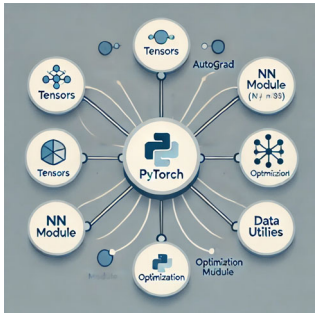
VISIÓN POR COMPUTADOR

Reconocimiento de Objetos:

Aprendizaje Profundo (Deep Learning)

Redes Neuronales Convolucionales (CNN)



Ingeniería de Sistemas y Automática

Universidad Miguel Hernández

Tabla de Contenidos

Aprendizaje Profundo (Deep Learning)

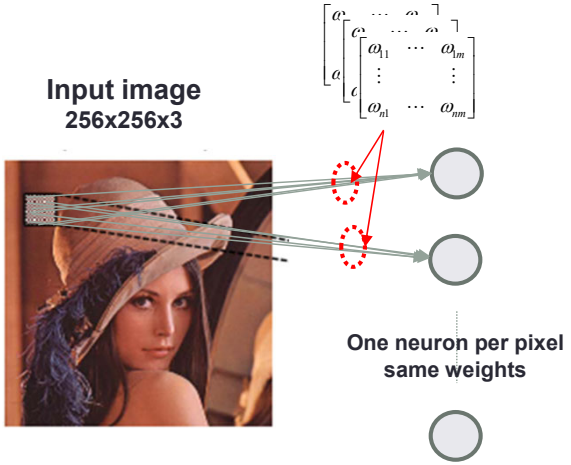
- Clasificación de imágenes sin detectar previamente descriptores
- Se trata de extraer patrones de la propia imagen durante el proceso de aprendizaje
- Arquitecturas:
  - Redes Neuronales Convolucionales (CNN)
    - Cada neurona actúa como un ***filtro convolucional*** (entorno de vecindad de cada pixel de la imagen)
    - Elevado número de capas ocultas para extraer información compleja (descriptores)
    - **Origen:** LeNet-5 (1998) Yann LeCun
    - Redes Residuales (RCNN): incorporan conexiones/saltos entre capas no contiguas (**Residuos**) para evitar el desvanecimiento del Gradiente

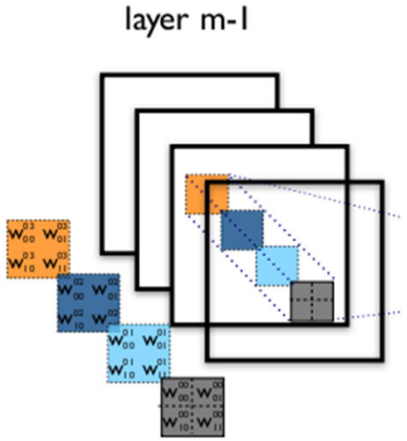
      "Gradient-based learning applied to document recognition" LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. Proceedings of the IEEE, 86(11) (1998)
  - Redes Transformer (ViT)
    - Utiliza el mecanismo de "***Atención***" que permite encontrar relaciones globales en la imagen
    - Propuesta en 2017 por *Google Brain Lab* en el artículo "*Attention is All You Need* " para LLMs
    - Se divide la imagen en "tokens" (parches) aprendiendo relaciones entre ellos

      "Attention is all You Need" Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017) Advances in Neural Information Processing Systems (NeurIPS)

Redes Neuronales Convolucionales (CNN)

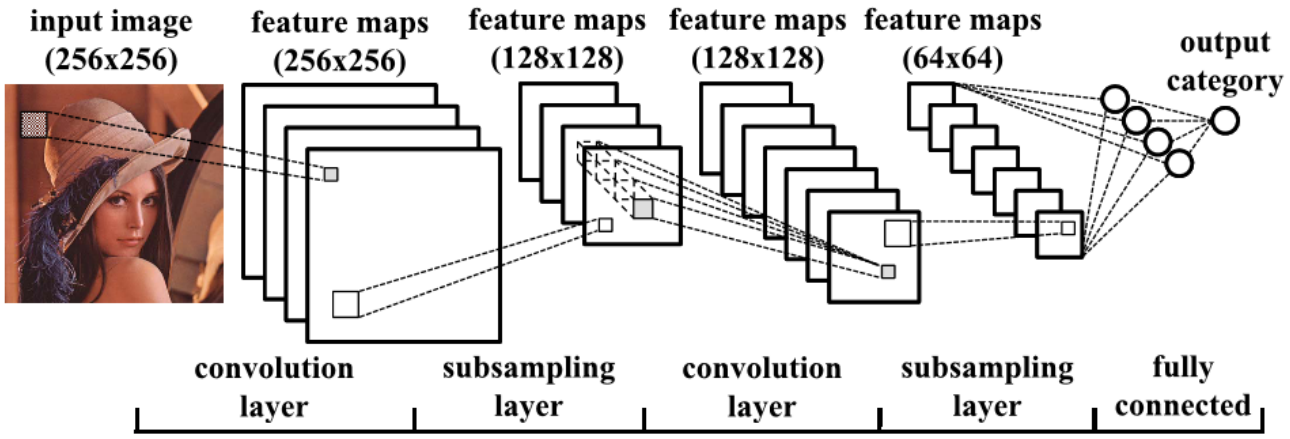
- Clasificación de imágenes sin usar descriptores:
  - La capa de entrada es la propia imagen
  - Segunda capa: una neurona por cada pixel de la imagen
    - En una Red Neuronal tradicional (MLP) todas las neuronas utilizarían toda la imagen (*explosión de parámetros*)
    - En una Red Neuronal Convolucional (CNN) cada neurona actúa sobre una ventana (entorno de vecindad de cada pixel de la imagen)
    - Los pesos en todas las neuronas en la misma capa son iguales
    - En cada capa hay varios niveles (***feature maps***) que realizan filtrados diferentes (diferentes pesos)
  - Cada nivel actúa como un filtro convolucional extrayendo características (la red aprende su propias características)





Redes Neuronales Convolucionales (CNN)

- LeNET-5 (1998): Yann LeCun y Yoshua Bengio (AT&T Labs) (5 capas - 60.000 parámetros)
  - La capa de entrada es la propia imagen
  - Cada neurona actúa sobre una ventana distinta de la imagen.
  - Cada capa esta formada por varios canales (agrupación de neuronas)
  - Las neuronas dentro del mismo canal utilizan los mismo pesos (mismo filtrado)
  - Diferentes canales ('*feature-maps*') de una capa realizan filtrados diferentes.
  - Se incorporan capas de submuestreo ('*max-pooling*') para reducir al dimensión de la red neuronal (máximo de la ventana)
  - Las capas de salida actúan como una capa normal de clasificación (MLP) (totalmente conectada)
  - Dataset MNIST: dígitos manuscritos (32x32)



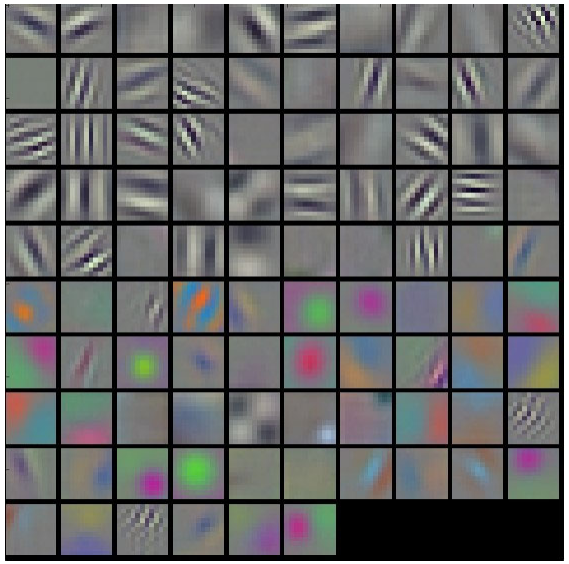
LeNet CNN

"Convolutional networks for images, speech, and time-series" Y. LeCun and Y. Bengio, The Handbook of Brain Theory and Neural Networks. MIT Press (1995)

Redes Neuronales Convolucionales (CNN)

- Propiedades:
  - El número de parámetros de la red crece exponencialmente
  - El ajuste suele presentar problemas de overfitting y requiere de un conjunto de entrenamiento muy grande
  - La función de activación de las capas de extracción de características se pueden representar como filtros de imagen:

Primera capa de convolución en AlexNet



Visualizador de capas Convolucionales (Microscope OpenAI)
https://microscope.openai.com

"ImageNet Classification with Deep Convolution" Alex Krizhevsky, Ilya Sutskever, Geoffrey E. Hinton, lournal of Computer and System Sciences 55 (1), (1997)

27 capas, filtros 3x3, 7x7
12M parámetros

19 capas, filtros 3x3
144M parámetros

7 capas, filtros 3x3, 5x5
60M parámetros

Image classification

Year	Codename	Error (percent)	99.9% Conf Int
2014	**GoogLeNet**	**6.66**	**6.40 - 6.92**
2014	VGG	7.32	7.05 - 7.60
2014	MSRA	8.06	7.78 - 8.34
2014	AHoward	8.11	7.83 - 8.39
2014	DeeperVision		
2013	Clarifai†		
2014	CASIAWS†		
2014	Trimps†	11.46	11.13 - 11.80
2014	Adobe†	11.58	11.25 - 11.91
2013	**Clarifai**	**11.74**	**11.41 - 12.08**
2013	NUS	12.95	12.60 - 13.30
2013	ZF		
2013	AHoward		
2013	OverFeat		
2014	Orange†	14.80	14.43 - 15.17
2012	SuperVision†	15.32	14.94 - 15.69
2012	**SuperVision**	**16.42**	**16.04 - 16.80**
2012	ISI	26.17	25.71 - 26.65
2012	VGG	26.98	26.53 - 27.43
2012	XRCE	27.06	26.60 - 27.52
2012	UvA	29.58	29.09 - 30.04

ImageNet Large Scale Visual Recognition Challenge
http://image-net.org/challenges/LSVRC/

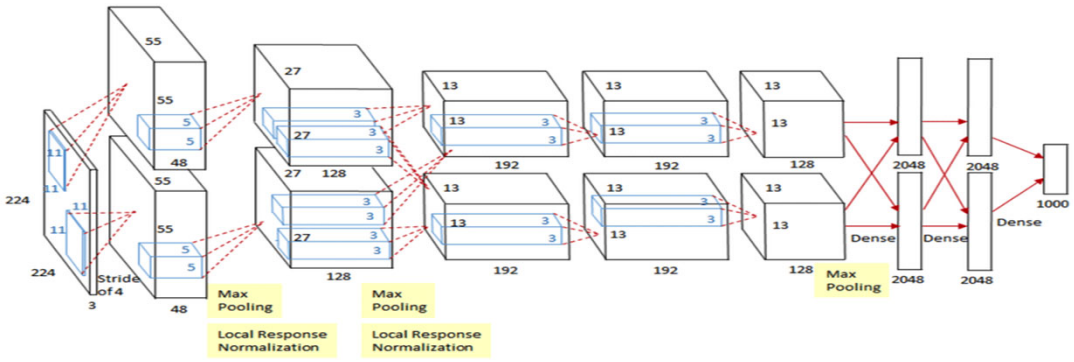
Simulador VGG16 (colab)

Redes Neuronales Convolucionales (CNN)

- ¿Qué aprende cada capa de estos modelos?
  - Capas Convoluciones Iniciales: Detectan formas básicas líneas diagonales, bordes y colores
  - Capas Convoluciones Medias: Combinan esas líneas para detectar texturas o partes de objetos (un círculo, una rejilla)
  - Capas Convoluciones Finales: extraen descriptores globales de la imagen. Detectan objetos complejos (un perro, una rueda, un rostro humano)
  - Capas Finales: Totalmente conectadas (MLP). Realizan la clasificación (asignación de patrones a clases) a partir de los descriptores

