Redes Neuronales (2025)

Ayudas Guía N°12: Autoencoder convolucional Fashion-MNIST con PyTorch
(Trabajo en progreso)*

Índice

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

1. ¿Qué es un autoencoder convolucional?

Un **autoencoder convolucional** es una red neuronal que aprende a comprimir y reconstruir imágenes utilizando **capas convolucionales** en lugar de capas lineales. Está compuesto por dos partes:

- **Encoder:** aplica convoluciones para extraer características espaciales y reducir la resolución.
- **Decoder:** aplica convoluciones transpuestas (o interpolaciones) para reconstruir la imagen original.

Este tipo de autoencoder preserva la estructura espacial de la imagen, lo que lo hace más adecuado para datos visuales como Fashion-MNIST.

2. Flujo general del autoencoder

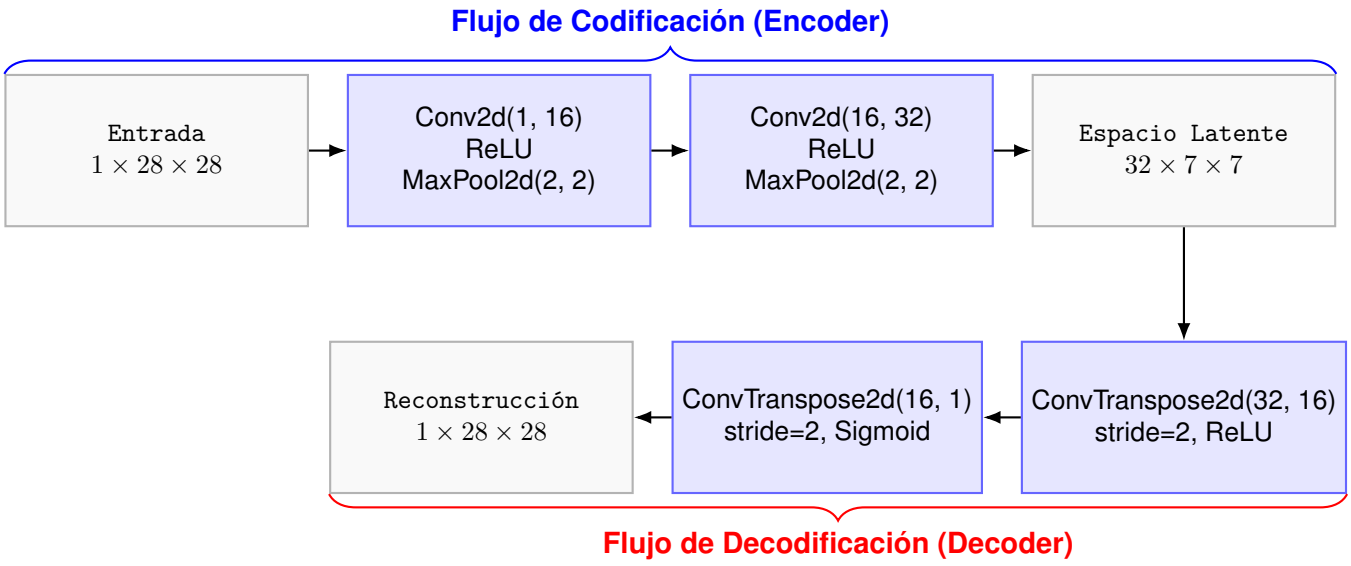

Figura 1: Arquitectura de un Autoencoder Convolucional (CAE) con flujo superior (Encoder) e inferior (Decoder).

3. ¿Qué es el espacio latente?

El **espacio latente** es el conjunto de representaciones internas que un autoencoder aprende para describir los datos de entrada de forma comprimida y significativa. Cada imagen se transforma en un punto dentro de este espacio, que captura sus características esenciales.

3.1. Forma del espacio latente

En autoencoders convolucionales, el espacio latente suele tener forma de tensor tridimensional:

$$\text{Espacio latente} \in \mathbb{R}^{C \times H' \times W'}$$

Donde:

- C es el número de canales (profundidad).
- H', W' son las dimensiones espaciales reducidas.

Por ejemplo, una imagen de entrada de $1 \times 28 \times 28$ puede codificarse como un tensor latente de $32 \times 7 \times 7$.

3.2. ¿Por qué se considera de dimensión reducida?

Aunque el tensor latente puede contener más valores que la imagen original, se considera de **dimensión reducida** porque:

- Los valores están organizados para capturar solo la información esencial.
- Hay menos **grados de libertad efectivos**: los valores están correlacionados y estructurados.
- El encoder fuerza una compresión semántica, no necesariamente una reducción numérica.

3.3. Compresión no lineal

El encoder aprende una transformación no lineal:

$$f_{\text{enc}} : \mathbb{R}^{H \times W} \to \mathbb{R}^{C \times H' \times W'}$$

Esta transformación proyecta las imágenes en una **variedad latente** de menor dimensión dentro del espacio total.

