人工神经网络 第3次作业报告

董又铭 计53 2015011294

1. **框架学习 & 代码补全**

本次作业采用了TensorFlow深度学习框架，需要在已有代码的基础上补全一些东西。由于TensorFlow框架的限制，要想像前两次作业一样自己实现网络层的正向和反向传播会非常困难，所以本次作业的本质上就是熟悉TensorFlow框架的使用。

奔着“熟悉TensorFlow的框架”的目的，可以对作业中原本给出的代码进行一些理解和学习。

* 1. Model.forward函数

本次作业的forward函数，与前两次作业的forward函数完全不同，本次作业的forward的函数是使用TensorFlow的变量和占位符“描述”一个网络的正向传播过程，真正训练时的正向传播并不是在这里进行的。

注意到forward函数在Model.\_\_init\_\_函数中调用了两次，分别描述训练用的网络和测试用的网络。显然这两个网络有很多相同的部分，比如网络的整体结构、linear和conv2d层的参数等；也有一些不同的部分，比如bn层和dropout层的具体过程。在两种网络有不同的实现方式的地方，用is\_train变量来区别对待。

在forward函数的第1行，tf.variable\_scope实际上是开启了一个命名空间，这个命名空间叫做model，并通过reuse来设定是否允许使用已有的变量。在此之后，合理使用tf.Variable和tf.get\_variable来创建变量，前者一定会创建新的变量，后者会检查是否已经有名称、形状、数据类型都完全相同的变量，如果已经有则使用原有的变量。

另一个需要处理的问题是，bn层在测试用的网络中需要使用训练时的均值和方差，这两个变量需要在训练的同时进行维护，这需要在创建训练优化器AdamOptimizer的时候增加一点小操作。

* 1. 网络层的实现

本次作业需要实现的层比较多，但因为有TensorFlow框架，我们并不太需要关心每个层内部的细节，大多数的层都只需要一句话就可以实现

|  |  |
| --- | --- |
| 网络层 | 用TensorFlow实现的例子 |
| Linear | output = tf.layers.dense(input, 10) |
| Relu | output = tf.nn.relu(input) |
| Conv2d | output = tf.layers.conv2d(input, 64, 3, 1, ‘same’) |
| Max\_pool\_2d | output = tf.layers.max\_pooling\_2d(input, 2, 2) |
| BN | output = tf.layers.batch\_normalization(input, training=True) |
| Dropout | output = tf.layers.dropout(input, 0.3, training=True) |
| Reshape | c = 1  for dim in input.shape[1: ]:  c \*= dim  output = tf.reshape(input, [-1, c]) |

* 1. 正确使用GPU加速

在安装正确版本的CUDA、cudNN以及GPU版本的TensorFlow之后，可以用Nvidia的GPU进行加速；在Ubuntu下正确安装ROCm和OpenCL也可以用AMD的GPU加速。我这里采用RTX2080Ti加速，搭配CUDA 9.0和cudNN 10.0，很遗憾TensorFlow目前不支持CUDA 10.0，无法利用RTX显卡的tensor核心进行更进一步的加速。

由于显卡除了被用来做计算以外，还要承担常规的显示器画面输出的计算，虽然常规使用占用的计算资源和显存资源都比较少，但也不可忽略。TensorFlow在默认的模式下，会先将几乎所有的显存全部申请占用，然后再进行计算。虽然这样可以减少动态申请的开销，但显然这是不明智的：比如我希望让一份作业代码在后台运行训练，同时手上还在Debug另一份代码；甚至直接运行多份代码，同时训练不同的模型；这些操作在默认模式下时可能导致Windows蓝屏的。

