Redes Neuronales (2025)

Ayudas Guía N°11: Aprendiendo Fashion-MNIST con PyTorch
(Trabajo en progreso)*

Índice

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

Objetivos

- Descargar y transformar (normalizar) los conjuntos de entrenamiento y validación de Fashion-MNIST.
- Explorar ejemplos y describir el formato de los datos.
- Crear un diccionario Python que asocie cada etiqueta numérica con su nombre legible.
- Graficar un mosaico de 3x3 imágenes con sus clasificaciones.

1. Ejercicio 1): Importando librerías

Asegúrese de tener Python 3 y las librerías necesarias instaladas. En muchos entornos (por ejemplo, Anaconda o un entorno virtual) ejecute:

```
pip install torch torchvision matplotlib
```

En Google Colab es suficiente ejecutar las siguientes líneas de código:

```
# Ejercicio 2: Descargar y explorar Fashion-MNIST (PyTorch)
import torch
from torchvision import datasets, transforms
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt
import numpy as np
import random
```

1.1. Explicación del bloque de código

Este bloque importa las bibliotecas necesarias para trabajar con redes neuronales, procesamiento de imágenes y visualización en Python. A continuación se explica cada línea en detalle.

1.1.1. `import torch`

- Importa la biblioteca **PyTorch**, que es una herramienta para construir y entrenar redes neuronales.
- Proporciona estructuras como `Tensor`, funciones matemáticas, y módulos para definir modelos.
- Es el núcleo del trabajo con aprendizaje profundo en este tutorial.

1.1.2. `from torchvision import datasets, transforms`

- `torchvision` es una biblioteca complementaria a PyTorch especializada en visión por computadora.
- `datasets`: contiene conjuntos de datos populares como MNIST, CIFAR, Fashion-MNIST, etc.
- `transforms`: permite aplicar transformaciones a las imágenes (por ejemplo, convertir a tensor, normalizar, rotar).
- Se usa para preparar los datos antes de entrenar el modelo.

1.1.3. `from torch.utils.data import DataLoader`

- `DataLoader` es una herramienta que divide el conjunto de datos en **lotes** (batches).
- Permite recorrer el dataset en bloques de ejemplos, lo cual es más eficiente para entrenar redes.
- También permite mezclar el orden de los datos y paralelizar la carga.

1.1.4. `import matplotlib.pyplot as plt`

- Importa la biblioteca **matplotlib**, usada para crear gráficos y visualizar datos.
- `pyplot` es el módulo que permite dibujar figuras, imágenes, curvas, etc.
- En este tutorial se usa para mostrar imágenes del dataset.

1.1.5. `import numpy as np`

- **NumPy** es una biblioteca para cálculos numéricos con arreglos multidimensionales.
- Se usa para convertir tensores a arreglos, manipular datos y realizar operaciones matemáticas.
- `np` es el alias comúnmente usado para acceder a sus funciones.

1.1.6. `import random`

- Importa el módulo estándar de Python para generar números aleatorios.
- Se usa para seleccionar ejemplos aleatorios del dataset.
- Permite reproducibilidad si se fija una semilla con `random.seed(...)`.

1.2. Comentarios

Este bloque de código prepara el entorno para trabajar con datos de imágenes, construir modelos de redes neuronales, y visualizar resultados. Cada biblioteca cumple un rol específico y es fundamental para el flujo de trabajo en aprendizaje profundo con PyTorch.

2. Ejercicio 2): Descargar y explorar Fashion-MNIST con PyTorch

```
# Ejercicio 2: Descargar y explorar Fashion-MNIST (PyTorch)
import torch
from torchvision import datasets, transforms
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt
import numpy as np
import random

# 1) Transformaciones: convertir a tensor y normalizar
# Fashion-MNIST tiene pixeles en [0,255], torchvision los entrega como [0,1]
   tras ToTensor.
# Normalizaremos con media 0.5 y desviación 0.5 para escalar a [-1,1] (práctica
    común para redes simples).
transform = transforms.Compose([
    transforms.ToTensor(),                    # convierte HxW (0..255) ->
       tensor CxHxW con valores en [0,1]
    transforms.Normalize((0.5,), (0.5,))      # normaliza cada canal: (x-0.5)
       /0.5 => [-1,1] aproximadamente
])

# Descargar datasets (si ya existen, torchvision no los baja de nuevo)
train_dataset = datasets.FashionMNIST(root='./data', train=True, download=True,
    transform=transform)
test_dataset  = datasets.FashionMNIST(root='./data', train=False, download=True
   , transform=transform)

# DataLoaders (opcional para entrenamiento; aquí solo para ejemplo)
```

```
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
test_loader  = DataLoader(test_dataset, batch_size=64, shuffle=False)
```

2.1. Explicación del bloque de código

2.1.1. Transformaciones

Antes de alimentar imágenes a una red neuronal, es necesario convertirlas a un formato que PyTorch pueda procesar: tensores normalizados.

a) `transforms.ToTensor()`

- Convierte una imagen PIL (formato tradicional en Python) o NumPy array en un tensor de PyTorch.
- Reescala los valores de los píxeles de `[0,255]` a `[0,1]` dividiendo por 255.
- Cambia el formato de `(alto, ancho)` a `(canal, alto, ancho)`.

b) `transforms.Normalize((0.5,), (0.5,))`

- Aplica la fórmula: `(x - media) / desviación`.
- En este caso: `(x - 0.5) / 0.5`, lo que transforma el rango `[0,1]` a aproximadamente `[-1,1]`.
- Esta normalización ayuda a que la red neuronal aprenda más rápido y de forma más estable.

c) `transforms.Compose([...])`

- Permite encadenar varias transformaciones en orden.
- Aquí primero se convierte a tensor y luego se normaliza.

2.1.2. Descarga del dataset Fashion-MNIST

- `datasets.FashionMNIST(...)` descarga el conjunto de datos si no está presente en el directorio `./data`.
- `train=True` indica que se quiere utilizar el conjunto de entrenamiento.
- `train=False` indica que se quiere utilizar el conjunto de validación.
- `download=True` permite que se descargue automáticamente si no existe.
- `transform=transform` aplica las transformaciones definidas anteriormente a cada imagen.

2.1.3. Creación de los DataLoaders

- `DataLoader` divide el dataset en lotes (`batches`) para procesarlos en bloques.
- `batch_size=64` indica que cada lote tendrá 64 imágenes.
- `shuffle=True` baraja los datos en cada epoch (solo en entrenamiento).
- `shuffle=False` mantiene el orden original (usado en testeo).

2.1.4. Resumen

Este bloque de código prepara los datos para ser utilizados en una red neuronal:

- Convierte las imágenes a tensores normalizados.
- Descarga el dataset Fashion-MNIST.
- Crea objetos `DataLoader` para recorrer los datos en lotes.

```
# 2) Exploración de ejemplos: formato y dimensiones
# Tomamos un ejemplo del dataset de entrenamiento
img_tensor, label = train_dataset[0]   # img_tensor: torch.Tensor de forma
    (1,28,28), label: int
print("Tipo de img_tensor:", type(img_tensor))
print("Shape (C,H,W):", img_tensor.shape)   # should be (1,28,28)
print("Rango de valores (tensor normalizado):", img_tensor.min().item(),
    img_tensor.max().item())
print("Etiqueta (numérica):", label)

# Para visualizar con matplotlib necesitamos convertir el tensor a numpy y "
    desnormalizar"
def tensor_to_image(tensor):
    # tensor tiene shape (1,28,28) y está normalizado en [-1,1]
    tensor = tensor.clone()                    # evitar modificar original
    tensor = tensor * 0.5 + 0.5                # desnormalizar a [0,1]
    array = tensor.numpy()                     # shape (1,28,28)
    array = np.squeeze(array)                  # shape (28,28)
    return array

sample_img = tensor_to_image(img_tensor)
plt.figure(figsize=(3,3))
plt.imshow(sample_img, cmap='gray')
plt.title(f"Etiqueta numérica: {label}")
plt.axis('off')
plt.show()
```

2.2. Explicación del bloque de código

2.2.1. Selección de un ejemplo del dataset

- `train_dataset[0]` accede al primer ejemplo del conjunto de entrenamiento.
- Devuelve un par: la imagen como tensor (`img_tensor`) y su etiqueta (`label`).
- `img_tensor` tiene forma `(1,28,28)`: un canal (escala de grises), 28 píxeles de alto y 28 de ancho.
- `label` es un número entero entre 0 y 9 que representa la clase de la imagen.

2.2.2. Inspección del tensor

- `type(img_tensor)` muestra que es un objeto `torch.Tensor`.
- `img_tensor.shape` confirma las dimensiones: canal, alto, ancho.
- `img_tensor.min().item()` y `img_tensor.max().item()` muestran el rango de valores.
- Como el tensor fue normalizado, sus valores están aproximadamente en `[-1,1]`.

2.2.3. Conversión para visualización

- `matplotlib` espera imágenes como arreglos NumPy en el rango `[0,1]`.
- La función `tensor_to_image` convierte el tensor normalizado a un arreglo visualizable.
- `tensor.clone()` evita modificar el tensor original.
- `tensor * 0.5 + 0.5` desnormaliza los valores a `[0,1]`.
- `np.squeeze(array)` elimina la dimensión del canal, dejando una imagen `(28,28)`.

2.2.4. Visualización con matplotlib

- `plt.figure(figsize=(3,3))` crea una figura pequeña.
- `plt.imshow(..., cmap='gray')` muestra la imagen en escala de grises.
- `plt.title(...)` agrega la etiqueta como título.
- `plt.axis('off')` oculta los ejes para una visualización más limpia.
- `plt.show()` muestra la imagen en pantalla.

