Deep Learning con Python y Keras

Parte 5. Redes Neuronales Convolucionales

2. Reconocimiento de dígitos

Manuel Castillo Cara

Índice

---

0. Contexto

En este proyecto, descubrirá cómo desarrollar un modelo de Deep Learning en la tarea de reconocimiento de dígitos manuscritos del MNIST. Después de completar esta clase sabrá:

- Cómo cargar MNIST y desarrollar un modelo de red neuronal.
- Cómo implementar y evaluar una CNN de línea base para MNIST.
- Cómo implementar un modelo de Deep Learning avanzado para MNIST.

```
import tensorflow as tf
# Eliminar warning
tf.compat.v1.logging.set_verbosity(tf.compat.v1.logging.ERROR)
```

---

1. MNIST dataset

MNIST toma imágenes de dígitos de una variedad de documentos escaneados, normalizados en tamaño y centrados.

Cada imagen es está dada en blanco y negro con 28×28 píxeles (784 píxeles en total). Se usan 60,000 imágenes para entrenar un modelo y 10,000 imágenes para validarlo.

Es una tarea de reconocimiento de dígitos. Como tal, hay 10 dígitos (0 a 9) o 10 clases para predecir.

En la página web de Rodrigo Benenson hay una lista de los resultados más avanzados y enlaces a los artículos relevantes sobre el MNIST y otros conjuntos de datos.

Más información sobre el dataset MNIST

Información de los resultados sobre MNIST de Rodrigo Benenson

---

2. Cargar MNIST

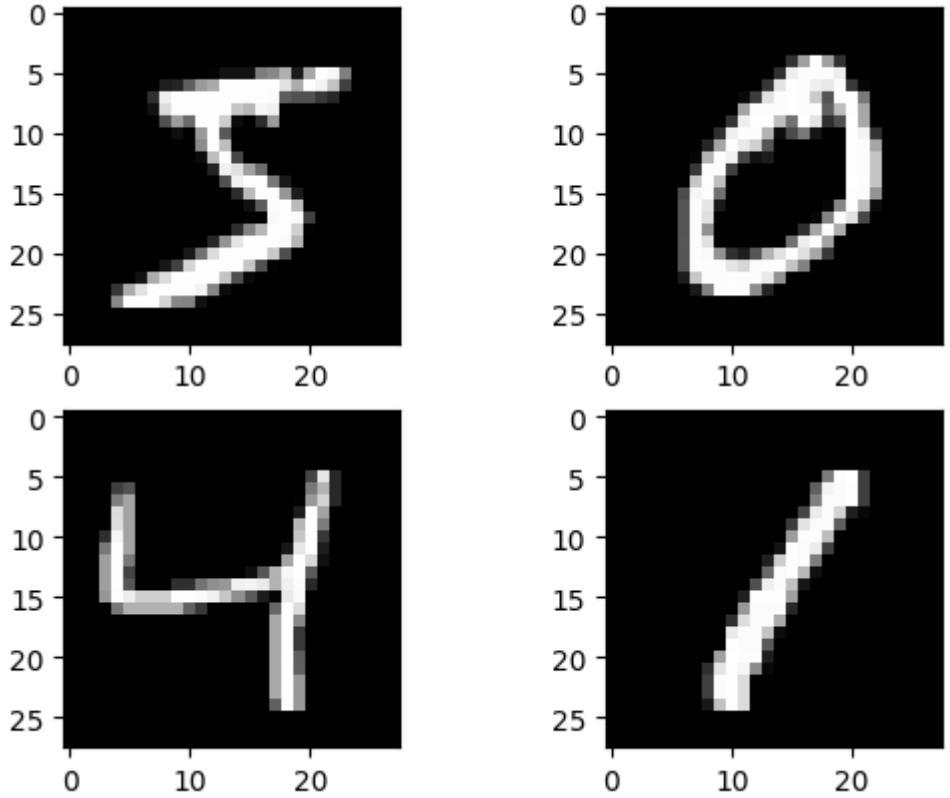
El conjunto de datos se descarga automáticamente la primera vez que se llama a esta función y se almacena en su directorio de inicio en `~/.keras/datasets/mnist.pkl.gz` como un archivo de 15 megabytes.

