

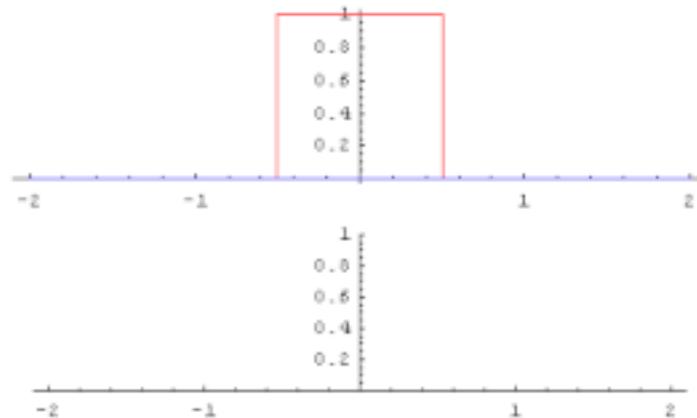
# *Redes Neuronales Convolucionales*

Prof. Wílmer Pereira

# Percepción visual

El sistema visual funciona mediante especialización estratificada ... ciertas neuronas se estimulan con ciertas áreas del campo visual, lo que, representa el **campo receptivo**. Las diferentes neuronas, del campo receptivo, funcionan como filtros del campo visual

- Se pretende tener una red neuronal artificial que manipule las imágenes como el cerebro. Es decir, con filtros **invariantes a la traslación** que capturen patrones organizados jerárquicamente.
- Este proceso de superposición de neuronas en el caso del cerebro humano es similar al mecanismo de **convolución**. Esta es una operación matemática que transforma dos funciones (filtro e imagen) en una tercera función (patrón), tal como se comportaría la neurona natural que recorre el campo visual y reconoce una forma.



# *Red Neuronal Convolucional*

## (CNN)

La idea es hacer una operación de convolución entre la matriz RGB de la imagen, con matrices que representan filtros, para obtener otra matriz/imagen donde se resalten patrones característicos de la imagen ... El proceso se extiende naturalmente a videos.

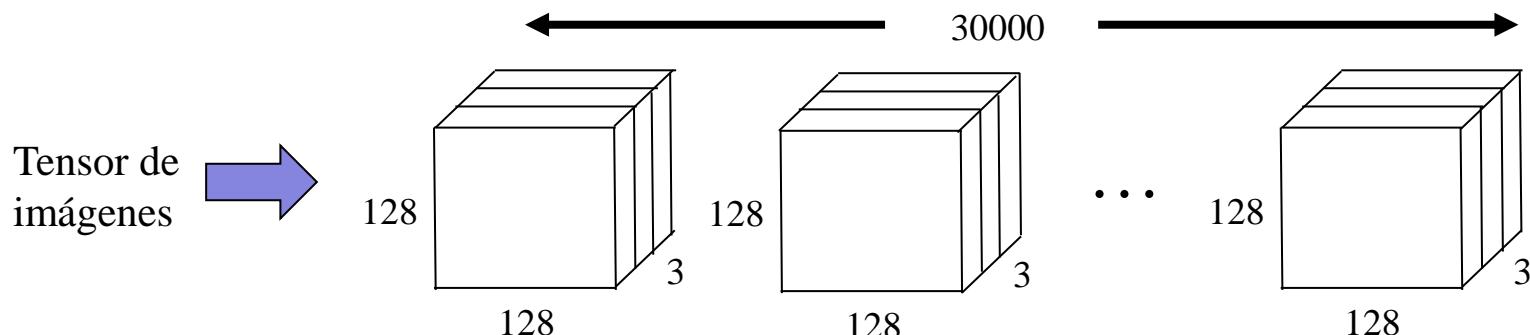
- Se definen varias capas convolucionales y se “aprenden” los filtros para identificar patrones. Además, desde el punto de vista práctico, son necesarias operaciones de reducción de información o *subsampling* para disminuir la complejidad del espacio visual.
- Las operaciones de matrices se pueden paralelizar, afortunadamente no explícitamente, cuando se usan librería que se valen de GPU o clusters
- Las CNN, pueden utilizarse en reconocimiento y clasificación de imágenes y videos donde requieren poco tratamiento previo de las imágenes. También han demostrado ser muy eficientes en procesamiento de lenguaje natural, sistemas de reconocimiento, series de tiempo, ...

# Tensores

- Es una representación matricial para datos multidimensionales. Un tensor de 0D es un escalar mientras que 1D es un arreglo que puede representar un ejemplo del conjunto de entrenamiento. Por ejemplo, los tensores 2D constituyen todos los ejemplos de entrada de datos numéricos, por ejemplo: (*muestra, atributos*)
- Datos un poco más complejos como series de tiempo pueden ser tensores 3D: (*muestra, atributos, pasosTiempo*). Finalmente imágenes y videos serían tensores 4D y 5D respectivamente:

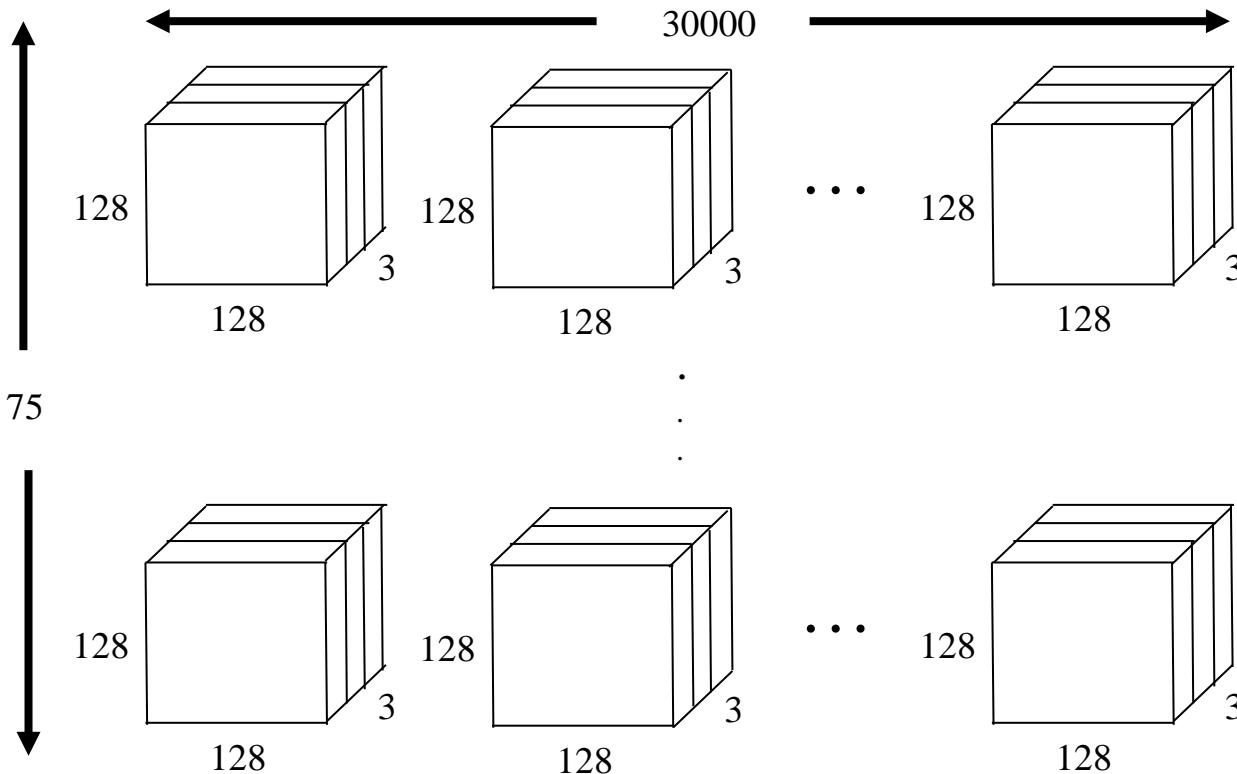
(*muestra, anchoPixel, altoPixel, coloresRGB*) → (30000,128,128,3)

(*muestra, marco, anchoPixel, altoPixel, coloresRGB*) → (30000,75,1024,1024,3)



- En consecuencia, para una CNN que procesa imágenes o video, los tensores necesarios son 4D o 5D ... Es de notar que la cantidad de datos es tan grande que se requerirán muchas capas de procesamiento, haciendo casi indispensable el procesamiento paralelo durante el entrenamiento y tener mecanismo de reducción de los datos para manejar el efecto Hughes (la cantidad de datos necesarios para entrenar, crece exponencialmente con la dimensionalidad)

# *Tensor 5D para video*

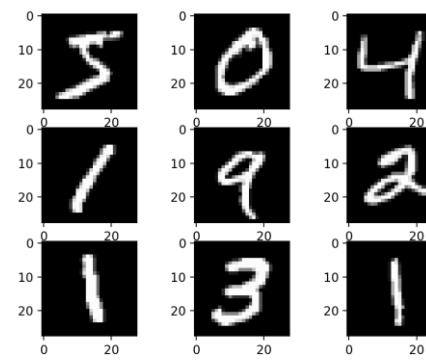


- Esto representa apenas un segundo para los 30000 videos, suponiendo que son 75 marcos/seg. En consecuencia, si desea procesar 1H30 de todos los videos, son 90x60 segundos, es decir, 5400 veces esta estructura de datos ...

# Ejemplo para CNN (MNIST)

- Partamos de un ejemplo muy conocido para reconocimiento de dígitos. Existe una base de conocimiento, [MNIST](#), formada por 70000 ejemplos etiquetados en 10 clases. Las imágenes son 28x28 píxeles de niveles de grises (256 → 1 byte). En consecuencia, los datos se pueden cargar en un tensor 4D (70000,28,28,1)

0 0 0 0 0 0 0 0 0 0  
1 1 1 1 1 1 1 1 1 1  
2 2 2 2 2 2 2 2 2 2  
3 3 3 3 3 3 3 3 3 3  
4 4 4 4 4 4 4 4 4 4  
5 5 5 5 5 5 5 5 5 5  
6 6 6 6 6 6 6 6 6 6  
7 7 7 7 7 7 7 7 7 7  
8 8 8 8 8 8 8 8 8 8  
9 9 9 9 9 9 9 9 9 9



- Si aplazamos la entrada de cada dígito, se tienen  $28 \times 28 = 784$  neuronas de entrada que se deben normalizar a valores entre 0 y 1 (en lugar de 0-255)
- Una vez preparados los datos, se definen los filtros para operar sobre píxeles cercanos y así identificar los patrones independiente de su posición (invarianza a la translación)
- Después de procesar una imagen, por cada capa de convolución, se hace una reducción para disminuir el tamaño de la imagen resultante. Finalmente, se termina con una red neuronal clásica para clasificación multiclase de los 10 dígitos posibles.



# Intuición de una convolución

- Imaginemos la operación que emula el campo receptivo como una máscara que se denomina comunmente **kernel** o parche. Para una CNN es una matriz  $m \times m$  con la que se hará la convolución a cada imagen

		0.6	0.6	
0.6				0.6
0.6	0.6	0.6	0.6	
0.6			0.6	

Imagen de entrada

1	0	-1
2	0	-2
1	0	-1

kernel

Los valores iniciales del kernel son aleatorios y se irán ajustando con el *backpropagation*

La operación o convolución se realiza entre el kernel contra la imagen con desplazamientos de izquierda a derecha y de arriba abajo, por defecto, un paso (*stride*) a la vez ...

