西安电子科技大学
XIDIAN UNIVERSITY

【MindSpore开源实习】共享参数预训练初始化

目 录

CONTENT

壹 项目介绍

贰 项目方案

叁 实验

肆 接口设计

背景描述：

训练Transformer模型的时间长，资源耗费多，因此减少Transformer模型的训练时间，加快其收敛速度非常重要。现有一些工作发现，在训练Transformer的过程中，不同层之间的参数具有非常高的相似性，且在训练过程中某些层的参数收敛较快，因此，可以通过共享不同层之间的参数的方式，加快其训练过程。

需求描述：

1、共享不同层之间的参数，加快模型训练过程。

2、实现训练策略API接口，集成到MindSpore中。

方案：

Efficient Training of BERT by Progressively Stacking

创新点：

提出了将知识从浅层模型转移到深层模型的堆叠算法；然后我们逐步应用叠加来加速BERT训练

Stacking算法：

如果我们有一个L层训练BERT，我们可以通过复制构建一个2L层BERT：对于i ≤ L，构造的BERT的第i层和(i+L)层与训练后的BERT的第i层具有相同的参数。

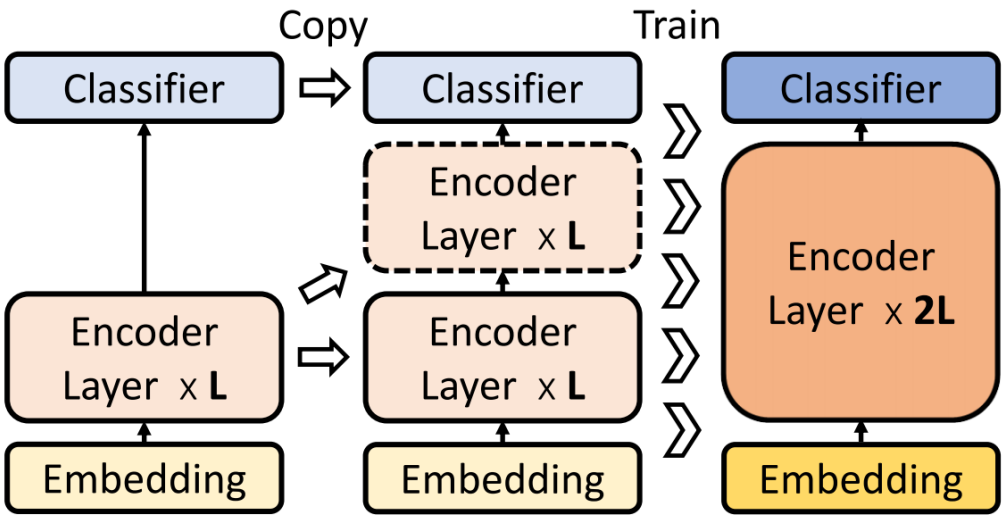


Figure 3. The diagram of the *stacking* algorithm.

方案

Algorithm 1 Progressive stacking

$M'_0 \leftarrow \text{InitBERT}(L/2^k)$
$M_0 \leftarrow \text{Train}(M'_0)$ {Train from scratch.}
for $i \leftarrow 1$ to k **do**
  $M'_i \leftarrow \text{Stack}(M_i)$ {Doubles the number of layers.}
  $M_i \leftarrow \text{Train}(M'_i)$ {M_i has $L/2^{k-i}$ layers.}
end for
return M_k

Progressive Stacking算法：

由于浅层模型通常比深度模型训练得更快（对于相同的步数），如果我们从浅层模型叠加来训练深度模型，训练时间将大大减少。同样，我们可以通过从较浅的模型堆叠来更快地训练这个浅层模型。通过递归，我们设计了一种基于堆叠技术的迭代训练算法，以更快地训练深度BERT。我们称这种算法为渐进式叠加。