3.4. Visualización del flujo hacia el espacio latente

3.5. Aplicaciones del espacio latente

Una vez entrenado el autoencoder, el espacio latente puede utilizarse para:

- **Reconstrucción:** el decoder traduce la representación latente de vuelta a la imagen original.
- **Interpolación:** se pueden generar transiciones suaves entre dos imágenes.
- **Clasificación:** se puede entrenar un clasificador sobre las representaciones latentes.
- **Visualización:** si el espacio latente tiene 2 o 3 dimensiones, puede graficarse para explorar agrupamientos.

4. Función de pérdida

La reconstrucción se compara con la imagen original usando el **error cuadrático medio (MSE)**:

$$J(x, \hat{x}) = \frac{1}{n} \sum_{i=1}^{n} (x_i - \hat{x}_i)^2$$

Donde x es la imagen original y $\hat{x}$ la reconstruida. El objetivo es minimizar esta pérdida durante el entrenamiento.

5. Ventajas del enfoque convolucional

- Preserva la estructura espacial de las imágenes.
- Aprende representaciones jerárquicas (de píxeles a formas o “estructuras”).
- Generaliza mejor en tareas visuales.

6. Aplicaciones del espacio latente

Una vez entrenado, el espacio latente puede usarse para:

- Visualizar reconstrucciones y evaluar compresión.
- Interpolar entre imágenes (generación).
- Clasificar prendas usando el vector latente como entrada.

7. Descargando y jugando con el dataset Fashion-MNIST.

Objetivo

Este ejercicio tiene como propósito familiarizarse con el dataset **Fashion-MNIST**, que contiene imágenes en escala de grises de prendas de vestir. Aprenderemos a:

- Descargar y transformar los datos.
- Visualizar un conjunto de imágenes con sus respectivas etiquetas.

7.1. Descarga y transformación del dataset

Usamos la clase `FashionMNIST` de `torchvision.datasets` para obtener los datos. La transformación `ToTensor()` convierte las imágenes en tensores con valores entre 0 y 1.

```
import torch
from torchvision import datasets, transforms
import matplotlib.pyplot as plt

# Transformación: convierte imágenes PIL a tensores
transform = transforms.ToTensor()

# Descarga de conjuntos de entrenamiento y testeo
train_dataset = datasets.FashionMNIST(root='./data', train=True,
                                      download=True, transform=transform)
test_dataset = datasets.FashionMNIST(root='./data', train=False,
                                     download=True, transform=transform)
```

7.2. Diccionario de clases

Fashion-MNIST tiene 10 clases. Usamos un diccionario para traducir los números de etiqueta a nombres legibles.

```
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"
}
```

7.3. Visualización de un mosaico de imágenes

Graficamos un mosaico de 3×3 imágenes del conjunto de entrenamiento, cada una con su etiqueta correspondiente.

```
# Crear un mosaico de 3x3 imágenes
fig, axes = plt.subplots(3, 3, figsize=(8, 8))
for i in range(3):
    for j in range(3):
        index = i * 3 + j
        image, label = train_dataset[index]
        axes[i, j].imshow(image.squeeze(), cmap='gray')
        axes[i, j].set_title(class_names[label], fontsize=10)
        axes[i, j].axis('off')

plt.suptitle("Mosaico de imágenes Fashion-MNIST", fontsize=14)
plt.tight_layout()
plt.show()
```

7.3.1. Explicación breve

- `image, label = train_dataset[index]`: accede a una imagen y su etiqueta.
- `image.squeeze()`: elimina la dimensión del canal para visualizar en 2D.
- `imshow(...)`: muestra la imagen en escala de grises.
- `set_title(...)`: coloca el nombre de la clase como título.
- `axis('off')`: oculta los ejes para una visualización más limpia.

Este ejercicio permite explorar visualmente el dataset y entender cómo se representan las clases.

8. Creando un Dataset Personalizado para Autoencoder

Objetivo

Este ejercicio tiene como propósito preparar los datos para entrenar un **autoencoder**, una red neuronal que aprende a reconstruir sus entradas. Para ello, crearemos un `Dataset` personalizado que retorne pares `(input, output)` donde ambos elementos son la misma imagen.

8.1. Descarga y transformación del dataset original

Usamos `FashionMNIST` de `torchvision.datasets` y aplicamos la transformación `ToTensor()` para convertir las imágenes en tensores.

```
from torchvision import datasets, transforms

# Transformación: convierte imágenes PIL a tensores
transform = transforms.ToTensor()

# Descarga del dataset original (con etiquetas)
train_raw = datasets.FashionMNIST(root='./data', train=True,
                                  download=True, transform=transform)
test_raw = datasets.FashionMNIST(root='./data', train=False,
                                 download=True, transform=transform)
```

8.2. Definición de la clase `CustomDataset`

Creamos una clase que hereda de `torch.utils.data.Dataset` y redefine el método `__getitem__` para retornar la imagen dos veces: como entrada y como salida.

```
from torch.utils.data import Dataset

class CustomDataset(Dataset):
    def __init__(self, original_dataset):
        self.data = original_dataset

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

    def __getitem__(self, idx):
        image, _ = self.data[idx]  # ignoramos la etiqueta
        return image, image        # input = output para autoencoder
```

