



UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

Introducción a PyTorch

Alfons Juan, Jorge Civera

DSIC

Departament de Sistemes
Informàtics i Computació

Índice

1	Introducción	1
2	Tensores	3
2.1	Inicialización de un tensor	4
2.2	Atributos de un tensor	7
2.3	Operaciones sobre tensores	8
3	Conjuntos de datos y carga	15
3.1	Carga de un conjunto de datos	16
3.2	Iteración y visualización	18
3.3	Conjunto de datos definido por el usuario	20
3.4	Preparación de los datos para entrenamiento	21
4	Transformaciones	23
5	Construcción de la red neuronal	26
5.1	Obtención del dispositivo para entrenamiento	27
5.2	Definición de la clase	28
5.3	Capas del modelo	31
5.4	Parámetros del modelo	37
6	Diferenciación automática	39
6.1	Tensores, funciones y grafo computacional	40

6.2	Cálculo de gradientes	42
6.3	Inhabilitación del seguimiento de gradientes	43
6.4	Más sobre grafos computacionales	44
7	Optimización de parámetros del modelo	45
7.1	Código previo	46
7.2	Hiperparámetros	47
7.3	Bucle de optimización	48
7.4	Función de pérdida	49
7.5	Optimizador	50
7.6	Implementación completa	51
8	Grabación y carga del modelo	53
8.1	Grabación y carga de los pesos del modelo	54
8.2	Grabación y carga del modelo completo	55

1. Introducción

- ▶ **PyTorch:** librería de aprendizaje automático de código abierto basada en la librería **Torch**, usada en aplicaciones de **visión artificial** y **procesamiento de lenguaje natural**
- ▶ Desarrollada por **Facebook AI Research (FAIR)** bajo una interfaz **Python**, también se ofrece a través de un **frontend C++** para la construcción de sistemas computacionalmente optimizados
- ▶ **Tensores:** ofrece la clase **torch.Tensor** para operar con arrays multidimensionales similares a los de **NumPy** en (CPU y) **GPU**
- ▶ **Redes neuronales profundas:** facilita su construcción mediante módulos de **diferenciación automática (torch.autograd)**, **optimización (torch.optim)** y **construcción de redes (torch.nn)**.
- ▶ Presentación basada en el **tutorial para principiantes oficial:**
<https://pytorch.org/tutorials/beginner/basics/intro.html>
- ▶ Polilabs: `PYTHONPATH=$PYTHONPATH:~/asigDSIC/ETSINF/apr/mlp/pylib`

- **Reproducción del tutorial:** por cada sección, se recomienda iniciar el intérprete y copiar-pegar los ejemplos enmarcados
- El código de algunos ejemplos se proporciona en `fichero.py` para ejecutarse con `exec(open("fichero.py").read())`

MLLP
tutorgs.pdf

61,6%

► A partir de otro tensor, reteniendo shape y dtype:

```
x_ones = torch.ones_like(x_data); x_ones
```

```
tensor([[1, 1],  
        [1, 1]])
```

```
x_data.shape; x_ones.shape
```

```
torch.Size([2, 2])  
torch.Size([2, 2])
```

```
x_data.dtype; x_ones.dtype
```

```
torch.int64  
torch.int64
```

► Crea un tensor de zeros, `x_zeros`, como `x_data`

```
.....
```

```
tensor([[0, 0],  
        [0, 0]])
```

Alfons Juan, Jorge Civera 4

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Terminal

```
$ python  
Python 3.10.6 (main, Nov 2 2022, 18:53:38) [GCC 11.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> import torch  
>>> import numpy as np  
>>> torch.h  
torch.half  
torch.hamming_window(  
torch.hann_window(  
torch.hardshrink(  
torch.has_cuda  
torch.has_cudnn  
torch.has_lapack  
torch.has_mkl  
torch.has_mkldnn  
torch.has_mlc  
torch.has_openmp  
torch.has_spectral  
torch.heaviside(  
torch.hinge_embedding_loss(  
torch.histc(  
torch.histogram(  
torch.histogramdd(  
torch.hsmm(  
torch.hsplit(  
torch.hspmm(  
torch.hstack(  
torch.hub(  
torch.hypot(  
>>>  
>>> data = [[1, 2],[3, 4]]  
>>> x_data = torch.tensor(data); x_data  
tensor([[1, 2],  
        [3, 4]])  
>>>  
>>> np_array = np.array(data)  
>>> x_np = torch.from_numpy(np_array); x_np  
tensor([[1, 2],  
        [3, 4]])  
>>>  
>>> x_ones = torch.ones_like(x_data); x_ones  
tensor([[1, 1],  
        [1, 1]])  
>>>  
>>> x_data.shape; x_ones.shape  
torch.Size([2, 2])  
torch.Size([2, 2])  
>>>  
>>> x_data.dtype; x_ones.dtype  
torch.int64  
torch.int64  
>>>  
>>>  
>>> []
```

2. Tensores

- **Tutorial oficial:** cuaderno jupyter *tensorqs_tutorial*

https://pytorch.org/tutorials/beginner/basics/tensorqs_tutorial.html

- Importación de las librerías torch y numpy:

```
import torch
import numpy as np
```

- Escribe `torch.h` seguido de tabulador:

```
torch.h # y Tab
```

<code>torch.half</code>	<code>torch.has_mkldnn</code>	<code>torch.histogramdd(</code>
<code>torch.hamming_window(</code>	<code>torch.has_mlc</code>	<code>torch.hsmm(</code>
<code>torch.hann_window(</code>	<code>torch.has_openmp</code>	<code>torch.hsplitt(</code>
<code>torch.hardshrink(</code>	<code>torch.has_spectral</code>	<code>torch.hspmm(</code>
<code>torch.has_cuda</code>	<code>torch.heaviside(</code>	<code>torch.hstack(</code>
<code>torch.has_cudnn</code>	<code>torch.hinge_embedding_loss(</code>	<code>torch.hub</code>
<code>torch.has_lapack</code>	<code>torch.histc(</code>	<code>torch.hypot(</code>
<code>torch.has_mkl</code>	<code>torch.histogram(</code>	

2.1. Inicialización de un tensor

- Creación a partir de datos:

```
data = [[1, 2], [3, 4]]  
x_data = torch.tensor(data); x_data
```

```
tensor([[1, 2],  
        [3, 4]])
```

- Creación a partir de un array NumPy:

```
np_array = np.array(data)  
x_np = torch.from_numpy(np_array); x_np
```

```
tensor([[1, 2],  
        [3, 4]])
```

- ▶ A partir de otro tensor, reteniendo shape y dtype:

```
x_ones = torch.ones_like(x_data); x_ones
```

```
tensor([[1, 1],  
        [1, 1]])
```

```
x_data.shape; x_ones.shape
```

```
torch.Size([2, 2])  
torch.Size([2, 2])
```

```
x_data.dtype; x_ones.dtype
```

```
torch.int64  
torch.int64
```

