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.

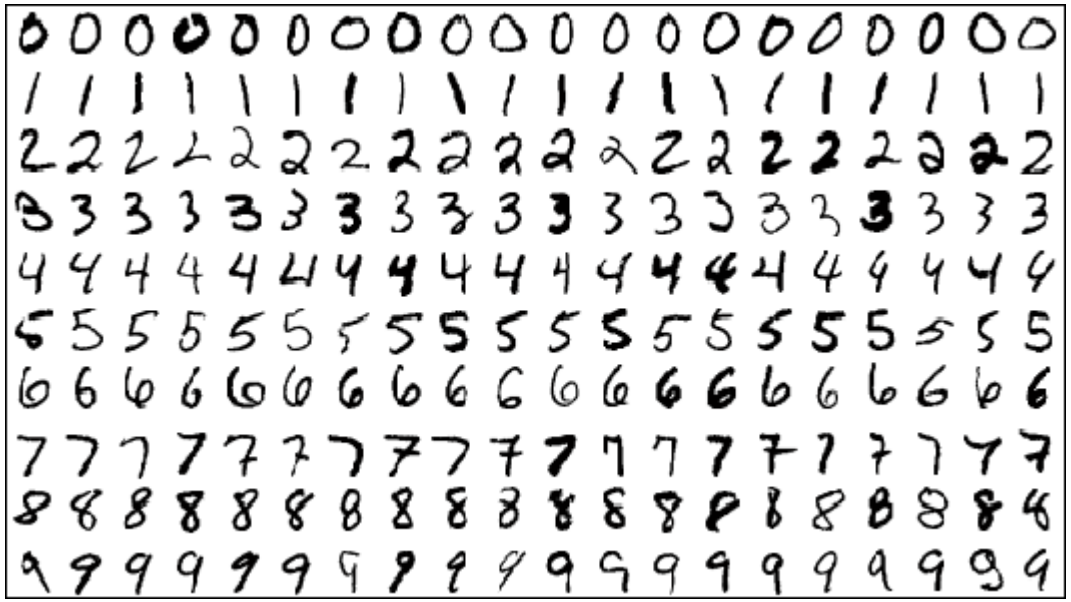
El objetivo es entrenar un modelo que, dada una imagen de entrada, asigne correctamente la probabilidad de pertenencia a cada una de las 10 clases posibles.

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

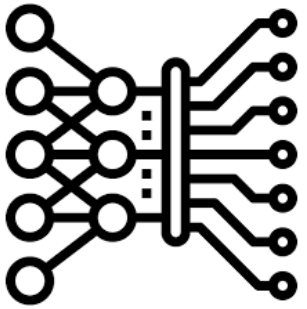
- Cargar y explorar la base de datos (ver ejemplos, tamaños, clases).
- Preprocesar (transformaciones, normalización, flatten si aplica).
- Definir conjuntos de entrenamiento y validación/prueba + DataLoader.
- Definir la arquitectura de la red neuronal (capas, activaciones).
- Configurar el aprendizaje: 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

