Prefix Tuning详解及代码实战

1. 背景

2. 技术细节

3. 优点

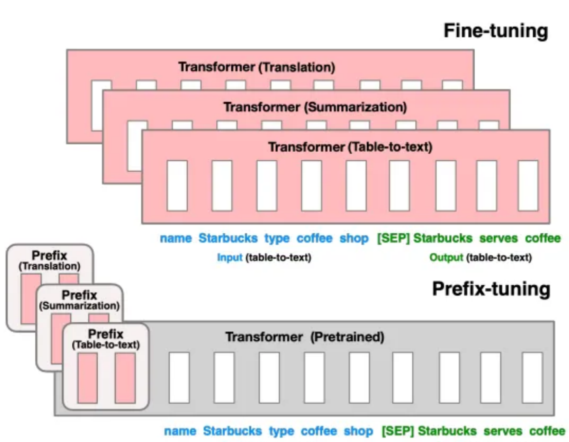
1. 背景

在Prefix Tuning之前的工作主要是人工设计离散的模版或者自动化搜索离散的模版存在缺陷：

- 人工离散模版缺点：模版的变化对模型最终的性能特别敏感，加一个少一个词或变动位置都会造成比较大的变化。
- 自动化搜索模版缺点：成本也比较高；同时，以前这种离散化的token搜索出来的结果可能并不是最优的。除此之外，传统的微调范式利用预训练模型去对不同的下游任务进行微调，对每个任务都要保存一份微调后的模型权重，一方面微调整个模型耗时长；另一方面也会占很多存储空间。

基于上述两点，Prefix Tuning优化：

- 提出固定预训练LM，为LM添加可训练，特定任务的前缀，可以为不同任务保存不同的前缀，微调成本也小；
- 同时，这种Prefix实际就是连续可微的Virtual Token（Soft Prompt/Continuous Prompt），相比离散的Token，更好优化，效果更好。



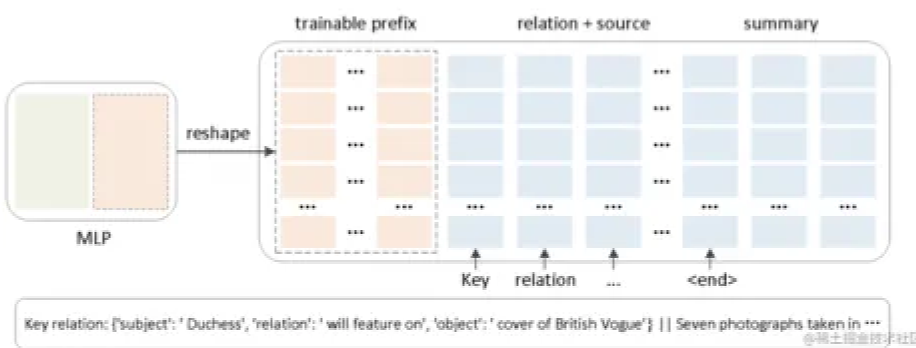
2. 技术细节

2021年斯坦福的研究人员在论文《Prefix-Tuning: Optimizing Continuous Prompts for Generation》中提出了 Prefix Tuning 方法。

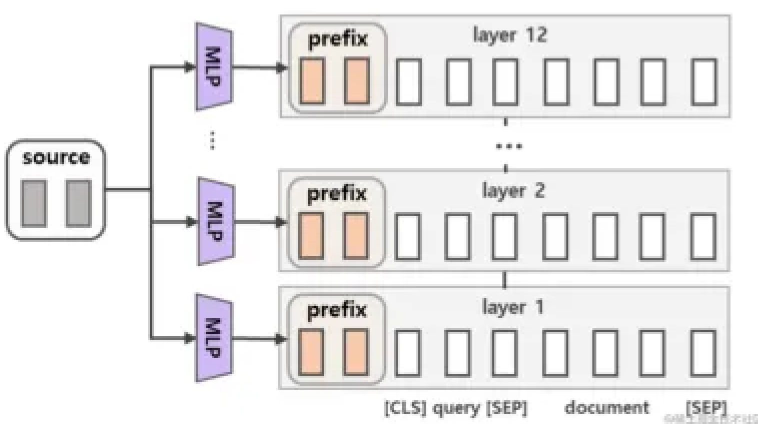
细节：

- 与Full-finetuning 更新所有参数的方式不同，该方法是**在输入 token 之前构造一段任务相关的 virtual tokens**（虚拟tokens不是真实的tokens，而是可学习的自由参数）**作为 Prefix**，然后训练的时候只更新 Prefix 部分的参数，而 Transformer 中的其他部分参数固定。

