

Algoritmos avanzados de Deep Learning.

Caso práctico



LookStudio (<https://www.freepik.es/autor/lookstudio>) (CC BY-SA (<http://creativecommons.org/licenses/?lang=es>))

Miguel y Lorena forman parte del equipo de desarrollo del departamento tecnológico de la empresa Pick&Deliver, y llevan varios meses desarrollando y probando aplicaciones de inteligencia artificial para mejorar procesos y optimizar recursos en la empresa.

Una vez han probado API (Application Programming Interfaces) de proveedores y modelos básicos de aprendizaje automático propios, antes de empezar a implementar y a hacer el despliegue en producción, deben cubrir también las opciones más avanzadas del deep learning, pues la comunidad está compartiendo impresiones muy positivas sobre los resultados que se consiguen con algunos modelos, especialmente en aplicaciones para datos no estructurados, como las imágenes o el lenguaje natural.

En esta unidad vamos a descubrir técnicas avanzadas de aprendizaje automático profundo que han ido surgiendo en los últimos años gracias al trabajo e investigación de la comunidad en torno al desarrollo de inteligencia artificial. En concreto, veremos en detalle:

- ✓ Redes neuronales convolucionales, sus parámetros, cómo se combinan con las de tipo neuronal y otras capas auxiliares a éstas.
- ✓ Modelos convolucionales ya pre-entrenados que podemos particularizar para un desarrollo propio
- ✓ Redes neuronales recurrentes para análisis del lenguaje.
- ✓ Redes generativas adversarias, que emulan la capacidad creativa del ser humano.

Al finalizar esta unidad, ya tendrás una imagen completa de los modelos avanzados que se están implementando en muchos desarrollos de proyectos actuales, y que marcan la diferencia con respecto a sistemas clásicos.



Ministerio de Educación y Formación Profesional (<https://www.educacionyfp.gob.es/portada.html>) (Dominio público)

Materiales formativos de FP Online propiedad del Ministerio de Educación y Formación Profesional.

[Aviso Legal](http://www.educacionyfp.gob.es/fpadistancia/comunes/aviso-legal.html) (<http://www.educacionyfp.gob.es/fpadistancia/comunes/aviso-legal.html>)

1.- Redes neuronales convolucionales.

Caso práctico



LookStudio (<https://www.freepik.es/autor/lookstudio>) (CC BY-SA (<http://creativecommons.org/licenses/?lang=es>))

"Vamos a empezar con las redes neuronales convolucionales" comenta Miguel al equipo en el sprint log (breve reunión de pie) de esa mañana. "Es una arquitectura fascinante, capaz de simplificar la interpretación de las imágenes y hacer el modelo tremadamente potente"

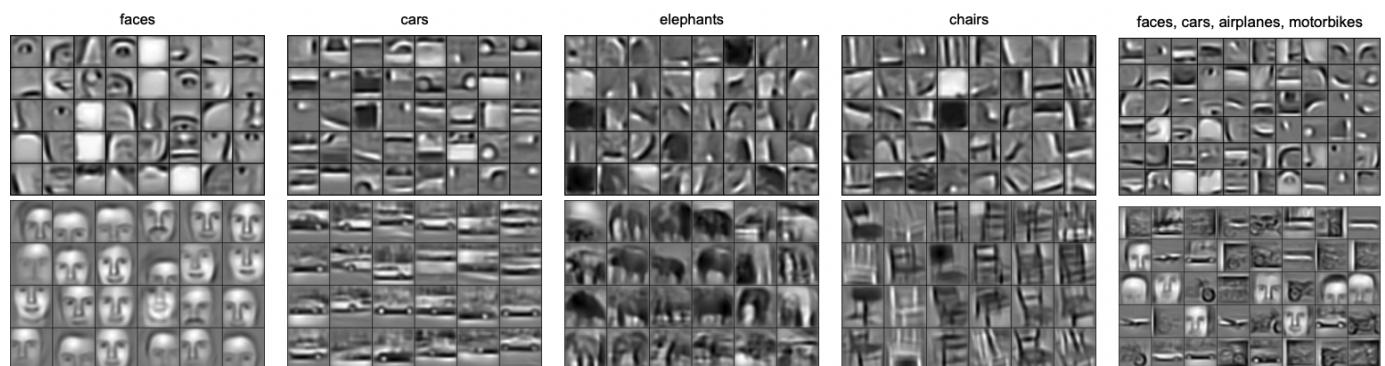
Aunque solo Lorena entiende a qué se refiere, Miguel contagia su entusiasmo al resto del equipo, que le desean suerte y le piden que les vaya manteniendo al día en el progreso que vaya haciendo.

"Parece casi cosa de magia" dice Andrew, que se suele encargar de la parte de sistemas de la empresa "Me encantaría aprender más sobre esto"

"Tranquilo, ¡Os vais a hartar de oírnos hablar sobre ello estos días!" dice Lorena entre risas de todo el equipo.

Las redes neuronales convolucionales han revolucionado el campo de la visión artificial, logrando resultados que muchas veces son verdaderamente sorprendentes. En una primera aproximación, podríamos decir que la principal ventaja de este tipo de redes es que se establecen las capas con un orden jerárquico, estando cada capa "especializada" en detectar un nivel de comprensión de la imagen. Es decir, la primera capa podría detectar simplemente bordes, la segunda capa podría ser capaz de detectar elementos circulares, prismáticos,... y una tercera capa podría estar especializada en distinguir ruedas, puertas,y otras partes de un vehículo.

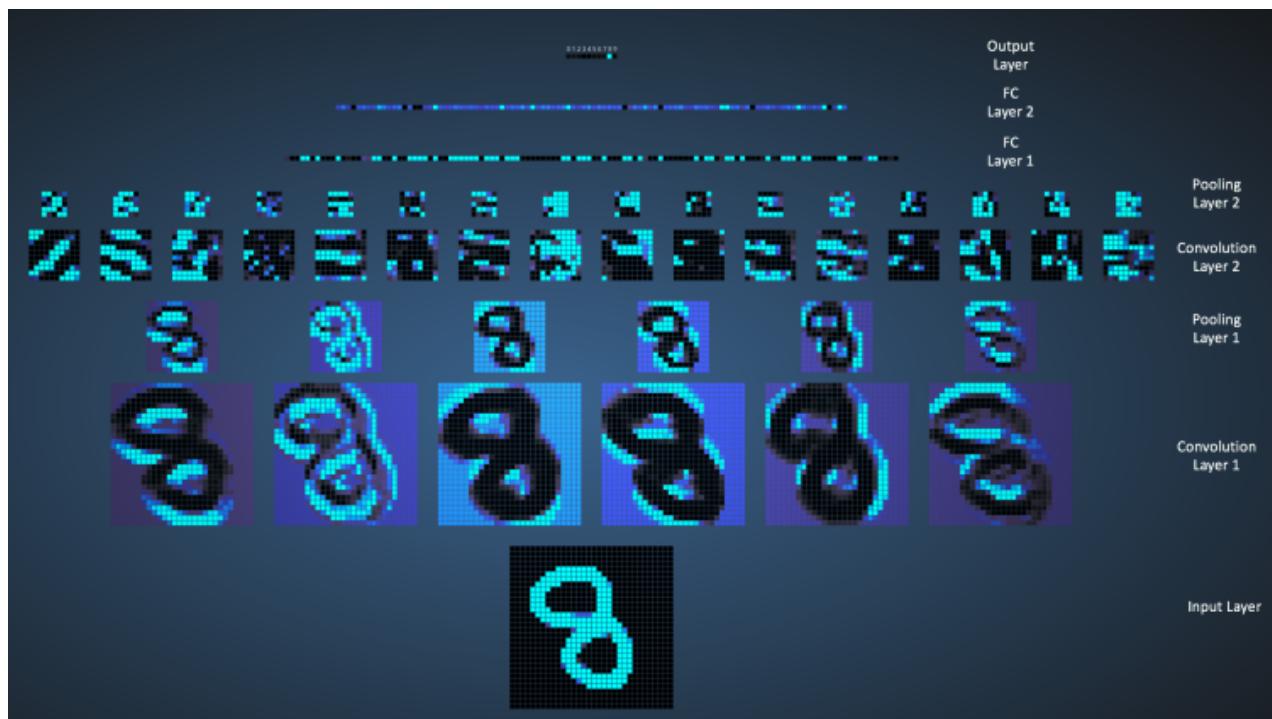
Otra ventaja de este tipo de modelos, es que los patrones que aprenden a reconocer pueden estar en cualquier parte de la imagen. Mientras una red neuronal profunda normal solo puede categorizar un elemento en una imagen si reconoce la imagen completa como continente de ese patrón, identificándolo en la zona de la imagen que suele estar, las redes convolucionales pueden haber aprendido a reconocer un elemento que suele estar en una esquina de la imagen y reconocerlo después en la parte central de otra imagen, y categorizarlo de forma única.



Andrew Ng (<http://web.eecs.umich.edu/~honglak/icml09-ConvolutionalDeepBeliefNetworks.pdf>) (CC BY-SA (<http://creativecommons.org/licenses/?lang=es>))

Recomendación

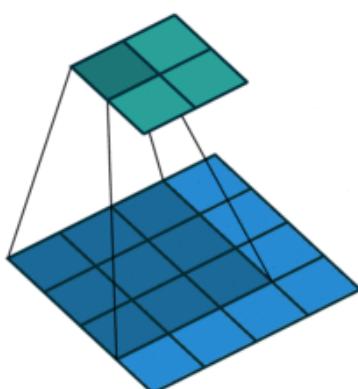
El profesor de Ciencias de la Computación Adam Harley, ha creado una representación interactiva de redes neuronales normales y redes neuronales convolucionales. Se basa en el famoso problema de reconocimiento de dígitos manuscritos. En [la interfaz](https://adamharley.com/nn_vis/cnn/2d.html) (https://adamharley.com/nn_vis/cnn/2d.html), se te pide que dibujes un dígito, y puedes ver cómo lo "ve" cada nodo y capa de la red, apareciendo también las conexiones entre ellas. Te recomendamos que entres y dediques tiempo a probarlo y explorar las distintas capas, que veremos en las siguientes secciones. Una vez hayas finalizado la unidad, puedes volver a la demo y volver a visualizar el modelo gráfico, ya con el conocimiento de la función y comportamiento de cada capa.



[A. Harley](https://github.com/aharley/nn_vis) (https://github.com/aharley/nn_vis) (CC BY (<http://creativecommons.org/licenses/?lang=es>))

Puedes consultar toda la documentación del proyecto en el [repositorio nn_vis en GitHub](https://github.com/aharley/nn_vis) (https://github.com/aharley/nn_vis).

1.1.- La capa convolucional.



[vdumoulin](https://github.com/vdumoulin/conv_arithmetic/blob/master/gif/no_padding_no_strides.gif)
(https://github.com/vdumoulin/conv_arithmetic/blob/master/gif/no_padding_no_strides.gif)
(CC BY (<http://creativecommons.org/licenses/?lang=es>))

La operación convolucional genera un mapa de nuevas variables aplicando una misma transformación a parches de variables del mapa de entrada. Es el funcionamiento típico de un filtro. Este mapa de salida, será un tensor de 2 o 3 dimensiones, a diferencia de las capas neuronales, que daban vectores unidimensionales de salida.

