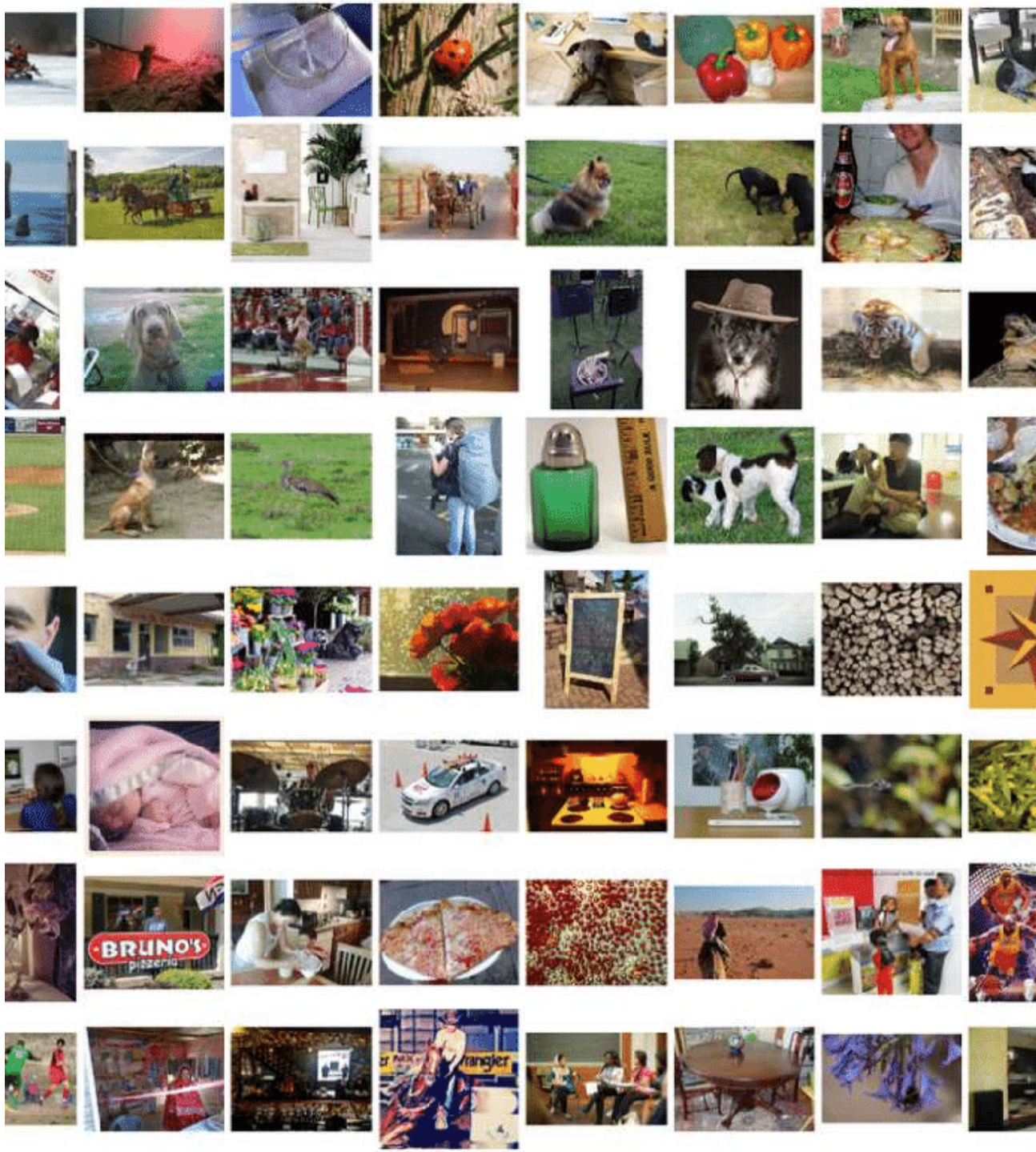
# 5.3 Usando un Convnet preentrenado

- Método para aplicar *Deep Learning* sobre bases de datos pequeñas.
- Un convnet preentrenado es una DNN guardada que ha sido entrenada previamente con una base de datos grande.
  - Aplicación en tareas de clasificación de imágenes a gran escala.
    -Una red entrenada para un tipo de trabajo puede ser *reutilizada* en un problema totalmente diferente.
  - Imagenet (animales y objetos) —> Reconocimiento mobiliario
  - La **portabilidad del aprendizaje** (*transfer learning*) es una gran ventaja de las DNN sobre otras técnicas.

