

Redes neuronales convolucionales

Aprendizaje profundo

Departamento de Sistemas Informáticos

E.T.S.I. de Sistemas Informáticos - UPM



Contexto actual

Las redes convolucionales (CNN) son una de las principales arquitecturas usadas



Figura 1. Tendencia de diferentes técnicas de deep learning a lo largo de los años. Fuente: Frontiers¹

Son la técnica predominante a la hora de procesar **imágenes** y **datos tabulares**

¹ Emmert-Streib, F., Yang, Z., Feng, H., Tripathi, S., & Dehmer, M. (2020). *An introductory review of deep learning for prediction models with big data*. *Frontiers in Artificial Intelligence*, 3, 4.

Motivación

Surgen para adaptar las redes neuronales al tratamiento de imágenes

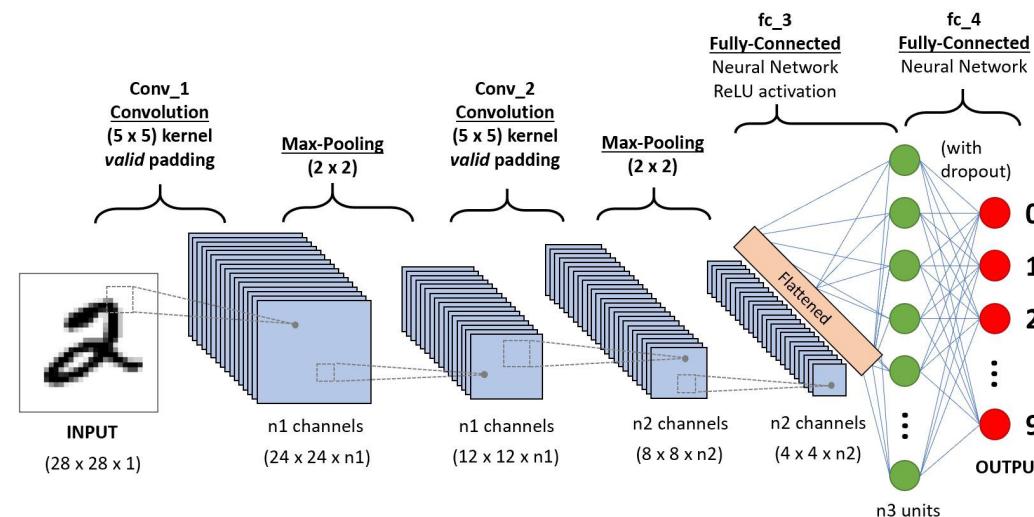
- Aprovechan las características de los datos espaciales para reducir el número de parámetros de la red
- Aprenden la invarianza de los datos, lo que les permite generalizar mejor
- Son capaces de extraer características jerárquicas de los datos, ayudando a identificar patrones complejos

Se apoyan en la operación de **convolución** para procesar los datos

Arquitectura de una CNN

Se divide en dos partes para el proceso de datos espaciales:

1. **Extracción de características:** Capas de convolución, *pooling* y normalización
2. **Inferencia:** Capas densas (MLP)



Arquitectura de una CNN. Fuente: Analytics Vidhya

Operación de convolución

¿Qué es una operación de convolución?

En nuestro contexto definiremos la convolución como **operación que procesa una matriz numérica manteniendo las relaciones espaciales de la misma**

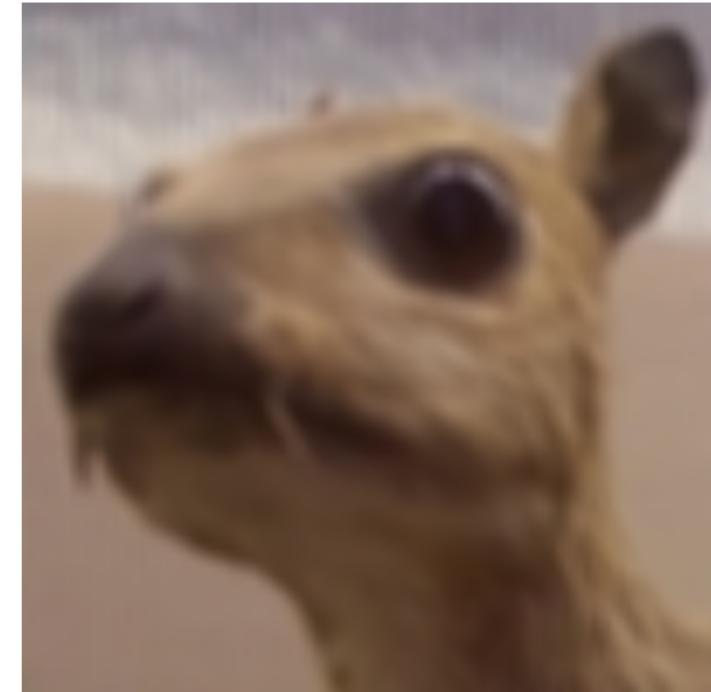
- Se aplica un **filtro** (o **kernel**) a la matriz de entrada produciendo una salida denominada **mapa de características**
- En visión artificial, se han utilizado tradicionalmente para producir efectos

