UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Redes neuronales multicapa

Alfons Juan

DSIC

Departament de Sistemes Informàtics i Computació

Índice

1. Introducción a PyTorch

- ***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).***
- ***Tutorial introductorio oficial:***

  `https://pytorch.org/tutorials/beginner/basics/intro.html`

1.1. Tensores

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

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

- Importación de las librerías torch y numpy:

```
37 import torch
38 import numpy as np
```

- ***Inicialización de un tensor***

  - Creación a partir de datos:

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

  - Creación a partir de un array NumPy:

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

- A partir de otro tensor, reteniendo *shape* y *datatype*:

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

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

- A partir de otro tensor, cambiando el *datatype*:

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

```
Random Tensor:
 tensor([[0.0692, 0.4579],
         [0.2360, 0.0959]])
```

- Con valores constantes o aleatorios:

```
97  shape = (2,3,)
98  rand_tensor = torch.rand(shape)
99  ones_tensor = torch.ones(shape)
100 zeros_tensor = torch.zeros(shape)
101
102 print(f"Random Tensor: \n {rand_tensor} \n")
103 print(f"Ones Tensor: \n {ones_tensor} \n")
104 print(f"Zeros Tensor: \n {zeros_tensor}")
```

```
Random Tensor:
 tensor([[0.1927, 0.5461, 0.9478],
         [0.6114, 0.3210, 0.8321]])

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

Zeros Tensor:
 tensor([[0., 0., 0.],
         [0., 0., 0.]])
```

▸ *Atributos de un tensor: shape, datatype y device*

```
122 tensor = torch.rand(3,4)
123
124 print(f"Shape of tensor: {tensor.shape}")
125 print(f"Datatype of tensor: {tensor.dtype}")
126 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
```

▸ *Operaciones sobre tensores:*

- ▹ Más de 100 operaciones disponibles:
  https://pytorch.org/docs/stable/torch.html
- ▹ Copia de un tensor (creado en CPU) a GPU:

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

- Indexación y trozeado al estilo NumPy:

```
171 tensor = torch.ones(4, 4)
172 print('First row: ', tensor[0])
173 print('First column: ', tensor[:, 0])
174 print('Last column:', tensor[..., -1])
175 tensor[:,1] = 0
176 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 en una dimensión dada:

```
188 t1 = torch.cat([tensor, tensor, tensor], dim=1)
189 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.]])
```

- Producto matricial: y1, y2, y3 tendrán el mismo valor

```
200 y1 = tensor @ tensor.T
201 y2 = tensor.matmul(tensor.T)
202
203 y3 = torch.rand_like(tensor)
204 torch.matmul(tensor, tensor.T, out=y3)
```

- Producto elemental: z1, z2, z3 tendrán el mismo valor

```
208 z1 = tensor * tensor
209 z2 = tensor.mul(tensor)
210
211 z3 = torch.rand_like(tensor)
212 torch.mul(tensor, tensor, out=z3)
```

- Tensores de un único elemento: conversión a número Python

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

```
12.0 <class 'float'>
```

- Operaciones en línea: mediante el sufijo "_"

```
238 print(tensor, "\n")
239 tensor.add_(5)
240 print(tensor)
```

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

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

▷ Memoria compartida entre tensores en CPU y arrays NumPy:

```
269 t = torch.ones(5)
270 print(f"t: {t}")
271 n = t.numpy()
272 print(f"n: {n}")
```

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

```
282 t.add_(1)
283 print(f"t: {t}")
284 print(f"n: {n}")
```

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

