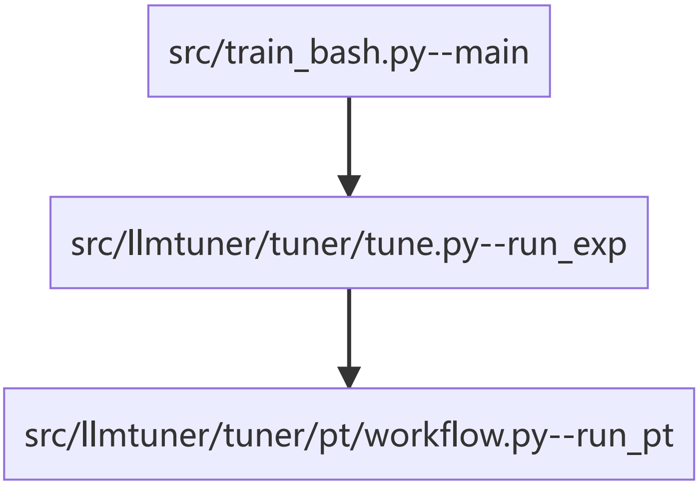
LLaMA-Efficient-Tuning

链接：https://github.com/hiyouga/LLaMA-Efficient-Tuning

```
LLaMA-Efficient-Tuning:
- assets: 微信群二维码
- data: 各种数据集 pt sft self-cognition multiturn rlhf
- src: 训练、推理代码
    - llmtuner: 与微调相关的各种代码
        - api: 创建FastAPI app的启动代码和数据对象定义
        - chat: 利用模型进行问答，返回形式包括流式和普通方式
        - dsets: 数据集的校验，加载，预处理
        - extras: 工具代码，包含模型名字定义，模板定义，日志，显存管理，可视化等
        - hparams: 阶段参数，数据参数，模型参数，微调参数，生成参数
        - tuner: 核心代码，包含dpo/ppo/pt/rm/sft
        - webui: 模型微调与模型部署展示UI代码
    - api_demo.py: 提供FastAPI接口
    - cli_demo.py: 提供命令行接口
    - export_model.py: 导出模型为单个或多个文件
    - train_bash.py: 利用命令行命令训练模型
    - train_web.py: 利用gradio Web UI来训练模型
    - web_demo.py: 利用gradio Web UI来部署模型进行聊天
- tests: 测试代码
    - evaluate_zh.py: 中文评测，包含选择、填空、开放问答
    - modeling_baichuan.py: 百川模型代码
    - quantize.py: 将模型利用GPTQ算法量化至4bit
    - template_encode.py: 测试模板编码
```

预训练代码解析

```
CUDA_VISIBLE_DEVICES=0 python src/train_bash.py \
    --stage pt \
    --model_name_or_path path_to_llama_model \
    --do_train \
    --dataset wiki_demo \
    --template default \
    --finetuning_type lora \
    --lora_target q_proj,v_proj \
    --output_dir path_to_pt_checkpoint \
    --overwrite_cache \
    --per_device_train_batch_size 4 \
    --gradient_accumulation_steps 4 \
    --lr_scheduler_type cosine \
    --logging_steps 10 \
    --save_steps 1000 \
    --learning_rate 5e-5 \
    --num_train_epochs 3.0 \
    --plot_loss \
    --fp16
```




```python
#加载数据集
dataset = get_dataset(model_args, data_args)

#加载模型与分词器
model, tokenizer = load_model_and_tokenizer(model_args, finetuning_args, training_args.do_train, stage="pt")

#处理数据集
dataset = preprocess_dataset(dataset, tokenizer, data_args, training_args, stage="pt")

#获取数据收集器
data_collator = DataCollatorForLanguageModeling(tokenizer=tokenizer, mlm=False)

# 初始化训练器
trainer = PeftTrainer(
    finetuning_args=finetuning_args,
    model=model,
    args=training_args,
    tokenizer=tokenizer,
    data_collator=data_collator,
    callbacks=callbacks,
    **split_dataset(dataset, data_args, training_args)
)

# 训练
if training_args.do_train:
    train_result = trainer.train(resume_from_checkpoint=training_args.resume_from_checkpoint)
    trainer.log_metrics("train", train_result.metrics)
    trainer.save_metrics("train", train_result.metrics)
    trainer.save_state()
    trainer.save_model()
    if trainer.is_world_process_zero() and model_args.plot_loss:
        plot_loss(training_args.output_dir, keys=["loss", "eval_loss"])

# 评估，使用困惑度
```

```python
if training_args.do_eval:
    metrics = trainer.evaluate(metric_key_prefix="eval")
    try:
        perplexity = math.exp(metrics["eval_loss"])
    except OverflowError:
        perplexity = float("inf")

    metrics["perplexity"] = perplexity
    trainer.log_metrics("eval", metrics)
    trainer.save_metrics("eval", metrics)
```

