Visión por Computador (1782)

Herramientas de programación de aplicaciones OpenCV – Python

DNN Transfer-Learning with PyTorch

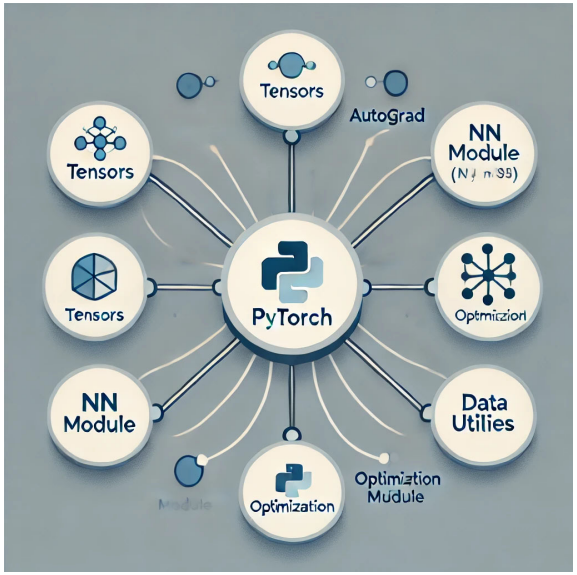
Luis M. Jiménez

Lab. Automática, Robótica y Visión por Computador
Universidad Miguel Hernández
http://arvc.umh.es

REDES NEURONALES CONVOLUCIONALES (CNN)

Transfer Learning (PyTorch)

https://pytorch.org



- Importar redes pre-entrenadas
- Clasificación: **VGG16**
- Modificar capas de salida y reentrenar
- Librerías: **torch, torchvision, torchinfo**

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

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

Deep Learning (CNN)

- Paquetes: torch, torchvision https://pytorch.org

1 Load data 2 Define model 3 Train model 4 Evaluate model

General

PyTorch is a open source machine learning framework. It uses **torch.Tensor** – multi-dimensional matrices – to process. A core feature of neural networks in PyTorch is the autograd package, which provides automatic derivative calculations for all operations on tensors.

import torch	Root package	**torch.randn(*size)**	Create random tensor
import torch.nn as nn	Neural networks	**torch.Tensor(L)**	Create tensor from list
from torchvision import datasets, models, transforms	Popular image datasets, architectures & transforms	**tnsr.view(a,b, ...)**	Reshape tensor to size (a, b, ...)
import torch.nn.functional as F	Collection of layers, activations & more	**requires_grad=True**	tracks computation history for derivative calculations

Layers

nn.Linear(m, n): Fully Connected layer (or dense layer) from m to n neurons

nn.ConvXd(m, n, s): X-dimensional convolutional layer from m to n channels with kernel size s; X ∈ {1, 2, 3}

nn.Flatten(): Flattens a contiguous range of dimensions into a tensor

nn.MaxPoolXd(s): X-dimensional pooling layer with kernel size s; X ∈ {1, 2, 3}

nn.Dropout(p=0.5): Randomly sets input elements to zero during training to prevent overfitting

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

nn.Embedding(m, n): Lookup table to map dictionary of size m to embedding vector of size n

nn.RNN/LSTM/GRU: Recurrent networks connect neurons of one layer with neurons of the same or a previous layer

torch.nn offers a bunch of other building blocks.
A list of state-of-the-art architectures can be found at https://paperswithcode.com/sota.

Load data

A dataset is represented by a class that inherits from **Dataset** (resembles a list of tuples of the form (features, label)).

DataLoader allows to load a dataset without caring about its structure.

Usually the dataset is split into training (e.g. 80%) and test data (e.g. 20%).

```
from torch.utils.data
 import Dataset, TensorDataset,
        DataLoader, random_split

train_data, test_data =
 random_split(
    TensorDataset(inps, tgts),
    [train_size,test_size]
 )

train_loader =
      DataLoader(
         dataset=train_data,
         batch_size=16,
         shuffle=True)
```

Activation functions

Common activation functions include **ReLU**, **Sigmoid** and **Tanh**, but there are other activation functions as well.

nn.ReLU() *creates a* nn.Module *for example to be used in Sequential models.* **F.relu()** *ist just a call of the ReLU function e.g. to be used in the* forward *method.*

nn.ReLU() or **F.relu()**
Output between 0 and ∞,
most frequently used activation function

nn.Sigmoid() or **F.sigmoid()**
Output between 0 and 1,
often used for predicting probabilities

nn.Tanh() or **F.tanh()**
Output between -1 and 1,
often used for classification with two classes

Define model

There are several ways to define a neural network in PyTorch, e.g. with **nn.Sequential** (a), as a class (b) or using a combination of both.

```
model = nn.Sequential(
  nn.Conv2D(■,■,■)
  nn.ReLU()
  nn.MaxPool2D(■)
  nn.Flatten()
  nn.Linear(■,■)
)
```
a

```
class Net(nn.Module):
  def __init__():
    super(Net, self).__init__()
    self.conv
      = nn.Conv2D(■,■,■)
    self.pool
      = nn.MaxPool2D(■)
    self.fc = nn.Linear(■,■)
  def forward(self, x):
    x = self.pool(
        F.relu(self.conv(x))
      )
    x = x.view(-1,■)
    x = self.fc(x)
    return x
model = Net()
```
b

Save/Load model

model = torch.load('PATH')	Load model
torch.save(model, 'PATH')	Save model

It is common practice to save only the model parameters, not the whole model using model.state_dict()

```
torch.save(model.state_dict(), 'params.ckpt')
model.load_state_dict(
                torch.load('params.ckpt'))
```

GPU Training

device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')

If a GPU with CUDA support is available, computations are sent to the GPU with ID 0 using **model.to(device)** or **inputs, labels = data[0].to(device), data[1].to(device).**

Train model

LOSS FUNCTIONS

PyTorch already offers a bunch of different loss fuctions, e.g.:

nn.L1Loss	Mean absolute error
nn.MSELoss	Mean squared error (L2Loss)
nn.CrossEntropyLoss	Cross entropy, e.g. for single-label classification or unbalanced training set
nn.BCELoss	Binary cross entropy, e.g. for multi-label classification or autoencoders

OPTIMIZATION (torch.optim)

Optimization algorithms are used to update weights and dynamically adapt the learning rate with gradient descent, e.g.:

optim.SGD	Stochastic gradient descent
optim.Adam	Adaptive moment estimation
optim.Adagrad	Adaptive gradient
optim.RMSProp	Root mean square prop

```
import torch.optim as optim

# Define loss function
loss_fn = nn.CrossEntropyLoss()

# Choose optimization method
optimizer = optim.SGD(model.parameters(),
                      lr=0.001, momentum=0.9)

# Loop over dataset multiple times (epochs)
for epoch in range(2):
    model.train() # activate training mode
    for i, data in enumerate(train_loader, 0):
        # data is a batch of [inputs, labels]
        inputs, labels = data

        # zero gradients
        optimizer.zero_grad()

        # calculate outputs
        outputs = model(inputs)
        # calculate loss & backpropagate error
        loss = loss_fn(outputs, labels)
        loss.backward()
        # update weights & learning rate
        optimizer.step()
```

```
correct = 0 # correctly classified
total   = 0 # classified in total

model.eval()
with torch.no_grad():
  for data in test_loader:
    inputs, labels = data
    outputs = model(inputs)
    _, predicted = torch.max(outputs.data, 1)
    total += labels.size(0) # batch size
    correct += (predicted==labels)
                        .sum().item()

print('Accuracy: %s' % (correct/total))
```

Evaluate model

The evaluation examines whether the model provides satisfactory results on previously withheld data. Depending on the objective, different metrics are used, such as acurracy, precision, recall, F1, or BLEU.

model.eval()	Activates evaluation mode, some layers behave differently
torch.no_grad()	Prevents tracking history, reduces memory usage, speeds up calculations

Deep Learning (CNN)

