

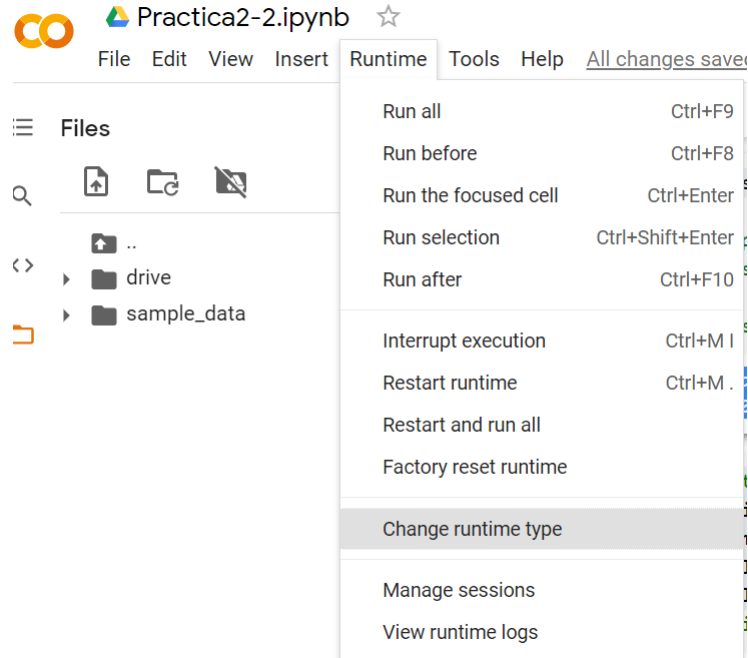
Redes convolucionales.
Uso del script en Colaboratory

Introducción

- Colaboratory es un entorno gratuito de Jupyter Notebook que no requiere configuración y que se ejecuta completamente en la nube.
- Es un Jupyter Notebook (JN) almacenado en Google Drive. Estos cuadernos están compuestos de celdas que pueden contener código, texto e imágenes entre otros elementos. Podremos ejecutar código Python sin necesitar una instalación específica en nuestro ordenador.
- Podemos utilizar un JN y guardar una copia en Drive. (Archivo/guardar una copia en Drive).
- Para poder utilizar Colaboratory es necesario tener abierta una cuenta de Google (la cuenta del correo de la Universidad bastaría).
- <https://colab.research.google.com/notebooks/welcome.ipynb>

