Prefix Tuning / P-Tuning v2 实战

为了不影响阅读体验，详细的代码放置在GitHub：llm-action 项目中 peft_prefix_tuning_clm.ipynb 和 peft_p_tuning_v2_clm.ipynb文件，这里仅列出关键步骤。

第一步，引进必要的库，如：Prefix Tuning / P-Tuning v2 配置类 PrefixTuningConfig。

```
from peft import get_peft_config, get_peft_model, PrefixTuningConfig, TaskType, PeftType
```

第二步，创建 Prefix Tuning / P-Tuning v2 微调方法对应的配置。

```
peft_config = PrefixTuningConfig(task_type=TaskType.CAUSAL_LM, num_virtual_tokens=30)
```

PrefixTuningConfig 配置类参数说明：

- task_type：指定任务类型。如：条件生成任务（SEQ_2_SEQ_LM），因果语言建模（CAUSAL_LM）等。
- num_virtual_tokens：虚拟token的数量，换句话说就是提示（prompt）。
- inference_mode：是否在推理模式下使用Peft模型。
- prefix_projection：是否投影前缀嵌入(token)，默认值为false，表示使用P-Tuning v2， 如果为true，则表示使用 Prefix Tuning。

第三步，通过调用 get_peft_model 方法包装基础的 Transformer 模型。

```
model = AutoModelForCausalLM.from_pretrained(model_name_or_path)
model = get_peft_model(model, peft_config)
model.print_trainable_parameters()
```

通过 print_trainable_parameters 方法可以查看到 P-Tuning v2 可训练参数的数量(仅为1,474,560)以及占比（仅为0.2629%）。

Plain Text

```
trainable params: 1,474,560 || all params: 560,689,152 || trainable%: 0.26299064191632515
```

PEFT 中 Prefix Tuning 相关的代码是基于清华开源的P-tuning-v2 进行的重构；同时，我们可以在chatglm-6b和chatglm2-6b中看到类似的代码。PEFT 中源码如下所示。

Plain Text

```
class PrefixEncoder(torch.nn.Module):
    def __init__(self, config):
        super().__init__()
        self.prefix_projection = config.prefix_projection
        token_dim = config.token_dim
        num_layers = config.num_layers
        encoder_hidden_size = config.encoder_hidden_size
        num_virtual_tokens = config.num_virtual_tokens
        if self.prefix_projection and not config.inference_mode:
            # Use a two-layer MLP to encode the prefix
            # 初始化重参数化的编码器
            self.embedding = torch.nn.Embedding(num_virtual_tokens, token_dim)
            self.transform = torch.nn.Sequential(
                torch.nn.Linear(token_dim, encoder_hidden_size),
                torch.nn.Tanh(),
                torch.nn.Linear(encoder_hidden_size, num_layers * 2 * token_dim),
            )
        else:
            self.embedding = torch.nn.Embedding(num_virtual_tokens, num_layers * 2 * token_dim)

    def forward(self, prefix: torch.Tensor):
        if self.prefix_projection:
            prefix_tokens = self.embedding(prefix)
            past_key_values = self.transform(prefix_tokens)
        else:
            past_key_values = self.embedding(prefix)
        return past_key_values
```

从上面的源码也可以看到 Prefix Tuning 与 P-Tuning v2 最主要的差别就是是否进行重新参数化编码。

第四步，模型训练的其余部分均无需更改，当模型训练完成之后，保存高效微调部分的模型权重以供模型推理即可。

```
peft_model_id = f"{model_name_or_path}_{peft_config.peft_type}_{peft_config.task_type}"
model.save_pretrained(peft_model_id)
```

输出的模型权重文件如下所示：

```
/data/nfs/llm/model/bloomz-560m_PREFIX_TUNING_CAUSAL_LM
├── [ 390]  adapter_config.json
├── [5.6M]  adapter_model.bin
└── [  93]  README.md

0 directories, 3 files
```