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

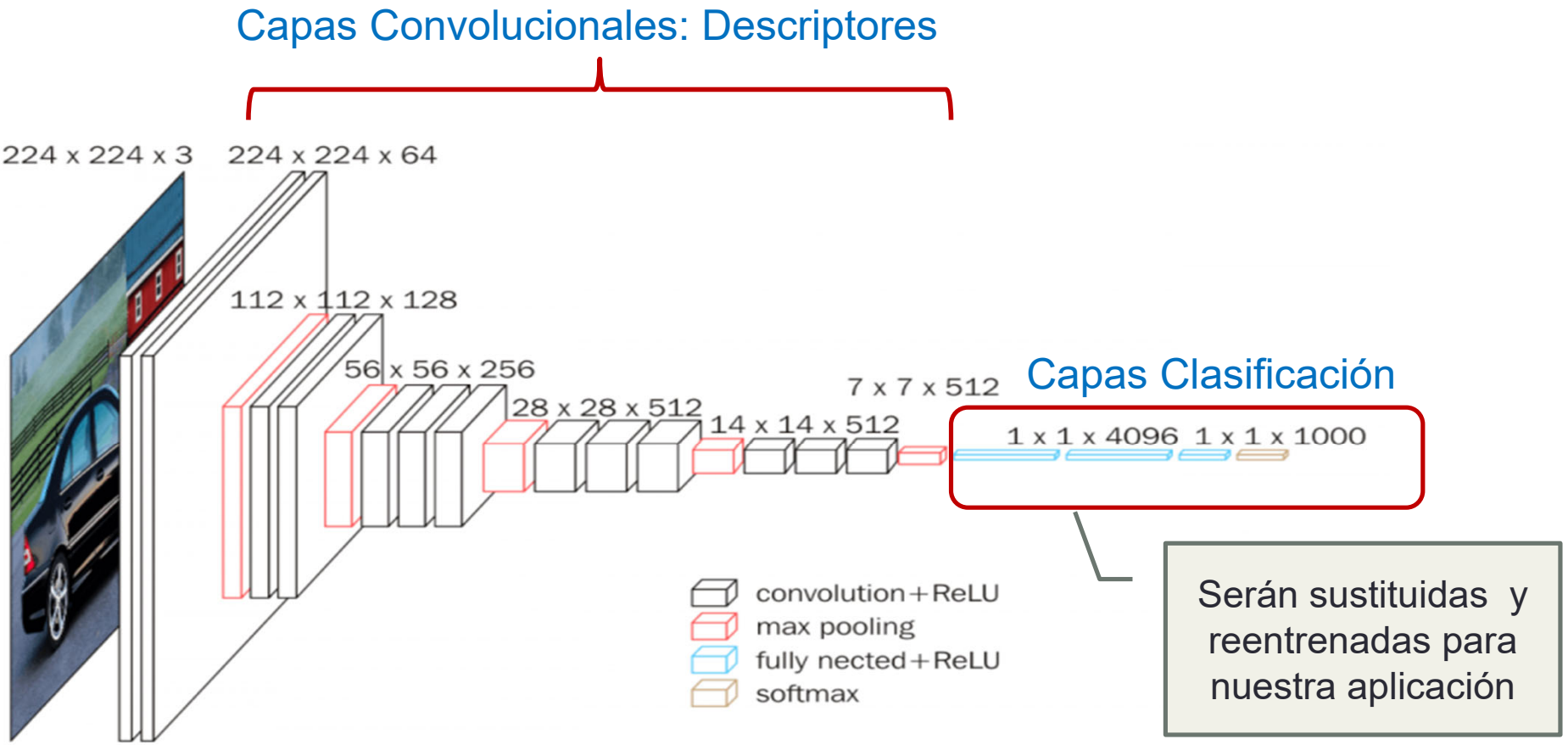
```
# Option A) Build model and download weights  (base network)
model = torchvision.models.vgg16(weights=torchvision.models.VGG16_Weights.DEFAULT)
```

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

Transfer Learning (CNN)

VGG16: (16 capas-138Millones parámetros) - input 3x224x224

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



PyTorch: Contenedores

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

- Modelos NN: torch.nn Clase Base: torch.nn.**Module**
  - 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)
)
```

```
from collections import OrderedDict

model = torch.nn.Sequential( OrderedDict( [
        ('layer1',  torch.nn.Linear(in_features=1000, out_features=100)),
        ('activation1',  torch.nn.Tanh()),
        ('layer2',  torch.nn.Linear(in_features=100, out_features=10)),
        ('activation2',  torch.nn.Softmax(dim=1))
] ) )
```

PyTorch: Contenedores

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

- Modelos NN: torch.nn Clase Base: torch.nn.**Module**
  - Modelo Funcional: torch.nn.functional
    - 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)

