Ajuste de una Red Neuronal Convolucional con MNIST en Keras

October 6, 2019

1 MNIST en Keras

Partiendo de la base de datos MNIST de imágenes de dígitos escritos a mano, vamos a realizar el ajuste de una Red Neuronal Convolucional utilizando la librería Keras. El proceso se realizará en varios pasos:

- Incluir librerías y definir parámetros.
- Cargar los datos.
- Ajustar los datos.
- Definir el modelo en Keras.
- Ajustar el modelo.
- Evaluar el modelo.

Adicionalmente vamos crear la matriz de confusión.
Finalmente vamos a revisar algunas particularidades con los datos.

1.1 Inclusión de librerías

Principalmente se incluye keras y desde keras se pueden cargar los datos MNIST directamente. Vamos a utilizar scikit learn para hacer la matriz de confusión y matplotlib para realizar los gráficos.

```
In [1]: import keras
        from keras.datasets import mnist
        from keras.models import Sequential
        from keras.layers import Dense, Dropout, Flatten
        from keras.layers import Conv2D, MaxPooling2D
        from keras import backend as K

        from sklearn import metrics

        import numpy as np
        from matplotlib import pyplot as plt
        %matplotlib inline
```

```
Using TensorFlow backend.
```

1.2 Definición de parámetros

- `batch_size` define el tamaño del lote para entrenar.
- `num_classes` indica cuantas clases genera la clasificación.
- `epochs` indica el número de vueltas completas que se hacen con el conjunto de datos de entrenamiento.
- `img_rows`, `img_cols` indica el tamaño de las imágenes en pixeles.

```
In [2]: batch_size = 128
        num_classes = 10
        epochs = 12

        # Dimensión de las imágenes
        img_rows, img_cols = 28, 28
```

1.3 Obtención de los datos

Los datos se obtienen con `mnist.load_data()` provisto por keras y se dividen en muestra de entrenamiento y prueba.

```
In [3]: # Dividir los datos entre los datos de entrenamiento y prueba.
        (x_train, y_train), (x_test, y_test) = mnist.load_data()
        (x_data, y_data), (x_data_test, y_data_test) = mnist.load_data()

        if K.image_data_format() == 'channels_first':
            x_train = x_train.reshape(x_train.shape[0], 1, img_rows, img_cols)
            x_test = x_test.reshape(x_test.shape[0], 1, img_rows, img_cols)
            input_shape = (1, img_rows, img_cols)
        else:
            x_train = x_train.reshape(x_train.shape[0], img_rows, img_cols, 1)
            x_test = x_test.reshape(x_test.shape[0], img_rows, img_cols, 1)
            input_shape = (img_rows, img_cols, 1)
```

1.4 Normalización de los datos

Los valores de las imágenes se pasan de una escala entre 0 y 1. Son 60 mil imágenes de entrenamiento y 10 mil de prueba de dimensión 28×28 en escala de grises y por eso es sólo 1 canal.

```
In [4]: x_train = x_train.astype('float32')
        x_test = x_test.astype('float32')
        x_train /= 255
        x_test /= 255
        print('x_train shape:', x_train.shape)
        print(x_train.shape[0], 'train samples')
        print(x_test.shape[0], 'test samples')
```

```
('x_train shape:', (60000, 28, 28, 1))
(60000, 'train samples')
(10000, 'test samples')
```

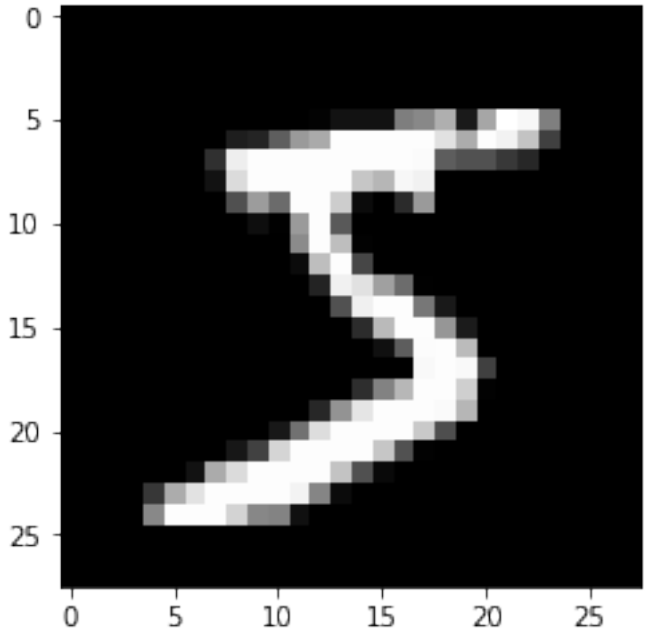
1.4.1 Rutina para dibujar un dígito

Vamos a definir una rutina para dibujar la imagen de un dígito. Se va a utilizar para explorar los datos y verificar los resultados de la inferencia realizada por el modelo.

```
In [5]: def plot_image(data):
            image = data
            image = np.array(image, dtype='float')
            pixels = image.reshape((28, 28))
            plt.imshow(pixels, cmap='gray')
            plt.show()
```

