Diseño de CNNs

Diseñaremos nuestra propia ConvNet y veremos algunas aplicaciones existentes.

También exploraremos los tres métodos diferentes para definir un modelo en Keras:

- Secuencial
- Funcional
- Orientado a Objetos

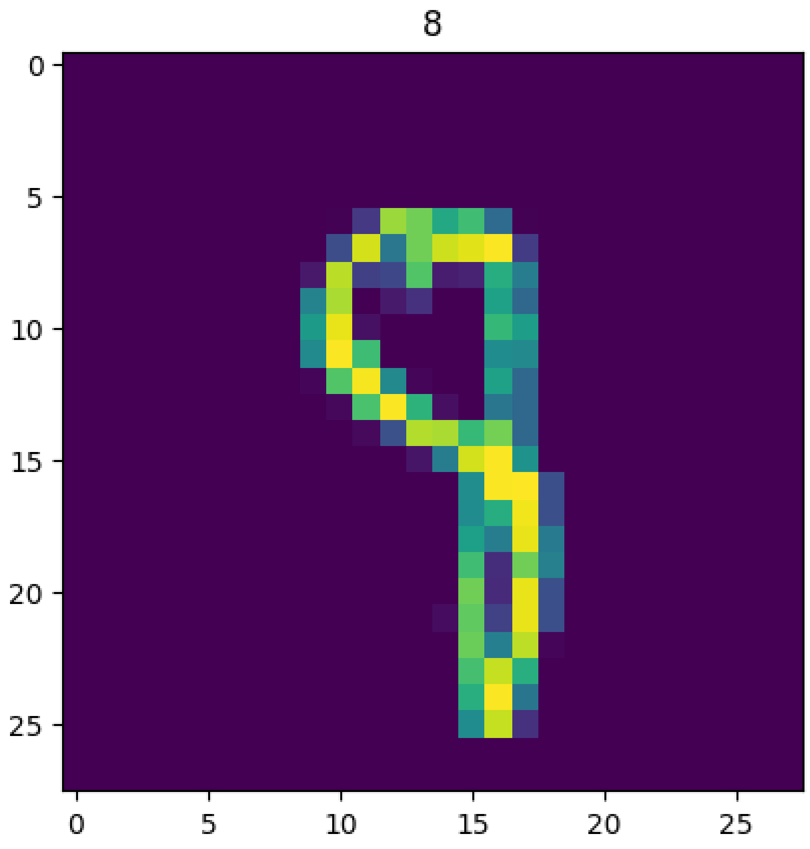
Dataset

```
from tensorflow.keras import datasets
from sklearn.model_selection import train_test_split

(x_train, y_train), (x_test, y_test) = datasets.mnist.load_data()
h, w = x_train.shape[1:]

x_train = x_train.reshape(x_train.shape[0], h, w, 1)
x_test = x_test.reshape(x_test.shape[0], h, w, 1)
input_shape = (h, w, 1)

x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255

x_train, x_val, y_train, y_val = train_test_split(
    x_train, y_train, test_size=10000, random_state=42)

(x_train.shape, y_train.shape), (x_val.shape, y_val.shape), (x_test.shape, y_test.shape)
```

```
(((50000, 28, 28, 1), (50000,)),
 ((10000, 28, 28, 1), (10000,)),
 ((10000, 28, 28, 1), (10000,)))
```

```
import matplotlib.pyplot as plt

plt.imshow(x_train[0].squeeze(-1))
plt.title(y_train[0])
```

```
Text(0.5, 1.0, '8')
```

