Manipulación de Datos en PyTorch II

Autor: Jorge García González (Universidad de Málaga)

Última Actualización: 1/10/2025

Asignatura: Programación para la Inteligencia Artificial

Antes de continuar con los modelos neuronales, debemos hacer un inciso para hablar otra vez de cómo gestionar los datos en PyTorch. Cuando entrenamos un modelo de aprendizaje profundo generalmente no tenemos el privilegio de poder ignorar la memoria. Tanto los modelos como los conjuntos de datos y los cálculos intermedios realizados durante el entrenamiento suelen requerir una cantidad no trivial de memoria y gestionarla bien (como hemos visto anteriormente) es determinante para que el entrenamiento sea eficiente. Además, hay ciertas manipulaciones de los datos que pueden ser convenientes durante el entrenamiento de un modelo, pero de eso hablaremos en el futuro.

Antes de hablar de ciertas herramientas de PyTorch para gestionar los datos durante el entrenamiento, vamos a entender unas funciones concretas de Python: los generadores.

Un generador es una función de Python que se comporta como un iterable eficiente. Veamos un ejemplo.

In [154…
```
iterable_lista = [0,1,2,3,4,5,6,7,8,9]
print(iterable_lista)
for n in iterable_lista:
  print(n)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
0
1
2
3
4
5
6
7
8
9
```

Ahí podemos ver la declaración explícita de una lista e iteramos sobre ella. A priori no hay problema con las listas. Sirven para almacenar información, poder acceder a ella, etc.

El "problema" es que las listas ocupan espacio. Suena obvio, pero es así. Si defino una lista de un millón de elementos, ese millón de elementos ocuparán espacio en memoria y en muchas ocasiones no sirve para nada tener ese millón de elementos en memoria a la vez porque el único uso que quiero hacer es iterarlos. Solo quiero usar los elementos de esa lista uno tras otro en un bucle bien definido. No voy a alterar el orden, añadir nuevos elementos, eliminar elementos, alterar los propios elementos... Solo tener acceso a uno tras otro.

Un generador es una función iterable que proporciona los elementos y los *genera* cuando se le pide sin tenerlos almacenados en memoria. Podríamos crear una clase para hacer eso, pero Python provee de una estructura muy simple para hacerlo. Un ejemplo:

In [155…
```
def firstn(n):
  num = 0
  while num < n:
    yield num
    num += 1

print(firstn(10))
for n in firstn(10):
  print(n)

generador = firstn(n)
print(f"Dame el siguiente: {next(generador)}")
print(f"Dame el siguiente: {next(generador)}")
print(f"Dame el siguiente: {next(generador)}")
```

```
<generator object firstn at 0x7f8ef81b2d40>
0
1
2
3
4
5
6
7
8
9
Dame el siguiente: 0
Dame el siguiente: 1
Dame el siguiente: 2
```

Una estructura simple, ¿verdad? La palabra clave **yield** le indica a la función que devuelva un valor, pero a diferencia de **return**, no termina la función. Si tenemos un **return** dentro de un bucle (decisión de programación MUY cuestionable), Python se limita a devolver lo indicado y dar por finalizada la función sin atender al estado del bucle. **yield** permite devolver lo indicado y mantener el bucle "a la espera" en segundo plano. Una vez el generador está creado, podemos usar la función de Python **next** para solicitarle el siguiente elemento.

En realidad llevamos usando generadores desde el principio. Es básicamente **range**.

Más información: https://wiki.python.org/moin/Generators

Ahora que hemos entendido la idea de generar los datos sobre la marcha, vamos a centrarnos en PyTorch.

Igual que en el uso de los generadores, al entrenar un modelo normalmente no necesitamos todo el conjunto de datos a la vez en memoria. Los entrenamientos funcionan por lotes (o *batch*), aunque de eso hablaremos un poco más adelante. Lo relevante ahora es que hace falta una herramienta para proveer los datos según se necesitan sin necesitar que estén todos cargados en memoria todo el tiempo. Para eso PyTorch tiene las clases *Dataset* y *DataLoader*.

Conceptualmente, el trabajo de la clase *Dataset* es cargar los datos de uno en uno y el de *DataLoader* hacer una gestión de esos datos adaptada al entrenamiento. Vamos a ver un ejemplo.

Para este ejemplo vamos a usar acceso a carpetas. Como este cuadernos e está realizando en Google colab, necesitamos montar en colab una carpeta de Google Drive para trabajar. Si se ejecuta en un entorno local, este paso se omitiría o cambiaría según necesidad.

