CAPÍTULO 5.

Redes neuronales en Keras

A continuación, pasaremos a un nivel más práctico con el ejemplo de reconocimiento de dígitos MNIST presentado en el capítulo anterior; haremos una primera presentación de los pasos que se siguen para entrenar una red neuronal usando la API de Keras.

5.1. Precarga de los datos en Keras

Para facilitar la iniciación al lector o lectora en Deep Learning, en Keras se dispone de un conjunto de datos precargados, como veremos más adelante con más detalle. Uno de ellos es el conjunto de datos MNIST, que se encuentra precargado en forma de cuatro *arrays* NumPy y se puede obtener con el siguiente código:

```
mnist = tf.keras.datasets.mnist

(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

`x_train` y `y_train` conforman el conjunto de entrenamiento, mientras que `x_test` y `y_test` contienen los datos de prueba. Las imágenes se encuentran codificadas como *arrays* NumPy y sus correspondientes etiquetas *(labels)*, que van desde 0 hasta 9. Siguiendo la estrategia del libro de ir introduciendo gradualmente los conceptos del tema (dejamos para más adelante, capítulo 9, cómo separar una parte de los datos de entrenamiento para guardarlos como los datos de validación, muy importante para afinar el modelo), de momento solo tendremos en cuenta los datos de entrenamiento y de prueba.

Si queremos comprobar qué valores hemos cargado, podemos elegir cualquiera de las imágenes del conjunto MNIST, por ejemplo la imagen 8 ya presentada en el capítulo anterior, usando el siguiente código Python:

```
import matplotlib.pyplot as plt
plt.imshow(x_train[8], cmap=plt.cm.binary)
```

Obtenemos la siguiente imagen:

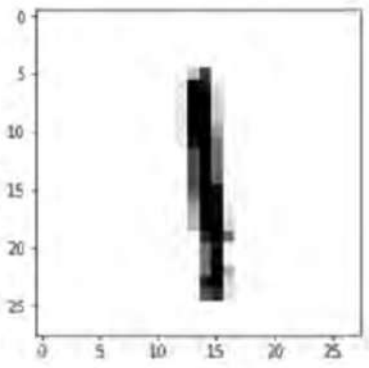

Figura 5.1 *Visualización de la muestra número 8 del conjunto de datos de entrenamiento MNIST.*

Si queremos ver su correspondiente etiqueta *(label)* podemos hacerlo mediante:

```
print(y_train[8])
```

```
1
```

Como vemos, nos devuelve el valor 1, como era de esperar.

Les propongo que pongamos en práctica la introducción a NumPy del capítulo 3 para conocer mejor nuestros datos. Primero, obtengamos el número de ejes y dimensiones del tensor `x_train` de nuestro ejemplo:

```
print(x_train.ndim)
```

```
3
```

```
print(x_train.shape)
```

```
(60000, 28, 28)
```

Si queremos saber qué tipo de datos contiene:

```
print(x_train.dtype)
```

```
uint8
```

En resumen, `x_train` es un tensor 3D de enteros de 8 bits. Más concretamente, se trata de un vector de 60 000 matrices 2D de 28 x 28 enteros. Cada una de esas matrices es una imagen en escala de grises, con coeficientes entre 0 y 255.

En este ejemplo estamos tratando un ejemplo en blanco y negro, pero en general una imagen en color suele tener tres dimensiones: altura, anchura y profundidad de color. Las imágenes en escala de grises (como nuestros dígitos MNIST) tienen un solo canal de color y, por lo tanto, pueden almacenarse en tensores 2D cada una de ellas. Pero habitualmente los tensores de imágenes son 3D, con un canal de color en una dimensión para las imágenes en escala de grises. Por ejemplo, 64 imágenes en escala de grises de tamaño 128 x 128 podrían almacenarse en un tensor de forma (64, 128, 128, 1). En cambio, un conjunto de 64 imágenes en color de tamaño 128 x 128 podría almacenarse en un tensor de forma (64, 128, 128, 3). En este caso tenemos tres canales usando la codificación RGB que trataremos más adelante. Es decir, en ambos casos estamos hablando de que los datos los tenemos en un tensor de 4D con forma `(samples, height, width, channels)`.

En realidad, podemos representar cualquier conjunto de datos en tensores; pensemos en un vídeo, por ejemplo. En este caso simplemente nos hace falta un tensor de 5D con la forma `(samples, frames, height, width, channels)`. Es decir, un vídeo puede ser entendido como una secuencia de fotogramas *(frame)*, en la que cada fotograma es una imagen de color.

También hemos comentado en el capítulo 3 que una vez tenemos los datos en tensores, podemos manipularlos fácilmente gracias a la librería NumPy. Por ejemplo, en el caso que nos ocupa hemos seleccionado el elemento `x_train[8]` del tensor. Pero, ¿cómo podemos seleccionar y manipular porciones del conjunto de datos? Imaginemos que queremos seleccionar los dígitos desde el 1 hasta el 99, y ponerlos en otro tensor. Esto lo podemos seleccionar (indexar) usando «:» de la siguiente manera:

```
my_slice = x_train [1:100:,:]
print(my_slice.shape)
```

```
(99, 28, 28)
```

En realidad, como hemos visto en el capítulo 3, es equivalente a esta notación más detallada, que especifica un índice de inicio y un índice de final a lo largo de cada eje tensorial:

```
my_slice = x_train [1:100,0:28, 0:28]
print(my_slice.shape)
```

```
(99, 28, 28)
```

En general, podemos seleccionar entre dos índices cualesquiera a lo largo de cada eje tensorial. Por ejemplo, para seleccionar 14 × 14 píxeles en la esquina inferior derecha de todas las imágenes podemos hacerlo con:

```
my_slice = x_train [:, 14:, 14:]
print(my_slice.shape)
```

```
(60000, 14, 14)
```

Recordemos también que NumPy, además, nos permite indicar una posición relativa al final del eje actual usando índices negativos. Por ejemplo, para recortar la parte central de 14 × 14 píxeles de las imágenes en parches de 14 × 14 píxeles centrados en el medio debe hacer esto:

