Redes convolucionales

Las redes convolucionales se basan en la operación de convolución que se usa para obtener información local de los datos. Su principal aplicación es en imágenes donde cada pixel tiene que ver con los pixeles de su vecindad.

Una convolución es una combinación de la imagen y una matríz llamada **filtro**. Un filtro tiene un tamaño típico de 3x3 ó 5x5 pixeles, mientras que las imágenes son mucho más grandes. El filtro hace un barrido sobre los pixeles de la imagen aplicando la operación de multiplicación (coordenada a coordenada) seguida de la suma.

El resultado de una convolución es una transformación de la imagen llamada **mapa de activación**. Diferentes filtros producen diferentes mapas de activación que se pueden volver a transformar usando otros filtros para obtener tranformaciones más complejas. Estos mapas de activación se pueden ver como características extraídas de las imágenes.

Al final, los mapas de acitvación se *aplanan* para convertirlos en un vector y luego aplicar capas densas antes de hacer la predicción.

Para correr este notebook se necesitará el archivo: - house.jpg

Importante

Se recomienda usar una GPU para ejecutar este notebook. En el menú **Entorno de ejecución**, elegir Cambiar tipo de entorno de ejecución y en la sección **Acelerador por hardware** elegir una GPU o TPU.

```
!wget https://raw.githubusercontent.com/msubrayada/MeIA2025/refs/heads/main/house.jpg
```

```
--2025-06-03 23:34:27--  https://raw.githubusercontent.com/msubrayada/MeIA2025/refs/heads/ma
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 185.199.110.133, 185.199.
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|185.199.110.133|:443... 
HTTP request sent, awaiting response... 200 OK
Length: 39019 (38K) [image/jpeg]
```

```
Saving to: ‘house.jpg.1’

house.jpg.1           0%[                    ]       0  --.-KB/s               house.jpg.1

2025-06-03 23:34:27 (3.59 MB/s) - ‘house.jpg.1’ saved [39019/39019]
```

Ejemplos de convoluciones

En el siguiente ejemplo tenemos la matriz `A` y se la aplicará el filtro `f`.

```
from scipy.ndimage import correlate
import numpy as np
```

```
A = np.array([[4,1,7,2,3], [8,3,1,2,2], [6,5,7,4,3], [2,2,3,1,3],[4,3,4,2,3]])
A
```

```
array([[4, 1, 7, 2, 3],
       [8, 3, 1, 2, 2],
       [6, 5, 7, 4, 3],
       [2, 2, 3, 1, 3],
       [4, 3, 4, 2, 3]])
```

```
f = np.array([[1, 0, -1], [1, 0, -1], [1, 0, -1]])
f
```

```
array([[ 1,  0, -1],
       [ 1,  0, -1],
       [ 1,  0, -1]])
```

```
correlate(A, f, mode='constant')
```

```
array([[ -4,   4,   0,   3,   4],
       [ -9,   3,   1,   7,   8],
       [-10,   5,   3,   3,   7],
       [-10,  -2,   3,   5,   7],
       [ -5,  -1,   2,   1,   3]])
```

Ejemplo con imágenes

Ahora veamos el efecto de aplicar una convolución a una imagen en lugar de una matriz.

```
from skimage.io import imread
import matplotlib.pyplot as plt
```

