Fundamentos de las Redes Neuronales

Redes Convolucionales (CNNs) II

Modelos generativos profundos

Clase 9

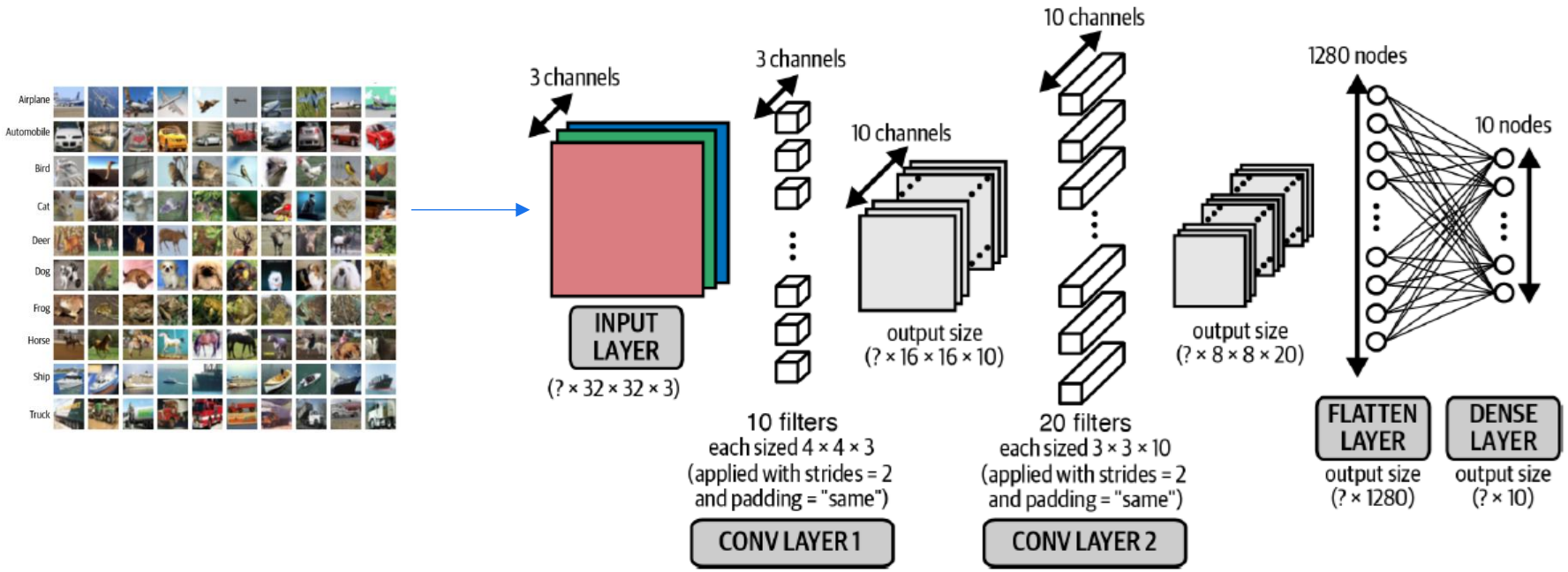
Dra. Wendy Aguilar

Modelos Generativos Profundos

UN ENFOQUE DESDE LA CREATIVIDAD COMPUTACIONAL

Ejercicio

De manera similar a MNIST, implementa esta red convolucional para CIFAR10.

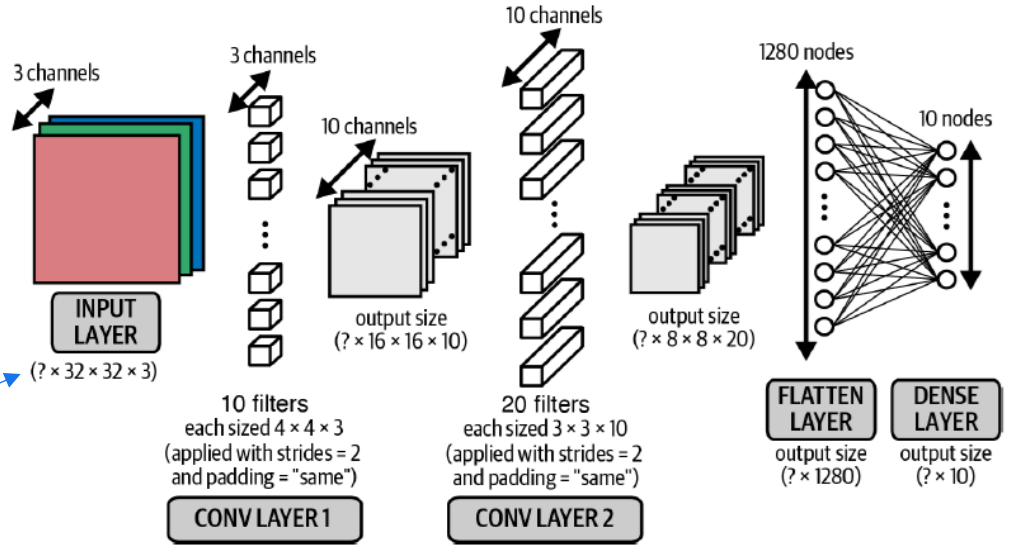


Predicción de la clase

* Nota: sin las mejoras de la diapositiva anterior.

CNN para Cifar10

Versión 1



Predicción de la clase

None indica que se le puede pasar cualquier cantidad de imágenes a la red de manera simultánea.

```
input_layer = layers.Input((32, 32, 3))

x = layers.Conv2D(filters=10, kernel_size=(4,4), strides=2, padding="same")(input_layer)
x = layers.Conv2D(filters=20, kernel_size=(3,3), strides=2, padding="same")(x)
x = layers.Flatten()(x)
output_layer = layers.Dense(NUM_CLASSES, activation="softmax")(x)

model = models.Model(input_layer, output_layer)
```

* Nota: Por default asigna una función de activación lineal.

Capas de pooling
(agrupamiento, reducción, submuestreo)

vs

Convolución con stride > 1

```python
input_layer = layers.Input(shape=(28, 28,1))
x = layers.ZeroPadding2D(padding=2)(input_layer)
x = data_aug(x)
x = layers.Conv2D(filters=16, kernel_size=(5,5), padding="valid", strides=(1,1), activation="relu")(x)
x = layers.MaxPooling2D(pool_size=(2, 2))(x)
x = layers.Conv2D(filters=32, kernel_size=(5,5), padding="same", strides=(1,1), activation="relu")(x)
x = layers.MaxPooling2D(pool_size=(2, 2))(x)
x = layers.Conv2D(64, kernel_size=(5, 5), padding="same", strides=(1,1), activation="relu")(x)
x = layers.MaxPooling2D(pool_size=(2, 2))(x)
x = layers.Flatten()(x)
x = layers.Dense(120, activation="relu")(x)
x = layers.Dense(84, activation="relu")(x)
output_layer = layers.Dense(NUM_CLASES, activation="softmax")(x)

model = models.Model(inputs=input_layer, outputs=output_layer)
```

```python
input_layer = layers.Input((32, 32, 3))

x = layers.Conv2D(filters=10, kernel_size=(4,4), strides=2, padding="same")(input_layer)
x = layers.Conv2D(filters=20, kernel_size=(3,3), strides=2, padding="same")(x)
x = layers.Flatten()(x)
output_layer = layers.Dense(NUM_CLASSES, activation="softmax")(x)

model = models.Model(input_layer, output_layer)
```

Ambas sirven para **reducir la resolución espacial** (alto × ancho) de los mapas de características en una red convolucional.

¿Qué hacen?

- Resumen información local sin aprender nada (no tienen pesos entrenables).
- Aplican una **regla fija** (máximo o promedio).

- Reducen resolución mientras se aprenden nuevas representaciones.
- Usan filtros entrenables para transformar los mapas de características y reducir su tamaño (saltando píxeles con stride).

Capas de pooling
(agrupamiento, reducción, submuestreo)

```
input_layer = layers.Input(shape=(28, 28,1))
x = layers.ZeroPadding2D(padding=2)(input_layer)
x = data_aug(x)
x = layers.Conv2D(filters=16, kernel_size=(5,5), padding="valid", strides=(1,1), activation="relu")(x)
x = layers.MaxPooling2D(pool_size=(2, 2))(x)
x = layers.Conv2D(filters=32, kernel_size=(5,5), padding="same", strides=(1,1), activation="relu")(x)
x = layers.MaxPooling2D(pool_size=(2, 2))(x)
x = layers.Conv2D(64, kernel_size=(5, 5), padding="same", strides=(1,1), activation="relu")(x)
x = layers.MaxPooling2D(pool_size=(2, 2))(x)
x = layers.Flatten()(x)
x = layers.Dense(120, activation="relu")(x)
x = layers.Dense(84, activation="relu")(x)
output_layer = layers.Dense(NUM_CLASES, activation="softmax")(x)

model = models.Model(inputs=input_layer, outputs=output_layer)
```

vs

Convolución con stride > 1

```
input_layer = layers.Input((32, 32, 3))

x = layers.Conv2D(filters=10, kernel_size=(4,4), strides=2, padding="same")(input_layer)
x = layers.Conv2D(filters=20, kernel_size=(3,3), strides=2, padding="same")(x)
x = layers.Flatten()(x)
output_layer = layers.Dense(NUM_CLASSES, activation="softmax")(x)

model = models.Model(input_layer, output_layer)
```

Útiles para:

- Invariancia a pequeñas traslaciones o deformaciones.
  Ej: Si un borde se mueve un poco, el valor máximo seguirá representando que "el borde está ahí".
- Reducir la dimensionalidad (de las entradas de las capas).
- Evitar sobreajuste al forzar a la red a enfocarse en las características más importantes.
- Suavizar ruido local (en el caso de AveragePooling).

- Aprender qué combinar y cómo al hacer el downsampling.
  no solo reducen tamaño, sino que crean representaciones más abstractas.
- Reducir resolución espacial mientras aumentan la profundidad (canales).
- Captar patrones complejos (bordes, texturas, formas) mientras se reduce la dimensionalidad.

Capas de pooling
(agrupamiento, reducción, submuestreo)

```
input_layer = layers.Input(shape=(28, 28,1))
x = layers.ZeroPadding2D(padding=2)(input_layer)
x = data_aug(x)
x = layers.Conv2D(filters=16, kernel_size=(5,5), padding="valid", strides=(1,1), activation="relu")(x)
x = layers.MaxPooling2D(pool_size=(2, 2))(x)
x = layers.Conv2D(filters=32, kernel_size=(5,5), padding="same", strides=(1,1), activation="relu")(x)
x = layers.MaxPooling2D(pool_size=(2, 2))(x)
x = layers.Conv2D(64, kernel_size=(5, 5), padding="same", strides=(1,1), activation="relu")(x)
x = layers.MaxPooling2D(pool_size=(2, 2))(x)
x = layers.Flatten()(x)
x = layers.Dense(120, activation="relu")(x)
x = layers.Dense(84, activation="relu")(x)
output_layer = layers.Dense(NUM_CLASES, activation="softmax")(x)

model = models.Model(inputs=input_layer, outputs=output_layer)
```

vs

Convolución con stride > 1

```
input_layer = layers.Input((32, 32, 3))

x = layers.Conv2D(filters=10, kernel_size=(4,4), strides=2, padding="same")(input_layer)
x = layers.Conv2D(filters=20, kernel_size=(3,3), strides=2, padding="same")(x)
x = layers.Flatten()(x)
output_layer = layers.Dense(NUM_CLASSES, activation="softmax")(x)

model = models.Model(input_layer, output_layer)
```

Limitaciones:

·No generan nuevas características, solo resumen las existentes.

·No aprenden qué es importante: siempre aplican la misma regla.

Ventajas:

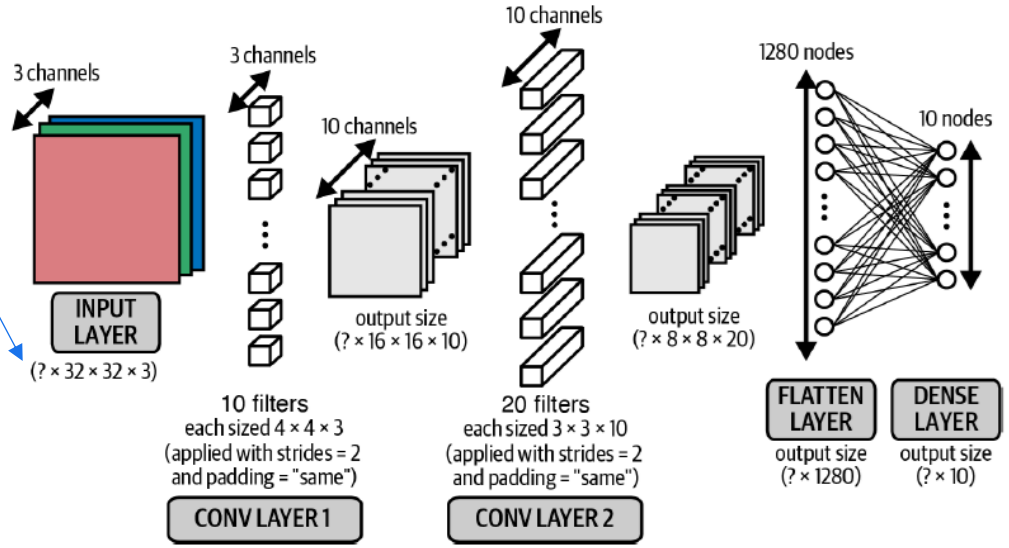
- Aprenden representaciones más potentes que las obtenidas por pooling.
- Reemplazan cada vez más al pooling en arquitecturas modernas (ResNet, EfficientNet, etc.).