```
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/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()**

```
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: 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: 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** → $y = x$

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

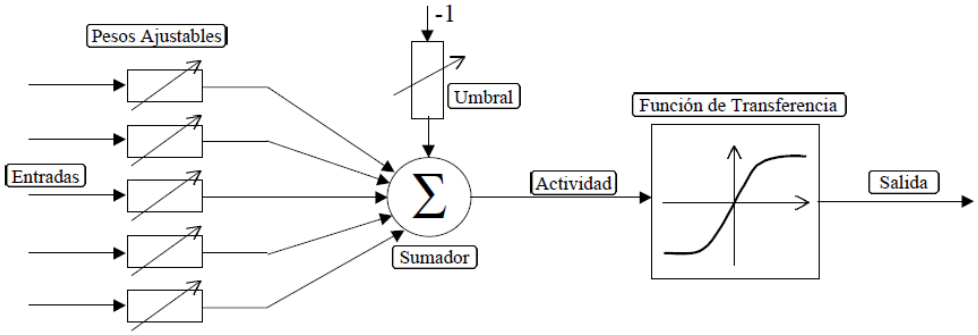
torch.nn.**LazyLinear** → no requiere configurar el tamaño de la entrada (se detecta con la primera entrada usada)

PyTorch: Layers

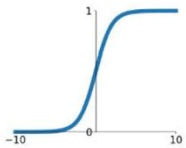
https://pytorch.org/docs/stable/nn.html

- Capas de Activación: torch.nn

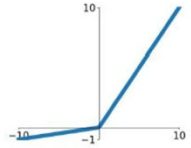
torch.nn.**AdaptiveLogSoftmaxWithLoss**
torch.nn.**ELU**
torch.nn.**Hardshrink**
torch.nn.**Hardsigmoid**
torch.nn.**Hardtanh**
torch.nn.**Hardswish**
torch.nn.**LeakyReLU**
torch.nn.**LogSigmoid**
torch.nn.**LogSoftmax**
torch.nn.**MultiheadAttention**
torch.nn.**PReLU**
torch.nn.**ReLU**
torch.nn.**ReLU6**
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**



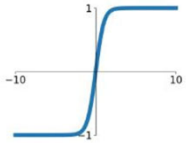
Sigmoid
$\sigma(x) = \frac{1}{1+e^{-x}}$



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

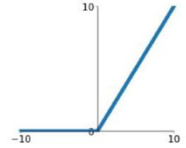


tanh
$\tanh(x)$

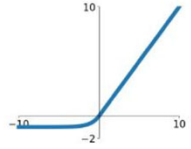


Maxout
$\max(w_1^T x + b_1, w_2^T x + b_2)$

ReLU
$\max(0, x)$



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



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

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

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

$$softsign = \frac{x}{|x| + 1}$$

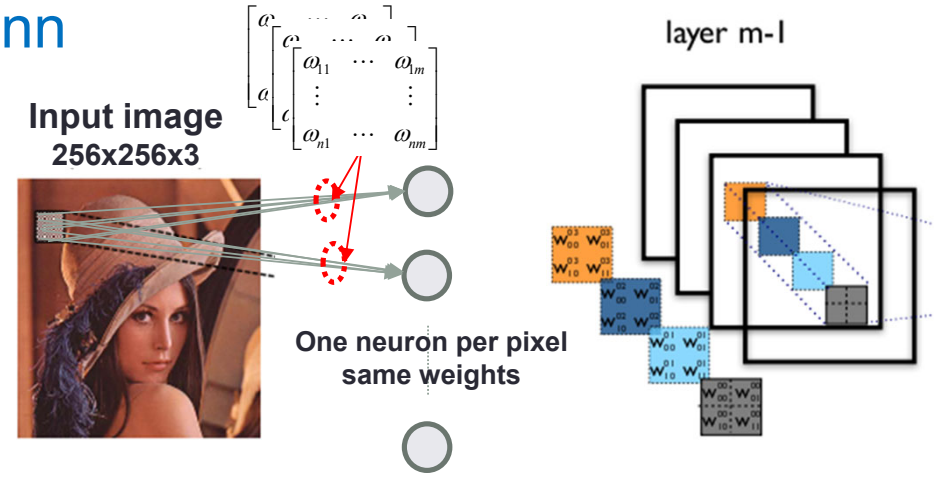
$$exponetial = e^x$$

$$\text{sgn}(z) = \begin{cases} +1 & si \quad x > 0 \\ 0 & si \quad x = 0 \\ -1 & si \quad x < 0 \end{cases}$$

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

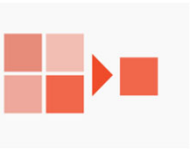
out_chanels : numero de conjunto de máscaras (pesos) distintas: canales de la capa
kernel_size: int or tuple (h,w) tamaño de la matriz de pesos
stride : int or tuple (1, 1): paso desplazamiento de la convolución en altura y anchura por la imagen
padding: int or {'valid', 'same'} *valid*: elimina pixels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

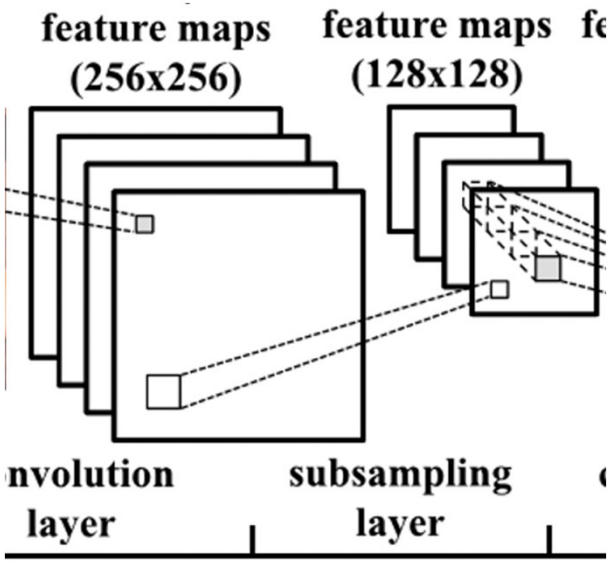
dilation : int or tuple (1, 1): espacios entre pixels a los que se aplica la convolución
groups=1: grupos del canales de la imagen de la entrada que se procesan separadamente
padding_mode : 'zeros', 'reflect', 'replicate' or 'circular'. Default: 'zeros'
dtype: tipo de los pesos Orden de las dimensiones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

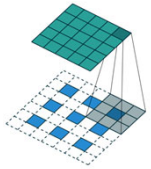
PyTorch: Layers

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

- Capas 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

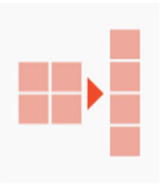
https://pytorch.org/docs/stable/nn.html

- 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

torch.nn.**CrossEntropyLoss**

Probabilistic losses

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**

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

PyTorch: Modelos

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

- Desensambado: 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 Secuenciales: Editar el método **forward**() o bien sustituirla por torch.nn.**Identity**()

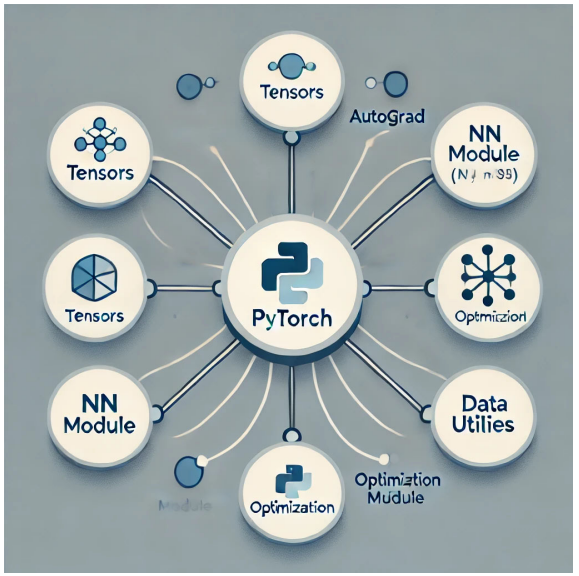
```
# Disassemble model
modules = dict( [(name, module) for name, module in model.named_modules()] )

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

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

PYTORCH

Ejemplo Clasificador PyTorch (et1.py):

- Modelo secuencial
- Red Neuronal MLP (Perceptrón Multicapa)
- Datos entrenamiento práctica 4 TITERE

https://pytorch.org




```
import torch
import torchinfo
import titere
import time
```

PyTorch: Datos de Entrenamiento

https://umh1782.umh.es/pytorch

- titere.**readWekaData()**
- titere.**readWekaDataTitere()**
- titere.**showPerformance()**
- titere.**plotLearningCurves()**

- Cargar datos entrenamiento titere
  - Carga de datos generados desde la aplicación TITERE
    - titere.**readWekaDataTitere**(file, labelRange=[]) → dict(numpy arrays)
  - Debemos convertir los datos a Tensores: torch.tensor()
  - Conversión de etiquetas vectoriales a matriciales (categóricas)
    - torch.nn.functional.**one_hot**(tensor, num_classes=-1) → LongTensor *(tensor debe ser de tipo **long**)*

```
DATA_FILE = 'train.arff'          # default train data file
TEST_FILE = 'test.arff'           # default test data file

# import titere reduced data with only 4 colums: ('Compacidad', 'Excentricidad', 'Rel_Invar_1', 'Rel_Invar_2'))
trainDataDict = titere.readWekaDataTitere(DATA_FILE)
trainData = torch.tensor( trainDataDict['dataMat'])
trainLabels = torch.tensor( trainDataDict['label'])     # vector with numeric labels (starts at 1)
labelRange = trainDataDict['labelRange']                # list with the numeric/string label association

testDataDict = titere.readWekaDataTitere(TEST_FILE, labelRange)
testData = torch.tensor( testDataDict['dataMat'] )
testLabels = torch.tensor( testDataDict['label'] )

numFeatures = trainData.size(-1)        # numFeatures trainData last dimension
numClasses = len(labelRange)

# adapt trainLabels/testLabels vector to a float32 matrix with one column per class
# train/test Labels: vector with numeric labels starting at 1, adjust initial index to 0 (trainLabels-1)
trainLabels = torch.nn.functional.one_hot (trainLabels.long()-1, num_classes=numClasses ).float()
testLabels = torch.nn.functional.one_hot (testLabels.long()-1, num_classes=numClasses ).float()
```

$$y \Rightarrow [y]$$

$$\begin{bmatrix}0\\2\\1\\3\end{bmatrix} \Rightarrow \begin{bmatrix}1&0&0&0\\0&0&1&0\\0&1&0&0\\0&0&0&1\end{bmatrix}$$

PyTorch: Dataloaders

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

- Cargar datos entrenamiento: DataLoaders
  - Clases:
    - torch.utils.data.DataLoader
    - torch.utils.data.TensorDataset

  - torch.utils.data.TensorDataset Crea un dataset que agrupa los tensores de datos y etiquetas
  - torch.utils.data.Dataloader Se encarga de la carga de datos de entrenamiento, agrupación en lotes (batch), barajado aleatorio y aumento de datos

```
# Create a DataLoader (internal object to manage batches/suffle/data augmentation)
train_loader = torch.utils.data.DataLoader( torch.utils.data.TensorDataset(trainData, trainLabels),
                                            batch_size=2, shuffle=True)
val_loader = torch.utils.data.DataLoader( torch.utils.data.TensorDataset(testData, testLabels),
                                            batch_size=2, shuffle=True)
```