Primero escribiremos un pequeño script para descargar y visualizar las primeras 4 imágenes mediante la función `mnist.load data()`.

```
# Plot ad hoc mnist instances
from keras.datasets import mnist
import matplotlib.pyplot as plt

# load (downloaded if needed) the MNIST dataset
(X_train, y_train), (X_test, y_test) = mnist.load_data()

# plot 4 images as gray scale
plt.subplot(221)
plt.imshow(X_train[0], cmap=plt.get_cmap('gray'))
plt.subplot(222)
plt.imshow(X_train[1], cmap=plt.get_cmap('gray'))
plt.subplot(223)
plt.imshow(X_train[2], cmap=plt.get_cmap('gray'))
```

```
plt.subplot(224)
plt.imshow(X_train[3], cmap=plt.get_cmap('gray'))

# show the plot
plt.show()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-da
11490434/11490434 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
```



---

3. MLP de línea base

Vamos a usar un MLP clásico como base para la comparación con modelos de redes neuronales convolucionales.

Importamos las clases, funciones y el dataset MNIST.

```
# Baseline MLP for MNIST dataset
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.datasets import mnist
# load data
(X_train, y_train), (X_test, y_test) = mnist.load_data()
```

Para un MLP clásico debemos reducir las imágenes a un vector de píxeles. En este caso, las imágenes de tamaño 28×28 serán vectores de entrada de 784 píxeles.

Realizamos esta transformación meidante la función `reshape()`.

Los valores de los píxeles son números enteros, por lo que los convertimos a punto flotante para poder normalizarlos.

```
# flatten 28*28 images to a 784 vector for each image
num_pixeles = X_train.shape[1] * X_train.shape[2]

X_train = X_train.reshape((X_train.shape[0], num_pixeles)).astype('float32')
X_test = X_test.reshape((X_test.shape[0], num_pixeles)).astype('float32')


X_train.shape[0]
```

```
60000
```

Los valores de los píxeles están en una escala de grises entre 0 y 255. Podemos normalizar los valores de los píxeles en el rango 0 y 1 dividiendo cada valor por el máximo valor, i.e., 255.

```
# normalize inputs from 0-255 to 0-1
X_train = X_train / 255
X_test = X_test / 255
```

Finalmente, la variable de salida es un número entero de 0 a 9. Por tanto, usaremos One-Hot Encoding para transformar el vector de enteros de clase en una matriz binaria.

Usaremos para ello la función de Keras `np_utils.to_categorical()`.

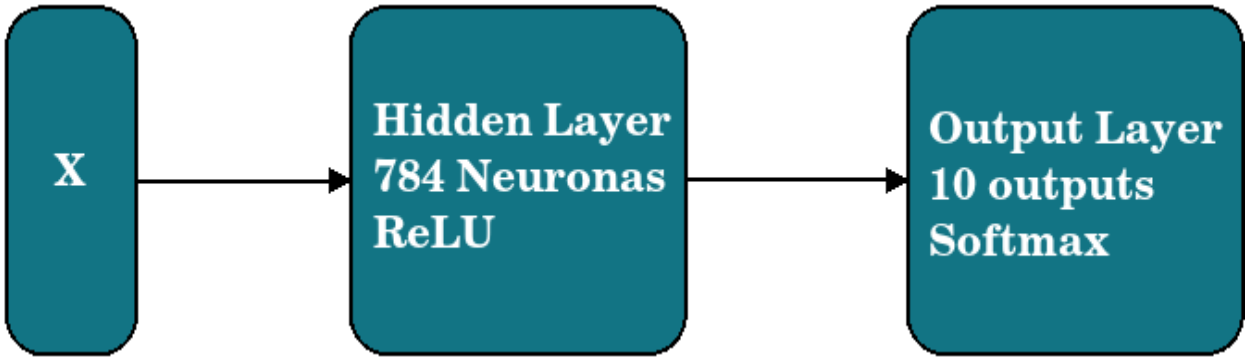
```
from tensorflow.keras.utils import to_categorical

# One hot encode outputs
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)

num_classes = y_test.shape[1]


y_test.shape[1]
```

```
10
```

Double-click (or enter) to edit

Vamos a definir nuestro modelo:

1. El número de entradas será el tamaño máximo de pixeles (784)
2. Tendrá una capa oculta con el mismo número de neuronas que entradas (784).
3. Se utiliza una función de activación ReLU en la capa oculta.
4. Se utiliza una función de activación Softmax en la capa de salida.
5. La función de pérdida será `categorical_crossentropy`.
6. Utilizaremos ADAM para aprender los pesos.




```
# define baseline model
def baseline_model():
    # create model
    model = Sequential()
    model.add(Dense(num_pixeles, input_dim=num_pixeles, activation='relu'))
    model.add(Dense(num_classes, activation='softmax'))
    # Compile model
    model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['

    return model
```

Entrenamos y evaluamos el modelo.