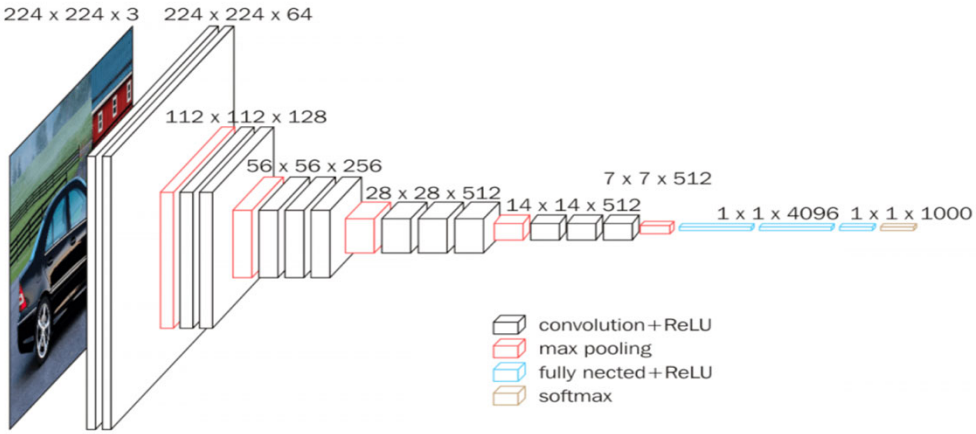
Redes Neuronales Convolucionales (CNN)

- **AlexNet (2012):** (8/24 capas - 60Millones parámetros)
  - Diseñada por Alex Krizhevsky, Ilya Sustkever y Geoffrey Hinton (Univ. Toronto)
  - Fue la primera en usar GPUs para el entrenamiento y la función de activación **ReLU**, que aceleró el aprendizaje



"ImageNet Classification with Deep Convolutional Neural Networks"
Alex Krizhevsky, Ilya Sutskever, Geoffrey E. Hinton
Communications of the ACM Volume 60 Issue 6 June 2017

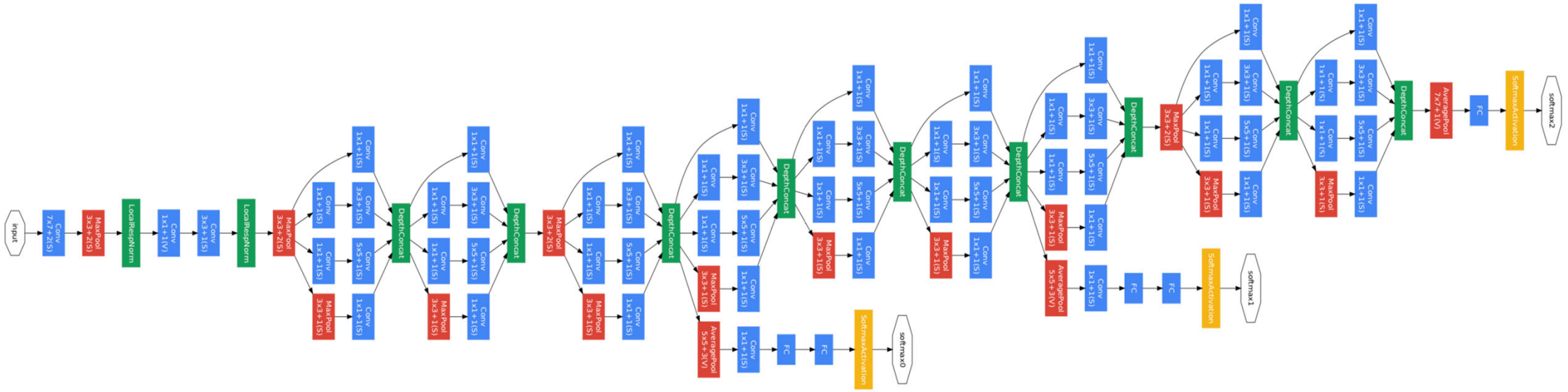
- **VGGNet (2014):** (16/19 capas - 138Millones parámetros)
  - Desarrollada por Karen Simonyan y Andrew Zisserman en el *Visual Geometry Group* (Univ. Oxford).
  - Su filosofía fue la simplicidad: usar filtros muy pequeños (3x3) pero apilarlos en una red muy profunda (hasta 19 capas)
  - Demostró que la profundidad (añadir más capas) era la clave para mejorar la precisión en la clasificación



"Very deep convolutional networks for large-scale image recognition"
Simonyan, K., & Zisserman, A, ICLR 2014

Redes Neuronales Convolucionales (CNN)

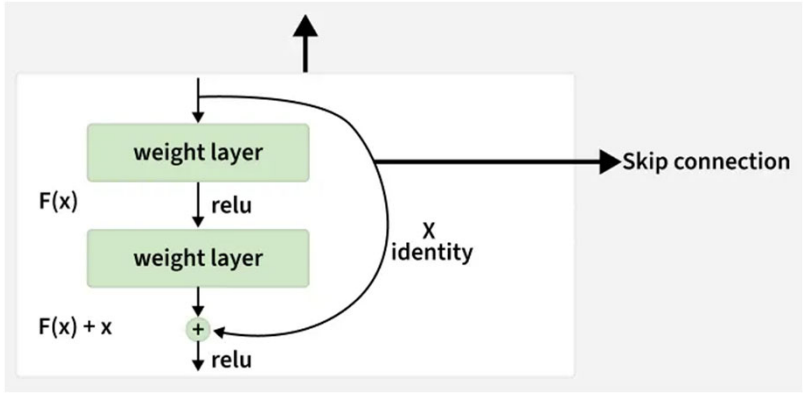
- **GoogLeNet / Inception (2014):** (22/144 capas - 12Millones parámetros)
  - Google Lab Introdujo los "*módulos Inception*", que permiten a la red elegir qué tamaño de filtro es mejor en cada etapa, procesando la información en paralelo.
  - Consiguió ser más profunda que AlexNet pero consumiendo mucha menos memoria y potencia de cómputo

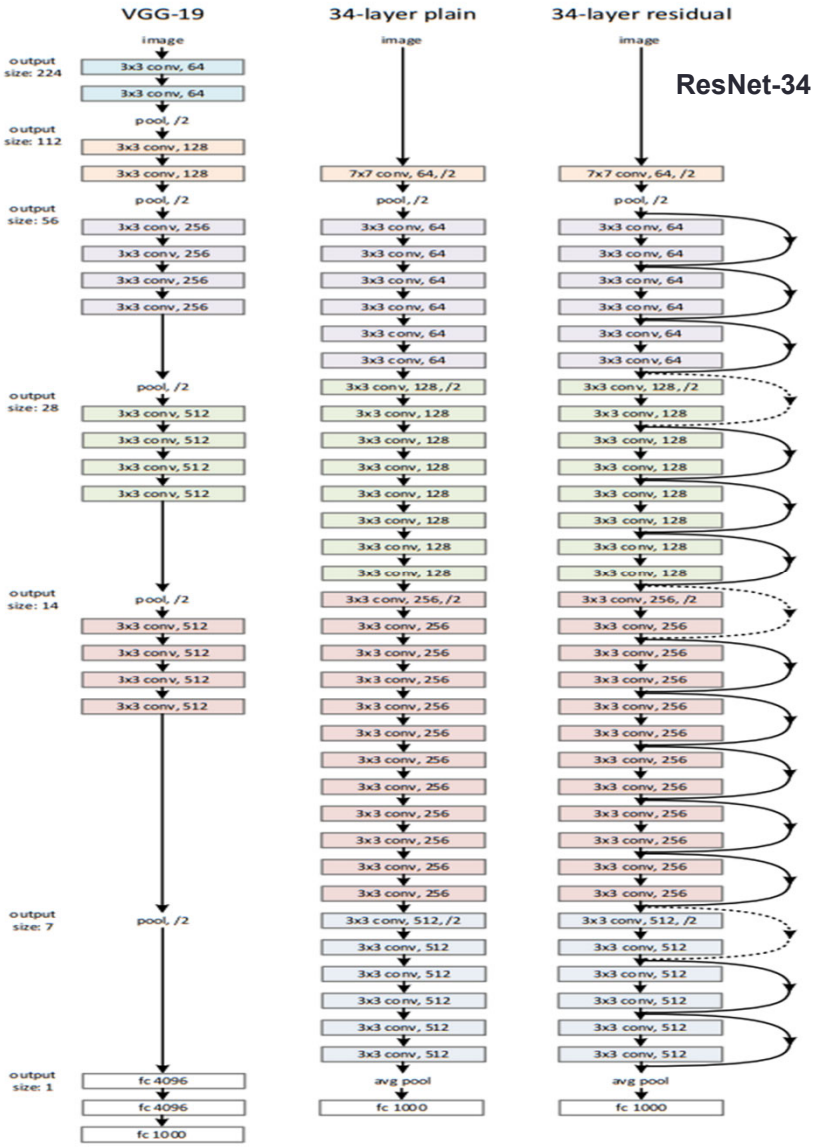


"Going Deeper with Convolutions" C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, A. Rabinovich
IEEE Conference on Computer Vision and Pattern Recognition (CVPR) 2015

Redes Neuronales Convolucionales (CNN)

- **ResNet (2015)** (34/152 capas - 22Millones parámetros)
  - A medida que las redes se hacían más profundas, empezaban a fallar (el gradiente se desvanecía en la **retropropagación**). Microsoft Research solucionó esto con las conexiones residuales o "saltos" (shortcuts)
  - Permitió crear redes de cientos de capas (como ResNet-152) que seguían aprendiendo de forma estable
  - ***Conexiones Residuales***: a la activación de una capa se le suman valores de entrada de capas previas.