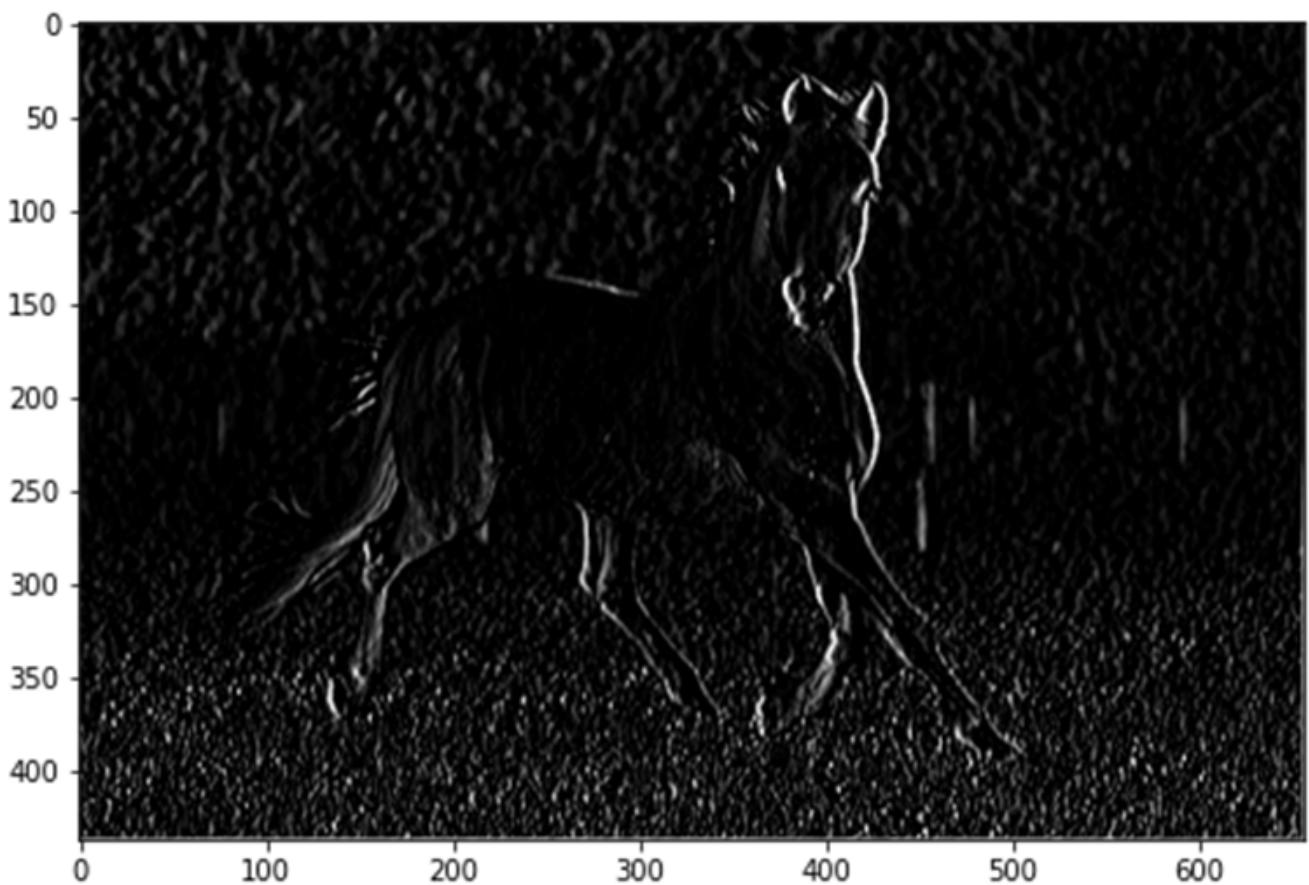
Esta transformación o filtro, se irá ajustando durante el entrenamiento, para extraer las características más fundamentales de la imagen o mapa de entrada, lo cual hace la red más ágil, pero también más precisa después en sus predicciones.

¿Y cómo funcionan estas transformaciones? En el siguiente ejemplo, puedes ver cómo actuaría un filtro de líneas verticales sobre la imagen de un caballo:



Miguel Fernández Zafra (<https://towardsdatascience.com/understanding-convolutions-and-pooling-in-neural-networks-a-simple-explanation-885a2d78f211>) (CC BY (<http://creativecommons.org/licenses/?lang=es>))

Vertical edge filter



Miguel Fernández Zafra (<https://towardsdatascience.com/understanding-convolutions-and-pooling-in-neural-networks-a-simple-explanation-885a2d78f211>) (CC BY (<http://creativecommons.org/licenses/?lang=es>))

Las capas convolucionales tienen dos parámetros principales:

- ✓ Dimensiones de la ventana que va aplicando la transformación: suelen ser de 3x3 o de 5x5.
- ✓ Profundidad del mapa de salida: es el número de filtros en paralelo que tiene la capa y que darán otros tantos mapas de salida.

En Keras, programamos este tipo de capa con estos parámetros:

```
Conv2D(output_depth, (window_height, window_width))
```

donde *output_depth* es la profundidad (recuerda, el número de imágenes de salida, igual que las capas neuronales tenían número de neuronas) y *window_height* y *window_width* son las dimensiones de la matriz de transformación o filtro.



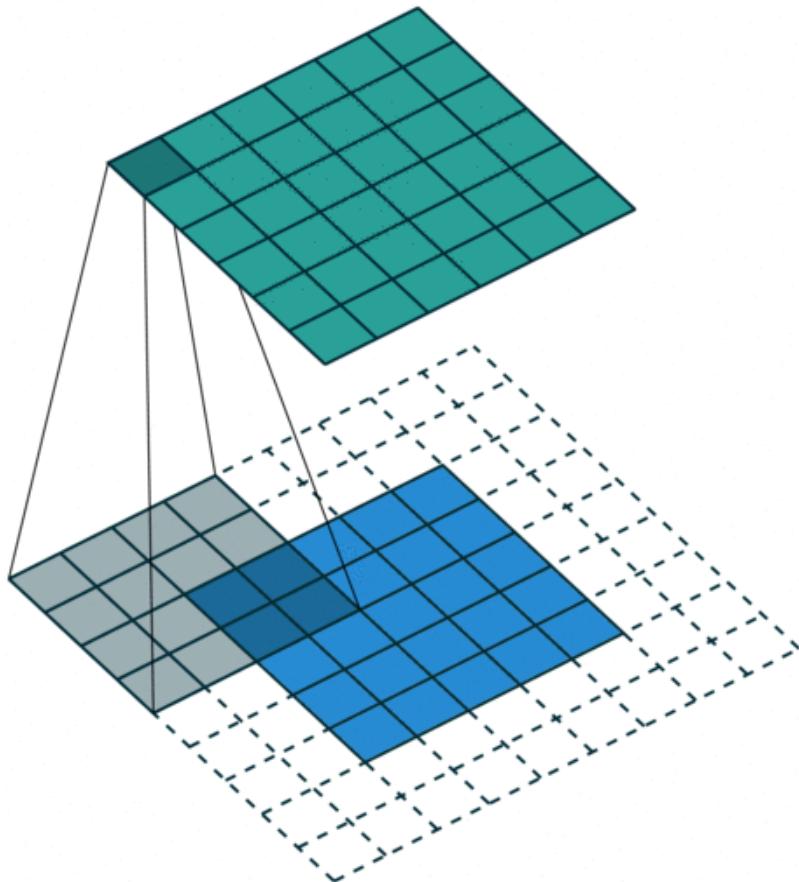
[Fergus](https://cs.nyu.edu/~fergus/tutorials/deep_learning_cvpr12/) (https://cs.nyu.edu/~fergus/tutorials/deep_learning_cvpr12/) (CC BY (<http://creativecommons.org/licenses/?lang=es>))

En esta animación se ve bastante bien cómo funciona el filtro. Vamos pasando esa ventana de 3x3 o 5x5, deslizando línea a línea sobre la imagen de entrada y extrayendo el resultado de la operación para cada conjunto de pixels.

No siempre van a coincidir las dimensiones de la imagen de salida con la de entrada, dependiendo esto de dos efectos: padding y strides

Efecto borde

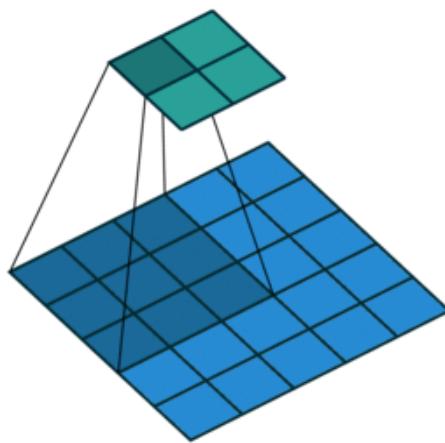
Un filtro de 3x3 no puede aplicarse centrado en los píxeles de los bordes, porque nos faltarían elementos para el cálculo. Es restricción de posiciones de la ventana del filtro, hace que se genere una imagen de salida de menor tamaño. Esto se resuelve con la técnica del padding, que consiste en añadir de forma artificial, unos píxeles extra de borde, aleatorios, que no aporten ningún patrón concreto.



[vdumoulin](https://github.com/vdumoulin/conv_arithmetic/blob/master/gif/arbitrary_padding_no_strides.gif) (https://github.com/vdumoulin/conv_arithmetic/blob/master/gif/arbitrary_padding_no_strides.gif) (CC BY
(<http://creativecommons.org/licenses/?lang=es>))

Uso de strides

Podemos hacer que la ventana de la convolución no vaya recorriendo los píxeles seguidos, sino que se vaya saltando un cierto número de ellos.



[vdumoulin](https://github.com/vdumoulin/conv_arithmetic/blob/master/gif/no_padding_strides.gif)
(https://github.com/vdumoulin/conv_arithmetic/blob/master/gif/no_padding_strides.gif).
(CC BY (<http://creativecommons.org/licenses/?lang=es>)))

Autoevaluación

¿Cómo llamamos a la técnica de añadir bordes artificiales a las imágenes para que la operación de convolución no reduzca el tamaño de la imagen de salida?

- Stride
- Depth
- Padding

El stride es el número de elementos que se saltan en cada aplicación del filtro

La profundidad o depth es el número de filtros distintos que se van a aplicar en la capa y que nos da ese número de imágenes de salida

Opción correcta

Solución

1. Incorrecto
2. Incorrecto
3. Opción correcta

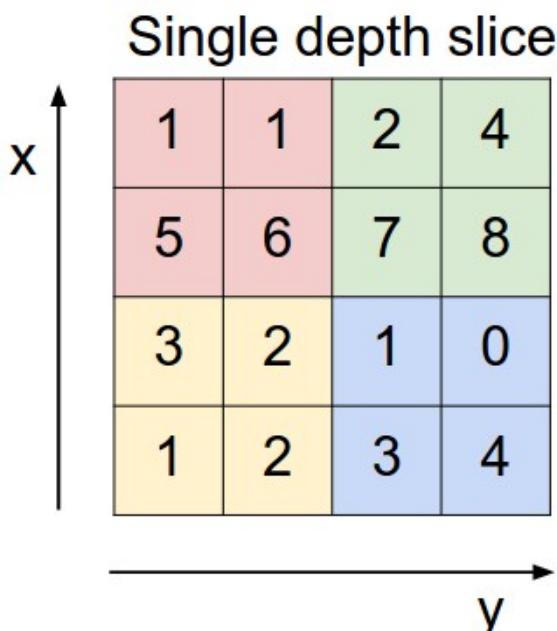
1.2.- La capa max-pooling.

Es bastante común introducir una capa de "pooling" o de reducción entre las capas convolucionales, para mantener bajo control el tamaño de la red y por tanto, del número de coeficientes o pesos en juego. Porque cada vez que se aplica una capa convolucional, hay un desdoble debido a la profundidad que aplicamos. Como viste en la unidad anterior, si la red es muy grande, aumenta el grado de overfitting y además se multiplican las cantidades de cómputo.

En realidad, las capas de pooling son muy parecidas a las de convolución excepto porque la operación de transformación es bastante simple (en general va a ser el valor máximo) y no hay desdoble. Además, se suele aplicar una ventana de 2×2 con un stride de dos elementos, lo cual reduce el tamaño a la mitad.