2.2.5. Resultados esperados

```
Tipo de img_tensor: <class 'torch.Tensor'>
Shape (C,H,W): torch.Size([1, 28, 28])
Rango de valores (tensor normalizado): -1.0 1.0
Etiqueta (numérica): 9
```

Figura 1: **Ejemplo individual de Fashion-MNIST.** Imagen de un único ejemplo extraído del conjunto de entrenamiento, desnormalizada para visualización. La imagen tiene formato 28×28 píxeles y un solo canal (escala de grises).

2.2.6. Resumen

Este bloque permite:

- Acceder a un ejemplo del dataset.
- Inspeccionar su formato y valores.
- Convertirlo para visualizarlo correctamente.

- Confirmar que las transformaciones aplicadas al dataset funcionan como se espera.

```
# 3) Crear diccionario de etiquetas (consultando la página de Fashion-MNIST)
# Clases oficiales de Fashion-MNIST:
class_names = {
    0: 'T-shirt/top',
    1: 'Trouser',
    2: 'Pullover',
    3: 'Dress',
    4: 'Coat',
    5: 'Sandal',
    6: 'Shirt',
    7: 'Sneaker',
    8: 'Bag',
    9: 'Ankle boot'
}
print("Diccionario de clases:", class_names)
```

2.3. Explicación del bloque de código

2.3.1. ¿Qué es Fashion-MNIST?

Fashion-MNIST es un conjunto de datos de imágenes en escala de grises de ropa y accesorios. Cada imagen tiene una etiqueta numérica entre 0 y 9 que representa una categoría de prenda.

2.3.2. ¿Por qué crear un diccionario de etiquetas?

- Las etiquetas en el dataset son números enteros.
- Para que los resultados sean legibles por humanos, se necesita traducir esos números a nombres de clase.
- El diccionario `class_names` permite hacer esa traducción fácilmente.

2.3.3. ¿Qué hace el diccionario `class_names`?

- Es un objeto de tipo `dict` en Python.
- Asocia cada número de etiqueta con su nombre correspondiente.
- Por ejemplo: `class_names[0]` devuelve `'T-shirt/top'`.

2.3.4. ¿Cómo se usa este diccionario?

- Para mostrar el nombre de la clase en gráficos, títulos o reportes.
- Para interpretar los resultados de clasificación.
- Para verificar visualmente si las imágenes están correctamente etiquetadas.

2.3.5. Ejemplo de uso

Supongamos que tenemos una imagen con etiqueta 7:

```
label = 7
print("Clase:", class_names[label])  # Salida: Sneaker
```

2.3.6. Visualización completa

El comando:

```
print("Diccionario de clases:", class_names)
```

muestra todo el diccionario en pantalla, lo cual es útil para confirmar que las clases están correctamente definidas.

Resultado esperado

```
Diccionario de clases: 0: 'T-shirt/top', 1: 'Trouser', 2: 'Pullover', 3: 'Dress',
4: 'Coat', 5: 'Sandal', 6: 'Shirt', 7: 'Sneaker', 8: 'Bag', 9: 'Ankle boot'
```

2.3.7. Comentarios

Crear un diccionario de etiquetas es una práctica esencial en clasificación de imágenes. Permite traducir resultados numéricos a nombres legibles, facilitando la interpretación y visualización de los datos.

```
# 4) Graficar un mosaico 3x3 de imágenes aleatorias con sus nombres
def plot_grid(dataset, class_dict, rows=3, cols=3, seed=123):
    random.seed(seed)
    indices = random.sample(range(len(dataset)), rows*cols)
    fig, axes = plt.subplots(rows, cols, figsize=(cols*2.5, rows*2.5))
    for ax, idx in zip(axes.flatten(), indices):
        img_t, lbl = dataset[idx]
        img = tensor_to_image(img_t)
        ax.imshow(img, cmap='gray')
        ax.set_title(class_dict[int(lbl)], fontsize=9)
        ax.axis('off')
    plt.tight_layout()
    plt.show()

plot_grid(train_dataset, class_names, rows=3, cols=3)
```

2.4. Explicación del bloque de código

2.4.1. Objetivo de la función

La función `plot_grid` permite visualizar un conjunto de imágenes del dataset en forma de mosaico. Cada imagen se muestra con su nombre de clase correspondiente, lo que facilita la inspección visual del conjunto de datos.

2.4.2. Parámetros de entrada

- `dataset`: conjunto de datos (por ejemplo, `train_dataset`).
- `class_dict`: diccionario que traduce etiquetas numéricas a nombres legibles.
- `rows`, `cols`: número de filas y columnas del mosaico.
- `seed`: semilla para generar aleatoriedad reproducible.

2.4.3. Explicación paso a paso

a) Fijar la semilla de aleatoriedad

```
random.seed(seed)
```

Esto asegura que los resultados sean siempre iguales si se usa la misma semilla. Es útil para reproducibilidad.

b) Seleccionar índices aleatorios

```
indices = random.sample(range(len(dataset)), rows*cols)
```

- `range(len(dataset))`: genera una lista de todos los índices posibles del dataset.
- `random.sample(...)`: selecciona aleatoriamente `rows*cols` índices sin repetir.

c) Crear la figura y los ejes

```
fig, axes = plt.subplots(rows, cols, figsize=(cols*2.5, rows*2.5))
```

- Crea una figura con una cuadrícula de subplots.
- `figsize`: controla el tamaño total de la figura.
- `axes`: es una matriz de objetos `Axes` donde se dibujan las imágenes.

d) Iterar sobre los ejes y los índices

```
for ax, idx in zip(axes.flatten(), indices):
```

- `axes.flatten()`: convierte la matriz de ejes en una lista.
- `zip(...)`: empareja cada eje con un índice aleatorio.

e) Extraer imagen y etiqueta

```
img_t, lbl = dataset[idx]
```

- Obtiene la imagen tensorial y su etiqueta desde el dataset.

f) Convertir imagen para visualización

```
img = tensor_to_image(img_t)
```

- Convierte el tensor normalizado a un arreglo NumPy en escala de grises.

g) Mostrar imagen en el eje

```
ax.imshow(img, cmap='gray')
```

- Dibuja la imagen en el eje correspondiente usando una paleta de grises.

h) Agregar título con nombre de clase

```
ax.set_title(class_dict[int(lbl)], fontsize=9)
```

- Traduce la etiqueta numérica a nombre legible usando el diccionario.

i) Ocultar ejes

```
ax.axis('off')
```

- Elimina los bordes, marcas y etiquetas del eje para que se vea solo la imagen.

j) Ajustar diseño y mostrar figura

```
plt.tight_layout()
plt.show()
```

- `tight_layout()`: ajusta los márgenes para evitar superposición.
- `show()`: muestra la figura en pantalla.

2.4.4. Resultados esperados

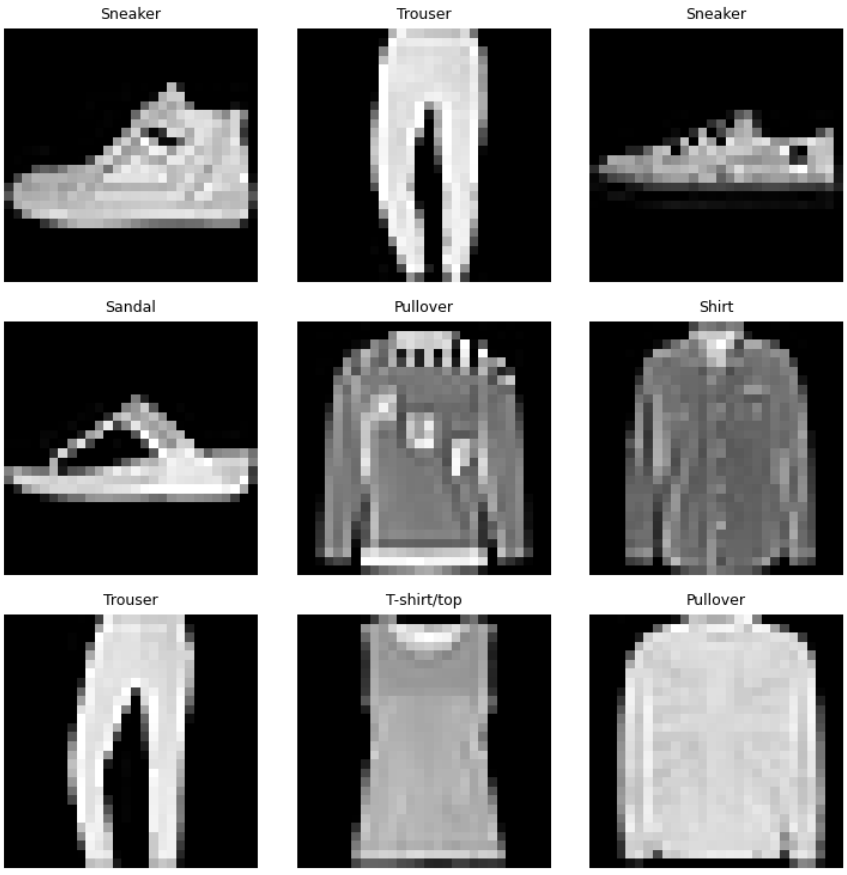

Figura 2: **Mosaico 3x3 de Fashion-MNIST.** Nueve imágenes aleatorias del conjunto de entrenamiento, cada una titulada con su nombre de clase legible, proveniente del diccionario de clases. El mosaico permite observar la variabilidad intra-clase y diferencias visuales entre categorías (por ejemplo, *Sneaker* vs *Ankle boot*).

3. Ejercicio 3: Creando un `DataLoader` para alimentar el modelo con lotes

Objetivos

- Crear los `DataLoader`s para los conjuntos de entrenamiento y testeo (validación).
- Explorar un batch de datos para entender su estructura.