PyTorch: Models

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

- Ejemplo Modelos Secuenciales: torch.nn
  - torch.nn.**Sequential**(**layers**=None) → model
    - Se pasan las capas como o una secuencia de parámetros, o bien un diccionario ordenado (**OrderedDict**) con el nombre y la capa
  - Utilidades de visualización:
    - **print**(model)
    - torchinfo.**summary**(model) → imprime los detalles del modelo

```
numFeatures = trainData.size(-1)        # numFeatures trainData last dimension
numClasses = len(labelRange)



# build Sequential torch model (Four features / Four classes)

model = torch.nn.Sequential(
          torch.nn.Linear(in_features=numFeatures, out_features=20),
          torch.nn.Tanh(),
          torch.nn.Linear(in_features=20, out_features=numClasses )

           # CrossEntropyLoss includes LogSoftmax internaly so we cannot use an output torch.nn.Softmax(dim=1)
)
print(f"MLP Sequential Model:\n", model)      # prints model summary
```

PyTorch: Entrenamiento

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

- Configurar entrenamiento:
  - Seleccionar dispositivo: '**cpu**, '**cuda**' y mover el modelo al dispositivo
  - Configurar función de pérdida y optimizador

```
# select model device (CPU/CUDA) once it is created and modified
DEVICE = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = model.to(DEVICE)

# Configure Loss function and optimizer
criterion = torch.nn.CrossEntropyLoss()
# SGD  optimizer  lr: Learning Rate, momentum: gradient inertia
optimizer = torch.optim.SGD(model.parameters(), lr=0.1, momentum=0.5)
```

- Ejecutar bucle de entrenamiento:
  - En Pytorch no hay disponible una función directa que realice el entrenamiento, debemos implementar un bucle que itere para cada época e implemente:
    1) Lectura de lotes de datos desde el DataLoader
    2) Mover los datos al dispositivo de aceleración, si disponemos de una GPU
    3) Realizar la inferencia (obtener la salida) para cada entrada (***Forward***).
    4) Calcular la función de pérdida con las salidas de la inferencia
    5) Reiniciar los gradientes en cada lote
    6) Calcular la retropropagación de la función de pérdida (***Backward***) y los gradientes
    7) Actualizar los pesos del modelo con el optimizador
    8) Calcular histórico de las métricas de entrenamiento (*accuracy/loss*) para los datos de entrenamiento y validación

PyTorch: Modelos

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

- Entrenamiento: función **trainModel()** disponible en torch_util.py
  - Vamos a crear la función **trainModel()** que realice el entrenamiento para poder usarla en diferentes proyectos.
  - Parámetros: *model, train_loader, val_loader, criterion, optimizer, epochs, device*
  - Iremos implementándola paso a paso:
    1) Bucle principal para cada época (actualización del modelo usando todos los datos)
    2) Configurar Modelo para entrenamiento (ciertas capas tienen comportamientos específicos)
    3) Inicializar métricas de la época (se acumulan para cada lote)
    4) Lectura de un lote de datos desde el DataLoader
    5) Mover los datos al dispositivo de aceleración, si disponemos de una GPU

```
def TrainModel(model, train_loader, val_loader, criterion, optimizer, epochs, device):

    print(f"\nTraining Model:")
    for epoch in range(epochs):
        model.train()   # configure model for training
        running_loss, correct, total = 0.0, 0, 0          # init epoch metrics

        print(f"Epoch [{epoch + 1}/{epochs}] ", end='')
        for batch_idx, (inputs, labels) in enumerate(train_loader):
            inputs, labels = inputs.to(device), labels.to(device)
            ……

        #end train batch loop
    # end for epoch  loop
#end def TrainModel()
```

PyTorch: Entrenamiento

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

- Entrenamiento: función **trainModel()** disponible en torch_util.py
  - Vamos a crear la función **trainModel()** que realice el entrenamiento para poder usarla en diferentes proyectos.
  - Parámetros: *model, train_loader, val_loader, criterion, optimizer, epochs, device*
  - Iremos implementándola paso a paso:
    6) Realizar la inferencia (obtener la salida) para cada entrada (***Forward***).
    7) Calcular la función de pérdida con las salidas de la inferencia
    8) Reiniciar los gradientes en cada lote
    9) Calcular la retropropagación de la función de pérdida (***Backward***) y los gradientes
    10) Actualizar los pesos del modelo con el optimizador

```
....
for batch_idx, (inputs, labels) in enumerate(train_loader):
        ......

        # Forward pass
        outputs = model(inputs)
        loss = criterion(outputs, labels)

        # Backward pass
        optimizer.zero_grad()     # clear out batch accumulative gradients before backpropagation
        loss.backward()           # loss backpropagation
        optimizer.step()          # update model parameters
        ......

#end train batch loop
```

PyTorch: Entrenamiento

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

- Entrenamiento: función **trainModel()** disponible en torch_util.py
  - Vamos a crear la función **trainModel()** que realice el entrenamiento para poder usarla en diferentes proyectos.
  - Iremos implementándola paso a paso:
    11) Calcular histórico de las métricas de entrenamiento (*accuracy/loss*) para los datos de entrenamiento, se acumulan para cada lote y se normalizan al acabar la época

```
….
for batch_idx, (inputs, labels) in enumerate(train_loader):
        ……

        # Accumulate batch loss/accuray metrics
        running_loss += loss.item() * inputs.size(0)          # add batch loss metric

        _, predicted = torch.max(outputs, 1)                  # categorical to vector
        if labels.dim() == 2 : _, labels = torch.max(labels, 1)   # categorical to vector

        correct += (predicted == labels).sum().item()  # add batch accuracy metric
        total += labels.size(0)
        # print progressing status
        print(f"\rEpoch [{epoch + 1}/{epochs}] - Batch [{batch_idx:02}] | "
              f"Train Loss: {running_loss/total:.4f}, Train Acc: {correct/total:.4f} ", end='', flush=True )
#end train batch loop

train_loss = running_loss / total          # training epoch loss
train_acc = correct / total                # training epoch accuracy
….
```

PyTorch: Entrenamiento

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

- Validación: **trainModel()**
  - Iremos implementándola paso a paso:
    12) Configurar Modelo para inferencia (ciertas capas tienen comportamientos específicos)
    13) Desactivar el calculo de gradientes: torch.**no_grad**()
    14) Realizar la inferencia (obtener la salida) para cada entrada (***Forward***).
    15) Calcular métricas para datos de validación

```
….
# Validation metrics
model.eval()    # configure model for inference
val_loss, val_correct, val_total = 0.0, 0, 0
with torch.no_grad():   # disable gradient calculation for inference
        for inputs, labels in val_loader:
                inputs, labels = inputs.to(device), labels.to(device)
                outputs = model(inputs)
                loss = criterion(outputs, labels)

                val_loss += loss.item() * inputs.size(0)          # add batch loss metric
                _, predicted = torch.max(outputs, 1)                          # categorical to vector
                if labels.dim() == 2 : _, labels = torch.max(labels, 1)        # categorical to vector
                val_correct += (predicted == labels).sum().item()   # add batch accuracy metric
                val_total += labels.size(0)
                print(f".", end='', flush=True)     # print progressing status
        #end validation batch loop
#end with torch.no_grad()
val_loss /= val_total                  # validation epoch loss
val_acc = val_correct / val_total   # validation epoch accuracy
….
```

PyTorch: Entrenamiento

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

- Entrenamiento: **trainModel()**
  - Iremos implementándola paso a paso:
    16) Almacenar el histórico con las métricas de entrenamiento y validación

```
def trainModel(model, train_loader, val_loader, criterion, optimizer, epochs, device):
    # init metrics history
    history = { 'loss': [],
        'val_loss': [],
        'accuracy': [],
        'val_accuracy': []
    }
    for epoch in range(epochs):
        …..

        # add epoch metrics to history dictionary
        history['loss'].append(train_loss)
        history['val_loss'].append(val_loss)
        history['accuracy'].append(train_acc)
        history['val_accuracy'].append(val_acc)

         # Verbose: print current epoch metrics
        print(f"\rEpoch [{epoch + 1}/{epochs}] - "
                f"Train Loss: {train_loss:.4f}, Train Acc: {train_acc:.4f}, "
                f"Val Loss: {val_loss:.4f}, Val Acc: {val_acc:.4f}")
     # end for epoch  loop

     return history
#end def trainModel()
```

