UNIVERSIDAD EAFIT

Ph. D. Juan David Martínez Vargas
Ph. D. Raul Andrés Castañeda

Escuela de Ciencias Aplicadas e Ingeniería

Lecture 05
Deep Learning

Agenda

- **Entrenar un modelo desde cero**
- **Actividad**
- **Quiz**
- **Sotmax**

El "Hello word" del DL

Clasificación de dígitos manuscritos usando PyTorch

En este Ejemplo se implementará un modelo de clasificación supervisada para reconocer dígitos manuscritos (0–9) a partir de imágenes en escala de grises, utilizando la base de datos **MNIST** y la librería PyTorch.

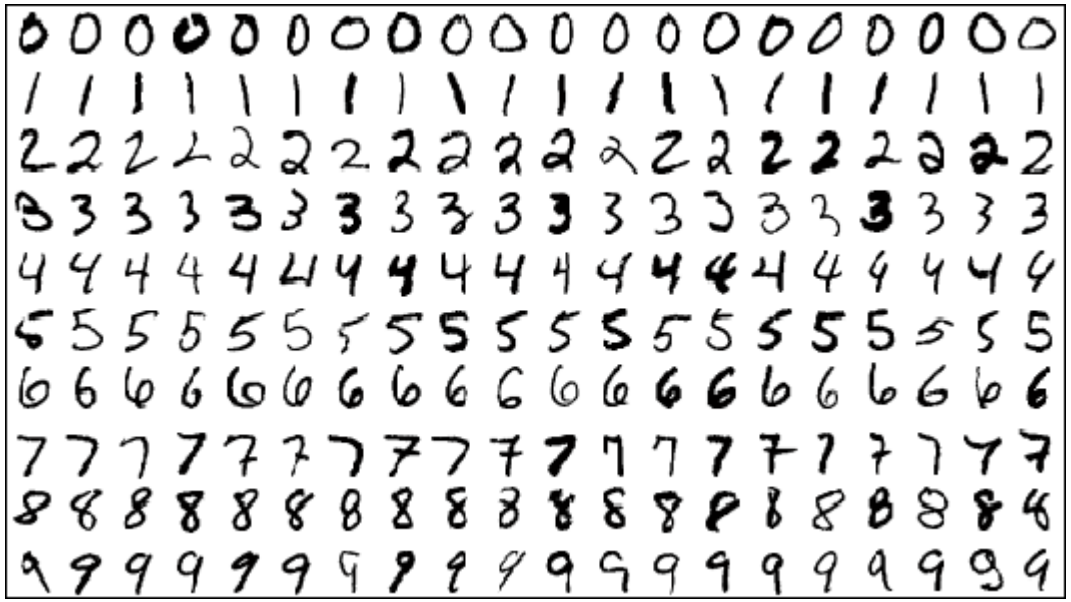
El objetivo es entrenar un modelo que, dada una imagen de entrada, asigne correctamente la probabilidad de pertenencia a cada una de las 10 clases posibles.

El ejercicio se desarrollará paso a paso, abordando los siguientes aspectos:

- Carga y exploración del conjunto de datos MNIST.
- Preparación de los datos para el entrenamiento (transformaciones y normalización).
- Definición de un modelo de clasificación basado en capas totalmente conectadas (Fullyconected).
- Implementación de la función de pérdida y del algoritmo de optimización.
- Entrenamiento del modelo y análisis de la evolución de la pérdida.
- Evaluación del desempeño del modelo sobre datos no vistos.

El "Hello word" del DL

¿ Qué es MNIST ?



Link

- La base de datos MNIST consta de 60.000 imágenes de entrenamiento y 10.000 imágenes de prueba.
- Cada imagen tiene un tamaño de 28×28 píxeles y representa un dígito escrito a mano.
- Las imágenes están en escala de grises.
- Cada imagen está asociada a una etiqueta.

El "Hello word" del DL

Para implementar un modelo DL end-to-end

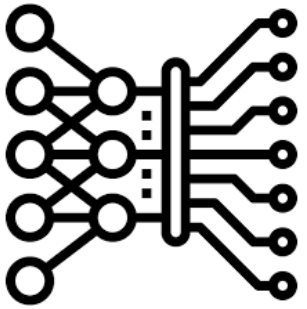
- Cargar y explorar la base de datos (ver ejemplos, tamaños, clases).
- Preprocesar (transformaciones, normalización, flatten si aplica).
- Definir conjuntos de entrenamiento y validación/prueba + DataLoader.
- Definir la arquitectura de la red neuronal (capas, activaciones).
- Configurar el aprendizaje: función de pérdida, optimizador, métricas (accuracy).
- Entrenar el modelo (épocas, monitoreo de loss/accuracy).
- Evaluar y discutir resultados (curvas, errores típicos, overfitting) (opcional: guardar modelo).

