数据集和优化器

蔚全爱

2025.8.5

1 dataset 和 dataloader

- 从本地文件夹中读取图像，并自动生成类别标签，可用于图像分类任务的数据加载，定义了如何获取单个样本。

```python
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))

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

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

- DataLoader 封装了 Dataset，它的核心功能是将单个样本的采样方式转换成按批次 (batch) 的采样方式封装的 DataLoader 的初始化函数接收多个参数来控制数据加载的行为。

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

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

```
])

dataset = my_dataset("path/to/images", transform)
loader = DataLoader(dataset, batch_size=64, shuffle=True,
    num_workers=4)
```

- 参数：collate_fn 用于将单个样本组成 batch 的函数，决定了采样方法

```
    def __init__(
    self,
    dataset: Dataset[_T_co],
    batch_size: Optional[int] = 1,
    shuffle: Optional[bool] = None,
    sampler: Union[Sampler, Iterable, None] = None,
    batch_sampler: Union[Sampler[List], Iterable[List], None] = None
        ,
    num_workers: int = 0,
    collate_fn: Optional[_collate_fn_t] = None,
    pin_memory: bool = False,
    drop_last: bool = False,
    timeout: float = 0,
    worker_init_fn: Optional[_worker_init_fn_t] = None,
    multiprocessing_context=None,
    generator=None,
    *,
    prefetch_factor: Optional[int] = None,
    persistent_workers: bool = False,
    pin_memory_device: str = "",
    in_order: bool = True
)
```

- 其中，collate_fn 重要，默认一般无法满足要求，需自定义 collate_fn 函数
- 默认的 collate_fn 会自动将常见类型聚合为 Tensor，具体如下：
  - Tensor：按第一个维度拼接为形如 (batch_size, ...) 的张量，32*32→1*32*32；
  - int / float：转换为对应标量 Tensor；
  - list：递归拼接为 Tensor，要求内部元素形状一致；
  - dict / Mapping：对每个字段分别递归调用 collate_fn；
  - NamedTuple / Sequence：对每个字段位置单独聚合；
  - str / bytes：保持原样，不进行拼接；
- 示例
  - **int** 列表：拼为一维张量

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

  - **str** 列表：保持原样

```
default_collate(['a', 'b', 'c']) # ['a', 'b', 'c']
```

  - **dict** 列表：按键拼接

```
default_collate([{'A': 0}, {'A': 100}]) # {'A': tensor([0,
    100])}
```

  - **NamedTuple** 列表：字段分别拼接

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

  - **Tuple** 列表：各位置独立拼接

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

  - **List** 列表：转换为二维张量

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

- 默认的 `collate_fn` 对以下复杂情况无法处理，常见于自然语言和生成式任务：
  - 样本中的字段是长度不一致的字符串序列（如自然语言的句子或段落），无法转换为规则形状的 Tensor；
  - 每个样本中存在嵌套结构(如 `dict` 中包含 `list[str]` 或 `list[list]`)，默认函数无法处理这种深层嵌套；
  - Few-shot Prompt 场景中，一个样本可能包含多个示例文本拼接而成的 prompt，导致结构不一致；
- 处理 `allenai/common_gen` 等类似数据集时，字段的值是一个长度不一的字符串列表，默认在尝试将不同长度的样本堆叠成一个批次时会出错，解决方案：打补丁 1. 列表转字符串；2. 字符串转定固定长 ID 序列；3. 设置最终格式为 `Tensor`，数据格式变得规整，就可以用默认的；
- 构造 few-shot prompt 数据

```
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))
        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})

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

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

返回值结构说明：`__getitem__` 返回一个字典，包含：

- "prompts"：长度为 k 的字符串列表；
- "labels"：对应的 k 个类别标签；

```
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}
```

- 典型用法：自定义数据加载器

```
image_data = my_dataset("path/to/images", transform)
image_loader = DataLoader(image_data, batch_size=128, shuffle=True)
```

```
my_data = llmCustomDataset(label_list, fg_example, 5, 10)
my_loader = DataLoader(my_data, batch_size=4, collate_fn=
    custom_collate_fn)
```

2 优化器

- 优化器：梯度和学习率，模块：参数管理、超参数与默认管理、状态管理（如动量、学习率衰减等）、参数更新规则、梯度清零模块。流程：正向传播、反向传播、参数更新。
- 主流优化器：带动量的 SGD、AdaGrad、Adam、RMSprop
- 带有动量的 SGD 实现
  - 继承自 Optimizer 带有动量的 SGD 实现，通过 `self.state` 管理动量；
  - 动量和参数更新公式：

$$
\begin{aligned}
v_t &= \mu \cdot v_{t-1} + g_t \\
\theta_t &= \theta_{t-1} - \eta \cdot v_t
\end{aligned}
$$