PyTorch: Entrenamiento

- Entrenamiento: torch_util.**trainModel()**
  - Llamada a la función **trainModel**():

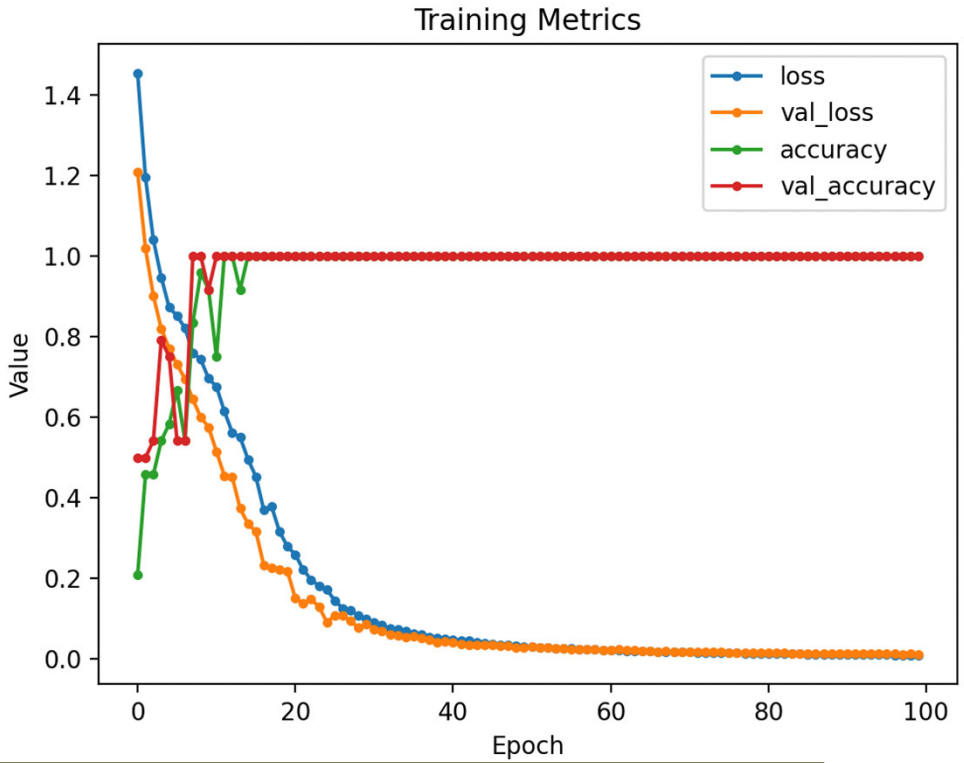
```
from torch_util import trainModel, predictModel

# Train model
start_time = time.time()

history = trainModel( model=model, train_loader=train_loader, val_loader=val_loader,
                      criterion=criterion, optimizer=optimizer, epochs=100, device=DEVICE )

print(f"Training completed in: {time.time() - start_time:.2f}s")
```

```
Training Model:
Epoch [2/100] - Train Loss: 1.1945, Train Acc: 0.4583, Val Loss: 1.0182, Val Acc: 0.5000...
Epoch [3/100] - Train Loss: 1.0396, Train Acc: 0.4583, Val Loss: 0.9010, Val Acc: 0.5417...
Epoch [4/100] - Train Loss: 0.9469, Train Acc: 0.5417, Val Loss: 0.8193, Val Acc: 0.7917...
Epoch [5/100] - Train Loss: 0.8723, Train Acc: 0.5833, Val Loss: 0.7708, Val Acc: 0.7500...
Epoch [6/100] - Train Loss: 0.8511, Train Acc: 0.6667, Val Loss: 0.7310, Val Acc: 0.5417...
 .......

Epoch [96/100] - Train Loss: 0.0106, Train Acc: 1.0000, Val Loss: 0.0128, Val Acc: 1.0000...
Epoch [97/100] - Train Loss: 0.0103, Train Acc: 1.0000, Val Loss: 0.0128, Val Acc: 1.0000...
Epoch [98/100] - Train Loss: 0.0102, Train Acc: 1.0000, Val Loss: 0.0125, Val Acc: 1.0000...
Epoch [99/100] - Train Loss: 0.0101, Train Acc: 1.0000, Val Loss: 0.0127, Val Acc: 1.0000...
Epoch [100/100] - Train Loss: 0.0099, Train Acc: 1.0000, Val Loss: 0.0126, Val Acc: 1.0000...
Training completed in: 1.02s
```

PyTorch: Entrenamiento

- Visualizar historial de entrenamiento:
  - **history** → dict ('**accuracy**', '**loss**', '**val_accuracy**', '**val_loss**')
  - matplotlib.**pyplot** : misma sintaxis que Matlab
  - Función **plotLearningCurves** disponible en el paquete **titere** y **torch_util**

```
import matplotlib.pyplot as plt
import matplotlib

def plotLearningCurves(metrics):

    plt.ion()   #interactive mode, non blocking plots

    for metric_name, values in metrics.items():
          plt.plot(values, '.-', label=metric_name)

    plt.title('Training Metrics')
    plt.xlabel('Epoch')
    plt.ylabel('Value')
    plt.legend(loc='best')
    plt.pause(0.1)      #  non-blocking window uptade

# end of plotLearningCurves function
```




```
from torch_util import showPerformance, plotLearningCurves

plotLearningCurves(history)
```

PyTorch: Inferencia

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

- Predicción: función **predictModel** disponible en torch_util.py
  - Implementaremos una función para la inferencia del modelo:
    1) Configurar Modelo para inferencia. Desactivar el calculo de gradientes: torch.**no_grad**()
    2) Lectura de un lote de datos desde el DataLoader
    3) Realizar la inferencia (obtener la salida) para cada entrada (***Forward***).
    4) Extraer vector de predicciones, probabilidad, clase real

```
def predictModel(model, data_loader, device):
        y_pred = []; prob = []; y_test = []
        model.eval()   # configure model for inference
        with torch.no_grad():   # disable gradient calculation for inference
                for inputs, labels in data_loader:
                        inputs, labels = inputs.to(device), labels.to(device)
                        outputs = model(inputs)
                        outputs = torch.softmax(outputs, dim=1)  # softmax normalization to get probabilities

                        probability, predicted = torch.max(outputs, 1)         # categorical to vector
                        if labels.dim() == 2 : _, labels = torch.max(labels, 1)        # categorical to vector

                        y_pred.extend(predicted.tolist())
                        prob.extend(probability.tolist())
                        y_test.extend(labels.tolist())
                        print(f".", end='', flush=True)     # print progressing status
                 #end validation batch loop
        #end with torch.no_grad()

        return y_pred, prob, y_test
```

PyTorch: Inferencia

- Predicción:

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

print(f"Predicted: ", y_pred)
print(f"Actual:    ", y_test)
print(f"Prob:      ", prob)

showPerformance(y_test, y_pred, labelRange)
```

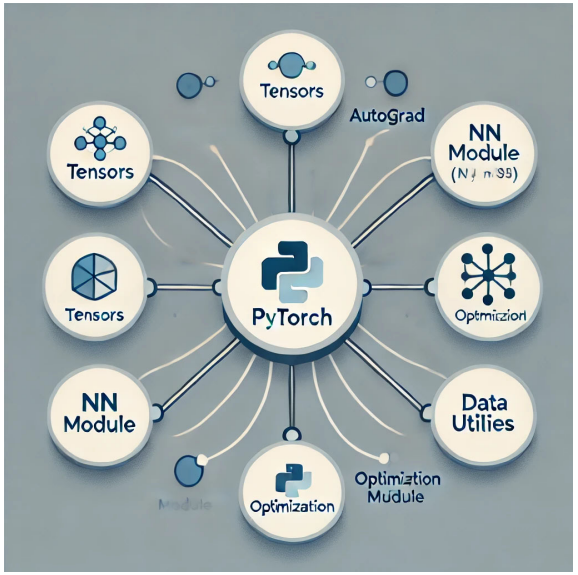
```
Predicted:  [3, 2, 0, 0, 1, 1, 0, 1, 0, 3, 2, 2, 2, 2, 0, 1, 3, 0, 1, 3, 3, 3, 2, 1]
Actual:     [3, 2, 0, 0, 1, 1, 0, 1, 0, 3, 2, 2, 2, 2, 0, 1, 3, 0, 1, 3, 3, 3, 2, 1]
Prob:       [0.9990304708480835, 0.9965984225273132, 0.9570725560188293,
0.9570606350898743, 0.9957953691482544, 0.9970447421073914, 0.961925745010376,
0.9875823259353638, 0.9707880020141602, 0.9988872408866882, 0.9926388263702393,
0.9937146306037903, 0.9949994087219238, 0.9948923587799072, 0.9588243961334229,
0.9947330951690674, 0.9989308714866638, 0.9691560864448547, 0.995020866394043,
0.9987531900405884, 0.999029278755188, 0.9989874958992004, 0.993822455406189,
0.9961236119270325]
```

REDES NEURONALES CONVOLUCIONALES (CNN)

Transfer Learning (PyTorch) (et2.py)

https://pytorch.org

- Importar redes pre-entrenadas
- Clasificación: **VGG16**
- Modificar capas de salida y reentrenar
- Librerías: **torch, torchvision, torchinfo**
  torch_util



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

Deep Learning (CNN)

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

Modelos específicos para imágenes en color o b/n

  - Datasets:
    - **ImageNet** Large Scale Visual Recognition Challenge (ILSVRC):
      - https://image-net.org/challenges/LSVRC/
    - Disponible una muestra en :
      - https://umh1782.umh.es/pytorch

Transfer Learning (CNN)

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

- Modelos PyTorch: torchvision.models
  - torchvision.models.**vgg16**(weights=None) → modelo

    weights: Pesos del modelo (None: construye el modelo sin valor de pesos)

    torchvision.models.**VGG16_Weights**.**DEFAULT**

    Pesos disponibles para varias configuraciones de entrenamiento de la red: (datsets / tipo de imagen: color/bn / tipo de datos de los pesos)

    DEFAULT: configuración de pesos por defecto

  - Cargar Pesos desde un fichero, método **load_state_dict**():
    - model.**load_state_dict**(torch.**load**("../pytorch-models/vgg16-weights.pth"))

  - Transformaciones de la entrada específicas de la red y pesos:
    - preprocess = torchvision.models.VGG16_Weights.DEFAULT.**transforms**()

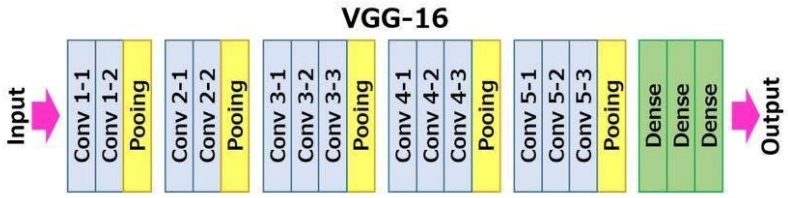
  - Leer/Guardar ficheros modelo en formato PyTorch (*.pth)
    - torch.**load**(**filepath**) → modelo (arquitectura+pesos) ó solo pesos
    - torch.**save**(model, filepath) Almacena modelo (arquitectura+pesos)
    - torch.**save**(model.**state_dict**(), filepath) Almacena solo pesos

Modelos disponibles en: http://umh1782.umh.es/python

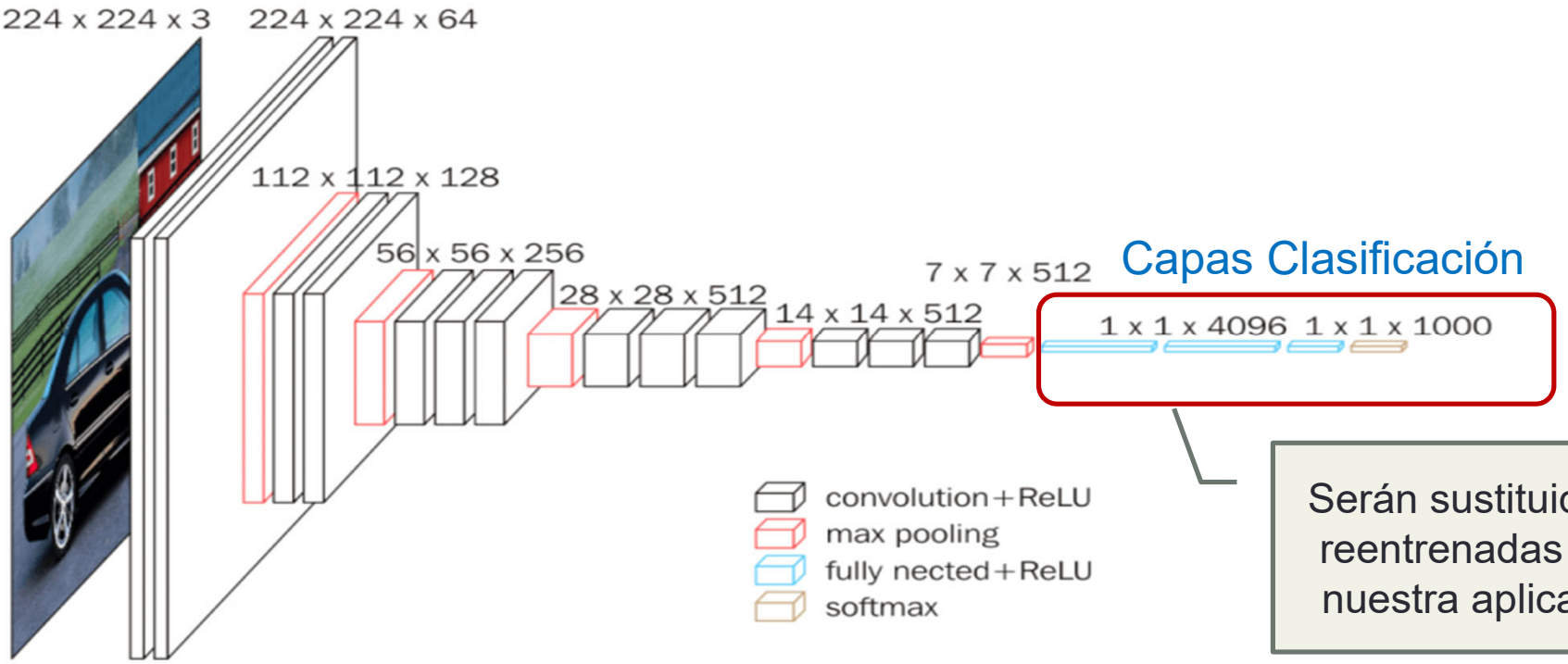
Transfer Learning (CNN)

VGG16: (16 capas-138 Millones de parámetros) - input 3x224x224

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



Capas Convolucionales: Descriptores



Capas Clasificación

Serán sustituidas y reentrenadas para nuestra aplicación

Transfer Learning (CNN)

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

- Cargar paquetes y definir datos de las clases a reconocer:

```
import torch
import torchvision
import torchinfo
import time
import inspect      # to show model code for forward() method


# local package for Course Computer Vision (1782): train, predict, plot metrics
from torch_util import trainModel, predictModel, plotLearningCurves

CLASSES_TEXT = { "7": "Shell",  "8": "Green Cube",  "249": "Red Cube",
                 "9": "Blue Shoe",   "15": "Piolin", "35": "Cup",
                 "62": "Duck", "69": "Tomato", "291": "Ball",
                 "138": "Blue Car", "160": "White Car", "156": "Red Clock",
                 "233": "Corn", "323": "Packet", "332": "Vase", "950": "Banana" }