		0.6	0.6	
0.6				0.6
0.6	0.6	0.6	0.6	
0.6			0.6	

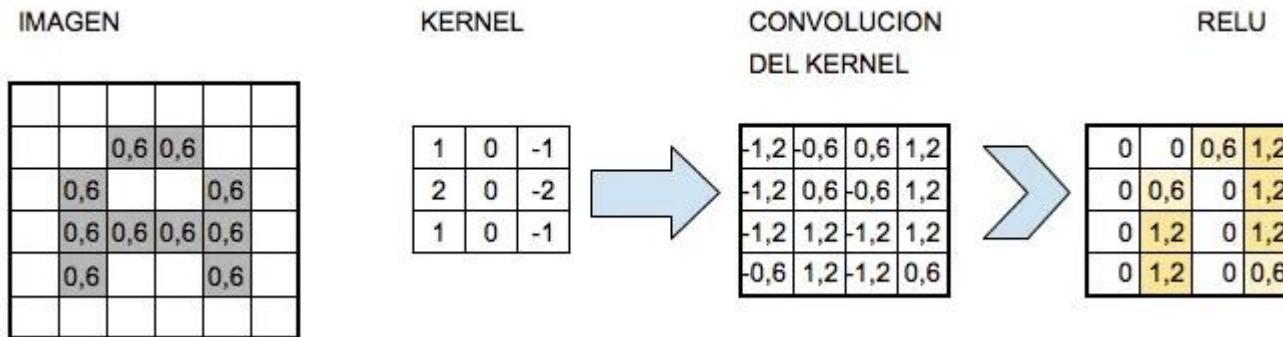
-1,2	-0,6	0,6	1,2
-1,2			

[Animación](#) para apreciar el desplazamiento del kernel o parche sobre la imagen haciendo las convoluciones

- Es de notar que si la imagen original es de  $n \times n$ , y el kernel de  $m \times m$ , la imagen resultante será de  $(n-m+1) \times (n-m+1)$  ... Los pasos son, normalmente, de una posición pero pueden ser saltos mayores (depende del diseñador) ...  $m$  es impar para que esté centrado ...

# Filtrado en CNN

- Aplicado a una capa de convolución de la CNN, el resultado sería:



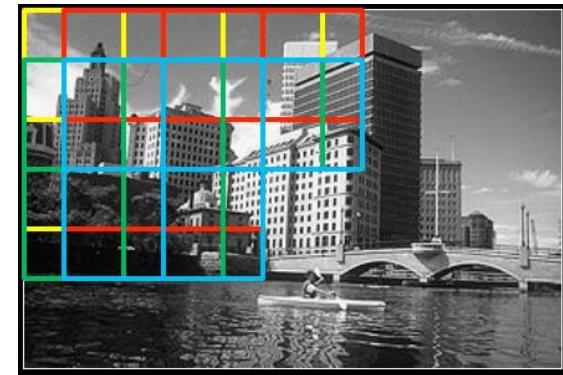
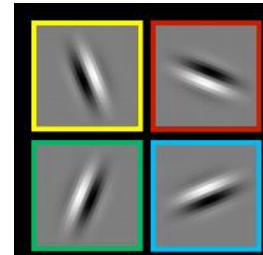
- Normalmente se aplican varios parches o kernels (que llamaremos **filtro**), supongamos 32, es decir que se desea detectar 32 características de la imagen original. Esto es conocido como la **profundidad** de la capa. En el caso de MNIST, de la imagen 28x28x1, al aplicar la convolución con 32 filtros, se tendría 26x26x32, es decir, 32 imágenes por lo que se necesitarían 21632 neuronas en la siguiente capa, en lugar de 784 neuronas.

Además lo común es aumentar la cantidad de kernel por capa para detectar más características

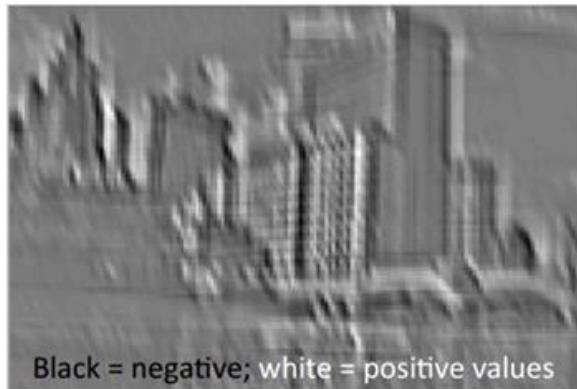


Por capa convolucional, aumentan rápidamente la cantidad de parámetros que debe aprender la red

# Efecto de Convolución + ReLU



Convolución



Black = negative; white = positive values

ReLU



Only non-negative values

# Ejemplos de algunos filtros

Sharpen:

0	0	0	0	0	0
0	0	-1	0	0	
0	-1	5	-1	0	
0	0	-1	0	0	
0	0	0	0	0	



Emboss:

-2	-1	0
-1	1	1
0	1	2



Blur:

0	0	0	0	0	0
0	1	1	1	0	
0	1	1	1	0	
0	1	1	1	0	
0	0	0	0	0	



Edge Enhance:

0	0	0
-1	1	0
0	0	0



Edge Detect:

0	1	0
1	-4	1
0	1	0



<https://docs.gimp.org/2.10/es/>

# Reducción: max-pooling

- Es necesario reducir la matriz convolucionada, recuperando sólo los datos más importante

0	0	0.6	1.2
0	0.6	0	1.2
0	1.2	0	1.2
0	1.2	0	0.6

0.6	1.2
1.2	1.2

En este caso, se hace *subsampling* con una matriz  $r \times r$  (en el ejemplo  $2 \times 2$ ) seleccionando el mayor valor. Esto se conoce como **max-pooling**. Es posible también **avg-pooling**, pero los mejores resultados son con el máximo

- En el caso de MNIST, después de un *max-pooling* con una matriz  $2 \times 2$ , y salto de 2, la matriz quedaría de  $13 \times 13 \times 32$  para la siguiente capa, es decir, 5408 neuronas en lugar de 21632 neuronas.
- Para no perder la información en los extremos de la imagen, el kernel pueden sobrepasar el marco de la imagen por lo que debe haber un **padding**. Puede ser 0, 1, extensión, ...

1	2	3
4	5	6
7	8	9
10	11	12

5
11

Opción valid (valor por defecto)

1	2	3	0
4	5	6	0
7	8	9	0
10	11	12	0

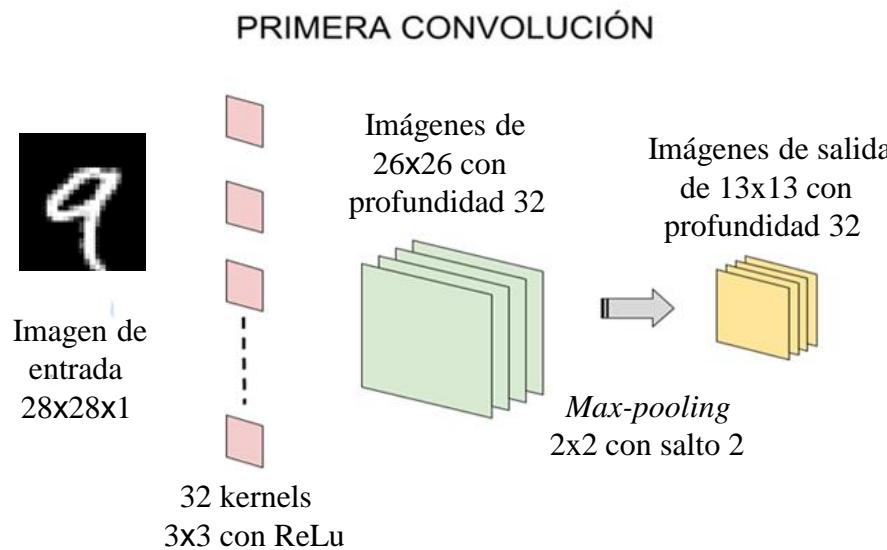
Opción same (con relleno)

5	6
11	12

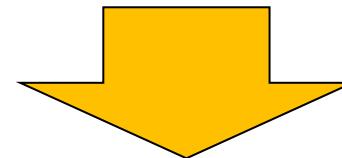
# Capas de convoluciones:

## 1<sup>era</sup> capa

- Si los saltos de la tabla de *pooling* no son de 1 posición, y el tamaño del kernel no es múltiplo del tamaño de la imagen, entonces será necesario considerar el valor de *padding* en `same`. Como en este caso, los saltos son de una posición no hace falta cambiar el valor de *padding* y quedará con su valor por defecto



Así, después del *Max-Pooling*, se tendrán 13x13x32 neuronas (5408)

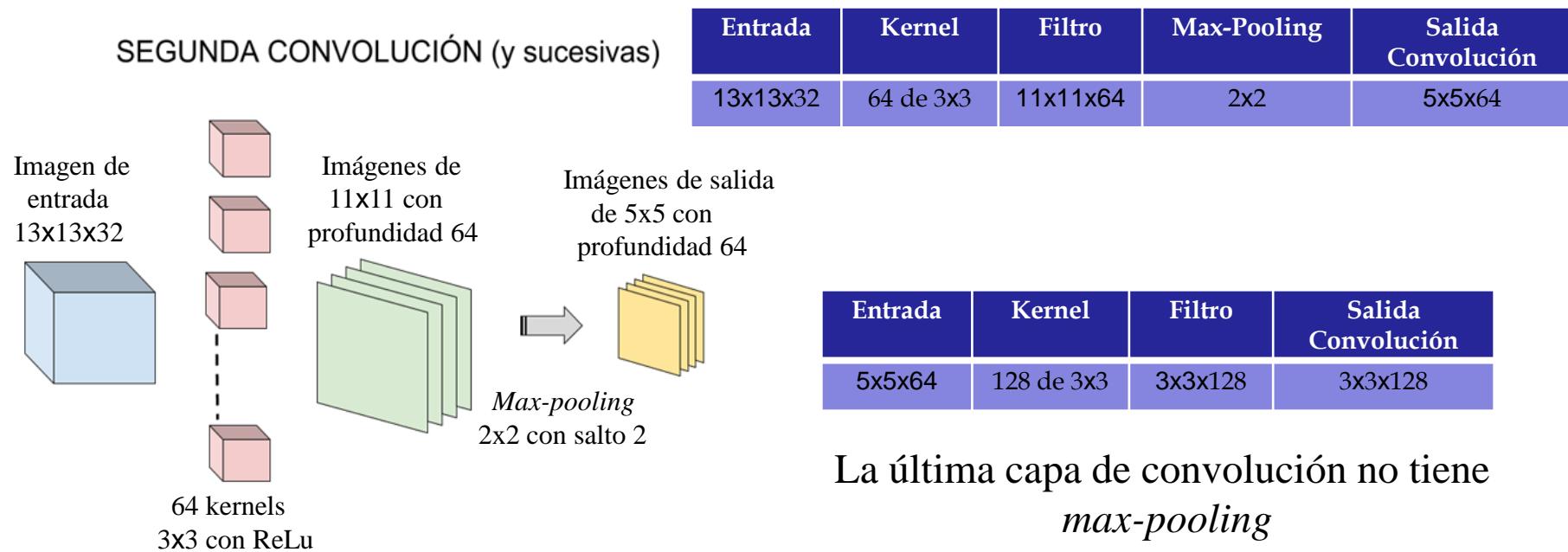


Hay menos riesgo de *overfitting* ...  
y además menos poder de procesamiento para el entrenamiento ...

