5. ARQUITECTURAS CNN DE REFERENCIA

En el capítulo anterior se presentó una de las arquitecturas de aprendizaje profundo que ayudó a sentar las bases de las redes neuronales modernas: LeNet. En términos generales, esta arquitectura desarrollada en los años 90 del siglo pasado presenta características similares a las arquitecturas secuenciales propuestas en la década del 2010, época en la cual se revolucionó el mundo del aprendizaje profundo. La pregunta aquí sería, por qué si ya se habían desarrollado las bases de las CNN, fue dos décadas después que empezó su aplicación e implementación a gran escala.

La respuesta a esta pregunta se fundamentó en dos aspectos: capacidad de cómputo y cantidad de datos. En dos décadas, la capacidad de cómputo creció exponencialmente de acuerdo con la ley de Moore, y permitió no solo aumentar la velocidad en el procesamiento de los datos, sino también disponer de unidades de procesamiento gráfico (GPU) para procesar datos en paralelo de manera eficaz. Este aspecto facilita la implementación de modelos con mayor número de capas y mayor profundidad, mejorando a su vez el rendimiento de los algoritmos.

Para entrenar este tipo de modelos un aspecto clave es contar con una gran cantidad de datos etiquetados que permitan en el entrenamiento ajustar los parámetros de la arquitectura. Mientras que la arquitectura LeNet fue entrenada y validada con un dataset de 60000 ejemplos con 10 clases, arquitecturas recientes han contado con datasets más complejos y cercanos a datos reales como lo son Pascal o ImageNet. La construcción y uso de estos datasets se potencializó a través de desafíos que invitaban a la comunidad a usar estos datos y resolver problemas relacionados con clasificación y reconocimiento de objetos.

En este sentido, ImageNet es el dataset de referencia más utilizado para clasificación de imágenes y reconocimiento de objetos a gran escala. Este dataset cuenta con más de 14 millones de imágenes etiquetadas manualmente con 1.000 categorías de objetos. Su amplia difusión radica en que este fue el dataset utilizado en el Reto ImageNet de reconocimiento visual a gran escala (ImageNet Large Scale Visual Recognition Challenge, ILSVRC). El ILSVRC fue una competencia anual que evaluó el progreso de los algoritmos de visión por

computador en cuanto a clasificación de imágenes y detección de objetos. La primera versión de este reto se realizó por primera vez en 2010[25].

Este reto se dividió en dos tareas: clasificación de imágenes y detección de objetos. En la tarea de clasificación de imágenes, los algoritmos se evalúan en función de su capacidad para clasificar correctamente las imágenes en una de las 1.000 categorías de objetos. En la tarea de detección de objetos, los algoritmos se evalúan por su capacidad para identificar y localizar objetos en las imágenes. El ILSVRC ha sido decisivo para impulsar el progreso de la visión por ordenador. El reto ha contribuido a impulsar el desarrollo de nuevos algoritmos de aprendizaje automático y ha dado lugar a mejoras significativas en la precisión de la clasificación de imágenes y la detección de objetos.

Este tipo de retos facilitó la comparación de nuevos modelos propuestos desde la investigación, de tal forma que en el año 2012 una arquitectura de aprendizaje profundo conocida como AlexNet superó a los modelos tradicionales de aprendizaje de máquina en el ILSVRC. En este año, AlexNet ganó el desafío de clasificación de imágenes con una tasa de error del 26,2%, superando ampliamente el resultado del año anterior que era del 37,5%. En 2014, de nuevo se presenta una mejora importante con la arquitectura VGG que obtuvo una tasa de error del 16,4%, demostrando la competitividad de los modelos de aprendizaje profundo. Ya en 2016, el ganador fue ResNet con una tasa de error del 3,5 %, lo que supuso un rendimiento similar al de un experto humano. En términos generales, las principales arquitecturas que significaron un aporte fundamental en esta área fueron las siguientes[26]:

1. AlexNet, que significó el gran avance en la implementación de arquitecturas de aprendizaje profundo (Krizhevsky, 2012)
2. VGG, a partir de la cual las redes neuronales se vuelven muy profundas mediante la implementación por bloques (Simonyan, 2014)
3. GoogLeNet (Szegedy, 2015)
4. ResNet, Redes neuronales de gran profundidad (He K. Z., 2016)

Estas arquitecturas que han sido pilares en el desarrollo del aprendizaje profundo se explicarán a continuación.

[25] https://www.image-net.org/challenges/LSVRC/
[26] https://github.com/floodsung/Deep-Learning-Papers-Reading-Roadmap

AlexNet

AlexNet es una arquitectura de CNN desarrollada por Alex Krizhevsky, Ilya Sutskever y Geoffrey Hinton en 2012, que utiliza *kernels* convolucionales más grandes de lo usual hoy en día, lo que le permite aprender características complejas de las imágenes. Para su entrenamiento una de las novedades en su momento fue la utilización de GPUs, lo que facilitó un entrenamiento mucho más rápido, en comparación con modelos anteriores. Lo anterior, dado que, operaciones como la convolución o la multiplicación de tensores pueden paralelizarse en hardware. En particular, para su entrenamiento se utilizaron convoluciones agrupadas para ajustar el modelo en dos GPU.