Vamos a mostrar la imagen del primer dígito de la muestra de entrenamiento.

```
In [6]: plot_image(x_train[0])
```



Ahora vamos a ver el valor del resultado observado en el primer dígito de la muestra de entrenamiento.

```
In [7]: y_train[0]

Out[7]: 5
```

Las clases se convierten en matrices binarias para utilizarlas con la función de pérdida `categorical_crossentropy`.

```
In [8]: y_train = keras.utils.to_categorical(y_train, num_classes)
        y_test = keras.utils.to_categorical(y_test, num_classes)
```

```
In [9]: y_train[0]

Out[9]: array([0., 0., 0., 0., 0., 1., 0., 0., 0., 0.], dtype=float32)
```

Podemos ver que el sexto valor es 1 indicando que la etiqueta de la primera imagen de la muestra de entrenamiento es un cinco. Recordemos que el vector es de 10 posiciones que inicia desde el 0.

Ahora vamos a crear el modelo con dos capas de convolución de 32 y 64 mapas de características respectivamente con kernel de 3×3 y ReLU como función de activación. Luego se aplica una capa de sub muestraje con ventana de 2×2 y finalmente una red reuronal conectada con 128 neuronas.

Para el modelo se utiliza la entropía, Adadelta como algoritmo de optimización y la métrica es la exactitud.

```
In [10]: model = Sequential()
         model.add(Conv2D(32, kernel_size=(3, 3),
                          activation='relu',
                          input_shape=input_shape))
         model.add(Conv2D(64, (3, 3), activation='relu'))
         model.add(MaxPooling2D(pool_size=(2, 2)))
         model.add(Flatten())
         model.add(Dense(128, activation='relu'))
         model.add(Dense(num_classes, activation='softmax'))

         model.compile(loss=keras.losses.categorical_crossentropy,
                       optimizer=keras.optimizers.Adadelta(),
                       metrics=['accuracy'])
```

Ahora podemos inspeccionar el modelo y ver las características y parámetros.

```
In [11]: model.summary()

_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 26, 26, 32)        320
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 24, 24, 64)        18496
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 12, 12, 64)        0
_________________________________________________________________
flatten_1 (Flatten)          (None, 9216)              0
_________________________________________________________________
dense_1 (Dense)              (None, 128)               1179776
_________________________________________________________________
dense_2 (Dense)              (None, 10)                1290
=================================================================
Total params: 1,199,882
Trainable params: 1,199,882
```

```
Non-trainable params: 0
_________________________________________________________________
```

El modelo se ajusta a los datos de entrenamiento en lotes de 128 y se realizarán 12 ciclos, para luego validar con los datos de prueba.

```
In [12]: model.fit(x_train, y_train,
                   batch_size=batch_size,
                   epochs=epochs,
                   verbose=1,
                   validation_data=(x_test, y_test))

Train on 60000 samples, validate on 10000 samples
Epoch 1/12
60000/60000 [==============================] - 90s 1ms/step - loss: 0.2090 - acc: 0.9370 - val
Epoch 2/12
60000/60000 [==============================] - 92s 2ms/step - loss: 0.0495 - acc: 0.9851 - val
Epoch 3/12
60000/60000 [==============================] - 87s 1ms/step - loss: 0.0310 - acc: 0.9902 - val
Epoch 4/12
60000/60000 [==============================] - 91s 2ms/step - loss: 0.0210 - acc: 0.9933 - val
Epoch 5/12
60000/60000 [==============================] - 96s 2ms/step - loss: 0.0141 - acc: 0.9960 - val
Epoch 6/12
60000/60000 [==============================] - 97s 2ms/step - loss: 0.0099 - acc: 0.9972 - val
Epoch 7/12
60000/60000 [==============================] - 100s 2ms/step - loss: 0.0062 - acc: 0.9984 - va
Epoch 8/12
60000/60000 [==============================] - 100s 2ms/step - loss: 0.0044 - acc: 0.9988 - va
Epoch 9/12
60000/60000 [==============================] - 97s 2ms/step - loss: 0.0031 - acc: 0.9992 - val
Epoch 10/12
60000/60000 [==============================] - 94s 2ms/step - loss: 0.0023 - acc: 0.9994 - val
Epoch 11/12
60000/60000 [==============================] - 97s 2ms/step - loss: 0.0015 - acc: 0.9996 - val
Epoch 12/12
60000/60000 [==============================] - 95s 2ms/step - loss: 8.0593e-04 - acc: 0.9999 -


Out[12]: <keras.callbacks.History at 0x1a31eeeb90>
```

