文件结构

- /src：代码目录

  main.py：主函数入口，包括训练和测试代码

  constants.py: 网络训练的各种选项常量定义，用于复现实验结果

  models.py, model_self.py, model_attention_self.py:模型定义和调用，其中models.py是引用pytorch的RNN,LSTM,GRU的库的网络，model_self.py是自己用pytorch底层函数实现的GRU网络和layer norm, model_attention_self.py是引入自注意力机制的模型

  data.py:数据读取

  plot_curves.py:测试训练结果可视化代码，使用tensorboard

- /data：数据集目录
- /result: 训练测试结果目录

实验结果

我对网络的各种可能的超参数/训练方法进行了严格控制变量的对比实验，选取出了以下的较优的网络参数，具体内容如下。

0.实验环境

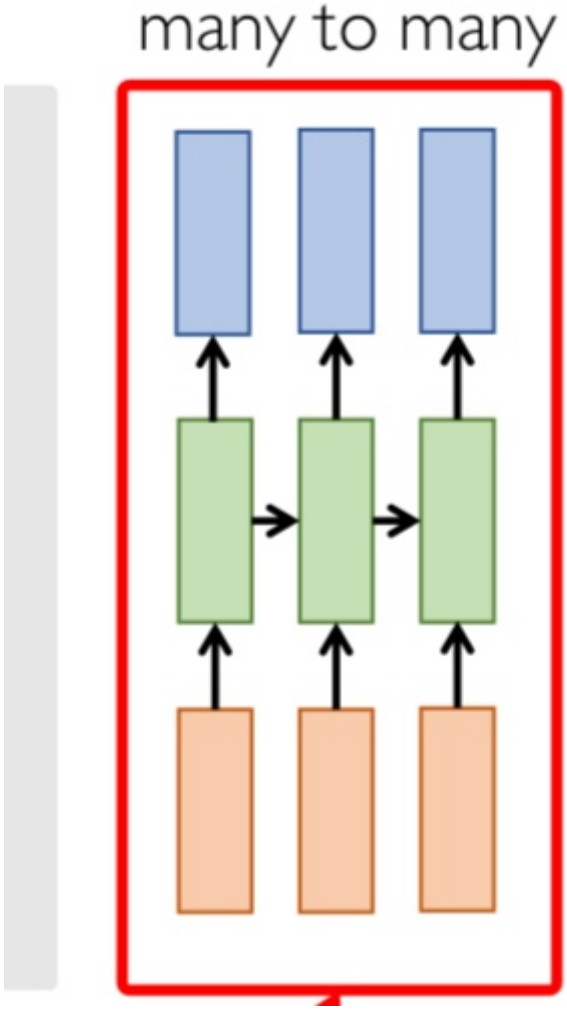
- ubuntu 18.04系统
- python 3.6.9
- pytorch 1.4.0

1.基本情况

1.1 网络整体结构

词向量先经过一个embedding构成的encoder，然后经过一个RNN网络，然后经过一个全连接层构成的decoder，得到结果。

因为这是一个Language Modelling任务，因此RNN应该选择自回归的many to many架构。假设一组输入的句子长度为n+1，那么，输入网络的应该分别为词1到词n,网络输出的结果应该分别和词2到词n+1进行比较，以求得语言模型的perplexity，进行反向传播。



1.2 超参数和优化方法的具体选择

- 句子最大长度：35
- batch大小：训练20测试10
- RNN网络类型：2层GRU网络，大小为50*50
- 学习率:10^-3，不衰减
- 优化方法：Adam，不带weight_decay
- 初始化：pytorch默认的均匀分布初始化
- dropout：对RNN层进行概率0.5的dropout
- 不进行gradient clip
- 不进行weight normalization
- 不进行layer normalization（因为pytorch没有在自己的RNN里封装layer normalization，我只能自己实现GRU网络和对应的layer normalization，但是我自己实现的网络没有进行并行优化，速度很慢）

