Redes Neuronales (2025)

Construcción de Redes Neuronales Feedforward en PyTorch
(Trabajo en progreso)*

Índice

*Reportar errores a: tristan.osan@unc.edu.ar

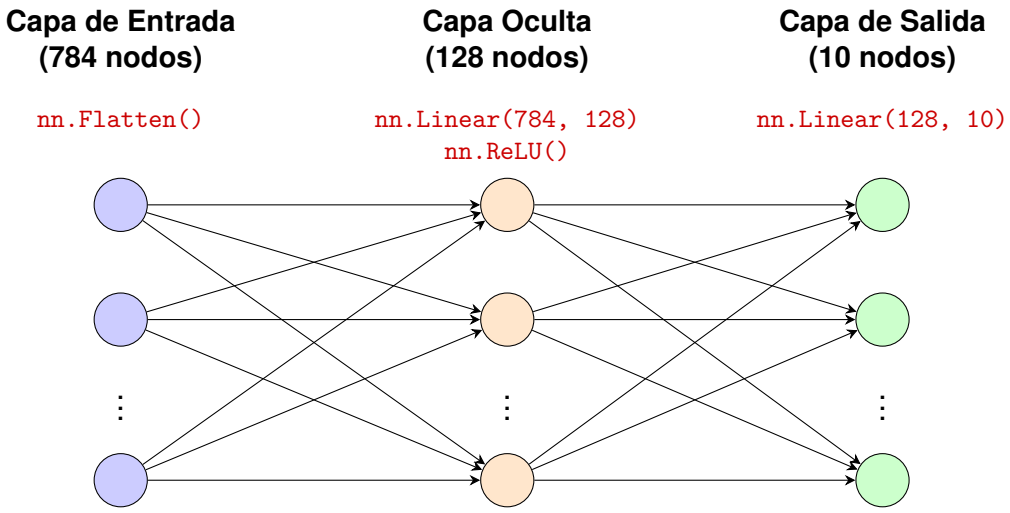
Introducción

En este documento exploraremos dos formas principales de construir redes neuronales feedforward en PyTorch. Estas redes son fundamentales para tareas de clasificación como Fashion-MNIST.

1. Método 1: Usar `nn.Sequential`

Este método es ideal para arquitecturas simples, donde las capas se aplican una tras otra sin bifurcaciones ni lógica condicional.

Ejemplo



Observaciones:

- Este diagrama representa una red neuronal feedforward construida con `nn.Sequential` en PyTorch.
- La entrada es una imagen de 28x28 píxeles, aplanada en un vector de 784 características.
- La capa oculta tiene 128 neuronas con activación ReLU.
- La salida tiene 10 neuronas, una por cada clase del conjunto Fashion-MNIST.
- Se muestran solo algunas neuronas por capa para simplificar la visualización.

Descripción: Por ejemplo, la red toma una imagen de 28x28 píxeles (784 entradas), la aplana, pasa por una capa oculta de 128 neuronas con activación ReLU, y finalmente produce 10 salidas correspondientes a las clases de Fashion-MNIST.

1.1. Bloque de código

```
import torch.nn as nn

modelo = nn.Sequential(
    nn.Flatten(),           # Convierte la imagen 28x28 en un vector de 784
       elementos
    nn.Linear(784, 128),    # Capa totalmente conectada con 128 neuronas
    nn.ReLU(),              # Función de activación ReLU
    nn.Linear(128, 10)      # Capa de salida con 10 clases
)
```

1.2. Explicación breve

Cada capa se agrega en orden. Por ejemplo, `nn.Flatten()` prepara los datos para la red, y `nn.Linear` define las conexiones entre neuronas. `nn.ReLU()` introduce no linealidad.

1.3. Ventajas

- Código conciso y legible.
- Ideal para modelos lineales.

1.4. Explicación ampliada

1.4.1. `import torch.nn as nn`

Importa el módulo `torch.nn`, que contiene las herramientas necesarias para construir redes neuronales en PyTorch.

1.4.2. `nn.Sequential(...)`

Define una red neuronal como una secuencia de capas. Cada capa se aplica en orden, una tras otra. Este método es ideal para arquitecturas simples y lineales.

