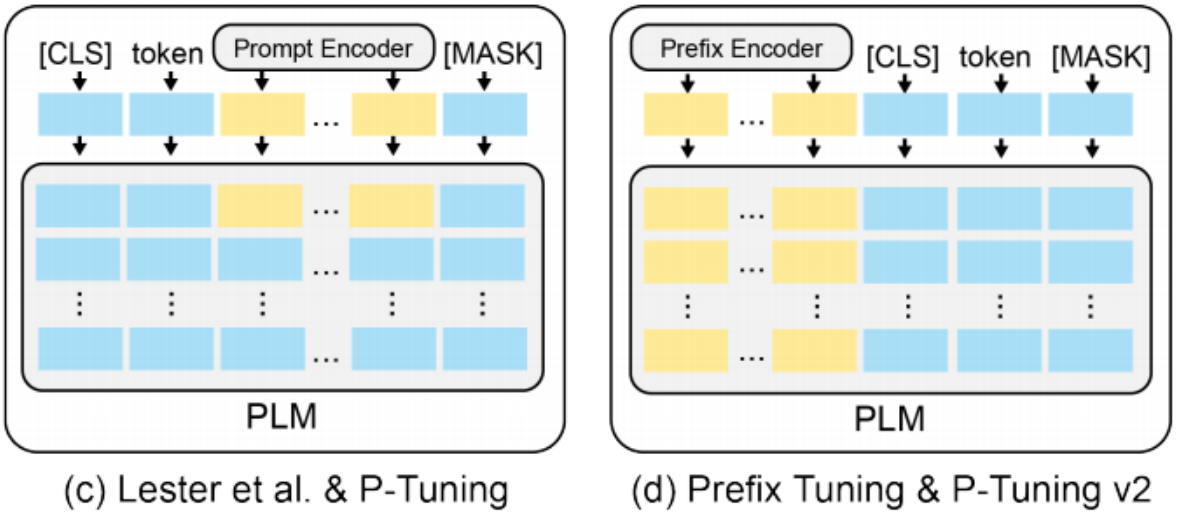
Parameter Efficient Fine-tuning

LoRA

P-tuning&P-tuning-V2



Prefix-tuning和P-tuning的差异：

1. PrefixTuning是为自然语言生成（NLG）和GPTs设计的，而P-tuning则针对自然语言理解（NLU）和所有类型的语言模
2. PrefixTuning只允许在输入序列开头添加提示符标记，而P-tuning可以在任何地方插入这些标记。
3. PrefixTuning在transformer的每一层中侵入式地连接连续的提示符标记，因为作者发现仅在输入中添加提示符无效；P-tuning非侵入式地仅在输入中添加连续提示符可行。

P-tuning-v2的改进：与P-tuning不同，第二版在模型每一层都插入了virtual token，并移除了Lstm作为prompt encoder。话不多说，直接上代码解读：P-tuning v2

主要代码有两块：model/prefix_encoder.py

```
class PrefixEncoder(torch.nn.Module):
    r'''
    The torch.nn model to encode the prefix
    Input shape: (batch-size, prefix-length)
    Output shape: (batch-size, prefix-length, 2*layers*hidden)
    '''
    def __init__(self, config):
        super().__init__()
        self.prefix_projection = config.prefix_projection
        if self.prefix_projection:
            # Use a two-layer MLP to encode the prefix
            self.embedding = torch.nn.Embedding(config.pre_seq_len,
config.hidden_size)
            self.trans = torch.nn.Sequential(
                torch.nn.Linear(config.hidden_size, config.prefix_hidden_size),
                torch.nn.Tanh(),
```

```
                torch.nn.Linear(config.prefix_hidden_size,
config.num_hidden_layers * 2 * config.hidden_size)
            )
        else:
            self.embedding = torch.nn.Embedding(config.pre_seq_len,
config.num_hidden_layers * 2 * config.hidden_size)

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

和P-Tuning相比，该方法的Prompt Encoder移除了LSTM网络，可以选择通过前馈神经网络编码或者直接通过嵌入层得到virtual token embedding。

model/sequence_classification.py

```
class BertPrefixForSequenceClassification(BertPreTrainedModel):
    def __init__(self, config):
        super().__init__(config)
        self.num_labels = config.num_labels
        self.config = config
        self.bert = BertModel(config)
        self.dropout = torch.nn.Dropout(config.hidden_dropout_prob)
        self.classifier = torch.nn.Linear(config.hidden_size, config.num_labels)

        for param in self.bert.parameters():
            param.requires_grad = False

        self.pre_seq_len = config.pre_seq_len
        self.n_layer = config.num_hidden_layers
        self.n_head = config.num_attention_heads
        self.n_embd = config.hidden_size // config.num_attention_heads

        self.prefix_tokens = torch.arange(self.pre_seq_len).long()
        self.prefix_encoder = PrefixEncoder(config)

        bert_param = 0
        for name, param in self.bert.named_parameters():
            bert_param += param.numel()
        all_param = 0
        for name, param in self.named_parameters():
            all_param += param.numel()
        total_param = all_param - bert_param
        print('total param is {}'.format(total_param)) # 9860105

    def get_prompt(self, batch_size):
        prefix_tokens = self.prefix_tokens.unsqueeze(0).expand(batch_size,
-1).to(self.bert.device)
        past_key_values = self.prefix_encoder(prefix_tokens)
        # bsz, seqlen, _ = past_key_values.shape
        past_key_values = past_key_values.view(
```

