Linear Regression Multiple Outputs

Objetivo

- Crear modelos complejos usando las funciones incorporadas de PyTorch.

Tabla de contenido

In [1]:
```
import os
os.environ['KMP_DUPLICATE_LIB_OK']='True'
```

In [2]:
```
import torch
import numpy as np
import matplotlib.pyplot as plt
from torch import nn,optim
from mpl_toolkits.mplot3d import Axes3D
from torch.utils.data import Dataset, DataLoader
import torchvision.transforms as transforms
```

Establecemos la semilla aleatoria:

In [3]:
```
torch.manual_seed(1)
```

Out[3]:
```
<torch._C.Generator at 0x288c71c1250>
```

Creamos algunos datos

Creamos una clase dataset con características 2-dimensionales y 2 objetivos:

In [4]:
```
from torch.utils.data import Dataset, DataLoader
class Data(Dataset):
    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],[1.0,3.0]])
            self.b=torch.tensor([[1.0,-1.0]])
            self.f=torch.mm(self.x,self.w)+self.b

            self.y=self.f+0.001*torch.randn((self.x.shape[0],1))
            self.len=self.x.shape[0]

    def __getitem__(self,index):

        return self.x[index],self.y[index]

    def __len__(self):
        return self.len
```

Creamos un objeto dataset:

In [5]:
```
data_set=Data()
```

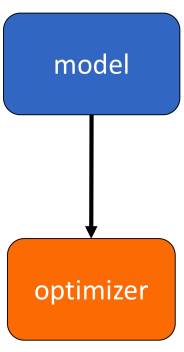
Creamos el modelo, el optimizador y el costo

Creamos un módulo personalizado:

In [6]:
```
class linear_regression(nn.Module):
    def __init__(self,input_size,output_size):
        super(linear_regression,self).__init__()
        self.linear=nn.Linear(input_size,output_size)
    def forward(self,x):
        yhat=self.linear(x)
        return yhat
```

In [7]:
```
model=linear_regression(2,2)
```

Creamos un objeto optimizer y establecemos la tasa de aprendizaje en 0.1.



In [8]:
```
optimizer = optim.SGD(model.parameters(), lr = 0.1)
```

Creamos la función de criterio que calcula el costo:

In [9]:
```
criterion = nn.MSELoss()
```

Creamos un objeto data loader y establecemos el batch size en 5:

In [10]:
```
train_loader=DataLoader(dataset=data_set,batch_size=5)
```

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 esto es una aproximación del costo verdadero.

In [11]:
```
LOSS=[]

epochs=100

for epoch in range(epochs):
    for x,y in train_loader:
        #make a prediction
        yhat=model(x)
        #calculate the loss
        loss=criterion(yhat,y)
        #store loss/cost
        LOSS.append(loss.item())
        #clear gradient
        optimizer.zero_grad()
        #Backward pass: compute gradient of the loss with respect to all the learnable
        loss.backward()
        #the step function on an Optimizer makes an update to its parameters
        optimizer.step()
```

Graficamos:

In [12]:
```
plt.plot(LOSS)
plt.xlabel("iterations ")
plt.ylabel("Cost/total loss ")
plt.show()
```