- **为了防止直接更新Prefix的参数导致训练不稳定和性能下降的情况，在Prefix层前面加了MLP结构，训练完成后，只保留Prefix的参数。**



- **通过消融实验证实，只调整embedding层的表现力不够，将导致性能显著下降，因此，在每层（所有layer的输入层）都加了prefix token，改动较大。**



针对不同的模型结构，需要构造不同的Prefix。

- 针对encoder-decoder架构，会在encoder的输入和decoder的输入embedding都加上prefix token；
    - **针对自回归架构模型：** 在句子前面添加前缀，得到 z = [PREFIX; x; y]，合适的上文能够在固定 LM 的情况下去引导生成下文（比如：GPT3的上下文学习）。
    - **针对编码器-解码器架构模型：** Encoder和Decoder都增加了前缀，得到 z = [PREFIX; x;

PREFIX0; y]。Encoder端增加前缀是为了引导输入部分的编码，Decoder 端增加前缀是为了引导后续token的生成。

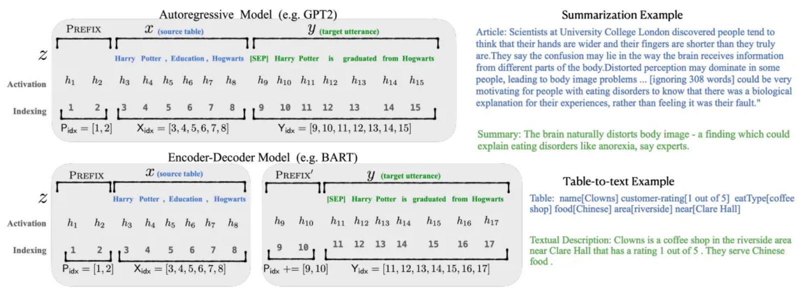


Figure 2: An annotated example of prefix-tuning using an autoregressive LM (top) and an encoder-decoder model (bottom). The prefix activations $\forall i \in \mathsf{P}_{\text{idx}}, h_i$ are drawn from a trainable matrix P_θ. The remaining activations are computed by the Transformer.