- ▶ Crea un tensor de zeros, **x_zeros**, como **x_data**

```
.....
```

```
tensor([[0, 0],  
        [0, 0]])
```


► A partir de otro tensor, cambiando el dtype:

```
torch.manual_seed(23)
x_rand = torch.rand_like(x_data, dtype=torch.float)
x_rand; x_rand.dtype
```

```
tensor([[0.4283, 0.2889],
        [0.4224, 0.3571]])
torch.float32
```

► Con valores constantes o aleatorios:

```
shape = (2, 3,)
rand_tensor = torch.rand(shape)
ones_tensor = torch.ones(shape)
zeros_tensor = torch.zeros(shape)
rand_tensor; ones_tensor; zeros_tensor
```

```
tensor([[0.9577, 0.1100, 0.2933],
        [0.9205, 0.5876, 0.1299]])
tensor([[1., 1., 1.],
        [1., 1., 1.]])
tensor([[0., 0., 0.],
        [0., 0., 0.]])
```

2.2. Atributos de un tensor

► *shape, dtype y device:*

```
tensor = torch.rand(3,4)
print(f"Shape of tensor: {tensor.shape}")
print(f"Datatype of tensor: {tensor.dtype}")
print(f"Device tensor is stored on: {tensor.device}")
```

```
Shape of tensor: torch.Size([3, 4])
Datatype of tensor: torch.float32
Device tensor is stored on: cpu
```

2.3. Operaciones sobre tensores

- Más de 100 operaciones disponibles:

<https://pytorch.org/docs/stable/torch.html>

- *Copia de un tensor (creado en CPU) a GPU:*

```
if torch.cuda.is_available():  
    tensor = tensor.to('cuda')  
  
tensor.device
```

```
device(type='cuda', index=0)
```

► Indexación y troceado al estilo numpy:

```
tensor = torch.ones(4, 4)
print('First row: ', tensor[0])
print('First column: ', tensor[:, 0])
print('Last column:', tensor[:, -1])
tensor[:,1] = 0
print(tensor)
```

```
First row:  tensor([1., 1., 1., 1.])
First column:  tensor([1., 1., 1., 1.])
Last column: tensor([1., 1., 1., 1.])
tensor([[1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.]])
```

► Extrae la última fila de **tensor**:

```
.....
tensor([1., 0., 1., 1.]])
```

► Concatenación de tensores en una dimensión dada:

```
t1 = torch.cat([tensor, tensor, tensor], dim=1); t1
```

```
tensor([[1., 0., 1., 1., 1., 0., 1., 1., 1., 0., 1., 1.],  
        [1., 0., 1., 1., 1., 0., 1., 1., 1., 0., 1., 1.],  
        [1., 0., 1., 1., 1., 0., 1., 1., 1., 0., 1., 1.],  
        [1., 0., 1., 1., 1., 0., 1., 1., 1., 0., 1., 1.]])
```

► Concatena **tensor** consigo mismo por filas (en vertical):

```
.....
```

```
tensor([[1., 0., 1., 1.],  
        [1., 0., 1., 1.],  
        [1., 0., 1., 1.],  
        [1., 0., 1., 1.],  
        [1., 0., 1., 1.],  
        [1., 0., 1., 1.],  
        [1., 0., 1., 1.],  
        [1., 0., 1., 1.]])
```

► **Producto matricial:** y1, y2, y3 tendrán el mismo valor

```
y1 = tensor @ tensor.T
y2 = tensor.matmul(tensor.T)
y3 = torch.rand_like(tensor)
torch.matmul(tensor, tensor.T, out=y3)
```

```
tensor([[3., 3., 3., 3.],
        [3., 3., 3., 3.],
        [3., 3., 3., 3.],
        [3., 3., 3., 3.]])
```

► **Producto elemental:** z1, z2, z3 tendrán el mismo valor

```
z1 = tensor * tensor
z2 = tensor.mul(tensor)
z3 = torch.rand_like(tensor)
torch.mul(tensor, tensor, out=z3)
```

```
tensor([[1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.]])
```

► *Tensores de un único elemento:* conversión a número python

```
agg = tensor.sum()
print(agg, type(agg))
agg_item = agg.item()
print(agg_item, type(agg_item))
```

```
tensor(12.) <class 'torch.Tensor'>
12.0 <class 'float'>
```

► Obtén la media de **tensor** como número python:

```
.....
```

```
0.75
```

► Operaciones en línea con el sufijo “_”:

```
tensor.add_(5)
```

```
tensor([ [6., 5., 6., 6.],  
         [6., 5., 6., 6.],  
         [6., 5., 6., 6.],  
         [6., 5., 6., 6.]])
```

► Réstale 5 a tensor:

```
.....
```

```
tensor([ [1., 0., 1., 1.],  
         [1., 0., 1., 1.],  
         [1., 0., 1., 1.],  
         [1., 0., 1., 1.]])
```


► Memoria compartida entre tensores en CPU y arrays numpy:

```
t = torch.ones(5); n = t.numpy(); t; n
```

```
tensor([1., 1., 1., 1., 1.])  
array([1., 1., 1., 1., 1.], dtype=float32)
```

```
t.add_(1); n # la modificación de t afecta a n
```

```
tensor([2., 2., 2., 2., 2.])  
array([2., 2., 2., 2., 2.], dtype=float32)
```

```
n = np.ones(5)  
t = torch.from_numpy(n)  
np.add(n, 1, out=n); t # la modificación de n afecta a t
```

```
array([2., 2., 2., 2., 2.])  
tensor([2., 2., 2., 2., 2.], dtype=torch.float64)
```

3. Conjuntos de datos y carga

- ▶ **Tutorial oficial:** cuaderno jupyter *data_tutorial*

https://pytorch.org/tutorials/beginner/basics/data_tutorial.html

- ▶ Dos clases destacadas:

- ▷ ***torch.utils.data.Dataset:*** conjuntos de datos pre-cargados y definidos por el usuario

↳ ***Imágenes:*** <https://pytorch.org/vision/stable/datasets.html>

↳ ***Texto:*** <https://pytorch.org/text/stable/datasets.html>

↳ ***Audio:*** <https://pytorch.org/audio/stable/datasets.html>

- ▷ ***torch.utils.data.DataLoader:*** iterable sobre conjunto de datos

- ▶ ***API torch.utils.data:***

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

3.1. Carga de un conjunto de datos

► Librerías:

```
import torch
from torch.utils.data import Dataset
from torchvision import datasets
from torchvision.transforms import ToTensor
import matplotlib.pyplot as plt
```

► Consulta el [repositorio oficial de Fashion-MNIST](#):

▷ ¿En qué se parece a MNIST?

.....

▷ ¿Cuál es la mejor precisión con sklearn verificada?

.....

▷ ¿Cuál es la precisión humana?

