Ajuste de una Red Neuronal Convolucional con CIFAR10 en Keras

October 6, 2019

1 CIFAR10 en Keras

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 CIFAR directamente. Se va a utilizar scikit learn para hacer la matriz de confusión y matplotlib para realizar los gráficos.

```
In [1]: from __future__ import print_function
        import keras
        from keras.datasets import cifar10
        from keras.models import Sequential
        from keras.layers import Dense, Dropout, Flatten
        from keras.layers import Conv2D, MaxPooling2D
        import os

        from sklearn import metrics

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

```
Using TensorFlow backend.
```

1.2 Definición de parámetros

Se definen los parámetros para realizar el proceso de entrenamiento.

- `batch_size` es el tamaño del lote que se va a procesar.
- `num_classes` es el número de clases que existen en los datos.
- `epochs` es el número de ciclos de entrenamiento que se hace con todos los datos.
- `class_names` es el nombre de los objetos de cada clase.

```
In [2]: batch_size = 32
        num_classes = 10
        epochs = 50
        class_names = np.array(['airplane', 'automobile', 'bird', 'cat',
                                'deer', 'dog', 'frog', 'horse', 'ship', 'truck'])
```

Hemos creado la función `get_name` para obtener la etiqueta correspondiente a algún dato.

```
In [3]: def get_name(data, index):
            return class_names[data[index]][0]
```

1.3 Obtención de los datos

Se cargan con `cifar10.load_data()` y se dividen es conjunto de entrenamiento y prueba. Recordemos que son 50 mil muestras de entrenamiento y 10 mil muestras de prueba. Cada imagen es de 32×32 pixeles y hay tres canales ya que son a color.

```
In [4]: # División de los datos en datos de entrenamiento y prueba
        (x_train, y_train), (x_test, y_test) = cifar10.load_data()
        print('x_train shape:', x_train.shape)
        print(x_train.shape[0], 'train samples')
        print(x_test.shape[0], 'test samples')
```

```
x_train shape: (50000, 32, 32, 3)
50000 train samples
10000 test samples
```

1.3.1 Rutina para dibujar una imagen

Vamos a definir una rutina para dibujar una imagen del conjunto de datos. 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, title):
            plt.axis('off')
            plt.title(title)
            plt.imshow(data)
            plt.show()
```

```
In [6]: plot_image(x_train[0], get_name(y_train, 0))
```


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

```
Out[7]: array([6], dtype=uint8)
```

Vamos a mostrar las primeras 9 imágenes del conjunto de datos de entrenamiento.

```
In [8]: fig, ax = plt.subplots()
        fig.set_size_inches(6, 6)

        for i in range(9):
            plt.subplot(330 + 1 + i)
            plt.axis('off')
            plt.imshow(x_train[i])
            plt.title(get_name(y_train, i))
        plt.show()
```

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

```
In [9]: y_train = keras.utils.to_categorical(y_train, num_classes)
        y_test = keras.utils.to_categorical(y_test, num_classes)
```

1.4 Creación del modelo

Ahora vamos a crear el modelo con cuatro Capas de Convolución de 32, 32, 64 y 64 mapas de características respectivamente con kernel de 3×3 y ReLU como función de activación. Primero se aplican dos Capas de Convolución y se aplica una Capa de Sub muestraje con una ventana de 2×2 y finalmente una red reuronal conectada con 512 neuronas.

