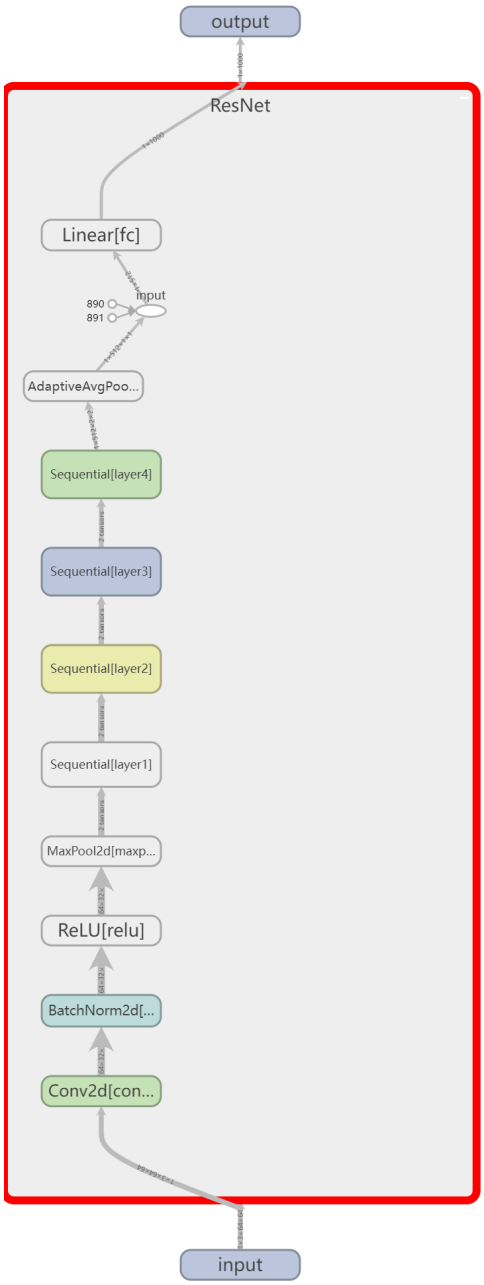
Python与深度学习基础第二次作业实验报告

贺维易PB20051035

一、在Tiny-ImageNet数据集上训练Resnet模型

- **实验任务**

1. 通过TensorBoard绘制Graph功能列出resnet18模型各层名称及输出大小，这部分代码一并在**Resnet.py**中。最终将训练好的模型参数**model_weights.pth**上传至Github.

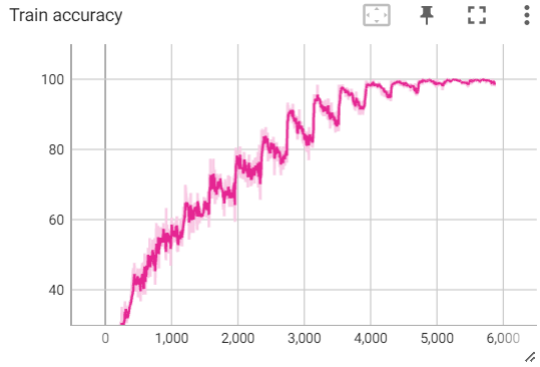


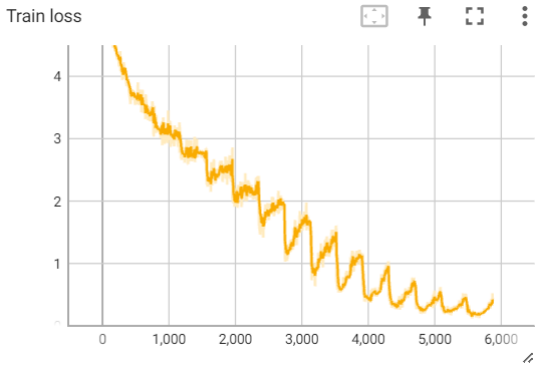
2. 在Github上提交了git diff生成的改动说明**patch.diff**文件。需要说明的是，对于验证集的处理，我选择了在本地修改val目录结构的方法，使之与train文件夹目录结构相同，从而有正确的标签，这部分代码将作为**modify.py**同时提交到Github，其中的路径名称是由本地目录决定的。

