muinar06_act1_individual

October 26, 2024

1 REDES NEURONALES

En esta actividad vamos a utilizar una red neuronal para clasificar imágenes de dígitos del 0 al 9 escritos a mano. Para ello, utilizaremos Keras con TensorFlow.

El dataset a utilizar es MNIST, una base de datos constituida por (como no) imágenes de dígitos escritos a mano. Este dataset es ampliamente utilizado en docencia como punto de entrada al entrenamiento de redes neuronales y otros, pero también es muy utilizado en trabajos reales de investigación para el entrenamiento de imágenes. Puedes consultar más información sobre el dataset en este enlace.

El código utilizado para contestar tiene que quedar claramente reflejado en el Notebook. Puedes crear nuevas celdas si así lo deseas para estructurar tu código y sus salidas. A la hora de entregar el notebook, **asegúrate de que los resultados de ejecutar tu código han quedado guardados y que son perfectamente visibles en la versión PDF que debes entregar adjunta**. Por ejemplo, a la hora de entrenar una red neuronal tiene que verse claramente un log de los resultados de cada epoch.

[4]:
```
from keras.datasets.mnist import load_data
import tensorflow as tf
import matplotlib.pyplot as plt
from tensorflow.keras.layers import Dense, Dropout, BatchNormalization
from tensorflow.keras.regularizers import l2
```

Tenemos la suerte de que el dataset MNIST, el que vamos a utilizar en esta actividad, está guardado en Keras, por lo que podemos utilizarlo sin necesidad de buscar el dataset de forma externa.

[5]:
```
mnist = tf.keras.datasets.fashion_mnist
```

Llamar a **load_data** en este dataset nos dará dos conjuntos de dos listas, estos serán los valores de entrenamiento y prueba para los gráficos que contienen los dígitos y sus etiquetas.

Nota: Aunque en esta actividad lo veis de esta forma, también lo vais a poder encontrar como 4 variables de esta forma: training_images, training_labels, test_images, test_labels = mnist.load_data()

[6]:
```
(training_images, training_labels), (test_images, test_labels) = load_data()
```

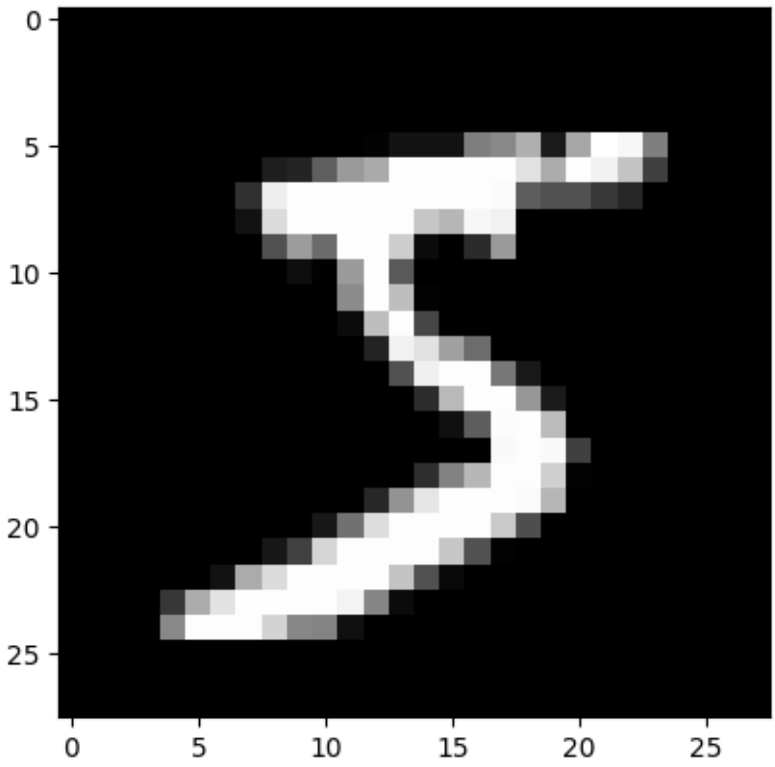
Antes de continuar vamos a dar un vistazo a nuestro dataset, para ello vamos a ver una imagen de entrenamiento y su etiqueta o clase.

[7]:

```python
import numpy as np
np.set_printoptions(linewidth=200)
plt.imshow(training_images[0], cmap="gray") # recordad que siempre es preferible trabajar en blanco y negro
#
print(training_labels[0])
print(training_images[0])
```

```
5
[[  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   3  18  18  18 126 136 175
26 166 255 247 127   0   0   0   0]
 [  0   0   0   0   0   0   0   0  30  36  94 154 170 253 253 253 253 253 225
172 253 242 195  64   0   0   0   0]
 [  0   0   0   0   0   0   0  49 238 253 253 253 253 253 253 253 253 251  93
82  82  56  39   0   0   0   0   0]
 [  0   0   0   0   0   0   0  18 219 253 253 253 253 253 198 182 247 241   0
0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0  80 156 107 253 253 205  11   0  43 154   0
0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0  14   1 154 253  90   0   0   0   0   0
0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0 139 253 190   2   0   0   0   0
0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0  11 190 253  70   0   0   0   0
0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0  35 241 225 160 108   1   0
0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0  81 240 253 253 119  25
0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0  45 186 253 253 150
27   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0  16  93 252 253
187   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0 249 253
249  64   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0  46 130 183 253 253
207   2   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0  39 148 229 253 253 253 250
```

```
182   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0  24 114 221 253 253 253 253 201  78
0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0  23  66 213 253 253 253 253 198  81   2   0
0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0  18 171 219 253 253 253 253 195  80   9   0   0   0
0   0   0   0   0   0   0   0   0]
 [  0   0   0   0  55 172 226 253 253 253 253 244 133  11   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0]
 [  0   0   0   0 136 253 253 253 212 135 132  16   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0]]
```



1.1 1. Información sobre el dataset

Una vez tenemos los datos cargados en memoria, vamos a obtener información sobre los mismos.

Pregunta 1.1 ***(0.25 puntos)*** ¿Cuántas imágenes hay de *training* y de *test*? ¿Qué tamaño tienen

las imágenes?

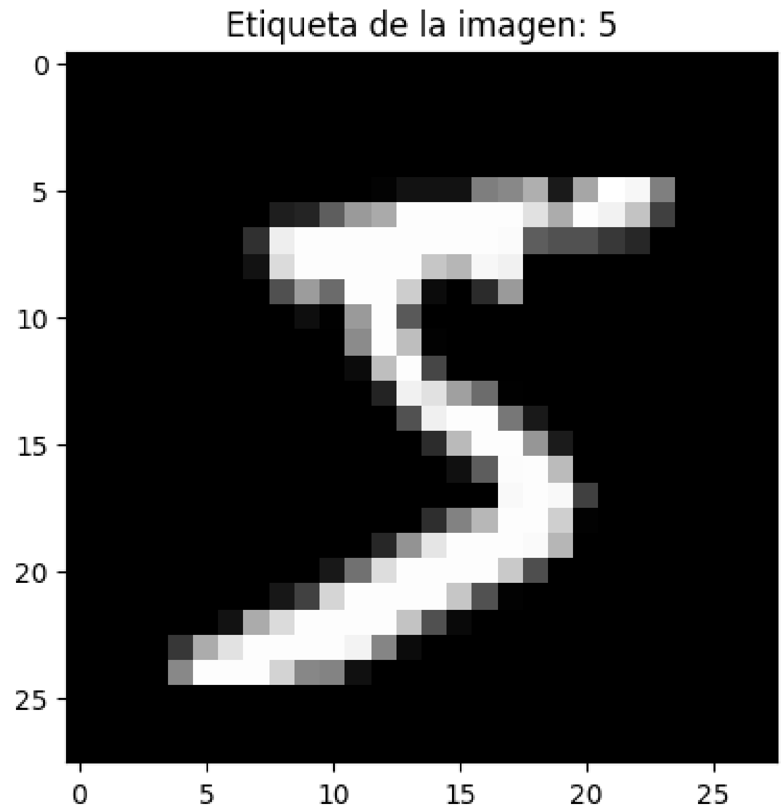
[8]:
```
num_training_images = training_images.shape[0]
num_test_images = test_images.shape[0]

image_shape = training_images.shape[1:]

print(f"Cantidad de imágenes de training: {num_training_images}")
print(f"Cantidad de imágenes de test: {num_test_images}")
print(f"Tamaño de las imágenes: {image_shape}")
```

```
Cantidad de imágenes de training: 60000
Cantidad de imágenes de test: 10000
Tamaño de las imágenes: (28, 28)
```

- La cantidad de imágenes en el conjunto de training es de 60,000.
- La cantidad de imágenes en el conjunto de test es de 10,000.
- Cada imagen tiene un tamaño de 28x28 píxeles. *Tu respuesta aquí*

Pregunta 1.2 ***(0.25 puntos)*** Realizar una exploración de las variables que contienen los datos. Describir en qué consiste un example del dataset (qué información se guarda en cada imagen) y describir qué contiene la información en y.

[9]:
```
example_image = training_images[0]
example_label = training_labels[0]

plt.imshow(example_image, cmap="gray")
plt.title(f"Etiqueta de la imagen: {example_label}")
plt.show()

print("Valores de los píxeles de la imagen:")
print(example_image)
print("\nEtiqueta asociada a la imagen:")
print(example_label)
```




```
Valores de los píxeles de la imagen:
[[  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   3  18  18  18 126 136 175
26 166 255 247 127   0   0   0   0]
 [  0   0   0   0   0   0   0   0  30  36  94 154 170 253 253 253 253 253 225
172 253 242 195  64   0   0   0   0]
 [  0   0   0   0   0   0   0  49 238 253 253 253 253 253 253 253 253 251  93
82  82  56  39   0   0   0   0   0]
 [  0   0   0   0   0   0   0  18 219 253 253 253 253 253 198 182 247 241   0
0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0  80 156 107 253 253 205  11   0  43 154   0
0   0   0   0   0   0   0   0   0]
```

```
 [  0   0   0   0   0   0   0   0   0  14   1 154 253  90   0   0   0   0   0
0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0 139 253 190   2   0   0   0   0
0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0  11 190 253  70   0   0   0   0
0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0  35 241 225 160 108   1   0
0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0  81 240 253 253 119  25
0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0  45 186 253 253 150
27   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0  16  93 252 253
187   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0 249 253
249  64   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0  46 130 183 253 253
207   2   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0  39 148 229 253 253 253 250
182   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0  24 114 221 253 253 253 253 201  78
0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0  23  66 213 253 253 253 253 198  81   2   0
0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0  18 171 219 253 253 253 253 195  80   9   0   0   0
0   0   0   0   0   0   0   0   0]
 [  0   0   0   0  55 172 226 253 253 253 253 244 133  11   0   0   0   0   0
0   0   0   0   0   0   0   0   0]
 [  0   0   0   0 136 253 253 253 212 135 132  16   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0]]

Etiqueta asociada a la imagen:
5
```

Cada imagen en el dataset MNIST está representada por una matriz de 28x28 píxeles, donde los valores indican la intensidad de los píxeles en escala de grises (de 0 a 255). En este caso, la imagen visualizada muestra un número 5, y los valores de la matriz reflejan las áreas blancas que forman el dígito.

La variable y (etiqueta) contiene el valor numérico que representa el dígito en la imagen. Para esta imagen, la etiqueta asociada es 5. Así, las imágenes contienen los datos visuales y las etiquetas proporcionan la clase correspondiente que el modelo utilizará para entrenarse.

1.2 2. Normalización y preprocesado de los datos

Pregunta 2.1 (0.25 puntos) Habreis notado que todos los valores numericos están entre 0 y 255. Si estamos entrenando una red neuronal, una buena practica es transformar todos los valores entre 0 y 1, un proceso llamado "normalización" y afortunadamente en Python es fácil normalizar una lista. ¿Cómo lo podemos hacer?