1.4.3. `nn.Flatten()`

Convierte cada imagen de entrada de tamaño 28×28 en un vector de 784 elementos. Esto es necesario porque las capas lineales esperan vectores como entrada, no matrices.

$$\text{Imagen } 28 \times 28 \longrightarrow \text{Vector } 784 \text{ elementos}$$

1.4.4. `nn.Linear(784, 128)`

Crea una capa totalmente conectada (también llamada *fully connected*) con 128 neuronas. Cada una de las 784 entradas se conecta a cada una de las 128 neuronas.

$$\text{Entrada: } \mathbf{x} \in \mathbb{R}^{784} \quad \longrightarrow \quad \text{Salida: } \mathbf{h} \in \mathbb{R}^{128}$$

1.4.5. `nn.ReLU()`

Aplica la función de activación ReLU (Rectified Linear Unit), que introduce no linealidad. Esta función transforma cada valor x según:

$$\text{ReLU}(x) = \max(0, x)$$

Esto ayuda a la red a aprender relaciones complejas entre los datos.

1.4.6. `nn.Linear(128, 10)`

Crea una segunda capa totalmente conectada que transforma los 128 valores de la capa oculta en 10 salidas, una por cada clase del problema de clasificación.

$$\text{Entrada: } \mathbf{h} \in \mathbb{R}^{128} \quad \longrightarrow \quad \text{Salida: } \mathbf{y} \in \mathbb{R}^{10}$$

1.5. Resumen

Esta red neuronal realiza los siguientes pasos:

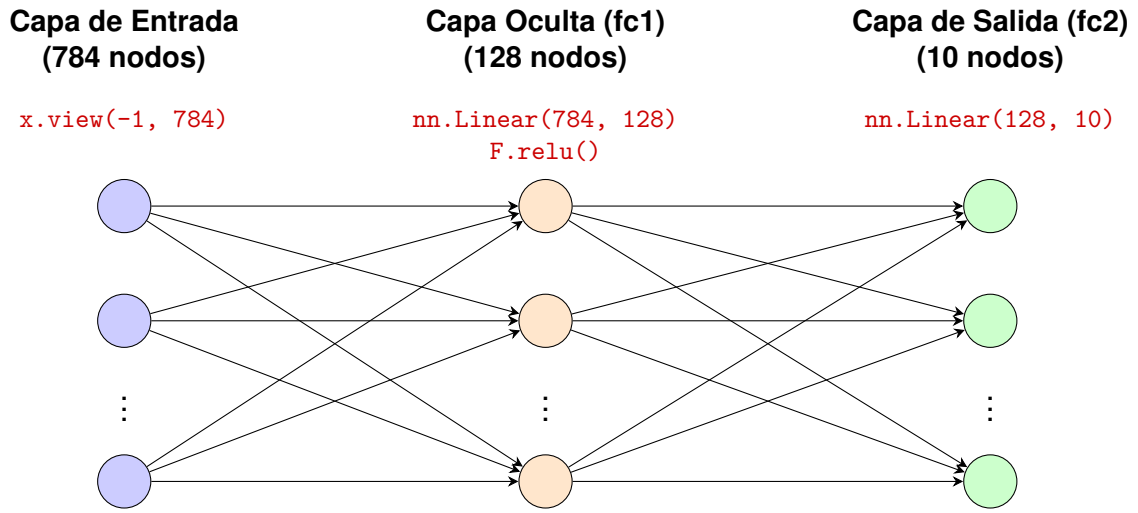
1. Aplana la imagen de entrada.
2. Aplica una capa lineal con 128 neuronas.
3. Aplica la activación ReLU.
4. Aplica una capa lineal de salida con 10 neuronas.

Es una estructura ideal para comenzar a trabajar con redes neuronales en PyTorch.

2. Método 2: Definir una clase personalizada con `nn.Module`

Este enfoque es más flexible y permite construir arquitecturas complejas, incluir lógica condicional, o reutilizar capas.

