TL01 pytorch

Índice

1 Introducción

pytorch: librería de aprendizaje profundo muy popular

Web: https://pytorch.org

- **Learn:** Get Started, Tutorials, Learn the Basics, Recipes, YouTube, Webinars
- **Community:** Landscape, Join the Ecosystem, Community Hub, Forums, Developer Resources, Contributor Awards, Community Events, Ambassadors
- **Projects:** PyTorch, vLLM, DeepSpeed, Host Your Project
- **Docs:** PyTorch, Domains
- **Blog & News:** Blog, Announcements, Case Studies, Events, Newsletter
- **About:** Foundation, Members, Governing Board, Technical Advisory Council, Cloud Credit Program, Staff, Contact

Github: https://github.com/pytorch/pytorch

- **More About PyTorch:** A GPU-Ready Tensor Library, Dynamic Neural Networks: Tape-Based Autograd, Python First, Imperative Experiences, Fast and Lean, Extensions Without Pain
- **Installation:** Binaries, From Source, Docker Image, Building the Documentation, Previous Versions
- **Getting Started:** Tutorials, Examples, API, Glossary
- **Resources:** PyTorch.org, Tutorials, Examples, Models, Courses, Twitter, Blog, YouTube
- **Communication:** Forums, GitHub Issues, Slack, Newsletter, Facebook Page, Brand Guidelines
- **Releases and Contributing:** three minor releases a year, filing an issue, Contribution page, Release page
- **The Team:** Soumith Chintala, Gregory Chanan, Dmytro Dzhulgakov, Edward Yang, and Nikita Shulga
- **License:** BSD-style (Berkeley Software Distribution), código abierto

Objetivo: familiarizarse con pytorch mediante los tutoriales **learn the basics**

2 Instalación

pip3 install torch torchvision torchaudio --index-url https://download.pytorch.org/whl/cu128

PyTorch Build	Stable (2.7.1)			Preview (Nightly)	
Your OS	Linux	Mac		Windows	
Package	Pip	LibTorch		Source	
Language	Python			C++ / Java	
Compute Platform	CUDA 11.8	CUDA 12.6	CUDA 12.8	ROCm 6.4	CPU
Run this Command:	pip3 install --pre torch torchvision torchaudio --index-url https://download.pytorch.org/whl/nightly/cu128				

In [ ]:
```python
import torch; print(torch.rand(1, 3), torch.__version__, torch.cuda.is_available())
```

```
tensor([[0.2587, 0.1582, 0.6875]]) 2.7.1+cu128 True
```

3 Quickstart

Quickstart: guía rápida para estudiantes ya familiarizados con otras librerías aprendizaje profundo

Datos: `torch.utils.data.DataLoader` y `torch.utils.data.Dataset`

In [ ]:
```python
import torch
from torch import nn
from torch.utils.data import DataLoader
from torchvision import datasets
from torchvision.transforms import ToTensor
```

`torchvision.datasets`: https://docs.pytorch.org/vision/stable/datasets.html

In [ ]:
```python
# Download training data from open datasets.
training_data = datasets.FashionMNIST(root="data", train=True, download=True, transform=ToTensor())
# Download test data from open datasets.
test_data = datasets.FashionMNIST(root="data", train=False, download=True, transform=ToTensor())
```

In [ ]:
```python
batch_size = 64
# Create data loaders.
train_dataloader = DataLoader(training_data, batch_size=batch_size)
test_dataloader = DataLoader(test_data, batch_size=batch_size)
for X, y in test_dataloader:
    print(f"Shape of X [N, C, H, W]: {X.shape}")
    print(f"Shape of y: {y.shape} {y.dtype}")
    break
```

```
Shape of X [N, C, H, W]: torch.Size([64, 1, 28, 28])
Shape of y: torch.Size([64]) torch.int64
```

Creación de modelos: `nn.Module`

In [ ]:
```
device = torch.accelerator.current_accelerator().type if torch.accelerator.is_available() else "cpu"
print(f"Using {device} device")
# Define model
class NeuralNetwork(nn.Module):
    def __init__(self):
        super().__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
model = NeuralNetwork().to(device)
print(model)
```

```
Using cuda device
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)
  )
)
```

Pérdida: https://pytorch.org/docs/stable/nn.html#loss-functions

In [ ]:
```
loss_fn = nn.CrossEntropyLoss()
```

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

In [ ]:
```
optimizer = torch.optim.SGD(model.parameters(), lr=1e-3)
```

Entrenamiento: iteración backprop, esto es, pasos forward y backward

In [ ]:
```
def train(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    model.train()
    for batch, (X, y) in enumerate(dataloader):
        X, y = X.to(device), y.to(device)

        # Compute prediction error
        pred = model(X)
        loss = loss_fn(pred, y)

        # Backpropagation
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()

        if batch % 100 == 0:
            loss, current = loss.item(), (batch + 1) * len(X)
            print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")
```