8.2.1. Explicación del bloque de código

- `Dataset`: clase base de PyTorch que permite definir conjuntos de datos personalizados.
- `CustomDataset`: clase que hereda de `Dataset` y redefine dos métodos obligatorios:
  - `__init__`: recibe un dataset original (por ejemplo, `FashionMNIST`) y lo guarda como atributo.
  - `__len__`: devuelve la cantidad total de muestras en el dataset.
  - `__getitem__`: accede a la muestra número `idx`, ignora la etiqueta y retorna la imagen dos veces.
- `image, image`: el autoencoder necesita aprender a reconstruir la imagen, por lo tanto la entrada y la salida son iguales.
- `_`: en Python, el guion bajo se usa para indicar que una variable no será utilizada.

Ejemplo de uso

```
from torchvision import datasets, transforms

# Dataset original con etiquetas
original_dataset = datasets.FashionMNIST(root='./data', train=True,
                                         download=True, transform=transforms.
                                            ToTensor())

# Dataset personalizado para autoencoder
autoencoder_dataset = CustomDataset(original_dataset)

# Acceder a una muestra
x_in, x_out = autoencoder_dataset[0]
```

Visualización

Podemos graficar la imagen de entrada y salida para verificar que son iguales:

```
import matplotlib.pyplot as plt

fig, axes = plt.subplots(1, 2, figsize=(6, 3))
axes[0].imshow(x_in.squeeze(), cmap='gray')
axes[0].set_title("Entrada")
axes[0].axis('off')
```

```
axes[1].imshow(x_out.squeeze(), cmap='gray')
axes[1].set_title("Salida")
axes[1].axis('off')
plt.suptitle("Par (input, output) para autoencoder")
plt.show()
```

Comentarios

Este diseño permite transformar cualquier dataset supervisado en uno no supervisado para entrenar autoencoders. Es una técnica fundamental para tareas de reconstrucción, compresión y aprendizaje de representaciones latentes.

8.3. Creación de los datasets para el Autoencoder

Usamos la clase personalizada para transformar los conjuntos de entrenamiento y testeo.

```
train_auto = CustomDataset(train_raw)
test_auto = CustomDataset(test_raw)
```

8.4. Inspección de un par `(input, output)`

Verificamos que el dataset personalizado retorna correctamente las imágenes duplicadas.

```
img_in, img_out = train_auto[0]
print(f"Forma de entrada: {img_in.shape}, forma de salida: {img_out.shape}")
```

8.5. Visualización de entrada y salida

Graficamos una imagen de entrada y su correspondiente salida para confirmar que son iguales.

```
import matplotlib.pyplot as plt

fig, axes = plt.subplots(1, 2, figsize=(6, 3))
axes[0].imshow(img_in.squeeze(), cmap='gray')
axes[0].set_title("Entrada")
axes[0].axis('off')
axes[1].imshow(img_out.squeeze(), cmap='gray')
axes[1].set_title("Salida")
axes[1].axis('off')
plt.suptitle("Par (input, output) para autoencoder")
plt.show()
```

8.6. Explicación breve

- `Dataset`: estructura que permite acceder a los datos de forma ordenada.
- `CustomDataset`: clase que modifica el comportamiento del dataset original.
- `__getitem__`: método que define qué retorna el dataset cuando se accede a un índice.
- `image, image`: usamos la misma imagen como entrada y salida, porque el autoencoder debe aprender a reconstruirla.
- `squeeze()`: elimina la dimensión del canal para visualizar la imagen en 2D.

Este ejercicio prepara los datos para entrenar un autoencoder sin etiquetas, enfocándose en la reconstrucción de imágenes.

9. Ejercicio 5: Autoencoder convolucional completo

Objetivo

Este ejercicio tiene como propósito entrenar y validar un modelo **autoencoder convolucional** utilizando el dataset `Fashion-MNIST`. Se implementan funciones de entrenamiento, evaluación, visualización de pérdidas y reconstrucciones, con explicaciones detalladas para estudiantes principiantes.

9.1. Bloque 1: Importación de librerías

```
# 1) Importación de librerías
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms
from torch.utils.data import DataLoader, Dataset
import matplotlib.pyplot as plt
```

Importamos las librerías necesarias para trabajar con PyTorch, cargar datos, construir redes neuronales y graficar resultados.

9.2. Bloques 2 y 3: Transformación de imágenes y dataset personalizado

```
# 2) Transformación: normaliza imágenes a [-1, 1]
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))  # (x - 0.5) / 0.5 -> [-1, 1]
])

# 3) Dataset personalizado para autoencoder
class CustomDataset(Dataset):
    def __init__(self, original_dataset):
        self.data = original_dataset

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

    def __getitem__(self, idx):
        image, _ = self.data[idx]
        return image, image  # input = output
```

9.2.1. Explicación breve

Este bloque prepara las imágenes del dataset `Fashion-MNIST` para entrenar un **autoencoder convolucional**. Incluye:

- Una transformación que normaliza las imágenes al rango $[-1, 1]$.
- Un dataset personalizado que retorna pares `(imagen, imagen)`.

