Caso práctico alrededor de la moda

1. Presentación de Kaggle

Kaggle ([www.kaggle.com](http://www.kaggle.com/)) es el sitio de referencia en términos de proposiciones de desafíos relacionados con Machine Learning. Este sitio está repleto de casos de estudio y de retos propuestos por grandes grupos, como McDonald’s o Netflix. Todos podemos intentar resolver los distintos desafíos proponiendo nuestros propios modelos de predicción. Este sitio también permite aprender mucho sobre las técnicas relacionadas con Machine Learning porque todas las propuestas de solución enviadas por los participantes a los distintos desafíos son visibles, comentadas y así se pueden estudiar. Este sitio es completamente gratuito, no hay motivos para no visitarlo.

Vamos a realizar nuestro caso práctico a partir de este sitio. Este caso nos permitirá ilustrar los principios de programación de una red neuronal convolucional.

2. Módulo Keras

En el capítulo anterior utilizamos la biblioteca TensorFlow. Hay un módulo complementario a esta biblioteca llamado Keras que, ante todo, es fácil de usar. Esta facilidad reside especialmente en la fase de creación y de parametrización de los distintos modelos de aprendizaje, sin olvidar que también es muy adecuado para el uso de redes neuronales convolucionales porque posee las funciones adecuadas para ello de manera nativa.

3. Clasificar vestidos, jerséis y zapatos

Uno de los casos de estudio más utilizados en la puesta en práctica de las redes neuronales convolucionales es la clasificación de cartas manuscritas con ayuda de la colección de observaciones MNIST (*Mixed National Institute of Standards and Technology*). Sin embargo, en 2018 Zalando publicó su propia colección de imágenes bajo el nombre Zalando-MNIST. El propósito de esta colección de imágenes era permitir a los algoritmos entrenarse en la clasificación de objetos de moda: jersey, vestido, bolso, etc. Ahora vamos a utilizar está colección para nuestro caso práctico.

Más allá del aspecto, muy concreto, del proyecto, la mayoría de los ordenadores pueden ejecutar la fase de aprendizaje de este caso de estudio. En efecto, como ya hemos mencionado al principio de este capítulo, la clasificación de las imágenes requiere características adquiridas en términos de memoria, potencia de procesador y rendimiento de la tarjeta gráfica para obtener buenos resultados de manera rápida.

También es posible utilizar la nube alquilando máquinas virtuales situadas en servidores dedicados a la inteligencia artificial, como ofrece Amazon. Estas máquinas se pueden configurar en términos de potencia en función del proyecto que se desea realizar.

Ahora le invitamos a descargar los datos disponibles en el sitio del editor y después a crear un proyecto nuevo de Python en el que será necesario crear un directorio datas para depositar el conjunto de los datos que hemos descargado previamente.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

*Copia de los datos de aprendizaje en el directorio datas del proyecto*

4. ¿De qué datos disponemos?

Antes de sumergirnos completamente en la fase de aprendizaje, tenemos que conocer los datos de los que disponemos.

Como queremos clasificar imágenes, deberíamos estar en posesión de algunas para realizar el aprendizaje. Pero está claro que no tenemos archivos de imagen como los que estamos acostumbrados a encontrar. En efecto, no hay archivos JPEG ni PNG.

Sin embargo, si abrimos el archivo fashion-mnist\_train.csv, podemos darnos cuenta de que contiene observaciones con las siguientes características:

* Una etiqueta.
* Píxeles numerados del 1 al 784 con distintos valores.

Ahora veamos qué nos dice la documentación de este conjunto de observaciones disponible en el sitio de Kaggle (<https://www.kaggle.com/plarmuseau/zalando-image-classifier/data>):

* El conjunto de observaciones contiene 60 000 imágenes de aprendizaje y 10 000 imágenes de prueba.
* Todas las imágenes tienen una altura de 28 píxeles y una anchura de 28 píxeles, con un total de 784 píxeles.
* Cada píxel está asociado a un solo valor de píxel que indica el brillo de este píxel. Este valor de píxel es un número entero comprendido entre 0 y 255.
* Los conjuntos de datos de entrenamiento y de pruebas contienen 785 columnas.
* La primera columna está formada por etiquetas de clase y representa el artículo de vestir. El resto de las columnas contienen los valores de los píxeles de la imagen asociada.