实验：

1、Bert模型的预训练加速

2、GPT模型的预训练加速

数据集预处理：

1、下载zhwiki或enwiki数据集进行预训练

enwik网址：

https://dumps.wikimedia.org/enwiki/latest/enwiki-latest-pages-articles.xml.bz2

zhwiki网址：

https://dumps.wikimedia.org/zhwiki/latest/zhwiki-latest-pages-articles.xml.bz2

这里采用的一部分小的数据集：

https://dumps.wikimedia.org/enwiki/latest/enwiki-latest-pages-articles1.xml-p1p41242.bz2

Bert实验

2、使用WikiExtractor提取和整理数据集中的文本，使用步骤如下：

```
pip install wikiextractor
python -m wikiextractor.WikiExtractor ****.bz2 -o -b
```

参数解释：

-o：输出的文件夹，默认为text

-b：输出生成的每个文件的大小，例如20M

最后会在输出的文件夹下生成一系列的输出文件

Bert实验

3、离线生成tfrecord文件

create_pretraining_data.py：

输入参数：输入文件、输出的tfrecord文件名，字典vocab.txt

思路是获取所有的输入文件，将输入文件的所有文档经过分词之后全部存入到all_documents列表中，然后通过all_documents列表生成mlm和nsp的实例，最后将所有的实例存入列表中，保存到tfrecord文件中。**存储到tfrecord中参数：**