.....

▷ ¿Cuál es la mejor precisión reportada?

.....

- **Carga de un conjunto de datos: FashionMNIST** con
 - ▷ **root:** directorio donde guardar los datos de train/test
 - ▷ **train:** training o test
 - ▷ **download=True:** descarga de internet si no está en root
 - ▷ **transform** y **target_transform:** transformaciones a aplicar a las características y etiquetas

```
training_data = datasets.FashionMNIST(root="data", train=True,  
    download=True, transform=ToTensor()); training_data
```

```
Dataset FashionMNIST  
  Number of datapoints: 60000  
  Root location: data  
  Split: Train  
  StandardTransform  
Transform: ToTensor()
```

```
test_data = datasets.FashionMNIST(root="data", train=False,  
    download=True, transform=ToTensor()); test_data
```

```
Dataset FashionMNIST  
  Number of datapoints: 10000  
  Root location: data  
  Split: Test  
  StandardTransform  
Transform: ToTensor()
```

3.2. Iteración y visualización

```
exec(open("data3.2.py").read())

labels_map = {
    0: "T-Shirt",
    1: "Trouser",
    2: "Pullover",
    3: "Dress",
    4: "Coat",
    5: "Sandal",
    6: "Shirt",
    7: "Sneaker",
    8: "Bag",
    9: "Ankle Boot",
}

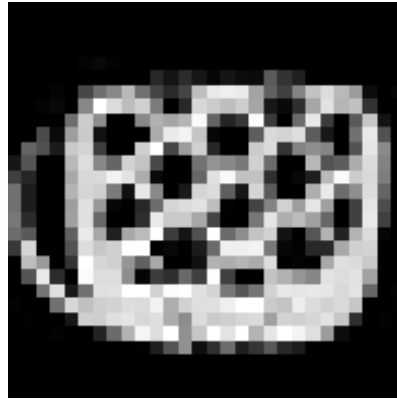
figure = plt.figure(figsize=(8, 8))
cols, rows = 3, 3
torch.manual_seed(23)
for i in range(1, cols * rows + 1):
    index = torch.randint(len(training_data), size=(1,)).item()
    img, label = training_data[index]
    figure.add_subplot(rows, cols, i)
    plt.title(labels_map[label])
    plt.axis("off")
    plt.imshow(img.squeeze(), cmap="gray")

plt.savefig('data3.2.pdf')
plt.show()
```

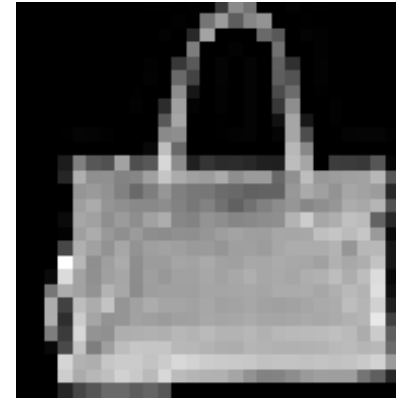
Sandal



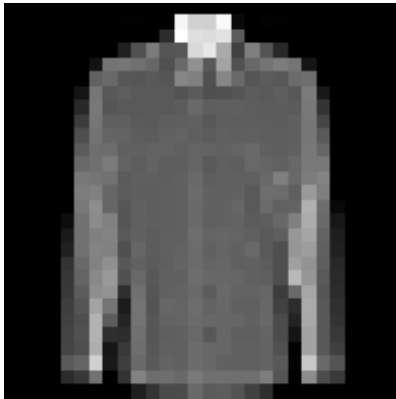
Bag



Bag



Shirt



Dress



Pullover



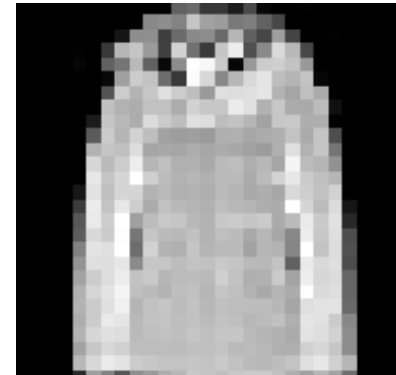
Pullover



Sandal



Coat



3.3. Conjunto de datos definido por el usuario

- ▶ *Subclase de Dataset con tres funciones:*

- ▷ *__init__*: se ejecuta una vez al instanciar un objeto Dataset
 - ▷ *__len__*: devuelve el número de muestras del dataset
 - ▷ *__getitem__*: devuelve la muestra de un índice dado
- ▶ El tutorial incluye un ejemplo genérico para imágenes

3.4. Preparación de los datos para entrenamiento

- **DataLoader**: en lugar de indexar los datos uno a uno, DataLoader los procesa en minibatches con posible barajado

```
from torch.utils.data import DataLoader
torch.manual_seed(23)
train_dataloader = DataLoader(training_data, batch_size=64,
                               shuffle=True)
test_dataloader = DataLoader(test_data, batch_size=64,
                              shuffle=True)
train_dataloader.dataset; test_dataloader.dataset
```

Dataset FashionMNIST

Number of datapoints: 60000

Root location: data

Split: Train

StandardTransform

Transform: ToTensor()

Dataset FashionMNIST

Number of datapoints: 10000

Root location: data

Split: Test

StandardTransform

Transform: ToTensor()

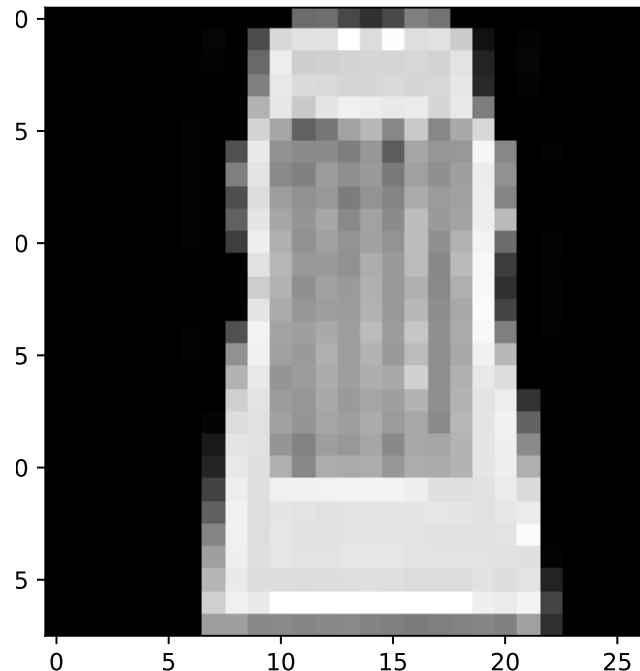
► *Iteración mediante DataLoader:* un minibatch por iteración

```
train_features, train_labels = next(iter(train_dataloader))
print(f"Feature batch shape: {train_features.size()}")
print(f"Labels batch shape: {train_labels.size()}")
img = train_features[0].squeeze()
label = train_labels[0]
plt.imshow(img, cmap="gray")
plt.savefig('data3.4.pdf'); plt.show()
print(f"Label: {label}")
```