Ejemplo de convolución: Desenfoque tipo caja

Antes de la
convolución



Después de la
convolución



Núcleo

| | | |
|---|---|---|
| 1 | 1 | 1 |
| 1 | 1 | 1 |
| 1 | 1 | 1 |

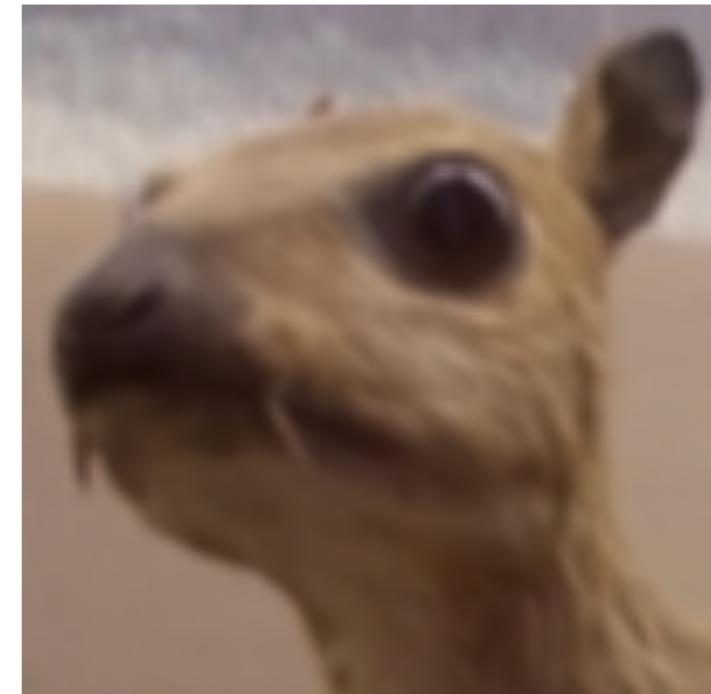
Ejemplo de desenfoque tipo caja.

Ejemplo de convolución: Desenfoque tipo caja

Antes de la
convolución



Después de la
convolución



Núcleo

| | | |
|---|---|---|
| 1 | 2 | 1 |
| 2 | 4 | 2 |
| 1 | 2 | 1 |

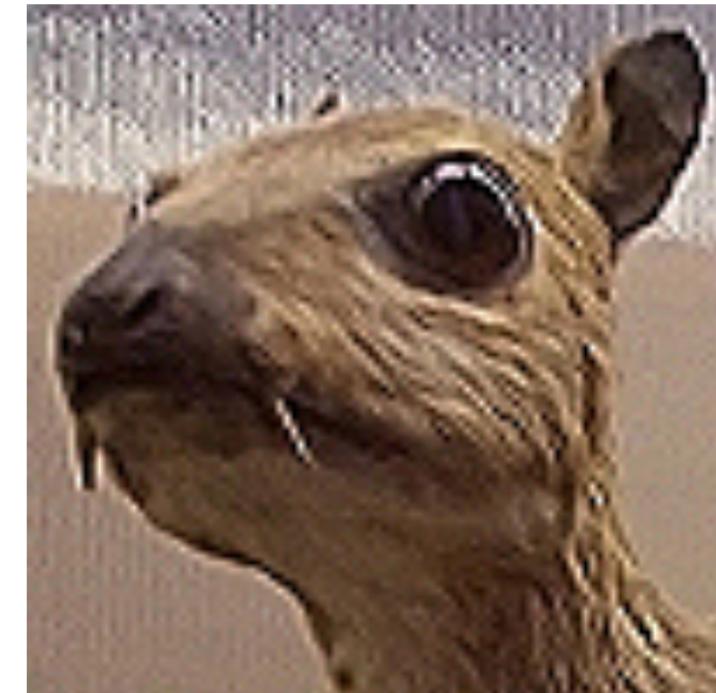
Ejemplo de desenfoque gaussiano.