```
my_slice = x_train [:, 7:-7, 7:-7]
print(my_slice.shape)
```

```
(60000, 14, 14)
```

5.2. Preprocesado de datos de entrada en una red neuronal

En general, siempre hay un preprocesamiento de datos con el objetivo de adaptarlos a un formato que permita un mejor aprovechamiento de estos por parte de las redes neuronales. Alguno de los preprocesamientos más habituales en Deep Learning son vectorización, normalización o extracción de características. En este ejemplo presentaremos algunos de ellos.

Estas imágenes de MNIST de 28 × 28 píxeles en nuestro ejemplo se representan como una matriz de números cuyos valores van entre [0, 255] de tipo *uint8*. Pero, como veremos en posteriores capítulos, es habitual escalar los valores de entrada de las redes neuronales a unos rangos determinados (esto se llama normalización). Por ejemplo, en el caso que nos ocupa en este capítulo los valores de entrada conviene escalarlos a valores de tipo *float32* dentro del intervalo [0, 1]:

```
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')

x_train /= 255
x_test /= 255
```

Este tipo de normalización se hace muy a menudo para facilitar que converja el proceso de entrenamiento de la red neuronal. Porque, en general, para alimentar a redes neuronales no se usan datos con valores que sean mucho más grandes que los valores de los pesos de una red, o datos que sean heterogéneos de rango entre ellos.

Otra transformación que se requiere a veces es cambiar la forma de los tensores sin cambiar los datos. Para ello tenemos la función `numpy.reshape()`, como hemos avanzado en el capítulo 3. Podemos usar este ejemplo para mostrar su utilidad. Para facilitar la entrada de datos a nuestra red neuronal debemos hacer una transformación del tensor (imagen) de 2 dimensiones (2D) a un vector de una dimensión (1D).

Es decir, la matriz de 28 × 28 números se puede representar con un vector *(array)* de 784 números (concatenando fila a fila), que es el formato que acepta como entrada una red neuronal densamente conectada como la que veremos a continuación.

Con la función `numpy.reshape()` se puede convertir cada imagen del conjunto de datos MNIST a un vector con 784 componentes:

```
x_train = x_train.reshape(60000, 784)
x_test = x_test.reshape(10000, 784)
```

Después de ejecutar estas dos líneas de código, podemos comprobar que *x_train.shape* toma la forma de (60000, 784) y *x_test.shape* toma la forma de (10000, 784), donde la primera dimensión indexa la imagen y la segunda indexa el píxel en cada imagen (ahora la intensidad del píxel es un valor entre 0 y 1):

```
print(x_train.shape)
print(x_test.shape)
```

```
(60000, 784)
(10000, 784)
```

Cambiar la forma de un tensor significa reorganizar sus filas y columnas para que coincidan con la forma deseada. Naturalmente, el tensor reformado tiene el mismo número total de datos que el tensor inicial.

Además, tenemos las etiquetas *(labels)* para cada dato de entrada — recordemos que en nuestro caso son números entre 0 y 9 que indican qué dígito representa la imagen, es decir, a qué clase se asocia—. En este ejemplo, y como ya hemos avanzado, vamos a representar esta etiqueta con un vector de 10 posiciones, donde la posición correspondiente al dígito que representa la imagen contiene un 1 y el resto de posiciones del vector contienen el valor 0.

Usaremos lo que se conoce como codificación *one-hot* (que explicaremos más adelante), que consiste en transformar las etiquetas en un vector de tantos ceros como número de etiquetas distintas, y que contiene el valor de 1 en el índice, que corresponde al valor de la etiqueta. Keras ofrece muchas funciones de soporte, entre ellas `to_categorical`, para realizar esta transformación. La podemos importar de `tensorflow.keras.utils`:

```
from tensorflow.keras.utils import to_categorical
```

Para ver el efecto de la transformación, podemos visualizar los valores antes y después de aplicar `to_categorical`:

```
print(y_test[0])
```

```
7
```

```
print(y_train[0])
```

```
5
```

```
print(y_train.shape)
```

```
(60000,)
```

```
print(x_test.shape)
```

```
(10000, 784)
```

```
y_train = to_categorical(y_train, num_classes=10)
y_test = to_categorical(y_test, num_classes=10)

print(y_test[0])
```

```
[0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]
```

```
print(y_train[0])
```

```
[0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
```

```
print(y_train.shape)
```

```
(60000, 10)
```

```
print(y_test.shape)
```

```
(10000, 10)
```

Ahora ya tenemos los datos preparados para ser usados en nuestro ejemplo de modelo simple, que vamos a programar usando la API Keras en la próxima sección.

5.3. Definición del modelo

La estructura de datos principal en Keras es la clase *Sequential*[93], que permite la creación de una red neuronal básica. Keras ofrece también una API[94] que permite implementar modelos más complejos en forma de grafo, que pueden tener múltiples entradas, múltiples salidas, conexiones arbitrarias en medio... Pero no lo presentaremos hasta el capítulo 12 del libro.

En este caso, el modelo en Keras se considera como una secuencia de capas; cada una de ellas va «destilando» gradualmente los datos de entrada para obtener la salida deseada. En Keras podemos encontrar todos los tipos de capas requeridas y se pueden agregar fácilmente al modelo.

La construcción en Keras de nuestro modelo para reconocer las imágenes de dígitos podría ser la siguiente:

```
model = Sequential()
model.add(Dense(10,activation='sigmoid',input_shape=(784,)))
model.add(Dense(10, activation='softmax'))
```

Aquí, la red neuronal se ha definido como una lista secuencia de dos capas densas que están completamente conectadas, es decir, todas las neuronas de la

[93] Véase https://keras.io/models/sequential/ [Consulta: 16/12/2019].

[94] Véase https://keras.io/getting-started/functional-api-guide/ [Consulta: 16/12/2019].

primera capa están conectadas con todas las neuronas de la siguiente. Visualmente podríamos representarlo de la siguiente manera:

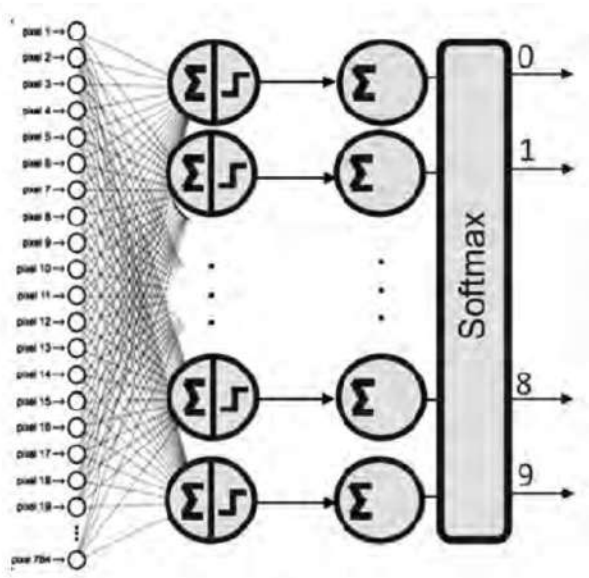

Figura 5.2 *Representación visual de una red neuronal de dos capas densamente conectadas y con función de activación* softmax *en la última capa.*

En este código expresamos explícitamente en el argumento `input_shape` de la primera capa cómo son los datos de entrada: un tensor que indica que tenemos 784 características (*features)* del modelo; en realidad el tensor que se está definiendo es de (None, 784,), como veremos más adelante.

Una característica muy interesante de la librería de Keras es que deducirá automáticamente la forma de los tensores entre capas después de la primera capa. Esto significa que solo tenemos que establecer esta información para la primera de ellas.

Para cada capa indicamos el número de nodos que tiene y la función de activación que aplicaremos en ella (en este caso, la función de activación `sigmoid` en la primera capa*)*.

La segunda capa es una capa con una función de activación `softmax` de 10 neuronas, lo que significa que devolverá una matriz de 10 valores de probabilidad, que representan a los 10 dígitos posibles (en general, la capa de salida de una red de clasificación tendrá tantas neuronas como clases —menos en una clasificación binaria, donde solo necesita una neurona—). Cada valor será la probabilidad de que la imagen del dígito actual pertenezca en cada una de las clases.

Un método de la clase `model` muy útil que proporciona Keras para comprobar la arquitectura de nuestra modelo es `summary()`:

```
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 10)                7850
_________________________________________________________________
dense_2 (Dense)              (None, 10)                110
=================================================================
Total params: 7,960
Trainable params: 7,960
Non-trainable params: 0
```

El método `summary()` del modelo muestra todas las capas del modelo, lo que incluye el nombre de cada capa (que se genera automáticamente, a menos que lo configuremos en un argumento al crear la capa), su forma de salida y su número de parámetros. El `summary()` termina con el número total de parámetros, incluidos los parámetros entrenables y no entrenables. Aquí solo tenemos parámetros entrenables (veremos ejemplos de parámetros no entrenables en el capítulo 11).

Más adelante, entraremos en más detalle en el análisis de la información que nos retorna el método `summary()`, pues este cálculo de parámetros y tamaños de los datos que tiene la red neuronal resulta muy valioso cuando empezamos a construir modelos de redes muy grandes. Para nuestro ejemplo simple, vemos que indica que se requieren 7960 parámetros, que corresponden a los 7850 parámetros para la primera capa y los 110 para la segunda (columna `Param #`).

Podemos aprovechar este ejemplo para observar que las capas densas a menudo tienen muchos parámetros. Por ejemplo, en la primera capa, por cada neurona *i* (entre 0 y 9) requerimos 784 parámetros para los pesos w_{ij} y, por tanto, 10 x 784 parámetros para almacenar los pesos de las 10 neuronas; además de los 10 parámetros adicionales para los 10 sesgos b_j correspondientes a cada una de ellas. La suma que nos da son los 7850 parámetros que nos muestra el método `summary()` para la primera capa.

En la segunda capa, al ser una función *softmax*, se requiere conectar todas sus 10 neuronas con las 10 neuronas de la capa anterior y, por tanto, se requieren 10 x 10 parámetros w_i; además de los 10 sesgos b_j correspondientes a cada nodo. Esto nos da un total de 110 parámetros que nos muestra el método `summary()` para la segunda capa.

En el manual de Keras se pueden encontrar los detalles de los argumentos que podemos indicar para la capa *Dense*[95]. En nuestro ejemplo, aparecen los más relevantes, donde el primer argumento indica el número de neuronas de la capa; el siguiente es la función de activación que usaremos en ella. En el capítulo 7 hablaremos con más detalle de otras posibles funciones de activación, más allá de las dos presentadas aquí: *sigmoid* y *softmax*.

[95] Véase https://keras.io/layers/core/#dense [Consulta: 16/12/2019].

También a menudo se indica la inicialización de los pesos como argumento de las capas *Dense*. Los valores iniciales deben ser adecuados para que el problema de optimización converja tan rápido como sea posible en el proceso de entrenamiento de la red. En el manual de Keras se pueden encontrar las diversas opciones de inicialización[96].

5.4. Configuración del proceso de aprendizaje

A partir del modelo *Sequential*, podemos definir las capas del modelo de manera sencilla, tal como hemos avanzado en el apartado anterior. Una vez que tengamos nuestro modelo definido, debemos configurar cómo será su proceso de aprendizaje con el método `compile()`, con el que podemos especificar algunas propiedades a través de argumentos del método.

El primero de estos argumentos es la función de coste *(loss function)*, que usaremos para evaluar el grado de error entre las salidas calculadas y las salidas deseadas de los datos de entrenamiento. Por otro lado, se especifica un optimizador que, como veremos, es la manera que tenemos de indicar los detalles del algoritmo de optimización que permite a la red neuronal calcular los pesos de los parámetros durante el entrenamiento a partir de los datos de entrada y de la función de coste definida. Entraremos en más detalle sobre el propósito exacto de la función decoste y el optimizador usados en el capítulo 6.

Finalmente, debemos indicar la métrica que usaremos para monitorizar el proceso de aprendizaje (y prueba) de nuestra red neuronal. En este primer ejemplo solo tendremos en cuenta la precisión (fracción de imágenes que son correctamente clasificadas). Por ejemplo, en nuestro ejemplo podemos especificar los siguientes argumentos en el método `compile()` para probarlo:

```
model.compile(loss="categorical_crossentropy",
              optimizer="sgd",
              metrics = ['accuracy'])
```

Especificamos que la función de coste es `categorical_crossentropy`, el optimizador usado es el *stocastic gradient descent (*`sgd`) y la métrica es `accuracy`.

5.5. Entrenamiento del modelo

Una vez definido nuestro modelo y configurado su método de aprendizaje, este ya está listo para ser entrenado. Para ello, podemos entrenar o ajustar el modelo a los datos de entrenamiento de que disponemos invocando al método `fit()` del modelo:

[96] Véase https://keras.io/initializers/#usage-of-initializers [Consulta: 16/12/2019].