Feature batch shape: torch.Size([64, 1, 28, 28])

Labels batch shape: torch.Size([64])

Label: 0



4. Transformaciones

- ▶ **Tutorial oficial:** cuaderno jupyter *transforms_tutorial*

https://pytorch.org/tutorials/beginner/basics/transforms_tutorial.html

- ▶ **Objetivo:** procesar datos en bruto (raw) dejándolos en un formato adecuado (processed) para entrenamiento y test de modelos
- ▶ **Conjuntos torchvision:** dos parámetros
 - ▷ **transform:** transforma las características
 - ▷ **target_transform:** transforma las etiquetas
- ▶ **Módulo torchvision.transforms:**
 - ▷ Operan sobre imágenes PIL, tensor o ambas
 - ▷ Encadenables mediante **Compose**
 - ▷ La mayoría de clases transform tienen funciones equivalentes
 - ▷ Info: <https://pytorch.org/vision/stable/transforms.html>

- **FashionMNIST:** *transform* pasa de formato PIL a tensor y *target_transform* de entero a tensor one-hot

```
import torch
from torchvision import datasets
from torchvision.transforms import ToTensor, Lambda
ds = datasets.FashionMNIST(
    root="data", train=True, download=True,
    transform=ToTensor(),
    target_transform=Lambda(lambda y: torch.zeros(10,
        dtype=torch.float).scatter_(0, torch.tensor(y), value=1)))
```

- **ToTensor()** convierte una imagen PIL en **FloatTensor** y normaliza intensidades en [0,1]
- **Lambda** aplica una función de usuario que crea un tensor nulo de talla 10 y llama a **scatter_** para asignar 1 en la posición y

▷ Lista los atributos y métodos de `ds` :

```
.....  
['__add__', '__class__', '__class_getitem__', '__delattr__',  
↳ '__dict__', '__dir__', '__doc__', '__eq__', '__format__',  
↳ '__ge__', '__getattribute__', '__getitem__', '__gt__',  
↳ '__hash__', '__init__', '__init_subclass__', '__le__',  
↳ '__len__', '__lt__', '__module__', '__ne__', '__new__',  
↳ '__orig_bases__', '__parameters__', '__reduce__',  
↳ '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',  
↳ '__slots__', '__str__', '__subclasshook__', '__weakref__',  
↳ '_check_exists', '_check_legacy_exist',  
↳ '_format_transform_repr', '_is_protocol', '_load_data',  
↳ '_load_legacy_data', '_repr_indent', 'class_to_idx',  
↳ 'classes', 'data', 'download', 'extra_repr', 'mirrors',  
↳ 'processed_folder', 'raw_folder', 'resources', 'root',  
↳ 'target_transform', 'targets', 'test_data', 'test_file',  
↳ 'test_labels', 'train', 'train_data', 'train_labels',  
↳ 'training_file', 'transform', 'transforms']
```

▷ ¿Cuál es la etiqueta de `ds.data[0]`?

```
.....  
tensor(9)
```

5. Construcción de la red neuronal

- **Tutorial oficial:** cuaderno jupyter *buildmodel_tutorial*

https://pytorch.org/tutorials/beginner/basics/buildmodel_tutorial.html

- ***torch.nn.Module*:** clase base para todos los módulos de redes

- Librerías:

```
import os
import torch
from torch import nn
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
```

5.1. Obtención del dispositivo para entrenamiento

- ▶ Entrenaremos en GPU si puede ser; si no, en CPU:

```
device = 'cuda' if torch.cuda.is_available() else 'cpu'  
print(f'Using {device} device')
```

Using cuda device

- ▶ Consulta brevemente, si no te suena:
 - ▷ Graphics processing unit (GPU)
 - ▷ Tensor Processing Unit (TPU)
 - ▷ Compute Unified Device Architecture (CUDA)

5.2. Definición de la clase

► **Red:** subclase de `nn.Module`

▷ `__init__` inicializa las capas

▷ `forward` procesa la entrada

```
exec(open("buildmodelNN.py").read())

class NeuralNetwork(nn.Module):
    def __init__(self):
        super(NeuralNetwork, self).__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(28*28, 512),
            nn.ReLU(),
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, 10),
        )
    def forward(self, x):
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)
        return logits
```

► *Instanciación y transferencia al dispositivo:*

```
torch.manual_seed(23)
model = NeuralNetwork().to(device)
print(model)
```

```
NeuralNetwork(
  (flatten): Flatten(start_dim=1, end_dim=-1)
  (linear_relu_stack): Sequential(
    (0): Linear(in_features=784, out_features=512, bias=True)
    (1): ReLU()
    (2): Linear(in_features=512, out_features=512, bias=True)
    (3): ReLU()
    (4): Linear(in_features=512, out_features=10, bias=True)
  )
)
```


- *Uso de la red:* no llamamos a `forward()` directamente!

```
torch.manual_seed(23)
X = torch.rand(1, 28, 28, device=device)
logits = model(X)
pred_probab = nn.Softmax(dim=1)(logits)
y_pred = pred_probab.argmax(1)
print(f"Predicted class: {y_pred}")
```

Predicted class: tensor([8], device='cuda:0')

- Repite el ejemplo con un tensor de unos como **x**:

```
.....
nn.Softmax(dim=1)(model(X_ones)).argmax(1)
```

tensor([4], device='cuda:0')

5.3. Capas del modelo

► *Entrada:* minibatch de 3 imágenes 28x28

```
torch.manual_seed(23)
input_image = torch.rand(3, 28, 28)
print(input_image.size())
```

```
torch.Size([3, 28, 28])
```

► Escribe la primera fila de la primera imagen del minibatch:

```
.....
tensor([0.4283, 0.2889, 0.4224, 0.3571, 0.9577, 0.1100, 0.2933, 0.9205, 0.5876,
        0.1299, 0.6729, 0.1028, 0.7876, 0.5540, 0.4653, 0.2311, 0.2214, 0.3348,
        0.4541, 0.2519, 0.6310, 0.1707, 0.3122, 0.1976, 0.5466, 0.0213, 0.9049,
        0.8444])
```

► *nn.Flatten*: convierte una imagen 28x28 en array 784D

```
flatten = nn.Flatten()  
flat_image = flatten(input_image)  
print(flat_image.size())
```

```
torch.Size([3, 784])
```

► Escribe las primeras 5 características de la primera imagen

```
.....
```

```
tensor([0.4283, 0.2889, 0.4224, 0.3571, 0.9577])
```