Recollect data

Data pre-processing

Data Analysis

Split data

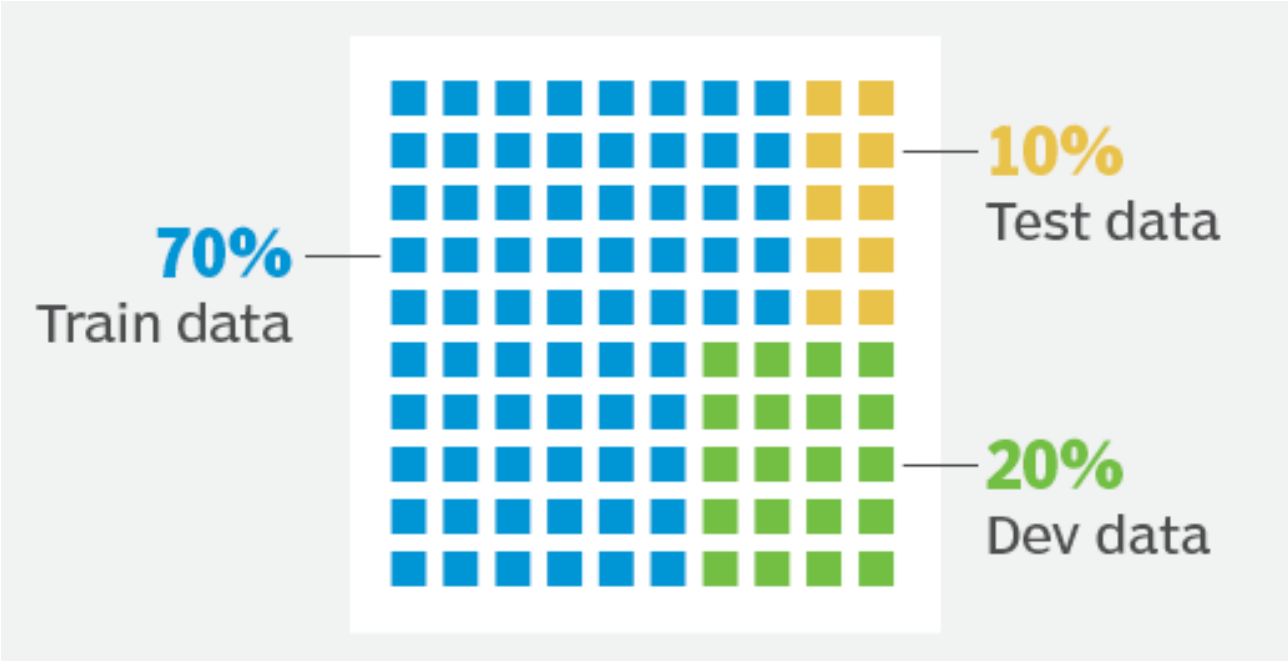


Model

Evaluation

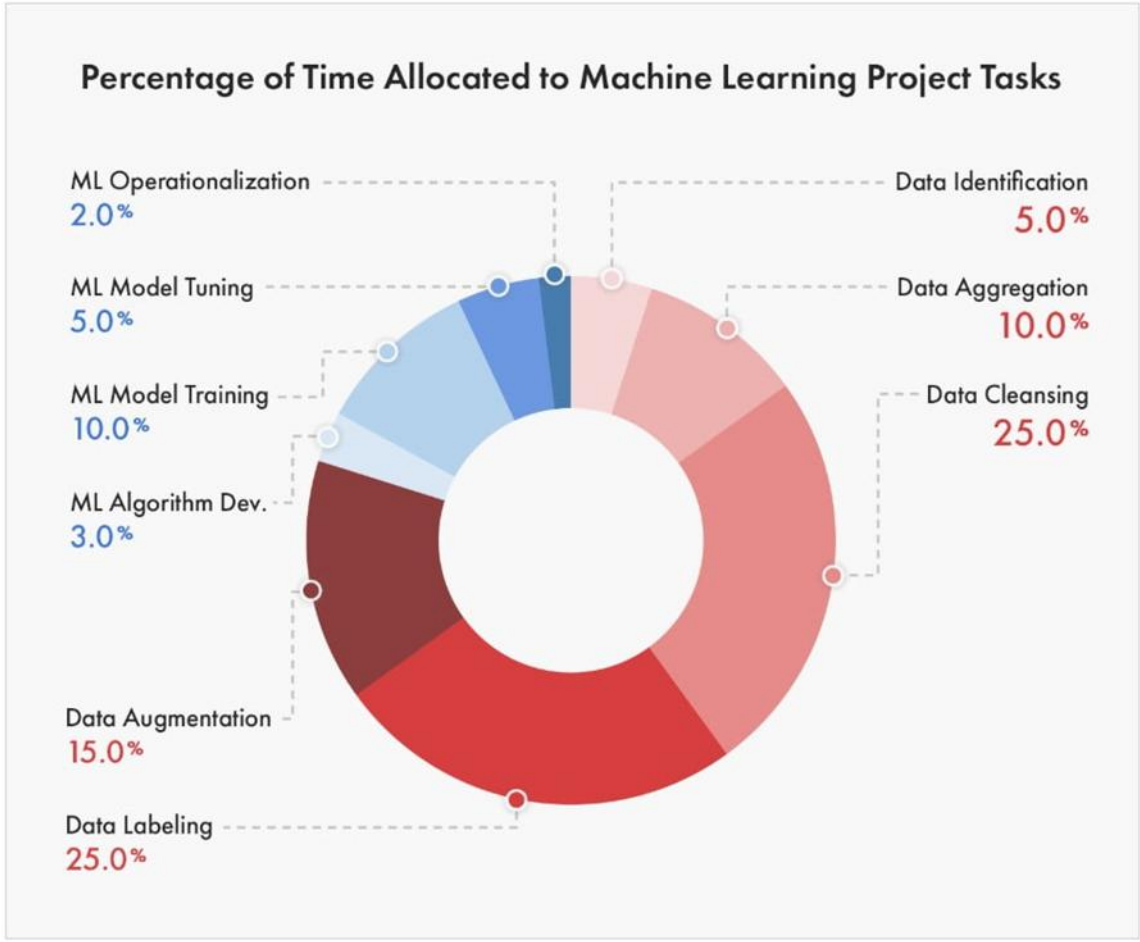
El "Hello word" del DL

Data Splitting



https://builtin.com/data-science/train-test-split

https://kili-technology.com/blog/training-validation-and-test-sets-how-to-split-machine-learning-data



Link

El "Hello word" del DL

Step-by-step

```
!pip install torch torchvision torchaudio
```

4m 12.2s Python

```
!pip install matplot
```

4.7s Python

Instalar Librerías

Importar Librerías

```
import torch
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader
from torch.optim import Adam

# Visualization tools
import torchvision
import torchvision.transforms.v2 as transforms
import torchvision.transforms.functional as F
import matplotlib.pyplot as plt
```

El "Hello word" del DL

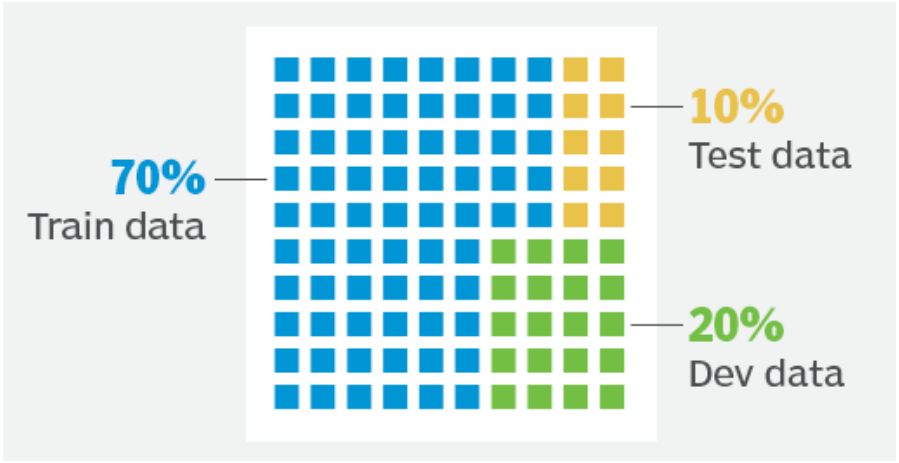
Step-by-step

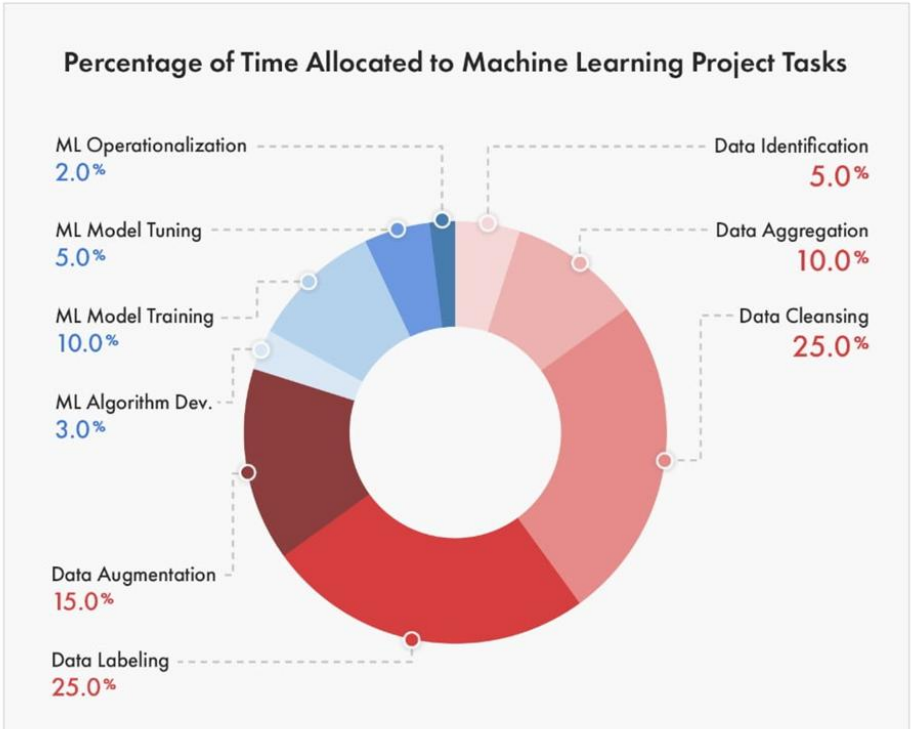
```python
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
torch.cuda.is_available()
```

Python

```
False
```

Verificar GPU disponible





Dividir los datos

```python
train_set = torchvision.datasets.MNIST("./data/", train=True, download=True)
valid_set = torchvision.datasets.MNIST("./data/", train=False, download=True)
```

Python

El "Hello word" del DL

Step-by-step

```python
train_set = torchvision.datasets.MNIST("./data/", train=True, download=True)
valid_set = torchvision.datasets.MNIST("./data/", train=False, download=True)
```

Python

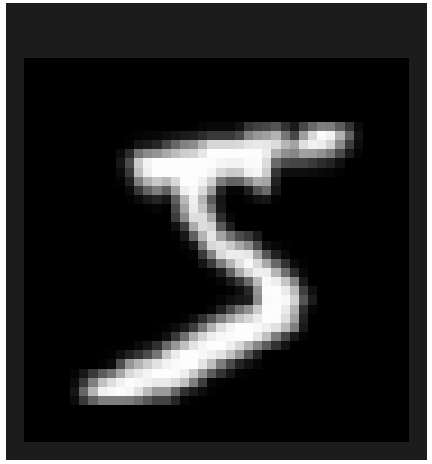
Dividir los datos

Dataset MNIST
Number of datapoints: 60000
Root location: ./data/
Split: Train

Dataset MNIST
Number of datapoints: 10000
Root location: ./data/
Split: Test

```python
x_0, y_0 = train_set[0]
x_0
type(x_0)
y_0
type(y_0)
```

```
5
int
```



