LoRA详解介绍及代码实战

LORA

1. 背景

论文标题:LoRA: Low-Rank Adaptation of Large Language Models

论文链接:https://arxiv.org/pdf/2106.09685.pdf

Aghajanyan的研究表明：**预训练模型拥有极小的内在维度(instrisic dimension)，即存在一个极低维度的参数，微调它和在全参数空间中微调能起到相同的效果**。同时发现在预训练后，越大的模型有越小的内在维度，这也解释了为何大模型都拥有很好的few-shot能力。

2. 出发点

随着大模型的发展，尤其是chatGPT出现之后，175B的参数量，训练起来非常昂贵。

微软提出了低秩自适应（LoRA），**LoRA的主要思想很简单，冻结预训练模型的权重参数，**在原始的预训练模型（PLM）旁边增加一个新的通路，通过前后两个矩阵A,B相乘，第一个矩阵A负责降维，第二个矩阵B负责升维，中间层维度为r，**在微调下游任务的时候，只更新A和B**，该方法的核心思想就是**通过低秩分解来模拟参数的改变量**，从而以极小的参数量来实现大模型的间接训练。

LoRA方法优点：

- **预训练模型可以共享**，针对下游任务可以**构建多个不同任务的LoRA模块**。冻结预训练模型参数共享，通过替换矩阵A和B来高效地切换任务，从而**显著降低存储需求和任务切换开销**。

- LoRA使训练更加的高效，将**硬件的进入门槛降低了3倍**，相同的内存下，可以微调更大参数的模型
- 线性设计允许我们在部署时将可训练矩阵与冻结权重合并，和完全微调的模型相比，**不会引入推理延迟。**

3. 具体实现方法

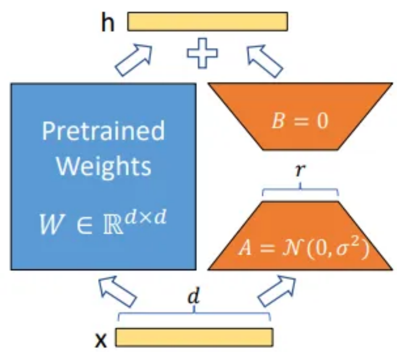


Figure 1: Our reparametrization. We only train A and B.

3.1. 公式

LoRA中是让模型学习BA，去近似SVD分解的结果

$$h = W_0x + \Delta Wx = W_0x + BAx$$

- 在训练过程中W_0被冻结，不接收梯度更新，而A和B包含可训练参数。
- 在初始化的时候，我们对使用A随机高斯初始化，对B使用零初始化。
- 因此，在训练开始时为0，然后我们可以通过$\frac{\alpha}{r}$对ΔW进行缩放， r是秩 在推理时，我们通过上图可知，将左右两部分进行相加即可，不会添加额外的计算资源。

在微调过程中，所有做lora适配器的module，它们的 r 都是一致的，且在训练过程中不会改变。

Transformer的权重矩阵包括Attention模块里用于计算query, key, value的Wq，Wk，Wv以及多头attention的Wo,以及MLP层的权重矩阵，**在LoRA原始论文中，作者**通过消融实验发现**最终选择对attention模块的 W_q, W_v 做低秩适配**产生最佳结果。

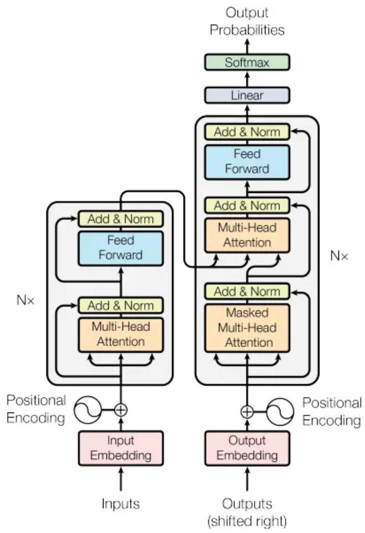


Figure 1: The Transformer - model architecture.