► *nn.Linear*: transformación lineal

```
torch.manual_seed(23)
layer1 = nn.Linear(in_features=28*28, out_features=20)
hidden1 = layer1(flat_image)
print(hidden1.size())
print(hidden1)
```

```
torch.Size([3, 20])
```

```
tensor([[ 4.7243, -0.3048,  0.3629, -0.3240,  0.0510, -0.3063,  0.0167,  0.2413,
        -0.1951,  0.1663,  0.1250, -0.2419, -0.1556,  0.5033,  0.1148, -0.4214,
        -0.1384, -0.2026, -0.2182,  0.2294],
        [ 0.0395,  4.5114, -0.1091, -0.1640, -0.0224, -0.1199,  0.3188, -0.2164,
        -0.0565, -0.0408,  0.0215, -0.3568, -0.0790,  0.2190,  0.2358, -0.6144,
        -0.1755, -0.2186,  0.0335,  0.2159],
        [ 0.3228, -0.4934,  4.9607, -0.4462,  0.2545, -0.2906,  0.4563,  0.0705,
        -0.3625, -0.1225,  0.0456, -0.0161,  0.0463,  0.0077,  0.3278, -0.4075,
         0.1262, -0.0512, -0.0390,  0.1706]], grad_fn=<AddmmBackward0>)
```

► *nn.ReLU*: activación no lineal que “apaga” negativos

```
print(f"Before ReLU:\n {hidden1}\n\n")
hidden1 = nn.ReLU()(hidden1)
print(f"After ReLU:\n {hidden1}")
```

Before ReLU:

```
tensor([[ 4.7243, -0.3048,  0.3629, -0.3240,  0.0510, -0.3063,  0.0167,  0.2413,
         -0.1951,  0.1663,  0.1250, -0.2419, -0.1556,  0.5033,  0.1148, -0.4214,
         -0.1384, -0.2026, -0.2182,  0.2294],
        [ 0.0395,  4.5114, -0.1091, -0.1640, -0.0224, -0.1199,  0.3188, -0.2164,
         -0.0565, -0.0408,  0.0215, -0.3568, -0.0790,  0.2190,  0.2358, -0.6144,
         -0.1755, -0.2186,  0.0335,  0.2159],
        [ 0.3228, -0.4934,  4.9607, -0.4462,  0.2545, -0.2906,  0.4563,  0.0705,
         -0.3625, -0.1225,  0.0456, -0.0161,  0.0463,  0.0077,  0.3278, -0.4075,
          0.1262, -0.0512, -0.0390,  0.1706]], grad_fn=<AddmmBackward0>)
```

After ReLU:

```
tensor([[4.7243, 0.0000, 0.3629, 0.0000, 0.0510, 0.0000, 0.0167, 0.2413, 0.0000,
         0.1663, 0.1250, 0.0000, 0.0000, 0.5033, 0.1148, 0.0000, 0.0000, 0.0000,
         0.0000, 0.2294],
        [0.0395, 4.5114, 0.0000, 0.0000, 0.0000, 0.0000, 0.3188, 0.0000, 0.0000,
         0.0000, 0.0215, 0.0000, 0.0000, 0.2190, 0.2358, 0.0000, 0.0000, 0.0000,
         0.0335, 0.2159],
        [0.3228, 0.0000, 4.9607, 0.0000, 0.2545, 0.0000, 0.4563, 0.0705, 0.0000,
         0.0000, 0.0456, 0.0000, 0.0463, 0.0077, 0.3278, 0.0000, 0.1262, 0.0000,
         0.0000, 0.1706]], grad_fn=<ReluBackward0>)
```

► *nn.Sequential*: contenedor de módulos ordenado

```
torch.manual_seed(23)
seq_modules = nn.Sequential(
    flatten,
    layer1,
    nn.ReLU(),
    nn.Linear(20, 10))
logits = seq_modules(input_image)
print(logits)
```

```
tensor([[ -0.0418,  0.4128,  0.1166, -0.7246,  0.1914,  0.8106,  1.0597,  0.5786,
          -0.1699,  1.1271],
        [-0.3771, -0.4748, -0.8368, -0.1844,  0.5892,  0.9324,  1.0034, -0.7360,
          -0.6787, -0.5469],
        [-0.0884, -0.1937, -0.4910,  0.5475, -0.8520,  0.1503, -0.9129, -1.0010,
          -0.3887,  0.2054]], grad_fn=<AddmmBackward0>)
```

► Clasifica el minibatch `input_image` por máximo logit:

```
.....
```

```
tensor([9, 6, 3])
```

► ***nn.Softmax***: convierte logits [-infty,infty] en probabilidades [0,1]

```
softmax = nn.Softmax(dim=1)
pred_probab = softmax(logits)
print(pred_probab)
```

```
tensor([[0.0594, 0.0936, 0.0696, 0.0300, 0.0750, 0.1394, 0.1788, 0.1105, 0.0523,
         0.1913],
        [0.0612, 0.0555, 0.0386, 0.0742, 0.1608, 0.2267, 0.2433, 0.0427, 0.0453,
         0.0516],
        [0.1097, 0.0987, 0.0733, 0.2072, 0.0511, 0.1393, 0.0481, 0.0440, 0.0812,
         0.1472]], grad_fn=<SoftmaxBackward0>)
```

► Clasifica el minibatch **input_image** por máxima probabilidad:

```
.....
```

```
tensor([9, 6, 3])
```

5.4. Parámetros del modelo

- La estructura del modelo revela las capas parametrizadas:

```
print("Model structure: ", model, "\n\n")
```

```
Model structure:  NeuralNetwork(  
  (flatten): Flatten(start_dim=1, end_dim=-1)  
  (linear_relu_stack): Sequential(  
    (0): Linear(in_features=784, out_features=512, bias=True)  
    (1): ReLU()  
    (2): Linear(in_features=512, out_features=512, bias=True)  
    (3): ReLU()  
    (4): Linear(in_features=512, out_features=10, bias=True)  
  )  
)
```

- ¿Dada una entrada, de qué dependerá la salida de la red?

.....