Luego del ajuste podemos salvar el modelo para utilizarlo sin realizar el ajuste de nuevo. Esta es una gran ventaja ya que se pueden compartir los resultados.

```
In [13]: model.save("MNIST-keras-no-drop.h5")
```

```
In [12]: from keras.models import load_model
         # Carga del modelo
```

```
        model = load_model('MNIST-keras-no-drop.h5')
        # Resumen de la estructura del modelo
        model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 26, 26, 32)        320
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 24, 24, 64)        18496
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 12, 12, 64)        0
_________________________________________________________________
flatten_1 (Flatten)          (None, 9216)              0
_________________________________________________________________
dense_1 (Dense)              (None, 128)               1179776
_________________________________________________________________
dense_2 (Dense)              (None, 10)                1290
=================================================================
Total params: 1,199,882
Trainable params: 1,199,882
Non-trainable params: 0
_________________________________________________________________
```

Luego del ajuste que tomó alrededor de un minuto y medio por vuelta podemos ver el desempeño de 99%.

```
In [13]: score = model.evaluate(x_test, y_test, verbose=0)
         print('Test loss:', score[0])
         print('Test accuracy:', score[1])
```

```
('Test loss:', 0.04357766299277946)
('Test accuracy:', 0.9902)
```

Si adicionalmente aplicamos `dropout` el modelo mejora ligeramente.

```
In [14]: model2 = Sequential()
         model2.add(Conv2D(32, kernel_size=(3, 3),
                           activation='relu',
                           input_shape=input_shape))
         model2.add(Conv2D(64, (3, 3), activation='relu'))
         model2.add(MaxPooling2D(pool_size=(2, 2)))
         model2.add(Dropout(0.25))
         model2.add(Flatten())
         model2.add(Dense(128, activation='relu'))
         model2.add(Dropout(0.5))
         model2.add(Dense(num_classes, activation='softmax'))
```

```
        model2.compile(loss=keras.losses.categorical_crossentropy,
                       optimizer=keras.optimizers.Adadelta(),
                       metrics=['accuracy'])
```

```
In [18]: model2.fit(x_train, y_train,
                    batch_size=batch_size,
                    epochs=epochs,
                    verbose=1,
                    validation_data=(x_test, y_test))
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/12
60000/60000 [==============================] - 103s 2ms/step - loss: 0.2684 - acc: 0.9188 - val
Epoch 2/12
60000/60000 [==============================] - 107s 2ms/step - loss: 0.0896 - acc: 0.9742 - val
Epoch 3/12
60000/60000 [==============================] - 108s 2ms/step - loss: 0.0661 - acc: 0.9805 - val
Epoch 4/12
60000/60000 [==============================] - 113s 2ms/step - loss: 0.0532 - acc: 0.9842 - val
Epoch 5/12
60000/60000 [==============================] - 106s 2ms/step - loss: 0.0485 - acc: 0.9857 - val
Epoch 6/12
60000/60000 [==============================] - 104s 2ms/step - loss: 0.0427 - acc: 0.9868 - val
Epoch 7/12
60000/60000 [==============================] - 106s 2ms/step - loss: 0.0380 - acc: 0.9887 - val
Epoch 8/12
60000/60000 [==============================] - 111s 2ms/step - loss: 0.0332 - acc: 0.9895 - val
Epoch 9/12
60000/60000 [==============================] - 102s 2ms/step - loss: 0.0314 - acc: 0.9902 - val
Epoch 10/12
60000/60000 [==============================] - 98s 2ms/step - loss: 0.0302 - acc: 0.9909 - val_
Epoch 11/12
60000/60000 [==============================] - 99s 2ms/step - loss: 0.0270 - acc: 0.9919 - val_
Epoch 12/12
60000/60000 [==============================] - 102s 2ms/step - loss: 0.0257 - acc: 0.9918 - val
```

```
Out[18]: <keras.callbacks.History at 0x1a43d89b10>
```

```
In [20]: model2.save("MNIST-keras.h5")
```