input_ids: 经过mlm预训练任务处理之后的tokens对应于字典的id列表

input_mask：表示哪些数据是有用的哪些数据是没有用的，1为有用，0为没用

segment_ids：段id，表示token属于第几句话

masked_lm_positions: mask的位置index

masked_lm_ids: 将mask对应的标签数据转为label id

masked_lm_weights: 哪些mask是有用的，1为有用，0为没用

next_sentence_labels：nsp任务的标签

训练Baseline：

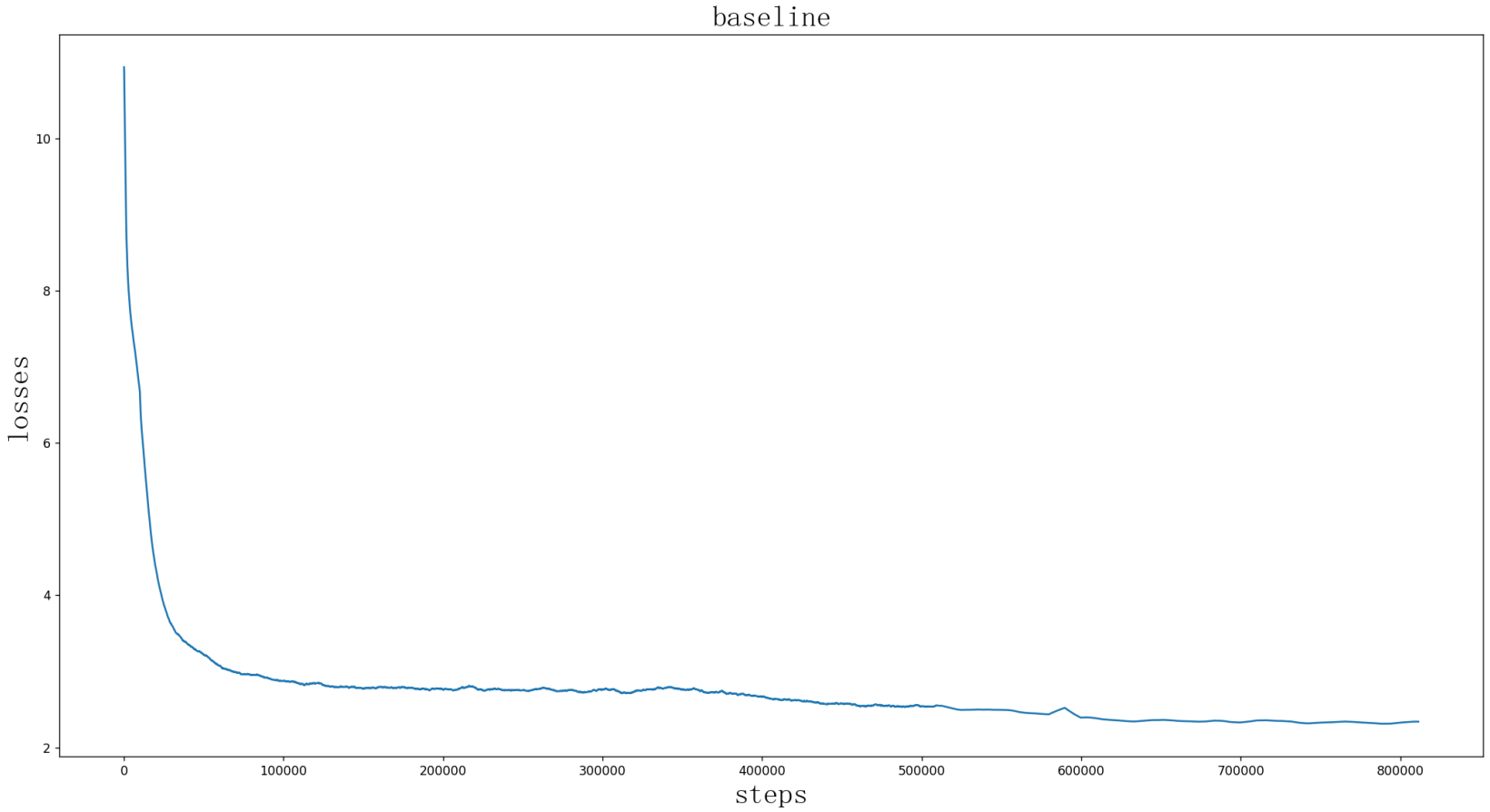
数据集：一共生成了5888559个instance存储在tfrecord中；

模型：BERT Base模型，Hidden layers为12，Hidden size为768，Attention Head为12

超参数设置：Batchsize为64，优化器为Adam，训练epoch为10，每个epoch下的step数量为92008