`PIL.Image.Image`

El "Hello word" del DL

Step-by-step

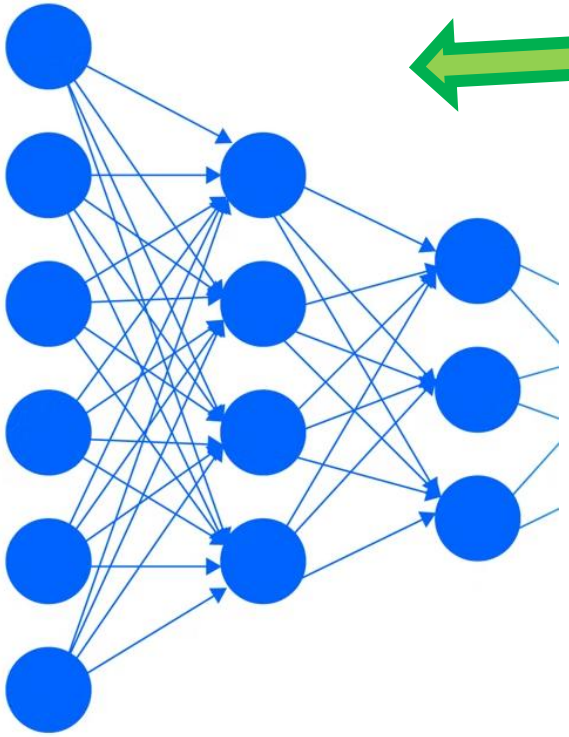
```
trans = transforms.Compose([transforms.ToTensor()])
x_0_tensor = trans(x_0)
```

Convertir los datos a tensores

El resultado '**x_0_tensor**' es: imagen convertida a tensor

¿Por qué es importante?** PyTorch solo trabaja con tensores → Es como traducir de español a inglés para que PyTorch "entienda" los datos.

Ademas, normaliza los valores: convierte los píxeles de rango [0, 255] a rango [0.0, 1.0],



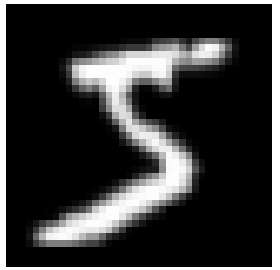
Input data

El modelo no "ve" imágenes, ve vectores.

$$x \in R^{784}$$

```
x_0_tensor.size()
```

```
torch.Size([1, 28, 28])
```



28

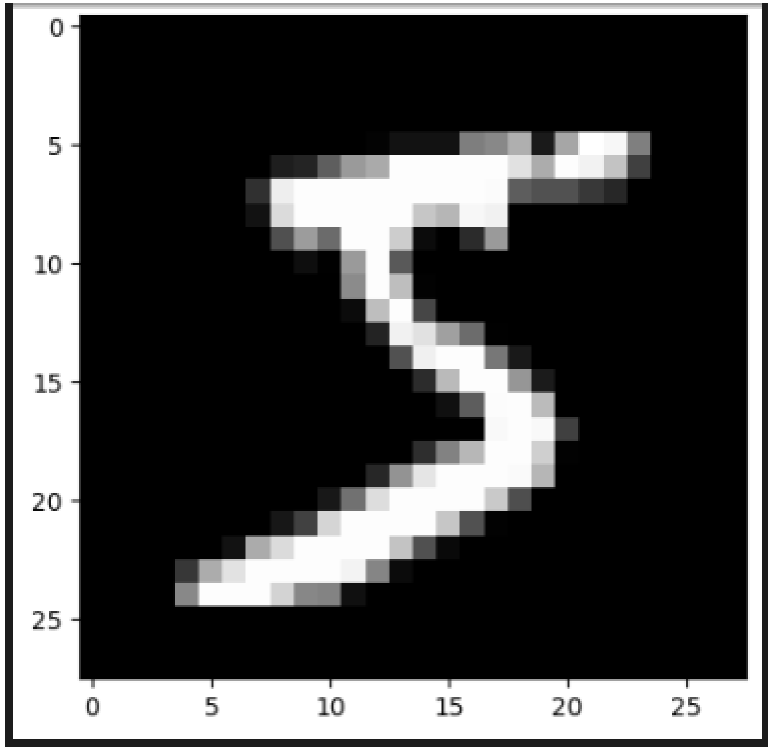
28

El "Hello word" del DL

Step-by-step

```
tensor([[[0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0118, 0.0706, 0.0706, 0.0706,
          0.4941, 0.5333, 0.6863, 0.1020, 0.6510, 1.0000, 0.9686, 0.4980,
          0.0000, 0.0000, 0.0000, 0.0000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
...
```

```
image = F.to_pil_image(x_0_tensor)
plt.imshow(image, cmap='gray')
```



Normalización de los datos

El "Hello word" del DL

Step-by-step

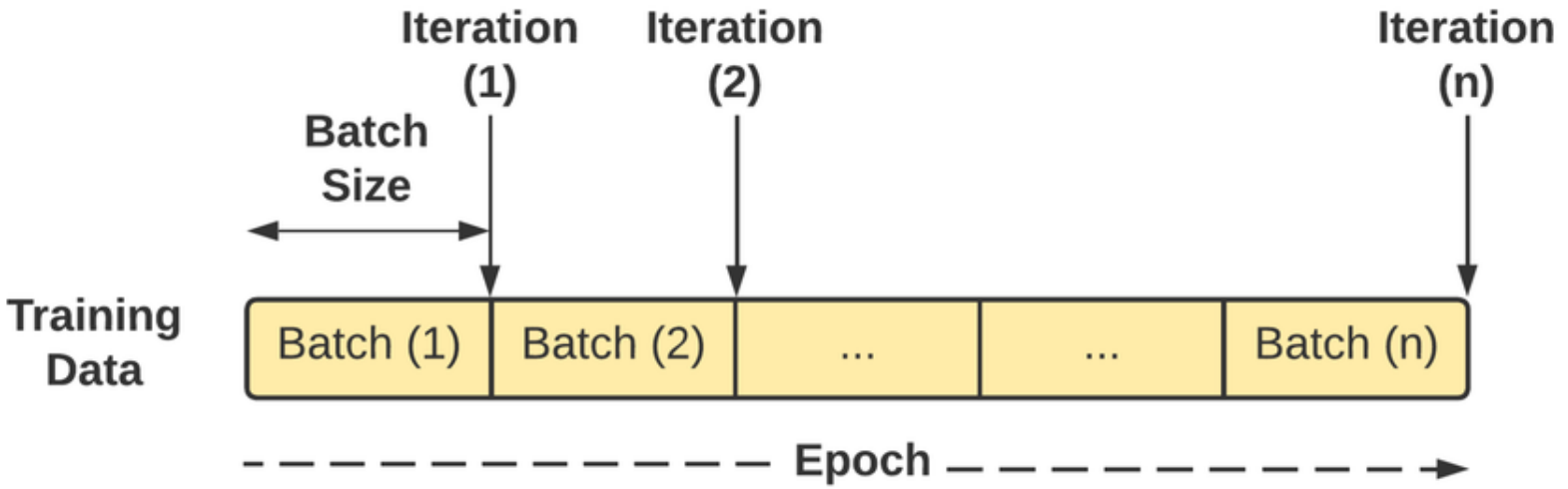
```
batch_size = 32

train_loader = DataLoader(train_set, batch_size=batch_size, shuffle=True)
valid_loader = DataLoader(valid_set, batch_size=batch_size)
```

El modelo no aprende con todos los datos a la vez, aprende por lotes

Balance entre velocidad, estabilidad y generalización

Entre los hiperparámetros, el **learning rate** y el **batch size** son dos parámetros directamente relacionados con el algoritmo del ***gradient descent***.



En DL, los datos de entrenamiento suelen dividirse en lotes más pequeños, cada uno de los cuales se procesa de forma independiente antes de actualizar los parámetros del modelo.

El **batchsize** se refiere al número de muestras utilizadas en cada uno de estos lotes durante el entrenamiento.