Esta red tiene ocho capas, cinco de ellas convolucionales y tres FC (Ver Figura 31). Las capas convolucionales y las dos capas FC previas a la clasificación utilizan funciones de activación ReLU, siendo una de las primeras arquitecturas que utilizaron esta función de activación. La última capa FC utiliza función de activación *softmax*.

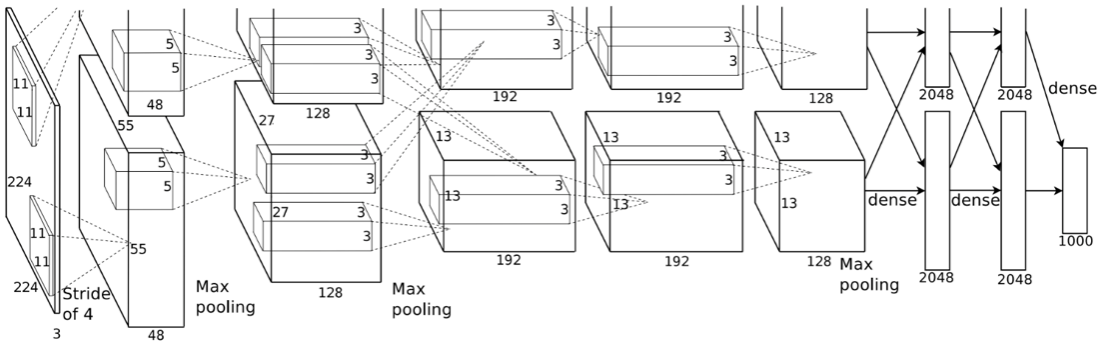


Figura 31. Ilustración de la arquitectura AlexNet y la operación entre dos GPUs para el entrenamiento (Figura tomada del artículo original (Krizhevsky, 2012))

Además de superar a los métodos de visión por computador de la época, esta arquitectura se caracteriza por tener alrededor de 60 millones de parámetros, que permitieron que los filtros de extracción de características obtenidos mediante el aprendizaje presentaran un rendimiento superior a los métodos diseñados manualmente. Este aspecto se puede ilustrar visualizando los 96 *kernels* convolucionales (11× 11×3) aprendidos por la primera capa convolucional de la red para imágenes de entrada de 224×224×3. Los 48 filtros superiores permiten extraer información espacial de la imagen y corresponden a

la primera GPU, mientras que los 48 filtros inferiores permiten extraer información espectral y de texturas y corresponden a la segunda GPU.

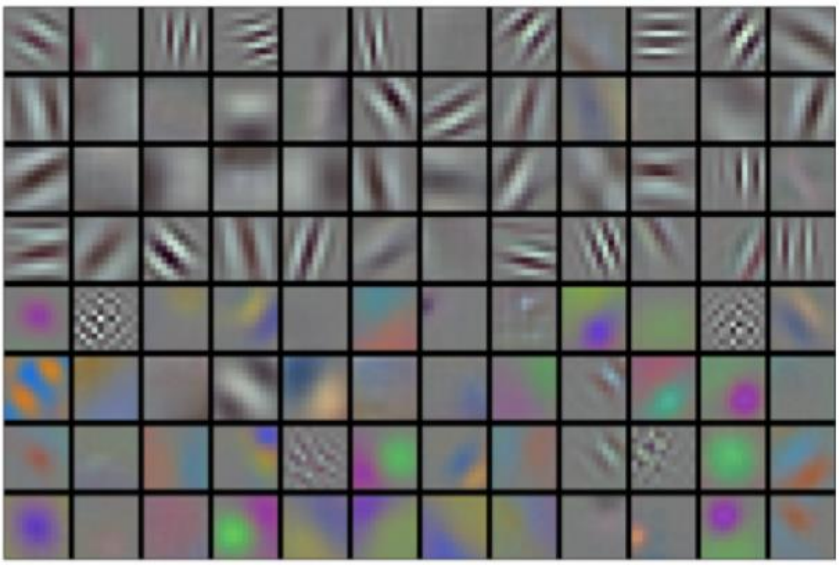


Figura 32. Filtros de imagen aprendidos por la primera capa de AlexNet con la GPU 1 (48 filtros superiores) y con la GPU 2 (48 filtros inferiores)
(Figura tomada del artículo original (Krizhevsky, 2012))

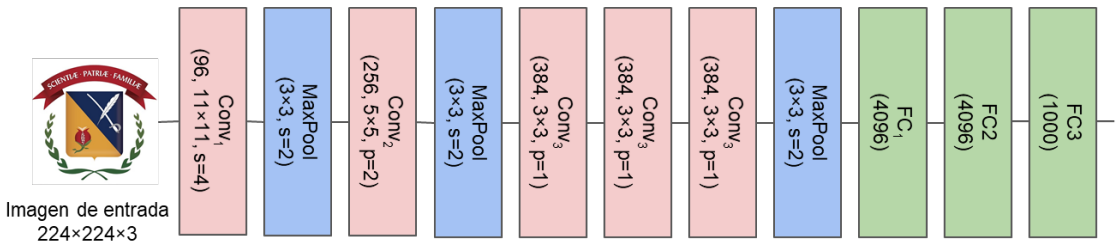


Figura 33. Esquema general de arquitectura AlexNet.