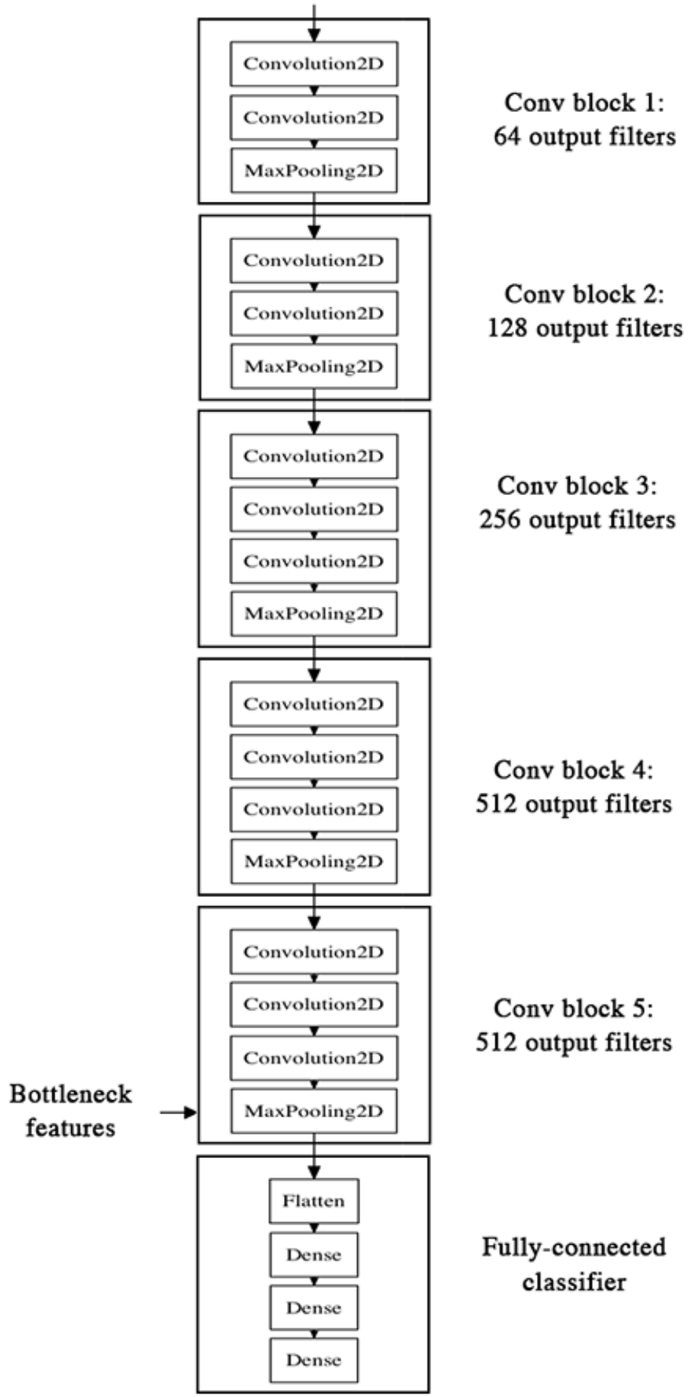
# Datos ImageNet

- 1.4 millones de imágenes, 1000 clases.
- Contiene diferentes razas de perros y gatos.
- Esperamos que funcione bien sobre el problema *dogs-vs-cats*.