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

```
In [10]: model = Sequential()
         model.add(Conv2D(32, kernel_size=(3, 3),
                          padding='same',
```

```
                 activation='relu',
                 input_shape=x_train.shape[1:]))
model.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

model.add(Conv2D(64, kernel_size=(3, 3),
                 padding='same',
                 activation='relu'))
model.add(Conv2D(64, kernel_size=(3, 3),
                 activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

model.add(Flatten())
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))

opt = keras.optimizers.RMSprop(lr=0.0001, decay=1e-6)

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

Ahora podemos corroborar el modelo.

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

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 32, 32, 32)        896
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 30, 30, 32)        9248
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 15, 15, 32)        0
_________________________________________________________________
dropout_1 (Dropout)          (None, 15, 15, 32)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 15, 15, 64)        18496
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 13, 13, 64)        36928
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 6, 6, 64)          0
_________________________________________________________________
dropout_2 (Dropout)          (None, 6, 6, 64)          0
```

```
_________________________________________________________________
flatten_1 (Flatten)          (None, 2304)              0
_________________________________________________________________
dense_1 (Dense)              (None, 512)               1180160
_________________________________________________________________
dropout_3 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_2 (Dense)              (None, 10)                5130
=================================================================
Total params: 1,250,858
Trainable params: 1,250,858
Non-trainable params: 0
_________________________________________________________________
```

Antes de pasar los datos al modelo se normalizan los valores.

```
In [12]: x_train = x_train.astype('float32')
         x_test = x_test.astype('float32')
         x_train /= 255
         x_test /= 255
```