9.2.2. Explicación ampliada

Transformación de imágenes

```
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))  # (x - 0.5) / 0.5 -> [-1, 1]
])
```

Explicación:

- `ToTensor()`: convierte la imagen de formato PIL a tensor, escalando los valores de píxeles de $[0, 255]$ a $[0{,}0, 1{,}0]$.
- `Normalize((0.5,), (0.5,))`: aplica la fórmula $(x - 0{,}5)/0{,}5$, transformando el rango $[0{,}0, 1{,}0]$ a $[-1{,}0, 1{,}0]$.
- Esta normalización es ideal cuando usamos funciones de activación como `Tanh`, que también producen salidas en ese rango.

Dataset personalizado para autoencoder

```
class CustomDataset(Dataset):
    def __init__(self, original_dataset):
        self.data = original_dataset

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

    def __getitem__(self, idx):
        image, _ = self.data[idx]
        return image, image  # input = output
```

Explicación:

- Hereda de `torch.utils.data.Dataset`, la clase base para definir datasets personalizados.
- `__init__`: guarda el dataset original como atributo interno.
- `__len__`: devuelve la cantidad total de imágenes.
- `__getitem__`: accede a la imagen número `idx`, ignora la etiqueta y retorna un par `(imagen, imagen)`.

9.3. Bloque 4: Carga de datos

```
# 4) Carga de datos
train_raw = datasets.FashionMNIST(root='./data', train=True, download=True,
    transform=transform)
valid_raw = datasets.FashionMNIST(root='./data', train=False, download=True,
    transform=transform)

train_set = CustomDataset(train_raw)
valid_set = CustomDataset(valid_raw)

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

9.3.1. Explicación breve

Este bloque de código prepara los datos necesarios para entrenar y validar un **autoencoder convolucional** usando el dataset `Fashion-MNIST`. Se descargan los datos, se transforman, se adaptan para autoencoder y se organizan en lotes. La separación en lotes permite entrenar de forma eficiente, y la adaptación del dataset asegura que las entradas y salidas sean iguales, como requiere el autoencoder.

9.3.2. Explicación ampliada

Descarga del dataset original

- `datasets.FashionMNIST(...)`: descarga el conjunto de imágenes de ropa en escala de grises.
- `train=True`: indica que se trata del conjunto de entrenamiento.
- `train=False`: indica que se trata del conjunto de validación (testeo).
- `transform=transform`: aplica la transformación definida previamente (por ejemplo, normalización a $[-1, 1]$).
- `download=True`: descarga automáticamente el dataset si no está presente en la carpeta `./data`.

Adaptación para autoencoder

- `CustomDataset(train_raw)`: convierte el dataset original en un conjunto de pares `(imagen, imagen)`.
- Esto es necesario porque el autoencoder no necesita etiquetas, sino aprender a reconstruir la imagen de entrada.

Organización en lotes

- `DataLoader(...)`: divide el dataset en lotes (batches) para entrenamiento eficiente.
- `batch_size=100`: cada lote contiene 100 imágenes.
- `shuffle=True`: mezcla aleatoriamente los datos en cada época (solo en entrenamiento).
- `shuffle=False`: mantiene el orden fijo en validación.

9.4. Bloque 5: Definición del modelo

```
# 5) Definición del autoencoder convolucional
class ConvAutoencoder(nn.Module):
    def __init__(self, latent_dim=128, dropout_p=0.2):
        super(ConvAutoencoder, self).__init__()

        # Encoder
        self.encoder = nn.Sequential(
            nn.Conv2d(1, 16, kernel_size=3),  # 28x28 -> 26x26
            nn.ReLU(),
            nn.MaxPool2d(2),                  # 26x26 -> 13x13
            nn.Dropout(p=dropout_p)
        )

        # Bottleneck
        self.bottleneck = nn.Sequential(
            nn.Flatten(),                     # 16x13x13 -> 2704
            nn.Linear(2704, latent_dim),
            nn.ReLU(),
            nn.Linear(latent_dim, 2704),
            nn.ReLU()
        )

        # Decoder
        self.decoder = nn.Sequential(
            nn.Unflatten(1, (16, 13, 13)),
            nn.ConvTranspose2d(16, 1, kernel_size=6, stride=2, padding=1),  #
                13x13 -> 28x28
```