Test: estimación del rendimiento teórico con datos de test

In [ ]:
```
def test(dataloader, model, loss_fn):
    size = len(dataloader.dataset)
    num_batches = len(dataloader)
    model.eval()
    test_loss, correct = 0, 0
    with torch.no_grad():
        for X, y in dataloader:
            X, y = X.to(device), y.to(device)
            pred = model(X)
            test_loss += loss_fn(pred, y).item()
            correct += (pred.argmax(1) == y).type(torch.float).sum().item()
    test_loss /= num_batches
    correct /= size
    print(f"Test Error: \n Accuracy: {(100*correct):>0.1f}%, Avg loss: {test_loss:>8f} \n")
```

Backprop: con un número de épocas dado

In [ ]:
```
epochs = 1
for t in range(epochs):
    print(f"Epoch {t+1}\n-------------------------------")
    train(train_dataloader, model, loss_fn, optimizer)
    test(test_dataloader, model, loss_fn)
```

```
Epoch 1
-------------------------------
loss: 2.311589  [   64/60000]
loss: 2.292657  [ 6464/60000]
loss: 2.276898  [12864/60000]
loss: 2.262604  [19264/60000]
loss: 2.249676  [25664/60000]
loss: 2.226302  [32064/60000]
loss: 2.226686  [38464/60000]
loss: 2.198057  [44864/60000]
loss: 2.186396  [51264/60000]
loss: 2.164053  [57664/60000]
Test Error: 
 Accuracy: 47.5%, Avg loss: 2.155500
```

Grabación y carga de modelos:

In [ ]:
```
torch.save(model.state_dict(), "model.pth")
print("Saved PyTorch Model State to model.pth")
```

```
Saved PyTorch Model State to model.pth
```

In [ ]:
```
model = NeuralNetwork().to(device)
model.load_state_dict(torch.load("model.pth", weights_only=True))
```

Out[ ]:
```
<All keys matched successfully>
```

In [ ]:
```
classes = [
    "T-shirt/top",
    "Trouser",
    "Pullover",
    "Dress",
    "Coat",
    "Sandal",
    "Shirt",
    "Sneaker",
    "Bag",
    "Ankle boot",
]

model.eval()
x, y = test_data[0][0], test_data[0][1]
with torch.no_grad():
    x = x.to(device)
    pred = model(x)
    predicted, actual = classes[pred[0].argmax(0)], classes[y]
    print(f'Predicted: "{predicted}", Actual: "{actual}"')
```

```
Predicted: "Ankle boot", Actual: "Ankle boot"
```

4 Learn the Basics: Tensors

In [ ]:
```
import torch; import numpy as np
```

Inicialización directamente a partir de datos:

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

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

Inicialización desde un array NumPy:

In [ ]:
```
np_array = np.array(data)
x_np = torch.from_numpy(np_array)
print(x_np)
```

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

Inicialización desde otro tensor:

In [ ]:
```
x_ones = torch.ones_like(x_data) # retains the properties of x_data
print(f"Ones Tensor: \n {x_ones} \n")

x_rand = torch.rand_like(x_data, dtype=torch.float) # overrides the datatype of x_data
print(f"Random Tensor: \n {x_rand} \n")
```

```
Ones Tensor:
 tensor([[1, 1],
        [1, 1]])

Random Tensor:
 tensor([[0.4684, 0.9267],
        [0.6666, 0.2466]])
```

Inicialización con valores constantes o aleatorios:

In [ ]:
```
shape = (2,3,)
rand_tensor = torch.rand(shape)
ones_tensor = torch.ones(shape)
zeros_tensor = torch.zeros(shape)
print(f"Random Tensor: \n {rand_tensor}")
print(f"Ones Tensor: \n {ones_tensor}")
print(f"Zeros Tensor: \n {zeros_tensor}")
```

```
Random Tensor:
 tensor([[0.7451, 0.4902, 0.2860],
        [0.3905, 0.7659, 0.6313]])
Ones Tensor:
 tensor([[1., 1., 1.],
        [1., 1., 1.]])
Zeros Tensor:
 tensor([[0., 0., 0.],
        [0., 0., 0.]])
```

Atributos de un tensor:

In [ ]:
```
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
```

Movimiento de un tensor al acelerador (GPU) actual si está disponible:

In [ ]:
```
if torch.accelerator.is_available():
    tensor = tensor.to(torch.accelerator.current_accelerator())
print(f"Device tensor is stored on: {tensor.device}")
```

```
Device tensor is stored on: cuda:0
```

Indexación y recorte al estilo numpy:

In [ ]:
```
tensor = torch.ones(4, 4)
print(f"First row: {tensor[0]}")
print(f"First column: {tensor[:, 0]}")
print(f"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.]])
```