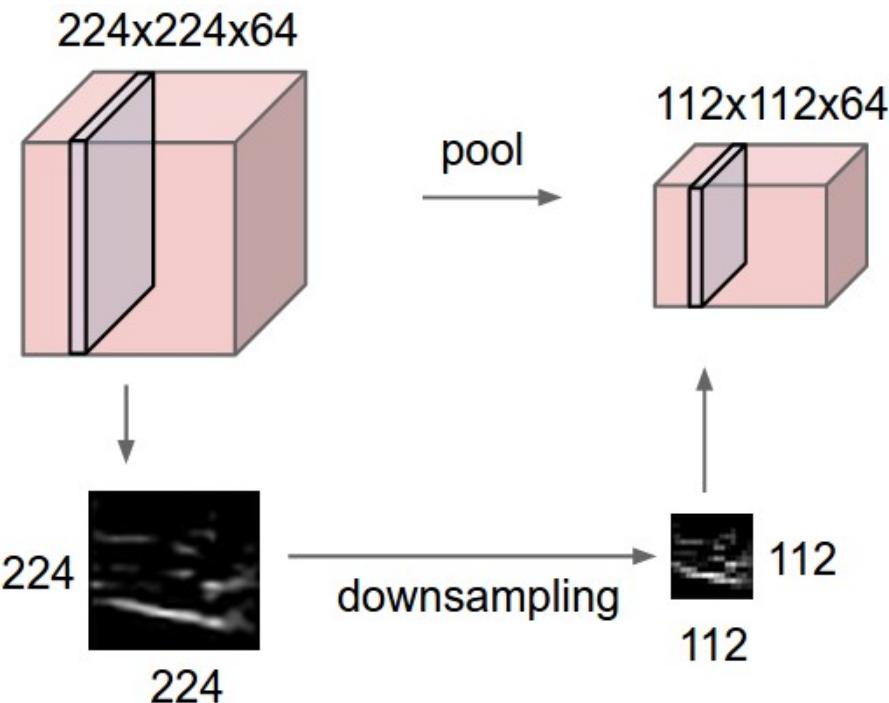
Estos dos parámetros se introducen en la capa de esta manera:

```
MaxPooling2D(window_dim, strides)
```



max pool with 2×2 filters
and stride 2

| | |
|---|---|
| 6 | 8 |
| 3 | 4 |



[cs231n \(<https://cs231n.github.io/convolutional-networks/>\)](https://cs231n.github.io/convolutional-networks/) ([CC BY-SA \(<http://creativecommons.org/licenses/?lang=es>\)](http://creativecommons.org/licenses/?lang=es))

Como puedes ver, esta capa no modifica ni afecta a la dimensión de la profundidad.

Otro efecto que se evita gracias a las capas max-pooling es la pérdida de información del contorno de la imagen, pues en cada capa, el efecto borde (si no se aplica padding), hace que se vaya reduciendo la ventana de variables.

Autoevaluación

La capa de max-pooling se añade para mantener el número de coeficientes limitado y evitar así el overfitting.

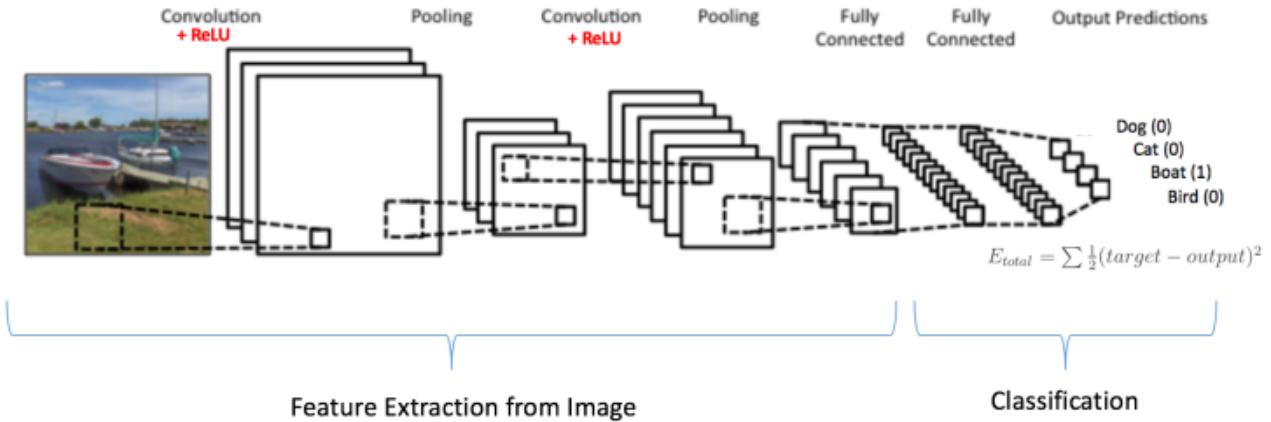
Verdadero Falso

Verdadero

Si no se aplica pooling, en cada convolución se desdobra la cantidad de pesos de la red y esto es perjudicial por cómputo y por aumentar el overfitting

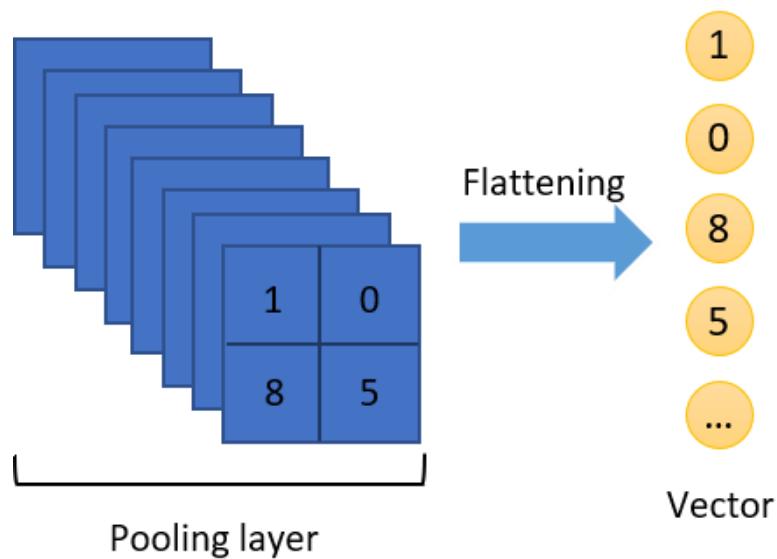
1.3.- El modelo completo.

El modelo de una red neuronal convolucional requiere que, tras las capas convolucionales y de pooling, se añadan capas neuronales densamente conectadas que son las que ponderan la contribución al error que introducen las diferentes variables de salida del bloque convolucional y que hace que la propagación hacia atrás o "back propagation" haga su trabajo.



Desconocido (<https://www.kdnuggets.com/2016/11/intuitive-explanation-convolutional-neural-networks.html/3>) (Dominio público)

Pero como la salida del bloque convolucional es un tensor 3D, es necesario "aplanarlo" con una capa de tipo Flatten.



TowardsAI (<https://towardsai.net/p/machine-learning/beginner-guides-to-convolutional-neural-network-from-scratch-kuzushiji-mnist-75f42c175b21>) (CC BY (<http://creativecommons.org/licenses/?lang=es>))

Vamos a ver un ejemplo en el que utilizamos todo lo visto hasta ahora. Para la preparación de los datos, como novedad, utilizaremos el módulo `ImageDataGenerator`, y tras un primer entrenamiento, vamos a recurrir a una técnica de aumentado del dataset. También utilizaremos una capa de dropout según lo visto en la unidad anterior. Puedes acceder al [notebook original](https://colab.research.google.com/drive/1crTPHtRGkQlVmCG-Pf1xS07iPS2Fgjce?usp=sharing) (<https://colab.research.google.com/drive/1crTPHtRGkQlVmCG-Pf1xS07iPS2Fgjce?usp=sharing>) y hacer una copia para editarla y trabajar sobre él.

Entrenamiento de una Convnet para clasificación de imágenes (Cats and Dogs)

Cargamos y exploramos los datos

Utilizamos un dataset a partir del que se utilizó en una competición en Kaggle en 2013 (el original contenía 25.000 imágenes), con 2000 imágenes de perros y gatos. Lo extraemos al directorio temporal.

In [15]:

```
# Ejecutamos Lo siguiente para descargar el dataset.  
# Podremos encontrar los archivos en el directorio correspondiente  
# en la pestaña de Archivos en el menú de la izquierda.  
  
!wget --no-check-certificate \  
    https://storage.googleapis.com/mledu-datasets/cats_and_dogs_filtered.zip \  
    -O /tmp/cats_and_dogs_filtered.zip  
  
# !unzip /tmp/cats_and_dogs_filtered.zip  
  
--2022-09-01 16:58:48-- https://storage.googleapis.com/mledu-datasets/cats_and_dogs_fil  
tered.zip  
Resolving storage.googleapis.com (storage.googleapis.com)... 209.85.147.128, 142.250.12  
5.128, 142.250.136.128, ...  
Connecting to storage.googleapis.com (storage.googleapis.com)|209.85.147.128|:443... con  
nected.  
HTTP request sent, awaiting response... 200 OK  
Length: 68606236 (65M) [application/zip]  
Saving to: '/tmp/cats_and_dogs_filtered.zip'  
  
/tmp/cats_and_dogs_ 100%[=====>] 65.43M 123MB/s in 0.5s  
  
2022-09-01 16:58:49 (123 MB/s) - '/tmp/cats_and_dogs_filtered.zip' saved [68606236/68606  
236]
```

In [16]:

```
import os  
import zipfile  
  
local_zip = '/tmp/cats_and_dogs_filtered.zip'  
zip_ref = zipfile.ZipFile(local_zip, 'r')  
zip_ref.extractall('/tmp')  
zip_ref.close()
```

ImageDataGenerator

En el caso de las convnets, es necesario un preprocesado especial de los datos, algo diferente al de las redes neuronales convencionales, pues se trabaja con tensores y no con vectores planos. En keras, la forma más cómoda de preparar los datos es a través del módulo ImageDataGenerator. Aquí puedes ver cómo lo aplicamos a nuestras imágenes de perros y gatos.

In [34]:

```
from keras.preprocessing.image import ImageDataGenerator

train_datagen = ImageDataGenerator(rescale=1./255)
valid_datagen = ImageDataGenerator(rescale =1./255)

train_dir = '/tmp/cats_and_dogs_filtered/train'
valid_dir = '/tmp/cats_and_dogs_filtered/validation'

train_gen = train_datagen.flow_from_directory(
    train_dir,
    target_size=(150,150),
    batch_size=20,
    class_mode='binary')

valid_gen = valid_datagen.flow_from_directory(
    valid_dir,
    target_size=(150,150),
    batch_size=20,
    class_mode='binary')
```

Found 2000 images belonging to 2 classes.
Found 1000 images belonging to 2 classes.