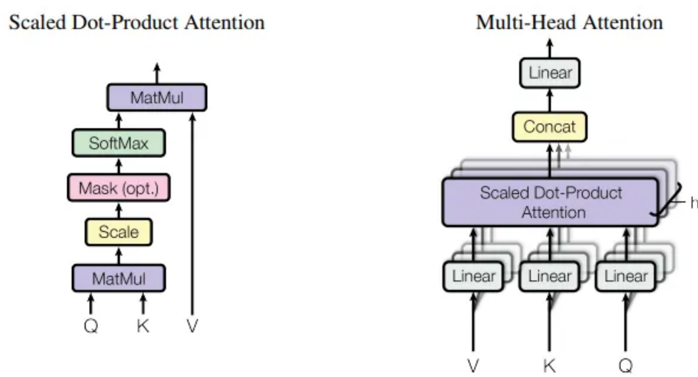
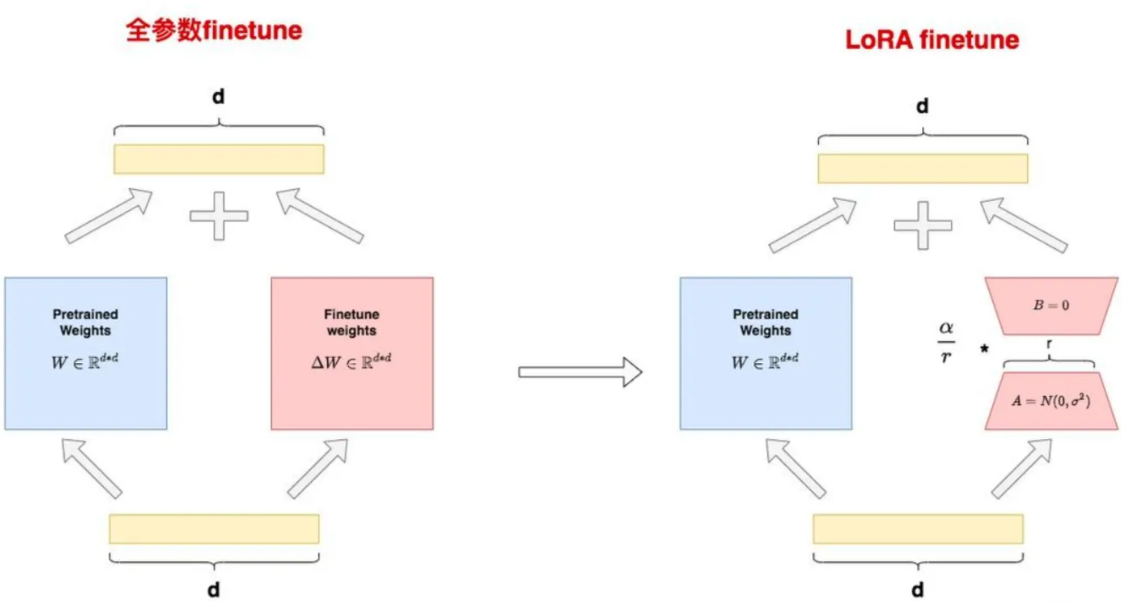


Figure 2: (left) Scaled Dot-Product Attention. (right) Multi-Head Attention consists of several attention layers running in parallel.



4. 4.代码实战

训练 Python

```python
from transformers import AutoModelForCausalLM
from peft import get_peft_config, get_peft_model, get_peft_model_state_dict, LoraConfig, TaskType

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"

####  加载配置 ################
peft_config = LoraConfig(task_type=TaskType.CAUSAL_LM,
                         inference_mode=False, r=8,
                         lora_alpha=32,
                         lora_dropout=0.1)

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]
```

```python
print(classes)
dataset = dataset.map(
    lambda x: {"text_label": [classes[label] for label in x["Label"]]},
    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
```

```python
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))

#####  加载模型 ############
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 = []
    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=}")

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)
```

推理

```python

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)

# tokenizer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
    )

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

```