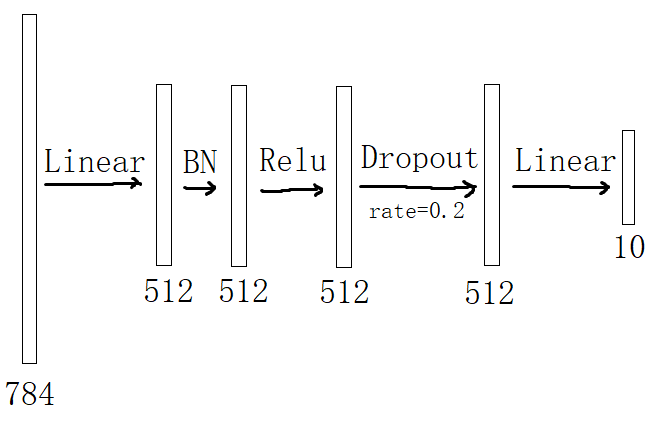
为了解决进程级别的并行对显存的抢占，可以在开启TensorFlow会话的时候加入一些限制：

|  |
| --- |
| sess\_config = tf.ConfigProto()  sess\_config.gpu\_options.per\_process\_gpu\_memory\_fraction = 0.45  sess\_config.gpu\_options.allow\_growth = True  with tf.Session(config=sess\_config) as sess:  ... |

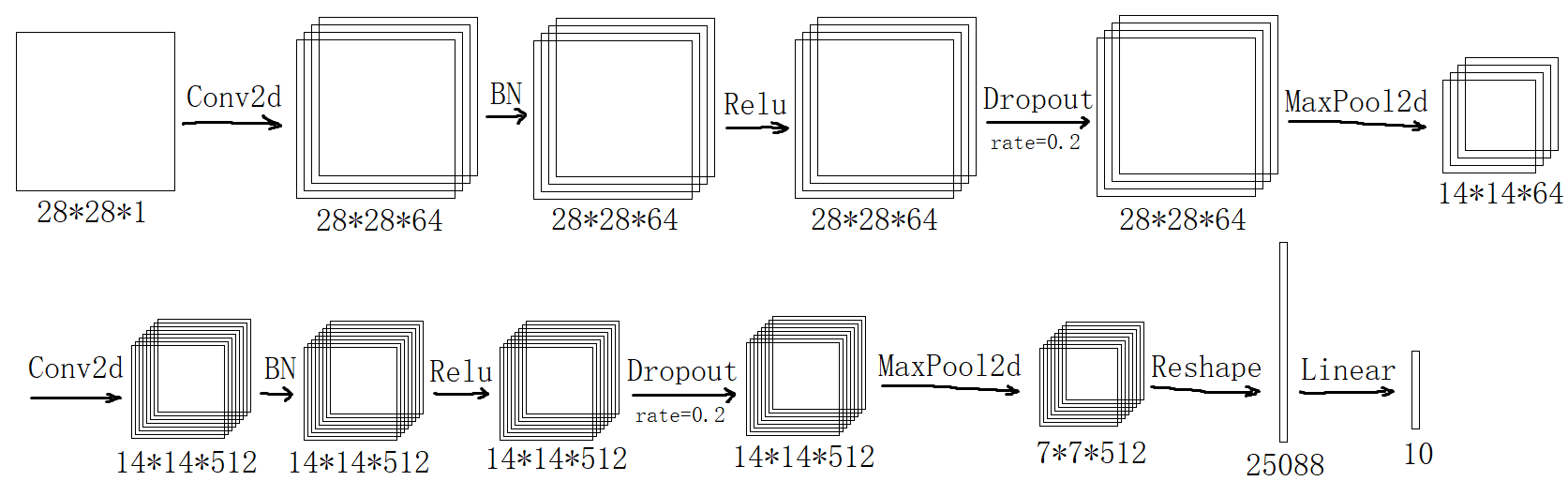
这里限制了两个方面，第2行是限制当前sess的显存占用不能超过总显存的45%，第3行是要求当前sess只有在必须用到这部分显存的时候再去申请占用。有了这样的限制之后，就可以同时运行多个TensorFlow进程了。