1.3 可视化与实验复现方法

两者都需要进入src文件夹下

复现最优结果只需要输入

```
python main.py
```

其余结果请参考以下各个实验的复现命令行指令

可视化复现需要先在src文件夹下，删除run文件夹，然后输入

```
python plot_curves.py
```

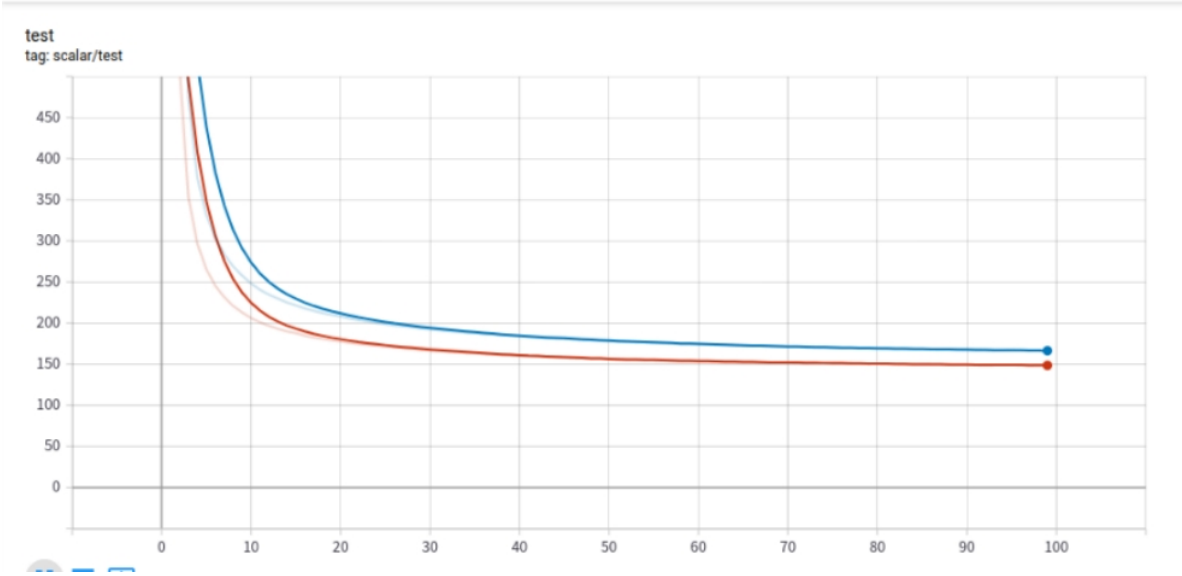
然后在命令行中输入想要可视化的实验对应的数字

然后在命令行输入

```
tensorboard --logdir runs
```

此时，进入命令行提示的链接即可看到结果

1.4 最优模型的结果



蓝色：训练的perplexity

红色：测试的perplexity

之后的每条曲线，都表示该实验的测试perplexity

2.RNN单元的选择和实验

2.1 选取原因

RNN一共有三种常用基本单元：RNN,LSTM,GRU。LSTM和GRU都解决了RNN的梯度爆炸与消失问题，准确率差不多，但是GRU更快。在实现网络的时候，我们也要在这三种基本单元中进行实验和选择。同时，我还用torch的基本单元和全连接网络实现了GRU单元，为了证明实现的正确性，我将pytorch自带的GRU单元和自己实现的GRU单元效果进行比较。

2.2 复现方法

基本RNN单元：

```
python main.py --network_type=2
```

LSTM单元：

```
python main.py --network_type=1
```

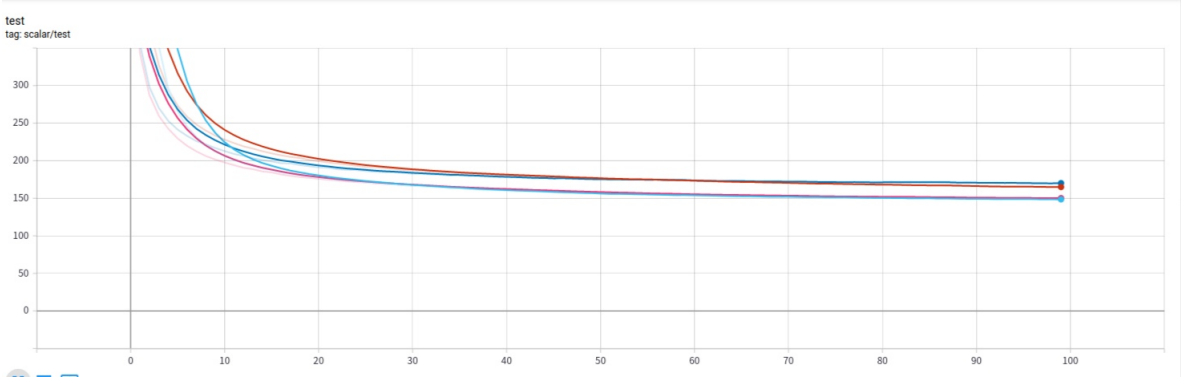
GRU单元：

```
python main.py --network_type=0
```