Ejemplo



Observaciones:

- Este diagrama representa una red neuronal feedforward construida con `nn.Module` en PyTorch.
- La entrada es una imagen de 28x28 píxeles, aplanada en un vector de 784 características.
- La capa oculta tiene 128 neuronas con activación ReLU.
- La salida tiene 10 neuronas, una por cada clase del conjunto Fashion-MNIST.
- Se muestran solo algunas neuronas por capa para simplificar la visualización.

Descripción: Por ejemplo, la red toma una imagen de 28x28 píxeles (784 entradas), la aplana, pasa por una capa oculta de 128 neuronas con activación ReLU, y finalmente produce 10 salidas correspondientes a las clases de Fashion-MNIST.

2.1. Bloque de código

```
import torch.nn as nn
import torch.nn.functional as F

class MiRed(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(784, 128)  # Capa oculta
        self.fc2 = nn.Linear(128, 10)   # Capa de salida

    def forward(self, x):
        x = x.view(-1, 784)             # Flatten manual
        x = F.relu(self.fc1(x))         # Activación ReLU
        x = self.fc2(x)                 # Salida sin softmax (se aplica en la
           función de pérdida)
        return x
```

2.2. Explicación breve

Aquí definimos una clase que hereda de `nn.Module`. En el método `forward`, especificamos cómo fluyen los datos. Esto permite insertar condiciones, múltiples caminos, o incluso bucles si fuera necesario.

2.3. Ventajas

- Permite mayor control sobre el flujo de datos.
- Ideal para modelos personalizados o más avanzados.

2.4. Explicación ampliada

2.4.1. `import torch.nn as nn`

Importa el módulo `torch.nn`, que contiene clases para construir redes neuronales, como `Linear`, `ReLU`, y `Module`.

2.4.2. `import torch.nn.functional as F`

Importa funciones de activación y operaciones sin parámetros, como `F.relu`, que se usan directamente en el método `forward`.

2.4.3. `class MiRed(nn.Module):`

Define una clase llamada `MiRed` que hereda de `nn.Module`, la clase base para todos los modelos en PyTorch.

2.4.4. `def __init__(self):`

Este es el constructor de la clase. Aquí se definen las capas que tendrá la red.

- `super().__init__()` llama al constructor de la clase base `nn.Module`.
- `self.fc1 = nn.Linear(784, 128)` define una capa lineal que transforma un vector de 784 entradas (una imagen de 28×28 píxeles aplanada) en 128 neuronas.
- `self.fc2 = nn.Linear(128, 10)` define una segunda capa lineal que transforma las 128 salidas anteriores en 10 salidas, una por cada clase.

2.4.5. `def forward(self, x):`

Este método define cómo fluye la información a través de la red.

- `x = x.view(-1, 784)` aplana la imagen de entrada. Si la entrada es un lote de imágenes de tamaño $[N, 1, 28, 28]$, se convierte en $[N, 784]$.
- `x = F.relu(self.fc1(x))` aplica la primera capa lineal y luego la función de activación ReLU:

$$\text{ReLU}(z) = \max(0, z)$$

  Esto introduce no linealidad en la red.

- `x = self.fc2(x)` aplica la segunda capa lineal, generando un vector de 10 elementos (logits), uno por clase.
- `return x` devuelve la salida sin aplicar `softmax`, ya que esta se incluye dentro de la función de pérdida como `CrossEntropyLoss`.

2.5. Resumen

Esta red neuronal realiza los siguientes pasos:

1. Aplana la imagen de entrada.
2. Aplica una transformación lineal con 128 neuronas.
3. Aplica la función de activación ReLU.
4. Aplica una segunda transformación lineal con 10 salidas.

Este enfoque permite modificar fácilmente la arquitectura, insertar condiciones, o reutilizar capas, lo que lo hace ideal para modelos más complejos.

3. Comparación

Característica	`nn.Sequential`	`nn.Module`
Simplicidad	Alta	Media
Flexibilidad	Baja	Alta
Ideal para principiantes	Sí	Sí, con guía
Permite lógica condicional	No	Sí

3.1. Comentarios

Ambos métodos son válidos y útiles. Para empezar, `nn.Sequential` es excelente. A medida que se avanza, usar clases personalizadas con `nn.Module` permite construir modelos más sofisticados.