ML_CLASIFICACION_CNN_01	Clasificación de dígitos escritos a mano	ML

El problema a resolver es la clasificación de imágenes (en escala de grises, 28 x 28 pixeles) de dígitos escritos a mano, en las 10 categorías (0 a 9). Utilizamos para ello la base de datos MNIST (*http://yann.lecun.com/exdb/mnist/*), elaborada por el NIST (National Institute of Standards and Technology) y compuesta por un conjunto de 60.000 imágenes para entrenamiento, y 10.000 imágenes para prueba y validación.

Ejemplos de imágenes (28x28) de dígitos escritos a mano

La base de datos MNIST ya viene precargada dentro de la librería Keras. La instalación de Keras se puede realizar directamente desde el entorno Anaconda.

En esta práctica para resolver el problema se construye una **red neuronal basada en convolución** (CNN – Convolutional Neural Network).

SOLUCIÓN

Definir las librerías a utilizar

```
# Importar librerías a utilizar
import numpy as np
import matplotlib.pyplot as plt
```

Cargamos en Python la base de datos MNIST, y asignamos las imágenes y etiquetas de los conjuntos para entrenamiento y prueba.

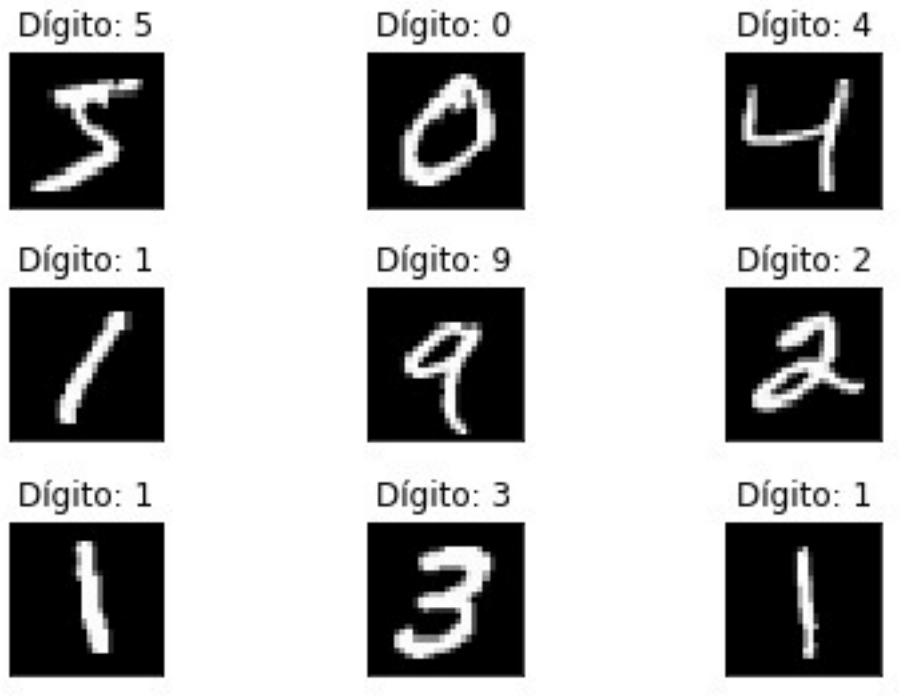
```
# Cargar la base de datos MNIST y asignar las imágenes y etiquetas de los conjuntos para
entrenamiento y prueba
from keras.datasets import mnist
(X_train, y_train), (X_test, y_test) = mnist.load_data()
```

Las imágenes están codificadas en arrays (0 o 1), y las etiquetas son un array de números (0 a 9). Realizamos la consulta de algunas imágenes y etiquetas para entrenamiento.

```
# Las imágenes están codificadas en arrays (0 o 1), y las etiquetas son un array números (0 a 9)
print("X_train shape", X_train.shape)
print("y_train shape", y_train.shape)
print("X_test shape", X_test.shape)
print("y_test shape", y_test.shape)

X_train shape (60000, 28, 28)
y_train shape (60000,)
X_test shape (10000, 28, 28)
y_test shape (10000,)
```