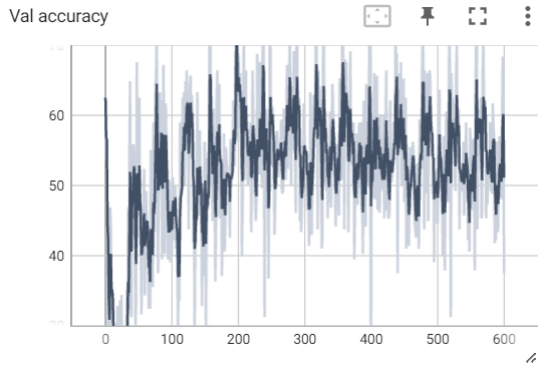
3. 设置**epoch=15**，将resnet18在训练集上的**Top5**精度训练到**99%**，展示TensorBoard中观察到的训练集Loss、训练集精度、验证集Loss、验证集精度变化。此外，我们使用训练时间作为指标量化评价训练速度的差异。

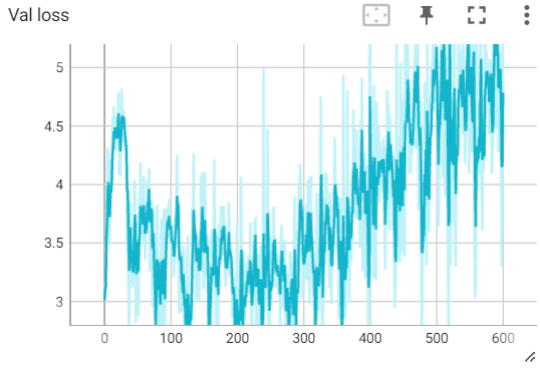
- **CPU**

  在个人电脑上训练，训练时间4h.



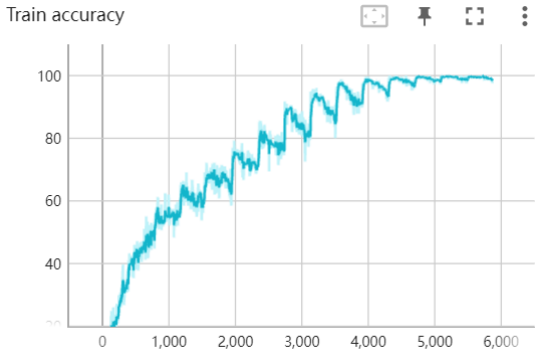


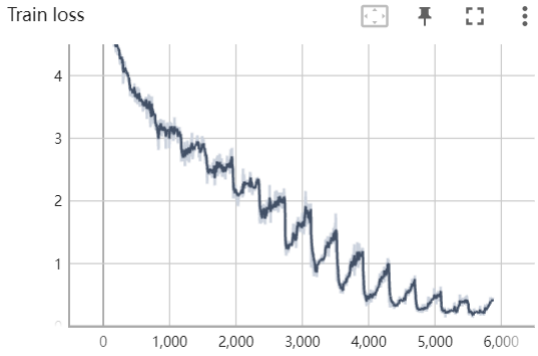


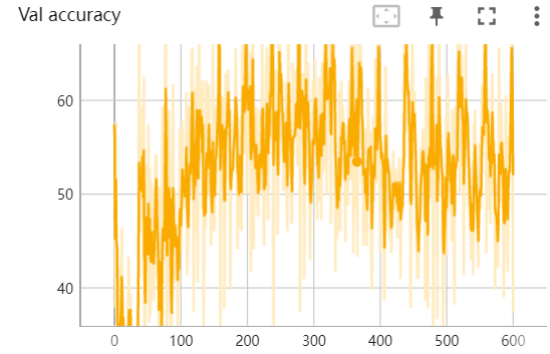


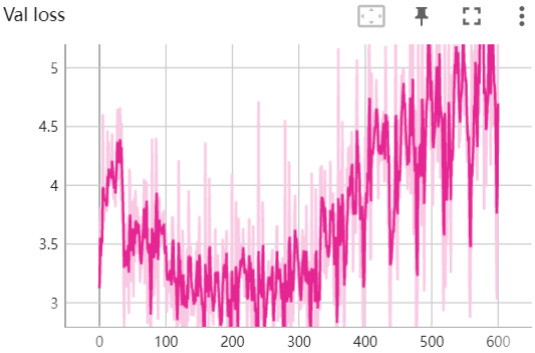
- **1个GPU**

  在Gitahub上训练，GPU编号#1，GPU类型gtx1080ti，训练时间1h 21m 19s.