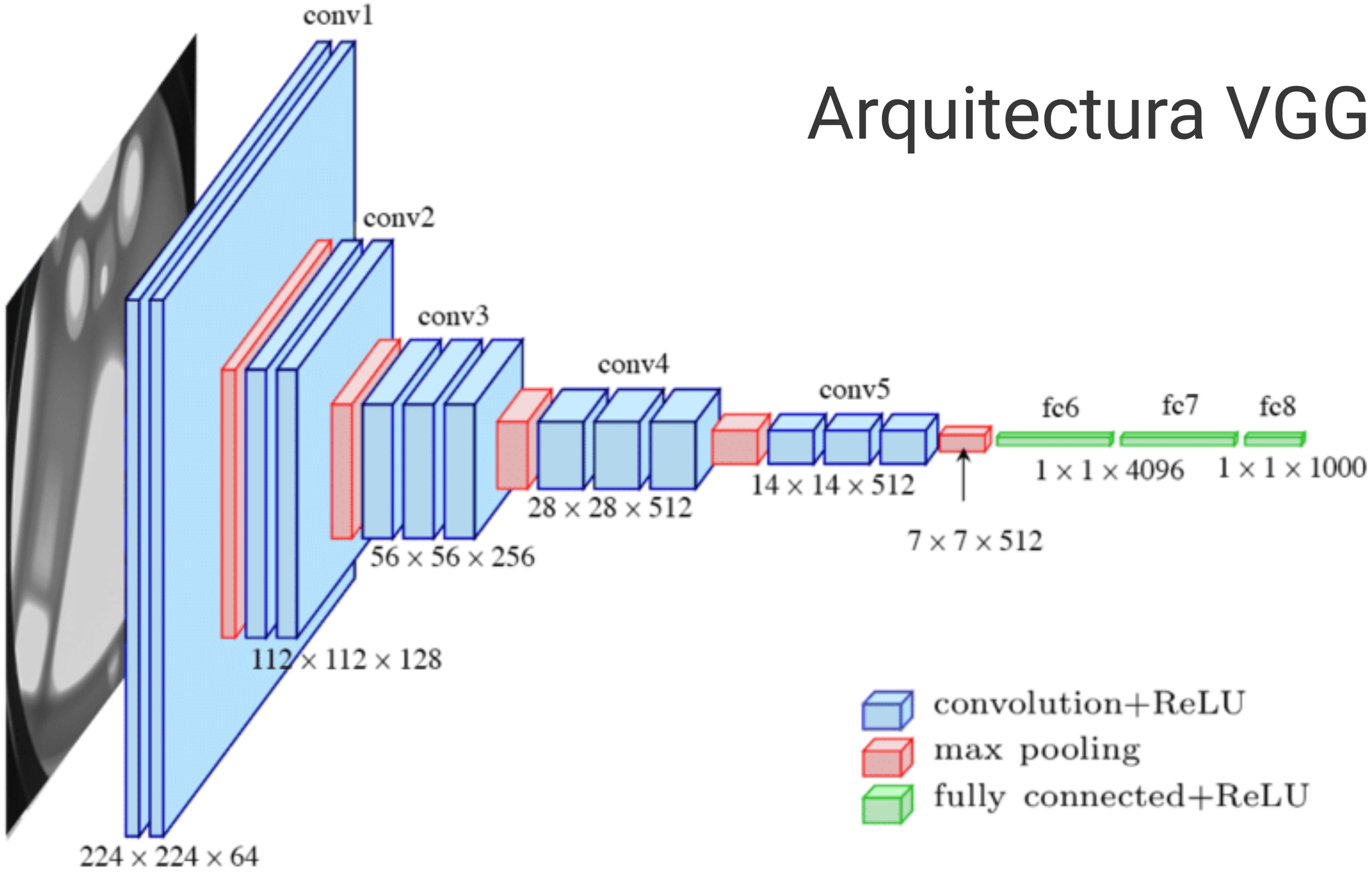


# Arquitectura VGG16

- Karen Simonyan & Andrew Zisserman (2014)
- Es un modelo algo viejo
  - Otros modelos son mejores y menos pesados
    - VGG, ResNet, Inception, Inception_esNet, Xception…



# Arquitectura VGG16



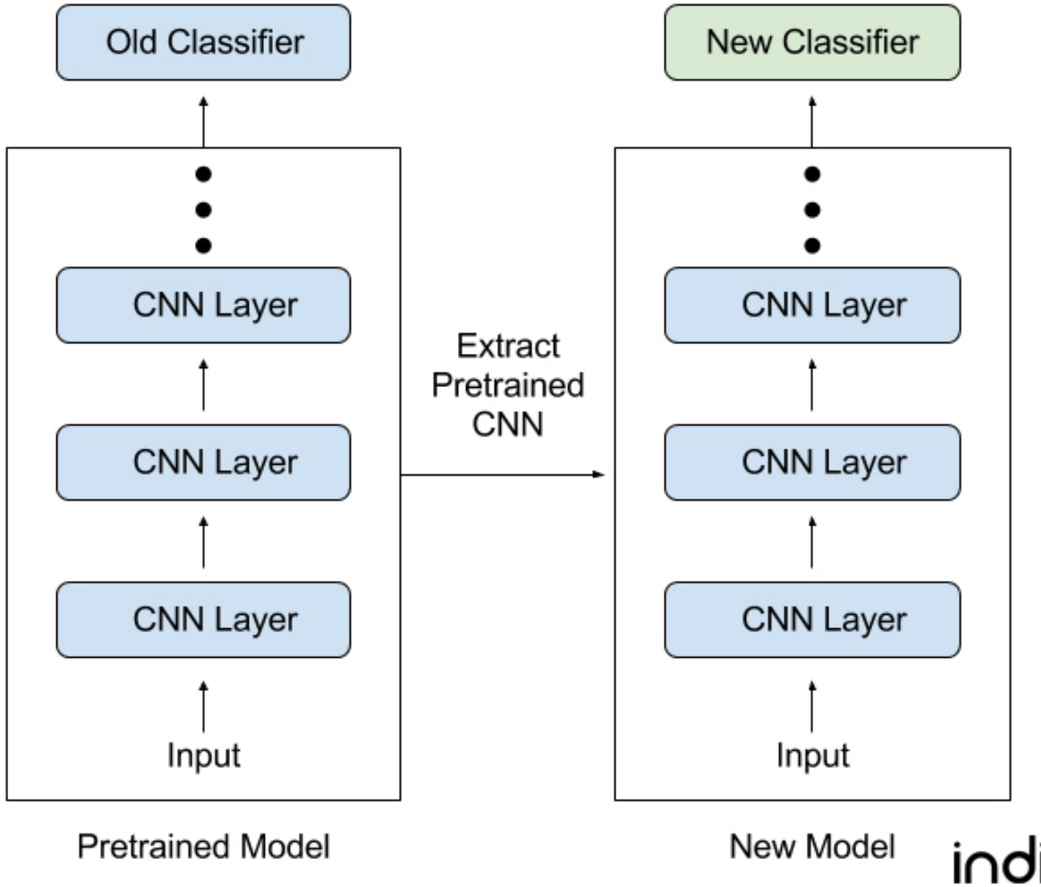
# *Feature extraction* (FE)

- FE y *fine-tuning*.
  - Dos maneras de usar una red preentrenada.
- FE: usar representaciones aprendidas previamente.
  - Ejecutar las representaciones con un nuevo clasificador.
- Estructura convnets.
  - Empiezan con una serie de capas *pooling* y de convolución: **base convolucional**.
  - Acaban con un clasificador conectado densamente.

- FE consiste en extraer la base convolucional.
  - Ejecutar DNN sobre nuevos datos
  - Extraer un nuevo clasificador

¿Por qué usar sólo la base convolucional y no la densa?