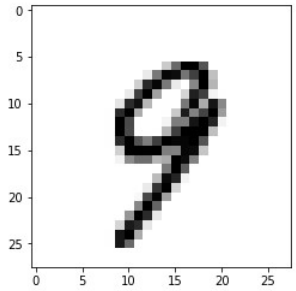
Visualizar algunos ejemplos de la base de datos

```
# Consultar algunas imágenes y etiquetas para entrenamiento
# Visualizar algunos ejemplos
fig = plt.figure()
for i in range(9):
  plt.subplot(3,3,i+1)
  plt.tight_layout()
  plt.imshow(X_train[i], cmap='gray', interpolation='none')
  plt.title("Dígito: {}".format(y_train[i]))
  plt.xticks([])
  plt.yticks([])
fig
plt.show()
```



Visualizar un ejemplo concreto de la base de datos

```
# Visualizar un ejemplo de las 60.000 imágenes
digit = X_train[4345]
plt.imshow(digit, cmap=plt.cm.binary)
plt.show()
```



Transformar y normalizar los datos a valores en el rango de 0 a 1. (Normalizar el histograma de la imagen)

```
# Antes de realizar el entrenamiento, preparar los datos transformando las imágenes
# iniciales con valores entre 0 y 255 (negro a blanco), a valores binarizados (0 a 1)
X_train = X_train.reshape((60000, 28 * 28))
X_train = X_train.astype('float32') / 255
X_test = X_test.reshape((10000, 28 * 28))
X_test = X_test.astype('float32') / 255
```

Transformar las etiquetas en categorías

```
# Preparar también las etiquetas en categorías:
from keras.utils import to_categorical
y_train_cat = to_categorical(y_train)
y_test_cat = to_categorical(y_test)
```

USANDO CONVOLUCIÓN

Como datos de entrada se consideran las imágenes de la base MNIST.

Configuramos la nueva red neuronal:

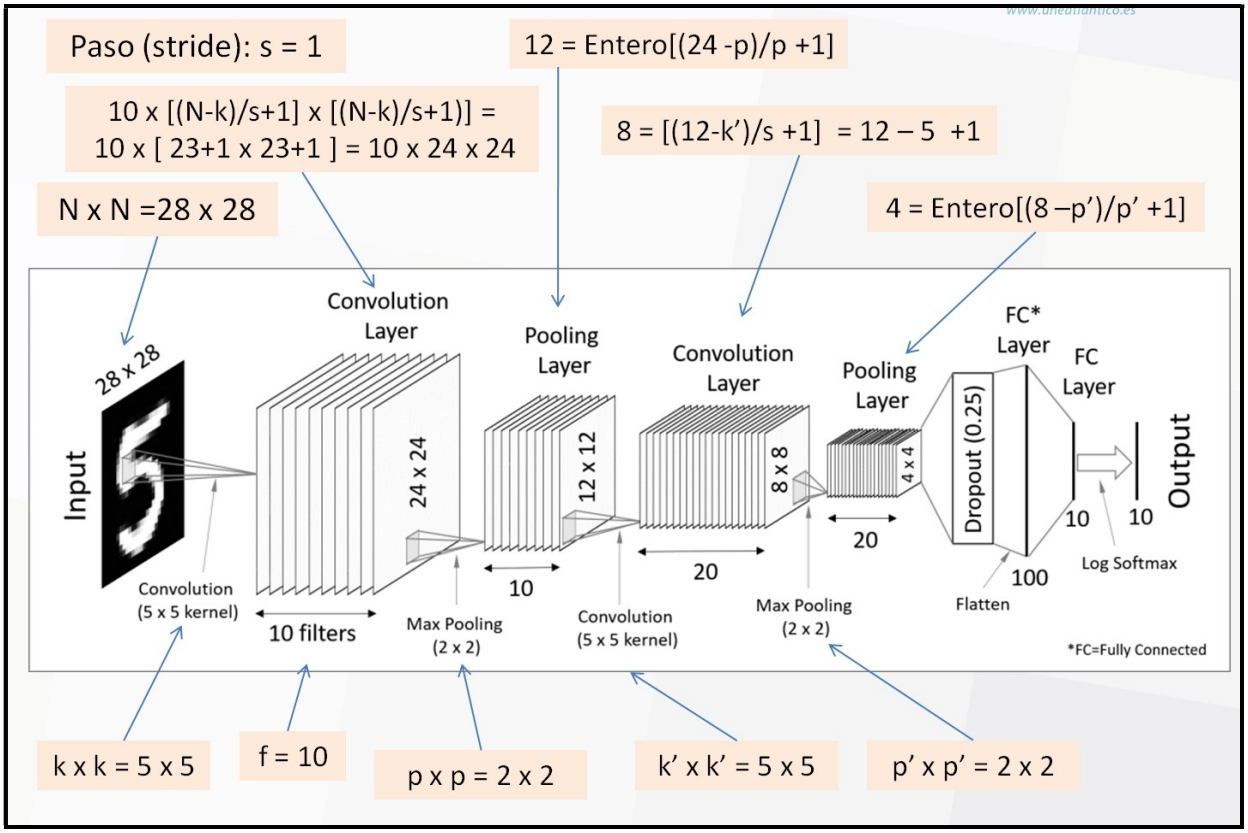
```
from keras import layers
from keras import models

model = models.Sequential()

model.add(layers.Conv2D(10, (5, 5), activation='relu', input_shape=(28, 28, 1)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(20, (5, 5), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
```

