Conjuntos de datos y cargadores de datos

Roberto López Castro

Diego Andrade Canosa

Índice

- Introducción
- Datasets en Pytorch
- Dataloaders en Pytorch
- Transformaciones
- Data Augmentation
- Técnicas avanzadas de carga de datos
- Manejo de datasets desbalanceados
- Distributed Data Parallelism
- Aplicaciones del mundo real

citic

Introducción a los Conjuntos de Datos

- (Pytorch) Colección de muestras de datos que se utilizan para entrenar o evaluar un modelo de ML.
- Contiene tanto los datos de entrada como las etiquetas correspondientes (si procede) para cada muestra.
- En PyTorch están representados por la clase ***torch.utils.data.Dataset***

- PyTorch proporciona una amplia variedad de conjuntos de datos predefinidos, como MNIST, CIFAR-10, ImageNet, etc.
- Es posible crear conjuntos de datos personalizados para adaptarse a necesidades específicas.

Estructura de un Conjunto de Datos

- Dos elementos principales: los datos de entrada y las etiquetas correspondientes.
- Los datos de entrada pueden ser imágenes, texto, señales de audio, etc., dependiendo del problema y el tipo de datos que estemos tratando.
- Las etiquetas representan las salidas deseadas o las clases a las que pertenecen los datos de entrada.

- En PyTorch se representan como una clase que hereda de ***torch.utils.data.Dataset*** y proporciona implementaciones para los métodos *__len__()* y *__getitem__()*.
  - *__len__()* devuelve la longitud del conjunto de datos (número total de muestras).
  - *__getitem__()* se utiliza para acceder a una muestra específica y su etiqueta utilizando un índice.

C

- P
  p
  a
- Algunos ejemplos de conjuntos de datos predefinidos en PyTorch son:
  - MNIST: Un conjunto de imágenes de dígitos escritos a mano, utilizado para reconocimiento de dígitos.
  - CIFAR-10: Un conjunto de imágenes en color de 10 clases diferentes, utilizado para clasificación de imágenes.
  - ImageNet: Un conjunto de imágenes de alta resolución clasificadas en miles de categorías, utilizado en desafíos de reconocimiento de imágenes.

Creación de un Conjunto de Datos Personalizado

• En PyTorch, también tenemos la flexibilidad de crear nuestros propios conjuntos de datos personalizados.

• Esto puede ser útil cuando tenemos datos en un formato específico o cuando queremos combinar diferentes fuentes de datos.

• Para crear un conjunto de datos personalizado, necesitamos crear una clase que herede de ***torch.utils.data.Dataset*** y proporcionar implementaciones para los métodos *___len___()* y *___getitem___()*.

Creación de un Conjunto de Datos Personalizado

```python
import torch
from torch.utils.data import Dataset

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

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

    def __getitem__(self, index):
        sample = self.data[index]
        label = self.labels[index]

        return sample, label

data = [...]  # Datos de entrada
labels = [...]  # Etiquetas correspondientes

custom_dataset = CustomDataset(data, labels)

# Acceder a una muestra específica y su etiqueta
sample, label = custom_dataset[0]
```

Introducción a los DataLoaders en PyTorch

- En PyTorch, los **DataLoaders** nos permiten cargar y gestionar eficientemente conjuntos de datos durante el entrenamiento de nuestros modelos.

- PyTorch proporciona la clase ***torch.utils.data.DataLoader*** para crear **DataLoaders** de manera sencilla.
- Al utilizar un **DataLoader**, podemos iterar sobre los datos en lotes, lo que facilita el procesamiento por lotes durante el entrenamiento.

Configuraciones comunes del DataLoader

```
# Configuraciones comunes del DataLoader
batch_size = 64
shuffle = True
num_workers = 4
drop_last = False  # Descartar el último lote incompleto

dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=shuffle,
num_workers=num_workers, drop_last=drop_last)
```

- El DataLoader ofrece va
  adaptarlo a nuestras nec
- Algunas configuraciones
  - Tamaño del lote (batch s
    cada lote que se utiliza pa
  - Aleatorización: Permite mezclar los datos en cada época (epoch) para evitar el sesgo del orden de los datos.
  - Multiprocesamiento: Habilita la carga de datos en paralelo utilizando múltiples hilos de ejecución para acelerar el proceso de carga.
  - Último lote incompleto: Maneja automáticamente el último lote que puede tener un tamaño menor al tamaño del lote especificado.

