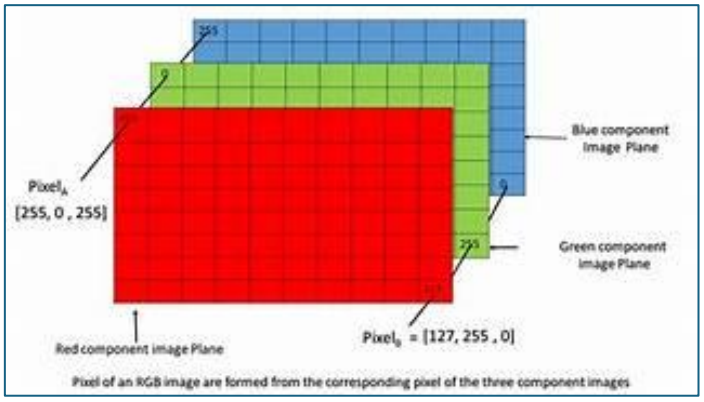
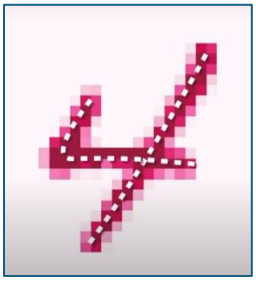
Comenzaremos con el proceso de convolución viendo el problema de por qué se inventó.

¿Cómo procesamos los humanos una imagen? Simplemente, detectando los ejes o las líneas de los objetos que la conforman.

Los ejes y bordes nos ayudan a a saber dónde están los objetos que existen en toda imagen. Nuestro ojo está acostumbrado a detectarlos. Para él es sencillo.

Informáticamente, la información de una imagen visual se descompone en una matriz de pixeles dispuestos en filas y columnas. Cada píxel representa un punto de la imagen en un sistema bidimensional y contiene información sobre el color (siendo el modelo RGB el más común en IA), y la intensidad luminosa y la opacidad. Cada una de las tres componentes (rojo, verde o azul) tiene un valor numérico entre 0 y 255 que indica la intensidad de ese color específico en el píxel.

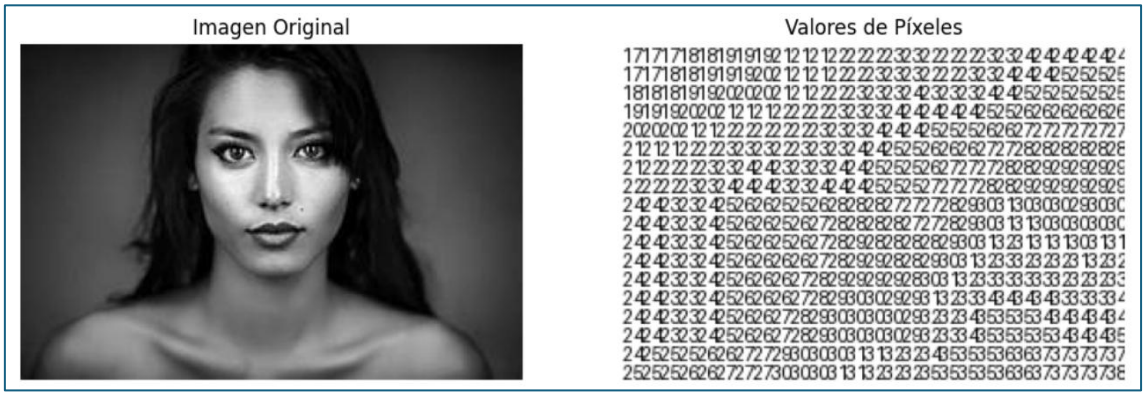


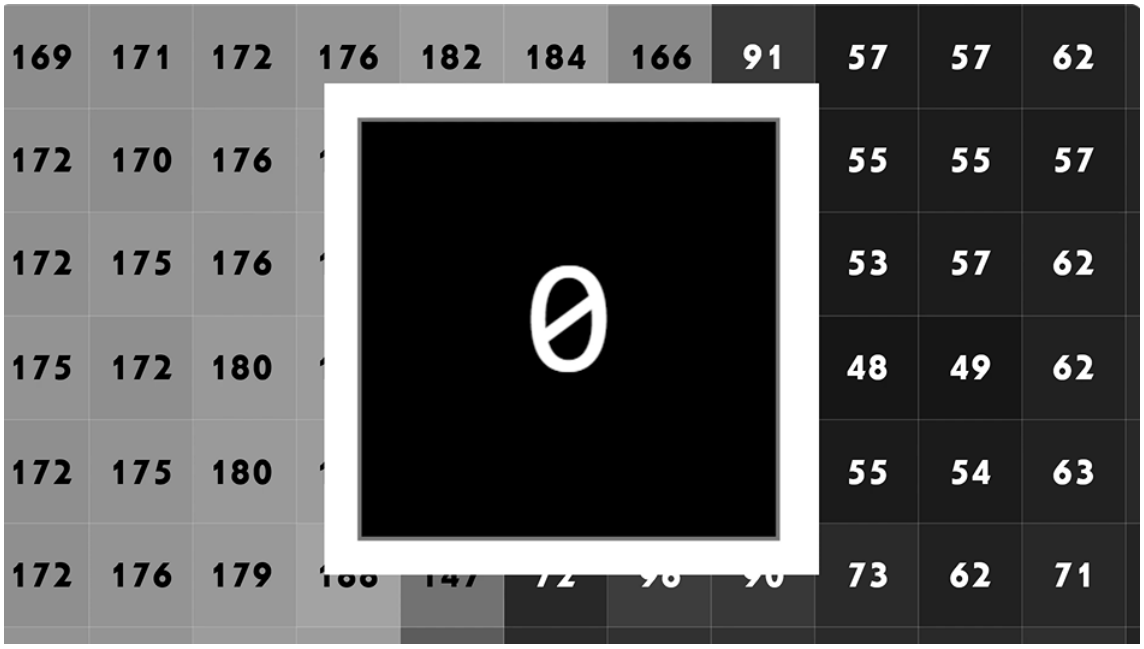


Si nuestro ojo detecta un cambio brusco de color (y valor entre pixeles cercanos), eso es un eje.

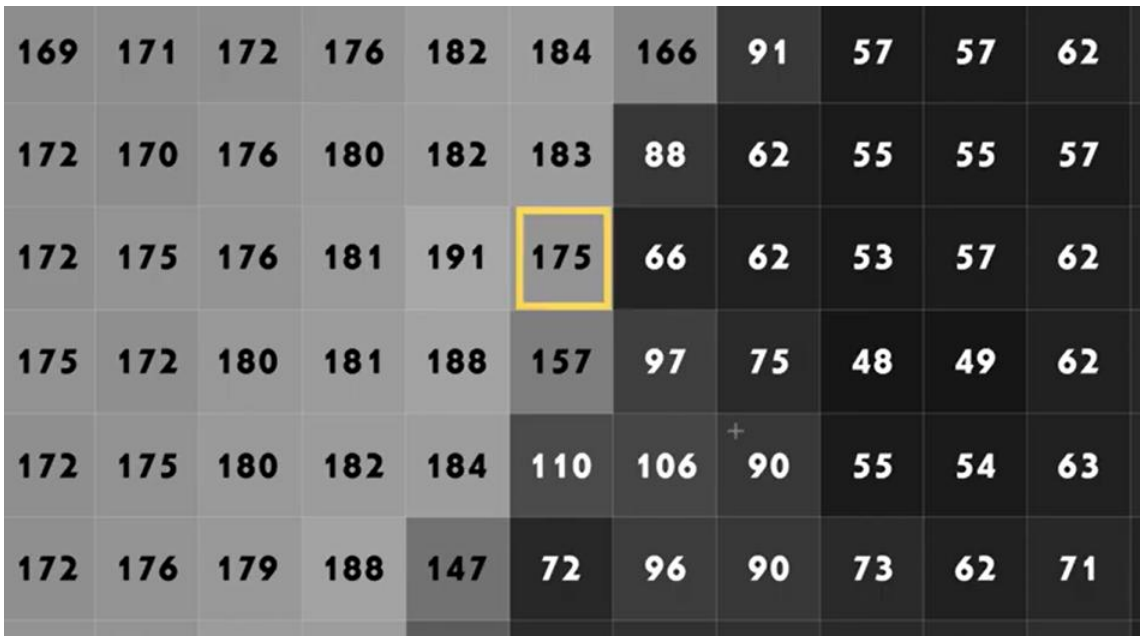
Los ordenadores no ven lo mismo que nosotros, ellos ven una serie de píxeles y cada píxel toma un valor numérico entre 0 y 255, que indica su nivel de color. Podemos trabajar con el modelo RGB y tendremos 3 valores numéricos, de 0 a 255, para los colores rojo, verde o azul.

Para simplificar y entender el proceso de "convolución", trabajaremos con imágenes en blanco y negro y, en este caso, el valor numérico 0 en el píxel significará negro y el valor numérico 255 blanco.

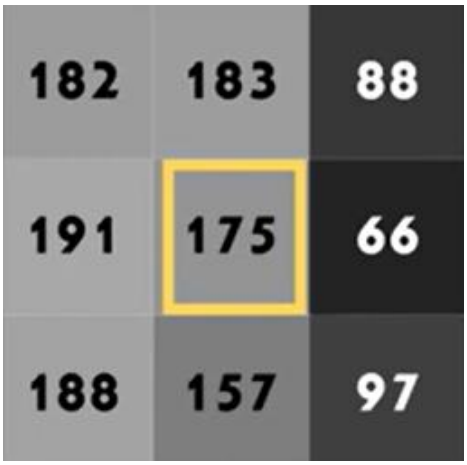




Pues bien, ¿cómo puedo saber si el píxel de valor numérico 175 (ver siguiente imagen) es un eje o borde en la imagen?

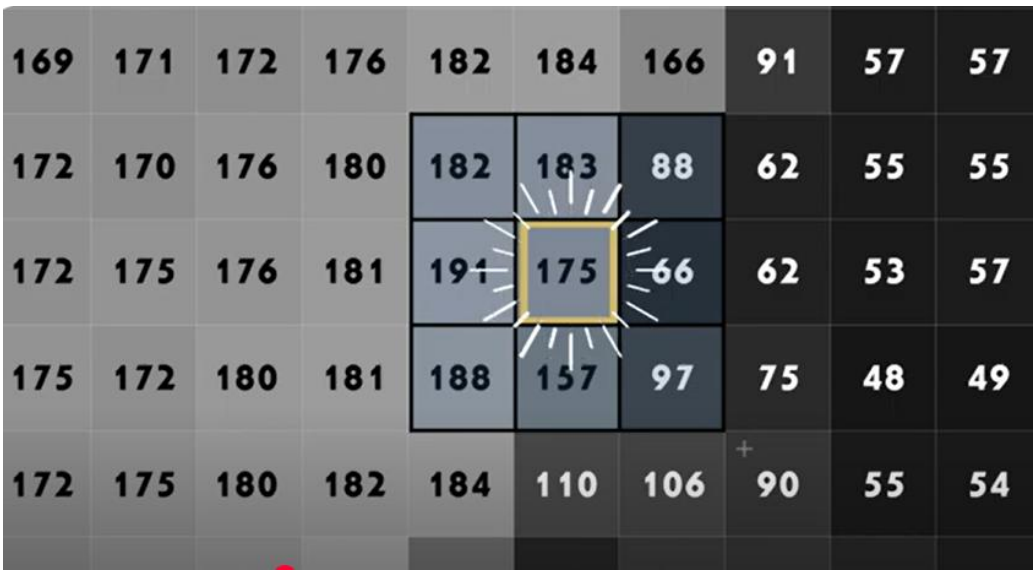


La única forma consiste en comparar el valor numérico de ese píxel con los valores numéricos de los pixeles que lo bordean y ver si hay cambios numéricos fuertes en los números de los pixeles que lo bordean. Es decir, no podemos basarnos en un píxel de forma individual.