El "Hello word" del DL

Step-by-step

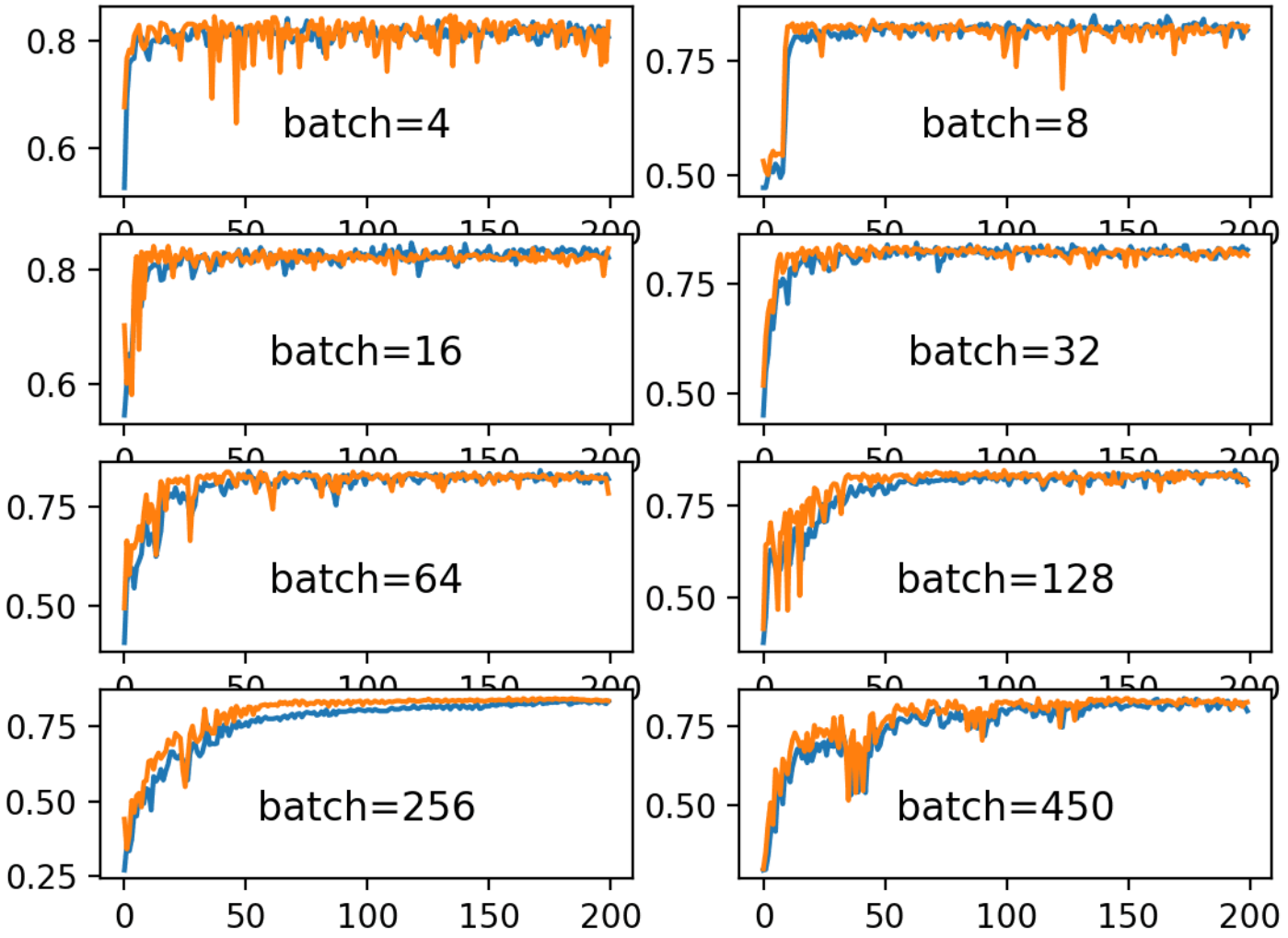
¿Por qué mezclar (`shuffle`)?

- En **entrenamiento**:
  - Se mezcla para evitar que el modelo aprenda patrones por orden

- En **validación**:
  - No es necesario mezclar

¿Por qué no usar todo el dataset de una vez?

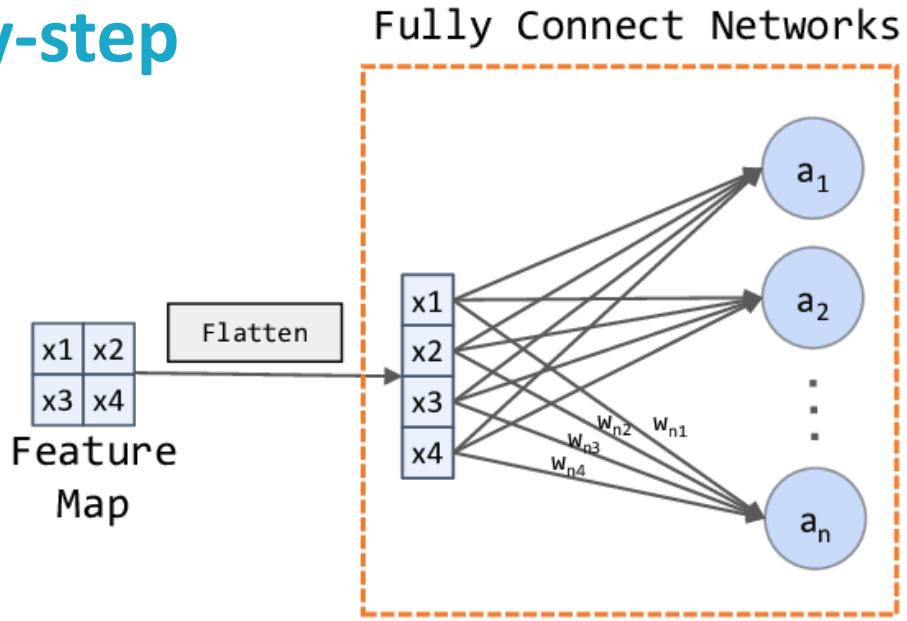
- Consume muchos recursos (RAM / GPU)
- Es menos eficiente
- Puede hacer el aprendizaje inestable
- El gradiente sería muy costoso de calcular

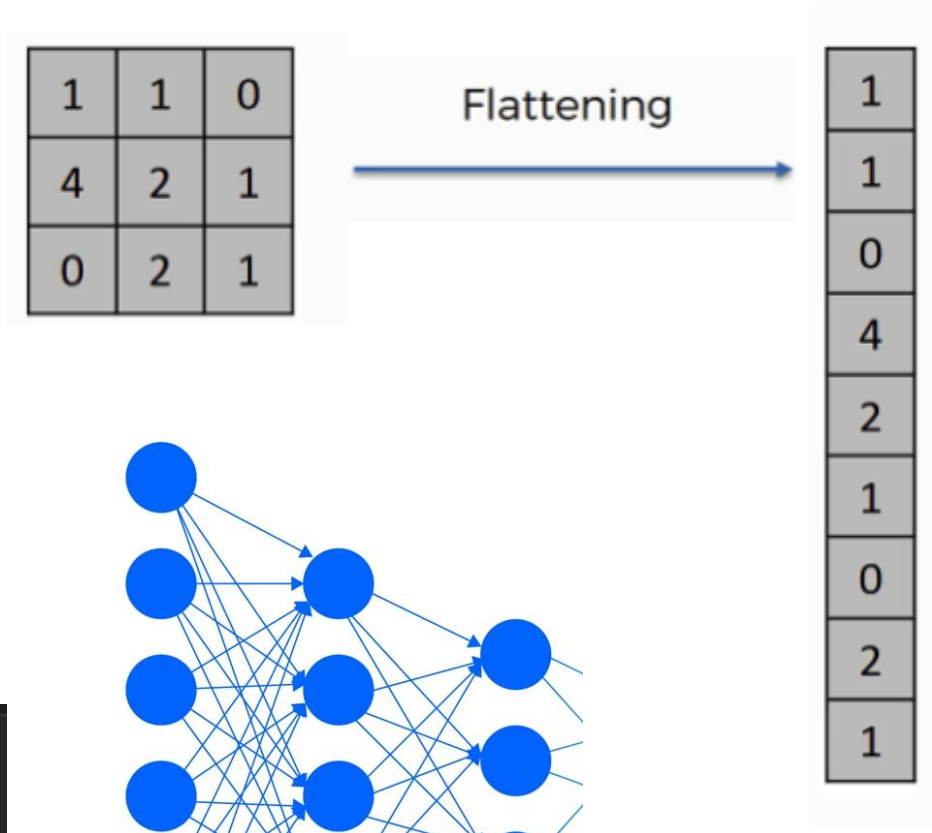


Link

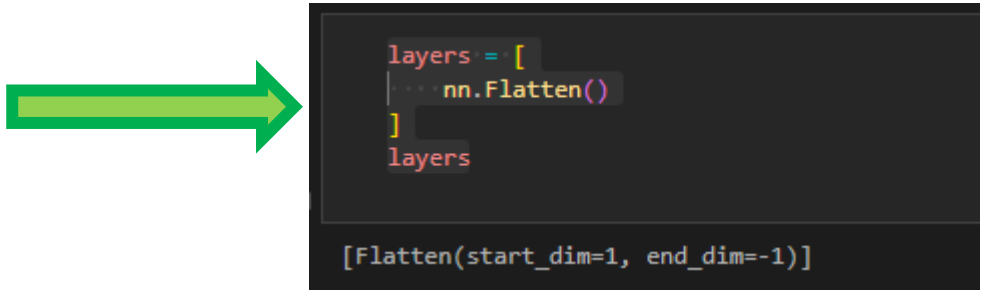
El "Hello word" del DL

Step-by-step





Convierte una imagen multidimensional en un vector de una sola dimensión.