1.5 Ajustar el modelo

Ahora se realiza el ajuste de modelo de acuerdo a los parámetros definidos previamente. En particular se van a realizar 10 pasadas a todos los datos para realizar el entrenamiento.

```
In [16]: model.fit(x_train, y_train,
                   batch_size=batch_size,
                   epochs=epochs,
                   validation_data=(x_test, y_test),
                   shuffle=True)

Train on 50000 samples, validate on 10000 samples
Epoch 1/50
50000/50000 [==============================] - 199s 4ms/step - loss: 0.6370 - acc: 0.7866 - val
Epoch 2/50
50000/50000 [==============================] - 204s 4ms/step - loss: 0.6374 - acc: 0.7876 - val
Epoch 3/50
50000/50000 [==============================] - 200s 4ms/step - loss: 0.6361 - acc: 0.7885 - val
Epoch 4/50
50000/50000 [==============================] - 196s 4ms/step - loss: 0.6365 - acc: 0.7888 - val
Epoch 5/50
50000/50000 [==============================] - 202s 4ms/step - loss: 0.6314 - acc: 0.7878 - val
Epoch 6/50
50000/50000 [==============================] - 200s 4ms/step - loss: 0.6339 - acc: 0.7892 - val
Epoch 7/50
50000/50000 [==============================] - 201s 4ms/step - loss: 0.6359 - acc: 0.7890 - val
Epoch 8/50
```

```
50000/50000 [==============================] - 196s 4ms/step - loss: 0.6290 - acc: 0.7910 - val
Epoch 9/50
50000/50000 [==============================] - 204s 4ms/step - loss: 0.6313 - acc: 0.7891 - val
Epoch 10/50
50000/50000 [==============================] - 202s 4ms/step - loss: 0.6326 - acc: 0.7900 - val
Epoch 11/50
50000/50000 [==============================] - 196s 4ms/step - loss: 0.6286 - acc: 0.7905 - val
Epoch 12/50
50000/50000 [==============================] - 195s 4ms/step - loss: 0.6328 - acc: 0.7896 - val
Epoch 13/50
50000/50000 [==============================] - 203s 4ms/step - loss: 0.6264 - acc: 0.7911 - val
Epoch 14/50
50000/50000 [==============================] - 200s 4ms/step - loss: 0.6258 - acc: 0.7908 - val
Epoch 15/50
50000/50000 [==============================] - 198s 4ms/step - loss: 0.6308 - acc: 0.7918 - val
Epoch 16/50
50000/50000 [==============================] - 205s 4ms/step - loss: 0.6268 - acc: 0.7903 - val
Epoch 17/50
50000/50000 [==============================] - 196s 4ms/step - loss: 0.6252 - acc: 0.7936 - val
Epoch 18/50
50000/50000 [==============================] - 194s 4ms/step - loss: 0.6304 - acc: 0.7907 - val
Epoch 19/50
50000/50000 [==============================] - 198s 4ms/step - loss: 0.6293 - acc: 0.7910 - val
Epoch 20/50
50000/50000 [==============================] - 222s 4ms/step - loss: 0.6263 - acc: 0.7940 - val
Epoch 21/50
50000/50000 [==============================] - 203s 4ms/step - loss: 0.6279 - acc: 0.7922 - val
Epoch 22/50
50000/50000 [==============================] - 203s 4ms/step - loss: 0.6344 - acc: 0.7907 - val
Epoch 23/50
50000/50000 [==============================] - 201s 4ms/step - loss: 0.6333 - acc: 0.7894 - val
Epoch 24/50
50000/50000 [==============================] - 204s 4ms/step - loss: 0.6320 - acc: 0.7924 - val
Epoch 25/50
50000/50000 [==============================] - 202s 4ms/step - loss: 0.6322 - acc: 0.7931 - val
Epoch 26/50
50000/50000 [==============================] - 193s 4ms/step - loss: 0.6253 - acc: 0.7918 - val
Epoch 27/50
50000/50000 [==============================] - 200s 4ms/step - loss: 0.6312 - acc: 0.7898 - val
Epoch 28/50
50000/50000 [==============================] - 196s 4ms/step - loss: 0.6289 - acc: 0.7904 - val
Epoch 29/50
50000/50000 [==============================] - 202s 4ms/step - loss: 0.6348 - acc: 0.7891 - val
Epoch 30/50
50000/50000 [==============================] - 197s 4ms/step - loss: 0.6267 - acc: 0.7920 - val
Epoch 31/50
50000/50000 [==============================] - 196s 4ms/step - loss: 0.6337 - acc: 0.7914 - val
Epoch 32/50
```

```
50000/50000 [==============================] - 200s 4ms/step - loss: 0.6353 - acc: 0.7900 - val
Epoch 33/50
50000/50000 [==============================] - 200s 4ms/step - loss: 0.6340 - acc: 0.7893 - val
Epoch 34/50
50000/50000 [==============================] - 192s 4ms/step - loss: 0.6413 - acc: 0.7898 - val
Epoch 35/50
50000/50000 [==============================] - 194s 4ms/step - loss: 0.6380 - acc: 0.7904 - val
Epoch 36/50
50000/50000 [==============================] - 199s 4ms/step - loss: 0.6344 - acc: 0.7899 - val
Epoch 37/50
50000/50000 [==============================] - 199s 4ms/step - loss: 0.6357 - acc: 0.7910 - val
Epoch 38/50
50000/50000 [==============================] - 196s 4ms/step - loss: 0.6409 - acc: 0.7898 - val
Epoch 39/50
50000/50000 [==============================] - 199s 4ms/step - loss: 0.6418 - acc: 0.7872 - val
Epoch 40/50
50000/50000 [==============================] - 197s 4ms/step - loss: 0.6382 - acc: 0.7907 - val
Epoch 41/50
50000/50000 [==============================] - 201s 4ms/step - loss: 0.6419 - acc: 0.7909 - val
Epoch 42/50
50000/50000 [==============================] - 201s 4ms/step - loss: 0.6470 - acc: 0.7870 - val
Epoch 43/50
50000/50000 [==============================] - 197s 4ms/step - loss: 0.6428 - acc: 0.7876 - val
Epoch 44/50
50000/50000 [==============================] - 193s 4ms/step - loss: 0.6452 - acc: 0.7875 - val
Epoch 45/50
50000/50000 [==============================] - 511s 10ms/step - loss: 0.6403 - acc: 0.7892 - va
Epoch 46/50
50000/50000 [==============================] - 412s 8ms/step - loss: 0.6455 - acc: 0.7875 - val
Epoch 47/50
50000/50000 [==============================] - 199s 4ms/step - loss: 0.6485 - acc: 0.7877 - val
Epoch 48/50
50000/50000 [==============================] - 200s 4ms/step - loss: 0.6387 - acc: 0.7892 - val
Epoch 49/50
50000/50000 [==============================] - 196s 4ms/step - loss: 0.6436 - acc: 0.7880 - val
Epoch 50/50
50000/50000 [==============================] - 193s 4ms/step - loss: 0.6480 - acc: 0.7875 - val
```

```
Out[16]: <keras.callbacks.History at 0x1a2dfb3f90>
```