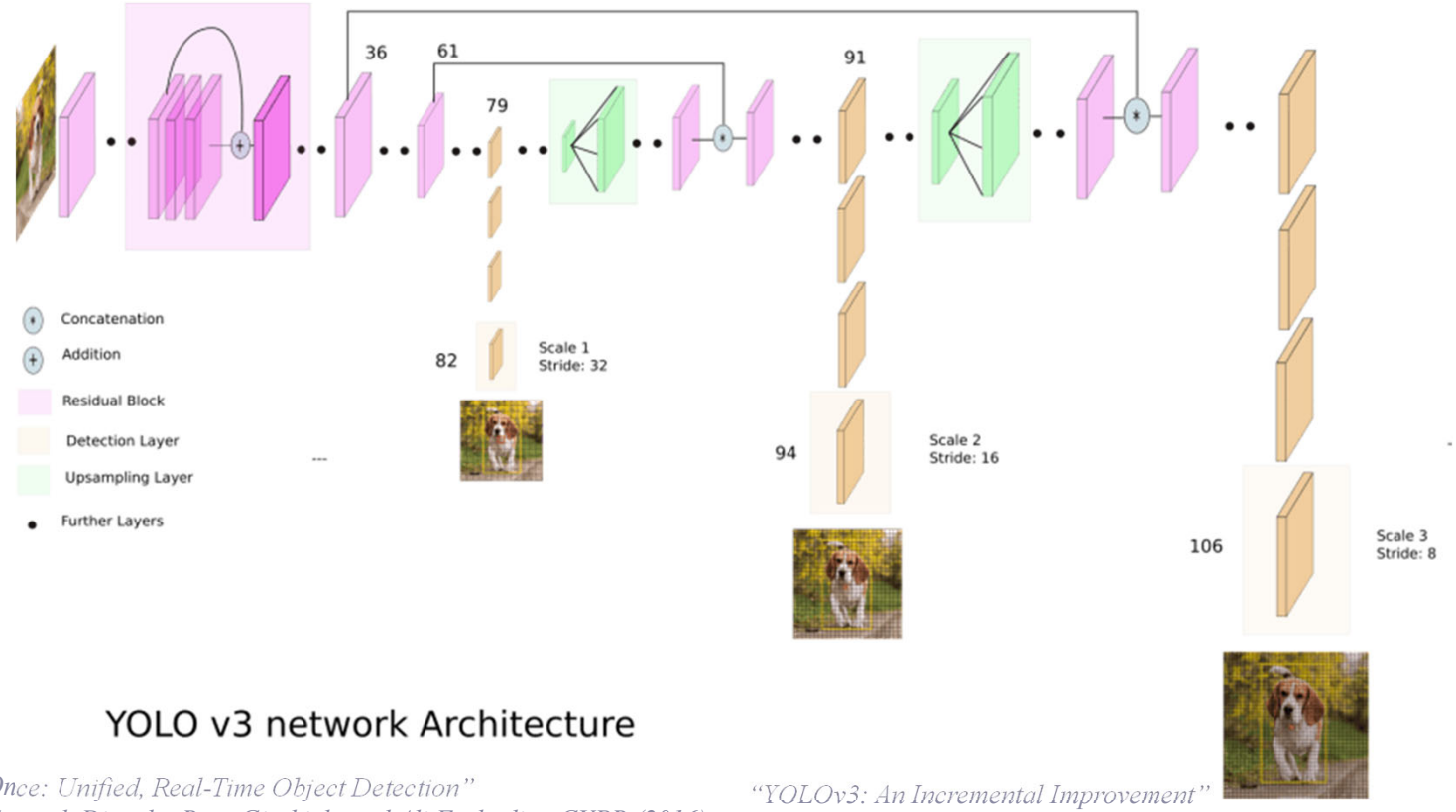


"Deep residual learning for image recognition" He, K., Zhang, X., Ren, S., & Sun, J. Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR) 2016

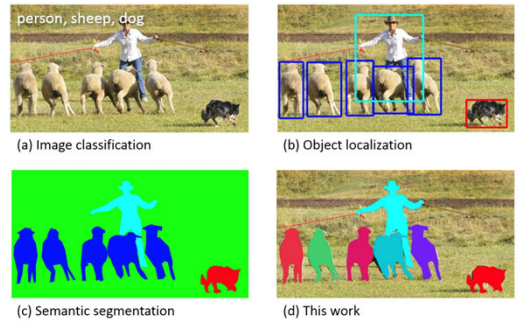
Redes Neuronales Convolucionales (CNN)

- YOLOv3 : (106 capas - 60Millones parámetros)
  - Reconocimiento y Detección de ventanas en la imagen (Univ. Washington / Microsoft Research)
    Entrenada con el dataset **MS COCO** (Common Objets in Context) 80 clases, etiquetado semántico de cada región

https://cocodataset.org



YOLO v3 network Architecture



"You Only Look Once: Unified, Real-Time Object Detection"
Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi CVPR (2016)

"YOLOv3: An Incremental Improvement"
Joseph Redmon, Ali Farhadi CVPR (2018)

Redes Neuronales Convolucionales (CNN)

- YOLO CNN Classifier

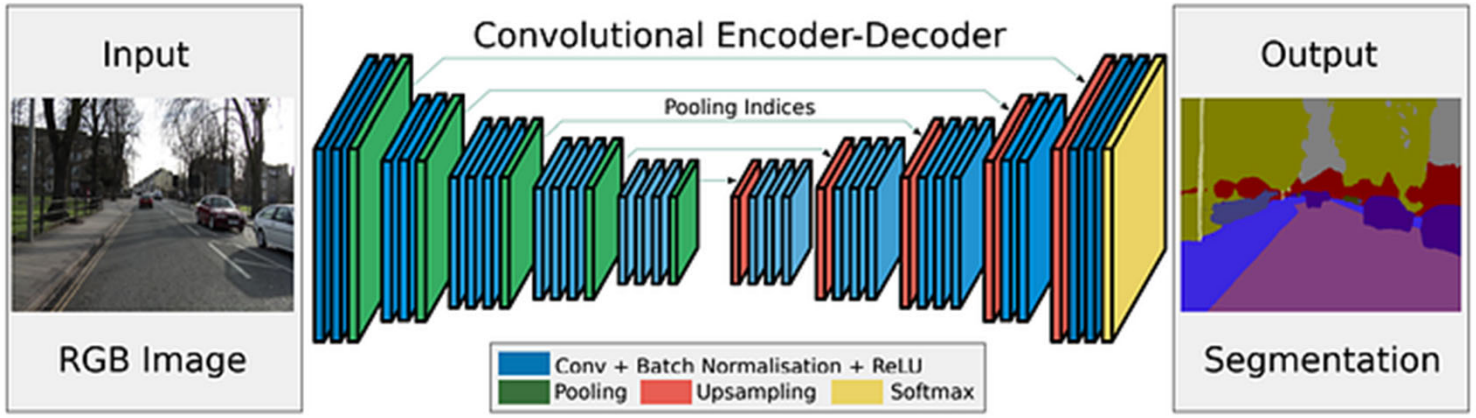
Video

"You Only Look Once: Unified, Real-Time Object Detection"
Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi CVPR (2016)

YOLO9000: Better, Faster, Stronger
Joseph Redmon, Ali Farhadi CVPR (2017)

Redes Neuronales Convolucionales (CNN)

- **SegNet (2015)** (26 capas - 30Millones parámetros)
  - Segmentación semántica: la salida es otra imagen en la que el valor de cada pixel corresponde a una etiqueta (clase) indicando a que objeto pertenece
  - Desarrollada por Badrinarayanan, Kendall y Cipolla en la Univ. Cambridge
  - Arquitectura ***Encoder-Decoder***:
    - **Encoder** (Codificador): Utiliza las primeras 13 capas convolucionales de VGG-16. Su función es extraer características de alto nivel y reducir la resolución de la imagen. Guarda los índices del Max-Pooling
    - **Decoder** (Decodificador): Es una copia "espejo" del codificador. Su función es mapear las características de baja resolución de vuelta al tamaño original de la imagen (***upsampling***). Utiliza los índices del Max-Pooling del encoder
    - **Capa Softmax Final**: Clasifica cada píxel individualmente en una de las categorías entrenadas

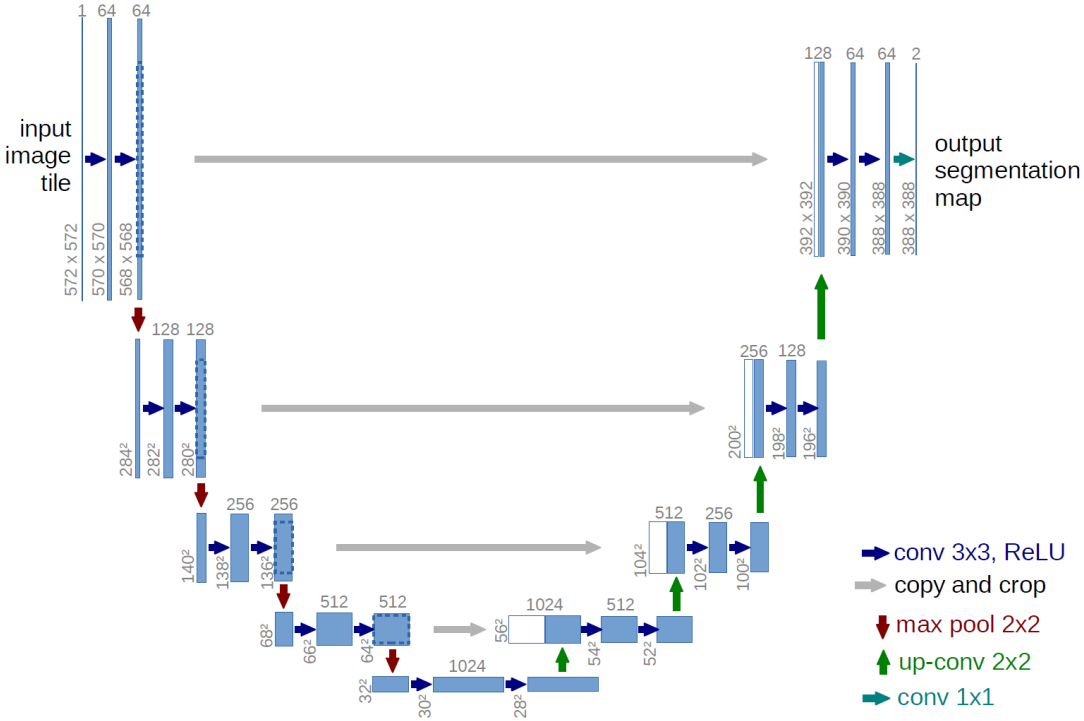


"SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation", Badrinarayanan, V., Kendall, A., & Cipolla, R. Transactions on Pattern Analysis and Machine Intelligence (PAMI) (2017)

Redes Neuronales Convolucionales (CNN)

- **U-Net (2015)** (23 capas - 31Millones parámetros)
  - Segmentación semántica en imágenes médicas
  - Objetivo: segmentar objetos con gran precisión cuando se tienen muy pocos datos de entrenamiento
  - Desarrollada por Ronneberger, Fischer y Broxen en la Univ. Friburgo
  - Arquitectura ***Encoder-Decoder U simétrica***:
    - **Encoder** (Codificador): parte izquierda de la "U", red convolucional similar a VGG para capturar las características de la imagen
    - **Decoder** (Decodificador): parte derecha de la "U". Recupera el tamaño original de la imagen para determinar la localización exacta de cada objeto
    - A diferencia de SegNet (que solo pasaba los índices del pooling), U-Net **copia y concatena los mapas de características completos** del encoder directamente al decoder

"U-Net: Convolutional networks for biomedical image segmentation", Ronneberger, O., Fischer, P., & Brox, T. Medical Image Computing and Computer-Assisted Intervention (MICCAI) (2015)



- **SAM2** (2024) (Segment Anything Model): (40-220Millones parámetros)
  - Red Híbrida Convolucional/Transformer. (Meta AI)

"SAM 2: Segment Anything in Images and Videos", Ravi, N., et al. European Conference on Computer Vision (ECCV) (2024)

