HuggingFace

蔚全爱

2025.8.4

1 Hugging Face 平台介绍

- 网站名称：**Hugging Face Hub** https://huggingface.co

2 dataset: 数据加载与处理

- 使用 Hugging Face 加载数据集：

```
from datasets import load_dataset

dataset = load_dataset("your_dataset_name",split="train")
```

  - 可先在本地电脑挂梯子下载，再上传至服务器：下载并缓存至本地 ~/.cache/huggingface/datasets，然后将整个缓存目录（如 ~/.cache/huggingface）打包上传至服务器相同路径。
  - 或者手动下载后指定本地路径加载：

```
dataset = load_dataset("/path/to/your/local_dataset", split="train"
    )
```

- 在下载前查看数据集结构，可使用 load_dataset_builder() 方法获取字段、划分等信息：

```
from datasets import load_dataset_builder

builder = load_dataset_builder("your_dataset_name")
print(builder.info.description) # 数据集描述
print(builder.info.features) # 字段信息（如 text、label）
print(builder.info.splits) # 划分情况（如 train/validation/test）
```

  - 可根据字段判断是否适合下游任务，如是否有标签字段、数据类型是否为文本等；

- 对加载好的原始数据集进行预处理（如文本分词）：

```python
from transformers import AutoTokenizer

# 加载模型对应的分词器（以 BERT 为例）
tokenizer = AutoTokenizer.from_pretrained("bert-base-uncased")

# 定义映射函数：对每个样本进行分词
def tokenize_function(example):
    return tokenizer(example["sentence1"], example["sentence2"],
        truncation=True)

# 使用 .map 批量映射处理整个数据集
tokenized_datasets = dataset.map(tokenize_function, batched=True)
```

  - `map(..., batched=True)` 表示对多个样本批量处理，提升效率，数据处理主要是使用 map；

3 diffusers：文本到图像生成

- 扩散模型的封装：`diffusers` 库将扩散模型中多个核心模块（如 VAE、UNet、Text Encoder、调度器等）封装为统一的 `Pipeline` 接口，用户只需调用少量 API 即可实现完整的文本生成图像流程。
  - 此处的 VAE、UNet、Text Encoder 等模块均为预训练模型
  - VAE 实现图像压缩，正向过程中将图像压缩为低维 latent 表示，反向生成时再恢复。可以减少 UNet 处理成本
- 以 **Stable Diffusion XL (SDXL)** 为例，其加载和使用方式如下：

```python
from diffusers import StableDiffusionXLPipeline

# 从 Hugging Face 模型库加载 SDXL 模型权重
pipe = StableDiffusionXLPipeline.from_pretrained(
    "stabilityai/stable-diffusion-xl-base-1.0",
    torch_dtype=torch.float16
).to("cuda")

# 使用 prompt 生成图像
prompt = "a futuristic landscape, vivid colors, ultra detailed"
image = pipe(prompt=prompt).images[0]
image.save("sdxl_output.png")
```

- 采样速度慢的问题：传统扩散模型（如 DDPM）需上百步去噪，采样速度较慢，尤其在高分辨率图像生成中明显。
- 常用采样器（**Scheduler**）：
  - `DDIMScheduler`：确定性采样，速度快；

- EulerAncestralDiscreteScheduler：细节表现好，适用于 SDXL；
- PNDMScheduler：兼顾速度与稳定性；
- DPM++、UniPC：采样质量与速度均优秀；

- 替换采样器示例：

```
from diffusers import StableDiffusionPipeline,
    EulerAncestralDiscreteScheduler

pipe = StableDiffusionPipeline.from_pretrained("stabilityai/stable-
    diffusion-2-1")
pipe.scheduler = EulerAncestralDiscreteScheduler.from_config(pipe.
    scheduler.config)
pipe.to("cuda")
image = pipe("a fantasy castle, ultra detailed").images[0]
```

- 扩散模型的核心过程是对潜变量 latent 进行多轮去噪迭代。以下代码是 diffusers 中 StableDiffusionPipeline 的 __call__() 函数，实现了逐步采样与噪声估计：

```
for i, t in enumerate(timesteps):
    if self.interrupt:
        continue

    # 若启用 Classifier-Free Guidance，则复制 latent 作为无条件和有条
        件输入
    latent_model_input = torch.cat([latents] * 2) if self.
        do_classifier_free_guidance else latents
    latent_model_input = self.scheduler.scale_model_input(
        latent_model_input, t)

    # 使用 UNet 预测噪声
    noise_pred = self.unet(
        latent_model_input,
        t,
        encoder_hidden_states=prompt_embeds,
        timestep_cond=timestep_cond,
        cross_attention_kwargs=self.cross_attention_kwargs,
        added_cond_kwargs=added_cond_kwargs,
        return_dict=False,
    )[0]
```

解释：UNet 模型的目标是学习函数 $\hat{\epsilon}_\theta(z_t, t)$，预测当前潜变量中的噪声；每一步使用 scheduler.step() 根据预测结果更新 latent，即 z_{t-1}；整个过程不断迭代，从高噪声 latent 开始，逐步生成清晰图像。

