Introducción a Pytorch

- Esteban Sierra Baccio | A00836286

Sección 1

1. Explica brevemente dos ventajas que ofrece PyTorch frente a otros frameworks como TensorFlow o JAX.

Las dos principales ventajas que ofrece PyTorch sobre frameworks como TensorFlow o JAX son su ejecución dinámica y su fuerte adopción en la investigación. La ejecución dinámica de PyTorch, a través de sus gráficos computacionales definidos por ejecución (run-by-run), le da una sintaxis muy "Pythonic" y permite una depuración más fácil e intuitiva, ya que el modelo se comporta como código Python normal, facilitando la impresión de variables y el uso de depuradores estándar. Además, PyTorch se ha convertido en el estándar de facto en la comunidad académica, lo que significa que la mayoría de los papers y modelos de inteligencia artificial de última generación se implementan primero y se publican con código en PyTorch, dando a los investigadores un acceso más rápido a las innovaciones.

2. Define qué es un grafo computacional dinámico y por qué es útil en PyTorch.

Un grafo computacional dinámico (dynamic computational graph, o DCG) es una estructura de datos que define las operaciones matemáticas en una red neuronal y se construye "sobre la marcha" a medida que el código ejecuta las operaciones. A diferencia de los grafos estáticos (usados tradicionalmente por TensorFlow), el grafo dinámico de PyTorch se redefine en cada iteración de forward pass y backward pass. Esta característica es fundamentalmente útil porque permite el uso de control de flujo de Python (como sentencias if, bucles for, o debugging estándar) directamente en la lógica del modelo. Esto hace que la depuración sea mucho más fácil y proporciona una flexibilidad extrema para construir arquitecturas de modelos complejas e irregulares, como redes neuronales recurrentes o modelos con longitudes de secuencia variables.

3. ¿Para qué sirve la clase `torch.utils.data.Dataset` y cómo se utiliza en el manejo de datos personalizados?

La clase torch.utils.data.Dataset es una clase abstracta fundamental en PyTorch que sirve como una interfaz estándar para acceder a los datos, encapsulando la lógica para recuperar muestras individuales y sus etiquetas. Para manejar datos personalizados, se debe heredar de esta clase e implementar dos métodos clave: **len**, que devuelve el número total de muestras, y **getitem**(idx), que carga, preprocesa y devuelve una única muestra y su target dado un índice. Esta estandarización permite que la clase torch.utils.data.DataLoader pueda tomar este objeto Dataset y, de manera eficiente y paralela, agrupar las muestras en mini-

lotes listos para el entrenamiento del modelo, manejando automáticamente el shuffling y la carga multiproceso.

Sección 2

Señala el código en PyTorch que realice lo siguiente:

1. Crear un tensor de forma (2, 3, 4) con valores aleatorios entre 0 y 1.

`torch.rand(2, 3, 4)`

2. Convertir un arreglo de NumPy a un tensor de PyTorch y luego convertirlo de nuevo a un arreglo de NumPy.

In [ ]:
```python
import numpy as np
import torch

# Create a NumPy array
numpy_array = np.array([1, 2, 3, 4])

# Convert to PyTorch tensor
tensor = torch.from_numpy(numpy_array)
print("NumPy Array:", numpy_array)
print("Torch Tensor:", tensor)
```

3. Sumar dos tensores x1 y x2 de forma (2, 3) usando una operación in-place. Explica la diferencia con una operación normal.

In [ ]:
```python
import torch

x = torch.rand(2, 3)
y = torch.rand(2, 3)

# operacion normal
z = x + y

# operacion in place
z += 1

print(z)
```

La diferencia entre una operación normal y una in-place es que una operación normal generalmente crea un nuevo objeto o espacio de memoria para almacenar el resultado, dejando el objeto original sin cambios.