Ambos son **herramientas válidas de diseño** en redes convolucionales.

CNN para Cifar10

Versión 1

None indica que se le puede pasar cualquier cantidad de imágenes a la red de manera simultánea.



Predicción de la clase

cnn_cifar10_version1.ipynb

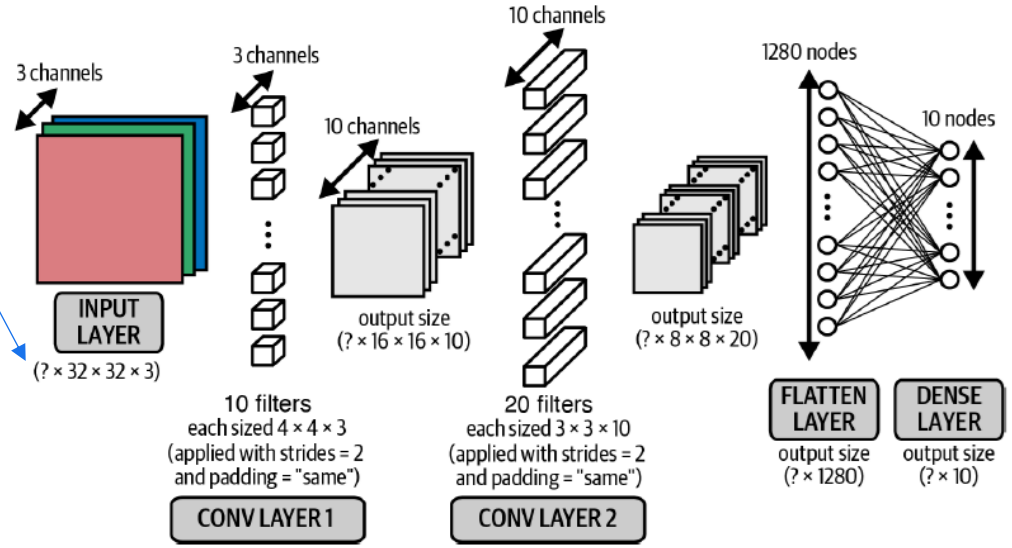
Google colab

```
input_layer = layers.Input((32, 32, 3))

x = layers.Conv2D(filters=10, kernel_size=(4,4), strides=2, padding="same")(input_layer)
x = layers.Conv2D(filters=20, kernel_size=(3,3), strides=2, padding="same")(x)
x = layers.Flatten()(x)
output_layer = layers.Dense(NUM_CLASSES, activation="softmax")(x)

model = models.Model(input_layer, output_layer)
```

CNN para Cifar10

Versión 1

None indica que se le puede pasar cualquier cantidad de imágenes a la red de manera simultánea.



Predicción de la clase

cnn_cifar10_version1.ipynb

Google colab

```
input_layer = layers.Input((32, 32, 3))

x = layers.Conv2D(filters=10, kernel_size=(4,4), strides=2, padding="same")(input_layer)
x = layers.Conv2D(filters=20, kernel_size=(3,3), strides=2, padding="same")(x)
x = layers.Flatten()(x)
output_layer = layers.Dense(NUM_CLASSES, activation="softmax")(x)

model = models.Model(input_layer, output_layer)
```

```
Epoch 10/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 4s 3ms/step - accuracy: 0.4348 - loss: 1.6719 - val_accuracy: 0.4130 - val_loss: 1.7112
10/10 ━━━━━━━━━━━━━━━━━━━━ 1s 16ms/step - accuracy: 0.4163 - loss: 1.7065
[1.71116304397583, 0.4129999876022339]
```

Limitaciones de nuestra red básica (2 conv + dense)

1. Sin funciones de activación

→ Las capas convolucionales son lineales, y dos transformaciones lineales siguen siendo lineales → la red no puede aprender patrones complejos.

Limitada a fronteras de decisión lineales

```
Epoch 20/25
1563/1563 ——— accuracy: 0.4465 - loss: 1.6335
Epoch 21/25
1563/1563 ——— accuracy: 0.4409 - loss: 1.6439
Epoch 22/25
1563/1563 ——— accuracy: 0.4460 - loss: 1.6322
Epoch 23/25
1563/1563 ——— accuracy: 0.4446 - loss: 1.6393
Epoch 24/25
1563/1563 ——— accuracy: 0.4423 - loss: 1.6411
Epoch 25/25
1563/1563 ——— accuracy: 0.4475 - loss: 1.6320
```

Se estanca

```
Epoch 45/45
1563/1563 ——— accuracy: 0.4578 - loss: 1.6033
```

Test: accuracy: 0.4020 - loss: 1.7559

```
input_layer = layers.Input((32, 32, 3))
x = layers.Conv2D(filters=10, kernel_size=(4,4), strides=2, padding="same")(input_layer)
x = layers.ReLU()(x)
x = layers.Conv2D(filters=20, kernel_size=(3,3), strides=2, padding="same")(x)
x = layers.ReLU()(x)
x = layers.Flatten()(x)
output_layer = layers.Dense(NUM_CLASSES, activation="softmax")(x)

model = models.Model(input_layer, output_layer)
```

cnn_cifar10_version2_relu.ipynb

Google colab

```
Epoch 20/25
1563/1563 ——— accuracy: 0.6642 - loss: 0.9660
Epoch 21/25
1563/1563 ——— accuracy: 0.6675 - loss: 0.9595
Epoch 22/25
1563/1563 ——— accuracy: 0.6692 - loss: 0.9447
Epoch 23/25
1563/1563 ——— accuracy: 0.6708 - loss: 0.9483
Epoch 24/25
1563/1563 ——— accuracy: 0.6721 - loss: 0.9451
Epoch 25/25
1563/1563 ——— accuracy: 0.6738 - loss: 0.9369
```

Sigue bajando

Test: accuracy: 0.6163 - loss: 1.0968

Limitaciones de nuestra red básica (2 conv + dense)

2. Muy poca profundidad

→ Solo cuenta con dos capas convolucionales, por lo que aprende patrones muy locales y simples

→ no logra construir representaciones jerárquicas necesarias para reconocer objetos complejos.

```
Epoch 20/25
1563/1563 — accuracy: 0.6642 - loss: 0.9660
Epoch 21/25
1563/1563 — accuracy: 0.6675 - loss: 0.9595
Epoch 22/25
1563/1563 — accuracy: 0.6692 - loss: 0.9447
Epoch 23/25
1563/1563 — accuracy: 0.6708 - loss: 0.9483
Epoch 24/25
1563/1563 — accuracy: 0.6721 - loss: 0.9451
Epoch 25/25
1563/1563 — accuracy: 0.6738 - loss: 0.9369
```

Sigue bajando

Test: accuracy: 0.6163 - loss: 1.0968

```
input_layer = layers.Input((32, 32, 3))
x = layers.Conv2D(filters=32, kernel_size=(3,3), strides=1, padding="same")(input_layer)
x = layers.ReLU()(x)
x = layers.Conv2D(filters=32, kernel_size=(3,3), strides=2, padding="same")(x)
x = layers.ReLU()(x)
x = layers.Conv2D(filters=64, kernel_size=(3,3), strides=1, padding="same")(x)
x = layers.ReLU()(x)
x = layers.Conv2D(filters=64, kernel_size=(3,3), strides=2, padding="same")(x)
x = layers.ReLU()(x)
x = layers.Flatten()(x)
x = layers.Dense(128)(x)
x = layers.ReLU()(x)
output_layer = layers.Dense(NUM_CLASSES, activation="softmax")(x)

model = models.Model(input_layer, output_layer)
```

cnn_cifar10_version3_mas_capas.ipynb

Google colab

Limitaciones de nuestra red básica (2 conv + dense)

2. Muy poca profundidad

→ Solo cuenta con dos capas convolucionales, por lo que aprende patrones muy locales y simples

→ no logra construir representaciones jerárquicas necesarias para reconocer objetos complejos.

```
Epoch 20/25
1563/1563 — accuracy: 0.6642 - loss: 0.9660
Epoch 21/25
1563/1563 — accuracy: 0.6675 - loss: 0.9595
Epoch 22/25
1563/1563 — accuracy: 0.6692 - loss: 0.9447
Epoch 23/25
1563/1563 — accuracy: 0.6708 - loss: 0.9483
Epoch 24/25
1563/1563 — accuracy: 0.6721 - loss: 0.9451
Epoch 25/25
1563/1563 — accuracy: 0.6738 - loss: 0.9369
```

Sigue bajando

Test: accuracy: 0.6163 - loss: 1.0968

```
input_layer = layers.Input((32, 32, 3))
x = layers.Conv2D(filters=32, kernel_size=(3,3), strides=1, padding="same")(input_layer)
x = layers.ReLU()(x)
x = layers.Conv2D(filters=32, kernel_size=(3,3), strides=2, padding="same")(x)
x = layers.ReLU()(x)
x = layers.Conv2D(filters=64, kernel_size=(3,3), strides=1, padding="same")(x)
x = layers.ReLU()(x)
x = layers.Conv2D(filters=64, kernel_size=(3,3), strides=2, padding="same")(x)
x = layers.ReLU()(x)
x = layers.Flatten()(x)
x = layers.Dense(128)(x)
x = layers.ReLU()(x)
output_layer = layers.Dense(NUM_CLASSES, activation="softmax")(x)

model = models.Model(input_layer, output_layer)
```

```
Epoch 20/25
1563/1563 — accuracy: 0.9803 - loss: 0.0558
Epoch 21/25
1563/1563 — accuracy: 0.9841 - loss: 0.0481 -
Epoch 22/25
1563/1563 — accuracy: 0.9835 - loss: 0.0472
Epoch 23/25
1563/1563 — accuracy: 0.9845 - loss: 0.0478 -
Epoch 24/25
1563/1563 — accuracy: 0.9847 - loss: 0.0436 -
Epoch 25/25
```

Está sobreajustando.

Test: accuracy: 0.6867 - loss: 2.5334

cnn_cifar10_version3_mas_capas.ipynb

Google colab

Necesitamos aplicar técnicas de regularización para reducir el sobreajuste y mejorar la generalización.

Limitaciones de nuestra red básica (2 conv + dense)

3. Dataset pequeño y muy variable

→ Las CNN tienden a sobreajustar y no aprenden invariancias; el **aumento de datos** amplía la variación e impone esas invariancias para generalizar mejor.

Ayuda a regularizar

```
Epoch 20/25
1563/1563 ━━━━━━━━━━━━━━━━━━━━ accuracy: 0.9803 - loss: 0.0558
Epoch 21/25
1563/1563 ━━━━━━━━━━━━━━━━━━━━ accuracy: 0.9841 - loss: 0.0481 -
Epoch 22/25
1563/1563 ━━━━━━━━━━━━━━━━━━━━ accuracy: 0.9835 - loss: 0.0472
Epoch 23/25
1563/1563 ━━━━━━━━━━━━━━━━━━━━ accuracy: 0.9845 - loss: 0.0478 -
Epoch 24/25
1563/1563 ━━━━━━━━━━━━━━━━━━━━ accuracy: 0.9847 - loss: 0.0436 -
Epoch 25/25
```

Está sobreajustando.

Test: accuracy: 0.6867 - loss: 2.5334

```python
# Aumento de datos (se activa solo en training)
aug = tf.keras.Sequential([
    layers.RandomFlip("horizontal"),
    layers.RandomTranslation(0.1, 0.1),
], name="data_augmentation")

input_layer = layers.Input((32, 32, 3))
x = aug(input_layer)
x = layers.Conv2D(filters=32, kernel_size=(3,3), strides=1, padding="same")(x)
x = layers.ReLU()(x)
x = layers.Conv2D(filters=32, kernel_size=(3,3), strides=2, padding="same")(x)
x = layers.ReLU()(x)
x = layers.Conv2D(filters=64, kernel_size=(3,3), strides=1, padding="same")(x)
x = layers.ReLU()(x)
x = layers.Conv2D(filters=64, kernel_size=(3,3), strides=2, padding="same")(x)
x = layers.ReLU()(x)
x = layers.Flatten()(x)
x = layers.Dense(128)(x)
x = layers.ReLU()(x)
output_layer = layers.Dense(NUM_CLASSES, activation="softmax")(x)

model = models.Model(input_layer, output_layer)
```

cnn_cifar10_version4_aumento_datos.ipynb

Google colab

Limitaciones de nuestra red básica (2 conv + dense)

3. Dataset pequeño y muy variable

→ Las CNN tienden a sobreajustar y no aprenden invariancias; el **aumento de datos** amplía la variación e impone esas invariancias para generalizar mejor.