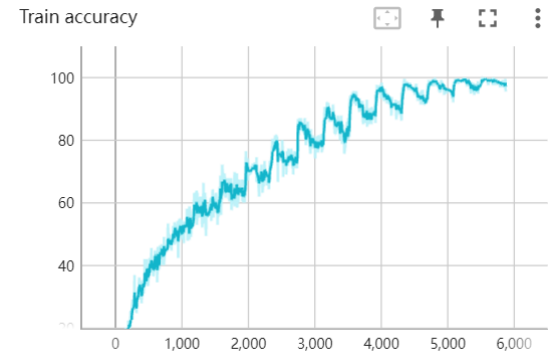


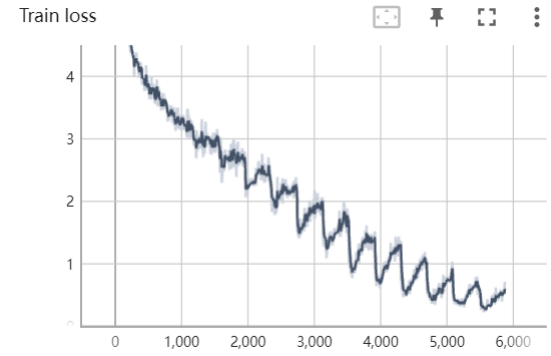


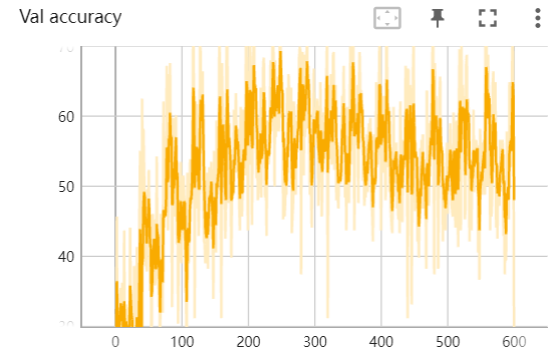


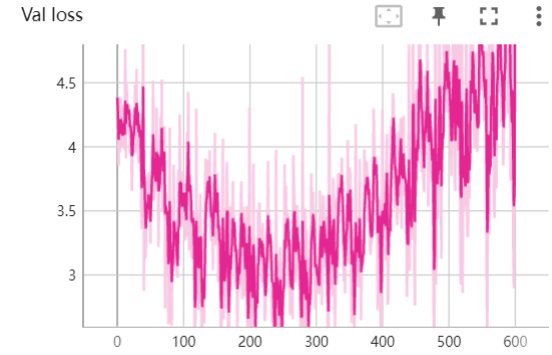
- **8个GPU**

  在Github上训练，GPU编号#0,#1,#2,#3,#4,#5,#6,#7，GPU类型gtx1080ti，训练时间1h 7m 29s.









分析：

- 在不同的GPU环境下曲线变化趋势基本相同，因此此处以8个GPU训练的loss和精度曲线为例进行分析。通过以上的图片可以看出Train accuracy前期是上升的，在中后期会出现一些波动，但整体上仍是逐渐上升的，且最终的训练精度可以达到99%左右，此后再增大epoch训练效果上不会再有显著提高，故我们选择设置参数epoch=15。Train loss前期是下降的，在中后期也出现了一些波动，但整体上仍是逐渐下降的。而Validation上的accuracy始终有较大波动，且最终准确率只能在60%左右；Val loss也一直存在波动，在整个过程中经历了一个先下降后上升的过程。
- 可以看出在GPU环境下训练速度将会显著提高，且GPU个数越多训练速度越快，当然所需的算力也越大。

4. 使用代码中的--evaluate选项对比两次评估的差异

```
# 使用--resume加载checkpoint保存的模型
python Resnet.py --resume checkpoint.pth.tar --evaluate
```

```
evaluation_results = []
...
output = model(images)
evaluation_results.append(output)
...
return evaluation_results # 在run_validate函数中保存输出结果
```

evaluation_results列表中的每个元素对应于每个批次的输出结果，即模型对于每个批次输入图像的预测输出.

```
if args.evaluate:
    results = validate(val_loader, model, criterion, args)
    import pickle
    # Save the evaluation results to a file
    with open('evaluation_results.pkl', 'wb') as f:
        pickle.dump(results, f)
    return
```

修改这部分，将输出结果保存至pkl文件中。输出结果有40个批次，每个批次256个数据，每个数据维度是200，其中最大值即为预测分类的类别。

按照上述步骤，对两个训练过程中的checkpoint进行加载并保存验证的输出结果。

```
# 打开并加载 pkl 文件
with open('evaluation_results1.pkl', 'rb') as f:
    results1 = pickle.load(f)
with open('evaluation_results2.pkl', 'rb') as f:
    results2 = pickle.load(f)
# 以第一个批次为例，事实上，遍历1:40批次即可得到所有图片的预测分类
predictions1 = torch.argmax(results1[1], dim=1)
predictions2 = torch.argmax(results2[1], dim=1)
```

即可对比两次评估的差异并找出评判结果不同的图片.