Tabla de Contenidos

Estrategias de Entrenamiento de Redes CNN

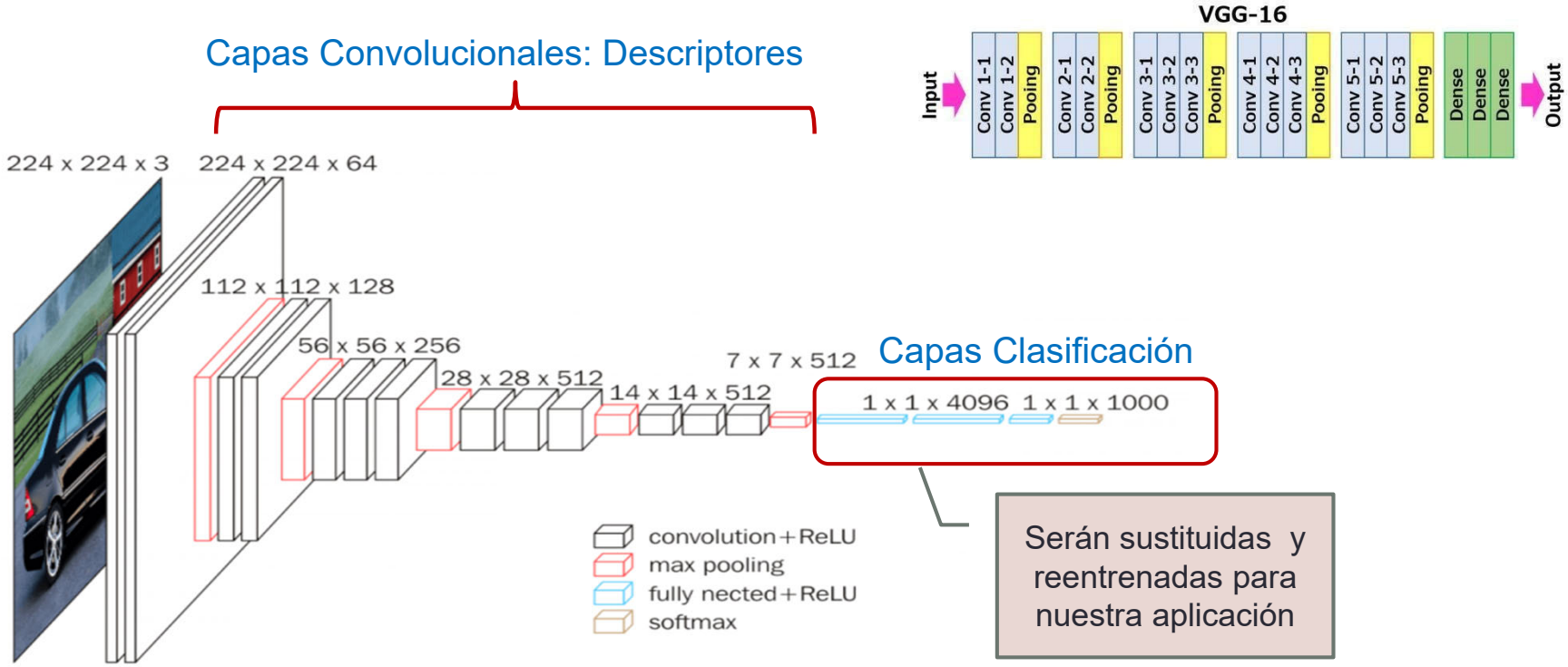
- Entrenamiento Completo:
  - Se parte de pesos aleatorios y se entrena toda la red neuronal
  - Precisa un conjunto de datos (‘dataset’) de entrenamiento muy grande para evitar **sobre-ajuste**
  - Número de iteraciones de entrenamiento ’epochs’ elevada para que el algoritmo SGD converja

- Entrenamiento Fino ‘Fine-tuning’:
  - Se parte de pesos previamente entrenados con un conjunto de datos general muy grande
  - Se reentrena toda la red o parte de ella partiendo de los pesos previos para ajustar la clasificación a imágenes de objetos específicos
  - Precisa un conjunto de datos de entrenamiento mas reducido

- Transferencia Conocimiento ‘Transfer Learning’:
  - Aprovecha los pesos ya entrenados de las capas convolucionales como extractor de características. Los pesos de estas capas quedan fijados
  - Se modifican y reentrenan las capas de clasificación finales para nuevos objetos

Transfer Learning (CNN)

- VGG16: (16 capas-138Millones parámetros) - input 3x224x224
  - Se trata de aprovechar los pesos ya entrenados de las capas convolucionales como extractor de características y modificar/reentrenar las capas de clasificación finales para nuevos objetos



"Very Deep Convolutional Networks for Large-Scale Image Recognition"
Karen Simonyan and Andrew Zisserman Univ. Oxford arXiv 1409.1556 ICLR 2015

Transfer Learning (CNN)

https://pytorch.org/vision/stable/models.html

```
Layer (type:depth-idx)                   Input Shape               Output Shape              Param #                   Trainable
==========================================================================================
VGG                                      [1, 3, 224, 224]          [1, 1000]                 --                        True
├─Sequential: 1-1                        [1, 3, 224, 224]          [1, 512, 7, 7]            --                        True
│    └─Conv2d: 2-1                       [1, 3, 224, 224]          [1, 64, 224, 224]         1,792                     True
│    └─ReLU: 2-2                         [1, 64, 224, 224]         [1, 64, 224, 224]         --                        --
│    └─Conv2d: 2-3                       [1, 64, 224, 224]         [1, 64, 224, 224]         36,928                    True
│    └─ReLU: 2-4                         [1, 64, 224, 224]         [1, 64, 224, 224]         --                        --
│    └─MaxPool2d: 2-5                    [1, 64, 224, 224]         [1, 64, 112, 112]         --                        --
│    └─Conv2d: 2-6                       [1, 64, 112, 112]         [1, 128, 112, 112]        73,856                    True
│    └─ReLU: 2-7                         [1, 128, 112, 112]        [1, 128, 112, 112]        --                        --
│    └─Conv2d: 2-8                       [1, 128, 112, 112]        [1, 128, 112, 112]        147,584                   True
│    └─ReLU: 2-9                         [1, 128, 112, 112]        [1, 128, 112, 112]        --                        --
│    └─MaxPool2d: 2-10                   [1, 128, 112, 112]        [1, 128, 56, 56]          --                        --
│    └─Conv2d: 2-11                      [1, 128, 56, 56]          [1, 256, 56, 56]          295,168                   True
│    └─ReLU: 2-12                        [1, 256, 56, 56]          [1, 256, 56, 56]          --                        --
│    └─Conv2d: 2-13                      [1, 256, 56, 56]          [1, 256, 56, 56]          590,080                   True
│    └─ReLU: 2-14                        [1, 256, 56, 56]          [1, 256, 56, 56]          --                        --
│    └─Conv2d: 2-15                      [1, 256, 56, 56]          [1, 256, 56, 56]          590,080                   True
│    └─ReLU: 2-16                        [1, 256, 56, 56]          [1, 256, 56, 56]          --                        --
│    └─MaxPool2d: 2-17                   [1, 256, 56, 56]          [1, 256, 28, 28]          --                        --
│    └─Conv2d: 2-18                      [1, 256, 28, 28]          [1, 512, 28, 28]          1,180,160                 True
│    └─ReLU: 2-19                        [1, 512, 28, 28]          [1, 512, 28, 28]          --                        --
│    └─Conv2d: 2-20                      [1, 512, 28, 28]          [1, 512, 28, 28]          2,359,808                 True
│    └─ReLU: 2-21                        [1, 512, 28, 28]          [1, 512, 28, 28]          --                        --
│    └─Conv2d: 2-22                      [1, 512, 28, 28]          [1, 512, 28, 28]          2,359,808                 True
│    └─ReLU: 2-23                        [1, 512, 28, 28]          [1, 512, 28, 28]          --                        --
│    └─MaxPool2d: 2-24                   [1, 512, 28, 28]          [1, 512, 14, 14]          --                        --
│    └─Conv2d: 2-25                      [1, 512, 14, 14]          [1, 512, 14, 14]          2,359,808                 True
│    └─ReLU: 2-26                        [1, 512, 14, 14]          [1, 512, 14, 14]          --                        --
│    └─Conv2d: 2-27                      [1, 512, 14, 14]          [1, 512, 14, 14]          2,359,808                 True
│    └─ReLU: 2-28                        [1, 512, 14, 14]          [1, 512, 14, 14]          --                        --
│    └─Conv2d: 2-29                      [1, 512, 14, 14]          [1, 512, 14, 14]          2,359,808                 True
│    └─ReLU: 2-30                        [1, 512, 14, 14]          [1, 512, 14, 14]          --                        --
│    └─MaxPool2d: 2-31                   [1, 512, 14, 14]          [1, 512, 7, 7]            --                        --
├─AdaptiveAvgPool2d: 1-2                 [1, 512, 7, 7]            [1, 512, 7, 7]            --                        --
├─Sequential: 1-3                        [1, 25088]                [1, 16]                   --                        True
│    └─Linear: 2-32                      [1, 25088]                [1, 4096]                 102,764,544               True
│    └─ReLU: 2-33                        [1, 4096]                 [1, 4096]                 --                        --
│    └─Dropout: 2-34                     [1, 4096]                 [1, 4096]                 --                        --
│    └─Linear: 2-35                      [1, 4096]                 [1, 4096]                 16,781,312                True
│    └─ReLU: 2-36                        [1, 4096]                 [1, 4096]                 --                        --
│    └─Dropout: 2-37                     [1, 4096]                 [1, 4096]                 --                        --
│    └─Linear: 2-38                      [1, 4096]                 [1, 1000]                 4,097,000                 True
```

Salida: torchinfo.**summary**(model)

```
def forward(self, x: torch.Tensor)
    x = self.features(x)
    x = self.avgpool(x)
    x = torch.flatten(x, 1)
    x = self.classifier(x)
    return x
```

Capas Convolucionales
Descriptores: submódulo 'features'

cambiaremos ***Trainable*** a False:

param.**requires_grad** = False

Serán sustituidas y reentrenadas para nuestra aplicación

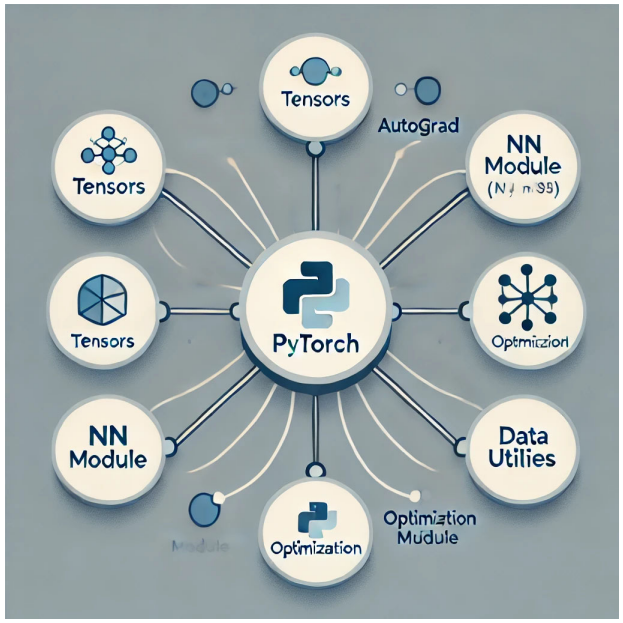
Tabla de Contenidos

Deep Learning (CNN) Pytorch

- Transfer Learning (PyTorch)
  - Importar redes pre-entrenadas
  - Clasificación: **VGG16**
  - Modificar/Añadir capas de salida y reentrenar
  - Librerías: **torch, torchvision, torchinfo**

```
import torch
import torchvision
import torchinfo
```

https://pytorch.org



torchvision: utilidades para Visión por Computador (CNNs)
datasets, modelos preentrenados, transformaciones de imágenes

Deep Learning (CNN) Pytorch

https://pytorch.org/vision/stable/models.html

- Redes CNN pre-entrenadas Clasificación
  - Modelos PyTorch: torchvision.models
    - AlexNet
    - ConvNeXt
    - DenseNet
    - EfficientNetV2
    - GoogleNet
    - Inception V3
    - MaxVit
    - MNASNet
    - MobileNet V2, V3
    - RegNet
    - ResNet 18/34/50/101
    - ResNeXt
    - ShuffleNet V2
    - SqueezNet
    - **VGG16** and VGG19
    - ViT (VisionTranformer)

se descargan desde el mismo código

```
# Option A) Build model and download weights  (base network)
model = torchvision.models.vgg16(weights=torchvision.models.VGG16_Weights.DEFAULT)
```

  - Datasets:
    - **ImageNet** Large Scale Visual Recognition Challenge (ILSVRC):
      - https://image-net.org/challenges/LSVRC/