Un ejemplo de implementación de la arquitectura AlexNet se muestra a continuación. Aspectos característicos de esta red incluyen en la primera capa un filtro de gran dimensión (11×11) articulado con una reducción de dimensiones significativa (*padding* de 4). Igualmente, el número de canales aumenta en comparación con LeNet. Las capas convolucionales posteriores utilizan un tamaño de filtro menor (5×5 y 3×3). Para las dos capas densas previas a la clasificación, esta arquitectura puede involucrar el uso de *dropout* con fines de reducción de *overfitting*.

```
model = Sequential([
        Conv2D(filters=96, kernel_size=11,strides=4,
activation='relu'),
        MaxPool2D(pool_size=3, strides=2),
```

```
        Conv2D(filters=256, kernel_size=5,padding='sa
me', activation='relu'),
        MaxPool2D(pool_size=3, strides=2),
        Conv2D(filters=384, kernel_size=3,padding='sa
me', activation='relu'),
        Conv2D(filters=384, kernel_size=3,padding='sa
me', activation='relu'),
        Conv2D(filters=256, kernel_size=3,padding='sa
me', activation='relu'),
        MaxPool2D(pool_size=3, strides=2),
        Flatten(),
        Dense(4096, activation='relu'),
        Dense(4096, activation='relu'),
        Dense(10)])
```

VGG

VGG es una arquitectura propuesta por el grupo de investigación en geometría visual (VGG) de la universidad de Oxford en 2014. Esta arquitectura conserva la estructura de las dos arquitecturas presentadas anteriormente, en cuanto a que está estructurada en dos partes: extracción de características (capas conv. + capas *pooling*) y clasificación (capas FC). Una de las principales novedades de esta arquitectura radica en que se puede construir a partir de bloques compuestos por una secuencia de capas convolucionales seguidas de una capa de *pooling*. Para cada bloque, es posible especificar el número de capas convolucionales y el número de filtros o canales de salida del bloque. La estructura general del bloque VGG se muestra en la Figura 34 (Simonyan, 2014).

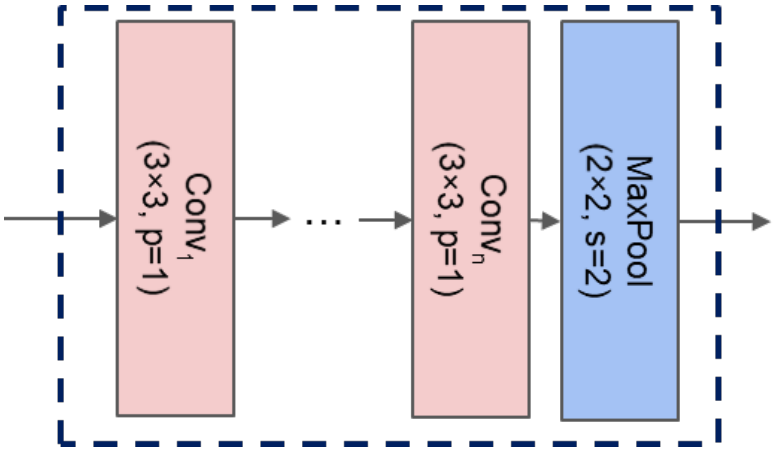


Figura 34. Esquema de bloque en arquitectura VGG con n capas convolucionales y número de canales variable.

La construcción de la arquitectura a partir de bloques reutilizables permite representar de manera compacta una red profunda. Dos de las arquitecturas VGG más difundidas corresponden a VGG16 y a VGG19, donde el número representa la profundidad del modelo. La representación de una arquitectura VGG se ilustra en la Figura 35 (Simonyan, 2014).

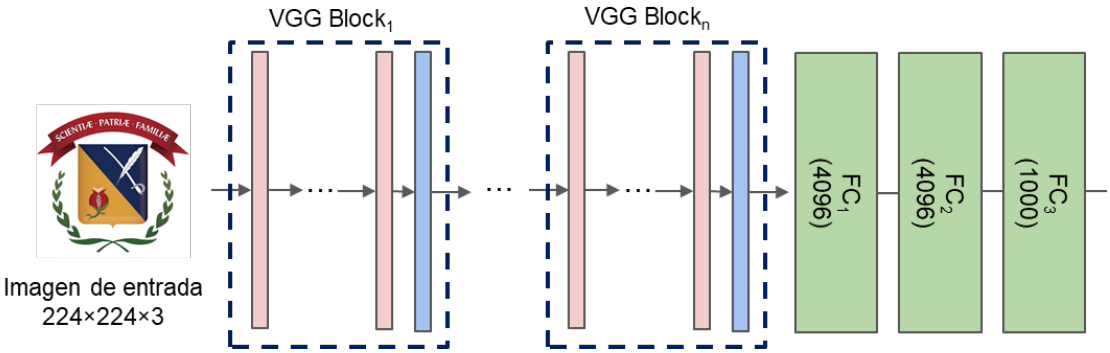


Figura 35. Arquitectura VGG conformada a partir de bloques.