Concatenación de tensores:

In [ ]:
```
t1 = torch.cat([tensor, tensor, tensor], dim=1)
print(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.]])
```

Multiplicación de matrices:

In [ ]:
```
y1 = tensor @ tensor.T
y2 = tensor.matmul(tensor.T)

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

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

Multiplicación de matrices elemento a elemento:

```
In [ ]: z1 = tensor * tensor
        z2 = tensor.mul(tensor)
        z3 = torch.rand_like(tensor)
        torch.mul(tensor, tensor, out=z3)

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

Operaciones in-place: se recomienda no usarlas

```
In [ ]: tensor.add_(5)

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

Memoria compartida: de tensores en CPU y arrays NumPy

```
In [ ]: t = torch.ones(5); n = t.numpy(); print(f"t: {t}  n: {n}")

t: tensor([1., 1., 1., 1., 1.])  n: [1. 1. 1. 1. 1.]

In [ ]: t.add_(1); print(f"t: {t}  n: {n}")

t: tensor([2., 2., 2., 2., 2.])  n: [2. 2. 2. 2. 2.]

In [ ]: n = np.ones(5); t = torch.from_numpy(n); print(f"t: {t}  n: {n}")

t: tensor([1., 1., 1., 1., 1.], dtype=torch.float64)  n: [1. 1. 1. 1. 1.]

In [ ]: np.add(n, 1, out=n); print(f"t: {t}  n: {n}")

t: tensor([2., 2., 2., 2., 2.], dtype=torch.float64)  n: [2. 2. 2. 2. 2.]
```

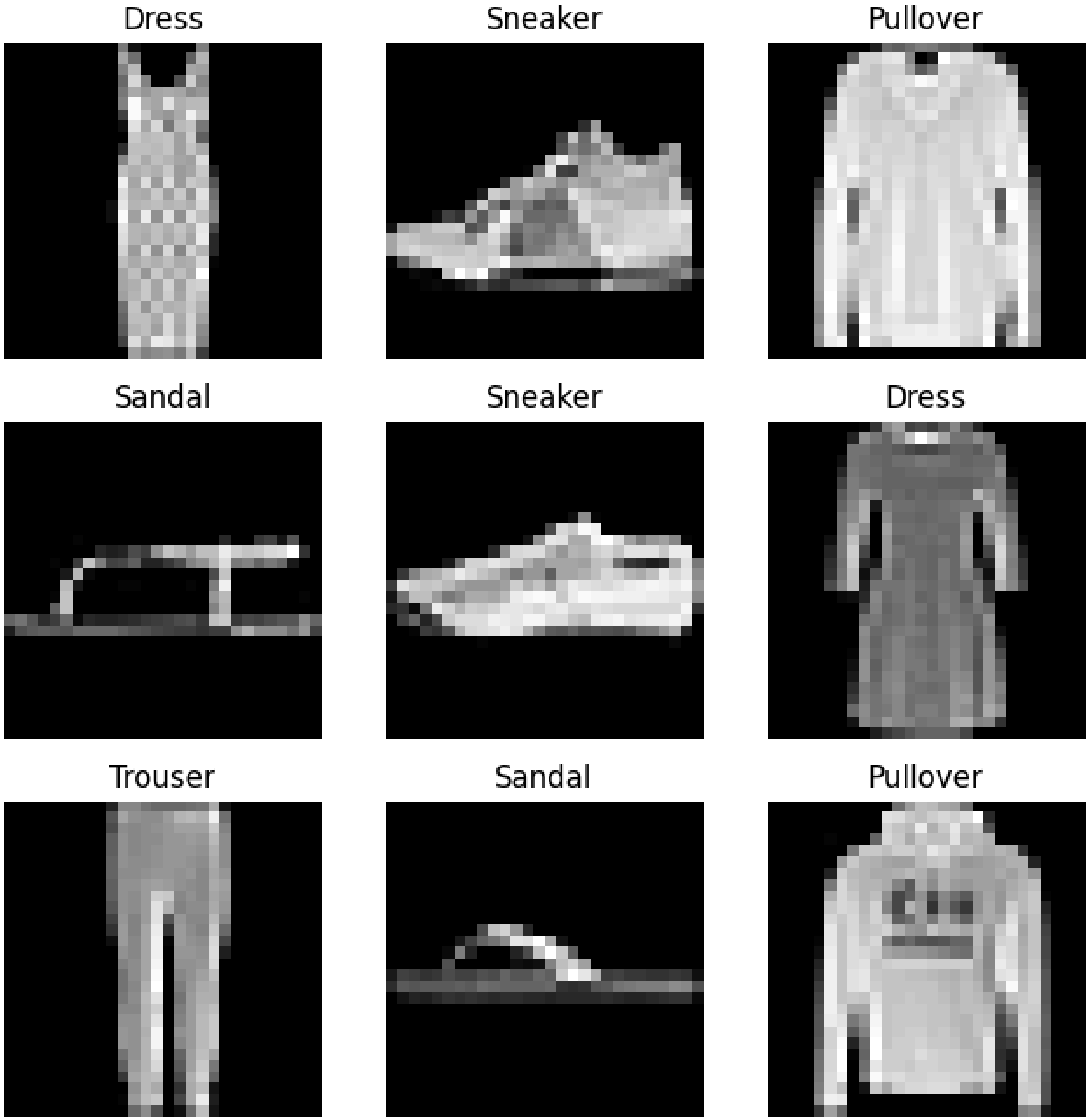
5 Learn the Basics: Datasets & DataLoaders

Lectura de un dataset:

In [ ]:
```
import torch
from torch.utils.data import Dataset
from torchvision import datasets
from torchvision.transforms import ToTensor
import matplotlib.pyplot as plt
training_data = datasets.FashionMNIST(root="data", train=True, download=True, transform=ToTensor())
test_data = datasets.FashionMNIST(root="data", train=False, download=True, transform=ToTensor())
```

Iteración y visualización del dataset:

In [ ]:
```
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
for i in range(1, cols * rows + 1):
    sample_idx = torch.randint(len(training_data), size=(1,)).item()
    img, label = training_data[sample_idx]
    figure.add_subplot(rows, cols, i)
    plt.title(labels_map[label])
    plt.axis("off")
    plt.imshow(img.squeeze(), cmap="gray")
plt.show()
```


Dress
Sneaker
Pullover
Sandal
Sneaker
Dress
Trouser
Sandal
Pullover

Creación de un dataset personalizado:

In [ ]:
```python
import os
import pandas as pd
from torchvision.io import decode_image