► Podemos acceder a ellos con `parameters()`:

```
for param in model.parameters():  
    print(f"Size: {param.size()}")
```

```
Size: torch.Size([512, 784])  
Size: torch.Size([512])  
Size: torch.Size([512, 512])  
Size: torch.Size([512])  
Size: torch.Size([10, 512])  
Size: torch.Size([10])
```

► También podemos usar `named_parameters()`:

```
for name, param in model.named_parameters():  
    print(f"Layer: {name} | Size: {param.size()}")
```

```
Layer: linear_relu_stack.0.weight | Size: torch.Size([512, 784])  
Layer: linear_relu_stack.0.bias | Size: torch.Size([512])  
Layer: linear_relu_stack.2.weight | Size: torch.Size([512, 512])  
Layer: linear_relu_stack.2.bias | Size: torch.Size([512])  
Layer: linear_relu_stack.4.weight | Size: torch.Size([10, 512])  
Layer: linear_relu_stack.4.bias | Size: torch.Size([10])
```

6. Diferenciación automática

- ▶ **Tutorial oficial:** cuaderno jupyter *autogradqs_tutorial*

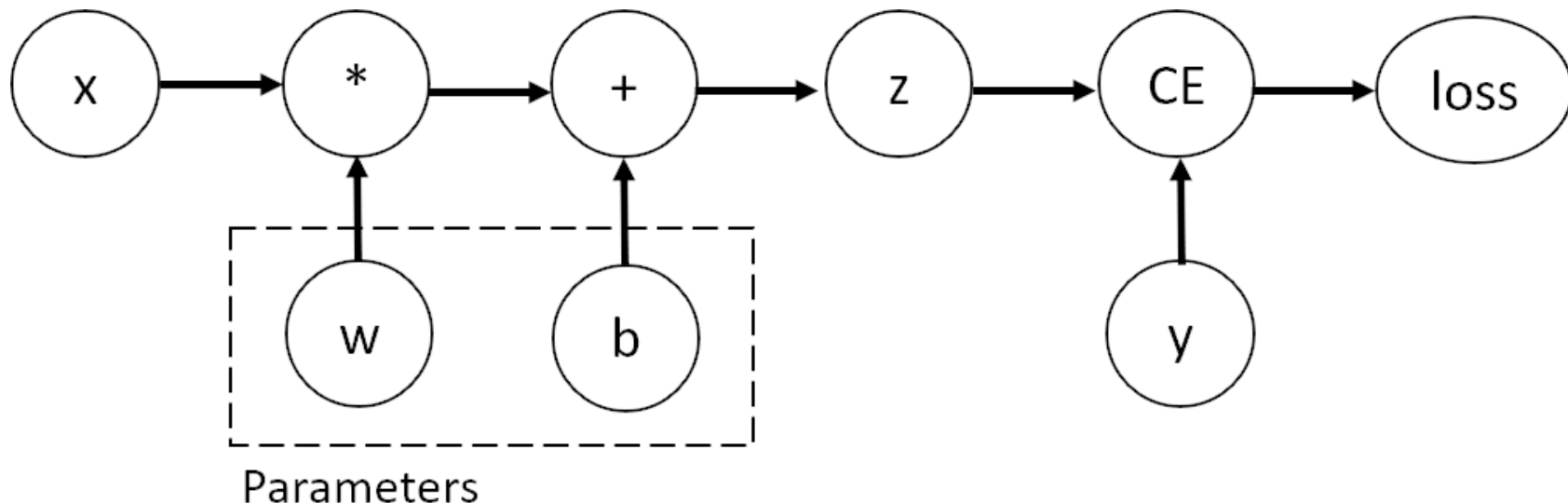
https://pytorch.org/tutorials/beginner/basics/autogradqs_tutorial.html

- ▶ **autograd:** cálculo automático del gradiente de la función de pérdida con respecto a los parámetros del modelo
 - ▷ **Backprop:** usa el gradiente para entrenar la red
 - ▷ **Grafo computacional:** representación de la red

6.1. Tensores, funciones y grafo computacional

- *Ejemplo:* red de una capa, con entrada x , parámetros w y b , y pérdida *entropía cruzada binaria*

```
import torch
x = torch.ones(5)  # input tensor
y = torch.zeros(3) # expected output
torch.manual_seed(23)
w = torch.randn(5, 3, requires_grad=True)
b = torch.randn(3, requires_grad=True)
z = torch.matmul(x, w)+b
loss = torch.nn.functional.binary_cross_entropy_with_logits(z, y)
```



- ▶ **requires_grad:** propiedad de tensores-parámetro a optimizar
- ▶ **Función de cálculo:** una función aplicada a tensores para construir un grafo computacional es un objeto de la clase **Function**
 - ▷ Permite el cálculo de la función hacia adelante (forward)
 - ▷ Permite el cálculo del gradiente de la pérdida con respecto a los parámetros
- ▶ **grad_fn:** propiedad de tensor con la función gradiente

```
print('Gradient function for z =', z.grad_fn)
print('Gradient function for loss =', loss.grad_fn)
```

```
Gradient function for z = <AddBackward0 object at 0x7f0ad92954b0>
Gradient function for loss =
↳ <BinaryCrossEntropyWithLogitsBackward0 object at
↳ 0x7f0ad939f5b0>
```

6.2. Cálculo de gradientes

- *Cálculo de gradientes:* `loss.backward()` calcula las derivadas de la pérdida con respecto a los parámetros, $\frac{\partial loss}{\partial w}$ y $\frac{\partial loss}{\partial b}$, bajo ciertos valores fijos de x e y , en `w.grad` y `b.grad`

```
loss.backward()  
print(w.grad)  
print(b.grad)
```

```
tensor([ [0.3256, 0.1307, 0.0733],  
        [0.3256, 0.1307, 0.0733],  
        [0.3256, 0.1307, 0.0733],  
        [0.3256, 0.1307, 0.0733],  
        [0.3256, 0.1307, 0.0733]])  
tensor([0.3256, 0.1307, 0.0733])
```

6.3. Inhabilitación del seguimiento de gradientes

- *Inhabilitación del seguimiento de gradientes:* para “congelar” parámetros en fine-tuning o acelerar cálculos en inferencia, evitando que tensores con `requires_grad=True` hagan seguimiento de su historia computacional a fin de calcular gradientes

► `torch.no_grad():`

```
z = torch.matmul(x, w)+b
print(z.requires_grad)
with torch.no_grad():
    z = torch.matmul(x, w)+b
print(z.requires_grad)
```

True

False

► `detach():`

```
z = torch.matmul(x, w)+b
z_det = z.detach()
print(z_det.requires_grad)
```

False

6.4. Más sobre grafos computacionales

- ▶ **DAG:** autograd mantiene un registro de datos (tensores) y todas las operaciones ejecutadas (junto con los tensores resultantes) en un grafo acíclico (DAG) de objetos **Function**
- ▶ **Cálculo automático de gradientes:** recorriendo el DAG desde las raíces (tensores de salida) a las hojas (tensores de entrada)
- ▶ **Forward:** autograd hace dos cosas simultáneamente
 - ▷ ejecuta la operación para calcular un tensor resultante
 - ▷ mantiene la función gradiente de la operación en el DAG
- ▶ **Backward:** tras **.backward()** en la raíz del DAG, autograd
 - ▷ calcula los gradientes de cada **.grad_fn**
 - ▷ mantiene la función gradiente de la operación en el DAG
 - ▷ retropropaga el error a los tensores hoja (regla de la cadena)

7. Optimización de parámetros del modelo

- **Tutorial oficial:** cuaderno jupyter *optimization_tutorial*

https://pytorch.org/tutorials/beginner/basics/optimization_tutorial.html

- **Entrenamiento:** proceso iterativo tal que, en cada época:

- ▷ el modelo predice la salida;
- ▷ calcula el error o pérdida (*loss*) de su predicción;
- ▷ obtiene las derivadas del error con respecto a sus parámetros;
- ▷ y **optimiza** los parámetros mediante descenso por gradiente.