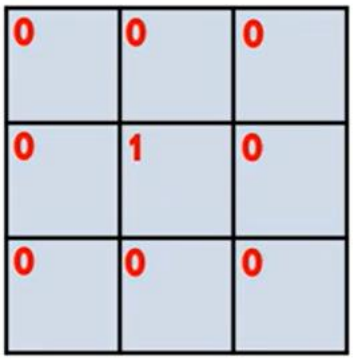


Pixeles que bordean al píxel principal

Vamos a entenderlo definiendo un cuadro de 3x3 y que llamaremos núcleo o kernel. A cada una de las casillas de ese kernel le indicamos un valor numérico. Y vamos a iterar el núcleo en todos los pixeles de la imagen, uno por uno. Pero en cada uno, en lugar de ver qué valor tienen, pondremos encima nuestro núcleo. Llamaremos al píxel que estamos revisando nuestro píxel principal:

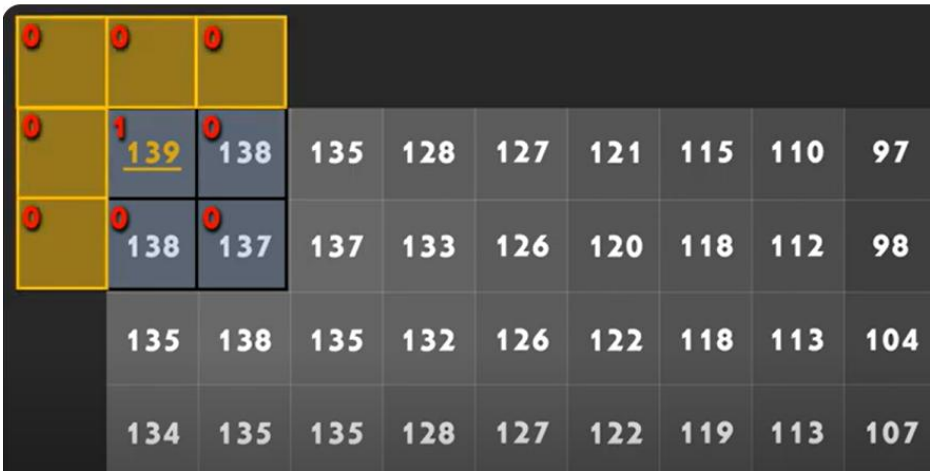


Asignemos los siguientes valores al núcleo:

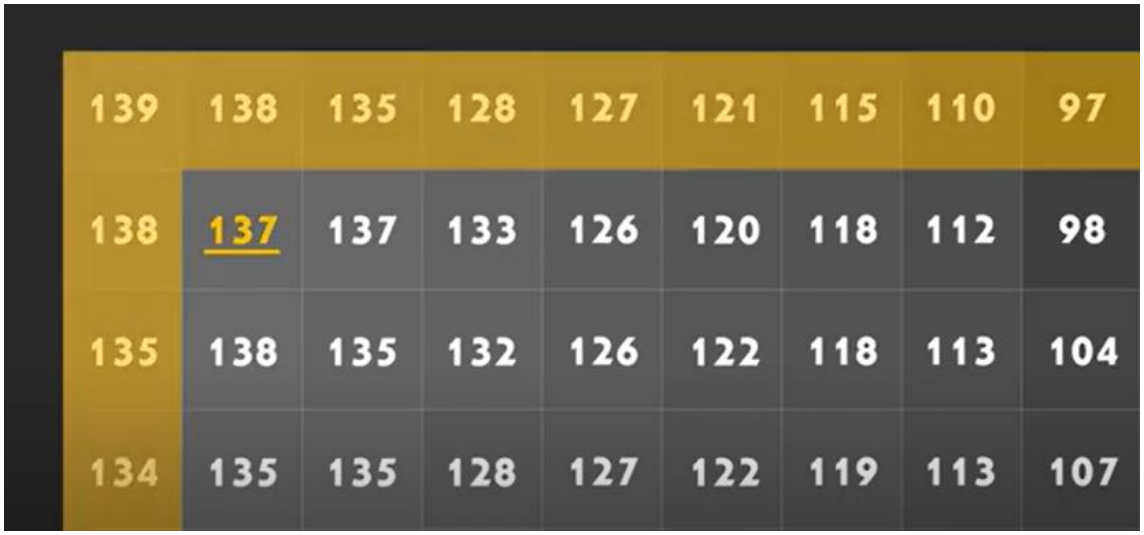


Núcleo o kernel

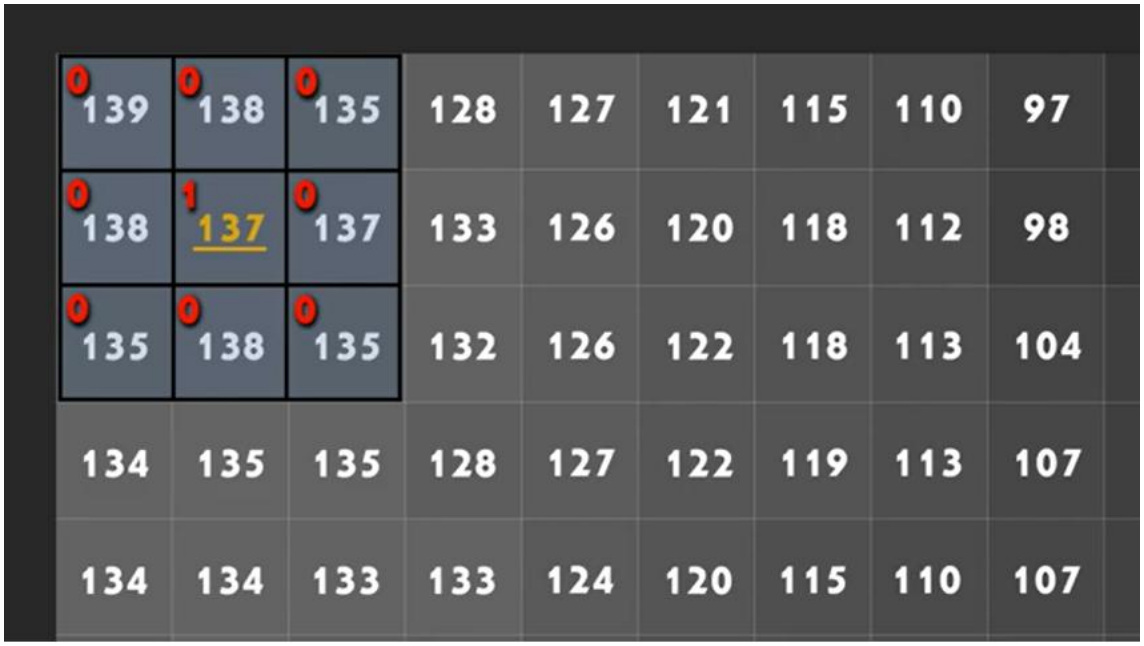
Si comenzamos con el primer píxel de la imagen (en su borde), nos encontramos con un problema: Lo solucionaremos descartando los píxeles de los bordes de la imagen.



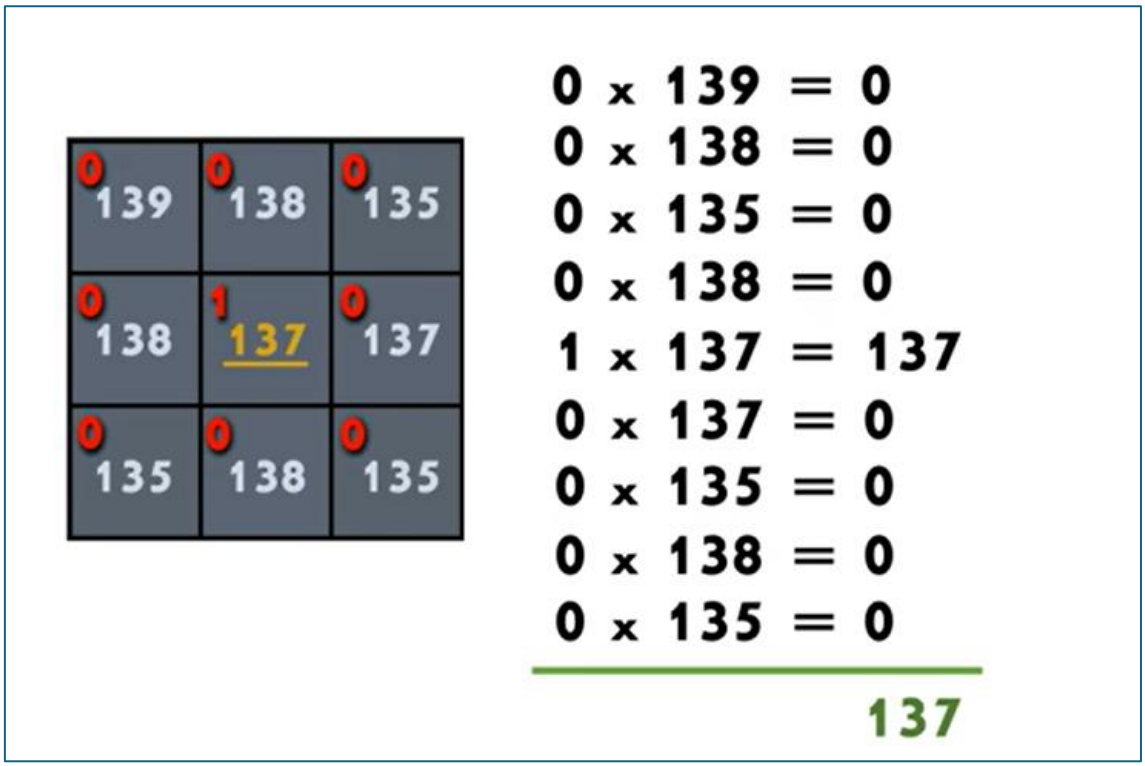
Nosotros ignoraremos los pixeles que bordean la imagen y comenzaremos el proceso con el píxel de valor 137, considerando este píxel como si fuera el primero:



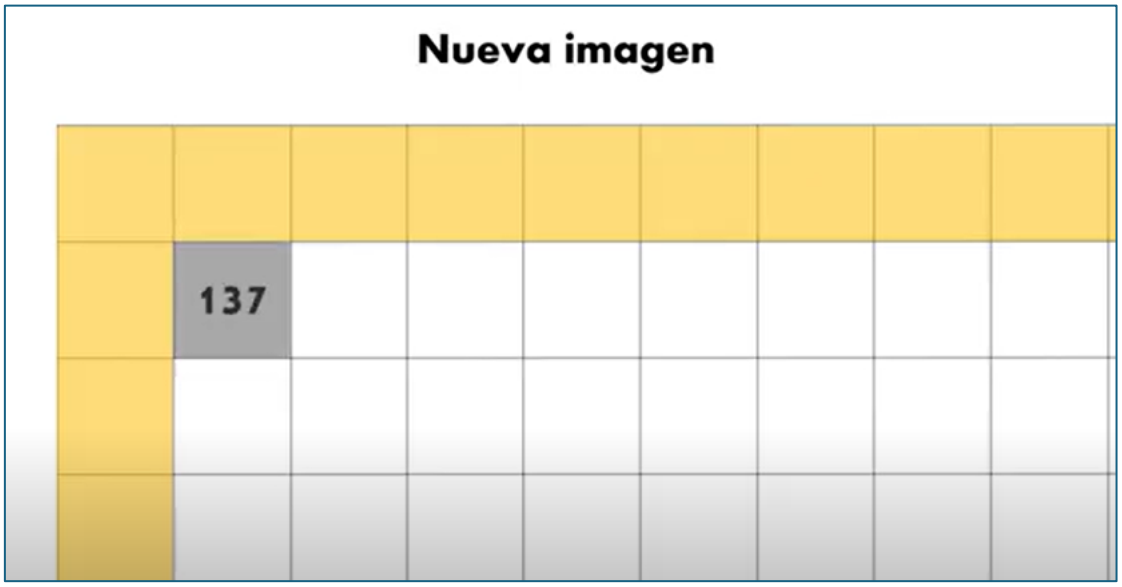
Ahora colocamos el núcleo con el píxel principal en el valor 137.