Ahora podemos cargar el modelo salvado para realizar inferencias. Lo interesante es que una vez que el modelo está ajustado, lo podemos utilizar para inferir. De esta forma se pueden crear bases de datos de modelos pre entrenados que permiten realizar inferencias. Es como contar con los parámetros de un programa que ahora se puede ejecutar en cualquier lugar.

```
In [15]: from keras.models import load_model
         # Cargar modelo
```

```
model = load_model('MNIST-keras.h5')
# Resumen de la estructura del modelo
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_3 (Conv2D)            (None, 26, 26, 32)        320
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 24, 24, 64)        18496
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 12, 12, 64)        0
_________________________________________________________________
dropout_1 (Dropout)          (None, 12, 12, 64)        0
_________________________________________________________________
flatten_2 (Flatten)          (None, 9216)              0
_________________________________________________________________
dense_3 (Dense)              (None, 128)               1179776
_________________________________________________________________
dropout_2 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_4 (Dense)              (None, 10)                1290
=================================================================
Total params: 1,199,882
Trainable params: 1,199,882
Non-trainable params: 0
_________________________________________________________________
```

```
In [16]: score = model.evaluate(x_test, y_test, verbose=0)
         print('Test loss:', score[0])
         print('Test accuracy:', score[1])
```

```
('Test loss:', 0.028851404254680527)
('Test accuracy:', 0.9907)
```

Vamos a almacenar en `y_pred` los valores inferidos de los datos de prueba `x_test`.

```
In [17]: y_pred = model.predict(x_test)
```

Con las inferencias ahora podemos hacer la matriz de confusión y a partir de esta generar métricas adicionales.

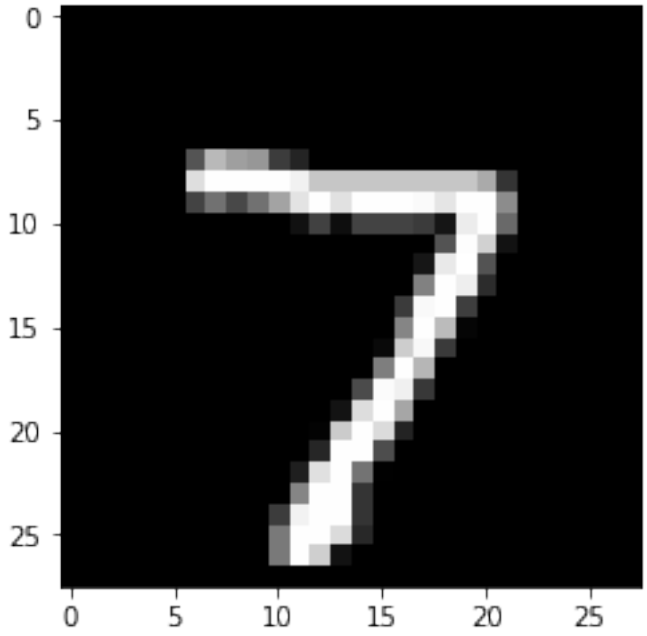
```
In [18]: matrix = metrics.confusion_matrix(y_test.argmax(axis=1), y_pred.argmax(axis=1))
         matrix
```

```
Out[18]: array([[ 977,    0,    0,    0,    0,    0,    1,    1,    1,    0],
                [   0, 1131,    1,    1,    0,    0,    2,    0,    0,    0],
```

```
[   1,    2, 1023,    0,    1,    0,    1,    4,    0,    0],
[   0,    0,    2, 1004,    0,    3,    0,    0,    1,    0],
[   0,    0,    0,    0,  974,    0,    4,    0,    1,    3],
[   1,    0,    0,    5,    0,  884,    2,    0,    0,    0],
[   4,    2,    0,    0,    1,    3,  948,    0,    0,    0],
[   0,    1,    5,    1,    0,    0,    0, 1020,    1,    0],
[   2,    0,    3,    1,    1,    1,    1,    1,  963,    1],
[   3,    2,    0,    1,    5,    6,    1,    4,    4,  983]])
```

Vamos a probar con el primer dato de prueba.

```
In [19]: plot_image(x_test[0])
```



Observemos el resultado esperado.

```
In [20]: y_test[0]
```

```
Out[20]: array([0., 0., 0., 0., 0., 0., 0., 1., 0., 0.], dtype=float32)
```

Ahora comparamos con el valor inferido.

```
In [21]: y_pred[0]
```

```
Out[21]: array([8.0566040e-12, 1.7861893e-10, 1.9713897e-09, 5.6523755e-09,
                1.5563544e-12, 4.1730020e-13, 4.4064523e-16, 1.0000000e+00,
                1.0673628e-12, 1.8242463e-09], dtype=float32)
```

Podemos corroborar que coincide.

```
In [22]: np.where(y_pred[0]==np.amax(y_pred[0]))[0][0]
```

```
Out[22]: 7
```

1.5 Matriz de confusión

Para dibujar la matriz de confusión se utiliza la rutina confusion_matrix de scikit-learn.