- 迭代结构：for 循环遍历调度器给定的时间步 timesteps，每一步都执行一次去噪操作；

- **UNet 输入构造：**
  - 若启用 Classifier-Free Guidance，则将 latent 复制两份（有条件与无条件）拼接输入；
  - 使用调度器的 scale_model_input 对 latent 进行缩放，保持数值稳定性；
- **UNet 去噪预测：**
  - 核心调用 self.unet(...),输出当前时间步下的噪声预测 noise_pred；
  - UNet 输入包含时间步 t、提示词嵌入 prompt_embeds 以及可选的 cross-attention 条件；

4 Transformers: 模型训练与语言训练、多模态模型的封装

- 轻量语言模型 phi-3-mini-4k-instruct 为例，使用自动加载接口快速加载

```
from transformers import AutoModelForCausalLM, AutoTokenizer

# 加载模型与分词器
model = AutoModelForCausalLM.from_pretrained("microsoft/Phi-3-mini
    -4k-instruct")
tokenizer = AutoTokenizer.from_pretrained("microsoft/Phi-3-mini-4k-
    instruct")

# 构建对话格式（支持 role-based 聊天）
messages = [{"role": "user", "content": "Can you provide ways to
    eat combinations of bananas and dragonfruits?"}]
inputs = tokenizer.apply_chat_template(messages,
    add_generation_prompt=True, return_tensors="pt")

# 调用生成接口
outputs = model.generate(inputs, max_new_tokens=32)
text = tokenizer.batch_decode(outputs)[0]

print(text)
```

  - 跨版本加载，自动调用对应的模型类；

- 模型训练：使用 Trainer API 进行模型微调

```
from transformers import Trainer

trainer = Trainer(
    model=model,
    args=training_args, # 使用 TrainingArguments 定义训练参数
    train_dataset=dataset["train"], # 训练数据集
```

```
    eval_dataset=dataset["test"], # 验证数据集
    processing_class=tokenizer, # 文本预处理模块（注：常见写法为
        tokenizer）
    data_collator=data_collator, # 批量处理器，自动补齐 padding 等
    compute_metrics=compute_metrics # 自定义评估函数
)

trainer.train() # 开始训练流程
```

- **Inference**：显存节省的各种策略

5 PEET: 高效微调

- 基于 prompt-tuning 的高效微调方法
  - 设原始输入为 $[x_1, x_2, \ldots, x_n]$,插入 m 个虚拟 token 表示为 $[\mathbf{p}_1, \ldots, \mathbf{p}_m]$，则输入模型为：

$$[\mathbf{p}_1, \ldots, \mathbf{p}_m;\ x_1, x_2, \ldots, x_n]$$

  每个 $\mathbf{p}_i = \mathrm{MLP}_\theta(\mathbf{e}_i)$，只更新 prompt 参数，最小化任务损失：

$$\min_\theta \mathcal{L}(f_{\text{frozen}}([\text{Prompt}_\theta;\ X]),\ y)$$

```
from peft import PromptEncoderConfig, get_peft_model

peft_config = PromptEncoderConfig(
    task_type="CAUSAL_LM",
    num_virtual_tokens=20,
    encoder_hidden_size=128
)

model = get_peft_model(model, peft_config)
model.print_trainable_parameters()
# 输出示例: trainable params: 300,288 || all params:
    559,514,880 || trainable%: 0.053%
```

- 基于 LoRA 的高效微调方法
  - **LoRA** 基本思想：假设参数空间是低秩的，冻结原始权重 W_0，仅学习低秩矩阵更新项 $\Delta W = AB$。
  - **LoRA** 更新形式：

$$W = W_0 + \Delta W = W_0 + AB$$

  其中：

$$A \in \mathbb{R}^{d\times r}, \quad B \in \mathbb{R}^{r\times k}, \quad r \ll \min(d, k)$$

  参数总数为 $r(d + k) \ll dk$

– **LoRA** 应用于注意力机制：在 Transformer 中，注意力头计算如下：

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

其中 $Q = XW_Q$，$K = XW_K$，$V = XW_V$ 使用 LoRA 更新 W_Q, W_V，即：

$$W_Q = W_Q^{(0)} + A_Q B_Q, \quad W_V = W_V^{(0)} + A_V B_V$$

```python
from peft import LoraConfig, get_peft_model

# 配置 LoRA 参数
config = LoraConfig(
    r=16,
    lora_alpha=16,
    target_modules=["query", "value"],
    lora_dropout=0.1,
    bias="none",
    modules_to_save=["classifier"],
)

# 应用 LoRA 到模型
model = get_peft_model(model, config)

# 打印可训练参数统计
model.print_trainable_parameters()
# 输出示例:
# "trainable params: 667,493 || all params: 86,543,818 ||
    trainable%: 0.77%"
```

– 如上例所示，在完整的 8650 万参数中，仅需训练约 66 万参数（不到 1%），即可完成任务适配。