- **实验总结**

1. 本次实验对Github上pytorch提供的example做了修改，实现了在Tiny-ImageNet数据集上训练Resnet模型。通过实验，完整地完成了第一次深度学习的训练，熟悉了深度学习的基本框架，学习了包括Tensorboard在内的可视化工具，并首次使用Bitahub进行模型训练。
2. 实验中遇到的困难及解决方案
    - 路径问题：在Bitahub上由于tiny-imagenet-200数据集太大不好上传，使用的是平台公开的数据集，但这和个人电脑上的路径是有差异的，因此开始总是报错。最后通过帮助文档学习了Bitahub的目录结构，并借鉴其它公开项目的路径设计成功运行。
    - Bitahub启动命令：缺乏一点计算机基本常识，刚开始不知道启动命令有什么用，每次都是随便写的，然后就一直运行失败，日志上也没有明确显示为什么失败，排除了各种可能猜是可能启动命令出了问题，最后仿照其它项目的启动命令按照网络结构-模型-训练集路径-测试集路径规范输入启动命令解决了问题。

- Validation曲线绘制：最开始Train的曲线没有问题，但Val同时画出来了很多条乱七八糟的曲线，猜想应该是代码中写入tensorboard的位置和横坐标出现了错误，为了方便画图，选择在validate函数中像train一样多加了epoch参数，然后在调用main()之前定义writer=SummaryWriter().
- 原始代码中保存的最优模型model_best.pth.tar太大无法上传Github，我采用如下方法仅保存模型参数（而非完整模型），并使用Github Desktop从本地上传.

```
import torch
import torch.nn as nn
import torchvision.models as models
# 定义模型的结构
model = models.resnet18()  # Resnet18 model
# 加载模型参数
checkpoint = torch.load('model_best.pth.tar')
model.load_state_dict(checkpoint['state_dict'])
# 保存模型参数
torch.save(model.state_dict(), 'model_weights.pth')
```

二、复现Word-level Language Model并讨论

- **实验任务**

1. Transformer模型训练和文本生成
   - 实验结果截图

```
| epoch   1 |   200/ 2983 batches | lr 20.00 | ms/batch 452.42 | loss 10.36 | ppl 31684.03
| epoch   1 |   400/ 2983 batches | lr 20.00 | ms/batch 469.61 | loss  9.00 | ppl  8087.28
| epoch   1 |   600/ 2983 batches | lr 20.00 | ms/batch 446.85 | loss  8.88 | ppl  7192.15
| epoch   1 |   800/ 2983 batches | lr 20.00 | ms/batch 449.66 | loss  8.70 | ppl  6005.77
| epoch   1 |  1000/ 2983 batches | lr 20.00 | ms/batch 447.38 | loss  8.75 | ppl  6285.95
| epoch   1 |  1200/ 2983 batches | lr 20.00 | ms/batch 445.81 | loss  8.60 | ppl  5434.32
| epoch   1 |  1400/ 2983 batches | lr 20.00 | ms/batch 444.25 | loss  8.44 | ppl  4643.33
| epoch   1 |  1600/ 2983 batches | lr 20.00 | ms/batch 449.84 | loss  8.70 | ppl  6003.81
| epoch   1 |  1800/ 2983 batches | lr 20.00 | ms/batch 457.42 | loss  8.81 | ppl  6693.99
| epoch   1 |  2000/ 2983 batches | lr 20.00 | ms/batch 465.95 | loss  8.66 | ppl  5762.23
| epoch   1 |  2200/ 2983 batches | lr 20.00 | ms/batch 462.81 | loss  8.76 | ppl  6368.89
| epoch   1 |  2400/ 2983 batches | lr 20.00 | ms/batch 455.82 | loss  8.32 | ppl  4092.73
| epoch   1 |  2600/ 2983 batches | lr 20.00 | ms/batch 449.09 | loss  8.33 | ppl  4167.07
| epoch   1 |  2800/ 2983 batches | lr 20.00 | ms/batch 450.81 | loss  8.33 | ppl  4144.04
-----------------------------------------------------------------------------------------
| end of epoch   1 | time: 1413.27s | valid loss  7.37 | valid ppl  1583.01
-----------------------------------------------------------------------------------------
```

   仅截取运行结果的一部分证明对代码的复现，训练好的模型作为**model.pth**上传至Github.