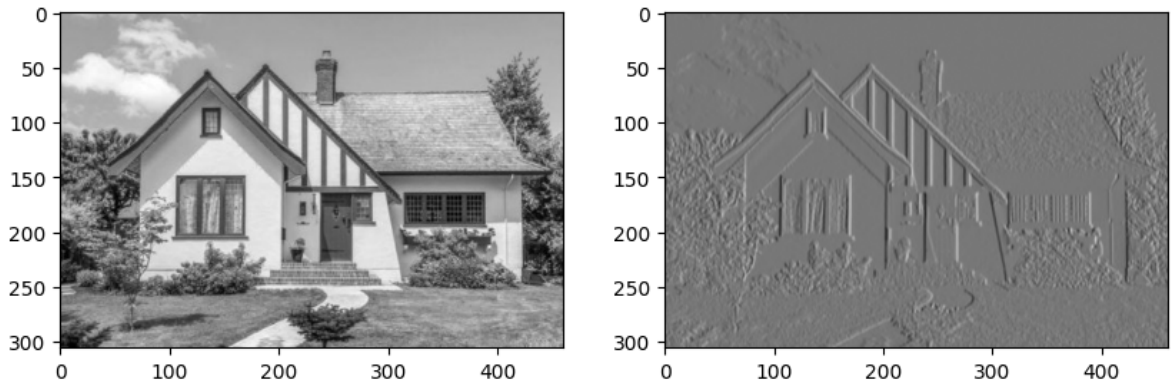
```
img = imread("house.jpg", as_gray=True)
plt.imshow(img, cmap='gray')
plt.show()
```

El siguiente kernel se usa para detectar bordes horizontales.

```
f = np.array([[-1, 0, 1], [-1, 0, 1], [-1, 0, 1]])
f
```

```
array([[-1,  0,  1],
       [-1,  0,  1],
       [-1,  0,  1]])
```

```
T = correlate(img, f, mode='constant')
plt.figure(figsize=(10, 5))
plt.subplot(1,2,1)
plt.imshow(img, cmap='gray')
plt.subplot(1,2,2)
plt.imshow(T, cmap='gray')
plt.show()
```

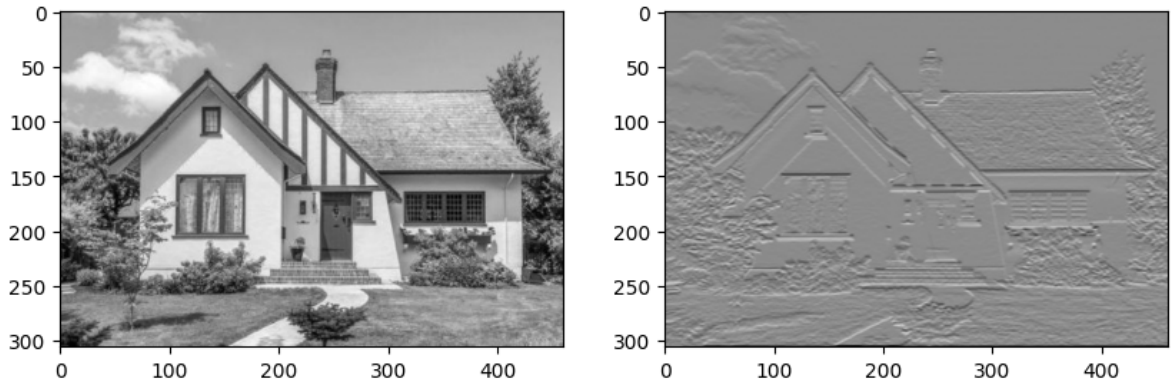


También se puede *voltear* para obtener un kernel que identifique bordes verticales. Note las diferencias en los mapas de activación.

```
f = np.array([[-1, -1, -1], [0, 0, 0], [1, 1, 1]])
f
```

```
array([[-1, -1, -1],
       [ 0,  0,  0],
       [ 1,  1,  1]])
```

```
T = correlate(img, f, mode='constant')
plt.figure(figsize=(10, 5))
plt.subplot(1,2,1)
plt.imshow(img, cmap='gray')
plt.subplot(1,2,2)
plt.imshow(T, cmap='gray')
plt.show()
```