Para acelerar la ejecución

- Activar la ejecución en TPU o GPU



Importar librerías

- Se trabajará con Keras que es una librería de alto nivel, escrita en Python, que utiliza la librería de más bajo nivel TensorFlow.
- Fue desarrollada para poder hacer experimentación de una forma muy rápida (prototipado).
- Importaremos TensorFlow, ciertos módulos de Keras , librerías de Python para hacer plots y ciertos módulos de la librería de aprendizaje automático scikit-learn que nos permitirá obtener las matrices de confusión y otros informes de una manera muy sencilla

```
import tensorflow as tf
import numpy as np

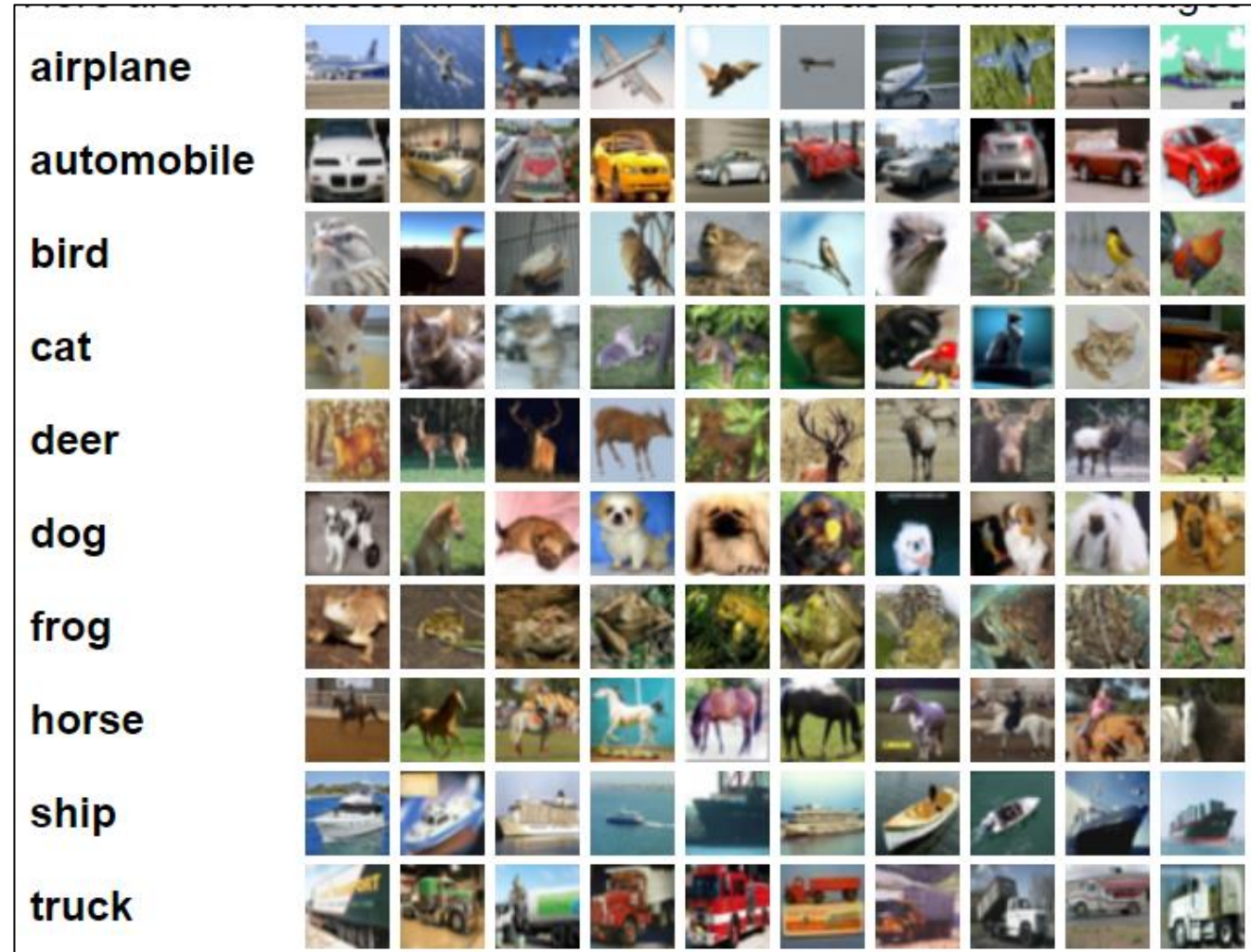
from tensorflow.keras import datasets, layers, models
import matplotlib.pyplot as plt

#Confusion Matrix and classification report
from sklearn.metrics import confusion_matrix, classification_report
```

Conjunto de datos a utilizar

- Utilizaremos el conjunto de datos CIFAR10

- 60000 imágenes en color
- 10 clases
- 6000 imágenes por clase
- Entrenamiento: 50000 imágenes
- Test: 10000 imágenes
- Cada imagen: 32 x 32 pixels y 3 canales (RGB)



Descargar los datos

- Obtendremos las imágenes de entrenamiento (`train_images`) y las de test (`test_images`) así como las salidas, es decir, las etiquetas correspondientes (`train_labels` y `test_labels`)

```
(train_images, train_labels), (test_images, test_labels) = datasets.cifar10.load_data()
```

```
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz  
170500096/170498071 [=====] - 2s 0us/step
```

Ver la estructura de los datos

```
#print the structure of data
print(train_images.shape)
print(test_images.shape)
print(train_labels.shape)
print(train_labels[1])
```

```
↳ (50000, 32, 32, 3)
   (10000, 32, 32, 3)
   (50000, 1)
   [9]
```