Siempre a partir de la documentación, podemos determinar las distintas etiquetas:

* 0 - Camiseta/top
* 1 - Pantalón
* 2 - Jersey
* 3 - Vestido
* 4 - Abrigo
* 5 - Sandalias
* 6 - Camisa
* 7 - Zapatillas
* 8 - Bolso
* 9 - Botines

A partir de esta información podemos afirmar que nuestras imágenes tan esperadas están presentes en este archivo bajo la forma de valores de píxeles.

Para asegurarnos, vamos a probar a mostrar la primera imagen.

import pandas as pnd

import numpy as np

#Definición del largo y ancho de la imagen

LARGO\_IMAGEN = 28

ANCHO\_IMAGEN = 28

#Carga de las imágenes

observaciones\_entrenamiento = pnd.read\_csv('datas/zalando/

fashion-mnist\_train.csv')

#Excluimos la primera columna (las etiquetas) para formar una

#tabla de píxeles

X = np.array(observaciones\_entrenamiento.iloc[:, 1:])

from matplotlib import pyplot as plt

primeraImagen = X[0]

primeraImagen = primeraImagen.reshape([LARGO\_IMAGEN,

ANCHO\_IMAGEN]);

plt.imshow(primeraImagen)

plt.show()

Después de importar los módulos Pandas y Numpy, hemos especificado las dimensiones de la imagen para utilizarlas después en varios lugares de nuestro programa.

A continuación, hemos cargado las imágenes en memoria dentro de una variable usando el módulo Pandas y hemos excluido la primera columna para formar una tabla que no contenga los datos de las imágenes (los píxeles). Después, hemos buscado la primera imagen en la tabla y luego hemos modificado su formato (reshape) para obtener un tamaño de 28 píxeles por 28 píxeles. Finalmente, la hemos mostrado con ayuda del módulo matplotlib.

Gráfico, Histograma

Descripción generada automáticamente

*Visualización de una imagen contenida en los datos de aprendizaje*

5. Preparación de los datos de aprendizaje

Ahora que tenemos una ligera idea de los datos que vamos a utilizar, hemos de prepararlos para que nuestra red neuronal pueda explotarlos.

La primera etapa consistirá en crear nuestros conjuntos de aprendizaje y conjuntos de pruebas de la siguiente manera:

import pandas as pnd

import numpy as np

**from keras.utils import to\_categorical**

**from sklearn.model\_selection import train\_test\_split**

#Definición del largo y ancho de la imagen

LARGO\_IMAGEN = 28

ANCHO\_IMAGEN = 28

#Carga de los datos de entrenamiento

observaciones\_entrenamiento = pnd.read\_csv('datas/zalando/

fashion-mnist\_train.csv')

#Solo se conservan las características "píxeles"

X = np.array(observaciones\_entrenamiento.iloc[:, 1:])

#Creamos categorías con ayuda del módulo Keras

y = to\_categorical(np.array(observaciones\_entrenamiento.iloc[:, 0]))

#Distribución de los datos de entrenamiento en datos

#de aprendizaje y datos de validación

#80 % de datos de aprendizaje y 20 % de datos de validación

X\_aprendizaje, X\_validacion, y\_aprendizaje, y\_validacion =

train\_test\_split(X, y, test\_size=0.2, random\_state=13)

Mediante estas líneas de programa acabamos de crear nuestro conjunto de aprendizaje compuesto por datos de aprendizaje y datos de validación. Hay que destacar el uso del módulo Keras y de la función to\_categorical, que permite crear una tabla binaria de las distintas categorías.

En nuestro ejemplo disponemos de 10 categorías, es decir, una tabla de 10 valores posibles: [0,0,0,0,0,0,0,0,0,0].

La primera imagen de nuestro conjunto de datos lleva el número 2 en su característica etiqueta; vamos a poner un 1 en la tercera posición (la primera correspondiente a la categoría que lleva el número 0):

[0,0,1,0,0,0,0,0,0,0]

La etiqueta de la segunda imagen es el número 9, lo que nos da la tabla siguiente:

[0,0,0,0,0,0,0,0,0,1]

Este proceso también se realiza para cada una de las observaciones con la finalidad de formar una tabla global:

[[0,0,1,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,0,0,1]…]

La siguiente fase consistirá en volver a dar formato a todos los datos de las imágenes con formato 28 \* 28 y en transformar los valores de cada píxel a un rango de valores comprendidos entre 0 y 1.