$x \in R^{784}$

Input data

El "Hello word" del DL

Step-by-step

Convierte una imagen multidimensional en un vector de una sola dimensión.

```
layers = [
    nn.Flatten()
]
layers
```

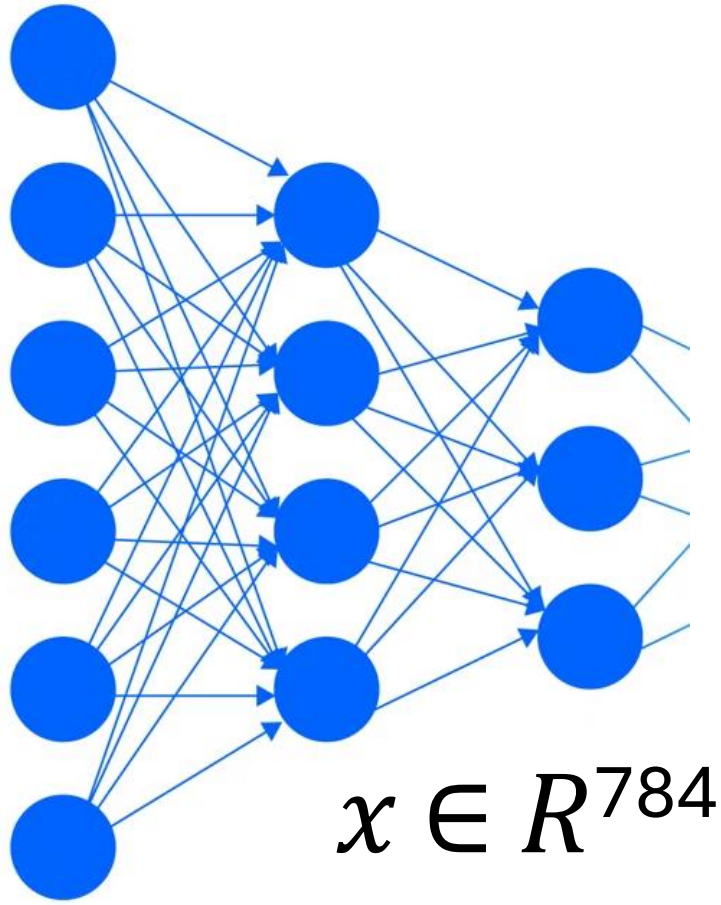
```
[Flatten(start_dim=1, end_dim=-1)]
```

Imagen MNIST original: Forma: (1, 28, 28) → 1 canal, 28x28 píxeles

```
[
[[pixel1, pixel2, ..., pixel28],
 [pixel29, pixel30, ..., pixel56],
 ...
 [pixel757, pixel758, ..., pixel784]]
]
```

Después de Flatten: Forma: (784,) → un vector largo

```
[
  pixel1, pixel2, pixel3, ..., pixel783, pixel784
]
```



$x \in R^{784}$

Input data

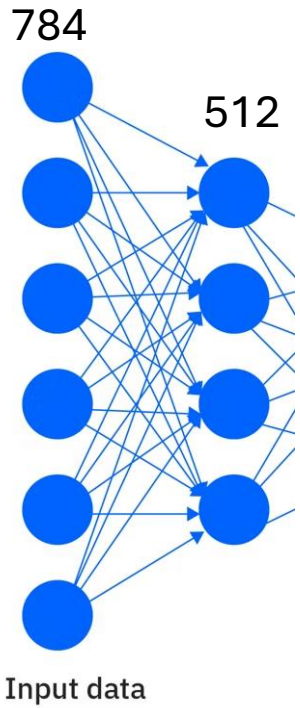
El "Hello word" del DL

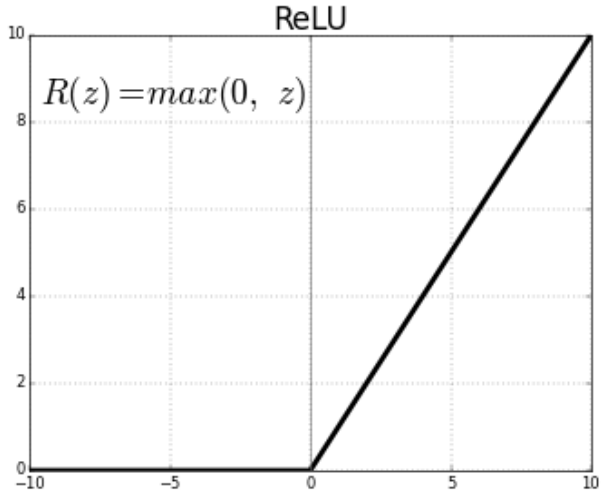
Step-by-step

```
layers = [
    nn.Flatten(),
    nn.Linear(input_size, 512),  # Input
    nn.ReLU(),  # Activation for input
]
layers
```

Entradas y primera capa

> *For example, the rectified linear function g(z) = max{0, z} is not differentiable at z = 0. This may seem like it invalidates g for use with a gradient-based learning algorithm. In practice, gradient descent still performs well enough for these models to be used for machine learning tasks.*
>
> — Page 192, Deep Learning, 2016






```
Antes de ReLU: [-2.3, 0.5, -1.1, 3.2]
Después de ReLU: [0, 0.5, 0, 3.2]
```

Try playing around with this value later to see how it affects training and to start developing a sense for what this number means

El "Hello word" del DL

Step-by-step

```python
n_classes = 10

layers = [
    nn.Flatten(),
    nn.Linear(input_size, 512),  # Input
    nn.ReLU(),  # Activation for input
    nn.Linear(512, 512),  # Hidden
    nn.ReLU(),  # Activation for hidden
    nn.Linear(512, n_classes)  # Output
]
layers
```

Agregar capas ocultas y de salida

$$\text{parámetros} = (\text{input} * \text{output}) + \text{bias}$$

- **Capa 1: Linear(784, 512)**

$$(784 * 512) + 512 = 401.920$$

- **Capa 2: Linear(512, 512)**

$$(512 * 512) + 512 = 262.656$$

- **Capa 3: Linear(512, 10)**

$$(512 * 10) + 10 = 5.130$$

parametros_gpt3 = 175.000.000.000
memoria_gpt3_gb = (parametros_gpt3 * 4) / (1024**3)

Resultado: 650 GB ❌ ¡No cabe en tu GPU de 8 GB

¿Por qué es importante saber el número de parámetros?
Más parámetros = más RAM/GPU necesaria- 669,706

parámetros × 4 bytes (float32) ≈ **2.7 MB**

Pocos parámetros → Underfitting (no aprende bien)
Muchos parámetros → Overfitting (memoriza, no generaliza)