Deep Learning (CNN) Pytorch

https://pytorch.org/docs/stable/nn.html#containers

- Crear Modelos NN: torch.nn
  - **Modelo Secuencial**: torch.nn.**Sequential**(**layers**=None) → model
    - Modelo simplificado para redes feed-forward. Cada capa con un solo **tensor** de entrada y de salida
    - Se pasan las capas como o una secuencia de parámetros, o bien un diccionario ordenado (**OrderedDict**) con el nombre y la capa

```
model = torch.nn.Sequential(
        torch.nn.Linear(in_features=1000, out_features=100),
        torch.nn.Tanh(),
        torch.nn.Linear(in_features=100, out_features=10),
        torch.nn.Softmax(dim=1)
)
```

  - **Modelo Funcional**: torch.nn.functional Clase Base: torch.nn.**Module**
    - Modelo general (Funcional), permite cualquier conexión (bifurcación, múltiples salidas o entradas) o topologías no lineales (Residual-realimentación, multi-rama, …)
    - La idea básica es crear la red como una clase derivada de la clase base torch.nn.**Module**.
    - Las capas/módulos se añaden como atributos de la clase en el constructor o bien se aplican directamente como funciones desde torch.nn.**functional** para capas simples sin pesos (activación)
    - Debemos definir el método **forward**() que implemente la inferencia del modelo (conexiones de capas)

PyTorch: Contenedores

https://pytorch.org/docs/stable/nn.html#containers

- Modelo Funcional: torch.nn.functional
  - Derivamos de la clase base torch.nn.**Module**
  - Definimos en el constructor las capas como atributos de la clase
  - El método **forward()** implementa la inferencia con las conexiones entre capas

```
import torch.nn.functional as F

class Net(torch.nn.Module):
        def __init__(self) -> None:
                super().__init__()          # call base class constructor

                self.layer1 = torch.nn.Linear(in_features=1000, out_features=100)
                self.activation1 = torch.nn.Tanh()
                self.layer2 = torch.nn.Linear(in_features=100, out_features=10)
                self.activation2 = torch.nn.Softmax(dim=1)

        def forward(self, x):
                x = self.activation1(self. layer1(x))
                x = self.activation2(self. layer2(x))
                return x

model = Net()
```

PyTorch: Contenedores

https://pytorch.org/docs/stable/nn.html

- Clase base: torch.nn.**Module**
  - El resto de modelos/capas deriva de esta clase
  - Atributos:
    - **training** → (bool) indica si el modelo esta en modo de entrenamiento o inferencia
  - Métodos:
    - **parameters**() → *Iterator[Parameter]. Parámetros/pesos de todos las capas del modelo*
    - **modules**() → *Iterator[Module]*
    - **named_modules**() → *Iterator[(str,Module)]*
    - **forward**(*input) calcula la inferencia del modelo
    - **train**() modo entrenamiento
    - **eval**() modo evaluación/inferencia
    - **type**(dst_type) cambia el formato de los parámetros
    - **to**() mueve parámetros a un dispositivo
    - **requires_grad_**(bool) set ***requires_grad*** for parameters (activa/desactiva entrenamiento)
    - **zero_grad_**(bool) limpia gradientes acumulados
    - **load_state_dict**(state_dict) carga pesos (parámetros) en el modelo
  - Parámetros (pesos): torch.nn.parameter.Parameter
    - **data (Tensor)**
    - **requires_grad** → (bool) Entrenable (True)

PyTorch: Contenedores

https://pytorch.org/docs/stable/cuda.html

- Aceleración GPU: torch.device, torch.cuda
  - Tipos aceleradores: CUDA, DML, MTIA, MPS, XPU
  - Testeo disponibilidad de aceleración por GPU:
    - "cuda" (NVIDIA GPU) torch.cuda.**is_available**()
    - Si disponemos de varias tarjetas podemos indicar el dispositivo añadiendo el numero: "cuda:0", "cuda:1", …
  - Para crear un dispositivo disponemos de la clase: torch.device(name) → device
  - Para indicar el dispositivo que usará para ejecutar un modelo disponemos del método en la clase torch.nn.Module:
    - torch.nn.Module.**to**(device)
    - torch.nn.Module.**cuda()**
    - torch.nn.Module.**cpu()**
  - Es importante ejecutarlo después de cualquier modificación del modelo

```
DEVICE = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("GPUs available:", torch.cuda.device_count())
for i in range(torch.cuda.device_count()):
        print(f"GPU: {torch.cuda.get_device_name(i)}, Type: {torch.cuda.get_device_capability(i)}")
model.to(DEVICE)
```

PyTorch: Tensores

https://pytorch.org/docs/stable/tensors.html

- Clase: torch.**Tensor**
  - Los datos que se procesan en una red se manejan como un tensor (array multidimensional con un álgebra asociada)
  - Formato por defecto para imágenes: 'channels_first'
    - (batch, channels, height, width)
  - Se disponen de funciones de creación y conversión:
    - torch.**tensor**(data) → (Tensor)
    - torch.**rand**() → (Tensor)
    - torch.**from_numpy**(np_array) → (Tensor)
    - …..
  - Atributos:
    - **shape** → torch.Size
    - **dtype**
    - **device**
    - **grad** → gradientes
    - **T** → transpuesta
    - **H** → conjugada transpuesta
    - …..
  - Métodos:
    - Operaciones aritméticas y transcendentales, algebra matricial
    - **transpose**() → *tensor transpuesto*
    - **histogram**() → *histograma del tensor*
    - **argmax**() → *matriz categórica a vector*
    - **type_as**(dst_type) cambia el formato del tensor
    - **reshape**() → *tensor redimensionado*
    - **view**() → *tensor redimensionado (sin copias)*
    - **to**() → *tensor* mueve una copia del tensor a un dispositivo
    - **to_**() mueve tensor a un dispositivo
    - …..

PyTorch: Layers

https://pytorch.org/docs/stable/nn.html

- Capas: torch.nn clase base: torch.nn.**Module**
  - Convolution Layers
  - Pooling layers
  - Padding Layers
  - Non-linear Activations (weighted sum, nonlinearity)
  - Non-linear Activations (other)
  - Normalization Layers
  - Recurrent Layers
  - Transformer Layers
  - Linear Layers
  - Dropout Layers
  - Sparse Layers
  - Distance Functions
  - Loss Functions
  - Vision Layers
  - Shuffle Layers
  - DataParallel Layers (multi-GPU, distributed)

- Capas Lineales:
  (totalmente conectadas)
  - torch.nn.**Linear** $y = x \cdot W^t + b$
  - torch.nn.**Bilinear** $y = x_1^T W\, x_2 + b.$
  - torch.nn.**LazyLinear**
  - torch.nn.**Identity** $\rightarrow$ $y = x$

torch.nn.**Linear**(**in_features**, **out_features**, bias=True, device=None, dtype=None)

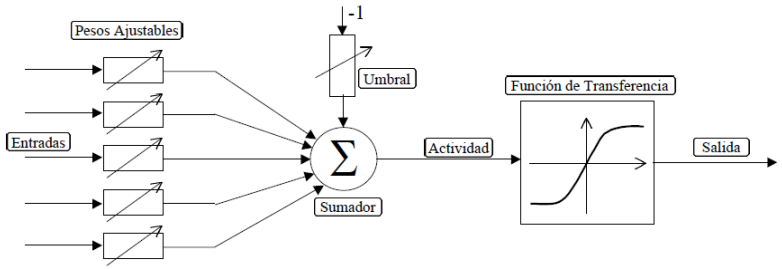
torch.nn.**LazyLinear** → no requiere configurar el tamaño de la entrada (se detecta con la primera entrada usada)

PyTorch: Layers

https://pytorch.org/docs/stable/nn.html

- Capas de Activación: torch.nn

torch.nn.**AdaptiveLogSoftmaxWithLoss**
torch.nn.**ELU**
torch.nn.**Hardshrink**
torch.nn.**Hardsigmoid**
torch.nn.**Hardtanh**
torch.nn.**Hardswish**
torch.nn.**LeakyReLU**
torch.nn.**LogSigmoid**
torch.nn.**LogSoftmax**
torch.nn.**MultiheadAttention**
torch.nn.**PReLU**
torch.nn.**ReLU**
torch.nn.**ReLU6**
torch.nn.**RReLU**
torch.nn.**SELU**
torch.nn.**CELU**
torch.nn.**GELU**
torch.nn.**Sigmoid**
torch.nn.**SiLU**
torch.nn.**Mish**
torch.nn.**Softmin**
torch.nn.**Softmax**
torch.nn.**Softmax2d**
torch.nn.**Softplus**
torch.nn.**Softshrink**
torch.nn.**Softsign**
torch.nn.**Tanh**
torch.nn.**Tanhshrink**
torch.nn.**Threshold**
torch.nn.**GLU**



Sigmoid
$\sigma(x) = \frac{1}{1+e^{-x}}$

Leaky ReLU
$\max(0.1x, x)$

tanh
$\tanh(x)$

Maxout
$\max(w_1^T x + b_1, w_2^T x + b_2)$

ReLU
$\max(0, x)$

ELU
$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$

$$softmax:\ p(i) = \frac{e^{z(i)}}{\sum_j e^{z(j)}}$$

$$\text{selu}\,(z) = \begin{cases} s * x & si \quad x \geq 0 \\ s * \alpha(e^x - 1) & si \quad x < 0 \end{cases}$$

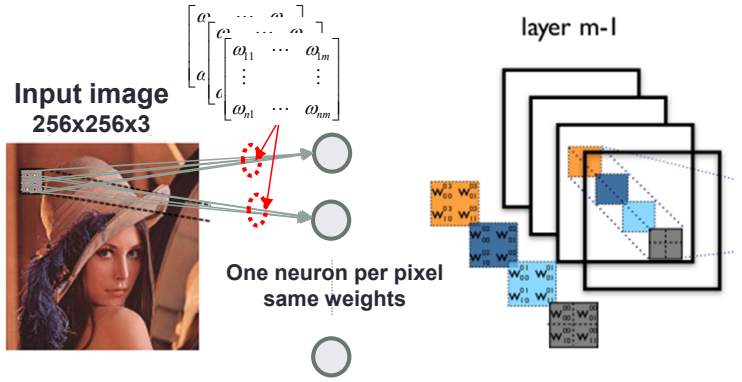
$$softplus = log(e^x + 1)$$

$$softsign = \frac{x}{|x| + 1} \qquad exponetial = e^x \qquad \text{sgn}(z) = \begin{cases} +1 & si \quad x > 0 \\ 0 & si \quad x = 0 \\ -1 & si \quad x < 0 \end{cases}$$

PyTorch: Layers

https://pytorch.org/docs/stable/nn.html

- Capas Convolucionales: torch.nn
  - torch.nn.**Conv1d**
  - torch.nn.**Conv2d**
  - torch.nn.**Conv3d**
  - torch.nn.**ConvTranspose1d**
  - torch.nn.**ConvTranspose2d**
  - torch.nn.**ConvTranspose3d**
  - torch.nn.**LazyConv1d, 2d, 3d**
  - torch.nn.**LazyConvTranspose1d, 2d, 3d**
  - torch.nn.**Fold**
  - torch.nn.**Unfold**



$$\text{out}(N_i, C_{\text{out}_j}) = \text{bias}(C_{\text{out}_j}) + \sum_{k=0}^{C_{\text{in}}-1} \text{weight}(C_{\text{out}_j}, k) \star \text{input}(N_i, k)$$

torch.nn.**Conv2d**(**in_chanels**, **out_channels**, **kernel_size**, stride=1, padding=0, dilation=1, groups=1, bias=True, paddind_mode='zeros', device=None, dtype=None)

in_chanels : canales de entrada: numero de máscaras de convolución para cada neurona

out_chanels : numero de conjunto de máscaras (pesos) distintas: canales de la capa
kernel_size: int or tuple (h,w) tamaño de la matriz de pesos
stride : int or tuple (1, 1): paso desplazamiento de la convolución en altura y anchura por la imagen
padding: int or {'valid', 'same'} *valid*: elimina pixels de borde (mascara fuera de la imagen)
same: rellena con ceros los pixels de la máscara fuera de la imagen

dilation : int or tuple (1, 1): espacios entre pixels a los que se aplica la convolución
groups=1: grupos del canales de la imagen de la entrada que se procesan separadamente
padding_mode : 'zeros', 'reflect', 'replicate' or 'circular'. Default: 'zeros'
dtype: tipo de los pesos Orden de las dimensiones de la entrada y salida: **(n, c, h, w)**