Ayuda a regularizar

```
Epoch 20/25
1563/1563 — · accuracy: 0.9803 - loss: 0.0558
Epoch 21/25
1563/1563 — accuracy: 0.9841 - loss: 0.0481 -
Epoch 22/25
1563/1563 —
Epoch 23/25 · accuracy: 0.9835 - loss: 0.0472
1563/1563 —
Epoch 24/25
1563/1563 — accuracy: 0.9845 - loss: 0.0478 -
Epoch 25/25
1563/1563 — accuracy: 0.9847 - loss: 0.0436 -
```

Está sobreajustando.

Test: accuracy: 0.6867 - loss: 2.5334

```python
# Aumento de datos (se activa solo en training)
aug = tf.keras.Sequential([
    layers.RandomFlip("horizontal"),
    layers.RandomTranslation(0.1, 0.1),
], name="data_augmentation")

input_layer = layers.Input((32, 32, 3))
x = aug(input_layer)
x = layers.Conv2D(filters=32, kernel_size=(3,3), strides=1, padding="same")(x)
x = layers.ReLU()(x)
x = layers.Conv2D(filters=32, kernel_size=(3,3), strides=2, padding="same")(x)
x = layers.ReLU()(x)
x = layers.Conv2D(filters=64, kernel_size=(3,3), strides=1, padding="same")(x)
x = layers.ReLU()(x)
x = layers.Conv2D(filters=64, kernel_size=(3,3), strides=2, padding="same")(x)
x = layers.ReLU()(x)
x = layers.Flatten()(x)
x = layers.Dense(128)(x)
x = layers.ReLU()(x)
output_layer = layers.Dense(NUM_CLASSES, activation="softmax")(x)

model = models.Model(input_layer, output_layer)
```

```
Epoch 20/25
1563/1563 — accuracy: 0.8173 - loss: 0.5246
Epoch 21/25
1563/1563 — accuracy: 0.8136 - loss: 0.5261
Epoch 22/25
1563/1563 — accuracy: 0.8198 - loss: 0.5136
Epoch 23/25
1563/1563 — accuracy: 0.8250 - loss: 0.4939
Epoch 24/25
1563/1563 — accuracy: 0.8220 - loss: 0.5044
Epoch 25/25
1563/1563 — accuracy: 0.8329 - loss: 0.4760
```

Test: accuracy: 0.8011 - loss: 0.6071

cnn_cifar10_version4_aumento_datos.ipynb

Google colab

Ya no hay sobreajuste

Limitaciones de nuestra red básica (2 conv + dense)

4. Solo escalamos los datos de entrada

¿Por qué escalamos los datos de entrada?

- Para que las activaciones iniciales estén en un rango controlado, evitando gradientes explosivos o inestables al inicio del entrenamiento.

Seguimos teniendo un problema

- A medida que los pesos cambian durante el entrenamiento, las distribuciones de activaciones se desplazan, y las capas dejan de recibir valores en rangos controlados.
- Causa varios problemas que afectan el aprendizaje.

Solución

Ayuda a regularizar

- Normalización por lotes

Normalización por lotes

Es una técnica muy usada en redes neuronales profundas porque **resuelve varios problemas importantes durante el entrenamiento**, especialmente relacionados con la **estabilidad y la velocidad del aprendizaje**.

Problema	Cómo afecta	Cómo lo resuelve BN
Sobreajuste	Las redes profundas con muchos parámetros pueden memorizar el conjunto de entrenamiento.	BN introduce ruido estocástico (por la media/varianza del batch), lo que tiene un efecto de **regularización ligera** que reduce el sobreajuste.
Gradientes inestables (explosión/desvanecimiento)	Los gradientes se vuelven muy grandes o muy pequeños cuando los valores intermedios crecen mucho o se contraen.	Al limitar la escala de las activaciones, BN mantiene gradientes en rangos manejables.
Desplazamiento de covariables internas (*internal covariate shift*)	Los datos que recibe cada capa cambian de distribución a medida que se actualizan los pesos de las capas anteriores, lo que ralentiza el aprendizaje. Porque las capas se están adaptando a entradas que no dejan de moverse.	BN mantiene la distribución de las activaciones intermedias más estable (media≈0, varianza≈1) y así acelera la convergencia.
Entrenamiento lento	Los optimizadores deben usar tasas de aprendizaje pequeñas para evitar inestabilidades.	BN permite usar tasas de aprendizaje más altas sin divergir, acelerando el entrenamiento.

Entrenando usando normalización por lotes

- La solución es sorprendentemente simple.
- Durante el entrenamiento, una capa de normalización por lotes **calcula el promedio y la desviación estándar de cada uno de sus canales de entrada sobre el lote y normaliza** al restar el promedio y dividiendo por la desviación estándar.
- Entonces **hay dos parámetros aprendidos para cada canal**, la escala (gama) y el shift (beta).
- La salida es simplemente la entrada normalizada, escalada por gama y desplazada por beta.

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Aplicado a la activación *x* sobre el mini-batch.

Nota: Podemos agregar capas de normalización por lotes después de capas densas o convolucionales para normalizar la salida.

Dropout

Todo algoritmo de ML que se considere exitoso debe de ser capaz de generalizar a datos no vistos, en vez de solo recordar los datos de entrenamiento.

Overfitting

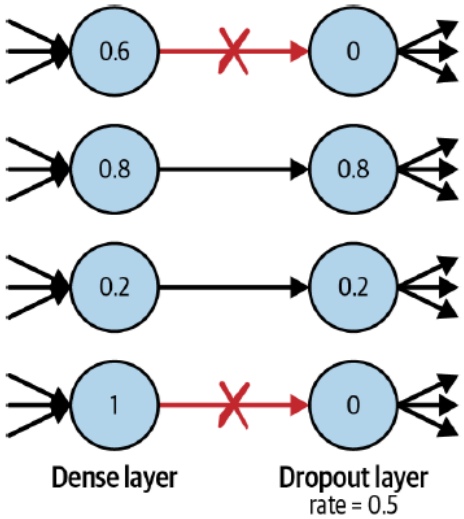
Cuando un algoritmo de ML se desempeña bien con los datos de entrenamiento, pero no con los de prueba.

Técnicas de regularización

Se aseguran de penalizar el modelo cuando comienza a sobreajustar.

Capas de Dropout

Durante el entrenamiento, cada capa de dropout elige aleatoriamente un conjunto de unidades de la capa que le precede y coloca sus salidas a 0.



La red ha sido entrenada para producir predicciones precisas incluso bajo condiciones no familiares, como aquellas causadas por el *dropping* de unidades aleatorias.
Solo sucede en la etapa de entrenamiento.

```
from tensorflow.keras import layers
layers.Dropout(rate = 0.25)
```

Proporción de unidades a tirar de la capa anterior.

Se usan comúnmente después de capas densas, aunque también las puedes usar después de las capas convolucionales.

```
Epoch 20/25
1563/1563 ━━━━ accuracy: 0.8173 - loss: 0.5246
Epoch 21/25
1563/1563 ━━━━ accuracy: 0.8136 - loss: 0.5261
Epoch 22/25
1563/1563 ━━━━ accuracy: 0.8198 - loss: 0.5136
Epoch 23/25
1563/1563 ━━━━ accuracy: 0.8250 - loss: 0.4939
Epoch 24/25
1563/1563 ━━━━ accuracy: 0.8220 - loss: 0.5044
Epoch 25/25
1563/1563 ━━━━ accuracy: 0.8329 - loss: 0.4760
```

Test: accuracy: 0.8011 - loss: 0.6071

cnn_cifar10_version4_aumento_datos.ipynb

Ya no hay sobreajuste

cnn_cifar10_version5_norm_drop.ipynb

```
Epoch 20/25
1563/1563 ━━━━ accuracy: 0.8173 - loss: 0.5246
Epoch 21/25
1563/1563 ━━━━ accuracy: 0.8136 - loss: 0.5261
Epoch 22/25
1563/1563 ━━━━ accuracy: 0.8198 - loss: 0.5136
Epoch 23/25
1563/1563 ━━━━ accuracy: 0.8250 - loss: 0.4939
Epoch 24/25
1563/1563 ━━━━ accuracy: 0.8220 - loss: 0.5044
Epoch 25/25
1563/1563 ━━━━ accuracy: 0.8329 - loss: 0.4760

Test:          accuracy: 0.8011 - loss: 0.6071
```

cnn_cifar10_version4_aumento_datos.ipynb

Ya no hay sobreajuste

```python
# Aumento de datos (se activa solo en training)
aug = tf.keras.Sequential([
    layers.RandomFlip("horizontal"),
    layers.RandomTranslation(0.1, 0.1),
], name="data_augmentation")

input_layer = layers.Input((32, 32, 3))
x = aug(input_layer)
x = layers.Conv2D(filters=32, kernel_size=(3,3), strides=1, padding="same")(x)
x = layers.BatchNormalization()(x)
x = layers.ReLU()(x)
x = layers.Conv2D(filters=32, kernel_size=(3,3), strides=2, padding="same")(x)
x = layers.BatchNormalization()(x)
x = layers.ReLU()(x)
x = layers.Conv2D(filters=64, kernel_size=(3,3), strides=1, padding="same")(x)
x = layers.BatchNormalization()(x)
x = layers.ReLU()(x)
x = layers.Conv2D(filters=64, kernel_size=(3,3), strides=2, padding="same")(x)
x = layers.BatchNormalization()(x)
x = layers.ReLU()(x)
x = layers.Flatten()(x)
x = layers.Dense(128)(x)
x = layers.BatchNormalization()(x)
x = layers.ReLU()(x)
x = layers.Dropout(rate = 0.5)(x)
output_layer = layers.Dense(NUM_CLASSES, activation="softmax")(x)

model = models.Model(input_layer, output_layer)
```

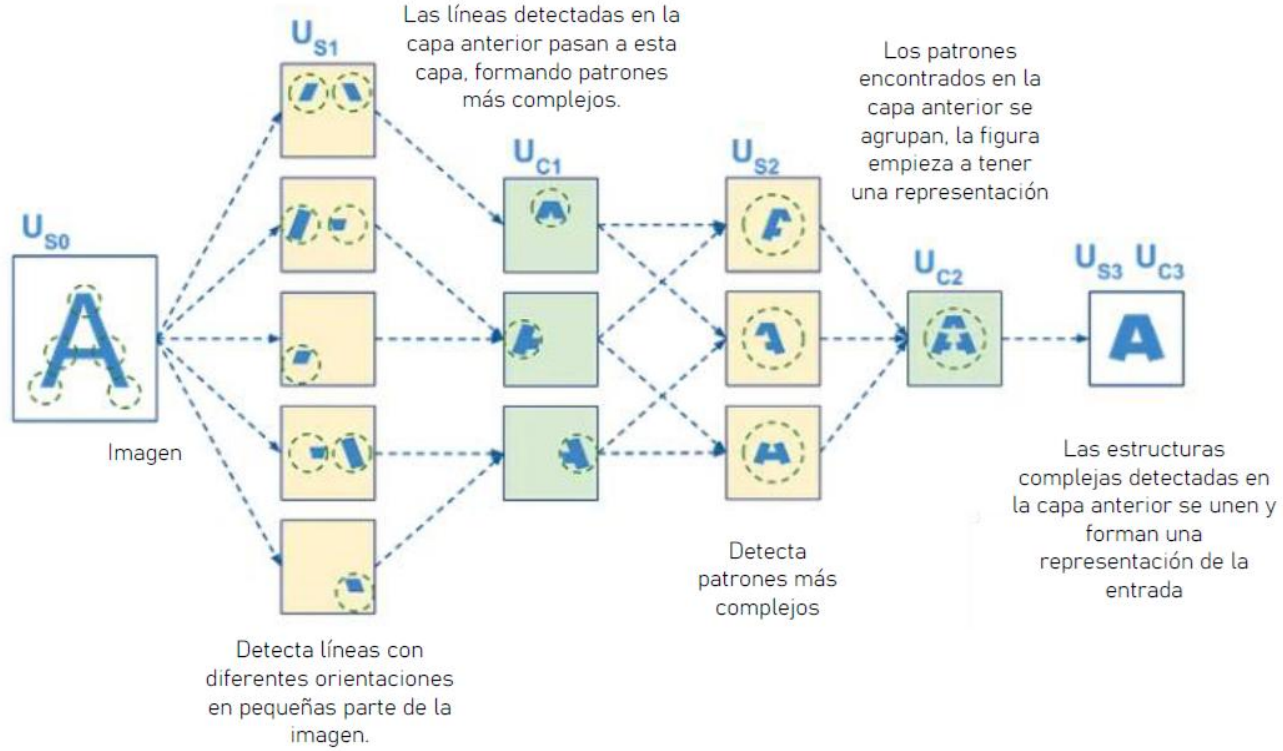
cnn_cifar10_version5_norm_drop.ipynb


```
Epoch 20/25
1563/1563 ━━━━ accuracy: 0.7806 - loss: 0.6370
Epoch 21/25
1563/1563 ━━━━ accuracy: 0.7849 - loss: 0.6224
Epoch 22/25
1563/1563 ━━━━ accuracy: 0.7853 - loss: 0.6218
Epoch 23/25
1563/1563 ━━━━ accuracy: 0.7892 - loss: 0.6121
Epoch 24/25
1563/1563 ━━━━ accuracy: 0.7921 - loss: 0.6057
Epoch 25/25
1563/1563 ━━━━ accuracy: 0.7942 - loss: 0.6005

Test:          accuracy: 0.7942 - loss: 0.6171
```

Menos sobreajuste

Visualización de filtros en CNN: ¿Qué patrones aprenden?