class CustomImageDataset(Dataset):
    def __init__(self, annotations_file, img_dir, transform=None, target_transform=None):
        self.img_labels = pd.read_csv(annotations_file)
        self.img_dir = img_dir
        self.transform = transform
        self.target_transform = target_transform

    def __len__(self):
        return len(self.img_labels)

    def __getitem__(self, idx):
        img_path = os.path.join(self.img_dir, self.img_labels.iloc[idx, 0])
        image = decode_image(img_path)
        label = self.img_labels.iloc[idx, 1]
        if self.transform:
            image = self.transform(image)
        if self.target_transform:
            label = self.target_transform(label)
        return image, label
```

`__init__`: instancia el objeto `Dataset`

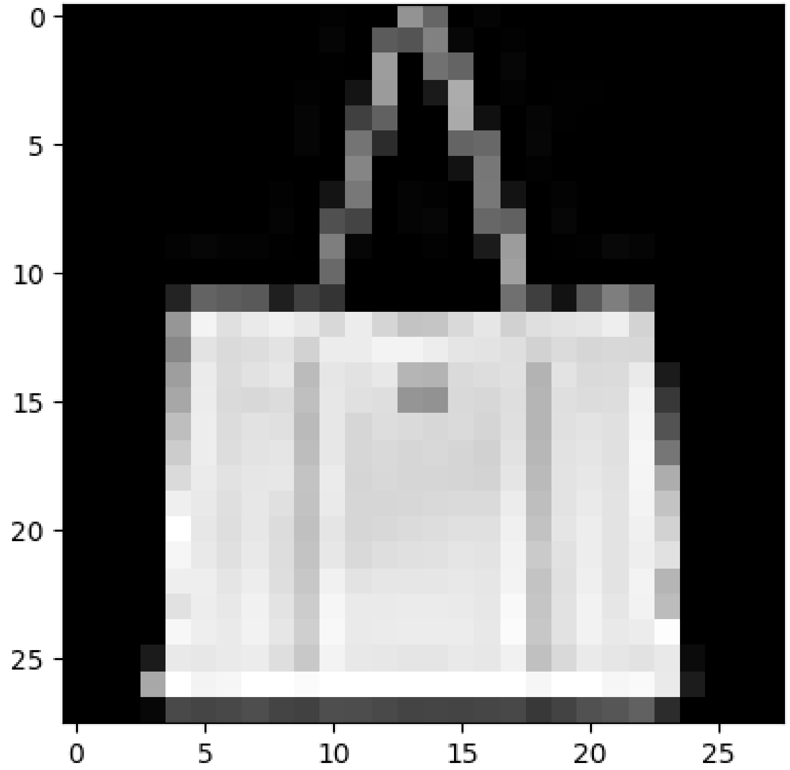
`__len__`: devuelve el número de datos del dataset

`__getitem__`: lee el dato de índice dado

Preparación de los datos para entrenamiento con DataLoaders:

```
In [ ]: from torch.utils.data import DataLoader
        train_dataloader = DataLoader(training_data, batch_size=64, shuffle=True)
        test_dataloader = DataLoader(test_data, batch_size=64, shuffle=True)

        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.show(); print(f"Label: {label}")
```

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




```
Label: 8
```

6 Learn the Basics: Transforms

`transform` y `target_transform`:

In [ ]:
```
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))
)
```

In [ ]:
```
for X, y in ds:
    print(f"Shape of X [N, C, H, W]: {X.shape}")
    print(f"Shape of y: {y.shape} {y.dtype}")
    break
```

```
Shape of X [N, C, H, W]: torch.Size([1, 28, 28])
Shape of y: torch.Size([10]) torch.float32
```

7 Learn the Basics: Build the Neural Network

Dispositivo para entrenamiento:

In [ ]:
```
import os; import torch; from torch import nn
from torch.utils.data import DataLoader; from torchvision import datasets, transforms
device = torch.accelerator.current_accelerator().type if torch.accelerator.is_available() else "cpu"
print(f"Using {device} device")
```

```
Using cuda device
```

Clase de la red: instancia de `nn.Module`

In [ ]:
```
class NeuralNetwork(nn.Module):
    def __init__(self):
        super().__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
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 del modelo: no hay que llamar al `forward` directamente

In [ ]:
```
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([3], device='cuda:0')
```

Capas del modelo: veamos cómo procesa un batch de 3 imágenes

In [ ]:
```
input_image = torch.rand(3,28,28)
print(input_image.size())
```

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

`nn.Flatten`: transforma cada imagen 28x28 en un vector de 784 dimensiones

In [ ]:
```
flatten = nn.Flatten()
flat_image = flatten(input_image)
print(flat_image.size())
```

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

`nn.Linear`: transforma linealmente la entrada con sus pesos y sesgos

In [ ]:
```
layer1 = nn.Linear(in_features=28*28, out_features=20)
hidden1 = layer1(flat_image)
print(hidden1.size())
```

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

`nn.ReLU`: transforma no-linealmente su entrada elemento a elemento

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

```
Before ReLU: tensor([[-0.1052,  0.7191,  0.1821, -0.3277,  0.3726,  0.1996, -0.2306,  0.0738,
         -0.8201, -0.5122,  0.0778,  0.0681, -0.6008, -0.7486, -0.1355, -0.4302,
          0.0179, -0.0800,  0.6629, -0.1785],
        [-0.0785,  0.4305,  0.2336,  0.0525,  0.3584, -0.0858,  0.3992,  0.4696,
         -0.3806, -0.7054, -0.1533, -0.0043, -0.5701, -0.3204, -0.0260, -0.5169,
          0.0939, -0.2174,  0.9401,  0.0230],
        [-0.3801,  0.5977,  0.2657, -0.3087, -0.1592,  0.3589, -0.2310, -0.1771,
         -0.7037, -0.5609, -0.0054, -0.3052, -0.5835, -0.4569, -0.2705, -0.5054,
          0.2032, -0.0107,  0.7808,  0.1818]], grad_fn=<AddmmBackward0>)


After ReLU: tensor([[0.0000, 0.7191, 0.1821, 0.0000, 0.3726, 0.1996, 0.0000, 0.0738, 0.0000,
         0.0000, 0.0778, 0.0681, 0.0000, 0.0000, 0.0000, 0.0000, 0.0179, 0.0000,
         0.6629, 0.0000],
        [0.0000, 0.4305, 0.2336, 0.0525, 0.3584, 0.0000, 0.3992, 0.4696, 0.0000,
         0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0939, 0.0000,
         0.9401, 0.0230],
        [0.0000, 0.5977, 0.2657, 0.0000, 0.0000, 0.3589, 0.0000, 0.0000, 0.0000,
         0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.2032, 0.0000,
         0.7808, 0.1818]], grad_fn=<ReluBackward0>)
```

`nn.Sequential`: contenedor ordenado de módulos

In [ ]:
```
seq_modules = nn.Sequential(
    flatten,
    layer1,
    nn.ReLU(),
    nn.Linear(20, 10)
)
input_image = torch.rand(3,28,28)
logits = seq_modules(input_image)
```

`nn.Softmax`: `dim` indica la dimensión a lo largo de la cual los valores deben sumar uno

In [ ]:
```
softmax = nn.Softmax(dim=1)
pred_probab = softmax(logits)
```

Parámetros del modelo: mediante `parameters()` y `named_parameters()`

In [ ]:
```
print(f"Model structure: {model}\n\n")
for name, param in model.named_parameters():
    print(f"Layer: {name} | Size: {param.size()} | Values : {param[:2]} \n")
```