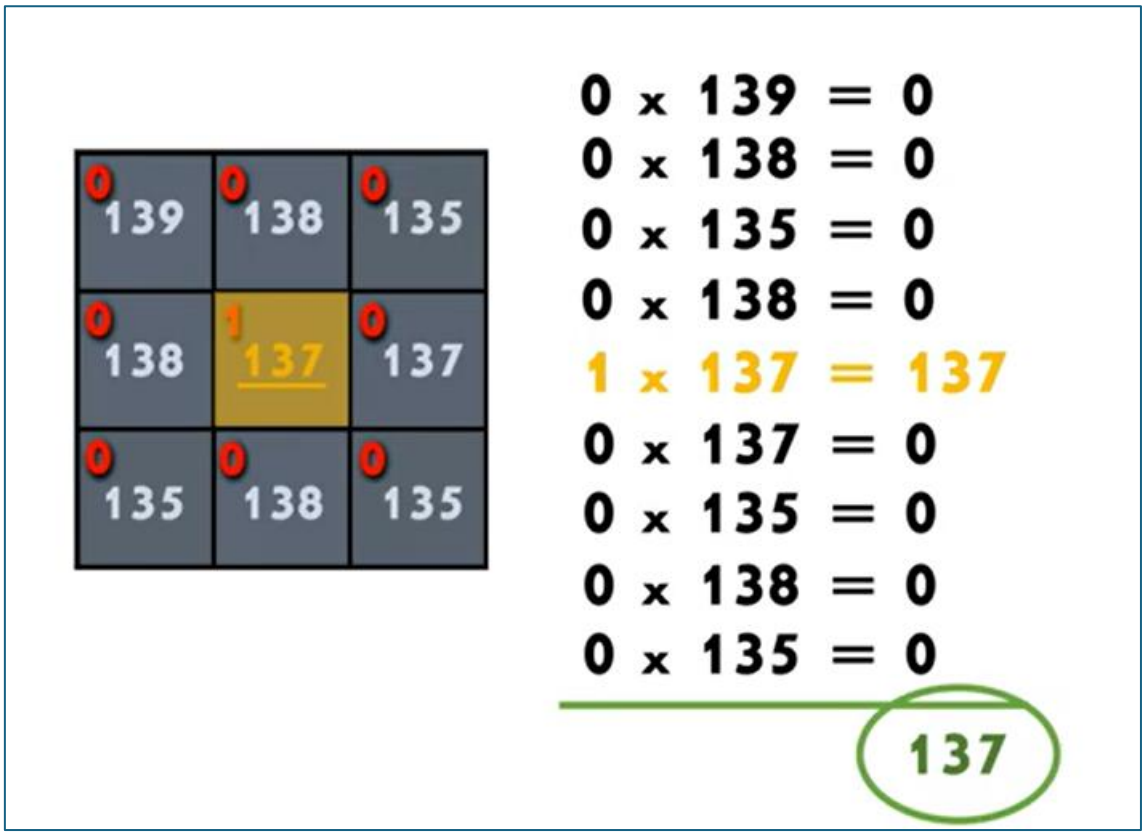
Por cada casilla, multiplicaremos el valor numérico del núcleo (color rojo) por el valor del píxel en escala de grises (que es un número entre 0 y 255) para finalmente, sumar esos productos:



El resultado en un número que pondremos en una **nueva imagen**, y en la misma posición que el píxel que estábamos revisando:

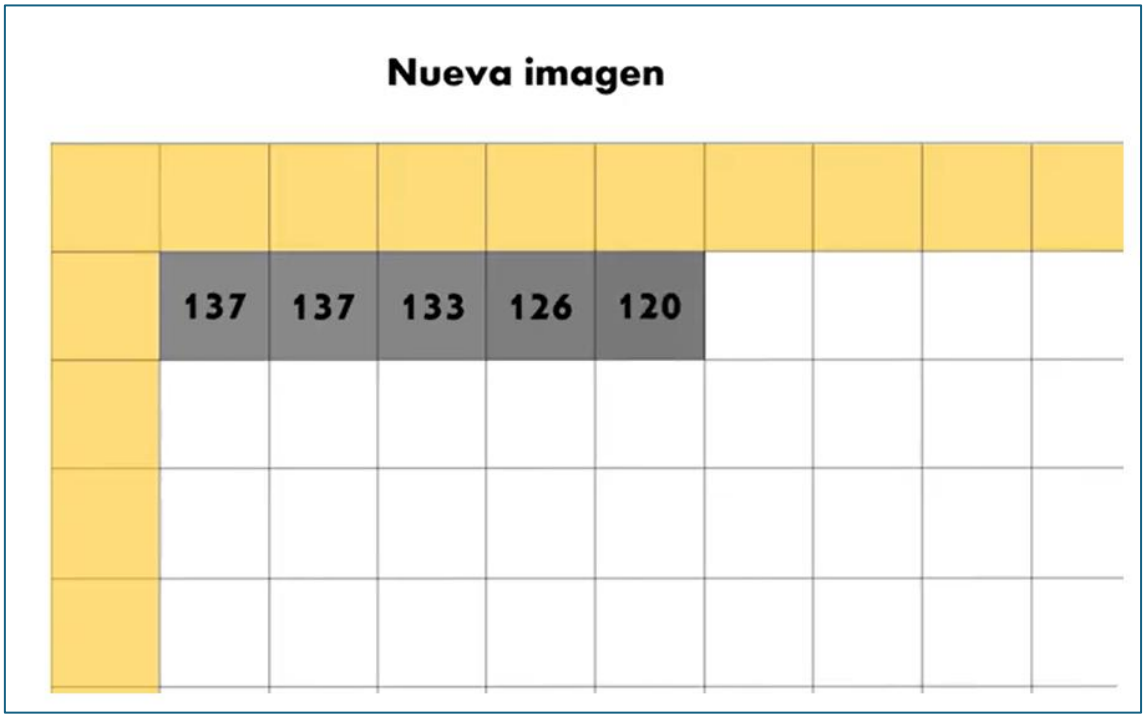


Como el núcleo tiene un valor 0 en las casillas que lo bordean, esto significa que sus valores no están siendo considerados. Es decir, en la nueva imagen, sólo se toma como información la que nos proporciona el píxel principal.

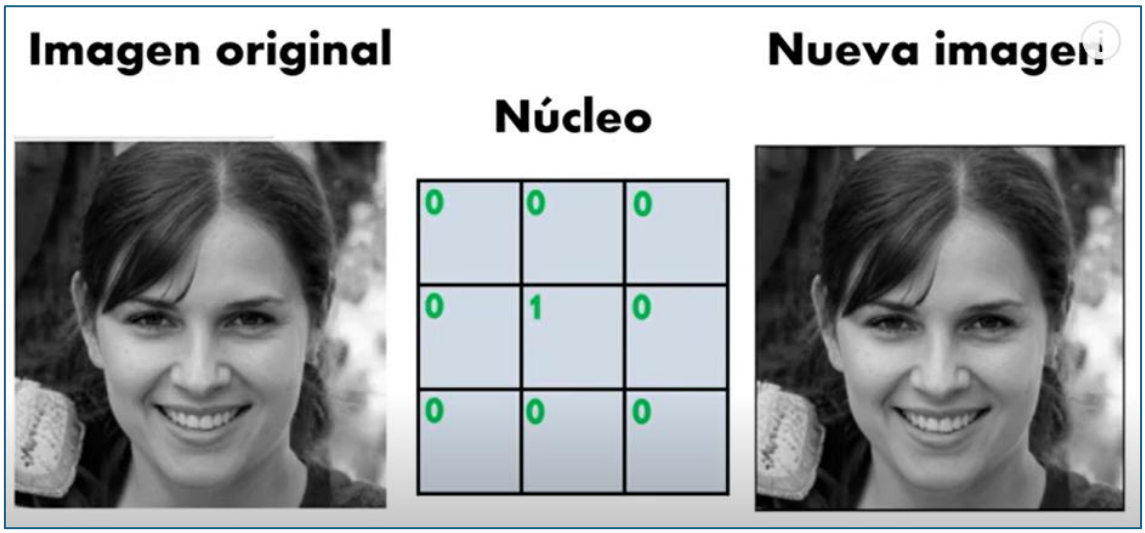


Ahora haremos esta operación con todos y cada uno de los pixeles. Uno por uno y así sucesivamente.

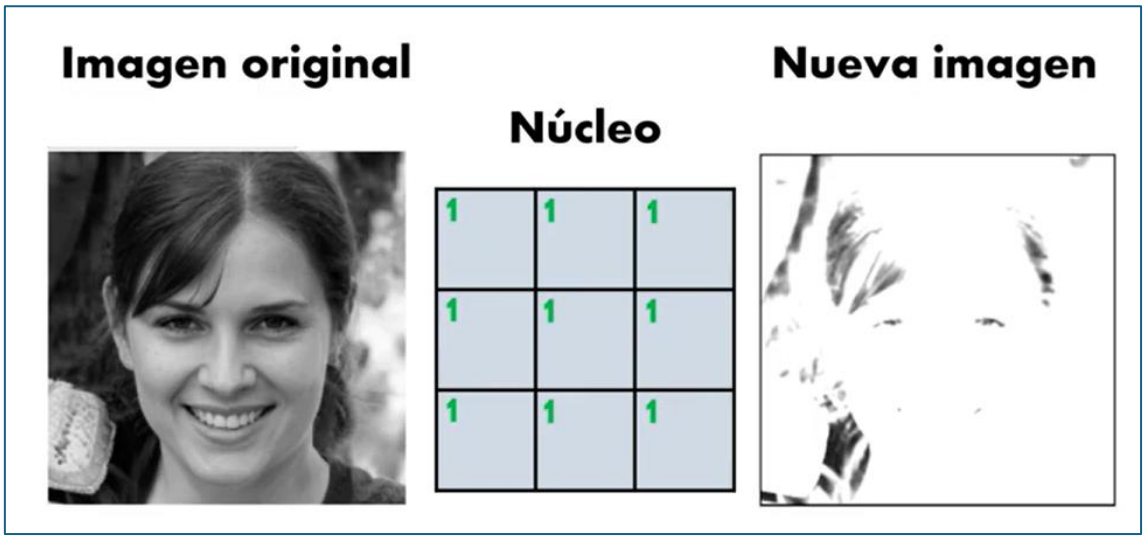
Los resultados de esas operaciones las colocaremos en la nueva imagen, también uno por uno:



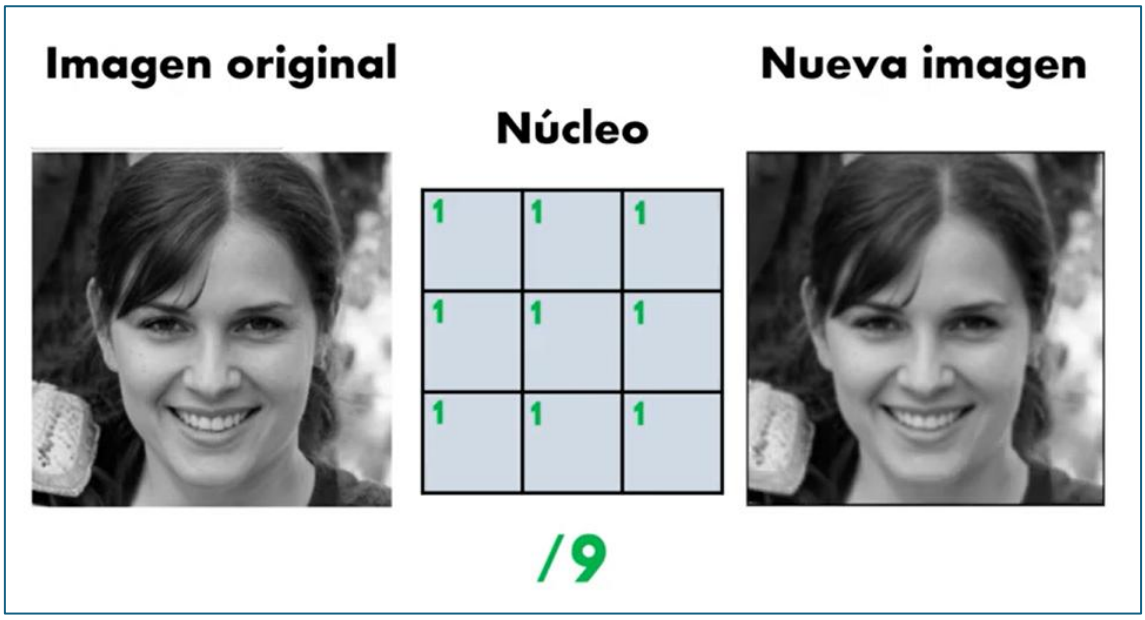
¿Qué estamos obteniendo? el resultado, aplicando ese núcleo, no es más que la imagen de partida.