Esta transformación es una adaptación a escala de estos valores de los píxeles, (scaling) para lo cual se divide cada valor de píxel entre 255 (valor máximo de un píxel).

#Se redimensionan las imágenes con formato 28\*28 y realizamos un

#scaling en los datos de los píxeles

X\_aprendizaje =

X\_ aprendizaje.reshape(X\_aprendizaje.shape[0], ANCHO\_IMAGEN,

LARGO\_IMAGEN, 1)

X\_aprendizaje = X\_aprendizaje.astype('float32')

X\_aprendizaje /= 255

A continuación, realizamos las mismas operaciones con los datos de validación:

X\_validacion = X\_validacion.reshape(X\_validacion.shape[0],

ANCHO\_IMAGEN, LARGO\_IMAGEN, 1)

X\_validacion = X\_validacion.astype('float32')

X\_validacion /= 255

6. Preparación de los datos de pruebas

Los datos de pruebas van a permitirnos verificar la eficacia de nuestro modelo de aprendizaje. Estos datos deben prepararse de la misma forma que los datos de aprendizaje:

#Preparación de los datos de pruebas

observaciones\_test = pnd.read\_csv('datas/zalando/fashion-

mnist\_test.csv')

X\_test = np.array(observaciones\_test.iloc[:, 1:])

y\_test = to\_categorical(np.array(observaciones\_test.iloc[:, 0]))

X\_test = X\_test.reshape(X\_test.shape[0], ANCHO\_IMAGEN,

LARGO\_IMAGEN, 1)

X\_test = X\_test.astype('float32')

X\_test /= 255

7. Una red con una sola capa de convolución

Vamos a efectuar un primer aprendizaje con una red neuronal que solo contiene una única capa de convolución.

**a. Configuración**

La configuración de nuestra primera red neuronal convolucional contendrá los siguientes elementos (ver figura siguiente):

* Una capa de convolución de 32 filtros de tamaño 3 x 3 con una función de activación de tipo ReLU.
* Un *pooling* compuesto de un filtro de 2 x 2.
* Una fase de *flatten* (colocación en columna de los resultados del *pooling*).
* Una red neuronal totalmente conectada de 128 neuronas con una función de activación de tipo ReLU.
* Una última red neuronal totalmente conectada que contiene 10 neuronas y una función de activación de tipo Softmax correspondiente a las 10 categorías de imágenes que se van a predecir.

Diagrama

Descripción generada automáticamente

*Red convolucional que contiene una sola capa de convoluciones*

Aquí puede ver el programa asociado a esta configuración:

from keras.models import Sequential

from keras.layers import Dense, Dropout, Flatten

from keras.layers import Conv2D, MaxPooling2D

#Especificamos las dimensiones de la imagen de entrada

dimensionImagen = (ANCHO\_IMAGEN, LARGO\_IMAGEN, 1)

#Creamos la red neuronal capa a capa

**redNeurona1Convolucion = Sequential()**

#1- Adición de la capa de convolución que contiene

#  32 filtros de tamaño 3x3 (Kernel) recorriendo la imagen

#  Una función de activación de tipo ReLU (Rectified Linear

#Activation)

#  Una imagen de entrada de 28 px \* 28 px

**redNeurona1Convolucion.add(Conv2D(32, kernel\_size=(3, 3),**

**activation='relu', input\_shape=dimensionImagen))**

#2- Definición de la función de pooling con una ventana de 2px

#por 2 px

**redNeurona1Convolucion.add(MaxPooling2D(pool\_size=(2, 2)))**

#3- Adición de una función de ignorancia

**redNeurona1Convolucion.add(Dropout(0.2))**

#5 - Se transforma en una sola línea

**redNeurona1Convolucion.add(Flatten())**

#6 - Adición de una red neuronal compuesta de 128 neuronas con

#una función de activación de tipo ReLU

**redNeurona1Convolucion.add(Dense(128, activation='relu'))**

#7 - Adición de una red de neuronas compuesta de 10 neuronas con

#una función de activación de tipo softmax

**redNeurona1Convolucion.add(Dense(10, activation='softmax'))**

El concepto Drop Out que aparece en el programa consiste en ignorar algunas neuronas en las fases de aprendizaje y de retropropagación para evitar el sobreajuste.