自定义GRU单元：

```
python main.py --network_type=3
```

2.3 运行结果



深蓝色：使用基本RNN单元

浅蓝色：使用GRU单元

红色：使用LSTM单元

粉色：使用自己实现的GRU单元

2.4 分析与结论

首先，GRU单元的确是比LSTM，RNN单元效果更好，因此我们的实验baseline选择。其次，自己实现的GRU单元曲线和自带的GRU单元曲线基本重合，证明了自己实现的正确性。

3.学习率策略

3.1 选取原因

我选取了10^-2, 10^-3, 10^-4三种学习率，并且在三种学习率中，选取最优的学习率，测试学习率衰减的效果。

3.2 复现方法

学习率10^-3：

```
python main.py --lr_strategy=0
```

学习率10^-2：

```
python main.py --lr_strategy=1
```

学习率10^-4：

```
python main.py --lr_strategy=2
```

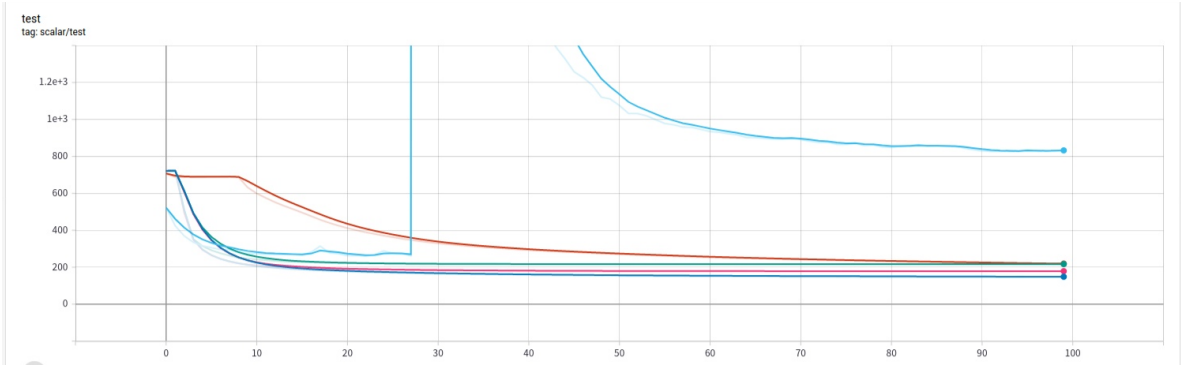
学习率10^-3, 快速衰减（每5个epoch减半）：

```
python main.py --lr_strategy=5
```

学习率10^-3，慢速衰减（每10个epoch减半）：

```
python main.py --lr_strategy=3
```

3.3 运行结果



深蓝色：学习率10^-3

浅蓝色：学习率10^-2

红色：学习率10^-4

绿色：学习率10^-3，快速衰减（每5个epoch减半）

粉色：学习率10^-3，慢速衰减（每10个epoch减半）

3.4 分析与结论

首先，10^-3学习率是三者之中最优的。

其次，学习率衰减的效果都不如不衰减。

4.优化策略

4.1 选取原因

pytorch有SGD，Adam两种常见优化方法。而且，优化中还可以使用weight decay策略，用L2余项来约束参数大小，缓解过拟合。因此，我对比了SGD，Adam，Adam带weight decay三组优化策略。