In [18]:

```
# Los índices serán los siguientes
# Para las imágenes de gatos (0) y
# para las imágenes de perros (1)
train_gen.class_indices
```

Out[18]:

```
{'cats': 0, 'dogs': 1}
```

In [39]:

```
# Visualizamos las imágenes que se generan
import matplotlib.pyplot as plt

batchX, batchY = next(train_gen)

plt.imshow(batchX[0])
print(batchY[0])
```

0.0



Carmen Bartolomé - Captura de pantalla (Dominio público)

In [20]:

```
batchX.shape
```

Out[20]:

```
(20, 150, 150, 3)
```

Construimos y entrenamos el modelo

In [40]:

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

model = models.Sequential()

model.add(layers.Conv2D(32, (3,3), activation="relu"))
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"))
```

In [41]:

```
opt = optimizers.rmsprop_v2.RMSprop(learning_rate=0.0001)

model.compile(loss='binary_crossentropy',
              optimizer=opt,
              metrics=['acc'])
```

In [42]:

```
hist = model.fit(train_gen, validation_data=valid_gen, epochs=30)

Epoch 1/30
100/100 [=====] - 10s 92ms/step - loss: 0.6901 - acc: 0.5250 -
val_loss: 0.6742 - val_acc: 0.5260
Epoch 2/30
100/100 [=====] - 9s 90ms/step - loss: 0.6545 - acc: 0.6210 - v
al_loss: 0.6299 - val_acc: 0.6590
Epoch 3/30
100/100 [=====] - 10s 98ms/step - loss: 0.6040 - acc: 0.6710 -
val_loss: 0.6333 - val_acc: 0.6470
Epoch 4/30
100/100 [=====] - 9s 94ms/step - loss: 0.5548 - acc: 0.7215 - v
al_loss: 0.5984 - val_acc: 0.6740
Epoch 5/30
100/100 [=====] - 10s 95ms/step - loss: 0.5300 - acc: 0.7360 -
val_loss: 0.6041 - val_acc: 0.6740
.
.
.
Epoch 26/30
100/100 [=====] - 9s 90ms/step - loss: 0.0621 - acc: 0.9825 - v
al_loss: 0.8028 - val_acc: 0.7360
Epoch 27/30
100/100 [=====] - 9s 90ms/step - loss: 0.0484 - acc: 0.9865 - v
al_loss: 0.8444 - val_acc: 0.7550
Epoch 28/30
100/100 [=====] - 9s 89ms/step - loss: 0.0409 - acc: 0.9900 - v
al_loss: 1.0128 - val_acc: 0.7250
Epoch 29/30
100/100 [=====] - 9s 89ms/step - loss: 0.0358 - acc: 0.9910 - v
al_loss: 0.9081 - val_acc: 0.7390
Epoch 30/30
100/100 [=====] - 9s 90ms/step - loss: 0.0300 - acc: 0.9950 - v
al_loss: 0.9246 - val_acc: 0.7390
```

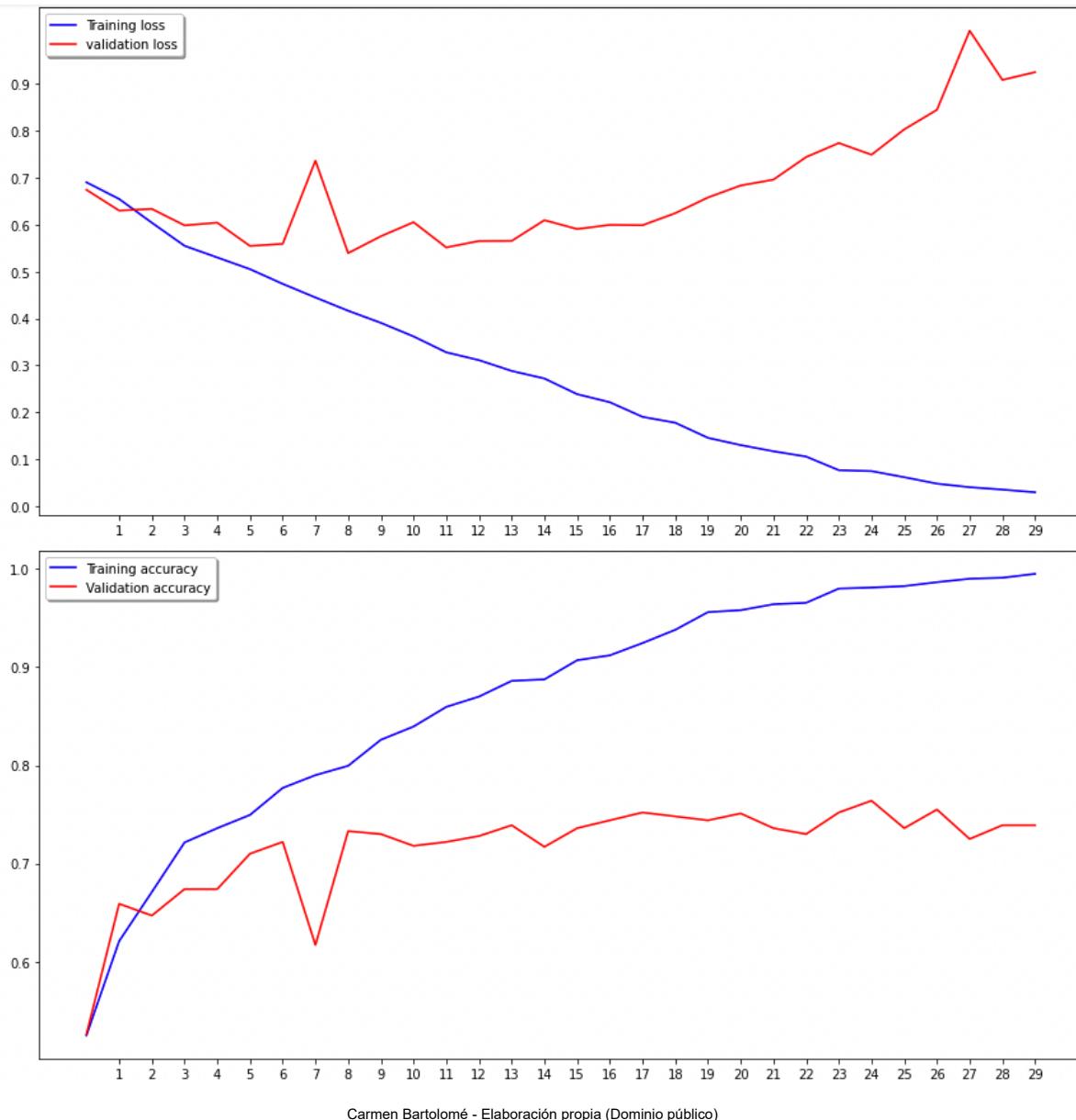
Evaluamos el modelo

In [43]:

```
import numpy as np

fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(12, 12))
ax1.plot(hist.history['loss'], color='b', label="Training loss")
ax1.plot(hist.history['val_loss'], color='r', label="validation loss")
ax1.set_xticks(np.arange(1, 30, 1))
ax1.set_yticks(np.arange(0, 1, 0.1))
ax1.legend(loc='best', shadow=True)

ax2.plot(hist.history['acc'], color='b', label="Training accuracy")
ax2.plot(hist.history['val_acc'], color='r', label="Validation accuracy")
ax2.set_xticks(np.arange(1, 30, 1))
ax2.legend(loc='best', shadow=True)
plt.tight_layout()
plt.show()
```



Técnica de Dataset aumentado

Ya hemos visto que el overfitting puede ser evitado si contamos con un conjunto de datos lo mayor posible. Si modificamos las imágenes de forma que sigan siendo representativas, pero que para el modelo sean nuevas, le ayudaremos a generalizar mejor.

Trucos para aumentar el dataset

- 1.- Rotación entre 0 y 180 grados.
- 2.- Cambios de altura o anchura
- 3.- Recortes
- 4.- Aumentos
- 5.- Simetrías

Tras las transformaciones, también es útil la funcionalidad de añadir pixeles de relleno en las zonas que quedan vacías.

Nuevo modelo con dropout

Definimos ahora una nueva red neuronal convolucional, añadiendo dropout tras el bloque convolucional

In [25]:

```
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'))

opt = optimizers.rmsprop_v2.RMSprop(learning_rate=0.0001)

model.compile(loss='binary_crossentropy',
              optimizer=opt,
              metrics=['acc'])
```

In [26]:

```
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,)  
  
valid_datagen = ImageDataGenerator(rescale =1./255)  
  
train_dir = '/tmp/cats_and_dogs_filtered/train'  
valid_dir = '/tmp/cats_and_dogs_filtered/validation'  
  
train_gen = train_datagen.flow_from_directory(  
    train_dir,  
    target_size= (150,150),  
    batch_size=32,  
    class_mode='binary')  
  
valid_gen = valid_datagen.flow_from_directory(  
    valid_dir,  
    target_size=(150,150),  
    batch_size=32,  
    class_mode='binary')
```

Found 2000 images belonging to 2 classes.
Found 1000 images belonging to 2 classes.

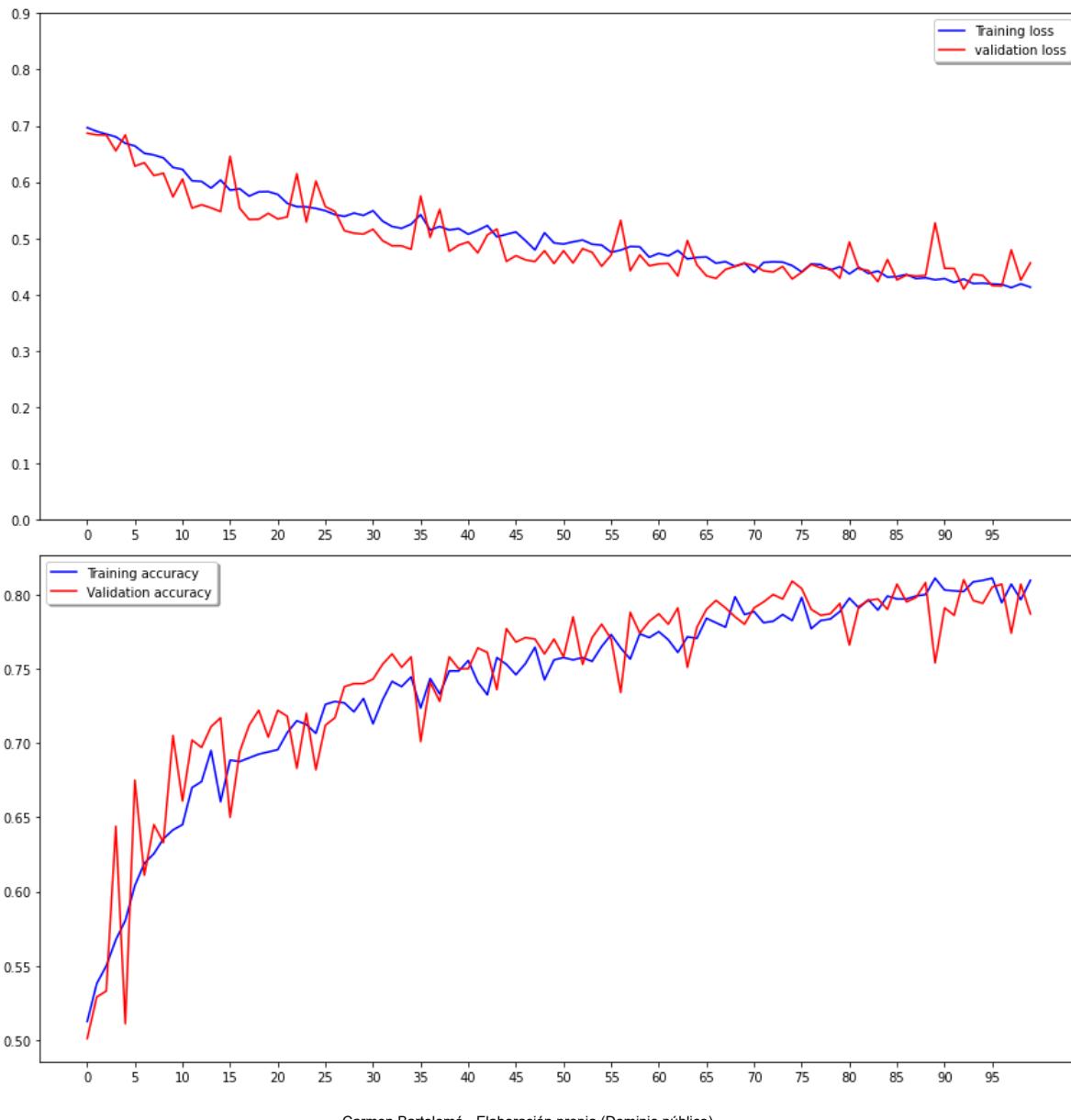
In [27]:

```
hist = model.fit(train_gen, validation_data=valid_gen, epochs=100)  
  
Epoch 1/100  
63/63 [=====] - 18s 273ms/step - loss: 0.6968 - acc: 0.5125 - val_loss: 0.6869 - val_acc: 0.5010  
Epoch 2/100  
63/63 [=====] - 17s 270ms/step - loss: 0.6901 - acc: 0.5380 - val_loss: 0.6839 - val_acc: 0.5290  
Epoch 3/100  
63/63 [=====] - 17s 267ms/step - loss: 0.6852 - acc: 0.5500 - val_loss: 0.6835 - val_acc: 0.5330  
Epoch 4/100  
63/63 [=====] - 17s 270ms/step - loss: 0.6805 - acc: 0.5675 - val_loss: 0.6555 - val_acc: 0.6440  
Epoch 5/100  
63/63 [=====] - 17s 270ms/step - loss: 0.6689 - acc: 0.5805 - val_loss: 0.6838 - val_acc: 0.5110  
. . .  
Epoch 96/100  
63/63 [=====] - 17s 269ms/step - loss: 0.4189 - acc: 0.8110 - val_loss: 0.4159 - val_acc: 0.8050  
Epoch 97/100  
63/63 [=====] - 17s 268ms/step - loss: 0.4181 - acc: 0.7945 - val_loss: 0.4154 - val_acc: 0.8070  
Epoch 98/100  
63/63 [=====] - 17s 267ms/step - loss: 0.4124 - acc: 0.8070 - val_loss: 0.4795 - val_acc: 0.7740  
Epoch 99/100  
63/63 [=====] - 17s 268ms/step - loss: 0.4191 - acc: 0.7965 - val_loss: 0.4257 - val_acc: 0.8070  
Epoch 100/100  
63/63 [=====] - 17s 269ms/step - loss: 0.4133 - acc: 0.8095 - val_loss: 0.4562 - val_acc: 0.7870
```

In [33]:

```
fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(12, 12))
ax1.plot(hist.history['loss'], color='b', label="Training loss")
ax1.plot(hist.history['val_loss'], color='r', label="validation loss")
ax1.set_xticks(np.arange(0, 100, 5))
ax1.set_yticks(np.arange(0, 1, 0.1))
ax1.legend(loc='best', shadow=True)

ax2.plot(hist.history['acc'], color='b', label="Training accuracy")
ax2.plot(hist.history['val_acc'], color='r', label="Validation accuracy")
ax2.set_xticks(np.arange(0, 100, 5))
ax2.legend(loc='best', shadow=True)
plt.tight_layout()
plt.show()
```

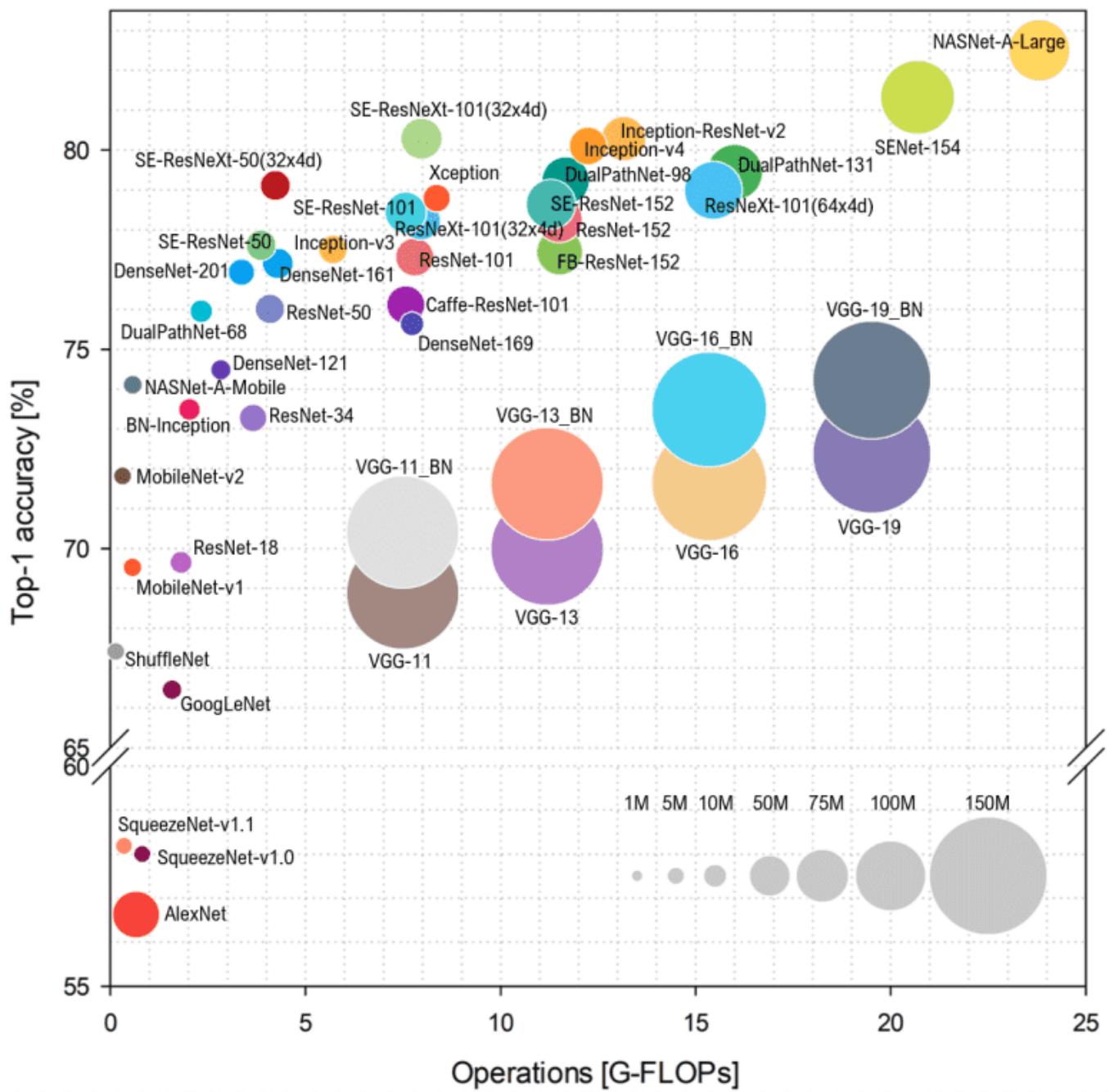


Carmen Bartolomé - Elaboración propia (Dominio público)

Según vemos en la gráfica, el modelo ha ganado en precisión y además ya no se aprecia overfitting.

1.4.- Modelos pre-entrenados y Fine-tunning.

Una de las técnicas que más se utilizan en la industria es utilizar redes pre-entrenadas con grandes datasets. Sus creadores han logrado un grado de generalización muy bueno y estas redes suelen tener un comportamiento muy bueno como parte de otros modelos orientados a tareas del mismo tipo.



Simone Bianco y otros autores (<https://arxiv.org/abs/1810.00736>) (CC BY-SA (<http://creativecommons.org/licenses/?lang=es>))

La principal ventaja de estas redes es que tienen bien jerarquizadas sus capas (o conjuntos de capas) lo que hace que sirvan como modelo de base para todo lo que tenga que ver con el mundo visual o lingüístico. Es como tener ojos y haber aprendido a ver formas, colores y objetos muy comunes. Una persona que no haya visto nunca animales marinos, tendrá que aprender a distinguir entre los distintos tipos de peces, pero sabrá ver el agua, y que son animales, distinguiéndolos de algas o rocas. Estas redes pre-entrenadas, se suelen "completar" para tener el modelo que necesitamos para un caso de aplicación concreto.

Son bastante conocidas las arquitecturas VGG, ResNet, Xception, etc, en el ámbito de la visión artificial. En concreto, vamos a utilizar las redes VGG16 y VGG19, incluidas en Keras, y entrenadas con el dataset ImageNet en 2014.

```
tf.keras.applications.VGG16(  
    include_top=True,  
    weights="imagenet",  
    input_tensor=None,  
    input_shape=None,  
    pooling=None,  
    classes=1000,  
    classifier_activation="softmax",  
)
```

[VGG Keras](https://keras.io/api/applications/vgg16/) (<https://keras.io/api/applications/vgg16/>). (Dominio público)

Las redes pre-entrenadas se puede utilizar de dos maneras, y vamos a aplicar ambas al dataset con el que hemos estado trabajando de imágenes de perros y gatos.

- ✓ "Feature Extraction" es una técnica con la que utilizamos el modelos pre-entrenado para extraer los patrones principales que dicho modelo detecta en las imágenes que le estamos pasando. Después, pasaremos ese mapa de "features" o características, a un modelo sencillo de red neuronal, para su entrenamiento.
- ✓ "Fine-tuning" es la técnica que consiste en componer un modelo híbrido en el que, al final de la red pre-entrenada, ponemos dos capas de red neuronal, y congelamos casi todas la capas de la red pre-entrenada para aprovechar sus primeras jerarquías de detección, y ya solo entrenar el último bloque y las capas que hemos añadido.

Veamos ambas técnicas en acción sobre nuestro dataset de gatos y perros reducido. Sería recomendable aplicar también técnicas de aumentado de dataset, así como mejorar la aplicación del "fine-tuning", y te animamos a que sigas explorando y trabajando sobre ello una vez hayas dominado la técnica.

Puedes acceder al [notebook original](#) (https://colab.research.google.com/drive/1wO_271A7FT2YngsDygDzi1z9LnWJa8n-?usp=sharing) y hacer una copia para editarlo y trabajar sobre él.

Entrenamiento de una Convnet para clasificación de imágenes (Cats and Dogs)

Cargamos los datos

Utilizamos un dataset a partir del que se utilizó en una competición en Kaggle en 2013 (el original contenía 25.000 imágenes), con 2000 imágenes de perros y gatos. Lo extraemos al directorio temporal.

In [1]:

```
# Ejecutamos Lo siguiente para descargar el dataset.  
# Podremos encontrar los archivos en el directorio correspondiente  
# en la pestaña de Archivos en el menú de la izquierda.  
  
