Entrenamiento Regresión Lineal múltiples

Objetivo

- Crear modelos complejos usando las funciones incorporadas de PyTorch.

Tabla de contenido

Crearemos modelos al estilo PyTorch.

- Crear algunos datos
- Crear el modelo y la función de costo al estilo PyTorch
- Entrenar el modelo: Batch Gradient Descent

Preparación

```
In [2]:
import os
os.environ['KMP_DUPLICATE_LIB_OK']='True'
```

```
In [3]:
# Import the libraries we need for this lab

from torch import nn,optim
import torch
import numpy as np
import matplotlib.pyplot as plt

from mpl_toolkits.mplot3d import Axes3D
from torch.utils.data import Dataset, DataLoader
```

Establecemos la semilla aleatoria:

```
In [4]:
# Set the random seed to 1.

torch.manual_seed(1)
```

```
Out[4]:
<torch._C.Generator at 0x26470118290>
```

Usaremos la siguiente función para graficar:

```
In [5]:
# The function for plotting 2D

def Plot_2D_Plane(model, dataset, n=0):
    w1 = model.state_dict()['linear.weight'].numpy()[0][0]
    w2 = model.state_dict()['linear.weight'].numpy()[0][1]
    b = model.state_dict()['linear.bias'].numpy()

    # Data
    x1 = data_set.x[:, 0].view(-1, 1).numpy()
    x2 = data_set.x[:, 1].view(-1, 1).numpy()
    y = data_set.y.numpy()

    # Make plane
    X, Y = np.meshgrid(np.arange(x1.min(), x1.max(), 0.05), np.arange(x2.min(), x2.max(), 0.05))
    yhat = w1 * X + w2 * Y + b

    # Plotting
    fig = plt.figure()
    ax = fig.gca(projection='3d')

    ax.plot(x1[:, 0], x2[:, 0], y[:, 0],'ro', label='y') # Scatter plot

    ax.plot_surface(X, Y, yhat) # Plane plot

    ax.set_xlabel('x1 ')
    ax.set_ylabel('x2 ')
    ax.set_zlabel('y')
    plt.title('estimated plane iteration:' + str(n))
    ax.legend()

    plt.show()
```

Creamos algunos datos

Creamos una clase dataset con características 2-dimensionales:

```
In [6]:
# Create a 2D dataset

class Data2D(Dataset):

    # Constructor
    def __init__(self):
        self.x = torch.zeros(20, 2)
        self.x[:, 0] = torch.arange(-1, 1, 0.1)
        self.x[:, 1] = torch.arange(-1, 1, 0.1)
        self.w = torch.tensor([[1.0], [1.0]])
        self.b = 1
        self.f = torch.mm(self.x, self.w) + self.b
        self.y = self.f + 0.1 * torch.randn((self.x.shape[0],1))
        self.len = self.x.shape[0]

    # Getter
    def __getitem__(self, index):
        return self.x[index], self.y[index]

    # Get Length
    def __len__(self):
        return self.len
```

Creamos un objeto dataset:

```
In [7]:
# Create the dataset object

data_set = Data2D()
```

Creamos el modelo, optimizador, y la función de pérdida total (costo)

Creamos un módulo regresión lineal personalizado:

```
In [8]:
# Create a customized linear

class linear_regression(nn.Module):

    # Constructor
    def __init__(self, input_size, output_size):
        super(linear_regression, self).__init__()
        self.linear = nn.Linear(input_size, output_size)

    # Prediction
    def forward(self, x):
        yhat = self.linear(x)
        return yhat
```

Creamos un modelo. Usamos 2 características: hacemos que el tamaño de la entrada sea 2 y el de la salida 1.

```
In [9]:
# Create the linear regression model and print the parameters

model = linear_regression(2,1)
print("The parameters: ", list(model.parameters()))
```

```
The parameters:  [Parameter containing:
tensor([[ 0.6209, -0.1178]], requires_grad=True), Parameter containing:
tensor([0.3026], requires_grad=True)]
```

Creamos un objeto optimizador. Establezca la tasa de aprendizaje en 0.1. No olvide ingresar los parámetros del modelo en el constructor.


```
In [10]:
# Create the optimizer

optimizer = optim.SGD(model.parameters(), lr=0.1)
```

Creamos la función de criterio que calcula el costo:

```
In [11]:
# Create the cost function

criterion = nn.MSELoss()
```

Creamos un objeto DataLoader. Establecemos el batch_size en 2.

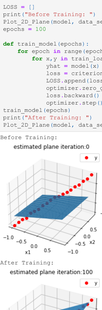
```
In [12]:
# Create the data loader

train_loader = DataLoader(dataset=data_set, batch_size=2)
```

Entrenamos el modelo vía Mini-Batch Gradient Descent

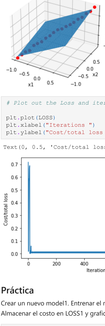
Ejecutamos 100 epochs de Mini-Batch Gradient Descent y almacenamos el costo para cada iteración. Recuerde que este es en realidad una aproximación del costo verdadero.

```
In [13]:
# Train the model

LOSS = []
print("Before Training: ")
Plot_2D_Plane(model, data_set)
epochs = 100

def train_model(epochs):
    for epoch in range(epochs):
        for x,y in train_loader:
            yhat = model(x)
            loss = criterion(yhat, y)
            LOSS.append(loss.item())
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
train_model(epochs)
print("After Training: ")
Plot_2D_Plane(model, data_set, epochs)
```

```
Before Training:
```

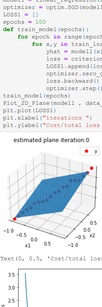



```
After Training:
```




```
In [14]:
# Plot out the Loss and iteration diagram

plt.plot(LOSS)
plt.xlabel("Iterations ")
plt.ylabel("Cost/total loss ")
```

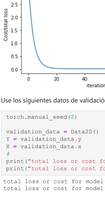
```
Out[14]:
Text(0, 0.5, 'Cost/total loss ')
```



Práctica

Crear un nuevo model1. Entrenar el modelo con un batch_size de 30 y tasa de aprendizaje de 0.1. Almacenar el costo en LOSS1 y graficar.

```
In [15]:
# Practice: create model1. Train the model with batch size 30 and learning rate 0.1, s

data_set = Data2D()
# solution:
train_loader = DataLoader(dataset = data_set, batch_size = 30)
model1 = linear_regression(2, 1)
optimizer = optim.SGD(model.parameters(), lr = 0.1)
LOSS1 = []
epochs = 100
def train_model(epochs):
    for epoch in range(epochs):
        for x,y in train_loader:
            yhat = model1(x)
            loss = criterion(yhat,y)
            LOSS1.append(loss.item())
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
train_model(epochs)
Plot_2D_Plane(model1 , data_set)
plt.plot(LOSS1)
plt.xlabel("iterations ")
plt.ylabel("Cost/total loss ")
```




```
Out[15]:
Text(0, 0.5, 'Cost/total loss ')
```

Use los siguientes datos de validación para calcular el costo para ambos modelos:

```
In [16]:
torch.manual_seed(2)

validation_data = Data2D()
Y = validation_data.y
X = validation_data.x
#
print("total loss or cost for model: ",criterion(model(X),Y))
print("total loss or cost for model: ",criterion(model1(X),Y))
```

```
total loss or cost for model:  tensor(0.0081, grad_fn=<MseLossBackward>)
total loss or cost for model:  tensor(0.0108, grad_fn=<MseLossBackward>)
```