```
            nn.Tanh()  # salida en [-1, 1]
        )

    def forward(self, x):
        z = self.encoder(x)
        z_latent = self.bottleneck(z)
        x_hat = self.decoder(z_latent)
        return x_hat
```

9.4.1. Explicación breve

Este bloque define una red neuronal **autoencoder convolucional** en PyTorch. Su tarea es aprender a reconstruir imágenes de entrada, comprimiéndolas en una representación latente y luego expandiéndolas nuevamente.

9.4.2. Explicación ampliada

Clase `ConvAutoencoder`

- Hereda de `nn.Module`, la clase base para modelos en PyTorch.
- El método `__init__` define la arquitectura.
- El método `forward` especifica cómo fluye la información.

Encoder

- `Conv2d`: aplica filtros para extraer características espaciales.
- `ReLU`: Aplica la función de activación ReLU, introduciendo no linealidad.
- `MaxPool2d`: reduce la resolución espacial.
- `Dropout`: apaga aleatoriamente neuronas para evitar posible sobreajuste.
- Resultado: imagen comprimida de tamaño `(16, 13, 13)`.

Bottleneck (cuello de botella)

- `Flatten`: convierte la imagen en un vector de dimensión 2704.
- `Linear`: reduce a una dimensión latente `n` (por ejemplo, 128).
- `ReLU`: Aplica la función de activación ReLU, introduciendo no linealidad.
- `Linear`: expande nuevamente a 2704.
- Resultado: vector listo para reconstrucción.

Decoder

- `Unflatten`: convierte el vector en un tensor de forma `(16, 13, 13)`.
- `ConvTranspose2d`: reconstruye la imagen original de tamaño `(1, 28, 28)`.
- `Tanh`: asegura que los valores estén en el rango $[-1, 1]$.

Método `forward`

- `x`: imagen de entrada.
- `z`: salida del encoder.
- `z_latent`: vector comprimido.
- `x_hat`: imagen reconstruida.

9.4.3. Cálculo del ancho de la imagen luego de un proceso de convolución

Cada vez que se aplica un filtro convolucional a una imagen *cuadrada*, el ancho de la imagen cambia según la siguiente expresión:

$$W_{\text{new}} = \left\lfloor \frac{W_{\text{ini}} - F + 2P}{S} \right\rfloor + 1,$$

donde todos los parámetros están expresados en número de píxeles:

- W_{ini}: ancho de la imagen de entrada,
- F: tamaño del filtro convolucional (se asume cuadrado),
- P: *padding* (relleno aplicado en los bordes),
- S: *stride* o paso, es decir, cuántos píxeles se desplaza el filtro en cada aplicación.

9.5. Bloque 6: Dispositivo de trabajo

```
# 6) Dispositivo
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = ConvAutoencoder(dropout_p=0.2).to(device)
```

9.5.1. Explicación breve

Selecciona automáticamente GPU si está disponible, o CPU en caso contrario, y luego transfiere el modelo al mismo. Esto permite aprovechar la aceleración por hardware si está disponible.

9.5.2. Explicación ampliada

Selección del dispositivo

- `torch.cuda.is_available()`: verifica si hay una GPU compatible con CUDA disponible.
- Si hay GPU, se selecciona `çuda"` como dispositivo.
- Si no hay GPU, se usa la CPU por defecto.
- El resultado se guarda en la variable `device`.

Transferencia del modelo al dispositivo

- `model = ConvAutoencoder(...)`: crea una instancia del modelo autoencoder.
- `.to(device)`: transfiere el modelo al dispositivo seleccionado (GPU o CPU).
- Esto es necesario para que tanto el modelo como los datos estén en el mismo lugar durante el entrenamiento.

9.5.3. ¿Por qué es importante?

- Entrenar en GPU puede ser hasta 10 veces más rápido que en CPU.
- PyTorch requiere que todos los tensores y modelos estén en el mismo dispositivo para operar correctamente.
- Este patrón hace que el código sea portátil y funcione en cualquier computadora.

9.6. Bloque 7: Función de pérdida y optimizador

```
# 7) Función de pérdida y optimizador
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=1e-3)
```

9.6.1. Explicación breve

Este bloque de código define dos elementos fundamentales para el entrenamiento de una red neuronal en PyTorch:

- La **función de pérdida**, que mide qué tan bien está funcionando el modelo.
- El **optimizador**, que ajusta los parámetros del modelo para minimizar dicha pérdida.

9.6.2. Explicación ampliada

Función de pérdida: `nn.MSELoss()`

- `MSE` significa **Mean Squared Error** o **Error Cuadrático Medio**.
- Compara la salida del modelo (`x_hat`) con la imagen original (`x_out`).
- Calcula la diferencia entre ambas y la eleva al cuadrado.
- Luego promedia todos los errores para obtener un único valor.
- Este valor indica qué tan bien el modelo está reconstruyendo las imágenes.

¿Por qué usamos `MSELoss` y no `CrossEntropyLoss`?

En el contexto de autoencoders, el objetivo del modelo es reconstruir la imagen de entrada lo más fielmente posible, píxel por píxel. Para ello, se compara la salida del modelo con la imagen original utilizando una métrica que mida la diferencia entre ambas. La función `MSELoss` (*Mean Squared Error*) es ideal para este propósito, ya que calcula el promedio del cuadrado de las diferencias entre los valores de cada píxel. Esta medida es continua y adecuada para salidas numéricas en el rango $[-1, 1]$ o $[0, 1]$, como ocurre en los autoencoders.

Por el contrario, la función `CrossEntropyLoss` está diseñada para tareas de clasificación, donde la salida del modelo representa probabilidades sobre clases discretas. Dado que un autoencoder no predice clases, sino reconstruye imágenes, `CrossEntropyLoss` no es apropiada en este caso. Usar `MSELoss` permite que el modelo aprenda una representación latente que minimiza directamente el error de reconstrucción entre la entrada y la salida.

Optimizador: `optim.Adam(...)`

- `Adam` es un algoritmo de optimización eficiente y ampliamente usado.
- Ajusta los parámetros del modelo para minimizar la función de pérdida.
- `model.parameters()`: indica que se deben optimizar todos los parámetros del modelo.
- `lr=1e-3`: establece la **tasa de aprendizaje** en 10^{-3}, que controla qué tan grandes son los pasos de ajuste.

9.7. Bloque 8: Funciones de entrenamiento y evaluación