La nueva red tiene la siguiente arquitectura:

```
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_3 (Conv2D)            (None, 24, 24, 10)        260
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 12, 12, 10)        0
```

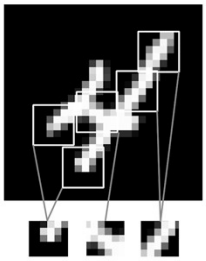
```
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 8, 8, 20)          5020
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 4, 4, 20)          0
=================================================================
Total params: 5,280
Trainable params: 5,280
Non-trainable params: 0
```



Le añadimos los clasificadores definidos con anterioridad:

```
model.add(layers.Flatten())
model.add(layers.Dense(100, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))
```

La diferencia entre usar capas densamente conectadas (Dense layers) y capas basadas en convolución (Convolution layers), es que las primeras buscan patrones en toda la imagen, en todos los pixeles, mientras que utilizando convolución se buscan patrones locales.



Definir la función de pérdida, el optimizador y las métricas para monitorizar el entrenamiento y la prueba de validación.

```
# Definir la función de pérdida, el optimizador y las métricas para monitorizar el entrenamiento
y la prueba de validación
network.compile(optimizer='rmsprop',loss='categorical_crossentropy',metrics=['accuracy'])
# Más información en:
# Optimizer: https://keras.io/optimizers/
# Loss function: https://keras.io/losses/
# Metrics: https://keras.io/metrics/
```

Realizar el entrenamiento y posteriormente la predicción sobre el conjunto de prueba y se comprueba el ajuste o error del modelo. Guardar el resultado en una variable denominada 'history'