Ejemplo de convolución: Desenfoque tipo caja

Antes de la
convolución



Después de la
convolución



Núcleo

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

Ejemplo de desenfoque realce de bordes.

Operación de convolución (I)

Producto escalar de una matriz con un **filtro (kernel)** que se desplaza por ella



Operación de
convolución

Figura 2. Convolución sobre imagen de un único canal. Fuente: Analytics Vidhya

Dos elementos fundamentales:

- **Matriz de entrada:** Dos (e.g. imagen en escala de grises) o tres dimensiones (e.g. imagen a color)
- **Filtro:** Ancho y alto determinado, mientras que coincide en profundidad con la matriz de entrada

El filtro recorre la matriz de entrada haciendo el producto escalar en cada posición

Operación de convolución (II)

La región que el filtro (kérnel) es capaz de observar se denomina **campo receptivo**



Figura 3. Convolución sobre una imagen de 3 canales. Fuente: SaturnCloud

Convolución en redes neuronales

¿Y si en lugar de filtros preconfigurados, los «aprendemos»?

- Esa es la idea detrás de las redes neuronales convolucionales
- Una **convolución neuronal** cambia los valores del núcleo por neuronas con sus propios pesos

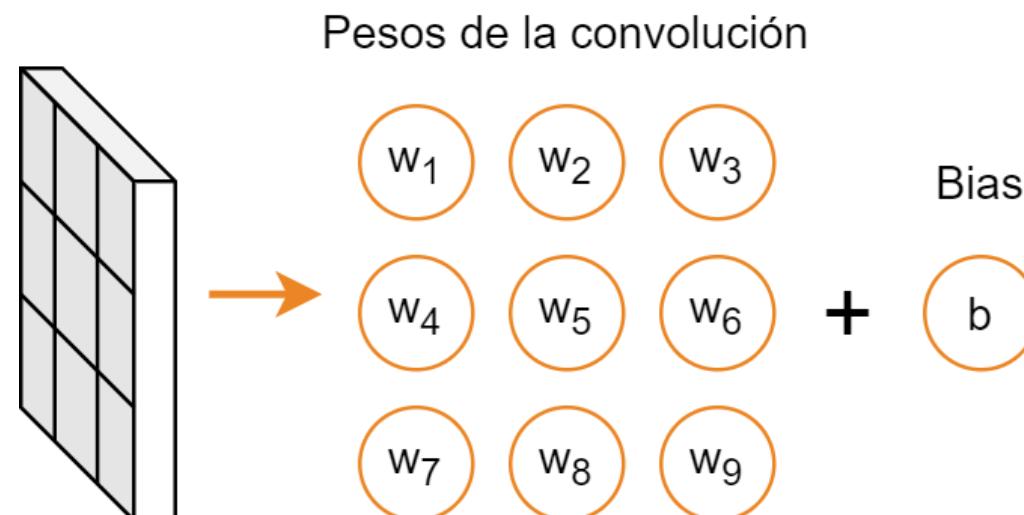


Figura 4. Pesos asociados a un filtro 2D

Activación de capas convolucionales

Tras la convolución, el resultado pasa por una función de activación no lineal



Figura 5. Proceso completo de obtención de mapa de características

La salida de la operación se denomina **mapa de características** del filtro

Hiperparámetros de la capa convolucional

Detalles de implementación

En PyTorch, la capa Conv2d se utiliza para definir las capas convolucionales

- También existe LazyConv2d para evitar especificar los canales de la entrada

```
import torch.nn as nn

conv_layer = nn.Conv2d(
    in_channels=3,           # Número de canales de la imagen de entrada (RGB = 3)
    out_channels=32,          # Número de filtros
    kernel_size=(3, 3),       # Tamaño del filtro
    stride=(1, 1),            # Salto del filtro
    padding='same',           # Relleno de la imagen
)
```

Estos parámetros son comunes en la mayoría de los *frameworks*

Número de convoluciones o filtros

El parámetro `out_channels` describe cuántos filtros que se aplicarán a la imagen

- Cada filtro es un conjunto de pesos y produce un mapa de características

```
conv_layer = nn.Conv2d(in_channels=3, out_channels=64, ...)
```