NUM_CLASSES = len(CLASSES_TEXT)
```

Transfer Learning (CNN)

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

- Cargar Modelo Tensorflow: torchvision.models
  torch.**load**(), torchinfo.**summary**()
  - Utilidades de visualización:
    - **print**(model) → mostrar resumen del modelo
    - **print**(inspect.**getsource**(model.**forward**)) → mostrar código del método forward()
    - torchinfo.**summary**(model) → imprime los detalles del modelo

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

# Option B) Build model and load weigths from local file (base network)
#model = torchvision.models.vgg16()
#model.load_state_dict(torch.load("../pytorch-models/vgg16-weights.pth"))

# Option C) alternatively  we can use pre-downloaded models (http://umh1782.umh.es/python)
#model = torch.load("../pytorch-models/vgg16_full_model.pth")

# prints model summary
print(f"VGG16 model summary:\n", model)
print(inspect.getsource(model.forward))     # show model code for forward() method

# Detailed model info
torchinfo.summary(model, col_names=["input_size", "output_size", "num_params", "trainable"],
                                input_size=(1, 3, 224, 224), col_width=15)
```

Transfer Learning (CNN)

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

Salida: print(model)

```
VGG16 model summary: VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace=True)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace=True)
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU(inplace=True)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU(inplace=True)
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU(inplace=True)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU(inplace=True)
    (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (18): ReLU(inplace=True)
    (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (20): ReLU(inplace=True)
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (22): ReLU(inplace=True)
    (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (25): ReLU(inplace=True)
    (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (27): ReLU(inplace=True)
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (29): ReLU(inplace=True)
    (30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
....
```

Capas Convolucionales Descriptores: submódulo 'features'

param.**requires_grad** = False

```
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
```

Serán sustituidas y reentrenadas para nuestra aplicación

```
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace=True)
    (2): Dropout(p=0.5, inplace=False)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace=True)
    (5): Dropout(p=0.5, inplace=False)
    (6): Linear(in_features=4096, out_features=1000, bias=True)
  )
)
```

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)

Capas Convolucionales
Descriptores: submódulo 'features'

cambiaremos ***Trainable*** a False:

param.**requires_grad** = False

Serán sustituidas y reentrenadas para nuestra aplicación

Transfer Learning (CNN)

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

- Configurar entrenamiento de pesos:

  torch.nn.Module.**parameters()** → *Iterator[Parameter].*
  torch.nn.parameter.Parameter.**requires_grad**

  - Congelaremos los pesos de las capas convolucionales: submodelo 'features'
  - Para ello iteraremos por todos los parámetros de la parte del modelo a congelar obtenidos con el método **parameters**() y configuraremos el campo **requires_grad** a False

```
# Freeze training weigths for layers in current feature block
for param in model.features.parameters():
    param.requires_grad = False
```

Transfer Learning (CNN)

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

- Modificar Modelo: torch.nn
  - Sustituiremos el submodelo 'classifier' que agrupa las capas de clasificación por un nuevo modelo Secuencial
  - Obtendremos las dimensiones del tensor de entrada de la capa **0** del submodelo 'classifier' previo:
    - model.classifier[0].**in_features**

```
# Modifiy the last block of VGG network (classifier)
model.classifier = torch.nn.Sequential(
    torch.nn.Dropout(0.5),
    torch.nn.Linear(in_features=model.classifier[0].in_features, out_features=1000),
    torch.nn.LeakyReLU(),
    torch.nn.Linear(in_features=1000, out_features=100),
    torch.nn.LeakyReLU(),
    torch.nn.Linear(in_features=100, out_features=NUM_CLASSES),

    # CrossEntropyLoss includes LogSoftmax internaly so we cannot use an output torch.nn.Softmax(dim=1)
)

# prints model summary
print(f"Modified VGG16 model summary:\n", model)

# Detailed model info
torchinfo.summary(model, col_names=["input_size", "output_size", "num_params", "trainable"],
                  input_size=(1, 3, 224, 224), col_width=18)
```

Transfer Learning (CNN)

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

```
VGG16 model summary: VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace=True)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace=True)
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU(inplace=True)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU(inplace=True)
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU(inplace=True)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU(inplace=True)
    (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (18): ReLU(inplace=True)
    (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (20): ReLU(inplace=True)
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (22): ReLU(inplace=True)
    (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (25): ReLU(inplace=True)
    (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (27): ReLU(inplace=True)
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (29): ReLU(inplace=True)
    (30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
....
```

- Modelo Modificado

Capas Convolucionales
Descriptores: submódulo 'features'

param.**requires_grad** = False

Nuevas Capas

param.**requires_grad** = True

```
  (classifier): Sequential(
    (0): Dropout(p=0.2, inplace=False)
    (1): Linear(in_features=25088, out_features=1000, bias=True)
    (2): LeakyReLU(negative_slope=0.01)
    (3): Linear(in_features=1000, out_features=100, bias=True)
    (4): LeakyReLU(negative_slope=0.01)
    (5): Linear(in_features=100, out_features=16, bias=True)
  )
)
```

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, 16]              --                   Partial
├─Sequential: 1-1                        [1, 3, 224, 224]     [1, 512, 7, 7]       --                   False
│    └─Conv2d: 2-1                       [1, 3, 224, 224]     [1, 64, 224, 224]    (1,792)              False
│    └─ReLU: 2-2                         [1, 64, 224, 224]    [1, 64, 224, 224]    --                   --
│    └─Conv2d: 2-3                       [1, 64, 224, 224]    [1, 64, 224, 224]    (36,928)             False
│    └─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)             False
│    └─ReLU: 2-7                         [1, 128, 112, 112]   [1, 128, 112, 112]   --                   --
│    └─Conv2d: 2-8                       [1, 128, 112, 112]   [1, 128, 112, 112]   (147,584)            False
│    └─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)            False
│    └─ReLU: 2-12                        [1, 256, 56, 56]     [1, 256, 56, 56]     --                   --
│    └─Conv2d: 2-13                      [1, 256, 56, 56]     [1, 256, 56, 56]     (590,080)            False
│    └─ReLU: 2-14                        [1, 256, 56, 56]     [1, 256, 56, 56]     --                   --
│    └─Conv2d: 2-15                      [1, 256, 56, 56]     [1, 256, 56, 56]     (590,080)            False
│    └─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)          False
│    └─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)          False
│    └─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)          False
│    └─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)          False
│    └─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)          False
│    └─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)          False
│    └─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
│    └─Dropout: 2-32                     [1, 25088]           [1, 25088]           --                   --
│    └─Linear: 2-33                      [1, 25088]           [1, 1000]            25,089,000           True
│    └─LeakyReLU: 2-34                   [1, 1000]            [1, 1000]            --                   --
│    └─Linear: 2-35                      [1, 1000]            [1, 100]             100,100              True
│    └─LeakyReLU: 2-36                   [1, 100]             [1, 100]             --                   --
│    └─Linear: 2-37                      [1, 100]             [1, 16]              1,616                True
```

- **Modelo Modificado**

torchinfo.**summary(**model**)**

Capas Convolucionales
Descriptores: submódulo 'features'

param.**requires_grad** = False

Nuevas Capas

param.**requires_grad** = True

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/vision/stable/transforms.html

- Preprocesamiento de las imágenes: torchvision.transforms
  - El parámetro **transform** de **ImageFolder** nos permite asignar la operación de preprocesamiento de las imágenes: típicamente se redimensiona y recorta la imagen, se convierte a tensor y se normalizan los valores restando la media y dividiendo por la desviación típica.
  - Para los modelos disponibles en torchvision.models disponemos de transformaciones predefinidas para distintas configuraciones de entrenamiento, accesibles en el método **transforms**() del objeto **weigths** usado para cargar los pesos del modelo
  - También podemos crearlas mediante la clase torchvision.transforms.**Compose**() pasándole una lista de las transformaciones (torchvision.transforms) a realizar en la imagen → Será útil para el **aumento de datos** mediante transformaciones aleatorias (lo veremos al final del tutorial)