```
# 8) Funciones de entrenamiento y evaluación
def train_epoch(model, loader, optimizer, criterion, device):
    model.train()
    running_loss = 0.0
    for x_in, x_out in loader:
        x_in, x_out = x_in.to(device), x_out.to(device)
        x_hat = model(x_in)
        loss = criterion(x_hat, x_out)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        running_loss += loss.item() * x_in.size(0)
    return running_loss / len(loader.dataset)

def evaluate(model, loader, criterion, device):
    model.eval()
    total_loss = 0.0
    with torch.no_grad():
        for x_in, x_out in loader:
            x_in, x_out = x_in.to(device), x_out.to(device)
            x_hat = model(x_in)
            loss = criterion(x_hat, x_out)
            total_loss += loss.item() * x_in.size(0)
    return total_loss / len(loader.dataset)
```

9.7.1. Explicación breve

Este bloque define dos funciones fundamentales para entrenar y evaluar un modelo autoencoder en PyTorch:

- `train_epoch`: realiza una pasada completa de entrenamiento sobre el conjunto de datos.
- `evaluate`: calcula la pérdida promedio sin modificar los parámetros del modelo.

Estas funciones permiten entrenar el autoencoder de forma eficiente y evaluar su rendimiento sin modificar sus parámetros. Son esenciales para construir el ciclo de entrenamiento por épocas y monitorear el progreso del modelo.

9.7.2. Explicación ampliada

`train_epoch`

- `model.train()`: activa el modo de entrenamiento (por ejemplo, habilita `Dropout`).
- Recorre los lotes del `DataLoader`:
  - Transfiere los tensores al dispositivo (`CPU` o `GPU`).

- Calcula la salida del modelo: `x_hat`.
- Calcula la pérdida entre `x_hat` y `x_out`.
- Reinicia los gradientes acumulados: `optimizer.zero_grad()`.
- Propaga el error hacia atrás: `loss.backward()`.
- Actualiza los parámetros del modelo: `optimizer.step()`.
- Acumula la pérdida total ponderada por el tamaño del lote.

- Devuelve la pérdida promedio sobre todo el conjunto.

`evaluate`

- `model.eval()`: activa el modo de evaluación (desactiva `Dropout`).
- `torch.no_grad()`: evita calcular gradientes (ahorra memoria y tiempo).
- Recorre los lotes del `DataLoader`:
  - Transfiere los tensores al dispositivo.
  - Calcula la salida del modelo.
  - Calcula la pérdida y acumula el total.
- Devuelve la pérdida promedio sobre todo el conjunto.

9.8. Bloque 9: Entrenamiento por épocas

```
# 9) Entrenamiento por épocas
num_epochs = 10
train_loss_during = []
train_loss_after = []
valid_loss_after = []

for epoch in range(num_epochs):
    loss_during = train_epoch(model, train_loader, optimizer, criterion, device
        )
    loss_train = evaluate(model, train_loader, criterion, device)
    loss_valid = evaluate(model, valid_loader, criterion, device)

    train_loss_during.append(loss_during)
    train_loss_after.append(loss_train)
    valid_loss_after.append(loss_valid)

    print(f"Época {epoch+1}: ECM durante = {loss_during:.4f}, ECM entrenamiento
        = {loss_train:.4f}, ECM validación = {loss_valid:.4f}")
```

9.8.1. Explicación breve

Este bloque ejecuta el ciclo de entrenamiento del modelo autoencoder durante varias **épocas**, registrando métricas de pérdida para analizar el progreso del aprendizaje. Es fundamental para analizar si el modelo está aprendiendo correctamente o si está sobreajustando.

9.8.2. Explicación ampliada

`num_epochs = 10`

- Define cuántas veces se recorrerá todo el conjunto de entrenamiento.
- Cada recorrido completo se llama una **época**.

Inicialización de listas

- `train_loss_during`: almacena el ECM calculado durante el entrenamiento.
- `train_loss_after`: almacena el ECM correcto sobre el conjunto de entrenamiento, calculado después de entrenar.
- `valid_loss_after`: almacena el ECM correcto sobre el conjunto de validación.

Ciclo de entrenamiento

- Para cada época:
  - Se entrena el modelo con `train_epoch`, y se guarda la pérdida promedio.
  - Se evalúa el modelo sobre el conjunto de entrenamiento con `evaluate`.
  - Se evalúa el modelo sobre el conjunto de validación con `evaluate`.
  - Se registran los tres valores de pérdida en sus respectivas listas.
  - Se imprime un resumen con los valores de ECM.

¿Por qué se calculan dos pérdidas sobre entrenamiento?

- `ECM durante entrenamiento`: se calcula mientras se actualizan los pesos, por lo tanto puede sobreestimar el valor de la pérdida.
- `ECM entrenamiento post`: se calcula después de entrenar, sin modificar el modelo, y refleja mejor el rendimiento real.

9.9. Bloque 10: Gráfico de pérdidas