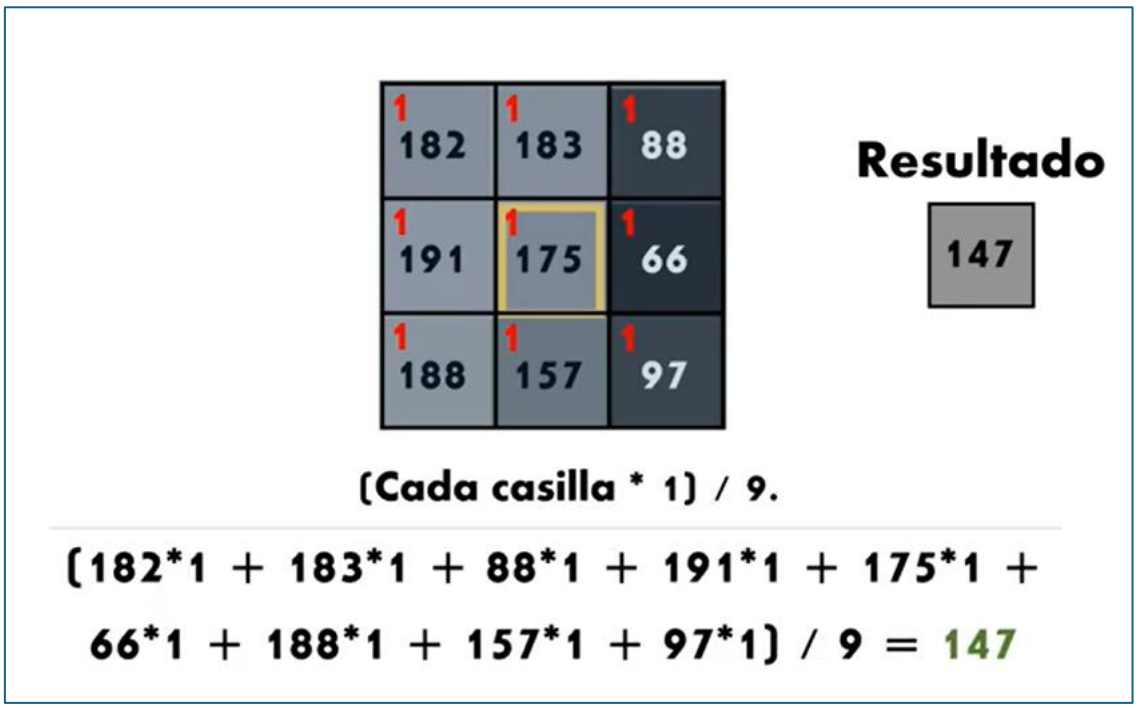
Y si cambiamos los valores numéricos del kernel de modo que cada casilla tenga el valor 1. Cada píxel de la nueva imagen se verá afectado por los pixeles que lo bordean y obtenemos una imagen muy muy desenfocada.



Pero si dividimos el resultado entre los 9 valores que tenemos en el kernel, obtenemos un desenfoque más suave. Estamos usando un filtro de desenfoque:



Al poner un 1 en todos los pixeles del kernel, le estamos diciendo que todos los pixeles son igualmente importantes que el píxel principal. Al hacer una media ponderada (sumando todos y dividiéndolos entre 9) conseguimos difuminar la imagen.



Por lo tanto, al poner el valor del cálculo en la nueva imagen, como se hace píxel por píxel, cada nuevo píxel se vuelve un promedio de todos los que tiene alrededor y conseguimos difuminar los bordes de la imagen y esta pierde claridad y el resultado es una imagen un poco más desenfocada:

El proceso de iterar un núcleo a través de cada píxel de una imagen (multiplicar los valores, sumarlos y pasarlos a una nueva imagen), recibe el nombre de **convolución**.

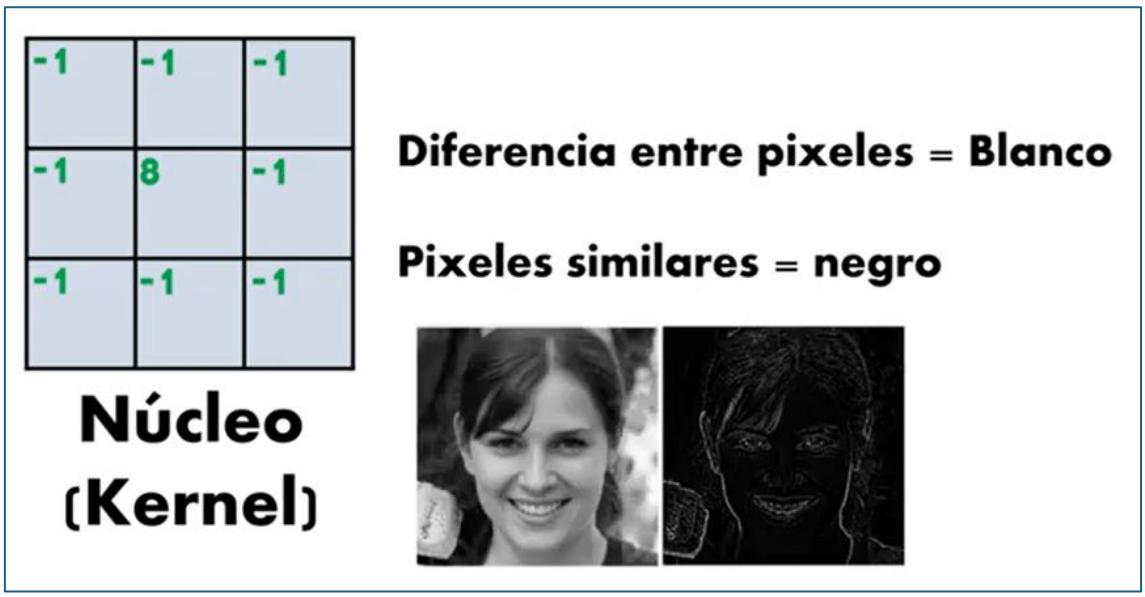
¿Cómo podemos usar las convoluciones para detectar ejes?

Como el resultado de las operaciones siempre lo ponemos en una nueva imagen. Si queremos detectar los ejes, necesitamos que cuando exista un eje se pasen pixeles muy blancos y cuando no hay ejes, se pasen pixeles muy negros:

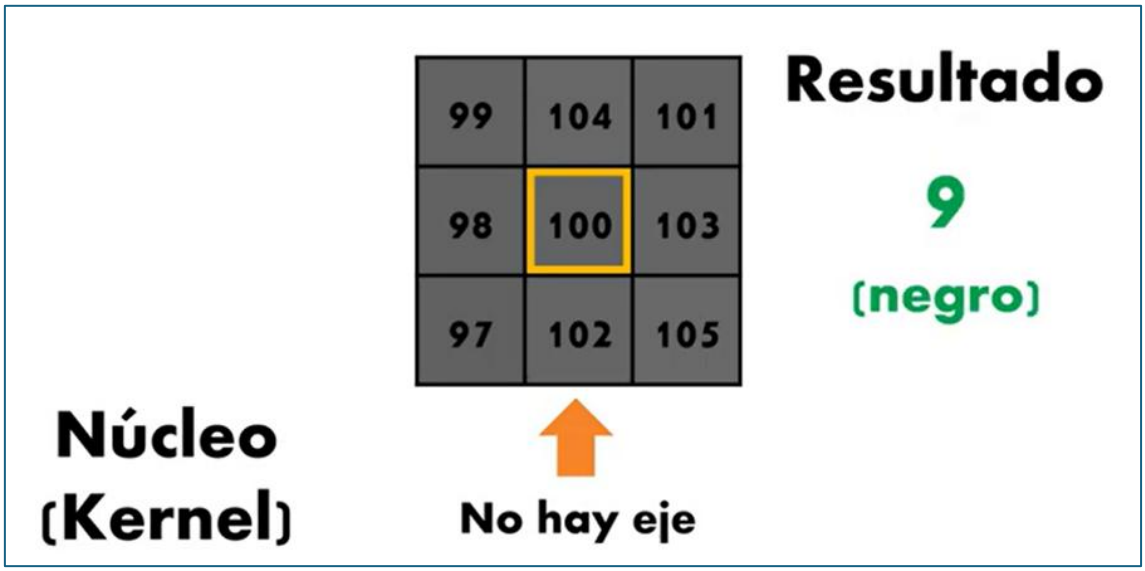
Imaginemos este núcleo:

-1	-1	-1
-1	8	-1
-1	-1	-1

Como debemos ejecutarlo píxel a píxel, lo que hará es que cuando el resultado del cálculo sea **muy alto** (que ocurrirá cuando tengamos mucha diferencia entre el píxel central y los que lo bordean) **lo tomará como blanco**, y cuando el resultado sea **muy pequeño** (que ocurrirá cuando los pixeles sean muy similares), **lo tomará como negro**. Un resultado alto significa blanco y un resultado pequeño, negro.

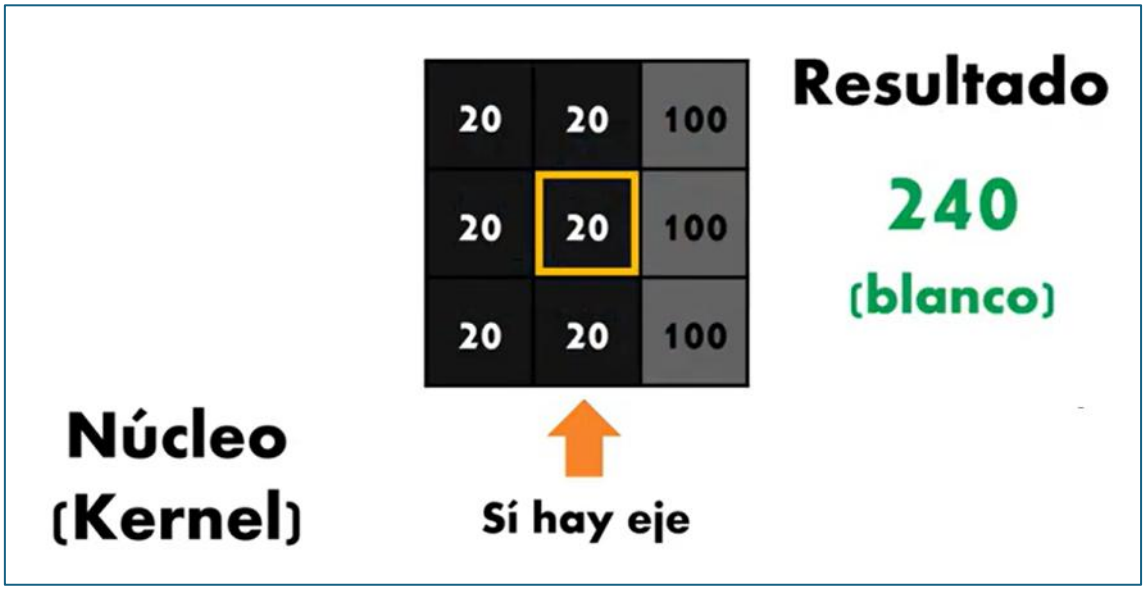


Si los valores de los pixeles son muy similares, el resultado es negro:



-99-104-101-103-105-102-97-98+(100x8)=-9

Y si son diferentes, el resultado es blanco:



-20-20-100-100-100-20-20-20+(8x20)=-240

Este núcleo aplicado de forma iterativa, con esta simple multiplicación se asegura que, en cada convolución, la salida nos aporte un valor numérico muy bajo o muy alto. Si aplicamos el núcleo a la imagen nos da como resultado la siguiente:

Podemos probarlo en vivo en los siguientes links:

- Con una imagen; https://ringa-tech.com/vision01/imagen.html
- Con nuestra cámara web: https://ringa-tech.com/vision01/camara.html

Hasta ahora hemos desenfocado, pero podemos hacer lo contrario y enfocar. El resultado sería el siguiente:

Filtro de enfoque

Enfoque

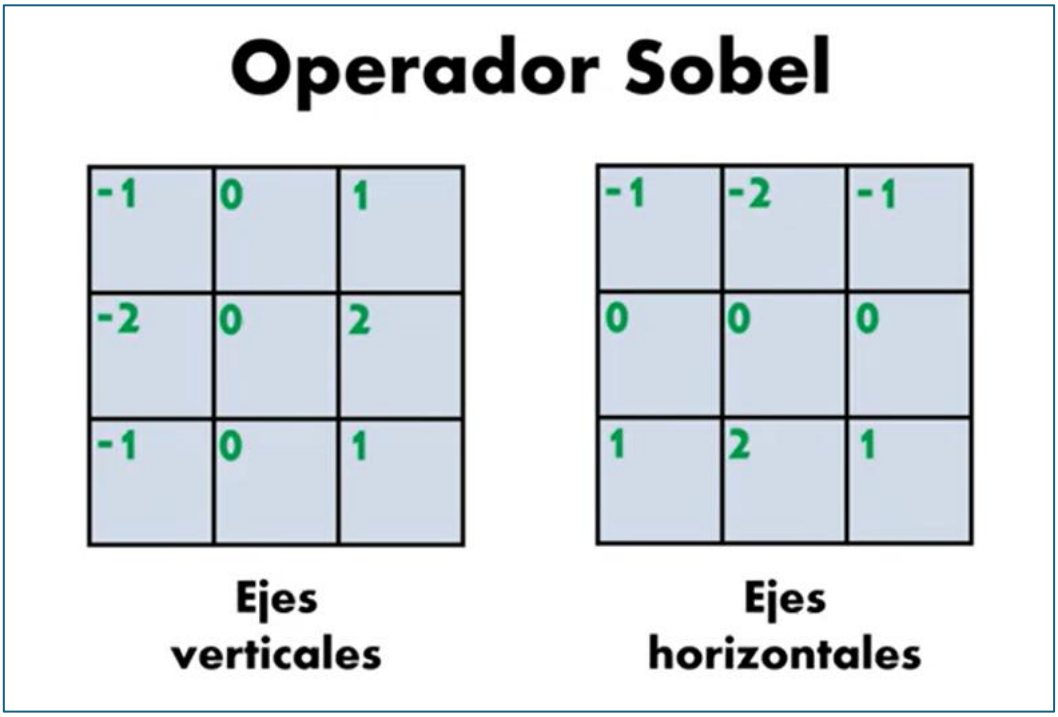
$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