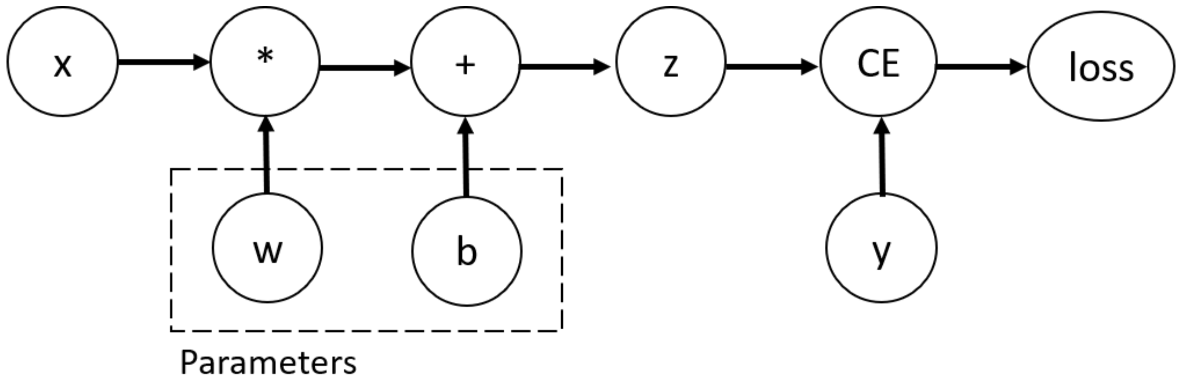
```
Model structure: NeuralNetwork(
  (flatten): Flatten(start_dim=1, end_dim=-1)
  (linear_relu_stack): Sequential(
    (0): Linear(in_features=784, out_features=512, bias=True)
    (1): ReLU()
    (2): Linear(in_features=512, out_features=512, bias=True)
    (3): ReLU()
    (4): Linear(in_features=512,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


Layer: linear_relu_stack.0.weight | Size: torch.Size([512, 784]) | Values : tensor([[ 0.0329,  0.0306, -0.0094,
...,  0.0007, -0.0146, -0.0047],
        [-0.0113,  0.0190, -0.0276,  ..., -0.0128,  0.0048, -0.0032]],
       device='cuda:0', grad_fn=<SliceBackward0>)

Layer: linear_relu_stack.0.bias | Size: torch.Size([512]) | Values : tensor([-0.0336, -0.0296], device='cuda:0', gra
d_fn=<SliceBackward0>)

Layer: linear_relu_stack.2.weight | Size: torch.Size([512, 512]) | Values : tensor([[ 0.0033, -0.0052,  0.0351,
..., -0.0437, -0.0233,  0.0184],
        [ 0.0422,  0.0076, -0.0153,  ..., -0.0122, -0.0086, -0.0186]],
       device='cuda:0', grad_fn=<SliceBackward0>)

Layer: linear_relu_stack.2.bias | Size: torch.Size([512]) | Values : tensor([ 0.0413, -0.0377], device='cuda:0', gra
d_fn=<SliceBackward0>)

Layer: linear_relu_stack.4.weight | Size: torch.Size([10, 512]) | Values : tensor([[-0.0013, -0.0241, -0.0068,  ...,
0.0153,  0.0208, -0.0422],
        [ 0.0175, -0.0135, -0.0155,  ..., -0.0078, -0.0262,  0.0253]],
       device='cuda:0', grad_fn=<SliceBackward0>)

Layer: linear_relu_stack.4.bias | Size: torch.Size([10]) | Values : tensor([-0.0150, -0.0217], device='cuda:0', grad
_fn=<SliceBackward0>)
```

8 Learn the Basics: Automatic Differentiation

`torch.autograd` : cálculo automático del gradiente de cualquier grafo computacional

In [ ]:
```
import torch; x = torch.ones(5); y = torch.zeros(3) # input ane expected tensors
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)
```

Tensores, funciones y grafo computacional:



`requires_grad` : propiedad de tensor que fijamos a `True` para los parámetros

`Function` : clase de la función representada por un grafo computacional, capaz de ejecutar los pasos forward y backward

`grad_fn` : propiedad de tensor con la función a aplicar para ejecutar el backward

In [ ]:
```
print(f"Gradient function for z = {z.grad_fn}")
print(f"Gradient function for loss = {loss.grad_fn}")
```

```
Gradient function for z = <AddBackward0 object at 0x74207ca49de0>
Gradient function for loss = <BinaryCrossEntropyWithLogitsBackward0 object at 0x74207ca49de0>
```

Cálculo de gradientes: `loss.backward()` y gradientes en `w.grad` y `b.grad`

In [ ]:
```
loss.backward(); print(w.grad); print(b.grad)
```

```
tensor([[0.0234, 0.2103, 0.3298],
        [0.0234, 0.2103, 0.3298],
        [0.0234, 0.2103, 0.3298],
        [0.0234, 0.2103, 0.3298],
        [0.0234, 0.2103, 0.3298]])
tensor([0.0234, 0.2103, 0.3298])
```