A esta rutina se le pasan los datos de prueba y las predicciones generadas por el modelo.

```
In [23]: matrix = metrics.confusion_matrix(y_test.argmax(axis=1), y_pred.argmax(axis=1))
         matrix
```

```
Out[23]: array([[ 977,    0,    0,    0,    0,    0,    1,    1,    1,    0],
                [   0, 1131,    1,    1,    0,    0,    2,    0,    0,    0],
                [   1,    2, 1023,    0,    1,    0,    1,    4,    0,    0],
                [   0,    0,    2, 1004,    0,    3,    0,    0,    1,    0],
                [   0,    0,    0,    0,  974,    0,    4,    0,    1,    3],
                [   1,    0,    0,    5,    0,  884,    2,    0,    0,    0],
                [   4,    2,    0,    0,    1,    3,  948,    0,    0,    0],
                [   0,    1,    5,    1,    0,    0,    0, 1020,    1,    0],
                [   2,    0,    3,    1,    1,    1,    1,    1,  963,    1],
                [   3,    2,    0,    1,    5,    6,    1,    4,    4,  983]])
```

Para normalizar los datos de la matriz de confusión se normaliza cada fila.

```
In [24]: matrix.astype('float') / matrix.sum(axis=1)[:, np.newaxis]
```

```
Out[24]: array([[9.96938776e-01, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00,
                 0.00000000e+00, 0.00000000e+00, 1.02040816e-03, 1.02040816e-03,
                 1.02040816e-03, 0.00000000e+00],
                [0.00000000e+00, 9.96475771e-01, 8.81057269e-04, 8.81057269e-04,
                 0.00000000e+00, 0.00000000e+00, 1.76211454e-03, 0.00000000e+00,
                 0.00000000e+00, 0.00000000e+00],
                [9.68992248e-04, 1.93798450e-03, 9.91279070e-01, 0.00000000e+00,
                 9.68992248e-04, 0.00000000e+00, 9.68992248e-04, 3.87596899e-03,
                 0.00000000e+00, 0.00000000e+00],
                [0.00000000e+00, 0.00000000e+00, 1.98019802e-03, 9.94059406e-01,
                 0.00000000e+00, 2.97029703e-03, 0.00000000e+00, 0.00000000e+00,
                 9.90099010e-04, 0.00000000e+00],
                [0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00,
                 9.91853360e-01, 0.00000000e+00, 4.07331976e-03, 0.00000000e+00,
                 1.01832994e-03, 3.05498982e-03],
                [1.12107623e-03, 0.00000000e+00, 0.00000000e+00, 5.60538117e-03,
                 0.00000000e+00, 9.91031390e-01, 2.24215247e-03, 0.00000000e+00,
                 0.00000000e+00, 0.00000000e+00],
                [4.17536534e-03, 2.08768267e-03, 0.00000000e+00, 0.00000000e+00,
                 1.04384134e-03, 3.13152401e-03, 9.89561587e-01, 0.00000000e+00,
                 0.00000000e+00, 0.00000000e+00],
                [0.00000000e+00, 9.72762646e-04, 4.86381323e-03, 9.72762646e-04,
                 0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 9.92217899e-01,
                 9.72762646e-04, 0.00000000e+00],
                [2.05338809e-03, 0.00000000e+00, 3.08008214e-03, 1.02669405e-03,
                 1.02669405e-03, 1.02669405e-03, 1.02669405e-03, 1.02669405e-03,
```

```
       9.88706366e-01, 1.02669405e-03],
      [2.97324083e-03, 1.98216056e-03, 0.00000000e+00, 9.91080278e-04,
       4.95540139e-03, 5.94648167e-03, 9.91080278e-04, 3.96432111e-03,
       3.96432111e-03, 9.74231913e-01]])
```