!wget --no-check-certificate \  
    https://storage.googleapis.com/mledu-datasets/cats_and_dogs_filtered.zip \  
    -O /tmp/cats_and_dogs_filtered.zip  
  
# !unzip /tmp/cats_and_dogs_filtered.zip  
  
--2022-09-02 07:00:21-- https://storage.googleapis.com/mledu-datasets/cats_and_dogs_filtered.zip  
Resolving storage.googleapis.com (storage.googleapis.com)... 172.217.194.128, 142.251.1.0.128, 142.251.12.128, ...  
Connecting to storage.googleapis.com (storage.googleapis.com)|172.217.194.128|:443... connected.  
HTTP request sent, awaiting response... 200 OK  
Length: 68606236 (65M) [application/zip]  
Saving to: '/tmp/cats_and_dogs_filtered.zip'  
  
/tmp/cats_and_dogs_ 100%[=====] 65.43M 246MB/s in 0.3s  
  
2022-09-02 07:00:22 (246 MB/s) - '/tmp/cats_and_dogs_filtered.zip' saved [68606236/68606236]
```

In [2]:

```
import os  
import zipfile  
  
local_zip = '/tmp/cats_and_dogs_filtered.zip'  
zip_ref = zipfile.ZipFile(local_zip, 'r')  
zip_ref.extractall('/tmp')  
zip_ref.close()
```

Creamos la red base con el bloque VGG16, pasando como argumentos los pesos que se quedaron configurados con el entrenamiento de ImageNet, que no incluya la red neuronal final (porque la vamos a poner nosotros) y las dimensiones del tensor de entrada.

In []:

```
from keras.applications.vgg16 import VGG16  
  
conv_base = VGG16(weights='imagenet',  
                  include_top=False,  
                  input_shape=(150, 150, 3))  
  
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/vgg16/vgg16_weights_tf_dim_ordering_tf_kernels_notop.h5  
58892288/58889256 [=====] - 1s 0us/step  
58900480/58889256 [=====] - 1s 0us/step
```

In []:

```
conv_base.summary()
```

Model: "vgg16"

| Layer (type) | Output Shape | Param # |
|----------------------------|-----------------------|---------|
| ===== | | |
| input_1 (InputLayer) | [(None, 150, 150, 3)] | 0 |
| block1_conv1 (Conv2D) | (None, 150, 150, 64) | 1792 |
| block1_conv2 (Conv2D) | (None, 150, 150, 64) | 36928 |
| block1_pool (MaxPooling2D) | (None, 75, 75, 64) | 0 |
| block2_conv1 (Conv2D) | (None, 75, 75, 128) | 73856 |
| block2_conv2 (Conv2D) | (None, 75, 75, 128) | 147584 |
| block2_pool (MaxPooling2D) | (None, 37, 37, 128) | 0 |
| block3_conv1 (Conv2D) | (None, 37, 37, 256) | 295168 |
| block3_conv2 (Conv2D) | (None, 37, 37, 256) | 590080 |
| block3_conv3 (Conv2D) | (None, 37, 37, 256) | 590080 |
| block3_pool (MaxPooling2D) | (None, 18, 18, 256) | 0 |
| block4_conv1 (Conv2D) | (None, 18, 18, 512) | 1180160 |
| block4_conv2 (Conv2D) | (None, 18, 18, 512) | 2359808 |
| block4_conv3 (Conv2D) | (None, 18, 18, 512) | 2359808 |
| block4_pool (MaxPooling2D) | (None, 9, 9, 512) | 0 |
| block5_conv1 (Conv2D) | (None, 9, 9, 512) | 2359808 |
| block5_conv2 (Conv2D) | (None, 9, 9, 512) | 2359808 |
| block5_conv3 (Conv2D) | (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 | |

Vemos que el tensor de salida tiene las dimensiones (4,4,512) y esto debemos tenerlo en cuenta para la red neuronal que pondremos después.

"Feature extraction"

La forma más rápida y menos pesada de utilizar el modelo pre.entrenado, es hacer una extracción de las principales variables fundamentales de las imágenes del dataset, y luego pasarlas a un modelo neuronal normal.

In []:

```
import numpy as np
from keras.preprocessing.image import ImageDataGenerator

train_datagen = ImageDataGenerator(rescale=1./255)
valid_datagen = ImageDataGenerator(rescale =1./255)

train_dir = '/tmp/cats_and_dogs_filtered/train'
valid_dir = '/tmp/cats_and_dogs_filtered/validation'

datagen = ImageDataGenerator(rescale=1./255)
batch_size = 20

def extract_features(directory, sample_count):
    features = np.zeros(shape=(sample_count, 4, 4, 512))
    labels = np.zeros(shape=(sample_count))
    generator = datagen.flow_from_directory(
        directory,
        target_size=(150, 150),
        batch_size=batch_size,
        class_mode='binary')
    i = 0
    for inputs_batch, labels_batch in generator:
        features[i * batch_size : (i + 1) * batch_size] = inputs_batch
        labels[i * batch_size : (i + 1) * batch_size] = labels_batch
        i += 1
        if i * batch_size >= sample_count:
            break
    return features, labels

train_features, train_labels = extract_features(train_dir, 2000)
validation_features, validation_labels = extract_features(valid_dir, 1000)
# test_features, test_labels = extract_features(test_dir, 1000)
```

Found 2000 images belonging to 2 classes.

Found 1000 images belonging to 2 classes.

Tenemos un tensor con los valores de salida tras la extracción de "features". Usamos reshape para aplanarlos y poder introducirlos en una red neuronal.

In []:

```
train_features = np.reshape(train_features, (2000, 4 * 4 * 512))
validation_features = np.reshape(validation_features, (1000, 4 * 4 * 512))
```

Creamos ahora un modelo neuronal sencillo con dropout en el que utilizaremos el mapa de "features" que hemos calculado.

In []:

```
from keras import models
from keras import layers
from keras.optimizers import rmsprop_v2

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=rmsprop_v2.RMSprop(learning_rate=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))

Epoch 1/30
100/100 [=====] - 2s 18ms/step - loss: 0.5949 - acc: 0.6780 - val_loss: 0.4408 - val_acc: 0.8240
Epoch 2/30
100/100 [=====] - 2s 16ms/step - loss: 0.4229 - acc: 0.8185 - val_loss: 0.3635 - val_acc: 0.8500
Epoch 3/30
100/100 [=====] - 2s 19ms/step - loss: 0.3525 - acc: 0.8470 - val_loss: 0.3323 - val_acc: 0.8690
.
.
.
100/100 [=====] - 2s 15ms/step - loss: 0.0946 - acc: 0.9720 - val_loss: 0.2660 - val_acc: 0.8860
Epoch 29/30
100/100 [=====] - 2s 15ms/step - loss: 0.0890 - acc: 0.9770 - val_loss: 0.2614 - val_acc: 0.8880
Epoch 30/30
100/100 [=====] - 2s 15ms/step - loss: 0.0874 - acc: 0.9705 - val_loss: 0.2612 - val_acc: 0.8890
```

Obtenemos una precisión de validación del 88%. Vamos a representar las curvas de precisión y pérdida para tener una imagen más completa de lo que está pasando.

In []:

```
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']

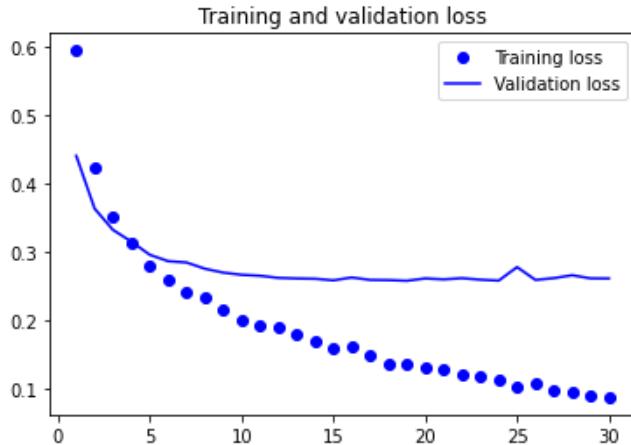
epochs = range(1, len(acc) + 1)

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()

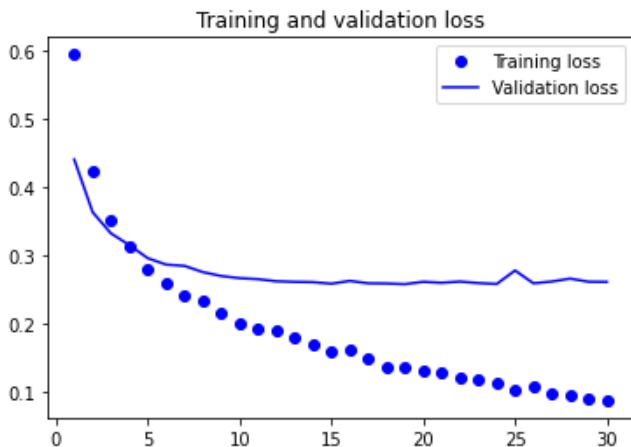
plt.figure()

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()
```



Carmen Bartolomé - Elaboración propia (Dominio público)



Carmen Bartolomé - Elaboración propia (Dominio público)

Fine-tuning

Utilizamos ahora otro modelo pre-entrenado, VGG19, congelando casi todas sus capas internas hasta el bloque 5. Haremos un entrenamiento sencillo de las tres capas del bloque 5 de VGG19 y de dos capas neuronales que vamos a añadir.

In [3]:

```
import numpy as np
from keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras import Sequential
from tensorflow.keras.layers import *
from tensorflow.keras.optimizers import Adam
```

In [4]:

```
from tensorflow.keras.applications.vgg19 import VGG19, preprocess_input

train_gen = ImageDataGenerator(preprocessing_function=preprocess_input).flow_from_directory('/tmp/cats_and_dogs_filtered/train')
valid_gen = ImageDataGenerator(preprocessing_function=preprocess_input).flow_from_directory('/tmp/cats_and_dogs_filtered/validation')
```

Found 2000 images belonging to 2 classes.

Found 1000 images belonging to 2 classes.

In [12]:

```
# Cargamos un modelo pre-entrenado con Imagenet
vgg_model = VGG19(include_top=False, weights="imagenet", input_shape=(256, 256, 3))

# Congelamos al modelo / que los parámetros no se actualicen
for layer in vgg_model.layers[:12]:
    layer.trainable = False
```

In []:

```
model = Sequential()

model.add(vgg_model)

model.add(Flatten())
model.add(Dense(units=100, activation="relu"))
model.add(Dense(units=2, activation="softmax"))
```

In [13]:

```
# Configuración
model.compile(optimizer=Adam(learning_rate=1e-5),
              loss="categorical_crossentropy",
              metrics=["acc"])