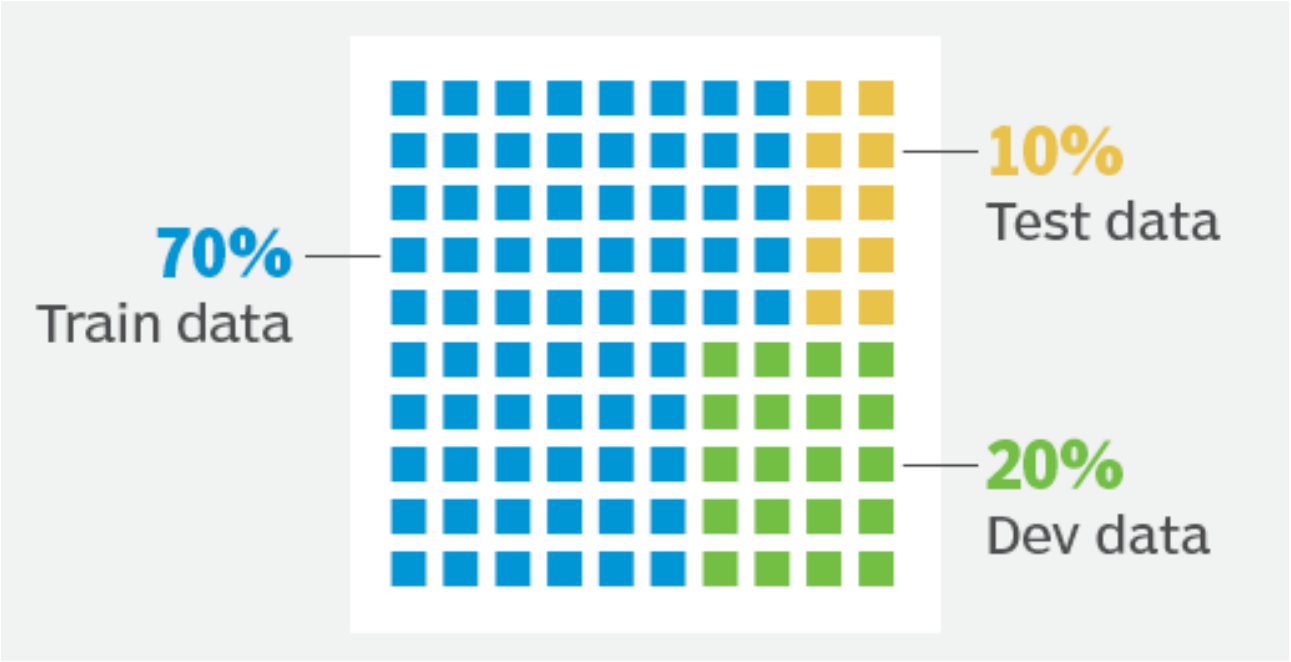


Model

Evaluation

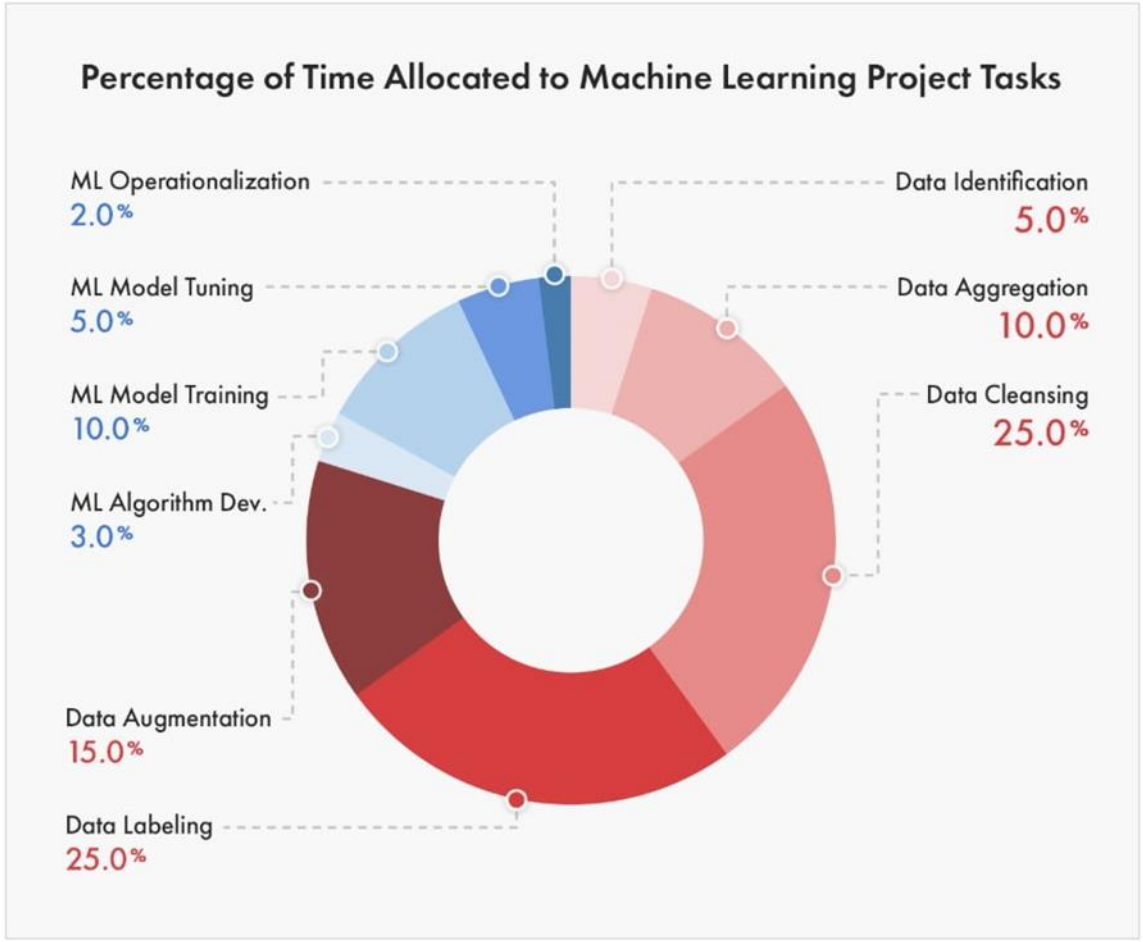
El "Hello word" del DL

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

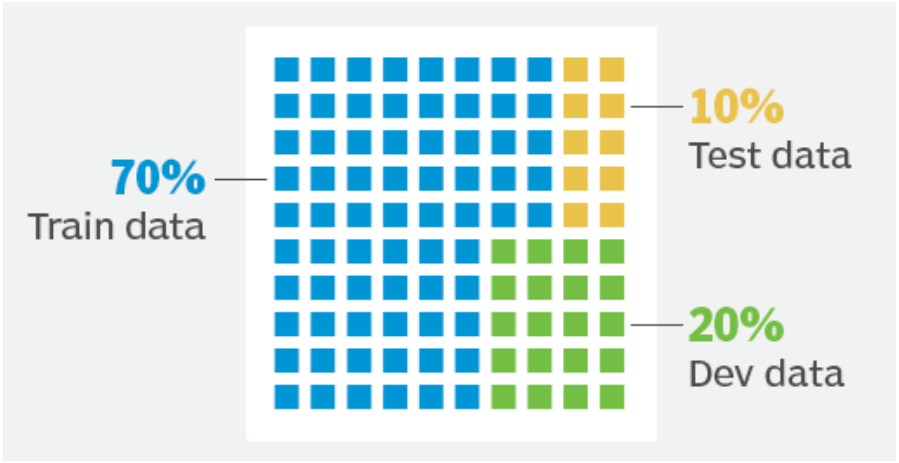
Step-by-step

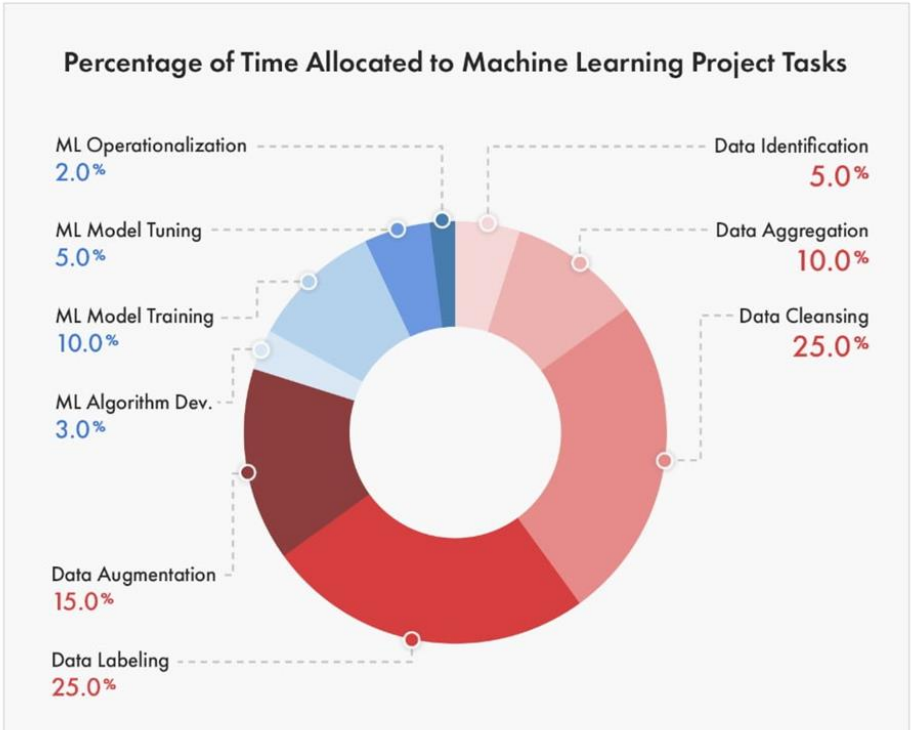
```python
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
torch.cuda.is_available()
```

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