GPT

```python
class ResidualAttentionBlock(nn.Module):
    def __init__(self, d_model: int, n_head: int, attn_mask: torch.Tensor = None):
        super().__init__()
        #多头注意力层
        self.attn = nn.MultiheadAttention(d_model, n_head)
        #层规范化
        self.ln_1 = LayerNorm(d_model)
        #FFN层
        self.mlp = nn.Sequential(OrderedDict([
            ("c_fc", nn.Linear(d_model, d_model * 4)),
            ("gelu", QuickGELU()),
            ("c_proj", nn.Linear(d_model * 4, d_model))
        ]))
        #层规范化
        self.ln_2 = LayerNorm(d_model)
        #mask掉上三角的所有词语
        self.attn_mask = attn_mask

    def attention(self, x: torch.Tensor):
        self.attn_mask = self.attn_mask.to(dtype=x.dtype, device=x.device) if self.attn_mask is not None else None
        return self.attn(x, x, x, need_weights=False, attn_mask=self.attn_mask)[0]

    def forward(self, x: torch.Tensor):
        x = x + self.attention(self.ln_1(x))
        x = x + self.mlp(self.ln_2(x))
        return x

class Transformer(nn.Module):
    def __init__(self, width: int, layers: int, heads: int, attn_mask: torch.Tensor = None):
        super().__init__()
        self.width = width
        self.layers = layers
        self.resblocks = nn.Sequential(*[ResidualAttentionBlock(width, heads, attn_mask) for _ in range(layers)])

    def forward(self, x: torch.Tensor):
        return self.resblocks(x)
```

```


def build_attention_mask(self):
    # lazily create causal attention mask, with full attention between the vision tokens
    # pytorch uses additive attention mask; fill with -inf
    # 相当于将上三角（不包括对角线）的所有数值设为负无穷
    mask = torch.empty(self.context_length, self.context_length)
    mask.fill_(float("-inf"))
    mask.triu_(1)  # zero out the lower diagonal
    return mask

self.transformer = Transformer(
    width=transformer_width,
    layers=transformer_layers,
    heads=transformer_heads,
    attn_mask=self.build_attention_mask()
)

def encode_text(self, text):
    # 词语编码
    x = self.token_embedding(text).type(self.dtype)  # [batch_size, n_ctx, d_model]
    # 位置编码加上词语编码得到输入
    x = x + self.positional_embedding.type(self.dtype)
    #将token长度维度放在第一维，Batchsize放在第二维
    x = x.permute(1, 0, 2)  # NLD -> LND
    #输入Transformer得到输出
    x = self.transformer(x)
    #将token长度维度还原回第二维
    x = x.permute(1, 0, 2)  # LND -> NLD
    #层规范化
    x = self.ln_final(x).type(self.dtype)
    # x.shape = [batch_size, n_ctx, transformer.width]
    # take features from the eot embedding (eot_token is the highest number in each sequence)
    x = x[torch.arange(x.shape[0]), text.argmax(dim=-1)] @ self.text_projection

    return x


```

permute理解

先将一个batch里同样位置的向量放在一起，这样直接使用第二维度的矩阵做乘法取对角线即为点积结果

最后再还原回来得到每个位置的输出向量

```
In [17]: x=torch.arange(24).reshape(2,3,4)
In [18]: x
Out[18]:
tensor([[[ 0,  1,  2,  3],
         [ 4,  5,  6,  7],
         [ 8,  9, 10, 11]],

        [[12, 13, 14, 15],
         [16, 17, 18, 19],
         [20, 21, 22, 23]]])
In [19]: x.permute(1,0,2)
Out[19]:
tensor([[[ 0,  1,  2,  3],
         [12, 13, 14, 15]],

        [[ 4,  5,  6,  7],
         [16, 17, 18, 19]],

        [[ 8,  9, 10, 11],
         [20, 21, 22, 23]]])
```