- Representaciones convolucionales son más generales.
- Representaciones del clasificador denso son específicas.




indico

# Nivel de generalización y reusabilidad

- El nivel de generalización y reusabilidad de las representaciones de cada capa depende de la profundidad en el modelo
  - Primeras capas extraen información más general: ejes, colores, texturas…
  - Capas más profundas extraen información más precisa: oreja de gato, ojo de perro.
  - Si la base de datos difiere mucho de la original utilizada para entrenar la red:
    - Utilizar sólo las primeras capas
- Nuestro caso
  - Imagenet contiene muchas imágenes de perros y gatos
  - Podríamos utilizar hasta la información contenida en las capas densas…
  - Pero no lo vamos a hacer.

# Modelos de clasificación de imágenes en Keras

- Todos preentrenados en la base ImageNet
- Importar desde el módulo `keras.applications`:

1. Xception
2. Interception V3
3. ResNet50
4. VGG16
5. VGG19
6. Mobile Net

# Importación de VGG16

- La importación del modelo VGG16 es muy pesada

  Solución aportada por
  *José Antonio Vázquez*

- Copiar la dirección[1] que aparece en el recuadro de *Jupyter* cuando se ejecuta por primera vez VGG16:

```
# Versión TensorFlow 1
from keras.applications import VGG16

# Versión Tensorflow 2
VGG16 = tf.keras.applications.VGG16
```

- Desde una terminal *Lynux*[2] descargar el modelo con `wget <<dire. https>>`
- Poner el modelo en el directorio `.keras/models`.

---

[1] https://github.com/fchollet/deep-learning-models/releases/download/v0.1/vgg16*weights*tf*dim*ordering*tf*kernels_notop.h5

[2] Desde Mac, instalar *Homebrew* e instalar *wget*: `> brew install wget`

# Notas sobre la importación del modelo

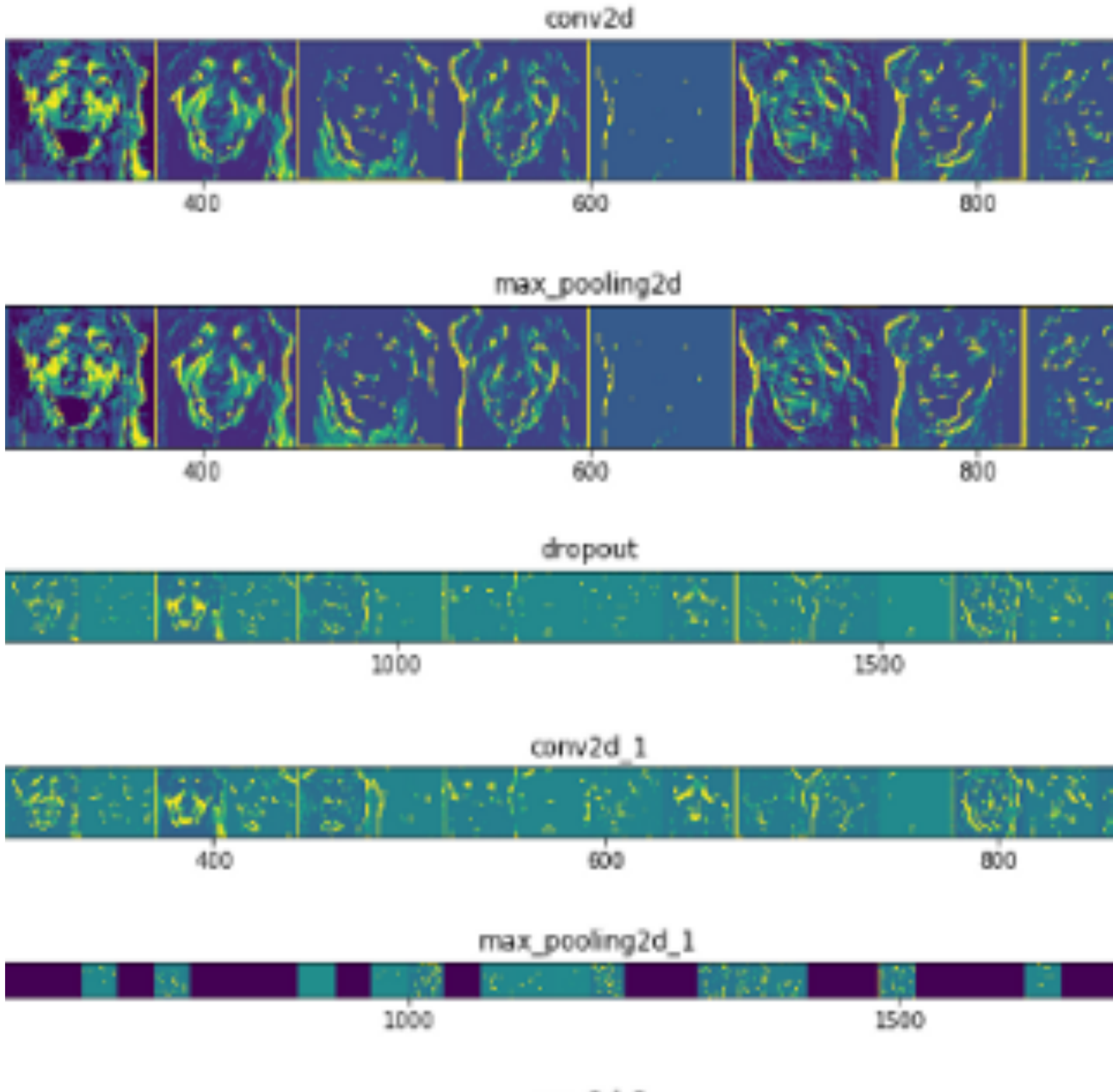
```
conv_base = VGG16(weights='imagenet',
                  include_top=False,
                  input_shape=(150, 150, 3))
```