- Las CNN aprenden filtros (kernels) que detectan patrones locales en las imágenes.
  — Ej.: bordes, texturas, esquinas en las primeras capas.
- Las representaciones se vuelven más abstractas en capas profundas.
  — De bordes → formas → partes de objetos → conceptos.
- Visualizar filtros y activaciones permite "ver por dentro" la red.
  — Nos ayuda a interpretar, depurar y entender su comportamiento.

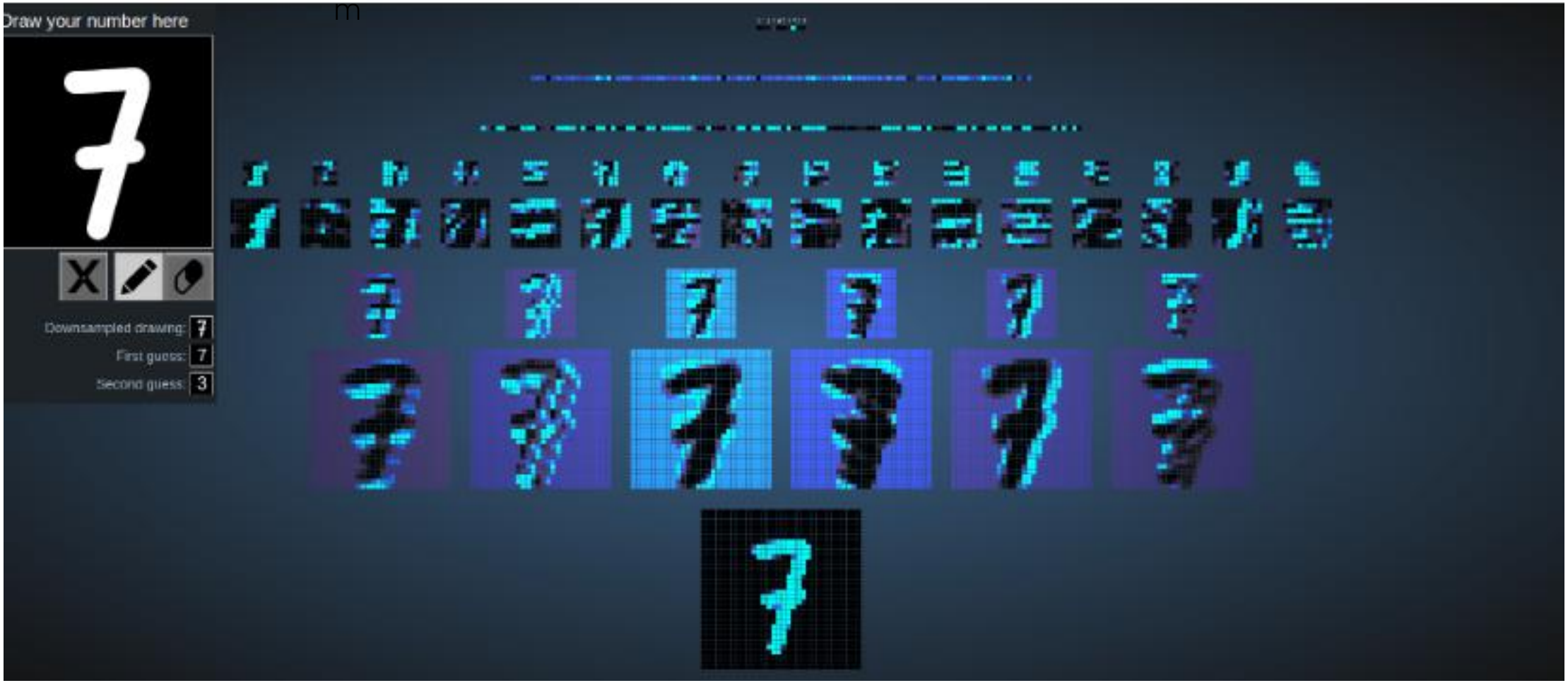


¿Qué patrones activan cada filtro y cómo se combinan para reconocer objetos?

Visualización de filtros en CNN: ¿Qué patrones aprenden?

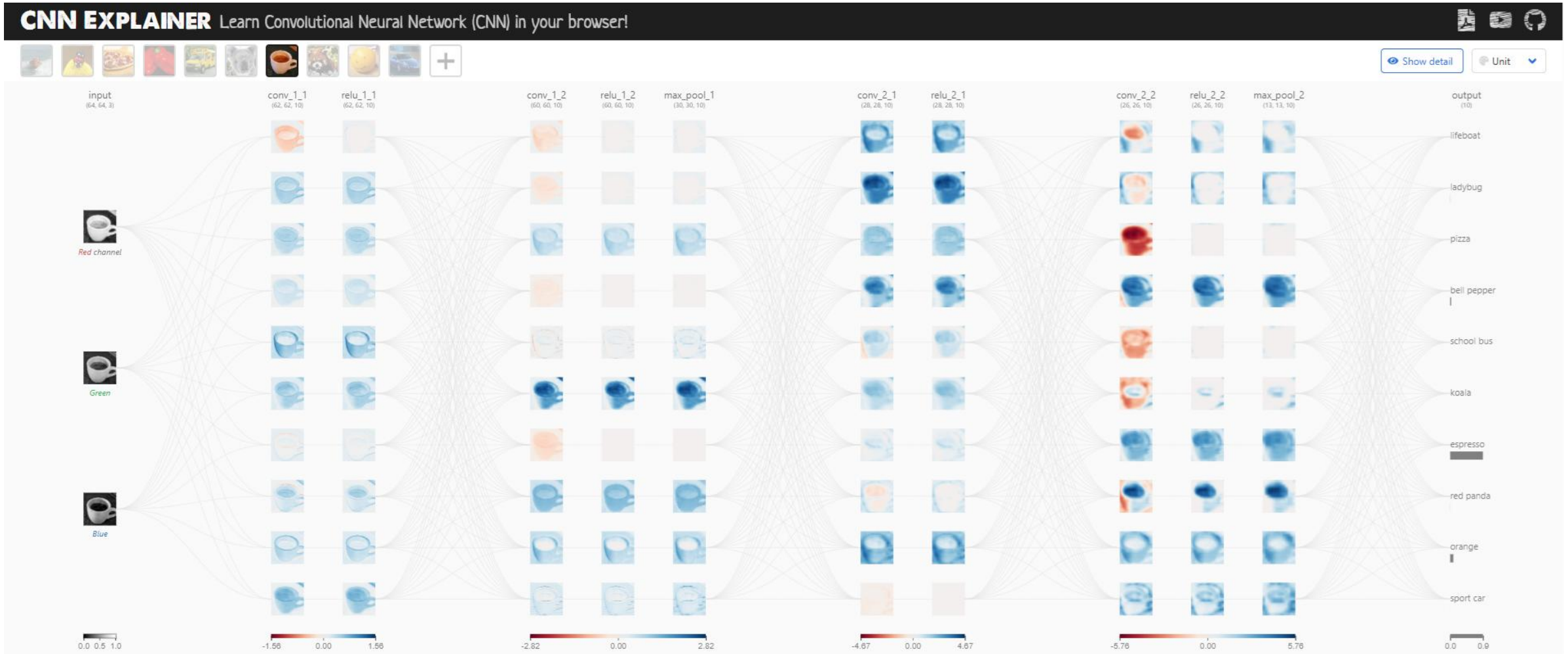
https://adamharley.com/nn_vis/cnn/3d.html

https://github.com/aharley/nn_vis?utm_source=chatgpt.com



Visualización de filtros en CNN: ¿Qué patrones aprenden?

https://poloclub.github.io/cnn-explainer/



Limitaciones de las CNNs hasta el 2012

- Desde finales de los 90, **las CNN (como LeNet-5)** funcionaban bien en MNIST, pero:
    - no escalaban a datasets grandes,
    - el cómputo era demasiado costoso,
    - y había **poco entusiasmo en la comunidad**, que prefería **métodos clásicos de visión** como SIFT, HOG, SVMs, random forests, etc.

•El término "deep learning" existía, pero **nadie usaba redes profundas en producción** porque **no podían entrenarse eficientemente** (problemas de gradientes y costo computacional).

Resultado:

Las CNN pasaron más de una década relegadas al margen de la visión por computadora.

AlexNet

(Krizhevsky, Sutskever & Hinton, 2012)

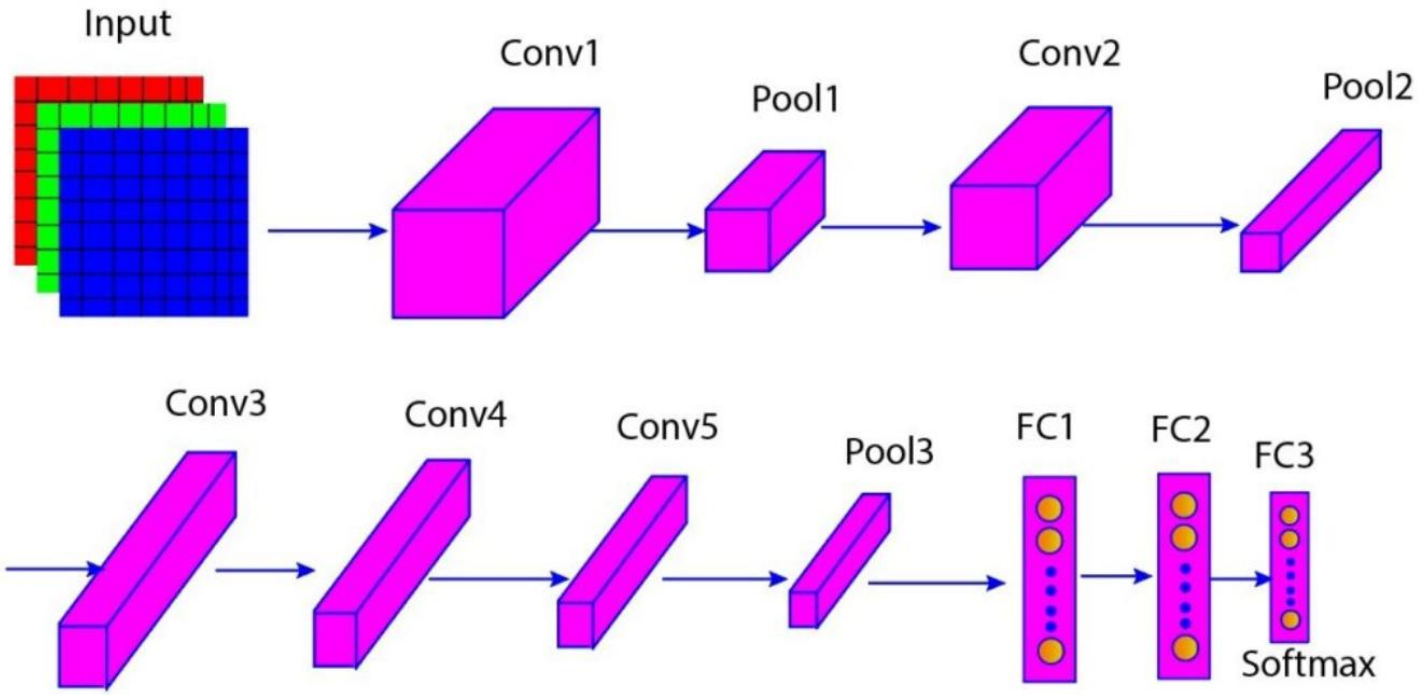
El renacimiento del Deep Learning en visión por computadora

- Entrenaron una CNN **mucho más grande y profunda**
  - 8 capas aprendibles: 5 convolucionales + 3 densas.
  - ≈ 60 millones de parámetros (gigantesca para su época).

- En el dataset **ImageNet** (1.2 millones de imágenes, 1000 clases).

Innovaciones clave que lo hicieron posible:

- Uso de GPUs (por primera vez a gran escala).
- ReLU → aceleró enormemente el entrenamiento.
- Dropout + data augmentation → mitigaron el sobreajuste.