- **Vídeo recomendado sobre el algoritmo Backprop:**

<https://www.youtube.com/watch?v=tIeHLnjs5U8>

7.1. Código previo

```
exec(open("optim7.1.py").read())

import torch
from torch import nn
from torch.utils.data import DataLoader
from torchvision import datasets
from torchvision.transforms import ToTensor, Lambda

training_data = datasets.FashionMNIST(root="data", train=True,
                                       download=True, transform=ToTensor())
test_data = datasets.FashionMNIST(root="data", train=False,
                                   download=True, transform=ToTensor())

train_dataloader = DataLoader(training_data, batch_size=64)
test_dataloader = DataLoader(test_data, batch_size=64)

class NeuralNetwork(nn.Module):
    def __init__(self):
        super(NeuralNetwork, self).__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(28*28, 512), nn.ReLU(),
            nn.Linear(512, 512), nn.ReLU(),
            nn.Linear(512, 10))
    def forward(self, x):
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)
        return logits

torch.manual_seed(23)
model = NeuralNetwork()
```

7.2. Hiperparámetros

- ▶ **Número de épocas:** número de veces a iterar sobre los datos

```
epochs = 5
```

- ▶ **Tamaño del batch:** número de muestras propagadas a través de la red antes de actualizar parámetros

```
batch_size = 64
```

- ▶ **Factor de aprendizaje:** magnitud de la actualización de parámetros en cada batch/época

```
learning_rate = 1e-3
```

- ▷ Demasiado pequeño: aprendizaje lento
- ▷ Demasiado grande: aprendizaje impredecible

7.3. Bucle de optimización

- ▶ ***Bucle de entrenamiento:*** itera sobre el conjunto de entrenamiento tratando de converger a parámetros óptimos
- ▶ ***Bucle de validación/test:*** itera sobre el conjunto de test para comprobar si el rendimiento del modelo está mejorando

7.4. Función de pérdida

- ▶ *Mean Square Loss:* `nn.MSELoss`, para regresión
- ▶ *Negative Log Likelihood:* `nn.NLLLoss`, para clasificación
- ▶ *Cross Entropy:* `nn.CrossEntropyLoss`, combina `nn.LogSoftmax` y `nn.NLLLoss`

```
loss_fn = nn.CrossEntropyLoss()
```

7.5. Optimizador

- *optimizer*: usamos SGD

```
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
```

- Tres pasos:
 - ▷ **optimizer.zero_grad()** reinicializa a cero los gradientes de los parámetros (y así evita sumas duplicadas)
 - ▷ **loss.backward()** retropropaga la pérdida de la predicción calculando sus gradientes respecto a los parámetros
 - ▷ **optimizer.step()** ajusta los parámetros mediante los gradientes hallados en retropropagación

7.6. Implementación completa

```
exec(open("optim7.6.py").read())

def train_loop(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    for batch, (X, y) in enumerate(dataloader):
        pred = model(X); loss = loss_fn(pred, y)
        optimizer.zero_grad(); loss.backward(); optimizer.step() # backprop
        if batch % 100 == 0:
            loss, current = loss.item(), batch * len(X)
            print(f"trloss: {loss:>7f} [{current:>5d}/{size:>5d}]")

def test_loop(dataloader, model, loss_fn):
    size = len(dataloader.dataset); nbatches = len(dataloader)
    teloss, correct = 0, 0
    with torch.no_grad():
        for X, y in dataloader:
            pred = model(X); teloss += loss_fn(pred, y).item()
            correct += (pred.argmax(1) == y).type(torch.float).sum().item()
    teloss /= nbatches; correct /= size
    print(f"teacc: {(100*correct):>0.1f}%, teloss: {teloss:>8f} \n")

loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
epochs = 10
for t in range(epochs):
    print(f"Epoch {t+1}\n-----")
    train_loop(train_dataloader, model, loss_fn, optimizer)
    test_loop(test_dataloader, model, loss_fn)
print("Done!")
```

Epoch 1

```
trloss: 2.311403 [ 0/60000]
trloss: 2.290950 [ 6400/60000]
trloss: 2.274235 [12800/60000]
trloss: 2.260890 [19200/60000]
trloss: 2.250175 [25600/60000]
trloss: 2.219779 [32000/60000]
trloss: 2.239611 [38400/60000]
trloss: 2.202065 [44800/60000]
trloss: 2.203018 [51200/60000]
trloss: 2.171447 [57600/60000]
teacc: 29.8%, teloss: 2.160771
```

Epoch 2

```
trloss: 2.177448 [ 0/60000]
trloss: 2.162522 [ 6400/60000]
trloss: 2.114367 [12800/60000]
trloss: 2.127496 [19200/60000]
trloss: 2.073345 [25600/60000]
trloss: 2.013937 [32000/60000]
trloss: 2.058795 [38400/60000]
trloss: 1.974498 [44800/60000]
trloss: 1.993499 [51200/60000]
trloss: 1.921251 [57600/60000]
teacc: 52.6%, teloss: 1.915133
```

Epoch 3

```
trloss: 1.948784 [ 0/60000]
trloss: 1.915269 [ 6400/60000]
trloss: 1.819419 [12800/60000]
trloss: 1.857720 [19200/60000]
trloss: 1.733476 [25600/60000]
trloss: 1.690374 [32000/60000]
trloss: 1.727686 [38400/60000]
trloss: 1.620707 [44800/60000]
trloss: 1.655109 [51200/60000]
trloss: 1.544735 [57600/60000]
teacc: 59.6%, teloss: 1.560143
```

Epoch 4

```
trloss: 1.625832 [ 0/60000]
trloss: 1.582964 [ 6400/60000]
trloss: 1.455607 [12800/60000]
trloss: 1.512096 [19200/60000]
trloss: 1.379335 [25600/60000]
trloss: 1.382386 [32000/60000]
trloss: 1.401563 [38400/60000]
trloss: 1.320725 [44800/60000]
trloss: 1.356428 [51200/60000]
trloss: 1.247718 [57600/60000]
teacc: 62.9%, teloss: 1.277477
```

Epoch 5

```
trloss: 1.354618 [ 0/60000]
trloss: 1.330229 [ 6400/60000]
trloss: 1.184364 [12800/60000]
trloss: 1.269096 [19200/60000]
trloss: 1.137335 [25600/60000]
trloss: 1.170170 [32000/60000]
trloss: 1.193289 [38400/60000]
trloss: 1.127051 [44800/60000]
trloss: 1.166408 [51200/60000]
trloss: 1.072755 [57600/60000]
teacc: 64.6%, teloss: 1.099308
```