4.2 复现方法

SGD优化：

```
python main.py --optim_strategy=2
```

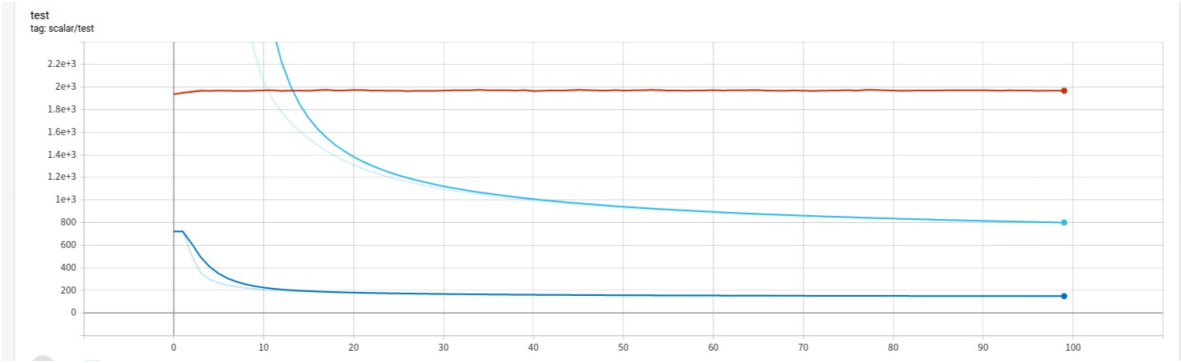
Adam优化：

```
python main.py --optim_strategy=1
```

Adam优化带weight decay：

```
python main.py --optim_strategy=0
```

4.3 运行结果



浅蓝色：SGD优化

深蓝色：Adam优化

红色：带weight decay的Adam优化

4.4 分析与结论

首先，Adam优化比SGD优化更优秀，收敛更快。

其次，因为这个网络过拟合现象并不严重，因此用weight decay的效果并不好。

5.初始化

5.1 选取原因

查阅pytorch官方文档可得，pytorch默认给RNN网络进行均匀分布初始化，有

$$u(-\sqrt{k},\sqrt{k}),k=\frac{1}{hiddensize}$$

但是，正交初始化也是一种在RNN网络中很好的初始化方式，能够有效的增加收敛速度，提高准确率（参考这篇文章）。因此，我比较了两种初始化方式。

5.2 复现方法

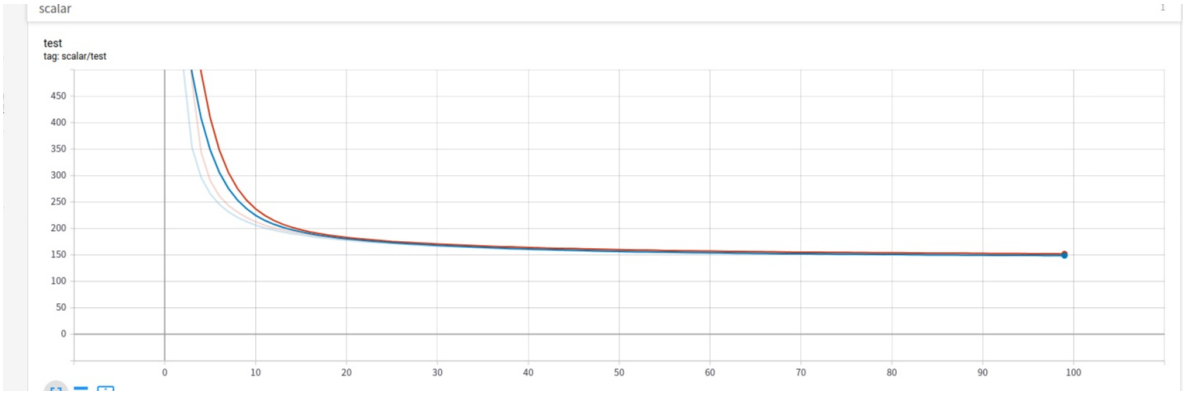
默认初始化：

```
python main.py --init_strategy=0
```

正交初始化：

```
python main.py --init_strategy=1
```

5.3 运行结果



蓝线：pytorch默认初始化

红线：正交初始化

5.4 分析与结论

经过比较，两种初始化方式的结果大致相当，使用正交初始化并不能有效提高网络效果。

6.Dropout

6.1 选取原因

Dropout能够让网络训练更快，同时减少过拟合现象，在RNN也不例外。因此，我对比了RNN层进行与不进行dropout（仅训练进行dropout）的结果。

6.2 复现方法

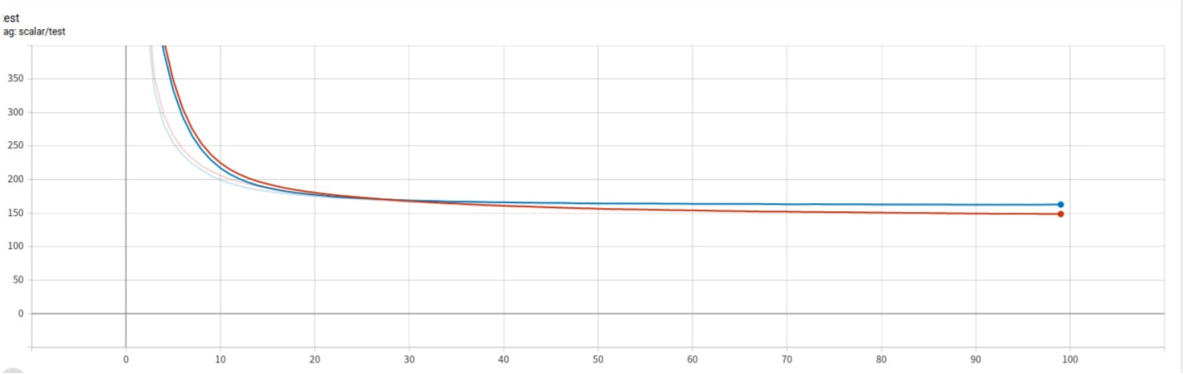
不dropout RNN层：

```
python main.py --dropout_strategy=0
```

dropout RNN层：

```
python main.py --dropout_strategy=1
```

6.3 运行结果



蓝线：不dropout

红线：对RNN进行dropout，概率0.5

6.4 分析与结论

RNN层dropout的确可以改进网络效果，因此我采用了dropout。

7.Gradient Clip

7.1 选取原因

在RNN网络中，使用Gradient Clip可以有效避免梯度爆炸。

7.2 复现方法

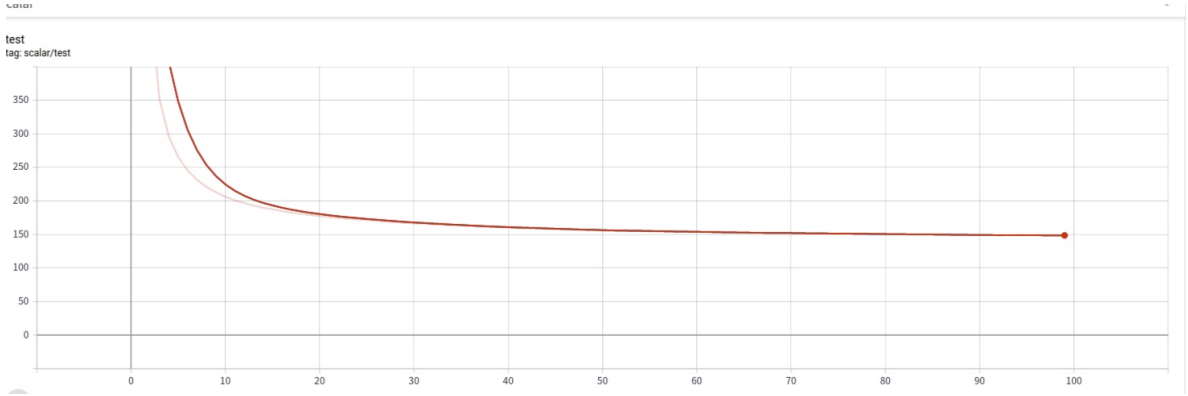
不Gradient Clip：

```
python main.py --clip_strategy=0
```

Gradient Clip：

```
python main.py --clip_strategy=1
```

7.3 运行结果



蓝线：不Gradient Clip

红线：Gradient Clip

7.4 分析与结论

是否使用gradient clip效果几乎一样。因为这个网络并没有遇到梯度爆炸的问题，因此，是否使用gradient clip没有任何区别，因此我没有使用gradient clip。

8.Weight Normalization

8.1 选取原因

在RNN中，一般不用（CNN中更常用的）Batch Normalization进行正规化。Weight Normalization可以降低网络的时间空间开销，减小小批量数据带来的噪声，缓解过拟合。因此，我比较了使用与不用Weight Normalization的实验结果。

8.2 复现方法

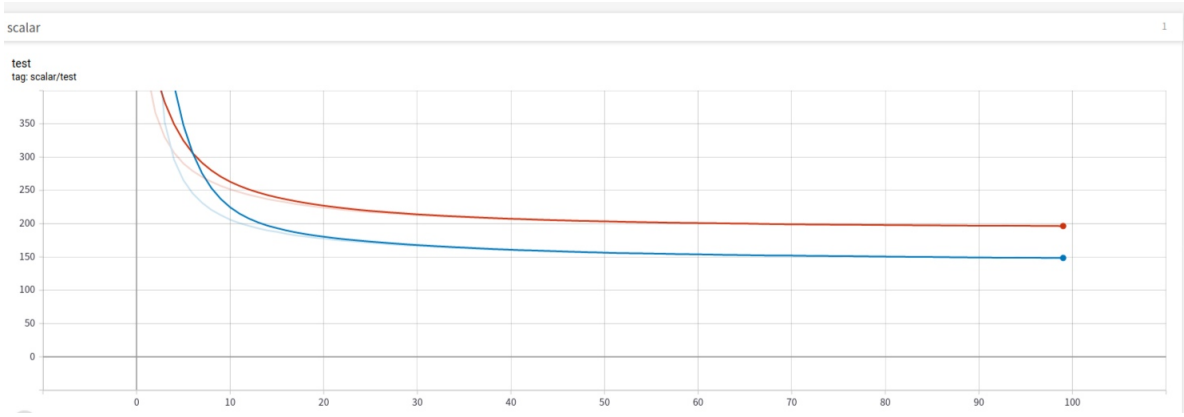
不Weight Normalization：

```
python main.py --weight_norm=0
```

Weight Normalization：

```
python main.py --weight_norm=1
```

8.3 运行结果



蓝线：不Weight Normalization

红线：Weight Normalization

8.4 分析与结论

在这个网络中，进行Weight Normalization会导致收敛更慢，因此我没有采用这种优化。

9.Layer Normalization

9.1 选取原因

除了Weight Normalization，在RNN网络中，Layer Normalization是另一种有效的正规化方法。因此，我比较了使用与不用Layer Normalization的实验结果。

但是，因为Layer Normalization需要对RNN运行的中间结果进行操作，而且pytorch的RNN单元也并没有封装这一操作方法，因此，我在自己的GRU框架下实现了Layer Normalization方法，并且与自己实现的GRU网络进行对比。

9.2 复现方法

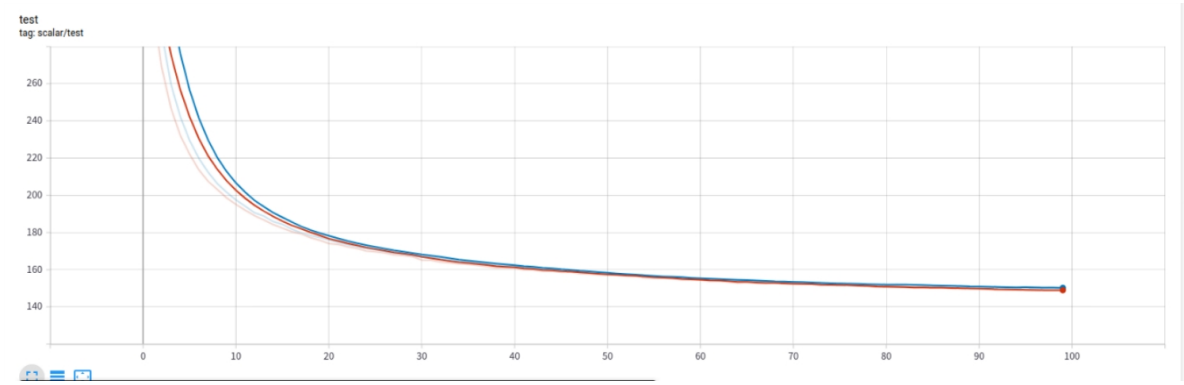
不Layer Normalization：

```
python main.py --network_type=3
```