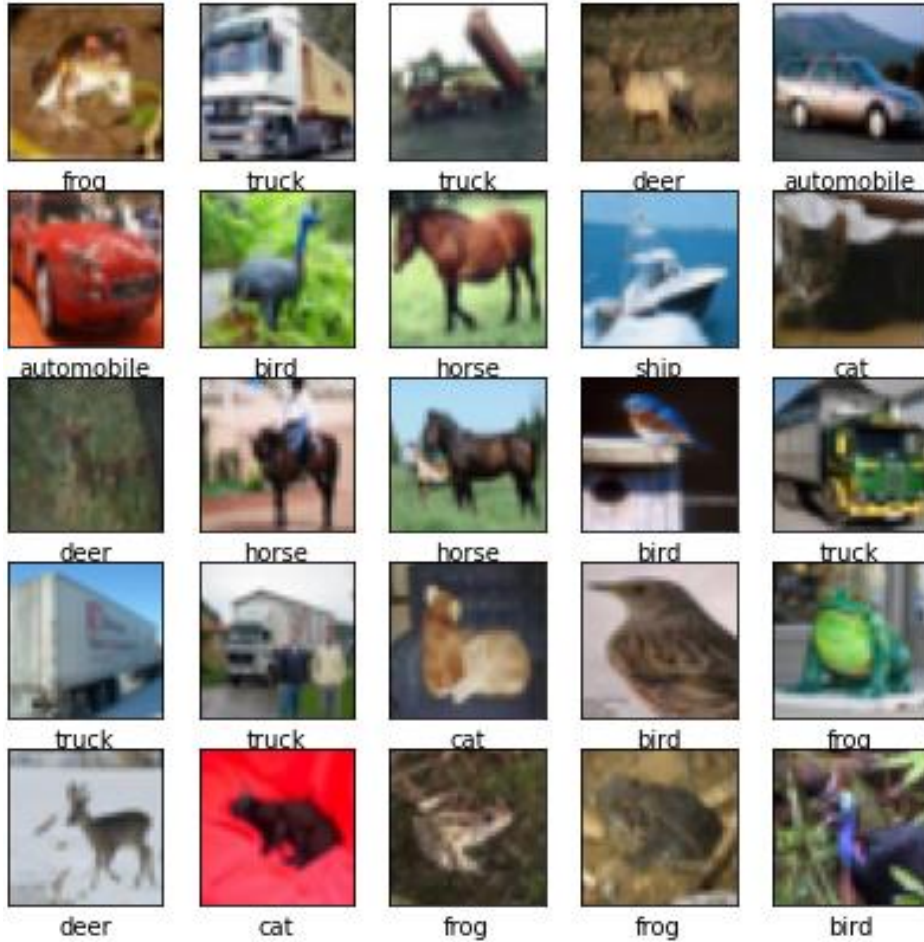
- Las imágenes son arrays de cuatro dimensiones (50000,32,32,3)
 - número de imágenes, anchura en pixels, altura en pixels, número de canales
- Las etiquetas se representan por un valor entero entre 0 y 9 (10 clases)

Ver algunas imágenes

- Representaremos las primeras 25 imágenes de entrenamiento

```
class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer',  
               'dog', 'frog', 'horse', 'ship', 'truck']  
  
plt.figure(figsize=(7,7))  
for i in range(25):  
    plt.subplot(5,5,i+1)  
    plt.xticks([])  
    plt.yticks([])  
    plt.grid(False)  
    plt.imshow(train_images[i], cmap=plt.cm.binary)  
    # The CIFAR labels happen to be arrays,  
    # which is why you need the extra index  
    # cada etiqueta es un array en este caso de un elemento, pero podrían ser más  
    plt.xlabel(class_names[train_labels[i][0]])  
plt.show()
```


Ver algunas imágenes



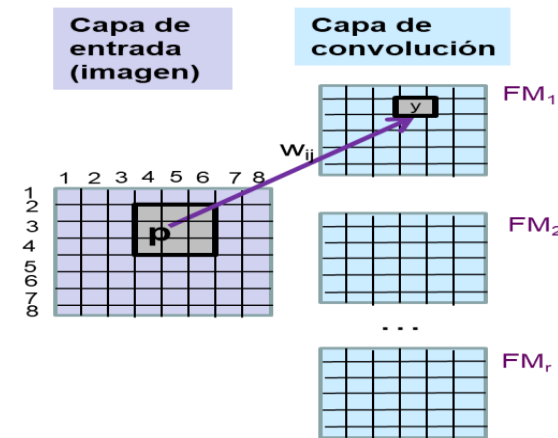
Definir los modelos de redes de neuronas. PM

- En esta sección definimos cada modelo utilizando Keras, mediante una función en Python. Podemos definir tantas funciones como queramos y luego utilizaremos la que nos convenga en cada caso.
- Usaremos un modelo secuencial y vamos añadiendo capas
- Ejemplo de PM muy básico
 - Aplicamos una normalización a la entrada (los píxeles podrían tener valores entre 0 y 255)
 - La capa de entrada tendrá 32 x 32 x 3 neuronas. Mediante *Flatten*, la estructura matricial se aplanar en un vector de entradas
 - Añadimos una capa oculta de 50 neuronas con función de activación sigmoideal (podremos usar otras funciones como 'tanh', 'relu', etc.
 - La última capa tendrá 10 neuronas (una por clase) con función de activación 'softmax'. Es una función similar a la sigmoide pero normalizada de forma que todas las salidas suman 1. Así se podrá corresponder cada valor con una probabilidad.

```
def create_simple_pm():  
    #modelo simple de pm:  
    model = tf.keras.models.Sequential()  
    model.add(tf.keras.layers.BatchNormalization(input_shape=(32, 32, 3)))  
    model.add(tf.keras.layers.Flatten(input_shape=(32, 32, 3), name="Input_layer"))  
    model.add(tf.keras.layers.Dense(50, activation='sigmoid', name="Hidden_layer"))  
    model.add(tf.keras.layers.Dense(10, activation='softmax', name="Output_layer"))  
    return model
```