[10]:
```python
training_images = training_images / 255
test_images = test_images / 255

print(training_images[0])
```

```
[[0.         0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.         0.         0.
0.         0.
  0.         0.         0.         0.         0.         0.         0.
0.         0.         0.         0.        ]
 [0.         0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.         0.         0.
0.         0.
  0.         0.         0.         0.         0.         0.         0.
0.         0.         0.         0.        ]
 [0.         0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.         0.         0.
0.         0.
  0.         0.         0.         0.         0.         0.         0.
0.         0.         0.         0.        ]
 [0.         0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.         0.         0.
0.         0.
  0.         0.         0.         0.         0.         0.         0.
0.         0.         0.         0.        ]
 [0.         0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.         0.         0.
0.         0.
  0.         0.         0.         0.         0.         0.         0.
0.         0.         0.         0.        ]
 [0.         0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.01176471 0.07058824
0.07058824 0.07058824 0.49411765
  0.53333333 0.68627451 0.10196078 0.65098039 1.         0.96862745 0.49803922
0.         0.         0.         0.        ]
 [0.         0.         0.         0.         0.         0.         0.
0.         0.11764706 0.14117647 0.36862745 0.60392157 0.66666667 0.99215686
0.99215686 0.99215686 0.99215686
  0.99215686 0.88235294 0.6745098  0.99215686 0.94901961 0.76470588 0.25098039
0.         0.         0.         0.        ]
 [0.         0.         0.         0.         0.         0.         0.
0.19215686 0.93333333 0.99215686 0.99215686 0.99215686 0.99215686 0.99215686
```

```
0.99215686 0.99215686 0.99215686
  0.98431373 0.36470588 0.32156863 0.32156863 0.21960784 0.15294118 0.
0.         0.         0.         0.        ]
 [0.         0.         0.         0.         0.         0.         0.
0.07058824 0.85882353 0.99215686 0.99215686 0.99215686 0.99215686 0.99215686
0.77647059 0.71372549 0.96862745
  0.94509804 0.         0.         0.         0.         0.         0.
0.         0.         0.         0.        ]
 [0.         0.         0.         0.         0.         0.         0.
0.         0.31372549 0.61176471 0.41960784 0.99215686 0.99215686 0.80392157
0.04313725 0.         0.16862745
  0.60392157 0.         0.         0.         0.         0.         0.
0.         0.         0.         0.        ]
 [0.         0.         0.         0.         0.         0.         0.
0.         0.         0.05490196 0.00392157 0.60392157 0.99215686 0.35294118 0.
0.         0.
  0.         0.         0.         0.         0.         0.         0.
0.         0.         0.         0.        ]
 [0.         0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.54509804 0.99215686 0.74509804
0.00784314 0.         0.
  0.         0.         0.         0.         0.         0.         0.
0.         0.         0.         0.        ]
 [0.         0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.04313725 0.74509804 0.99215686
0.2745098  0.         0.
  0.         0.         0.         0.         0.         0.         0.
0.         0.         0.         0.        ]
 [0.         0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.1372549  0.94509804
0.88235294 0.62745098 0.42352941
  0.00392157 0.         0.         0.         0.         0.         0.
0.         0.         0.         0.        ]
 [0.         0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.         0.31764706
0.94117647 0.99215686 0.99215686
  0.46666667 0.09803922 0.         0.         0.         0.         0.
0.         0.         0.         0.        ]
 [0.         0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.         0.
0.17647059 0.72941176 0.99215686
  0.99215686 0.58823529 0.10588235 0.         0.         0.         0.
0.         0.         0.         0.        ]
 [0.         0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.         0.         0.
0.0627451  0.36470588
  0.98823529 0.99215686 0.73333333 0.         0.         0.         0.
0.         0.         0.         0.        ]
```

```
 [0.         0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.         0.         0.
0.         0.
  0.97647059 0.99215686 0.97647059 0.25098039 0.         0.         0.
0.         0.         0.         0.        ]
 [0.         0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.         0.
0.18039216 0.50980392 0.71764706
  0.99215686 0.99215686 0.81176471 0.00784314 0.         0.         0.
0.         0.         0.         0.        ]
 [0.         0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.15294118 0.58039216
0.89803922 0.99215686 0.99215686
  0.99215686 0.98039216 0.71372549 0.         0.         0.         0.
0.         0.         0.         0.        ]
 [0.         0.         0.         0.         0.         0.         0.
0.         0.         0.         0.09411765 0.44705882 0.86666667 0.99215686
0.99215686 0.99215686 0.99215686
  0.78823529 0.30588235 0.         0.         0.         0.         0.
0.         0.         0.         0.        ]
 [0.         0.         0.         0.         0.         0.         0.
0.         0.09019608 0.25882353 0.83529412 0.99215686 0.99215686 0.99215686
0.99215686 0.77647059 0.31764706
  0.00784314 0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.        ]
 [0.         0.         0.         0.         0.         0.         0.07058824
0.67058824 0.85882353 0.99215686 0.99215686 0.99215686 0.99215686 0.76470588
0.31372549 0.03529412 0.
  0.         0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.        ]
 [0.         0.         0.         0.         0.21568627 0.6745098  0.88627451
0.99215686 0.99215686 0.99215686 0.99215686 0.95686275 0.52156863 0.04313725 0.
0.         0.
  0.         0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.        ]
 [0.         0.         0.         0.         0.53333333 0.99215686 0.99215686
0.99215686 0.83137255 0.52941176 0.51764706 0.0627451  0.         0.         0.
0.         0.
  0.         0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.        ]
 [0.         0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.         0.         0.
0.         0.
  0.         0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.        ]
 [0.         0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.         0.         0.
0.         0.
```

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

Pregunta 2.2 (0.25 puntos) Utiliza la función ***reshape*** de Numpy para convertir las imágenes en vectores de características de un tamaño de (N, 784). Explica con tus palabras por qué es necesario hacer esto.

[11]:
```
training_images = training_images.reshape(-1, 28 * 28)
test_images = test_images.reshape(-1, 28 * 28)

print(f"Nueva forma de training_images: {training_images.shape}")
print(f"Nueva forma de test_images: {test_images.shape}")
```

```
Nueva forma de training_images: (60000, 784)
Nueva forma de test_images: (10000, 784)
```

Es necesario usar reshape para convertir las imágenes de 28x28 en vectores de 784 elementos porque las redes neuronales no entienden bien las imágenes en formato de matriz. Ellas prefieren trabajar con listas largas (vectores), donde cada número representa un píxel de la imagen. Al aplanarlas, logramos que la red pueda procesar mejor la información y aprender más fácilmente. Básicamente, le damos a la red los datos de una manera que puede entender mejor.

Pregunta 2.3 (0.25 puntos) Para facilitar el desarrollo de la actividad, vamos a expresar las etiquetas así:

[12]:
```
training_labels = tf.keras.utils.to_categorical(training_labels)
test_labels = tf.keras.utils.to_categorical(test_labels)
```

Muestra cómo son ahora los datos, como resultado de este cambio y también de los realizados en las dos preguntas anteriores. Debate cómo se beneficiará la red neuronal de todos estos cambios.

[13]:
```
print(f"Ejemplo de etiquetas de entrenamiento después de la conversión a␣
 ↪categóricas:\n{training_labels[:5]}")
print(f"\nEjemplo de etiquetas de prueba después de la conversión a categóricas:
 ↪\n{test_labels[:5]}")
print(f"\nForma de los datos de entrenamiento después de la normalización y␣
 ↪reshape:\n{training_images.shape}")
print(f"\nForma de las etiquetas de entrenamiento después de la conversión a␣
 ↪categóricas:\n{training_labels.shape}")
```

```
Ejemplo de etiquetas de entrenamiento después de la conversión a categóricas:
[[0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
 [1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
```

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

Ejemplo de etiquetas de prueba después de la conversión a categóricas:
[[0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]
 [0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]]

Forma de los datos de entrenamiento después de la normalización y reshape:
(60000, 784)

Forma de las etiquetas de entrenamiento después de la conversión a categóricas:
(60000, 10)
```

[14]:
```
training_images.shape[1]
```

[14]:
```
784
```

Al normalizar los datos, evitamos que grandes diferencias en los valores de los píxeles afecten el aprendizaje, lo que hace que la red sea más eficiente y estable. Al convertir las imágenes en vectores, la red puede procesarlas de manera lineal, como lo espera, y con la transformación de las etiquetas a formato categórico, facilitamos que la red use funciones de activación como softmax para predecir la clase correcta. Todos estos cambios mejoran la precisión y aceleran el proceso de aprendizaje de la red neuronal, optimizando su rendimiento en la tarea de clasificar los dígitos.

1.3 3. Creación del Modelo

Ahora vamos a definir el modelo, pero antes vamos a repasar algunos comandos y conceptos muy útiles: * **Sequential**: Eso define una SECUENCIA de capas en la red neuronal * **Dense**: Añade una capa de neuronas * **Flatten**: ¿Recuerdas cómo eran las imágenes cuando las imprimiste para poder verlas? Un cuadrado, Flatten toma ese cuadrado y lo convierte en un vector de una dimensión.

Cada capa de neuronas necesita una función de activación. Normalmente se usa la función relu en las capas intermedias y softmax en la ultima capa (en problemas de clasificación de más de dos items) * **Relu** significa que "Si X>0 devuelve X, si no, devuelve 0", así que lo que hace es pasar sólo valores 0 o mayores a la siguiente capa de la red. * **Softmax** toma un conjunto de valores, y escoge el más grande.

Pregunta 3.1 (0.5 puntos). Utilizando Keras, y preparando los datos de X e Y como fuera necesario, define y entrena una red neuronal que sea capaz de clasificar imágenes de MNIST con las siguientes características:

- Una capa de entrada del tamaño adecuado.
- Una capa oculta de 512 neuronas.
- Una capa final con 10 salidas.

[15]:
```
import tensorflow as tf
```

```python
# Definir el modelo secuencial con una capa oculta de 512 neuronas y activación␣
 ↪'relu'
model = tf.keras.models.Sequential([
    tf.keras.layers.Dense(512, activation='relu', input_shape=(training_images.
 ↪shape[1],)),
    tf.keras.layers.Dense(10, activation='softmax')  # Capa de salida con 10␣
 ↪neuronas, una por cada clase
])

# Optimizador 'adam' y categorical_crossentropy como pérdida
model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# Entrenar el modelo con los datos de entrenamiento
model.fit(training_images, training_labels, epochs=20)

# Evaluar el modelo con los datos de prueba y mostrar la precisión
test_loss, test_acc = model.evaluate(test_images, test_labels)
print(f'\nPrecisión en los datos de prueba: {test_acc}')
```

```
c:\Users\contr\.conda\envs\env_deep\Lib\site-
packages\keras\src\layers\core\dense.py:87: UserWarning: Do not pass an
`input_shape`/`input_dim` argument to a layer. When using Sequential models,
prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

```
Epoch 1/20
1875/1875             9s 4ms/step -
accuracy: 0.8954 - loss: 0.3432
Epoch 2/20
1875/1875             7s 4ms/step -
accuracy: 0.9755 - loss: 0.0806
Epoch 3/20
1875/1875             6s 3ms/step -
accuracy: 0.9843 - loss: 0.0510
Epoch 4/20
1875/1875             8s 4ms/step -
accuracy: 0.9890 - loss: 0.0348
Epoch 5/20
1875/1875             10s 5ms/step -
accuracy: 0.9913 - loss: 0.0263
Epoch 6/20
1875/1875             8s 4ms/step -
accuracy: 0.9946 - loss: 0.0170
Epoch 7/20
1875/1875             9s 5ms/step -
accuracy: 0.9957 - loss: 0.0144
```

```
Epoch 8/20
1875/1875            10s 5ms/step -
accuracy: 0.9959 - loss: 0.0128
Epoch 9/20
1875/1875            10s 5ms/step -
accuracy: 0.9967 - loss: 0.0101
Epoch 10/20
1875/1875             9s 5ms/step -
accuracy: 0.9971 - loss: 0.0088
Epoch 11/20
1875/1875             7s 4ms/step -
accuracy: 0.9977 - loss: 0.0074
Epoch 12/20
1875/1875             7s 4ms/step -
accuracy: 0.9979 - loss: 0.0063
Epoch 13/20
1875/1875             7s 4ms/step -
accuracy: 0.9980 - loss: 0.0060
Epoch 14/20
1875/1875             7s 4ms/step -
accuracy: 0.9979 - loss: 0.0070
Epoch 15/20
1875/1875             7s 4ms/step -
accuracy: 0.9980 - loss: 0.0069
Epoch 16/20
1875/1875             7s 4ms/step -
accuracy: 0.9975 - loss: 0.0072
Epoch 17/20
1875/1875            11s 6ms/step -
accuracy: 0.9984 - loss: 0.0046
Epoch 18/20
1875/1875            11s 6ms/step -
accuracy: 0.9987 - loss: 0.0040
Epoch 19/20
1875/1875             7s 4ms/step -
accuracy: 0.9973 - loss: 0.0081
Epoch 20/20
1875/1875             7s 4ms/step -
accuracy: 0.9987 - loss: 0.0037
313/313              0s 1ms/step -
accuracy: 0.9753 - loss: 0.1449

Precisión en los datos de prueba: 0.9786999821662903
```

Pregunta 3.2 (0.25 puntos): ¿crees conveniente utilizar una capa flatten en este caso? Motiva tu respuesta.

[16]:
```
### Tu código para incluir una capa flatten si lo ves necesario ###
```

Dado que ya he usado la función reshape para aplanar las imágenes de formato (28, 28) a un vector de 784 características en la pregunta 2.2, no es necesario utilizar la capa Flatten en este caso. La operación de reshape ya cumple con la misma función que Flatten haría dentro del modelo.

Flatten tiene sentido cuando las imágenes están en formato bidimensional, pero dado que ya están convertidas en vectores de 784 elementos, el uso de Flatten sería redundante.

Pregunta 3.3 (0.25 puntos): Utiliza la función summary() para mostrar la estructura de tu modelo.

[17]:
```
model.summary()
```

```
Model: "sequential"
```

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 512)	401,920
dense_1 (Dense)	(None, 10)	5,130

```
Total params: 1,221,152 (4.66 MB)

Trainable params: 407,050 (1.55 MB)

Non-trainable params: 0 (0.00 B)

Optimizer params: 814,102 (3.11 MB)
```

1.4 4: Compilación y entrenamiento

Pregunta 4.1 (0.5 puntos): Compila tu modelo. Utiliza ***categorical_crossentropy*** como función de pérdida, ***Adam*** como optimizador, y monitoriza la ***tasa de acierto*** durante el entrenamiento. Explica qué hace cada cosa en la compilación.