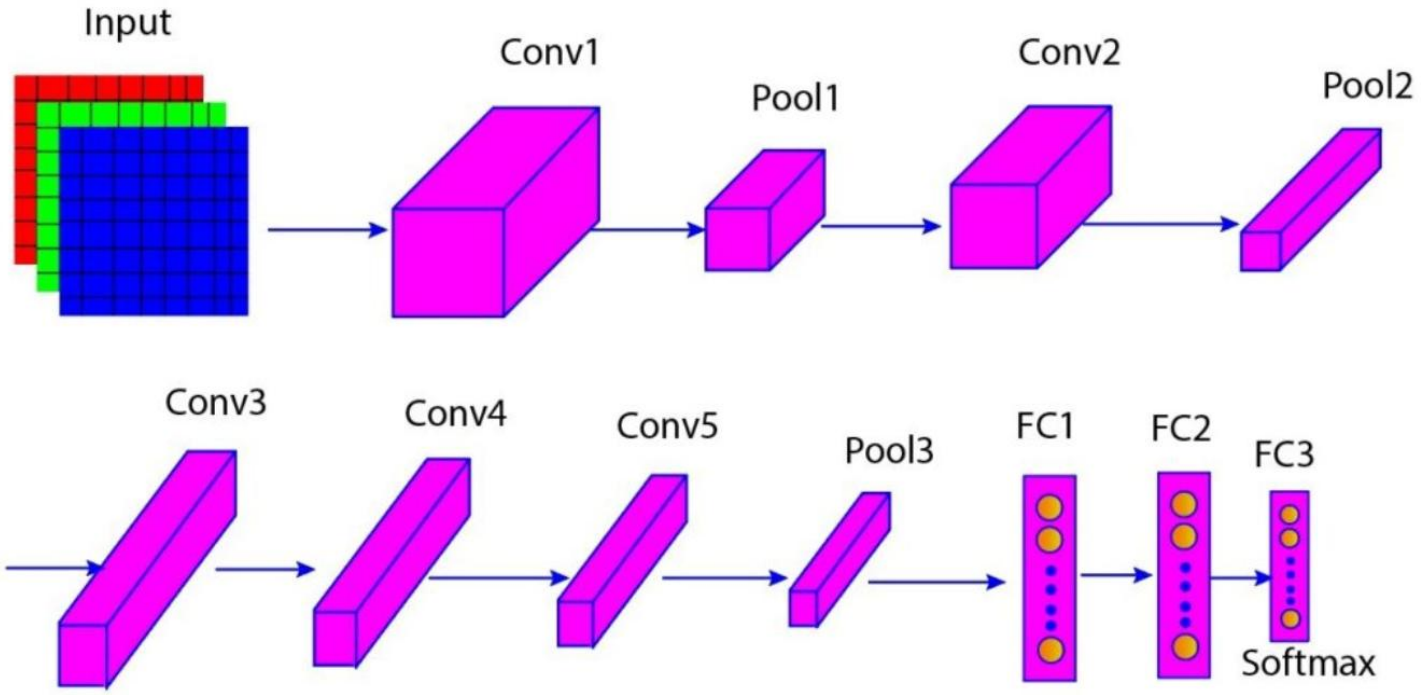


AlexNet

(Krizhevsky, Sutskever & Hinton, 2012)

El renacimiento del Deep Learning en visión por computadora

- Ganaron el concurso ImageNet 2012
  - Redujo el error top-5 de **26% → 15%**, superando por amplio margen a todos los competidores.
- Fue la **primera vez que un modelo de deep learning superó ampliamente a todos los métodos clásicos** en visión por computadora a gran escala.
- **Demostró que el Deep Learning podía escalar y ganar** en problemas reales de gran escala.



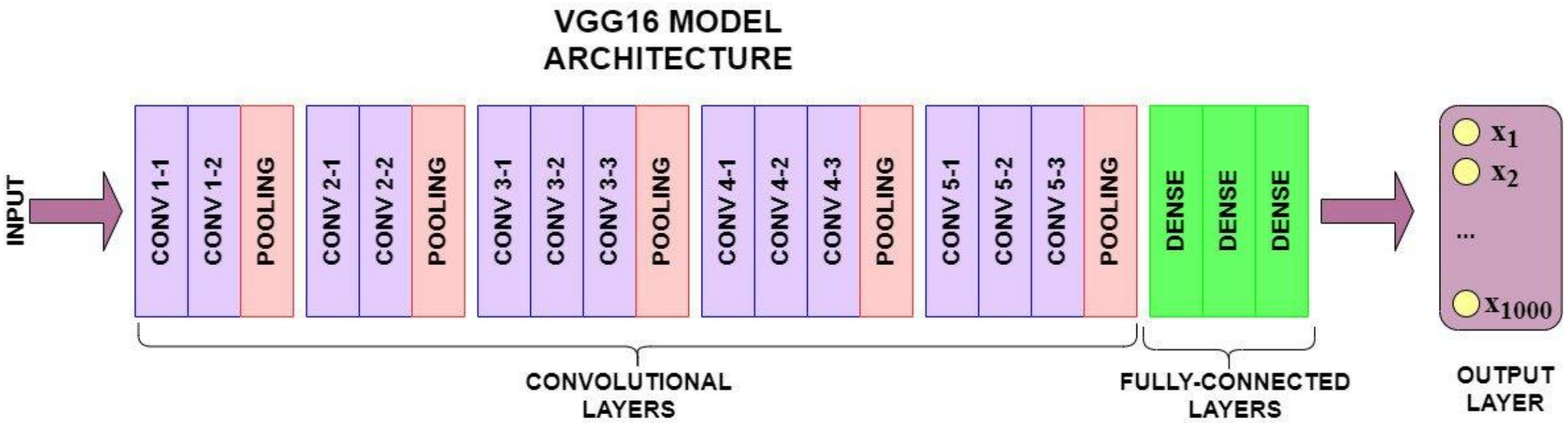
Impacto histórico

- Disparó el auge de redes profundas en visión por computadora.
- Inspiró directamente a todas las arquitecturas modernas.

VGGNet-16

(2014 — Simonyan & Zisserman)

Profundidad extrema con simplicidad estructural



Arquitectura muy uniforme y simple

- Entrada: imágenes de tamaño 224 x 224 a color.
- Solo usa **convoluciones 3×3** apiladas y **pooling 2×2**.
- Arquitectura tipo "bloques" repetidos: fácil de entender, implementar y extender.
- De 16 capas con pesos entrenables.

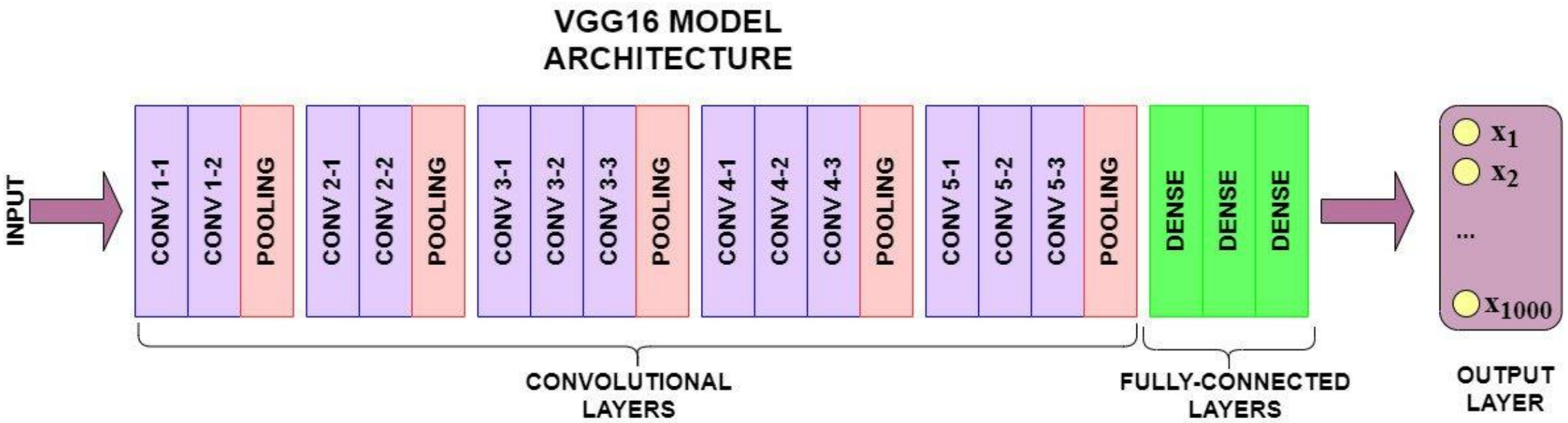
Segundo lugar en ImageNet 2014

Modelo	Año	Top-5 accuracy
AlexNet	2012	84.7 %
VGG-16	**2014**	**92.7 %**

VGGNet-16

(2014 — Simonyan & Zisserman)

Profundidad extrema con simplicidad estructural



Demostró que aumentar la profundidad mejora el desempeño

- Mientras haya suficientes datos y cómputo, **más capas → mejores resultados**.
- Cambió la percepción: profundidad ≠ sobreajuste si se entrena correctamente.

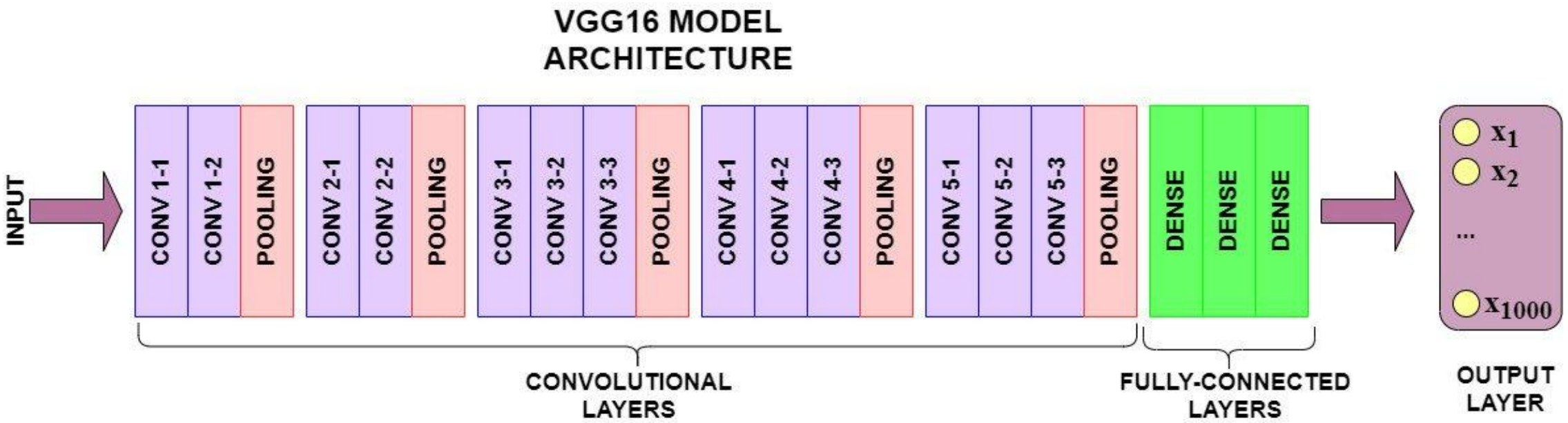
Alta reutilización en transfer learning

- Pesos preentrenados de VGG se usan ampliamente como **extractores de características** en visión por computadora.
- Base de muchos modelos modernos hasta hoy.

VGGNet-16

(2014 — Simonyan & Zisserman)

Profundidad extrema con simplicidad estructural



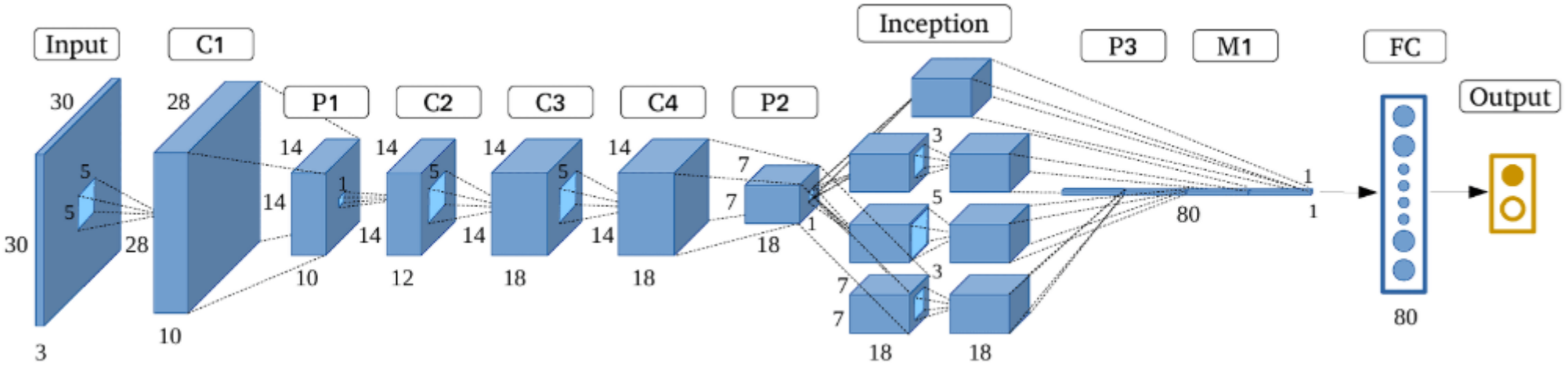
Limitación

- ≈ 138 millones de parámetros → **muy costosa en memoria y tiempo de entrenamiento**.