Definir CNN. Capa convolucional

- Las CNN tendrán capas convolucionales, capas de 'pooling' y capas 'fully connected'
- Capa de convolución: para imágenes se utiliza la convolución 2D. En Keras utilizamos la función conv2D
 - (<https://keras.io/layers/convolutional/>)
- Argumentos más importantes de conv2D
 - **filters**: número de filtros
 - **kernel_size**: tamaño de los filtros. Ej: (3,3), (5,5), etc
 - **activation**: función de activación
 - **strides**: los saltos que da el filtro. Por defecto vale 1.
 - **Padding**: se utiliza para mantener la dimensión de la imagen que recibe la capa de convolución, añadiendo pixels a 0 alrededor de la imagen. Puede ser 'valid' o 'same'. Valor por defecto 'valid', que equivale a no utilizar padding. Si es 'same' se añaden pixels con valor 0 alrededor de la imagen para que la anchura y altura de la salida sean las mismas.



Ejemplo: `model.add(layers.Conv2D(16, (3, 3), activation='relu', padding='same'))`

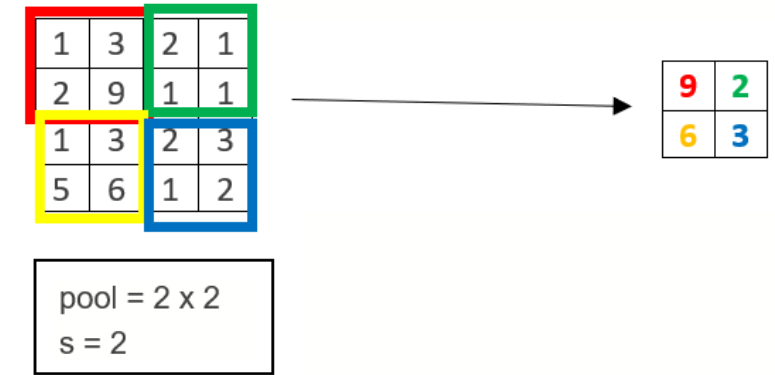
núm de filtros

tam de
los filtros

Definir CNN. Capa de 'pooling'

Capa de Pooling

- Sirve para reducir el tamaño de la representación. Ej, con maxPooling:
- Las capas de pooling no tienen parámetros (pesos) solo tienen hiperparámetros
- Las capas de pooling siempre mantienen el número de canales
Se aplican a cada canal por separado
- Por defecto, el valor de strides coincide con pool_size, y no se solapan las ventanas.
- Esta capa reduce a la mitad la anchura y altura del input.
- Ejemplo: `model.add(layers.MaxPooling2D((2, 2)))`
por defecto, `strides = (2,2)`



Definir CNN. Capa 'fully connected'

- Son capas de un perceptron multicapa convencional, donde todas las neuronas de una capa se conectan con todas las de la capa siguiente.

Funciones de Keras:

- **Flatten**: realiza el aplanado de las neuronas de la capa anterior. Equivale a la capa de entrada de un PM.

