Redes Neuronales Profundas (Cifar10)

Joel Isaias Solano Ocampo | A01639289

Importar Tensorflow:

Importamos diferentes librerias necesarias para realizar y visualizar la aplicacion de redes neuronales profundas en base a nuestros datos:

In [4]:
```
import tensorflow as tf
import matplotlib.pyplot as plt
from tensorflow.keras import datasets, layers, models
```

Preparar Dataset CIFAR10:

Obtenemos los datos de *CIFAR10* de la libreria *datasets* obtenida a traves de *tensorflow.keras*:

In [5]:
```
(train_images, train_labels), (test_images, test_labels) = datasets.cifar10.load_data(
```

Normalizar:

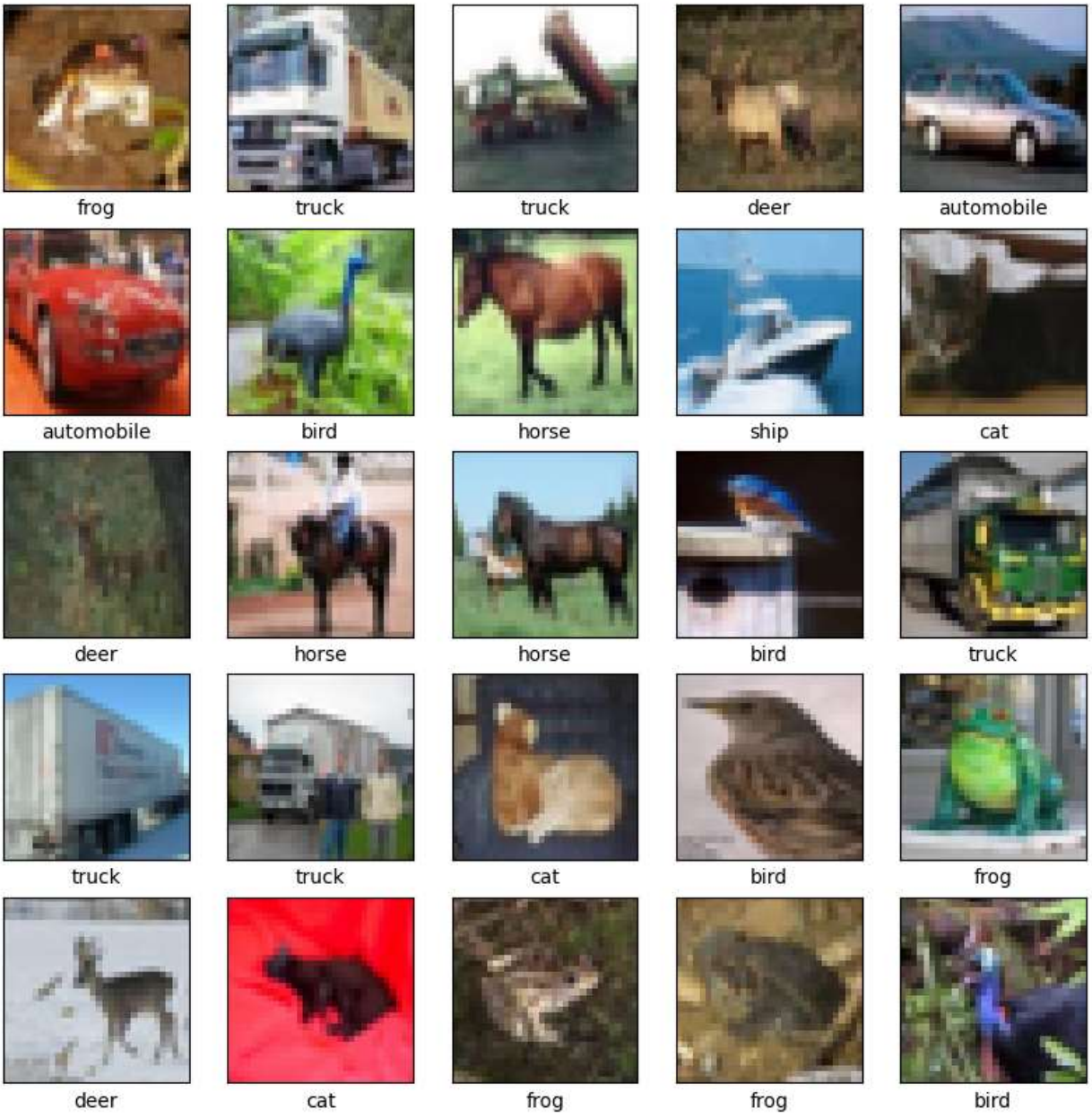
Normalizamos los datos dividiendo _train*images* y _test*images* en 255.0:

In [6]:
```
train_images, test_images = train_images/255.0, test_images/255
```

Validacion de datos:

Validamos que todos los datos ya normalizados correspondan en cuanto visualizacion y nombre de las clases para proceder:

In [7]:
```
class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse'
plt.figure(figsize = (10,10))
for i in range(25):
  plt.subplot(5, 5, i+1)
  plt.xticks([])
  plt.yticks([])
  plt.grid(False)
  plt.imshow(train_images[i])
  plt.xlabel(class_names[train_labels[i][0]])
plt.show()
```



Capas de convolucion:

Para nuestro modelo, importamos una nueva libreria llamada *regularizers* y emepzamos a agregar diferentes capas de convolucion con diferentes numero de neuronas pero con el mismo metodo de activacion:

In [8]:
```
from tensorflow.keras import regularizers

model = models.Sequential()
model.add(layers.Conv2D(64, (3,3), activation = 'relu', input_shape = (32, 32, 3)))
model.add(layers.MaxPooling2D((2,2)))
model.add(layers.Conv2D(128, (3,3), activation = 'relu'))
model.add(layers.MaxPooling2D((2,2)))
model.add(layers.Conv2D(128, (3,3), activation = 'relu'))
```

Arquitectura:

Verificamos que las capas de convolucion hayan sido añadidas correctamente verificando difenrentes parametros como el tipo de capa, forma de la salida de datos y sus respectivos parametros:

In [9]:
```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 30, 30, 64)        1792

 max_pooling2d (MaxPooling2  (None, 15, 15, 64)        0
 D)

 conv2d_1 (Conv2D)           (None, 13, 13, 128)       73856

 max_pooling2d_1 (MaxPoolin  (None, 6, 6, 128)         0
 g2D)

 conv2d_2 (Conv2D)           (None, 4, 4, 128)         147584

=================================================================
Total params: 223232 (872.00 KB)
Trainable params: 223232 (872.00 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Capas densas:

En nuestro modelo, continuamos agregando diferentes capas densas con diferentes numero de neuronas y metodo de activacion, usando *relu* y *sigmoid* respectivamente:

In [10]:
```
model.add(layers.Flatten())
model.add(layers.Dense(128, activation='relu'))
model.add(layers.Dense(20, activation='sigmoid'))
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 30, 30, 64)        1792

 max_pooling2d (MaxPooling2  (None, 15, 15, 64)        0
 D)

 conv2d_1 (Conv2D)           (None, 13, 13, 128)       73856

 max_pooling2d_1 (MaxPoolin  (None, 6, 6, 128)         0
 g2D)

 conv2d_2 (Conv2D)           (None, 4, 4, 128)         147584

 flatten (Flatten)           (None, 2048)              0

 dense (Dense)               (None, 128)               262272

 dense_1 (Dense)             (None, 20)                2580

=================================================================
Total params: 488084 (1.86 MB)
Trainable params: 488084 (1.86 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Entrenamiento:

Una vez llegado a este punto, por fin podemos comenzar con el entrenamiento del modelo con los datos, usando como optimizador a *adam*, usando un loss de *SparseCategoricalCrossentropy* y tomando como metricas el *accuracy*.

Para el historial del entrenamiento, tomamos los *labels* e *images* de entrenamiento y realizamos un *epochs* de 10.