GoogLeNet

Las arquitecturas presentadas hasta aquí están diseñadas de tal forma que extraen las características espaciales y espectrales de la imagen a través de una serie de capas de convolucionales y de pooling para reducir dimensiones. Estas características extraídas se procesan mediante un perceptrón multicapa, con una secuencia de capas de FC, que finalizan con la clasificación de los datos de entrada. Adicionalmente, este tipo de estructura fue compactada mediante bloques a partir de la propuesta de la arquitectura VGG.

Un enfoque alternativo propuso representar de una manera más amplia la estructura espacial de la imagen mediante el uso de capas convolucionales combinadas con capas FC (Lin, 2013) desde los primeros bloques de la arquitectura. De esta forma, los bloques Network in Network (NiN) están conformados por una capa convolucional seguida de múltiples capas convolucionales 1×1, que se comportan como capas totalmente interconectadas.

Una vez se implementa una arquitectura a partir de bloques NiN, el número de canales de salida se reduce al número de clases del problema por medio de las capas convolucionales 1×1. Esto implica que no sea necesario utilizar las capas FC de salida de los modelos anteriores, solo basta con utilizar una capa de *pooling* promedio global (*Global average pooling*). Su ventaja radica en un menor número

de parámetros y el precio a pagar puede implicar un mayor tiempo de entrenamiento.

A partir de estos fundamentos principales: la filosofía por bloques utilizada tanto en VGG como en NiN, y el uso de representaciones de diferentes dimensiones unida al *pooling* global de salida en NiN, Google propuso la arquitectura GoogLeNet. La idea en esta arquitectura es utilizar una combinación de filtros de distintos tamaños, que se articulan en un solo bloque conocido como *Inception*. Este bloque permite extraer información en paralelo mediante capas convolucionales de diferentes dimensiones que incluso involucran una capa de *max pooling*. La arquitectura completa se conforma a partir de la conexión de estos bloques con otras capas en serie. La arquitectura del bloque Inception se muestra en la Figura 36 (Szegedy, 2015).

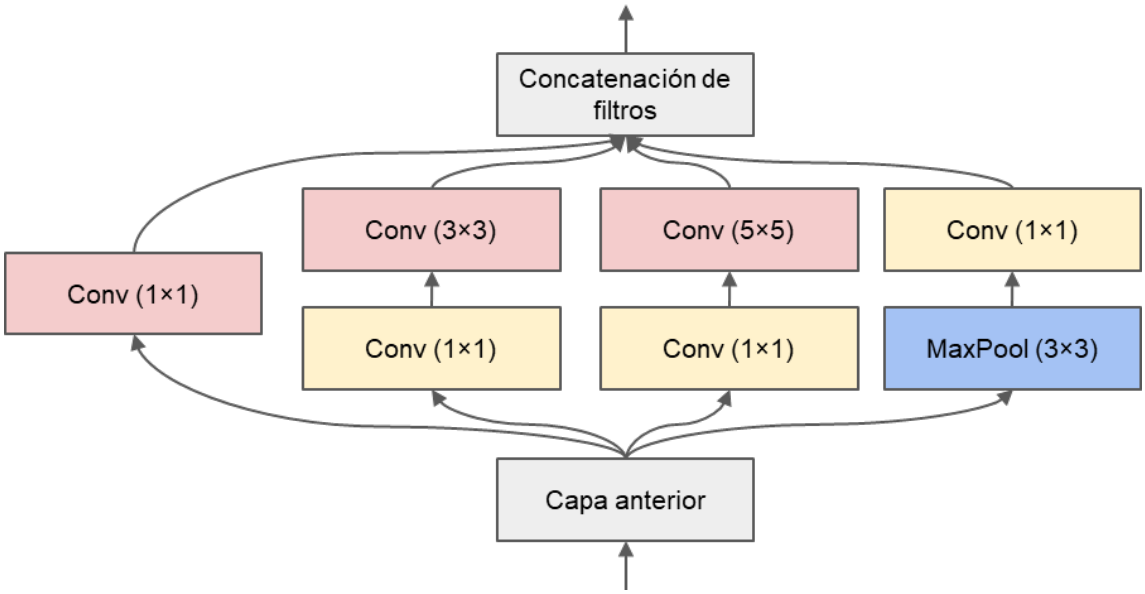


Figura 36. Esquema de bloque Inception para arquitectura GoogleNet.

El esquema general de la arquitectura GoogLeNet conformado a partir de bloques *Inception*, algunas capas convolucionales y *pooling* de entrada, más la capas de *global average pooling* y capa FC de clasificación se muestra en la Figura 37. En el intermedio de algunos bloques Inception se utilizan capas de *max pooling* para reducir la resolución de los datos a nivel de filas y columnas. Al interior de los bloques Inception se utilizan capas convolucionales 1×1 que permiten reducir la dimensión, pero en este caso a nivel de canales (Szegedy, 2015).