[18]:
```
# Cabe mencionar que ya compilé el modelo previamente con estos requerimientos,
# pero como te lo piden nuevamente, lo defnifiré para explicar los parámetros.
model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

- **categorical_crossentropy (función de pérdida):** Es ideal para clasificación multiclase, como en MNIST. Mide la diferencia entre las predicciones del modelo y las etiquetas verdaderas. El objetivo es minimizar esta pérdida ajustando los pesos del modelo.

- **Adam (optimizador):** Es un optimizador eficiente que ajusta los pesos del modelo usando los gradientes. Combina lo mejor de AdaGrad y RMSProp, lo que lo hace rápido y eficaz para muchos problemas.
- **accuracy (métrica):** Monitorea el porcentaje de aciertos del modelo durante el entrenamiento, lo que ayuda a saber qué tan bien está prediciendo las clases correctas.

Pregunta 4.2 (0.5 puntos): Utiliza la función ***fit()*** para entrenar tu modelo. Para ayudarte en tu primer entrenamiento, utiliza estos valores: * epochs = 5 * batch_size = 32 * validation_split = 0.25

[19]:
```
# Este sería el segundo entrenamiento del modelo, pero como ya se pidió␣
 ↪entrenarlo en la pregunta 3.1
model.fit(training_images, training_labels,
          epochs=5,
          batch_size=32,
          validation_split=0.25)

test_loss, test_acc = model.evaluate(test_images, test_labels)
print(f'\nPrecisión en los datos de prueba: {test_acc}')
```

```
Epoch 1/5
1407/1407             7s 4ms/step -
accuracy: 0.9979 - loss: 0.0062 - val_accuracy: 0.9971 - val_loss: 0.0082
Epoch 2/5
1407/1407             6s 4ms/step -
accuracy: 0.9990 - loss: 0.0023 - val_accuracy: 0.9965 - val_loss: 0.0133
Epoch 3/5
1407/1407             6s 4ms/step -
accuracy: 0.9986 - loss: 0.0048 - val_accuracy: 0.9937 - val_loss: 0.0241
Epoch 4/5
1407/1407             6s 4ms/step -
accuracy: 0.9984 - loss: 0.0047 - val_accuracy: 0.9963 - val_loss: 0.0119
Epoch 5/5
1407/1407             6s 5ms/step -
accuracy: 0.9985 - loss: 0.0056 - val_accuracy: 0.9961 - val_loss: 0.0142
313/313             0s 1ms/step -
accuracy: 0.9785 - loss: 0.1657

Precisión en los datos de prueba: 0.9815999865531921
```

2 5: Impacto al variar el número de neuronas en las capas ocultas

En este ejercicio vamos a experimentar con nuestra red neuronal cambiando el numero de neuronas por 512 y por otros valores. Para ello, utiliza la red neuronal de la pregunta 3, y su capa oculta cambia el número de neuronas:

- **216 neuronas en la capa oculta
- **1024 neuronas en la capa oculta

y entrena la red en ambos casos.

[20]:
```
# Definir el modelo con 216 neuronas en la capa oculta
model_216 = tf.keras.models.Sequential([
    tf.keras.layers.Dense(216, activation='relu', input_shape=(training_images.
 ↪shape[1],)),
    tf.keras.layers.Dense(10, activation='softmax')  # Capa de salida con 10␣
 ↪neuronas
])

model_216.compile(optimizer='adam',
                  loss='categorical_crossentropy',
                  metrics=['accuracy'])

model_216.fit(training_images, training_labels,
              epochs=5,
              batch_size=32,
              validation_split=0.25)

test_loss, test_acc = model.evaluate(test_images, test_labels)
print(f'\nPrecisión en los datos de prueba: {test_acc}')
```

```
Epoch 1/5
1407/1407             5s 3ms/step -
accuracy: 0.8779 - loss: 0.4352 - val_accuracy: 0.9548 - val_loss: 0.1503
Epoch 2/5
1407/1407             4s 3ms/step -
accuracy: 0.9636 - loss: 0.1223 - val_accuracy: 0.9663 - val_loss: 0.1133
Epoch 3/5
1407/1407             4s 3ms/step -
accuracy: 0.9773 - loss: 0.0750 - val_accuracy: 0.9666 - val_loss: 0.1101
Epoch 4/5
1407/1407             3s 2ms/step -
accuracy: 0.9851 - loss: 0.0518 - val_accuracy: 0.9692 - val_loss: 0.1011
Epoch 5/5
1407/1407             3s 2ms/step -
accuracy: 0.9892 - loss: 0.0368 - val_accuracy: 0.9742 - val_loss: 0.0917
313/313             0s 1ms/step -
accuracy: 0.9785 - loss: 0.1657

Precisión en los datos de prueba: 0.9815999865531921
```

[21]:
```
# Definir el modelo con 1024 neuronas en la capa oculta
model_1024 = tf.keras.models.Sequential([
    tf.keras.layers.Dense(1024, activation='relu', input_shape=(training_images.
 ↪shape[1],)),
    tf.keras.layers.Dense(10, activation='softmax')  # Capa de salida con 10␣
 ↪neuronas
```

```
])

model_1024.compile(optimizer='adam',
                   loss='categorical_crossentropy',
                   metrics=['accuracy'])

model_1024.fit(training_images, training_labels,
               epochs=5,
               batch_size=32,
               validation_split=0.25)

test_loss, test_acc = model.evaluate(test_images, test_labels)
print(f'\nPrecisión en los datos de prueba: {test_acc}')
```

```
Epoch 1/5
1407/1407             15s 10ms/step -
accuracy: 0.8941 - loss: 0.3535 - val_accuracy: 0.9627 - val_loss: 0.1250
Epoch 2/5
1407/1407             13s 9ms/step -
accuracy: 0.9764 - loss: 0.0793 - val_accuracy: 0.9686 - val_loss: 0.1035
Epoch 3/5
1407/1407             16s 11ms/step -
accuracy: 0.9841 - loss: 0.0504 - val_accuracy: 0.9737 - val_loss: 0.0916
Epoch 4/5
1407/1407             15s 10ms/step -
accuracy: 0.9908 - loss: 0.0312 - val_accuracy: 0.9697 - val_loss: 0.1017
Epoch 5/5
1407/1407             14s 10ms/step -
accuracy: 0.9921 - loss: 0.0250 - val_accuracy: 0.9758 - val_loss: 0.0885
313/313             0s 1ms/step -
accuracy: 0.9785 - loss: 0.1657

Precisión en los datos de prueba: 0.9815999865531921
```

Pregunta 5.1 (0.5 puntos): ¿Cual es el impacto que tiene la red neuronal?

El impacto de variar el número de neuronas en la capa oculta es que, con 216, el entrenamiento es más rápido, mientras que con 1024 tarda mucho más, aunque la precisión final es casi la misma en ambos casos. Más neuronas no siempre significan mejor rendimiento. Quizá, si los datos fueran más complejos o si el modelo tuviera más capas, añadir más neuronas tendría un impacto positivo en la precisión. Pero en este caso, parece que un número más alto solo hace que el entrenamiento sea más lento sin una mejora significativa en el resultado.

3 6: Número de neuronas de la capa de salida

Considerad la capa final, la de salida de la red neuronal de la pregunta 3.

Pregunta 6.1 (0.25 puntos): ¿Por qué son 10 las neuronas de la última capa?

Pregunta 6.2 (0.25 puntos): ¿Qué pasaría si tuvieras una cantidad diferente a 10?

Por ejemplo, intenta entrenar la red con 5, para ello utiliza la red neuronal de la pregunta 1 y cambia a 5 el número de neuronas en la última capa.

[22]:
```
model = tf.keras.models.Sequential([
    tf.keras.layers.Dense(512, activation='relu', input_shape=(training_images.
 ↪shape[1],)),
    tf.keras.layers.Dense(5, activation='softmax')  # Capa de salida con 5␣
 ↪neuronas
])

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

history = model.fit(training_images, training_labels,
                    epochs=5,
                    batch_size=32,
                    validation_split=0.25)

test_loss, test_acc = model.evaluate(test_images, test_labels)
print(f'\nPrecisión en los datos de prueba: {test_acc}')
```

```
Epoch 1/5
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[22], line 10
      1 model = tf.keras.models.Sequential([
      2     tf.keras.layers.Dense(512, activation='relu',␣
 ↪input_shape=(training_images.shape[1],)),
      3     tf.keras.layers.Dense(5, activation='softmax')  # Capa de salida co␣
 ↪5 neuronas
      4 ])
      6 model.compile(optimizer='adam',
      7               loss='categorical_crossentropy',
      8               metrics=['accuracy'])
---> 10 history = model.fit(training_images, training_labels,
     11                     epochs=5,
     12                     batch_size=32,
     13                     validation_split=0.25)
     15 test_loss, test_acc = model.evaluate(test_images, test_labels)
     16 print(f'\nPrecisión en los datos de prueba: {test_acc}')

File c:\Users\contr\.
 ↪conda\envs\env_deep\Lib\site-packages\keras\src\utils\traceback_utils.py:122,␣
 ↪in filter_traceback.<locals>.error_handler(*args, **kwargs)
    119     filtered_tb = _process_traceback_frames(e.__traceback__)
```

```
    120     # To get the full stack trace, call:
    121     # `keras.config.disable_traceback_filtering()`
--> 122     raise e.with_traceback(filtered_tb) from None
    123 finally:
    124     del filtered_tb

File c:\Users\contr\.
 ↪conda\envs\env_deep\Lib\site-packages\keras\src\backend\tensorflow\nn.py:587,
 ↪in categorical_crossentropy(target, output, from_logits, axis)
    585 for e1, e2 in zip(target.shape, output.shape):
    586     if e1 is not None and e2 is not None and e1 != e2:
--> 587         raise ValueError(
    588             "Arguments `target` and `output` must have the same shape.
    589             "Received: "
    590             f"target.shape={target.shape}, output.shape={output.shape}"
    591         )
    593 output, from_logits = _get_logits(
    594     output, from_logits, "Softmax", "categorical_crossentropy"
    595 )
    596 if from_logits:

ValueError: Arguments `target` and `output` must have the same shape. Received:
 ↪target.shape=(None, 10), output.shape=(None, 5)
```

Tu respuestas a la pregunta 6.1 aquí: La capa de salida tiene 10 neuronas porque estamos clasificando 10 dígitos (0 al 9), por lo que cada neurona representa una clase.

Tu respuestas a la pregunta 6.2 aquí: Si cambiamos el número de neuronas a un valor diferente, como 5, el modelo no podrá clasificar correctamente los 10 dígitos. En este caso, el error se debe a que las etiquetas originales contienen 10 clases y el modelo, al tener solo 5 neuronas de salida, no es capaz de interpretar todas las categorías, lo cual es necesario para esta tarea de clasificación.

4 7: Aumento de epoch y su efecto en la red neuronal

En este ejercicio vamos a ver el impacto de aumentar los epoch en el entrenamiento. Usando la red neuronal de la pregunta 3:

Pregunta 7.1 (0.25 puntos) * Intentad 15 epoch para su entrenamiento, probablemente obtendras un modelo con una pérdida mucho mejor que el que tiene 5.

Pregunta 7.2 (0.25 puntos) * Intenta ahora con 30 epoch para su entrenamiento.

Pregunta 7.3 (0.25 puntos) * ¿Qué está pasando en la pregunta anterior? Explica tu respuesta y da el nombre de este efecto si lo conoces.

[42]:
```
model_15_epochs = tf.keras.models.Sequential([
    tf.keras.layers.Dense(512, activation='relu', input_shape=(training_images.
 ↪shape[1],)),
    tf.keras.layers.Dense(10, activation='softmax')
```

```
])

model_15_epochs.compile(optimizer='adam',
                        loss='categorical_crossentropy',
                        metrics=['accuracy'])

model_15_epochs.fit(training_images, training_labels,
                    epochs=15, # Entrenar el modelo con 15 epochs
                    batch_size=32,
                    validation_split=0.25)

test_loss, test_acc = model_15_epochs.evaluate(test_images, test_labels)
print(f'\nPrecisión en los datos de prueba con 15 epochs: {test_acc}')
```

```
Epoch 1/15
1407/1407             9s 6ms/step -
accuracy: 0.8869 - loss: 0.3885 - val_accuracy: 0.9611 - val_loss: 0.1291
Epoch 2/15
1407/1407             7s 5ms/step -
accuracy: 0.9726 - loss: 0.0918 - val_accuracy: 0.9691 - val_loss: 0.0991
Epoch 3/15
1407/1407             8s 5ms/step -
accuracy: 0.9835 - loss: 0.0546 - val_accuracy: 0.9722 - val_loss: 0.0894
Epoch 4/15
1407/1407             8s 6ms/step -
accuracy: 0.9889 - loss: 0.0373 - val_accuracy: 0.9733 - val_loss: 0.0962
Epoch 5/15
1407/1407             7s 5ms/step -
accuracy: 0.9907 - loss: 0.0290 - val_accuracy: 0.9734 - val_loss: 0.0961
Epoch 6/15
1407/1407             6s 4ms/step -
accuracy: 0.9931 - loss: 0.0208 - val_accuracy: 0.9718 - val_loss: 0.1077
Epoch 7/15
1407/1407             6s 4ms/step -
accuracy: 0.9951 - loss: 0.0163 - val_accuracy: 0.9753 - val_loss: 0.0980
Epoch 8/15
1407/1407             6s 5ms/step -
accuracy: 0.9972 - loss: 0.0099 - val_accuracy: 0.9654 - val_loss: 0.1530
Epoch 9/15
1407/1407             14s 7ms/step -
accuracy: 0.9958 - loss: 0.0136 - val_accuracy: 0.9718 - val_loss: 0.1256
Epoch 10/15
1407/1407             9s 6ms/step -
accuracy: 0.9959 - loss: 0.0128 - val_accuracy: 0.9779 - val_loss: 0.1019
Epoch 11/15
1407/1407             8s 6ms/step -
accuracy: 0.9987 - loss: 0.0042 - val_accuracy: 0.9749 - val_loss: 0.1173
Epoch 12/15
```

```
1407/1407             7s 5ms/step -
accuracy: 0.9978 - loss: 0.0080 - val_accuracy: 0.9760 - val_loss: 0.1169
Epoch 13/15
1407/1407             8s 5ms/step -
accuracy: 0.9972 - loss: 0.0084 - val_accuracy: 0.9758 - val_loss: 0.1165
Epoch 14/15
1407/1407             10s 7ms/step -
accuracy: 0.9983 - loss: 0.0052 - val_accuracy: 0.9797 - val_loss: 0.1082
Epoch 15/15
1407/1407             9s 6ms/step -
accuracy: 0.9976 - loss: 0.0076 - val_accuracy: 0.9775 - val_loss: 0.1194
313/313             0s 1ms/step -
accuracy: 0.9764 - loss: 0.1203

Precisión en los datos de prueba con 15 epochs: 0.9794999957084656
```

[43]:
```
model_30_epochs = tf.keras.models.Sequential([
    tf.keras.layers.Dense(512, activation='relu', input_shape=(training_images.
  ↪shape[1],)),
    tf.keras.layers.Dense(10, activation='softmax')
])

model_30_epochs.compile(optimizer='adam',
                        loss='categorical_crossentropy',
                        metrics=['accuracy'])

model_30_epochs.fit(training_images, training_labels,
                    epochs=30, # Entrenar el modelo con 30 epochs
                    batch_size=32,
                    validation_split=0.25)

test_loss, test_acc = model_30_epochs.evaluate(test_images, test_labels)
print(f'\nPrecisión en los datos de prueba con 30 epochs: {test_acc}')
```