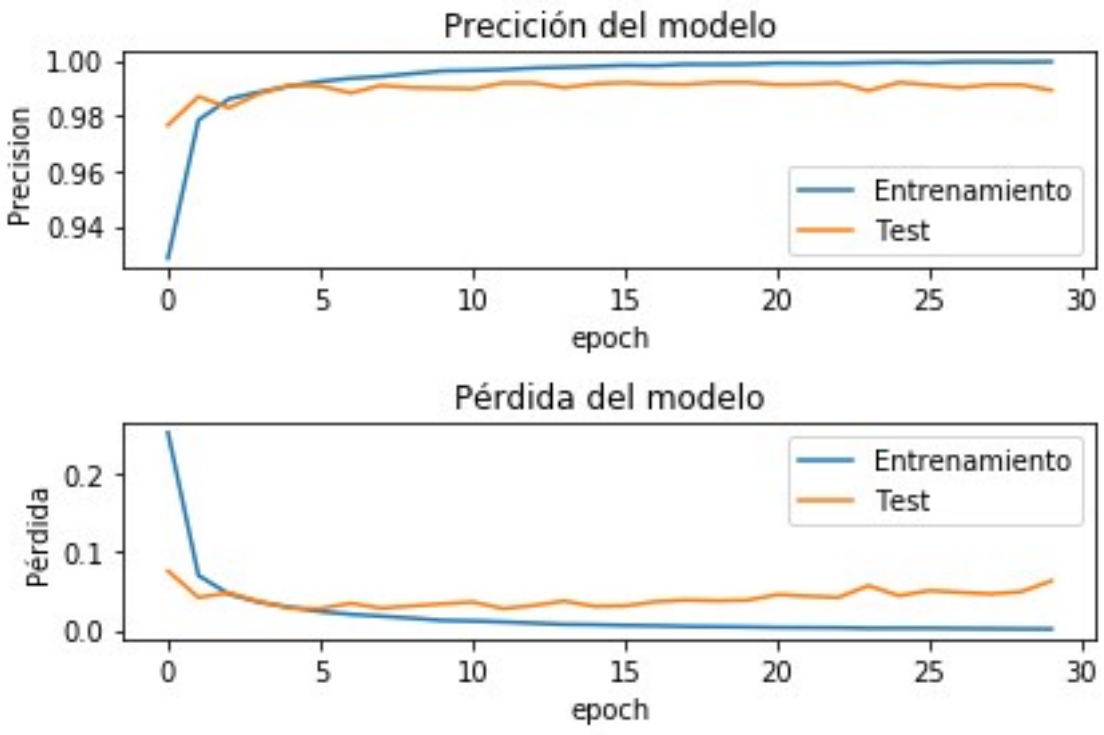
```
# Realizar el entrenamiento. Guardar el resultado en una variable denominada 'history'

history = network.fit(X_train, y_train_cat, epochs=5, batch_size=128, validation_data=(X_test,
y_test_cat))

Train on 60000 samples, validate on 10000 samples

Epoch 1/30
60000/60000 [==============================] - 38s 626us/step - loss: 0.2521 - acc: 0.9284 -
val_loss: 0.0757 - val_acc: 0.9765
Epoch 2/30
60000/60000 [==============================] - 38s 639us/step - loss: 0.0700 - acc: 0.9784 -
val_loss: 0.0419 - val_acc: 0.9869
Epoch 3/30
60000/60000 [==============================] - 39s 655us/step - loss: 0.0462 - acc: 0.9860 -
val_loss: 0.0477 - val_acc: 0.9828
Epoch 4/30
60000/60000 [==============================] - 39s 657us/step - loss: 0.0359 - acc: 0.9885 -
val_loss: 0.0365 - val_acc: 0.9879
Epoch 5/30
60000/60000 [==============================] - 40s 673us/step - loss: 0.0295 - acc: 0.9907 -
val_loss: 0.0282 - val_acc: 0.9910
Epoch 6/30
60000/60000 [==============================] - 40s 666us/step - loss: 0.0239 - acc: 0.9923 -
val_loss: 0.0276 - val_acc: 0.9907
Epoch 7/30
60000/60000 [==============================] - 39s 652us/step - loss: 0.0204 - acc: 0.9935 -
val loss: 0.0343 - val acc: 0.9882
```

```
Epoch 8/30
60000/60000 [==============================] - 39s 650us/step - loss: 0.0180 - acc: 0.9941 -
val_loss: 0.0283 - val_acc: 0.9909
Epoch 9/30
60000/60000 [==============================] - 41s 686us/step - loss: 0.0156 - acc: 0.9952 -
val_loss: 0.0310 - val_acc: 0.9901
Epoch 10/30
60000/60000 [==============================] - 40s 674us/step - loss: 0.0126 - acc: 0.9962 -
val_loss: 0.0337 - val_acc: 0.9899
Epoch 11/30
60000/60000 [==============================] - 40s 658us/step - loss: 0.0119 - acc: 0.9963 -
val_loss: 0.0361 - val_acc: 0.9897
Epoch 12/30
60000/60000 [==============================] - 39s 656us/step - loss: 0.0107 - acc: 0.9966 -
val_loss: 0.0277 - val_acc: 0.9918
Epoch 13/30
60000/60000 [==============================] - 36s 607us/step - loss: 0.0092 - acc: 0.9972 -
val_loss: 0.0316 - val_acc: 0.9918
Epoch 14/30
60000/60000 [==============================] - 38s 639us/step - loss: 0.0076 - acc: 0.9975 -
val_loss: 0.0373 - val_acc: 0.9901