GoogLeNet / Inception

(Szegedy et al., 2014)

Más profundidad con eficiencia computacional



Primer lugar en ImageNet 2014

Modelo	Año	Top-5 accuracy
AlexNet	2012	84.7 %
VGG-16	**2014**	**92.7 %**
GoogLeNet (Inception)	2014	93.3 %

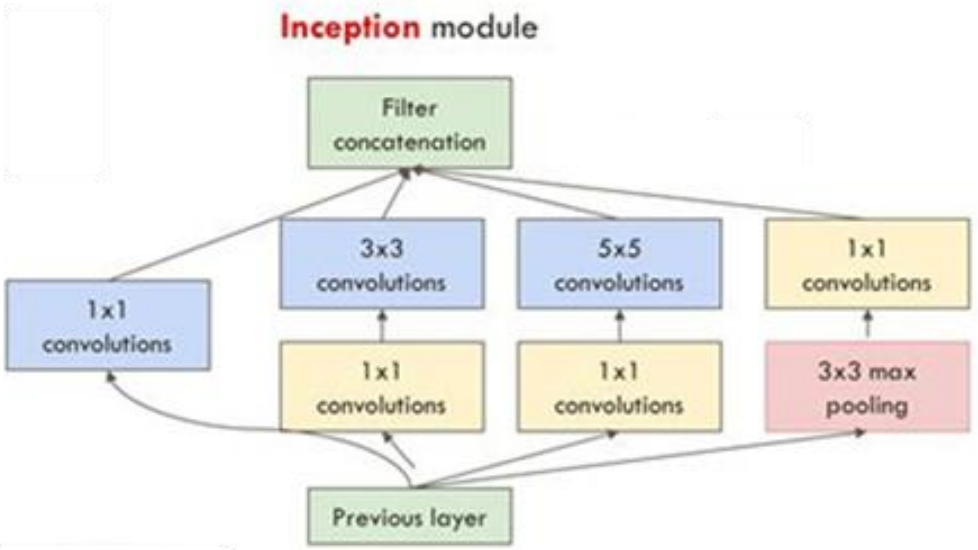
Demostró que **no solo importa la profundidad, sino también la eficiencia.**

- ≈ 4 millones de parámetros (vs ≈138 millones en VGG-16).
- Profundidad efectiva ≈ 22 capas, pero con mucho menor coste computacional.

GoogLeNet / Inception

(Szegedy et al., 2014)

Más profundidad con eficiencia computacional



- Un bloque Inception **procesa la misma entrada con filtros de varios tamaños en paralelo** y concatena sus salidas, lo que permite a la red **capturar patrones a múltiples escalas con pocos parámetros.**

Ventajas

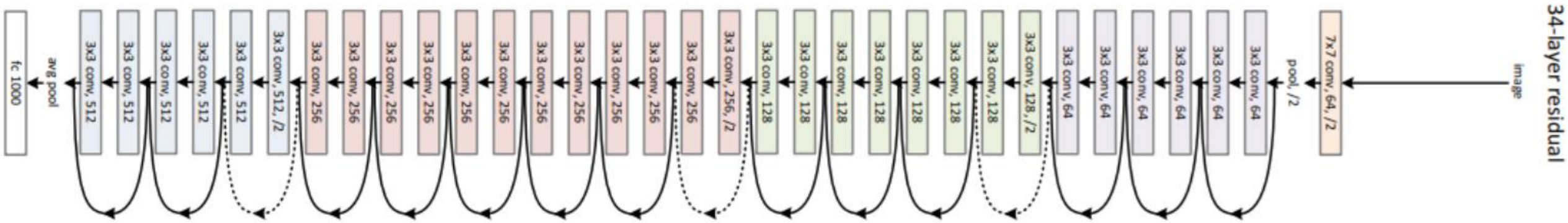
- Multiescala:
  Capta patrones pequeños y grandes al mismo tiempo.
- Eficiencia:
  Las conv 1×1 reducen canales → bajan mucho los parámetros.
- Profundidad efectiva:
  Permiten redes más profundas sin sobrecoste de memoria.

Fue adoptada extensamente en aplicaciones industriales de visión.

ResNet

(He et al., 2015)

Redes ultra profundas con conexiones residuales



Arquitectura modular

- Formada por **bloques residuales básicos o "bottleneck"** apilados.
- Permite construir versiones de **18, 34, 50, 101, 152 capas**.

Gran innovación: conexiones residuales (skip connections)

- Cada bloque residual aprende solo la "diferencia" (residuo) entre la entrada y la salida deseada.
- Añade un atajo (shortcut) que salta una o varias capas y suma la entrada a la salida.

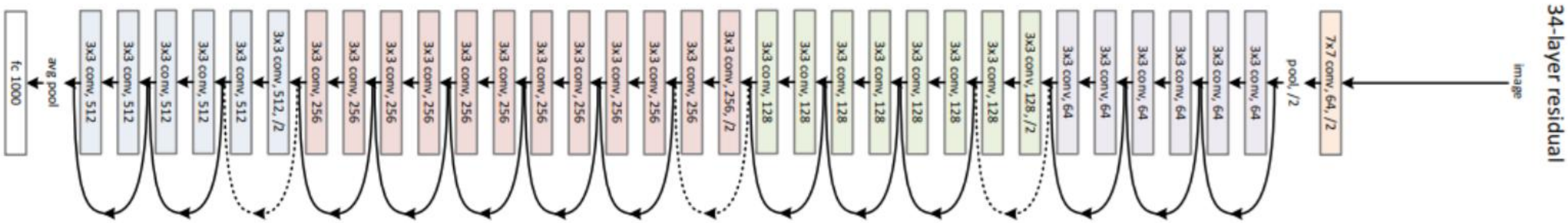
Solución al problema del desvanecimiento del gradiente:

- Antes, al aumentar las capas, **las redes dejaban de entrenar correctamente**.
- Las conexiones residuales **facilitan el flujo del gradiente hacia capas muy profundas**.

ResNet

(He et al., 2015)

Redes ultra profundas con conexiones residuales



- Ganadora de ImageNet 2015

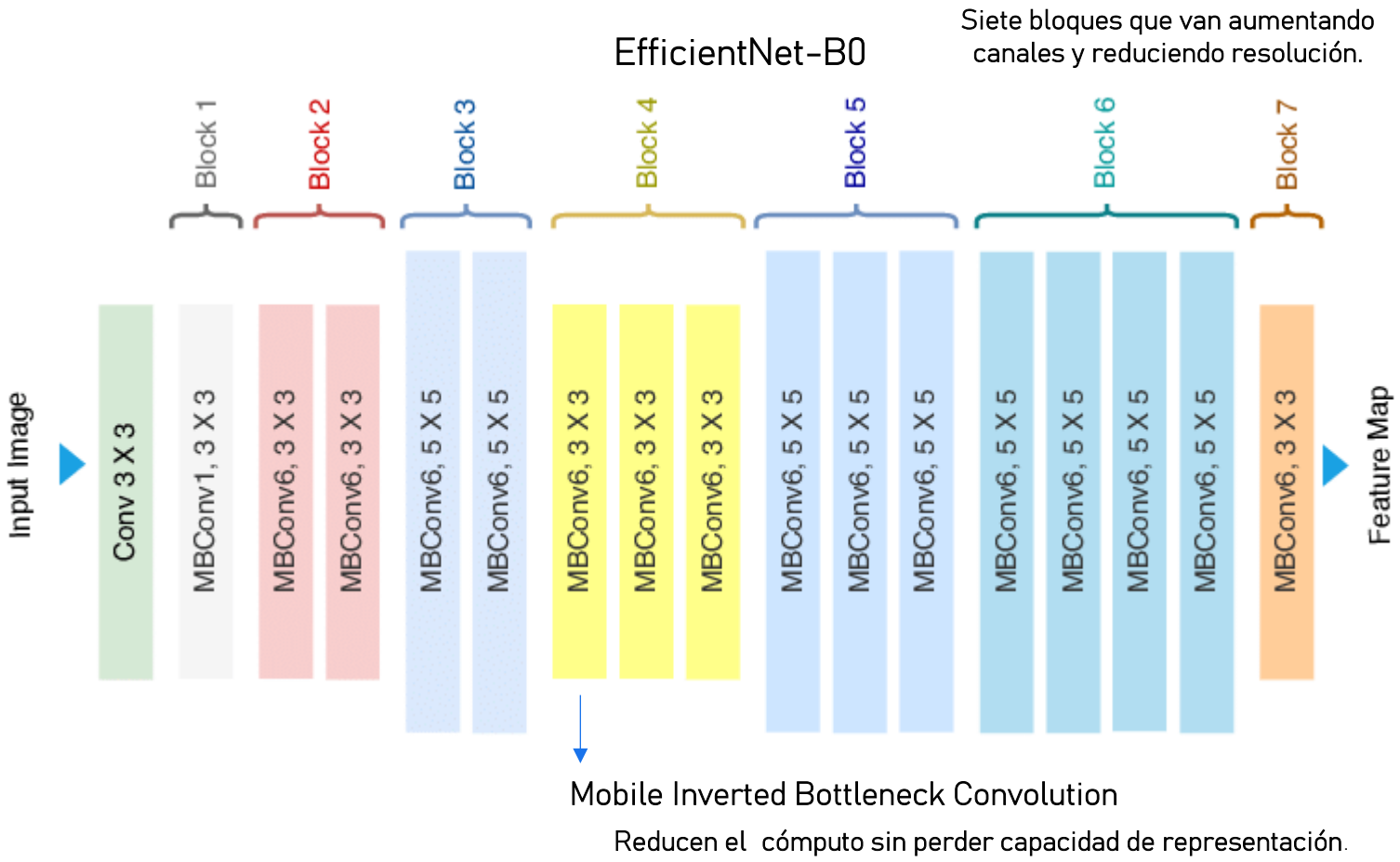
Modelo	Año	Top-5 accuracy
AlexNet	2012	84.7 %
VGG-16	**2014**	**92.7 %**
GoogLeNet (Inception)	2014	93.3 %
ResNet	2015	96.4 %

EfficientNet

(Tan & Le, 2019)

Demostró que no basta con hacer redes más profundas: escalar profundidad, ancho y resolución de forma conjunta y balanceada permite lograr más precisión con menos recursos.

- EfficientNet no nació como una red gigante:
  - Primero los autores buscaron **una arquitectura base óptima pero muy pequeña** a la que luego pudieran **escalar sistemáticamente**.
  - Esa red inicial es **EfficientNet-B0**, y fue encontrada mediante **Neural Architecture Search (NAS)**
    - **algoritmo que explora automáticamente el espacio de arquitecturas posibles**, entrenando y evaluando muchas redes candidatas y eligiendo la mejor.



EfficientNet

(Tan & Le, 2019)

Demostró que no basta con hacer redes más profundas: escalar profundidad, ancho y resolución de forma conjunta y balanceada permite lograr más precisión con menos recursos.

- Escalamiento compuesto
  - En redes anteriores (VGG, ResNet, Inception), cuando se querían mejorar resultados se hacía:
    - Más **profundas** (más capas), o
    - Más **anchas** (más filtros), o
    - Con **imágenes más grandes** (más resolución de entrada).
  - Pero esto se hacía de forma **manual y desequilibrada**, lo que no siempre mejoraba los resultados.

  - EfficientNet propone **escalar las tres dimensiones a la vez de forma balanceada**,

    Usa constantes α, β, γ y un factor ϕ:

    depth $= \alpha^{\phi}$, $width = \beta^{\phi}$, $resolution = \gamma^{\phi}$

    Cada incremento de $\phi \approx$ **duplica el costo computacional** de forma controlada.

Dimensión	Qué significa	Qué aporta	Limitaciones si se usa sola
Profundidad	Más **capas** secuenciales (conv, BN, ReLU, etc.)	Capta **patrones jerárquicos** y de **alto nivel** (formas completas, semántica)	Difícil de entrenar (gradientes inestables), sobreajuste si no hay muchos datos
Anchura	Más **filtros/canales** por capa	Capta más **patrones locales y variados** en paralelo (texturas, bordes, colores)	Más costo de memoria, no escala jerarquía por sí sola
Resolución	Imágenes de **mayor tamaño de entrada**	Permite **ver detalles más finos** (pequeños objetos, bordes nítidos)	Aumenta mucho el cómputo si no se acompaña de más capacidad en la red

Si solo aumentas **una** de estas dimensiones, obtienes ganancias limitadas:

- Solo más profundo → aprende jerarquías, pero se entrena mal.
- Solo más ancho → aprende más detalles locales, pero no abstracción.
- Solo más resolución → ve más detalle, pero no sabe procesarlo.