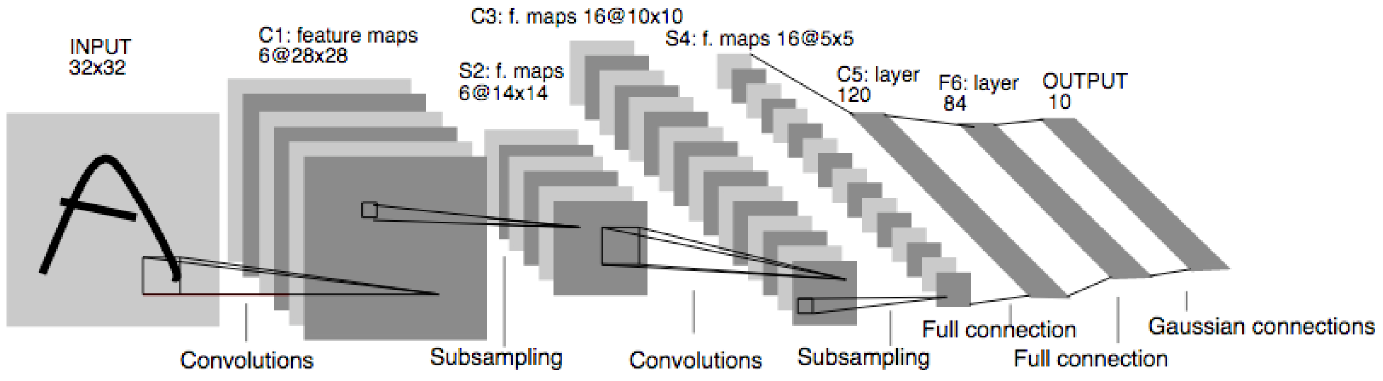


```python
import numpy as np

print(f"{np.unique(y_train)} unique labels.")
```

```
[0 1 2 3 4 5 6 7 8 9] unique labels.
```

1. LeNet

Yann Le Cun in 1998 (paper url).



lenet archi

```python
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import AvgPool2D, Conv2D, MaxPool2D, Dense, Flatten
from tensorflow.keras import optimizers

lenet = Sequential(name="LeNet-5")

lenet.add(Conv2D(6, kernel_size=(5, 5), activation="tanh", padding="same", name="C1"))
lenet.add(MaxPool2D(pool_size=(2, 2), name="S2"))
lenet.add(Conv2D(16, kernel_size=(5, 5), activation='tanh', name="C3"))
lenet.add(AvgPool2D(pool_size=(2, 2), name="S4"))
lenet.add(Conv2D(120, kernel_size=(5, 5), activation='tanh', name="C5"))
lenet.add(Flatten())
lenet.add(Dense(84, activation='tanh', name="F6"))
lenet.add(Dense(10, activation='softmax'))

lenet.summary()
```

```
Model: "LeNet-5"

┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━┓
┃ Layer (type)                     ┃ Output Shape           ┃       Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━┩
│ C1 (Conv2D)                      │ ?                      │   0 (unbuilt) │
├──────────────────────────────────┼────────────────────────┼───────────────┤
│ S2 (MaxPooling2D)                │ ?                      │             0 │
├──────────────────────────────────┼────────────────────────┼───────────────┤
│ C3 (Conv2D)                      │ ?                      │   0 (unbuilt) │
├──────────────────────────────────┼────────────────────────┼───────────────┤
│ S4 (AveragePooling2D)            │ ?                      │             0 │
├──────────────────────────────────┼────────────────────────┼───────────────┤
│ C5 (Conv2D)                      │ ?                      │   0 (unbuilt) │
├──────────────────────────────────┼────────────────────────┼───────────────┤
│ flatten (Flatten)                │ ?                      │   0 (unbuilt) │
├──────────────────────────────────┼────────────────────────┼───────────────┤
│ F6 (Dense)                       │ ?                      │   0 (unbuilt) │
├──────────────────────────────────┼────────────────────────┼───────────────┤
│ dense (Dense)                    │ ?                      │   0 (unbuilt) │
└──────────────────────────────────┴────────────────────────┴───────────────┘

 Total params: 0 (0.00 B)

 Trainable params: 0 (0.00 B)

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