En este ejercicio aprenderemos a utilizar el objeto `DataLoader` de PyTorch para preparar los datos del conjunto Fashion-MNIST en lotes de tamaño fijo. Esto es fundamental para entrenar redes neuronales de manera eficiente.

3.1. Creación de los `DataLoader`s

El objeto `DataLoader` permite dividir el dataset en lotes (*batches*) y mezclar el orden de los datos si se desea. Esto es útil para entrenar modelos en bloques de ejemplos, lo cual mejora la eficiencia computacional y la estabilidad del entrenamiento.

```
from torch.utils.data import DataLoader
from torchvision import datasets, transforms

# Transformación: convertir a tensor y normalizar
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])

# Descargar datasets
train_dataset = datasets.FashionMNIST(root='./data', train=True, download=True,
    transform=transform)
test_dataset  = datasets.FashionMNIST(root='./data', train=False, download=True
   , transform=transform)

# Crear DataLoaders con batch_size=100 y shuffle=True
train_loader = DataLoader(train_dataset, batch_size=100, shuffle=True)
test_loader  = DataLoader(test_dataset, batch_size=100, shuffle=True)
```

Explicación

- `batch_size=100`: cada lote contiene 100 imágenes.
- `shuffle=True`: se mezcla el orden de los datos antes de cada época (epoch), lo cual mejora la generalización del modelo.
- `train_loader` y `test_loader` son iterables que devuelven pares `(images, labels)`.

3.2. Exploración de un batch

Vamos a inspeccionar un lote de datos para entender su estructura. Usamos la función `next(iter(...))` para obtener el primer batch.

```
# Obtener un batch del conjunto de entrenamiento
images, labels = next(iter(train_loader))

# Mostrar dimensiones
print("Shape de images:", images.shape)   # (100, 1, 28, 28)
print("Shape de labels:", labels.shape)   # (100,)
```

3.2.1. Resultados esperados

```
Shape de images: torch.Size([100, 1, 28, 28])
Shape de labels: torch.Size([100])
```

Línea de código a explicar

```
images, labels = next(iter(train_loader))
```

¿Qué es `train_loader`?

- Es un objeto de tipo `DataLoader` de PyTorch.
- Su función es entregar los datos del conjunto de entrenamiento en **lotes** (batches).
- Cada vez que se recorre, devuelve un par de tensores:
  - `images`: lote de imágenes (por ejemplo, 100 si usamos `batch_size=100`).
  - `labels`: lote de etiquetas correspondientes a esas imágenes.

¿Qué hace `iter(train_loader)`?

- Convierte el objeto `train_loader` en un **iterador**.
- Esto permite acceder a los lotes uno por uno usando la función `next(...)`.

¿Qué hace `next(...)`?

- Devuelve el **primer lote** del iterador.
- Es equivalente a decir: “Dame el primer bloque de datos del conjunto de entrenamiento”.

¿Qué hace la asignación `images, labels = ...`?

- El lote devuelto por `next(...)` es un par de tensores.
- Esta línea separa ese par en dos variables:
  - `images`: tensor de forma `(100, 1, 28, 28)`.
  - `labels`: tensor de forma `(100,)`.

Interpretación de las dimensiones

- `images` tiene forma `(100, 1, 28, 28)`:
  - 100: número de ejemplos en el lote.
  - 1: número de canales (escala de grises).
  - 28x28: tamaño de cada imagen en píxeles.
- `labels` tiene forma `(100,)`:
  - Cada entrada `labels[i]` es un número entero en el conjunto $\{0, 1, \ldots, 9\}$.
  - Este número indica la clase a la que pertenece la imagen `images[i]`.

¿Por qué es útil esta línea?

- Permite acceder rápidamente a un lote de datos sin recorrer todo el conjunto.
- Es útil para:
  - Verificar que los datos están bien cargados.
  - Visualizar ejemplos.
  - Probar el modelo con un lote de entrada.

Ejemplo ilustrativo

Supongamos que `labels[0] = 7`. Entonces:

- `images[0]` es una imagen de un **Sneaker** (zapatilla deportiva).
- `labels[0]` es el número 7, que representa esa clase.

Comentarios

Esta línea de código es una forma compacta y eficiente de obtener un lote de datos desde el `DataLoader`. Comprender su funcionamiento es clave para trabajar con datos en PyTorch y alimentar modelos de manera correcta.

3.3. Visualización de algunas imágenes del batch

Podemos graficar algunas imágenes del batch para verificar que están correctamente cargadas y etiquetadas.

```
# Función para des-normalizar y convertir a imagen
def tensor_to_image(tensor):
    tensor = tensor * 0.5 + 0.5
    array = tensor.numpy()
    array = np.squeeze(array)
    return array

# Graficar las primeras 9 imágenes del batch
fig, axes = plt.subplots(3, 3, figsize=(8,8))
for i, ax in enumerate(axes.flatten()):
    img = tensor_to_image(images[i])
    label = labels[i].item()
    ax.imshow(img, cmap='gray')
    ax.set_title(class_names[label])
    ax.axis('off')
plt.tight_layout()
plt.show()
```

3.4. Explicación del bloque de código

Objetivo del bloque

Este código permite visualizar las primeras 9 imágenes de un lote (`batch`) de datos obtenidos con un `DataLoader`. Cada imagen se muestra con su etiqueta correspondiente en un mosaico de 3 filas por 3 columnas.

3.4.1. Función `tensor_to_image`

Esta función convierte un tensor de PyTorch en una imagen NumPy lista para ser graficada.

Explicación paso a paso

- `tensor = tensor * 0.5 + 0.5`:
  - Deshace la normalización aplicada previamente: si los valores estaban en $[-1, 1]$, ahora vuelven a estar en $[0, 1]$.
- `array = tensor.numpy()`:
  - Convierte el tensor a un arreglo NumPy, que es el formato que `matplotlib` puede graficar.
- `array = np.squeeze(array)`:
  - Elimina la dimensión del canal (de forma `(1,28,28)` a `(28,28)`), ya que es una imagen en escala de grises.
- `return array`:
  - Devuelve la imagen lista para ser graficada.

3.4.2. Crear figura y subplots

```
fig, axes = plt.subplots(3, 3, figsize=(8,8))
```

- Crea una figura con una cuadrícula de 3 filas y 3 columnas.
- `figsize=(8,8)` define el tamaño total de la figura en pulgadas.
- `axes` es una matriz de objetos donde se dibujarán las imágenes.

3.4.3. Iterar sobre las imágenes del batch

```
for i, ax in enumerate(axes.flatten()):
```

- `axes.flatten()` convierte la matriz de ejes en una lista lineal.
- `enumerate(...)` permite recorrer los primeros 9 elementos y tener acceso al índice `i` y al eje `ax`.

3.4.4. Procesar y graficar cada imagen

- `img = tensor_to_image(images[i])`:
  - Convierte el tensor normalizado a una imagen NumPy en escala de grises.
- `label = labels[i].item()`:
  - Extrae el valor numérico de la etiqueta (por ejemplo, 0 a 9).
- `ax.imshow(img, cmap='gray')`:
  - Muestra la imagen en el eje correspondiente usando una paleta de grises.
- `ax.set_title(class_names[label])`:
  - Agrega el nombre de la clase como título del subplot.
- `ax.axis('off')`:
  - Oculta los ejes para una visualización más limpia.

3.4.5. Ajustar y mostrar la figura

```
plt.tight_layout()
plt.show()
```

- `tight_layout()` ajusta los márgenes para evitar que los títulos se superpongan.
- `show()` muestra la figura en pantalla.

3.4.6. Comentarios

Este bloque de código permite visualizar de forma clara y ordenada un conjunto de imágenes del dataset, junto con sus etiquetas. Es una herramienta fundamental para verificar que los datos están bien cargados y etiquetados antes de entrenar un model

3.4.7. Resultados esperados

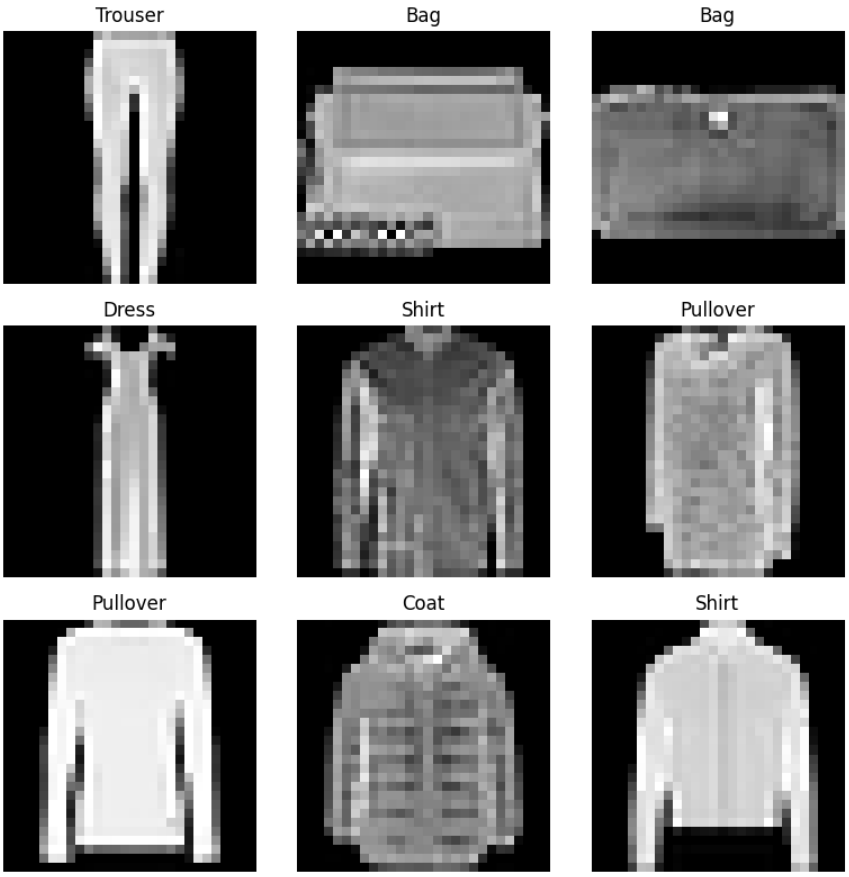

Figura 3: **Exploración visual de un batch del DataLoader.** Se muestran 9 imágenes extraídas del primer lote (`batch`) del conjunto de entrenamiento. Cada imagen está desnormalizada para su visualización y titulada con su clase correspondiente, según el diccionario de etiquetas de Fashion-MNIST.