整体训练下来的step数量为921000

Baseline训练loss可视化：



第一步：

模型：BERT Base模型，**Hidden layers为3**

超参数设置：**训练epoch为1**

将第一步训练的模型参数进行复制生成六层模型对应的权重

第二步：

模型：BERT Base模型，**Hidden layers为6，**

超参数设置：**训练epoch为2**

加载复制的六层模型的权重参数

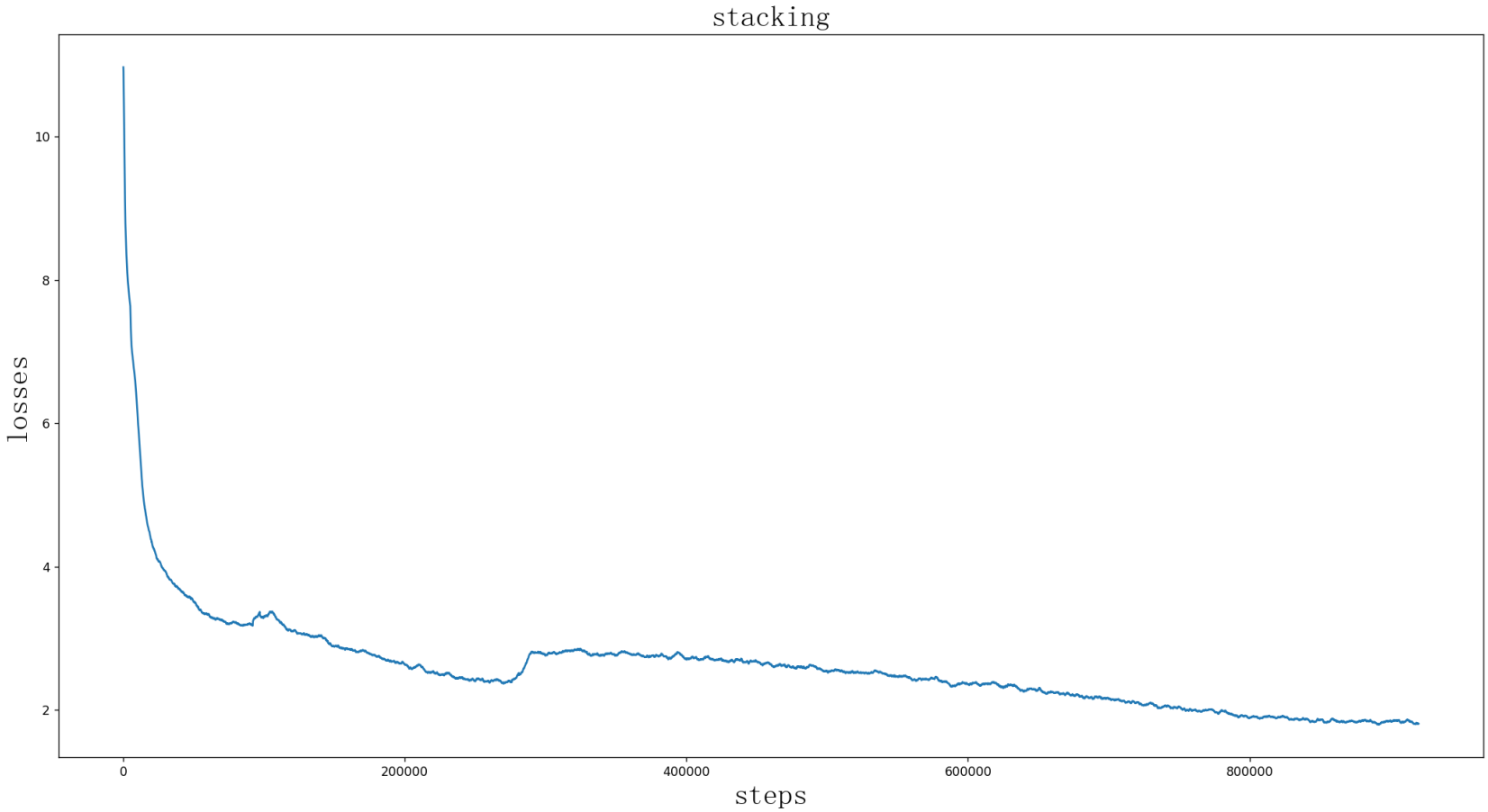
将第二步训练的模型参数进行复制生成十二层模型对应的权重

第三步：

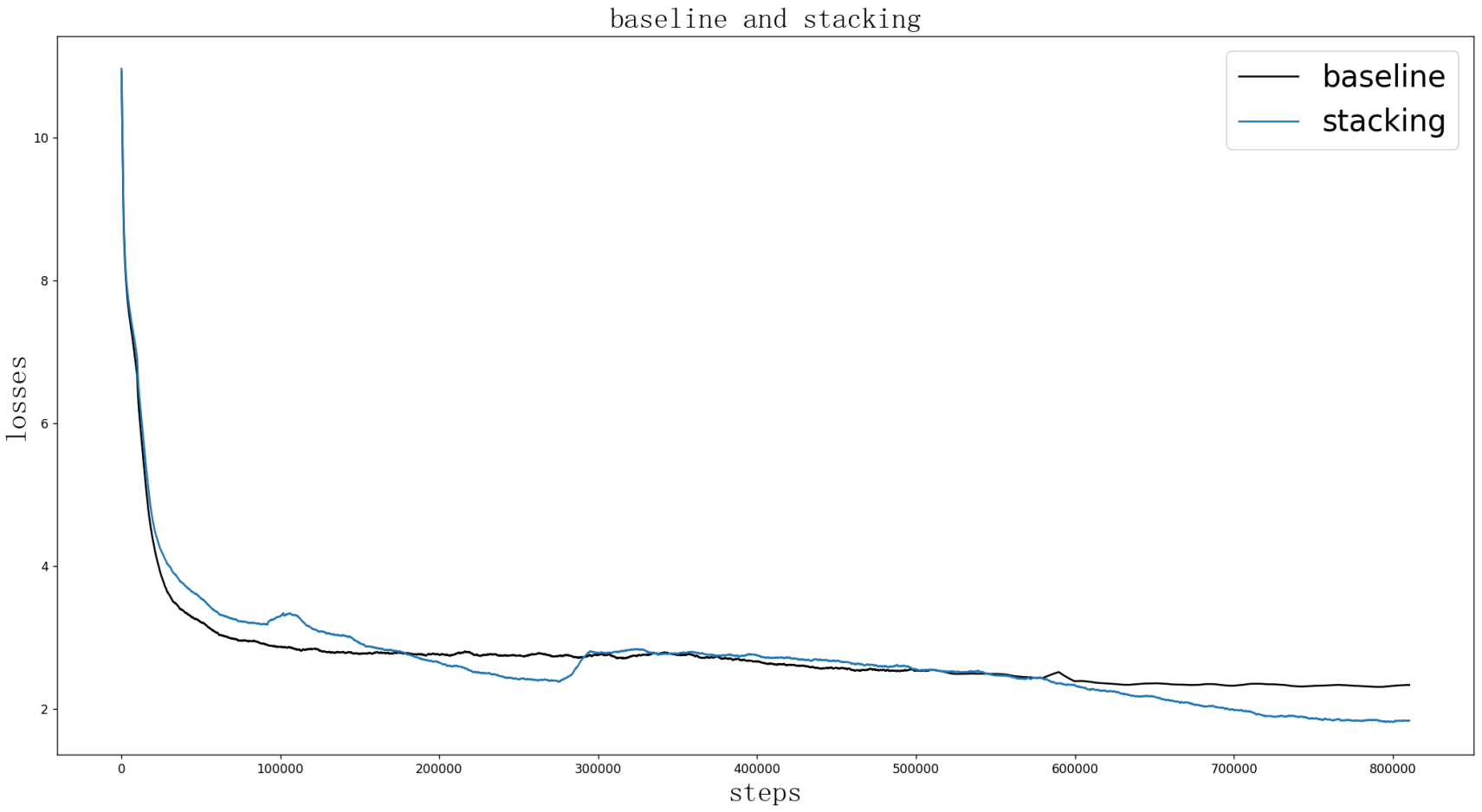
模型：BERT Base模型，**Hidden layers为12**

超参数设置：**训练epoch为7**

Stacking训练loss可视化：



Loss对比图：


baseline and stacking
baseline
stacking
losses
steps
10
8
6
4
2
0
100000
200000
300000
400000
500000
600000
700000
800000

时间：设备TeslaV10032GB 节约了15.23%训练时间

模型层数	训练epoch数量	每个step运行时间(ms)	总时间(min)
baseline12	10	175	2683.87
3层	1	61	96.432
6层	2	100	309.269
12层	7	175	1869.327

Bert实验

Squad下游任务：

微调加载的预训练模型	微调是否冻结主干	微调epoch数量	exact_match	f1
无	否	10	7.833	15.767
baseline	否	3	9.00	17.423
baseline	否	10	11.466	20.225
baseline	是	10	10.974	19.712
stacking	否	10	52.999	65.2288
stacking	是	10	53.141	64.9172