```
model.fit(x_train, y_train, epochs=5)
```

En los dos primeros argumentos hemos indicado los datos con los que entrenaremos el modelo en forma de *arrays* NumPy. Con *epochs* estamos indicando el número de veces que usaremos todos los datos en el proceso de aprendizaje. (este último argumento se explicará con mucho más detalle en el capítulo 7).

Este método encuentra el valor de los parámetros de la red mediante el algoritmo iterativo de entrenamiento que hemos espeficado en el argumento `optimizer` del método `compile()`. A grandes rasgos, en cada iteración de este algoritmo, este coge datos de entrenamiento de `x_train`, los pasa a través de la red neuronal (con los valores que en aquel momento tengan sus parámetros), compara el resultado obtenido con el esperado (indicado en `y_train`) y calcula la *loss* con la función de coste para guiar el proceso de ajuste de los parámetros del modelo. Intuitivamente consiste en aplicar el optimizador especificado anteriormente en el método `compile()` para calcular un nuevo valor de cada uno de los parámetros (pesos y sesgos) del modelo en cada iteración, de tal forma de que se reduzca el valor de la *loss* en siguientes iteraciones.

Este es el método que, como veremos, puede llegar a tardar más tiempo. Keras nos permite ver su avance usando el argumento *verbose* (por defecto, igual a 1), además de indicar una estimación de cuánto tarda cada época (*epoch)*:

```
Epoch 1/5
60000/60000 [==========] - 4s 71us/sample - loss: 1.9272 - accuracy: 0.4613
Epoch 2/5
60000/60000 [==========] - 4s 68us/sample - loss: 1.3363 - accuracy: 0.7286
Epoch 3/5
60000/60000 [==========] - 4s 69us/sample - loss: 0.9975 - accuracy: 0.8066
Epoch 4/5
60000/60000 [==========] - 4s 68us/sample - loss: 0.7956 - accuracy: 0.8403
Epoch 5/5
60000/60000 [==========] - 4s 68us/sample - loss: 0.6688 - accuracy: 0.8588
10000/10000 [==================] - 0s 22us/step
```

Este es un ejemplo simple para que el lector o lectora, al acabar el capítulo, haya podido programar ya su primera red neuronal pero, como veremos, el método `fit()` permite muchos más argumentos que tienen un impacto muy importante en el resultado del aprendizaje.

Además, este método retorna un objeto `History` que hemos omitido en este ejemplo. Su atributo `History.history` es el registro de los valores de *loss* para los datos de entrenamiento y resto de métricas en sucesivas *epochs*, así como otras métricas para los datos de validación si se han especificado. En posteriores capítulos, veremos lo valioso de esta información para evitar, por ejemplo, el sobreajuste (*overfitting)* del modelo.

5.6. Evaluación del modelo

En este punto ya se ha entrenado la red neuronal y ahora se puede evaluar cómo se comporta con datos nuevos de prueba *(test)* con el método `evaluate()`. Este método devuelve dos valores:

```
test_loss, test_acc = model.evaluate(x_test, y_test)
```

que indican cómo de bien o de mal se comporta nuestro modelo con datos nuevos que nunca ha visto (que hemos almacenado en *x_test* y *y_test* cuando hemos realizado el `mnist.load_data()`). De momento, fijémonos solo en uno de ellos, la precisión:

```
print('Test accuracy:', test_acc)
```

```
Test accuracy: 0.8661
```

La precisión *(accuracy)* nos está indicando que el modelo que hemos creado en este capítulo aplicado sobre datos que nunca ha visto anteriormente clasifica el 90 % de ellos correctamente.

El lector o lectora debe fijarse en que, en este ejemplo, para evaluar este modelo solo nos hemos centrado en su precisión, es decir, en la proporción entre las predicciones correctas que ha hecho el modelo y el total de predicciones. Sin embargo, aunque en ocasiones resulta suficiente, otras veces es necesario profundizar un poco más y tener en cuenta los tipos de predicciones incorrectas que realiza el modelo en cada una de sus categorías, ya que pueden tener un impacto muy diferente.

Una herramienta muy utilizada en Machine Learning para evaluar el rendimiento de modelos es la matriz de confusión *(confusion matrix)*, una tabla con filas y columnas que contabilizan las predicciones en comparación con los valores reales. Usamos esta tabla para entender mejor cómo de bien o de mal el modelo se comporta, y es muy útil para mostrar de forma explícita cuándo una clase es confundida con otra. Una matriz de confusión para un clasificador binario como el explicado en el capítulo 4 tiene esta estructura:

		Predicción	
		Positivos	**Negativos**
Observación	**Positivos**	Verdaderos Positivos (VP)	Falsos Negativos (FN)
	Negativos	Falsos Positivos (FP)	Verdaderos Negativos (VN)

Figura 5.3. *Matriz de confusión para una clasificación binaria.*

Se trata de una matriz en la que se informa del recuento de las predicciones:

- VP es la cantidad de positivos que fueron clasificados correctamente como positivos por el modelo.
- VN es la cantidad de negativos que fueron clasificados correctamente como negativos por el modelo.
- FN es la cantidad de positivos que fueron clasificados incorrectamente como negativos.
- FP es la cantidad de negativos que fueron clasificados incorrectamente como positivos.

Con esta matriz de confusión, la precisión se puede calcular sumando los valores de la diagonal y dividiendo por el total:

$$\text{Precisión} = (\text{VP} + \text{VN}) / (\text{VP} + \text{FP} + \text{VN} + \text{FN})$$

Ahora bien, la precisión *(accuracy)* puede ser engañosa en la calidad del modelo porque al medirla para el modelo concreto no distinguimos entre los errores de tipo falso positivo y falso negativo —como si ambos tuvieran la misma importancia—. Por ejemplo, piensen en un modelo que predice si una seta es venenosa. En este caso el coste de un falso negativo, es decir, una seta venenosa dada por comestible, podría ser dramático. En cambio, al revés, un falso positivo, tiene un coste muy diferente.

Por ello tenemos otra métrica llamada *recall* que nos indica cómo de bien el modelo evita los falsos negativos:

$$\textit{Recall} = \text{VP} / (\text{VP} + \text{FN})$$

Es decir, del total de observaciones positivas (setas venenosas), cuántas detecta el modelo realmente.

A partir de la matriz de confusión se pueden obtener diversas métricas para focalizar otros casos, tal como se muestra en este enlace[97], pero queda fuera del alcance de este libro entrar más detalladamente en este tema. La conveniencia de usar una métrica u otra dependerá de cada caso en particular y, en concreto, del «coste» asociado a cada error de clasificación del modelo.

El lector o lectora se preguntará cómo es esta matriz de confusión en nuestro clasificador, donde tenemos 10 posibles valores. En este caso, propongo usar el paquete *Scikit-Learn*[98] (que ya hemos mencionado anteriormente) para evaluar la

---

[97] Confusion Matrix. Wikipedia. [*online*]. Disponible en: https://en.wikipedia.org/wiki/Confusion_matrix [Consultado: 30/04/2018].

[98] Véase http://scikit-learn.org/stable/ [Consulta 12/12/2019].

calidad del modelo calculando la matriz de confusión[99], presentada por la figura siguiente:

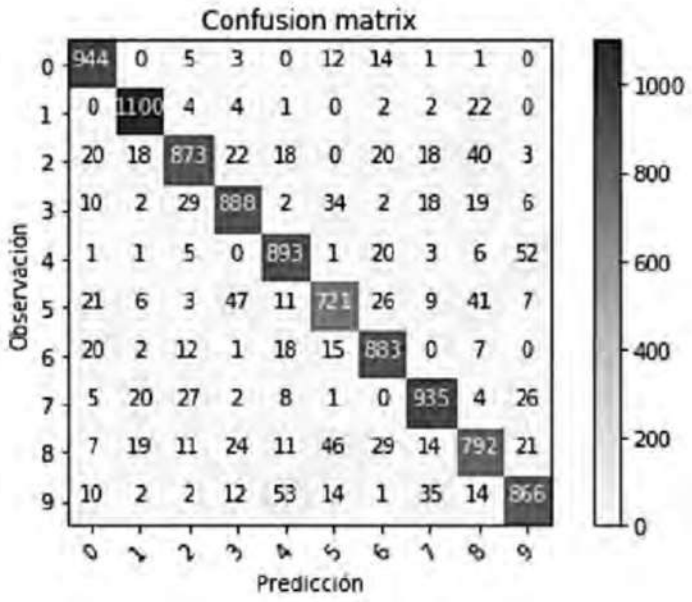

Figura 5.4 *Matriz de confusión para el modelo de este capítulo aplicado al problema de dígitos MNIST.*

En este caso, los elementos de la diagonal representan el número de puntos en que la etiqueta que predice el modelo coincide con el valor real de la etiqueta, mientras que los otros valores nos indican los casos en que el modelo ha clasificado incorrectamente. Por tanto, cuanto más altos son los valores de la diagonal mejor será la predicción. En este ejemplo, si el lector o lectora calcula la suma de los valores de la diagonal dividido por el total de valores de la matriz, observará que coincide con la precisión que nos ha retornado el método `evaluate()`.

En el GitHub del libro también pueden encontrar el código usado para calcular esta matriz de confusión.

5.7. Generación de predicciones

Finalmente, nos queda el paso de usar el modelo creado en los anteriores apartados para realizar predicciones sobre qué dígitos representan nuevas imágenes. Para ello, Keras ofrece el método `predict()` de un modelo que ya ha sido previamente entrenado.

---

[99] Véase http://scikit-learn.org/stable/modules/generated/sklearn.metrics.confusion_matrix.html [Consulta 12/12/2019].

Para probar este método podemos elegir un elemento cualquiera, por ejemplo uno del conjunto de datos de prueba `x_test` que ya tenemos cargado. Elijamos el elemento 11 de este conjunto de datos `x_test` y veamos a qué clase corresponde según el modelo entrenado de que disponemos.

Antes, vamos a visualizar la imagen para poder comprobar nosotros mismos si el modelo está haciendo una predicción correcta:

```
plt.imshow(x_test[11], cmap=plt.cm.binary)
```

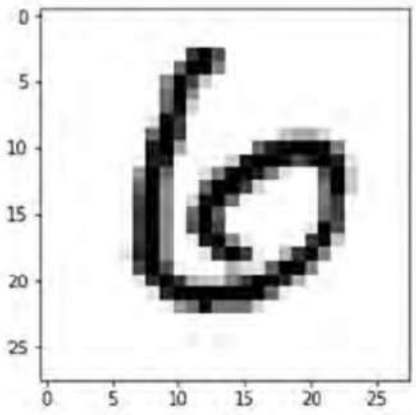

Figura 5.5 *Imagen de la muestra 11 del conjunto de prueba de MNIST.*

Creo que el lector o lectora estará de acuerdo en que, en este caso, se trata del número 6.

Ahora comprobemos que el método `predict()` del modelo prediga correctamente el valor que acabamos de estimar nosotros. Para ello, ejecutamos la siguiente línea de código:

```
predictions = model.predict(x_test)
```

Una vez calculado el vector resultado de la predicción para este conjunto de datos, podemos saber a qué clase le da más probabilidad de pertenencia mediante la función `argmax` de NumPy, que retorna el índice de la posición que contiene el valor más alto de la función. En concreto, para el elemento 11:

```
np.argmax(predictions[11])
```

Podemos comprobarlo imprimiendo el *array*:

```
print(predictions[11])
```

```
[0.06 0.01 0.17 0.01 0.05 0.04 0.54 0.   0.11 0.02]
```

Vemos que nos ha devuelto el índice 6, correspondiente a la clase «6», la que habíamos estimado nosotros, porque tiene el valor más alto.

También podemos comprobar con el siguiente código que el resultado de la predicción es un vector cuya suma de todos sus componentes es igual a 1, como era de esperar:

```
np.sum(predictions[11])
```

```
1.0
```

Hasta aquí el lector o lectora ha podido crear su primer modelo en Keras que clasifica correctamente los dígitos MNIST el 90 % de las veces.

5.8. Todos los pasos de una tirada

Antes de acabar este capítulo, proponemos hacer un repaso de cómo se crea un modelo en Keras aplicando de una tirada todos los pasos a otro conjunto de datos. En concreto proponemos el conjunto de datos Fashion-MNIST —también precargado en Keras y muy parecido al anterior en cuanto a la preparación requerida de los datos— para poder centrarnos en los pasos relacionados con el modelo. En el capítulo 10 ya entraremos en más detalle en la problemática que presentan habitualmente la carga de datos y la preparación que estos requieren para poder ser consumidos por una red neuronal.

Fashion-MNIST[100] es un conjunto de datos de las imágenes de los artículos de Zalando, una tienda de moda *online* alemana especializada en venta de ropa y zapatos. El conjunto de datos contiene 70 000 imágenes en escala de grises en 10 categorías. Las imágenes muestran prendas individuales de ropa en baja resolución (28 x 28 píxeles). Se usan 60 000 imágenes para entrenar la red y 10 000 imágenes para evaluar la precisión con la que la red aprende a clasificar las imágenes. Como se puede ver, este conjunto de datos comparte el mismo tamaño de imagen y estructura de entrenamiento que el conjunto de datos anterior.

[100] Véase https://github.com/zalandoresearch/fashion-mnist [Consulta: 16/12/2019].

Preparar los datos

Como siempre, antes de empezar a programar nuestra red neuronal debemos importar todas las librerías que se van a requerir (y asegurarnos de que estamos ejecutando la versión correcta de TensorFlow en nuestro Colab).

```
%tensorflow_version 2.x

import tensorflow as tf
from tensorflow import keras

import numpy as np
import matplotlib.pyplot as plt

print(tf.__version__)
```

```
2.0.0
```

A partir de este punto ya podemos cargar los datos:

```
fashion_mnist = keras.datasets.fashion_mnist

(train_images, train_labels), (test_images, test_labels)
       = fashion_mnist.load_data()
```

Igual que en el ejemplo anterior, la carga del conjunto de datos devuelve cuatro matrices NumPy. Las matrices `train_images` y `train_labels` son el conjunto de entrenamiento. Las matrices `test_images` y `test_labels` son el conjunto de prueba para evaluar la precisión del modelo.

Como hemos avanzado, las imágenes son matrices NumPy de 28 x 28 píxeles, con valores que van de 0 a 255. Las etiquetas son una matriz de enteros, que van de 0 a 9. Estos corresponden a la clase de ropa que representa la imagen:

Clase	Tipo
0	T-shirt/top
1	Trouser
2	Pullover
3	Dress
4	Coat

Clase	Tipo
5	Sandal
6	Shirt
7	Sneaker
8	Bag
9	Ankle boot

Dado que los nombres de clase no se incluyen con el conjunto de datos, podemos crear una lista con ellos para usarlos más adelante al visualizar las imágenes:

```
class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress',
               'Coat', 'Sandal', 'Shirt', 'Sneaker', 'Bag',
               'Ankle boot']
```

Al igual que en el ejemplo anterior, vamos a escalar los valores de entrada en el rango 0-1:

```
train_images = train_images.astype('float32')
test_images = test_images.astype('float32')

train_images = train_images / 255.0
test_images = test_images / 255.0
```

Recordemos que es una buena práctica comprobar que los datos tienen la forma que esperamos:

```
print("train_images.shape:",train_images.shape)
print("len(train_labels:",len(train_labels))
print("test_images.shape:",test_images.shape)
print("len(test_labels):",len(test_labels))
```

```
train_images.shape: (60000, 28, 28)
len(train_labels: 60000
test_images.shape: (10000, 28, 28)
len(test_labels): 10000
```

Y que las muestras y etiquetas son los valores que esperamos:

```
train_labels
```

```
array([9, 0, 0, ..., 3, 0, 5], dtype=uint8)
```

```
plt.figure(figsize=(12,12))
for i in range(50):
    plt.subplot(10,5,i+1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(train_images[i], cmap=plt.cm.binary)
    plt.xlabel(class_names[train_labels[i]])
plt.show()
```

La Figura 5.6 muestra la salida de este código.

Figura 5.6 *Visualización de las 50 primeras imágenes del conjunto de datos Fashion-MNIST.*

Definir el modelo

Dado que los datos son de la misma dimensión y forma, podríamos usar el mismo modelo que en el ejemplo anterior. Pero antes recordemos que en el modelo anterior hemos preprocesado los datos de entrada con la función `numpy.reshape()`. En realidad, Keras nos facilita este paso de reconvertir las muestras de entrada de 28 × 28 a un vector *(array)* de 784 números (concatenando fila a fila) con el uso de la capa `keras.layers.Flatten()`.

```
model = Sequential()
model.add(Flatten(input_shape=(28, 28)))
model.add(Dense(10, activation='sigmoid'))
model.add(Dense(10, activation='softmax'))
```

Podemos comprobar con el método `summary()` que esta capa no requiere parámetros para aplicar la transformación (columna `Param #`). En general, siempre usaremos esta capa del modelo para hacer esta operación en lugar de redimensionar el tensor de datos antes de la entrada.

```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_1 (Flatten)          (None, 784)               0
_________________________________________________________________
dense_2 (Dense)              (None, 10)                7850
_________________________________________________________________
dense_3 (Dense)              (None, 10)                110
=================================================================
Total params: 7,960
Trainable params: 7,960
Non-trainable params: 0
_________________________________________________________________
```

Configurar el modelo

Antes de que el modelo esté listo para ser entrenado, se requiere especificar el valor de algunos argumentos del método de compilación:

```
model.compile(optimizer='sgd',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

Recordemos que en este paso se especifica la función de coste *(loss)* que «dirige» el entrenamiento del modelo en la dirección correcta durante el proceso de entrenamiento. También especificamos el tipo de optimización que usaremos para actualizar los parámetros del modelo durante el proceso de aprendizaje. Y, finalmente, se indica la métrica que se usará para monitorear los pasos de entrenamiento y prueba. En este ejemplo nuevamente proponemos usar la precisión *(accuracy)*, es decir, la fracción de las imágenes que están clasificadas correctamente.