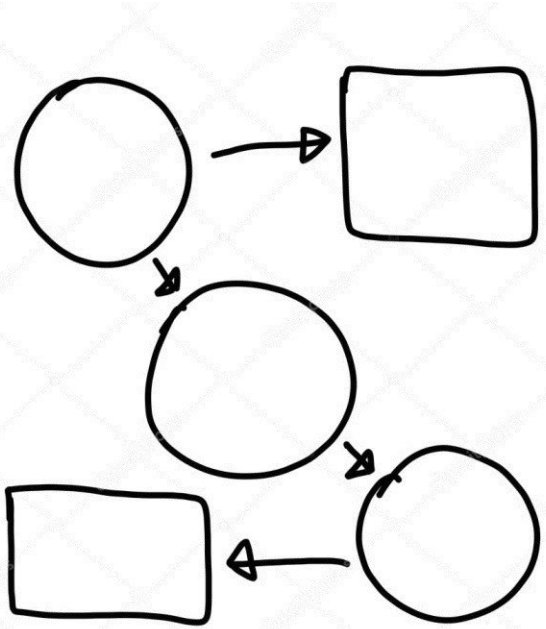
El "Hello word" del DL

Step-by-step

Sequential → es la forma más simple de construir una red neuronal en PyTorch.

```
model = nn.Sequential(*layers)
model
```
⊗ 0.2s

Modelo secuencial



❓ Sin asterisco

```
nn.Sequential(layers)
```

Pasa UNA lista

✓ Con asterisco

```
nn.Sequential(*layers)
```

Desempaqueta → argumentos individuales

Analogía: Desempaquetar una caja

```
lista = ['a', 'b', 'c']

funcion(lista)     → pasa la caja completa
funcion(*lista)    → pasa 'a', 'b', 'c' por separado
```

Link

El "Hello word" del DL

Step-by-step

```
model = torch.compile(model)
```

Optimización del modelo

Ventajas:

Más rápido - Especialmente en GPUs modernas.
Fácil de usar - Solo una línea de código.
Compatible - Funciona con la mayoría de modelos.
Sin cambiar código - Tu modelo sigue funcionando igual.

Desventajas:

No siempre más rápido - En modelos muy pequeños puede ser peor.

```
loss_function = nn.CrossEntropyLoss()
```

Definir la función de perdida

CrossEntropy: está diseñado para calificar si un modelo predijo la categoría correcta de un grupo de categorías.

$$CE = -\sum_{i=1}^{N} y_i log(p_i)$$

Feature (Color)
Red
Green
Yellow
Green
Red

One Hot Encoding →

One Hot Encoded Vector	Red	Green	Yellow
[1,00]	1	0	0
[0,1,0]	0	1	0
[0,0,1]	0	0	1
[0,1,0]	0	1	0
[1,00]	1	0	0

El "Hello word" del DL

Step-by-step

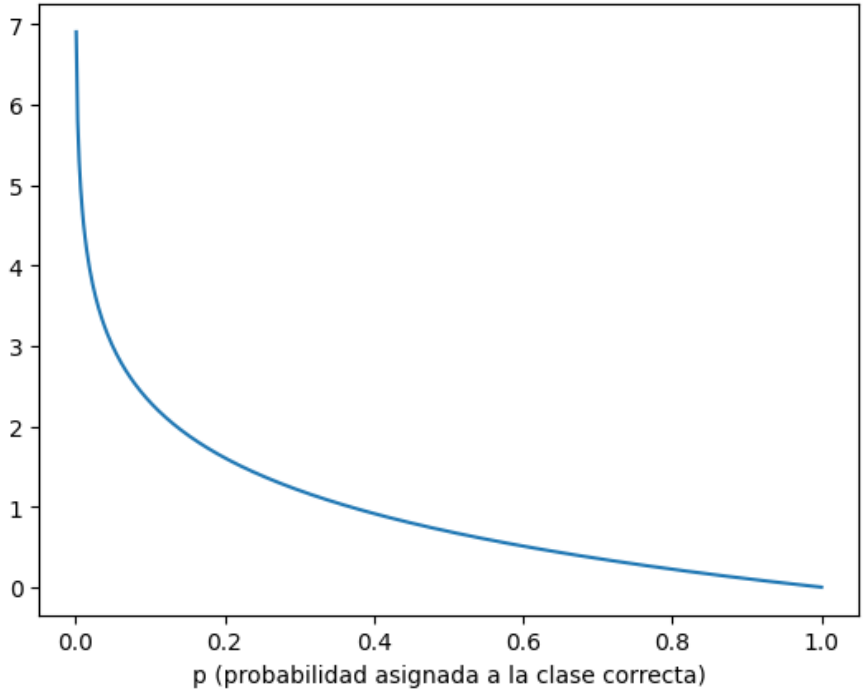
$$CE = -\sum_{i=1}^{N} y_i log(p_i)$$

Supongamos→ Etiqueta verdadera: $y = 6$
one-hot: y = [0,0,0,0,0,0,1,0,0,0]

Softmax produce:

$$p = [0.01, 0.02, 0.03, 0.04, 0.50, 0.10, 0.20, 0.05, 0.03, 0.02]$$

CE = −(0)log(0.01) − (0)log(0.02) … −(1)log(0.20)…



TODO se anula excepto: CE=−log(0.20)

[1,0,0,0,0,0,0,0,0,0]

[0,1,0,0,0,0,0,0,0,0]

[0,0,1,0,0,0,0,0,0,0]

[0,0,0,1,0,0,0,0,0,0]

[0,0,0,0,1,0,0,0,0,0]

[0,0,0,0,0,0,0,0,1,0]

[0,0,0,0,0,0,0,0,0,1]

El "Hello word" del DL

Step-by-step

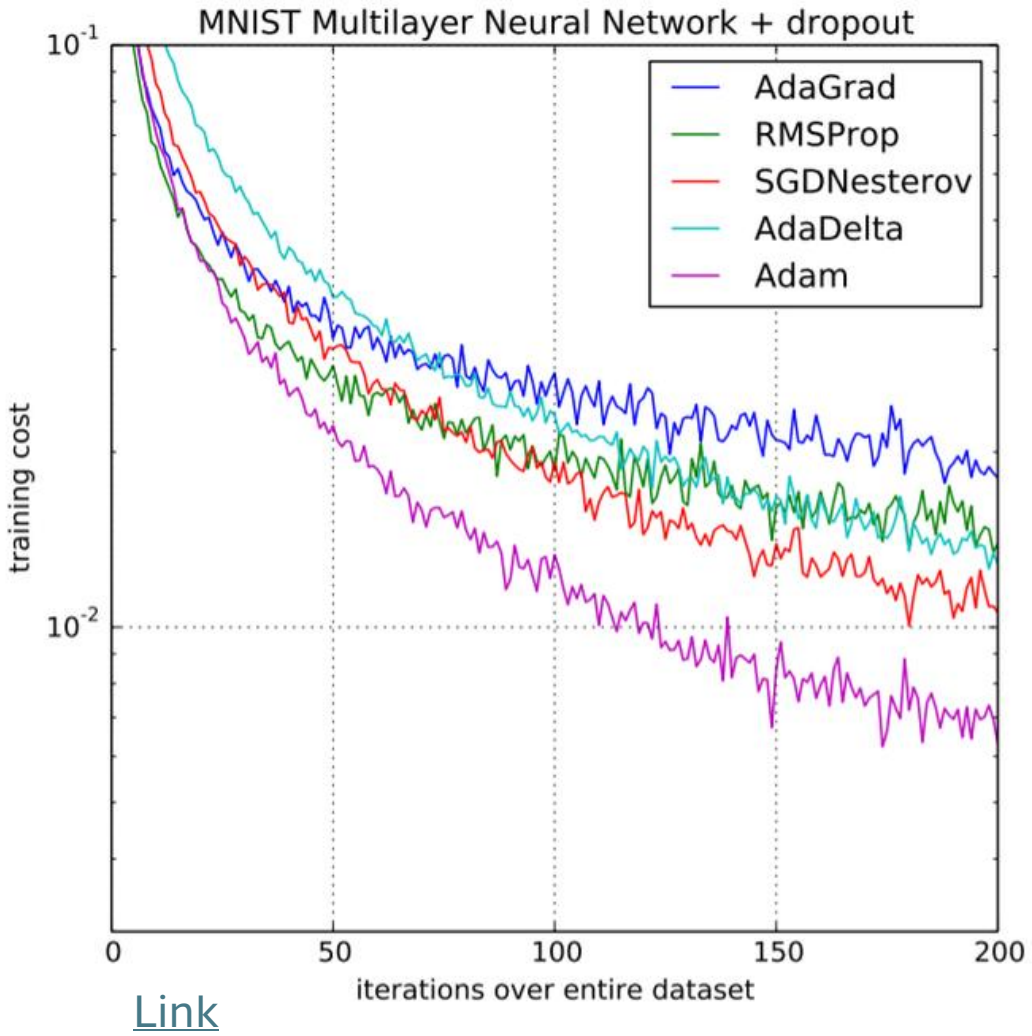
```
optimizer = Adam(model.parameters())
```