In [11]:
```
model.compile(optimizer='adam',
              loss = tf.keras.losses.SparseCategoricalCrossentropy(from_logits = True)
              metrics = ['accuracy'])

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

```
Epoch 1/10
/usr/local/lib/python3.10/dist-packages/keras/src/backend.py:5714: UserWarning: "`spa
rse_categorical_crossentropy` received `from_logits=True`, but the `output` argument
was produced by a Softmax activation and thus does not represent logits. Was this int
ended?
  output, from_logits = _get_logits(
```

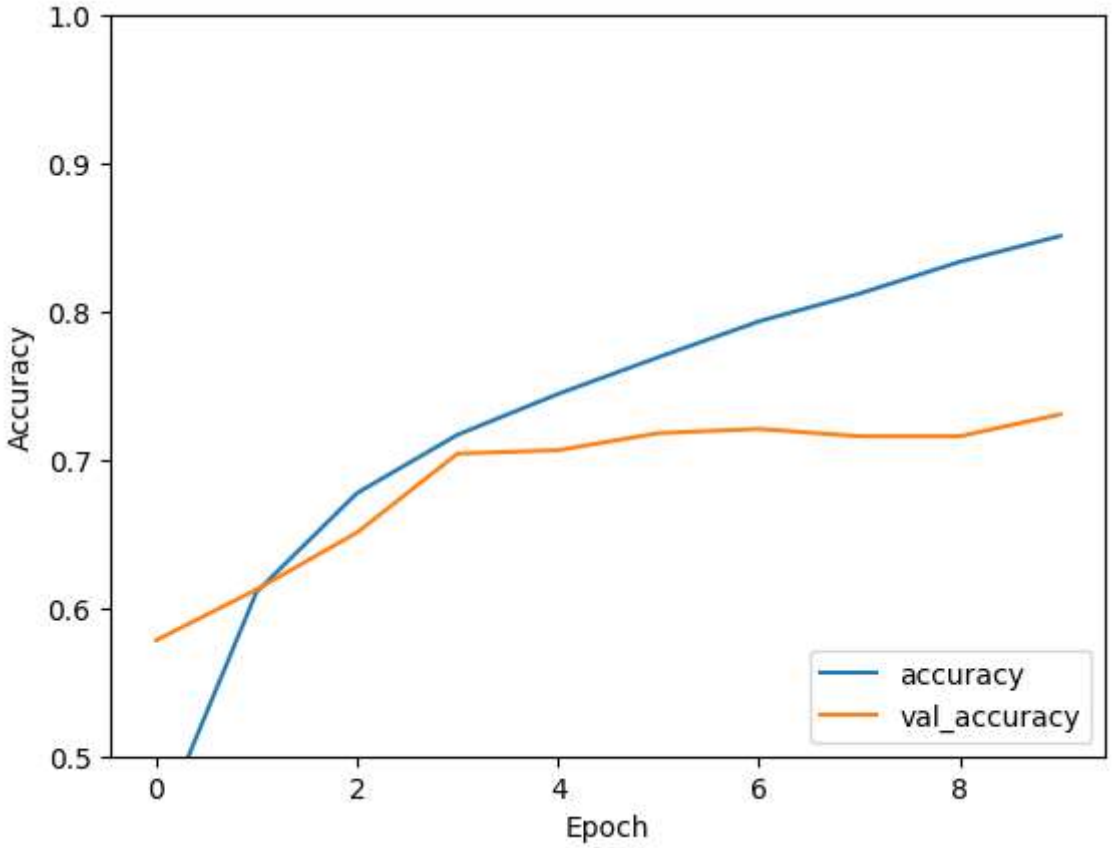
```
1563/1563 [==============================] - 194s 124ms/step - loss: 1.5261 - accurac
y: 0.4473 - val_loss: 1.2125 - val_accuracy: 0.5782
Epoch 2/10
1563/1563 [==============================] - 184s 118ms/step - loss: 1.1006 - accurac
y: 0.6108 - val_loss: 1.1034 - val_accuracy: 0.6125
Epoch 3/10
1563/1563 [==============================] - 185s 119ms/step - loss: 0.9204 - accurac
y: 0.6776 - val_loss: 0.9987 - val_accuracy: 0.6509
Epoch 4/10
1563/1563 [==============================] - 187s 120ms/step - loss: 0.8066 - accurac
y: 0.7169 - val_loss: 0.8537 - val_accuracy: 0.7042
Epoch 5/10
1563/1563 [==============================] - 185s 119ms/step - loss: 0.7228 - accurac
y: 0.7444 - val_loss: 0.8505 - val_accuracy: 0.7065
Epoch 6/10
1563/1563 [==============================] - 186s 119ms/step - loss: 0.6527 - accurac
y: 0.7693 - val_loss: 0.8338 - val_accuracy: 0.7180
Epoch 7/10
1563/1563 [==============================] - 188s 120ms/step - loss: 0.5884 - accurac
y: 0.7934 - val_loss: 0.8428 - val_accuracy: 0.7209
Epoch 8/10
1563/1563 [==============================] - 186s 119ms/step - loss: 0.5303 - accurac
y: 0.8122 - val_loss: 0.8657 - val_accuracy: 0.7159
Epoch 9/10
1563/1563 [==============================] - 187s 120ms/step - loss: 0.4659 - accurac
y: 0.8336 - val_loss: 0.9009 - val_accuracy: 0.7159
Epoch 10/10
1563/1563 [==============================] - 189s 121ms/step - loss: 0.4185 - accurac
y: 0.8511 - val_loss: 0.8940 - val_accuracy: 0.7307
```

Resultados de entrenamiento:

Visualizamos en una grafica (donde el eje x representa el numero de *epoch* y el eje y el puntaje del *accuracy*) los resultados obtenidos de *accuracy* y _val*accuracy* en base al entrenamiento previamente realizado.

In [12]:
```
plt.plot(history.history['accuracy'], label = 'accuracy')
plt.plot(history.history['val_accuracy'], label = 'val_accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.ylim([0.5,1])
plt.legend(loc='lower right')
test_loss, test_acc = model.evaluate(test_images, test_labels, verbose = 2)
```