2. 利用TensorBoard绘制Transformer结构

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.tensorboard import SummaryWriter

class Transformer(nn.Module):
    def __init__(self, input_size, hidden_size, num_heads, num_layers):
        super(Transformer, self).__init__()
        self.embedding = nn.Linear(input_size, hidden_size)
        self.encoder_layers = nn.ModuleList([
            nn.TransformerEncoderLayer(d_model=hidden_size, nhead=num_heads)
            for _ in range(num_layers)
        ])
```

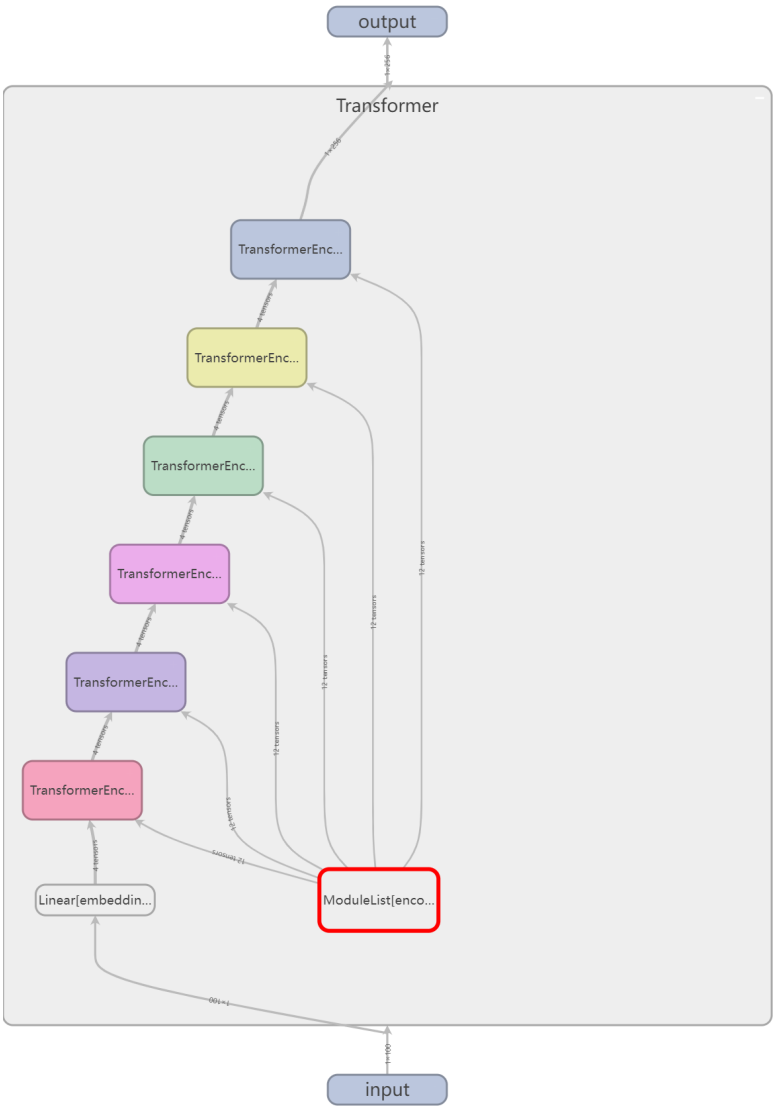
```
    def forward(self, x):
        x = self.embedding(x)
        for layer in self.encoder_layers:
            x = layer(x)
        return x

# Create a dummy input tensor
input_size = 100
input_tensor = torch.randn(1, input_size)

# Create a Transformer model
model = Transformer(input_size=input_size, hidden_size=256, num_heads=4,
num_layers=6)

writer = SummaryWriter()
writer.add_graph(model, input_tensor)
writer.close()
```

这只是显示Transformer结构的一个例子.



3. 讨论Transformer和CNN在捕捉上下文依赖上的差异

CNN主要通过卷积层和池化层来捕捉局部特征，它在空间维度上共享权重，能够有效地捕捉图像中的局部模式和空间关系，但由于卷积和池化操作的局部性质，CNN在处理序列数据等具有长距离依赖性的任务时存在一定的限制。

相比之下，Transformer是一种基于自注意力机制的序列模型，它能够捕捉全局依赖关系。Transformer通过自注意力机制，即通过对序列中不同位置的元素进行加权聚合，来建立元素之间的长距离依赖关系。这使得Transformer在处理序列数据时能够更好地捕捉全局上下文依赖关系，适用于机器翻译、语言模型等任务。