```
295 n = np.ones(5)
296 t = torch.from_numpy(n)
```

```
306 np.add(n, 1, out=n)
307 print(f"t: {t}")
308 print(f"n: {n}")
```

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

1.2. 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

- ***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

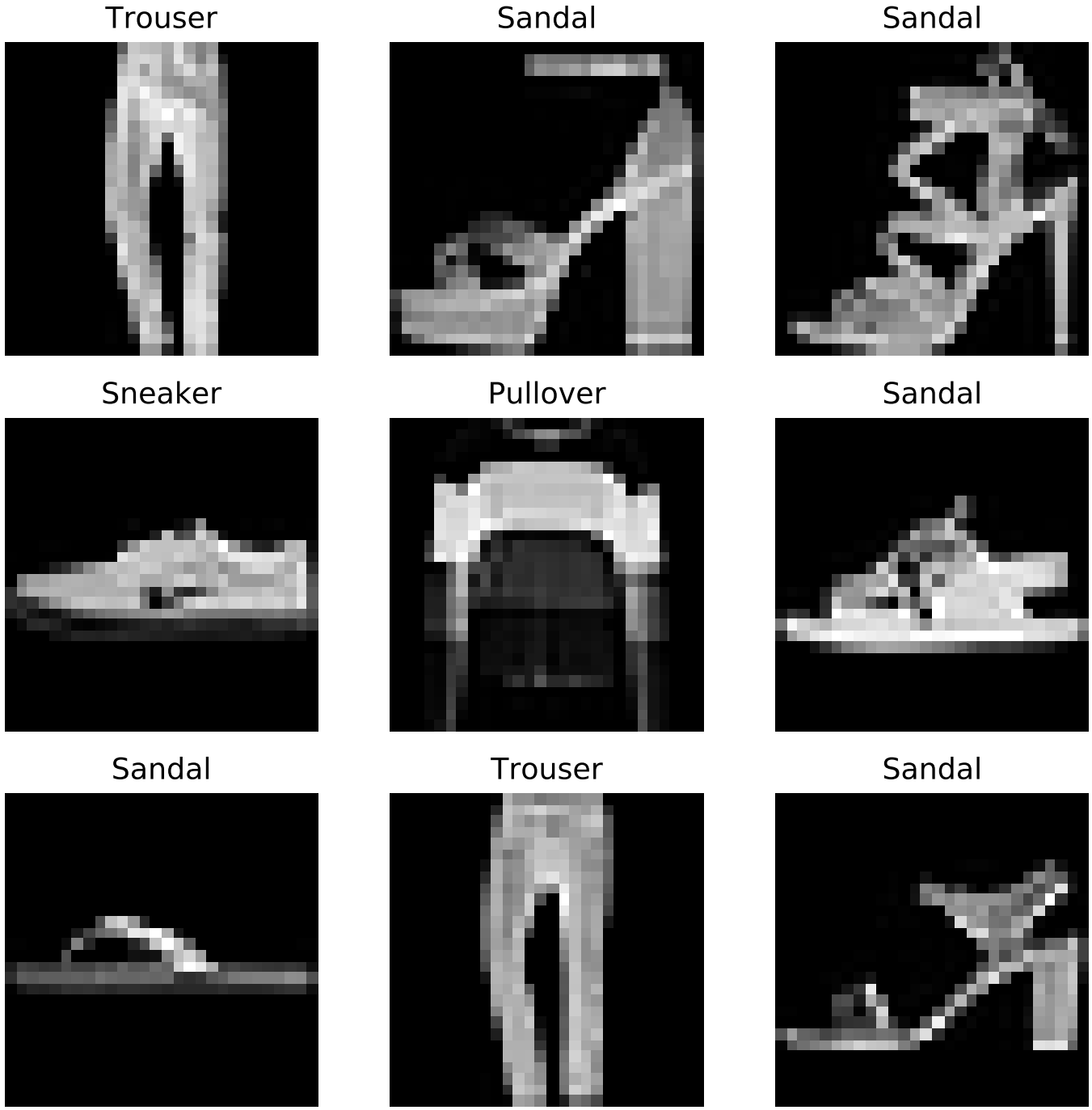
```
62 import torch
63 from torch.utils.data import Dataset
64 from torchvision import datasets
65 from torchvision.transforms import ToTensor
66 import matplotlib.pyplot as plt
```

```
69 training_data = datasets.FashionMNIST(
70   root="data",
71   train=True,
72   download=True,
73   transform=ToTensor()
74 )
```

```
76 test_data = datasets.FashionMNIST(
77   root="data",
78   train=False,
79   download=True,
80   transform=ToTensor()
81 )
```

▸ *Iteración y visualización:* con `training_data[index]`

```
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.savefig('data_tutorial_fig.eps'); plt.show()
```


Trouser
Sandal
Sandal
Sneaker
Pullover
Sandal
Sandal
Trouser
Sandal

- ***Conjunto de datos del usuario:***

  `__init__` `__len__` `__getitem__`

```
144 import os
145 import pandas as pd
146 from torchvision.io import read_image
```

```
148 class CustomImageDataset(Dataset):
149   def __init__(self, annotations_file, img_dir,
      ↪  transform=None, target_transform=None):
150     self.img_labels = pd.read_csv(annotations_file)
151     self.img_dir = img_dir
152     self.transform = transform
153     self.target_transform = target_transform
```

```
155   def __len__(self):
156     return len(self.img_labels)
```

```
158   def __getitem__(self, idx):
159     img_path = os.path.join(self.img_dir,
        ↪  self.img_labels.iloc[idx, 0])
160     image = read_image(img_path)
161     label = self.img_labels.iloc[idx, 1]
162     if self.transform:
163       image = self.transform(image)
164     if self.target_transform:
165       label = self.target_transform(label)
166     return image, label
```

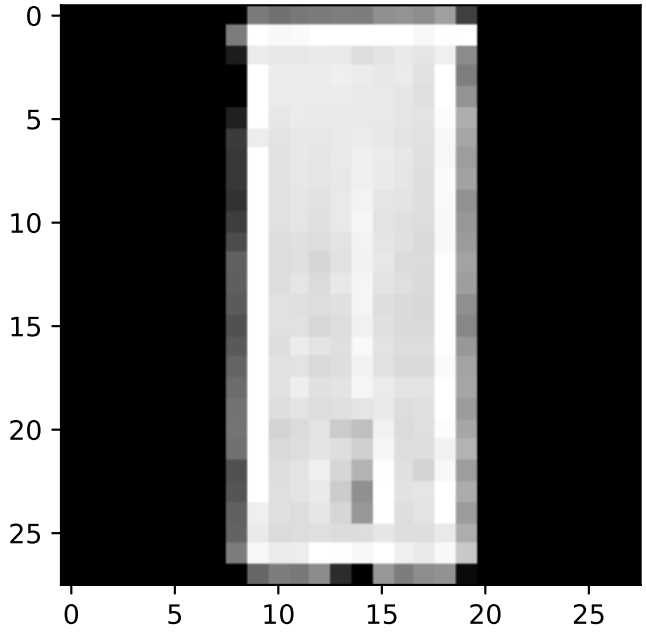
- ***Preparación de los datos de entrenamiento:*** **`DataLoader`**
  - **`Dataset`** indexa los datos uno a uno
  - **`DataLoader`** los procesa en minibatches y baraja por época

```
253 from torch.utils.data import DataLoader
254
255 train_dataloader = DataLoader(training_data, batch_size=64,
    ↪  shuffle=True)
256 test_dataloader = DataLoader(test_data, batch_size=64,
    ↪  shuffle=True)
```

▸ *Iteración mediante DataLoader:* un minibatch por iteración

```
273 train_features, train_labels = next(iter(train_dataloader))
274 print(f"Feature batch shape: {train_features.size()}")
275 print(f"Labels batch shape: {train_labels.size()}")
276 img = train_features[0].squeeze()
277 label = train_labels[0]
278 plt.imshow(img, cmap="gray")
279 plt.savefig('data_tutorial_fig2.eps'); plt.show()
280 print(f"Label: {label}")
```

```
Feature batch shape: torch.Size([64, 1, 28, 28])
Labels batch shape: torch.Size([64])
<Figure size 432x288 with 1 Axes>
Label: 1
```



1.3. 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:*** de formato PIL a tensor
  - ***target_transform:*** de entero a tensor one-hot

```
42 import torch
43 from torchvision import datasets
44 from torchvision.transforms import ToTensor, Lambda
45
46 ds = datasets.FashionMNIST(
47   root="data",
48   train=True,
49   download=True,
50   transform=ToTensor(),
51   target_transform=Lambda(lambda y: torch.zeros(10,
     ↪  dtype=torch.float).scatter_(0, torch.tensor(y),
     ↪  value=1))
52 )
```

  - ***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

1.4. 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

- ***Red para FashionMNIST:***

```
38 import os
39 import torch
40 from torch import nn
41 from torch.utils.data import DataLoader
42 from torchvision import datasets, transforms
```

  - ***Dispositivo para entrenamiento:***

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

```
Using cpu device
```

- ***Red:*** subclase de **`nn.Module`** con capas inicializadas mediante **`__init__`** y método **`forward`** para procesar datos

```
72 class NeuralNetwork(nn.Module):
73   def __init__(self):
74     super(NeuralNetwork, self).__init__()
75     self.flatten = nn.Flatten()
76     self.linear_relu_stack = nn.Sequential(
77       nn.Linear(28*28, 512),
78       nn.ReLU(),
79       nn.Linear(512, 512),
80       nn.ReLU(),
81       nn.Linear(512, 10),
82       )
83
84   def forward(self, x):
85     x = self.flatten(x)
86     logits = self.linear_relu_stack(x)
87     return logits
```

▷ ***Instanciación de la red:*** (y transferencia al dispositivo)

```
98 model = NeuralNetwork().to(device)
99 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:*** sin llamar a `model.forward()` directamente!

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

```
Predicted class: tensor([7])
```

- Minibatch de 3 imágenes 28x28 para probar las capas de la red:

```
138 input_image = torch.rand(3,28,28)
139 print(input_image.size())
```

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

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

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

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

- ***nn.Linear:*** transformación lineal

```
169 layer1 = nn.Linear(in_features=28*28, out_features=20)
170 hidden1 = layer1(flat_image)
171 print(hidden1.size())
```

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

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

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

```
Before ReLU: tensor([[ 0.1273,  0.2672,  0.3574,  0.3020, -0.1052,  0.1517, -0.1105,  0.0767,
          0.1336,  0.1753,  0.0544,  0.4097, -0.1525, -0.1203, -0.6834, -0.1714,
          0.0906, -0.3521,  0.2346,  0.3087],
        [-0.3015,  0.1236,  0.4443,  0.2092, -0.0203, -0.1279, -0.1368, -0.0398,
         -0.0414, -0.0998,  0.0626,  0.1779, -0.1200, -0.2351, -0.2217,  0.1297,
         -0.1608,  0.3531,  0.0319, -0.0061],
        [-0.0182, -0.0440,  0.0421, -0.0962, -0.0679,  0.2174,  0.0236,  0.1717,
          0.0334,  0.0672,  0.2517,  0.3544, -0.3450,  0.0033, -0.5189, -0.3197,
          0.2171, -0.2576,  0.0865,  0.0896]], grad_fn=<AddmmBackward>)


After ReLU: tensor([[0.1273, 0.2672, 0.3574, 0.3020, 0.0000, 0.1517, 0.0000, 0.0767, 0.1336,
         0.1753, 0.0544, 0.4097, 0.0000, 0.0000, 0.0000, 0.0000, 0.0906, 0.0000,
         0.2346, 0.3087],
        [0.0000, 0.1236, 0.4443, 0.2092, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
         0.0000, 0.0626, 0.1779, 0.0000, 0.0000, 0.0000, 0.1297, 0.0000, 0.3531,
         0.0319, 0.0000],
        [0.0000, 0.0000, 0.0421, 0.0000, 0.0000, 0.2174, 0.0236, 0.1717, 0.0334,
         0.0672, 0.2517, 0.3544, 0.0000, 0.0033, 0.0000, 0.0000, 0.2171, 0.0000,
         0.0865, 0.0896]], grad_fn=<ReluBackward0>)
```

- ***nn.Sequential:*** contenedor de módulos ordenado

```
204 seq_modules = nn.Sequential(
205     flatten,
206     layer1,
207     nn.ReLU(),
208     nn.Linear(20, 10)
209 )
210 input_image = torch.rand(3,28,28)
211 logits = seq_modules(input_image)
```

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

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

- ***Parámetros:*** `parameters()` `named_parameters()`

```
245 print("Model structure: ", model, "\n\n")
246
247 for name, param in model.named_parameters():
248     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([[
↪  0.0258,  0.0100, -0.0098,  ...,  0.0356,  0.0075, -0.0312],
        [-0.0270,  0.0208, -0.0270,  ...,  0.0253, -0.0113,  0.0235]],
       grad_fn=<SliceBackward>)

Layer: linear_relu_stack.0.bias | Size: torch.Size([512]) | Values : tensor([ 0.0256,
↪  -0.0135], grad_fn=<SliceBackward>)

Layer: linear_relu_stack.2.weight | Size: torch.Size([512, 512]) | Values : tensor([[
↪  0.0257, -0.0190,  0.0312,  ..., -0.0165, -0.0153, -0.0055],
        [-0.0319,  0.0410, -0.0223,  ...,  0.0059, -0.0043, -0.0406]],
       grad_fn=<SliceBackward>)

Layer: linear_relu_stack.2.bias | Size: torch.Size([512]) | Values : tensor([0.0287,
↪  0.0239], grad_fn=<SliceBackward>)

Layer: linear_relu_stack.4.weight | Size: torch.Size([10, 512]) | Values :
↪  tensor([[-0.0174,  0.0245,  0.0148,  ..., -0.0113, -0.0366, -0.0281],
        [-0.0389, -0.0332,  0.0430,  ..., -0.0261,  0.0423, -0.0020]],
       grad_fn=<SliceBackward>)

Layer: linear_relu_stack.4.bias | Size: torch.Size([10]) | Values : tensor([0.0002,
↪  0.0212], grad_fn=<SliceBackward>)
```

1.5. Diferenciación automática

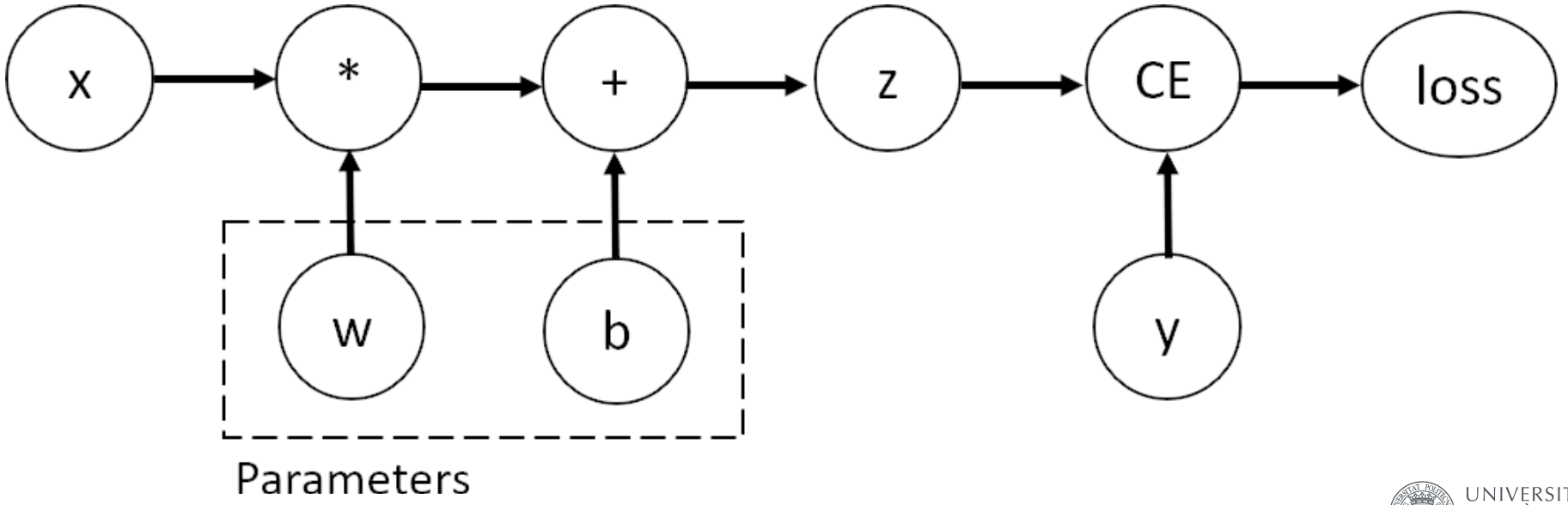
- ***Tutorial oficial:*** cuaderno jupyter ***autogradqs_tutorial***

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

- ***Grafo computacional:*** con ***torch.autograd***

```
42 import torch
```

```
44 x = torch.ones(5)  # input tensor
45 y = torch.zeros(3)  # expected output
46 w = torch.randn(5, 3, requires_grad=True)
47 b = torch.randn(3, requires_grad=True)
48 z = torch.matmul(x, w)+b
49 loss = torch.nn.functional.binary_cross_entropy_with_logits(z, y)
```



- ***Tensores, funciones y grafo computacional:***
  - ***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, así como el cálculo del gradiente de la pérdida con respecto a los parámetros.
  - ***grad_fn:*** propiedad de tensor con la función gradiente

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

```
Gradient function for z = <AddBackward0 object at 0x7f3665790310>
Gradient function for loss = <BinaryCrossEntropyWithLogitsBackward
↪  object at 0x7f3665790310>
```

- ***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`

```
107 loss.backward()
108 print(w.grad)
109 print(b.grad)
```

```
tensor([[0.1306, 0.0071, 0.3008],
        [0.1306, 0.0071, 0.3008],
        [0.1306, 0.0071, 0.3008],
        [0.1306, 0.0071, 0.3008],
        [0.1306, 0.0071, 0.3008]])
tensor([0.1306, 0.0071, 0.3008])
```

- ***Inhabilitación del seguimiento de gradientes:*** necesario cuando se quiere "congelar" parámetros en fine-tuning o acelerar cálculos inferencia, evitando que tensores con **`requires_grad=True`** hagan seguimiento de su historia computacional para permitir el cálculo de gradientes
  - **`torch.no_grad()`:**

```
141 z = torch.matmul(x, w)+b
142 print(z.requires_grad)
143
144 with torch.no_grad():
145   z = torch.matmul(x, w)+b
146 print(z.requires_grad)
```

```
True
False
```

  - **`detach()`:**

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

```
False
```

- ***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)

1.6. 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

- Código previo:

```
40 import torch
41 from torch import nn
42 from torch.utils.data import DataLoader
43 from torchvision import datasets
44 from torchvision.transforms import ToTensor, Lambda
```

```
46 training_data = datasets.FashionMNIST(
47     root="data",
48     train=True,
49     download=True,
50     transform=ToTensor()
51 )
```

```
53 test_data = datasets.FashionMNIST(
54     root="data",
55     train=False,
56     download=True,
57     transform=ToTensor()
58 )
```

```
60 train_dataloader = DataLoader(training_data,
    ↪  batch_size=64)
61 test_dataloader = DataLoader(test_data, batch_size=64)
```

- Código previo (cont.):

```
63 class NeuralNetwork(nn.Module):
64   def __init__(self):
65     super(NeuralNetwork, self).__init__()
66     self.flatten = nn.Flatten()
67     self.linear_relu_stack = nn.Sequential(
68       nn.Linear(28*28, 512),
69       nn.ReLU(),
70       nn.Linear(512, 512),
71       nn.ReLU(),
72       nn.Linear(512, 10),
73     )
74
75   def forward(self, x):
76     x = self.flatten(x)
77     logits = self.linear_relu_stack(x)
78     return logits
79
80 model = NeuralNetwork()
```

- ***Hiperparámetros:***
  - ***Número de épocas:*** número de veces a iterar sobre los datos
  - ***Tamaño del batch:*** número de muestras propagadas a través de la red antes de actualizar parámetros
  - ***Factor de aprendizaje:*** magnitud de la actualización de parámetros en cada batch/época
    - ↦ Demasiado pequeño: aprendizaje lento
    - ↦ Demasiado grande: aprendizaje impredecible

```
101 learning_rate = 1e-3
102 batch_size = 64
103 epochs = 5
```

- ***Bucle de optimización:*** cada época consta de dos partes
  - ***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

- ***Función de pérdida:*** error de predicción a minimizar
  - ***Mean Square Loss:*** `nn.MSELoss`, para regresión
  - ***Negative Log Likelihood:*** `nn.NLLLoss`, para clasificación
  - ***Cross Entropy:*** `nn.CrossEntropyLoss`, combina `nn.LogSoftmax` y `nn.NLLLoss`

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

- ***Optimización:*** proceso de ajuste de los parámetros del modelo en cada paso de entrenamiento

- ***Algoritmos de optimización:*** objeto **`optimizer`**; usamos SGD

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

- La optimización se lleva acabo en tres pasos:
  - **`optimizer.zero_grad()`** para reinicializar a cero los gradientes de los parámetros (y así evitar sumas duplicadas).
  - **`loss.backwards()`** para retropropagar la pérdida de la predicción calculando sus gradientes respecto a los parámetros.
  - **`optimizer.step()`** para ajustar los parámetros mediante los gradientes hallados en retropropagación.

▸ *Bucle de entremamiento:*

```
177 def train_loop(dataloader, model, loss_fn, optimizer):
178   size = len(dataloader.dataset)
179   for batch, (X, y) in enumerate(dataloader):
180     # Compute prediction and loss
181     pred = model(X)
182     loss = loss_fn(pred, y)
183 
184     # Backpropagation
185     optimizer.zero_grad()
186     loss.backward()
187     optimizer.step()
188 
189     if batch % 100 == 0:
190       loss, current = loss.item(), batch * len(X)
191       print(f"loss: {loss:>7f}
          ↪  [{current:>5d}/{size:>5d}]")
```

▸ *Bucle de test:*

```
194 def test_loop(dataloader, model, loss_fn):
195   size = len(dataloader.dataset)
196   num_batches = len(dataloader)
197   test_loss, correct = 0, 0
198
199   with torch.no_grad():
200     for X, y in dataloader:
201       pred = model(X)
202       test_loss += loss_fn(pred, y).item()
203       correct += (pred.argmax(1) ==
          ↪  y).type(torch.float).sum().item()
204
205   test_loss /= num_batches
206   correct /= size
207   print(f"Test Error: \n Accuracy: {(100*correct):>0.1f}%,
      ↪  Avg loss: {test_loss:>8f} \n")
```

- Inicialización de la función de pérdida y optimizador para los bucles de entrenamiento y test:

```
218 loss_fn = nn.CrossEntropyLoss()
219 optimizer = torch.optim.SGD(model.parameters(),
     ↪  lr=learning_rate)
220
221 epochs = 10
222 for t in range(epochs):
223   print(f"Epoch {t+1}\n-------------------------------")
224   train_loop(train_dataloader, model, loss_fn, optimizer)
225   test_loop(test_dataloader, model, loss_fn)
226 print("Done!")
```

```
Epoch 1
-------------------------------
loss: 2.314330  [    0/60000]
loss: 2.298410  [ 6400/60000]
loss: 2.278487  [12800/60000]
loss: 2.269283  [19200/60000]
loss: 2.252476  [25600/60000]
loss: 2.224926  [32000/60000]
loss: 2.230276  [38400/60000]
loss: 2.195671  [44800/60000]
loss: 2.198123  [51200/60000]
loss: 2.147159  [57600/60000]
Test Error:
 Accuracy: 44.7%, Avg loss: 2.155972
```

2. Tarea MNIST

- Las redes ofrecen los mejores resultados:
  `http://devres.zoomquiet.top/data/20160422121512/index.html`

- ***mlp_exp.ipynb*** y ***mlp_exp.py:*** experimento MNIST basado en una partición del conjunto de entrenamiento oficial con un 90 % para entrenamiento y un 10 % para validación.
  - MLP de arquitectura propuesta por Geoffrey E. Hinton en 2005.
  - Dos capas ocultas de 500 y 300 neuronas con activación ReLU, entropía cruzada como función de pérdida y optimización Adam.
  - Error en validación por debajo del 2 % cuando se entrena durante 20 épocas con un tamaño de batch de 100 muestras.

- Configuración previa en Polilabs:

```
export PYTHONPATH=$PYTHONPATH:$HOME/asigDSIC/ETSINF/apr/mlp/pylib
```

mlp_exp.py

```
1  #!/usr/bin/env python
2  # coding: utf-8
3
4  # # Aplicación de redes MLP a la tarea MNIST
5
6  # Importamos las librerías necesarias para la realización de un
7  # experimento con una red MLP en MNIST
8
9  import torch
10 import torchvision
```

mlp_exp.py

```
12 # ## Carga de datos de MIST
13
14 # La tarea MNIST está disponible desde la libreria
15 # `torchvision`. Mediante el parámetro `train=True` seleccionamos la
16 # partición de entrenamiento o evaluación. Finalmente, mediante el
17 # parámetro `transform=torchvision.transforms.ToTensor()` indicamos
18 # qué transformación se aplica a MNIST, en este caso convertimos las
19 # imágenes en tensores.
20
21 # MNIST Dataset (Images and Labels)
22 training_dataset = torchvision.datasets.MNIST(
23     root='./data',
24     train=True,
25     transform=torchvision.transforms.ToTensor(),
26     download=True
27 )
28 test_dataset = torchvision.datasets.MNIST(
29     root='./data',
30     train=False,
31     transform=torchvision.transforms.ToTensor(),
32     download=True
33 )
```

mlp_exp.py

```
36 # ## Diseño experimental y preparación de datos
37
38 # Realizamos una partición del conjunto de entrenamiento oficial
39 # dedicando un 90% (54000 muestras) para entrenamiento y un 10% (6000
40 # muestras) para validación.
41
42 train_dataset, val_dataset =
   ↪  torch.utils.data.random_split(training_dataset, [54000, 6000])
```

mlp_exp.py

```
45 # A la hora de entrenar el modelo pasaremos los datos por dicho modelo
46 # varias veces. Cada vez que pasemos un lote de datos (*batch*) por el
47 # modelo le llamaremos iteración, o *forward pass*. Cada vez que
48 # pasemos todos los datos por el modelo le llamaremos *epoch*.
49 #
50 # Entre cada epoch los datos son barajados y divididos en lotes
51 # (batches) de nuevo. Esta gestión de los datos está implementada en
52 # la clase *DataLoader* de PyTorch.
53
54 batch_size = 100
55
56 # Preparamos los conjuntos de entrenamiento y validación en lotes y
57 # barajamos el conjunto de entrenamiento
58 train_loader = torch.utils.data.DataLoader(dataset=train_dataset,
   ↪  batch_size=batch_size, shuffle=True)
59 val_loader = torch.utils.data.DataLoader(dataset=val_dataset,
   ↪  batch_size=batch_size, shuffle=False)
```

mlp_exp.py

```
62 # ## Definición de una red MLP
63 #
64 # Una red MLP está compuesta por una secuencia de capas de neuronas
65 # conectadas entre si. La capa de entrada tiene una dimensionalidad
66 # que depende de los datos utilizados. En el caso de MNIST son
67 # imágenes de 28x28 píxeles, es decir, 784 dimensiones. La capa de
68 # salida tiene una dimensionalidad que depende del número de clases a
69 # predecir. En MNIST, tenemos los dígitos del 0 al 9, es decir, 10
70 # clases. Entre las capas de entrada y de salida se definen $L$ capas
71 # ocultas con un número de neuronas $M_1$. A la salida de cada neurona
72 # se aplica una función de activación para poder aproximar funciones
73 # no lineales.
74 #
75 # Definimos una red MLP con las siguientes características:
76 #
77 # - Entrada: 784
78 # - Primera capa oculta ($M_1$): 500
79 # - Segunda capa oculta ($M_2$): 300
80 # - Capa de salida: 10
81 #
82 # Usaremos como función de activación ReLU tras cada oculta, pero se
83 # pueden utilizar otras definidas en el paquete `torch.nn.functional`.
```

mlp_exp.py

```
85  from torch import nn, optim
86  from torch.nn.modules import Module
87
88  class MLP(nn.Module):
89      # layers_data is a list of pairs: number of neurons and activation
        ↪  function
90      def __init__(self, input_size, layers_data: list, num_classes,
        ↪  learning_rate=1e-3, optimizer=optim.Adam):
91          super().__init__()
92
93          self.layers = nn.ModuleList()
94          self.input_size = input_size
95          # Layer and activation function are appended in a list
96          for output_size, activation_function in layers_data:
97              self.layers.append(nn.Linear(input_size, output_size))
98              input_size = output_size
99              self.layers.append(activation_function)
100         # Finally, the output layer is appended
101         self.layers.append(nn.Linear(input_size, num_classes))
102         self.device = torch.device('cuda' if torch.cuda.is_available()
            ↪  else 'cpu')
103         self.to(self.device)
104         self.learning_rate = learning_rate
105         self.optimizer = optimizer(params=self.parameters(),
            ↪  lr=learning_rate)
106         self.criterion = nn.CrossEntropyLoss()
107
108     def forward(self, input_data):
109         for layer in self.layers:
110             output_data = layer(input_data)
111             input_data=output_data
112         return output_data
```

mlp_exp.py

```
115 input_size=28 * 28
116 M1, M2 = 500, 300
117 num_classes=10
118 mlp = MLP(input_size, [(M1, nn.ReLU()), (M2, nn.ReLU())], num_classes)
```

mlp_exp.py

```
121 # ##Entrenamiento de la red MLP
122 #
123 # Se recorre el conjunto de entrenamiento en lotes (batches) y por
124 # cada batch se efectúan los siguientes pasos:
125 #
126 # 1) Se transfiere el batch a la CPU o GPU.
127 # 2) Se calcula el paso forward para obtener la predicción.
128 # 3) Se calcula la función de pérdida y el gradiente utilizando BackProp
    ↪  (backward).
129 # 4) Se actualizan los valores de los parámetros con el gradiente.
130 # 5) Se resetea el gradiente.
```

mlp_exp.py

```
134 # Loop for a number of epochs
135 for epoch in range(20):
136     total_loss = 0.0
137 
138     # Loop over the training set in batch mode
139     for (inputs, labels) in train_loader:
140 
141         # Transfering data to GPU or CPU
142         inputs = inputs.to(mlp.device)
143         labels = labels.to(mlp.device)
144 
145         # Converting from 28x28 data samples to 768 data samples
146         inputs = inputs.view(-1, 28*28)
147 
148         # Forward pass
149         outputs = mlp(inputs)
150         # Computing loss function
151         loss = mlp.criterion(outputs, labels)
152         # Computing gradient
153         loss.backward()
154         # Updating parameter values with gradient
155         mlp.optimizer.step()
156         # Reset gradient
157         mlp.optimizer.zero_grad()
158 
159         # Accumulated loss function over an epoch
160         total_loss += loss.item()
161 
162     print("Epoch %d, Loss=%.4f" % (epoch+1,
    ↪  total_loss/len(train_loader)))
```

mlp_exp.py

```
165 # ### Evaluación
166 # Estimación del error empírico en un conjunto de datos.
167
168 def error(model, data_loader, device):
169     with torch.no_grad():
170         errors = 0
171         total = 0
172         for inputs, labels in data_loader:
173             inputs = inputs.to(device)
174             inputs = inputs.view(-1, 28*28)
175
176             outputs = model(inputs)
177             _, predicted = outputs.max(1)
178
179             errors += (predicted.cpu() != labels).sum().item()
180             total += labels.size(0)
181
182     err = errors / total
183     return err
184
185
186 # Estimación del error en el conjunto de entrenamiento.
187
188 err=error(mlp, train_loader, mlp.device)
189 print("Tasa de error en entrenamiento: %.2f%%" % (err*100))
```

mlp_exp.out

```
Epoch 1, Loss=0.2722
Epoch 2, Loss=0.0965
Epoch 3, Loss=0.0629
Epoch 4, Loss=0.0465
Epoch 5, Loss=0.0328
Epoch 6, Loss=0.0259
Epoch 7, Loss=0.0216
Epoch 8, Loss=0.0160
Epoch 9, Loss=0.0172
Epoch 10, Loss=0.0120
Epoch 11, Loss=0.0135
Epoch 12, Loss=0.0116
Epoch 13, Loss=0.0109
Epoch 14, Loss=0.0071
Epoch 15, Loss=0.0099
Epoch 16, Loss=0.0114
Epoch 17, Loss=0.0045
Epoch 18, Loss=0.0140
Epoch 19, Loss=0.0062
Epoch 20, Loss=0.0082
Tasa de error en entrenamiento: 0.23%
Tasa de error en validación: 2.25%
```

3. Ejercicios

3.1. Ejercicio 1 (0.25 puntos)

- A partir del código base proporcionado, representa gráficamente la evolución del error en entrenamiento y validación en función del número de épocas.

- Si es necesario, incrementa el número máximo de iteraciones para poder observar el fenómeno de sobreentrenamiento.

3.2. Ejercicio 2 (0.5 puntos)

- Para ajustar los parámetros del MLP, estudia el comportamiento de la tasa de error en el conjunto de validación en función de:
  - Algoritmo de optimización: SGD, Adadelta, Adagrad, Adam, etc.
  - Función de activación: ReLU, Sigmoid, Tanh, etc.
  - Número de capas ocultas: 1, 2, 3, etc.
  - Número de neuronas por capa, en múltiplos de 100 hasta 800.
  - Criterio de parada: iteraciones, error en validación, etc.

- En función del número de resultados obtenidos como consecuencia de la exploración de los valores de los parámetros, utiliza una representación adecuada de los mismos, ya sea gráfica o tabular, que muestre no solo el mejor resultado obtenido, sino también otros resultados relevantes que permitan poner de manifiesto la tendencia a mejorar o empeorar del modelo según varían los valores de los parámetros considerados.

3.3. Ejercicio 3 (0.25 puntos)

- Tras el ajuste de parámetros en el conjunto de validación, utiliza los valores óptimos de los parámetros del clasificador para entrenar y evaluar un clasificador final en los conjuntos oficiales MNIST de entrenamiento y test, respectivamente.

- Recuerda que toda estimación de (la probabilidad de) error de un clasificador final, debe ir acompañada de sus correspondientes intervalos de confianza al 95 %.

- Discute los resultados obtenidos comparándolos con los obtenidos en los clasificadores estudiados y con otros clasificadores basados en redes neuronales reportados en la tarea MNIST.