```
# 10) Gráfico de pérdidas
plt.figure(figsize=(8, 5))
plt.plot(range(1, num_epochs+1), train_loss_during, label='ECM durante
    entrenamiento')
plt.plot(range(1, num_epochs+1), train_loss_after, label='ECM entrenamiento (
    post)')
plt.plot(range(1, num_epochs+1), valid_loss_after, label='ECM validación')
plt.xlabel('Época')
plt.ylabel('Error Cuadrático Medio')
plt.title('Curvas de pérdida: entrenamiento vs validación')
plt.legend()
plt.grid(True)
plt.show()
```

9.9.1. Explicación breve

Este bloque de código genera un gráfico que muestra cómo evoluciona el **Error Cuadrático Medio (ECM)** durante el entrenamiento del autoencoder. Este gráfico es una herramienta fundamental para monitorear el aprendizaje del modelo. Permite detectar problemas como sobreajuste, subentrenamiento o estancamiento, y tomar decisiones informadas sobre el número de épocas o la arquitectura del modelo.

9.9.2. Explicación ampliada

- `plt.figure(figsize=(8, 5))`: crea una figura de tamaño 8x5 pulgadas.
- `plt.plot(...)`: dibuja una curva para cada lista de pérdidas:
  - `train_loss_during`: pérdida promedio calculada durante el entrenamiento.
  - `train_loss_after`: pérdida promedio sobre el conjunto de entrenamiento, calculada después de entrenar.
  - `valid_loss_after`: pérdida promedio sobre el conjunto de validación.
- `range(1, num_epochs+1)`: genera los valores del eje X (épocas).
- `label=...`: asigna una etiqueta a cada curva para la leyenda.
- `plt.xlabel(...)` y `plt.ylabel(...)`: nombran los ejes.
- `plt.title(...)`: agrega un título al gráfico.
- `plt.legend()`: muestra la leyenda con las etiquetas de cada curva.
- `plt.grid(True)`: agrega una cuadrícula para facilitar la lectura.
- `plt.show()`: muestra el gráfico en pantalla.

¿Qué se puede observar en este gráfico?

- Si las curvas de entrenamiento y validación bajan juntas, el modelo está aprendiendo correctamente.
- Si la curva de validación se estabiliza o sube mientras la de entrenamiento sigue bajando, puede haber sobreajuste.
- Comparar `train_loss_during` con `train_loss_after` permite ver si hay mucha variabilidad durante el entrenamiento.

9.9.3. Resultados típicos esperados

9.10. Bloque 11: Visualización de reconstrucciones (*"desnormalizadas"*

```
# 11) Visualización de reconstrucciones (''desnormalizadas'')
model.eval()
imgs, _ = next(iter(valid_loader))
imgs = imgs.to(device)
with torch.no_grad():
    recon = model(imgs)