Epoch 15/30
60000/60000 [==============================] - 43s 712us/step - loss: 0.0072 - acc: 0.9978 -
val_loss: 0.0306 - val_acc: 0.9915
Epoch 16/30
60000/60000 [==============================] - 39s 657us/step - loss: 0.0063 - acc: 0.9982 -
val_loss: 0.0311 - val_acc: 0.9919
Epoch 17/30
60000/60000 [==============================] - 39s 656us/step - loss: 0.0059 - acc: 0.9981 -
val_loss: 0.0365 - val_acc: 0.9914
Epoch 18/30
60000/60000 [==============================] - 38s 641us/step - loss: 0.0050 - acc: 0.9986 -
val_loss: 0.0384 - val_acc: 0.9913
Epoch 19/30
60000/60000 [==============================] - 39s 649us/step - loss: 0.0047 - acc: 0.9986 -
val_loss: 0.0376 - val_acc: 0.9920
Epoch 20/30
60000/60000 [==============================] - 38s 639us/step - loss: 0.0043 - acc: 0.9986 -
val_loss: 0.0383 - val_acc: 0.9920
Epoch 21/30
60000/60000 [==============================] - 38s 640us/step - loss: 0.0036 - acc: 0.9990 -
val_loss: 0.0454 - val_acc: 0.9912
Epoch 22/30
60000/60000 [==============================] - 39s 653us/step - loss: 0.0034 - acc: 0.9989 -
val_loss: 0.0436 - val_acc: 0.9913
Epoch 23/30
60000/60000 [==============================] - 38s 638us/step - loss: 0.0033 - acc: 0.9989 -
val_loss: 0.0418 - val_acc: 0.9918
Epoch 24/30
60000/60000 [==============================] - 40s 661us/step - loss: 0.0026 - acc: 0.9991 -
val_loss: 0.0569 - val_acc: 0.9890
Epoch 25/30
60000/60000 [==============================] - 39s 653us/step - loss: 0.0027 - acc: 0.9992 -
val_loss: 0.0441 - val_acc: 0.9920
Epoch 26/30
60000/60000 [==============================] - 40s 671us/step - loss: 0.0026 - acc: 0.9991 -
val_loss: 0.0506 - val_acc: 0.9911
Epoch 27/30
60000/60000 [==============================] - 40s 671us/step - loss: 0.0023 - acc: 0.9994 -
val_loss: 0.0484 - val_acc: 0.9901
Epoch 28/30
```

```
60000/60000 [==============================] - 40s 675us/step - loss: 0.0021 - acc: 0.9995 -
val_loss: 0.0464 - val_acc: 0.9912
Epoch 29/30
60000/60000 [==============================] - 39s 647us/step - loss: 0.0018 - acc: 0.9993 -
val_loss: 0.0491 - val_acc: 0.9911
Epoch 30/30
60000/60000 [==============================] - 39s 652us/step - loss: 0.0017 - acc: 0.9995 -
val_loss: 0.0630 - val_acc: 0.9892
```