La Figura 3 permite observar cómo se estructuran los datos dentro de un lote entregado por el `DataLoader`. Cada lote contiene 100 ejemplos, pero aquí se grafican solo 9 para facilitar la inspección. Las imágenes tienen formato `(1,28,28)` y fueron desnormalizadas para recuperar su escala original de grises. Las etiquetas numéricas fueron traducidas a nombres legibles usando un diccionario de clases.

4. Ejercicio 4: Definición de una red neuronal feedforward

Objetivo

Definir una red neuronal artificial de tipo **feedforward** con 4 capas totalmente conectadas. Esta red será utilizada para clasificar imágenes del conjunto Fashion-MNIST.

4.1. Arquitectura de la red

La red tendrá la siguiente estructura:

- **Capa de entrada**: recibe imágenes de tamaño 28×28 píxeles, lo que equivale a 784 neuronas (ya que $28 \times 28 = 784$).
- **Primera capa oculta**: $n_1 = 128$ neuronas, activación ReLU, dropout con $p = 0{,}2$.
- **Segunda capa oculta**: $n_2 = 64$ neuronas, activación ReLU, dropout con $p = 0{,}2$.
- **Capa de salida**: 10 neuronas (una por cada clase de Fashion-MNIST), sin activación ni dropout.

4.2. Código en PyTorch

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

# Definición de la red neuronal
class FeedforwardNN(nn.Module):
    def __init__(self):
        super(FeedforwardNN, self).__init__()

        # Capa de entrada: 784 -> 128
        self.fc1 = nn.Linear(784, 128)
        self.dropout1 = nn.Dropout(p=0.2)

        # Capa oculta: 128 -> 64
        self.fc2 = nn.Linear(128, 64)
        self.dropout2 = nn.Dropout(p=0.2)

        # Capa de salida: 64 -> 10
        self.fc3 = nn.Linear(64, 10)

    def forward(self, x):
        # Aplanar imagen: (batch_size, 1, 28, 28) -> (batch_size, 784)
        x = x.view(x.size(0), -1)

        # Primera capa oculta
        x = F.relu(self.fc1(x))
        x = self.dropout1(x)

        # Segunda capa oculta
        x = F.relu(self.fc2(x))
        x = self.dropout2(x)

        # Capa de salida (sin activación)
        x = self.fc3(x)
        return x
```

4.3. Explicación

`nn.Linear(in_features, out_features)`

Define una capa totalmente conectada. Cada neurona de la capa anterior se conecta con todas las neuronas de la siguiente.

`F.relu(...)`

Aplica la función de activación ReLU (Rectified Linear Unit), que convierte valores negativos en cero y deja pasar los positivos. Es útil para introducir no linealidad.

`nn.Dropout(p=0.2)`

Durante el entrenamiento, `Dropout` apaga aleatoriamente el 20 % de las neuronas para evitar que la red se sobreajuste (*"overfitting"*). No se aplica durante la inferencia.

`x.view(x.size(0), -1)`

Convierte cada imagen de forma `(1,28,28)` en un vector plano de 784 elementos. Esto es necesario para alimentar la capa lineal de entrada.

Capa de salida

La última capa tiene 10 neuronas, una por cada clase. No se aplica activación aquí porque la función de pérdida `CrossEntropyLoss` en PyTorch ya incluye `softmax` internamente.

Comentarios

Hemos definido una red neuronal simple pero funcional para clasificación de imágenes. Esta arquitectura es adecuada para comenzar a entrenar modelos en Fashion-MNIST y entender cómo se conectan las capas en PyTorch.

5. Logits y Softmax en Clasificación con PyTorch

5.1. ¿Qué son los logits?

En una red neuronal de clasificación, la última capa suele ser una capa lineal que produce un vector de salida con tantos elementos como clases. Ese vector se llama **logits**.

- Los logits son números reales sin restricciones.
- Representan la "evidencia" que la red tiene para cada clase.
- No son probabilidades.

Ejemplo de logits para 5 clases:

$$\text{logits} = [2{,}1,\ -0{,}3,\ 0{,}8,\ 4{,}2,\ 1{,}5]$$

5.2. ¿Por qué no podemos usar los logits directamente?

- Pueden ser negativos o mayores que 1.
- No suman 1.
- No se pueden interpretar como probabilidades.

5.3. ¿Qué hace la función softmax?

La función `softmax` transforma los logits en probabilidades. La fórmula es:

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

Donde:

- z_i: logit de la clase i
- K: número total de clases
- e^{z_i}: exponencial del logit (siempre positivo)
- El denominador asegura que la suma total sea 1

5.4. Ejemplo numérico

Supongamos que tenemos 3 clases y la red produce:

$$\text{logits} = [2{,}0,\ 1{,}0,\ 0{,}1]$$

Aplicamos softmax:

$$e^{2{,}0} = 7{,}39 \tag{1}$$
$$e^{1{,}0} = 2{,}72 \tag{2}$$
$$e^{0{,}1} = 1{,}10 \tag{3}$$
$$\text{suma total} = 7{,}39 + 2{,}72 + 1{,}10 = 11{,}21 \tag{4}$$

Entonces:

$$\text{softmax} = \left[\frac{7{,}39}{11{,}21},\ \frac{2{,}72}{11{,}21},\ \frac{1{,}10}{11{,}21}\right] \approx [0{,}66,\ 0{,}24,\ 0{,}10]$$

La clase 0 tiene la mayor probabilidad: 66 %.

5.5. Uso en PyTorch

Durante inferencia (predicción)

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

logits = torch.tensor([2.0, 1.0, 0.1])
probs = F.softmax(logits, dim=0)
print(probs)  # tensor([0.6590, 0.2424, 0.0986])
```

Durante entrenamiento

- No aplicamos *softmax* explícitamente.
- Usamos `nn.CrossEntropyLoss`, que:
  - Toma los *logits* directamente.
  - Aplica *softmax* internamente.
  - Calcula la *pérdida* usando la probabilidad de la clase correcta.

5.6. Predicción final

Una vez que tenemos las probabilidades, usamos:

```
predicted_class = torch.argmax(probs).item()
```

Esto devuelve el índice de la clase con mayor probabilidad.

5.7. Resumen visual

Concepto	Qué es	Forma	Uso
Logits	Salida cruda de la red	Tensor de reales	Entrada para softmax o CrossEntropy
Softmax	Función de activación	Probabilidades en [0,1] que suman 1	Para interpretar o visualizar
Predicción	Clase más probable	Índice entero	`argmax(softmax(logits))`

5.8. Comentarios

- Los *logits* son la salida directa de la red antes de aplicar cualquier activación.
- La función `softmax` convierte esos logits en probabilidades interpretables.
- En PyTorch, usamos `softmax` para visualizar y `CrossEntropyLoss` para entrenar.
- Comprender esta transformación es clave para interpretar y evaluar modelos de clasificación.

6. One-hot y Cálculo de la Entropía Cruzada (*Cross Entropy*)

6.1. Concepto: *one-hot* (codificación indicadora)

En clasificación supervisada, las etiquetas de las muestras suelen representarse de manera discreta: por ejemplo, para C clases las etiquetas reales son enteros $\{0, 1, \ldots, C-1\}$. **One-hot** es una representación vectorial de la etiqueta en la que la clase verdadera aparece con valor 1 y las demás con 0.

Si $C = 4$ y la etiqueta verdadera es la clase $k = 2$, la codificación one-hot es:

$$y = [0,\ 0,\ 1,\ 0]^\top.$$

Ventajas:

- Facilita cálculos vectoriales en la función de pérdida.
- Es la representación convencional cuando usamos softmax y cross-entropy.

En la práctica (p. ej. PyTorch) muchas funciones aceptan la etiqueta como entero k y realizan internamente la comparación con el vector one-hot; no siempre es necesario construir explícitamente el vector one-hot.

6.2. Recordatorio: logits y softmax

Sea un modelo que por entrada x produce un vector de *logits* $z \in \mathbb{R}^C$. Los logits son salidas no normalizadas; para convertirlos en probabilidades se aplica softmax:

$$p_i = \text{softmax}(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{C} e^{z_j}}, \qquad i = 1, \ldots, C,$$

con $p_i \in (0, 1)$ y $\sum_i p_i = 1$.

6.3. Definición de la Cross Entropy (una muestra)

Si la etiqueta verdadera (one-hot) es y y las probabilidades predichas son $p = \text{softmax}(z)$, la pérdida por una muestra es:

$$J = -\sum_{i=1}^{C} y_i \log p_i.$$

Si la clase verdadera es k (es decir $y_k = 1$), esto se reduce a:

$$J = -\log p_k.$$

6.4. Ejemplo numérico completo (mini-batch)

6.4.1. Configuración del ejemplo

Definimos:

- Número de clases $C = 4$.
- Mini-batch de $B = 3$ muestras.
- Las etiquetas verdaderas (enteros) para las 3 muestras son: $[2,\ 0,\ 3]$.
- Supondremos que el modelo (por ejemplo una capa lineal simple) produce los siguientes *logits* z (matriz $B \times C$):

$$Z = \begin{bmatrix} 2{,}0 & 0{,}5 & 0{,}1 & -1{,}2 \\ 0{,}2 & 1{,}5 & -0{,}5 & 0{,}0 \\ -1{,}0 & 0{,}0 & 0{,}2 & 2{,}0 \end{bmatrix}.$$

- Para cada muestra convertiremos z en probabilidades p mediante softmax, calcularemos la pérdida por muestra $-\log p_k$ y la pérdida promedio del mini-batch.