- Propiedad `grad` solo en nodos hoja con `requires_grad=True`
- `retain_graph=True` para hacer más de una llamada `backward` en el mismo grafo

Deshabilitación del seguimiento de gradientes: con `torch.no_grad()` o `detach()` (para congelar parámetros o eficiencia en inferencia)

In [ ]:
```
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
```

In [ ]:
```
z = torch.matmul(x, w)+b; z_det = z.detach(); print(z_det.requires_grad)
```

```
False
```

9 Learn the Basics: Optimization

Código prerrequisito:

In [ ]:
```
import torch; from torch import nn
from torch.utils.data import DataLoader
from torchvision import datasets
from torchvision.transforms import ToTensor

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().__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

model = NeuralNetwork()
```

Hiperparámetros:

```
In [ ]: learning_rate = 1e-3
        batch_size = 64
        epochs = 5
```

Bucle de optimización:

```
In [ ]: loss_fn = nn.CrossEntropyLoss()
        optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
```

```
In [ ]: def train_loop(dataloader, model, loss_fn, optimizer):
            size = len(dataloader.dataset)
            # Set the model to training mode - important for batch normalization and dropout layers
            # Unnecessary in this situation but added for best practices
            model.train()
            for batch, (X, y) in enumerate(dataloader):
                # Compute prediction and loss
                pred = model(X)
                loss = loss_fn(pred, y)

                # Backpropagation
                loss.backward()
                optimizer.step()
                optimizer.zero_grad()

                if batch % 100 == 0:
                    loss, current = loss.item(), batch * batch_size + len(X)
                    print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")
```

In [ ]:
```python
def test_loop(dataloader, model, loss_fn):
    # Set the model to evaluation mode - important for batch normalization and dropout layers
    # Unnecessary in this situation but added for best practices
    model.eval()
    size = len(dataloader.dataset)
    num_batches = len(dataloader)
    test_loss, correct = 0, 0
    # Evaluating the model with torch.no_grad() ensures that no gradients are computed during test mode
    # also serves to reduce unnecessary gradient computations and memory usage for tensors with requires_grad=True
    with torch.no_grad():
        for X, y in dataloader:
            pred = model(X)
            test_loss += loss_fn(pred, y).item()
            correct += (pred.argmax(1) == y).type(torch.float).sum().item()
    test_loss /= num_batches
    correct /= size
    print(f"Test Error: \n Accuracy: {(100*correct):>0.1f}%, Avg loss: {test_loss:>8f} \n")
```

In [ ]:
```python
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
epochs = 1
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
-------------------------------
loss: 2.305895  [   64/60000]
loss: 2.285977  [ 6464/60000]
loss: 2.269203  [12864/60000]
loss: 2.267075  [19264/60000]
loss: 2.265107  [25664/60000]
loss: 2.229418  [32064/60000]
loss: 2.238681  [38464/60000]
loss: 2.209979  [44864/60000]
loss: 2.208000  [51264/60000]
loss: 2.178957  [57664/60000]
Test Error: 
 Accuracy: 43.0%, Avg loss: 2.171826 

Done!
```

10 Learn the Basics: Save and Load the Model

In [ ]:
```python
import torch; import torchvision.models as models
```

Grabación de pesos del modelo: `state_dict` y `torch.save`

In [ ]:
```python
model = models.vgg16(weights='IMAGENET1K_V1')
torch.save(model.state_dict(), 'model_weights.pth')
```

Carga de pesos: `load_state_dict()` con `weights_only=True`

In [ ]:
```python
model = models.vgg16() # we do not specify ``weights``, i.e. create untrained model
model.load_state_dict(torch.load('model_weights.pth', weights_only=True))
model.eval() # to set the dropout and batch normalization layers to evaluation mode
```

Out[ ]:
```
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)
    ...
    (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))
  (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)
  )
)
```

Grabación del modelo completo: `torch.save` con `model` en lugar de `model.state_dict()`

```
In [ ]: torch.save(model, 'model.pth')
```

Carge del modelo completo: `torch_load` con `weights_only=False`

```
In [ ]: model = torch.load('model.pth', weights_only=False),
```

11 Ejercicio: CIFAR10

Ejercicio: entrena una red convolucional sencilla para CIFAR10 que supere el 55% de acierto en test

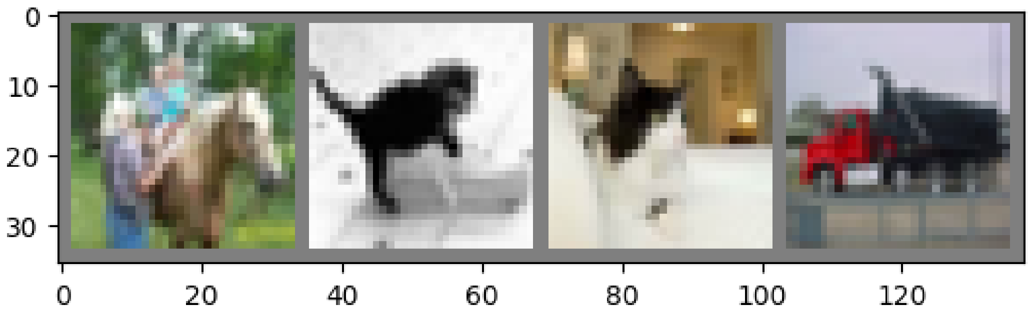
In [ ]:
```
import numpy as np; import matplotlib.pyplot as plt
import torch; import torchvision; import torchvision.transforms as transforms
```

Los datasets torchvision devuelven imágenes PILImage en $[0, 1]$ que normalizamos a $[-1, 1]$:

In [ ]:
```
transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])
batch_size = 4
trainset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=batch_size, shuffle=True, num_workers=2)
testset = torchvision.datasets.CIFAR10(root='./data', train=False, download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=batch_size, shuffle=False, num_workers=2)
classes = ('plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
```

Visualización de algunas imágenes:

In [ ]:
```
import matplotlib.pyplot as plt; import numpy as np
def imshow(img):
    img = img / 2 + 0.5; npimg = img.numpy(); plt.imshow(np.transpose(npimg, (1, 2, 0))); plt.show()
dataiter = iter(trainloader); images, labels = next(dataiter)
imshow(torchvision.utils.make_grid(images)); print(' '.join(f'{classes[labels[j]]:5s}' for j in range(batch_size)))
```




```
horse cat   cat   truck
```

Solución:

```
In [ ]: import torch.nn as nn; import torch.nn.functional as F
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)
    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = torch.flatten(x, 1) # flatten all dimensions except batch
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
device = torch.accelerator.current_accelerator().type if torch.accelerator.is_available() else "cpu"
model = Net().to(device); print(device, model)
```

```
cuda Net(
  (conv1): Conv2d(3, 6, kernel_size=(5, 5), stride=(1, 1))
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv2): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
  (fc1): Linear(in_features=400, out_features=120, bias=True)
  (fc2): Linear(in_features=120, out_features=84, bias=True)
  (fc3): Linear(in_features=84, out_features=10, bias=True)
)
```

```
In [ ]: def train_loop(dataloader, model, loss_fn, optimizer):
            model.train(); size = len(dataloader.dataset)
            num_batches = len(dataloader); train_loss, correct = 0, 0
            for X, y in dataloader:
                X, y = X.to(device), y.to(device)
                pred = model(X); loss = loss_fn(pred, y)
                loss.backward(); optimizer.step(); optimizer.zero_grad()
```

```
In [ ]: def eval_loop(dataloader, model, loss_fn):
            model.eval(); size = len(dataloader.dataset)
            num_batches = len(dataloader); test_loss, correct = 0, 0
            with torch.no_grad():
                for X, y in dataloader:
                    X, y = X.to(device), y.to(device)
                    pred = model(X); test_loss += loss_fn(pred, y).item()
                    correct += (pred.argmax(1) == y).type(torch.float).sum().item()
            test_loss /= num_batches; correct /= size
            return test_loss, correct
```

```
In [ ]: def exp(loss_fn, optimizer, epochs):
            train_losses = []; train_accs = []; test_losses = []; test_accs = []
            for t in range(epochs):
                train_loop(trainloader, model, loss_fn, optimizer)
                train_loss, train_acc = eval_loop(trainloader, model, loss_fn)
                train_losses.append(train_loss); train_accs.append(train_acc)
                test_loss, test_acc = eval_loop(testloader, model, loss_fn)
                test_losses.append(test_loss); test_accs.append(test_acc)
            return train_losses, train_accs, test_losses, test_accs
```

```
In [ ]: def plot_exp(train_losses, train_accs, test_losses, test_accs):
            fig, axs = plt.subplots(1, 2, figsize=(10, 2.25))
            fig.tight_layout(); plt.subplots_adjust(wspace=0.3)
            xx = np.arange(1, len(train_losses)+1); ax = axs[0]; ax.grid(); ax.set_ylabel('loss')
            ax.plot(xx, train_losses, 'b-', xx, test_losses, 'r-'); ax = axs[1]; ax.grid()
            ax.set_ylabel('accuracy'); ax.plot(xx, train_accs, 'b-', xx, test_accs, 'r-')
```

In [ ]:
```
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=1e-3)
trl, tra, tel, tea = exp(loss_fn, optimizer, 10)
```

In [ ]:
```
plot_exp(trl, tra, tel, tea); print(f'Acierto en test: {tea[-1]:.2%}')
```

```
Acierto en test: 57.30%
```