Y también podemos realzar

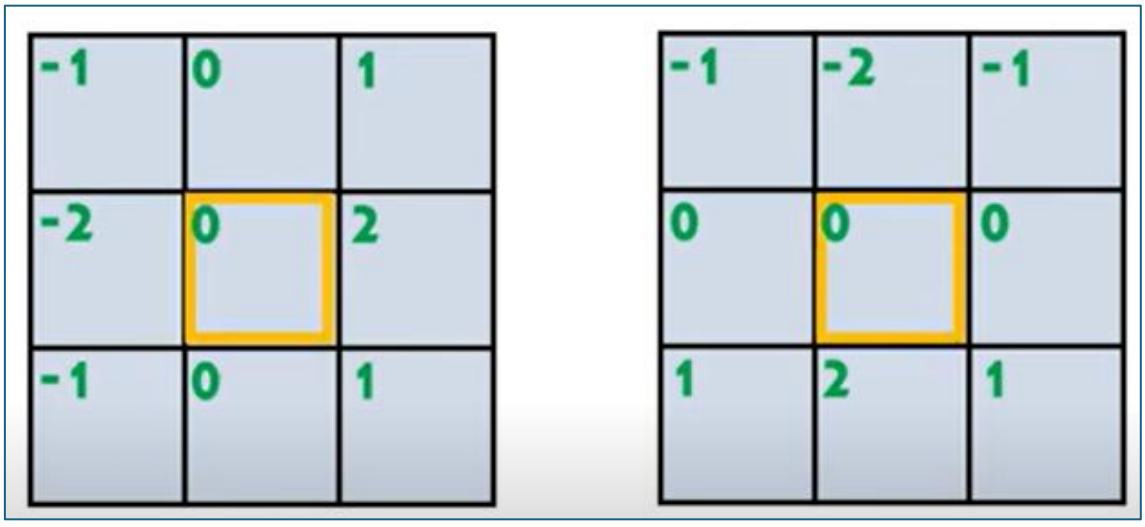
Los kernels (o matrices de convolución) pueden ser matrices de 3x3, de 5x5, etc.

Como los ejes son muy importantes en las redes neuronales convolucionales, se ha seguido trabajando en la **búsqueda de matrices que los detecten mejor**. Hay unos detectores de ejes que son muy eficientes y son el operador Sobel y el algoritmo de Canny. El algoritmo de Canny es una mejora de Sobel en la detección de bordes.

Sobel se basa en el uso de núcleos, pero en lugar de usar sólo uno, utiliza estos dos núcleos:

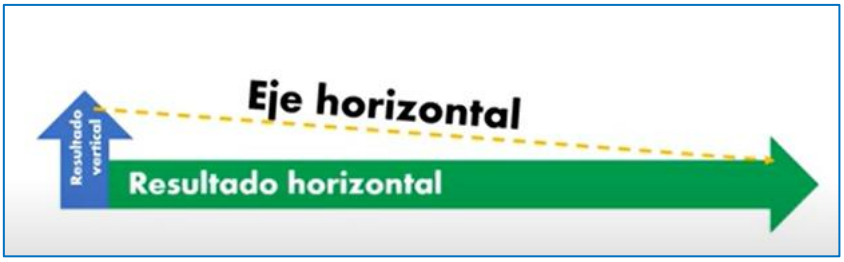


Uso se especializa en detectar ejes verticales y el otro en detectar ejes horizontales. Si nos damos cuenta, funcionan un poco diferente a los anteriores porque ignoran el píxel principal:

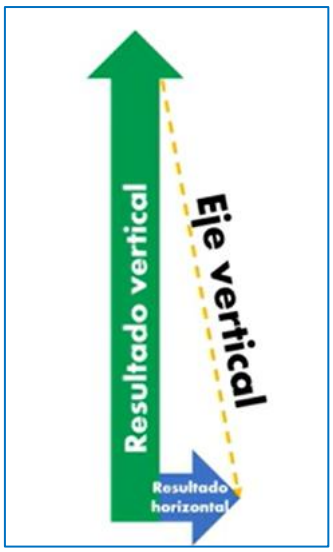


Con estos núcleos tenemos por separado el resultado del filtro vertical y del horizontal y, por lo tanto, sabemos, por cada píxel, que valor se le otorga al resultado (convolución) horizontal y al vertical:

Si el píxel resulta ser un eje horizontal, obtendrá un valor grande en ese kernel y pequeño en el vertical. Y tomarán valores positivos o negativos según se desplacen a la derecha o izquierda.



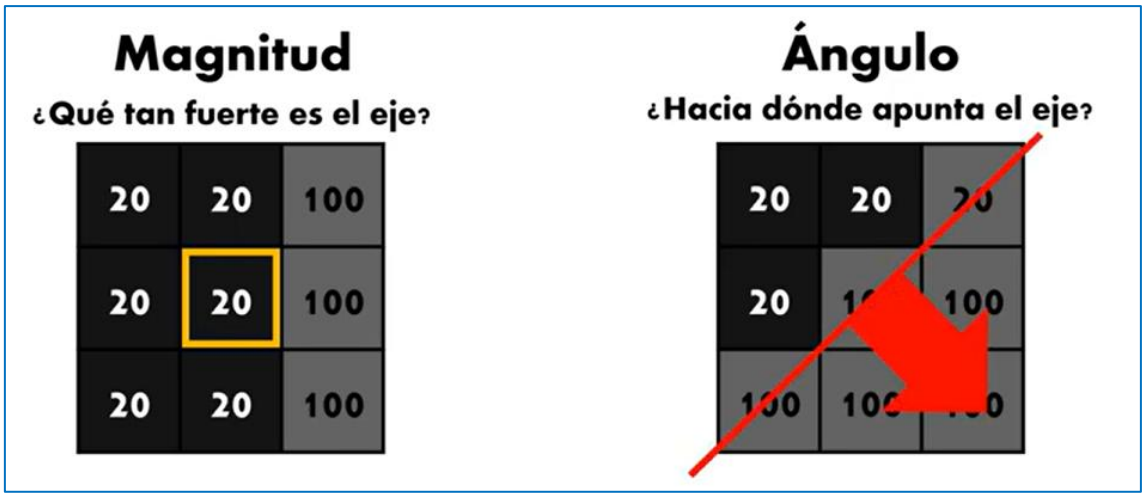
Pero si el píxel forma parte de un eje vertical, será pequeña la componente horizontal y muy grande la vertical:



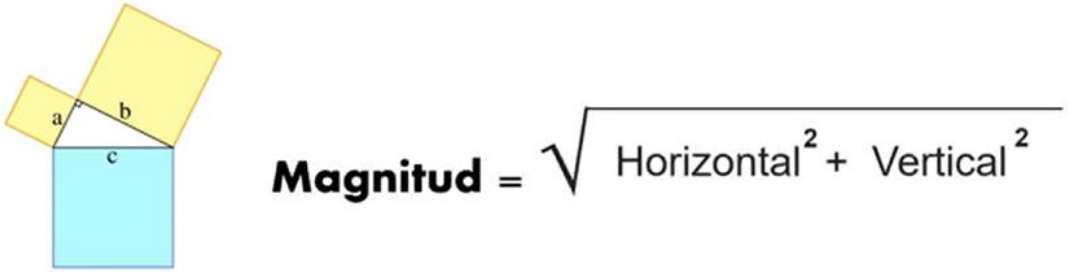
En ejes inclinados, ambos componentes tendrán un valor grande:

Por lo tanto, viendo los resultados de los dos ejes podemos saber si se trata de un eje horizontal, vertical o inclinado y conocer hacia qué lado apunta el eje. Y esto, píxel por píxel.

La detección de los ejes se reconoce atendiendo a los valores numéricos de sus dos parámetros principales, que son **la magnitud y el ángulo**. El valor numérico de la magnitud nos dirá qué fuerte es el eje y el ángulo nos dirá hacia donde apunta.

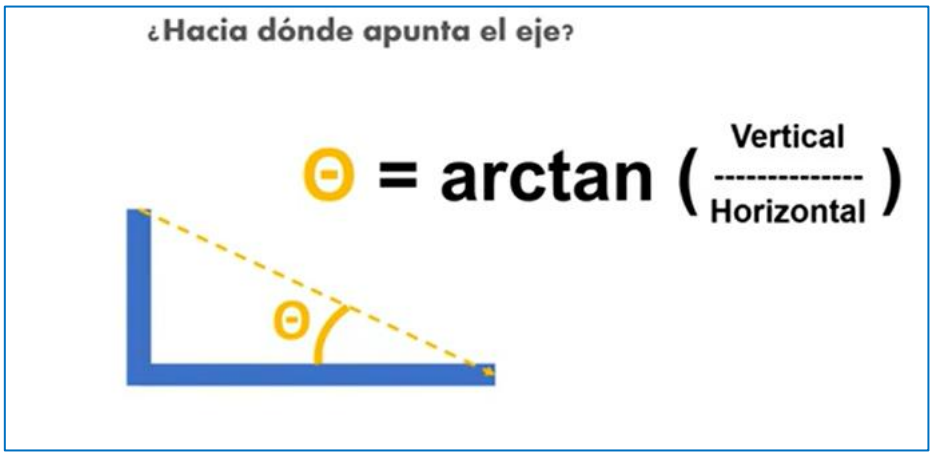


La magnitud la podemos calcular usando el teorema de Pitágoras. Pensar que conocemos los catetos, que son las componentes horizontal y vertical.



Teorema de Pitágoras

También podemos saber cuál es el ángulo o la orientación del eje usando trigonometría, con la función inversa a la tangente de un ángulo:



Es decir, con estos dos parámetros podremos definir mejor qué es un eje o un borde de una imagen.

Las redes neuronales convolucionales (CNN) utilizan filtros convolucionales para extraer o conocer los objetos que componen la imagen.

Un ejemplo de redes convolucionales es la clasificación de imágenes. Esa red, tomará una imagen como entrada y le aplica una serie de filtros convolucionales para detectar características como bordes, texturas y formas en la imagen. Estas características se combinan en capas posteriores para realizar la clasificación final de la imagen en una de clases predefinidas, como por ejemplo perro, gato, ave, etc.