PyTorch: Layers

https://pytorch.org/docs/stable/nn.html

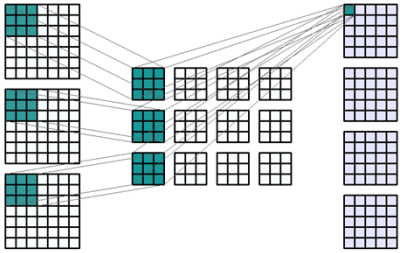
- Capas Convolucionales: torch.nn
  - torch.nn.**Conv2d**

$$\text{out}(N_i, C_{\text{out}_j}) = \text{bias}(C_{\text{out}_j}) + \sum_{k=0}^{C_{\text{in}}-1} \text{weight}(C_{\text{out}_j}, k) \star \text{input}(N_i, k)$$

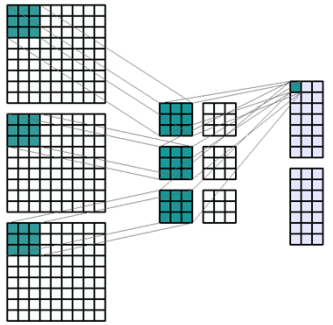
torch.nn.**Conv2d**(**in_chanels**, **out_channels**, **kernel_size**, stride=1, padding=0, dilation=1, groups=1, bias=True, paddind_mode='zeros', device=None, dtype=None)

stride : int or tuple (1, 1): paso desplazamiento de la convolución en altura y anchura por la imagen
padding: int or {'valid', 'same'} *valid*: elimina pixels de borde (mascara fuera de la imagen)
same: rellena con ceros los pixels de la máscara fuera de la imagen
dilation : int or tuple (1, 1): espacios entre pixels a los que se aplica la convolución
groups=1: grupos del canales de la imagen de la entrada que se procesan separadamente
padding_mode : 'zeros', 'reflect', 'replicate' or 'circular'. Default: 'zeros'

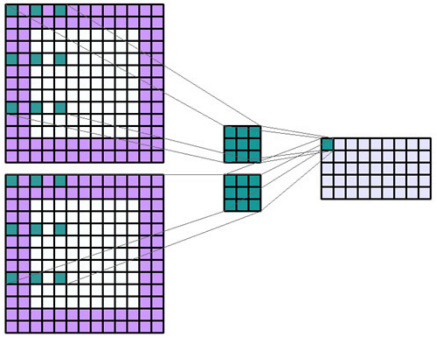
stride=(1,1)
dilation=(1,1)



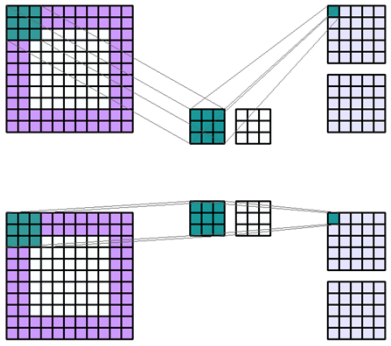
stride=(1,3)
dilation=(1,1)



dilation=(4,2)
stride=(1,1)



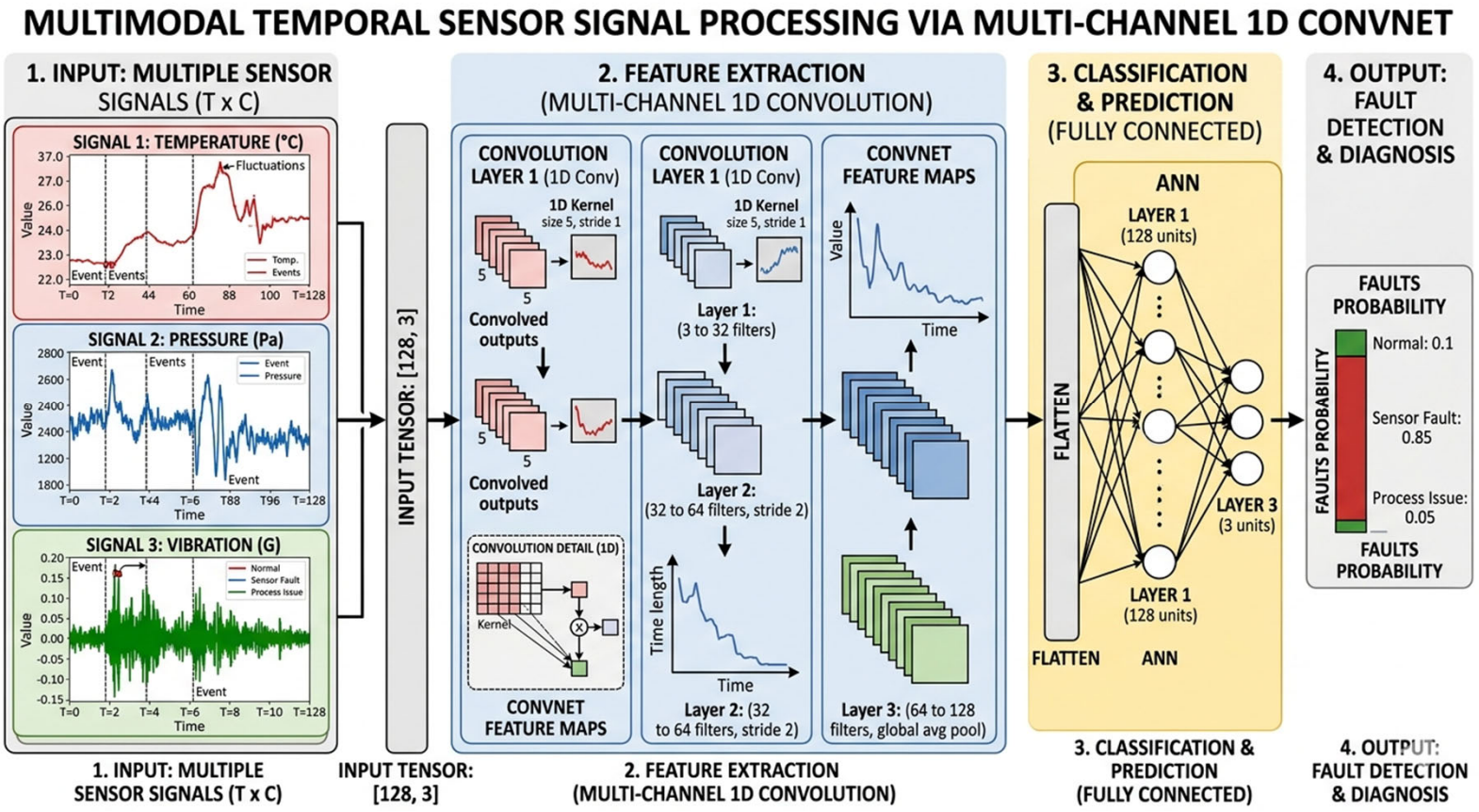
groups=2
stride=(1,1)
dilation=(1,1)



PyTorch: Layers

https://pytorch.org/docs/stable/nn.html

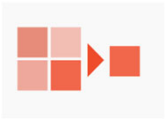
- Capas Conv1d:

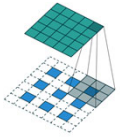


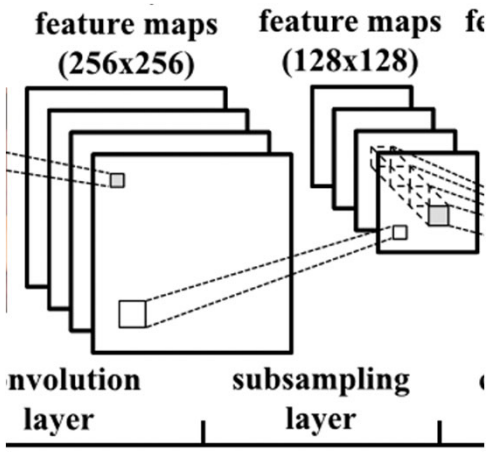
PyTorch: Layers

https://pytorch.org/docs/stable/nn.html

- Capas de Muestreo: torch.nn
  - torch.nn.**MaxPool1d**
  - torch.nn.**MaxPool2d**
  - torch.nn.**MaxPool3d**
  - torch.nn.**AvgPool1d**
  - torch.nn.**AvgPool2d**
  - torch.nn.**AvgPool3d**
  - torch.nn.**AdaptativeMaxPool1d, 2d, 3d**
  - torch.nn.**AdaptativeAvgPool1d, 2d, 3d**
  - torch.nn.**FractionalMaxPool2d, 3d**
  - torch.nn.**LPPool1d, 2d, 3d**
  - torch.nn.**MaxUnpool1d, 2d, 3d**







torch.nn.**MaxPool2d**(kernel_size=(2, 2), stride=None, padding=0, dilation=1,
return_índices=False, ceil_mode=False)

kernel_size int or tuple (2, 2): ventana en la que calcular el máximo (muestreo vertical y horizontal)

stride : int or tuple (1, 1): paso desplazamiento del muestreo por la imagen, altura y anchura

padding: int or tuple (0, 0): elimina pixels borde / añade pixel de borde (ceros)

dilation : int or tuple (1, 1): espacios entre pixels a los que se aplica el muestreo

return_indices : (bool) almacena los índices del máximo seleccionado.
Usados en **MaxUnpool2d / ConvTranspose2d**

ceil_mode : (bool). Uso de ***ceil/floor*** para calcular el tamaño de la salida

PyTorch: Layers

https://pytorch.org/docs/stable/nn.html

- Capas Realimentadas (Recurrentes):
  - torch.nn.**RNNBase** **clase base**
  - torch.nn.**RNN**
  - torch.nn.**LSTM**
  - torch.nn.**GRU**
  - torch.nn.**RNNCell**
  - torch.nn.**LSTMCell**
  - torch.nn.**GRUCell**
- Capas Transformer:
  - torch.nn.**Transformer** (Modelo)
  - torch.nn.**TransformerEncoderLayer**
  - torch.nn.**TransformerDecoderLayer**

- Capas Regularización:
  (solo actúan en el entrenamiento)
  - torch.nn.**Dropout**
  - torch.nn.**Dropout1D**
  - torch.nn.**Dropout2D**
  - torch.nn.**Dropout3D**
  - torch.nn.**AlphaDropout**
  - torch.nn.**FeatureAlphaDropout**
- Capas “Sparse”:
  - torch.nn.**Embedding**
  - torch.nn.**EmbeddingBag**

- torch.nn.**Dropout**(**p**=0.5, inplace=False)

  Pone aleatoriamente entradas (salida neuronas capa previa) a **0** con probabilidad **p** en cada paso de la fase de entrenamiento. Reescala el resto para que la suma se mantenga.