```
Epoch 1/30
1407/1407             8s 5ms/step -
accuracy: 0.8892 - loss: 0.3822 - val_accuracy: 0.9618 - val_loss: 0.1264
Epoch 2/30
1407/1407             6s 4ms/step -
accuracy: 0.9722 - loss: 0.0942 - val_accuracy: 0.9677 - val_loss: 0.1008
Epoch 3/30
1407/1407             6s 4ms/step -
accuracy: 0.9839 - loss: 0.0553 - val_accuracy: 0.9689 - val_loss: 0.0984
Epoch 4/30
1407/1407             5s 4ms/step -
accuracy: 0.9895 - loss: 0.0353 - val_accuracy: 0.9745 - val_loss: 0.0902
Epoch 5/30
1407/1407             6s 4ms/step -
```

```
accuracy: 0.9914 - loss: 0.0272 - val_accuracy: 0.9728 - val_loss: 0.0925
Epoch 6/30
1407/1407            7s 5ms/step -
accuracy: 0.9938 - loss: 0.0200 - val_accuracy: 0.9775 - val_loss: 0.0839
Epoch 7/30
1407/1407            6s 4ms/step -
accuracy: 0.9942 - loss: 0.0185 - val_accuracy: 0.9772 - val_loss: 0.0879
Epoch 8/30
1407/1407            6s 5ms/step -
accuracy: 0.9971 - loss: 0.0101 - val_accuracy: 0.9754 - val_loss: 0.1000
Epoch 9/30
1407/1407            6s 4ms/step -
accuracy: 0.9966 - loss: 0.0102 - val_accuracy: 0.9764 - val_loss: 0.0931
Epoch 10/30
1407/1407            6s 5ms/step -
accuracy: 0.9978 - loss: 0.0081 - val_accuracy: 0.9740 - val_loss: 0.1152
Epoch 11/30
1407/1407            8s 5ms/step -
accuracy: 0.9971 - loss: 0.0103 - val_accuracy: 0.9773 - val_loss: 0.1064
Epoch 12/30
1407/1407            6s 4ms/step -
accuracy: 0.9975 - loss: 0.0072 - val_accuracy: 0.9766 - val_loss: 0.1114
Epoch 13/30
1407/1407            7s 5ms/step -
accuracy: 0.9982 - loss: 0.0056 - val_accuracy: 0.9737 - val_loss: 0.1331
Epoch 14/30
1407/1407            7s 5ms/step -
accuracy: 0.9976 - loss: 0.0069 - val_accuracy: 0.9766 - val_loss: 0.1159
Epoch 15/30
1407/1407            5s 4ms/step -
accuracy: 0.9979 - loss: 0.0060 - val_accuracy: 0.9793 - val_loss: 0.1056
Epoch 16/30
1407/1407            6s 4ms/step -
accuracy: 0.9986 - loss: 0.0048 - val_accuracy: 0.9767 - val_loss: 0.1230
Epoch 17/30
1407/1407            6s 4ms/step -
accuracy: 0.9966 - loss: 0.0088 - val_accuracy: 0.9776 - val_loss: 0.1186
Epoch 18/30
1407/1407            6s 4ms/step -
accuracy: 0.9989 - loss: 0.0036 - val_accuracy: 0.9763 - val_loss: 0.1275
Epoch 19/30
1407/1407            6s 4ms/step -
accuracy: 0.9993 - loss: 0.0022 - val_accuracy: 0.9781 - val_loss: 0.1240
Epoch 20/30
1407/1407            6s 4ms/step -
accuracy: 0.9972 - loss: 0.0078 - val_accuracy: 0.9779 - val_loss: 0.1314
Epoch 21/30
1407/1407            6s 4ms/step -
```

```
accuracy: 0.9986 - loss: 0.0047 - val_accuracy: 0.9765 - val_loss: 0.1494
Epoch 22/30
1407/1407             5s 4ms/step -
accuracy: 0.9984 - loss: 0.0054 - val_accuracy: 0.9781 - val_loss: 0.1489
Epoch 23/30
1407/1407             6s 4ms/step -
accuracy: 0.9987 - loss: 0.0046 - val_accuracy: 0.9765 - val_loss: 0.1573
Epoch 24/30
1407/1407             6s 4ms/step -
accuracy: 0.9992 - loss: 0.0029 - val_accuracy: 0.9791 - val_loss: 0.1417
Epoch 25/30
1407/1407             6s 4ms/step -
accuracy: 0.9985 - loss: 0.0049 - val_accuracy: 0.9769 - val_loss: 0.1589
Epoch 26/30
1407/1407             7s 5ms/step -
accuracy: 0.9983 - loss: 0.0046 - val_accuracy: 0.9790 - val_loss: 0.1442
Epoch 27/30
1407/1407             6s 4ms/step -
accuracy: 0.9988 - loss: 0.0039 - val_accuracy: 0.9769 - val_loss: 0.1664
Epoch 28/30
1407/1407             6s 4ms/step -
accuracy: 0.9993 - loss: 0.0025 - val_accuracy: 0.9769 - val_loss: 0.1673
Epoch 29/30
1407/1407             6s 4ms/step -
accuracy: 0.9982 - loss: 0.0053 - val_accuracy: 0.9797 - val_loss: 0.1527
Epoch 30/30
1407/1407             6s 4ms/step -
accuracy: 0.9986 - loss: 0.0040 - val_accuracy: 0.9763 - val_loss: 0.1849
313/313             0s 1ms/step -
accuracy: 0.9765 - loss: 0.1803

Precisión en los datos de prueba con 30 epochs: 0.9786999821662903
```

Tu respuesta a la pregunta 7.3 aquí: Al aumentar las épocas a 30, la precisión en los datos de prueba no mejora significativamente y la pérdida comienza a aumentar en el conjunto de validación. Este fenómeno se conoce como overfitting o sobreajuste, es decir, el modelo se ajusta demasiado a los datos de entrenamiento, capturando detalles y ruido específicos de esos datos. Como resultado, pierde su capacidad para generalizar bien con datos nuevos, causando un descenso en el rendimiento en el conjunto de validación y prueba.

5 8: Early stop

En el ejercicio anterior, cuando entrenabas con epoch extras, tenías un problema en el que tu pérdida podía cambiar. Puede que te haya llevado un poco de tiempo esperar a que el entrenamiento lo hiciera, y puede que hayas pensado "¿no estaría bien si pudiera parar el entrenamiento cuando alcance un valor deseado?", es decir, una precisión del 85% podría ser suficiente para ti, y si alcanzas eso después de 3 epoch, ¿por qué sentarte a esperar a que termine muchas más épocas? Como cualquier otro programa existen formas de parar la ejecución

A partir del código de ejemplo, hacer una nueva función que tenga en cuenta la perdida (loss) y que pueda parar el código para evitar que ocurra el efeto secundario que vimos en el ejercicio 5.

[ ]:
```
### Ejemplo de código

class myCallback(tf.keras.callbacks.Callback):
      def on_epoch_end(self, epoch, logs={}):
        if(logs.get('accuracy')> 0.85):
              print("\nAlcanzado el 85% de precisión, se cancela el␣
 ↪entrenamiento!!")
              self.model.stop_training = True
```

Pregunta 8.1. ***(0.75 puntos)***: Consulta la documentación de Keras y aprende cómo podemos utilizar Early stop en nuestro modelos.

[49]:
```
model = tf.keras.models.Sequential([
    tf.keras.layers.Dense(512, activation='relu', input_shape=(training_images.
 ↪shape[1],)),
    tf.keras.layers.Dense(10, activation='softmax')
])

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

# Configuración del Early Stopping
early_stop = tf.keras.callbacks.EarlyStopping(
    monitor='val_loss',       # Monitorea la pérdida de validación
    patience=3,               # Número de épocas sin mejora tras el cual se␣
 ↪detiene el entrenamiento
    verbose=1,                # Mostrar mensajes sobre el estado del Early␣
 ↪Stopping
    restore_best_weights=True # Restaurar los pesos con el mejor resultado de␣
 ↪validación
)

model.fit(training_images, training_labels,
          epochs=30,
          batch_size=32,
          validation_split=0.25,
          callbacks=[early_stop])  # Incluye el callback para early stopping

test_loss, test_acc = model.evaluate(test_images, test_labels)
print(f'\nPrecisión en los datos de prueba con 30 epochs: {test_acc}')
```

```
Epoch 1/30
1407/1407             7s 5ms/step -
accuracy: 0.8909 - loss: 0.3742 - val_accuracy: 0.9611 - val_loss: 0.1301
```

```
Epoch 2/30
1407/1407             9s 6ms/step -
accuracy: 0.9711 - loss: 0.0942 - val_accuracy: 0.9710 - val_loss: 0.0944
Epoch 3/30
1407/1407             8s 6ms/step -
accuracy: 0.9819 - loss: 0.0581 - val_accuracy: 0.9735 - val_loss: 0.0887
Epoch 4/30
1407/1407             7s 5ms/step -
accuracy: 0.9879 - loss: 0.0362 - val_accuracy: 0.9693 - val_loss: 0.1036
Epoch 5/30
1407/1407             8s 5ms/step -
accuracy: 0.9915 - loss: 0.0271 - val_accuracy: 0.9745 - val_loss: 0.0924
Epoch 6/30
1407/1407             8s 6ms/step -
accuracy: 0.9939 - loss: 0.0191 - val_accuracy: 0.9736 - val_loss: 0.0979
Epoch 6: early stopping
Restoring model weights from the end of the best epoch: 3.
313/313             1s 2ms/step -
accuracy: 0.9727 - loss: 0.0895

Precisión en los datos de prueba con 30 epochs: 0.9771000146865845
```

5.1 9. Unidades de activación

En este ejercicio, vamos a evaluar la importancia de utilizar las unidades de activación adecuadas. Como hemos visto en clase, funciones de activación como sigmoid han dejado de utilizarse en favor de otras unidades como ReLU.

Pregunta 9.1 ***(0.75 puntos)***: Utilizando la red realizada en el ejercicio 3, escribir un breve análisis comparando la utilización de unidades sigmoid y ReLU (por ejemplo, se pueden comentar aspectos como velocidad de convergencia, métricas obtenidas…). Explicar por qué pueden darse estas diferencias. Opcionalmente, comparar con otras activaciones disponibles en Keras.

Pista: Usando redes más grandes se hace más sencillo apreciar las diferencias. Es mejor utilizar al menos 3 o 4 capas densas.

[56]:
```
# Red con activación Sigmoid
# Modelo con función de activación Sigmoid en capas ocultas
model_sigmoid = tf.keras.models.Sequential([
    tf.keras.layers.Dense(512, activation='sigmoid',␣
 ↪input_shape=(training_images.shape[1],)),
    tf.keras.layers.Dense(256, activation='sigmoid'),
    tf.keras.layers.Dense(128, activation='sigmoid'),
    tf.keras.layers.Dense(10, activation='softmax')
])

model_sigmoid.compile(optimizer='adam',
                      loss='categorical_crossentropy',
                      metrics=['accuracy'])
```

```
model_sigmoid.fit(training_images, training_labels,
                  epochs=10,
                  batch_size=32,
                  validation_split=0.25)

test_loss, test_acc = model_sigmoid.evaluate(test_images, test_labels)
print(f'\nPrecisión en los datos de prueba: {test_acc}')
```

```
Epoch 1/10
1407/1407             11s 7ms/step -
accuracy: 0.7502 - loss: 0.8157 - val_accuracy: 0.9387 - val_loss: 0.2060
Epoch 2/10
1407/1407             9s 7ms/step -
accuracy: 0.9444 - loss: 0.1819 - val_accuracy: 0.9408 - val_loss: 0.1939
Epoch 3/10
1407/1407             10s 7ms/step -
accuracy: 0.9657 - loss: 0.1166 - val_accuracy: 0.9643 - val_loss: 0.1231
Epoch 4/10
1407/1407             9s 7ms/step -
accuracy: 0.9768 - loss: 0.0774 - val_accuracy: 0.9680 - val_loss: 0.1122
Epoch 5/10
1407/1407             9s 7ms/step -
accuracy: 0.9813 - loss: 0.0594 - val_accuracy: 0.9644 - val_loss: 0.1191
Epoch 6/10
1407/1407             10s 7ms/step -
accuracy: 0.9861 - loss: 0.0433 - val_accuracy: 0.9735 - val_loss: 0.0920
Epoch 7/10
1407/1407             9s 7ms/step -
accuracy: 0.9890 - loss: 0.0352 - val_accuracy: 0.9743 - val_loss: 0.0952
Epoch 8/10
1407/1407             10s 7ms/step -
accuracy: 0.9906 - loss: 0.0284 - val_accuracy: 0.9719 - val_loss: 0.1079
Epoch 9/10
1407/1407             11s 8ms/step -
accuracy: 0.9945 - loss: 0.0198 - val_accuracy: 0.9700 - val_loss: 0.1199
Epoch 10/10
1407/1407             10s 7ms/step -
accuracy: 0.9951 - loss: 0.0165 - val_accuracy: 0.9759 - val_loss: 0.1013
313/313              1s 2ms/step -
accuracy: 0.9714 - loss: 0.1093

Precisión en los datos de prueba: 0.9764000177383423
```

[57]:
```
# Red con activación ReLU
# Modelo con función de activación ReLU en capas ocultas
model_relu = tf.keras.models.Sequential([
```

```
    tf.keras.layers.Dense(512, activation='relu', input_shape=(training_images.
 ↪shape[1],)),
    tf.keras.layers.Dense(256, activation='relu'),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(10, activation='softmax')
])

model_relu.compile(optimizer='adam',
                   loss='categorical_crossentropy',
                   metrics=['accuracy'])

model_relu.fit(training_images, training_labels,
               epochs=10,
               batch_size=32,
               validation_split=0.25)

test_loss, test_acc = model_relu.evaluate(test_images, test_labels)
print(f'\nPrecisión en los datos de prueba: {test_acc}')
```

