Redes Neuronales en PyTorch

Tinoco Sergio.
Instituto Politécnico Nacional, Escuela Superior de Cómputo.
Redes neuronales y aprendizaje profundo

28 de marzo de 2023

En el presente documento se va a presentar el desarrollo de una red neuronal completa utilizando el paquete de PyTorch.

Primeramente se va a implementar el perceptrón en PyTorch.

Implementación de un perceptrón en PyTorch

Primeramente se importan los paquetes necesarios para esta implementación.

```
import numpy as np
import torch
```

Se definen las entradas y pesos de la red, así como el sesgo.

Se realiza la operación para generar nuestro perceptrón.

```
#Parametros en Numpy
x=np.array([2.0,3.0,4.0])   #Entrada
w=np.array([0.1,0.1,0.2])   #Pesos
b=np.array([1.0])   #Sesgo

h_np = 1/(1 + np.exp(-1*(np.dot(x,w)+b)))
print(h_np)
```

Posteriormente se convierten cada uno de los valores de entrada a tensores de PyTorch.

```
#Convirtiendo a tensor
X=torch.from_numpy(x)
W=torch.from_numpy(w)
B=torch.from_numpy(b)
```

Mediante los métodos del paquete torch se genera el producto punto de los vectores y posteriormente se calcula el valor de la función sigmoide a dicho producto punto.

```
#Perceptron usando tensores

H=torch.add(torch.dot(X,W),B) #Producto punto y sesgo
Y=torch.sigmoid(H)   #Funcion de activacion sigmoide

print(Y)
```

Resultados

```
[0.90887704]
```

Salida del perceptron mediante funciones de numpy

```
tensor([0.9089], dtype=torch.float64)
```

Salida del perceptron mediante funciones de PyTorch

Implementación de una red neuronal multicapa usando PyTorch

Se importan los paquetes necesarios

```
import matplotlib.pyplot as plt

import numpy as np
import torch
from torchvision import datasets, transforms
```

Se cargan los conjuntos de datos para entrenamiento y pruebas.

```
# Descargamos el conjunto de datos de entrenamiento
trainset = datasets.MNIST('MNIST_data/', download=True, train=True, transform=transform)
# Cargamos el conjunto
batch_size=64
trainloader = torch.utils.data.DataLoader(trainset, batch_size=batch_size, shuffle=True)

# Descargamos y cargamos el conjunto de prueba
testset = datasets.MNIST('MNIST_data/', download=True, train=False, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=batch_size, shuffle=True)
```

Se genera una clase “RedNeuronal” junto con su constructor en el cual se definen las capas que van a constituir nuestro modelo.

```
class RedNeuronal(nn.Module):
    def __init__(self):
        super().__init__()
        # Definir las capas. Cada una con 128, 64 y 10 unidades respectivamente
        self.fc1 = nn.Linear(784, 128)
        self.fc2 = nn.Linear(128, 64)
        # Capa de salida con 10 units (una para cada dígito)
        self.fc3 = nn.Linear(64, 10)
```

Se define la función para el pase frontal de la red.

```
def forward(self, x):
    x = self.fc1(x)
    x = F.relu(x)
    x = self.fc2(x)
    x = F.relu(x)
    x = self.fc3(x)
    x = F.softmax(x, dim=1)

    return x
```

Se crea una instancia de nuestra clase Red Neuronal

```
model = RedNeuronal()
print(model)
```

Se inicializan los pesos y sesgos de la primera capa del modelo. Para el caso de los sesgos inicializan con un valor de 0 y para los pesos se inicializan con una distribución normal aleatoria.

```
model.fc1.bias.data.fill_(0)

model.fc1.weight.data.normal_(std=0.01)
```

Se cargan los datos de entrada correspondientes a las imágenes junto con la etiqueta de cada una.

```
dataiter = iter(trainloader)
images, labels = next(dataiter)
```

Se redimensiona las imágenes a un vector fila de 784 elementos que corresponden al nivel de intensidad de la imagen antes de pasarla por la red.

```
images.resize_(batch_size, 1, 784)
```

Se hace el pase frontal de la primera imagen del conjunto de entrenamiento y se obtiene la predicción estimada por el modelo.

```
img_idx = 0
prediction = model.forward(images[img_idx,:])
```

Resultados.

```
RedNeuronal(
  (fc1): Linear(in_features=784, out_features=128, bias=True)
  (fc2): Linear(in_features=128, out_features=64, bias=True)
  (fc3): Linear(in_features=64, out_features=10, bias=True)
)
```

Objeto de la clase RedNeuronal

```
tensor([[-2.1486e-02, -1.2253e-02, -4.0777e-03,  ...,  2.3371e-02,
         -2.3779e-02, -3.5763e-03],
        [ 1.2511e-03, -2.4997e-02, -1.2716e-03,  ..., -4.7973e-03,
          2.9357e-02,  5.2831e-03],
        [ 9.4265e-04, -2.1907e-02,  1.7580e-02,  ..., -1.0016e-02,
         -1.4664e-02,  1.4871e-02],
        ...,
        [-1.3497e-02,  2.5237e-02, -2.3458e-02,  ...,  3.4679e-02,
          6.1907e-03,  2.4193e-02],
        [ 1.4800e-02,  4.7736e-03, -2.3931e-02,  ..., -8.1078e-04,
          4.1197e-03, -2.3007e-05],
        [-4.3584e-03, -1.7603e-02, -3.2744e-02,  ..., -2.6177e-02,
         -3.3125e-02,  4.5237e-03]], requires_grad=True)
```

Pesos de la red a modo de tensores

```
tensor([[0.0965, 0.0983, 0.0997, 0.1040, 0.0852, 0.1108, 0.0923, 0.1001, 0.1056,
         0.1076]], grad_fn=<SoftmaxBackward0>)
```

Salida de la red con el vector de probabilidades de cada clase.

Conclusiones

Como se pudo observar en los resultados, los tensores nos permiten realizar operaciones matriciales lo cual nos facilita el trabajo a la hora de implementar un modelo de red neuronal. Cabe mencionar que estos tensores tienen ciertas propiedades que nos permiten verlos como un objeto algebraico ya sea tipo matricial, vector o incluso escalar y que de alguna forma tienen la capacidad de modificar su valor como si se tratara de un simple número.