**b. Compilación, aprendizaje y prueba**

Ahora que está configurada nuestra red, podemos compilarla y realizar el aprendizaje:

import keras

**redNeurona1Convolucion.compile(loss=keras.losses.categorical\_**

**crossentropy,**

**optimizer=keras.optimizers.Adam(),**

**metrics=['accuracy'])**

La compilación se hace precisando los siguientes parámetros:

* La función de minimización del error.
* El optimizador utiliza el algoritmo Adam (método de cálculo de descenso de gradiente).
* Y el valor medido, en nuestro caso la precisión.

Antes de ejecutar nuestro programa, vamos a parametrizar el aprendizaje de la siguiente manera:

* 10 pasadas completas del conjunto de datos a través de la red neuronal (epoch).
* 256 imágenes tratadas simultáneamente (batch\_size).
* Una visualización de los registros de aprendizaje en la consola (verbose=1).
* La validación del modelo se basa en las imágenes y etiquetas de validación definidas anteriormente en este capítulo (80 % de datos de aprendizaje y 20 % de datos de validación).

historico\_aprendizaje =

redNeurona1Convolucion.fit(X\_aprendizaje, y\_aprendizaje,

         batch\_size=256,

         epochs=10,

         verbose=1,

         validation\_data=(X\_validation, y\_validation))

En otras palabras, sabiendo que nuestro conjunto de aprendizaje contiene 60 000 imágenes y que solo hemos retenido el 80 % para la fase de aprendizaje, esta última se realizará sobre 48 000 imágenes.

Como no podemos pasar las 48 000 imágenes de una vez a través de la red neuronal, tenemos que enviarlas mediante un *batch* (sistema por lotes). Según nuestra parametrización, el *batch* enviará 256 imágenes a través de la red de neuronas. Por lo tanto, para hacer una pasada completa del conjunto de las imágenes dentro de la red neuronal (epoch), necesitaremos 187 iteraciones (48 000/256). Dejamos esta cifra a un lado y ya veremos cómo nos será útil un poco más adelante en este capítulo.

También añadimos la evaluación del modelo de aprendizaje en los datos de pruebas para determinar la precisión de aprendizaje:

**evaluation = redNeurona1Convolucion.evaluate(X\_test, y\_test,**

**verbose=0)**

print('Error:', evaluation[0])

print('Precision:', evaluation[1])

Y terminamos con la visualización de un gráfico donde figurarán las curvas de precisión y de error después de cada pasada completa del conjunto de datos a través de la red neuronal (epoch).

#Datos de precisión (accurary)

plt.plot(historico\_aprendizaje.history['acc'])

plt.plot(historico\_aprendizaje.history['val\_acc'])

plt.title('Precisión del modelo')

plt.ylabel('Precisión')

plt.xlabel('Epoch')

plt.legend(['Aprendizaje', 'Test'], loc='upper left')

plt.show()

# Gráfico entrenamiento y validación valores perdidos

plt.plot(historico\_aprendizaje.history['loss'])

plt.plot(historico\_aprendizaje.history['val\_loss'])

plt.title('Error')

plt.ylabel('Error')

plt.xlabel('Epoch')

plt.legend(['Aprendizaje', 'Test'], loc='upper left')

plt.show()

¡Ahora podemos iniciar el aprendizaje!

Hay que destacar que, en nuestro caso, la fase de aprendizaje dura alrededor de 5 minutos utilizando un PC equipado con un procesador Intel Core i5 2,6G GHz, 8 GB de memoria y una tarjeta gráfica Intel HD Graphics 620 equipada con 64 MB de RAM.

Este tiempo puede variar en función de las características de su equipo.

**c. Conclusión sobre el aprendizaje**

Una vez terminado el aprendizaje y constituido nuestro modelo de aprendizaje, obtenemos las siguientes cifras sobre nuestro conjunto de pruebas (pueden variar).

Error: 0.23872388958334922

Precisión: 0.9154

Significa que tenemos una precisión de clasificación del 91,54 %, con una tasa de error del 23,87 %.

Esta información se retoma en los dos gráficos siguientes, donde se comprueba un aumento de la precisión en la fase de aprendizaje hasta alcanzar el 93 %, así como una disminución del error. Estos dos datos son completamente lógicos.

