P-Tuning 微调实战

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

第一步，引进必要的库，如：P-Tuning 配置类 PromptEncoderConfig。

```
from peft import (
    get_peft_config,
    get_peft_model,
    get_peft_model_state_dict,
    set_peft_model_state_dict,
    PeftType,
    TaskType,
    PromptEncoderConfig,
)
```

第二步，创建 P-Tuning 微调方法对应的配置。

```
peft_config = PromptEncoderConfig(task_type=TaskType.CAUSAL_LM, num_virtual_tokens=20, encoder_hidden_size=128)
```

P-tuning 使用提示编码器（PromptEncoder）来优化提示参数，因此，您需要使用如下几个参数初始化 PromptEncoderConfig：

- task_type：训练的任务类型，如：序列分类（SEQ_CLS），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）。
- encoder_hidden_size：编码器的隐藏大小，用于优化提示参数。
- encoder_reparameterization_type：指定如何重新参数化提示编码器，可选项有：MLP 或 LSTM，默认值为 MLP。

当使用 LSTM 时， 提示编码器结构如下：

```
(prompt_encoder): ModuleDict(
    (default): PromptEncoder(
      (embedding): Embedding(20, 1024)
      (lstm_head): LSTM(1024, 128, num_layers=2, batch_first=True, bidirectional=True)
      (mlp_head): Sequential(
        (0): Linear(in_features=256, out_features=256, bias=True)
        (1): ReLU()
        (2): Linear(in_features=256, out_features=1024, bias=True)
      )
    )
  )
```

当使用 MLP 时， 提示编码器结构如下：

```
(prompt_encoder): ModuleDict(
    (default): PromptEncoder(
      (embedding): Embedding(20, 1024)
      (mlp_head): Sequential(
        (0): Linear(in_features=1024, out_features=128, bias=True)
        (1): ReLU()
        (2): Linear(in_features=128, out_features=128, bias=True)
        (3): ReLU()
        (4): Linear(in_features=128, out_features=1024, bias=True)
      )
    )
  )
```

PEFT 中的 P-tuning 的提示编码器是基于英伟达的NeMo库中 prompt_encoder.py 进行的重构，源码如下所示。

```
class PromptEncoder(torch.nn.Module):
    def __init__(self, config):
        super().__init__()
        self.token_dim = config.token_dim
        self.input_size = self.token_dim
        self.output_size = self.token_dim
        self.hidden_size = config.encoder_hidden_size
        self.total_virtual_tokens = config.num_virtual_tokens * config.num_transformer_submodules
        self.encoder_type = config.encoder_reparameterization_type

        # 初始化 embedding 层
        self.embedding = torch.nn.Embedding(self.total_virtual_tokens, self.token_dim)
        if not config.inference_mode:
            # 根据PromptEncoder重参数化类型初始化相应的lstm和mlp
            if self.encoder_type == PromptEncoderReparameterizationType.LSTM:
                lstm_dropout = config.encoder_dropout
                num_layers = config.encoder_num_layers
                # LSTM
                self.lstm_head = torch.nn.LSTM(
                    input_size=self.input_size,
                    hidden_size=self.hidden_size,
                    num_layers=num_layers,
                    dropout=lstm_dropout,
                    bidirectional=True,
                    batch_first=True,
                )

                self.mlp_head = torch.nn.Sequential(
                    torch.nn.Linear(self.hidden_size * 2, self.hidden_size * 2),
                    torch.nn.ReLU(),
                    torch.nn.Linear(self.hidden_size * 2, self.output_size),
                )

            elif self.encoder_type == PromptEncoderReparameterizationType.MLP:
                warnings.warn(
                    f"for {self.encoder_type}, the `encoder_num_layers` is ignored. Exactly 2 MLP layers are used."
                )
                layers = [
                    torch.nn.Linear(self.input_size, self.hidden_size),
                    torch.nn.ReLU(),
                    torch.nn.Linear(self.hidden_size, self.hidden_size),
                    torch.nn.ReLU(),
                    torch.nn.Linear(self.hidden_size, self.output_size),
                ]
                self.mlp_head = torch.nn.Sequential(*layers)

            else:
                raise ValueError("Prompt encoder type not recognized. Please use one of MLP (recommended) or LSTM.")

    def forward(self, indices):
        input_embeds = self.embedding(indices)
        if self.encoder_type == PromptEncoderReparameterizationType.LSTM:
            output_embeds = self.mlp_head(self.lstm_head(input_embeds)[0])
        elif self.encoder_type == PromptEncoderReparameterizationType.MLP:
            output_embeds = self.mlp_head(input_embeds)
        else:
            raise ValueError("Prompt encoder type not recognized. Please use one of MLP (recommended) or LSTM.")

        return output_embeds
```

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 方法可以查看可训练参数的数量(仅为300,288)以及占比（仅为0.05366%）。

```
trainable params: 300,288 || all params: 559,514,880 || trainable%: 0.05366935013417338
```