6.4.2. Paso 1: calcular softmax por fila

Muestra 1 logits $z^{(1)} = [2{,}0,\ 0{,}5,\ 0{,}1,\ -1{,}2]$.

$$\begin{aligned} \text{denom}_1 &= e^{2{,}0} + e^{0{,}5} + e^{0{,}1} + e^{-1{,}2} \\ &\approx 7{,}3891 + 1{,}6487 + 1{,}1052 + 0{,}3010 \\ &\approx 10{,}4440. \end{aligned}$$

Probabilidades:

$$p^{(1)} \approx \frac{1}{10{,}4440}[7{,}3891,\ 1{,}6487,\ 1{,}1052,\ 0{,}3010] \approx [0{,}7074,\ 0{,}1579,\ 0{,}1059,\ 0{,}0288].$$

Muestra 2 logits $z^{(2)} = [0{,}2,\ 1{,}5,\ -0{,}5,\ 0{,}0]$.

$$\text{denom}_2 \approx e^{0{,}2} + e^{1{,}5} + e^{-0{,}5} + e^{0{,}0} \approx 1{,}2214 + 4{,}4817 + 0{,}6065 + 1{,}0000 \approx 7{,}3096.$$

$$p^{(2)} \approx [0{,}1671,\ 0{,}6134,\ 0{,}0830,\ 0{,}1365].$$

Muestra 3 logits $z^{(3)} = [-1{,}0,\ 0{,}0,\ 0{,}2,\ 2{,}0]$.

$$\text{denom}_3 \approx e^{-1{,}0} + e^{0{,}0} + e^{0{,}2} + e^{2{,}0} \approx 0{,}3679 + 1{,}0000 + 1{,}2214 + 7{,}3891 \approx 9{,}9784.$$

$$p^{(3)} \approx [0{,}0369,\ 0{,}1002,\ 0{,}1224,\ 0{,}7405].$$

6.4.3. Paso 2: calcular la pérdida por muestra (Cross Entropy)

Las etiquetas verdaderas son $[2, 0, 3]$, es decir:

$$y^{(1)} = [0, 0, 1, 0], \quad y^{(2)} = [1, 0, 0, 0], \quad y^{(3)} = [0, 0, 0, 1].$$

La pérdida por muestra es $-\log p_k$:

$$\begin{aligned} J^{(1)} &= -\log p_2^{(1)} \quad (\text{clase } k = 2 \text{ para muestra } 1) \\ &= -\log(0{,}1059) \approx 2{,}245, \\ J^{(2)} &= -\log p_0^{(2)} = -\log(0{,}1671) \approx 1{,}790, \\ J^{(3)} &= -\log p_3^{(3)} = -\log(0{,}7405) \approx 0{,}301. \end{aligned}$$

Pérdida promedio del mini-batch:

$$J_{\text{batch}} = \frac{1}{B} \sum_{b=1}^{B} J^{(b)} \approx \frac{2{,}245 + 1{,}790 + 0{,}301}{3} \approx 1{,}445.$$

6.4.4. Interpretación

- La muestra 1 tuvo alta pérdida porque la probabilidad asignada a la clase verdadera (clase 2) fue baja (0.1059).
- La muestra 3 obtuvo baja pérdida pues el modelo predijo con alta confianza la clase correcta (0.7405).
- El objetivo del entrenamiento es *minimizar* la pérdida promedio: ajustar pesos para aumentar p_k en las muestras correctas.

6.4.5. Cálculo del gradiente básico: $\partial J / \partial z$

Para usar retropropagación necesitamos el gradiente de la pérdida con respecto a los logits z. Una propiedad clave es que (para Cross Entropy con softmax) el gradiente por muestra es muy simple:

$$\frac{\partial J}{\partial z_i} = p_i - y_i.$$

Esto se deriva aplicando la regla de la cadena a $J = -\sum_j y_j \log p_j$ con $p = \text{softmax}(z)$. Es una ventaja práctica enorme: no necesitamos derivar manualmente softmax compuesta con el log — el resultado es $p - y$.

Gradientes por muestra (ejemplo numérico):

$$\begin{aligned} g^{(1)} &= p^{(1)} - y^{(1)} = [0{,}7074,\ 0{,}1579,\ 0{,}1059,\ 0{,}0288] - [0, 0, 1, 0] \\ &= [0{,}7074,\ 0{,}1579,\ -0{,}8941,\ 0{,}0288]. \\ g^{(2)} &= p^{(2)} - y^{(2)} = [0{,}1671,\ 0{,}6134,\ 0{,}0830,\ 0{,}1365] - [1, 0, 0, 0] \\ &= [-0{,}8329,\ 0{,}6134,\ 0{,}0830,\ 0{,}1365]. \\ g^{(3)} &= p^{(3)} - y^{(3)} = [0{,}0369,\ 0{,}1002,\ 0{,}1224,\ 0{,}7405] - [0, 0, 0, 1] \\ &= [0{,}0369,\ 0{,}1002,\ 0{,}1224,\ -0{,}2595]. \end{aligned}$$

Estos vectores $g^{(b)} \in \mathbb{R}^C$ son los gradientes locales que se propagan hacia las capas previas (multiplicados por las entradas) para obtener gradientes de pesos.

6.5. Ejemplo de *ciclo* de entrenamiento (conceptual)

A modo de resumen, un ciclo de entrenamiento con un mini-batch implica:

1. **Forward:** calcular logits $z^{(b)}$ para cada muestra b, transformar a probabilidades $p^{(b)} = \text{softmax}(z^{(b)})$, y calcular pérdida por muestra $J^{(b)} = -\log p^{(b)}_{k^{(b)}}$. Calcular pérdida promedio J_{batch}.
2. **Backward:** calcular gradientes locales $g^{(b)} = p^{(b)} - y^{(b)}$, combinar con entradas para obtener $\partial J/\partial W$ y $\partial J/\partial b$, y propagar más atrás si hay capas previas.
3. **Actualización:** aplicar regla de actualización (p.ej. SGD/Adam) para modificar W y b.
4. Repetir para siguientes batches y épocas.

6.6. Consejos prácticos

- Para depurar: tome un mini-batch pequeño y calcule manualmente los pasos (como en este ejemplo) y compare con la salida de su implementación automática.
- Observe que la suma de p por fila debe ser 1; la predicción se obtiene por $\arg\max$ de los logits (o de p).
- No aplique softmax antes de `CrossEntropyLoss` en PyTorch: pase logits directamente para aprovechar la estabilidad numérica.
- Entender que $\partial J/\partial z = p - y$ es una identidad práctica que facilita comprender la retropropagación en clasificación.

6.7. Comentarios

La codificación *one-hot* convierte la etiqueta discreta en un vector indicador compatible con operaciones vectoriales. La *Cross Entropy* cuantifica la discrepancia entre la probabilidad predicha por softmax y la etiqueta verdadera; su forma combinada con softmax produce una expresión de gradiente simple y eficiente ($p - y$), lo que facilita el cálculo de actualizaciones por retropropagación. El ejemplo numérico ofrecido ilustra paso a paso cómo se computa la pérdida y su gradiente en un mini-batch, ayudando a estudiantes principiantes a conectar la teoría con los cálculos concretos que realiza una implementación en PyTorch.

7. Ejercicio 5: Entrenamiento y validación de una red neuronal

Objetivo

Entrenar y validar una red neuronal feedforward para clasificar imágenes del conjunto Fashion-MNIST. Se implementan funciones para recorrer lotes, calcular pérdida y precisión, y visualizar resultados.

7.1. Definición del modelo

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# 1) Definición del modelo
```

```
class FeedforwardNN(nn.Module):
    def __init__(self):
        super(FeedforwardNN, self).__init__()
        self.fc1 = nn.Linear(784, 128)
        self.dropout1 = nn.Dropout(p=0.2)
        self.fc2 = nn.Linear(128, 64)
        self.dropout2 = nn.Dropout(p=0.2)
        self.fc3 = nn.Linear(64, 10)

    def forward(self, x):
        x = x.view(x.size(0), -1)
        x = F.relu(self.fc1(x))
        x = self.dropout1(x)
        x = F.relu(self.fc2(x))
        x = self.dropout2(x)
        x = self.fc3(x)
        return x
```

Explicación:

- `nn.Linear(a, b)`: capa totalmente conectada con a entradas y b salidas.
- `Dropout(p=0.2)`: apaga aleatoriamente el 20 % de las neuronas durante entrenamiento.
- `F.relu(...)`: activa neuronas con ReLU (positivos pasan, negativos se anulan).
- `x.view(...)`: aplana la imagen de 28×28 a un vector de 784 elementos.
- La capa de salida tiene 10 neuronas, una por clase.

7.2. Función de entrenamiento

```
# 2) Función de entrenamiento
def train_one_epoch(model, dataloader, loss_fn, optimizer, device):
    model.train()
    running_loss = 0.0
    correct = 0
    total = 0

    for images, labels in dataloader:
        images, labels = images.to(device), labels.to(device)

        optimizer.zero_grad()
        outputs = model(images)
        loss = loss_fn(outputs, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item() * images.size(0)
        _, predicted = torch.max(outputs, 1)
        correct += (predicted == labels).sum().item()
        total += labels.size(0)

    avg_loss = running_loss / total
    accuracy = correct / total
    return avg_loss, accuracy
```

7.2.1. Explicación breve

- `model.train()`: activa modo entrenamiento.

- `optimizer.zero_grad()`: borra gradientes anteriores.
- `loss.backward()`: calcula gradientes.
- `optimizer.step()`: actualiza pesos.
- `torch.max(outputs, 1)`: predicción más probable.
- `accuracy`: fracción de aciertos.

7.2.2. Explicación ampliada

¿Qué hace la función de entrenamiento?

Esta función entrena el modelo durante una época completa. Recorre todos los lotes (`batches`) del conjunto de entrenamiento, actualiza los pesos del modelo y calcula dos métricas:

- **Pérdida promedio** (`avg_loss`): mide qué tan bien está aprendiendo el modelo.
- **Precisión** (`accuracy`): mide cuántas predicciones fueron correctas.

Explicación paso a paso

`model.train()`

Activa el modo entrenamiento del modelo. Esto es importante porque algunas capas como `Dropout` o `BatchNorm` se comportan distinto en entrenamiento que en validación.

Inicialización de acumuladores

- `running_loss`: acumula la pérdida total.
- `correct`: cuenta cuántas predicciones fueron correctas.
- `total`: cuenta cuántos ejemplos se procesaron.

Loop sobre los lotes

```
for images, labels in dataloader:
```

Recorre todos los lotes del conjunto de entrenamiento. Cada lote contiene:

- `images`: tensor con las imágenes.
- `labels`: tensor con las etiquetas verdaderas.

Copiar datos al dispositivo

```
images, labels = images.to(device), labels.to(device)
```

Esto asegura que los datos estén en el mismo dispositivo que el modelo (CPU o GPU).

Forward pass y cálculo de pérdida

```
optimizer.zero_grad()
outputs = model(images)
loss = loss_fn(outputs, labels)
```

- `zero_grad()`: borra gradientes anteriores.
- `model(images)`: calcula las predicciones del modelo.
- `loss_fn(...)`: calcula la pérdida entre las predicciones y las etiquetas verdaderas.

Backward pass y actualización de pesos

```
loss.backward()
optimizer.step()
```

- `backward()`: calcula los gradientes de la pérdida respecto a los pesos.
- `step()`: actualiza los pesos del modelo usando los gradientes.

Acumulación de métricas

```
running_loss += loss.item() * images.size(0)
_, predicted = torch.max(outputs, 1)
correct += (predicted == labels).sum().item()
total += labels.size(0)
```

- `loss.item()`: convierte la pérdida a número Python.
- `images.size(0)`: número de ejemplos en el lote.
- `torch.max(outputs, 1)`: selecciona la clase con mayor probabilidad.
- `(predicted == labels)`: compara predicciones con etiquetas reales.
- `sum().item()`: cuenta cuántas fueron correctas.

`running_loss += loss.item() * images.size(0)`: Esta línea acumula la pérdida total del modelo durante una época de entrenamiento. Es parte del cálculo del promedio de pérdida sobre todos los ejemplos procesados.

Desglose paso a paso

a) `loss.item()`

- `loss` es un tensor que representa la pérdida promedio del lote actual.
- `.item()` convierte ese tensor en un número Python (float).
- Por ejemplo, si la pérdida del lote es `tensor(0.45)`, entonces `loss.item()` devuelve `0.45`.

b) `images.size(0)`

- `images` es un tensor de forma `(batch_size, canales, alto, ancho)`.
- `images.size(0)` devuelve el tamaño del primer eje, es decir, el número de imágenes en el lote.
- Por ejemplo, si el lote tiene 100 imágenes, entonces `images.size(0)` devuelve `100`.

c) Multiplicación: `loss.item() * images.size(0)`

- La pérdida que devuelve `loss.item()` es promedio por imagen.
- Multiplicarla por el número de imágenes nos da la pérdida total del lote.
- Ejemplo: si la pérdida promedio es `0.45` y hay `100` imágenes, entonces la pérdida total es `0.45 × 100 = 45.0`.

d) Acumulación: `running_loss += ...`

- `running_loss` es una variable que acumula la pérdida total de todos los lotes procesados.
- Al sumar la pérdida total del lote actual, vamos construyendo la pérdida total de la época.

¿Por qué se hace esto?

Porque al final de la época queremos calcular la **pérdida promedio por imagen**, no por lote. Para eso:

- Sumamos la pérdida total de todos los lotes.
- Luego dividimos por el número total de imágenes procesadas.

Ejemplo numérico

Supongamos que procesamos 3 lotes:

- Lote 1: pérdida promedio = 0.4, tamaño = 100 $\rightarrow$ pérdida total = 40
- Lote 2: pérdida promedio = 0.5, tamaño = 100 $\rightarrow$ pérdida total = 50
- Lote 3: pérdida promedio = 0.6, tamaño = 100 $\rightarrow$ pérdida total = 60

Entonces:

$$\text{running\_loss} = 40 + 50 + 60 = 150$$

Y si procesamos 300 imágenes:

$$\text{avg\_loss} = \frac{150}{300} = 0{,}5$$

Cálculos finales

```
avg_loss = running_loss / total
accuracy = correct / total
return avg_loss, accuracy
```

Se calcula la pérdida promedio y la precisión total sobre todos los ejemplos procesados.

7.2.3. Comentarios

La función de entrenamiento permite que el modelo aprenda ajustando sus pesos en función de los errores que comete. Además, devuelve métricas que ayudan a monitorear el progreso del aprendizaje. **Importante:** Recordar que los errores calculados durante la etapa de entrenamiento están estimados por exceso (los vamos calculando a medida que entrenamos la red con el conjunto de datos de entrenamiento), de manera que sólo sirven para monitorear cualitativamente si el procedimiento de entrenamiento de la red evoluciona de la manera esperada (Cómo se pueden calcular correctamente?).

7.3. Función de validación

```python
# 3) Función de validación
def validate(model, dataloader, loss_fn, device):
    model.eval()
    running_loss = 0.0
    correct = 0
    total = 0

    with torch.no_grad():
        for images, labels in dataloader:
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            loss = loss_fn(outputs, labels)

            running_loss += loss.item() * images.size(0)
            _, predicted = torch.max(outputs, 1)
            correct += (predicted == labels).sum().item()
            total += labels.size(0)

    avg_loss = running_loss / total
    accuracy = correct / total
    return avg_loss, accuracy
```

7.3.1. Explicación breve:

- `model.eval()`: desactiva dropout.
- `torch.no_grad()`: evita cálculo de gradientes.
- Mismo cálculo de pérdida y precisión que en entrenamiento.

7.3.2. Explicación ampliada de la función `validate`

¿Qué hace esta función?

La función `validate` evalúa el rendimiento de un modelo entrenado sobre un conjunto de datos de validación. Calcula dos métricas:

- **Pérdida promedio** (`avg_loss`): mide el error del modelo.
- **Precisión** (`accuracy`): mide cuántas predicciones fueron correctas.

Explicación paso a paso

a) `model.eval()`

Activa el modo evaluación del modelo. Esto desactiva capas como `Dropout` y `BatchNorm`, que se comportan distinto en entrenamiento.

b) Inicialización de acumuladores

- `running_loss`: suma total de pérdidas.
- `correct`: número de predicciones correctas.
- `total`: número total de ejemplos procesados.

c) `with torch.no_grad()`

Desactiva el cálculo de gradientes. Esto ahorra memoria y tiempo porque en validación no se actualizan los pesos del modelo.

d) Loop sobre los lotes

```
for images, labels in dataloader:
```

Recorre todos los lotes del conjunto de validación. Cada lote contiene:

- `images`: tensor con las imágenes.
- `labels`: tensor con las etiquetas verdaderas.

e) Copiar datos al dispositivo

```
images, labels = images.to(device), labels.to(device)
```

Asegura que los datos estén en el mismo dispositivo que el modelo (CPU o GPU).

f) Forward pass y cálculo de pérdida

```
outputs = model(images)
loss = loss_fn(outputs, labels)
```

- `model(images)`: genera predicciones.
- `loss_fn(...)`: calcula la pérdida entre las predicciones y las etiquetas verdaderas.

g) Acumulación de pérdida

```
running_loss += loss.item() * images.size(0)
```

Multiplica la pérdida promedio del lote por el número de imágenes para obtener la pérdida total del lote. Luego la suma a `running_loss`.

h) Cálculo de predicciones correctas

```
_, predicted = torch.max(outputs, 1)
correct += (predicted == labels).sum().item()
total += labels.size(0)
```

- `torch.max(outputs, 1)`: selecciona la clase con mayor probabilidad.
- `(predicted == labels)`: compara predicciones con etiquetas reales.
- `sum().item()`: cuenta cuántas fueron correctas.
- `total`: suma el número de ejemplos procesados.

i) Cálculo final

```
avg_loss = running_loss / total
accuracy = correct / total
return avg_loss, accuracy
```

Calcula la pérdida promedio por imagen y la precisión total sobre el conjunto de validación.

7.3.3. Comentarios

Esta función permite evaluar objetivamente el rendimiento del modelo sin modificar sus parámetros. Es esencial para monitorear el aprendizaje y detectar sobreajuste (*"overfitting"*).

7.4. Preparación de los datos

```
# 4) Preparación de datos
from torchvision import datasets, transforms #(Esta línea se coloca al
    principio del programa)

transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])

train_set = datasets.FashionMNIST(root='./data', train=True, download=True,
    transform=transform)
valid_set = datasets.FashionMNIST(root='./data', train=False, download=True,
    transform=transform)

train_loader = DataLoader(train_set, batch_size=100, shuffle=True)
valid_loader = DataLoader(valid_set, batch_size=100, shuffle=False)
```

¿Qué hace este bloque?

Este bloque prepara los datos del conjunto **Fashion-MNIST** para ser utilizados en el entrenamiento y validación de una red neuronal. Incluye:

- Descarga del dataset.
- Transformación de las imágenes.
- Creación de los `DataLoader`s para iterar por lotes.

7.4.1. Explicación paso a paso

a) Importación de módulos

```
from torchvision import datasets, transforms
```

- `datasets`: contiene conjuntos de datos listos para usar.
- `transforms`: permite aplicar transformaciones a las imágenes (como convertirlas a tensores o normalizarlas).

b) Definición de transformaciones

```
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])
```

- `Compose([...])`: encadena varias transformaciones.
- `ToTensor()`: convierte la imagen PIL (formato imagen) a un tensor de PyTorch con valores en $[0, 1]$.

- `Normalize((0.5,), (0.5,))`: normaliza los valores a $[-1, 1]$ usando la fórmula:

$$x_{\text{normalizado}} = \frac{x - 0{,}5}{0{,}5}$$

c) Carga del conjunto de entrenamiento

```
train_set = datasets.FashionMNIST(root='./data', train=True, download=True,
    transform=transform)
```

- `root='./data'`: carpeta donde se guardan los datos.
- `train=True`: indica que se quiere el conjunto de entrenamiento.
- `download=True`: descarga el dataset si no está presente.
- `transform=transform`: aplica las transformaciones definidas.

d(Carga del conjunto de validación

```
valid_set = datasets.FashionMNIST(root='./data', train=False, download=True,
    transform=transform)
```

- `train=False`: indica que se quiere el conjunto de prueba (usado como validación).

e) Creación del `DataLoader` de entrenamiento

```
train_loader = DataLoader(train_set, batch_size=100, shuffle=True)
```

- Divide el conjunto de entrenamiento en lotes de 100 imágenes.
- `shuffle=True`: mezcla aleatoriamente los datos en cada época para mejorar el aprendizaje.

f) Creación del `DataLoader` de validación

```
valid_loader = DataLoader(valid_set, batch_size=100, shuffle=False)
```

- Divide el conjunto de validación en lotes de 100 imágenes.
- `shuffle=False`: mantiene el orden original (no es necesario mezclar en validación).

7.4.2. Comentarios

Este bloque es esencial para preparar los datos antes de entrenar una red neuronal. Convierte las imágenes en tensores normalizados, las organiza en lotes y permite iterar sobre ellas de forma eficiente durante el entrenamiento y la validación.

7.5. Inicialización de componentes

```
# 5) Inicialización de componentes
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = FeedforwardNN().to(device)
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=1e-3)
```

7.5.1. Explicación detallada

a) Selección del dispositivo

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

- Verifica si hay una GPU disponible con `torch.cuda.is_available()`.
- Si hay GPU, usa çuda" como dispositivo.
- Si no hay GPU, usa çpu".
- Esto permite que el código funcione en cualquier computadora, aprovechando la aceleración si está disponible.

b) Crear el modelo y moverlo al dispositivo

```
model = FeedforwardNN().to(device)
```

- Se crea una instancia del modelo definido previamente (`FeedforwardNN`).
- `.to(device)` mueve el modelo al dispositivo seleccionado (CPU o GPU).
- Esto es necesario para que los datos y el modelo estén en el mismo lugar durante el entrenamiento.

c) Definir la función de pérdida

```
loss_fn = nn.CrossEntropyLoss()
```

- Se utiliza la función de pérdida de entropía cruzada.
- Es adecuada para clasificación multiclase.
- Compara las predicciones del modelo (logits) con las etiquetas verdaderas.
- Internamente aplica `log_softmax`, por lo que no se necesita activación en la capa de salida.

7.5.2. d) Definir el optimizador

```
optimizer = torch.optim.SGD(model.parameters(), lr=1e-3)
```

- Se utiliza el método de descenso de gradiente estocástico (SGD).
- `model.parameters()` indica que se van a optimizar los pesos del modelo.
- `lr=1e-3` establece la tasa de aprendizaje en $0{,}001$.
- El optimizador ajustará los pesos del modelo para minimizar la pérdida.

Comentarios

Este bloque de código prepara todos los componentes necesarios para entrenar una red neuronal en PyTorch:

- Selecciona el dispositivo de cómputo.
- Crea y mueve el modelo al dispositivo.
- Define la función de pérdida.
- Configura el optimizador.

Estos pasos son esenciales antes de comenzar el ciclo de entrenamiento.

7.6. Bucle de entrenamiento y validación

```
# 6) Bucle de entrenamiento y validación
num_epochs = 100
train_losses = []
train_accuracies = []
valid_losses = []
valid_accuracies = []

for epoch in range(num_epochs):
    train_loss, train_acc = train_one_epoch(model, train_loader, loss_fn,
        optimizer, device)
    valid_loss, valid_acc = validate(model, valid_loader, loss_fn, device)

    train_losses.append(train_loss)
    train_accuracies.append(train_acc)
    valid_losses.append(valid_loss)
    valid_accuracies.append(valid_acc)

    print(f"Época {epoch+1}:")
    print(f"  Entrenamiento -> Pérdida: {train_loss:.4f}, Precisión: {train_acc
        :.4f}")
    print(f"  Validación    -> Pérdida: {valid_loss:.4f}, Precisión: {valid_acc
        :.4f}")
```

7.6.1. Explicación detallada

¿Qué hace este bloque?

Este bloque ejecuta el ciclo de entrenamiento y validación de una red neuronal durante 100 épocas. En cada época:

- Se entrena el modelo con los datos de entrenamiento.
- Se evalúa el modelo con los datos de validación.
- Se registran las métricas de pérdida y precisión.
- Se imprime el progreso en pantalla.

Explicación paso a paso

a) `num_epochs = 100`

Define cuántas veces se repite el ciclo completo de entrenamiento y validación. Cada repetición se llama **época**.

b) Inicialización de listas

```
train_losses = []
train_accuracies = []
valid_losses = []
valid_accuracies = []
```

Estas listas guardan los valores de pérdida y precisión por época, tanto para entrenamiento como para validación. Luego se pueden usar para graficar el progreso.

c) Bucle principal

```
for epoch in range(num_epochs):
```

Recorre las épocas desde 0 hasta 99 (100 en total).

d) Entrenamiento del modelo

```
train_loss, train_acc = train_one_epoch(...)
```

Llama a la función que entrena el modelo con los datos de entrenamiento. Devuelve:

- `train_loss`: pérdida promedio en entrenamiento.
- `train_acc`: precisión en entrenamiento.

e) Validación del modelo

```
valid_loss, valid_acc = validate(...)
```

Llama a la función que evalúa el modelo con los datos de validación. Devuelve:

- `valid_loss`: pérdida promedio en validación.
- `valid_acc`: precisión en validación.

f) Registro de métricas

```
train_losses.append(train_loss)
train_accuracies.append(train_acc)
valid_losses.append(valid_loss)
valid_accuracies.append(valid_acc)
```

Agrega los valores de la época actual a las listas correspondientes.

g) Visualización en pantalla de los resultados

```
print(f"Época {epoch+1}:")
print(f"  Entrenamiento -> Pérdida: {train_loss:.4f}, Precisión: {train_acc:.4f
    }")
print(f"  Validación    -> Pérdida: {valid_loss:.4f}, Precisión: {valid_acc:.4f
    }")
```

Muestra en pantalla los resultados de la época actual. El formato `:.4f` limita los decimales a 4 cifras.

¿Por qué es importante este bucle?

- Permite que el modelo aprenda progresivamente.
- Monitorea el rendimiento en entrenamiento y validación.
- Ayuda a detectar sobreajuste (*"overfitting"*) si la precisión en entrenamiento sube pero en validación baja.
- Facilita la visualización del progreso mediante gráficos.

7.6.2. Comentarios

Este bucle es el corazón del proceso de entrenamiento. Ejecuta las funciones de entrenamiento y validación, guarda las métricas y muestra el progreso. Es fundamental para evaluar cómo mejora el modelo a lo largo de las épocas.

7.7. Visualización de las métricas de pérdida y precisión

```
# 7) Gráficos de pérdida y precisión
plt.figure(figsize=(8,4))
plt.plot(train_losses, label='Entrenamiento')
plt.plot(valid_losses, label='Validación')
plt.title("Pérdida (Cross Entropy) por época")
plt.xlabel("Época")
plt.ylabel("Pérdida")
plt.legend()
plt.grid(True)
plt.show()

plt.figure(figsize=(8,4))
plt.plot(train_accuracies, label='Entrenamiento')
plt.plot(valid_accuracies, label='Validación')
plt.title("Precisión por época")
plt.xlabel("Época")
plt.ylabel("Precisión")
plt.ylim(0,1)
plt.legend()
plt.grid(True)
plt.show()
```

7.7.1. Explicación detallada

¿Qué hace este bloque?

Este bloque genera dos gráficos usando la biblioteca `matplotlib.pyplot`:

- El primero muestra cómo evoluciona la **pérdida** (error) durante el entrenamiento y la validación.
- El segundo muestra cómo evoluciona la **precisión** (porcentaje de aciertos) en ambas fases.

a) `plt.figure(figsize=(8,4))`

Crea una nueva figura de tamaño 8×4 pulgadas. Esto define el espacio donde se dibujará el gráfico.

b) `plt.plot(...)`

Dibuja una curva en el gráfico:

- `train_losses`: lista con la pérdida por época en entrenamiento.
- `valid_losses`: lista con la pérdida por época en validación.
- `label`: nombre que aparecerá en la leyenda.

c) `plt.title(...)`

Agrega un título descriptivo al gráfico. En este caso: `"Pérdida (Cross Entropy) por época"`.

d) `plt.xlabel(...)` **y** `plt.ylabel(...)`

Etiquetas para los ejes:

- Eje X: número de época.
- Eje Y: valor de pérdida o precisión.

e) `plt.legend()`

Muestra la leyenda con los nombres de las curvas (Entrenamiento y Validación).

f) `plt.grid(True)`

Agrega una cuadrícula al fondo del gráfico para facilitar la lectura de valores.

g) `plt.show()`

Muestra el gráfico en pantalla.

h) Segundo gráfico: Precisión

El segundo bloque repite el proceso para graficar la precisión. Se agregan dos detalles importantes:

- `plt.ylim(0,1)`: fija el eje Y entre 0 y 1, ya que la precisión es una fracción.

7.7.2. ¿Por qué es útil graficar estas métricas?

- Permite visualizar si el modelo está aprendiendo correctamente.
- Ayuda a detectar sobreajuste (*"overfitting"*): cuando la precisión en entrenamiento sube pero en validación baja.
- Facilita la comparación entre diferentes modelos o configuraciones.

7.7.3. Comentarios

Este bloque convierte las listas de métricas en gráficos claros y comparables. Es una herramienta fundamental para monitorear el progreso del entrenamiento y tomar decisiones informadas sobre ajustes al modelo.

7.7.4. Resultados esperados

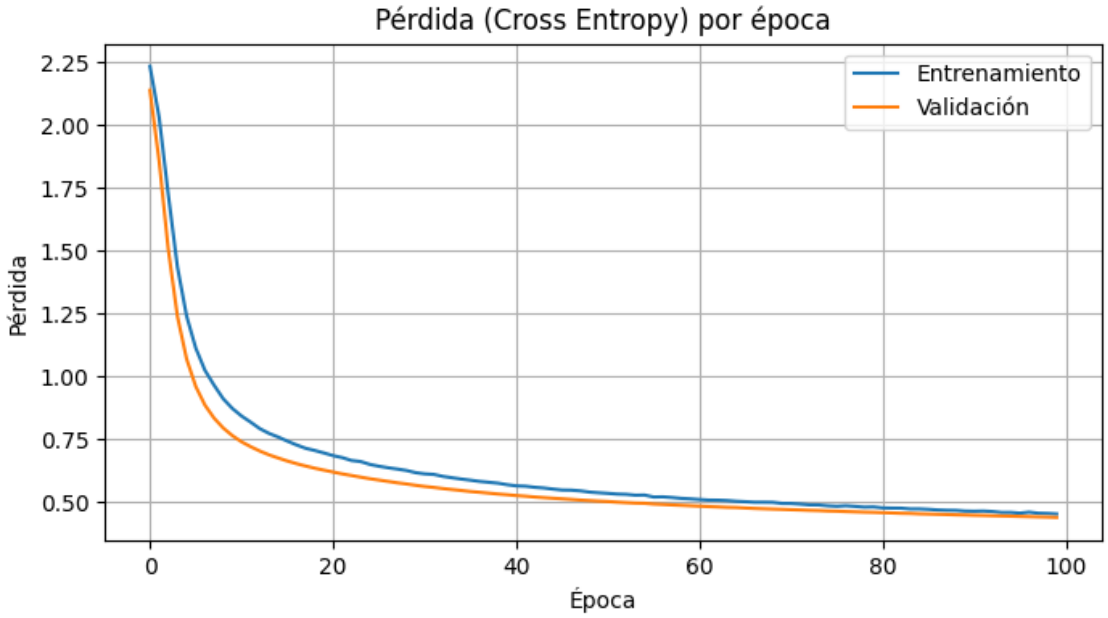

Figura 4: Evolución de la pérdida (Cross Entropy) por época durante el entrenamiento y la validación del modelo. Una pérdida decreciente indica que el modelo está aprendiendo a clasificar mejor.

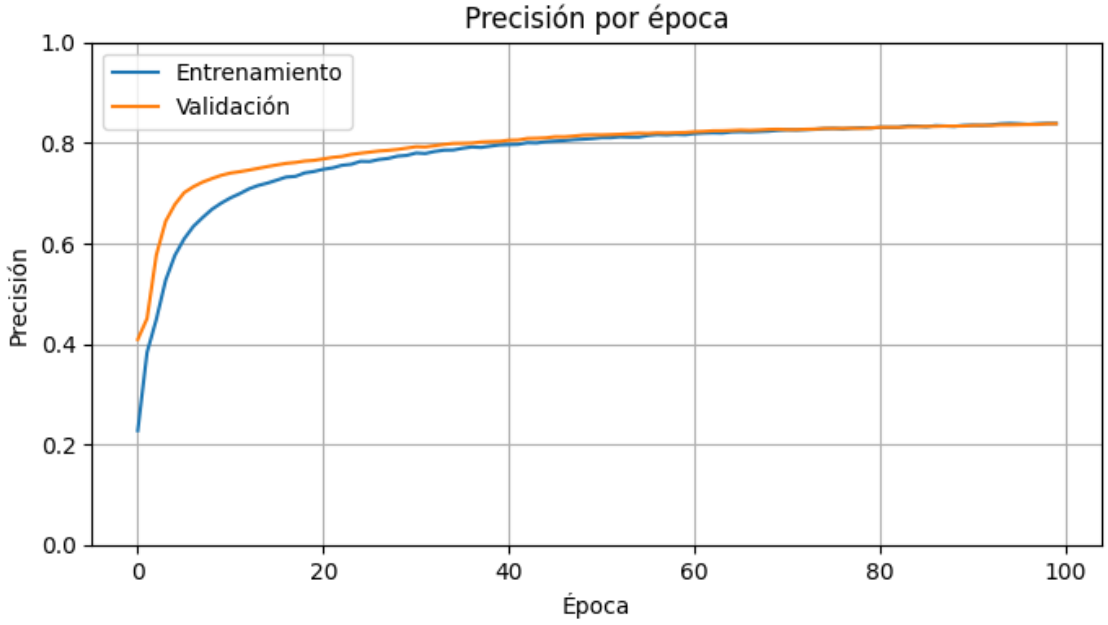

Figura 5: Evolución de la precisión por época durante el entrenamiento y la validación. Una precisión creciente indica que el modelo está acertando más en sus predicciones.

7.8. Evaluación del modelo con una sola imagen del conjunto de validación

Objetivo

Evaluar el modelo entrenado utilizando una sola imagen del conjunto de validación y visualizar la predicción junto con la imagen.

7.9. Código paso a paso

```python
# 1. Obtener un batch del conjunto de validación
images, labels = next(iter(valid_loader))

# 2. Seleccionar una imagen y su etiqueta
indice = 0 #(podemos cambiar el 0 por otro número natural entre 0 y 100)
image = images[indice]
label = labels[indice]

# 3. Agregar dimensión de batch
image = image.unsqueeze(0)  # Forma: [1, 1, 28, 28]

# 4. Mover al dispositivo
image = image.to(device)
label = label.to(device)

# 5. Evaluar el modelo
model.eval()
with torch.no_grad():
    output = model(image)
    _, predicted_class = torch.max(output, 1)

# 6. Mostrar resultados
print(f"Etiqueta verdadera: {label.item()}")
print(f"Clase predicha: {predicted_class.item()}")
```

7.10. Visualización de la imagen

```python
import matplotlib.pyplot as plt

# Desnormalizar para visualizar correctamente
image_np = image.cpu().squeeze().numpy() * 0.5 + 0.5

plt.imshow(image_np, cmap="gray")
plt.title(f"Predicción: {predicted_class.item()} | Etiqueta: {label.item()}")
plt.axis("off")
plt.show()
```

7.11. Explicación paso a paso

- `next(iter(valid_loader))`: toma un lote del conjunto de validación.
- `images[0]`: selecciona la primera imagen del lote.
- `unsqueeze(0)`: agrega una dimensión para simular un lote de tamaño 1.
- `model.eval()`: activa el modo evaluación del modelo.
- `torch.no_grad()`: evita calcular gradientes (más eficiente).
- `torch.max(output, 1)`: selecciona la clase con mayor probabilidad.
- `.item()`: convierte el tensor escalar a número Python.
- `image.cpu().squeeze().numpy() * 0.5 + 0.5`: desnormaliza la imagen para mostrarla correctamente.

7.12. Resultados esperados

```
Etiqueta verdadera: 9
Clase predicha: 9
```

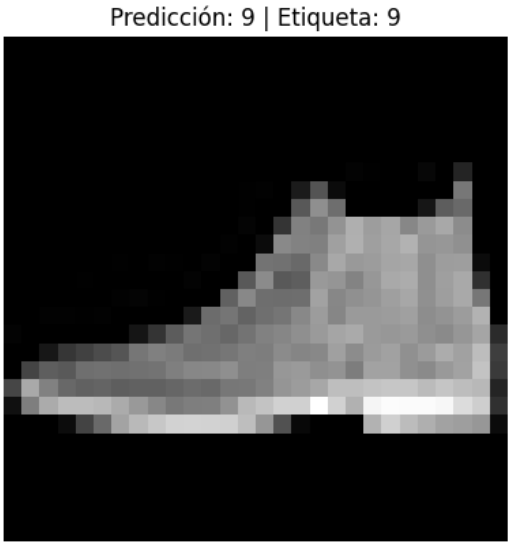

Figura 6: Evaluación del modelo con una sola imagen del conjunto de validación. Se muestra la imagen original (desnormalizada) junto con la clase predicha por la red neuronal y la etiqueta verdadera. Esta visualización permite verificar si el modelo está clasificando correctamente ejemplos individuales.

7.13. Comentarios

Este procedimiento permite evaluar el modelo con una sola imagen real del conjunto de validación, visualizarla y verificar si la predicción es correcta. Es útil para hacer pruebas rápidas y entender cómo responde el modelo a ejemplos individuales.