Cuando se aplica nuestro modelo en los datos de pruebas, se originan las gráficas siguientes:

Gráfico, Gráfico de líneas

Descripción generada automáticamente

*Precisión de la red neuronal en los datos de aprendizaje y de pruebas*

Gráfico, Gráfico de líneas

Descripción generada automáticamente

*Errores realizados por la red neuronal en los datos de aprendizaje y de pruebas*

**d. Aumento de la cantidad de datos**

Es posible que dentro de nuestro conjunto de datos de pruebas haya imágenes de las que no disponga nuestro conjunto de entrenamiento. Eso influye sobre la precisión de clasificación de nuestro modelo.

En ese caso, la solución consiste en aumentar la cantidad de imágenes de nuestro conjunto de aprendizaje. Por eso no hay ninguna necesidad de buscar fotografías en Internet: vamos a utilizar una función ImageDataGenerator del módulo Keras adecuada para este propósito.

**from keras.preprocessing.image import ImageDataGenerator**

generador\_imagenes = **ImageDataGenerator(rotation\_range=8,**

**width\_shift\_range=0.08,**

**shear\_range=0.3,**

**height\_shift\_range=0.08,**

**zoom\_range=0.08)**

El principio de esta función es el siguiente:

A partir de una imagen contenida en el conjunto de aprendizaje, la función va a proceder a realizar acciones de giro, zoom y desplazamiento de manera aleatoria para crear imágenes nuevas. Aquí podemos ver los parámetros de esta función:

* rotation\_range es el ángulo de giro de la imagen, comprendido entre 0 y 180.
* width\_shift y height\_shift son los rangos de desplazamiento vertical y horizontal de la imagen.
* shear\_range es el nivel de inclinación de la imagen (figura siguiente).
* zoom\_range es el nivel de zoom de la imagen.

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

*Inclinación de la imagen*

Ahora vamos a generar imágenes nuevas y procederemos a realizar un aprendizaje nuevo.

nuevas\_imagenes\_aprendizaje =

generador\_imagenes.flow(X\_aprendizaje, y\_aprendizaje,

batch\_size=256)

nouvas\_imagenes\_validacion =

generador\_imagenes.flow(X\_validacion, y\_validacion,

batch\_size=256)

historico\_aprendizaje =**redNeurona1Convolucion.fit\_generator(nuevas\_imagenes\_aprendizaje,**

**steps\_per\_epoch=48000//256,**

**epochs=50,**

**validation\_data=nuevas\_imagenes\_validacion,**

**validation\_steps=12000//256,**

**use\_multiprocessing=False,**

**verbose=1 )**

evaluacion = redNeurona1Convolucion.evaluate(X\_test, y\_test,

verbose=0)

print('Error:', evaluacion[0])

print('Precision:', evaluacion[1])

Para nuestro nuevo aprendizaje, utilizamos la función fit\_generator del módulo Keras en lugar de la función fit utilizada en el aprendizaje anterior. Esto se debe al hecho de que las imágenes que utilizamos son diferentes para cada subtarea de aprendizaje (batch) porque han sido generadas de manera aleatoria con nuestro generador de imágenes. La función fit no es capaz de gestionar este tipo de cambio sobre la marcha, a diferencia de la función fit\_generator, adecuada para este caso.

También hay que destacar un aumento de la cantidad de pasadas completas del conjunto de datos a través de la red neuronal, aumentando de 10 a 50 porque disponemos de una cantidad de imágenes mayor.

La función fit\_generator se puede considerar como un bucle de aprendizaje sin fin. Para que este bucle se detenga, tenemos que indicarle la cantidad de iteraciones que debe efectuar por epoch; esa es la función del parámetro steps\_per\_epoch que toma el valor 187, como habíamos calculado durante el primer aprendizaje. Así, cuando una epoch alcance la cantidad de iteraciones, se iniciará la epoch siguiente hasta alcanzar la cantidad total de epoch especificadas, en nuestro caso son 50.

Ahora podemos ejecutar nuestro nuevo aprendizaje y comprobar en un primer momento que requiere mucho más tiempo (alrededor de 40 minutos frente a los 5 minutos que duraba el aprendizaje anterior). Esto se debe a que se han generado imágenes nuevas en cada batch como complemento de aprendizaje; por eso se necesitan mejores recursos materiales y se ralentiza el conjunto del proceso.