# Entrenamos al modelo
history = model.fit(train_gen, validation_data=valid_gen, epochs=10)

Epoch 1/10
63/63 [=====] - 31s 487ms/step - loss: 5.3426e-07 - acc: 1.0000
- val_loss: 0.1775 - val_acc: 0.9680
Epoch 2/10
63/63 [=====] - 32s 505ms/step - loss: 0.0027 - acc: 0.9985 - val_loss: 0.2276 - val_acc: 0.9660
Epoch 3/10
63/63 [=====] - 31s 490ms/step - loss: 0.0082 - acc: 0.9985 - val_loss: 0.3145 - val_acc: 0.9590
Epoch 4/10
63/63 [=====] - 31s 490ms/step - loss: 0.0172 - acc: 0.9970 - val_loss: 0.2801 - val_acc: 0.9620
Epoch 5/10
63/63 [=====] - 31s 495ms/step - loss: 0.0021 - acc: 0.9995 - val_loss: 0.1627 - val_acc: 0.9710
Epoch 6/10
63/63 [=====] - 31s 492ms/step - loss: 5.6681e-05 - acc: 1.0000
- val_loss: 0.1864 - val_acc: 0.9720
Epoch 7/10
63/63 [=====] - 31s 492ms/step - loss: 9.1397e-07 - acc: 1.0000
- val_loss: 0.1827 - val_acc: 0.9720
Epoch 8/10
63/63 [=====] - 31s 492ms/step - loss: 5.6479e-07 - acc: 1.0000
- val_loss: 0.1826 - val_acc: 0.9720
Epoch 9/10
63/63 [=====] - 31s 494ms/step - loss: 4.8529e-07 - acc: 1.0000
- val_loss: 0.1828 - val_acc: 0.9720
Epoch 10/10
63/63 [=====] - 31s 494ms/step - loss: 4.2843e-07 - acc: 1.0000
- val_loss: 0.1828 - val_acc: 0.9720
```

Logramos una precisión final del 97%, mucho mejor que la que se obtuvo inicialmente con este dataset y un modelo convolucional sencillo, que rondaba el 74%.

2.- Redes neuronales recurrentes.

Caso práctico

Los resultados de las redes neuronales convolucionales para reconocimiento de imagen, han sido realmente mejores que los modelos anteriores que habían estado probando en Pick&Deliver.

Toda la empresa ya sabe que el equipo de inteligencia artificial va a tope y desde algunos departamentos les van haciendo sugerencias de procesos que les afectan y que también se podrían mejorar.

Desde el área de administración, les han pedido una herramienta que les ayude a procesar toda la documentación que les llega. Una especie de gestor inteligente que encuentre información de valor, datos que son útiles para otros procesos, etc (Etcétera), dentro del texto de distintos tipos de documentos. Un OCR (Optical Character Recognition) que "entienda" lo que está leyendo y solo devuelva lo que es importante para la empresa.

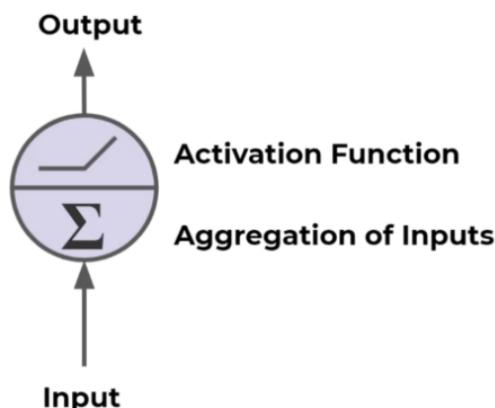


[katemangostar \(\[https://www.freepik.es/foto gratis/negocios-que-controla-documentos-mesa_1027035.htm#query=documentos&position=13&from_view=search\]\(https://www.freepik.es/foto gratis/negocios-que-controla-documentos-mesa_1027035.htm#query=documentos&position=13&from_view=search\)\) \(CC BY-NC-SA \(<http://creativecommons.org/licenses/?lang=es>\)\)](https://www.freepik.es/foto gratis/negocios-que-controla-documentos-mesa_1027035.htm#query=documentos&position=13&from_view=search)

Las redes neuronales recurrentes (RNN) fueron modelizadas en la década de los 80, pero estas redes han sido muy difíciles de entrenar por sus requerimientos en computación. Con la llegada de los avances de estos últimos años, en materia de hardware, se han vuelto más accesibles y se ha popularizado su uso por la industria.

En el perceptrón simple que hemos estudiado, el flujo de cálculo es en una única dirección, sin registrar ni recordar valores previos de los datos de entrada. Solo los pesos van quedando registrados.

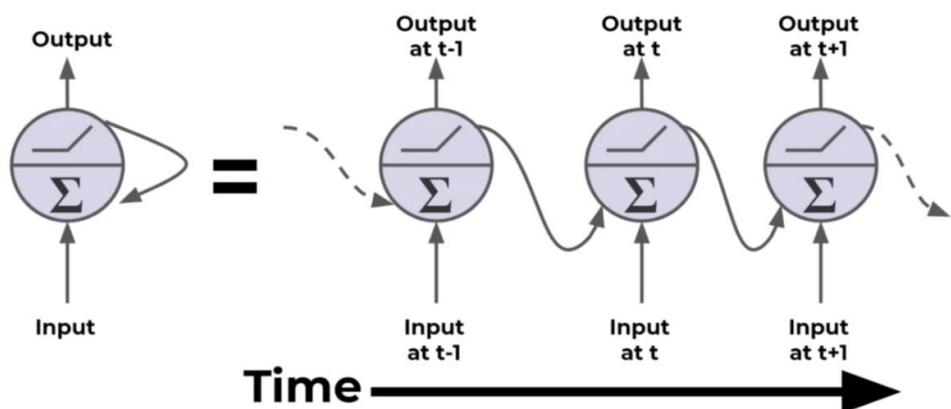
En la neurona recurrente, se combina el cálculo de las entradas (que provienen de la capa previa), con la salida que la propia neurona tuvo en su ejecución del paso anterior.



[Rishi Kumar \(<https://medium.com/nerd-for-tech/recurrent-neural-network-an-overview-1128ffc34ce7>\) \(CC BY-SA \(<http://creativecommons.org/licenses/?lang=es>\)\)](https://medium.com/nerd-for-tech/recurrent-neural-network-an-overview-1128ffc34ce7)

Una forma más sencilla de entenderlo, es representando los estados en una línea en función del tiempo:

- Recurrent Neuron



Dado que la salida de una neurona recurrente en un instante de tiempo determinado es una función de entradas de los instantes de tiempo anteriores, se podría decir que una neurona recurrente tiene en cierta forma memoria. La parte de una red neuronal que preserva un estado a través del tiempo se suele llamar memory cell(o simplemente cell)

Esta forma de "retener" en cierta forma la información de forma acumulativa es muy útil a la hora de trabajar con casos formados por secuencias de palabras que están relacionadas entre sí, como a la hora de :

- ✓ Traducción de textos sin perder el sentido de toda la frase.
- ✓ Reconocimiento de voz.
- ✓ Chatbots.
- ✓ Generar textos con un tema concreto.
- ✓ De forma combinada con Convnets para generar descripciones de imágenes.

En Keras, la clase SimpleRNN nos permite instanciar una capa de tipo recurrente:

```
from keras.models import Sequential
from keras.layers import Embedding, SimpleRNN
model = Sequential()
model.add(Embedding(10000, 32))
model.add(SimpleRNN(32))
```

En caso de ampliar varias capas de recurrencia, es importante añadir el parámetro que indica que se devuelva toda la secuencia de salidas, porque si no, por defecto, solo devolverían la salida del último paso. La última capa solo tiene que devolver la salida del último paso:

```
model = Sequential()
model.add(Embedding(10000, 32))
model.add(SimpleRNN(32, return_sequences=True))
model.add(SimpleRNN(32, return_sequences=True))
model.add(SimpleRNN(32, return_sequences=True))
model.add(SimpleRNN(32))
```

Este tipo de modelos se enfrentan a comportamientos anómalos del gradiente en las capas más profundas, que pueden hacer que el resultado sea igual o peor que si no utilizaremos redes recurrentes. Para evitar estos problemas, surgen redes como la "Long Short Term Memory Networks" o LSTM, que veremos en la siguiente sección.

Autoevaluación

En las redes neuronales recurrentes, las neuronas solo trabajan con las entradas de la capa previa.

Verdadero Falso

Falso

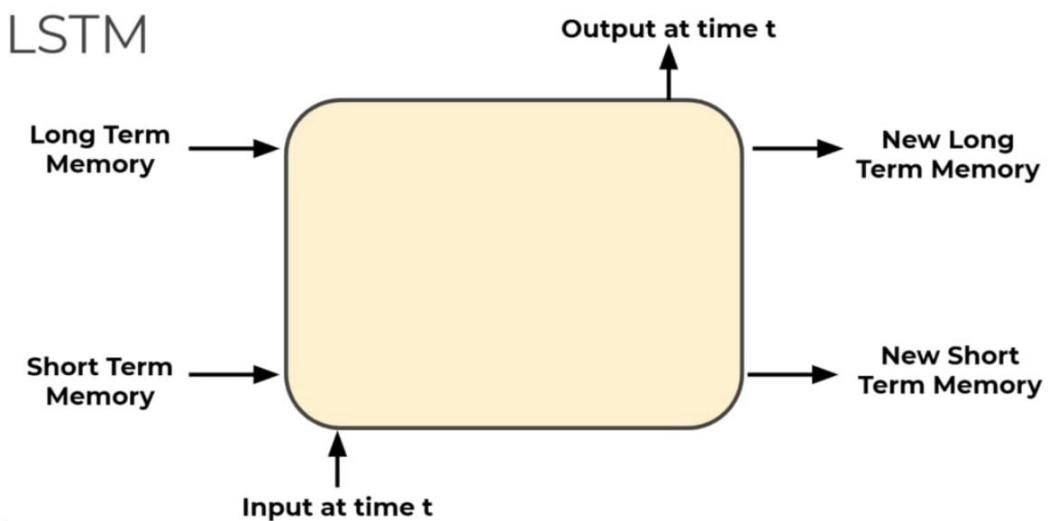
Las RNN también reciben y computan como entrada su propia salida de la etapa temporal anterior.

2.1.- La capa LSTM.

Los modelos Long-Short Term Memory (LSTM) son una extensión de las redes neuronales recurrentes, que básicamente amplían su memoria para tener en cuenta los estados cronológicamente más antiguos. Las LSTM (Long Short Term Memory) permiten a las RNN (Redes Neuronales Recurrentes) recordar sus entradas durante un largo período de tiempo. Esto se debe a que LSTM contiene su

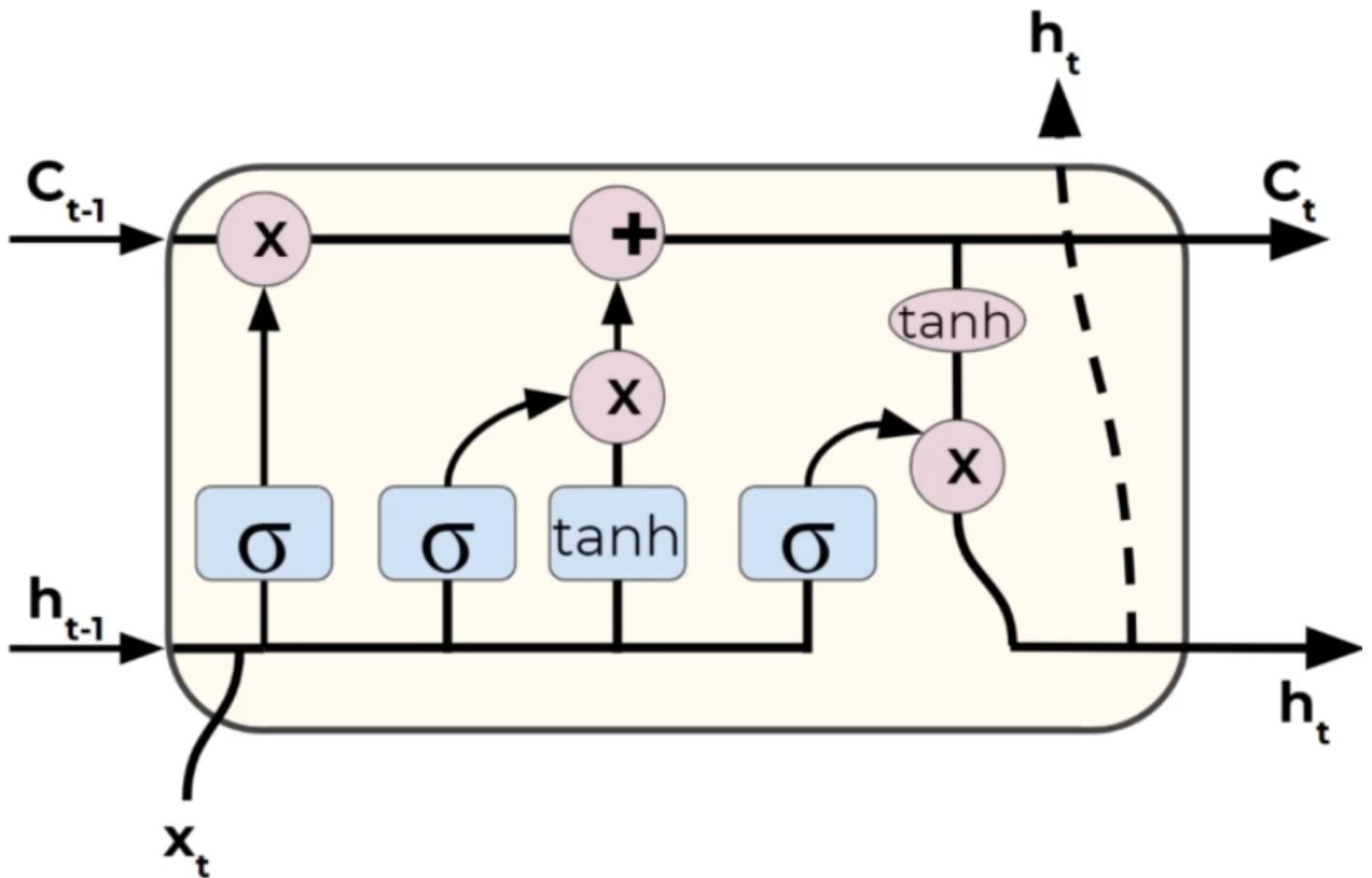
información en la memoria, que puede considerarse similar a la memoria de un ordenador , en el sentido que una neurona de una LSTM puede leer, escribir y borrar información de su memoria.

Esta memoria se puede ver como una “celda” bloqueada, donde “bloqueada” significa que la célula decide si almacenar o eliminar información dentro (abriendo la puerta o no para almacenar), en función de la importancia que asigna a la información que está recibiendo. La asignación de importancia se decide a través de los pesos, que también se aprenden mediante el algoritmo. Esto lo podemos ver como que aprende con el tiempo qué información es importante y cuál no.



Rishi Kumar (<https://medium.com/nerd-for-tech/recurrent-neural-network-an-overview-1128ffc34ce7>) (CC BY-SA (<http://creativecommons.org/licenses/?lang=es>))

En una neurona LSTM hay tres puertas a estas “celdas” de información: puerta de entrada (input gate), puerta de olvidar (forget gate) y puerta de salida (output gate). Estas puertas determinan si se permite o no una nueva entrada, se elimina la información porque no es importante o se deja que afecte a la salida en el paso de tiempo actual.



Rishi Kumar (<https://medium.com/nerd-for-tech/recurrent-neural-network-an-overview-1128ffc34ce7>) (CC BY-SA (<http://creativecommons.org/licenses/?lang=es>))

En Keras, introducimos una capa LSTM con la clase del mismo nombre, como en el siguiente ejemplo de modelo:

```
from keras.layers import LSTM
model = Sequential()
model.add(Embedding(max_features, 32))
model.add(LSTM(32))
model.add(Dense(1, activation='sigmoid'))
```

Para saber más

Un caso muy interesante para ver en acción la arquitectura LSTM es el del análisis de sentimiento en comentarios de películas de la web IMBD.

Puedes acceder a [este notebook](https://colab.research.google.com/github/markwest1972/LSTM-Example-Google-Colaboratory/blob/master/LSTM_IMDB_Sentiment_Example.ipynb) (https://colab.research.google.com/github/markwest1972/LSTM-Example-Google-Colaboratory/blob/master/LSTM_IMDB_Sentiment_Example.ipynb) y copiarlo para editarla y trabajar sobre él. Si quieras entender mejor cada parte, puedes consultar la documentación de su autor en [este repositorio](https://github.com/markwest1972/LSTM-Example-Google-Colaboratory) (<https://github.com/markwest1972/LSTM-Example-Google-Colaboratory>) de GitHub.

3.- Redes generativas adversarias.

Caso práctico



LookStudio (<https://www.freepik.es/autor/lookstudio>) (CC BY-SA (<http://creativecommons.org/licenses/?lang=es>))

Lorena está entusiasmada con las redes neuronales convolucionales. Ha estado rescatando todos sus modelos anteriores de clasificación de imágenes y ha probado cómo las redes convolucionales dan resultados mucho mejores en la precisión si se tiene cuidado con el overfitting.

Cuando le cuenta a Miguel cómo ha logrado mejorar los resultados, él se contagia de su entusiasmo y le habla de las redes GAN (Generative Adversarial Network). "¿Las redes GAN? ¿En qué consisten?" le pregunta Lorena.

"¿Llegaste a probar aquella aplicación móvil que se puso tan de moda que, a partir de una foto tuya, te mostraba tu rostro con mucha más edad?" le responde Miguel "Estaba creada con una red GAN"

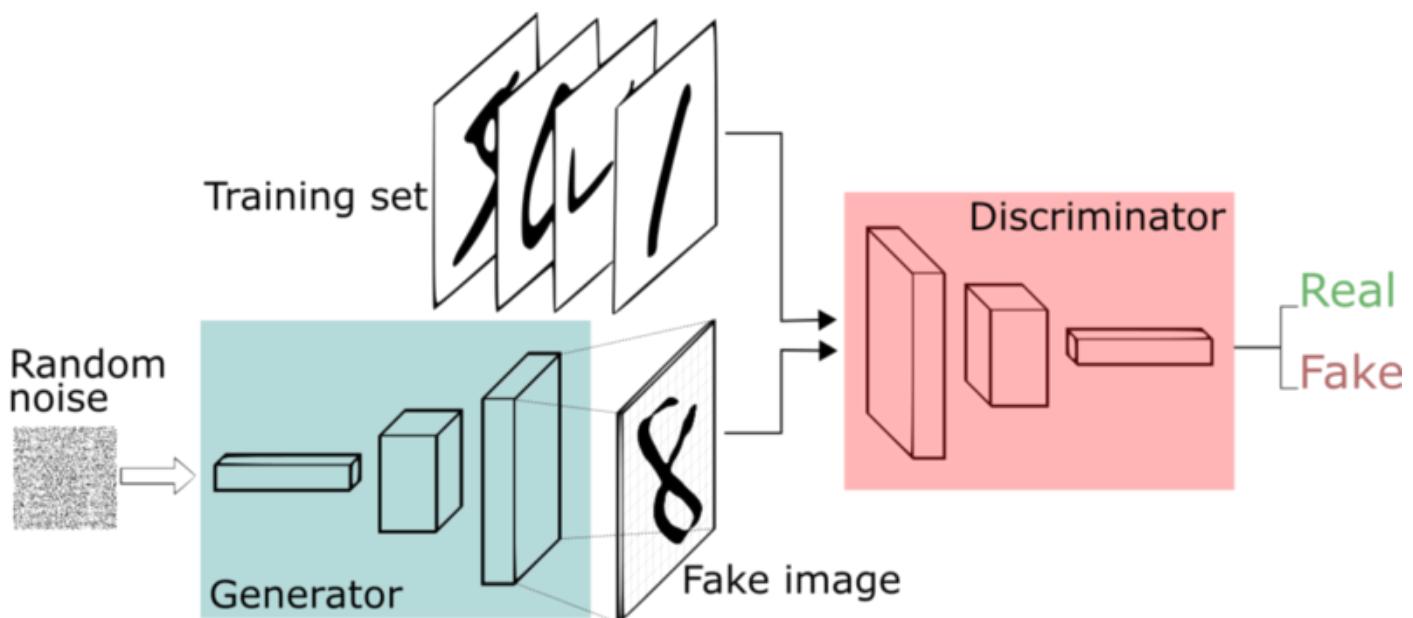
Lorena no ve, a priori, si eso les puede ayudar en Pick&Deliver, pero siente una gran curiosidad y decide investigar y probar a programar una GAN. Tal vez después de conocer mejor este tipo de arquitecturas vea su posible aplicación en la empresa.



[infobae \(https://www.infobae.com/america/tecnologia/2018/09/19/quien-es-el-cientifico-que-lidera-la-revolucion-creativa-de-las-maquinas/\)](https://www.infobae.com/america/tecnologia/2018/09/19/quien-es-el-cientifico-que-lidera-la-revolucion-creativa-de-las-maquinas/) (Dominio público)

Las redes generativas adversarias o redes GAN son una arquitectura que implican, en realidad, la interacción entre dos modelos de redes neuronales profundas. Fueron diseñadas por Ian Goodfellow y otros autores, y descritas en un [paper](https://arxiv.org/abs/1406.2661) (<https://arxiv.org/abs/1406.2661>) publicado en 2014, como un juego de suma cero entre un generador y un discriminador. En su publicación, Goodfellow propone una arquitectura de dos modelos de deep learning que incluyen redes convolucionales en las que sustituyen las capas de pooling por strides, y hacen hincapié en aplicar, también la técnica Batch Normalization que vimos en la unidad anterior.

La función del Generador es generar instancias de datos a partir de "nada" (consideraremos el ruido aleatorio como entrada), mientras que el Discriminador comprueba si los datos aleatorios son consistentes con las instancias de datos reales, y devuelve probabilidades entre 0 y 1, donde 1 indica una predicción de realidad (real) y 0 indica una falsificación (fake). Esquemáticamente se podría resumir visualmente de la siguiente manera:



[Thalles Silva \(https://www.freecodecamp.org/news/an-intuitive-introduction-to-generative-adversarial-networks-gans-7a2264a81394\)](https://www.freecodecamp.org/news/an-intuitive-introduction-to-generative-adversarial-networks-gans-7a2264a81394/) (CC BY-SA (<http://creativecommons.org/licenses/?lang=es>))

El sistema está listo cuando el generador consigue generar casos que el discriminador da por válidos.

En Keras, un ejemplo de definición del modelo Generador podría ser:

```

import keras
from keras import layers
import numpy as np
latent_dim = 32
height = 32
width = 32
channels = 3
generator_input = keras.Input(shape=(latent_dim,))
x = layers.Dense(128 * 16 * 16)(generator_input)
x = layers.LeakyReLU()(x)
x = layers.Reshape((16, 16, 128))(x)
x = layers.Conv2D(256, 5, padding='same')(x)
x = layers.LeakyReLU()(x)
x = layers.Conv2DTranspose(256, 4, strides=2, padding='same')(x)
x = layers.LeakyReLU()(x)
x = layers.Conv2D(256, 5, padding='same')(x)
x = layers.LeakyReLU()(x)
x = layers.Conv2D(256, 5, padding='same')(x)
x = layers.LeakyReLU()(x)
x = layers.Conv2D(channels, 7, activation='tanh', padding='same')(x)
generator = keras.models.Model(generator_input, x)
generator.summary()

```

Y un ejemplo de programación del modelo Discriminador sería algo así:

```

discriminator_input = layers.Input(shape=(height, width, channels))
x = layers.Conv2D(128, 3)(discriminator_input)
x = layers.LeakyReLU()(x)
x = layers.Conv2D(128, 4, strides=2)(x)
x = layers.LeakyReLU()(x)
x = layers.Conv2D(128, 4, strides=2)(x)
x = layers.LeakyReLU()(x)
x = layers.Conv2D(128, 4, strides=2)(x)
x = layers.LeakyReLU()(x)
x = layers.Flatten()(x)
x = layers.Dropout(0.4)(x)
x = layers.Dense(1, activation='sigmoid')(x)
discriminator = keras.models.Model(discriminator_input, x)
discriminator.summary()
discriminator_optimizer = keras.optimizers.RMSprop(
    lr=0.0008,
    clipvalue=1.0,
    decay=1e-8)
discriminator.compile(optimizer=discriminator_optimizer,
    loss='binary_crossentropy')

```

Finalmente, construir el sistema completo implica "conectar" ambos modelos:

```

discriminator.trainable = False
gan_input = keras.Input(shape=(latent_dim,))
gan_output = discriminator(generator(gan_input))
gan = keras.models.Model(gan_input, gan_output)
gan_optimizer = keras.optimizers.RMSprop(lr=0.0004, clipvalue=1.0, decay=1e-8)
gan.compile(optimizer=gan_optimizer, loss='binary_crossentropy')

```

El proceso de entrenamiento no es sencillo. Requiere ajustes finos y una monitorización cuidadosa.

Para saber más

Si quieras ver la programación y entrenamiento de una red GAN, te recomendamos acceder a estos ejemplos explicados, guardando una copia para poder editarlos y trabajar sobre ellos:

[Tutorial_gans](https://colab.research.google.com/github/lexfridman/mit-deep-learning/blob/master/tutorial_gans/tutorial_gans.ipynb) (https://colab.research.google.com/github/lexfridman/mit-deep-learning/blob/master/tutorial_gans/tutorial_gans.ipynb)

TF-GAN

Tutorial

(https://colab.research.google.com/github/tensorflow/gan/blob/master/tensorflow_gan/examples/colab_notebooks/tfgan_tutorial.ipynb)

Autoevaluación

Rellena los huecos para completar esta frase del tema

El sistema GAN está listo cuando el consigue generar casos que el da por válidos.

Enviar

El sistema está listo cuando el generador consigue generar casos que el discriminador da por válidos.