1.6 Prueba del modelo

Luego del ajuste ahora se prueba el modelo con los datos de prueba.

```
In [17]: scores = model.evaluate(x_test, y_test, verbose=1)
         print('Test loss:', scores[0])
         print('Test accuracy:', scores[1])
```

```
10000/10000 [==============================] - 8s 766us/step
Test loss: 0.6841973246574402
Test accuracy: 0.7784
```

Siempre es importante salvar el modelo para trabajar posteriormente y evitar realizar procesos de ajuste de nuevo. De esta forma se ahorra tiempo.

```
In [18]: model.save('keras_cifar10_trained_model-100.h5')
```

Se puede cargar el modelo salvado y se pueden corroborar las características de la arquitectura.

```
In [13]: from keras.models import load_model
         # Cargar el modelo del archivo salvado.
         # Hay un modelo entrenado con 10 iteraciones keras_cifar10_trained_model.h5
         # otro con 50 iteraciones keras_cifar10_trained_model-50.h5
         # y uno con 100 iteraciones keras_cifar10_trained_model-100.h5
         model = load_model('keras_cifar10_trained_model-50.h5')
         model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 32, 32, 32)        896
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 30, 30, 32)        9248
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 15, 15, 32)        0
_________________________________________________________________
dropout_1 (Dropout)          (None, 15, 15, 32)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 15, 15, 64)        18496
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 13, 13, 64)        36928
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 6, 6, 64)          0
_________________________________________________________________
dropout_2 (Dropout)          (None, 6, 6, 64)          0
_________________________________________________________________
flatten_1 (Flatten)          (None, 2304)              0
_________________________________________________________________
dense_1 (Dense)              (None, 512)               1180160
_________________________________________________________________
dropout_3 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_2 (Dense)              (None, 10)                5130
=================================================================
Total params: 1,250,858
Trainable params: 1,250,858
```

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

Podemos corroborar de nuevo los valores

```
In [14]: # Evaluación del modelo
         scores = model.evaluate(x_test, y_test, verbose=1)
         print('Test loss:', scores[0])
         print('Test accuracy:', scores[1])

10000/10000 [==============================] - 7s 674us/step
Test loss: 0.66253902053833
Test accuracy: 0.78


In [15]: from keras.models import load_model
         # Cargar el modelo del archivo salvado.
         # Hay un modelo entrenado con 10 iteraciones keras_cifar10_trained_model.h5
         # otro con 50 iteraciones keras_cifar10_trained_model-50.h5
         # y uno con 100 iteraciones keras_cifar10_trained_model-100.h5
         model = load_model('keras_cifar10_trained_model-100.h5')
         model.summary()

_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 32, 32, 32)        896
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 30, 30, 32)        9248
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 15, 15, 32)        0
_________________________________________________________________
dropout_1 (Dropout)          (None, 15, 15, 32)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 15, 15, 64)        18496
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 13, 13, 64)        36928
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 6, 6, 64)          0
_________________________________________________________________
dropout_2 (Dropout)          (None, 6, 6, 64)          0
_________________________________________________________________
flatten_1 (Flatten)          (None, 2304)              0
_________________________________________________________________
dense_1 (Dense)              (None, 512)               1180160
_________________________________________________________________
dropout_3 (Dropout)          (None, 512)               0
_________________________________________________________________
```

```
dense_2 (Dense)              (None, 10)                5130
=================================================================
Total params: 1,250,858
Trainable params: 1,250,858
Non-trainable params: 0
_________________________________________________________________
```

```
In [16]: # Evaluación del modelo
         scores = model.evaluate(x_test, y_test, verbose=1)
         print('Test loss:', scores[0])
         print('Test accuracy:', scores[1])
```

```
10000/10000 [==============================] - 7s 677us/step
Test loss: 0.6841973246574402
Test accuracy: 0.7784
```