Visualizar las métricas

```
# Visualizar las métricas
fig = plt.figure()
plt.subplot(2,1,1)
plt.plot(history.history['acc'])
plt.plot(history.history['val_acc'])
plt.title('Precición del modelo')
plt.ylabel('Precision')
plt.xlabel('epoch')
plt.legend(['Entrenamiento', 'Test'], loc='lower right')

plt.subplot(2,1,2)
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Pérdida del modelo')
plt.ylabel('Pérdida')
plt.xlabel('epoch')
plt.legend(['Entrenamiento', 'Test'], loc='upper right')
plt.tight_layout()
plt.show()
```



Comprobar el error respecto del conjunto de prueba

```
# Comprobar el ajuste o error del modelo respecto del conjunto de prueba
test_loss, test_acc = network.evaluate(X_test, y_test_cat)
print('test_acc:', test_acc)

10000/10000 [==============================] - 3s 341us/step
test_acc: 0.9892
```

Guardar el modelo (en formato JSON) y los pesos del modelo (en formato HDF5)

```
# Guardar el modelo en formato JSON
from keras.models import model_from_json
model_json = network.to_json()
with open("network.json", "w") as json_file:
    json_file.write(model_json)

# Guardar los pesos (weights) a formato HDF5
network.save_weights("network_weights.h5")
print("Guardado el modelo a disco")

Guardado el modelo a disco
Modelo cargado desde el disco
```

Cargar el modelo (en formato JSON) y los pesos del modelo (en formato HDF5)

```
# Leer JSON y crear el modelo
json_file = open("network.json", "r")
loaded_model_json = json_file.read()
json_file.close()
loaded_model = model_from_json(loaded_model_json)

# Cargar los pesos (weights) en un nuevo modelo
loaded_model.load_weights("network_weights.h5")
print("Modelo cargado desde el disco")
```

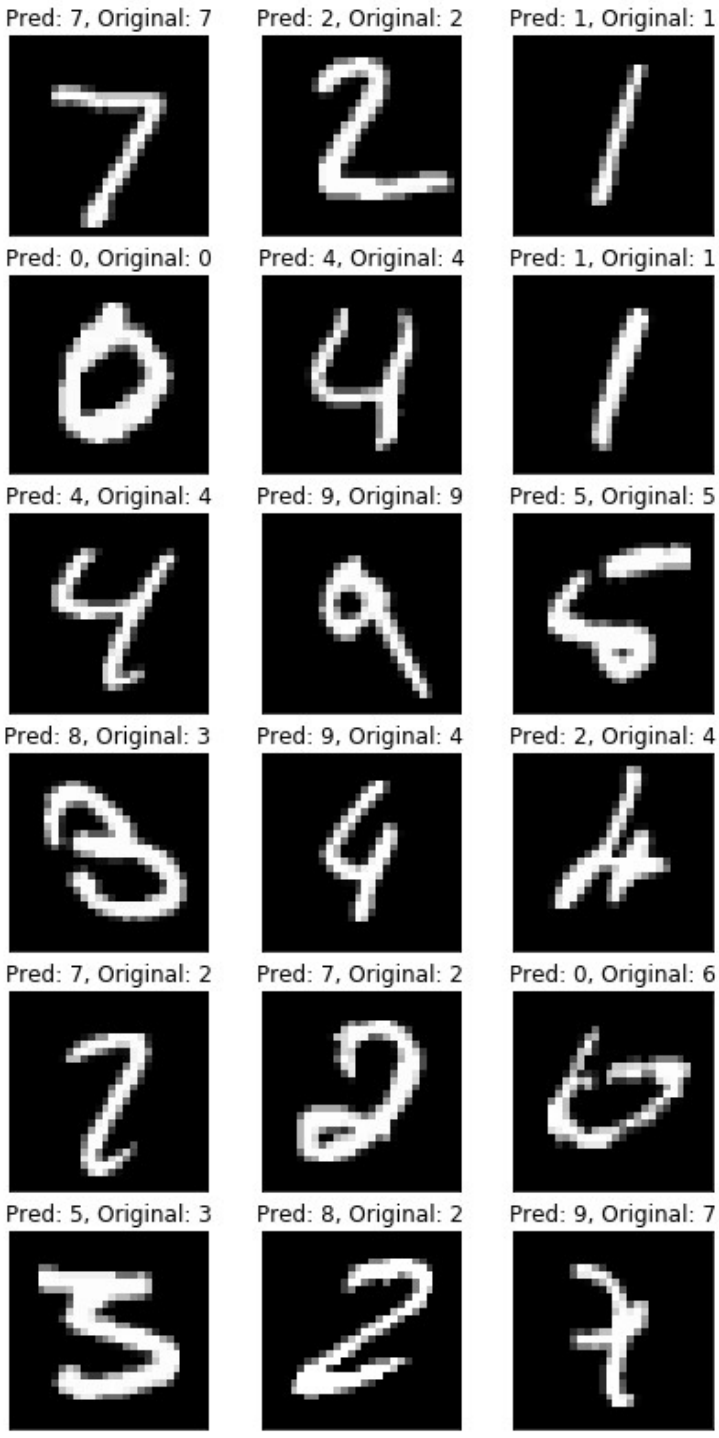
Predecir sobre el conjunto de test

```
# Predecir sobre el conjunto de test
predicted_classes = loaded_model.predict_classes(X_test)
```