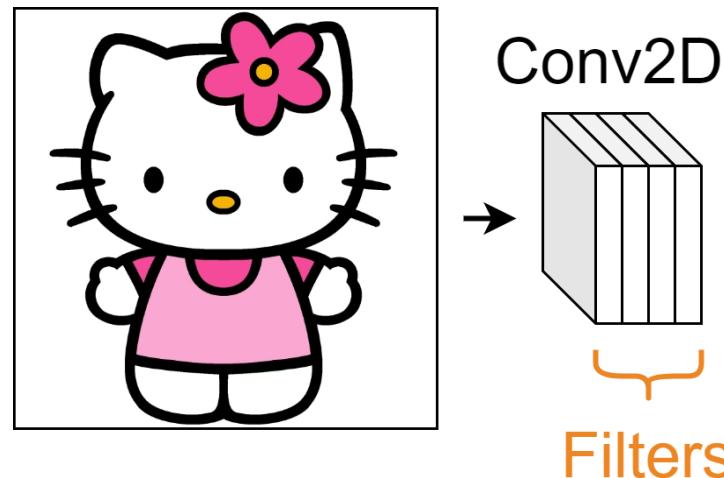


Figura 6. Capa convolucional generando \$N\$ mapas de características (imágenes)

Tamaño de filtro

El parámetro `kernel_size` dice el tamaño de cada filtro como (alto, ancho)

- Si sólo se indica un número N , se asume un filtro cuadrado de tamaño (N, N)

```
conv_layer = nn.Conv2d(..., kernel_size=(5, 5))
```

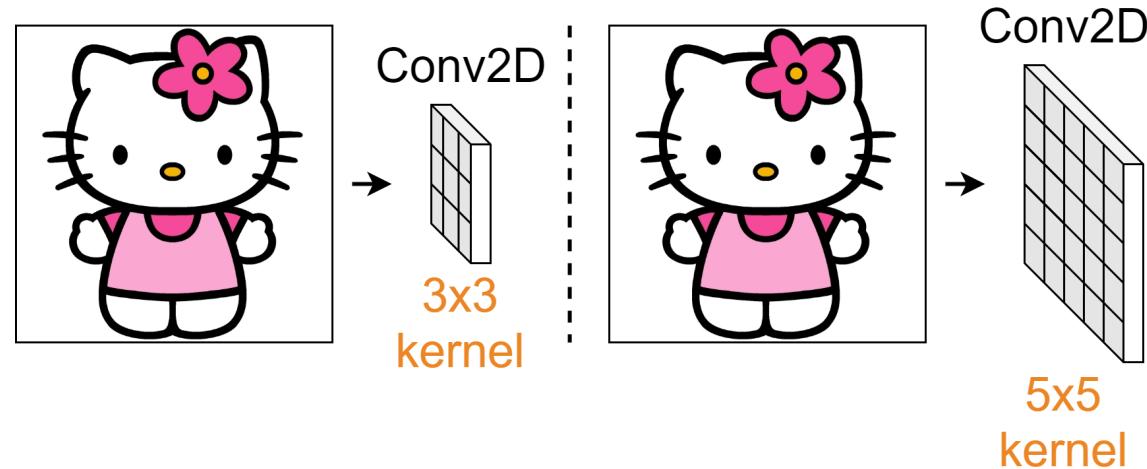


Figura 7. Dos filtros de tamaño 3 y 5

Salto de convolución

El parámetro `stride` determina el salto del filtro al deslizarse sobre la imagen

- También especificado como una tupla (`alto, ancho`) o un único entero `salto`
- Si no se especifica, el salto es de 1 en ambas direcciones

```
conv_layer = nn.Conv2d(..., stride=2)
```



Figura 8. Salto de 2 para el deslizamiento del filtro. Fuente: Towards Data Science

Relleno

El parámetro `padding` indica cómo se rellena el borde de la imagen

- `'valid'`: No se rellena la imagen
- `'same'`: Se rellena la imagen con ceros para mantener el tamaño de la salida

```
conv_layer = nn.Conv2d( . . . , padding='same' )
```



Figura 9. Relleno tipo `same` para mantener el tamaño de la salida. Fuente: Towards Data Science

Activación

El parámetro activation permite añadir una función de activación directamente a la salida de la capa

