机器学习编程课笔记

2025 年 8 月 5 日

1 Dataset 与 DataLoader 的基本实现

在使用 PyTorch 进行深度学习任务时，`Dataset` 和 `DataLoader` 是数据加载的核心组件。其作用包括两个方面：一是确定采样范围，即定义整个数据集中样本的数量及其索引方式；二是通过按 `batch` 批次进行采样，实现高效训练流程。通常我们会从本地按文件夹结构组织的方式加载图像数据，每个子文件夹代表一个类别标签。在实际训练中，`DataLoader` 中的 `collate_fn` 函数扮演着关键角色，负责决定如何从 `Dataset` 中采样数据，并将其打包成可直接供模型使用的 batch 结构。在 PyTorch 中，自定义数据集需要继承 `torch.utils.data.Dataset` 类，并实现 `__len__` 与 `__getitem__` 方法。以下是一个图像分类任务的数据集定义示例：

1.1 图像数据集实现：my_dataset 类

```
class my_dataset(Dataset):
    def __init__(self, path, preprocess):
        self.preprocess = preprocess
        self.image_paths = []
        self.labels = []
        label_list = os.listdir(path)
        for label in label_list:
            image_folder = os.path.join(path, label)
            for file_names in os.listdir(image_folder):
                if file_names.endswith(("png", "jpg", "jpeg")):
                    self.image_paths.append(os.path.join(image_folder,
                        file_names))
                    self.labels.append(label_list.index(label))
```

Listing 1: my_dataset 类

该类用于从本地文件夹结构中按类别加载图像路径与标签，预处理函数 `preprocess` 用于对图像进行标准化等变换。

```
    def __len__(self): # 采样范围
        return len(self.image_paths)

    def __getitem__(self, item): # 如何采样
```

```
        image = Image.open(self.image_paths[item])
        image = self.preprocess(image)
        label = self.labels[item]
        return image, label
```

`__len__` 返回样本数量，`__getitem__` 定义了采样方式。

1.2 文本任务数据集：llmCustomDataset 类

`llmCustomDataset` 用于 LLM 类模型的训练数据生成，构建包含多个子任务的输入样本。在深度学习中，一个 `Dataset` 对象的每一个样本通常是一个 `tuple`，其中包括输入数据和对应的标签信息。以图像分类任务为例，一个典型的样本结构为 `(image, label)`，其中：

- **image**: 是一个 `torch.Tensor` 类型的数据张量，其维度为 (C, H, W)，例如在 CIFAR-100 数据集中为 $(3, 32, 32)$，代表 3 个颜色通道、32×32 像素大小的彩色图像；
- **label**: 是一个整数（`int` 类型），表示图像所属的类别编号。

当使用 `DataLoader` 进行 batch 加载时，`collate_fn` 函数会自动将这些样本组合成批量数据。其作用是：对一组样本 `[(image1, label1), (image2, label2), (image3, label3)]`，分别将图像组成一个张量 batch，即 `collate_fn([image1, image2, image3])`，将标签组成一个 batch，即 `collate_fn([label1, label2, label3])`，最终返回一个二元组 `(batched_images, batched_labels)`。

在更复杂的任务中，如基于语言模型的文本生成，还可以设计更灵活的 Dataset 结构。例如，在下游多标签任务中（如 ImageNet 的子集分类），可以从完整的 `label_list` 中随机选择一个包含 k 个类的子集，构造为一个小任务（sub-task）。此时每个 sub-task 内的每个目标类别（label）都通过多个示例句子（例如 3 个）作为提示进行引导生成。

这样的设计可用于构造 few-shot 的输入样本，其结构为：

- 每个样本中包含多个 `prompt-label` 对；
- 每条 prompt 包含 3 个示例（即 3-shot），格式如下：
  `object1 => description1; object2 => description2; object3 => description3; target =>`

```
class llmCustomDataset(Dataset):
    def __init__(self, label_list, example, k_subset=10, shuffle_nums=1):
        self.data = []
        for _ in range(shuffle_nums):
            labels = random.sample(label_list, len(label_list)) # 打乱
            sub_tasks = [labels[i: i + k_subset] for i in range(0, len(
                labels), k_subset)]
            for sub_task in sub_tasks:
                prompts = []
```

```
                for label in sub_task:
                    chosen_idx = random.sample(range(len(example)), 3)
                    current_prompt = """Given an object category, Generate
                        one sentence about an image description: {} => {};{}
                         => {};{} => {};{} =>""".format(
                         example[chosen_idx[0]][0], example[chosen_idx
                            [0]][1],
                         example[chosen_idx[1]][0], example[chosen_idx
                            [1]][1],
                         example[chosen_idx[2]][0], example[chosen_idx
                            [2]][1],
                         label,
                    )
                    prompts.append(current_prompt)
                self.data.append({"prompts": prompts, "labels": sub_task})
```

Listing 2: llmCustomDataset 类

该数据集用于构造 **few-shot prompting** 格式样本，其中每个 prompt 带有 3 个例子。

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

    def __getitem__(self, idx):
        return self.data[idx]
```

1.3 collate_fn 机制

PyTorch 中的 `DataLoader` 可以使用 `collate_fn` 参数来自定义如何将若干样本合并为一个 batch。其默认行为如下：

- 如果是 Tensor 类型，会增加一个 batch 维；
- 如果是字典，会对每个键值递归调用 `default_collate`；
- 如果是 List/NamedTuple，也会进行相应整合。

功能说明： 该函数接受一个样本列表（例如长度为 batch size），并将其中每个元素的对应字段打包为一个 `Tensor`，最终形成一个包含 batch size 的张量。

输入输出映射： 根据输入类型，`default_collate` 的行为如下：

- `torch.Tensor` → `torch.Tensor`: 添加一个 batch 维度。
- NumPy 数组 → `torch.Tensor`: 自动转换并添加 batch 维度。
- Python 基本类型（`int`, `float`）→ `torch.Tensor`: 转换并添加 batch 维度。

- `str`, `bytes` → 保持不变。
- `Mapping[K, V]`（如字典）→ 对每个 key 分别调用 `default_collate`。
- `NamedTuple` 或普通 `Tuple/List` → 按位置逐元素调用 `default_collate`。

```
default_collate([0, 1, 2, 3])
# 输出: tensor([0, 1, 2, 3])

default_collate(['a', 'b', 'c'])
# 输出: ['a', 'b', 'c']  (字符串不变)

default_collate([{'A': 0, 'B': 1}, {'A': 100, 'B': 100}])
# 输出: {'A': tensor([0, 100]), 'B': tensor([1, 100])}

Point = namedtuple('Point', ['x', 'y'])
default_collate([Point(0, 0), Point(1, 1)])
# 输出: Point(x=tensor([0, 1]), y=tensor([0, 1]))

default_collate([(0, 1), (2, 3)])
# 输出: [tensor([0, 2]), tensor([1, 3])]

default_collate([[0, 1], [2, 3]])
# 输出: [tensor([0, 2]), tensor([1, 3])]
```

直观理解： `default_collate` 会“提取第一维”的同类型元素聚合到一起形成一个新的 batch。例如对于两个 `Tuple`，`(x1, y1)` 和 `(x2, y2)`，其输出为 `[tensor([x1, x2]), tensor([y1, y2])]`。

示例：collate_fn 会自动扩充 batch 维度 默认的 `collate_fn` 会在每个样本张量的最前面新增一维作为 batch 维度。例如，在图像数据中，每个图像原本的尺寸是 `(3, 32, 32)`，即通道数为 3，高宽均为 32 的彩色图像。

通过如下代码我们可以验证其效果：

```
image_data = my_dataset(r"D:\dataset\cifar100_images\train", transform)
image_loader = torch.utils.data.DataLoader(image_data, batch_size=128,
    shuffle=True, num_workers=0)

for batch in image_loader:
    x, y = batch
    print(x.shape, y.shape)
    break
```

输出结果如下：

```
torch.Size([128, 3, 32, 32]) torch.Size([128])
```

这表明 collate_fn 已将原本每张图片的维度 (3, 32, 32) 扩展为 batch 维 (128, 3, 32, 32)，即每一批包含 128 张图像。

同理，标签值本为整数，如 $0, 1, 2, \ldots$，也会被聚合为一个 Tensor，其维度变为 (128)，表示一批图像对应的 128 个标签值。

Mapping 示例： 当每个样本是一个字典(Mapping)类型时，例如 {"input": ..., "output": ...}，default_collate 会将相同键下的值分别聚合成一个 batch。

```
# 三个样本
data1 = {"input": torch.Tensor([1, 2]), "output": 3}
data2 = {"input": torch.Tensor([1, 3]), "output": 2}
data3 = {"input": torch.Tensor([1, 4]), "output": 1}

# 调用 default_collate 后：
batch = {
    "input": default_collate([torch.Tensor([1, 2]), torch.Tensor([1, 3]),
       torch.Tensor([1, 4])]),
    "output": default_collate([3, 2, 1])
}

# 结果：
batch["input"] = tensor([[1., 2.],
                         [1., 3.],
                         [1., 4.]])
batch["output"] = tensor([3, 2, 1])
```

这说明 default_collate 会自动提取所有样本中相同字段（如 "input" 和 "output"），分别聚合成 batch，并保持结构不变，适用于如字典、JSON、字典嵌套等结构的数据预处理。

1.4 自定义 collate_fn

在实际数据处理中，尤其是使用 Huggingface Datasets 时，我们常会遇到以下问题：如果样本返回的是 list 类型，而不同样本 list 的长度不一，则默认的 collate_fn 将无法正确打包 batch，会引发错误。因此，对于 list 结构的样本，推荐使用自定义 collate_fn 或手动设置 Dataset 的格式为 torch 类型：

- 手动设置数据格式为 torch 类型，使得返回值为 tensor 而非 list；
- 或者自定义 collate_fn 函数来对返回的 list 数据进行加工处理。

首先我们加载数据：

```
text_data = load_dataset("allenai/common_gen", split="train")
```

原始数据集的格式如下：

```
{'concept_set_idx': 0, 'concepts': ['ski', 'mountain', 'skier'],
 'target': 'Skier␣skis␣down␣the␣mountain'}
```

原因分析： 因为 text_data 中的样本字段如 "concepts" 是字符串列表，其长度不一，导致默认 collate_fn 无法将多个 list 打包为 tensor，从而报错。

解决方式：使用 map 构造结构化样本 + set_format 下面通过 map 操作将数据中的概念与目标串拼接成输入、输出样本：

```
def add_eos_to_examples(example):
    string = ",".join(example['concepts'])
    example['input_text'] = '%s␣.' % string
    example['target_text'] = '%s␣' % example['target']
    return example

text_data = text_data.map(add_eos_to_examples, batched=False,
                          remove_columns=text_data.column_names)
```

转换后的样本格式如下：

```
{'input_text': 'ski,mountain,skier␣.',
 'target_text': 'Skier␣skis␣down␣the␣mountain␣'}
```

然后使用 AutoTokenizer 将其编码为数值型数据：

```
def convert_to_features(example_batch):
    input_encodings = tokenizer(example_batch['input_text'],
                                padding="max_length",
                                max_length=16,
                                truncation=True,
                                return_tensors="pt")
    target_encodings = tokenizer(example_batch['target_text'],
                                 padding="max_length",
                                 max_length=16,
                                 truncation=True,
                                 return_tensors="pt").input_ids

    labels_with_ignore_index = []
    for labels_example in target_encodings:
        labels_example = [label if label != 0 else -100 for label in
            labels_example]
        labels_with_ignore_index.append(labels_example)

    encodings = {
        'input_ids': input_encodings['input_ids'],
        'attention_mask': input_encodings['attention_mask'],
        'labels': labels_with_ignore_index
    }
    return encodings

```

```
text_data = text_data.map(convert_to_features, batched=True,
                          remove_columns=text_data.column_names)
```

编码后的格式如下：

```
{
 'input_ids': [49406, 3428, 267, 3965, 267, 42585, 269, 49407, 49407, ...],
 'attention_mask': [1, 1, 1, 1, 1, 1, 1, 1, 0, 0, ...],
 'labels': [49406, 42585, 32843, 1136, 518, 3965, 49407, 49407, ...]
}
```

说明：编码函数通过设置最大编码长度（如 `max_length=16`），将所有句子长度统一补零到最大长度，从而形成形状一致的 `tensor`，方便后续的批量处理和模型训练。

最后，我们通过以下命令将数据转换为 Tensor 格式，以便 `collate_fn` 能正确打包成 batch：

```
text_data.set_format(type="torch")
```

若不加该语句，`DataLoader` 会返回嵌套的 list，可能导致结构错乱或训练错误。使用该命令后，返回的内容会是如下形式的 Tensor：

```
torch.Size([4, 16])  # 四个样本，每个样本为长度为16的句子
```

效果对比：

- 不加该语句，`DataLoader` 返回的是 list，不能直接进行 batch；
- 加上该语句后，返回的是 tensor，形状如 `(batch_size, seq_len)`，更易于模型输入。

例子说明：

若原始字段为：

```
[101, 1005, 1012, 1006, 1012, 1007, 102]
```

则 batch 后：

- 加上 `set_format`：得到一个 tensor，形状为 `(4, 7)`；
- 不加：将返回一个 list，其中每列为 `tensor([101, 101, 101, 101])`、`tensor([1005, 1005, ...])` 等。

1.5 文本数据的动态 padding

在文本建模任务中，输入句子的长度可能差异显著。为保证批量训练时 `tensor` 尺寸统一，常采用 **padding 补齐**的方式来对齐输入的序列长度。

固定长度 Padding

我们通常使用如下方式在编码阶段完成固定长度补齐：

```
input_encodings = tokenizer(
    example_batch['input_text'],
    padding="max_length",
    max_length=16,
    truncation=True,
    return_tensors="pt"
)
```

- `padding="max_length"` 表示所有句子都强制补齐到 `max_length` 的长度；
- `max_length=16` 指定最大长度为 16，过长的句子将被截断；
- `truncation=True` 表示允许截断，防止句子超长；
- `return_tensors="pt"` 返回 PyTorch 的 `tensor` 格式；

潜在问题：若设置的 `max_length` 太短，长句会被截断导致信息损失；若设置太长，则导致内存浪费，训练效率下降。

动态 Padding

为提升灵活性和效率，我们也可以采用 **动态 padding**，即在 `collate_fn` 中根据每个 batch 内的句子最大长度进行补齐：

这时不进行提前补齐，而是保留原始编码长度。在 `DataLoader` 中定义如下自定义 collate 函数：

```
def collate_fn(batch):
    return tokenizer.pad(
        batch,                  # 输入为一个列表
        padding=True,           # 自动根据最长句子补齐
        return_tensors="pt"     # 输出 tensor 格式
    )
```

使用说明

若采用动态 padding，需确保：

- 预处理阶段不进行固定长度 padding；
- 使用 `DataLoader` 时传入自定义 `collate_fn`；

完整流程示例如下：

```
# 不设置 max_length
text_data = text_data.map(tokenizer, remove_columns=text_data.column_names)

```

```
# 加载器中使用动态 padding
loader = torch.utils.data.DataLoader(
    text_data,
    batch_size=8,
    shuffle=True,
    collate_fn=collate_fn
)
```

文本通常长度不一，默认 collate 函数无法处理，需自定义函数动态 padding：

```
def collate_fn(batch):
    return tokenizer.pad(
        batch,
        padding=True,
        return_tensors="pt",
    )
```

1.6 自定义 collate：prompt 拼接示例

```
def custom_collate_fn(batch):
    prompts_batch = []
    labels_batch = []
    for item in batch:
        prompts_batch += item["prompts"]
        labels_batch += item["labels"]
    return {"prompts": prompts_batch, "labels": labels_batch}
```

此函数将嵌套结构的 List[”prompts”] 与 [”labels”] 拉平成统一形式，适配后续模型输入。

2 优化器的原理与实现

2.1 优化器简介

优化器（**Optimizer**）是深度学习中用于 **迭代更新模型参数**的关键组件，其目的是通过最小化损失函数，逐步逼近最优解。优化器根据参数的梯度信息，计算出更新步长，并施加在参数本身上以实现权重的迭代优化。

优化器的核心功能

- 管理参数组（`param_groups`）；
- 跟踪参数状态（如动量、均方梯度）；
- 提供 `step()` 方法进行参数更新；
- 支持调度器和 `state_dict` 的存取；

常见优化器类型

- **SGD（随机梯度下降）**：最基础的优化方法；
- **Momentum（动量法）**：在 SGD 基础上引入历史梯度累积；
- **Adam**：结合了 Momentum 与 RMSProp 的优点，具有自适应学习率能力；

2.2 SGD 与动量法实现：`MySGDWithMomentum`

`MySGDWithMomentum` 是基于 PyTorch `Optimizer` 的自定义动量优化器，其核心思想为在参数更新中加入历史梯度：

$$v_t = \mu v_{t-1} + \nabla L(\theta) \quad , \quad \theta = \theta - \eta v_t \tag{1}$$

其中，μ 为动量因子，η 为学习率，v_t 为当前动量。

```
buf.mul_(momentum).add_(d_p)
p.data.add_(-lr, buf)
```

此实现中使用 `momentum_buffer` 存储历史梯度累积，能够加快收敛、减少震荡。
注意：初始化阶段需判断 `momentum_buffer` 是否存在，否则使用当前梯度初始化。

2.3 手动实现 SGD：`MySGDManual`

`MySGDManual` 是更原始的手动实现版本，仅执行最基本的梯度下降：

$$\theta = \theta - \eta \nabla L(\theta) \tag{2}$$

```
p.data -= self.lr * p.grad
```

该实现无状态记录、无动量、也不支持参数组，仅适用于教学或极简场景。
附加说明：需手动调用 `zero_grad()` 清空上一轮的梯度。

2.4 自定义 Adam 优化器：`MyAdamOptim`

Adam（Adaptive Moment Estimation）结合了一阶动量和二阶动量估计：

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\nabla L(\theta) v_t = \beta_2 v_{t-1} + (1-\beta_2)(\nabla L(\theta))^2 \tag{3}$$

使用偏差校正后更新参数：

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1-\beta_2^t} \theta = \theta - \eta \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \tag{4}$$

对应代码片段为：

```
exp_avg.mul_(beta1).add_(grad, alpha=1 - beta1)
exp_avg_sq.mul_(beta2).addcmul_(grad, grad, value=1 - beta2)
bias_correction1 = 1 - beta1 ** step
bias_correction2 = 1 - beta2 ** step
p.data.addcdiv_(exp_avg, denom, value=-step_size)
```

2.5 手动实现 Adam：MyAdam

该版本未继承 torch.optim.Optimizer，而是手动管理：

- 使用 dict 储存每个参数的 m, v；
- 手动实现偏差校正；
- 支持对每个参数单独更新；

```
m_hat = m / (1 - beta1 ** self.t)
v_hat = v / (1 - beta2 ** self.t)
update = self.lr * m_hat / (v_hat.sqrt() + self.eps)
p.data.add_(-update)
```

2.6 优化器与模型联动机制

无论是继承版还是手写版，优化器都必须接收模型的 parameters()：

```
optimizer = MyAdam(model.parameters(), lr=1e-3)
for epoch in range(n_epochs):
    loss = compute_loss(model(inputs), labels)
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```

注意事项：

- 必须调用 zero_grad()，否则梯度将累积；
- loss.backward() 会自动将梯度存储在 param.grad 中；
- 优化器操作的是模型的“引用参数”，参数会在原地更新；