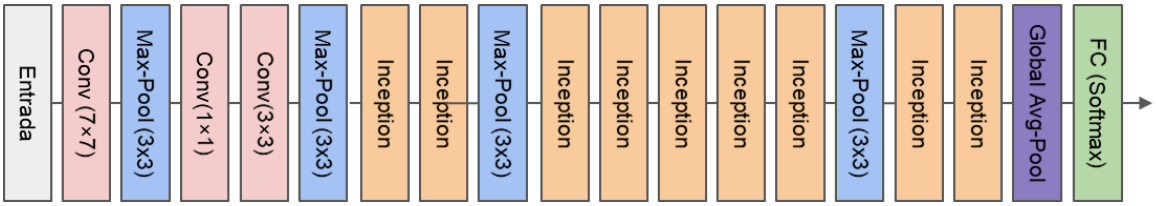
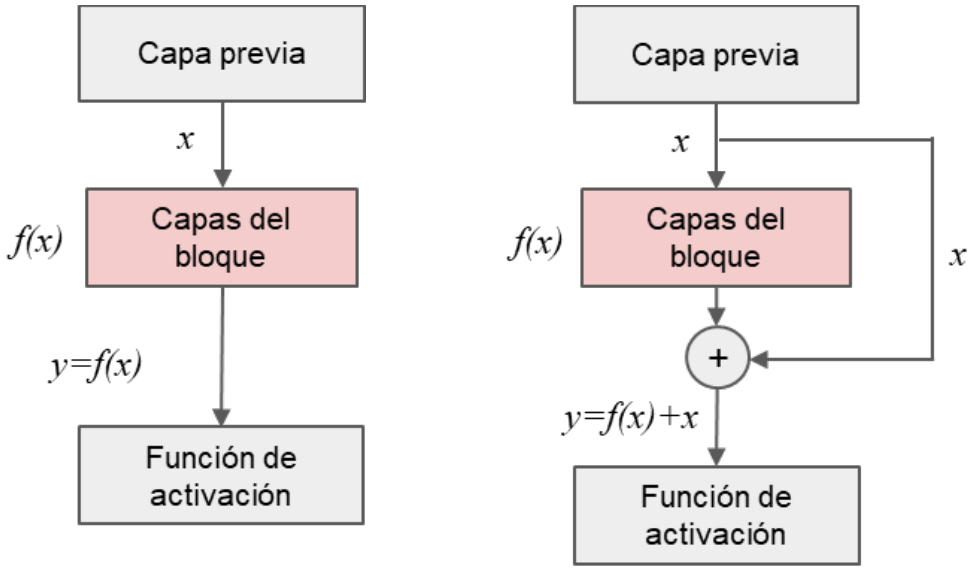


Figura 37. Diagrama general arquitectura GoogleNet.

ResNet

Las arquitecturas estudiadas hasta aquí conectan de manera secuencial los bloques definidos en cada una de ellas, tal como lo ilustra la Figura 38(a).



(a) Sin conexión residual (b) Con conexión residual

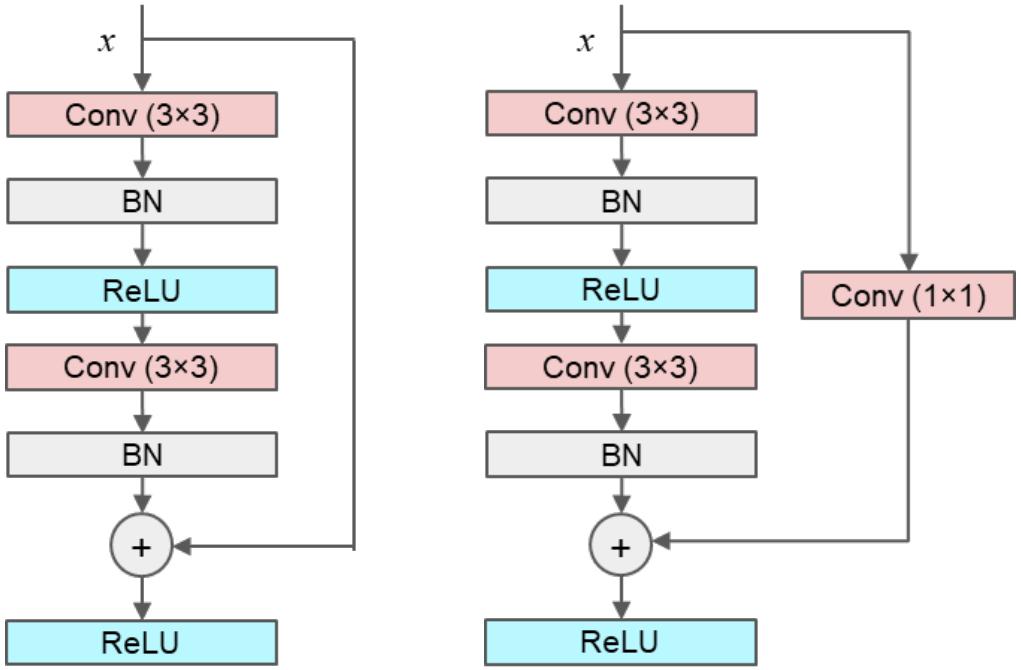
Figura 38. Esquema de conexión de bloques en arquitecturas CNN.