Iteración sobre un DataLoader

- Una vez que tenemos un **DataLoader**, podemos iterar sobre él en un bucle para obtener los lotes de datos para el entrenamiento.
- En cada iteración, el **DataLoader** nos devuelve un lote de datos.
- Podemos acceder a las muestras de datos y sus etiquetas correspondientes utilizando la sintaxis de desempaquetado.
- Podemos realizar operaciones en cada lote, como pasar los datos a un modelo para la propagación hacia adelante y el cálculo de pérdidas.

Creando un DataLoader en PyTorch

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

# Crear un DataLoader
dataset = CustomDataset(data, labels)  # Suponiendo que tienes un conjunto de datos personalizado
batch_size = 32
shuffle = True  # Aleatorizar las muestras
num_workers = 4  # Utilizar 4 hilos de ejecución para la carga de datos

dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=shuffle, num_workers=num_workers)

# Iterar sobre el DataLoader
for batch in dataloader:
    inputs, labels = batch
    # Realizar operaciones en el lote de datos, como pasarlos a un modelo para la propagación hacia adelante
    outputs = model(inputs)
    loss = criterion(outputs, labels)
    # Realizar el cálculo de pérdida y otras operaciones de entrenamiento
```

citic

Técnicas Avanzadas de Carga de Datos

- Además de los DataLoaders básicos, PyTorch ofrece algunas técnicas avanzadas de carga de datos.
- Estas técnicas permiten una mayor personalización y flexibilidad en el manejo de los conjuntos de datos.

Batch Sampling

- El muestreo de lotes personalizado se puede lograr mediante la implementación de la clase Sampler en PyTorch.
- Esto permite controlar cómo se seleccionan y ordenan los lotes de datos durante el entrenamiento.
- Es útil cuando se tienen requisitos especiales de orden o agrupamiento de los datos.

Data Shuffling

- El mezclado de datos es una técnica importante para evitar el sesgo en el orden de los datos de entrenamiento.
- PyTorch proporciona la opción shuffle en el DataLoader para mezclar aleatoriamente los datos en cada época de entrenamiento.
- Esto ayuda a garantizar una distribución más equitativa y aleatoria de los datos en cada lote.

Data Loading en Dispositivos Múltiples

- Cuando se trabaja con dispositivos múltiples, como GPUs, se puede utilizar la clase DataParallel para facilitar la carga de datos en paralelo en cada dispositivo.
- Esto ayuda a optimizar la eficiencia y el rendimiento de la carga de datos al distribuir la carga en diferentes dispositivos.
- La clase DataParallel se encarga automáticamente de dividir los lotes de datos y enviarlos a los dispositivos correspondientes.

- DataParallel != Distributed Data Parallel (a continuación)
- single-process multi-thread (a wrapper of scatter + paralllel_apply + gather) vs. multi-process parallelism (different machines)

Carga Eficiente de Datos

- Utiliza la opción *num_workers* en el **DataLoader** para cargar datos en paralelo en múltiples subprocesos.
- Ajusta el tamaño del lote (*batch size*) para equilibrar la utilización de la memoria y el rendimiento del modelo.
- Si es posible, almacena los datos en una unidad de almacenamiento rápida, como un SSD, para acelerar la carga de datos.

Transformaciones en PyTorch

- En PyTorch, las transformaciones son operaciones aplicadas a los datos para modificarlos o prepararlos antes de utilizarlos en el modelo.

- Pueden incluir el reescalado de las imágenes, la normalización de los valores, la aplicación de técnicas de aumento de datos, entre otros.
- PyTorch proporciona la clase ***torchvision.transforms*** que ofrece una amplia variedad de transformaciones comunes para trabajar con conjuntos de datos de imágenes.
- Las transformaciones se pueden aplicar tanto a nivel de conjunto de datos como a nivel de muestra.

Transformaciones en el Conjunto de Datos

```python
import torchvision.transforms as transforms
from torchvision.datasets import CIFAR10

# Definir las transformaciones a aplicar en el conjunto de datos
transform = transforms.Compose([
    transforms.Resize((32, 32)),  # Reescalado a tamaño 32x32
    transforms.ToTensor(),  # Conversión de la imagen a un tensor
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))  # Normalización de
    los valores de los canales
])

# Cargar el conjunto de datos con las transformaciones aplicadas
dataset = CIFAR10(root='./data', train=True, transform=transform, download=True)
```

- Podemos aplicar transform
  datos utilizando la clase ***to***
  transform.
- Las transformaciones se aplican al momento de cargar los datos del conjunto de datos.
- Al aplicar transformaciones al conjunto de datos, podemos realizar operaciones como reescalado, recorte, normalización, etc., en todas las muestras del conjunto de datos.

Transformaciones en el Conjunto de Datos

Geometry

`Resize`(size[, interpolation, max_size, ...])	Resize the input image to the given size.
`v2.Resize`(size[, interpolation, max_size, ...])	[BETA] Resize the input to the given size.
`v2.ScaleJitter`(target_size[, scale_range, ...])	[BETA] Perform Large Scale Jitter on the input according to "Simple Copy-Paste is a Strong Data Augmentation Method for Instance Segmentation".
`v2.RandomShortestSize`(min_size[, max_size, ...])	[BETA] Randomly resize the input.
`v2.RandomResize`(min_size, max_size[, ...])	[BETA] Randomly resize the input.
`RandomCrop`(size[, padding, pad_if_needed, ...])	Crop the given image at a random location.
`v2.RandomCrop`(size[, padding, ...])	[BETA] Crop the input at a random location.
`RandomResizedCrop`(size[, scale, ratio, ...])	Crop a random portion of image and resize it to a given size.
`v2.RandomResizedCrop`(size[, scale, ratio, ...])	[BETA] Crop a random portion of the input and resize it to a given size.

https://pytorch.org/vision/stable/transforms.html

Transformaciones en la Muestra

- También podemos aplicar transformaciones a nivel de muestra utilizando la clase ***torchvision.transforms***.
- Las transformaciones a nivel de muestra se aplican durante el proceso de obtención de la muestra utilizando el método *___getitem___()* de la clase del conjunto de datos.
- Esto nos permite aplicar transformaciones específicas a cada muestra individualmente.
- Las transformaciones a nivel de muestra pueden incluir la conversión de la imagen a tensores, la normalización de los valores, la aplicación de técnicas de aumento de datos, entre otros.

Transformaciones en la Muestra

```python
import torchvision.transforms as transforms
from torchvision.datasets import CIFAR10

class CustomDataset(Dataset):
    def __init__(self, data, labels, transform=None):
        self.data = data
        self.labels = labels
        self.transform = transform

    def __getitem__(self, index):
        sample = self.data[index]
        label = self.labels[index]

        if self.transform:
            sample = self.transform(sample)

        return sample, label

# Definir las transformaciones a aplicar en la muestra
transform = transforms.Compose([
    transforms.RandomHorizontalFlip(),  # Volteo horizontal aleatorio
    transforms.RandomCrop(32, padding=4),  # Recorte aleatorio de tamaño 32x32
    con relleno de 4 píxeles
    transforms.ToTensor(),  # Conversión de la imagen a un tensor
])

# Crear una instancia del conjunto de datos personalizado con las
transformaciones aplicadas
dataset = CustomDataset(data, labels, transform=transform)
```

Combinación de Transformaciones

- En PyTorch, podemos combi
  aplicarlas secuencialmente a
- Podemos utilizar la clase ***tor***
  para combinar transformaci
  compuesta.
- Las transformaciones se apli
  especifican en la transforma
- Al combinar transformaciones, podemos construir flujos de transformación personalizados que se ajusten a nuestras necesidades específicas.

```python
import torchvision.transforms as transforms

# Definir transformaciones individuales
resize_transform = transforms.Resize((256, 256))
crop_transform = transforms.RandomCrop(224)
to_tensor_transform = transforms.ToTensor()

# Combinar las transformaciones en una transformación compuesta
composed_transform = transforms.Compose([
    resize_transform,
    crop_transform,
    to_tensor_transform
])

# Aplicar la transformación compuesta a una muestra de datos
transformed_sample = composed_transform(sample)
```

Aplicación de Transformaciones en un DataLoader

- Podemos aplicar transformaciones a un **DataLoader** utilizando el parámetro *transform* al crear la instancia del **DataLoader**.
- Las transformaciones se aplican al momento de cargar los datos en el **DataLoader**.
- Esto nos permite aplicar transformaciones personalizadas a los datos en cada lote mientras se cargan en el **DataLoader**.

Beneficios de las Transformaciones en PyTorch

- Realizar operaciones de preprocesamiento y aumento de datos de manera eficiente.
- Preparar los datos antes de introducirlos en el modelo.
- Aplicar técnicas de aumento de datos, como la rotación, el recorte, el cambio de brillo, etc., para enriquecer el conjunto de datos y mejorar la generalización del modelo.
- Son flexibles y personalizables, lo que nos permite adaptarlas a nuestras necesidades específicas.

Carga y Transformación de Datos

- Lab 1: 01_dataloaders.ipynb
- Lab 2: 01_transformations.ipynb

Utilizar Transformaciones Adecuadas

- Asegúrate de elegir las transformaciones adecuadas para tus datos y tareas específicas.
- Considera las transformaciones de reescalado, recorte, volteo, rotación, normalización y otras según sea necesario.
- Experimenta con diferentes transformaciones para mejorar el rendimiento y la precisión del modelo.

Visualización y Análisis de Datos

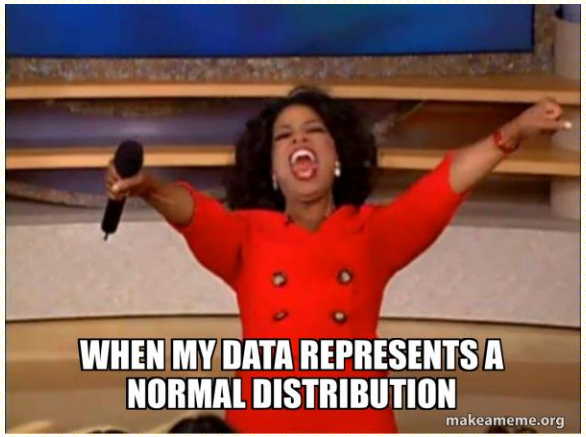
- Visualiza algunas muestras de datos para comprender mejor su formato, distribución y características.
- Utiliza bibliotecas como matplotlib, seaborn o tensorboard para trazar histogramas, gráficos de dispersión, etc.
- Analiza las estadísticas de los datos, como la media, la desviación estándar, para ajustar las transformaciones o la normalización.

Verificación de los DataLoaders

- Verifica que los **DataLoaders** estén funcionando correctamente antes de comenzar el entrenamiento o la evaluación.
- Itera sobre los **DataLoaders** y visualiza algunas imágenes y etiquetas para asegurarte de que los datos se carguen correctamente.
- Realiza un seguimiento de las estadísticas clave, como el tamaño del lote y el número de clases, para confirmar la coherencia de los datos -> Data augmentation + balanceo de datos

Data Augmentation

- Data augmentation es una técnica comúnmente utilizada para aumentar la cantidad y variedad de datos de entrenamiento.
- Consiste en aplicar transformaciones aleatorias a las imágenes durante el entrenamiento para introducir variabilidad.
- Esto ayuda a mejorar la generalización del modelo y a reducir el sobreajuste.



citic

Tipos de Data Augmentation

- Existen diferentes tipos de data augmentation que se pueden aplicar a las imágenes, como:
  - Rotación: rotar la imagen en un ángulo aleatorio.
  - Volteo horizontal: voltear la imagen horizontalmente.
  - Cambio de brillo, contraste y saturación: ajustar los niveles de brillo, contraste y saturación de la imagen.
  - Recorte y redimensionamiento aleatorio: recortar y redimensionar la imagen a tamaños aleatorios.
  - Ruido aleatorio: agregar ruido aleatorio a la imagen.
  - ...

Implementación de Data Augmentation en PyTorch

- En PyTorch, la data augmentation se puede implementar utilizando la clase *transforms* de la biblioteca *torchvision*.
- Puedes combinar y encadenar diferentes transformaciones para crear una secuencia de data augmentation.
- Asegúrate de aplicar la data augmentation **solo en el conjunto de datos de entrenamiento** y no en el conjunto de datos de prueba o validación.

Ejemplo de Data Augmentation en PyTorch

- Lab3: 01_augmentation.ipynb

Manejo de Conjuntos de Datos Desbalanceados

- En muchas aplicaciones del mundo real, los conjuntos de datos pueden estar desbalanceados, lo que significa que hay una diferencia significativa en el número de muestras entre las diferentes clases.
- Esto puede llevar a un sesgo en el entrenamiento del modelo hacia las clases dominantes y afectar su rendimiento en las clases minoritarias.

Métodos de Manejo de Datos Desbalanceados

- Oversampling: aumentar la cantidad de muestras en las clases minoritarias mediante duplicación o generación sintética de datos.
- Undersampling: reducir la cantidad de muestras en las clases dominantes al eliminar o submuestrear datos.
- Uso de pesos de clase: asignar pesos diferentes a las clases durante el entrenamiento para equilibrar su influencia en la función de pérdida.

Implementación en PyTorch

- Oversampling y Undersampling: se pueden aplicar transformaciones personalizadas a los conjuntos de datos o utilizar bibliotecas como i*mbalanced-learn*.
- Uso de pesos de clase: se pueden especificar los pesos de clase al calcular la función de pérdida o utilizar la opción *class_weight* en algunas funciones de pérdida incorporadas de PyTorch.

Implementación en PyTorch

- Lab4: 01_unbalanced.ipynb

Distributed Data Parallelism

- Distributed Data Parallelism es una técnica avanzada para entrenar modelos en PyTorch en múltiples GPUs o nodos de manera eficiente.
- Permite distribuir el modelo y los datos entre varios dispositivos para acelerar el entrenamiento y mejorar el rendimiento.

Beneficios de Distributed Data Parallelism

- Distribuir el entrenamiento en varias GPUs o nodos tiene varios beneficios:
- Mayor velocidad de entrenamiento al aprovechar el poder de procesamiento de múltiples dispositivos.
- Capacidad para entrenar modelos más grandes que no cabrían en una sola GPU.
- Mejor escalabilidad al utilizar múltiples recursos de hardware para el entrenamiento.

Configuración de Distributed Data Parallelism

- Para utilizar Distributed Data Parallelism en PyTorch:

1. Configurar el entorno de ejecución para la distribución, como definir el número de GPUs o nodos disponibles.
2. Crear una instancia del modelo y envolverlo con la clase *torch.nn.parallel.DistributedDataParallel*.
3. Configurar la inicialización y sincronización entre los procesos distribuidos.
4. Dividir los datos y las tareas de entrenamiento entre los dispositivos utilizando el **DataLoader** y la función de entrenamiento personalizada.

Ejemplo de Uso de Distributed Data Parallelism

- Lab5: 01_DDP.ipynb
- Más en profundidad en el siguiente curso de Pytorch

Aplicaciones del Mundo Real

- PyTorch y sus capacidades de manipulación de conjuntos de datos y carga de datos son ampliamente utilizadas en diversas aplicaciones del mundo real.
- Aquí presentamos algunos ejemplos de cómo PyTorch se utiliza en diferentes dominios:

Aplicación 1: Visión por Computadora

- En visión por computadora, PyTorch se utiliza para entrenar y desplegar modelos de reconocimiento de objetos, detección de objetos, segmentación semántica, entre otros.
- PyTorch permite cargar grandes conjuntos de datos de imágenes y aplicar transformaciones y técnicas de aumento de datos para mejorar el rendimiento del modelo.

Aplicación 2: Procesamiento del Lenguaje Natural (NLP)

- En el procesamiento del lenguaje natural, PyTorch es ampliamente utilizado para tareas como clasificación de texto, generación de texto, traducción automática y análisis de sentimientos.
- PyTorch permite cargar y procesar conjuntos de datos de texto, aplicar técnicas de tokenización y atención, y entrenar modelos de lenguaje con arquitecturas complejas como Transformers.

Aplicación 3: Aprendizaje por Refuerzo

- En el aprendizaje por refuerzo, PyTorch se utiliza para entrenar agentes que toman decisiones secuenciales en entornos dinámicos.
- PyTorch proporciona herramientas para construir modelos de agentes de aprendizaje profundo, como redes neuronales y funciones de pérdida personalizadas, y permite cargar y procesar datos de transiciones de estados y recompensas.