1. **对比MLP和CNN**

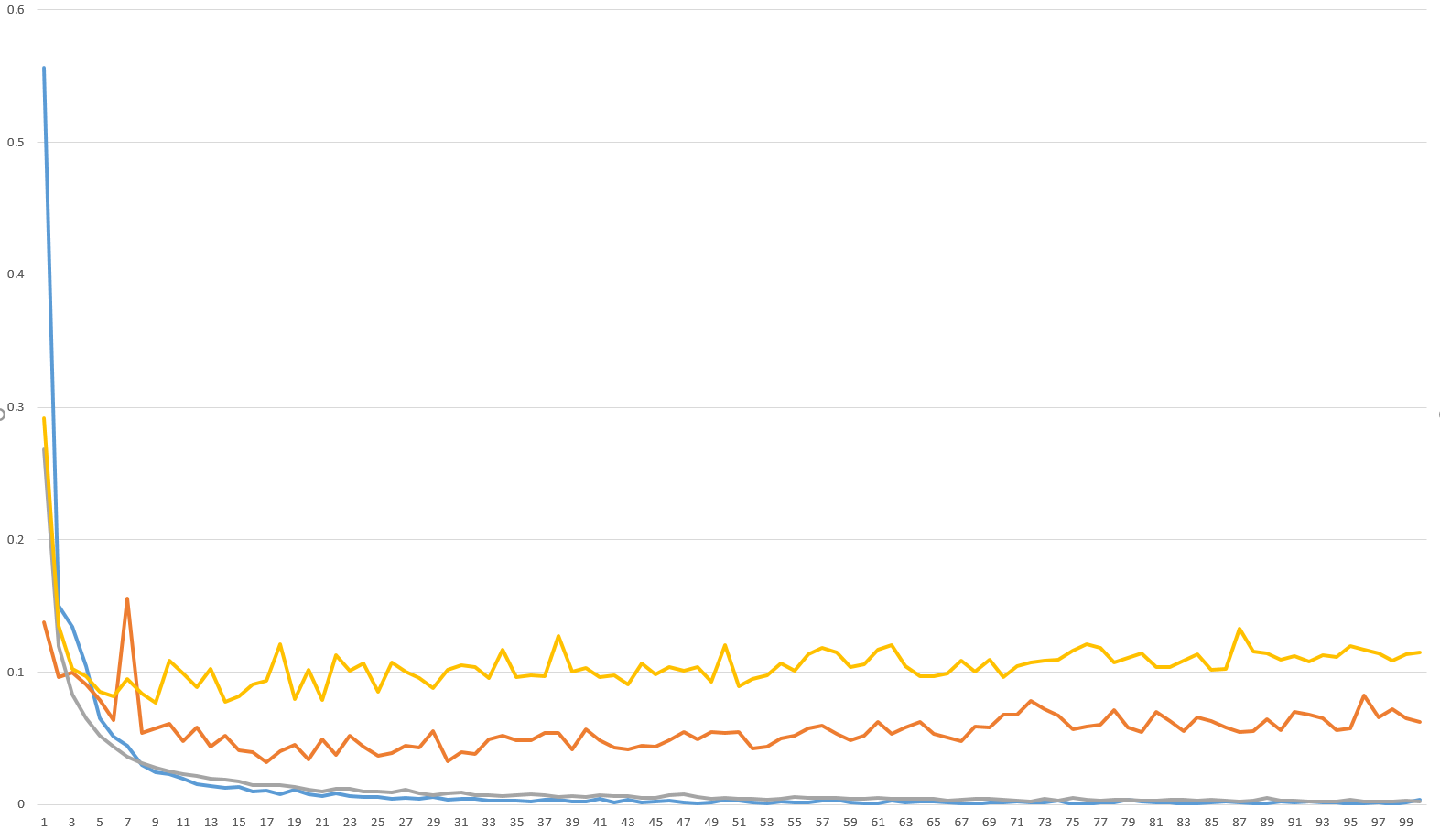
MLP的结构如下图



CNN的结构如下图



训练100个epoch，每个网络每个epoch在训练集合测试集上的loss值如下图。其中蓝色和橙色分别是CNN的训练集和测试集loss函数值，灰色和黄色分别是MLP的训练集和测试集的loss函数值。