Ejemplo: `model.add(layers.Flatten())`

- **Dense**: Añade una capa con un determinado número de neuronas

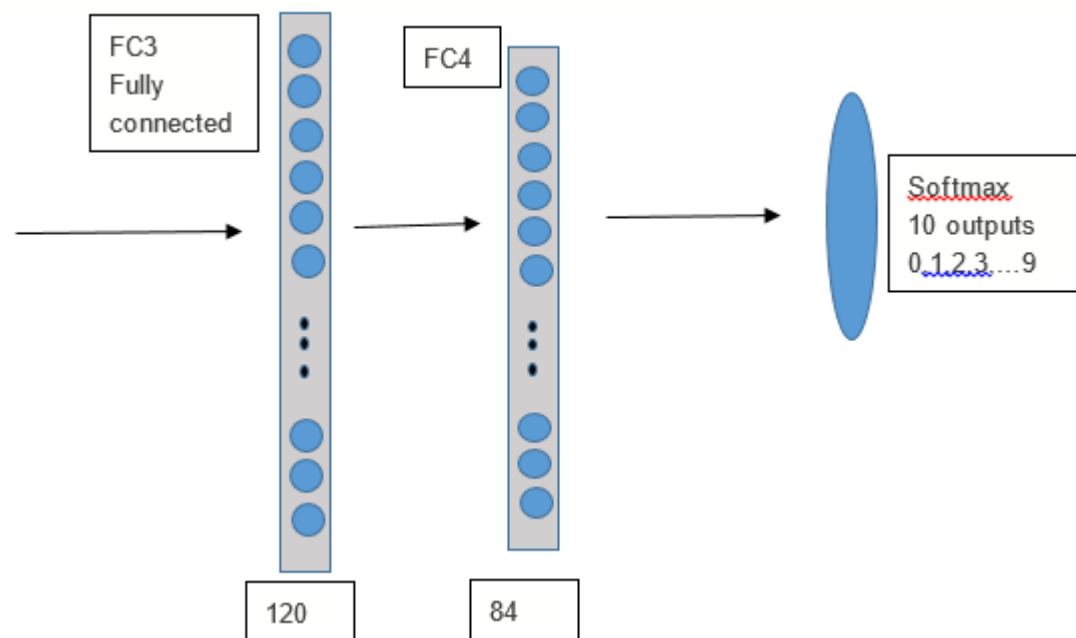
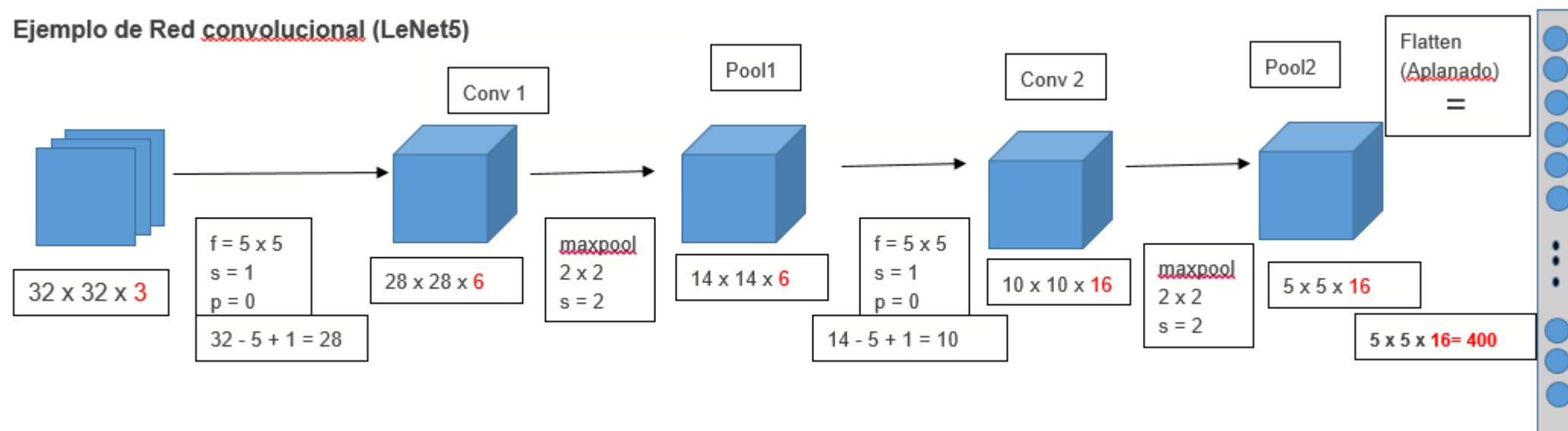
Ejemplo: Capa con 32 neuronas y función de activación 'relu'

```
model.add(layers.Dense(32, activation='relu'))
```

- Habría que añadir la última capa con 10 neuronas (una por clase) y con función de activación 'softmax'

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

Ejemplo de Red convolucional (LeNet5)