Vamos a entrenar un clasificador multiclase con una red neuronal convolucional para clasificar el *dataset MINIST digits*. Los pasos, en Colab con Python podrían ser los siguientes:

1. Importamos las librerías necesarias

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten,
Dense #Para poder implementar nuestras capas convolucionales
from tensorflow.keras.datasets import mnist
from tensorflow.keras.utils import to_categorical
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix,
ConfusionMatrixDisplay
import numpy as np
```

2. Cargamos el dataset

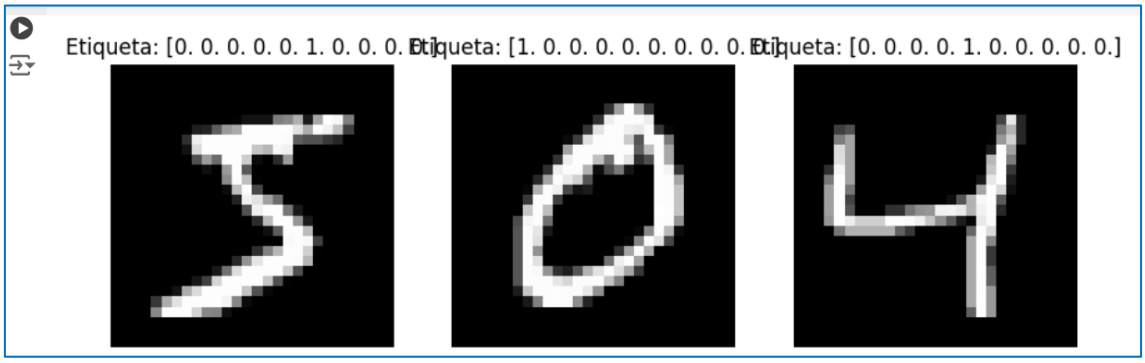
```
# Cargar el dataset MNIST
(x_train, y_train), (x_test, y_test) = mnist.load_data()
#Podemos ver cuántos datos son y como son:
print(x_train.shape)
print(y_train.shape)
print(x_test.shape)
print(y_test.shape)
```

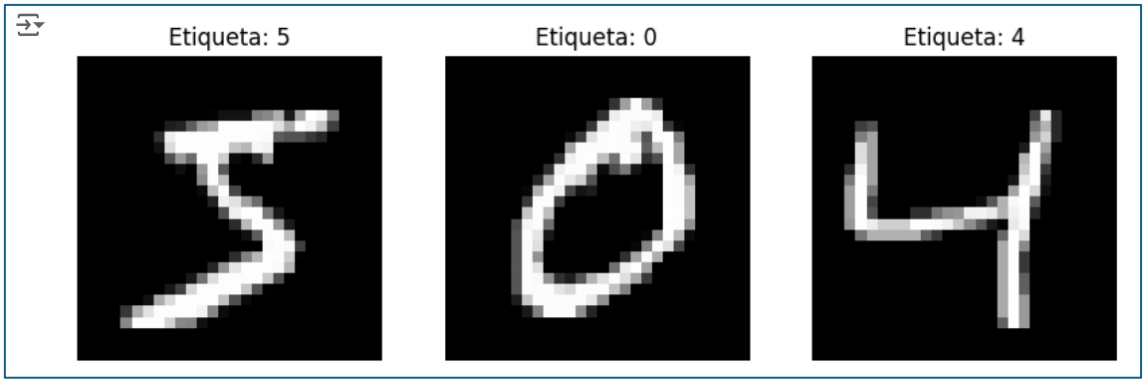
```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 ━━━━━━━━━━━━━━━━━━━━ 2s 0us/step
(60000, 28, 28)
(60000,)
(10000, 28, 28)
(10000,)
```

Hay 60000 imágenes en los datos de entrenamiento o *train* y 10000 en los datos de *test*, y cada una de 28x28.

3. Mostramos algunos datos de entrenamiento

```
# Mostrar algunos datos de entrenamiento
plt.figure(figsize=(10, 10))
for i in range(9):
    plt.subplot(3, 3, i + 1)
    plt.imshow(x_train[i], cmap='gray')
    plt.title(f"Etiqueta: {y_train[i]}")
    plt.axis('off')
plt.show()
```





4. Preprocesamiento de datos: Normalización

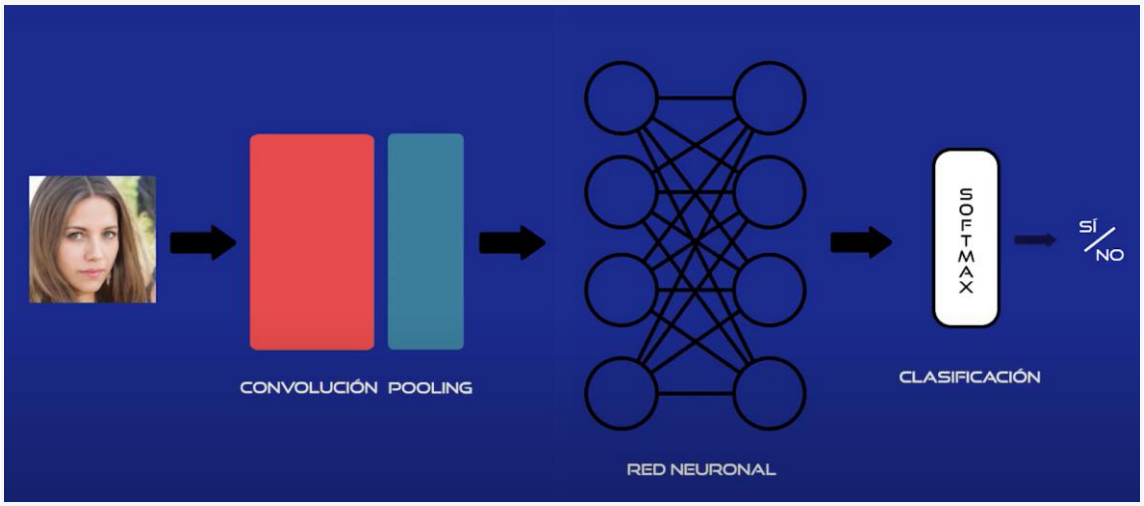
```
# Preprocesamiento de datos
x_train = x_train.reshape((x_train.shape[0], 28, 28,
1)).astype('float32') / 255
x_test = x_test.reshape((x_test.shape[0], 28, 28,
1)).astype('float32') / 255

y_train = to_categorical(y_train, 10) # Por ejemplo, la etiqueta 3
se convierte en [0, 0, 0, 1, 0, 0, 0, 0, 0, 0]
y_test = to_categorical(y_test, 10)
```

Los datos originales del conjunto MNIST (x_train y x_test) tienen dimensiones (60000, 28, 28) y (10000, 28, 28) respectivamente, porque son imágenes en escala de grises de 28x28 píxeles. Aquí, añadimos un canal adicional (1) para indicar que son imágenes monocromáticas. El formato resultante será (número de ejemplos, alto, ancho, canales), que es el formato esperado por los modelos de TensorFlow/Keras para datos de imágenes.

Los valores de los píxeles originalmente están entre 0 y 255 (valores de intensidad en escala de grises). Dividir entre 255 escala los valores entre 0 y 1, lo que ayuda a estabilizar el entrenamiento del modelo y permite que las operaciones matemáticas (como derivadas y actualizaciones de pesos) sean más eficientes. Y esto se conoce como normalización.

5. Definimos el modelo

```
# Definir el modelo
model = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)),
    MaxPooling2D((2, 2)),
    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    Flatten(),
    Dense(128, activation='relu'),
    Dense(10, activation='softmax')
])
```



Por lo general, existen varias capas de convolución (para detectar diferentes patrones) y pooling (redimensionado):

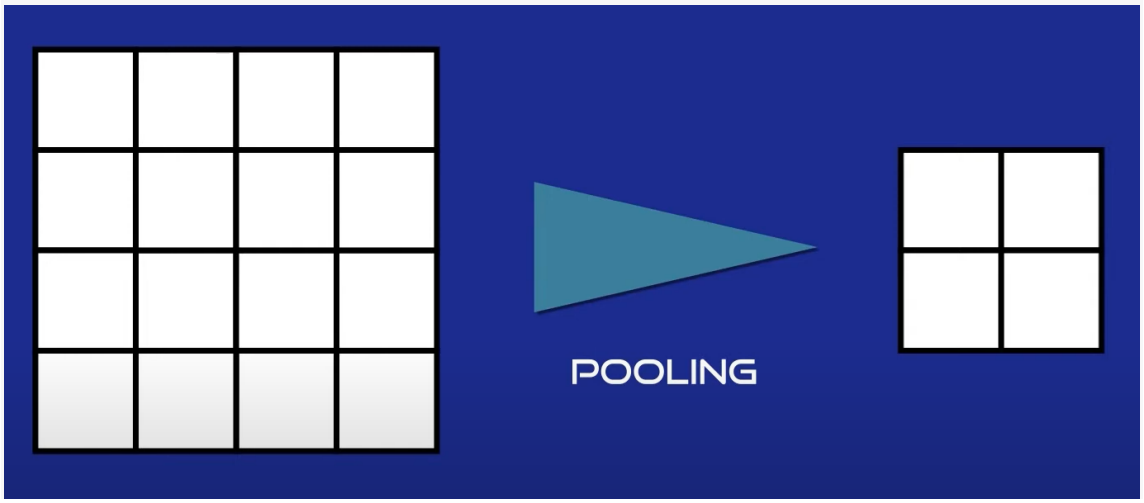
En nuestro modelo, de forma literal, la estructura de capas es la siguiente:

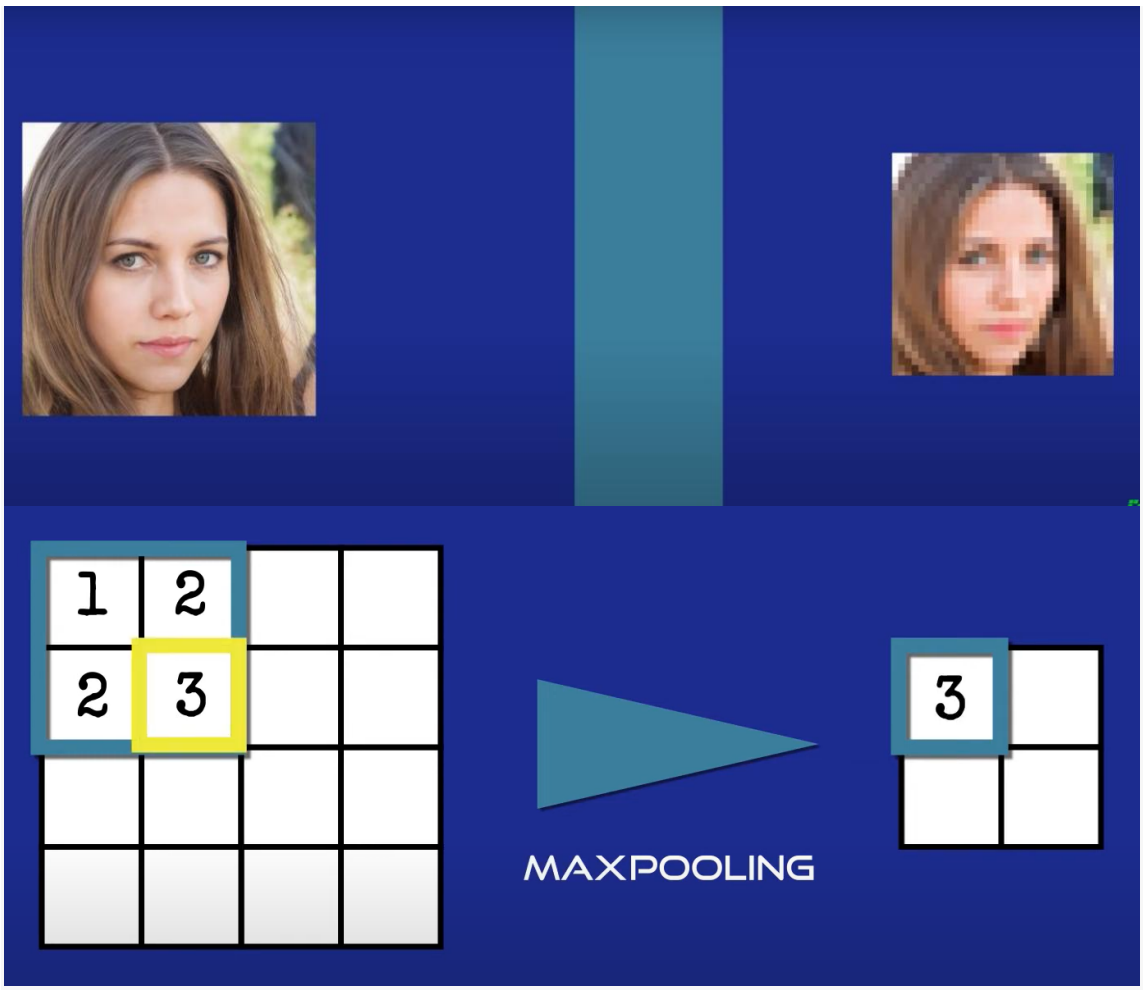
```
# Definir el modelo
model = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)),
    MaxPooling2D((2, 2)),
    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    Flatten(),
    Dense(128, activation='relu'),
    Dense(10, activation='softmax')
])
```

Cada bloque de la imagen corresponde a una capa en el modelo, y la información en cada bloque incluye:

1. **Nombre de la capa y tipo**:
    - conv2d (Convolutional Layer): Capa convolucional para trabajar con imágenes en 2D y que detecta características espaciales como bordes o texturas.
    - max_pooling2d (MaxPooling Layer): Capa de agrupamiento que reduce la dimensionalidad para mejorar la eficiencia y evitar el sobreajuste.

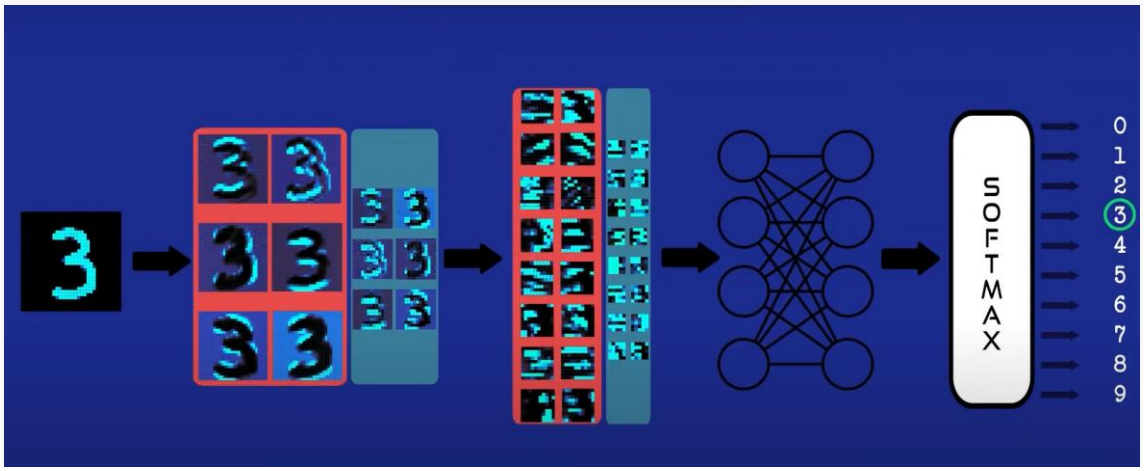
El pooling podemos entenderlo como una redimensión de la imagen, reduciendo su resolución):





El maxpooling permite analizar el contenido de una región de la imagen por bloques y elige en número más alto, es decir, preserva la información más relevante, que corresponderá a un píxel en la imagen resultante:

- flatten (Flatten Layer): Convierte las características 2D en un vector 1D para pasarlo a las capas densas.



Convertimos las imágenes en vectores y eso se conoce como flattering.

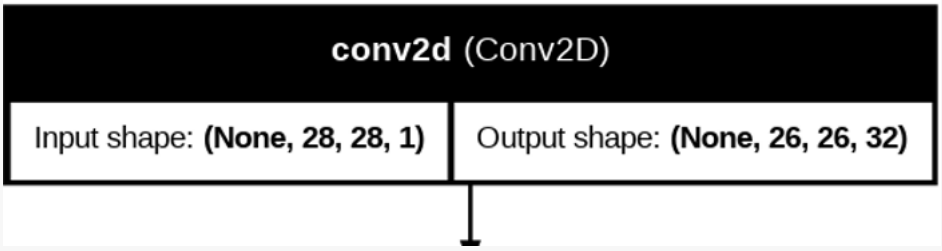
- dense (Fully Connected Layer): Capas densas donde se toman las características y se procesan para generar predicciones.

2. **Input Shape (Forma de entrada)**:
   - Muestra cómo llegan los datos a la capa (dimensiones de las imágenes o características).
3. **Output Shape (Forma de salida)**:
   - Describe las dimensiones de los datos después de ser procesados por esa capa.

Detalles del flujo del modelo:

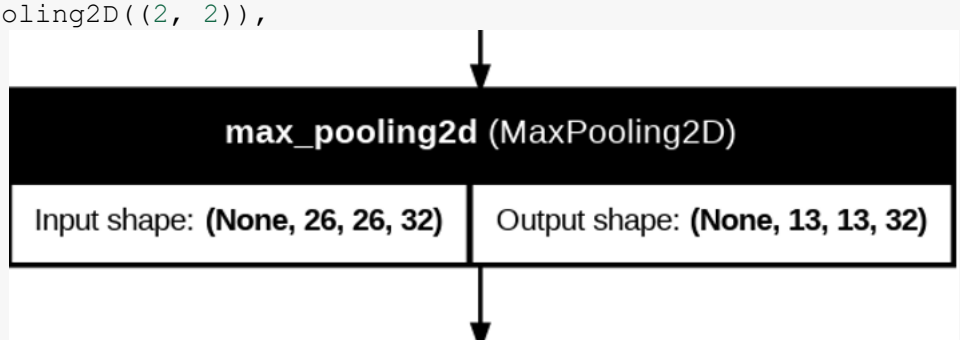
- **Primera capa (conv2d)**: Procesa imágenes de entrada 2D de tamaño (28, 28, 1) (28x28 píxeles, escala de grises). **Produce 32 filtros de tamaño (26, 26, 32**), reduciendo la dimensión debido a la convolución. El tamaño del kernel es una matriz de 3x3