La arquitectura ResNet va un paso más allá y propone combinar la función no lineal entregada por un bloque ($f(x)$) con el componente lineal de entrada al bloque (x) (Ecuación (78)) (Zhang, 2021). Esta combinación se conecta posteriormente a la función de activación (ver Figura 38(b)).

$$g(x) = x + f(x) \tag{78}$$

En ResNet (He K. Z., 2016), el bloque de la arquitectura está conformado por dos capas convolucionales 3×3 con igual número de canales, seguidas cada una por

un proceso de *batch normalization* y una función de activación ReLU. Previo a la conexión de esta última función de activación se realiza la conexión residual. En este punto, la arquitectura contempla dos posibilidades. La primera consiste en conservar el número de canales de la entrada, por lo cual el número de canales de las dos capas convolucionales será igual al número de canales de entrada. De esta forma, la señal de salida del bloque (*f(x)*) puede adicionarse a esta misma entrada (*x*). La segunda alternativa permite variar el número de canales de salida, por lo cual es necesario adicionar en la conexión residual una capa convolucional 1×1 que permita ajustar el número de canales de la entrada *x* al número de canales configurados en las dos capas convolucionales del bloque (He K. Z., 2016).



(a) Conexión conservando el número de canales de entrada

(b) Conexión variando el número de canales de entrada

Figura 39. Esquemas de conexión residual en arquitectura ResNet.

DenseNet

En la arquitectura ResNet se estableció el uso de conexiones residuales que implican una adición entre el dato de entrada y una combinación no lineal de este. Tomando este tipo de conexión como referencia, la arquitectura DenseNet

propone el uso de conexiones similares, con la diferencia que los datos no se suman, sino que se combinan mediante concatenación (Huang, 2017). De acuerdo con sus autores, este tipo de conexión está orientado a mejorar el flujo de información y los gradientes a través de la red, lo que facilita su entrenamiento, además de tener un efecto de regularización sobre la red.

La conexión *feedforward* de la arquitectura DenseNet se realiza hacia varios dentro de un bloque, a diferencia de las conexiones residuales en ResNet. La concatenación de los datos y la conexión en diversos puntos hace que la cantidad de datos se incremente, por lo cual es necesario utilizar una red tipo perceptrón multicapa en la salida con el fin de reducir el número de atributos. Esto significa que la última capa del bloque está densamente conectada a todas las capas anteriores (Huang, 2017). El bloque fundamental de la arquitectura general de conexiones densas de DenseNet se ilustra en la Figura 40 y en la Ecuación (79). La idea de este tipo de bloque es permitir una conexión directa dese cualquier capa a todas las capas subsecuentes del bloque. De esta forma, la salida es una función de diferentes procesos previos.

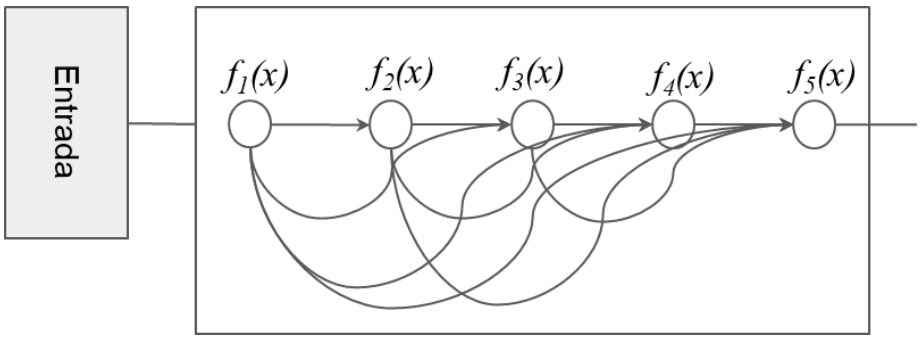


Figura 40. Ejemplo de un bloque denso de 5 capas de una red DenseNet. Cada capa toma como entradas las salidas de todas las capas que la preceden en el bloque.

Como lo ilustra la Figura 40, en el bloque DenseNet se conectan las salidas de todas las capas intermedias (con tamaños de mapa de características coincidentes) hacia las siguientes capas, para garantizar el máximo flujo de información entre ellas. Para ello, las unidades de convolución al interior de los bloques Dense tienen el mismo número de canales de salida, con el fin de realizar la concatenación a lo largo de esta dimensión. De acuerdo con lo anterior, la última capa del bloque recibe los mapas de características de todas las capas anteriores. En consecuencia, el mapa de características de salida del bloque DenseNet estará dado por la concatenación de las salidas de cada una de las capas intermedias del bloque, así:

$$x \to \left[x, f_1(x), f_2\left(x, f_1(x)\right), f_3\left(x, f_1(x), f_2\left(x, f_1(x)\right), \cdots\right)\right] \quad (79)$$

De acuerdo a la estructura de la red DenseNet, cada una de las funciones compuestas $f_n(x)$ involucran tres capas: *Batch normalization*, función de activación (ej. ReLU) y una capa de convolución 3x3. Para interconectar los bloques densos, la red incluye capas de transición conformadas por una capa convolucional 1x1 que permite controlar el número de canales y una capa de *pooling* promedio 2x2 para realizar *downsampling*. En conclusión, las principales unidades que componen DenseNet corresponden a los bloques densos y las capas de transición como se muestra a continuación.

