机器学习编程课笔记

2025 年 8 月 4 日

一、datasets：高效加载与预处理数据

1.1 加载数据集

使用 HuggingFace 的 `datasets` 库只需一行代码即可加载标准数据集：

```
from datasets import load_dataset
dataset = load_dataset("imdb")
```

- 数据集通过名字在 Hugging Face Hub 上查找并下载；
- 本地缓存路径一般位于 `~/.cache/huggingface/datasets`；
- 如果网络不可用，可先在本地下载并通过 `cache_dir` 指定目录加载。

- `load_dataset_builder` *feature*

1.2 数据查看与切片

数据加载后可像 Python 字典或列表一样操作：

```
from datasets import load_dataset, get_dataset_split_names
splits = get_dataset_split_names("cornell-movie-review-data/
    rotten_tomatoes")
print(splits)  # 例如 ['train', 'validation', 'test']
dataset_all = load_dataset("cornell-movie-review-data/rotten_tomatoes
    ")
print(dataset['train'][0])        # 查看第一条样本
print(dataset['train'].features) # 查看字段
```

未指定 split 时返回包含多个子集的 `DatasetDict`，指定 split 则返回单一 `Dataset` 对象:contentReference[oaicite:2]index=2。

1.3 数据预处理

datasets 提供 map() 函数用于预处理，支持批处理和多进程：

```
def preprocess(example):
    return {"text": example["text"].lower()}

dataset = dataset.map(preprocess)
```

map() 在大规模数据集上通常比逐条预处理更快、更节省内存。

- example 是字典类型，表示单个样本；
- 返回值也应为字典，key 为原有或新字段名；
- 默认情况下，map() 会对每一个样本逐个调用处理函数。

1.4 与 PyTorch 集成

```
from torch.utils.data import DataLoader

def tokenize_fn(examples):
    return tokenizer(examples["text"], padding="max_length",
       truncation=True)

tokenized = dataset.map(tokenize_fn, batched=True)
tokenized.set_format(type="torch", columns=["input_ids", "label"])
loader = DataLoader(tokenized, batch_size=16, shuffle=True)
```

支持与 PyTorch、TensorFlow、NumPy、Pandas 等框架无缝集成的高效数据 pipeline。

二、diffusers：扩散模型与流水线封装

2.1 核心思想

diffusers 是 HuggingFace 提供的扩散模型库，将文本到图像生成封装为标准 pipeline，核心组件包括：

- **VAE（变分自编码器）**：压缩图像为隐空间；
- **UNet**：扩散和反扩散过程的主模型；
- **Text Encoder（如 CLIP）**：将文本转化为向量；

- **Scheduler**：控制扩散采样过程；
- **Safety Checker**：检测输出图像是否违规。

2.2 解构基础 Pipeline：以 DDPM 为例

官方示例使用 DDPMPipeline 展示推理流程：

```
from diffusers import DDPMPipeline
ddpm = DDPMPipeline.from_pretrained("google/ddpm-cat-256").to("cuda")
image = ddpm(num_inference_steps=25).images[0]
```

Pipeline 内部封装了 `UNet2DModel` 和 `DDPMScheduler`，依次执行以下步骤：

1. 从随机噪声生成初始样本
2. `scheduler.set_timesteps(num_steps)` 创建时间步序列；
3. 每个时间步调用 `model(image, t).sample` 预测残差；
4. 用 scheduler 的 `step()` 方法反推上一个时间步骤的图像；
5. 重复迭代至结束，输出去噪图像；
6. 图像映射到 $[0, 1]$ 范围并转换为 NumPy 数组返回:contentReference[oaicite:2]index=2。

2.3 手写自定义推理流程代码

以下是官方演示的“自己写推理流程”的流程结构：

```
from diffusers import DDPMScheduler, UNet2DModel
scheduler = DDPMScheduler.from_pretrained("google/ddpm-cat-256")
model = UNet2DModel.from_pretrained("google/ddpm-cat-256").to("cuda")
scheduler.set_timesteps(50)
noise = torch.randn((1, model.config.sample_size, model.config.
    sample_size))
image = noise
for t in scheduler.timesteps:
    model_out = model(image, t).sample
    image = scheduler.step(model_out, t, image).prev_sample
# 输出去噪结果并转换为图像
```

说明了 Pipeline 封装流程中每一步关键操作：模型与 scheduler 分离、循环去噪、生成输出图像:contentReference[oaicite:3]index=3。