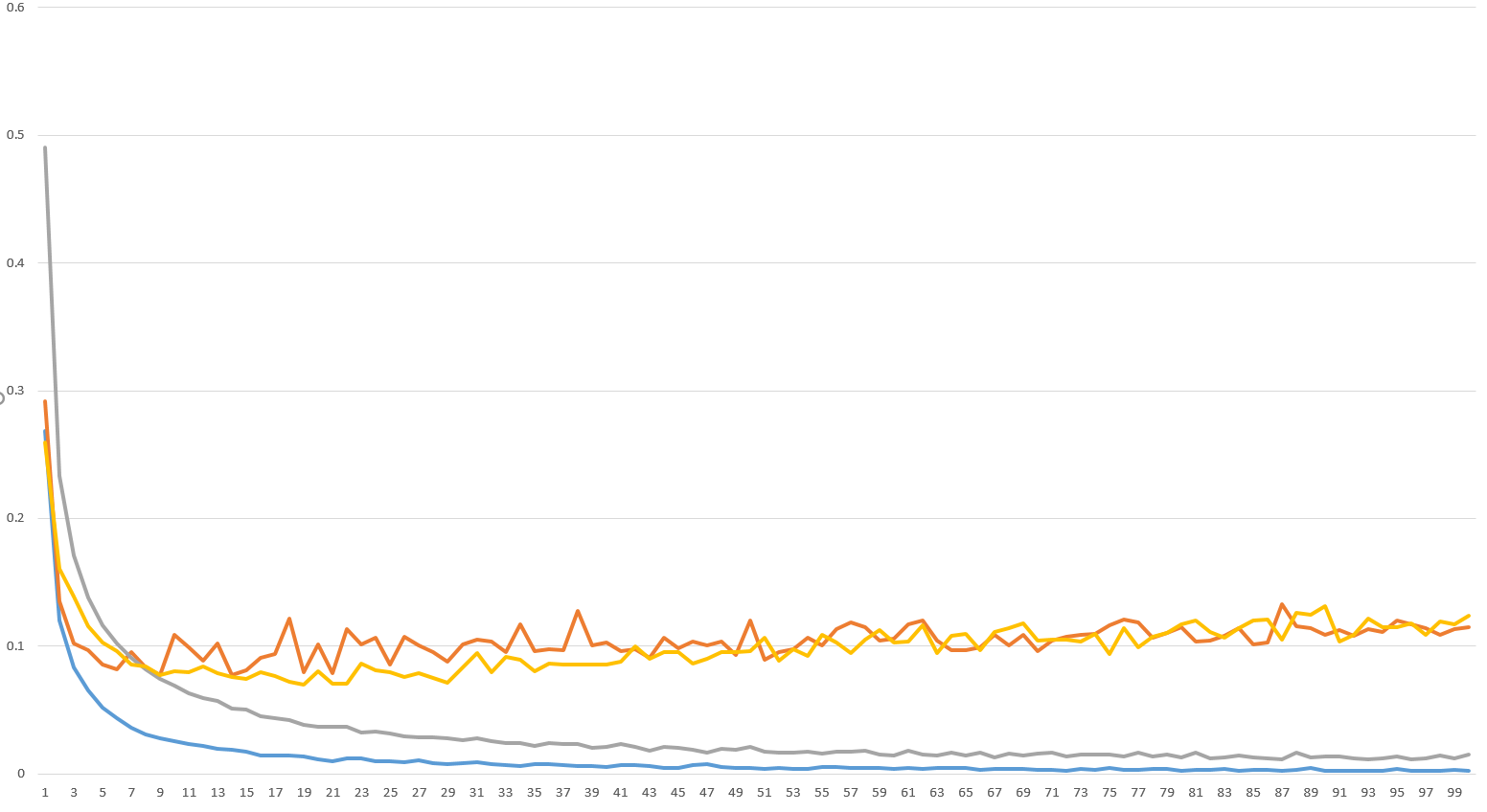


可以看到，在训练一段之间之后，虽然训练集上的loss函数值仍然在缓慢的下降，但在测试集上的loss函数值却在随机的波动，甚至还有缓缓上升的趋势。但是总体来看，CNN的情况要比MLP稍好一些。