注意：这里只会保存经过训练的增量 PEFT 权重。其中，adapter_config.json 为 P-Tuning v2 / Prefix Tuning 配置文件；adapter_model.bin 为 P-Tuning v2 / Prefix Tuning 权重文件。

第五步，加载微调后的权重文件进行推理。

```
from peft import PeftModel, PeftConfig

peft_model_id = f"{model_name_or_path}_{peft_config.peft_type}_{peft_config.task_type}"
config = PeftConfig.from_pretrained(peft_model_id)
# 加载基础模型
model = AutoModelForCausalLM.from_pretrained(config.base_model_name_or_path)
# 加载PEFT模型
model = PeftModel.from_pretrained(model, peft_model_id)

# 编码
inputs = tokenizer(f'{text_column} : {dataset["test"][i]["Tweet text"]} Label : ', return_tensors="pt")

# 模型推理
outputs = model.generate(
        input_ids=inputs["input_ids"],
        attention_mask=inputs["attention_mask"],
        max_new_tokens=10,
        eos_token_id=3
    )

# 解码
print(tokenizer.batch_decode(outputs.detach().cpu().numpy(), skip_special_tokens=True))
```

至此，我们完成了 Prefix Tuning / P-Tuning v2 的训练及推理。

~~~python
# coding=utf-8
# Copyright 2023-present the HuggingFace Inc. team.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

# Based on https://github.com/THUDM/P-tuning-v2/blob/main/model/prefix_encoder.py
# with some refactor
import torch


class PrefixEncoder(torch.nn.Module):
    r"""
    The `torch.nn` model to encode the prefix.

    Args:
        config ([`PrefixTuningConfig`]): The configuration of the prefix encoder.

    Example:

    ```py
    >>> from peft import PrefixEncoder, PrefixTuningConfig

    >>> config = PrefixTuningConfig(
    ...     peft_type="PREFIX_TUNING",
    ...     task_type="SEQ_2_SEQ_LM",
    ...     num_virtual_tokens=20,
    ...     token_dim=768,
    ...     num_transformer_submodules=1,
    ...     num_attention_heads=12,
    ...     num_layers=12,
    ...     encoder_hidden_size=768,
    ... )
    >>> prefix_encoder = PrefixEncoder(config)
~~~

```
    ```

    Attributes:
        - **embedding** (`torch.nn.Embedding`) -- The embedding layer of the prefix encoder.
        - **transform** (`torch.nn.Sequential`) -- The two-layer MLP to transform the prefix embeddings if
          `prefix_projection` is `True`.
        - **prefix_projection** (`bool`) -- Whether to project the prefix embeddings.

    Input shape: (`batch_size`, `num_virtual_tokens`)

    Output shape: (`batch_size`, `num_virtual_tokens`, `2*layers*hidden`)
    """

    def __init__(self, config):
        super().__init__()
        self.prefix_projection = config.prefix_projection
        token_dim = config.token_dim
        num_layers = config.num_layers
        encoder_hidden_size = config.encoder_hidden_size
        num_virtual_tokens = config.num_virtual_tokens
        if self.prefix_projection and not config.inference_mode:
            # Use a two-layer MLP to encode the prefix
            self.embedding = torch.nn.Embedding(num_virtual_tokens, token_dim)
            self.transform = torch.nn.Sequential(
                torch.nn.Linear(token_dim, encoder_hidden_size),
                torch.nn.Tanh(),
                torch.nn.Linear(encoder_hidden_size, num_layers * 2 * token_dim),
)
        else:
            self.embedding = torch.nn.Embedding(num_virtual_tokens, num_layers * 2 * token_dim)

    def forward(self, prefix: torch.Tensor):
        if self.prefix_projection:
            prefix_tokens = self.embedding(prefix)
            past_key_values = self.transform(prefix_tokens)
        else:
            past_key_values = self.embedding(prefix)
        return past_key_values
```
```