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

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

1.7 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 [18]: matrix = metrics.confusion_matrix(y_test.argmax(axis=1), y_pred.argmax(axis=1))
         matrix
```

```
Out[18]: array([[818,   9,  31,  14,  16,   2,  15,   6,  41,  48],
                [ 14, 866,   2,   3,   2,   2,   7,   1,  12,  91],
                [ 63,   4, 646,  30,  87,  41, 107,  14,   2,   6],
                [ 33,   2,  62, 530,  75, 113, 139,  10,  16,  20],
                [ 12,   0,  32,  24, 813,   8,  86,  16,   5,   4],
                [ 12,   1,  47, 135,  62, 662,  46,  28,   2,   5],
                [  7,   1,  22,  18,  21,   5, 923,   0,   2,   1],
                [ 12,   1,  35,  37,  87,  53,  19, 734,   1,  21],
                [ 61,  20,   4,   7,   3,   2,  13,   0, 866,  24],
                [ 12,  31,   2,   4,   2,   1,   9,   2,  11, 926]])
```

La matriz se puede generar normalizada.

```
In [19]: matrix.astype('float') / matrix.sum(axis=1)[:, np.newaxis]
```

```
Out[19]: array([[0.818, 0.009, 0.031, 0.014, 0.016, 0.002, 0.015, 0.006, 0.041,
                 0.048],
                [0.014, 0.866, 0.002, 0.003, 0.002, 0.002, 0.007, 0.001, 0.012,
                 0.091],
```