GPT实验数据：

使用OpenWebText数据集，预处理并转换为Mindrecord格式的数据。

Openwebtext.mindrecord：30.5GB

Openwebtext.mindspore.db：739MB

GPT模型：

Transformer Decoder模型

Num_layers：12层

Num_heads：12

Hidden_size：768

Seq_length：1024

Baseline：

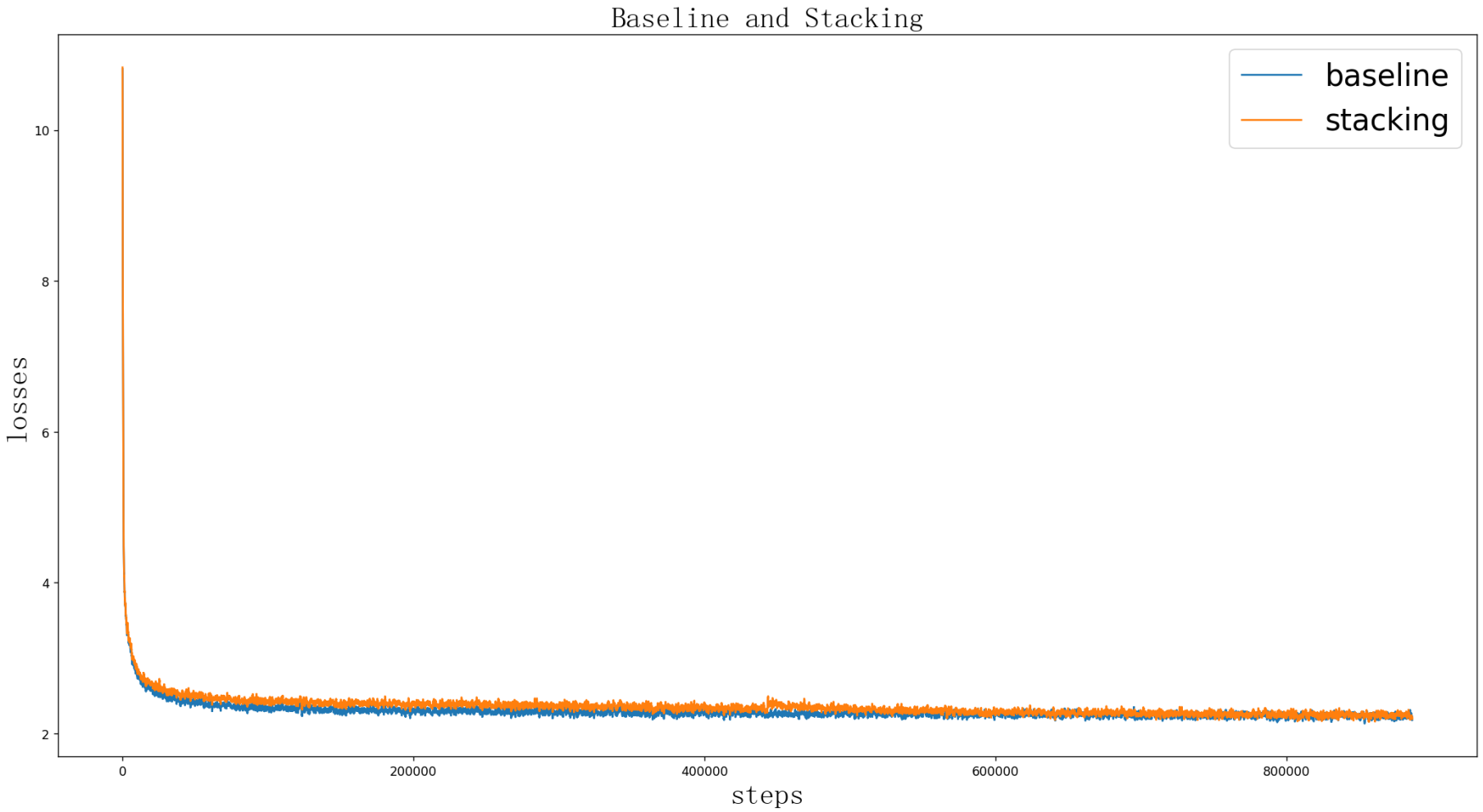
完整的数据集下12层GPT模型训练2个epoch，batchsize为32，每个epoch下的step数量为443330

Progressively Stacking：

第一阶段：完整的数据集下6层GPT模型训练1个epoch，batchsize为32，每个epoch下的step数量为443330;

第二阶段：12层GPT模型训练1个epoch，训练前加载第一阶段训练好的6层GPT模型权重并进行stacking。

GPT实验


Baseline and Stacking
baseline
stacking
losses
steps
10
8
6
4
2
0
200000
400000
600000
800000

时间：设备TeslaV10032GB 节约了19.08%训练时间

模型层数	训练epoch数量	每个step运行时间(s)	总时间(h)
baseline12	2	1.52	374.3676
6层	1	0.94	115.758
12层	1	1.52	187.184

wikitext-2下游任务：

微调加载的预训练模型	微调epoch数量	Loss	PPL
Hugging face	1	2.97833	19.655
baseline	1	3.239338	25.5168
stacking	1	3.232555	25.344