1. El modelo se ajusta a más de 10 épocas con actualizaciones cada 200 imágenes.
2. Los datos de test se utilizan como conjunto de datos de validación.
3. Se utiliza un valor `verbose` de 2.
4. Evaluamos en test e imprimimos las métricas.

```
# build the model
model = baseline_model()

# Fit the model
# verbose=0 will show you nothing (silent)
# verbose=1 will show you an animated progress bar like this:
# verbose=2 will just mention the number of epoch like this: Epoch 1/10
model.fit(X_train, y_train, validation_data=(X_test, y_test), epochs=10, batch_

# Final evaluation of the model
scores = model.evaluate(X_test, y_test, verbose=0)
print("Error del modelo de la linea base: %.2f%%" % (100-scores[1]*100))
```

```
print( Error del modelo de la linea base: %.2f%%  % (100-scores[1] 100))
```

```
Epoch 1/10
300/300 - 7s - 25ms/step - accuracy: 0.9216 - loss: 0.2781 - val_accuracy:
Epoch 2/10
300/300 - 5s - 17ms/step - accuracy: 0.9678 - loss: 0.1105 - val_accuracy:
Epoch 3/10
300/300 - 7s - 22ms/step - accuracy: 0.9794 - loss: 0.0703 - val_accuracy:
Epoch 4/10
300/300 - 5s - 17ms/step - accuracy: 0.9856 - loss: 0.0500 - val_accuracy:
Epoch 5/10
300/300 - 6s - 20ms/step - accuracy: 0.9901 - loss: 0.0358 - val_accuracy:
Epoch 6/10
300/300 - 10s - 32ms/step - accuracy: 0.9932 - loss: 0.0261 - val_accuracy:
Epoch 7/10
300/300 - 6s - 21ms/step - accuracy: 0.9956 - loss: 0.0188 - val_accuracy:
Epoch 8/10
300/300 - 5s - 18ms/step - accuracy: 0.9966 - loss: 0.0146 - val_accuracy:
Epoch 9/10
300/300 - 6s - 21ms/step - accuracy: 0.9975 - loss: 0.0110 - val_accuracy:
Epoch 10/10
300/300 - 9s - 31ms/step - accuracy: 0.9984 - loss: 0.0084 - val_accuracy:
Error del modelo de la linea base: 1.85%
```

---

4. CNN para MNIST

Ahora que hemos visto cómo cargar el conjunto de datos MNIST y entrenar un modelo simple de perceptrón multicapa en él, es hora de desarrollar una red neuronal convolucional más sofisticada o un modelo CNN.

Crearemos una CNN simple para MNIST que demuestra cómo utilizar todos los aspectos de una implementación de CNN moderna, incluidas las capas convolucionales, las capas de agrupación y las capas de dropout.

El primer paso es importar las clases y funciones necesarias.

```
# Simple CNN for the MNIST Dataset ( Cambie a la version mas reciente con tenso
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, Flatten, Conv2D, MaxPooling
from tensorflow.keras.utils import to_categorical
```

En Keras, las capas utilizadas para convoluciones bidimensionales esperan valores de píxeles con las dimensiones `[muestras]-[ancho]-[alto]-[canales]`.

En cuanto al canal en MNIST, ya que está dada en escala de grises, la dimensión de píxel se establece en 1.

```
# load data
(X_train, y_train), (X_test, y_test) = mnist.load_data()

# reshape to be [samples][width][height][channels]
X_train = X_train.reshape((X_train.shape[0], 28, 28, 1)).astype('float')
X_test = X_test.reshape((X_test.shape[0], 28, 28, 1)).astype('float')
```

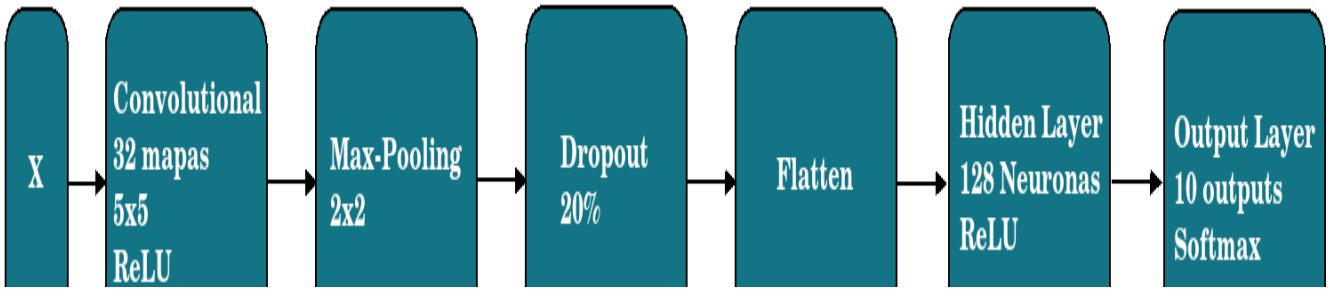
```
Start coding or generate with AI.
```