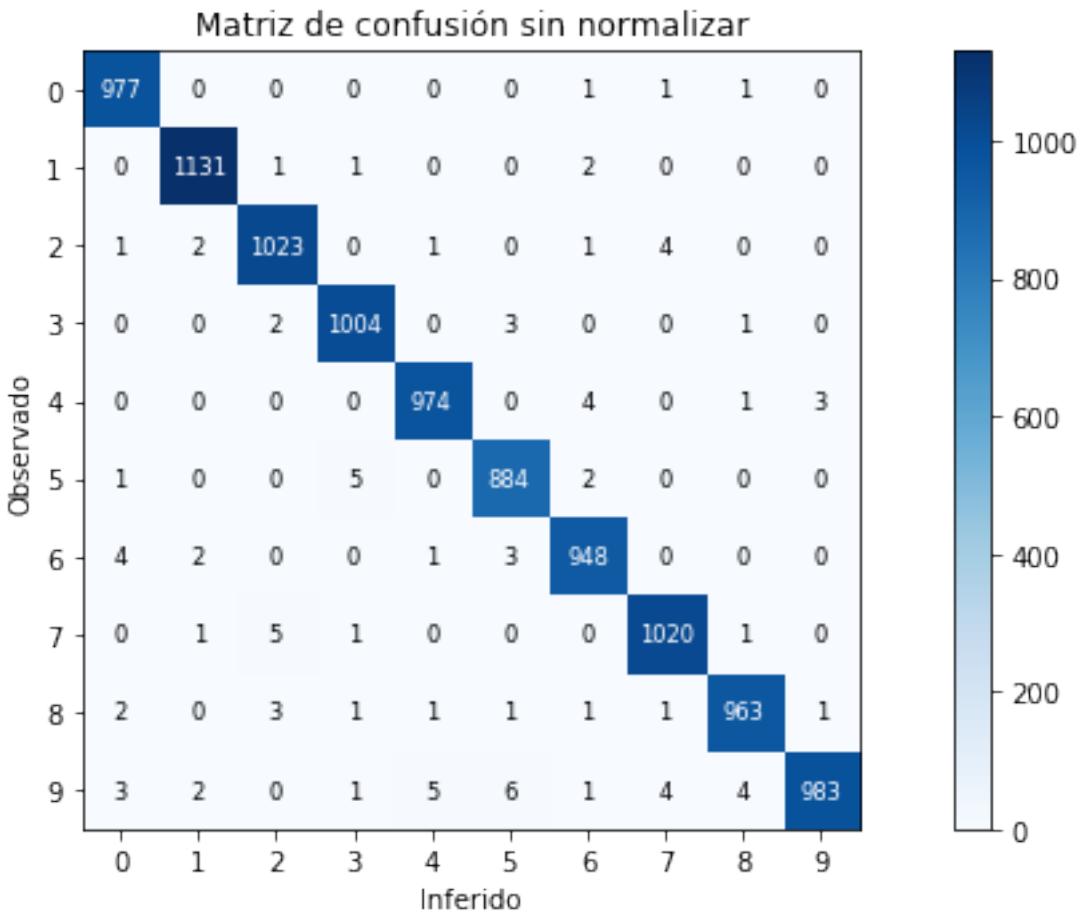
Hemos modificado el ejemplo para mostrar la matriz de confusión del ajuste con los datos MNIST.

```
In [25]: from sklearn.model_selection import train_test_split
         from sklearn.metrics import confusion_matrix
         from sklearn.utils.multiclass import unique_labels

         class_names = np.array(['0', '1', '2', '3', '4', '5', '6', '7', '8', '9'])

         def plot_confusion_matrix(y_true, y_pred, classes,
                                   normalize=False,
                                   title=None,
                                   cmap=plt.cm.Blues,
                                   printtext=False):
             """
             This function prints and plots the confusion matrix.
             Normalization can be applied by setting `normalize=True`.
             """
             if not title:
                 if normalize:
                     title = 'Matriz de confusión normalizada'
                 else:
                     title = 'Matriz de confusión sin normalizar'

             # Compute confusion matrix
             cm = confusion_matrix(y_true.argmax(axis=1), y_pred.argmax(axis=1))
             # Only use the labels that appear in the data
             classes = classes[unique_labels(y_true)]

             if normalize:
                 cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]

             if printtext:
                 if normalize:
                     print("Matriz de confusión normalizada")
                 else:
                     print('Matriz de confusión sin normalizar')
                 print(cm)

             fig, ax = plt.subplots()
             im = ax.imshow(cm, interpolation='nearest', cmap=cmap)
             ax.figure.colorbar(im, ax=ax)
             fig.set_size_inches(10, 5)
```