```
       [0.063, 0.004, 0.646, 0.03 , 0.087, 0.041, 0.107, 0.014, 0.002,
        0.006],
       [0.033, 0.002, 0.062, 0.53 , 0.075, 0.113, 0.139, 0.01 , 0.016,
        0.02 ],
       [0.012, 0.   , 0.032, 0.024, 0.813, 0.008, 0.086, 0.016, 0.005,
        0.004],
       [0.012, 0.001, 0.047, 0.135, 0.062, 0.662, 0.046, 0.028, 0.002,
        0.005],
       [0.007, 0.001, 0.022, 0.018, 0.021, 0.005, 0.923, 0.   , 0.002,
        0.001],
       [0.012, 0.001, 0.035, 0.037, 0.087, 0.053, 0.019, 0.734, 0.001,
        0.021],
       [0.061, 0.02 , 0.004, 0.007, 0.003, 0.002, 0.013, 0.   , 0.866,
        0.024],
       [0.012, 0.031, 0.002, 0.004, 0.002, 0.001, 0.009, 0.002, 0.011,
        0.926]])
```

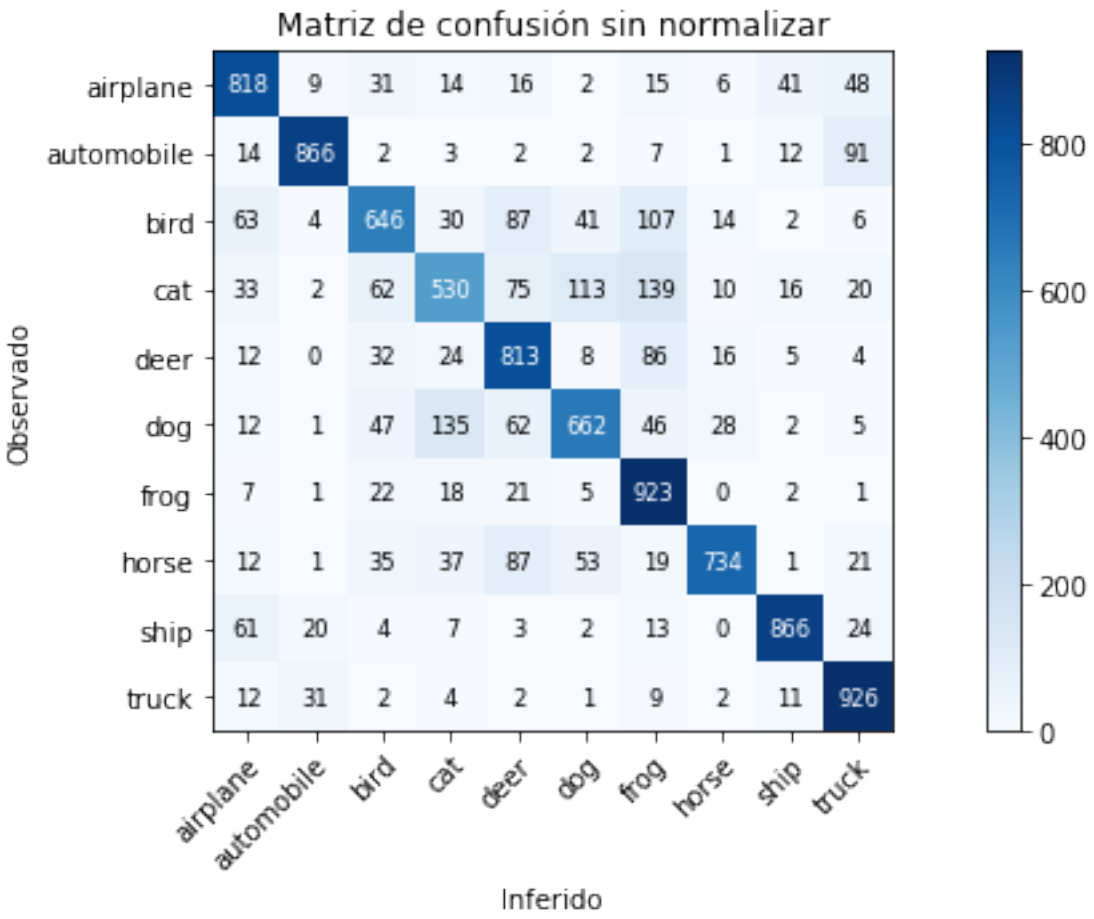
Hemos modificado el ejemplo para mostrar la matriz de confusión del ajuste con los datos CIFAR.

```
In [20]: from sklearn.model_selection import train_test_split
         from sklearn.metrics import confusion_matrix
         from sklearn.utils.multiclass import unique_labels

         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')

    # Rotate the tick labels and set their alignment.
    plt.setp(ax.get_xticklabels(), rotation=45, ha="right",
             rotation_mode="anchor")