Entrenamiento del modelo

Ahora el modelo ya está listo para entrenar mediante el método `fit()`, actualizando los parámetros de tal manera que aprenda a asociar imágenes a etiquetas:

```
model.fit(train_images, train_labels, epochs=5)
```

```
Train on 60000 samples
Epoch 1/5
60000/60000 [==========] - 5s 85us/sample - loss: 1.7468 - accuracy: 0.4803
Epoch 2/5
60000/60000 [==========] - 5s 85us/sample - loss: 1.2168 - accuracy: 0.6512
Epoch 3/5
60000/60000 [==========] - 5s 85us/sample - loss: 0.9804 - accuracy: 0.6962
Epoch 4/5
60000/60000 [==========] - 5s 85us/sample - loss: 0.8597 - accuracy: 0.7269
Epoch 5/5
60000/60000 [==========] - 5s 84us/sample - loss: 0.7738 - accuracy: 0.7458
```

A medida que el modelo entrena, se muestran las métricas de `loss` y `accuracy`. Como vemos, este modelo alcanza una precisión de, aproximadamente, 0.7958 (o 79.5 %) en los datos de entrenamiento, pasando todas las imágenes por la red neuronal 5 veces (5 épocas, o *epochs*).

Evaluación y mejora del modelo

El siguiente paso es comparar el rendimiento del modelo en el conjunto de datos de prueba:

```
test_loss, test_acc = model.evaluate(test_images,  test_labels)

print('Test accuracy:', test_acc)
```

```
Test accuracy: 0.7463
```

Vemos que es aproximadamente la misma precisión que en los datos de entrenamiento. En siguientes capítulos trataremos en más detalles lo que implica que estos valores no coincidan.

Uso del modelo para hacer predicciones

Con el modelo entrenado, podemos empezar a usarlo para hacer predicciones sobre algunas imágenes (usemos por comodidad alguna de las imágenes de prueba que ya tenemos cargadas en el *notebook*).

```
predictions = model.predict(test_images)
```

En `predictions` se ha almacenado la predicción de la etiqueta para cada imagen en el conjunto de prueba. Echemos un vistazo a la primera predicción:

```
predictions[5]
```

```
array([5.5544176e-03, 9.3776268e-01, 6.8228887e-03, 1.0295090e-02,
       2.3263685e-02, 7.3104594e-03, 4.9446058e-03, 3.7988315e-03,
       1.2928306e-04, 1.1813057e-04], dtype=float32)
```

Puede ver qué etiqueta tiene el valor de confianza más alto con la función `argmax`:

```
np.argmax(predictions[5])
```

```
1
```

El modelo está más seguro de que esta imagen son unos pantalones *(Trouser)*. Al examinar la etiqueta que le corresponde muestra que esta clasificación es correcta:

```
test_labels[5]
```

```
1
```

Aprovechemos que este conjunto de datos es más rico visualmente para presentar gráficamente cómo de bien o de mal se comporta el modelo. Para ello, usaremos esta función extraída del tutorial de TensorFlow[101]:

[101] Véase https://www.tensorflow.org/tutorials/keras/classification [Consulta: 16/12/2019].

```
def plot_image(i, predictions_array, true_label, img):
  predictions_array, true_label, img = predictions_array,
  true_label[i], img[i]
  plt.grid(False)
  plt.xticks([])
  plt.yticks([])

  plt.imshow(img, cmap=plt.cm.binary)

  predicted_label = np.argmax(predictions_array)
  if predicted_label == true_label:
    color = 'blue'
  else:
    color = 'red'

  plt.xlabel("{} {:2.0f}% ({})".format(class_names[predicted_label],
                                100*np.max(predictions_array),
                                class_names[true_label]),
                                color=color)

def plot_value_array(i, predictions_array, true_label):
  predictions_array, true_label = predictions_array, true_label[i]
  plt.grid(False)
  plt.xticks(range(10))
  plt.yticks([])
  thisplot = plt.bar(range(10), predictions_array, color="#007700")
  plt.ylim([0, 1])
  predicted_label = np.argmax(predictions_array)

  thisplot[predicted_label].set_color('red')
  thisplot[true_label].set_color('black')
```

Las etiquetas de predicción correcta las pintaremos de negro (para que se pueda ver en la edición en blanco y negro) y las etiquetas de predicción incorrecta las colorearemos de rojo (gris en la edición en blanco y negro). El número da el porcentaje (de 100) para la etiqueta predicha.

```
i = 5
plt.figure(figsize=(6,3))
plt.subplot(1,2,1)
plot_image(i, predictions[i], test_labels, test_images)
plt.subplot(1,2,2)
plot_value_array(i, predictions[i], test_labels)
plt.show()
```

La Figura 5.7 muestra la salida de este código.

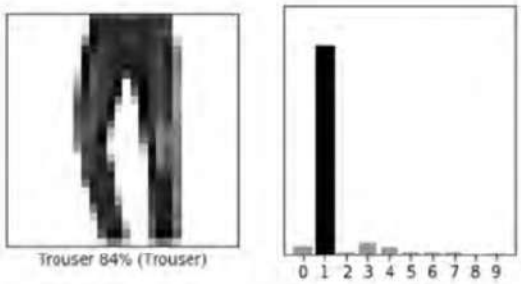

Figura 5.7 *Predicción del modelo para la imagen 5.*

```
i = 8
plt.figure(figsize=(6,3))
plt.subplot(1,2,1)
plot_image(i, predictions[i], test_labels, test_images)
plt.subplot(1,2,2)
plot_value_array(i, predictions[i],  test_labels)
plt.show()
```

La Figura 5.8 muestra la salida de este código.

