

# Clase20 IMA539

Mg. Alejandro Ferreira Vergara

June 12, 2023

## 1 Pytorch para implementar Redes Neuronales

[PyTorch](#) es una de las librerías de aprendizaje profundo más populares disponibles actualmente (junto con [TensorFlow](#)) y nos permite implementar redes neuronales de manera mucho más eficiente que cualquiera de nuestras implementaciones anteriores en NumPy.

Pytorch nos permite realizar cómputo en GPU. Podemos pensar en una tarjeta gráfica como un pequeño clúster informático dentro de nuestra máquina. El reto es que escribir código para dirigirlo a las GPU no es tan sencillo como ejecutar código Python en nuestro intérprete. Existen paquetes especiales, como **CUDA** y **OpenCL**, que nos permiten dirigirnos a la GPU. Sin embargo, escribir código en CUDA u OpenCL no es el entorno más conveniente para implementar y ejecutar algoritmos de aprendizaje automático.

### 1.1 ¿Qué es Pytorch?

**Pytorch** es una interfaz de programación escalable y multiplataforma para implementar y ejecutar algoritmos de aprendizaje automático y, principalmente, de **aprendizaje profundo**.

Fue desarrollado principalmente por investigadores e ingenieros del laboratorio de inteligencia artificial de Facebook (Facebook AI). Su desarrollo también involucra muchas contribuciones de la comunidad.

Fue lanzado inicialmente en septiembre de 2016, es gratuito y de código abierto.

Para mejorar el rendimiento del entrenamiento de los modelos de aprendizaje automático, particularmente modelos de aprendizaje profundo, Pytorch permite su ejecución tanto en CPU como en GPU (además de dispositivos TPU). Sin embargo, su mayor capacidad de rendimiento se puede descubrir cuando se utilizan las GPU's (también las TPU's).

#### Instalación:

En el entorno virtual ima539, ejecutar la siguiente sentencia:

```
(ima539) ~ $ conda install pytorch=2.0.1 torchvision=0.15.2 torchaudio=2.0.2 pytorch-cuda=11.7
```

```
[ ]: import torch

gpu_pytorch = torch.cuda.device_count()
if gpu_pytorch:
    ngpus = torch.cuda.device_count()

print(f'PyTorch está usando la GPU?: {"SI! :)" if gpu_pytorch else "No :("}')
```

```
if gpu_pytorch:
    print(f'{ngpus} GPU reconocida por PyTorch')
```


**Pytorch** está construido para maniúlar **tensores**. Los tensores pueden entenderse como una **generalización de escalares, vectores, matrices, etc.** En este sentido, un escalar puede definirse como un tensor de orden (rango) 0, un vector como un tensor de orden 1, una matriz como un tensor de orden 2 y las matrices apiladas en una tercera dimensión como tensores de orden 3.


Los tensores en PyTorch son similares a los arrays de NumPy, excepto que los tensores de Pytorch están optimizados para la diferenciación automática y pueden ejecutarse en GPU's.

```
[1]: from IPython.display import Image

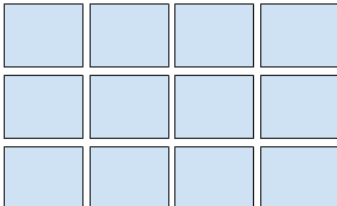
Image(filename=r'clase20/19_1.png', width=500)
```

[1]:

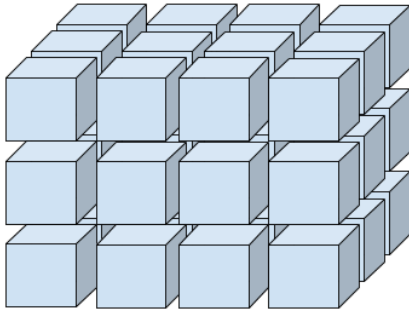
Rank 0:   
(scalar)

Rank 1:   
(vector)

Rank 2: (matrix)



Rank 3:



## 1.2 Creación de tensores con Pytorch

```
[ ]: import numpy as np

a = [1, 2, 3]
b = np.array([4, 5, 6], dtype=np.int32)

t_a = torch.tensor(a)
t_b = torch.from_numpy(b)

print(t_a)
print(t_b)
```

```
[ ]: t_ones = torch.ones(2, 3)

print(t_ones.shape)
print(t_ones)

[ ]: rand_tensor = torch.rand(2,3)

print(rand_tensor)

[ ]: m = np.array([[4, 5, 6],[3, 4, 5]], dtype=np.int32)

t_m = torch.from_numpy(m)
t_m

[ ]: m = np.array([[4, 5, 6],[3, 4, 5]],[[4, 5, 6],[3, 4, 5]]], dtype=np.float64)

t_m = torch.from_numpy(m)
t_m

[ ]: t_m.shape

[ ]: torch.manual_seed(1.)

t1 = 2 * torch.rand(5, 2) - 1
t2 = torch.normal(mean=0, std=1, size=(5, 2))
print(t1, '\n', t2)

[ ]: t3 = torch.multiply(t1, t2)
print(t3)

[ ]: t4 = torch.matmul(t1, torch.transpose(t2, 0, 1))
print(t4)

[ ]: t5 = torch.mean(t1, axis=0)
print(t5)

[ ]: t5 = torch.mean(t1, axis=1)
print(t5)

[ ]: norm_t1 = torch.linalg.norm(t1, ord=2, dim=1)
print(norm_t1)

[ ]: np.sqrt(np.sum(np.square(t1.numpy()), axis=1))
```

### 1.2.1 Regresión Lineal por Mínimos Cuadrados

$$z = w \times x + b$$

```
[ ]: import matplotlib.pyplot as plt

X_train = np.arange(10, dtype='float32').reshape((10, 1))
y_train = np.array([1.0, 1.3, 3.1, 2.0, 5.0, 6.3, 6.6,
                    7.4, 8.0, 9.0], dtype='float32')

plt.plot(X_train, y_train, 'o', markersize=10)
plt.xlabel('x')
plt.ylabel('y')

#plt.savefig('figures/12_07.pdf')
plt.show()
```

```
[ ]: from torch.utils.data import TensorDataset
from torch.utils.data import DataLoader

X_train_norm = (X_train - np.mean(X_train)) / np.std(X_train)
X_train_norm = torch.from_numpy(X_train_norm)

# On some computers the explicit cast to .float() is necessary
y_train = torch.from_numpy(y_train).float()

train_ds = TensorDataset(X_train_norm, y_train)

batch_size = 1
train_dl = DataLoader(train_ds, batch_size, shuffle=True)
```

```
[ ]: torch.manual_seed(1)

weight = torch.randn(1)
weight.requires_grad_()

bias = torch.zeros(1, requires_grad=True)

def model(xb):
    return xb @ weight + bias
```

```
[ ]: def loss_fn(input, target):
    return (input-target).pow(2).mean()
```

```
[ ]: learning_rate = 0.001
num_epochs = 200
log_epochs = 10

for epoch in range(num_epochs):
    for x_batch, y_batch in train_dl:
        pred = model(x_batch)
```

```

    loss = loss_fn(pred, y_batch)
    loss.backward()

    with torch.no_grad():
        weight -= weight.grad * learning_rate
        bias -= bias.grad * learning_rate
        weight.grad.zero_()
        bias.grad.zero_()

    if epoch % log_epochs==0:
        print(f'Epoch {epoch} Loss {loss.item():.4f}')

```

```

[ ]: print('Parámetros finales:', weight.item(), bias.item())

X_test = np.linspace(0, 9, num=100, dtype='float32').reshape(-1, 1)
X_test_norm = (X_test - np.mean(X_train)) / np.std(X_train)
X_test_norm = torch.from_numpy(X_test_norm)
y_pred = model(X_test_norm).detach().numpy()

fig = plt.figure(figsize=(13, 5))
ax = fig.add_subplot(1, 2, 1)
plt.plot(X_train_norm, y_train, 'o', markersize=10)
plt.plot(X_test_norm, y_pred, '--', lw=3)
plt.legend(['Muestras de entrenamiento', 'Recta de Regresión'], fontsize=10)
ax.set_xlabel('x', size=15)
ax.set_ylabel('y', size=15)
ax.tick_params(axis='both', which='major', labelsize=15)

#plt.savefig('figures/12_08.pdf')
plt.show()

```

### 1.3 Implementación de un Multilayer Perceptron

Módulo torch.nn: <https://pytorch.org/docs/stable/nn.html?highlight=nn#module-torch.nn>

```

[ ]: from sklearn.datasets import load_iris
    from sklearn.model_selection import train_test_split

    iris = load_iris()
    X = iris['data']
    y = iris['target']

    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=1./3,
        ↪random_state=1)