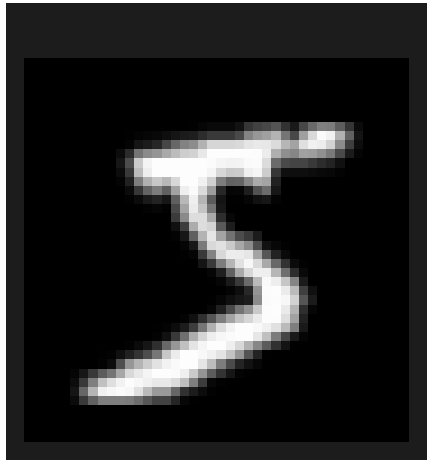
Dividir los datos

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

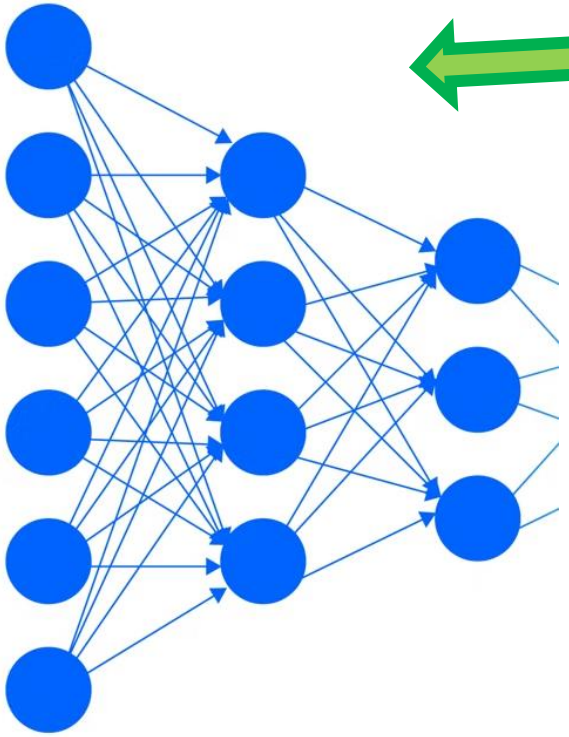
```
trans = transforms.Compose([transforms.ToTensor()])
x_0_tensor = trans(x_0)
```

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],



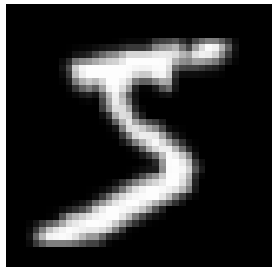
Input data

El modelo no "ve" imágenes, ve vectores.