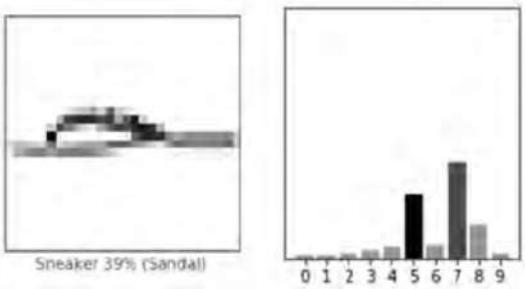

Figura 5.8 *Predicción del modelo para la imagen 8.*

Visualicemos varias imágenes con sus predicciones. Para ello, usaremos el código mostrado a continuación, cuya salida se representa en la Figura 5.9. Se debe tener en cuenta que el modelo puede estar equivocado, incluso cuando tiene mucha confianza en la clasificación sobre una de las clases.

```
num_rows = 7
num_cols = 2
num_images = num_rows*num_cols
plt.figure(figsize=(2*2*num_cols, 2*num_rows))
for i in range(num_images):
  plt.subplot(num_rows, 2*num_cols, 2*i+1)
  plot_image(i, predictions[i], test_labels, test_images)
  plt.subplot(num_rows, 2*num_cols, 2*i+2)
  plot_value_array(i, predictions[i], test_labels)
plt.tight_layout()
plt.show()
```

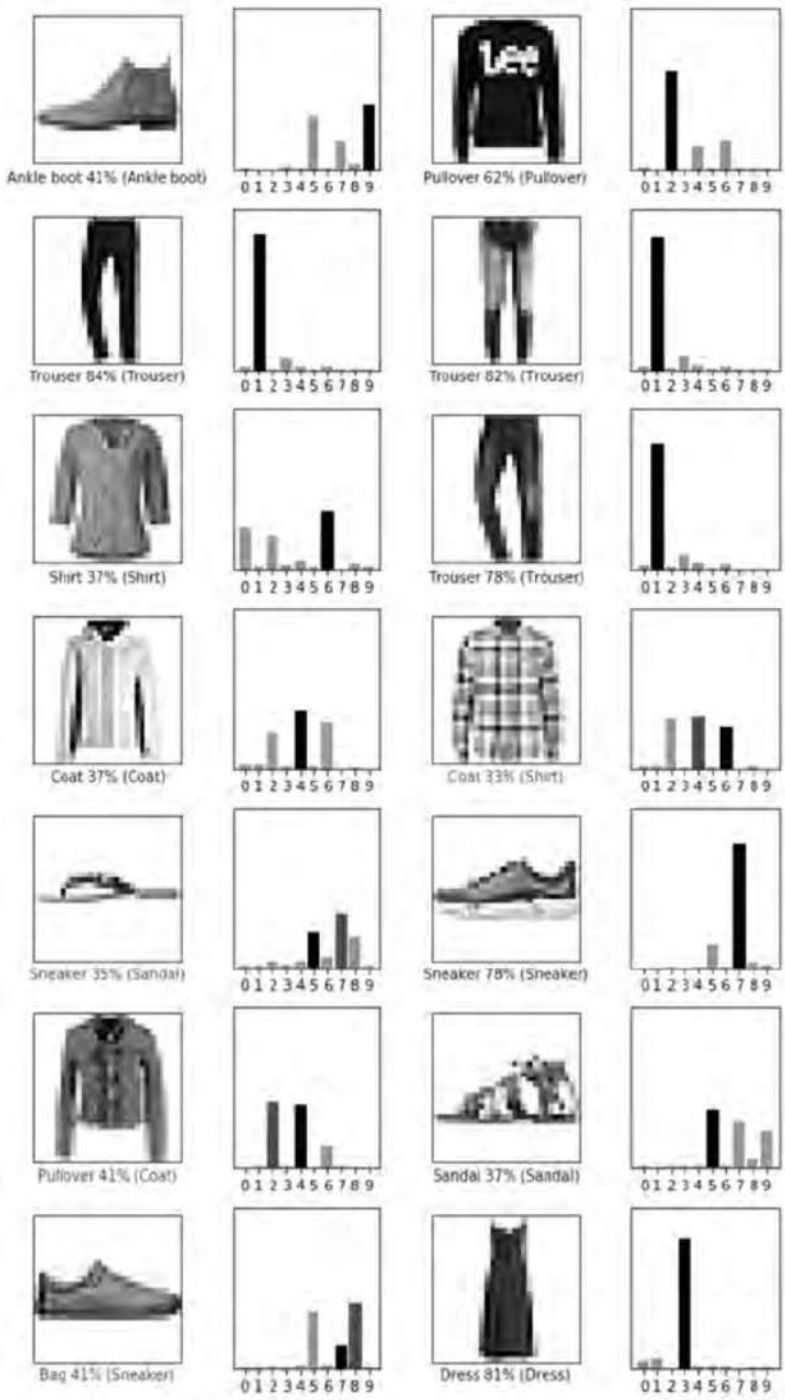

Figura 5.9 *Ejemplos de predicciones del modelo para el conjunto de datos Fashion-MNIST. Para cada imagen, el histograma representa la probabilidad de pertenencia de la clase correspondiente calculada por el modelo.*

Mejorar el modelo

Podemos observar que la precisión obtenida de este modelo para estos datos (75 %) dista mucho de la obtenida para el ejemplo previo de los dígitos (alrededor de un 86 %). Es decir, aunque este modelo fuera bueno para el problema de los dígitos MNIST, no lo es para clasificar los datos que nos ocupan.

Podríamos decir que es un resultado esperado, puesto que no hay una solución única para todos los problemas, sino que cada problema requiere su propia solución.

Intentemos, por ejemplo, cambiar el optimizador usado. Recordemos que el optimizador es el algoritmo usado por el modelo para actualizar los pesos de cada una de sus capas en el proceso de entrenamiento. Una elección bastante habitual es el optimizador sgd, pero hay muchos más, como por ejemplo el optimizador `Adam`, que a veces puede hacer converger mejor el proceso de optimización.

```
model =  Sequential()
model.add(Flatten(input_shape=(28, 28)))
model.add(Dense(10, activation='sigmoid'))
model.add(Dense(10, activation='softmax'))

model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

model.fit(train_images, train_labels, epochs=5)

test_loss, test_acc = model.evaluate(test_images,  test_labels)

print('\nTest accuracy:', test_acc)
```

```
Test accuracy: 0.8373
```

Como vemos, cambiando solo el optimizador ya hemos mejorado un 9 % adicional la precisión del modelo. Esto nos hace pensar que hay muchos elementos a tener en cuenta cuando definimos y configuramos el proceso de aprendizaje de una red neuronal. Lo cual nos ofrece motivación para los dos siguientes capítulos, en los que entraremos en más detalle en el proceso de aprendizaje y en todos los hiperparámetros que podemos configurar para un buen entrenamiento de la red. En el capítulo 8 veremos cómo podemos mejorar estos resultados de clasificación para el caso de imágenes usando redes neuronales convolucionales.