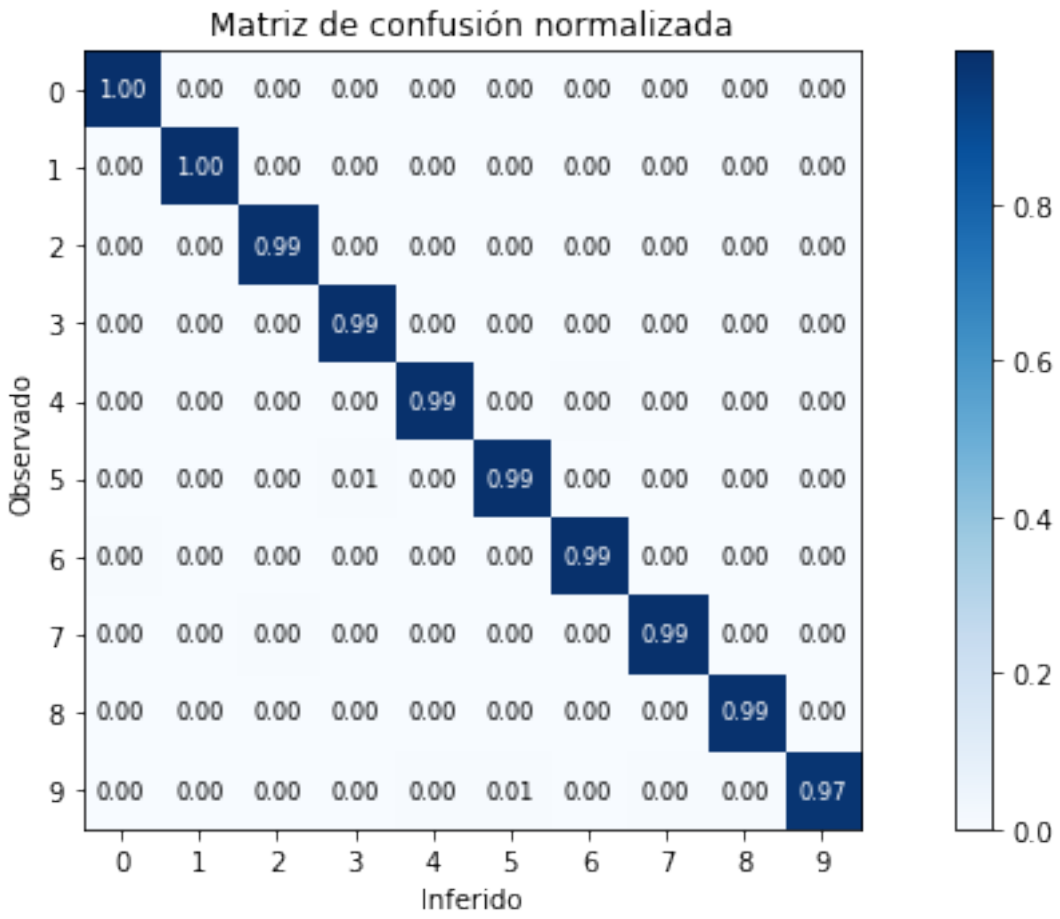
```
        # We want to show all ticks...
        ax.set(xticks=np.arange(cm.shape[1]),
               yticks=np.arange(cm.shape[0]),
               # ... and label them with the respective list entries
               xticklabels=classes, yticklabels=classes,
               title=title,
               ylabel='Observado',
               xlabel='Inferido')

        # Loop over data dimensions and create text annotations.
        fmt = '.2f' if normalize else 'd'
        thresh = cm.max() / 2.
        for i in range(cm.shape[0]):
            for j in range(cm.shape[1]):
                ax.text(j, i, format(cm[i, j], fmt),
                        fontsize='smaller',
                        ha="center", va="center",
                        color="white" if cm[i, j] > thresh else "black")
        fig.tight_layout()
        return ax
```

Ahora podemos ver de forma visual la matriz de confusión normalizada o sin normalizar.

```
In [26]: np.set_printoptions(precision=2)
         # Graficar la matriz de confusión sin normalizar
         plot_confusion_matrix(y_test, y_pred, classes=class_names,
                               title=u"Matriz de confusi\u00f3n sin normalizar")
         plt.show()
```



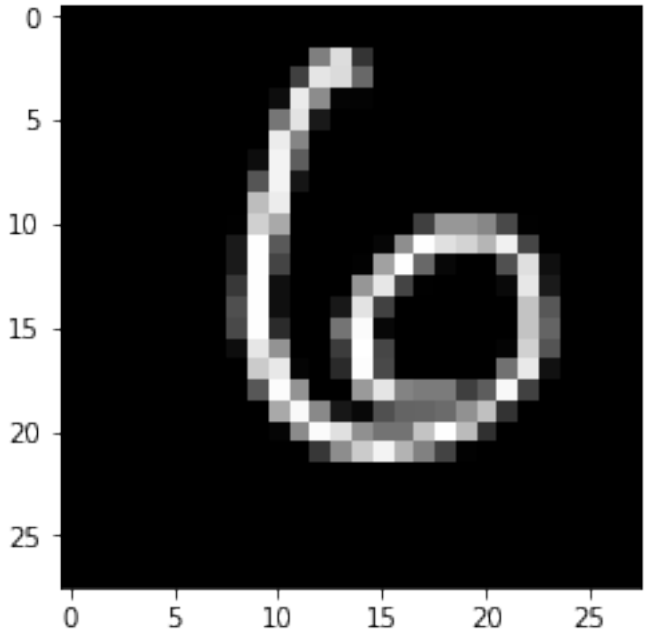
In [27]:
```
np.set_printoptions(precision=0)
# Matriz de confusión normalizada
plot_confusion_matrix(y_test, y_pred, classes=class_names, normalize=True,
                      title=u"Matriz de confusi\u00f3n normalizada")

plt.show()
```



1.6 Errores del modelo

Encontramos un ejemplo donde el modelo no genera el resultado esperado y en lugar de inferir un 6 infiere un 0.

```
In [28]: plot_image(x_test[100])
```




```
In [29]: y_test[100]
```

```
Out[29]: array([0., 0., 0., 0., 0., 0., 1., 0., 0., 0.], dtype=float32)
```

```
In [30]: result=model.predict(np.expand_dims(x_test[100], axis=0))
         result
```