  - 代码实现：

```python
class MySGDWithMomentum(Optimizer):
    def __init__(self, params, lr=1e-3, momentum=0.9):
        if lr <= 0.0:
            raise ValueError("Invalid learning rate")
        if momentum < 0.0:
            raise ValueError("Invalid momentum")

        defaults = dict(lr=lr, momentum=momentum)
        super(MySGDWithMomentum, self).__init__(params, defaults)

    @torch.no_grad()
    def step(self, closure=None):
        loss = None
        if closure is not None:
            with torch.enable_grad():
                loss = closure()

        for group in self.param_groups:
            lr = group["lr"]
            momentum = group["momentum"]

            for p in group["params"]:
                if p.grad is None:
                    continue
```

```
                d_p = p.grad

                state = self.state[p]

                if "momentum_buffer" not in state:
                    buf = state["momentum_buffer"] = torch.clone(d_p).
                        detach()
                else:
                    buf = state["momentum_buffer"]
                    buf.mul_(momentum).add_(d_p)

                p.data.add_(-lr, buf)

        return loss
```

- MySGDManual: 手动实现、不继承 Optimizer
  - 参数实现：

$$\theta_t = \theta_{t-1} - \eta \cdot g_t$$

  - 代码实现：

```
        # Optimizer作用：参数组的保存，state_dict存储，调度器支持等
class MySGDManual:
    def __init__(self, params, lr=1e-3):
        self.params = list(params)
        self.lr = lr
        # self.params.data,self.params.grad

    def step(self):
        for p in self.params:
            if p.grad is not None:
                p.data -= self.lr * p.grad

    def zero_grad(self):
        for p in self.params:
            if p.grad is not None:
                p.grad.zero_()
```

- MyAdamOptim
  - 继承自 Optimizer 基类,Adam 结合了动量和 RMSprop 的思想,betas 分别用于控制一阶矩和二阶矩估计的指数衰减率

– 更新公式:

$$
\begin{aligned}
m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t \\
v_t &= \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \\
\hat{m}_t &= \frac{m_t}{1-\beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1-\beta_2^t} \\
\theta_t &= \theta_{t-1} - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}
\end{aligned}
$$

– 代码实现:

```
class MyAdamOptim(Optimizer):
def __init__(self, params, lr=1e-3, betas=(0.9, 0.999), eps=1e
    -8):
    if lr <= 0.0:
        raise ValueError(f"Invalid learning rate: {lr}")
    if not 0.0 <= betas[0] < 1.0 or not 0.0 <= betas[1] < 1.0:
        raise ValueError(f"Invalid betas: {betas}")
    if eps < 0.0:
        raise ValueError(f"Invalid epsilon value: {eps}")

    defaults = dict(lr=lr, betas=betas, eps=eps)
    super(MyAdamOptim, self).__init__(params, defaults)

@torch.no_grad()
def step(self, closure=None):
    loss = None
    if closure is not None:
        with torch.enable_grad():
            loss = closure()

    for group in self.param_groups:
        for p in group["params"]:
            if p.grad is None:
                continue
            grad = p.grad.data

            lr = group["lr"]
            beta1, beta2 = group["betas"]
            eps = group["eps"]

            state = self.state[p]

            if len(state) == 0:
                state["step"] = 0
                state["exp_avg"] = torch.zeros_like(p.data) # m
                state["exp_avg_sq"] = torch.zeros_like(p.data) # v

            exp_avg, exp_avg_sq = state["exp_avg"], state["
                exp_avg_sq"]
```

```
            state["step"] += 1
            step = state["step"]

            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

            denom = (exp_avg_sq.sqrt() / (bias_correction2 ** 0.5)
                ).add_(eps)
            step_size = lr / bias_correction1

            p.data.addcdiv_(exp_avg, denom, value=-step_size)

    return loss
```

- MyAdam
  - 纯手动实现的 Adam 优化器
  - 代码实现

```
class MyAdam:
def __init__(self, params, lr=1e-3, betas=(0.9, 0.999), eps=1e
    -8):
    self.params = list(params)
    self.lr = lr
    self.betas = betas
    self.eps = eps
    self.state = {}
    self.t = 0

def step(self):
    self.t += 1
    beta1, beta2 = self.betas

    for p in self.params:
        if p.grad is None:
            continue

        grad = p.grad.data
        if p not in self.state:
            self.state[p] = {
                'm': torch.zeros_like(p.data),
                'v': torch.zeros_like(p.data)
            }

        m = self.state[p]['m']
```

```python
            v = self.state[p]['v']

            m.mul_(beta1).add_(grad, alpha=1 - beta1)
            v.mul_(beta2).addcmul_(grad, grad, value=1 - beta2)

            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)

    def zero_grad(self):
        for p in self.params:
            if p.grad is not None:
                p.grad.zero_()
```