# Desnormalizar: x = 0.5 * tanh_output + 0.5
imgs_vis = 0.5 * imgs + 0.5
recon_vis = 0.5 * recon + 0.5

fig, axes = plt.subplots(2, 5, figsize=(12, 3))
for i in range(5):
    axes[0, i].imshow(imgs_vis[i][0].cpu(), cmap='gray')
    axes[0, i].axis('off')
    axes[1, i].imshow(recon_vis[i][0].cpu(), cmap='gray')
    axes[1, i].axis('off')
axes[0, 2].set_title("Originales", fontsize=12)
axes[1, 2].set_title("Reconstruidas", fontsize=12)
plt.suptitle("Reconstrucciones después del entrenamiento (normalización [-1,
    1])", fontsize=14)
```

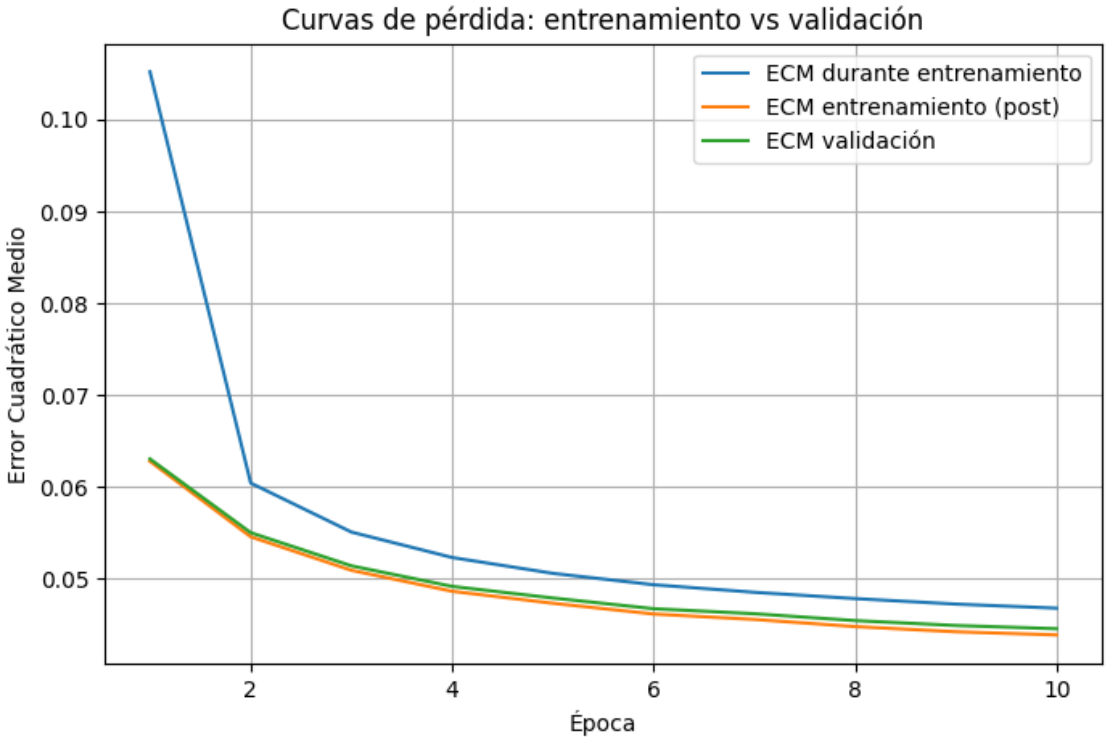

Figura 2: Curvas de pérdida por época. Se observa cómo evoluciona el Error Cuadrático Medio (ECM) durante el entrenamiento del autoencoder. La curva azul representa el ECM calculado durante la actualización de pesos, la curva naranja muestra el ECM sobre el conjunto de entrenamiento sin modificar el modelo, y la curva verde corresponde al ECM sobre el conjunto de validación. Este gráfico permite evaluar si el modelo está aprendiendo correctamente y detectar posibles signos de sobreajuste.

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

9.10.1. Explicación breve

Este bloque permite visualizar cómo el autoencoder reconstruye las imágenes de entrada después del entrenamiento. Se comparan las imágenes originales con sus reconstrucciones, desnormalizadas al rango $[0, 1]$ para poder graficarlas correctamente. Este bloque permite evaluar visualmente la calidad de las reconstrucciones generadas por el autoencoder. Comparar las imágenes originales con las reconstruidas ayuda a entender qué tan bien ha aprendido el modelo a representar y regenerar los datos.

9.10.2. Explicación ampliada

Evaluación del modelo

- `model.eval()`: pone el modelo en modo evaluación (desactiva `Dropout`, etc.).
- `next(iter(valid_loader))`: obtiene un lote de imágenes del conjunto de validación.
- `imgs.to(device)`: transfiere las imágenes al mismo dispositivo que el modelo (CPU o GPU).
- `with torch.no_grad()`: evita calcular gradientes, lo cual ahorra memoria y tiempo.
- `recon = model(imgs)`: genera las reconstrucciones de las imágenes.

Desnormalización

- Las imágenes fueron normalizadas a $[-1, 1]$ durante la carga.
- Para visualizarlas correctamente en escala de grises, deben transformarse de nuevo a $[0, 1]$.
- Esto se logra con la fórmula: `0.5 * imagen + 0.5`.
- Se aplica tanto a las imágenes originales como a las reconstruidas.

Visualización con `matplotlib`

- `plt.subplots(2, 5)`: crea una figura con 2 filas y 5 columnas de subgráficos.
- Fila 0: muestra las imágenes originales.
- Fila 1: muestra las imágenes reconstruidas por el autoencoder.
- `imshow(..., cmap='gray')`: muestra cada imagen en escala de grises.
- `axis('off')`: oculta los ejes para una visualización más limpia.
- `set_title(...)`: agrega títulos a las filas.
- `plt.suptitle(...)`: agrega un título general a la figura.
- `plt.tight_layout()`: ajusta automáticamente los márgenes.
- `plt.show()`: muestra el gráfico en pantalla.

9.10.3. Resultados típicos esperados

Figura 3: Comparación visual entre imágenes originales (fila superior) y sus reconstrucciones generadas por el autoencoder (fila inferior), después de entrenar el autoencoder convolucional durante 10 épocas. Las imágenes fueron desnormalizadas desde el rango $[-1, 1]$ al rango $[0, 1]$ para poder visualizarse correctamente en escala de grises. Esta figura permite evaluar la calidad de reconstrucción del modelo y observar si ha capturado correctamente las características visuales de los datos de entrada.