```
Epoch 1/10
1407/1407             9s 5ms/step -
accuracy: 0.8868 - loss: 0.3745 - val_accuracy: 0.9647 - val_loss: 0.1161
Epoch 2/10
1407/1407             10s 7ms/step -
accuracy: 0.9703 - loss: 0.0973 - val_accuracy: 0.9661 - val_loss: 0.1185
Epoch 3/10
1407/1407             10s 7ms/step -
accuracy: 0.9788 - loss: 0.0660 - val_accuracy: 0.9697 - val_loss: 0.1022
Epoch 4/10
1407/1407             10s 7ms/step -
accuracy: 0.9855 - loss: 0.0464 - val_accuracy: 0.9728 - val_loss: 0.0991
Epoch 5/10
1407/1407             11s 8ms/step -
accuracy: 0.9889 - loss: 0.0328 - val_accuracy: 0.9724 - val_loss: 0.1068
Epoch 6/10
1407/1407             10s 7ms/step -
accuracy: 0.9894 - loss: 0.0346 - val_accuracy: 0.9745 - val_loss: 0.1030
Epoch 7/10
1407/1407             8s 5ms/step -
accuracy: 0.9927 - loss: 0.0234 - val_accuracy: 0.9729 - val_loss: 0.1118
Epoch 8/10
1407/1407             8s 6ms/step -
accuracy: 0.9926 - loss: 0.0239 - val_accuracy: 0.9701 - val_loss: 0.1427
Epoch 9/10
1407/1407             8s 6ms/step -
accuracy: 0.9924 - loss: 0.0238 - val_accuracy: 0.9700 - val_loss: 0.1304
Epoch 10/10
1407/1407             7s 5ms/step -
```

```
accuracy: 0.9941 - loss: 0.0185 - val_accuracy: 0.9767 - val_loss: 0.1049
313/313             0s 1ms/step -
accuracy: 0.9751 - loss: 0.1155

Precisión en los datos de prueba: 0.979200005531311
```

Se observa que la red con ReLU converge más rápido y obtiene mejores métricas en general. Esto se debe a que ReLU evita el problema de "vanishing gradients" que suele ocurrir con sigmoid, especialmente en redes profundas. Además, ReLU permite una dinámica de aprendizaje más efectiva, ayudando a la red a alcanzar precisiones más altas en menos tiempo. En cambio, sigmoid es más lenta y tiende a saturarse, lo que limita la eficiencia en la optimización.

5.2 10. Inicialización de parámetros

En este ejercicio, vamos a evaluar la importancia de una correcta inicialización de parámetros en una red neuronal.

Pregunta 10.1 ***(0.75 puntos)***: Partiendo de una red similar a la del ejercicio anterior (usando ya ReLUs), comentar las diferencias que se aprecian en el entrenamiento al utilizar distintas estrategias de inicialización de parámetros. Para ello, inicializar todas las capas con las siguientes estrategias, disponibles en Keras, y analizar sus diferencias:

- Inicialización con ceros.
- Inicialización con una variable aleatoria normal.
- Inicialización con los valores por defecto de Keras para una capa Dense (estrategia *glorot uniform*)

[58]:
```
# Modelo con inicialización de parámetros en cero
model_zeros = tf.keras.models.Sequential([
    tf.keras.layers.Dense(512, activation='relu', kernel_initializer='zeros',␣
 ↪input_shape=(training_images.shape[1],)),
    tf.keras.layers.Dense(256, activation='relu', kernel_initializer='zeros'),
    tf.keras.layers.Dense(128, activation='relu', kernel_initializer='zeros'),
    tf.keras.layers.Dense(10, activation='softmax')
])

model_zeros.compile(optimizer='adam',
                    loss='categorical_crossentropy',
                    metrics=['accuracy'])

model_zeros.fit(training_images, training_labels,
                epochs=10,
                batch_size=32,
                validation_split=0.25)

test_loss, test_acc = model_zeros.evaluate(test_images, test_labels)
print(f'\nPrecisión en los datos de prueba: {test_acc}')
```

```
Epoch 1/10
1407/1407             10s 6ms/step -
```

```
accuracy: 0.1110 - loss: 2.3018 - val_accuracy: 0.1076 - val_loss: 2.3021
Epoch 2/10
1407/1407             9s 6ms/step -
accuracy: 0.1137 - loss: 2.3012 - val_accuracy: 0.1076 - val_loss: 2.3019
Epoch 3/10
1407/1407             9s 6ms/step -
accuracy: 0.1110 - loss: 2.3017 - val_accuracy: 0.1076 - val_loss: 2.3023
Epoch 4/10
1407/1407             9s 6ms/step -
accuracy: 0.1132 - loss: 2.3013 - val_accuracy: 0.1076 - val_loss: 2.3021
Epoch 5/10
1407/1407             11s 8ms/step -
accuracy: 0.1145 - loss: 2.3007 - val_accuracy: 0.1076 - val_loss: 2.3019
Epoch 6/10
1407/1407             17s 6ms/step -
accuracy: 0.1122 - loss: 2.3013 - val_accuracy: 0.1076 - val_loss: 2.3021
Epoch 7/10
1407/1407             8s 6ms/step -
accuracy: 0.1141 - loss: 2.3011 - val_accuracy: 0.1076 - val_loss: 2.3018
Epoch 8/10
1407/1407             8s 6ms/step -
accuracy: 0.1154 - loss: 2.3009 - val_accuracy: 0.1076 - val_loss: 2.3019
Epoch 9/10
1407/1407             9s 6ms/step -
accuracy: 0.1156 - loss: 2.3008 - val_accuracy: 0.1076 - val_loss: 2.3019
Epoch 10/10
1407/1407             8s 6ms/step -
accuracy: 0.1141 - loss: 2.3009 - val_accuracy: 0.1076 - val_loss: 2.3021
313/313             1s 2ms/step -
accuracy: 0.1160 - loss: 2.3009

Precisión en los datos de prueba: 0.11349999904632568
```

[59]:
```
# Modelo con inicialización de parámetros aleatoria normal
model_random_normal = tf.keras.models.Sequential([
    tf.keras.layers.Dense(512, activation='relu',␣
 ↪kernel_initializer='random_normal', input_shape=(training_images.shape[1],)),
    tf.keras.layers.Dense(256, activation='relu',␣
 ↪kernel_initializer='random_normal'),
    tf.keras.layers.Dense(128, activation='relu',␣
 ↪kernel_initializer='random_normal'),
    tf.keras.layers.Dense(10, activation='softmax')
])

model_random_normal.compile(optimizer='adam',
                            loss='categorical_crossentropy',
                            metrics=['accuracy'])
```

```
model_random_normal.fit(training_images, training_labels,
                        epochs=10,
                        batch_size=32,
                        validation_split=0.25)

test_loss, test_acc = model_random_normal.evaluate(test_images, test_labels)
print(f'\nPrecisión en los datos de prueba: {test_acc}')
```

```
Epoch 1/10
1407/1407             12s 7ms/step -
accuracy: 0.8804 - loss: 0.3911 - val_accuracy: 0.9566 - val_loss: 0.1433
Epoch 2/10
1407/1407             10s 7ms/step -
accuracy: 0.9694 - loss: 0.0982 - val_accuracy: 0.9643 - val_loss: 0.1179
Epoch 3/10
1407/1407             12s 8ms/step -
accuracy: 0.9794 - loss: 0.0642 - val_accuracy: 0.9661 - val_loss: 0.1181
Epoch 4/10
1407/1407             18s 7ms/step -
accuracy: 0.9846 - loss: 0.0496 - val_accuracy: 0.9711 - val_loss: 0.1084
Epoch 5/10
1407/1407             10s 7ms/step -
accuracy: 0.9885 - loss: 0.0354 - val_accuracy: 0.9737 - val_loss: 0.1042
Epoch 6/10
1407/1407             13s 10ms/step -
accuracy: 0.9893 - loss: 0.0320 - val_accuracy: 0.9735 - val_loss: 0.1051
Epoch 7/10
1407/1407             11s 8ms/step -
accuracy: 0.9932 - loss: 0.0234 - val_accuracy: 0.9757 - val_loss: 0.1087
Epoch 8/10
1407/1407             7s 5ms/step -
accuracy: 0.9933 - loss: 0.0214 - val_accuracy: 0.9764 - val_loss: 0.1072
Epoch 9/10
1407/1407             8s 5ms/step -
accuracy: 0.9944 - loss: 0.0190 - val_accuracy: 0.9749 - val_loss: 0.1161
Epoch 10/10
1407/1407             8s 5ms/step -
accuracy: 0.9945 - loss: 0.0173 - val_accuracy: 0.9757 - val_loss: 0.1181
313/313             0s 1ms/step -
accuracy: 0.9759 - loss: 0.1113

Precisión en los datos de prueba: 0.9797000288963318
```

[60]:
```
# Modelo con inicialización por defecto glorot_uniform
model_glorot = tf.keras.models.Sequential([
```

```
    tf.keras.layers.Dense(512, activation='relu', input_shape=(training_images.
 ↪shape[1],)),  # glorot_uniform por defecto
    tf.keras.layers.Dense(256, activation='relu'),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(10, activation='softmax')
])

model_glorot.compile(optimizer='adam',
                     loss='categorical_crossentropy',
                     metrics=['accuracy'])

model_glorot.fit(training_images, training_labels,
                 epochs=10,
                 batch_size=32,
                 validation_split=0.25)

test_loss, test_acc = model_glorot.evaluate(test_images, test_labels)
print(f'\nPrecisión en los datos de prueba: {test_acc}')
```

```
Epoch 1/10
1407/1407             9s 5ms/step -
accuracy: 0.8863 - loss: 0.3729 - val_accuracy: 0.9641 - val_loss: 0.1179
Epoch 2/10
1407/1407             7s 5ms/step -
accuracy: 0.9694 - loss: 0.0952 - val_accuracy: 0.9684 - val_loss: 0.1109
Epoch 3/10
1407/1407             6s 4ms/step -
accuracy: 0.9809 - loss: 0.0632 - val_accuracy: 0.9663 - val_loss: 0.1173
Epoch 4/10
1407/1407             7s 5ms/step -
accuracy: 0.9844 - loss: 0.0473 - val_accuracy: 0.9665 - val_loss: 0.1204
Epoch 5/10
1407/1407             6s 4ms/step -
accuracy: 0.9887 - loss: 0.0361 - val_accuracy: 0.9695 - val_loss: 0.1145
Epoch 6/10
1407/1407             7s 5ms/step -
accuracy: 0.9906 - loss: 0.0297 - val_accuracy: 0.9655 - val_loss: 0.1484
Epoch 7/10
1407/1407             6s 4ms/step -
accuracy: 0.9910 - loss: 0.0287 - val_accuracy: 0.9748 - val_loss: 0.1100
Epoch 8/10
1407/1407             7s 5ms/step -
accuracy: 0.9933 - loss: 0.0204 - val_accuracy: 0.9676 - val_loss: 0.1414
Epoch 9/10
1407/1407             10s 7ms/step -
accuracy: 0.9924 - loss: 0.0239 - val_accuracy: 0.9723 - val_loss: 0.1221
Epoch 10/10
1407/1407             9s 6ms/step -
```

```
accuracy: 0.9936 - loss: 0.0191 - val_accuracy: 0.9765 - val_loss: 0.1117
313/313             1s 2ms/step -
accuracy: 0.9758 - loss: 0.1072

Precisión en los datos de prueba: 0.9801999926567078
```

La comparación muestra que la inicialización con ceros es muy ineficiente. La red prácticamente no aprende, ya que los pesos iguales no permiten una representación adecuada. La inicialización aleatoria normal funciona mucho mejor y permite que el modelo alcance una buena precisión, pero puede ser inestable, ya que a veces los pesos iniciales son muy grandes o muy pequeños. Finalmente, la inicialización glorot_uniform, ofrece el mejor balance, permite una convergencia rápida y estable, logrando una precisión alta sin los problemas de saturación.

5.3 11. Optimizadores

Problema 11.1 ***(0.75 puntos)***: Partiendo de una red similar a la del ejercicio anterior (utilizando la mejor estrategia de inicialización observada), comparar y analizar las diferencias que se observan al entrenar con varios de los optimizadores vistos en clase, incluyendo SGD como optimizador básico (se puede explorar el espacio de hiperparámetros de cada optimizador, aunque para optimizadores más avanzados del estilo de RMSprop es buena idea dejar los valores por defecto provistos por Keras).

[61]:
```
# Modelo con optimizador SGD
model_sgd = tf.keras.models.Sequential([
    tf.keras.layers.Dense(512, activation='relu',␣
 ↪kernel_initializer='glorot_uniform', input_shape=(784,)),
    tf.keras.layers.Dense(256, activation='relu',␣
 ↪kernel_initializer='glorot_uniform'),
    tf.keras.layers.Dense(128, activation='relu',␣
 ↪kernel_initializer='glorot_uniform'),
    tf.keras.layers.Dense(10, activation='softmax')
])

model_sgd.compile(optimizer=tf.keras.optimizers.SGD(learning_rate=0.01),
                  loss='categorical_crossentropy',
                  metrics=['accuracy'])

model_sgd.fit(training_images, training_labels,
              epochs=10,
              batch_size=32,
              validation_split=0.25)

test_loss, test_acc = model_sgd.evaluate(test_images, test_labels)
print(f'\nPrecisión en los datos de prueba: {test_acc}')
```

```
Epoch 1/10
1407/1407             5s 3ms/step -
accuracy: 0.7027 - loss: 1.1141 - val_accuracy: 0.9147 - val_loss: 0.2995
Epoch 2/10
```