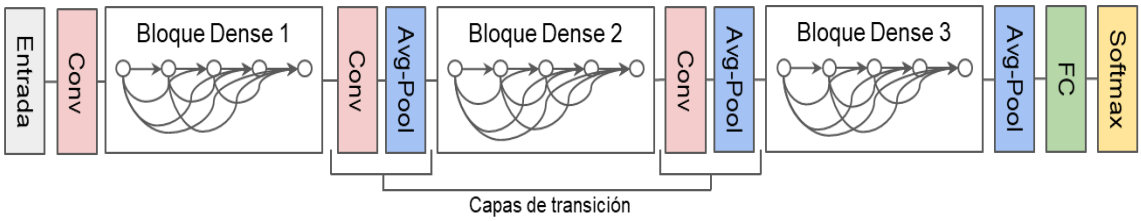


Figura 41. Arquitectura DenseNet.

Transferencia de aprendizaje

El aprendizaje por transferencia es una estrategia de diseño en la que la información de un modelo pre-entrenado para una tarea específica se utiliza en un nuevo problema, es decir, es posible aprovechar los conocimientos de dicho modelo. Los dos modelos pueden diferenciarse ya sea en la complejidad de las dos tareas, o también, es posible realizar la transferencia desde un modelo entrenado con más datos a otro con menos datos (utilizando un conjunto de datos diferente). En este sentido, la transferencia de aprendizaje puede ser útil para modelos de última generación y también cuando no hay suficientes datos de entrenamiento.

A nivel de clasificación de imágenes, es posible encontrar modelos top en el estado del arte, que por lo general han sido entrenados con ImageNet y están disponibles para descarga tanto en repositorios como para importarlos directamente en los *frameworks* de desarrollo de modelos de aprendizaje profundo. Algunas opciones que permiten obtener estos modelos se listan a continuación:

- TensorFlow Hub: https://www.tensorflow.org/hub/
- PyTorch Hub: https://pytorch.org/hub/
- HuggingFace: https://huggingface.co
- Keras applications: https://keras.io/api/applications/

A septiembre de 2023, *keras applications* dispone los siguientes modelos para realizar transferencia de aprendizaje:

- Xception
- VGG (16, 19)
- ResNet (50 (v1, v2), 101 (v1, v2), 152 (v1, v2))
- Inception v3
- InceptionResNetv2
- MobileNet (v1, v2)
- DenseNet (121, 169, 201)
- NASNetMobile
- NAsNetLarge
- EfficientNet (B0-B7)
- EfficientNetV2 (B0-B3, S, M, L)
- ConvNext (Tiny, small, base, Large, Xlarge)

Estos modelos que han sido entrenados en un problema sirven como punto de partida en un problema relacionado, reduciendo el tiempo de entrenamiento y por lo general disminuyendo el error de generalización. Al importarlos pueden aplicarse directamente en la predicción, para extraer características o también para realizar un ajuste fino del nuevo modelo sobre un problema diferente.

Predicción

Los modelos pre-entrenados pueden utilizarse directamente para realizar predicciones sobre las clases del dataset en las cuales fue entrenado. En el caso de ImageNet, el modelo por transferencia de aprendizaje puede utilizarse para realizar predicciones sobre datos de algunas de las 1000 clases de este dataset.

Para importar un modelo por transferencia de aprendizaje en Colaboratory, es necesario importar el modelo específico desde keras/tensorflow. También es necesario importar los métodos que permiten pre-procesar y adaptar las imágenes a los requerimientos del modelo y su predicción.

Al instanciar la clase que importa el modelo (ej. VGG16) es posible especificar si se importa solamente la estructura del modelo, o también se importan los

parámetros (pesos) del modelo pre-entrenado con ImageNet. Si se incluyen estos pesos, el modelo estará listo para realizar predicciones sobre nuevos datos. El siguiente bloque de código muestra el proceso para realizar estas tareas. En este caso se lee una imagen cargada en el entorno de ejecución, la cual se pre-procesa y se estructura a las dimensiones de entrada de la arquitectura (224×224×3).

```
from tensorflow.keras.applications.vgg16 import
 VGG16
from tensorflow.keras.preprocessing import imag
e
from tensorflow.keras.applications.vgg16 import
 preprocess_input, decode_predictions
import numpy as np

model = VGG16(weights='imagenet')

img_path = 'Gato1.jpg'
img = image.load_img(img_path, target_size=(224
, 224))
x = image.img_to_array(img)
x = np.expand_dims(x, axis=0)
x = preprocess_input(x)

preds = model.predict(x)
print('Predicted:', decode_predictions(preds, t
op=3)[0])
```

El parámetro `top` especificado en la decodificación de predicciones (`decode_predictions`), determina el número de clases más probables que pueden corresponder a los datos de entrada.