Seleccione el optimizador que actualiza los parámetros del modelo durante el entrenamiento

Ajusta el learning rates durante el training.

```
def get_batch_accuracy(output, y, N):
    pred = output.argmax(dim=1, keepdim=True)
    correct = pred.eq(y.view_as(pred)).sum().item()
    return correct / N
```

Función para calcular la precisión (accuracy)



Link

El "Hello word" del DL

Step-by-step

Función para calcular la precisión (accuracy)

```python
def get_batch_accuracy(output, y, N):
    pred = output.argmax(dim=1, keepdim=True)
    correct = pred.eq(y.view_as(pred)).sum().item()
    return correct / N
```

Supongamos que la red procesa 5 imágenes

OUTPUT: Salidas de la red neuronal (logits o probabilidades)
Forma: (5, 10) -> 5 imágenes, 10 posibles dígitos

```
output = torch.tensor([
                        [0.1, 0.05, 0.05, 0.7, 0.02, 0.03, 0.01, 0.02, 0.01, 0.01], # Imagen 1
                        [0.9, 0.02, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01], # Imagen 2
                        [0.05, 0.8, 0.05, 0.02, 0.02, 0.01, 0.01, 0.01, 0.01, 0.02], # Imagen 3
                        [0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.85, 0.05, 0.04], # Imagen 4
                       [0.02, 0.01, 0.75, 0.05, 0.05, 0.03, 0.03, 0.02, 0.02, 0.02] # Imagen 5
                        ])
# Y: Etiquetas verdaderas (qué dígito es realmente cada
imagen)
y = torch.tensor([3, 0, 1, 7, 2])
```

El "Hello word" del DL

Step-by-step

```
pred = output.argmax(dim=1, keepdim=True)
```

```
output = torch.tensor([
                        [0.1, 0.05, 0.05, 0.7, 0.02, 0.03, 0.01, 0.02, 0.01, 0.01], # Imagen 1     → 3
                        [0.9, 0.02, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01], # Imagen 2    → 0
                        [0.05, 0.8, 0.05, 0.02, 0.02, 0.01, 0.01, 0.01, 0.01, 0.02], # Imagen 3    → 1
                        [0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.85, 0.05, 0.04], # Imagen 4 → 7
                       [0.02, 0.01, 0.75, 0.05, 0.05, 0.03, 0.03, 0.02, 0.02, 0.02] # Imagen 5    → 2
                        ])
```

```
pred = [[3], [0], [1], [7], [2]]
```

```
correct = pred.eq(y.view_as(pred)).sum().item()
```

```
pred.eq(y.view_as(pred)) = [[True], # 3 == 3 ✓
                            [True], # 0 == 0 ✓
                            [True], # 1 == 1 ✓
                            [True], # 7 == 7 ✓
                            [True]] # 2 == 2 ✓
```

```
return correct / N # return 5 / 5 = 1.0 (100% Accuracy)
```

```
.sum().item() = 5 # ¡Todas correctas!
```

Step-by-step

Entrenar el modelo

```
epochs = 15

for epoch in range(epochs):
    print('Epoch: {}'.format(epoch))
    train()
    validate()
```

Train = aprender

Validate = examinar

```
def train():
    loss = 0
    accuracy = 0

    model.train()
    for x, y in train_loader:
        x, y = x.to(device), y.to(device)
        output = model(x)
        optimizer.zero_grad()
        batch_loss = loss_function(output, y)
        batch_loss.backward()
        optimizer.step()

        loss += batch_loss.item()
        accuracy += get_batch_accuracy(output, y, train_N)
    print('Train - Loss: {:.4f} Accuracy: {:.4f}'.format(loss, accuracy))
```

```
def validate():
    loss = 0
    accuracy = 0

    model.eval()
    with torch.no_grad():
        for x, y in valid_loader:
            x, y = x.to(device), y.to(device)
            output = model(x)

            loss += loss_function(output, y).item()
            accuracy += get_batch_accuracy(output, y, valid_N)
    print('Valid - Loss: {:.4f} Accuracy: {:.4f}'.format(loss, accuracy))
```

El "Hello word" del DL

Step-by-step

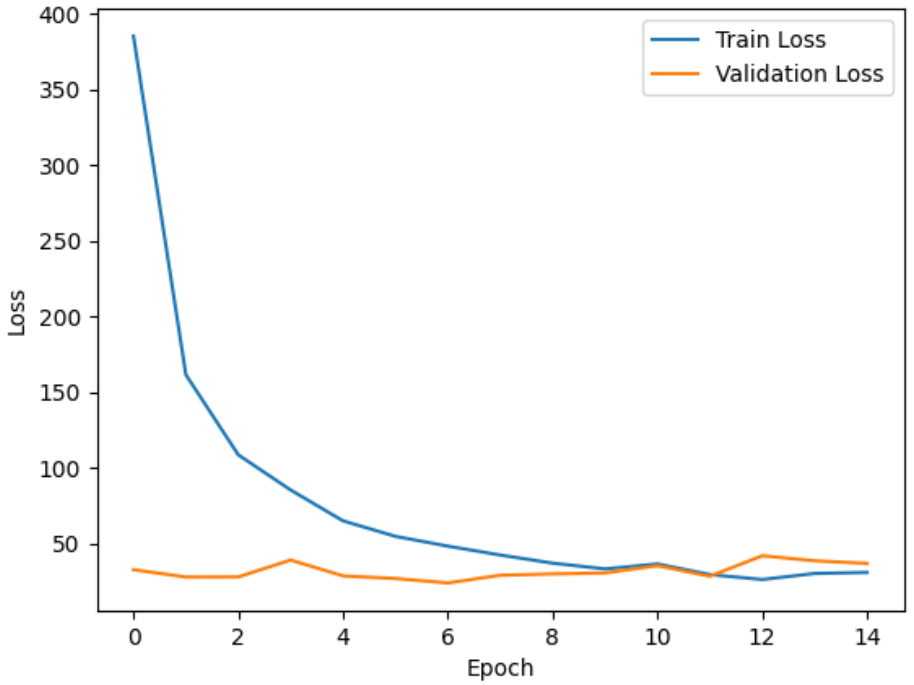
Epoch: 0 Train - Loss: 385.4550 Accuracy: 0.9378	Valid - Loss: 32.4810 Accuracy: 0.9672
Epoch: 1 Train - Loss: 161.4343 Accuracy: 0.9733	Valid - Loss: 27.6381 Accuracy: 0.9744
Epoch: 2 Train - Loss: 108.4484 Accuracy: 0.9808	Valid - Loss: 27.6959 Accuracy: 0.9767
Epoch: 3 Train - Loss: 85.1808 Accuracy: 0.9852	Valid - Loss: 38.8592 Accuracy: 0.9664
Epoch: 4 Train - Loss: 64.7815 Accuracy: 0.9892	Valid - Loss: 28.3647 Accuracy: 0.9755
Epoch: 5 Train - Loss: 54.5347 Accuracy: 0.9907	Valid - Loss: 26.6371 Accuracy: 0.9791
Epoch: 6 Train - Loss: 48.0643 Accuracy: 0.9922	Valid - Loss: 23.7243 Accuracy: 0.9804
Epoch: 7 Train - Loss: 42.1750 Accuracy: 0.9926	Valid - Loss: 28.8064 Accuracy: 0.9782
Epoch: 8 Train - Loss: 36.8053 Accuracy: 0.9938	Valid - Loss: 29.6902 Accuracy: 0.9792
Epoch: 9 Train - Loss: 33.0069 Accuracy: 0.9946	Valid - Loss: 30.3106 Accuracy: 0.9825
Epoch: 10 Train - Loss: 36.2901 Accuracy: 0.9938	Valid - Loss: 35.1231 Accuracy: 0.9800
Epoch: 11 Train - Loss: 29.2552 Accuracy: 0.9956	Valid - Loss: 28.1439 Accuracy: 0.9831
Epoch: 12 Train - Loss: 25.9809 Accuracy: 0.9959	Valid - Loss: 41.6936 Accuracy: 0.9774
Epoch: 13 Train - Loss: 29.9886 Accuracy: 0.9953	Valid - Loss: 38.3812 Accuracy: 0.9783
Epoch: 14 Train - Loss: 30.6800 Accuracy: 0.9957	Valid - Loss: 36.6485 Accuracy: 0.9820