接口设计：

Progressively Stacking Train为多阶段训练，每一个阶段训练过程中构建的模型（模型的层数）、优化器的状态、epoch数量的设定、学习率调度器等都不一样，因此在每个阶段的训练过程中都要对这些进行单独的构建。

接口设计思路：

1、拷贝config文件：

各个阶段在构建相应的模型、优化器等时，可能会pop一些配置文件里面的参数，为了防止其影响下一阶段模型、优化器等的构建，需要提前将配置文件进行深度拷贝

2、通过循环来进行Progressively Stacking Train：

循环遍历来进行Progressively Stacking训练的各个阶段，每个阶段单独构建相应的模型、优化器、epoch size、学习率调度器等

3、阶段过渡时权重的加载：

在各个阶段之间过渡时，后一阶段的模型需要加载前一阶段的模型权重并进行stcking，例如第二阶段6层的模型需要加载第一阶段3层模型的权重并将此权重进行stacking，前一阶段的模型权重的路径可以通过前一阶段的ModelCheckpoint回调函数的latest_ckpt_file_name属性来获得。

4、不同的模型设置各自的stacking函数：

对于GPT和Bert而言，其模型权重字典的key值都不一样，因此需要针对不同的大模型来构建不同的stacking函数

接口代码：

```
def deepcopy_opt(self, opt):
    temp_logger = opt.logger
    del opt.logger
    temp_opt = copy.deepcopy(opt)
    opt.logger = temp_logger
    temp_opt.logger = temp_logger
    return temp_opt
```

接口代码：

```
for i in range(len(self.config.stages)):
    # 参数的复制
    self.config = temp_opt
    # 修改一些参数(每个阶段的epoch数量与网络层数)
    self.config.epoch_size = self.config.stage_epochs[i]
    # Build the model with loss
    self.logger.info("Start to build model")
    model_config = self.check_and_build_model_config()
    model_config.is_training = True
    # 修改网络层数
    model_config.num_layers = self.config.stages[i]
    net_with_loss = self.build_model(model_config)
    self.logger.info("Build model finished")
```

接口代码：

```
# 是否需要叠加权重
if pre_load_model_path and i >= 1:
    self.logger.info(f"Start to load the ckpt from {pre_load_model_path}")
    # 权重参数翻倍
    ckpt = self.double_model_weights(pre_load_model_path, self.config.stages[i - 1])
    # 权重加载
    load_param_into_net(net_with_loss, ckpt)


# 保存最新权重路径
pre_load_model_path = callback[-1].latest_ckpt_file_name
```

接口代码：

```
def double_model_weights(self, pre_load_model_path, pre_num_layers):
    # GPT模型权重的翻倍
    param_dict = load_checkpoint(pre_load_model_path)
    lst = []
    for k, v in param_dict.items():
        k_split = k.split('.')
        # print(k_split)
        if k_split[0] == 'backbone' and k_split[1] == "transformer" and k_split[2] == 'encoder' \
                and k_split[3] == 'blocks':
            l_id = int(k_split[4])
            k_split[4] = str(l_id + pre_num_layers)
            new_k = ".".join(k_split)
            lst.append([new_k, v.clone()])
    # 将叠加的层权重添加到param_dict中
    for k, v in lst:
        param_dict[k] = v
    return param_dict
```

接口代码：

```
export GLOG_v=3
export DEVICE_ID=$1
DATA_DIR=$2

python -m mindtransformer.models.gpt.gpt_stack_trainer \
    --stages=[3,6,12]  \
    --stage_epochs=[1,1,3]  \
    --train_data_path=$DATA_DIR \
    --optimizer="adam"  \
    --seq_length=1024 \
    --parallel_mode="stand_alone" \
    --checkpoint_prefix="gpt" \
    --global_batch_size=16 \
    --vocab_size=50257 \
    --hidden_size=768 \
    --num_heads=12 \
    --full_batch=False \
    --device_target="GPU" > standalone_train_gpu_log.txt 2>&1 &
```

西安电子科技大学
XIDIAN UNIVERSITY

谢谢！