- Si no la salida es únicamente la suma ponderada de los pesos
- Suele usarse por claridad, según gusto o por ayudar a la definición de modelos dinámicos

```
conv_layer = nn.Sequential(  
    nn.Conv2d(in_channels=3, out_channels=32, kernel_size=3, padding='same'),  
    nn.ReLU()  
)
```

Era trampa, en otros *frameworks* existe esta posibilidad, en PyTorch no

- Pero ahora ya sabéis qué significa cuando estéis traduciendo de Keras u otros

Upsampling y downsampling

Cambios de dimensionalidad

A la hora de diseñar una red convolucional, las capas que cambian las dimensiones de la información son fundamentales

- ***Downsampling***: Reducción de la resolución espacial
- ***Upsampling***: Aumento de la resolución espacial

Las capas de convolución por defecto realizan *downsampling* de dos maneras:

- Mediante el uso del parámetro `stride`
- Mediante el uso de `padding` en la imagen de tipo `valid`

Pero en DL a veces necesitamos:ç

- Muchas capas de convolución, para extraer características cada vez más complejas, y
- Aumentar o, sobre todo, disminuir, la resolución espacial.

Reducción dimensional con *pooling*

El *pooling* es una operación que reduce la resolución espacial de la imagen



padding same

Figura 10. Diferentes operaciones de *pooling* sobre la misma matriz. Fuente: Towards Data Science

Devuelve un valor de cada región visitada por el filtro

- **Max pooling:** Máximo
 - **Average pooling:** Media
-

```
pooling_layer = nn.MaxPool2d(kernel_size=2, stride=2)
```

Si el stride no se indica, se asume que es igual al tamaño del filtro

Aumento dimensional con *upsampling*

El *upsampling* es una operación que aumenta la resolución espacial de la imagen



padding same

Figura 11. Ejemplo de *upsampling* mediante la técnica *bed of nails*. Fuente: Towards Data Science

Existen una amplia multitud de técnicas

- **Vecinos cercanos:** Se copia el valor de un píxel a toda la región generada
- **Interpolación:** Se rellena con valores interpolados de los píxeles vecinos
- ***Bed of nails*:** Se rellena con ceros

Ejemplo:

```
upsampling_layer = nn.Upsample(scale_factor=2, mode='nearest')
```

Strides para el cambio dimensional

Es otra alternativa para reducir la dimensión de la entrada

- Si el *stride* es mayor que 1, el filtro se desplaza más rápido por la imagen
- La salida de la convolución es más pequeña que la entrada



Figura 12. Un *stride* de 2 se puede usar como *downsampling* «inteligente». Fuente: Towards Data Science

La principal ventaja respecto al *pooling* es que se aprenden los pesos de los filtros

- Podemos decir que se usa un filtro «inteligente»

Filtros 1×1

Se usan en ocasiones para reducir la dimensionalidad de la imagen²

- Disminuye la cantidad de canales (menos complejidad y cálculos),
- Aplicación de operaciones no lineales sin alterar el tamaño de la imagen, y
- Cada canal de cada píxel como entrada de una red neuronal, así aprenden transformaciones complejas a nivel de canal.

Stride de 2x2

Figura 13. Una convolución de 1 único peso también se puede usar para realizar como *downsampling* «inteligente», pero en canales, no tamaño. Fuente: Towards Data Science

² Lin, M., Chen, Q., & Yan, S. (2013). *Network in network*. arXiv preprint arXiv:1312.4400.