```
1407/1407             5s 4ms/step -
accuracy: 0.9179 - loss: 0.2887 - val_accuracy: 0.9308 - val_loss: 0.2418
Epoch 3/10
1407/1407             5s 3ms/step -
accuracy: 0.9367 - loss: 0.2195 - val_accuracy: 0.9425 - val_loss: 0.1999
Epoch 4/10
1407/1407             5s 4ms/step -
accuracy: 0.9475 - loss: 0.1857 - val_accuracy: 0.9464 - val_loss: 0.1902
Epoch 5/10
1407/1407             5s 3ms/step -
accuracy: 0.9541 - loss: 0.1557 - val_accuracy: 0.9528 - val_loss: 0.1614
Epoch 6/10
1407/1407             5s 4ms/step -
accuracy: 0.9612 - loss: 0.1333 - val_accuracy: 0.9499 - val_loss: 0.1688
Epoch 7/10
1407/1407             5s 4ms/step -
accuracy: 0.9660 - loss: 0.1180 - val_accuracy: 0.9605 - val_loss: 0.1386
Epoch 8/10
1407/1407             5s 4ms/step -
accuracy: 0.9705 - loss: 0.1032 - val_accuracy: 0.9611 - val_loss: 0.1306
Epoch 9/10
1407/1407             5s 3ms/step -
accuracy: 0.9748 - loss: 0.0896 - val_accuracy: 0.9625 - val_loss: 0.1261
Epoch 10/10
1407/1407             5s 3ms/step -
accuracy: 0.9763 - loss: 0.0789 - val_accuracy: 0.9638 - val_loss: 0.1247
313/313             1s 2ms/step -
accuracy: 0.9626 - loss: 0.1314

Precisión en los datos de prueba: 0.9674000144004822
```

[62]:
```python
# Definir el modelo con inicialización glorot_uniform
model_adam = tf.keras.models.Sequential([
    tf.keras.layers.Dense(512, activation='relu',␣
 ↪kernel_initializer='glorot_uniform', input_shape=(784,)),
    tf.keras.layers.Dense(256, activation='relu',␣
 ↪kernel_initializer='glorot_uniform'),
    tf.keras.layers.Dense(128, activation='relu',␣
 ↪kernel_initializer='glorot_uniform'),
    tf.keras.layers.Dense(10, activation='softmax')
])

model_adam.compile(optimizer='adam',
                   loss='categorical_crossentropy',
                   metrics=['accuracy'])

model_adam.fit(training_images, training_labels,
```

```
                epochs=10,
                batch_size=32,
                validation_split=0.25)

test_loss, test_acc = model_adam.evaluate(test_images, test_labels)
print(f'\nPrecisión en los datos de prueba: {test_acc}')
```

```
Epoch 1/10
1407/1407             10s 6ms/step -
accuracy: 0.8897 - loss: 0.3689 - val_accuracy: 0.9621 - val_loss: 0.1278
Epoch 2/10
1407/1407             9s 6ms/step -
accuracy: 0.9714 - loss: 0.0920 - val_accuracy: 0.9675 - val_loss: 0.1087
Epoch 3/10
1407/1407             9s 6ms/step -
accuracy: 0.9799 - loss: 0.0641 - val_accuracy: 0.9718 - val_loss: 0.1042
Epoch 4/10
1407/1407             9s 6ms/step -
accuracy: 0.9857 - loss: 0.0454 - val_accuracy: 0.9717 - val_loss: 0.1071
Epoch 5/10
1407/1407             8s 6ms/step -
accuracy: 0.9880 - loss: 0.0378 - val_accuracy: 0.9720 - val_loss: 0.1086
Epoch 6/10
1407/1407             9s 6ms/step -
accuracy: 0.9906 - loss: 0.0275 - val_accuracy: 0.9715 - val_loss: 0.1199
Epoch 7/10
1407/1407             8s 6ms/step -
accuracy: 0.9912 - loss: 0.0283 - val_accuracy: 0.9749 - val_loss: 0.1047
Epoch 8/10
1407/1407             8s 6ms/step -
accuracy: 0.9921 - loss: 0.0243 - val_accuracy: 0.9685 - val_loss: 0.1482
Epoch 9/10
1407/1407             9s 6ms/step -
accuracy: 0.9932 - loss: 0.0227 - val_accuracy: 0.9744 - val_loss: 0.1274
Epoch 10/10
1407/1407             8s 6ms/step -
accuracy: 0.9953 - loss: 0.0143 - val_accuracy: 0.9727 - val_loss: 0.1429
313/313             1s 2ms/step -
accuracy: 0.9729 - loss: 0.1328

Precisión en los datos de prueba: 0.9761000275611877
```

[63]:
```
# Modelo con optimizador RMSprop
model_rmsprop = tf.keras.models.Sequential([
    tf.keras.layers.Dense(512, activation='relu',␣
 ↪kernel_initializer='glorot_uniform', input_shape=(784,)),
```

```
    tf.keras.layers.Dense(256, activation='relu',␣
  ↪kernel_initializer='glorot_uniform'),
    tf.keras.layers.Dense(128, activation='relu',␣
  ↪kernel_initializer='glorot_uniform'),
    tf.keras.layers.Dense(10, activation='softmax')
])

model_rmsprop.compile(optimizer='rmsprop',
                      loss='categorical_crossentropy',
                      metrics=['accuracy'])

model_rmsprop.fit(training_images, training_labels,
                  epochs=10,
                  batch_size=32,
                  validation_split=0.25)

test_loss, test_acc = model_rmsprop.evaluate(test_images, test_labels)
print(f'\nPrecisión en los datos de prueba: {test_acc}')
```

```
Epoch 1/10
1407/1407             9s 6ms/step -
accuracy: 0.8820 - loss: 0.3748 - val_accuracy: 0.9545 - val_loss: 0.1733
Epoch 2/10
1407/1407             8s 6ms/step -
accuracy: 0.9718 - loss: 0.0997 - val_accuracy: 0.9694 - val_loss: 0.1151
Epoch 3/10
1407/1407             7s 5ms/step -
accuracy: 0.9815 - loss: 0.0656 - val_accuracy: 0.9694 - val_loss: 0.1244
Epoch 4/10
1407/1407             7s 5ms/step -
accuracy: 0.9865 - loss: 0.0487 - val_accuracy: 0.9730 - val_loss: 0.1250
Epoch 5/10
1407/1407             8s 5ms/step -
accuracy: 0.9889 - loss: 0.0413 - val_accuracy: 0.9743 - val_loss: 0.1366
Epoch 6/10
1407/1407             7s 5ms/step -
accuracy: 0.9904 - loss: 0.0354 - val_accuracy: 0.9763 - val_loss: 0.1421
Epoch 7/10
1407/1407             7s 5ms/step -
accuracy: 0.9915 - loss: 0.0315 - val_accuracy: 0.9742 - val_loss: 0.1582
Epoch 8/10
1407/1407             9s 6ms/step -
accuracy: 0.9930 - loss: 0.0260 - val_accuracy: 0.9773 - val_loss: 0.1692
Epoch 9/10
1407/1407             7s 5ms/step -
accuracy: 0.9939 - loss: 0.0234 - val_accuracy: 0.9748 - val_loss: 0.1997
Epoch 10/10
```

```
1407/1407              7s 5ms/step -
accuracy: 0.9947 - loss: 0.0204 - val_accuracy: 0.9741 - val_loss: 0.2291
313/313              1s 2ms/step -
accuracy: 0.9731 - loss: 0.2244

Precisión en los datos de prueba: 0.9779999852180481
```

Los resultados muestran diferencias claras entre los optimizadores. SGD tiene una convergencia más lenta y menor precisión en comparación con Adam y RMSprop. Aunque SGD mejora gradualmente, toma más épocas para alcanzar una buena precisión, lo que indica que no es tan eficiente para redes más profundas. Adam destaca al lograr una alta precisión rápidamente y mantener la estabilidad de la pérdida durante el entrenamiento, lo que demuestra su capacidad para adaptarse mejor a redes complejas. RMSprop también ofrece un buen rendimiento, con alta precisión y estabilidad, aunque muestra un poco más de variabilidad en la pérdida de validación en comparación con Adam. En conjunto, Adam parece ser el mejor optimizador para este caso, equilibrando velocidad de convergencia y precisión sin desventajas importantes.

5.4 12. Regularización y red final *(1.25 puntos)*

Problema 12.1 ***(2 puntos)***: Entrenar una red final que sea capaz de obtener una accuracy en el validation superior al 95%. Para ello, combinar todo lo aprendido anteriormente y utilizar técnicas de regularización para evitar overfitting. Algunos de los elementos que pueden tenerse en cuenta son los siguientes.

- Número de capas y neuronas por capa
- Optimizadores y sus parámetros
- Batch size
- Unidades de activación
- Uso de capas dropout, regularización L2, regularización L1…
- Early stopping (se puede aplicar como un callback de Keras, o se puede ver un poco "a ojo" cuándo el modelo empieza a caer en overfitting y seleccionar el número de epochs necesarias)
- Batch normalization

Si los modelos entrenados anteriormente ya se acercaban al valor requerido de accuracy, probar distintas estrategias igualmente y comentar los resultados.

Explicar brevemente la estrategia seguida y los modelos probados para obtener el modelo final, que debe verse entrenado en este Notebook. No es necesario guardar el entrenamiento de todos los modelos que se han probado, es suficiente con explicar cómo se ha llegado al modelo final.

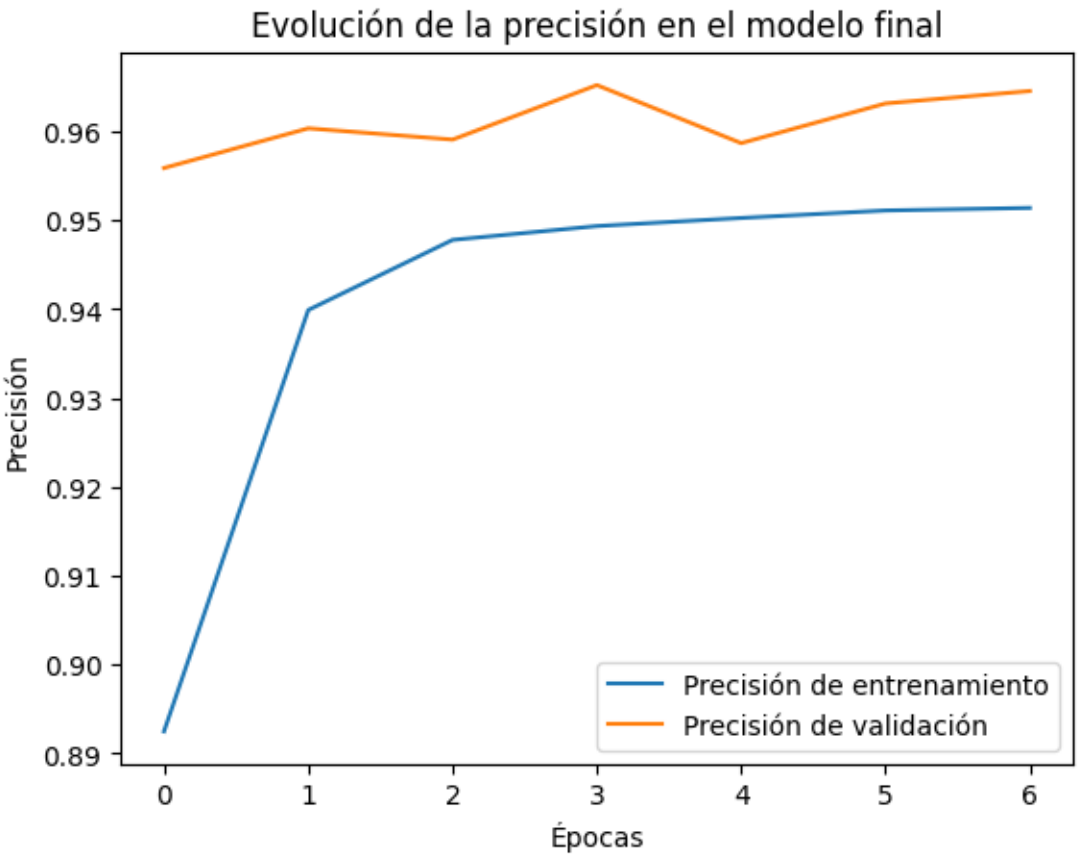
[23]:
```
from tensorflow.keras.layers import Dense, Dropout, BatchNormalization
from tensorflow.keras.regularizers import l2


model_final = tf.keras.models.Sequential([
    Dense(512, activation='relu', kernel_initializer='glorot_uniform',␣
 ↪kernel_regularizer=l2(0.001), input_shape=(784,)),
    BatchNormalization(),
    Dropout(0.3),
```

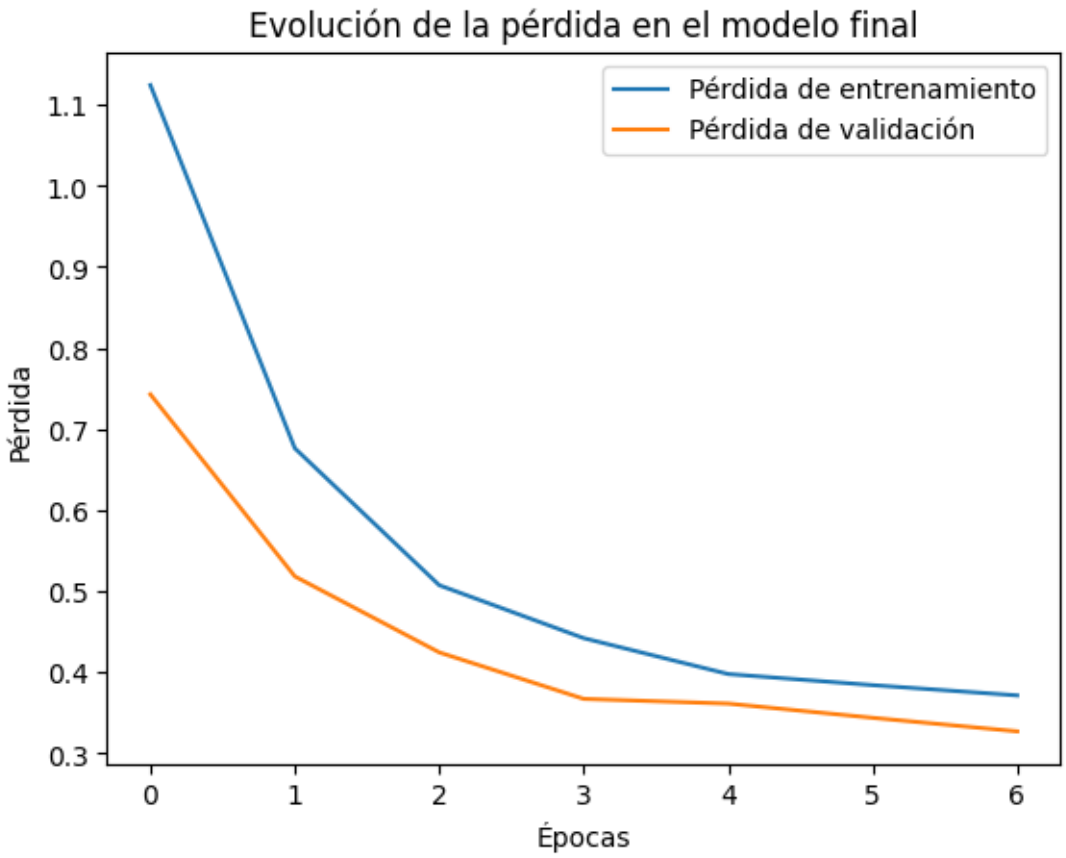
```
    Dense(256, activation='relu', kernel_initializer='glorot_uniform',␣
 ↪kernel_regularizer=l2(0.001)),
    BatchNormalization(),
    Dropout(0.3),