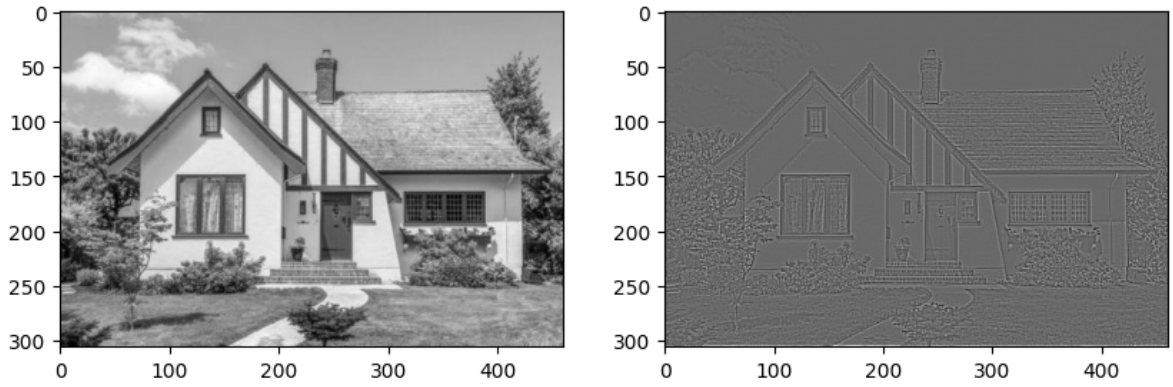


El siguiente filtro resalta los bordes y elimina todo lo demás.

```
f = np.array([[-1, -1, -1], [-1, 8, -1], [-1, -1, -1]])
f
```

```
array([[-1, -1, -1],
       [-1,  8, -1],
       [-1, -1, -1]])
```

```
T = correlate(img, f, mode='constant')
plt.figure(figsize=(10, 5))
plt.subplot(1,2,1)
plt.imshow(img, cmap='gray')
plt.subplot(1,2,2)
plt.imshow(T, cmap='gray')
plt.show()
```



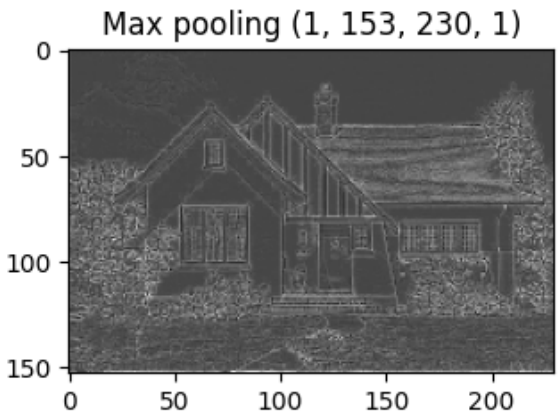
Pooling

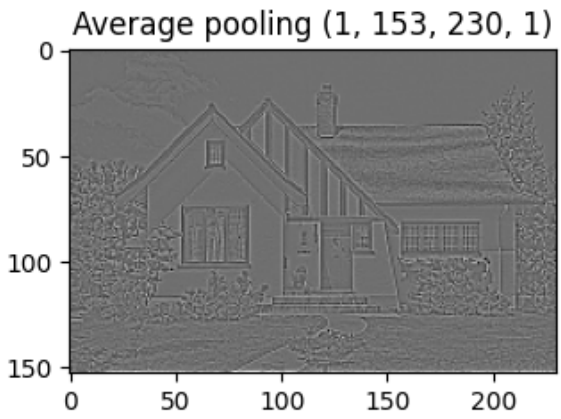
La operación de pooling permite trabajar con mapas de activación más pequeños. Lo más común es dividir los mapas a la mitad en alto y ancho, logrando una reducción del 75% del tamaño en pixeles. Esto permite trabajar con mapas más pequeños disminuyendo las operaciones necesarias.