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

```
x_0_tensor.size()
```

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



28

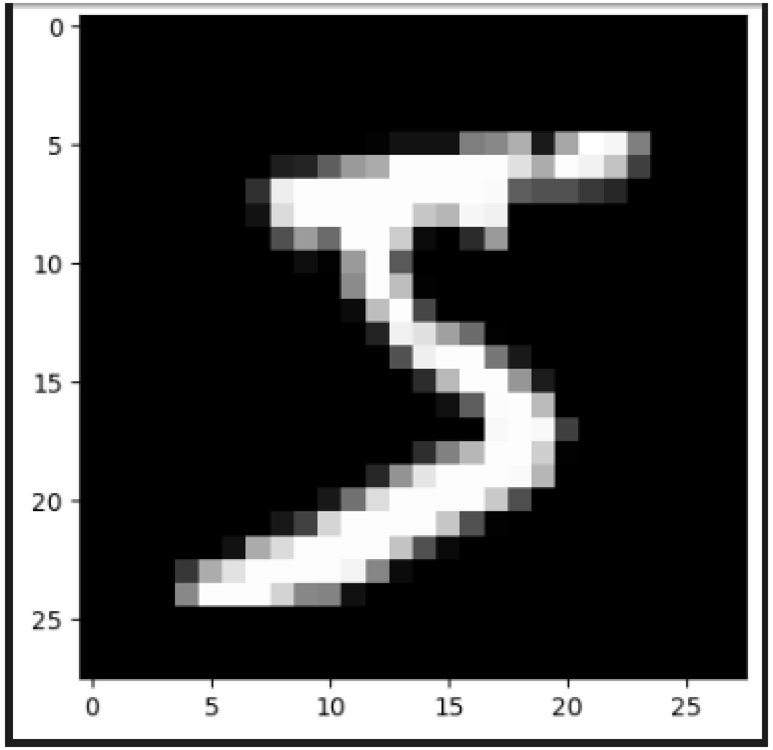
28

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,
...
```

```python
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

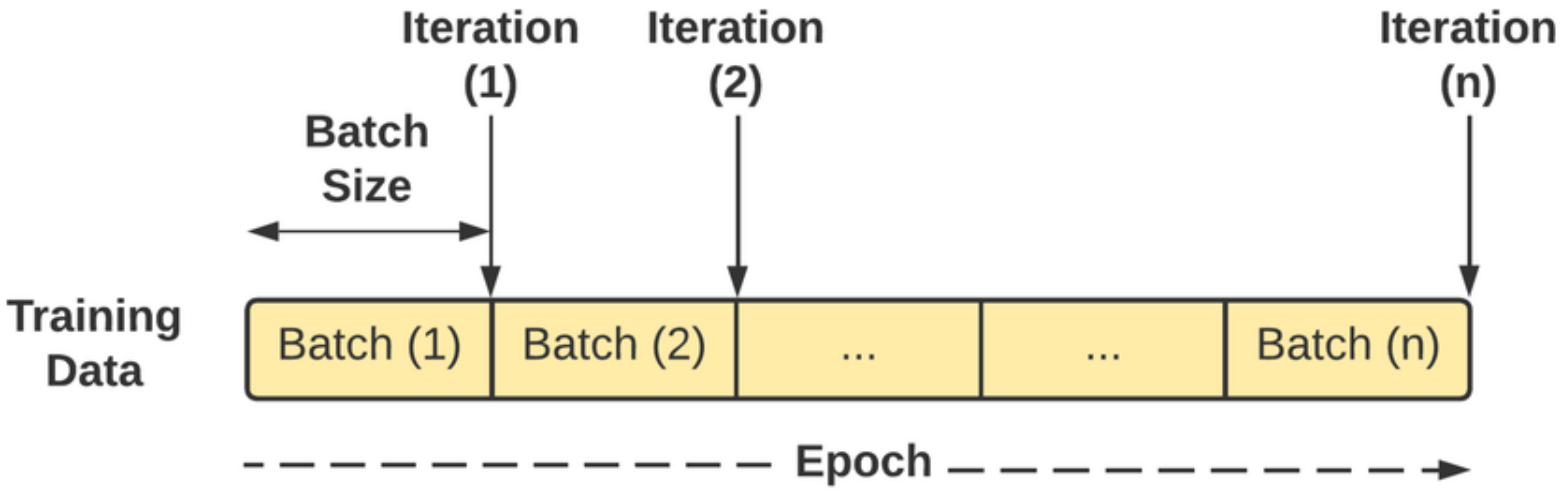