Entrada	Kernel	Filtro	Max-Pooling	Salida Convolución
28x28x1	32 de 3x3	26x26x32	2x2 salto 2	13x13x32

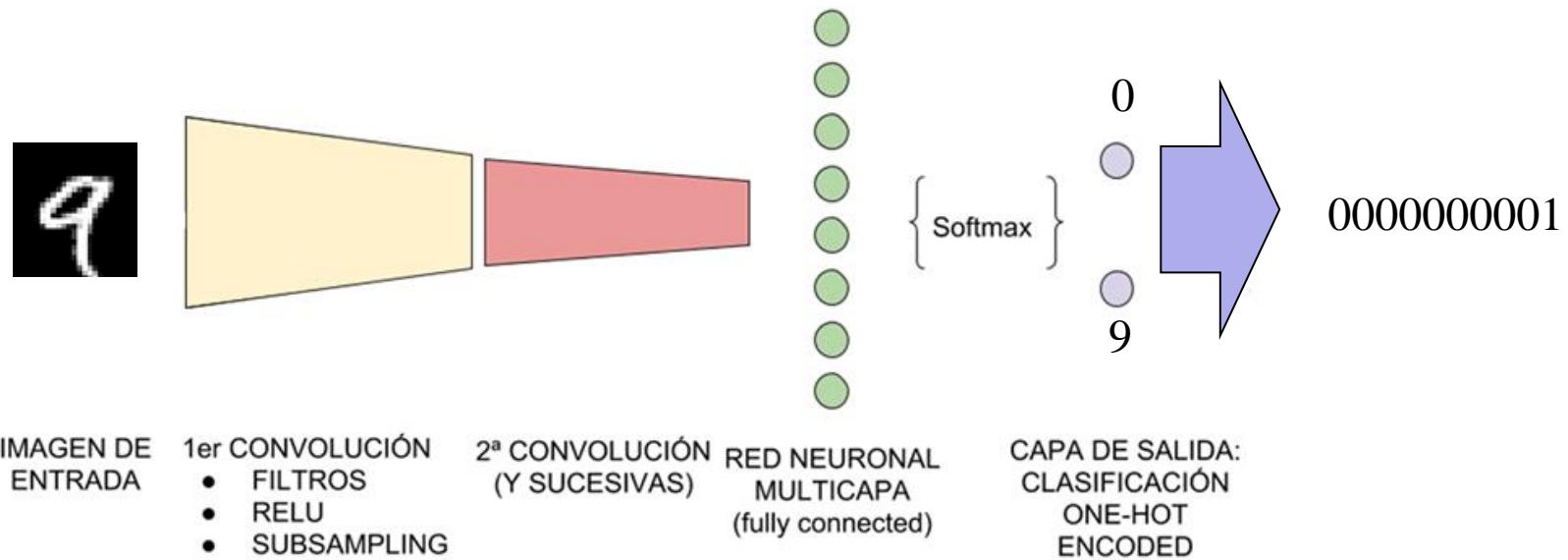
# Siguientes capas de convoluciones

- La primera convolución detectó líneas y curvas. Las siguientes capas reconocerán formas más complejas

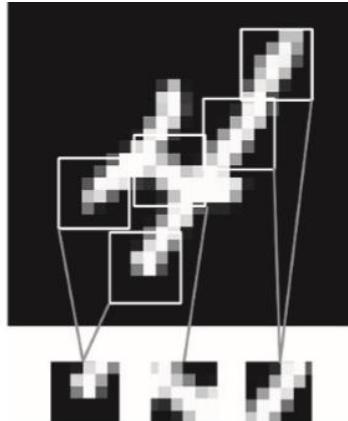


- Ahora se agrega la segunda parte de la red neuronal para clasificar. Aplanando la salida de la tercera capa convolucional, la siguiente capa debe tener 1152 neuronas (ReLU) y finalmente la última capa por ser una clasificador multiclasa exclusivo para MNIST, debe tener 10 neuronas (softmax).

# Estructura global de la CNN



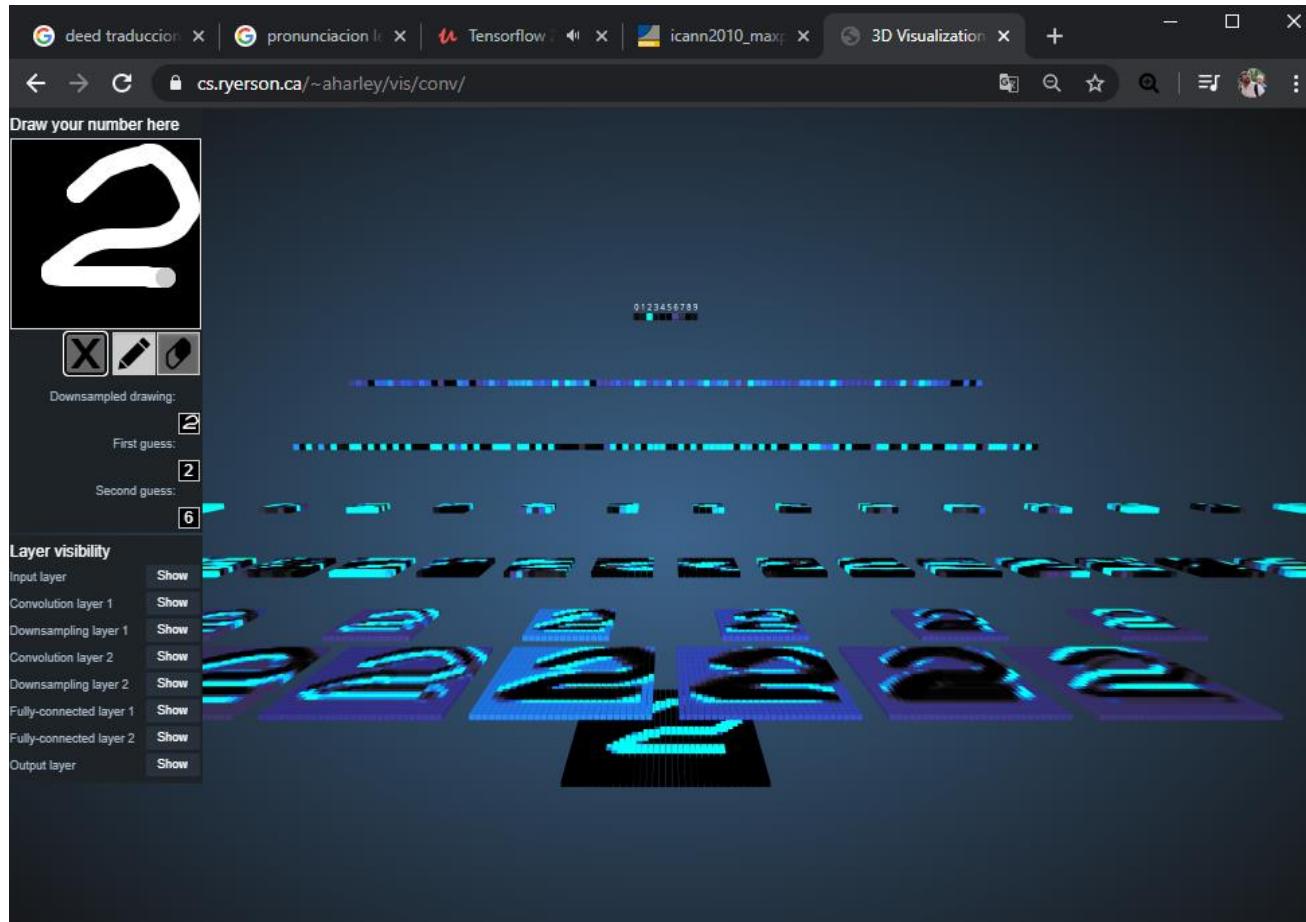
- El *backpropagation* consiste en ajustar los pesos para aprender los kernels y tener el filtrado adecuado para reconocer el dígito introducido en la entrada



Por ejemplo, se podría tener un kernel que reconozca los tramos en diagonal del dígito. En este caso identificaría, cuatro segmentos para el dígito 4

El siguiente [ejemplo](#) resume bien toda la teoría de redes neuronales convolucionales

# *Ejemplo gráfico de una CNN*



<https://www.cs.ryerson.ca/~aharley/vis/conv/>

# Hiperparámetros de una CNN

- Las decisiones a priori que debe tomar el diseñador antes de entrenar la red son:
  - Tamaño de los kernels. Lo común es de 2x2 hasta 5x5 ...
  - Cantidad de kernels (filtro) que se harán en cada fase de convolución.  
Lo recomendado es pocos kernels en las primeras fases e ir aumentando hacia las últimas fases de convolución.
  - Pasos o movimiento de cada kernel sobre la imagen (*stride*). Por defecto es un paso, con superposición, aunque es configurable.
  - Como llenar los bordes de la imagen después de la convolución.  
Las opciones son llenar de ceros o unos en los bordes
  - Tamaño de la tabla de *pooling* que va habitualmente de 2x2 a 4x4 ...  
Mientras más grande sea más información es susceptible de perderse ... Lo ideal es sin sobreposición con el valor máximo (no *average*)
  - Es posible tener *dropout* (eliminación de neuronas) o *dropconnect* (eliminación de arcos en las entradas). Este último no es recomendable si la resolución de las imágenes es baja. También puede agregarse regularización (L1 y/o L2) si se sospecha de *overfitting*.