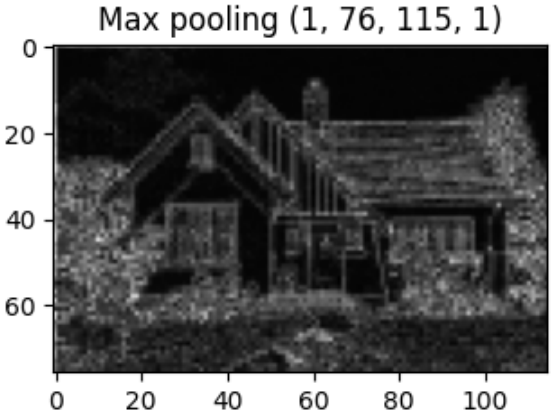
```
from tensorflow.keras import layers

def apply_pooling(A, k=3):
    print("Imagen original:", A.shape)
    max_pool = layers.MaxPooling2D((2, 2))
    avg_pool = layers.AveragePooling2D((2, 2))

    B = A.copy()
    C = A.copy()
    for i in range(k):
        B = max_pool(B)
        C = avg_pool(C)
        plt.figure(figsize=(8, 8))
        plt.subplot(k,2,1)
        plt.imshow(B[0,:,:,0], cmap='gray')
        plt.title(f"Max pooling {B.shape}")

        plt.subplot(k,2,2)
        plt.imshow(C[0,:,:,0], cmap='gray')
        plt.title(f"Average pooling {C.shape}")

        plt.show()
```

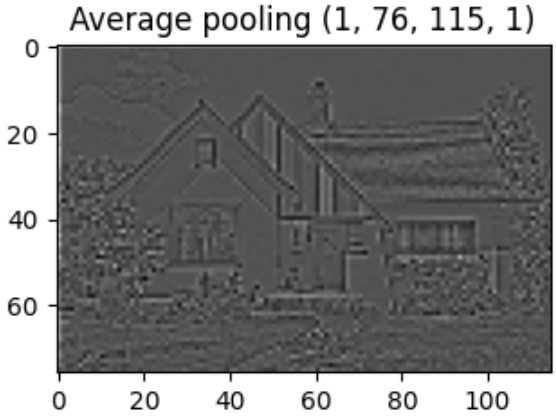
```
apply_pooling(T.reshape(1, T.shape[0], T.shape[1], 1))
```

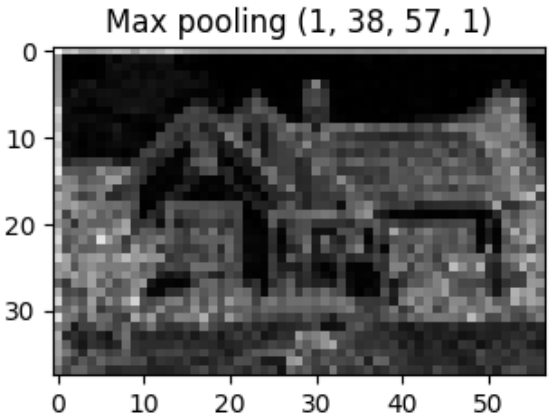
```
Imagen original: (1, 306, 461, 1)
```

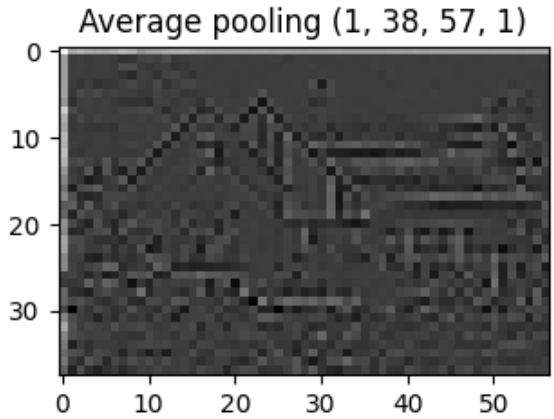

Max pooling (1, 153, 230, 1)
0
50
100
150
0
50
100
150
200


Average pooling (1, 153, 230, 1)
0
50
100
150
0
50
100
150
200


Max pooling (1, 76, 115, 1)
0
20
40
60
0
20
40
60
80
100


Average pooling (1, 76, 115, 1)
0
20
40
60
0
20
40
60
80
100


Max pooling (1, 38, 57, 1)
0
10
20
30
0
10
20
30
40
50


Average pooling (1, 38, 57, 1)
0
10
20
30
0
10
20
30
40
50

Ejemplo de Perros y gatos

Ahora veremos un ejemplo de como se usa la operación convolución en una red neuronal para clasificar imágenes de perros y gatos. Las primeras capas de la red serán de convolución seguidas de MaxPooling y al final una capa densa para hacer la clasificación.

Primero debemos obtener el conjunto de datos que está en la colección de `tensorflow_datasets`. Se trata de imágenes de perros y gatos y la tarea es diferenciarlos. Vamos a partir los datos en 80% entrenamiento y 20% de validación.

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.models import Sequential
from tensorflow.keras import layers, optimizers
import tensorflow_datasets as tfds
import tensorflow as tf
```

```
ds_train, ds_val = tfds.load('cats_vs_dogs', split=['train[:80%]', 'train[80%:]'], shuffle_f
len(ds_train), len(ds_val)
```

```
(18610, 4652)
```

Con la función `normalize_img` se reescalan las imágenes a 150x150 y que tengan valores entre 0 y 1. Se hacen los lotes de tamaño 128.

```
def normalize_img(data):
  image = tf.image.convert_image_dtype(data['image'], tf.float32)
  return tf.image.resize(image, [150, 150]), data['label']

ds_train = ds_train.map(normalize_img, num_parallel_calls=tf.data.AUTOTUNE)
ds_train = ds_train.cache()
ds_train = ds_train.batch(128)
ds_train = ds_train.prefetch(tf.data.AUTOTUNE)

ds_val = ds_val.map(normalize_img)
ds_val = ds_val.cache()
ds_val = ds_val.batch(128)
ds_val = ds_val.prefetch(tf.data.AUTOTUNE)
```

Aquí se crea la estructura de la red convolucional. La primera capa es de convolución (Conv2D) tiene 32 filtros de 3x3 con una activación ReLU. El tamaño de las imágenes de entreada es de 150x150 pixeles en los 3 canales (RGB). Luego viene una capa de MaxPooling que divide a la mitad las dimensiones de los mapas de activación.

Siguen 3 bloque de capas de convolución seguidas de una MaxPooling.

Se aplanan los mapas de activación que entran a una capa densa de 128 neuronas.

La salida de la red es una neurona, que se usará para hacer la predicción.

```
model = Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu', padding='same', input_shape=(150, 150
model.add(layers.MaxPooling2D((2, 2)))

model.add(layers.Conv2D(64, (3, 3), activation='relu', padding='same'))
model.add(layers.MaxPooling2D((2, 2)))

model.add(layers.Conv2D(128, (3, 3), activation='relu', padding='same'))
model.add(layers.MaxPooling2D((2, 2)))

model.add(layers.Conv2D(64, (3, 3), activation='relu', padding='same'))
model.add(layers.MaxPooling2D((2, 2)))

model.add(layers.Flatten())
model.add(layers.Dropout(0.5))
model.add(layers.Dense(128, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
model.summary()
```

```
/usr/local/lib/python3.11/dist-packages/keras/src/layers/convolutional/base_conv.py:107: User
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

```
Model: "sequential"
```

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 150, 150, 32)	896
max_pooling2d_1 (MaxPooling2D)	(None, 75, 75, 32)	0
conv2d_1 (Conv2D)	(None, 75, 75, 64)	18,496
max_pooling2d_2 (MaxPooling2D)	(None, 37, 37, 64)	0
conv2d_2 (Conv2D)	(None, 37, 37, 128)	73,856

```
 max_pooling2d_3 (MaxPooling2D)    (None, 18, 18, 128)                0

 conv2d_3 (Conv2D)                 (None, 18, 18, 64)             73,792

 max_pooling2d_4 (MaxPooling2D)    (None, 9, 9, 64)                   0

 flatten (Flatten)                 (None, 5184)                       0

 dropout (Dropout)                 (None, 5184)                       0

 dense (Dense)                     (None, 128)                  663,680

 dense_1 (Dense)                   (None, 1)                        129


 Total params: 830,849 (3.17 MB)

 Trainable params: 830,849 (3.17 MB)

 Non-trainable params: 0 (0.00 B)
```

Note que la primera capa de convolución tiene una salida de 32 mapas de activación (de 150x150). La resolución de los mapas de activación se va reduciendo a la mitad hasta llegar a los 9x9 pixeles. Al final, se tienen 9x9x64 = 5184 dimensiones del vector aplanado.

```
model.compile(loss='binary_crossentropy',
              optimizer=optimizers.Adam(learning_rate=.001),
              metrics=['accuracy'])
```

```
history = model.fit(ds_train,
                    validation_data=ds_val,
                    epochs=10)
```

```
Epoch 1/10
146/146            65s 356ms/step - accuracy: 0.5557 - loss: 0.6810 - val_accuracy: 0.7208 -
Epoch 2/10
146/146            15s 103ms/step - accuracy: 0.7177 - loss: 0.5507 - val_accuracy: 0.7702 -
Epoch 3/10
146/146            16s 109ms/step - accuracy: 0.7661 - loss: 0.4851 - val_accuracy: 0.8080 -
```

```
Epoch 4/10
146/146             15s 101ms/step - accuracy: 0.8032 - loss: 0.4274 - val_accuracy: 0.8132 -
Epoch 5/10
146/146             16s 111ms/step - accuracy: 0.8241 - loss: 0.3938 - val_accuracy: 0.8474 -
Epoch 6/10
146/146             15s 102ms/step - accuracy: 0.8512 - loss: 0.3429 - val_accuracy: 0.8648 -
Epoch 7/10
146/146             15s 102ms/step - accuracy: 0.8673 - loss: 0.3098 - val_accuracy: 0.8650 -
Epoch 8/10
146/146             15s 103ms/step - accuracy: 0.8830 - loss: 0.2809 - val_accuracy: 0.8762 -
Epoch 9/10
146/146             16s 111ms/step - accuracy: 0.8870 - loss: 0.2640 - val_accuracy: 0.8766 -
Epoch 10/10
146/146             15s 102ms/step - accuracy: 0.9026 - loss: 0.2316 - val_accuracy: 0.8897 -
```

Guardamos el modelo para usarlo después.

```
model.save("cnn-dogs-cats.keras")
```

Luego lo podemos leer con:

```
from tensorflow.keras.models import load_model
model = load_model("cnn-dogs-cats.keras")
```

Mostrar algunas predicciones

A continuación vamos a mostrar algunas de las imágenes y cómo fueron clasificadas por la red convolucional. Las imágenes fueron tomadas del conjunto de validación.

```
for batch in ds_val:
    break
images = batch[0]
predictions = model.predict(images)
predictions.shape
```

```
4/4             1s 14ms/step
```

```
(128, 1)
```

```
c = 0
fig = plt.figure(figsize=(8, 4))
for im, p in zip(images[0:8], predictions[0:8]):
    c += 1
    fig.add_subplot(2, 4, c)
    plt.axis('off')
    plt.imshow(im)
    plt.title(f"Predicción: {p[0]:.2f}")
plt.show()
```

Ejercicio

Modificar la configuración de la red para obtener el mayor accuracy posible. **Dejar el parámetro de 10 epochs fijo.** Los cambios que se sugieren son: - Cambiar el número de filtros por capa Conv2D. - Cambiar el tamaño del filtro. Usar 3x3 o 5x5. - Modificar la cantidad de capas convolucionales de la red. - Cambiar el número de neuronas de la capa densa. - Cambiar el learning rate. - Probar con diferentes tamaños de lote (`batch_size`).