Comprobar predicciones correctas y falsas. Visualizar algunas correctas e incorrectas

```
# Comprobar que predicciones son correctas y cuales no
indices_correctos = np.nonzero(predicted_classes == y_test)[0]
indices_incorrectos = np.nonzero(predicted_classes != y_test)[0]
print()
print(len(indices_correctos)," clasificados correctamente")
print(len(indices_incorrectos)," clasificados incorrectamente")

9892  clasificados correctamente
108  clasificados incorrectamente
```

```
# Adaptar el tamaño de la figura para visualizar 18 subplots
plt.rcParams['figure.figsize'] = (7,14)

figure_evaluation = plt.figure()

# Visualizar 9 predicciones correctas
for i, correct in enumerate(indices_correctos[:9]):
    plt.subplot(6,3,i+1)
    plt.imshow(X_test[correct].reshape(28,28), cmap='gray', interpolation='none')
    plt.title(
      "Pred: {}, Original: {}".format(predicted_classes[correct],
                                      y_test[correct]))
    plt.xticks([])
    plt.yticks([])

# Visualizar 9 predicciones incorrectas
for i, incorrect in enumerate(indices_incorrectos[:9]):
    plt.subplot(6,3,i+10)
    plt.imshow(X_test[incorrect].reshape(28,28), cmap='gray', interpolation='none')
    plt.title(
      "Pred: {}, Original: {}".format(predicted_classes[incorrect],
                                      y_test[incorrect]))
    plt.xticks([])
    plt.yticks([])

figure_evaluation
```


Pred: 7, Original: 7
Pred: 2, Original: 2
Pred: 1, Original: 1
Pred: 0, Original: 0
Pred: 4, Original: 4
Pred: 1, Original: 1
Pred: 4, Original: 4
Pred: 9, Original: 9
Pred: 5, Original: 5
Pred: 8, Original: 3
Pred: 9, Original: 4
Pred: 2, Original: 4
Pred: 7, Original: 2
Pred: 7, Original: 2
Pred: 0, Original: 6
Pred: 5, Original: 3
Pred: 8, Original: 2
Pred: 9, Original: 7