## Argumentos

- `weights`: indica con qué pesos se inicializa el modelo.
  - `None`: aleatorios
  - `imagenet`: ImageNet preentrenado
  - `<file>`: ruta a un archivo con los pesos
- `include_top`: `True` o `False` según se use o no el clasificador denso de la red.
  - Ponemos `False` porque no nos interesan 1000 clases de ImageNet sino sólo 2 (perros y gatos).
- `input_shape`: Características de los tensores de imágenes.
  - Si no se incluye, el DNN puede procesar imágenes de cualquier tamaño.

# *Feature maps*

- Se importan múltiples capas `Convolution2D` y `Maxpooling2D`
- Total $\approx 15\mathrm{M}$ de parámetros.
- FM final de (4, 4, 512)
- Sobre este FM ponemos el clasificador conectado densamente.



# Alternativa 1

- Ejecutar base convolutiva sobre el conjunto de datos
  - Grabar salida sobre *array* Numpy.
  - Utilizar este array como entrada sobre un clasificador conectado densamente.
- Solución rápida y sencilla
  - La base convolutiva se ejecuta sólo una vez sobre cada imagen.
    - La base convolutiva es la parte más lenta del proceso.
- No permite *Data Augmentation*.

# Alternativa 2

## No ejecutar en clase

- Extender el modelo `con_base` que hemos construido, añadiendo capas densas.
  - Permite usar *Data Augmentation*
  - Es mucho más tardado que la alternativa 1 (5h)

# Extracción rápida sin *Data Augmentation*

- Ejecutar `ImageDataGenerator`
  - Imágenes —> *arrays* Numpy y etiquetas.
  - Obtener características de las imágenes.
    - Método `predict` del modelo `conv_base`
    - *Output* formato (4, 4, 512)
- Alimentar clasificador densamente conectado
  - Entrada FM (4, 4, 512)
  - Aplanar a $4 \times 4 \times 512 = 8192$.
  - Definir modelo de clasificador
    -Utilizar `dropout`para regularizar
- $Validation\ accuracy \sim 90\%$
  - Mucho mejor que los resultados directos (Notebook 5.2, *Bases de datos pequeñas*, $acc \approx 70\%$)
  - Mucho mejor que los resultados con *data augmentation* ($acc \approx 82\%$)
- Sobreajuste después de muy pocas épocas.
  - Incluso con regularización *Dropout* del 50%:
    `model.add(layers.Dropout(0.5))`

# NO EJECUTAR DURANTE LA CLASE el modelo convolutivo

- El primer modelo denso se ejecuta sin problemas, pero el segundo modelo convolutivo es muy tardado:
  - **10 min por época**
  - **5h en total**

# Extracción con *Data Augmentation*

# NO EJECUTAR

La ejecución es tan lenta que debe disponerse de GPUs.

- Extender el modelo `conv_base`con clasificador
- Simplemente añadir unas capas a `conv_base`:
- Parámetros: ~15M `conv_base` + ~2M clasificador

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

# Compilación

- Es muy importante **congelar** (*freeze*) el modelo.
  - Impedir actualización de pesos durante entrenamiento.
    - Los pesos iniciales de las capas densas son aleatorios.
    - Propagarán grandes cambios y destruirán lo aprendido por VGG16
  - `conv_base.trainable = False`
    - Sólo se actualizan pesos de capas densas.
- Compilar el modelo
  - Si se modifica el atributo `trainable` después de compilar, no se tendrá en cuenta a menos que se recompile.