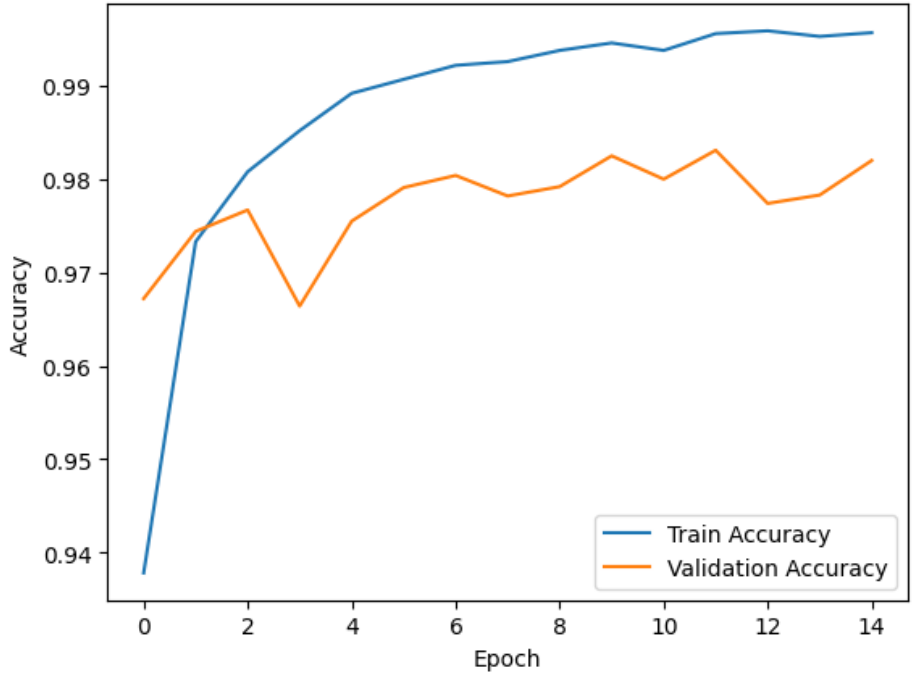
El "Hello word" del DL

Early Stopping + Checkpoints = Garantía de tener el mejor modelo

¿Qué garantiza cada uno?

Técnica	¿Qué hace?	¿Qué garantiza?
Checkpoints	Guarda los pesos del modelo periódicamente	Puedes **recuperar** el modelo de cualquier época guardada
Early Stopping	Para el entrenamiento cuando no mejora	**Ahorra tiempo** y evita overfitting excesivo



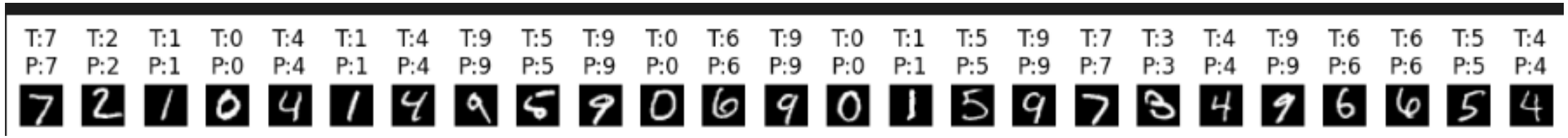


El "Hello word" del DL

tensor([[-56.3176, -47.4944, -58.1196, 2.6680, -54.7834, 32.3323, -29.6542, -40.3872, -37.2434, -25.3665]],

prediction.argmax(dim=1, keepdim=True) `tensor([[5]])`

`y_0` `5`



¿Cómo guardar el modelo e implementarlo para futuras predicciones ?

Keras and TensorFlow

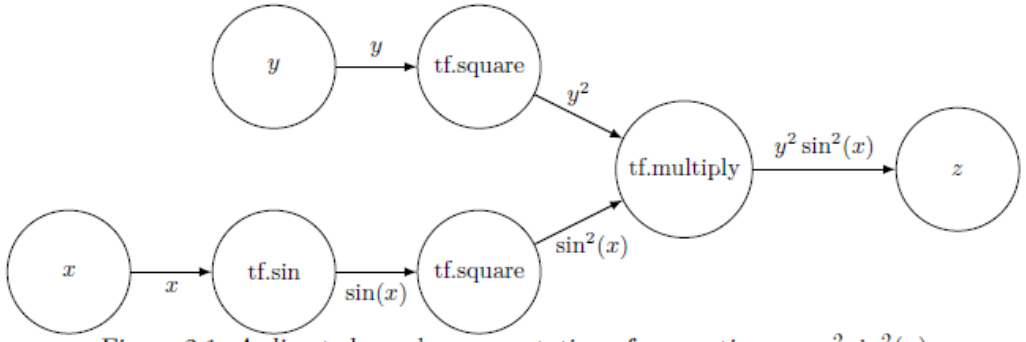
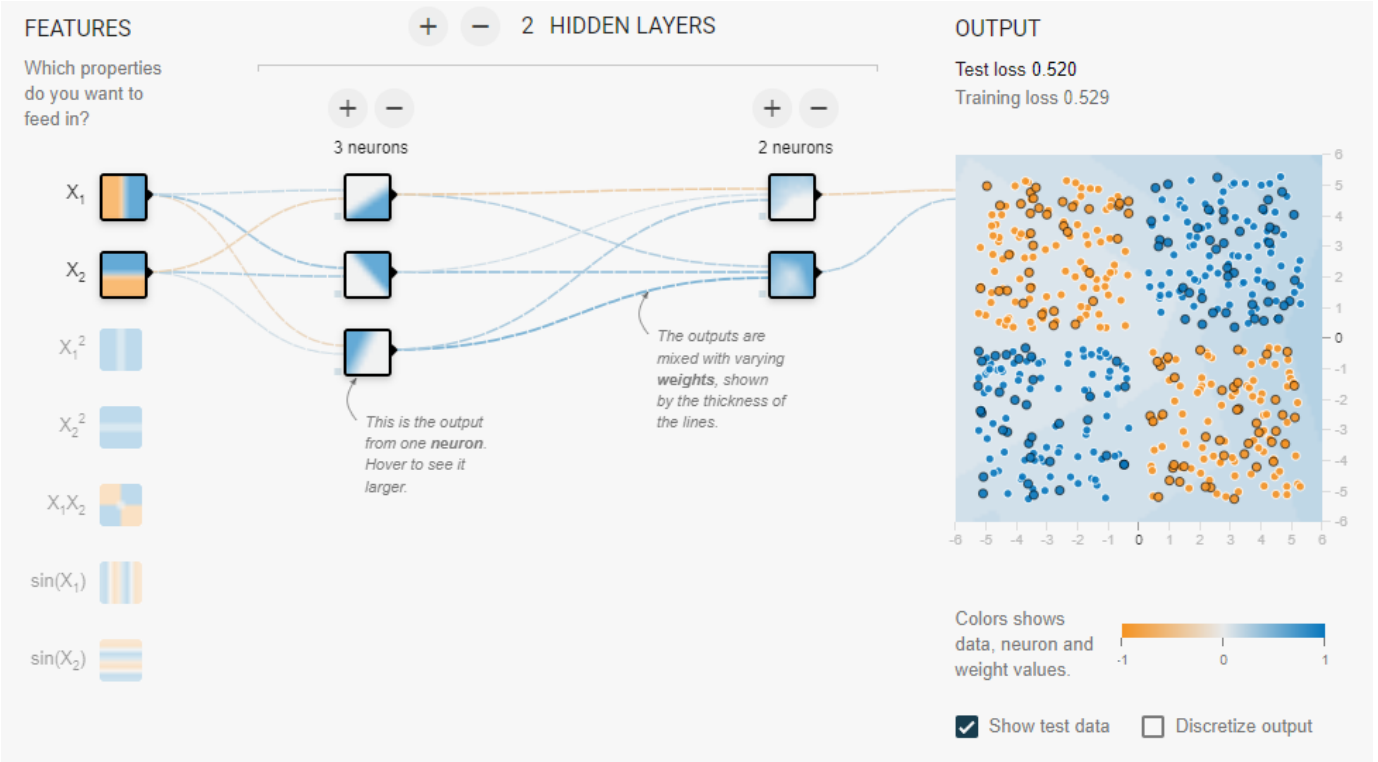


Figure 2.1: A directed graph representation of computing $z = y^2 \sin^2(x)$



A Neural Network Playground (tensorflow.org)