```
batch_size = 32

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

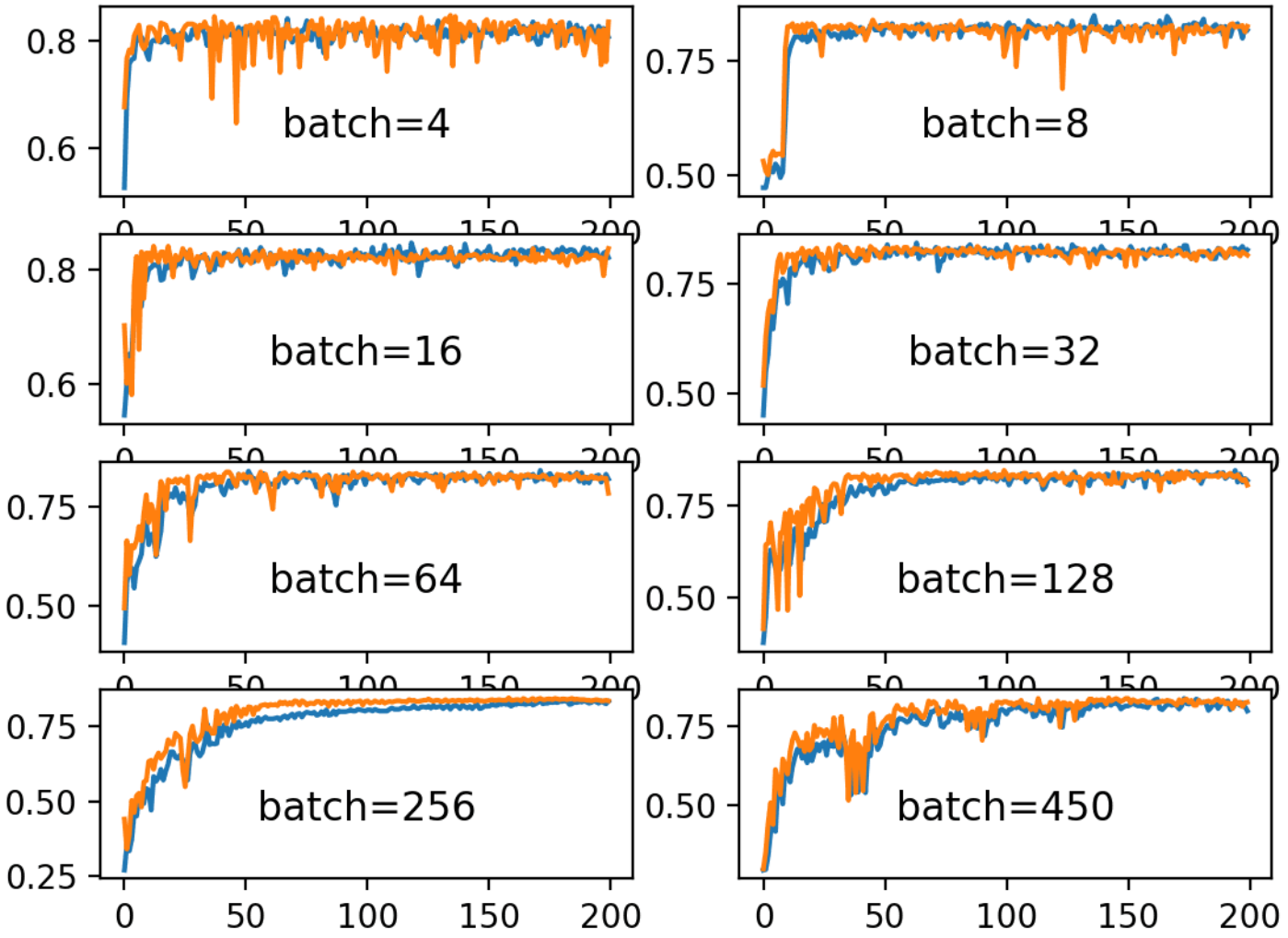
¿Por qué mezclar (`shuffle`)?

- 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?

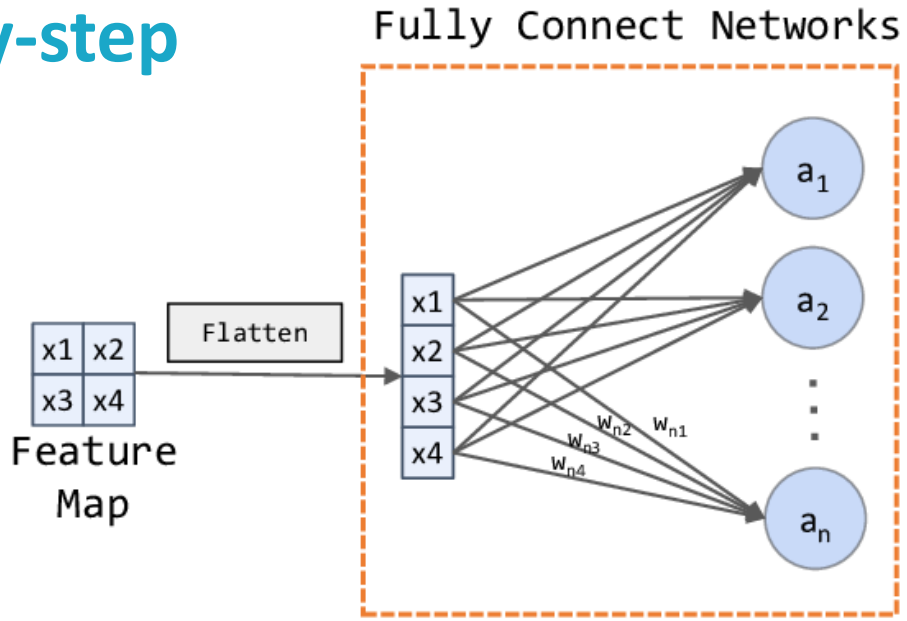
- Consume muchos recursos (RAM / GPU)
- Es menos eficiente
- Puede hacer el aprendizaje inestable