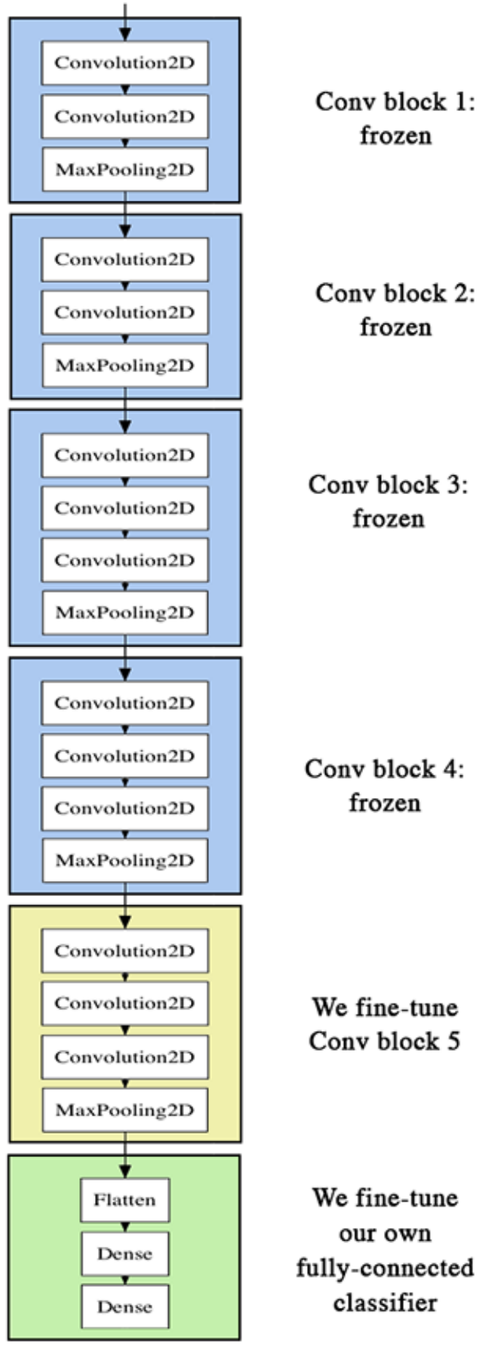
| | **Scratch** | **Data Augmentation** | **Pretrained DA** | **Pretrained no-DA** |
|---|---|---|---|---|
| Acc | 70% | 82% | 92% | **96%** |

# *Fine-tuning* (FT)

NO EJECUTAR

Modelo alternativo a *Feature Extraction*

- Consiste en *descongelar* algunas capas convolutivas.
- Se entrenan a la vez el clasificador y las capas descongeladas.
- Permite que las capas convolutivas con la información más abstracta se puedan reajustar para el problema actual.
- La parte de la red susceptible a FT son los últimos bloques de la base convolutiva.
  - Después de la penúltima capa de `MaxPooling2D`.

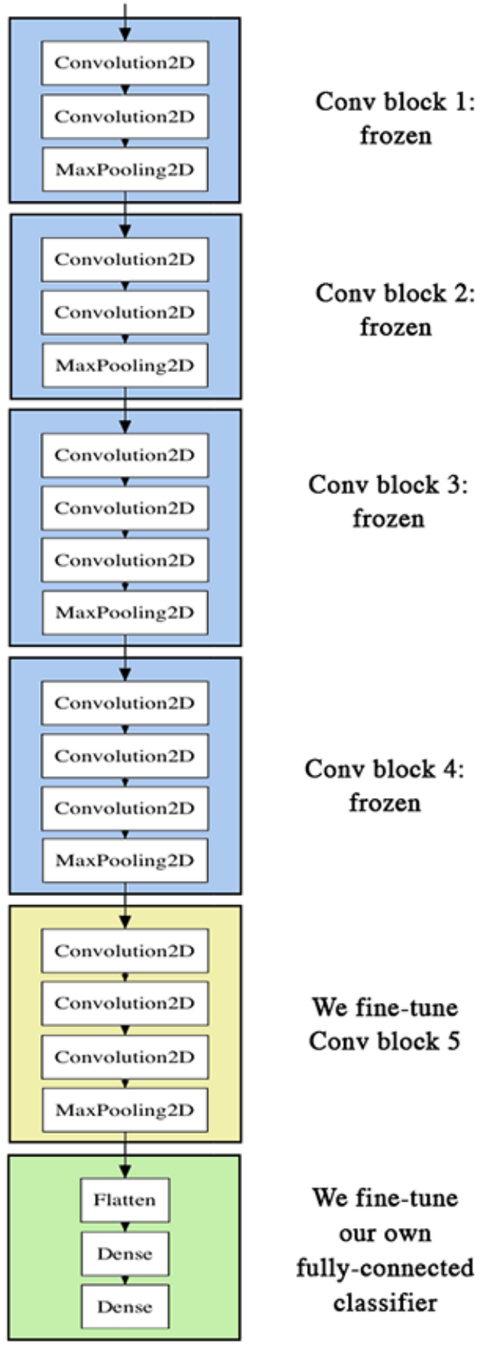


La parte descongelada se especifica después de un entrenamiento inicial para no perder información de la red original.

1. Añadir red densa tras la base convolutiva.
2. Congelar la base convolutiva.
3. Entrenar la red densa (clasificador).
4. Descongelar *algunas* de la últimas capas de la base convolutiva.
5. Entrenar juntos el clasificador y las capas descongeladas.