2.4 编写自定义 Pipeline 类

你还可以定义完全手写的 Pipeline 类继承自 DiffusionPipeline：

```
import torch
from diffusers import DiffusionPipeline

class MyPipeline(DiffusionPipeline):
    def __init__(self, unet, scheduler):
        super().__init__()
        self.register_modules(unet=unet, scheduler=scheduler)

    @torch.no_grad()
    def __call__(self, batch_size=1, num_inference_steps=50):
        img = torch.randn((batch_size, self.unet.in_channels,
                           self.unet.sample_size, self.unet.
                              sample_size))
        self.scheduler.set_timesteps(num_inference_steps)
        for t in self.scheduler.timesteps:
            model_out = self.unet(img, t).sample
            img = self.scheduler.step(model_out, t, img).prev_sample
        img = (img / 2 + 0.5).clamp(0, 1)
        return img.cpu().permute(0,2,3,1).numpy()
```

然后你可以将该类保存为 pipeline.py 并上传至 Hub 的自定义 repo，即可支持如下加载：

```
from diffusers import DiffusionPipeline
pipe = DiffusionPipeline.from_pretrained(
    "google/ddpm-cat-256",
    custom_pipeline="my_pipeline_repo"
)
```

其中 my pipeline repo 包含定义该类的 pipeline.py 文件，DiffusionPipeline 会自动加载其中的流程代码进行推理:contentReference[oaicite:4]index=4。

2.5 社区 Pipeline 与 Hub 加载方式

- 若已有社区 pipeline 名称，比如 clip guided stable diffusion，可以直接使用：
- 可加载社区 pipelines 或本地 pipeline 目录，实现高度自定义；使用时注意设置 trust_remote_code=True（如 Transformers pipeline）以加载自定义代码

  :contentReference[oaicite:5]index=5。

```
from diffusers import DiffusionPipeline
from transformers import CLIPModel, CLIPFeatureExtractor

pipe = DiffusionPipeline.from_pretrained(
  "CompVis/stable-diffusion-v1-4",
  custom_pipeline="clip_guided_stable_diffusion",
  clip_model=CLIPModel.from_pretrained(...),
  feature_extractor=CLIPFeatureExtractor.from_pretrained(...)
)
```

2.6 pipeline 结构说明

Pipeline 封装了多个模块，每步可自定义替换，如：

- 替换 VAE 以加速采样；
- 替换调度器实现跳步采样（如 EulerDiscreteScheduler）；
- 替换 UNet 或使用 ControlNet、IPAdapter 等增强模型能力。

三、transformers：模型加载与训练器

3.1 自动模型加载

transformers 库支持一行加载预训练模型和分词器：

```
from transformers import AutoModel, AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained("bert-base-uncased")
model = AutoModel.from_pretrained("bert-base-uncased")
```

3.2 使用 Trainer 训练

通过 Trainer 类快速完成模型训练，支持 GPU 并行、logging、checkpoint、早停等功能。

```
from transformers import Trainer, TrainingArguments

args = TrainingArguments(
```

```
    output_dir="outputs",
    evaluation_strategy="epoch",
    per_device_train_batch_size=16,
    num_train_epochs=3,
    save_steps=500,
)

trainer = Trainer(
    model=model,
    args=args,
    train_dataset=train_data,
    eval_dataset=eval_data,
)

trainer.train()
```

四、高效微调策略（PEFT）

4.1 LoRA

LoRA 假设：模型的权重更新 ΔW 可以表示为两个低秩矩阵 $A \in R^{d\times r}$ 和 $B \in R^{r\times k}$ 的乘积：

$$W' = W + \Delta W = W + AB$$

其中 $r \ll d, k$，表示秩的降低，从而极大减少了参数量。

LoRA 通常作用于 Transformer 的 attention 模块中，如 Q、K、V、O 权重矩阵：

- 替换原来的 W_q 为 $W_q + A_q B_q$
- 冻结原始 W_q，仅训练 A_q 与 B_q
- **参数高效**：只需训练少量参数（如 0.1% 原始模型参数）；
- **与全参性能接近**：在许多任务上效果与 full tuning 相当；
- **模块化部署**：LoRA 权重文件小，可快速部署在不同下游任务。

LoRA（Low-Rank Adaptation）用两个小矩阵 A, B 替代大矩阵更新，只需训练极少参数：

$$W' = W + A \cdot B, \quad A \in R^{d\times r},\ B \in R^{r\times k}$$

使用 `peft` 快速封装：

```
from peft import get_peft_model, LoraConfig

config = LoraConfig(r=4, alpha=16, target_modules=["q_proj", "v_proj"
    ])
model = get_peft_model(model, config)
```

4.2 Prompt Tuning

传统的 fine-tuning 需要对模型所有参数进行更新，而 Prompt Tuning 则直接向输入嵌入中插入可训练的“提示向量”（prompt embedding）：

$$\text{Input Embedding} = [P_1, P_2, \ldots, P_n, x_1, x_2, \ldots, x_m]$$

其中 P_i 为可学习的 soft prompt 向量，x_j 为原始文本 token 的 embedding，整个模型参数保持冻结，仅训练 P。Prompt Tuning 仅优化 prompt embedding 而不更新模型参数，适用于少量数据下的快速适配。支持 Soft Prompt、Prefix Tuning 等方式。