```python
n_epochs = 5
batch_size = 256

lenet.compile(
    optimizer=optimizers.SGD(learning_rate=0.1),
    loss="sparse_categorical_crossentropy",
    metrics=["accuracy"]
)

lenet.fit(
    x_train, y_train,
    epochs=n_epochs,
    batch_size=batch_size,
    validation_data=(x_val, y_val)
)
```

```
Epoch 1/5

196/196 ━━━━━━━━━━━━━━━━━━━━ 4s 21ms/step - accuracy: 0.7120 - loss: 1.0518 - val_accuracy: 0.9033

Epoch 2/5

196/196 ━━━━━━━━━━━━━━━━━━━━ 4s 20ms/step - accuracy: 0.9145 - loss: 0.2941 - val_accuracy: 0.9298

Epoch 3/5

196/196 ━━━━━━━━━━━━━━━━━━━━ 4s 20ms/step - accuracy: 0.9368 - loss: 0.2104 - val_accuracy: 0.9449

Epoch 4/5

196/196 ━━━━━━━━━━━━━━━━━━━━ 4s 20ms/step - accuracy: 0.9507 - loss: 0.1629 - val_accuracy: 0.9581

Epoch 5/5

196/196 ━━━━━━━━━━━━━━━━━━━━ 4s 20ms/step - accuracy: 0.9612 - loss: 0.1321 - val_accuracy: 0.9677
```

```
<keras.src.callbacks.history.History at 0x3217e1ca0>
```

```
lenet.evaluate(x_test, y_test, verbose=0)
```

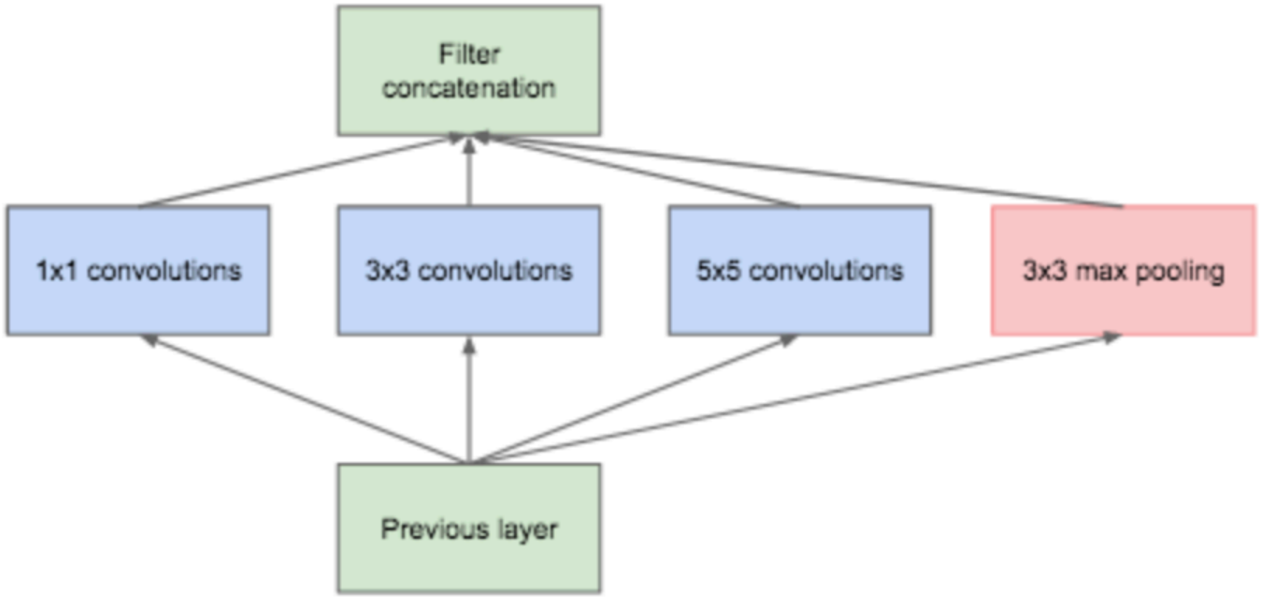
```
[0.10733101516962051, 0.9675999879837036]
```

Aunque LeNet fue definido inicialmente usando `tanh` o `sigmoid`, esas funciones de activación ahora se usan muy poco. Ambas se saturan con valores muy pequeños o muy grandes, lo que hace que su gradiente sea casi nulo.

Actualmente, la mayoría de las redes utilizan `ReLU` como función de activación en las capas ocultas, o alguna de sus variantes (https://keras.io/layers/advanced-activations/).

2. Inception

Szegedy et al. (paper url).



inception archi

Ejemplo usando la API **Functional**:

```
a = Input(shape=(32,))
b = Dense(32)(a)
model = Model(inputs=a, outputs=b)
```

```
from tensorflow.keras.layers import Concatenate, Flatten, Input
from tensorflow.keras.models import Model


def inception_layer(tensor, n_filters):
    branch1x1 = Conv2D(n_filters, kernel_size=(1, 1), activation="relu", padding="same")(tensor)
    branch5x5 = Conv2D(n_filters, kernel_size=(5, 5), activation="relu", padding="same")(tensor)
    branch3x3 = Conv2D(n_filters, kernel_size=(3, 3), activation="relu", padding="same")(tensor)

    branch_pool = MaxPool2D(pool_size=(3, 3), strides=(1, 1), padding="same")(tensor)

    output = Concatenate(axis=-1)(
        [branch1x1, branch5x5, branch3x3, branch_pool]
    )
    return output


input_tensor = Input(shape=input_shape)
x = Conv2D(16, kernel_size=(5, 5), padding="same")(input_tensor)
x = inception_layer(x, 32)
x = Flatten()(x)
output_tensor = Dense(10, activation="softmax")(x)

mini_inception = Model(inputs=input_tensor, outputs=output_tensor)
```

```
mini_inception.summary()
```

Model: "functional_1"

Layer (type)	Output Shape	Param #	Connected to
input_layer_1 (InputLayer)	(None, 28, 28, 1)	0	-
conv2d (Conv2D)	(None, 28, 28, 16)	416	input_layer_1[0]…
conv2d_1 (Conv2D)	(None, 28, 28, 32)	544	conv2d[0][0]
conv2d_2 (Conv2D)	(None, 28, 28, 32)	12,832	conv2d[0][0]
conv2d_3 (Conv2D)	(None, 28, 28, 32)	4,640	conv2d[0][0]
max_pooling2d (MaxPooling2D)	(None, 28, 28, 16)	0	conv2d[0][0]
concatenate (Concatenate)	(None, 28, 28, 112)	0	conv2d_1[0][0], conv2d_2[0][0], conv2d_3[0][0], max_pooling2d[0]…
flatten_1 (Flatten)	(None, 87808)	0	concatenate[0][0]
dense_1 (Dense)	(None, 10)	878,090	flatten_1[0][0]

Total params: 896,522 (3.42 MB)

Trainable params: 896,522 (3.42 MB)

Non-trainable params: 0 (0.00 B)

```
n_epochs = 5
batch_size = 256

mini_inception.compile(
    optimizer=optimizers.SGD(learning_rate=0.1),
    loss="sparse_categorical_crossentropy",
    metrics=["accuracy"]
)

mini_inception.fit(
    x_train, y_train,
    epochs=n_epochs,
    batch_size=batch_size,
    validation_data=(x_val, y_val)
)
```