Clasificación de dígitos con redes convolucionales.ipynb

Diferentes arquitecturas de CNN

LeNet - La primera arquitectura de CNN

Desarrollada por LeCun et al.³ en 1998 para reconocer de dígitos escritos a mano

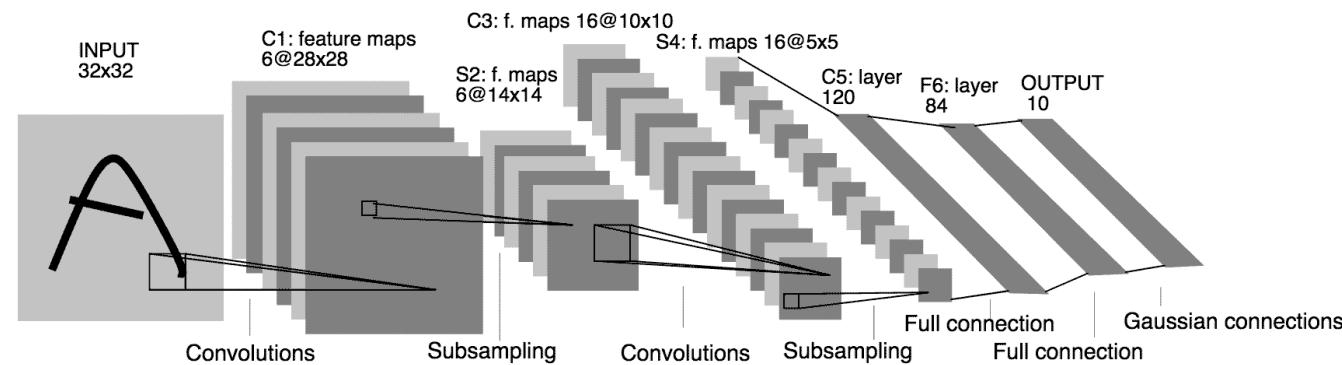


Figura 14. Arquitectura LeNet-5. Fuente: [Anatomies of Intelligence](#)

Es considerada el «Hola Mundo» del aprendizaje profundo

- **Arquitectura:** Dos capas convolucionales, dos capas *pooling* y tres capas densas
- **Principal problema:** *Vanishing gradients*

³ LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. Proceedings of the IEEE, 86(11), 2278-2324.. Proceedings of the IEEE, 86(11), 2278-2324.

AlexNet - Arquitectura que popularizó las CNN

Desarrollada por Alex Krizhevsky et al.⁴ en 2012, ganadora de ImageNet 2012



Arquitectura AlexNet

Figura 15. Arquitectura AlexNet. Fuente: Medium

- **Arquitectura:** Cinco convolucionales, 2 *pooling* y tres densas
- **Principal aportación:** Uso de *ReLU* y *dropout* para evitar el sobreajuste
- **Problema:** Muchos parámetros

Esta arquitectura ha inspirado el diseño de muchas arquitecturas posteriores

⁴ Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). *ImageNet classification with deep convolutional neural networks*. Advances in neural information processing systems, 25, 1097-1105.

GoogLeNet (Inception v1) - Bloques *inception*

Desarrollada por Szegedy et al.⁶ en 2014, ganadora de ImageNet 2014



Arquitectura GoogLeNet

Figura 16. Arquitectura GoogLeNet. Fuente: Medium

- **Arquitectura:** 22 capas con bloques *inception* (principal aportación)
- **Problema:** Complejidad de implementación

⁶ Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., ... & Rabinovich, A. (2015). *Going deeper with convolutions*. Proceedings of the IEEE conference on computer vision and pattern recognition, 1, 3.

Bloque *inception*

Estructura que permite la extracción de características a diferentes escalas

- Nos permite utilizar múltiples tipos de tamaño de filtro, en lugar de uno solo
- Luego se concatena el resultado de cada filtro y pasarlo a la siguiente capa
- Sucesivas versiones han ido añadiendo mejoras al bloque



Bloque _inception_ v1

Figura 17. Bloque *inception*. Fuente: The Startup

VGGNet - Arquitectura con muchas capas

Desarrollada por Simonyan y Zisserman⁷ en 2014



Figura 18. Bloque *inception*. Fuente: Electronics (MDPI)

- **Arquitectura:** 16 convoluciones (o más) y 3 densas
- **Problema:** Muchos parámetros, *vanishing gradients*
- **Ventaja:** Fácil de entender y de implementar

⁷ Simonyan, K., & Zisserman, A. (2014). *Very deep convolutional networks for large-scale image recognition*. arXiv preprint arXiv:1409.1556.

ResNet - Redes residuales

Desarrollada por He et al.⁸ en 2015



Figura 19. Arquitectura ResNet-34. Fuente: Medium

- **Arquitectura:** 34+ capas convolucionales con bloques residuales
- **Principal aportación:** Conexiones residuales para evitar el *vanishing gradient*
- **Ventaja:** Permite el entrenamiento de redes muy profundas

⁸ He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. Proceedings of the IEEE conference on computer vision and pattern recognition, 770-778.

Ejercicios

- Clasificación alternativa
- Implementando LeNet

Licencia

Esta obra está licenciada bajo una licencia Creative Commons
Atribución-NoComercial-CompartirlGual 4.0 Internacional.

Puedes encontrar su código en el siguiente enlace:
<https://github.com/blazaid/aprendizaje-profundo>