Pero a pesar de este tiempo de aprendizaje, se puede notar una pequeña mejora en el aprendizaje, que roza el 92 %.

Error: 0.2299670210123062

Precisión: 0.9175

Y una disminución de la tasa de error entre la fase de aprendizaje y la fase de prueba. Entonces, podemos considerar que este modelo tiene un rendimiento mayor que el anterior.

Interfaz de usuario gráfica

Descripción generada automáticamente con confianza baja

*Mejora de la precisión de la red neuronal*

Interfaz de usuario gráfica

Descripción generada automáticamente

*Disminución de la cantidad de errores de la red neuronal*

**e. Guardar el modelo**

Ahora que tenemos un modelo con un rendimiento un poco mejor, podemos guardarlo para utilizarlo después.

Para ello, es recomendable crear un directorio nuevo dentro del proyecto al que llamaremos «modelo», y que será el encargado de contener la definición modelo en el formato JSON. Los distintos pesos procedentes del aprendizaje se guardarán dentro de un archivo con la extensión H5. Estos dos archivos se pueden utilizar más tarde en una aplicación adicional.

#Guardar el modelo

# Serialización del modelo **model\_json =**

**redNeurona1Convolucion.to\_json() with open("modelo/modelo.json",**

**"w") as json\_file:**

**json\_file.write(model\_json)**

**# Serialización de los pesos**

**redNeurona1Convolucion.save\_weights("modelo/modelo.h5")**

**print("¡Modelo guardado!")**

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

*Guardado de la red neuronal convolucional*

8. Un modelo más eficaz

Ahora vamos a utilizar un modelo más complejo, pero que ofrece un rendimiento mejor. Este modelo contiene 4 capas de convolución definidas de la siguiente manera:

redNeuronas4Convolucion = Sequential()

redNeuronas4Convolucion.add(Conv2D(32, kernel\_size=(3, 3),

activation='relu', input\_shape=dimentionImage))

redNeuronas4Convolucion.add(BatchNormalization())

redNeuronas4Convolucion.add(Conv2D(32, kernel\_size=(3, 3),

activation='relu'))

redNeuronas4Convolucion.add(BatchNormalization())

redNeuronas4Convolucion.add(MaxPooling2D(pool\_size=(2, 2)))

redNeuronas4Convolucion.add(Dropout(0.25))

redNeuronas4Convolucion.add(Conv2D(64, kernel\_size=(3, 3),

activation='relu'))

redNeuronas4Convolucion.add(BatchNormalization())

redNeuronas4Convolucion.add(Dropout(0.25))

redNeuronas4Convolucion.add(Conv2D(128, kernel\_size=(3, 3),

activation='relu'))

redNeuronas4Convolucion.add(BatchNormalization())

redNeuronas4Convolucion.add(MaxPooling2D(pool\_size=(2, 2)))

redNeuronas4Convolucion.add(Dropout(0.25))

redNeuronas4Convolucion.add(Flatten())

redNeuronas4Convolucion.add(Dense(512, activation='relu'))

redNeuronas4Convolucion.add(BatchNormalization())

redNeuronas4Convolucion.add(Dropout(0.5))

redNeuronas4Convolucion.add(Dense(128, activation='relu'))

redNeuronas4Convolucion.add(BatchNormalization())

redNeuronas4Convolucion.add(Dropout(0.5))

redNeuronas4Convolucion.add(Dense(10, activation='softmax'))

Destacamos el uso de la función BatchNormalisation() que permite normalizar los datos. En las fases de actualización de los pesos, es posible que los datos sean demasiado pequeños y demasiado grandes en relación a los otros. Gracias a esta función, los valores se normalizarán, es decir, que su media será casi 0 y su desviación típica casi 1. Esta fase también es la causante de la mejora en la realización del tratamiento.

Estos son los resultados obtenidos:

Tiempo de aprendizaje = 53044.6571731

Error: 0.19346623014211656

Precisión: **0.9302**

Es decir, un 93 % de clasificaciones correctas. Pero esto tiene un coste: con la configuración de nuestro PC que hemos mencionado en este capítulo, ¡la fase de aprendizaje ha durado alrededor de 8 horas!