  Permite evitar el Sobreajuste ‘*Overfitting*’

  inplace==True modifica directamente el tensor de entrada sin crear un nuevo tensor

  inplace==False crea un nuevo tensor de salida que es el que modifica

PyTorch: Layers

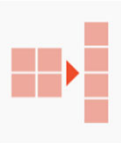
https://pytorch.org/docs/stable/nn.html

- Capas: torch.nn
  - Capas Normalización:
    - torch.nn.**BatchNorm1d, 2d, 3d**
    - torch.nn.**LazyBatchNorm1d, 2d, 3d**
    - torch.nn.**GroupNorm**
    - torch.nn.**SyncBatchNorm**
    - torch.nn.**InstanceNorm1d, 2d, 3d**
    - torch.nn.**LazyInstanceNorm1d, 2d, 3d**
    - torch.nn.**LayerNorm**
    - torch.nn.**LocalResponseNorm**
    - torch.nn.**RMSNorm**
  - Capas Relleno Contornos:
    - torch.nn.**ReflectionPad1d, 2d, 3d**
    - torch.nn.**ReplicationPad1d, 2d, 3d**
    - torch.nn.**ZeroPad1d, 2d, 3d**
    - torch.nn.**ConstantPad1d, 2d, 3d**
    - torch.nn.**CircularPad1d, 2d, 3d**

nn.BatchNormXd(n): Normalizes a X-dimensional input batch with n features; X ∈ {1, 2, 3}

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

  - Funciones Distancia:
    - torch.nn.**CosineSimilarity**
    - torch.nn.**PariwiseDistance**
  - Capas Redimensionado:
    - torch.nn.**Flatten**
    - torch.nn.**Unflatten**
    - torch.nn.**PixelShuffle**
    - torch.nn.**PixelUnshuffle**
    - torch.nn.**Unsample**
    - torch.nn.**UnsampleNearest2d**
    - torch.nn.**UnsampleBilinear2d**
    - torch.nn.**ChannelShuffle**



- torch.nn.**Flatten**(star_dim=1, end_dim=-1)

  Convierte capas convolucionales en vectoriales (planas)

PyTorch: Optimizadores

https://pytorch.org/docs/stable/optim.html

- Configuración modelo:
  - Seleccionar el tipo de función de coste (‘***loss***’) y el optimizador
  - Clase base optimizadores: torch.optim.Optimizer(params, defaults)
  - Atributos:
    - **params** → (iterator) indica los tensores que deben ser optimizados
    - **defaults** → opciones optimizador
  - Optimizadores:
    - torch.optim.**SGD** Stochastic Gradient Descent
    - torch.optim.**Adadelta** (SGD-based) Adaptative learning rate per dimensión
    - torch.optim.**Adafactor** (SGD-based) Adaptative learning rate sublinear memory cost
    - torch.optim.**Adagrad** (SGD-based) Adaptative Gradient Algorithm
    - torch.optim.**Adam** (SGD) Adaptative estimation of first-order and second-order moments
    - torch.optim.**Adamax** ADAM with infinite norm (max)
    - torch.optim.**RMSprop** Root Mean Square Propagation
    - torch.optim.**NAdam** Nesterov-accelerated Adaptive Moment Estimation,
    - Otros: **AdamW, SparseAdam , ASGD**, **LBFGS** , **RAdam, Rprop**

$$\omega_{t+1} = \omega_t - \eta \nabla L(\omega)$$

$$\nabla L(\omega) = \beta\, \nabla L(\omega_{t-1}) + \nabla L(\omega_t)$$

$\eta \rightarrow learning\ rate$

$\beta \rightarrow momentum$

```
optimizer = torch.optim.SGD(model.parameters(), lr=0.01, momentum=0.1)
```

PyTorch: Funciones de Pérdida 'loss'

- Funciones de coste entrenamiento: 'loss'

torch.nn.**L1Loss**
torch.nn.**MSELoss**

Regression losses

$$\ell(x,y) = L = \{l_1,\dots,l_N\}^\top, \quad l_n = |x_n - y_n|,$$
$$\ell(x,y) = L = \{l_1,\dots,l_N\}^\top, \quad l_n = (x_n - y_n)^2,$$

torch.nn.**CrossEntropyLoss**
torch.nn.**CTCLoss**
torch.nn.**NLLLoss**
torch.nn.**PoissonNLLLoss**
torch.nn.**GaussianNLLLoss**
torch.nn.**KLDivLoss**
torch.nn.**BCELoss**
torch.nn.**BCEWithLogitsLoss**
torch.nn.**MarginRankingLoss**
torch.nn.**HingeEmbeddingLoss**
torch.nn.**MultiLabelMarginLoss**
torch.nn.**HuberLoss**
torch.nn.**SmoothL1Loss**
torch.nn.**SoftMarginLoss**
torch.nn.**MultiLabelSoftMarginLoss**
torch.nn.**CosineEmbeddingLoss**
torch.nn.**MultiMarginLoss**
torch.nn.**TripletMarginLoss**
torch.nn.**TripletMarginWithDistanceLoss**

Probabilistic losses

$$\ell(x,y) = L = \{l_1,\dots,l_N\}^\top, \quad l_n = -\sum_{c=1}^{C} w_c \log \frac{\exp(x_{n,c})}{\sum_{i=1}^{C}\exp(x_{n,i})} y_{n,c}$$

```
criterion = torch.nn.CrossEntropyLoss()
```

PyTorch: Modelos

https://pytorch.org/docs/stable/nn.html

- Desensamblado: torch.nn.**Module**
  - torch.nn.Module.**modules()** → *Iterator[Module]*
  - torch.nn.Module.**named_modules()** → *Iterator[(name,Module)]*
  - torch.nn.Module.**parameters()** → *Iterator[Parameter].*

  - Los módulos/capas con etiqueta pueden ser accedidas como un atributo del objeto
  - Se puede acceder también a las capas indexando el objeto torch.nn.**Module**
  - Nos permitirá modificar una red previa o evaluar la respuesta de una capa
  - Borrado de capas:
    - Modelos Secuenciales: **del** model[idx]
    - Modelos funcionales: Editar el método **forward**() o bien sustituirla por torch.nn.**Identity**()

```
# Disassemble model
modules = dict( [(name, module) for name, module in model.named_modules()] )

# first module is the main model
print("Model Layers")

for i in range(len(modules)-1):
        print(f"- {i}: {model[i]}")
```

Transfer Learning(CNN)

https://pytorch.org/vision/stable/datasets.html

- Cargar Datos Entrenamiento: torchvision.datasets.**ImageFolder**
  - Clase derivada de torchvision.datasets.**DatasetFolder**
    Para datasets especiales: debemos definir la función de lectura de datos
  - torchvision.datasets.**ImageFolder**(**root**, **transform**=None, **target_transform**=None, **allow_empty**=False) → **torchvision.datasets.ImageFolder**

    Crea un objeto **torchvision.datasets.ImageFolder** a partir de un directorio de imágenes. No se cargan inmediatamente las imágenes en memoria, solo es un listado con el path de cada imagen y sus etiquetas. Configura la transformación de preprocesamiento si se indica. Cuando se indexa el objeto es cuando se carga y aplica la transformación: img, label = dataset[i]

    root: Directorio raíz. Los subdirectorios de primer nivel son los nombres de las clases.
    transform: transformaciones de preprocesamiento para la imagen
    target_transform: transformaciones para la etiqueta (por defecto enteros **labels:** 0…n)
    allow_empy: True: un directorio vacío se considera clase válida

    ```
    root/
    ...class_a/
    ......a_image_1.jpg
    ......a_image_2.jpg
    ...class_b/
    ......b_image_1.jpg
    ......b_image_2.jpg
    ```

  - Propiedades:
    class_to_idx: lista con nombres de clase asociada a índices 0..n
  - Disponemos también de cargadores de imágenes de Datasets públicos: se detalla su uso al final de este tutorial

Transfer Learning(CNN)

https://pytorch.org/docs/stable/data.html

- Dividir el dataset: entrenamiento / validación
  - torch.utils.data.**random_split**(dataset, lengths) → (dataset1, dataset2)
    - Divide de forma aleatoria un **dataset** en dos nuevos datases no solapados

      lengths: [dataset1_size, datset2_size] dimesiones o procentajes

- Crear Cargador de Datos: clase torch.utils.data.**DataLoader**
  - La clase **DataLoader** gestiona la lectura de datos a partir del **dataset** cargado con **ImageFolder** y nos permite distribuir los datos en lotes de entrenamiento asignándolos aleatoriamente en cada iteración de lectura de completa de los datos (*shuffe*)
  - torch.utils.data.**DataLoader**(dataset, batch_size=1, shuffle=False) → DataLoader

    dataset (Dataset)
    batch_size (int, optional) – muestras por lote
    shuffle (bool, optional) True: se barajan aleatoriamente los datos de nuevo en cada época
  - El objeto **DataLoader** es iterable y devuelve una tupla de tensores con un lote de datos

Transfer Learning(CNN)

https://pytorch.org/docs/stable/data.html

- Cargar Datos Entrenamiento (Resumen)
  - Crear el objeto **Dataset**, dividir los conjuntos de entrenamiento y validación y crear los objetos **DataLoader** para asignar lotes y lectura aleatoria**:**
    - torchvision.datasets.**ImageFolder**
    - torch.utils.data.**random_Split**
    - torch.utils.data.**DataLoader**

```
# Load image dataset (list of image files)
full_dataset = torchvision.datasets.ImageFolder(root="../images/", transform=preprocess_vgg16)

# extract labels idx  dictionary: (label -> idx)
class_names = full_dataset.class_to_idx
print(f"ClassNames ({len(class_names)}):", class_names)


# split train/validation subsets are simple references to base full_dataset
data_size = len(full_dataset)
train_size = int(0.8 * data_size)        # 80% train
test_size = data_size - train_size       # 20% validation

train_dataset, val_dataset = torch.utils.data.random_split(full_dataset, [train_size, test_size])

# Data Loaders
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=10, shuffle=True)
val_loader = torch.utils.data.DataLoader(val_dataset, batch_size=10, shuffle=False)
```

```
root/
...class_a/
......a_image_1.jpg
......a_image_2.jpg
...class_b/
......b_image_1.jpg
......b_image_2.jpg
```

Transfer Learning (CNN)

https://pytorch.org/docs/stable/nn.html

- Entrenamiento: torch_util.**trainModel()**
  - Implementa el bucle de entrenamiento: **epoch** (iteración completa de actualización de los pesos con todos los datos de entrenamiento)
  - **Proceso**: 1) leer lotes de imágenes, 2) calcular la inferencia, 3) calcular función de pérdida, 4) retro-propagar los gradientes de la función de pérdida, 5) actualizar los parámetros (pesos) del modelo, 6) calcular métricas de la iteración (loss/accuracy).