4. Reorganizar x = torch.arange(6) a forma (2, 3) y luego permutar sus dimensiones.

In [4]:
```
import torch
x = torch.arange(6).reshape(2, 3)
print(x)
x = x.permute(1, 0)
print(x)
```

```
tensor([[0, 1, 2],
        [3, 4, 5]])
tensor([[0, 3],
        [1, 4],
        [2, 5]])
```

5. Multiplicar matrices entre x de forma (2, 3) y W de forma (3, 3).

In [8]:
```
import torch
x = torch.rand(6).reshape(2, 3)
y = torch.rand(9).reshape(3, 3)
z = torch.matmul(x, y)
print(z)
```

```
tensor([[0.7757, 1.1916, 0.8522],
        [0.7672, 0.9700, 1.1284]])
```

6. Dado x = torch.arange(12).view(3, 4), obtener:
   A. La segunda columna
   B. La primera fila
   C. Las dos primeras filas y la última columna

In [9]:
```
import torch
x = torch.arange(12).view(3, 4)
print("Matriz x:\n", x)
segunda_columna = x[:, 1]
print("Segunda columna:\n", segunda_columna)
primera_fila = x[0, :]
print("Primera fila:\n", primera_fila)
ultimas_dos_filas_ultima_columna = x[0:2, 3]
print("Dos primeras filas y última columna:\n", ultimas_dos_filas_ultima_columna)
```

```
Matriz x:
 tensor([[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]])
Segunda columna:
 tensor([1, 5, 9])
Primera fila:
 tensor([0, 1, 2, 3])
Dos primeras filas y última columna:
 tensor([3, 7])
```

7. Crear un tensor que requiera gradientes desde su creación y verificar si los está calculando.

In [11]:
```
import torch
x = torch.randn(3, 3, requires_grad=True)
print("Tensor x:\n", x)
y = x + 2
print("Tensor y (x + 2):\n", y)
z = y.mean()
print("Tensor z (mean of y):\n", z)
z.backward()
print("Gradientes de x:\n", x.grad)
```

```
Tensor x:
 tensor([[ 0.5784,  0.5376,  1.7394],
        [-1.5807,  0.4363, -1.0191],
        [ 1.1168,  0.4584, -1.0253]], requires_grad=True)
Tensor y (x + 2):
 tensor([[2.5784, 2.5376, 3.7394],
        [0.4193, 2.4363, 0.9809],
        [3.1168, 2.4584, 0.9747]], grad_fn=<AddBackward0>)
Tensor z (mean of y):
 tensor(2.1380, grad_fn=<MeanBackward0>)
Gradientes de x:
 tensor([[0.1111, 0.1111, 0.1111],
        [0.1111, 0.1111, 0.1111],
        [0.1111, 0.1111, 0.1111]])
```

8. Mover un tensor x a GPU si está disponible. Verificar primero si hay GPU disponible.

In [14]:
```
import torch
x = torch.randn(3, 3)
if torch.cuda.is_available():
    x = x.to('cuda')
    print("Tensor movido a GPU:", x)
else:
    print("GPU no disponible. Tensor en CPU:", x)
```

```
GPU no disponible. Tensor en CPU: tensor([[-0.2808, -0.7496,  0.6518],
        [ 1.8922,  0.8290,  0.4128],
        [ 0.8473, -2.2379,  1.3122]])
```

9. Convertir un tensor que está en GPU a NumPy.

In [15]:
```
import torch
import numpy as np

# Crear un tensor en GPU (si está disponible)
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
gpu_tensor = torch.tensor([1, 2, 3], device=device)

# Mover el tensor a la CPU y convertirlo a NumPy
numpy_array = gpu_tensor.cpu().numpy()
print(numpy_array)  # Salida: [1 2 3]
```

```
[1 2 3]
```

10. Crear una red neuronal simple con una capa lineal de entrada de tamaño 10 y salida de tamaño 5.

In [16]:
```
import torch
import torch.nn as nn

class RedNeuronalSimple(nn.Module):
    def __init__(self):
        super(RedNeuronalSimple, self).__init__()
        self.capa_lineal = nn.Linear(10, 5)

    def forward(self, x):
        return self.capa_lineal(x)

modelo = RedNeuronalSimple()
print("Modelo creado:\n", modelo)
```

```
Modelo creado:
 RedNeuronalSimple(
  (capa_lineal): Linear(in_features=10, out_features=5, bias=True)
)
```

11. Aplicar una función de activación ReLU a un tensor x.

In [18]:
```
import torch
x = torch.tensor([-1.0, 0.0, 1.0, 2.0])
relu = torch.nn.ReLU()
y = relu(x)
print(y)
```

```
tensor([0., 0., 1., 2.])
```