```
mini_inception.evaluate(x_test, y_test, verbose=0)
```

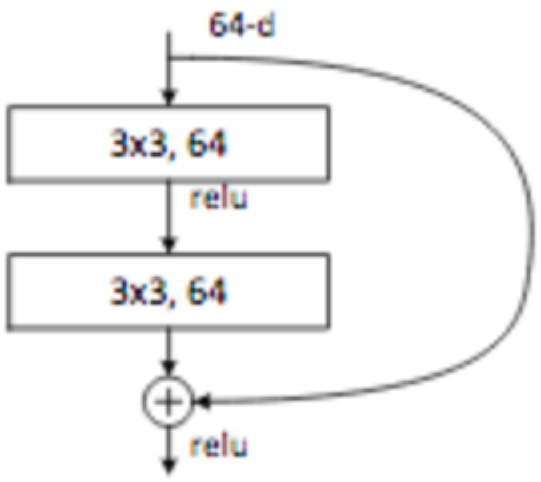
```
Epoch 1/5

196/196 ━━━━━━━━━━━━━━━━━━━━ 39s 197ms/step - accuracy: 0.8029 - loss: 0.6269 - val_accuracy: 0.954

Epoch 2/5

196/196 ━━━━━━━━━━━━━━━━━━━━ 38s 191ms/step - accuracy: 0.9704 - loss: 0.1030 - val_accuracy: 0.973

Epoch 3/5

196/196 ━━━━━━━━━━━━━━━━━━━━ 40s 206ms/step - accuracy: 0.9815 - loss: 0.0657 - val_accuracy: 0.979

Epoch 4/5

196/196 ━━━━━━━━━━━━━━━━━━━━ 38s 192ms/step - accuracy: 0.9844 - loss: 0.0513 - val_accuracy: 0.981

Epoch 5/5

196/196 ━━━━━━━━━━━━━━━━━━━━ 38s 193ms/step - accuracy: 0.9877 - loss: 0.0416 - val_accuracy: 0.981
```

```
[0.05889404937624931, 0.9814000129699707]
```

3. ResNet

Los modelos ResNet (*Residual Networks*) fueron introducidos por He et al. en 2015 (paper url). Ellos descubrieron que añadir más capas mejoraba el rendimiento, pero resultaba difícil retropropagar los gradientes hasta las primeras capas.

Un truco para permitir que los gradientes *fluyan* más fácilmente es utilizar conexiones de cortocircuito (*shortcut connections*) que dejan el tensor de entrada sin modificar (también conocidas como *residuales*).



resnet archi

Ejemplo usando la API orientada a objetos:

```
class MyModel(Model):
    def __init__(self):
        self.classifier = Dense(10, activation="softmax")

    def call(self, inputs):
        return self.classifier(inputs)
```

```
from tensorflow.keras.layers import Add, Layer, Activation

class ResidualBlock(Layer):
    def __init__(self, n_filters):
        super().__init__(name="ResidualBlock")

        self.conv1 = Conv2D(n_filters, kernel_size=(3, 3), activation="relu", padding="same")
        self.conv2 = Conv2D(n_filters, kernel_size=(3, 3), padding="same")
        self.add = Add()
        self.last_relu = Activation("relu")

    def call(self, inputs):
        x = self.conv1(inputs)
        x = self.conv2(inputs)

        y = self.add([x, inputs])
        y = self.last_relu(y)

        return y


class MiniResNet(Model):
    def __init__(self, n_filters):
        super().__init__()

        self.conv = Conv2D(n_filters, kernel_size=(5, 5), padding="same")
        self.block = ResidualBlock(n_filters)
        self.flatten = Flatten()
        self.classifier = Dense(10, activation="softmax")

    def call(self, inputs):
        x = self.conv(inputs)
        x = self.block(x)
        x = self.flatten(x)
        y = self.classifier(x)

        return y
```

```
mini_resnet = MiniResNet(32)
# mini_resnet.build((None, *input_shape))
mini_resnet.summary()
```

Model: "mini_res_net"

Layer (type)	Output Shape	Param #
conv2d_4 (Conv2D)	?	0 (unbuilt)
ResidualBlock (ResidualBlock)	?	0 (unbuilt)
flatten_2 (Flatten)	?	0 (unbuilt)
dense_2 (Dense)	?	0 (unbuilt)

Total params: 0 (0.00 B)

Trainable params: 0 (0.00 B)

Non-trainable params: 0 (0.00 B)