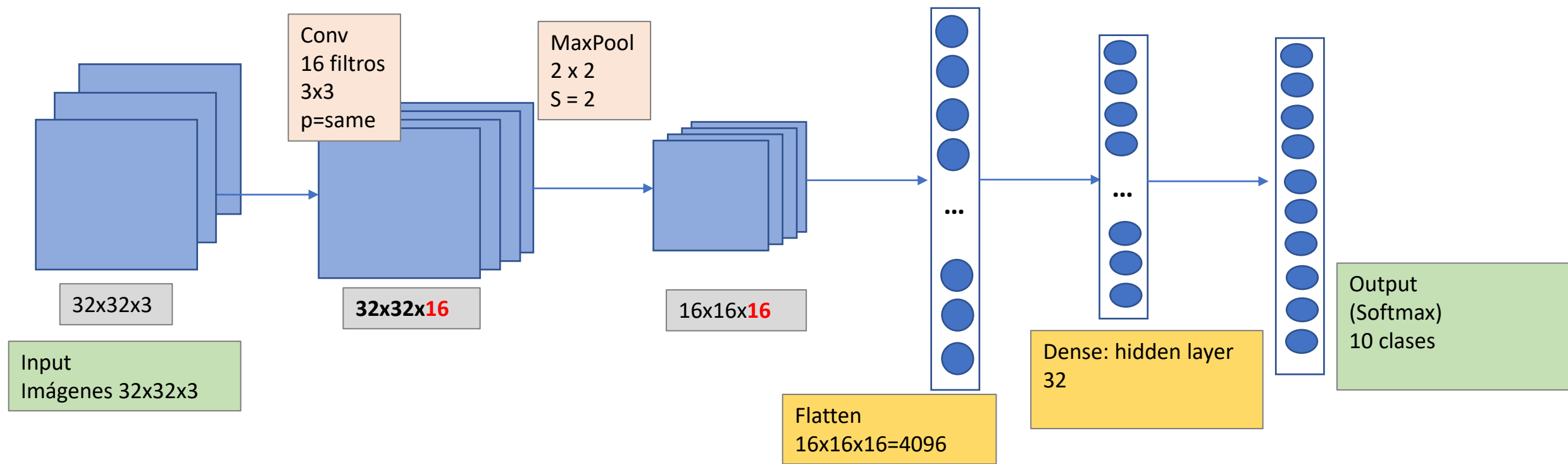


Definir CNN

- Ejemplo de una red CNN muy simple: normalización, capa convolucional con 16 filtros de 3 x 3. La FullyConnected con 32 neuronas ocultas y 10 neuronas de salida

```
def create_model_simple_cnn():  
    model = models.Sequential()  
    model.add(layers.BatchNormalization(input_shape=(32, 32, 3)))  
    model.add(layers.Conv2D(16, (3, 3), activation='relu',padding='same'))  
    model.add(layers.MaxPooling2D((2, 2)))  
  
    model.add(layers.Flatten())  
    model.add(layers.Dense(32, activation='relu'))  
    model.add(layers.Dense(10, activation='softmax'))  
    return model
```

Red CNN muy básica utilizada en el script



Incluir Dropout en los modelos

- Dropout: técnica de regularización. Se eliminan neuronas (y por tanto los pesos asociados) en la fase de entrenamiento con una determinada probabilidad p , para prevenir el sobreaprendizaje.

```
def create_model_simple_cnn():
    model = models.Sequential()
    model.add(layers.BatchNormalization(input_shape=(32, 32, 3)))
    model.add(layers.Conv2D(16, (3, 3), activation='relu', padding='same'))
    model.add(layers.Dropout(0.25))
    model.add(layers.MaxPooling2D((2, 2)))
    model.add(layers.Flatten())
    model.add(layers.Dense(32, activation='relu'))
    model.add(layers.Dropout(0.25))
    model.add(layers.Dense(10, activation='softmax'))
    return model
```

```
def create_simple_pm():
    #modelo simple de pm:
    model = models.Sequential()
    model.add(layers.BatchNormalization(input_shape=(32, 32, 3)))
    model.add(layers.Flatten(input_shape=(32, 32, 3), name="Input_layer"))
    model.add(layers.Dense(50, activation='sigmoid', name="Hidden_layer"))
    model.add(layers.Dropout(0.25))
    model.add(layers.Dense(10, activation='softmax', name="Output_layer"))
    return model
```

Crear y visualizar el modelo

- Una vez definidas las funciones, creamos el modelo mediante una llamada a la función que nos interese

```
model = create_model_simple_cnn()  
#model = create_simple_pm()
```

- Podemos visualizar la arquitectura del modelo creado, así como el número de parámetros (pesos) de cada capa y total.

```
model.summary()
```

Layer (type)	Output Shape	Param #
=====		
batch_normalization_4 (Batch Normalization)	(None, 32, 32, 3)	12

conv2d_4 (Conv2D)	(None, 32, 32, 16)	448

max_pooling2d_4 (MaxPooling2D)	(None, 16, 16, 16)	0

flatten_4 (Flatten)	(None, 4096)	0

dense_8 (Dense)	(None, 32)	131104

dense_9 (Dense)	(None, 10)	330
=====		
Total params: 131,894		
Trainable params: 131,888		
Non-trainable params: 6		