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_P_TUNING_CAUSAL_LM
├── [ 451]  adapter_config.json
├── [ 81K]  adapter_model.bin
└── [ 129]  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 配置文件；adapter_model.bin 为 P-Tuning 权重文件。

第五步，加载微调后的权重文件进行推理。

Plain Text

```
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 P-Tuning 的训练及推理。

p_tuning

https://github.com/huggingface/peft/blob/main/src/peft/tuners/p_tuning/model.py

Python

~~~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/NVIDIA/NeMo/blob/main/nemo/collections/nlp/modules/common/prompt_encoder.py
# with some refactor
import warnings

import torch

from .config import PromptEncoderConfig, PromptEncoderReparameterizationType


class PromptEncoder(torch.nn.Module):
    """
    The prompt encoder network that is used to generate the virtual token embeddings for p-tuning.

    Args:
        config ([`PromptEncoderConfig`]): The configuration of the prompt encoder.

    Example:

    ```py
    >>> from peft import PromptEncoder, PromptEncoderConfig

    >>> config = PromptEncoderConfig(
    ...     peft_type="P_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_reparameterization_type="MLP",
    ...     encoder_hidden_size=768,
    ... )

    >>> prompt_encoder = PromptEncoder(config)
    ```

    Attributes:
        - **embedding** (`torch.nn.Embedding`) -- The embedding layer of the prompt encoder.
        - **mlp_head** (`torch.nn.Sequential`) -- The MLP head of the prompt encoder if `inference_mode=False`.
        - **lstm_head** (`torch.nn.LSTM`) -- The LSTM head of the prompt encoder if `inference_mode=False` and
        `encoder_reparameterization_type="LSTM"`.
        - **token_dim** (`int`) -- The hidden embedding dimension of the base transformer model.
        - **input_size** (`int`) -- The input size of the prompt encoder.
        - **output_size** (`int`) -- The output size of the prompt encoder.
        - **hidden_size** (`int`) -- The hidden size of the prompt encoder.
        - **total_virtual_tokens** (`int`): The total number of virtual tokens of the
        prompt encoder.
        - **encoder_type** (Union[[`PromptEncoderReparameterizationType`], `str`]): The encoder type of the prompt
          encoder.

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

    Output shape: (`batch_size`, `total_virtual_tokens`, `token_dim`)
    """

    def __init__(self, config):
        super().__init__()
        self.token_dim = config.token_dim
        self.input_size = self.token_dim
        self.output_size = self.token_dim
        self.hidden_size = config.encoder_hidden_size
        self.total_virtual_tokens = config.num_virtual_tokens * config.num_transformer_submodules
        self.encoder_type = config.encoder_reparameterization_type
```
```

```python
    # embedding
    self.embedding = torch.nn.Embedding(self.total_virtual_tokens, self.token_dim)
    if not config.inference_mode:
      if self.encoder_type == PromptEncoderReparameterizationType.LSTM:
        lstm_dropout = config.encoder_dropout
        num_layers = config.encoder_num_layers
        # LSTM
        self.lstm_head = torch.nn.LSTM(
          input_size=self.input_size,
          hidden_size=self.hidden_size,
          num_layers=num_layers,
          dropout=lstm_dropout,
          bidirectional=True,
          batch_first=True,
        )

        self.mlp_head = torch.nn.Sequential(
          torch.nn.Linear(self.hidden_size * 2, self.hidden_size * 2),
          torch.nn.ReLU(),
          torch.nn.Linear(self.hidden_size * 2, self.output_size),
        )

      elif self.encoder_type == PromptEncoderReparameterizationType.MLP:
        encoder_num_layers_default = PromptEncoderConfig.encoder_num_layers
        if config.encoder_num_layers != encoder_num_layers_default:
          warnings.warn(
            f"for {self.encoder_type.value}, the argument `encoder_num_layers` is ignored. "
            f"Exactly {encoder_num_layers_default} MLP layers are used."
          )
        layers = [
          torch.nn.Linear(self.input_size, self.hidden_size),
          torch.nn.ReLU(),
          torch.nn.Linear(self.hidden_size, self.hidden_size),
          torch.nn.ReLU(),
          torch.nn.Linear(self.hidden_size, self.output_size),
        ]
        self.mlp_head = torch.nn.Sequential(*layers)
```

```
            else:
                raise ValueError("Prompt encoder type not recognized. Please use one of MLP (recommended) or LSTM.")

    def forward(self, indices):
        input_embeds = self.embedding(indices)
        if self.encoder_type == PromptEncoderReparameterizationType.LSTM:
            output_embeds = self.mlp_head(self.lstm_head(input_embeds)[0])
        elif self.encoder_type == PromptEncoderReparameterizationType.MLP:
            output_embeds = self.mlp_head(input_embeds)
        else:
            raise ValueError("Prompt encoder type not recognized. Please use one of MLP (recommended) or LSTM.")

        return output_embeds
```