```
# Image transform  Resize/normalize input image color (net specific) [scale*(x-mean)]
# OPTION A: download default VGG16 preprocess transformation
preprocess_vgg16 = torchvision.models.VGG16_Weights.DEFAULT.transforms()
print("VGG16 Preprocess Transform: ", preprocess_vgg16)

# OPTION B: manually compose preprocess transformation
# preprocess_vgg16 = torchvision.transforms.Compose([
#     torchvision.transforms.Resize(size=256),
#     torchvision.transforms.CenterCrop(size=(224, 224)),
#     torchvision.transforms.ToTensor(),
#     torchvision.transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
# ])
```

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 da

Transfer Learning(CNN)

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

- Cargar Datos entrenamiento
  - 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()**

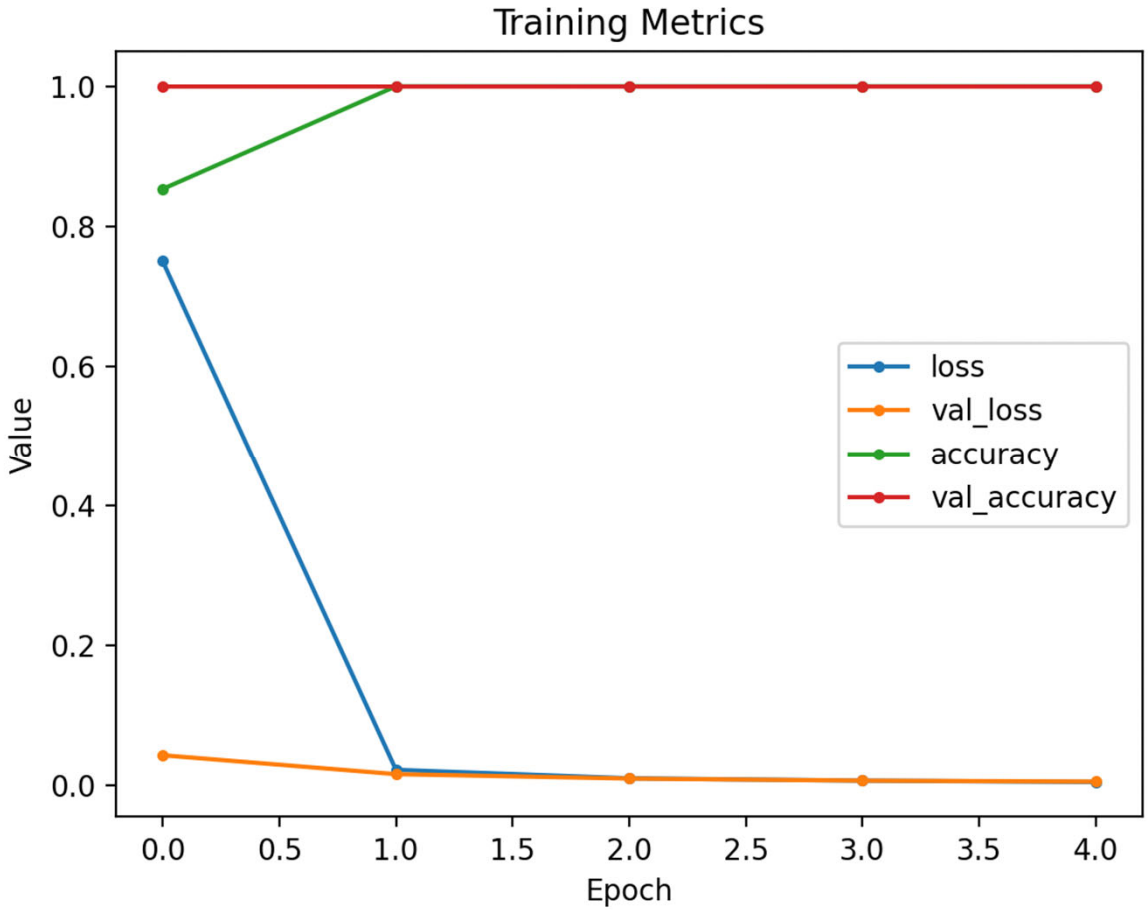
```
from torch_util import trainModel, predictModel, plotLearningCurves

# select model device (CPU/CUDA) once it is created and modified
DEVICE = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = model.to(DEVICE)

# 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)

# Train model (we'll start with  a few epochs as training without GPU acceleration will be slow)
start_time = time.time()

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

print(f"Training completed in: {time.time() - start_time:.2f}s")
```

```
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

```
# 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')
```

Transfer Learning (CNN)

- Aceleración:
  - El entrenamiento es lento ya que en cada iteración es preciso realizar la inferencia de todas las imágenes de entrenamiento en las capas convolucionales de la red VGG16.
  - Solución sin aceleración por GPU:
    - Construir un modelo de red separado (**subred1**) para las primeras capas de la red convolucional VGG16 (**Features** → **Pooling**) (*capas ya entrenadas*)
    - Construir un modelo de red separado (**subred2**) para las últimas capas de clasificación (*capas a entrenar*)
    - Precalcular los descriptores a la salida VGG16 (**subred1**) para las imágenes de entrenamiento
    - Usar estos descriptores de las imágenes de entrenamiento para entrenar la **subred2** de clasificación.
    - Testear la red completa con las imágenes de Test
  - Este método no permite aprovechar las opciones de aumento de datos mediante rotaciones y traslaciones aleatorias de PyTorch (torchvision.transforms.**Compose**())

Transfer Learning (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()

Transfer Learning(CNN)

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

- Preprocesamiento (‘*Data Augmentation*’): torchvision.transforms.**Compose**()
  - Crearemos dos conjuntos de transformaciones:
    - preprocess_vgg16 : solo contiene la normalización de imágenes de VGG16, la usaremos para el dataset de validación o la predicción
    - preprocess_train : contiene las transformaciones aleatorias de aumento de datos además de las de normalización de imágenes de VGG16 la usaremos para el datset de entrenamiento

```
# Image transform  Resize/normalize input image color (net specific) [scale*(x-mean)]
#  Download default VGG16 preprocess transformation
preprocess_vgg16 = torchvision.models.VGG16_Weights.DEFAULT.transforms()
print("VGG16 Preprocess Transform: ", preprocess_vgg16)

# Compose preprocess transformation with Data augmentation
preprocess_train = torchvision.transforms.Compose([
        # Random Data Augmentation
        torchvision.transforms.RandomHorizontalFlip(),
        torchvision.transforms.RandomVerticalFlip(),
        torchvision.transforms.RandomRotation(degrees=(0, 180)),
        torchvision.transforms.RandomAffine(degrees=(30, 70), translate=(0.1, 0.3), scale=(0.5, 0.75)),

        # VGG16 preprocess
        preprocess_vgg16
])
```

Transfer Learning(CNN)

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

- Crear Datsets de datos:
  - Cargar full_datset configurando la transformacion base: preprocess_vgg16
  - Hacer una copia de full_datset para poder cambar la transformación a preprocess_train (los subsets son referencias al dataset original).
    - Usaremos la función **deepcopy** disponible en el paquete **copy**

```
import copy

# 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)

# Copy dataset to configure transform for train_dataset with data augmentation
full_dataset_train = copy.deepcopy(full_dataset)
full_dataset_train.transform = preprocess_train
```

Transfer Learning(CNN)

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

- Crear Datsets de datos:
  - Seleccionar índices aleatorios para separar los subsets
  - Divide los índices en los conjuntos: train_indx, val_indx
  - Construye los subsets con las selecciones aleatorias
  - Crea los DataLoaders

```
# split train/validation subsets are simple references to base full_dataset
data_size = len(full_dataset)
rand_indx = torch.randperm(data_size)    # Random permutation indexes for full_dataset
train_size = int(0.8 * data_size)   # 80% train / 20% validation

train_indx,  val_indx = rand_indx[ : train_size],  rand_indx[train_size : ]

# Build Subsets with selected random permutation indexes
train_dataset = torch.utils.data.Subset(full_dataset_train, train_indx)
val_dataset = torch.utils.data.Subset(full_dataset, val_indx)

# 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)
```

Leer Datasets predefinidos 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 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 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)