Normalizamos los valores de los píxeles en el rango 0 y 1 y realizar OHE en el target.

```
# normalize inputs from 0-255 to 0-1
X_train = X_train / 255
X_test = X_test / 255

# one hot encode outputs
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)

num_classes = y_test.shape[1]
```

A continuación, definimos nuestro modelo de red neuronal:

1. La primera capa oculta es una capa convolucional llamada `Conv2D`.
    - Tiene 32 mapas de características, con un tamaño de $5 × 5$ y una función de activación ReLu.
2. Capa Pooling `MaxPooling2D`.
    - Tamaño de pacht de $2 × 2$.
3. Capa de regularización `Dropout`.
4. Capa `Flatten` para conversión de la matriz 2D en un vector (1D).
5. Capa `Dense` con 128 neuronas y la función de activación ReLU.
6. Capa de salida con 10 neuronas para las 10 clases y una función de activación **Softmax**.
7. La compilación con ADAM, pérdida logarítmica como función de coste y Accuracy como métrica.



```
# define a simple CNN model
def baseline_model():
    # create model
    model = Sequential()
    model.add(Conv2D(32, (5,5), input_shape=(28,28,1), activation='relu'))
    model.add(MaxPooling2D())
    model.add(Dropout(0.2))
    model.add(Flatten())
    model.add(Dense(128, activation='relu'))
    model.add(Dense(num_classes, activation='softmax'))

    # Compile model
    model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['


    return model
```

Entrenamos con 10 épocas a un tamaño de batch de 200.

```
# build the model
model = baseline_model()

# Fit the model
model.fit(X_train, y_train, validation_data=(X_test, y_test), epochs=10, batch_

# Final evaluation of the model
scores = model.evaluate(X_test, y_test, verbose=0)
print("Error del modelo de CNN linea base: %.2f%%" % (100-scores[1]*100))
```

```
/usr/local/lib/python3.11/dist-packages/keras/src/layers/convolutional/base
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
Epoch 1/10
300/300 ━━━━━━━━━━━━━━━━━━━━ 35s 110ms/step - accuracy: 0.8556 - loss: 0.51
Epoch 2/10
300/300 ━━━━━━━━━━━━━━━━━━━━ 31s 105ms/step - accuracy: 0.9775 - loss: 0.07
Epoch 3/10
300/300 ━━━━━━━━━━━━━━━━━━━━ 33s 109ms/step - accuracy: 0.9834 - loss: 0.05
Epoch 4/10
300/300 ━━━━━━━━━━━━━━━━━━━━ 31s 105ms/step - accuracy: 0.9869 - loss: 0.04
Epoch 5/10
300/300 ━━━━━━━━━━━━━━━━━━━━ 41s 104ms/step - accuracy: 0.9885 - loss: 0.03
Epoch 6/10
300/300 ━━━━━━━━━━━━━━━━━━━━ 32s 108ms/step - accuracy: 0.9919 - loss: 0.02
Epoch 7/10
300/300 ━━━━━━━━━━━━━━━━━━━━ 41s 108ms/step - accuracy: 0.9922 - loss: 0.02
Epoch 8/10
300/300 ━━━━━━━━━━━━━━━━━━━━ 40s 103ms/step - accuracy: 0.9929 - loss: 0.02
Epoch 9/10
300/300 ━━━━━━━━━━━━━━━━━━━━ 31s 103ms/step - accuracy: 0.9950 - loss: 0.01
Epoch 10/10
300/300 ━━━━━━━━━━━━━━━━━━━━ 42s 107ms/step - accuracy: 0.9955 - loss: 0.01
```

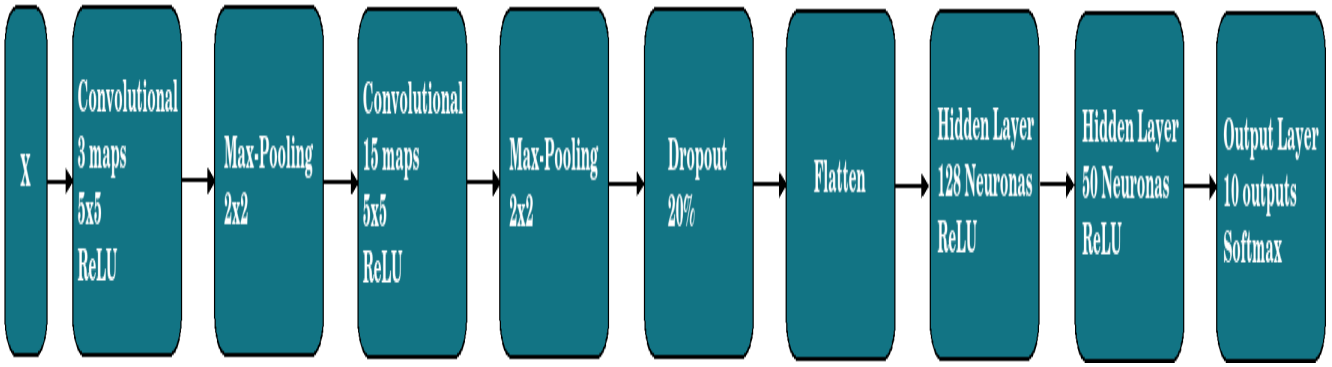
```
Error del modelo de CNN linea base: 1.02%
```

---

5. CNN más profunda para MNIST

Esta vez definimos una arquitectura con más capas de convolucionales, Max-pooling y capas completamente conectadas.

1. Capa convolucional con 30 mapas de tamaño 5×5.
2. Capa de Pooling con patch de 2×2.
3. Capa convolucional con 15 mapas de tamaño 3×3.
4. Capa de Pooling con patch de 2×2.
5. Capa de Dropout del 20%.
6. Capa Flatten.
7. Capa completamente conectada con 128 neuronas y ReLu.
8. Capa completamente conectada con 50 neuronas y ReLu
9. Capa de salida cpm activación Softmax.
10. La compilación con ADAM, pérdida logarítmica como función de coste y Accuracy como métrica.




```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Dropout, Flatten, Dense
from tensorflow.keras.optimizers import Adam

def larger_model():
    # Crear modelo
    model = Sequential()
    model.add(Conv2D(30, (5, 5), input_shape=(28, 28, 1), activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))

    model.add(Conv2D(15, (3, 3), activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))

    model.add(Dropout(0.2))
```

```
    model.add(Dropout(0.2))
    model.add(Flatten())

    model.add(Dense(128, activation='relu'))
    model.add(Dense(50, activation='relu'))
    model.add(Dense(num_classes, activation='softmax'))

    # Compilar modelo
    model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['a

    return model

# Construir el modelo
model = larger_model()

# Entrenar el modelo
model.fit(X_train, y_train, validation_data=(X_test, y_test), epochs=10, batch_s

# Evaluar el modelo
scores = model.evaluate(X_test, y_test, verbose=0)
print("Error del modelo CNN profunda: %.2f%%" % (100 - scores[1] * 100))
```

```
Epoch 1/10
300/300 - 39s - 131ms/step - accuracy: 0.8802 - loss: 0.3881 - val_accuracy
Epoch 2/10
300/300 - 37s - 122ms/step - accuracy: 0.9693 - loss: 0.0978 - val_accuracy
Epoch 3/10
300/300 - 35s - 117ms/step - accuracy: 0.9777 - loss: 0.0707 - val_accuracy
Epoch 4/10
300/300 - 41s - 136ms/step - accuracy: 0.9825 - loss: 0.0557 - val_accuracy
Epoch 5/10
300/300 - 41s - 135ms/step - accuracy: 0.9847 - loss: 0.0481 - val_accuracy
Epoch 6/10
300/300 - 39s - 129ms/step - accuracy: 0.9864 - loss: 0.0423 - val_accuracy
Epoch 7/10
300/300 - 42s - 141ms/step - accuracy: 0.9879 - loss: 0.0394 - val_accuracy
Epoch 8/10
300/300 - 42s - 141ms/step - accuracy: 0.9884 - loss: 0.0351 - val_accuracy
Epoch 9/10
300/300 - 40s - 132ms/step - accuracy: 0.9897 - loss: 0.0321 - val_accuracy
Epoch 10/10
300/300 - 40s - 135ms/step - accuracy: 0.9907 - loss: 0.0282 - val_accuracy
Error del modelo CNN profunda: 0.86%
```