    Dense(128, activation='relu', kernel_initializer='glorot_uniform',␣
 ↪kernel_regularizer=l2(0.001)),
    BatchNormalization(),
    Dropout(0.3),

    Dense(10, activation='softmax')
])

# Compilar el modelo con Adam y early stopping
model_final.compile(optimizer='adam',
                    loss='categorical_crossentropy',
                    metrics=['accuracy'])

# Configuración de Early Stopping
early_stop = tf.keras.callbacks.EarlyStopping(monitor='val_accuracy',␣
 ↪patience=3, restore_best_weights=True, verbose=1)

# Entrenar el modelo
history_final = model_final.fit(training_images, training_labels,
                                epochs=50,             # Hasta 50 épocas para␣
 ↪darle espacio de mejora
                                batch_size=64,         # Batch size más grande␣
 ↪para optimizar tiempos
                                validation_split=0.25, # Usamos 25% de los␣
 ↪datos para validación
                                callbacks=[early_stop])

test_loss, test_accuracy = model_final.evaluate(test_images, test_labels)
print(f'Precisión en los datos de prueba: {test_accuracy}')
```

```
Epoch 1/50
704/704             10s 10ms/step -
accuracy: 0.8209 - loss: 1.4608 - val_accuracy: 0.9559 - val_loss: 0.7429
Epoch 2/50
704/704             5s 8ms/step -
accuracy: 0.9382 - loss: 0.7351 - val_accuracy: 0.9603 - val_loss: 0.5183
Epoch 3/50
704/704             5s 7ms/step -
accuracy: 0.9478 - loss: 0.5350 - val_accuracy: 0.9591 - val_loss: 0.4245
Epoch 4/50
704/704             5s 8ms/step -
accuracy: 0.9501 - loss: 0.4520 - val_accuracy: 0.9652 - val_loss: 0.3672
```

```
Epoch 5/50
704/704             6s 9ms/step -
accuracy: 0.9501 - loss: 0.4093 - val_accuracy: 0.9587 - val_loss: 0.3614
Epoch 6/50
704/704             6s 9ms/step -
accuracy: 0.9526 - loss: 0.3815 - val_accuracy: 0.9631 - val_loss: 0.3439
Epoch 7/50
704/704             7s 9ms/step -
accuracy: 0.9517 - loss: 0.3714 - val_accuracy: 0.9645 - val_loss: 0.3272
Epoch 7: early stopping
Restoring model weights from the end of the best epoch: 4.
313/313             1s 2ms/step -
accuracy: 0.9584 - loss: 0.3843
Precisión en los datos de prueba: 0.9653000235557556
```

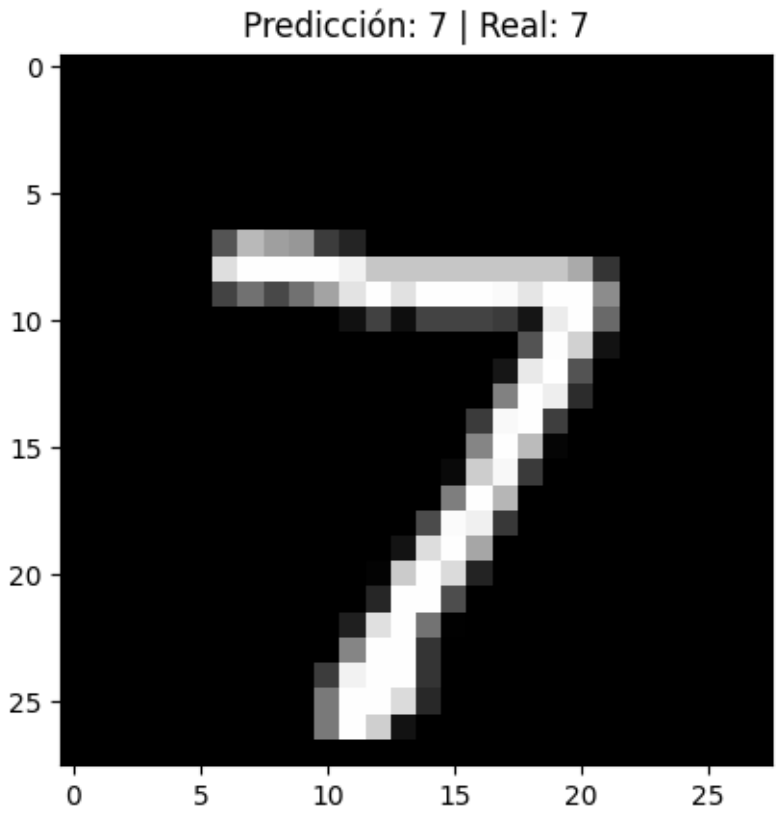
[24]:
```python
import matplotlib.pyplot as plt

# Gráfico la precisión
plt.plot(history_final.history['accuracy'], label='Precisión de entrenamiento')
plt.plot(history_final.history['val_accuracy'], label='Precisión de validación')
plt.xlabel('Épocas')
plt.ylabel('Precisión')
plt.legend()
plt.title('Evolución de la precisión en el modelo final')
plt.show()

# Gráfico la pérdida
plt.plot(history_final.history['loss'], label='Pérdida de entrenamiento')
plt.plot(history_final.history['val_loss'], label='Pérdida de validación')
plt.xlabel('Épocas')
plt.ylabel('Pérdida')
plt.legend()
plt.title('Evolución de la pérdida en el modelo final')
plt.show()
```

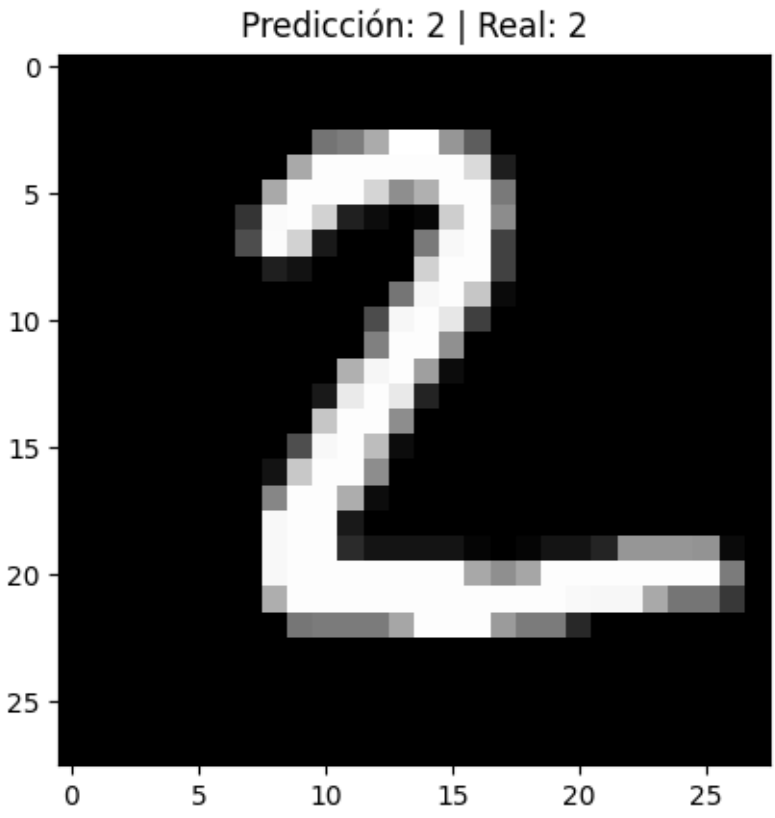

Evolución de la precisión en el modelo final
Precisión
Épocas
0.96
0.95
0.94
0.93
0.92
0.91
0.90
0.89
0
1
2
3
4
5
6
Precisión de entrenamiento
Precisión de validación

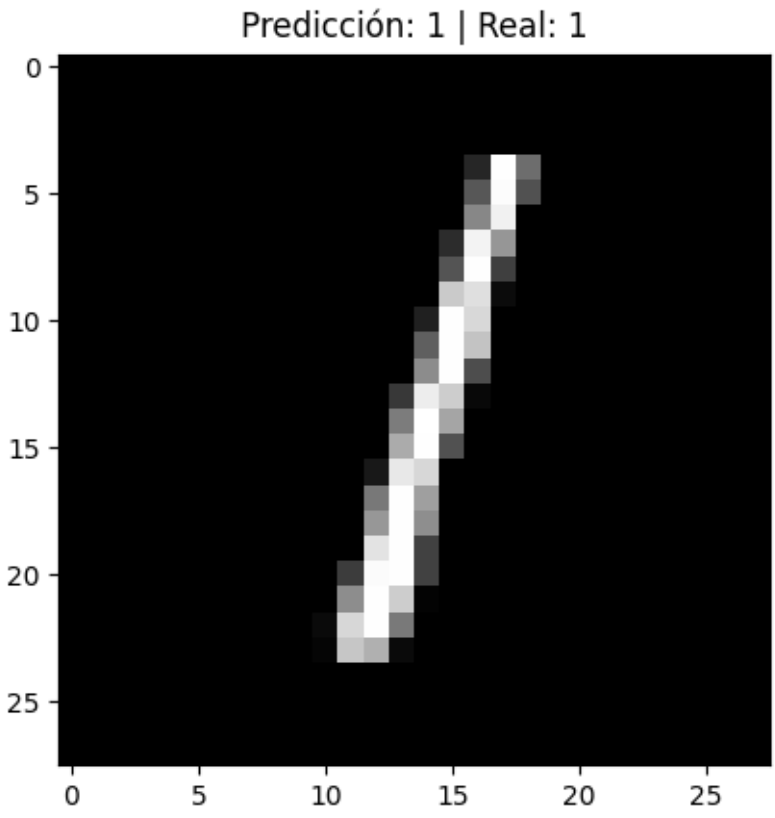


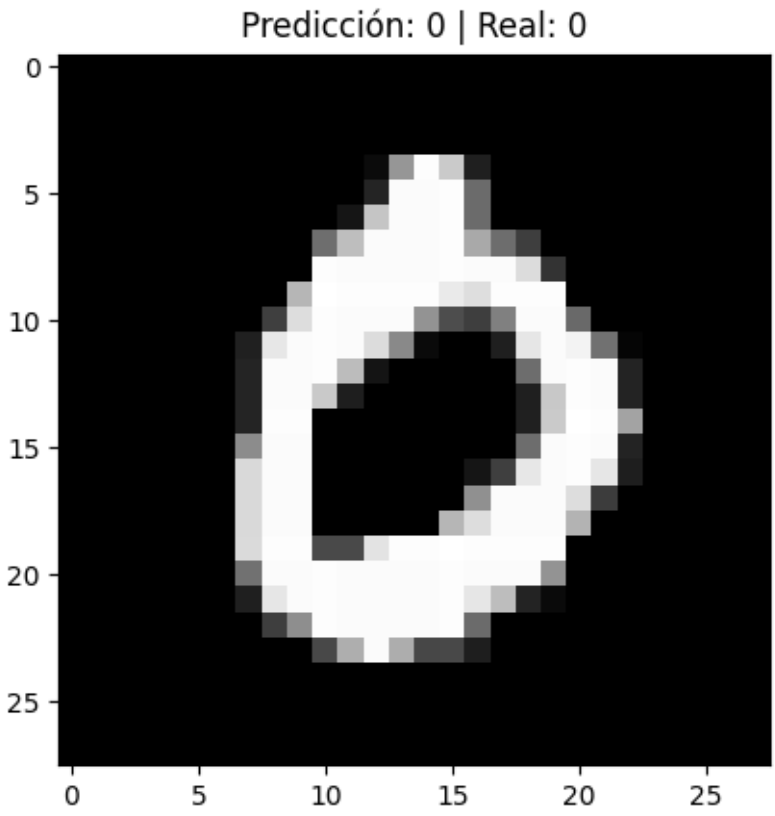
[26]:
```python
# Realizar predicciones en las primeras 10 imágenes del conjunto de prueba
predictions = model_final.predict(test_images[:10])

predicted_labels = np.argmax(predictions, axis=1)
true_labels = np.argmax(test_labels[:10], axis=1)

print(f'Predicciones: {predicted_labels}')
print(f'Etiquetas reales: {true_labels}')