In [156…
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).
```

Una vez montado en colab, definimos una ruta en la que trabajar y vamos a crear un conjunto de datos para resolver un hipotético problema en el que se relaciona una imagen con cuánto ruido uniforme tiene.

Vamos a importar algunas librerías útiles.

In [157…
```
working_path = '/content/drive/MyDrive/Work/Docencia UMA/2025-2026/Programacion para la IA/data'
import torch
from torchvision.io import decode_image
from torchvision.utils import save_image
from tqdm import tqdm
import csv
import os
```

In [158…
```
original_image_path = os.path.join(working_path, "Shin Chan/shin_chan_2.png")
original_image = decode_image(original_image_path, mode='RGBA')/255.                # Cargo la imagen original.
alpha = original_image[-1]                                                          # Separo el alfa.
original_image = original_image[:-1] + (1-alpha[None,:])                            # Me quedo solo con RGB y le
dataset_size = 100                                                                  # Defino el número de imáge
maximum_noise_range = 100                                                           # Defino el máximo de ruido
labels = []                                                                         # Inicializo la lista que a

for i in tqdm(range(dataset_size)):
  noise_range = torch.randint(low=0, high=maximum_noise_range, size=(1,))[0]        # Decido cuánto va a ser el
  noise = torch.rand_like(original_image)*noise_range/255.                          # Genero el ruido.
  image_with_noise = torch.clamp(original_image+noise*alpha, 0,1)                   # Añado el ruido a la imagen
  image_path = os.path.join(working_path, f"Shin Chan/images/shin_chan_{i}.png")    # Creo el path de la imagen
  save_image(image_with_noise, image_path)                                          # Guardo la imagen desnormal
  labels.append(int(noise_range))                                                   # Añado la etiqueta a la lis

# Guardo la lista como un csv.
with open(os.path.join(working_path, f"Shin Chan/labels.csv"), mode='w', newline='') as csvfile:
  f = csv.writer(csvfile, quoting=csv.QUOTE_ALL)
  f.writerow(labels)
```

```
100%|██████████| 100/100 [00:13<00:00,  7.52it/s]
```

Una vez con un conjunto de datos que manipular, podemos ver para qué sirve la clase *Dataset*.

Como hemos dicho, su objetivo es cargar los datos de uno en uno. Sirve para enmascarar todo el proceso para lidiar con los archivos. Vamos a importar la clase base.

In [159…
```
from torch.utils.data import Dataset
```

Un *Dataset* de PyTorch tiene que tener implementdos tres metodos: *__init__* (el constructor), *__len__* para que devuelva su tamaño cuando se use la función **len** y *__getitem__* para solicitar un elemento.

Nota: que los métodos tengan nombres con guiones bajos es una manera de señalar que esos métodos no están pensados para ser llamados directamente por quien use la clase, sino que son utilidades intermedias.

Dentro de esos métodos uno puede hacer lo que considere oportuno. Típicamente en el constructor se guarda la información necesaria para trabajar. Un *Dataset* para nuestro conjunto de datos podría ser el siguiente.

In [164…
```
class ShinChanDataset(Dataset):
  def __init__(self, images_path, labels_path, transform=None, target_transform=None):
    self.images_path = images_path
    self.labels_path = labels_path
    self.transform = transform
    self.target_transform = target_transform

    # Vamos a cargar las etiquetas en el constructor. No ocupan mucho en memoria y
    # ahorramos tiempo de acceso a disco duro durante la ejecución.
    with open(self.labels_path, newline='') as csvfile:
      f = csv.reader(csvfile)
      self.labels = [int(l) for l in list(f)[0]]

  def __len__(self):
    # La longitud del Dataset coincide con el número de etiquetas.
    return len(self.labels)

  def __getitem__(self, idx):
    img_path = os.path.join(self.images_path, f"shin_chan_{idx}.png")
    image = decode_image(img_path)
    label = self.labels[idx]
    #Si hay transformaciones, las aplico.
    if self.transform:
      image = self.transform(image)
    if self.target_transform:
      label = self.target_transform(label)
    return image, label
```

Obviando el argumento *self* necesario en los métodos de Python, al constructor se le pueden alterar los argumentos según sea necesario y no devuelve nada, *__len__* es un método que no tiene argumentos y devuelve un entero (se asume que 0 o positivo), mientras que *__getitem__* tiene como argumento el índice del elemento a devolver y devuelve una tupla con el dato y su etiqueta.

Los transform que se utilizan son transformaciones. Una convención de PyTorch para referirise a las funciones que transforman la información. Por ahora no vamos a usarlas y podrían ser omitidas, pero mejor si nos acostumbramos a verlas.

Vamos a crear una instancia de nuestro *Dataset* y a ver si funciona.

In [165…
```
shin_chan_dataset = ShinChanDataset(
    os.path.join(working_path, "Shin Chan",  "images"),
    os.path.join(working_path, "Shin Chan", "labels.csv")
)
print(len(shin_chan_dataset))
print(shin_chan_dataset.__getitem__(0))
```