Extracción de características

Un modelo pre-entrenado puede utilizarse también como bloque de extracción de características hacia otro clasificador. Para esto es necesario especificar al importar el modelo, que no se utilizarán las capas top o capas de clasificación de la arquitectura (últimas capas del modelo), esto mediante el argumento "`include_top=False`".


```
from tensorflow.keras.applications.vgg16 import
 VGG16
from tensorflow.keras.preprocessing import imag
e
from tensorflow.keras.applications.vgg16 import
 preprocess_input
import numpy as np

model = VGG16(weights='imagenet', include_top=F
alse)
```

Al realizar la predicción sobre un modelo importado sin sus capas top, la salida no entrega un vector de probabilidades de pertenencia a la clase, sino que entrega un mapa de características. Esto se ilustra en el siguiente ejemplo que da como resultado un mapa de características 7×7 por cada canal de salida.

```
img_path = 'Gato1.jpg'
img = image.load_img(img_path, target_size=(224
, 224))
x = image.img_to_array(img)
x = np.expand_dims(x, axis=0)
x = preprocess_input(x)


features = model.predict(x)

# Ver las dimensiones de salida del modelo en
el summary
print(features.shape)
#print(features)
print(features[0,0:7,0:7,128])
```



```
[[ 0.         5.7264395  0.         0.         0.         0.
   4.35953  ]
 [ 0.         8.568925  41.75738   50.764385   0.         0.
   0.       ]
 [ 0.         5.896989   0.         0.         0.         0.
   0.       ]
 [ 0.         0.         0.         0.         0.         0.
   0.       ]
 [ 0.         0.         0.         0.         0.         0.
   0.       ]
 [ 0.         0.         0.         0.         0.         0.
   0.       ]
 [ 0.         0.         0.         0.         0.         1.1514584
   8.352236 ]]
```

En este sentido, la salida de un modelo de este tipo puede concatenarse con un modelo de aprendizaje tradicional, o también es posible especificar una nueva estructura de capas top que permita realizar un ajuste fino del modelo sobre un problema diferente (con un conjunto de datos diferente) y la posterior clasificación sobre el nuevo conjunto de datos.

Ajuste fino (*fine tuning*)

Como se ha comentado hasta aquí, los modelos pre-entrenados pueden importarse tanto en estructura como en parámetros, con la posibilidad de excluir la capa de salida (la que realiza la clasificación) incluida en las capas top del modelo. En este caso será necesario agregar al nuevo modelo una capa de salida cuyo tamaño corresponda al número de clases del nuevo conjunto de datos de destino. Una vez realizado este ajuste, es posible entrenar el nuevo modelo con el nuevo conjunto de datos.

Además de poder entrenar la capa de salida, es posible entrenar capas anteriores del modelo. De hecho, las capas de un modelo de keras/tensorflow cuentan un parámetro denominado "*Trainable*", que define si los parámetros de esa capa se conservan constantes (conocido también como congelar, con la opción False), o si por el contrario se actualizan durante cada iteración del ciclo de entrenamiento (se descongelan con la opción True). En este sentido, los parámetros de las capas nuevas que se han agregado al modelo se inicializan de forma aleatoria y se entrenan desde cero. Los parámetros de las capas que se descongelan inician con los valores que traían del modelo original, y se empiezan a ajustar a los nuevos datos durante el entrenamiento en un proceso conocido como ajuste fino (*fine tuning*).

La importación de un modelo para *fine tuning* es similar a la importación para extracción de características (es decir, no se importan las capas top):

```
from tensorflow.keras.applications.vgg16 import V
GG16
from tensorflow.keras.preprocessing import image
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Dense, Global
AveragePooling2D

base_model = VGG16(weights='imagenet', include_to
p=False)
```

Luego de importar el modelo, se agregan las capas top para la clasificación. Para este ejemplo, se agregará una capa de *average pooling,* una capa densa de 1024 unidades y una capa densa de 2 unidades. Este último valor corresponde al número de clases del nuevo problema a abordar, que para el ejemplo es un caso binario.

```
# Agregar una capa de average pooling
x = base_model.output
x = GlobalAveragePooling2D()(x)

# Agregar una capa Fully Connected
x = Dense(1024, activation='relu')(x)

# Agregar una capa FC del número de clases
predictions = Dense(2, activation='softmax')(x)

# Modelo final para entrenar
model = Model(inputs=base_model.input,
outputs=predictions)
```

Una vez se ha consolidado el modelo (modelo base + nuevas capas top) es posible especificar cuáles de las capas se entrenarán y cuáles no. Una alternativa, por ejemplo, consiste en congelar todas las capas que se importaron del modelo original (modelo base) y entrenar solamente las nuevas capas top, como lo muestra el siguiente bloque de código.

```
For layer in base_model.layers:
    layer.trainable = False

model.compile(...)
```



```
model.fit(...)
```

La segunda alternativa consiste en definir cuáles capas se entrenarán y cuáles no. Para ello, primero se pueden visualizar los nombres de las capas y sus índices para determinar cuáles capas congelar:

```
for i, layer in enumerate(base_model.layers):
   print(i, layer.name)
```

A continuación, se definen las capas que no se entrenarán con el parámetro en `False` y las capas que sí se entrenarán con el parámetro en `True`. En el siguiente ejemplo, se congelan las primeras quince capas y se entrenarán las capas restantes.

```
for layer in model.layers[:15]:
  layer.trainable = False
for layer in model.layers[15:]:
  layer.trainable = True
```

A partir de esto, es posible compilar y entrenar el modelo de la forma usual.