```
Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)),
```



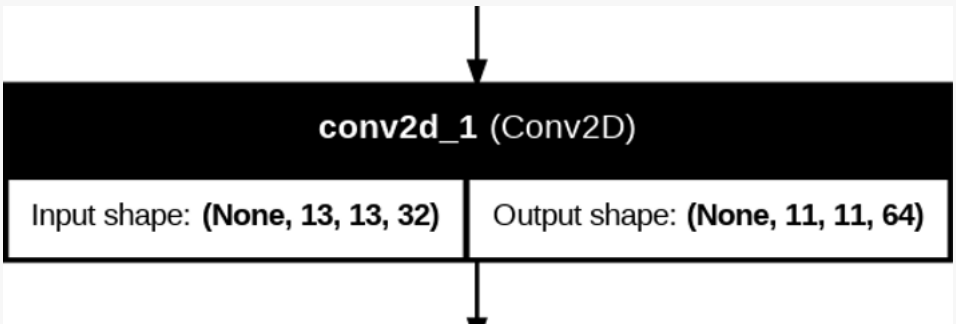
- **Pooling (max_pooling2d)**: Esta capa de agrupamiento reduce las dimensiones a la mitad, pasando a (13, 13, 32). Sólo toma el pixel con mayor valor de la matriz de 2x2.

```
MaxPooling2D((2, 2)),
```



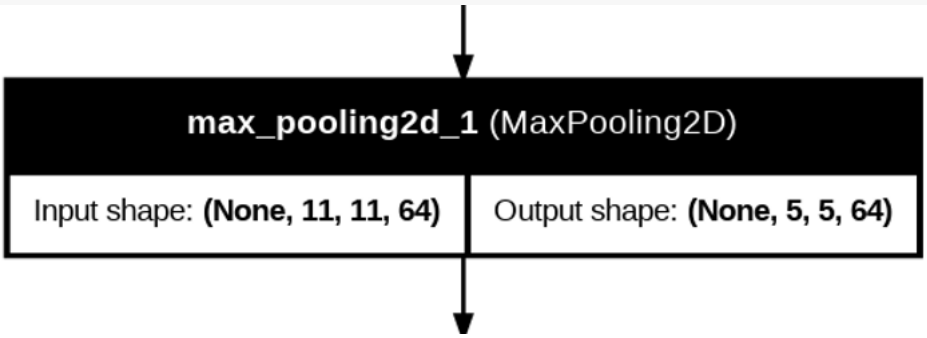
- **Segunda capa convolucional (conv2d_1)**:Como la imagen es más pequeña, en esta convolución podemos poner más filtros. En este caso le hemos puesto 64 filtros y su kernel es también de 3x3 **Genera 64 filtros de características de tamaño (11, 11, 64**).

```
Conv2D(64, (3, 3), activation='relu'),
```



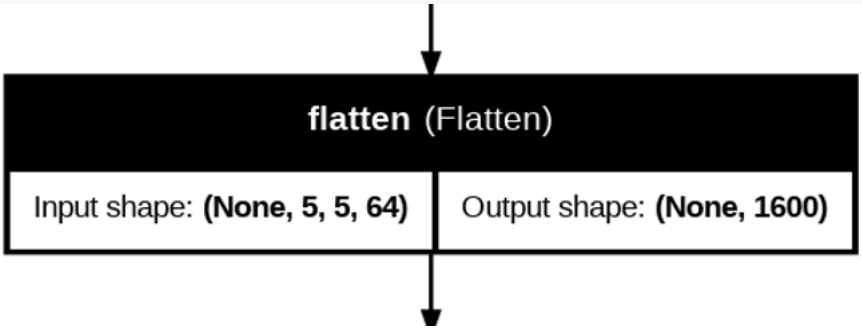
- **Segunda capa de pooling (max_pooling2d_1)**: Nuevamente reduce las dimensiones de (11,11,64) a la mitad, obteniendo imágenes de **(5, 5, 64)**. Sólo toma el pixel de mayor valor en una matriz de 2x2.

```
MaxPooling2D((2, 2)),
```



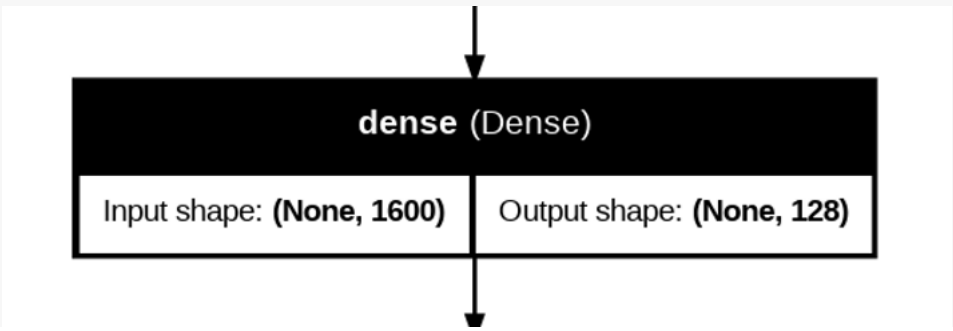
- **flatten**: Convierte la salida (5,5,64) en un vector unidimensional de tamaño 1600= 5x5x64

```
Flatten(),
```



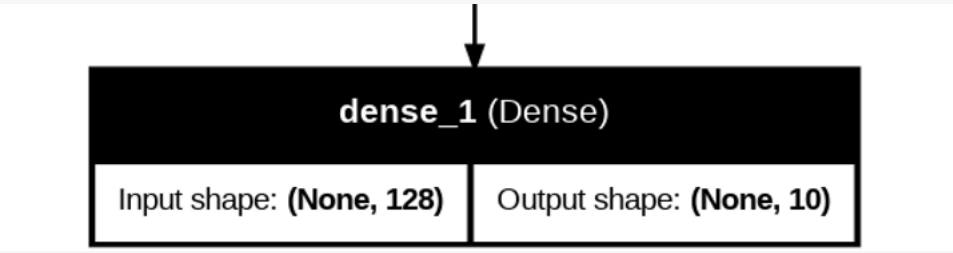
- **Capa densa (dense)**: Procesa las características reducidas y produce una representación de 128 neuronas. Podríamos usar 100 neuronas en lugar de 128, si quisiéramos.

```
Dense(128, activation='relu'),
```



- **Última capa densa (dense_1)**: Genera 10 neuronas, una por cada clase, con activación softmax para obtener probabilidades.