```
100
(tensor([[[255, 255, 255,  ..., 255, 255, 255],
         [255, 255, 255,  ..., 255, 255, 255],
         [255, 255, 255,  ..., 255, 255, 255],
         ...,
         [255, 255, 255,  ..., 255, 255, 255],
         [255, 255, 255,  ..., 255, 255, 255],
         [255, 255, 255,  ..., 255, 255, 255]],

        [[255, 255, 255,  ..., 255, 255, 255],
         [255, 255, 255,  ..., 255, 255, 255],
         [255, 255, 255,  ..., 255, 255, 255],
         ...,
         [255, 255, 255,  ..., 255, 255, 255],
         [255, 255, 255,  ..., 255, 255, 255],
         [255, 255, 255,  ..., 255, 255, 255]],

        [[255, 255, 255,  ..., 255, 255, 255],
         [255, 255, 255,  ..., 255, 255, 255],
         [255, 255, 255,  ..., 255, 255, 255],
         ...,
         [255, 255, 255,  ..., 255, 255, 255],
         [255, 255, 255,  ..., 255, 255, 255],
         [255, 255, 255,  ..., 255, 255, 255]]], dtype=torch.uint8), 30)
```

Todo parece funcionar. Tenemos un objeto que nos devuelve los datos según se los pedimos. Una vez con un *Dataset*, podemos aprovechar la clase *Dataloader*.

In [162…
```
from torch.utils.data import DataLoader
```

Como hemos dicho, el *DataLoader* nos permite hacer una gestión de los conjuntos de datos de alto nivel. Se encarga de "organizar" los datos que el *Dataset* proporciona. Vamos a ver qué hace con nuestro *Dataset* si lo recorremos. ¡*DataLoader* también es un iterable!

In [167…
```
dataloader = DataLoader(shin_chan_dataset, batch_size=64, shuffle=True)
for i, batch in enumerate(dataloader):
  print(f"Iteración {i}")
  print(f"Tipo de batch: {type(batch)}")
  print(f"Longitud de batch: {len(batch)}")
  print(f"Tamaño de batch[0]: {batch[0].shape}")
  print(f"Tamaño de batch[1]: {batch[1].shape}")
```

```
Iteración 0
Tipo de batch: <class 'list'>
Longitud de batch: 2
Tamaño de batch[0]: torch.Size([64, 3, 700, 600])
Tamaño de batch[1]: torch.Size([64])
Iteración 1
Tipo de batch: <class 'list'>
Longitud de batch: 2
Tamaño de batch[0]: torch.Size([36, 3, 700, 600])
Tamaño de batch[1]: torch.Size([36])
```

Vamos a ver qué está pasando aquí por partes. Primero observemos la llamada al constructor. Como podemos ver, lo primero que le proporcionamos como argumento es un *Dataset*. De ahí obtendrá los datos. El segundo argumento es el *batch_size* que determinará el tamaño de los lotes que irá generando el *DataLoader*. Por último con *shuffle* le hemos indicado que ordene aleatoriamente los datos cada *época*. Ahora vamos a explicar qué significa esto.

El trabajo del *DataLoader* es ir proporcionando al bucle de entrenamiento los datos con los que realizar el *forward* y posterior *backward* en cada iteración. Los datos de cada una de esas iteraciones es lo que entendemos por un *batch*. Como podemos ver en la ejecución, el *batch* (no nos engañemos, nadie lo llama lote) es una lista de tamaño 2. El primer elemento es un tensor con todos los inputs (el primer elemento de la tupla que devuelve el *Dataset*) y el segundo un tensor con todas las etiquetas (el segundo elemento que devuelve el *Dataset*). Alguno de ellos podría ser otro iterable si PyTorch no los identificara con un tipo apto para hacer un tensor.

Es interesante ver que en la primera iteración el *batch* tiene 64 datos y en la segunda tiene solo 36. Esto se debe simplemente a que nuestro conjunto de datos tiene 100 elementos (64+36=100). Cuando iteramos el *DataLoader* devuelve un *batch* tras otro hasta haber devuelto todos los elementos del conjunto de datos. A recorrer todos los elementos del conjunto de datos es lo que llamamos una *época* de entrenamiento y el argumento *shuffle* del *DataLoader* le indica que cada vez que inice una nueva época, devuelva los datos en un orden distinto.

En el siguiente cuaderno hablaremos de la importancia del tamaño del *batch* y que sea aleatorio. *DatasLoader* tiene otros argumentos que pueden ser relevantes desde el punto de vista de su eficiencia a la hora de acceder a los datos, pero ya hablaremos de ellos en el futuro si hay tiempo.