```
n_epochs = 5
batch_size = 256

mini_resnet.compile(
    optimizer=optimizers.SGD(learning_rate=0.1),
    loss="sparse_categorical_crossentropy",
    metrics=["accuracy"]
)

mini_resnet.fit(
    x_train, y_train,
    epochs=n_epochs,
    batch_size=batch_size,
    validation_data=(x_val, y_val)
)

mini_resnet.evaluate(x_test, y_test, verbose=0)
```

```
Epoch 1/5

/Users/aveloz/miniconda3/lib/python3.9/site-packages/keras/src/optimizers/base_optimizer.py:774:
UserWarning: Gradients do not exist for variables ['mini_res_net/ResidualBlock/conv2d_5/kernel',
'mini_res_net/ResidualBlock/conv2d_5/bias'] when minimizing the loss. If using `model.compile()`,
did you forget to provide a `loss` argument?
  warnings.warn(

196/196 ━━━━━━━━━━━━━━━━━━━━ 16s 81ms/step - accuracy: 0.7973 - loss: 0.7147 - val_accuracy: 0.9618

Epoch 2/5
```

```
196/196 ━━━━━━━━━━━━━━━━━━━━ 16s 79ms/step - accuracy: 0.9665 - loss: 0.1171 - val_accuracy: 0.9213

Epoch 3/5

196/196 ━━━━━━━━━━━━━━━━━━━━ 16s 81ms/step - accuracy: 0.9712 - loss: 0.0945 - val_accuracy: 0.9679

Epoch 4/5

196/196 ━━━━━━━━━━━━━━━━━━━━ 15s 79ms/step - accuracy: 0.9788 - loss: 0.0716 - val_accuracy: 0.9684

Epoch 5/5

196/196 ━━━━━━━━━━━━━━━━━━━━ 15s 79ms/step - accuracy: 0.9829 - loss: 0.0592 - val_accuracy: 0.9723

 [0.08727332949638367, 0.9731000065803528]
```

4. Batch Normalization

Batch Normalization no es una arquitectura, es una capa. En el artículo de Ioffe et al. in 2015 (paper url) se señala:

> Training Deep Neural Networks is complicated by the fact that the **distribution of each layer's inputs changes during training, as the parameters of the previous layers change**. This slows down the training by requiring lower learningrates and careful parameter initialization, and makes it notoriously hard to train models with saturating nonlinearities. We refer to this phenomenon as **internal covariate shift**, and address the problem by **normalizing layer inputs**.

El resultado es que las CNNs que se entrenan con BatchNorm convergen más rápido con resultados mejores. Actualmente, todas o casi todas usan alguna forma de BatchNorm. Ver el artículo.

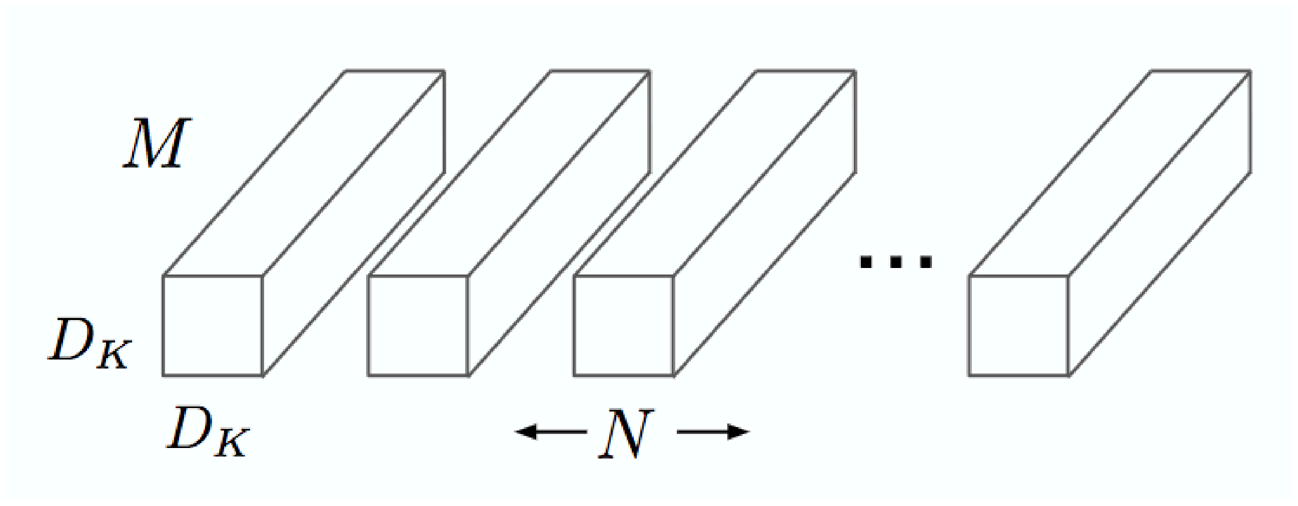
Un bloque clásico es:

```python
class ConvBlock(Layer):
    def __init__(n_filters, kernel_size):
        super().__init__()

        self.conv = Conv2D(n_filters, kernel_size=kernel_size, use_bias=False)
        self.bn = BatchNormalization(axis=3)
        self.activation = Activation("relu")

    def call(self, inputs):
        return self.activation(
            self.bn(self.conv(inputs))
        )
```

5. Convoluciones separables

Las CNNs suelen tener muchos parámetros debido a su gran profundidad. Un truco para reducir el número de parámetros con una pérdida mínima en el rendimiento es usar **convolución separable**.

La convolución estándar tiene muchos parámetros (aunque sigue siendo mucho menos que una capa densa):



(a) Standard Convolution Filters

conv

```
conv_model = Sequential(name="Conv Model")
conv_model.add(Conv2D(8, kernel_size=(3, 3), use_bias=False))
```

Cuántos parámetros?

```
conv_model.build((None, *input_shape))
conv_model.summary()
```

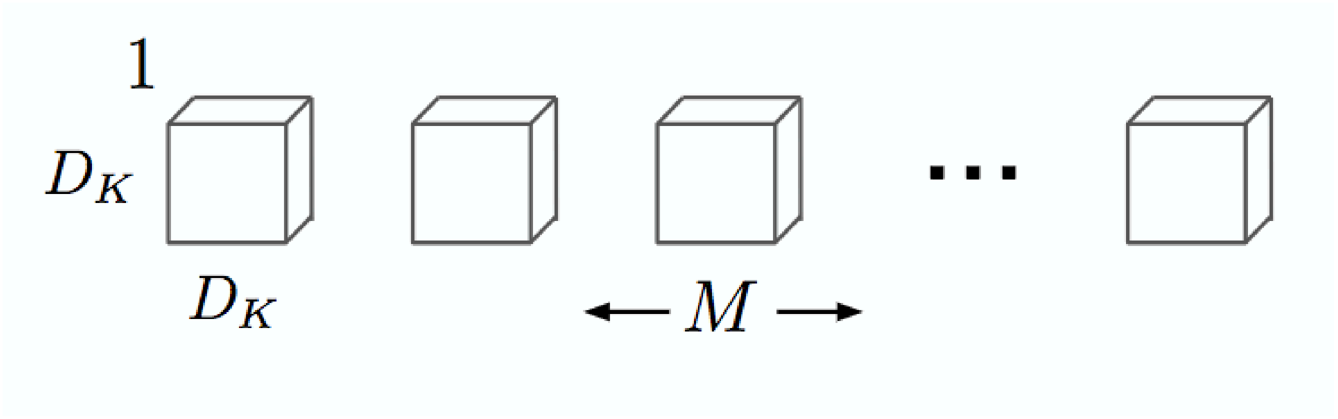
```
Model: "Conv Model"
```

Layer (type)	Output Shape	Param #
conv2d_7 (Conv2D)	(None, 26, 26, 8)	72

