#1 云计算实验报告

邹弘嘉、曹汇杰、吴双

#2 项目目标

利用云主机完成模型的训练以及模型的预测与用户完成交互，我们组选定的项目是下一位置预测，我们将根据用户自己给出的之前的一系列的路径预测其下一可能出现的位置

#2 AI train的使用

我们首先使用Ucloud上的交互式训练，使用其中的pytorch环境，在安装必要的库（如tqdm）之后，对模型开始训练并成功跑通代码，但代码中没有对模型进行保存

#2 部署上云

这一部分主要是对环境的配置，因为原先安装的python 2，所以我们要重新安装python 3，并用如下命令安装pytorch：

```
conda install pytorch torchvision cudatoolkit=9.2 -c pytorch
```

#2 训练与模型

由于官方提供的代码并不具有模型的交互、保存与测试使用的功能，所以我们进行了相应的修改。其中保存和使用这两部分，由于涉及到对整个模型的理解，所以进行了相当大的修改。

#3 模型的交互

其中，由于ucloud的ai-inference并不支持pytorch的基础环境，所以不支持我们的基础模型的部署，更由于相应的一些涉及memory、gpu的问题导致ucloud的训练不太便于使用，于是我们斟酌后决定使用ucloud云主机进行交互式训练以进行代码调试，调试成功后再使用极客云云主机进行训练。

由于ucloud上并没有gpu和cuda环境的比较方便的支持，我们将代码进行了如下的变化，以使得程序在cpu环境中也可以运行，方便我们的代码调试。

```
    strnn_model = STRNNCell(dim).cuda()
    checkpoint = torch.load('./checkpoint.pth.tar', map_location='cpu')
```

之后我们由于模型训练的连续性要求将交互式训练转变为普通训练，这一部分的具体原因我们会在下一部分说明。

#3 模型的保存

在交互式的代码调试过程中，我们发现官方给出的代码不具有模型的保存功能，于是我们使用checkpoint来进行模型的保存。

```
    # Save
    torch.save(strnn_model, os.path.join("./", "epoch_" + str(start_epoch + i+1) + ".pt"))
    # Get bool not ByteTensor
    is_best = bool(total_loss > best_loss)
    # Get greater Tensor to keep track best acc
    best_loss = min(total_loss, best_loss)
    # Save checkpoint if is a new best
    save_checkpoint({
        'epoch': start_epoch + i + 1,
        'state_dict': model.state_dict(),
        'best_loss': best_loss
    }, is_best)
```

通过保存所在的`epoch`和`best_loss`来保证程序每一次保存的都是loss最低的模型。

这一部分涉及两个困难点：1. 模型训练连续性难以保证；2. 无法在每一个epoch都保证继承的是之前训练出来的最好的模型。

1. 市面上大部分的云主机为了保护自己token分配的安全性以及迎合大部分用户的使用习惯，都默认缺省值情况下的交互式训练是不需要后台运行的，所以需要通过命令行执行一些指令来将任务转为后台运行。我们的训练任务时间复杂度非常高，平均一个epoch就要训练三个小时，于是我们不可能将终端一直打开或者实时监控训练进度，于是我们对代码进行了一些连续性上的优化，即后台运行机制。
   这个机制中，我们将交互式的训练转化为普通训练，将所有print输出都设置实时的重定向

   ```
   print(out, flush=True)
   ```

   然后将所有的进度显示和运行状态写入云端的一个临时文件中，即输出`log`到`/tmp/log3`文件：

   ```
   setsid python lucky_dog.py > /tmp/log3 2>&1 &
   ```

   并在下次进入终端时可以用如下指令看训练任务打出来的`log`：

   ```
   tail -f /tmp/log3
   ```

   这样我们便可以将云主机的训练与客户端的操作分离开来，保证训练的连续性不会被客户端打扰。
2. 在模型的每一个epoch的迭代帧中，我们发现模型只是直接的继承上一个epoch的训练结果，并没有判断这一个结果是否是有效的即相对于之前的迭代是增长的，所以很有可能会出现过拟合的情况，于是我们设计了一个新的方法，即除了第一次每一次epoch都选取最好的checkpoint作为上一个的epoch的训练结果，在此基础上进行接下来的训练。但由于考虑到可能会落入局部最优的不理想情况，于是我们预先训练了十个epoch，将这十个epoch的模型分别保存，比较最好的那一个设置为checkpoint，然后再在十一个epoch以后的每一轮训练中进行模型与checkpoint的取优：

   ```
       # Load Checkpoint
       if(os.path.exists('./checkpoint.pth.tar')):
           checkpoint = torch.load('./checkpoint.pth.tar')
           start_epoch = checkpoint['epoch']
           best_accuracy = checkpoint['best_loss']
           strnn_model.load_state_dict(checkpoint['state_dict'])
           print("=> loaded checkpoint '{}' (trained for {} epochs)".format('./checkpoint.pth.tar', checkpoint['epoch']), flush=True)
   ```

最终我们的模型达到了

```
Training End..
Test:
recall@1: 0.0016934547654726034
recall@5: 0.003484628548254774
recall@10: 0.003897527834104618
recall@100: 0.00457426672545655
recall@1000: 0.01357672547208301
recall@10000: 0.07937842017610481
```

如下的准确率，部分recall比官方的Performance要好，但是没有显著的提升。

#3 模型的使用

容易看出，在preprocess过程中，模型实际上将gowalla数据集转化成按用户分类的时序轨迹数据集，于是数据集和测试集的数据格式都需要统一成这一格式。后端传输给模型的数据集具有如下的格式，并保存在云主机本地的一个文件夹`"demo.txt"`里

时间间隔	地点间隔	上一个地点	下一个地点

由于时间间隔和上一个地点都可以通过实际过程中直接得到，于是我们直接将这两个项填入实际的值便可。由于地点间隔和下一个地点都属于实际中没办法得到的项，于是我们使用随机数来填入这两个项，以防用户在第一次行动时就想要进行预测，我们将第一行的时间间隔和地点间隔设置为0。

在模型的使用过程中，我们发现模型的返回值形式为地点序号，于是我们将返回值通过一个地点序号和地点经纬度的字典进行相应的映射。

```
id2lalo = pd.read_csv("id2latilongi.csv")

    for output in outputs:
        temp_out.append(','.join(str(i) for i in id2lalo.iloc[output]))
#reflect the locations' ids to their latitudes and longitudes
    return '\t'.join(str(i) for i in temp_out)
```

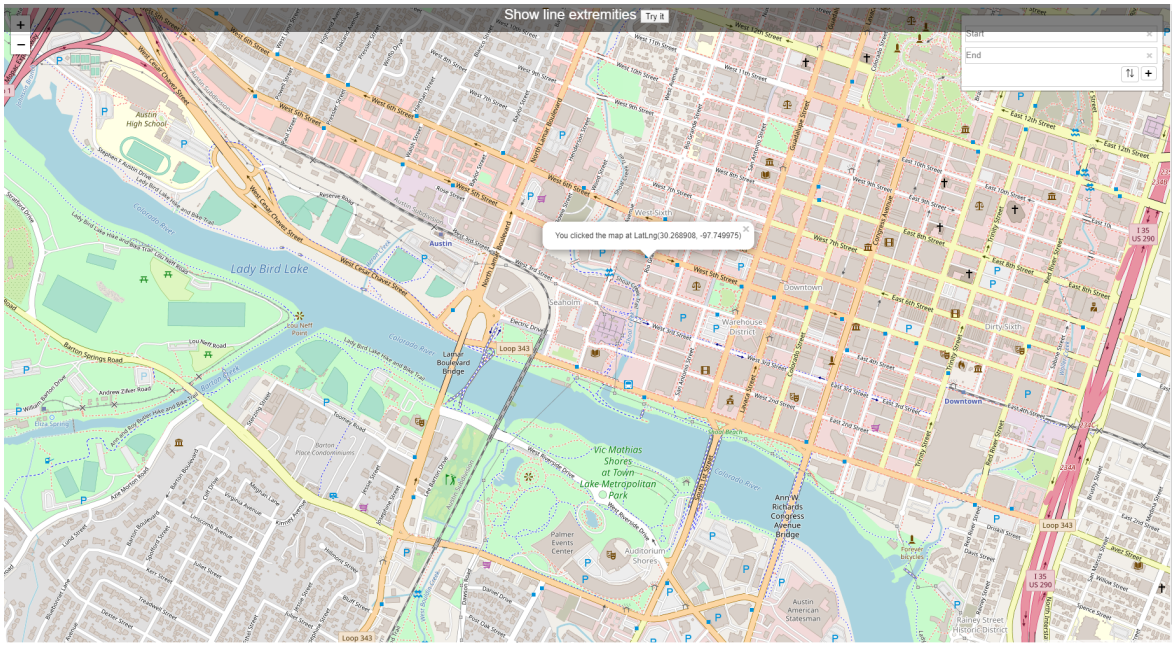
由于模型的精度问题以及一些训练算法的缺陷，使得对于任意的测试数据，模型预测的可能性最大的结果都固定在一个点上不会发生变化，我们编写了100个随机数据集进行比对，验证了这一点，在这里就不多做赘述。于是我们将训练的返回，改写为返回可能性降序的前100个地点的向量，将其映射为逗号区分的经纬度，以'\t'间隔：

```
def evaluation(train_file="demo.txt"):
    # print("Loading data â€¦")
    train_user, train_td, train_ld, train_loc, train_dst =
data_loader.treat_prepro(
        train_file, step=1)
    # strnn_model = STRNNCell(dim).cuda()

#     checkpoint = torch.load('./checkpoint.pth.tar', map_location='cpu')

    train_batches = list(
        zip(train_user, train_td, train_ld, train_loc, train_dst))
#     print(train_batches)
    outputs = print_score(train_batches, 3)
    temp_out = []
    for output in outputs:
        temp_out.append(','.join(str(i) for i in id2lalo.iloc[output]))
    return '\t'.join(str(i) for i in temp_out)
```

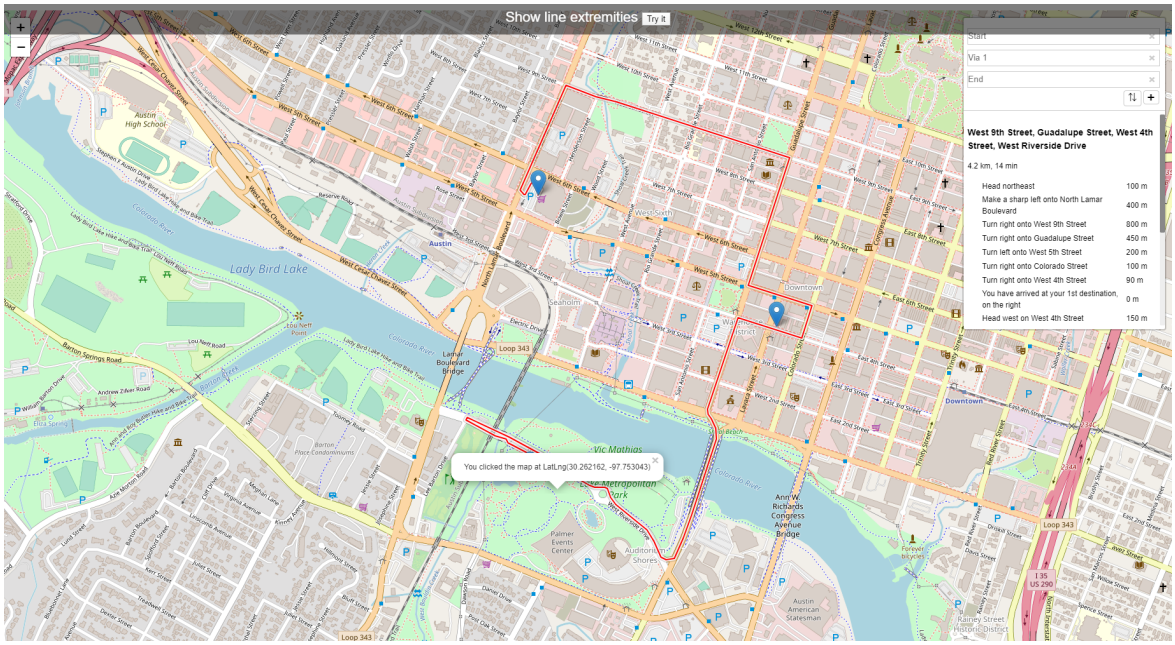
#2 前端

我们使用`jquery`框架进行了整体前端的部署，`ajax`协议进行前后端的消息传输，并使用`leaflet`第三方框架、`mapbox`的地图资源及其插件`leaflet-routing-machine-master`进行前端功能的实现。

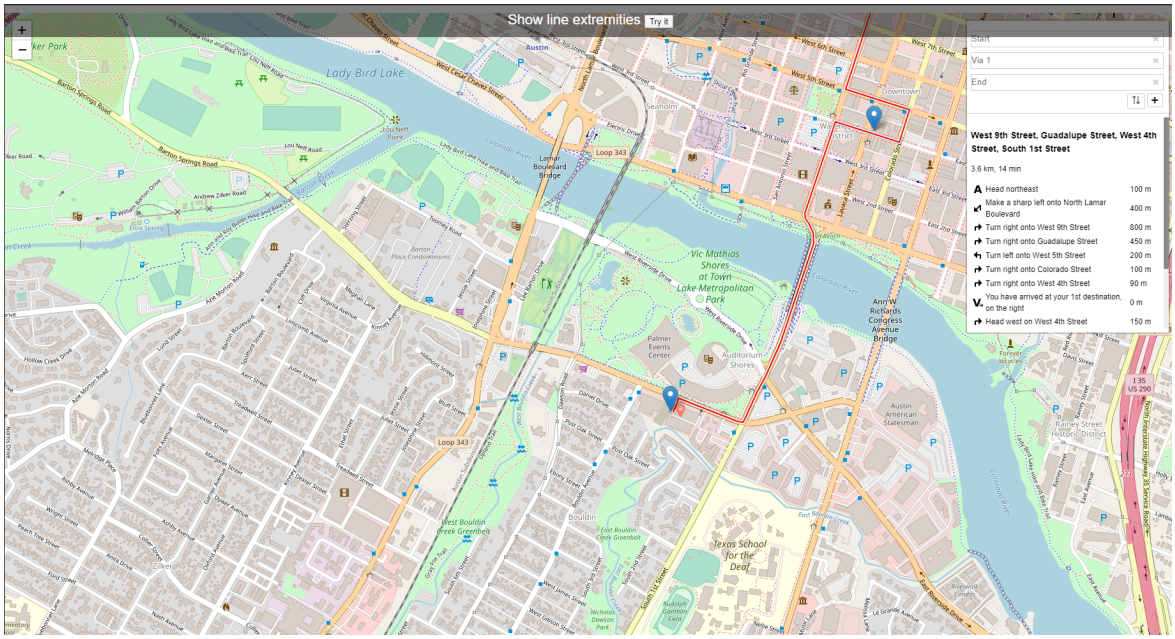
交互的方式为点击地图，前端将所点击的地点坐标传输给后端，后端进行相应的判断，选取周围的有效点返回给前端，再由前端将其显示在地图上。



在之后的每一次点击，都会按顺序显示地图规划好的路线，路线红色显示，备选路线蓝色显示，相关的具体信息在右上角显示。



在地图上拥有立足点后的任何时间点都可以进行地点预测，在地图的顶部点击`try it`按钮，就可以将现有的所有信息提交给后端，然后后端将这些信息进行整合，得到一个预测结果，返回给前端。前端将预测的点以popup的形式显示在地图上。



#2 后端

后端代码使用了Restful 风格的beego 框架, 主要功能由以下几个函数实现.

1. 接收前端发送过来的地址,并存储下来,以便日后的训练

```
func (this *DataController) Post() {

    longitude := this.GetString("longi")
    latitude := this.GetString("lati")

    res1,_ := strconv.ParseFloat(longitude,32)
    res2,_ := strconv.ParseFloat(latitude,32)

    var point Point
    point.Longitude = res1
    point.Latitude = res2
    pointSlice = append(pointSlice, point)
    p,the_num := getNearstPoint(point)
    //根据point 第三个参数
    endPlace = strconv.Itoa(the_num)
    strToWrite := strconv.FormatFloat(p.Latitude,'E',-1,32)+"\t"+strconv.FormatFloat(p.Longitude,'E',-1,32)+"\t"+startPlace+"\t"+endPlace
    startPlace = endPlace
    tracefile(strToWrite)
    data := &Point{p.Latitude,p.Longitude}
    lastPoint = *data
    this.Data["json"] = data
    this.ServeJSON()
}
```

2. 接收前端发送过来的计算请求,根据之前存储的结果计算出最后的结果并且返回给前端,同时将之前存储的计算结果清零饥渴

```
/*获得最后的结果*/
func (this * ComputeController) Post() {
    /*运行python脚本获得字符串式输出*/
    cmd := exec.Command("python","/Users/hjzhou/Gocode/newServer/src/app/controllers/main.py")
    out,err := cmd.CombinedOutput()
    if err!=nil {
```

```
        log.Println(err.Error())
    }
    p := strings.Replace(string(out),"\n","",100)
    a := strings.Split(p,"\t")
    min := 1000.0
    num :=-1
    for i := 0; i< len(a);i=i+2  {
        Lati, _ := strconv.ParseFloat(a[i],32)
        Longi,_ := strconv.ParseFloat(a[i+1],32)
        newPoint := Point{Latitude:Lati,Longitude:Longi}
        dis := calDistance(lastPoint,newPoint)
        if min<dis {
            min = dis
            num = i
        }
    }
    Lati,_ :=strconv.ParseFloat(a[num],32)
    Longi,_ := strconv.ParseFloat(a[num+1],32)

    //以json方式传递给前段
    data := &Point{Latitude:Lati,Longitude:Longi}
    this.Data["json"] = data
    this.ServeJSON()
}
```