```
Dense(10, activation='softmax')
```



6. Compilamos el modelo

```
# Compilar el modelo
model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

7. Entrenamos el modelo

```
# Entrenar el modelo
history = model.fit(x_train, y_train, epochs=10, batch_size=32,
validation_split=0.2)
```

```
Epoch 4/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 4s 3ms/step - accuracy: 0.9919 - loss: 0.0251 - val_accuracy: 0.9878 - val_loss: 0.0458
Epoch 5/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.9951 - loss: 0.0150 - val_accuracy: 0.9884 - val_loss: 0.0465
Epoch 6/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 4s 3ms/step - accuracy: 0.9957 - loss: 0.0125 - val_accuracy: 0.9881 - val_loss: 0.0478
Epoch 7/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.9967 - loss: 0.0096 - val_accuracy: 0.9897 - val_loss: 0.0410
Epoch 8/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 4s 3ms/step - accuracy: 0.9979 - loss: 0.0069 - val_accuracy: 0.9899 - val_loss: 0.0467
Epoch 9/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 4s 3ms/step - accuracy: 0.9978 - loss: 0.0064 - val_accuracy: 0.9889 - val_loss: 0.0525
Epoch 10/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.9979 - loss: 0.0068 - val_accuracy: 0.9891 - val_loss: 0.0540
```

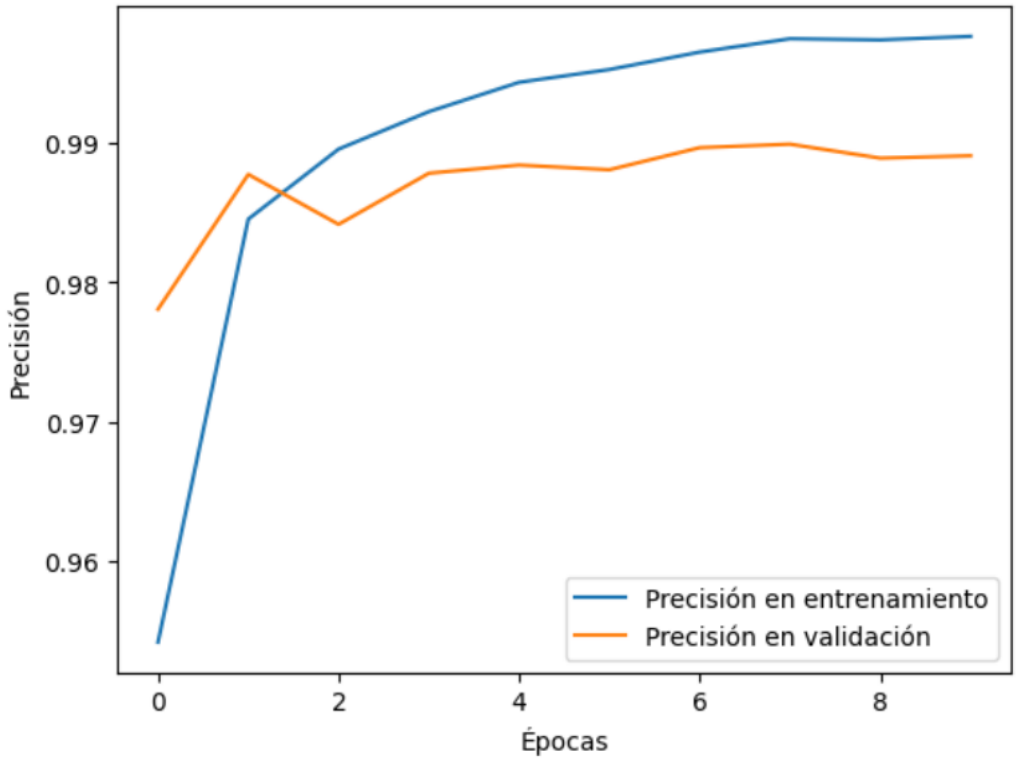
8. Evaluar el modelo.

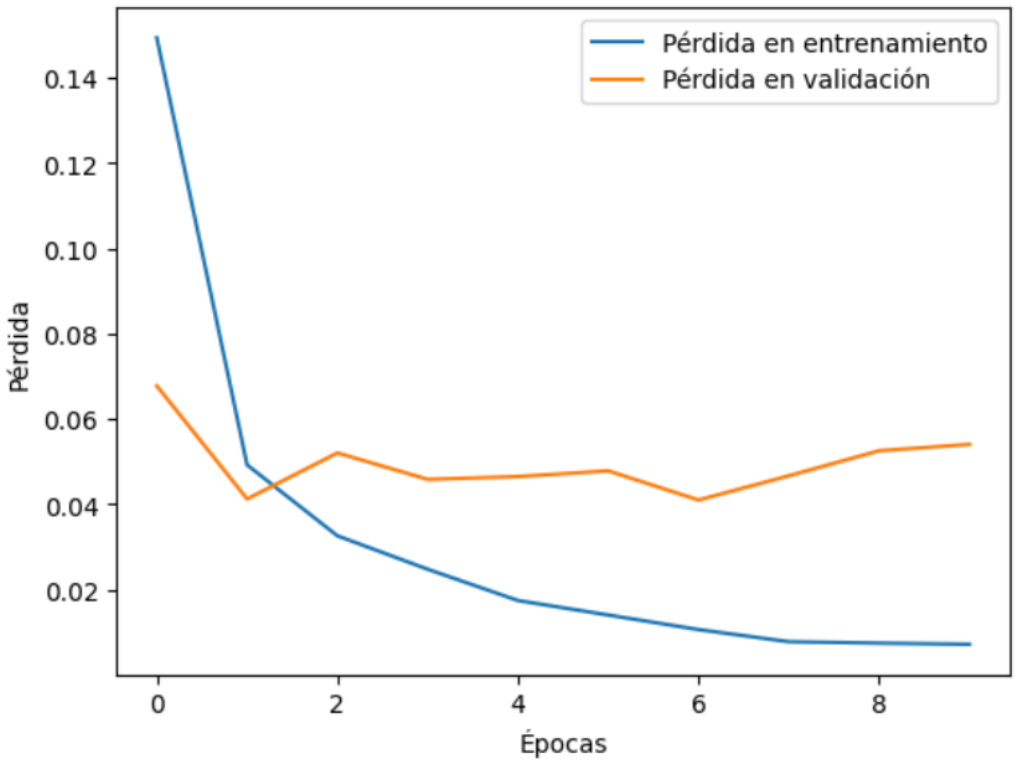
```
# Evaluar el modelo
test_loss, test_acc = model.evaluate(x_test, y_test)
print(f"\nPérdida en el conjunto de prueba: {test_loss}")
print(f"Precisión en el conjunto de prueba: {test_acc}")
```

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9857 - loss: 0.0643

Pérdida en el conjunto de prueba: 0.05203856900334358
Precisión en el conjunto de prueba: 0.9886000156402588
```

9. Graficar las métricas: precisión y pérdidas

```
# Graficar las métricas
import matplotlib.pyplot as plt

# Precisión
plt.plot(history.history['accuracy'], label='Precisión en
entrenamiento')
plt.plot(history.history['val_accuracy'], label='Precisión en
validación')
plt.xlabel('Épocas')
plt.ylabel('Precisión')
plt.legend()
plt.show()

# Pérdida
plt.plot(history.history['loss'], label='Pérdida en entrenamiento')
plt.plot(history.history['val_loss'], label='Pérdida en
validación')
plt.xlabel('Épocas')
plt.ylabel('Pérdida')
plt.legend()
plt.show()
```





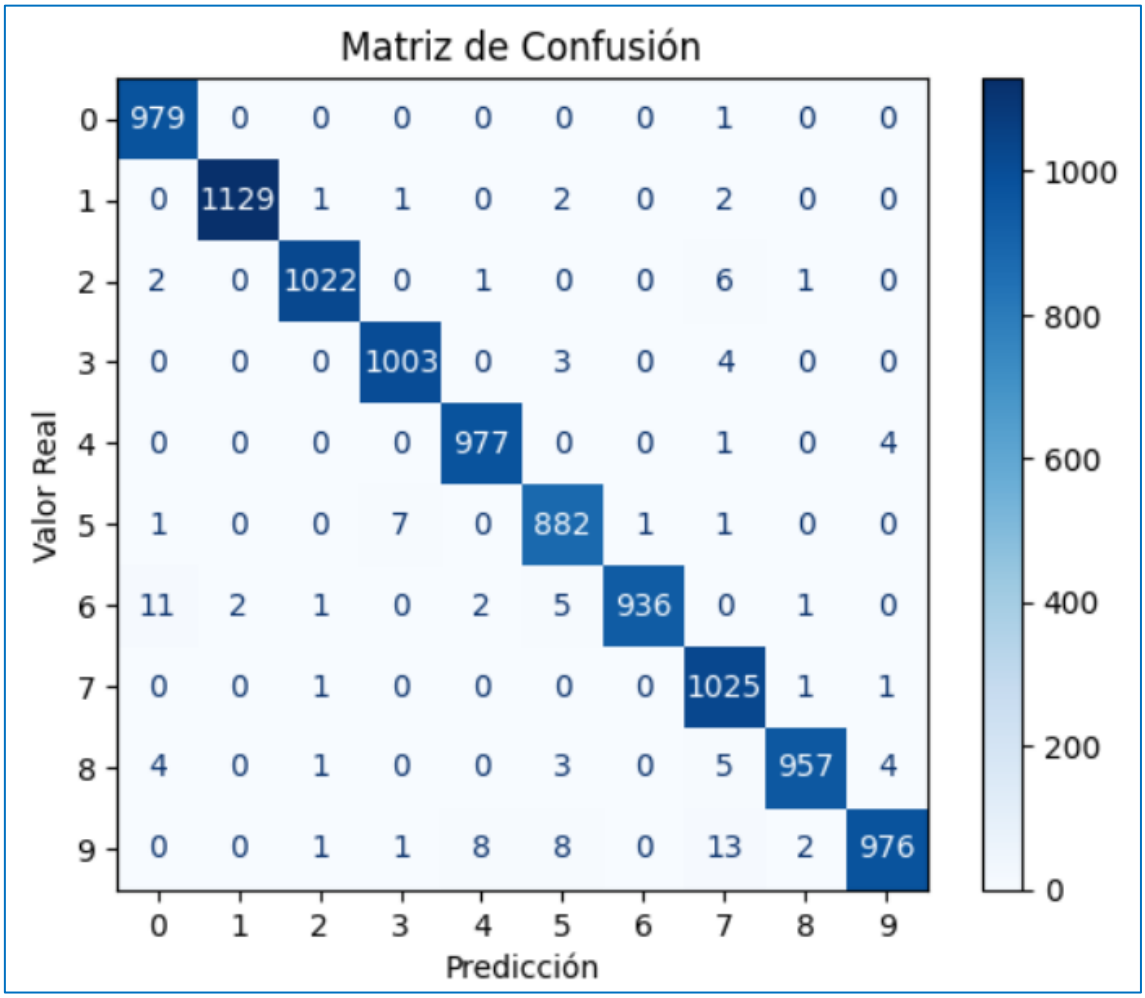
10. Matriz de confusión: La idea es conocer qué dígitos se han clasificado mejor

```
# Convertir las predicciones en etiquetas
predictions = model.predict(x_test)
predicted_classes = np.argmax(predictions, axis=1)
```

```
true_classes = np.argmax(y_test, axis=1)
```

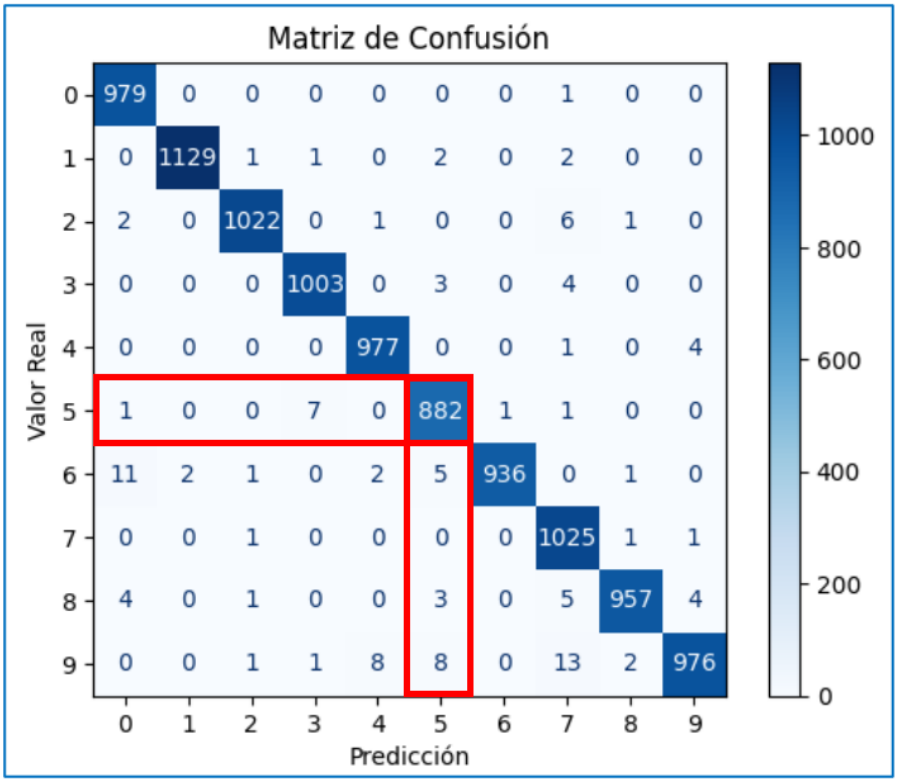
```
# Calcular matriz de confusión
cm = confusion_matrix(true_classes, predicted_classes)
cmd = ConfusionMatrixDisplay(cm, display_labels=range(10))
```

```
# Mostrar matriz de confusión
cmd.plot(cmap=plt.cm.Blues)
plt.xlabel("Predicción")
plt.ylabel("Valor Real")
plt.title("Matriz de Confusión")
plt.show()
```



Fijémonos, por ejemplo, en el número 5:

- Observamos que lo ha clasificado bien 882 veces, pero ha fallado 7 equivocando su predicción y confundiéndolo con un 3.



- Vemos que el número en el que más se ha equivocado ha sido el 9. Nada más y nada menos que 13 veces lo ha confundido con el número 7: