PyTorch en Acción

Diego Quezada

Machine Learning Engineer

22 de Febrero, 2024

Acerca de mí

- Ingeniero Civil Informático 2023.
- Estudiante Magíster en Ciencias de la Ingeniería Informática.
- 2+ años de experiencia en la industria.
- `/diegoquezadac` en LinkedIn.

0. Contenido

1. Introducción a PyTorch

- Librería de Python *open source* e **imperativa** para aprendizaje profundo.
- Desarrollado por Facebook AI Research (FAIR) en el 2017.
- Basado en Torch: un framework para computación científica escrito en Lua.

1.1. Principios de diseño

- Ser Pythonic.
- Poner a los investigadores primero.
- Proveer **desempeño pragmático**.

1.2. Diseño centrado en la usabilidad:

- *Everything is just a program*.
- Interoperabilidad y extensiblidad.
- Diferenciación automática.

2. Fundamentos de PyTorch

- `Tensor` : Matriz multi-dimensional que contiene elementos de un **único** tipo de datos.
- `Dataset` : Colección de datos.
- `DataLoader` : Iterador que permite cargar datos de un `Dataset` .
- `nn.Module` : Clase base para todos los módulos de red neuronal.

2.1. Tensor

- Los tensores son la estructura de datos principal de PyTorch.
- En el contexto del aprendizaje automático, los tensores son utilizados para almacenar y manipular datos, así como los parámetros de los modelos.
- torch.Tensor permite cálculos eficientes y flexibles, integración con NumPy, optimización en GPU y manipulación intuitiva.

```
A = torch.tensor([[1,2,3], [4,5,6], [7,8,9]])
B = torch.zeros(3, 3)
C = torch.randn(3, 3)
D = torch.eye(3)

print(f"Shape of tensor: {A.shape}")
print(f"Datatype of tensor: {A.dtype}")
print(f"Device tensor is stored on: {A.device}")
```

2.2. Dataset

- Los conjuntos de datos son esenciales en el aprendizaje automático.
- Requieren una estructura flexible para su definición.
- La clase torch.utils.data.Dataset ofrece una interfaz estandarizada para esta definición.

```python
class TestDataset(Dataset):

    def __init__(self):
        self.data = torch.randn(10_000, 5)
        self.target = torch.randint(0, 2, (10_000,3))


    def __getitem__(self, index: int):
        x = self.data[index]
        y = self.target[index]
        return x, y

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

2.3. DataLoader

- Iterar a través del conjunto de datos es una práctica común en el proceso de entrenamiento.
- Manejar grandes conjuntos de datos presenta el desafío de una gestión eficiente de la memoria y los recursos del sistema.
- torch.utils.data.DataLoader permite cargar conjuntos de datos eficientemente en *batches*.

```python
dataloader = DataLoader(dataset, batch_size=64, shuffle=True)

for index, (x, y) in enumerate(dataloader):

    print(f'Batch {index}: {y}')
```

2.4. nn.Module

- Controlar la arquitectura de los modelos es crucial para adaptarlos a tareas específicas y mejorar su rendimiento.
- El rápido avance en deep learning demanda una estructura flexible y escalable para la implementación de modelos.
- nn.Module permite implementar modelos de manera imperativa, brindando flexibilidad y control sobre cada componente del modelo en PyTorch.

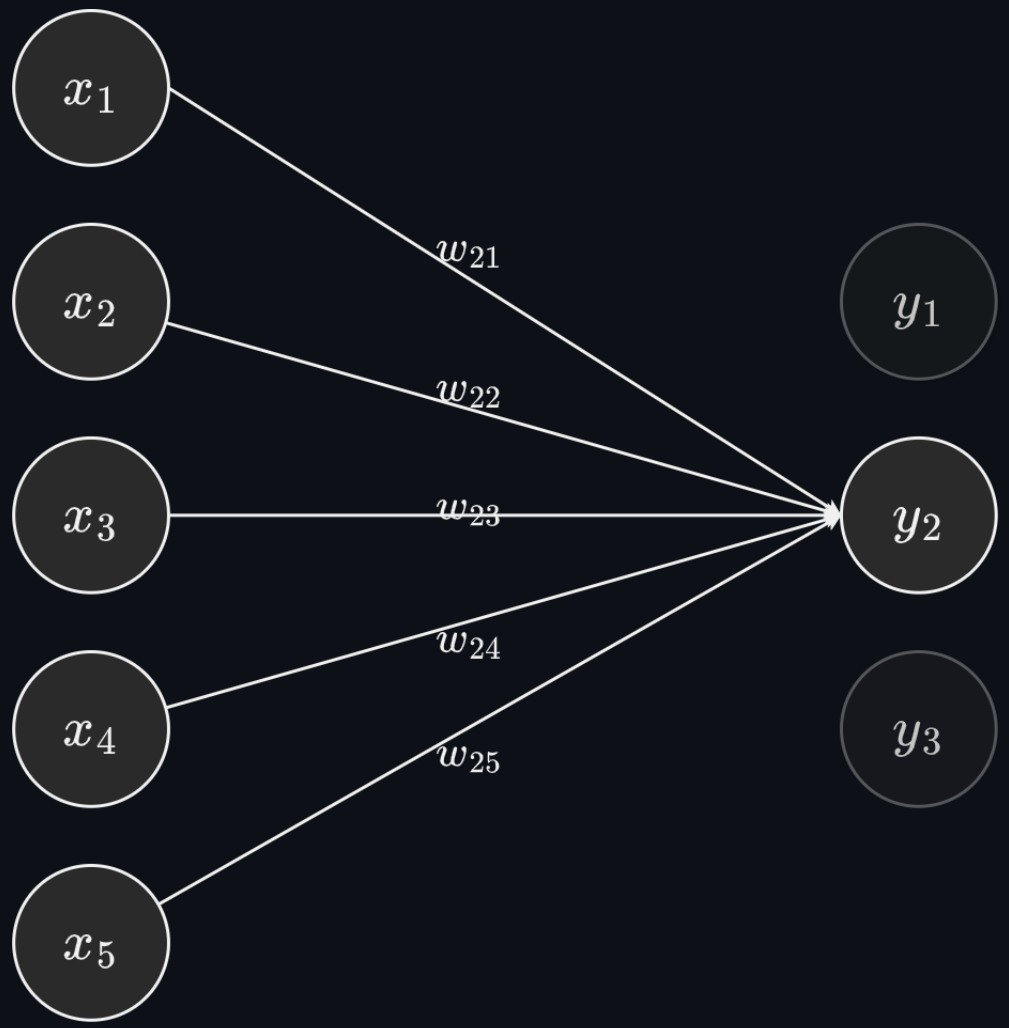
```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.input_layer = nn.Sequential(nn.Linear(5, 32), nn.ReLU())
        self.hidden_layer = nn.Sequential(nn.Linear(32, 32), nn.ReLU())
        self.output_layer = nn.Sequential(nn.Linear(32, 3), nn.Sigmoid())

    def forward(self, x):
        x = self.input_layer(x)
        x = self.hidden_layer(x)
        x = self.output_layer(x)
        return x
```

3. torch.Autograd

- El objetivo del aprendizaje automático es minimizar una función de costo $J(\boldsymbol{\theta})$.
- Al utilizar gradiente descendente, es necesario calcular $\nabla J(\boldsymbol{\theta})$.
- torch.autograd permite calcular gradientes automáticamente.

4. torch.Optim

- Una vez calculado $\nabla J(\boldsymbol{\theta})$, es necesario actualizar los parámetros del modelo.
- torch.optim implementa métodos de optimización populares como SGD y Adam.
- torch.optim.lr_scheduler ofrece varios métodos para ajustar la tasa de aprendizaje.


x_1
x_2
x_3
x_4
x_5
w_{21}
w_{22}
w_{23}
w_{24}
w_{25}
y_1
y_2
y_3

```
X = torch.randn(64, 5)  # Batch (64) x Features (5)
Y = torch.ones(64, 3)  # Batch (64) x Classes (3)

W = torch.randn(5, 3, requires_grad=True)
b = torch.randn(3, requires_grad=True)

optimizer = torch.optim.SGD([W, b], lr=0.1)
```

```
optimizer.zero_grad()

Z = torch.matmul(X, W) + b # Forward pass

loss = torch.nn.functional.binary_cross_entropy_with_logits(Z, Y)

loss.backward() # Backward pass
optimizer.step()
```


X
*
+
Z
σ
CE
Loss
W
b
Y

5. PyTorch 2.X

- PyTorch 2.0 introdujo `torch.compile` y el backend MPS para Apple M1/M2.
- PyTorch 2.1 introdujo múltiples mejoras a `torch.compile`, entre ellas, la capacidad de compilar operaciones en NumPy.
- PyTorch 2.2 expandió el soporte de `torch.compile` a optimizadores.

6. Experiencias Prácticas