# Código de *Fine tunning*

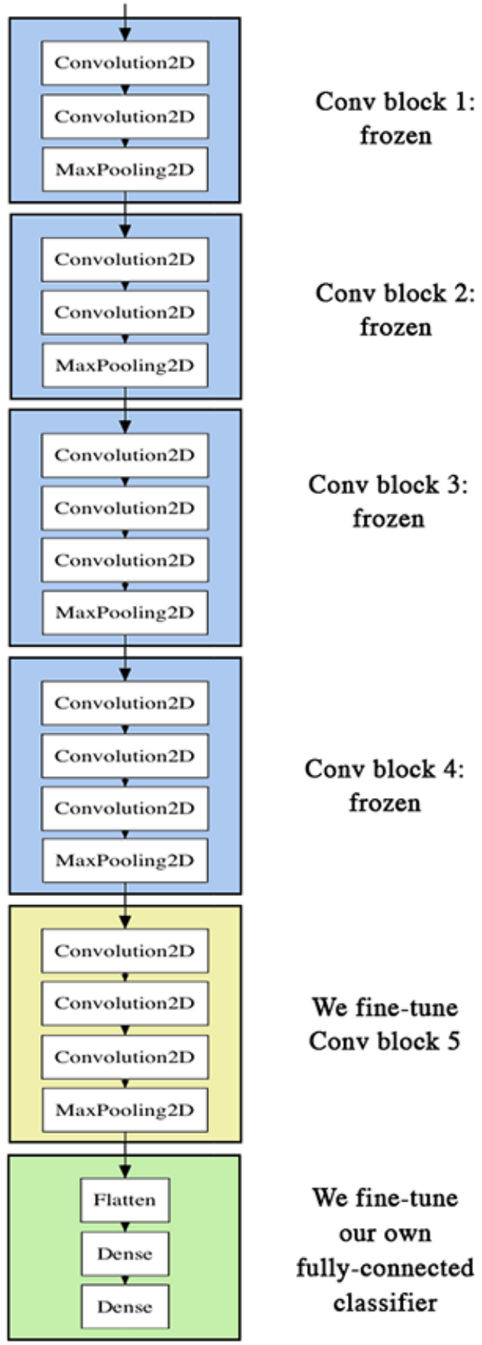
- Descongelamos las 4 últimas capas.
  - Bloques de capas congeladas del 1 al 4.
  - Descongelamos bloque 5.
- Las primeras capas de la base convolutiva representan características generables y reutilizabas.
- Sólo necesitamos reentrenar las capas más específicas.
- Cuanto más parámetros se entrenen, mayor el riesgo de sobreajuste.



# Cambios al código para *fine tuning*

```
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
```

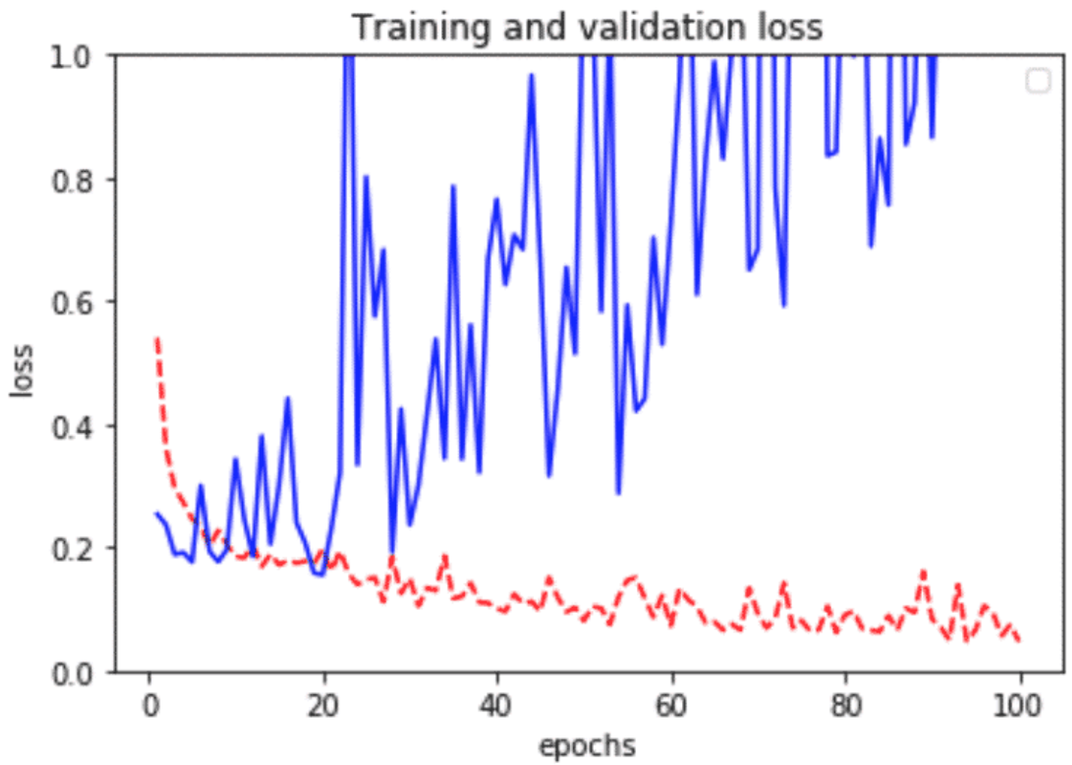


# Compilación

- Optimizador `RMSProp`
- Ritmo de aprendizaje muy pequeño
  - `lr = 1e-5`

# Gráficas

- Las curvas para `accuracy` y `loss` son muy ruidosas

- Suaviza las curvas con una función de media móvil exponencial



- Las curvas de validación se ven más limpias, con una mejora del ~1% respecto al *Feature Extraction*.

# ¿Por qué mejora la *accuracy* si el *loss* no decrece?

- Lo que importa para la *accuracy* es la distribución de los valores de *loss*, no su media
- *Accuracy* es el resultado de poner un límite binario a la probabilidad de clase predicha por el modelo.
  - El modelo puede mejorar incluso sin hacerlo el promedio de loss.