```
313/313 - 9s - loss: 0.8940 - accuracy: 0.7307 - 9s/epoch - 28ms/step
```



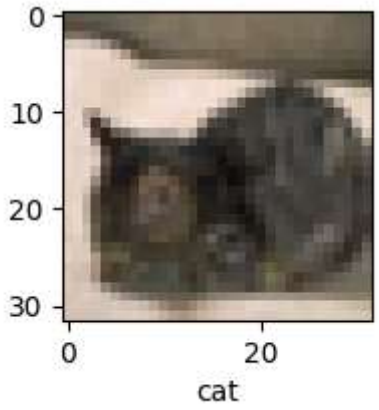
In [13]:
```
print(test_acc)
```

```
0.7307000160217285
```

Prediccion:

Para verificar la prediccion de nuestro modelo ya entrenado con los datos de verificacion, obtenemos un dato aleatorio y validamos su contenido:

In [14]:
```
n = 115
plt.figure(figsize=(2,2))
plt.imshow(test_images[n])
plt.xlabel(class_names[test_labels[n][0]])
plt.show()
```



Resultados de prediccion:

Para terminar, realizamos la prediccion de nuestro modelo ya entrenado con dicho dato aleatorio y verificamos si la imagen pertenece a dicha clase y checamos la probabilidad:

In [15]:
```
predictions = model.predict(test_images)
print(predictions[n])

import numpy as np

print('La imagen pertenece al grupo {} con una probabilidad de {:2f} %'
      .format(class_names[np.argmax(predictions[n])], 100 * np.max(predictions[n])))
```

```
313/313 [==============================] - 9s 28ms/step
[9.4179499e-01 1.1382879e-03 3.6326054e-02 6.7334211e-01 9.9726957e-01
 9.3871224e-01 9.8350698e-01 4.1348833e-01 3.0492949e-01 2.4412250e-02
 6.6223743e-10 1.3205430e-09 2.2750102e-09 7.8901663e-10 2.7550000e-09
 3.2185876e-09 2.4639636e-08 1.7967511e-10 8.9739647e-09 1.0894573e-08]
La imagen pertenece al grupo deer con una probabilidad de 99.726957 %
```