```
Total params: 72 (288.00 B)

Trainable params: 72 (288.00 B)

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

Las convoluciones separables consisten en dos tipos de convolución:

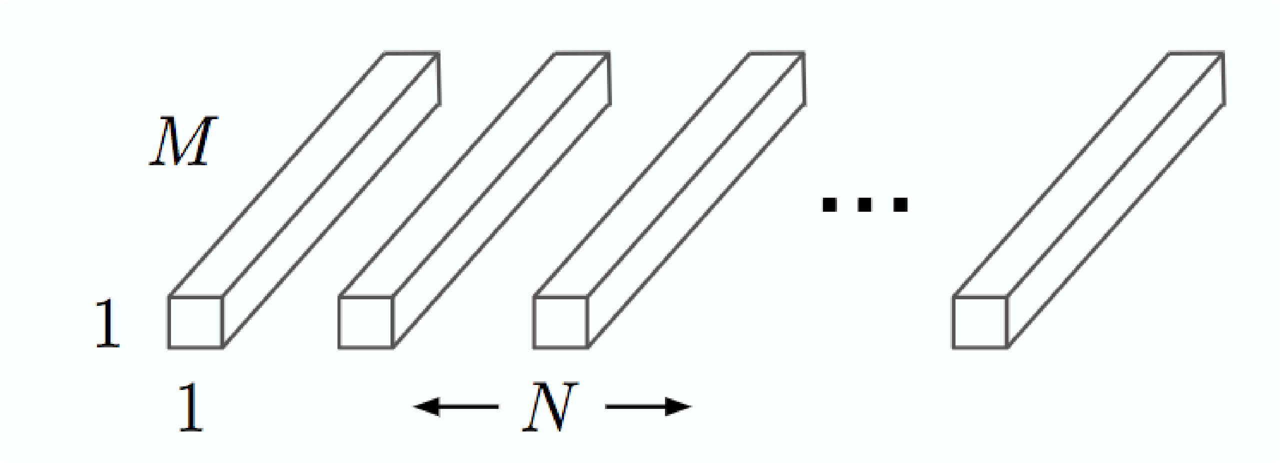
- Una **convolución depthwise**: se crea un único kernel por canal de entrada, por lo que se afecta la información espacial, pero no se comparte información entre canales.



(b) Depthwise Convolutional Filters

depthwise conv

- Una **convolución pointwise**: es una convolución habitual con un kernel de tamaño (1, 1). Aquí, la información espacial no se ve afectada, pero sí se comparte información entre los canales.



(c) 1×1 Convolutional Filters called Pointwise Convolution in the context of Depthwise Separable Convolution

pointwise conv

```
from tensorflow.keras.layers import DepthwiseConv2D

separable_model = Sequential(name="Separable Model")
separable_model.add(DepthwiseConv2D(kernel_size=(3, 3), use_bias=False))
separable_model.add(Conv2D(8, kernel_size=(1, 1), use_bias=False))
```

- Cuantos parámetros tienen las convoluciones Depthwise y Pointwise?

```
separable_model.build((None, *input_shape))
separable_model.summary()
```

```
Model: "Separable Model"
```

```
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━┓
┃ Layer (type)                          ┃ Output Shape               ┃         Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━┩
│ depthwise_conv2d                      │ (None, 26, 26, 1)          │               9 │
│ (DepthwiseConv2D)                     │                            │                 │
├───────────────────────────────────────┼────────────────────────────┼─────────────────┤
│ conv2d_8 (Conv2D)                     │ (None, 26, 26, 8)          │               8 │
└───────────────────────────────────────┴────────────────────────────┴─────────────────┘

 Total params: 17 (68.00 B)

 Trainable params: 17 (68.00 B)

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