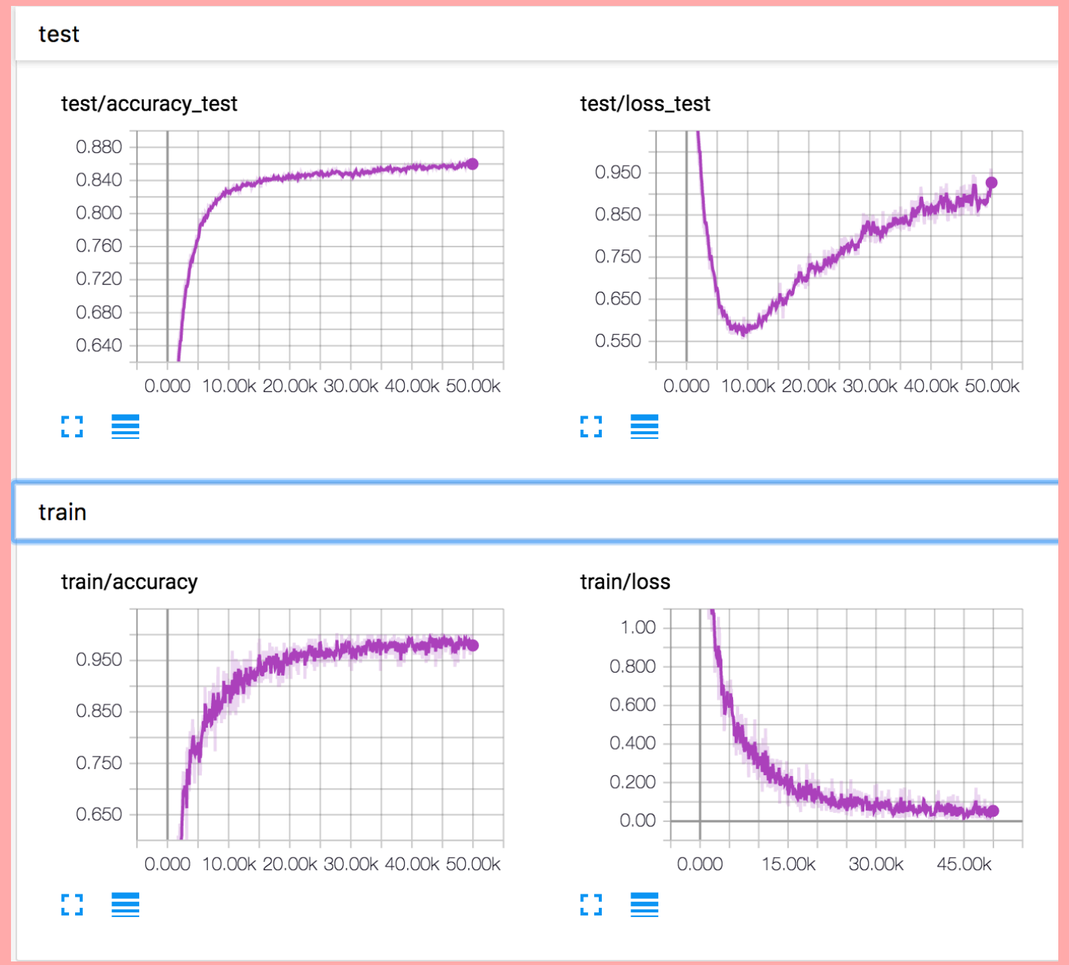
# ¿Por qué mejora la *accuracy* si la pérdida no decrece?

## Ampliación

No están totalmente correlacionados:

- La pérdida mide la diferencia entre la predicción (probabilidad) y la clase (0 ó 1).
- La *accuracy* mide la diferencia entre la clase predicha (0 ó 1) y la clase real (0 ó 1).

Si la predicción cruda cambia, también lo hace la pérdida, pero la *accuracy* es más *resiliente* en cuanto que el cambio debe situar la probabilidad al otro lado del límite de predición (a menudo $p_{thr} = 0.5$).



# Evaluación del modelo

- 97% accuracy similar a resultados de *Kaggle* **usando sólo el 10% de las imágenes** de la base de datos.

Dogs vs. Cats

Create an algorithm to distinguish dogs from cats

k Kaggle • 213 teams • 2014

| # | Team Name | Score |
|---|---|---|
| 1 | Pierre Sermanet | 0.98914 |
| 8 | fastml.com/ cats-and-dogsn | 0.98000 |
| 23 | mymo | 0.97017 |

# Recapitulación

- *Convnets* son actualmente el **mejor modelo** de *Machine Learning* para problemas de *computer-vision*.
- Para bases de datos pequeñas, el principal problema es el sobreajuste.
  - Data augmentation.
  - Reusar un *convnet* ya existente
    - *Feature extraction*
    - *Fine-tuning*

# Lecturas adicionales

- Rohit Thakur, 2019, Towards Data Science: Step by step VGG16 implementation in Keras for beginners
- Renu Khandelwall, 2020, Towards Data Science: Convolutional Neural Network: Feature Map and Filter Visualization

# Visualizando cómo aprenden los *convnets*

*El próximo día*

KEEP
CALM
AND
SEE YOU
NEXT CLASS