image.png

3. 优点

Prefix-Tuning只需训练和存储0.1%的新增参数（VS adapter 3.6%，fine-tuning 100%）。

代码实践

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

训练

Python

```python
from transformers import AutoModelForCausalLM
from peft import get_peft_config, get_peft_model, PrefixTuningConfig, TaskType, PeftType
import torch
from datasets import load_dataset
import os
from transformers import AutoTokenizer
from torch.utils.data import DataLoader
from transformers import default_data_collator, get_linear_schedule_with_warmup
from tqdm import tqdm
from datasets import load_dataset


device = "cuda"

model_name_or_path = "/data/nfs/llm/model/bloomz-560m"
tokenizer_name_or_path = "/data/nfs/llm/model/bloomz-560m"

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

dataset_name = "twitter_complaints"
checkpoint_name = f"{dataset_name}_{model_name_or_path}_{peft_config.peft_type}_{peft_config.task_type}_v1.pt".replace("/", "_")
text_column = "Tweet text"
label_column = "text_label"
max_length = 64
lr = 3e-2
num_epochs = 10
batch_size = 8


from datasets import load_dataset

# dataset = load_dataset("ought/raft", dataset_name)
dataset = load_dataset("/home/guodong.li/data/peft/raft/raft.py", dataset_name, cache_dir="/home/guodong.li/data/peft/data")

classes = [k.replace("_", " ") for k in dataset["train"].features["Label"].names]
print(classes)
dataset = dataset.map(
    lambda x: {"text_label": [classes[label] for label in x["Label"]]},
```

```python
        batched=True,
        num_proc=1,
)
print(dataset)
dataset["train"][0]


# data preprocessing
tokenizer = AutoTokenizer.from_pretrained(model_name_or_path)
if tokenizer.pad_token_id is None:
    tokenizer.pad_token_id = tokenizer.eos_token_id
target_max_length = max([len(tokenizer(class_label)["input_ids"]) for class_label in classes])
print("target_max_length:", target_max_length)


def preprocess_function(examples):
    batch_size = len(examples[text_column])
    inputs = [f"{text_column} : {x} Label : " for x in examples[text_column]]
    targets = [str(x) for x in examples[label_column]]
    model_inputs = tokenizer(inputs)
    labels = tokenizer(targets)
    for i in range(batch_size):
        sample_input_ids = model_inputs["input_ids"][i]
        label_input_ids = labels["input_ids"][i] + [tokenizer.pad_token_id]
        # print(i, sample_input_ids, label_input_ids)
        model_inputs["input_ids"][i] = sample_input_ids + label_input_ids
        labels["input_ids"][i] = [-100] * len(sample_input_ids) + label_input_ids
        model_inputs["attention_mask"][i] = [1] * len(model_inputs["input_ids"][i])
    # print(model_inputs)
    for i in range(batch_size):
        sample_input_ids = model_inputs["input_ids"][i]
        label_input_ids = labels["input_ids"][i]
        model_inputs["input_ids"][i] = [tokenizer.pad_token_id] * (
            max_length - len(sample_input_ids)
        ) + sample_input_ids
        model_inputs["attention_mask"][i] = [0] * (max_length - len(sample_input_ids)) + model_inputs[
            "attention_mask"
        ][i]
        labels["input_ids"][i] = [-100] * (max_length - len(sample_input_ids)) + label_input_ids
        model_inputs["input_ids"][i] = torch.tensor(model_inputs["input_ids"][i][:max_length])
```

```
        model_inputs["attention_mask"][i] = torch.tensor(model_inputs["attention_mask"][i][:max_length])
        labels["input_ids"][i] = torch.tensor(labels["input_ids"][i][:max_length])
    model_inputs["labels"] = labels["input_ids"]
    return model_inputs


processed_datasets = dataset.map(
    preprocess_function,
    batched=True,
    num_proc=1,
    remove_columns=dataset["train"].column_names,
    load_from_cache_file=False,
    desc="Running tokenizer on dataset",
)

train_dataset = processed_datasets["train"]
eval_dataset = processed_datasets["train"]


train_dataloader = DataLoader(train_dataset, shuffle=True, collate_fn=default_data_collator, batch_size=batch_size, pin_memory=True)
eval_dataloader = DataLoader(eval_dataset, collate_fn=default_data_collator, batch_size=batch_size, pin_memory=True)

def test_preprocess_function(examples):
    batch_size = len(examples[text_column])
    inputs = [f"{text_column} : {x} Label : " for x in examples[text_column]]
    model_inputs = tokenizer(inputs)
    # print(model_inputs)
    for i in range(batch_size):
        sample_input_ids = model_inputs["input_ids"][i]
        model_inputs["input_ids"][i] = [tokenizer.pad_token_id] * (max_length - len(sample_input_ids)) + sample_input_ids
        model_inputs["attention_mask"][i] = [0] * (max_length - len(sample_input_ids)) + model_inputs["attention_mask"][i]

        model_inputs["input_ids"][i] = torch.tensor(model_inputs["input_ids"][i][:max_length])
        model_inputs["attention_mask"][i] = torch.tensor(model_inputs["attention_mask"][i][:max_length])
    return model_inputs


test_dataset = dataset["test"].map(
    test_preprocess_function,
```

```
    batched=True,
    num_proc=1,
    remove_columns=dataset["train"].column_names,
    load_from_cache_file=False,
    desc="Running tokenizer on dataset",
)

test_dataloader = DataLoader(test_dataset, collate_fn=default_data_collator, batch_size=batch_size, pin_memory=True)
next(iter(test_dataloader))


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

# model
# optimizer and lr scheduler
optimizer = torch.optim.AdamW(model.parameters(), lr=lr)
lr_scheduler = get_linear_schedule_with_warmup(
    optimizer=optimizer,
    num_warmup_steps=0,
    num_training_steps=(len(train_dataloader) * num_epochs),
)


# training and evaluation
model = model.to(device)

for epoch in range(num_epochs):
    model.train()
    total_loss = 0
    for step, batch in enumerate(tqdm(train_dataloader)):
        batch = {k: v.to(device) for k, v in batch.items()}
        #         print(batch)
        #         print(batch["input_ids"].shape)
        outputs = model(**batch)
        loss = outputs.loss
        total_loss += loss.detach().float()
        loss.backward()
        optimizer.step()
        lr_scheduler.step()
        optimizer.zero_grad()

    model.eval()
    eval_loss = 0
    eval_preds = []
```

```python
    for step, batch in enumerate(tqdm(eval_dataloader)):
        batch = {k: v.to(device) for k, v in batch.items()}
        with torch.no_grad():
            outputs = model(**batch)
        loss = outputs.loss
        eval_loss += loss.detach().float()
        eval_preds.extend(
            tokenizer.batch_decode(torch.argmax(outputs.logits, -1).detach().cpu().numpy(), skip_special_tokens=True)
        )

    eval_epoch_loss = eval_loss / len(eval_dataloader)
    eval_ppl = torch.exp(eval_epoch_loss)
    train_epoch_loss = total_loss / len(train_dataloader)
    train_ppl = torch.exp(train_epoch_loss)
    print(f"{epoch=}: {train_ppl=} {train_epoch_loss=} {eval_ppl=} {eval_epoch_loss=}")
```

推理 Python

```python
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))
```