Layer Normalization：

```
python main.py --network_type=4
```

9.3 运行结果



蓝线：不Layer Normalization

红线：Layer Normalization

9.4 分析与结论

在这个网络中，使用Layer Normalization与否，实验结果变化不大。因此Layer Normalization没什么用，我没有选取这种方法。

10.注意力机制

10.1 选取原因

我选取了self attention机制。因为在RNN中，使用temporal attention是将RNN的中间结果和词向量求得分数/相似度，但是RNN封装模块并不能任意获取中间结果，因此这样会大大降低网络的计算速度。

我的实现如下：

$$Alignment = Sortmax(score) = Softmax(a(q,k))$$
$$RNNInput = Alignment * Embedding$$

因为首先，和课上讲的不同，这个模型不是encoder-decoder模式，而是auto regressive模式，因此，我没有先通过一个RNN encoder，而是直接把embedding的结果词向量用来求attention。同样也因为这个是auto regressive模式，我也直接把attention求得的对齐结果作为RNN网络的输入，而不是作为引入的额外变量。

我尝试了两种a(q,k)的形式

$$a(q,k) = \frac{q^T k}{\sqrt{dim}}$$
$$a(q,k) = \frac{q^T W k}{\sqrt{dim}}$$

10.2 复现方法

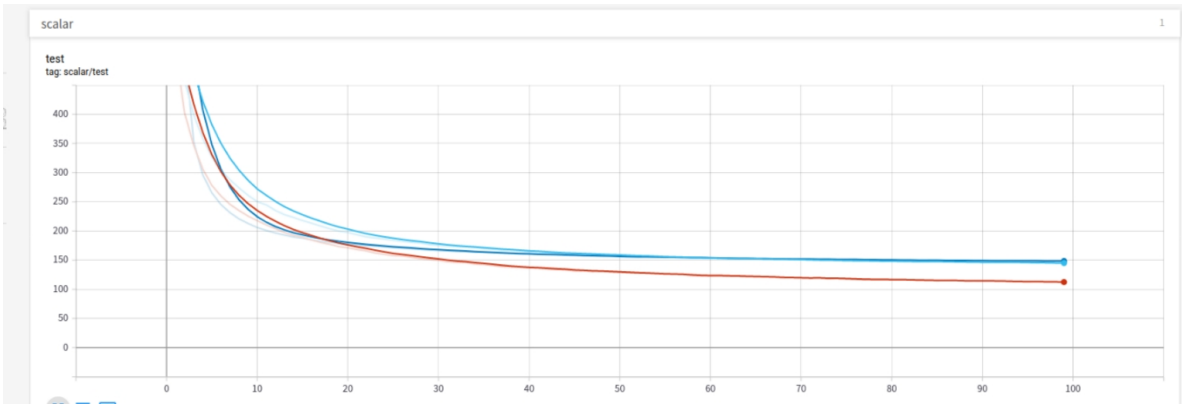
使用内积直接求attention：

```
python main.py --network_type=6
```

用一个中间可学习矩阵：

```
python main.py --network_type=7
```

10.3 运行结果



深蓝色：没有Attention机制

浅蓝色：用内积直接求Attention

红色：用了一个中间矩阵

10.4 分析

可以看出，使用内积直接求Attention并没有改进结果。因为这样求Attention，实际上是根据和当前待输入的词向量的相关性来分配权重，那么最终就会收敛到“只有当前待输入词向量得分1，其余都是0”，这就和没有Attention机制差不多了。

但是用一个中间矩阵的效果就比较好，中间矩阵的表达能力让我们进一步优化了RNN的输入，让RNN的输入能更好的关联其余单词的信息，使得网络得到了改进。

但是，这种直接求Attention的方法其实是有问题的。因为这是一个语言建模任务，每个地方单词出现的概率只应该和其前面单词出现的概率有关，不应该和其后面单词出现的概率有关。因此，我修改了Attention，使得出现在该处后面的词汇对Attention贡献为0.