    # 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
```

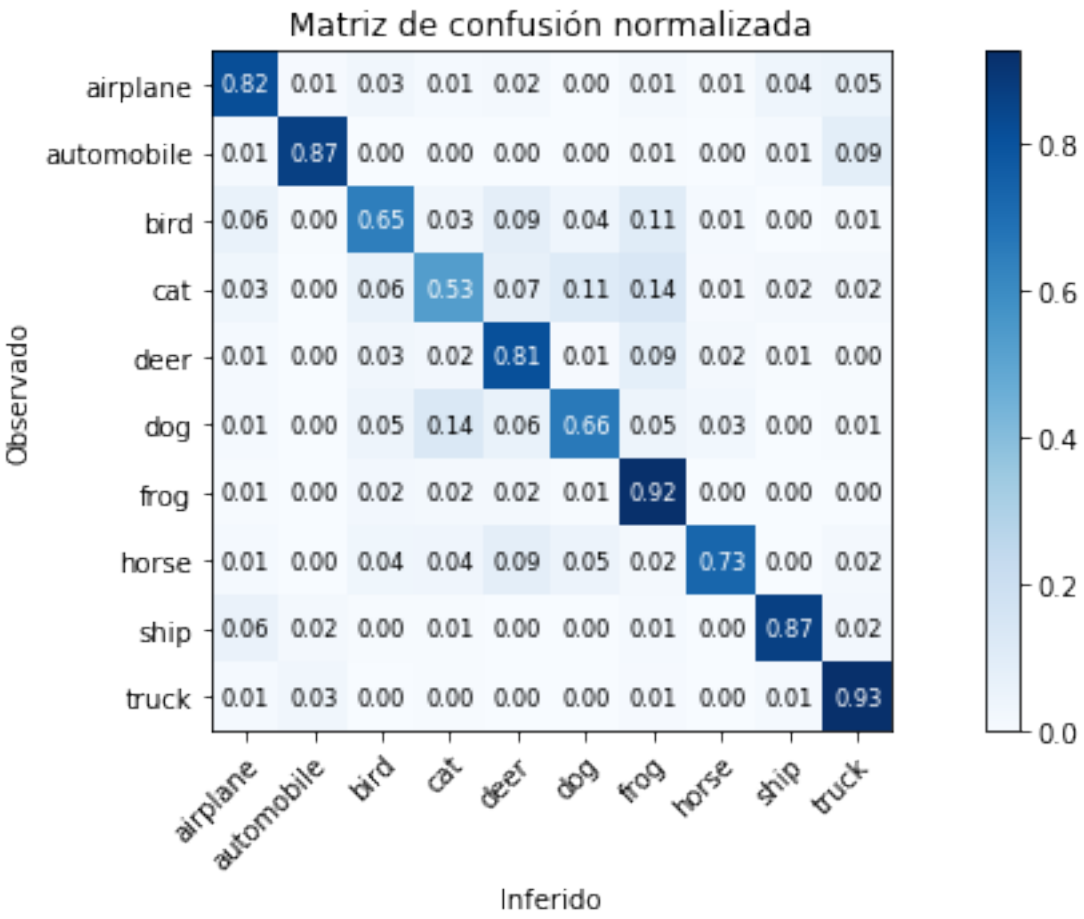
Ahora utilizamos la rutina recien creada para visualizar en una imagen la matriz de confusión.

```
In [21]: np.set_printoptions(precision=2)
         plot_confusion_matrix(y_test, y_pred, classes=class_names,
                               title=u"Matriz de confusi\u00f3n sin normalizar")
         plt.show()
```



Cambiando el parámetro `normalize` a `True` podemos ver la imagen con valores normalizados.

```
In [22]: np.set_printoptions(precision=0)
         plot_confusion_matrix(y_test, y_pred, classes=class_names, normalize=True,
                               title=u"Matriz de confusi\u00f3n normalizada")

         plt.show()
```


Matriz de confusión normalizada
airplane 0.82 0.01 0.03 0.01 0.02 0.00 0.01 0.01 0.04 0.05
automobile 0.01 0.87 0.00 0.00 0.00 0.00 0.01 0.00 0.01 0.09
bird 0.06 0.00 0.65 0.03 0.09 0.04 0.11 0.01 0.00 0.01
cat 0.03 0.00 0.06 0.53 0.07 0.11 0.14 0.01 0.02 0.02
deer 0.01 0.00 0.03 0.02 0.81 0.01 0.09 0.02 0.01 0.00
dog 0.01 0.00 0.05 0.14 0.06 0.66 0.05 0.03 0.00 0.01
frog 0.01 0.00 0.02 0.02 0.02 0.01 0.92 0.00 0.00 0.00
horse 0.01 0.00 0.04 0.04 0.09 0.05 0.02 0.73 0.00 0.02
ship 0.06 0.02 0.00 0.01 0.00 0.00 0.01 0.00 0.87 0.02
truck 0.01 0.03 0.00 0.00 0.00 0.00 0.01 0.00 0.01 0.93
Observado
airplane
automobile
bird
cat
deer
dog
frog
horse
ship
truck
Inferido
0.8
0.6
0.4
0.2
0.0