- El gradiente sería muy costoso de calcular

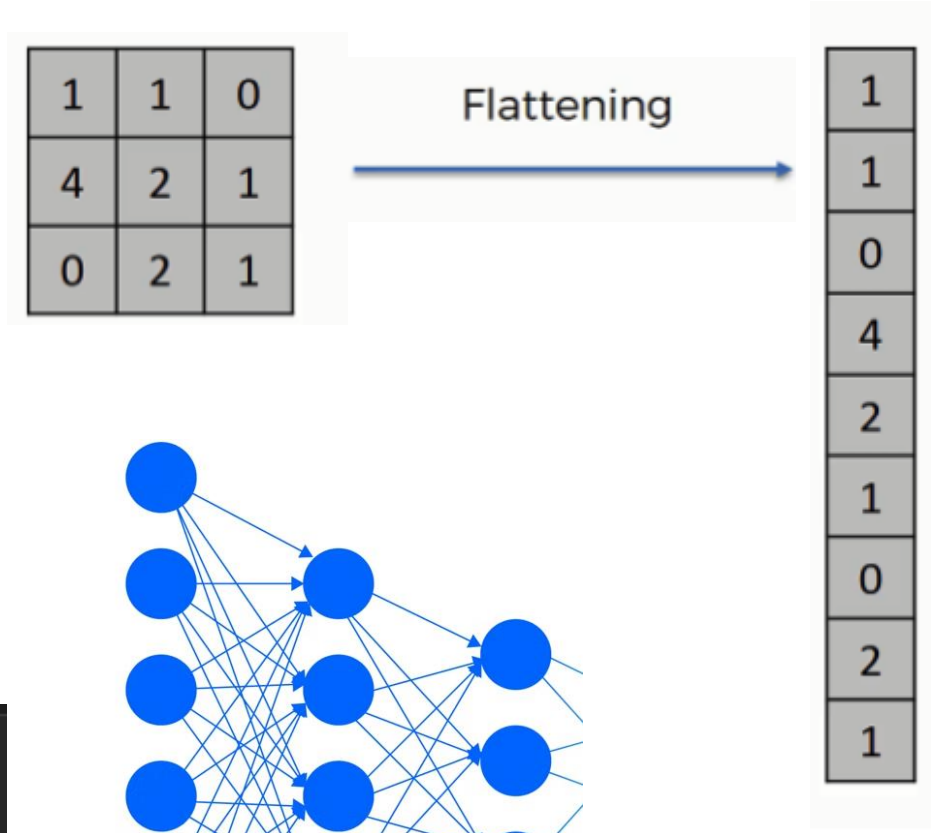


Link

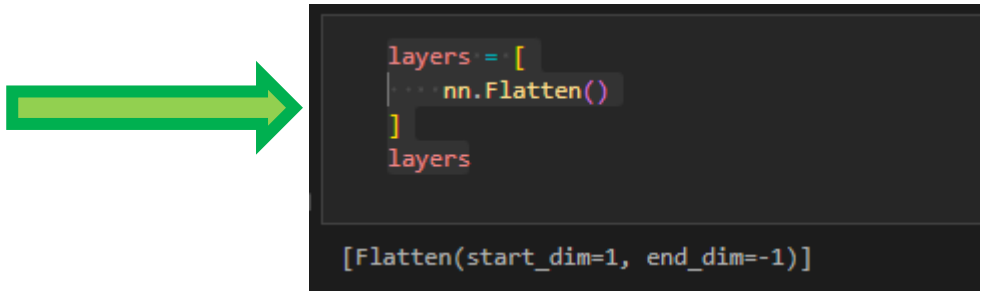
El "Hello word" del DL

Step-by-step





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



$x \in R^{784}$

Input data

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

[

El "Hello word" del DL

Step-by-step