```

```
[ ]: from torch.utils.data import TensorDataset
from torch.utils.data import DataLoader

X_train_norm = (X_train - np.mean(X_train)) / np.std(X_train)
X_train_norm = torch.from_numpy(X_train_norm).float()
y_train = torch.from_numpy(y_train)

train_ds = TensorDataset(X_train_norm, y_train)

torch.manual_seed(1)
batch_size = 4
train_dl = DataLoader(train_ds, batch_size, shuffle=True)
```

```
[ ]: import torch.nn as nn

class Model(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super().__init__()
        self.layer1 = nn.Linear(input_size, hidden_size)
        self.layer2 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = self.layer1(x)
        x = nn.Sigmoid()(x)
        x = self.layer2(x)
        x = nn.Softmax(dim=1)(x)
        return x

input_size = X_train_norm.shape[1]
hidden_size = 16
output_size = 3

model = Model(input_size, hidden_size, output_size)
```

```
[ ]: learning_rate = 0.001
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

num_epochs = 100
loss_hist = [0] * num_epochs
accuracy_hist = [0] * num_epochs

for epoch in range(num_epochs):

    for x_batch, y_batch in train_dl:
        pred = model(x_batch)
        loss = loss_fn(pred, y_batch.long())
```

```

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

        loss_hist[epoch] += loss.item()*y_batch.size(0)
        is_correct = (torch.argmax(pred, dim=1) == y_batch).float()
        accuracy_hist[epoch] += is_correct.sum()

    loss_hist[epoch] /= len(train_dl.dataset)
    accuracy_hist[epoch] /= len(train_dl.dataset)

```

```

[ ]: fig = plt.figure(figsize=(8, 3))
    ax = fig.add_subplot(1, 2, 1)
    ax.plot(loss_hist, lw=3)
    ax.set_title('Training loss', size=10)
    ax.set_xlabel('Epoch', size=10)
    ax.tick_params(axis='both', which='major', labelsize=15)

    ax = fig.add_subplot(1, 2, 2)
    ax.plot(accuracy_hist, lw=3)
    ax.set_title('Training accuracy', size=15)
    ax.set_xlabel('Epoch', size=15)
    ax.tick_params(axis='both', which='major', labelsize=15)
    plt.tight_layout()

    #plt.savefig('figures/12_09.pdf')
    plt.show()

```

```

[ ]: X_test_norm = (X_test - np.mean(X_train)) / np.std(X_train)
    X_test_norm = torch.from_numpy(X_test_norm).float()
    y_test = torch.from_numpy(y_test)
    pred_test = model(X_test_norm)

    correct = (torch.argmax(pred_test, dim=1) == y_test).float()
    accuracy = correct.mean()

    print(f'Test Acc.: {accuracy:.4f}')

```

```

[ ]: path = 'iris_classifier.pt'
    torch.save(model, path)

```

```

[ ]: model_new = torch.load(path)

    model_new.eval()

```

```

[ ]: pred_test = model_new(X_test_norm)

```

```

correct = (torch.argmax(pred_test, dim=1) == y_test).float()
accuracy = correct.mean()

print(f'Test Acc.: {accuracy:.4f}')

```

## 2 Examinando Funciones de Activación

### 2.1 Sigmoide

$$\phi(z) = \frac{1}{1 + e^{-z}}$$

```

[ ]: X = np.array([1, 1.4, 2.5])
     w = np.array([0.4, 0.3, 0.5])

def net_input(X, w):
    return np.dot(X, w)

def logistic(z):
    return 1.0 / (1.0 + np.exp(-z))

def logistic_activation(X, w):
    z = net_input(X, w)
    return logistic(z)

print('P(y=1|x) = %.3f' % logistic_activation(X, w))

```

```

[ ]: W = np.array([[1.1, 1.2, 0.8, 0.4],
                  [0.2, 0.4, 1.0, 0.2],
                  [0.6, 1.5, 1.2, 0.7]])

A = np.array([[1, 0.1, 0.4, 0.6]])

Z = np.dot(W, A[0])
y_probab = logistic(Z)

print('Net Input: \n', Z)

print('Output Units:\n', y_probab)

```

```

[ ]: y_class = np.argmax(Z, axis=0)
     print('Predicted class label: %d' % y_class)

```

### 2.2 Softmax

$$\phi(z)_i = \frac{e^{z_i}}{\sum_{j=1}^t e^{z_j}}, \quad i = 1, 2, \dots, t$$



```
[ ]: def softmax(z):
      return np.exp(z) / np.sum(np.exp(z))
```

```
y_probas = softmax(Z)
print('Probabilities:\n', y_probas)
```

```
[ ]: np.sum(y_probas)
```

## 2.3 ReLU

$$\phi(z) = \max(0, z)$$

```
[ ]: def relu(z):
      return np.maximum(z, 0.)
```

```
y_relu = relu(Z)
print(Z)
print(y_relu)
```

```
[ ]: z = np.arange(-3, 3, 0.1)
phi_z = relu(z)

plt.plot(z, phi_z)
plt.axvline(0.0, color='k')
plt.ylim(-0.5, 3.1)
plt.xlabel('z')
plt.ylabel('$\phi(z)$')

plt.yticks([0., 1.5, 3.])
ax = plt.gca()
ax.yaxis.grid(True)

plt.tight_layout()
plt.show()
```