Compilar el modelo

- Debemos compilar el modelo mediante el método ***compile***
- Debemos especificar:
 - El **optimizador** a utilizar (Adam, RMSprop, Adagrad, etc.)
 - La función objetivo a minimizar (**loss**): Utilizaremos como función loss **'sparse_categorical_crossentropy'**, apropiada para problemas de clasificación multiclase, ya que no es necesario transformar la salida o target de los datos para igualar el número de neuronas de salida del modelo. Basta con un número entero que indique la clase del dato correspondiente.
 - La métrica a utilizar para medir el rendimiento de la red. La más adecuada para un problema de clasificación es **'sparse_categorical_accuracy'**

```
model.compile(  
    optimizer=tf.keras.optimizers.Adam(learning_rate=1e-3, ),  
    #optimizer=tf.keras.optimizers.RMSprop(learning_rate=0.001, rho=0.9),  
    loss='sparse_categorical_crossentropy',  
    metrics=['sparse_categorical_accuracy'])
```

Entrenar el modelo

- Finalmente, entrenamos el modelo mediante el método *fit*
- Debemos pasar como argumento los datos de entrenamiento y los de test, así como especificar:
 - El número de épocas o iteraciones (`epochs`)
 - Opcionalmente, el número de batches por época (`steps_per_epoch`) o el tamaño de cada batch (`batch_size`). Si no se especifican, se toman valores por defecto. Consultar https://keras.io/api/models/model_training_apis/
 - En cada batch se actualizan los pesos.
 - Cada cuantas épocas se ejecuta una validación (`validation_freq`)
Si `validation_freq=1` se valida al finalizar cada época. Se puede hacer cada 10 épocas, por ejemplo.

Entrenar el modelo

- Recopilamos toda la información generada durante el entrenamiento y la guardamos en 'historico'

```
historico = model.fit(train_images, train_labels, epochs=10, validation_freq=10,  
                      validation_data=(test_images, test_labels))
```

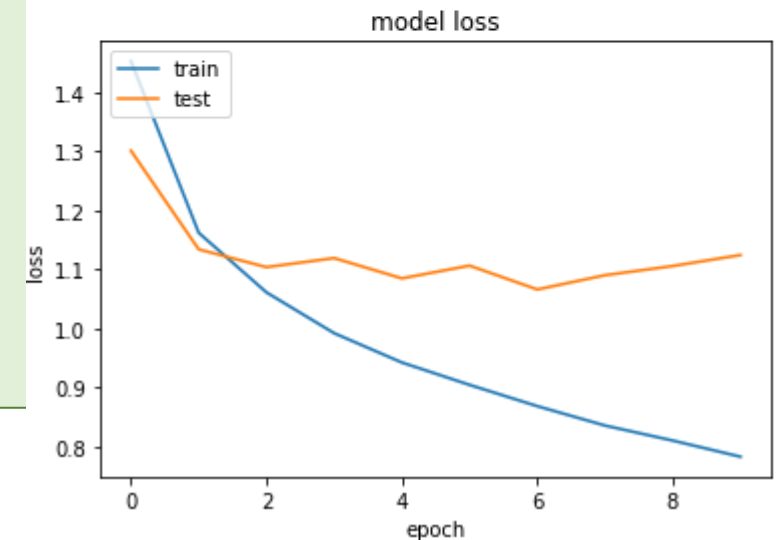
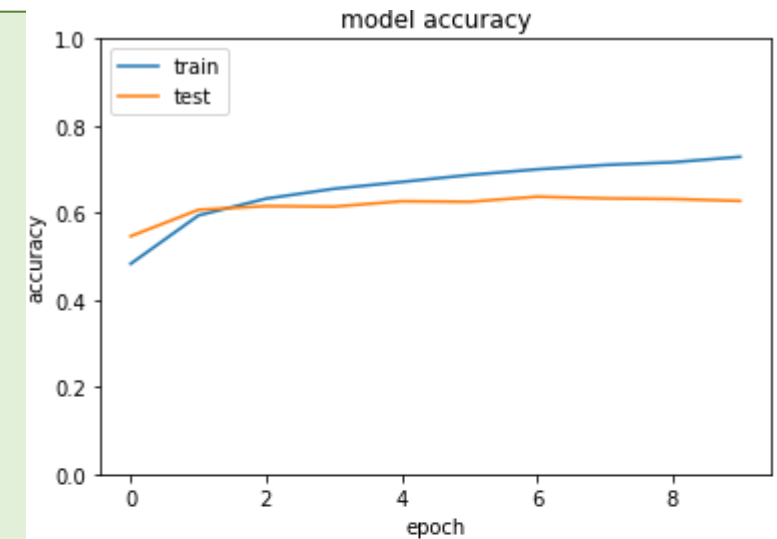
```
Epoch 1/10  
1563/1563 [=====] - 5s 3ms/step - loss: 0.7610 - sparse_categorical_accuracy: 0.7332  
Epoch 2/10  
1563/1563 [=====] - 5s 3ms/step - loss: 0.7340 - sparse_categorical_accuracy: 0.7431  
Epoch 3/10  
1563/1563 [=====] - 5s 3ms/step - loss: 0.7178 - sparse_categorical_accuracy: 0.7472  
Epoch 4/10  
1563/1563 [=====] - 5s 3ms/step - loss: 0.6974 - sparse_categorical_accuracy: 0.7553  
Epoch 5/10  
1563/1563 [=====] - 5s 3ms/step - loss: 0.6783 - sparse_categorical_accuracy: 0.7603  
Epoch 6/10  
1563/1563 [=====] - 5s 3ms/step - loss: 0.6592 - sparse_categorical_accuracy: 0.7662  
Epoch 7/10  
1563/1563 [=====] - 5s 3ms/step - loss: 0.6433 - sparse_categorical_accuracy: 0.7714  
Epoch 8/10
```

Generar plots de evolución del entrenamiento

- Generamos plots de la evolución de 'loss' y 'accuracy'

```
# summarize history for accuracy
plt.plot(historico.history['sparse_categorical_accuracy'])
plt.plot(historico.history['val_sparse_categorical_accuracy'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.ylim([0, 1])
plt.show()

# summarize history for loss
plt.plot(historico.history['loss'])
plt.plot(historico.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
```



Evaluar el modelo y obtener predicciones

- Evaluar el modelo:

```
evaluacion=model.evaluate(test_images, test_labels)
```

```
313/313 [=====] - 1s 2ms/step - loss: 1.1052 - sparse_categorical_accuracy: 0.6475
```

- Obtener predicciones en bruto

```
#predicciones en bruto:  
raw_testPred = model.predict(test_images)  
#prediccion en bruto de los 5 primeros patrones de test: 5 vectores de valores reales  
print(raw_testPred[:5])
```

```
[[3.02679600e-05 6.54377664e-06 4.47412720e-04 9.84654546e-01 2.25518420e-06 1.14351027e-02  
3.37629695e-03 2.06992922e-06 4.03496015e-05 5.06099013e-06]
```

```
[1.66831902e-04 6.28211442e-03 1.71215112e-10 1.12229726e-10 1.08514794e-11 7.44749694e-13  
1.54153343e-10 1.29557403e-13 9.93528068e-01 2.29697598e-05]
```

```
.....
```

Obtener predicciones por clase y matriz de confusión

- Para obtener predicciones con la clase asignada

```
#predicciones de la clase:  
class_testPred = np.argmax(raw_testPred, axis=1)  
#predicción de la clase de los 10 primeros patrones de test  
print(class_testPred[:20])      #predicciones  
print(test_labels[:20,0])      #valores esperados
```

```
[3 8 8 0 6 6 1 6 3 1 0 9 5 7 9 6 5 7 8 6]  
[3 8 8 0 6 6 1 6 3 1 0 9 5 7 9 8 5 7 8 6]
```

- Obtener la matriz de confusión

```
#Confusion Matrix  
  
cm=confusion_matrix(test_labels, class_testPred)  
print(cm)
```

```
[[698  40  60  28  13   8   5  10  82  56]  
 [ 29 801  12  13   6   2   6   9  16 106]  
 [ 77  14 531  70 101  59  62  48  21  17]  
 [ 29  30  95 497  73 130  68  41  19  18]  
 [ 42   6  88  77 578  29  72  82  18   8]  
 [ 18  12  97 230  54 462  38  67  11  11]  
 [ 18  20  59  81  54  20 726   5   9   8]  
 [ 30  18  43  47  53  46  13 720   4  26]  
 [ 94  75  17  15   4   4   3   4 717  67]  
 [ 46 134  10  19   6   4   5  26  23 727]]
```


Guardar los resultados en ficheros

- Evolución de `loss` y `accuracy` de entrenamiento y **test** y la matriz de confusión

```
#guardar resultados
np.savetxt('evaluacion.txt',evaluacion,newline='\t')
np.savetxt('historicoTrainLoss.txt',historico.history['loss'])
np.savetxt('historicoTestLoss.txt',historico.history['val_loss'])
np.savetxt('historicoTrainAcc.txt',historico.history['sparse_categorical_accuracy'])
np.savetxt('historicoTestAcc.txt',historico.history['val_sparse_categorical_accuracy'])
# guardar matriz de confusión
np.savetxt('matrizConf.txt', cm, fmt='%-3d')
```

- Guardar el modelo

```
#guarda el modelo completo
model.save('modelo.h5')
#guarda solo pesos
model.save_weights('pesos.h5')
```

Se pueden ver los archivos de la carpeta actual. Se pueden descargar en nuestro explorador de ficheros o realizar una copia en Drive