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

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

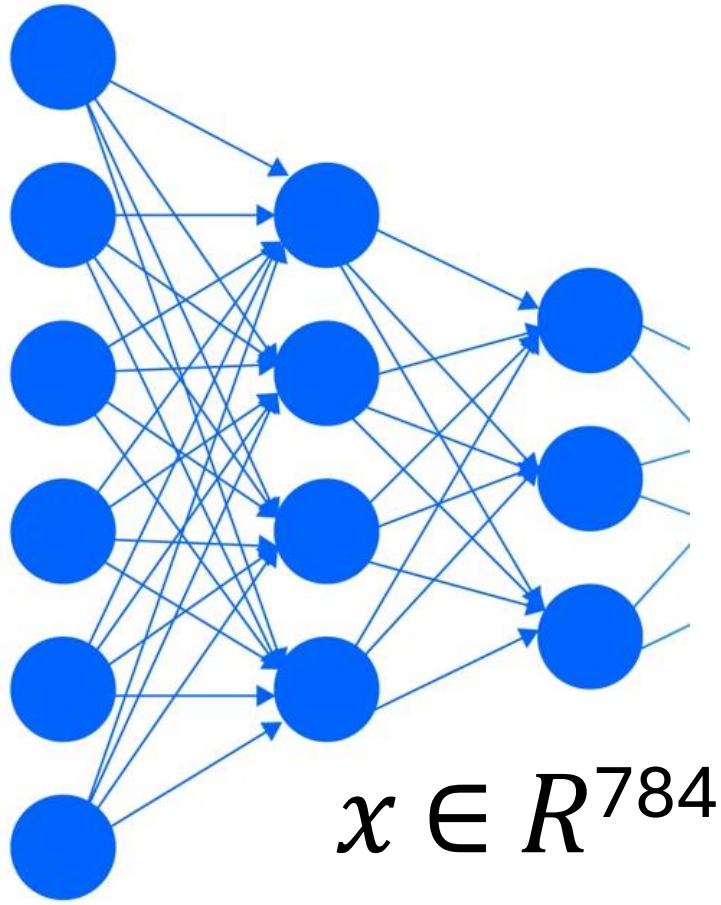
```
[Flatten(start_dim=1, end_dim=-1)]
```

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

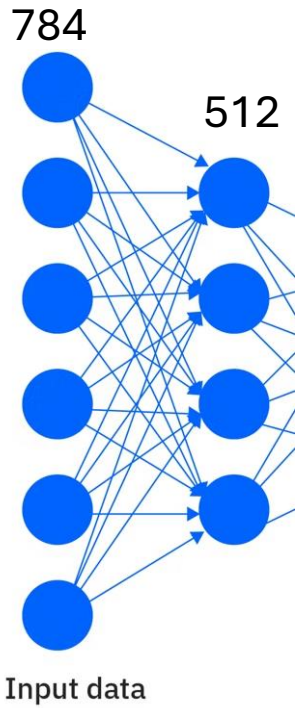
El "Hello word" del DL

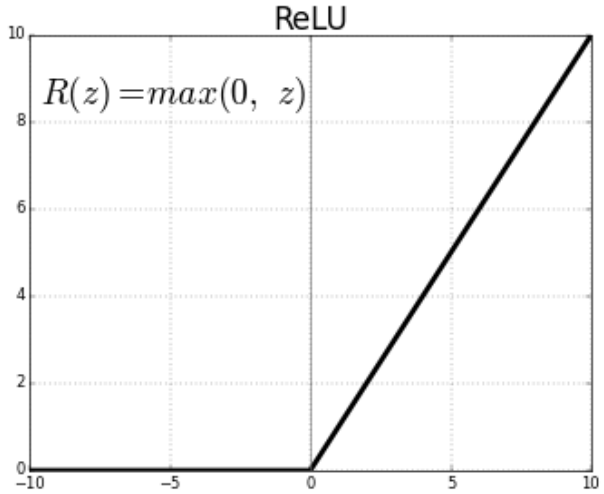
Step-by-step