# MNIST con Keras

- El primer paso es definir las capas convolucionales (2D porque se trata de imágenes) y las especificaciones de la reducción (*Max-Pooling*) hasta una imagen altamente comprimida de 3x3. Luego se aplana, de nuevo, para la fase de clasificación con 10 clases.

```
from keras import layers
from keras import models

model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))

model.add(layers.Flatten())
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))
```

```
>>> model.summary()

Layer (type)          Output Shape         Param #
-----  
conv2d_1 (Conv2D)     (None, 26, 26, 32)      320  
maxpooling2d_1 (MaxPooling2D) (None, 13, 13, 32)      0  
conv2d_2 (Conv2D)     (None, 11, 11, 64)      18496  
maxpooling2d_2 (MaxPooling2D) (None, 5, 5, 64)      0  
conv2d_3 (Conv2D)     (None, 3, 3, 64)      36928  
flatten_1 (Flatten)   (None, 576)            0  
dense_1 (Dense)       (None, 64)             36928  
dense_2 (Dense)       (None, 10)             650  
-----  
Total params: 93,322  
Trainable params: 93,322  
Non-trainable params: 0
```

# Estructura global de la CNN

- Se recuperan los datos, ya separados en conjunto de entrenamiento y conjunto de prueba, se normalizan, se categorizan las clases para que sean etiquetas no numéricas. Por último se definen los hiperparámetros para el *backpropagation* y se entrena:

```
from keras.datasets import mnist
from keras.utils import to_categorical

(train_images, train_labels), (test_images, test_labels) = mnist.load_data()

train_images = train_images.reshape((60000, 28, 28, 1))
train_images = train_images.astype('float32') / 255

test_images = test_images.reshape((10000, 28, 28, 1))
test_images = test_images.astype('float32') / 255

train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)

model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
model.fit(train_images, train_labels, epochs=5, batch_size=64)
```

- Finalmente se mide el *accuracy* en la predicción del conjunto de prueba, obteniéndose una precisión del 0.993, es decir, del 99.3% ...

```
>>> test_loss, test_acc = model.evaluate(test_images, test_labels)
>>> test_acc
```

- Para texto se usan capas convolucionales 1D y para volúmenes convolucionales 3D. Un buen [ejemplo](#) con el código más detallado para MNIST ...

# Entrenar una CNN con pocas imágenes

Este escenario es bastante común ... hay varias soluciones como aumentar los datos con variaciones de las imágenes sobre el conjunto original o usar capas convolucionadas pre-entrenadas con muchas imágenes disponibles ya clasificadas ...

- Se parte de un conjunto de imágenes de *kaggle*, disponible desde el 2013, el cual tiene 25000 imágenes de perros y gatos. Se toman sólo 2000 para entrenamiento, 1000 para validación y 1000 de prueba para justamente mostrar las estrategias cuando se tienen pocos ejemplos de entrada. A diferencia de MNIST (base de datos dígitos manuscritos, 28x28x1) las imágenes son más grandes y a colores de 150x150x3



# Estructura de la red para perros y gatos

```
from keras import layers
from keras import models

model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu',
                      input_shape=(150, 150, 3)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
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'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Flatten())
model.add(layers.Dense(512, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

```
>>> model.summary()

Layer (type)          Output Shape         Param #
conv2d_1 (Conv2D)     (None, 148, 148, 32)  896
maxpooling2d_1 (MaxPooling2D) (None, 74, 74, 32)  0
conv2d_2 (Conv2D)     (None, 72, 72, 64)   18496
maxpooling2d_2 (MaxPooling2D) (None, 36, 36, 64)  0
conv2d_3 (Conv2D)     (None, 34, 34, 128)  73856
maxpooling2d_3 (MaxPooling2D) (None, 17, 17, 128)  0
conv2d_4 (Conv2D)     (None, 15, 15, 128)  147584
maxpooling2d_4 (MaxPooling2D) (None, 7, 7, 128)  0
flatten_1 (Flatten)   (None, 6272)        0
dense_1 (Dense)       (None, 512)         3211776
dense_2 (Dense)       (None, 1)           513

Total params: 3,453,121
Trainable params: 3,453,121
Non-trainable params: 0
```

Para evitar el riesgo de *overfitting* se trata de tener el menor número de capas convolucionadas por los pocos datos de entrada. Noté que en la etapa de clasificación sólo hay dos capas, donde la última es con función de activación sigmoide por tratarse de una clasificación binaria.

Es de notar que hay una gran cantidad de parámetros a ajustar (3,453,123), a diferencia de MNIST que eran apenas 93,322 ...

Los parámetros de una capa convolucional se calculan como:

neuronasSalida\*(neuronasEntrada\*TamañoKernel+1)

y de un capa convencional sería:

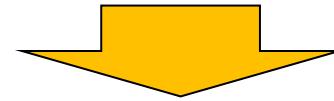
neuronasSalida\*(neuronasEntrada + 1)

# Preprocesamiento y entrenamiento

- Previo a la definición de la red convolucional, se debe leer las imágenes de dimensiones 150x150, normalizada y recuperarla en grupos de tamaño 20.

```
from keras.preprocessing.image import ImageDataGenerator  
  
# All images will be rescaled by 1./255  
train_datagen = ImageDataGenerator(rescale=1./255)  
test_datagen = ImageDataGenerator(rescale=1./255)  
  
train_generator = train_datagen.flow_from_directory(  
    # This is the target directory  
    train_dir,  
    # All images will be resized to 150x150  
    target_size=(150, 150),  
    batch_size=20,  
    # Since we use binary_crossentropy loss, we need binary labels  
    class_mode='binary')  
  
validation_generator = test_datagen.flow_from_directory(  
    validation_dir,  
    target_size=(150, 150),  
    batch_size=20,  
    class_mode='binary')
```

```
history = model.fit_generator(  
    train_generator,  
    steps_per_epoch=100,  
    epochs=30,  
    validation_data=validation_generator,  
    validation_steps=50)
```



2000 casos de entrenamiento 100x20  
1000 casos de validación 50x20.

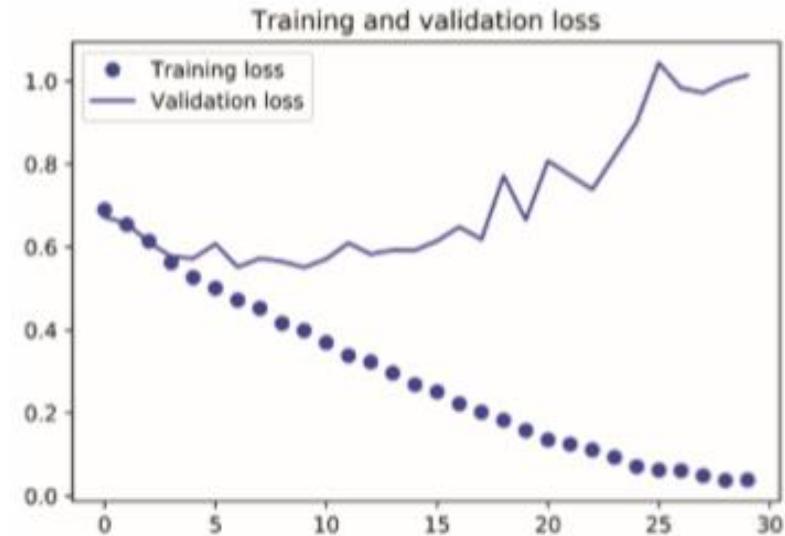
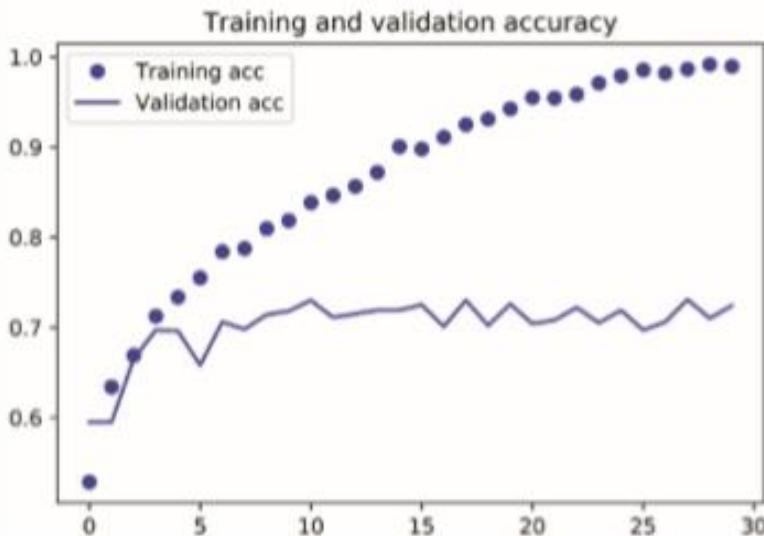
Se usa `fit_generator` si cada dato de entrada es grande y son pocos

- Es necesario guardar esta base de imágenes reducida con:

```
model.save('cat_dogs_small.h5')
```

# *Accuracy y pérdida del entrenamiento y la validación*

- Gracias a la librería `matplotlib`, se puede graficar la relación entre *accuracy* y *loss*, para detectar la idoneidad de red convolucional propuesta.

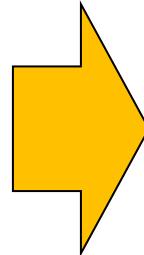


- A partir de la quinta época (de las 100 previstas) ya hay *overfitting* seguramente por tener tan pocos datos de entrada (2000 ejemplos de entrenamiento) y el *accuracy* apenas llega al **72%** ... Ya que hay *overfitting* podemos usar regularización L1/L2 o *dropout*, pero específicamente para procesamiento de imágenes, se puede usar **expansión de datos**.

# Expansión de los datos

Consiste en aumentar los datos de entrada sobre el pequeño conjunto disponible, realizando un número aleatorio de transformaciones que producen imágenes creíbles: rotaciones, traslaciones, cortes, ensanchamientos, zoom, efecto espejo, ...

```
datagen = ImageDataGenerator(  
    rotation_range=40,  
    width_shift_range=0.2,  
    height_shift_range=0.2,  
    shear_range=0.2,  
    zoom_range=0.2,  
    horizontal_flip=True,  
    fill_mode='nearest')
```



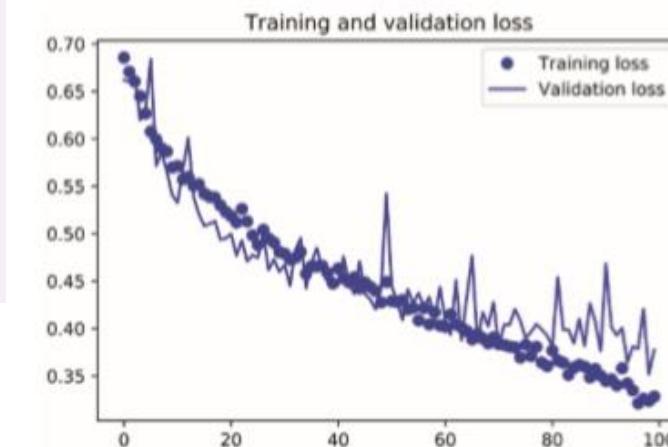
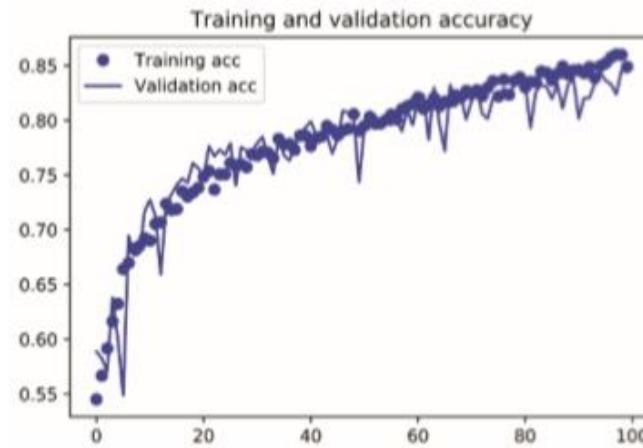
La entrada no es exactamente igual. No es infalible para combatir el *overfitting*, por lo que se puede utilizar conjuntamente *dropout* y/o regularización

- Ahora utilizaremos la misma estructura de la red convolucional anterior pero agregando la expansión de los datos de entrada y un procesamiento de *dropout* en la última capa convolucional. La expansión debe realizarse antes de definir la estructura de la red neuronal convolucional que también puede ser guardada con `model.save`.

# Resultados de la expansión

```
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu',
                      input_shape=(150, 150, 3)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
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'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Flatten())
model.add(layers.Dropout(0.5)) ←
model.add(layers.Dense(512, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))

model.compile(loss='binary_crossentropy',
              optimizer=optimizers.RMSprop(lr=1e-4),
              metrics=['acc'])
```



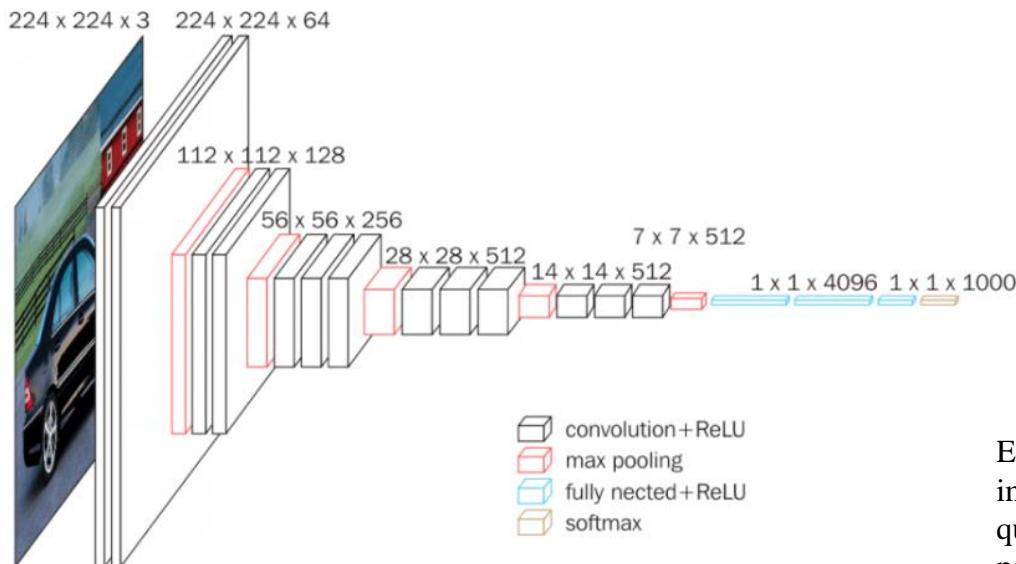
- Ahora la exactitud mejora al 82% ... Cambiando otros hiperparámetros como: número de kernel, cantidad de capas convolucionales, ... se podría llegar a un *accuracy* del 87%, el cual sigue siendo relativamente bajo ... El siguiente paso sería usar un modelo pre-entrenado

# CNN pre-entrenada

Consiste en una gran red neuronal convolucional entrenada con muchos datos de entrada, con muchas clases y lo “suficientemente general” para transferir aprendizaje.

ImageNet ofrece muchas CNN pre-entrenadas para construir otras CNN's

- Dentro de las CNN pre-entrenadas de ImageNet están: ResNet, Inception, Xception, MobileNet ... Para el ejemplo que hemos estado desarrollando, usaremos VGG16 (2014) que se entrenó con una base de datos de 14 millones de imágenes (224x224x3) de 1000 clases. El entrenamiento tomó varias semanas en GPU Titan Black de NVIDIA y consiguió una precisión del 92.7% ...

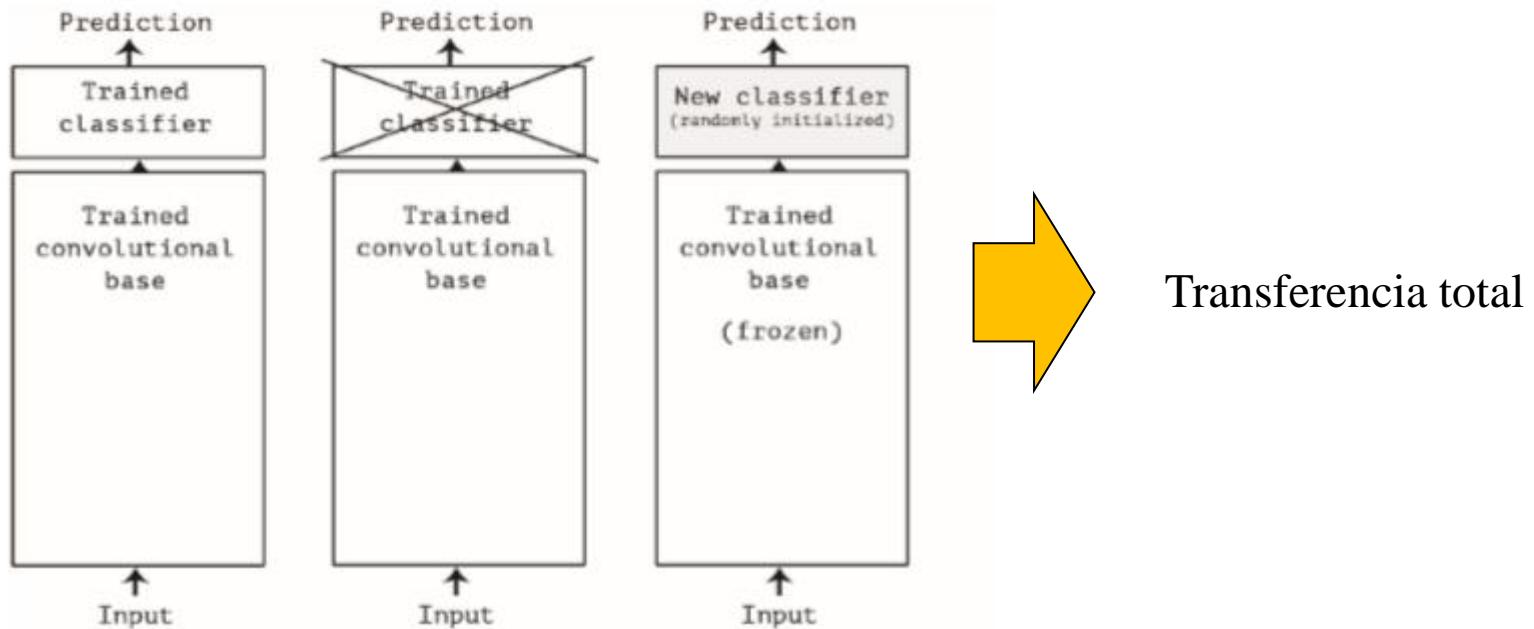


```
from keras.applications import VGG16  
  
conv_base = VGG16(weights='imagenet',  
                  include_top=False,  
                  input_shape=(150, 150, 3))
```

Este parámetro es muy importante para evitar que se modifiquen los pesos de VGG16

# Usos de CNN pre-entrenada

- Hay dos maneras de usar las CNN pre-entrenadas: por **transferencia total** o por **entonación fina**. En ambos casos se elimina la capa de clasificación y se entrena con las nuevas capas de clasificación que conviene al problema específico.



- Es claro que las primeras capas convolucionadas son más generales y además detectan más características mientras más profunda sea la convolución. Si las últimas convoluciones son demasiado específicas, es mejor entrenar esas capas con las nuevas imágenes. Esto es justamente el método de entonación fina.

# VGG16

```
>>> conv_base.summary()

Layer (type)          Output Shape         Param #
=====                ======           =====
input_1 (InputLayer)   (None, 150, 150, 3)  0
block1_conv1 (Convolution2D) (None, 150, 150, 64) 1792
block1_conv2 (Convolution2D) (None, 150, 150, 64) 36928
block1_pool (MaxPooling2D)  (None, 75, 75, 64)  0
block2_conv1 (Convolution2D) (None, 75, 75, 128) 73856
block2_conv2 (Convolution2D) (None, 75, 75, 128) 147584
block2_pool (MaxPooling2D)  (None, 37, 37, 128)  0
block3_conv1 (Convolution2D) (None, 37, 37, 256) 295168
block3_conv2 (Convolution2D) (None, 37, 37, 256) 590080
block3_conv3 (Convolution2D) (None, 37, 37, 256) 590080
block3_pool (MaxPooling2D)  (None, 18, 18, 256)  0
block4_conv1 (Convolution2D) (None, 18, 18, 512) 1180160
block4_conv2 (Convolution2D) (None, 18, 18, 512) 2359808
block4_conv3 (Convolution2D) (None, 18, 18, 512) 2359808
block4_pool (MaxPooling2D)  (None, 9, 9, 512)  0
block5_conv1 (Convolution2D) (None, 9, 9, 512) 2359808
block5_conv2 (Convolution2D) (None, 9, 9, 512) 2359808
block5_conv3 (Convolution2D) (None, 9, 9, 512) 2359808
block5_pool (MaxPooling2D)  (None, 4, 4, 512)  0
=====
Total params: 14,714,688
Trainable params: 14,714,688
Non-trainable params: 0
```

La **transferencia total** es conservando o congelando los pesos de todas las capas convolucionadas (con expansión o no). Se usa cuando la base de datos es pequeña y la pre-entrenada es similar

Para la **entonación fina** se permite el entrenamiento de las últimas capas y también se puede o no utilizar expansión de datos. Generalmente se usa cuando la base de datos es pequeña y la pre-entrenada es diferente o cuando la base de datos es grande y la pre-entrenada similar.

Para extraer las características de VGG16 se debe llamar el método `conv_base.predict...` para usar la salida de VGG16 como entrada a la red de clasificación

# Transferencia total sin expansión de datos

```
from keras import models
from keras import layers
from keras import optimizers

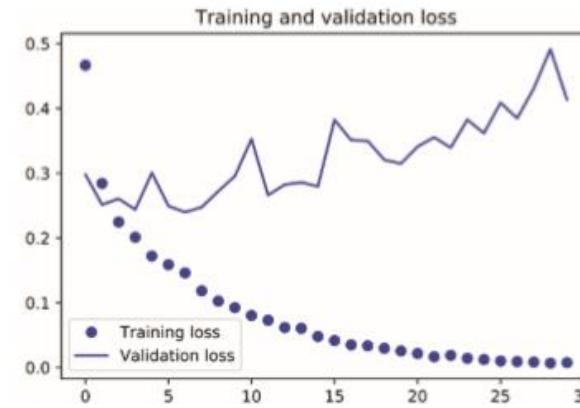
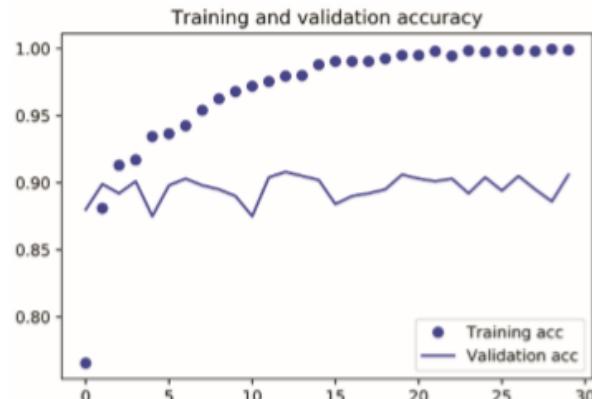
model = models.Sequential()
model.add(layers.Dense(256, activation='relu', input_dim=4 * 4 * 512)) ←
model.add(layers.Dropout(0.5))
model.add(layers.Dense(1, activation='sigmoid'))

model.compile(optimizer=optimizers.RMSprop(lr=2e-5), ←
              loss='binary_crossentropy',
              metrics=['acc'])

history = model.fit(train_features, train_labels,
                     epochs=30,
                     batch_size=20,
                     validation_data=(validation_features, validation_labels))
```

Con la dimensión de la salida de la red pre-entrenada VGG16, es decir, 8192 neuronas ...

Es importante que la tasa de aprendizaje sea baja ...



- Esta vez la precisión aumenta al **90%** pero continúa el *overfitting* en apenas 5 épocas a pesar del *dropout* que se colocó en la primera capa de clasificación. En consecuencia, es necesaria la expansión de datos.

# *Transferencia total con expansión de datos*

```
from keras import models
from keras import layers

model = models.Sequential()
model.add(conv_base)
model.add(layers.Flatten())
model.add(layers.Dense(256, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

Aún con una red pre-entrenada pero con expansión de datos, se requiere mucho tiempo de procesamiento por lo que es imprescindible usar GPU's ... Se agrega VGG16 antes de las capas de clasificación

```
>>> model.summary()

Layer (type)          Output Shape       Param #
=====
vgg16 (Model)         (None, 4, 4, 512)    14714688
flatten_1 (Flatten)   (None, 8192)        0
dense_1 (Dense)       (None, 256)         2097408
dense_2 (Dense)       (None, 1)           257
=====
Total params: 16,812,353
Trainable params: 16,812,353
Non-trainable params: 0
```

Es de notar la gran cantidad de parámetros (casi 17 millones ...) por ello la necesidad de GPU's

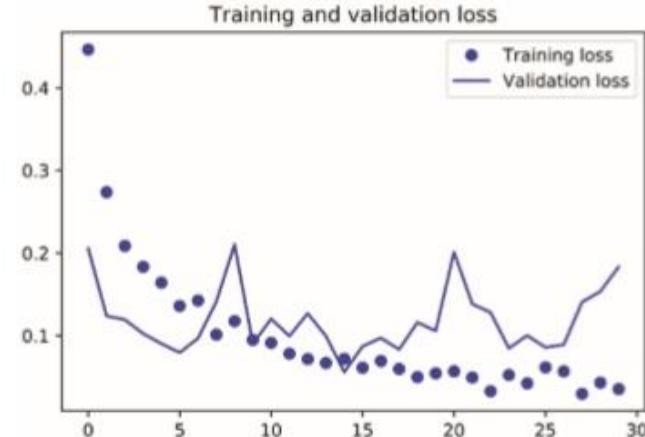
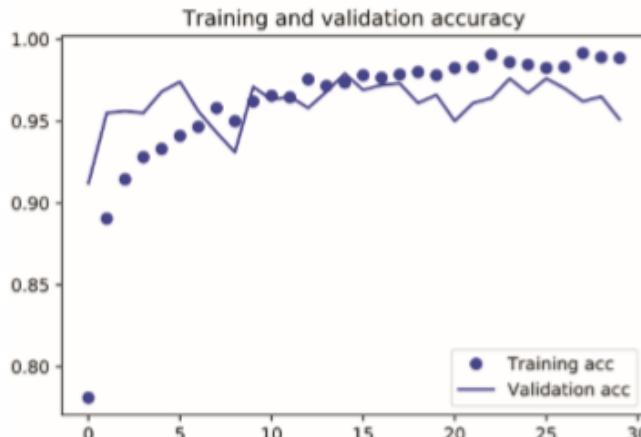
- Con transferencia total, VGG16 debe tener sus pesos congelados lo que se indica al momento de la carga de imágenes pre-entrenadas. Así que solamente se modifican los pesos de las capas de clasificación. Esta vez se logra una precisión del (96%) ...

# Transferencia total con expansión de datos

```
from keras.preprocessing.image import ImageDataGenerator  
  
train_datagen = ImageDataGenerator(  
    rescale=1./255,  
    rotation_range=40,  
    width_shift_range=0.2,  
    height_shift_range=0.2,  
    shear_range=0.2,  
    zoom_range=0.2,  
    horizontal_flip=True,  
    fill_mode='nearest')
```

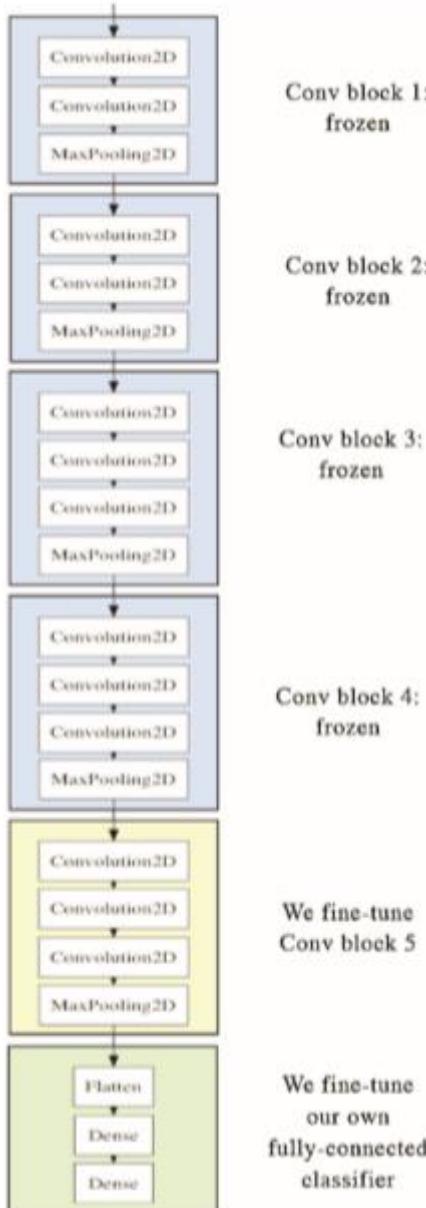
```
import matplotlib.pyplot as plt  
  
acc = history.history['acc']  
val_acc = history.history['val_acc']  
loss = history.history['loss']  
val_loss = history.history['val_loss']  
  
epochs = range(1, len(acc) + 1)  
  
plt.plot(epochs, acc, 'bo', label='Training acc')  
plt.plot(epochs, val_acc, 'b', label='Validation acc')  
plt.title('Training and validation accuracy')  
plt.legend()  
  
plt.figure()  
  
plt.plot(epochs, loss, 'bo', label='Training loss')  
plt.plot(epochs, val_loss, 'b', label='Validation loss')  
plt.title('Training and validation loss')  
plt.legend()  
  
plt.show()
```

Así se expande la base de datos de imágenes de perros y gatos ...

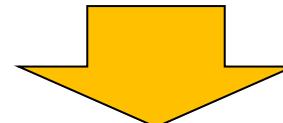


No se aprecia *overfitting* ... sin embargo se puede mejorar aún más el *accuracy*

# Entonación fina



Con esta estrategia se permite el ajuste de pesos del último tramo de capas convolucionadas porque se suponen que son las más específicas, dejando la mayoría del modelo pre-entrenado. Los nuevos datos ajustan sólo parte de las capas convolucionada y por supuesto las capas de clasificación.



La fase de entrenamiento sólo será sobre los dos últimos bloques de la red neuronal completa ...

```
conv_base.trainable = True  
  
set_trainable = False  
for layer in conv_base.layers:  
    if layer.name == 'block5_conv1':  
        set_trainable = True  
    if set_trainable:  
        layer.trainable = True  
    else:  
        layer.trainable = False
```

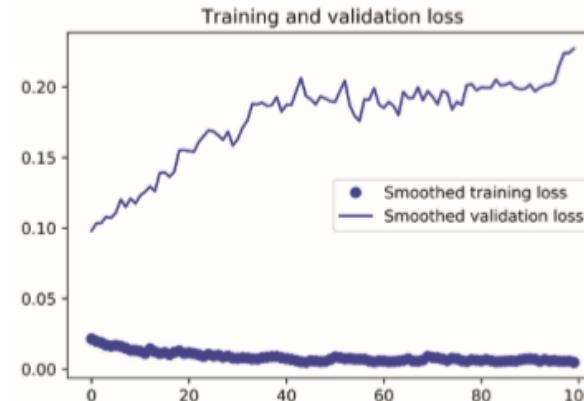
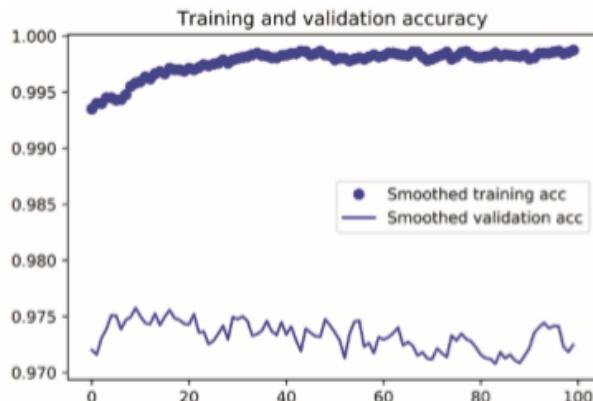
Se ha demostrado, empíricamente, que una tasa de aprendizaje baja permite mantener parte de lo aprendido por la red pre-entrenada

# Entonación fina

```
model.compile(loss='binary_crossentropy',
              optimizer=optimizers.RMSprop(lr=1e-5),
              metrics=['acc'])

history = model.fit_generator(
    train_generator,
    steps_per_epoch=100,
    epochs=100,
    validation_data=validation_generator,
    validation_steps=50)
```

Con estas condiciones de entrenamiento y suavizando las curvas de precisión y pérdida (opción `smoothed_points`) se aprecia una mejora en el *accuracy* del 97%.



- En resumen el *accuracy* fue:

Con pocos datos	72%	
Pocos datos con expansión de datos	82%	
Transferencia total sin expansión	90%	Mejora progresiva ...
Transferencia total con expansión	96%	
Entonación fina con expansión	97%	

# *Entonación fina o Transferencia total*

Base de datos grande y pre-entrenada diferente

Entrenar una gran red completa

Base de datos grande y pre-entrenada similar

Entonación fina

Base de datos pequeña y pre-entrenada diferente

Entonación fina

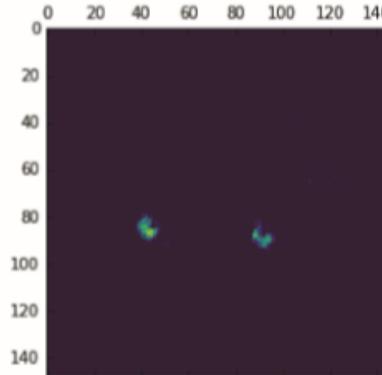
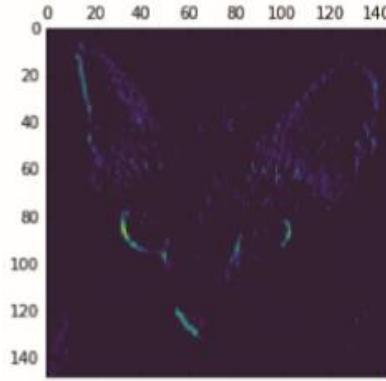
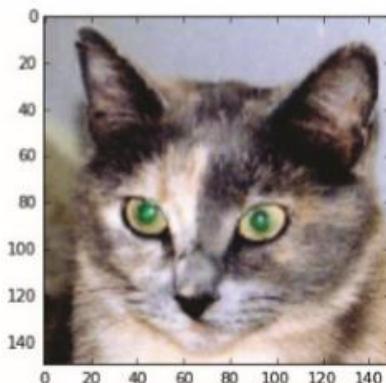
Base de datos pequeña y pre-entrenada similar

Transferencia total

# *Visualización de las capas convolucionadas*

Es conocida la incapacidad de las redes neuronales para explicar sus respuestas. La consulta en reversa, a partir de la salida remontando hasta la entrada, no aporta explicación. Esta limitación conocida como enfoque **caja negra**, no permite a una red neuronal, por su estructura interna, extraer la justificación a partir de su respuesta.

- Afortunadamente en las redes neuronales convolucionales, por tratarse de imágenes, se puede visualizar tres aspectos: (1) la transición de las imágenes producto de los kernels (2) como se van transformando los filtros (3) como localizar objetos particulares.
- Lo ideal en el primer caso, sería mostrar como una imagen va cambiando en su transitar por cada capa convolucionada producto de cada kernel ...



# Transición de las imágenes por capa

```
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu',
                      input_shape=(150, 150, 3)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
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'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Flatten())
model.add(layers.Dropout(0.5))
model.add(layers.Dense(512, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))

model.compile(loss='binary_crossentropy',
              optimizer=optimizers.RMSprop(lr=1e-4),
              metrics=['acc'])
```

```
>>> from keras.models import load_model
>>> model = load_model('cats_and_dogs_small_2.h5')
>>> model.summary() # As a reminder.

Layer (type)                 Output Shape              Param # 
conv2d_5  (Conv2D)            (None, 148, 148, 32)    896    
maxpooling2d_5 (MaxPooling2D) (None, 74, 74, 32)      0      
conv2d_6  (Conv2D)            (None, 72, 72, 64)     18496   
maxpooling2d_6 (MaxPooling2D) (None, 36, 36, 64)      0      
conv2d_7  (Conv2D)            (None, 34, 34, 128)    73856   
maxpooling2d_7 (MaxPooling2D) (None, 17, 17, 128)    0      
conv2d_8  (Conv2D)            (None, 15, 15, 128)    147584  
maxpooling2d_8 (MaxPooling2D) (None, 7, 7, 128)     0      
flatten_2 (Flatten)          (None, 6272)           0      
dropout_1 (Dropout)          (None, 6272)           0      
dense_3  (Dense)             (None, 512)            3211776  
dense_4  (Dense)             (None, 1)              513    

Total params: 3,453,121
Trainable params: 3,453,121
Non-trainable params: 0
```

Partiendo de la primera red neuronal convolucional, sin ninguna de las variantes, se puede, para una imagen, visualizar el resultado de todos los kernel por cada capa ...

Se trabajará sobre la imagen de gato 1700

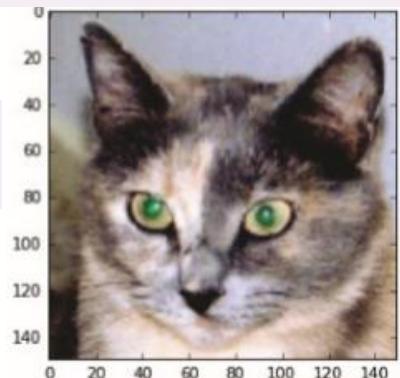
```
img_path = '/Users/fchollet/Downloads/cats_and_dogs_small/test/cats/cat.1700.jpg'

# We preprocess the image into a 4D tensor
from keras.preprocessing import image
import numpy as np

img = image.load_img(img_path, target_size=(150, 150))
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)
# Remember that the model was trained on inputs
# that were preprocessed in the following way:
img_tensor /= 255.

# Its shape is (1, 150, 150, 3)
print(img_tensor.shape)
```

```
import matplotlib.pyplot as plt
plt.imshow(img_tensor[0])
```



# Separación de las capas convolucionadas

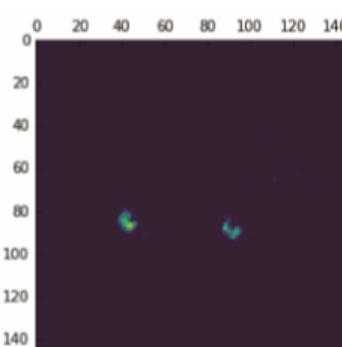
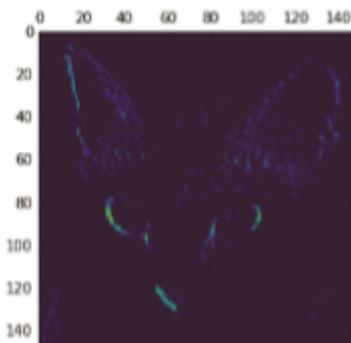
```
from keras import models  
  
# Extracts the outputs of the top 8 layers:  
layer_outputs = [layer.output for layer in model.layers[:8]]  
# Creates a model that will return these outputs, given the model input:  
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)  
  
# This will return a list of 8 Numpy arrays:  
# one array per layer activation  
activations = activation_model.predict(img_tensor)
```

Una entrada varias salidas

Define las 8 capas, 4 convolucionales y 4 de *max-pooling*, con el modelo Model en lugar de usar Sequential.

- Así la primera capa se accede con la variable `activations[0]`, cuya dimensión es  $(1, 148, 148, 32)$
- En consecuencia, se puede visualizar el resultado de la salida de cualquier filtro o kernel con la función `matshow` de `matplotlib`:

```
plt.matshow(activations[0, :, :, 4], cmap='viridis')  
plt.matshow(activations[0, :, :, 7], cmap='viridis')
```



# Visualización de cada filtro por capa

```
# These are the names of the layers, so can have them as part of our plot
layer_names = []
for layer in model.layers[:8]:
    layer_names.append(layer.name)

images_per_row = 16

# Now let's display our feature maps
for layer_name, layer_activation in zip(layer_names, activations):
    # This is the number of features in the feature map
    n_features = layer_activation.shape[-1]

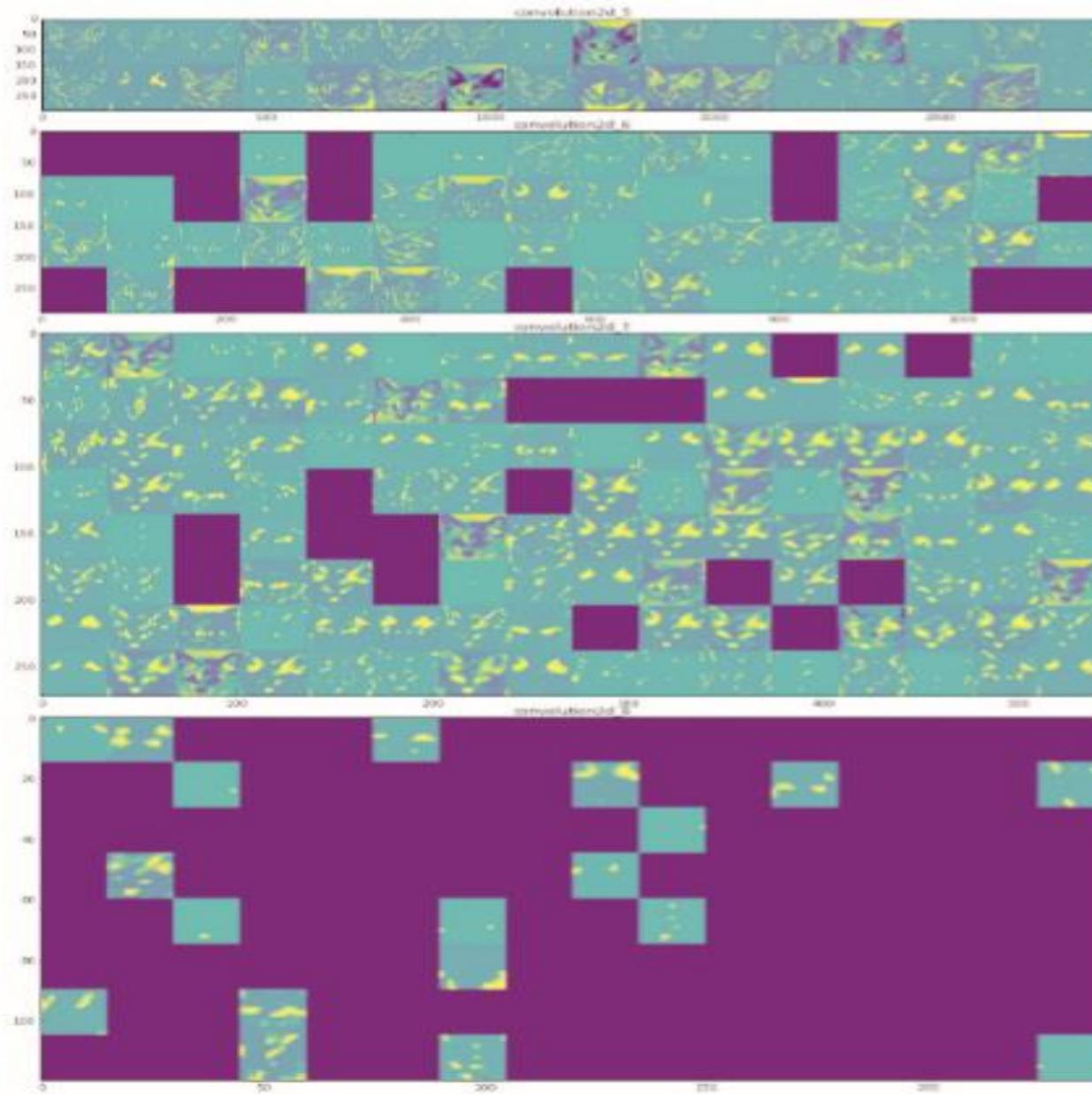
    # The feature map has shape (1, size, size, n_features)
    size = layer_activation.shape[1]

    # We will tile the activation channels in this matrix
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

    # We'll tile each filter into this big horizontal grid
    for col in range(n_cols):
        for row in range(images_per_row):
            channel_image = layer_activation[0,
                                              :, :,
                                              col * images_per_row + row]
            # Post-process the feature to make it visually palatable
            channel_image -= channel_image.mean()
            channel_image /= channel_image.std()
            channel_image += 64
            channel_image += 128
            channel_image = np.clip(channel_image, 0, 255).astype('uint8')
            display_grid[col * size : (col + 1) * size,
                         row * size : (row + 1) * size] = channel_image

    # Display the grid
    scale = 1. / size
    plt.figure(figsize=(scale * display_grid.shape[1],
                       scale * display_grid.shape[0]))
    plt.title(layer_name)
    plt.grid(False)
    plt.imshow(display_grid, aspect='auto', cmap='viridis')
```

# *Transición del gato 1700 por capas*



En la primera capa se ven claramente rasgos de la imagen original. A medida se avanza. Mientras la capa es más profunda más abstracto es la extracción de la característica

Los filtros en violeta significa que la información que debería filtrarse no está presente en la imagen

# *Visualización de filtros o kernels*

- Así como se puede ver la salida de cada capa convolucional, también es posible visualizar los filtros aprendidos. La idea es construir una función de pérdida que maximice el valor de cada filtro por cada capa convolucional.
- El objetivo es mostrar todos los filtros de la red pre-entrenada VGG16 para los primeros cuatro niveles de capas convolucionales ... Comencemos con sólo el primer filtro del 3<sup>er</sup> bloque de la primera capa convolucional: `block3_conv1`.

```
from keras.applications import VGG16
from keras import backend as K

model = VGG16(weights='imagenet',
               include_top=False)

layer_name = 'block3_conv1'
filter_index = 0

layer_output = model.get_layer(layer_name).output
loss = K.mean(layer_output[:, :, :, filter_index])
```

- Se deben hacer algunos ajustes a los pesos de los filtros para maximizarlos con ascenso de gradiente.

# Funciones auxiliares

```
def deprocess_image(x):
    # normalize tensor: center on 0., ensure std is 0.1
    x -= x.mean()
    x /= (x.std() + 1e-5)
    x *= 0.1

    # clip to [0, 1]
    x += 0.5
    x = np.clip(x, 0, 1)

    # convert to RGB array
    x *= 255
    x = np.clip(x, 0, 255).astype('uint8')
    return x
```

```
def generate_pattern(layer_name, filter_index, size=150):
    # Build a loss function that maximizes the activation
    # of the nth filter of the layer considered.
    layer_output = model.get_layer(layer_name).output
    loss = K.mean(layer_output[:, :, :, filter_index]) ← Ajuste de los pesos

    # Compute the gradient of the input picture wrt this loss
    grads = K.gradients(loss, model.input)[0] ← Aprendizaje para maximizar filtros

    # Normalization trick: we normalize the gradient
    grads /= (K.sqrt(K.mean(K.square(grads))) + 1e-5) ← Ajuste de los pesos

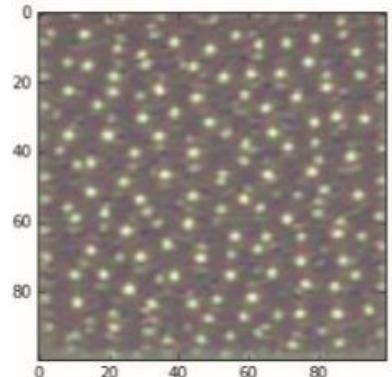
    # This function returns the loss and grads given the input picture
    iterate = K.function([model.input], [loss, grads])

    # We start from a gray image with some noise
    input_img_data = np.random.random((1, size, size, 3)) * 20 + 128.

    # Run gradient ascent for 40 steps
    step = 1.
    for i in range(40):
        loss_value, grads_value = iterate([input_img_data])
        input_img_data += grads_value * step

    img = input_img_data[0]
    return deprocess_image(img)
```

```
plt.imshow(generate_pattern('block3_conv1', 0))
```



# Visualización de todos los filtros

- Ahora el código para mostrar los filtros de la primera capa convolucional de los cuatro primeros bloques (se excluye el último bloque porque sobre este se hace entonación fina) ...

```
layer_name = 'block1_conv1'
size = 64
margin = 5

# This a empty (black) image where we will store our results.
results = np.zeros((8 * size + 7 * margin, 8 * size + 7 * margin, 3))

for i in range(8): # iterate over the rows of our results grid
    for j in range(8): # iterate over the columns of our results grid
        # Generate the pattern for filter `i + (j * 8)` in `layer_name`
        filter_img = generate_pattern(layer_name, i + (j * 8), size=size)

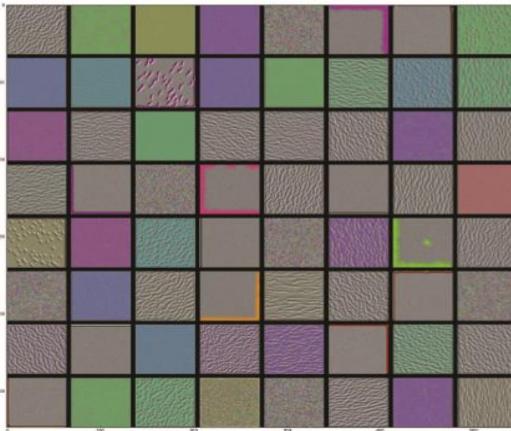
        # Put the result in the square `(i, j)` of the results grid
        horizontal_start = i * size + i * margin
        horizontal_end = horizontal_start + size
        vertical_start = j * size + j * margin
        vertical_end = vertical_start + size
        results[horizontal_start: horizontal_end, vertical_start: vertical_end, :] = filter_img

# Display the results grid
plt.figure(figsize=(20, 20))
plt.imshow(results)
```

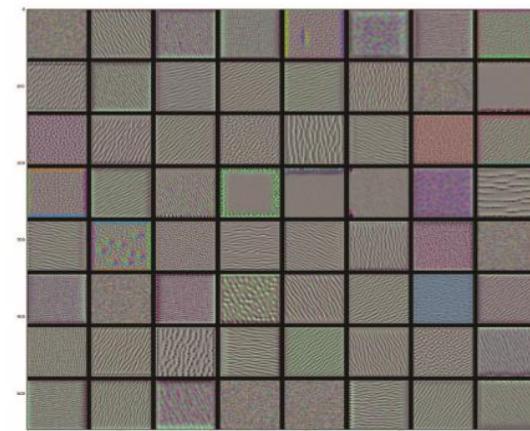
- Sólo se consideran 64 filtros en cada capa convolucional ...

# *Filtros de la primera capa convolucional por cada bloque*

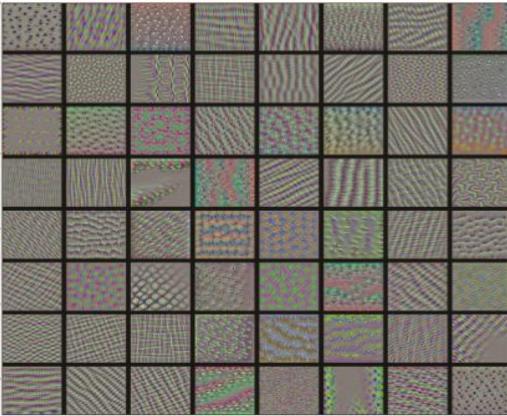
Primera capa convolucional  
1<sup>er</sup> bloque



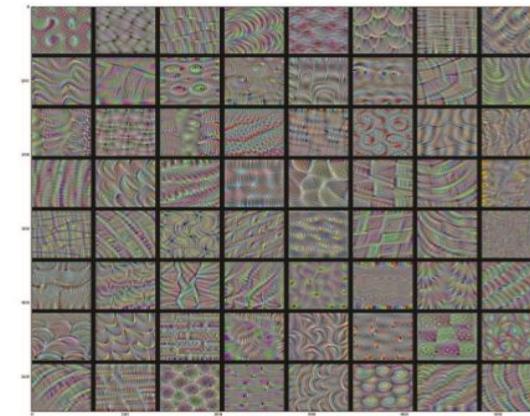
Primera capa convolucional  
2<sup>do</sup> bloque



Primera capa convolucional  
3<sup>er</sup> bloque

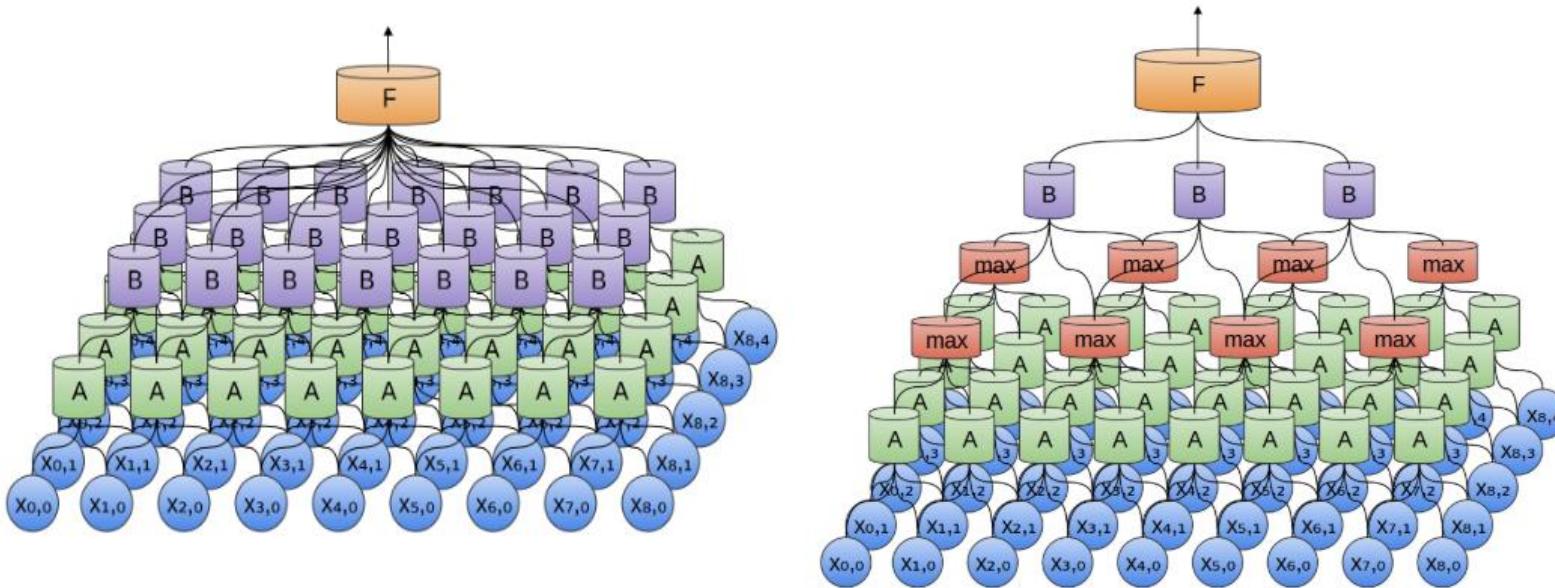


Primera capa convolucional  
4<sup>to</sup> bloque



Diversos métodos para visualización de filtros

# Recapitulando CNN



- Excelente herramienta para visión por computadora que puede aplicarse a navegación robótica. Además hay ciertas modificaciones a las CNN que pueden ser utilizadas para el arte y el entretenimiento ([deepArt](#) y [Prisma](#))
- Para los videos, en lugar de imágenes, el inconveniente es el seguimiento, de la secuencia de imágenes, en la línea de tiempo, es decir, las CNN no tienen la capacidad de recordar que sucedió en el pasado para así procesar patrones en el tiempo ... afortunadamente existen las Redes Neuronales Recurrente ...