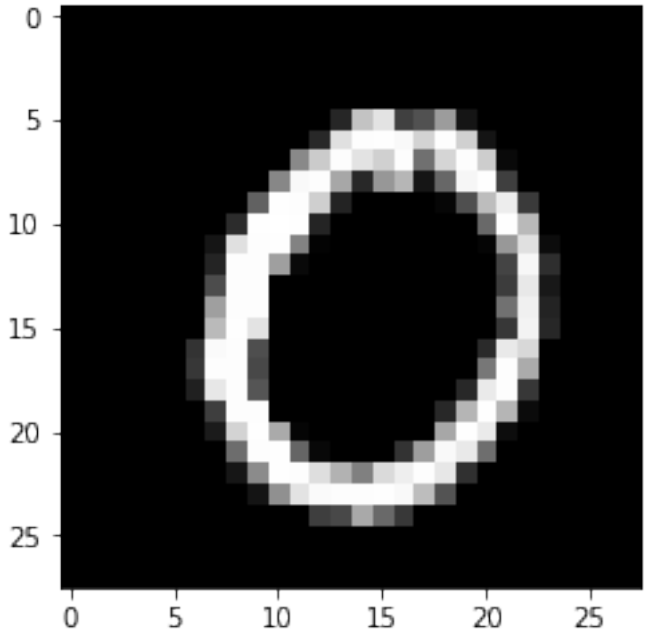
```
Out[30]: array([[4.e-11, 7.e-15, 6.e-17, 1.e-17, 1.e-13, 2.e-11, 1.e+00, 5.e-17,
                 7.e-13, 9.e-18]], dtype=float32)
```

```
In [31]: np.where(result==np.amax(result))[0][0]
```

```
Out[31]: 0
```

1.6.1 Otra prueba con un 0

```
In [32]: plot_image(x_test[101])
```




```
In [33]: y_test[101]
```

```
Out[33]: array([1., 0., 0., 0., 0., 0., 0., 0., 0., 0.], dtype=float32)
```

```
In [34]: result=model.predict(np.expand_dims(x_test[101], axis=0))
         result
```

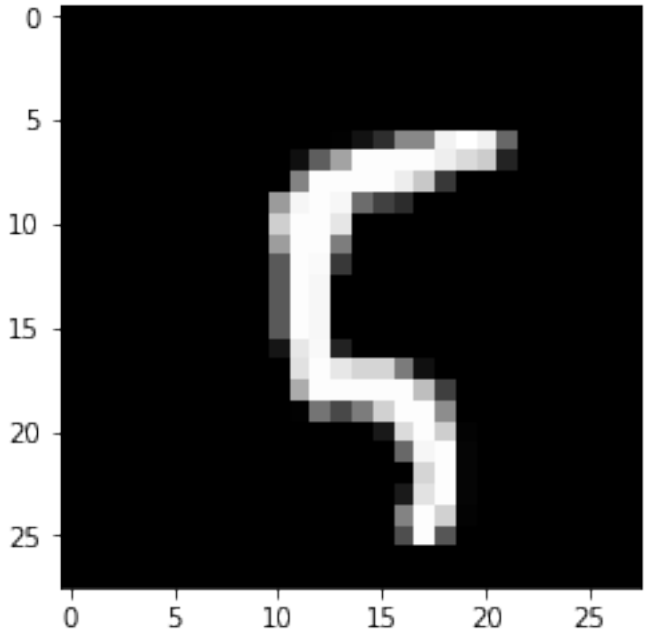
```
Out[34]: array([[1.e+00, 1.e-14, 3.e-13, 4.e-14, 9.e-16, 2.e-11, 3.e-07, 1.e-12,
                 7.e-11, 3.e-10]], dtype=float32)
```

```
In [35]: np.where(result==np.amax(result))[0][0]
```

```
Out[35]: 0
```

Ejemplo con figura incompleta

```
In [36]: plot_image(x_train[100])
```




```
In [37]: y_train[100]
```

```
Out[37]: array([0., 0., 0., 0., 0., 1., 0., 0., 0., 0.], dtype=float32)
```

```
In [38]: number=np.expand_dims(x_train[100], axis=0)
```

```
In [39]: arr=model.predict(number)[0]
         arr
```

```
Out[39]: array([8.e-08, 7.e-09, 2.e-11, 4.e-08, 9.e-09, 1.e+00, 5.e-05, 9.e-10,
                6.e-06, 6.e-06], dtype=float32)
```

```
In [40]: np.where(arr==np.amax(arr))[0][0]
```

```
Out[40]: 5
```