Epoch 6

```
trloss: 1.170938 [ 0/60000]
trloss: 1.167573 [ 6400/60000]
trloss: 1.003188 [12800/60000]
trloss: 1.117761 [19200/60000]
trloss: 0.987984 [25600/60000]
trloss: 1.027530 [32000/60000]
trloss: 1.066352 [38400/60000]
trloss: 1.003134 [44800/60000]
trloss: 1.044621 [51200/60000]
trloss: 0.964757 [57600/60000]
teacc: 66.0%, teloss: 0.985917
```

Epoch 7

```
trloss: 1.046596 [ 0/60000]
trloss: 1.063737 [ 6400/60000]
trloss: 0.880441 [12800/60000]
trloss: 1.018553 [19200/60000]
trloss: 0.895659 [25600/60000]
trloss: 0.929185 [32000/60000]
trloss: 0.984901 [38400/60000]
trloss: 0.922667 [44800/60000]
trloss: 0.961700 [51200/60000]
trloss: 0.893683 [57600/60000]
teacc: 67.3%, teloss: 0.910060
```

Epoch 8

```
trloss: 0.956910 [ 0/60000]
trloss: 0.993390 [ 6400/60000]
trloss: 0.793749 [12800/60000]
trloss: 0.949914 [19200/60000]
trloss: 0.835471 [25600/60000]
trloss: 0.858853 [32000/60000]
trloss: 0.928901 [38400/60000]
trloss: 0.868920 [44800/60000]
trloss: 0.902640 [51200/60000]
trloss: 0.843204 [57600/60000]
teacc: 68.5%, teloss: 0.856261
```

Epoch 9

```
trloss: 0.888783 [ 0/60000]
trloss: 0.941961 [ 6400/60000]
trloss: 0.729432 [12800/60000]
trloss: 0.900020 [19200/60000]
trloss: 0.793433 [25600/60000]
trloss: 0.806567 [32000/60000]
trloss: 0.887255 [38400/60000]
trloss: 0.831703 [44800/60000]
trloss: 0.858814 [51200/60000]
trloss: 0.804780 [57600/60000]
teacc: 69.7%, teloss: 0.815984
```

Epoch 10

```
trloss: 0.834625 [ 0/60000]
trloss: 0.901355 [ 6400/60000]
trloss: 0.679746 [12800/60000]
trloss: 0.862242 [19200/60000]
trloss: 0.762120 [25600/60000]
trloss: 0.766526 [32000/60000]
trloss: 0.853988 [38400/60000]
trloss: 0.804405 [44800/60000]
trloss: 0.824825 [51200/60000]
trloss: 0.774202 [57600/60000]
teacc: 70.8%, teloss: 0.784274
```

Done!

8. Grabación y carga del modelo

- **Tutorial oficial:** cuaderno jupyter *saveloadrun_tutorial*

https://pytorch.org/tutorials/beginner/basics/saveloadrun_tutorial.html

- **torchvision.models:** subpaquete de torchvision con definiciones de modelos para diferentes tareas de visión

```
import torch
import torchvision.models as models
dir(models)
```

```
['AlexNet', 'ConvNeXt', 'DenseNet', 'EfficientNet', 'GoogLeNet', 'GoogLeNetOutputs', 'Inception3',
↳ 'InceptionOutputs', 'MNASNet', 'MobileNetV2', 'MobileNetV3', 'RegNet', 'ResNet',
↳ 'ShuffleNetV2', 'SqueezeNet', 'VGG', 'VisionTransformer', '_GoogLeNetOutputs',
↳ '_InceptionOutputs', '__builtins__', '__cached__', '__doc__', '__file__', '__loader__',
↳ '__name__', '__package__', '__path__', '__spec__', '_utils', 'alexnet', 'convnext',
↳ 'convnext_base', 'convnext_large', 'convnext_small', 'convnext_tiny', 'densenet',
↳ 'densenet121', 'densenet161', 'densenet169', 'densenet201', 'detection', 'efficientnet',
↳ 'efficientnet_b0', 'efficientnet_b1', 'efficientnet_b2', 'efficientnet_b3', 'efficientnet_b4',
↳ 'efficientnet_b5', 'efficientnet_b6', 'efficientnet_b7', 'feature_extraction', 'googlenet',
↳ 'inception', 'inception_v3', 'mnasnet', 'mnasnet0_5', 'mnasnet0_75', 'mnasnet1_0',
↳ 'mnasnet1_3', 'mobilenet', 'mobilenet_v2', 'mobilenet_v3_large', 'mobilenet_v3_small',
↳ 'mobilenetv2', 'mobilenetv3', 'optical_flow', 'quantization', 'regnet', 'regnet_x_16gf',
↳ 'regnet_x_1_6gf', 'regnet_x_32gf', 'regnet_x_3_2gf', 'regnet_x_400mf', 'regnet_x_800mf',
↳ 'regnet_x_8gf', 'regnet_y_128gf', 'regnet_y_16gf', 'regnet_y_1_6gf', 'regnet_y_32gf',
↳ 'regnet_y_3_2gf', 'regnet_y_400mf', 'regnet_y_800mf', 'regnet_y_8gf', 'resnet', 'resnet101',
↳ 'resnet152', 'resnet18', 'resnet34', 'resnet50', 'resnext101_32x8d', 'resnext50_32x4d',
↳ 'segmentation', 'shufflenet_v2_x0_5', 'shufflenet_v2_x1_0', 'shufflenet_v2_x1_5',
↳ 'shufflenet_v2_x2_0', 'shufflenetv2', 'squeezenet', 'squeezenet1_0', 'squeezenet1_1', 'vgg',
↳ 'vgg11', 'vgg11_bn', 'vgg13', 'vgg13_bn', 'vgg16', 'vgg16_bn', 'vgg19', 'vgg19_bn', 'video',
↳ 'vision_transformer', 'vit_b_16', 'vit_b_32', 'vit_l_16', 'vit_l_32', 'wide_resnet101_2',
↳ 'wide_resnet50_2']
```


8.1. Grabación y carga de los pesos del modelo

- ***torch.save***: para grabar los parámetros de un modelo en su diccionario de estado `state_dict`

```
_____ son 528M de pesos! _____  
model = models.vgg16(pretrained=True)  
torch.save(model.state_dict(), 'model_weights.pth')
```

- ***load_state_dict()***: carga los pesos de un modelo ya instanciado

```
model = models.vgg16() # we do not specify pretrained=True,  
↳ i.e. do not load default weights  
model.load_state_dict(torch.load('model_weights.pth'))  
model.eval() # to set the dropout and batch normalization  
↳ layers to evaluation mode
```

8.2. Grabación y carga del modelo completo

- ***torch.save:*** para graba el modelo completo y no solo su diccionario de estado

```
torch.save(model, 'model.pth')
```

- ***torch.load:*** para cargar el modelo completo

```
model = torch.load('model.pth')
```