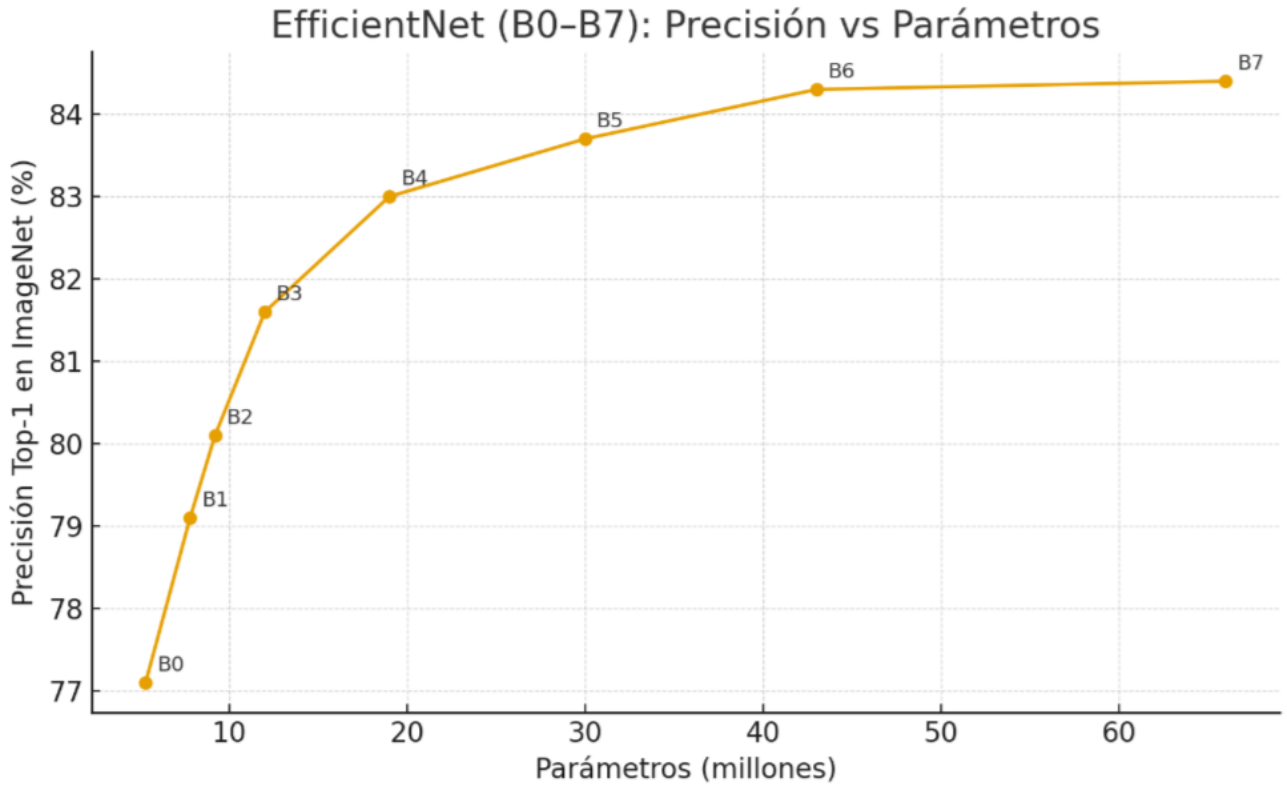
EfficientNet

(Tan & Le, 2019)

Demostró que no basta con hacer redes más profundas: escalar profundidad, ancho y resolución de forma conjunta y balanceada permite lograr más precisión con menos recursos.

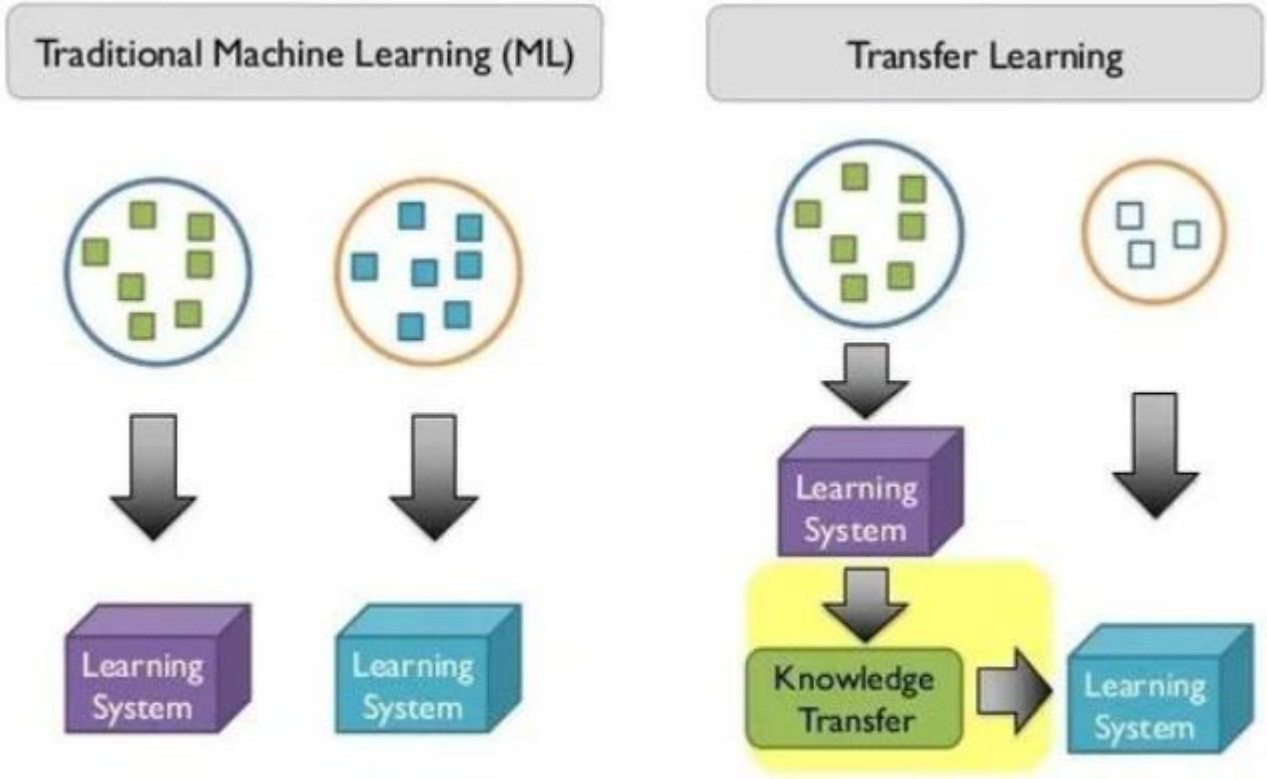
- Crearon la familia EfficientNet (B0-B7)
  - B0 → red pequeña, rápida, 224×224.
  - B7 → red grande, muy precisa, 600×600.

Todas tienen la **misma arquitectura base**, solo **más capas, más filtros y mayor resolución de entrada**.



Transfer Learning
con redes convolucionales pre-entrenadas

- Técnica que permite usar una red grande ya entrenada (ej. en ImageNet) como punto de partida para resolver otra tarea diferente.
- Aprovecha que las primeras capas aprenden características genéricas (bordes, texturas, formas), útiles para muchos problemas de visión.
- En lugar de entrenar una CNN grande desde cero, aprovechamos lo que ya aprendió en un *dataset* grande (*ImageNet*) y solo adaptamos su últimas capas a nuestro nuevo problema.



transfer_learning_ResNet50_cats_vs_dogs.ipynb

Google colab

Dejarlo ejecutando

Transfer Learning
con redes convolucionales pre-entrenadas

¿Cómo se hace?

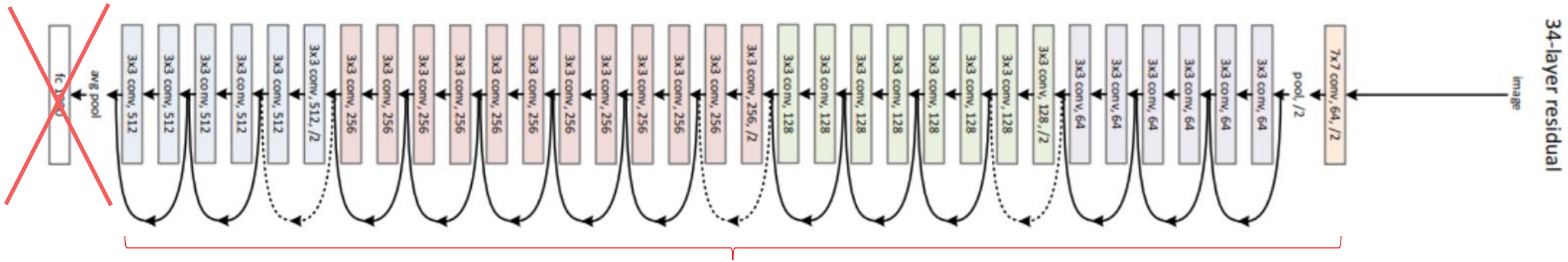
1. Seleccionar el modelo base preentrenado
   - Ej. ResNet50, VGG16, EfficientNet con pesos de ImageNet.
   - Este modelo aportará representaciones visuales ya aprendidas (bordes, texturas, formas, etc.).
   - Se descarga sin su capa final de clasificación (solo la parte de extracción de características o base convolucional).

```
base_model = tf.keras.applications.ResNet50(
    include_top=False,
    weights="imagenet",
    input_shape=(IMG_SIZE, IMG_SIZE, 3)
)
```

None → inicializa pesos aleatorios (entrenar desde cero).

VGG16, VGG19, ResNet101, ResNet152 EfficientNetB0 … EfficientNetB7

≥ 32×32
Recomendado: 224×224



Extracción de características

Transfer Learning
con redes convolucionales pre-entrenadas

transfer_learning_ResNet50_cats_vs_dogs.ipynb

Google colab

Dejarlo ejecutando

2. Congelar el modelo base preentrenado
   - Se congela la base convolucional para que sus pesos **no se actualicen durante el entrenamiento inicial**.

```
base_model.trainable = False
```

3. Se construye el modelo completo
   - Incluye una nueva cabeza de clasificación encima (capa global pooling + densa), que será la que se entrenará con el nuevo dataset.

```
inputs = layers.Input(shape=(IMG_SIZE, IMG_SIZE, 3))
x = data_augmentation(inputs)
x = tf.keras.applications.resnet50.preprocess_input(x * 255.0)
x = base_model(x, training=False)
x = layers.GlobalAveragePooling2D()(x)
x = layers.Dropout(0.4)(x)
outputs = layers.Dense(1, activation="sigmoid")(x)

model = models.Model(inputs, outputs)
```

Fuerza al modelo base a comportarse en **modo inferencia** (BatchNorm no actualice estadísticas de media e invarianza y desactiva dropout)

Reduce cada mapa de características 2D a un solo valor (su promedio)
shape = (batch=32, height=5, width=5, channels=2048)
x = layers.GlobalAveragePooling2D()(x)
shape = (32, 2048)

Reduce drásticamente el número de parámetros comparado con aplanar (`Flatten`).
Evita sobreajuste y mejora la generalización.

transfer_learning_ResNet50_cats_vs_dogs.ipynb

Google colab

Dejarlo ejecutando

Transfer Learning

con redes convolucionales pre-entrenadas

4. Entrenamiento de la cabeza
   - Aprende a clasificar con las características existentes.
   - Solo los pesos de las capas nuevas se actualizan.

```
model.compile(
    optimizer=optimizers.Adam(learning_rate=LEARNING_RATE_FE),
    loss="binary_crossentropy",
    metrics=["accuracy"]
)

history_fe = model.fit(
    ds_train,
    validation_data=ds_val,
    epochs=EPOCHS_FE
)
```

5. Descongelar parte de la base (fine-tunning)
   - Una vez que la cabeza ya aprendió, **se descongela parte o toda la base convolucional**.
   - Esto permite ajustar finamente los pesos del modelo pre-entrenado a tu dominio específico.

```
FINE_TUNE_AT = 100
```

→ Número de capas a congelar en el fine-tunning.

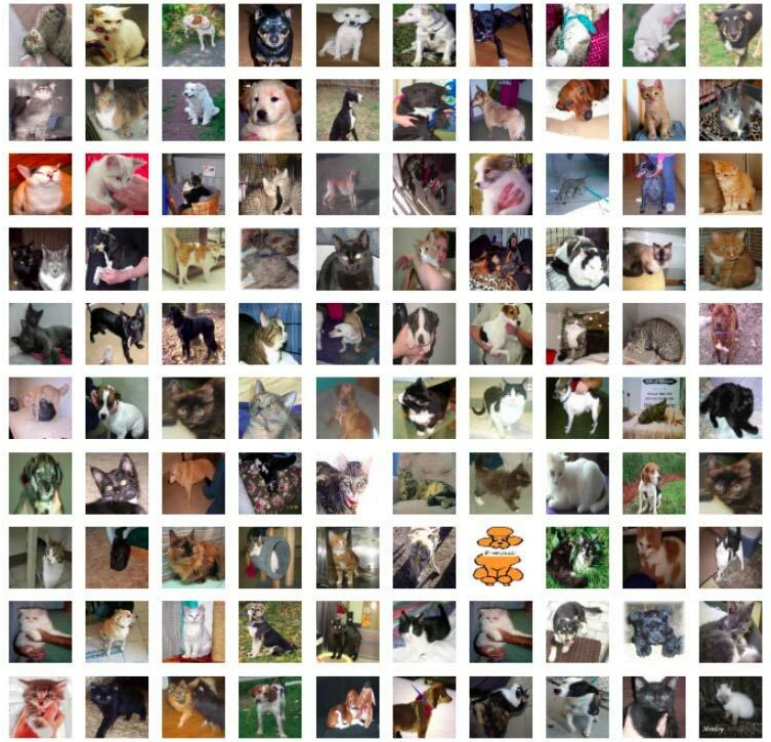
```
base_model.trainable = True
for layer in base_model.layers[:FINE_TUNE_AT]:
    layer.trainable = False

model.compile(
    optimizer=optimizers.Adam(learning_rate=LEARNING_RATE_FT),
    loss="binary_crossentropy",
    metrics=["accuracy"]
)
```

Transfer Learning

En dataset Cats vs Dogs

¿Por qué usar *Transfer Learning* en Cats vs Dogs?