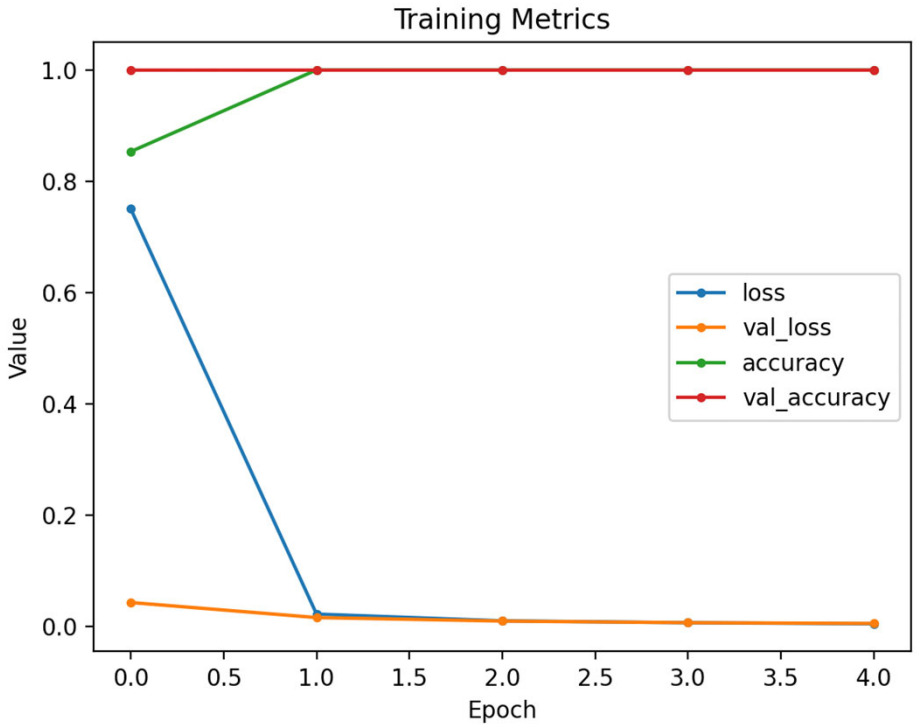
```
from torch_util import trainModel, predictModel, plotLearningCurves

# Configure Loss function and optimizer
criterion = torch.nn.CrossEntropyLoss()
# SGD  optimizer  lr: Learning Rate, momentum: gradient inertia
optimizer = torch.optim.SGD(model.parameters(), lr=0.01, momentum=0.5)

history = trainModel( model=model, train_loader=train_loader, val_loader=val_loader,
                      criterion=criterion, optimizer=optimizer, epochs=5, device=DEVICE )
```

```
Training Model:
Epoch [1/5] - Train Loss: 0.7504, Train Acc: 0.8533, Val Loss: 0.0432, Val Acc: 1.0000
Epoch [2/5] - Train Loss: 0.0224, Train Acc: 1.0000, Val Loss: 0.0162, Val Acc: 1.0000
Epoch [3/5] - Train Loss: 0.0103, Train Acc: 1.0000, Val Loss: 0.0099, Val Acc: 1.0000
Epoch [4/5] - Train Loss: 0.0069, Train Acc: 1.0000, Val Loss: 0.0070, Val Acc: 1.0000
Epoch [5/5] - Train Loss: 0.0050, Train Acc: 1.0000, Val Loss: 0.0056, Val Acc: 1.0000
Training completed in: 238.00s
```

PyTorch: Entrenamiento

- Visualizar historial de entrenamiento:
  - **history** → dict ('**accuracy**', '**loss**', '**val_accuracy**', '**val_loss**')
  - Función **plotLearningCurves** disponible en el paquete **torch_util**

```
plotLearningCurves(history)
```



PyTorch: Inferencia

- Predicción:

```
# Prediction for val_dataset
y_pred, prob, y_test =  predictModel(model, val_loader, DEVICE)

print(f"(Predicted,Actual): \n", list(zip(y_pred, y_test)))
print(f"Prob:      ", prob)
```

```
(Predicted,Actual):
 [(4, 4), (10, 10), (15, 15), (0, 0), (4, 4), (15, 15), (7, 7), (2, 2), (2, 2), (5, 5), (3, 3), (8, 8), (15, 15),
(7, 7), (9, 9), (11, 11), (10, 10), (6, 6), (7, 7), (12, 12), (8, 8), (7, 7), (14, 14), (10, 10), (5, 5), (3, 3),
(4, 4), (3, 3), (2, 2), (9, 9), (7, 7), (8, 8), (1, 1), (8, 8), (11, 11), (9, 9), (1, 1), (13, 13), (4, 4), (14, 14),
(3, 3), (7, 7), (0, 0), (12, 12), (9, 9), (3, 3), (12, 12), (8, 8), (7, 7), (15, 15), (11, 11), (6, 6), (13, 13),
(15, 15), (1, 1), (11, 11), (5, 5), (13, 13), (4, 4), (8, 8), (13, 13), (11, 11), (4, 4), (7, 7), (6, 6), (0, 0),
(13, 13), (3, 3), (5, 5), (9, 9), (8, 8), (4, 4), (12, 12), (14, 14), (15, 15)]
Prob:       [0.9869616031646729, 0.9914218783378601, 0.9939790964126587,
0.9952548742294312, 0.9953244924545288, 0.9922415018081665, 0.9939061999320984,
0.9951816201210022, 0.9953709244728088, 0.9897963404655457, 0.9941461086273193,
0.9954494833946228, 0.9944823980331421, 0.9948833584785461, 0.9923409223556519,
0.9928567409515381, 0.9917910099029541, 0.9959593415260315, 0.9950034022331238,
0.991814911365509, 0.9953832030296326, 0.9946852922439575, 0.9920428395271301,
0.9915589094161987, 0.9919025301933289, 0.9942554831504822, 0.9946278929710388,
```

Guardar Modelo

https://pytorch.org/docs/main/generated/torch.save.html

- Guardar modelo en formato PyTorch
  - PyTorch almacena los modelos en formato propio .pth
    - torch.**save**(model, path)
  - Podemos exportarlo también a otros formatos como **ONNX**: torch.onnx.**export**(), en este caso se requiere indicar las dimensiones de la entrada (los modelos PyTorch no la incluyen)
  - Debemos almacenar también las etiquetas usadas: cadena de texto asociada a los índices de clase
  - Almacenar modelo:

```
# Save model
torch.save(model, "MyNet_vgg16.pth")


# save keys/class names file
with open("aloi-16-keys-labels.txt", "w") as f:
    for label, idx in class_names.items():
        f.write(label+'\n')

with open("aloi-16-labels.txt", "w") as f:
    for label, idx in class_names.items():
        f.write(CLASSES_TEXT[label]+'\n')
```

Data Augmentation (CNN)

https://pytorch.org/vision/stable/transforms.html

- Preprocesamiento ('*Data Augmentation*'): torchvision.transforms
  - Para el aumento de datos podemos generar transformaciones aleatorias de los datos entre cada época. Para ello utilizamos el parámetro **transform** de **ImageFolder**
  - Crearemos una composición con la clase torchvision.tranforms.**Compose**() que añada al preprocesamiento básico visto en el apartado previo, las transformaciones aleatorias deseadas
  - Disponemos de dos versiones en espacios de nombre separados:
    - v1: torchvision.transforms
    - v2: torchvision.transforms.v2 incorpora transformaciones adicionales y más rápidas
  - Transformaciones disponibles como Clases o como Funciones

RandomResize()
RandomShortestSize()
RandomCrop()
RandomResizedCrop()
RandomIoUCrop()
RandomHorizontalFlip()
RandomVerticalFlip()
ElasticTransform()
RandomRotation()
RandomVerticalFlip()
RandomZoomOut()
RandomRotation()

RandomAffine()
RandomPerspective()
RandomChannelPermutation()
RandomPhotometricDistort(()
GaussianBlur(()
GaussianNoise()
RandomInvert()
RandomPosterize()
RandomRotation()
RandomSolarize()
RandomAdjustSharpness()
RandomAutocontrast()

RandomEqualize()
Compose(()
RandomApply()
RandomPhotometricDistort(()
RandomChoice(()
RandomOrder()
RandomErasing()

AutoAugment(()
RandAugment()
TrivialAugmentWide()
AugMix()

Leer Datasets Predefinidos Pytorch

https://pytorch.org/vision/stable/datasets.html

- Módulo datasets (clasificación): torchvision.dataset
  - torchvision.dataset.**Caltech101**(root, target_type='category', transform=None, target_transform=None, donwload=False) → (image, target)
    La tupla devuelta se pasa como parámetro a torch.utils.data.DataLoader()
    root: path donde esté almacenado o bien donde se va a descargar si **donwload** == True
    dataset 101 categorías, 40-800 imágenes por categoría, imágenes en color de 300x200
  - torchvision.dataset.**Caltech256**() → (image, target) 256 categorías
  - torchvision.dataset.**CelebA**() → (image, metadata) Large-scale CelebFaces Attributes Dataset
  - torchvision.dataset.**CIFAR10**() → (image, target) 10 categorías 60000 32x32 colour images
  - torchvision.dataset.**CIFAR100**() → (image, target) 100 categorías 60000 32x32 colour images
  - torchvision.dataset.**Country211**() → (image, target) 300 imágenes con coordenadas GPS
  - torchvision.dataset.**DTD**() → (image, target) Describable Textures Dataset
  - torchvision.dataset.**EuroSAT**() → (image, target) Imágenes satélite Sentinel-2 con GPS
  - torchvision.dataset.**FashionMNIST**() → (image, target) Imágenes ropa Zalando
  - torchvision.dataset.**FER2013**() → (image, target) Dataset caras 48x48 en 7 categorías
  - torchvision.dataset.**FDVCAircraft**() → (image, target) Dataset de aviones
  - torchvision.dataset.**Flickr8k**() → (image, target)
  - torchvision.dataset.**Flickr30k**() → (image, target)
  - torchvision.dataset.**Flowers102**() → (image, target) Oxford 102 clases de flores
  - torchvision.dataset.**Food101**() → (image, target) 101 clases comida
  - torchvision.dataset.**GTSRB**() → (image, target) German Traffic Sign Recognition Benchmark
  - torchvision.dataset.**INaturalist**() → (image, target)

Leer Datasets Predefinidos Pytorch

https://pytorch.org/vision/stable/datasets.html

- Módulo datasets (clasificación): torchvision.dataset
  - torchvision.dataset.**ImageNet**() → (image, target) 14 Millones imágenes con 20.000 categorías
  - torchvision.dataset.**Imagenette**() → (image, target)
  - torchvision.dataset.**KMNIST**() → (image, target)
  - torchvision.dataset.**LFWPeople**() → (image, target)
  - torchvision.dataset.**LSUN**() → (image, target)
  - torchvision.dataset.**MNIST**() → (image, target) caracteres escritos a mano
  - torchvision.dataset.**Omniglot**() → (image, target)
  - torchvision.dataset.**OxfordIIIPet**() → (image, metadata)
  - torchvision.dataset.**Places365**() → (image, target)
  - torchvision.dataset.**PCAM**() → (image, target)
  - torchvision.dataset.**QMNIST**() → (image, target)
  - torchvision.dataset.**RenderSST2**() → (image, target)
  - torchvision.dataset.**SEMEION**() → (image, target)
  - torchvision.dataset.**SBU**() → (image, target)
  - torchvision.dataset.**SatanfordCars**() → (image, target)
  - torchvision.dataset.**STL10**() → (image, target)
  - torchvision.dataset.**SUN397**() → (image, target)
  - torchvision.dataset.**SVHN**() → (image, target)
  - torchvision.dataset.**USPS**() → (image, target)

Leer Datasets Predefinidos Pytorch

https://pytorch.org/vision/stable/datasets.html

- Módulo datasets (Detección/Segmentación): torchvision.dataset

  imágenes etiquetas con ventanas de detección y mapas de segmentación

  - torchvision.dataset.**CocoDetection**() → (image, metadata)
  - torchvision.dataset.**CelebA**() → (image, metadata)
  - torchvision.dataset.**Cytuscapes**() → (image, metadata)
  - torchvision.dataset.**Kitti**() → (image, metadata)
  - torchvision.dataset.**OxfordIIIPet**() → (image, metadata)
  - torchvision.dataset.**SBDataset**() → (image, metadata)
  - torchvision.dataset.**VOCSegmentation**() → (image, metadata)
  - torchvision.dataset.**VOCDetection**() → (image, metadata)
  - torchvision.dataset.**WIDERFace**() → (image, metadata)

Tabla de Contenidos

- Introducción al Aprendizaje Profundo
- Redes Neuronales Convolucionales
- Transfer Learning
- Diseño y Entrenamiento de CNNs en Pytorch

….Fin