for i in range(10):
    plt.imshow(test_images[i].reshape(28, 28), cmap='gray')
    plt.title(f'Predicción: {predicted_labels[i]} | Real: {true_labels[i]}')
    plt.show()
```

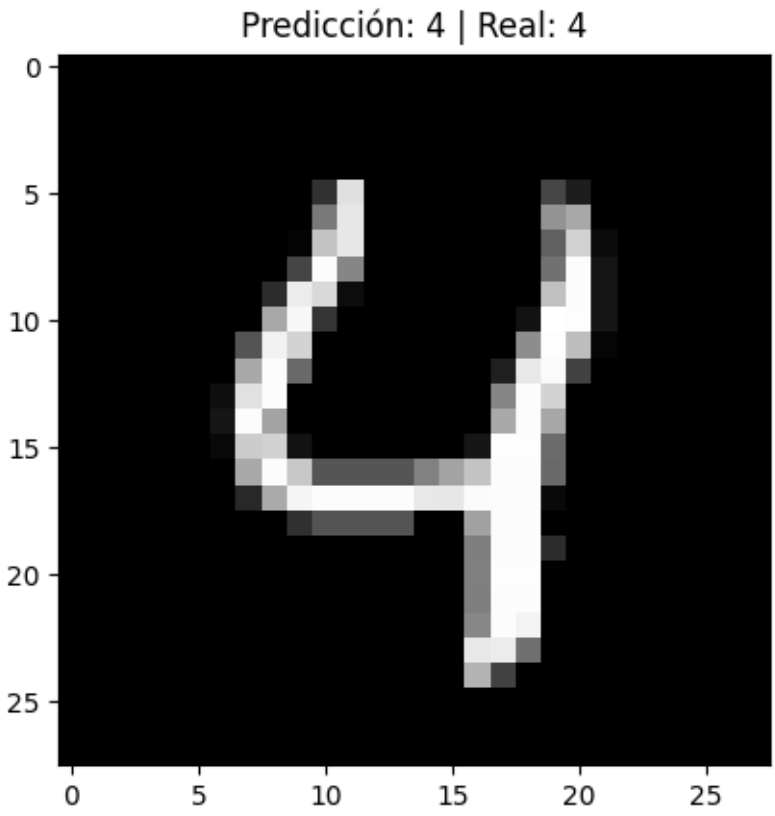
```
1/1             0s 21ms/step
Predicciones: [7 2 1 0 4 1 4 9 6 9]
Etiquetas reales: [7 2 1 0 4 1 4 9 5 9]
```

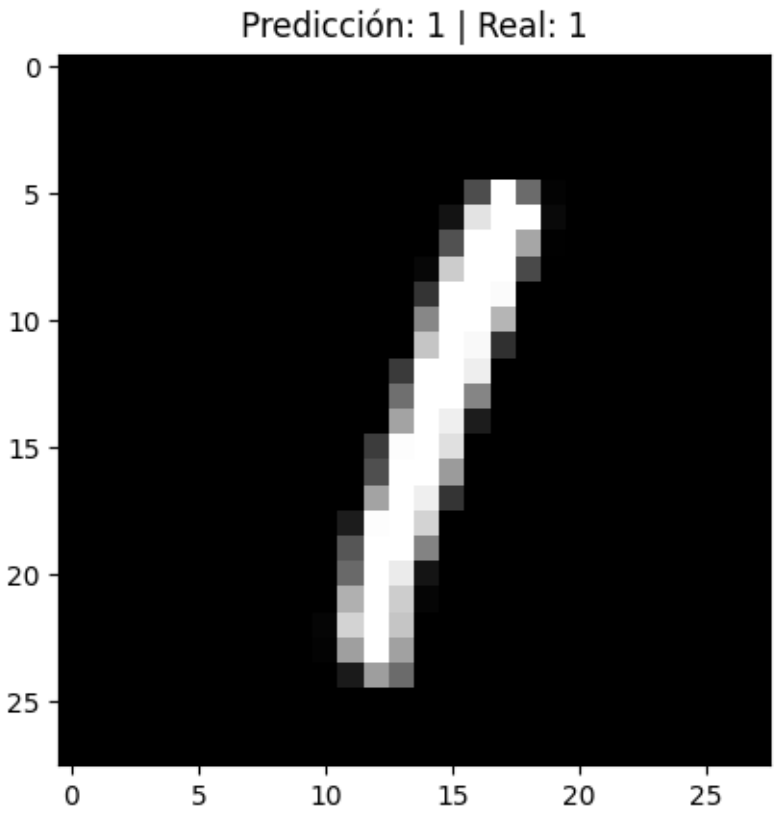

Predicción: 7 | Real: 7
0
5
10
15
20
25
0
5
10
15
20
25

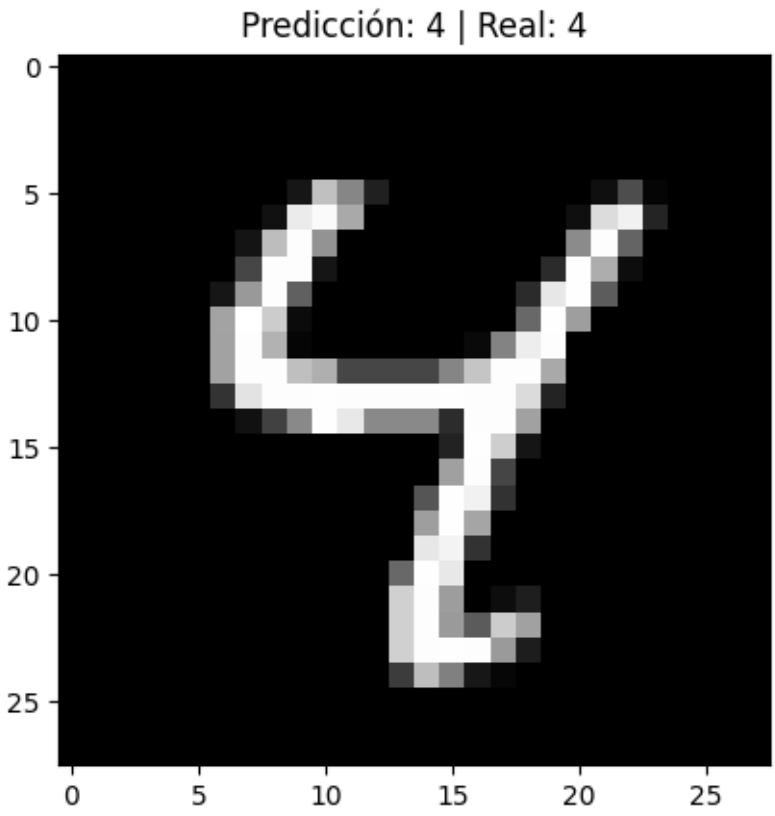

Predicción: 2 | Real: 2
0
5
10
15
20
25
0
5
10
15
20
25

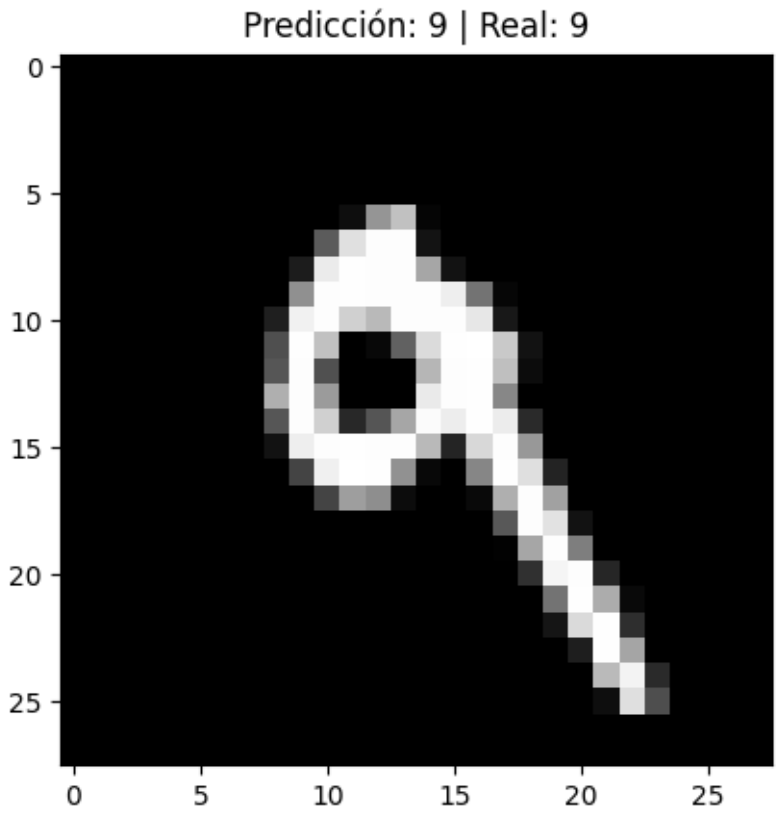

Predicción: 1 | Real: 1
0
5
10
15
20
25
0
5
10
15
20
25

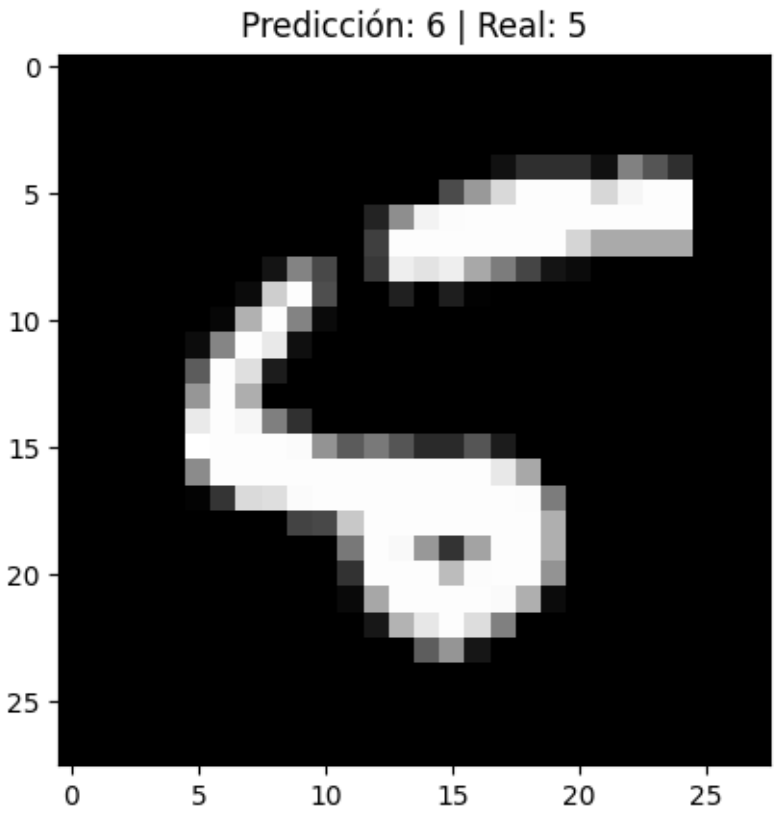

Predicción: 0 | Real: 0
0
5
10
15
20
25
0
5
10
15
20
25

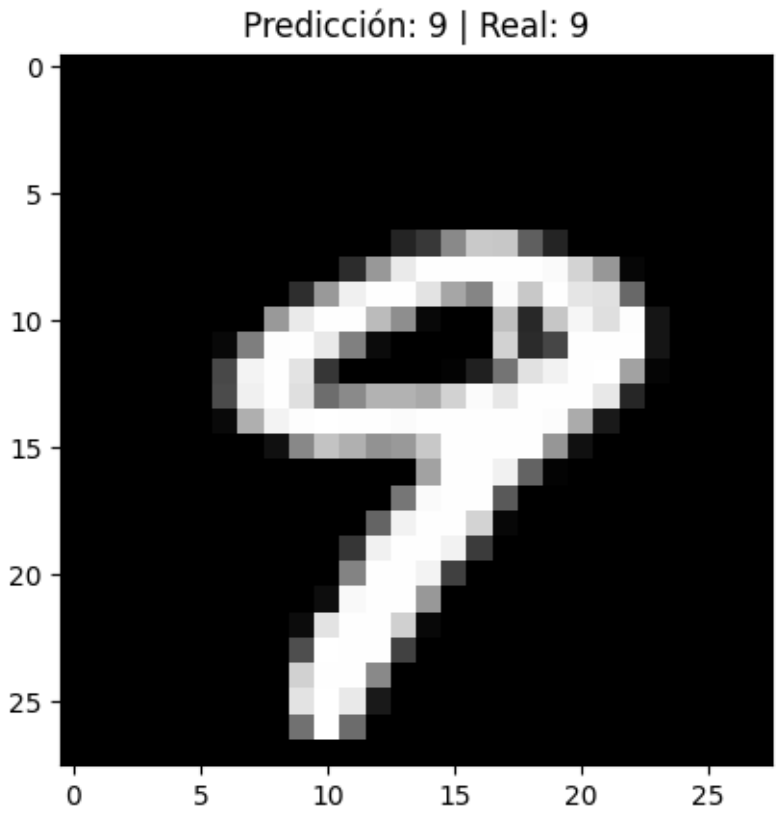

Predicción: 4 | Real: 4
0
5
10
15
20
25
0
5
10
15
20
25


Predicción: 1 | Real: 1
0
5
10
15
20
25
0
5
10
15
20
25


Predicción: 4 | Real: 4
0
5
10
15
20
25
0
5
10
15
20
25


Predicción: 9 | Real: 9
0
5
10
15
20
25
0
5
10
15
20
25


Predicción: 6 | Real: 5
0
5
10
15
20
25
0
5
10
15
20
25



El modelo final logró una precisión del 96.53% en los datos de prueba, gracias a una combinación efectiva de técnicas. Se usaron capas densas con ReLU y la inicialización glorot_uniform para mejorar la estabilidad. Para evitar el sobreajuste, aplicamos dropout del 30% y regularización L2 en las capas, lo que mantuvo al modelo generalizando bien. Además, batch normalization ayudó a estabilizar el entrenamiento y acelerar la convergencia.

El uso del optimizador Adam fue clave, ya que permitió una rápida mejora en las primeras épocas, y el early stopping detuvo el entrenamiento en la época 10, cuando el rendimiento dejó de mejorar, evitando entrenar de más. En conjunto, este enfoque permitió obtener un modelo robusto, con una excelente capacidad de generalización y una alta precisión, superando el 95% en validación sin sobreentrenar.

Además, al probar el modelo con datos de prueba, se observó que las predicciones fueron bastante precisas en la mayoría de los casos. Por ejemplo, para las primeras 10 imágenes, las predicciones fueron: [7, 2, 1, 0, 4, 1, 4, 9, 6, 9], mientras que las etiquetas reales eran: [7, 2, 1, 0, 4, 1, 4, 9, 5, 9]. Solo hubo un pequeño error en la predicción del dígito 6 en lugar del 5, lo que refuerza que el modelo tiene un excelente rendimiento.