10.5 修改后的复现方法

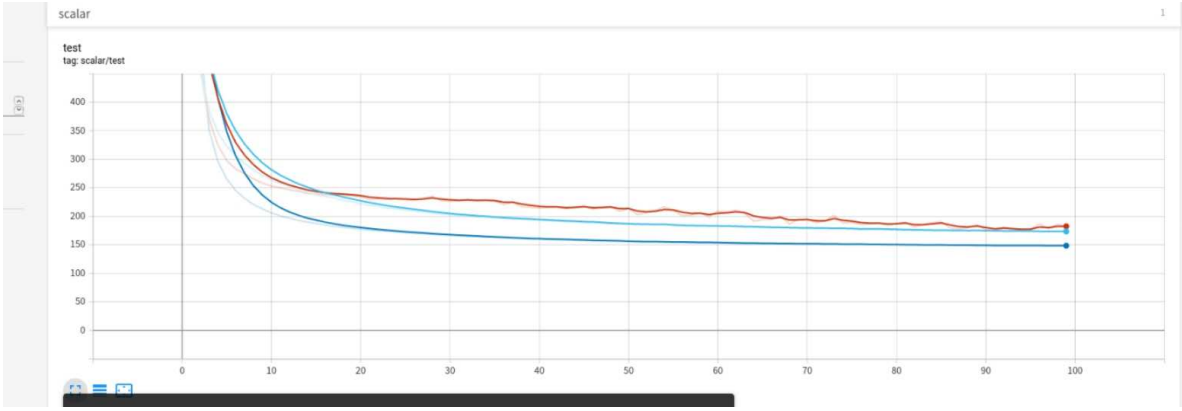
使用内积直接求attention：

```
python main.py --network_type=6
```

用一个中间可学习矩阵：

```
python main.py --network_type=7
```

10.6 修改后的运行结果



深蓝色：没有Attention机制

浅蓝色：用内积直接求Attention

红色：用了一个中间矩阵

10.7 修改后的结果分析

此时，使用Attention的收敛速度就更慢了，无论是用内积还是用一个中间矩阵。

对于只用内积的网络，我猜测是修改Attention的操作增加了网络的复杂性，使得网络收敛更慢，更容易过拟合了，导致其比不用Attention还差。

对于使用中间矩阵的网络，根据10.3的结果，我猜测这样做理论上是能改进网络效果的，但是，因为修改Attention的操作增加了网络复杂性，网络的表现反而更差了。

10.8 总结

综上所述，按照我的方式在这个auto regressive的任务中使用self attention，如果只使用内积，并不能改进网络，因为按照和当前输入词的相关性分配权重，和直接输入这个词差别不大。如果使用一个可学习的参数矩阵，能够一定程度上学习到一些需要注意的其他单词，有助于改进效果。

但是这种attention的益处，因为这个任务--language modelling的特殊性质被抵消了。language modelling要求对每个位置出现单词的概率只和前面的词有关，和后面的词无关。因此，就必须在求softmax前，对attention的结果进行对角化操作，保证后面的词不会被注意到。这样增加了网络复杂性，也使得网络更难收敛，更容易过拟合。

综上，在language modelling任务中使用self attention改进RNN不是一个很好的选择。

总结

这次实验，让我深入了解了RNN网络的内部单元和架构，还有attention机制，尤其是用于语言模型的Auto Regressive的架构。在CNN作业的基础上，我对于网络训练技巧---学习率，优化方式，初始化方式，dropout，各种Normalization，都能熟练运用了。同时，通过实现GRU网络的正向传播，我也对pytorch有了更加深入的了解，但是自己实现的正向传播并行度比较差，这是需要将来进一步深入学习改进的。

但是，我也发现了pytorch框架的一些问题：pytorch框架不支持RNN中的Layer Normalization，也不支持读取RNN网络的中间变量，这样给很多操作带来了很大的困难。因为这些原因，我必须手动实现GRU网络来测试Layer Normalization，以及难以使用temporal attention（自己实现的网络并行性什么的很差，容易复杂度远远大于框架，训练速度非常慢）。