```
            batch_size,
            self.pre_seq_len,
            self.n_layer * 2,
            self.n_head,
            self.n_embd
        )
        past_key_values = self.dropout(past_key_values)
        past_key_values = past_key_values.permute([2, 0, 3, 1, 4]).split(2)
        return past_key_values

    def forward(
        self,
        input_ids=None,
        attention_mask=None,
        token_type_ids=None,
        position_ids=None,
        head_mask=None,
        inputs_embeds=None,
        labels=None,
        output_attentions=None,
        output_hidden_states=None,
        return_dict=None,
    ):
        return_dict = return_dict if return_dict is not None else
self.config.use_return_dict
        batch_size = input_ids.shape[0]
        past_key_values = self.get_prompt(batch_size=batch_size)
        prefix_attention_mask = torch.ones(batch_size,
self.pre_seq_len).to(self.bert.device)
        attention_mask = torch.cat((prefix_attention_mask, attention_mask),
dim=1)

        outputs = self.bert(
            input_ids,
            attention_mask=attention_mask,
            token_type_ids=token_type_ids,
            position_ids=position_ids,
            head_mask=head_mask,
            inputs_embeds=inputs_embeds,
            output_attentions=output_attentions,
            output_hidden_states=output_hidden_states,
            return_dict=return_dict,
            past_key_values=past_key_values,
        )

        pooled_output = outputs[1]

        pooled_output = self.dropout(pooled_output)
        logits = self.classifier(pooled_output)

        loss = None
        if labels is not None:
            if self.config.problem_type is None:
                if self.num_labels == 1:
                    self.config.problem_type = "regression"
```

```
                elif self.num_labels > 1 and (labels.dtype == torch.long or
labels.dtype == torch.int):
                    self.config.problem_type = "single_label_classification"
                else:
                    self.config.problem_type = "multi_label_classification"

            if self.config.problem_type == "regression":
                loss_fct = MSELoss()
                if self.num_labels == 1:
                    loss = loss_fct(logits.squeeze(), labels.squeeze())
                else:
                    loss = loss_fct(logits, labels)
            elif self.config.problem_type == "single_label_classification":
                loss_fct = CrossEntropyLoss()
                loss = loss_fct(logits.view(-1, self.num_labels),
labels.view(-1))
            elif self.config.problem_type == "multi_label_classification":
                loss_fct = BCEWithLogitsLoss()
                loss = loss_fct(logits, labels)
        if not return_dict:
            output = (logits,) + outputs[2:]
            return ((loss,) + output) if loss is not None else output

        return SequenceClassifierOutput(
            loss=loss,
            logits=logits,
            hidden_states=outputs.hidden_states,
            attentions=outputs.attentions,
        )
```

我们直接将重心放到get_prompt和forward函数。首先由prefix encoder得到对应的virtual token的语义向量，然后改变形状得到past_key_values:[Batch,prefix_len,2*num_lauyers,num_heads,n_emb]，假设对应的Batch,prefix_len,num_lauyers,num_heads,n_emb分别是32, 128, 12, 12, 64，那么有past_kv:[32,128,24,12,64]。再通过permute函数变成past_kv:[24,32,12,128,64]，同时split(2)会把past_kv在第0个维度进行划分得到一个长度为12的元组。其中每一个元素形状为[2,32,12,128,64]，对应于BERT模型每一层的past_key_values。

接下来再看forward函数，attention_mask会由prefix_attention_mask和原始的attention_mask合并得到（显而易见），然后再把past_key_values传入bert模型即可。最后取 BERT 模型的[CLS]位置对应的输出接全连接层完成分类任务即可。整体比较简单，无需过多解析。

在transformers/models/bert/modeling_bert.py中的BertSelfAttention中对past_key_values的处理如下：

```
    def forward(
        self,
        hidden_states: torch.Tensor,
        attention_mask: Optional[torch.FloatTensor] = None,
        head_mask: Optional[torch.FloatTensor] = None,
        encoder_hidden_states: Optional[torch.FloatTensor] = None,
        encoder_attention_mask: Optional[torch.FloatTensor] = None,
        past_key_value: Optional[Tuple[Tuple[torch.FloatTensor]]] = None,
        output_attentions: Optional[bool] = False,
    ) -> Tuple[torch.Tensor]:
```

```
        mixed_query_layer = self.query(hidden_states)

        # If this is instantiated as a cross-attention module, the keys
        # and values come from an encoder; the attention mask needs to be
        # such that the encoder's padding tokens are not attended to.
        is_cross_attention = encoder_hidden_states is not None

        if is_cross_attention and past_key_value is not None:
            # reuse k,v, cross_attentions
            key_layer = past_key_value[0]
            value_layer = past_key_value[1]
            attention_mask = encoder_attention_mask
        elif is_cross_attention:
            key_layer =
self.transpose_for_scores(self.key(encoder_hidden_states))
            value_layer =
self.transpose_for_scores(self.value(encoder_hidden_states))
            attention_mask = encoder_attention_mask
        elif past_key_value is not None:
            key_layer = self.transpose_for_scores(self.key(hidden_states)) #
[Batch,num_heads,seqlen,n_em]
            value_layer = self.transpose_for_scores(self.value(hidden_states))
            #[Batch,num_heads,seqlen,n_em]
            key_layer = torch.cat([past_key_value[0], key_layer], dim=2)
            #[Batch,num_heads,seqlen+prefix_len,n_em]
            value_layer = torch.cat([past_key_value[1], value_layer], dim=2)
            #[Batch,num_heads,seqlen+prefix_len,n_em]
        else:
            key_layer = self.transpose_for_scores(self.key(hidden_states))
            value_layer = self.transpose_for_scores(self.value(hidden_states))

        query_layer = self.transpose_for_scores(mixed_query_layer)
```

IA3