En lugar de entrenar una CNN desde cero



Dataset relativamente pequeño y simple

- Cats vs Dogs tiene ≈25 000 imágenes (pocas para entrenar una CNN grande desde cero).
- Entrenar redes profundas requiere **muchos más datos para no sobreajustar**.

Las redes grandes ya "saben ver"

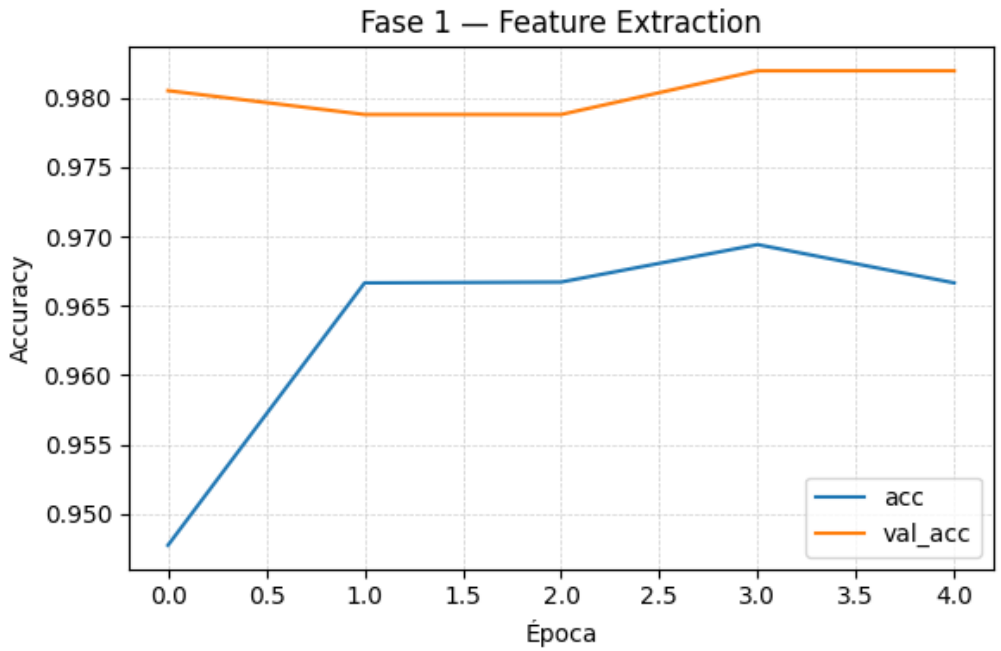
- Modelos preentrenados (ImageNet) ya aprendieron bordes, texturas, formas, colores.
- Estas características genéricas son reutilizables en casi cualquier tarea de visión.

Mucho más rápido y eficiente

- Solo se entrena un pequeño *head* final al inicio y opcionalmente se realiza un fine-tunning.
- Reduce drásticamente el tiempo y el costo computacional.

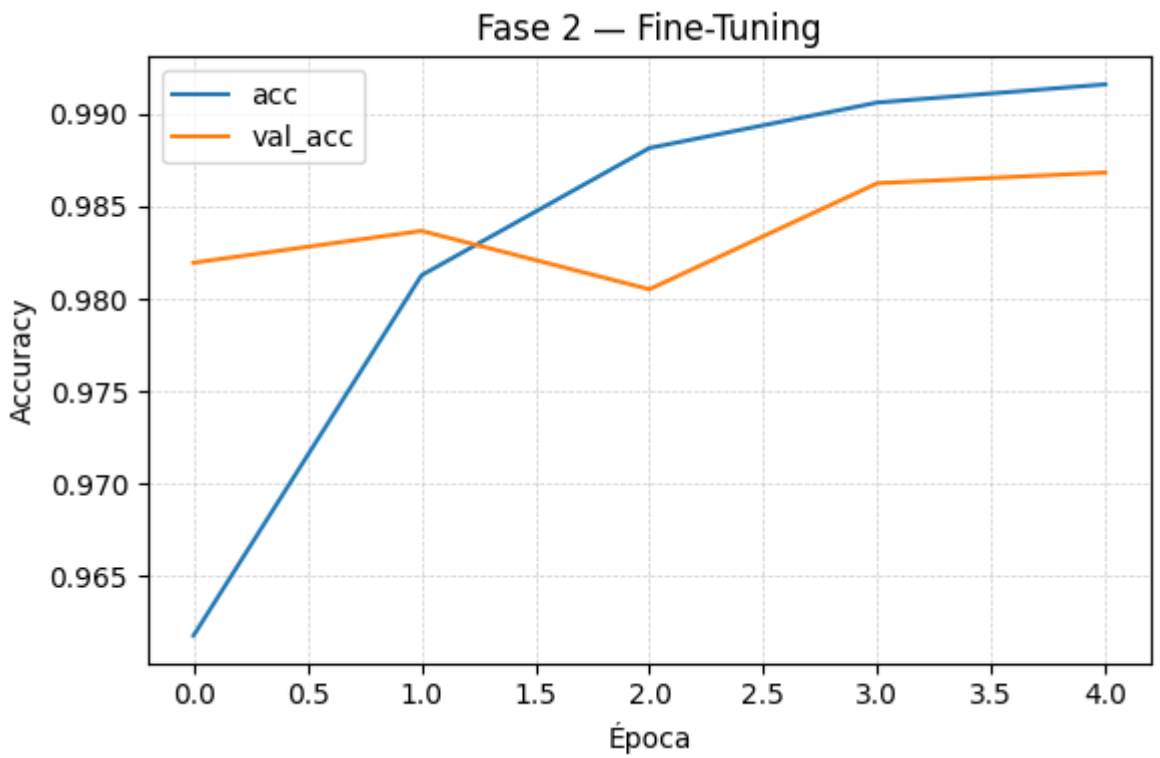
Entrenamiento Fase 1 — *Feature Extraction* (solo el head)

```
Epoch 1/5
509/509 ━━━━━━━━━━━━━━━━━━━━ 72s 101ms/step - accuracy: 0.9197 - loss: 0.1912 - val_accuracy: 0.9805 - val_loss: 0.0554
Epoch 2/5
509/509 ━━━━━━━━━━━━━━━━━━━━ 55s 91ms/step - accuracy: 0.9660 - loss: 0.0935 - val_accuracy: 0.9788 - val_loss: 0.0653
Epoch 3/5
509/509 ━━━━━━━━━━━━━━━━━━━━ 57s 94ms/step - accuracy: 0.9684 - loss: 0.0829 - val_accuracy: 0.9788 - val_loss: 0.0568
Epoch 4/5
509/509 ━━━━━━━━━━━━━━━━━━━━ 57s 94ms/step - accuracy: 0.9686 - loss: 0.0857 - val_accuracy: 0.9819 - val_loss: 0.0577
Epoch 5/5
509/509 ━━━━━━━━━━━━━━━━━━━━ 55s 92ms/step - accuracy: 0.9676 - loss: 0.0860 - val_accuracy: 0.9819 - val_loss: 0.0570
```

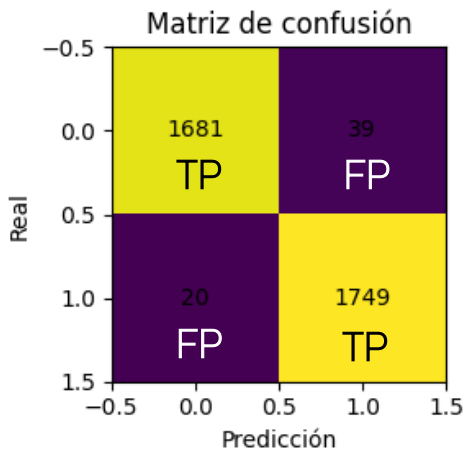


Entrenamiento Fase 2 — *Fine-Tuning* (descongelar parte de la base)

```
Epoch 1/5
509/509 ━━━━━━━━━━━━━━━━━━━━ 120s 176ms/step - accuracy: 0.9533 - loss: 0.1228 - val_accuracy: 0.9819 - val_loss: 0.0589
Epoch 2/5
509/509 ━━━━━━━━━━━━━━━━━━━━ 96s 169ms/step - accuracy: 0.9801 - loss: 0.0480 - val_accuracy: 0.9837 - val_loss: 0.0513
Epoch 3/5
509/509 ━━━━━━━━━━━━━━━━━━━━ 95s 168ms/step - accuracy: 0.9886 - loss: 0.0322 - val_accuracy: 0.9805 - val_loss: 0.0552
Epoch 4/5
509/509 ━━━━━━━━━━━━━━━━━━━━ 95s 169ms/step - accuracy: 0.9897 - loss: 0.0269 - val_accuracy: 0.9862 - val_loss: 0.0457
Epoch 5/5
509/509 ━━━━━━━━━━━━━━━━━━━━ 98s 175ms/step - accuracy: 0.9911 - loss: 0.0213 - val_accuracy: 0.9868 - val_loss: 0.0408
```




```
Test accuracy: 0.9831
Confusion matrix:\n [[1681   39]
 [  20 1749]]
```



	Predicho	
	cat	dog
Real cat	[1681	39]
Real dog	[20	1749]

Fin de la era CNN

El surgimiento de los Vision Transformers

- EfficientNet Marcó en muchos sentidos el **último gran hito "clásico" de las CNNs puras.**
- Después de ella el foco de la comunidad empezó a migrar hacia los **Vision Transformers (2020)**.
  - Usan atención auto-regresiva en lugar de convoluciones.
- Desde 2021 en adelante, casi todos los nuevos state-of-the-art en visión usan:
  - ViTs puros
  - o híbridos CNN + atención.

Época	Arquitecturas clave	Aportes principales
🧩 **2012–2014**	AlexNet → VGG	Inician la era del *deep learning* en visión; redes más profundas y grandes
⚡ **2015–2017**	ResNet → DenseNet	Redes muy profundas gracias a conexiones residuales y densas
📱 **2018–2019**	MobileNetV2 → EfficientNet	Redes ligeras y eficientes; escalamiento balanceado (depth, width, resolution)
⚙️ **2021–2022**	EfficientNetV2 → ConvNeXt	Últimas CNN competitivas con ViTs; diseño simplificado y moderno
🧠 **2020–hoy**	ViT → DeiT → Swin → BEiT...	Reemplazan convoluciones por atención; dominan benchmarks grandes

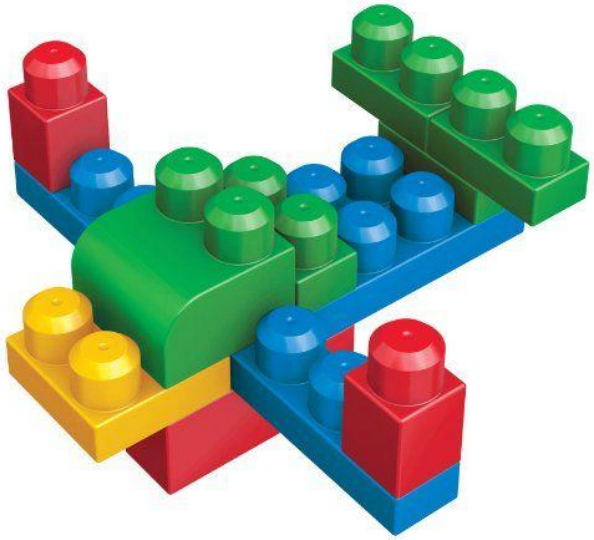
Temario

1. Introducción.
2. Matemáticas esenciales para el aprendizaje profundo
3. Fundamentos de las redes neuronales profundas
4. Autocodificadores variacionales
5. Redes generativas adversarias (GANs)
6. Modelos autoregresivos
7. Modelos de normalización de flujo
8. Modelos basados en energía
9. Modelos de difusión

Introdujimos los conceptos más básicos que necesitaremos para comenzar a construir modelos generativos profundos.

Conclusiones

- Las redes neuronales profundas son completamente flexibles por diseño.
- No hay realmente reglas fijas para sus arquitecturas.
- Existen guías y mejores prácticas.
- Pero, debes sentirte libre de experimentar con los tipos de capas y con el orden en el que aparecen.
- No te sientas restringido a usar solo las arquitecturas que conoces — **úsalas como punto de partida, no como límites.**.



Muchos avances vinieron justamente de romper el esquema secuencial clásico:

- **ResNet** introdujo *skip connections*.
- **Inception** usó ramas en paralelo.
- **Transformers** reemplazaron convoluciones y recurrencias por atención pura.

Todo esto nació de **experimentar más allá de las arquitecturas conocidas.**