```
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

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

Agregar capas ocultas

```
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 capa 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)**

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

¿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)

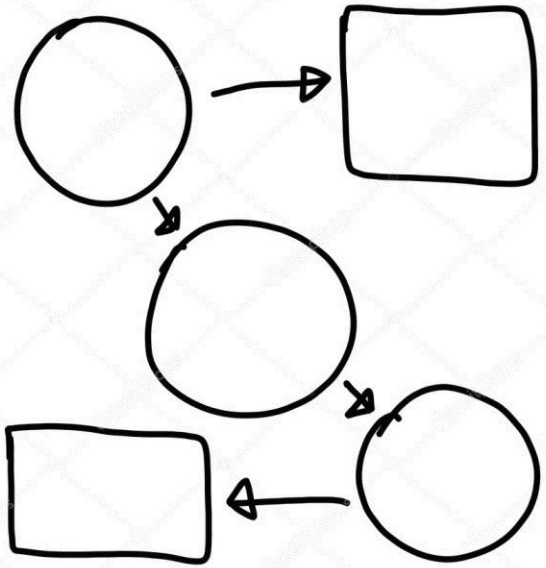
El "Hello word" del DL

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

```
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.

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.

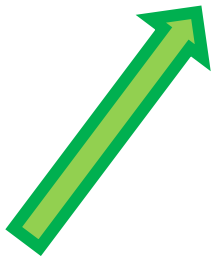
Desventajas:

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

El "Hello word" del DL

Step-by-step

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



Entrenar el modelo

```
epochs = 15

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

Keras and TensorFlow

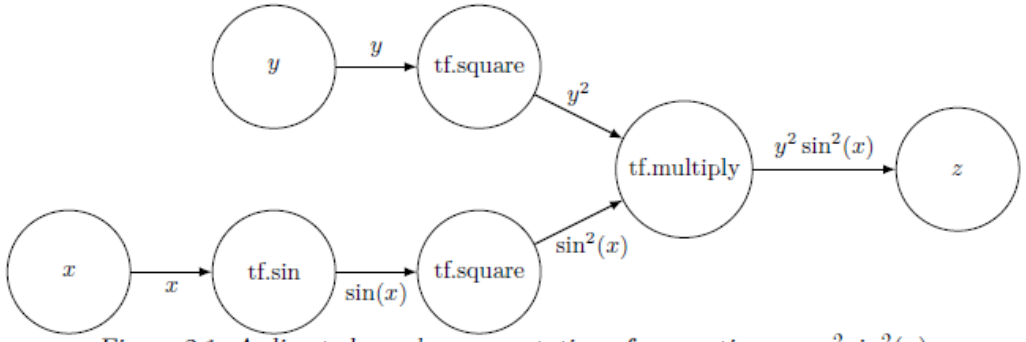
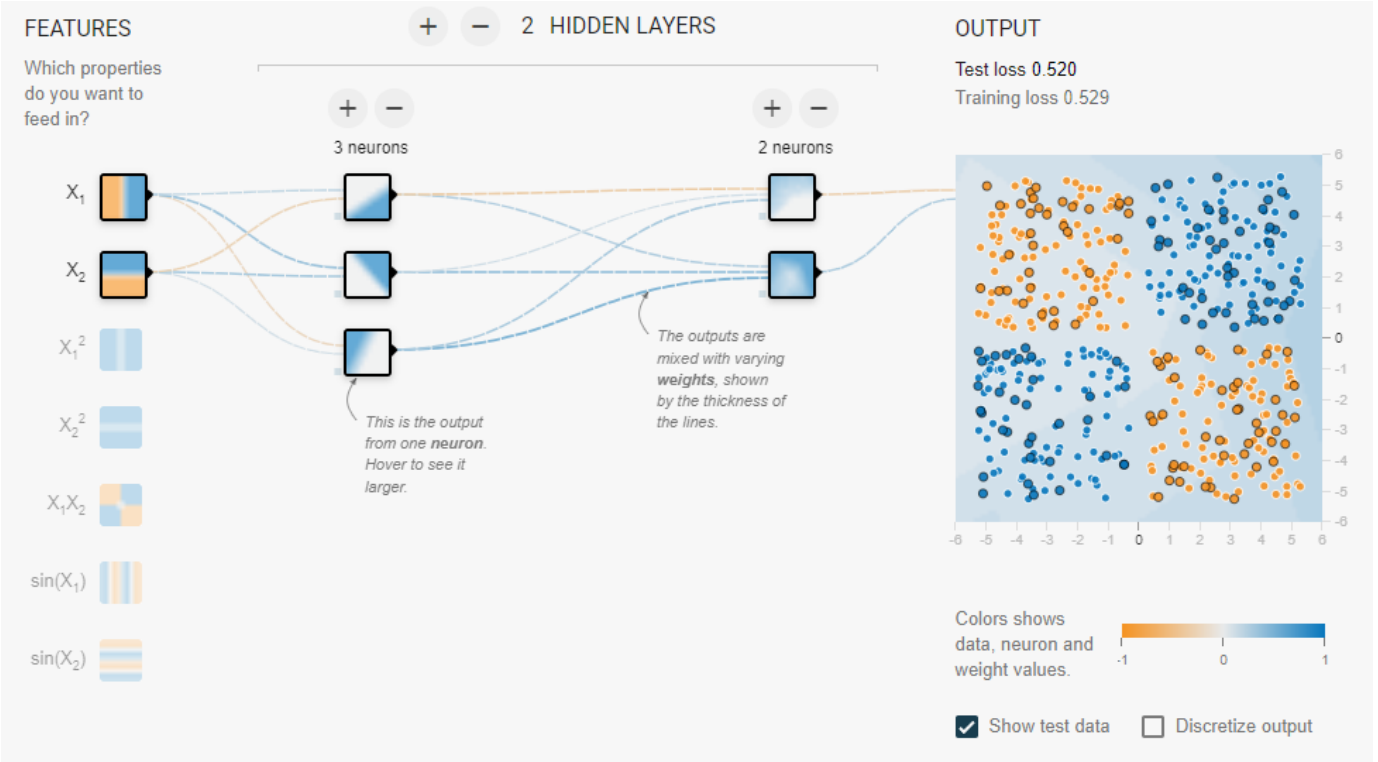


Figure 2.1: A directed graph representation of computing $z = y^2 \sin^2(x)$



[A Neural Network Playground (tensorflow.org)]