1. **BN层对CNN和MLP的影响**

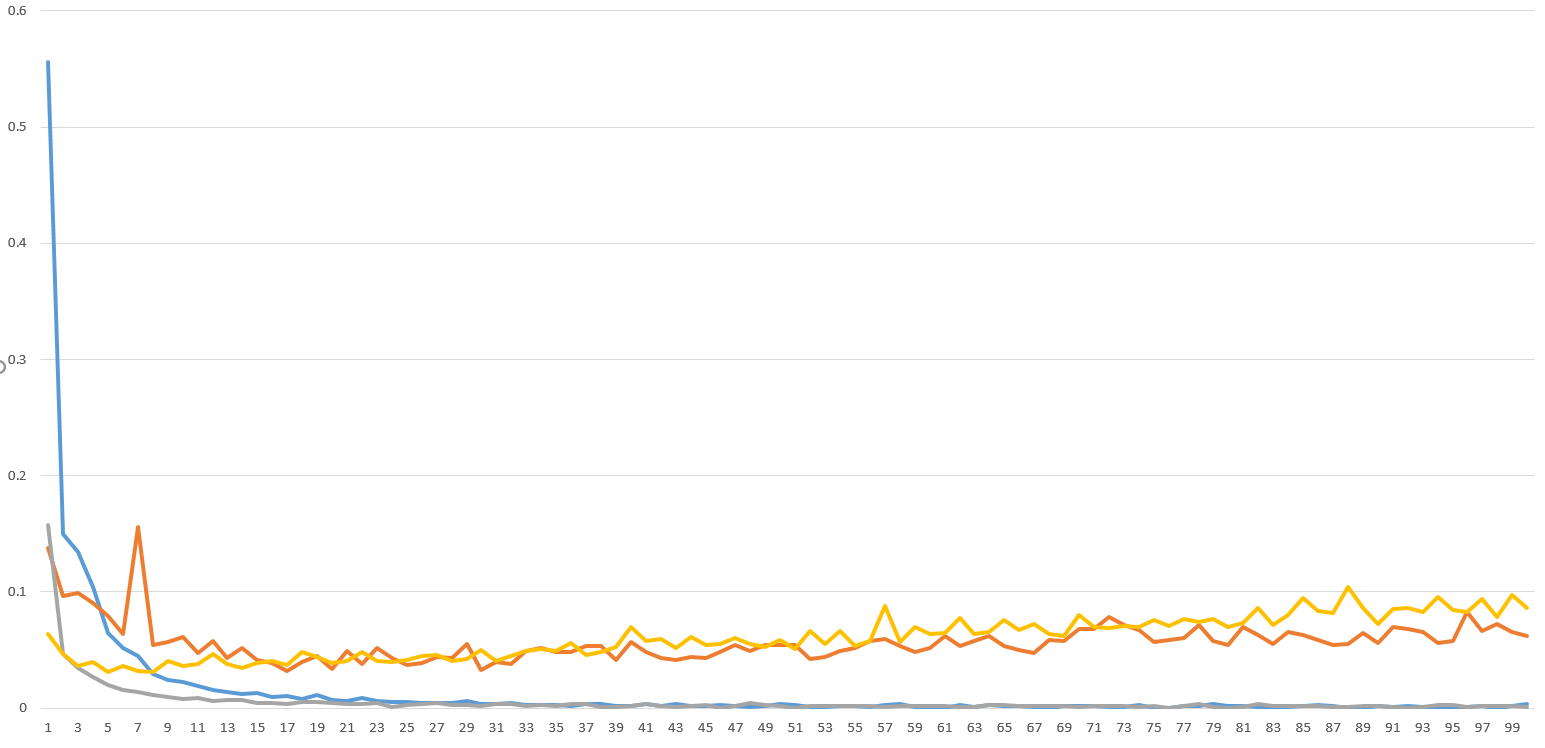
使用BN层时网络结构和上一小节的网络结构保持不变，不使用BN层的网络结构就只需要移除所有BN层。

两种情况下MLP的loss函数值如下图，其中蓝色和橙色是使用BN层的训练集和测试集loss函数值，灰色和黄色是不使用BN层的训练集和测试集的loss函数值。



可以看出，使用BN层之后，网络模型的收敛速度有大幅提升，在训练集上的现象尤其明显，在测试集上也有一定的体现。同时，在训练集上的最终精度也有一定的帮助，但测试集上的精度没有明显的变化。

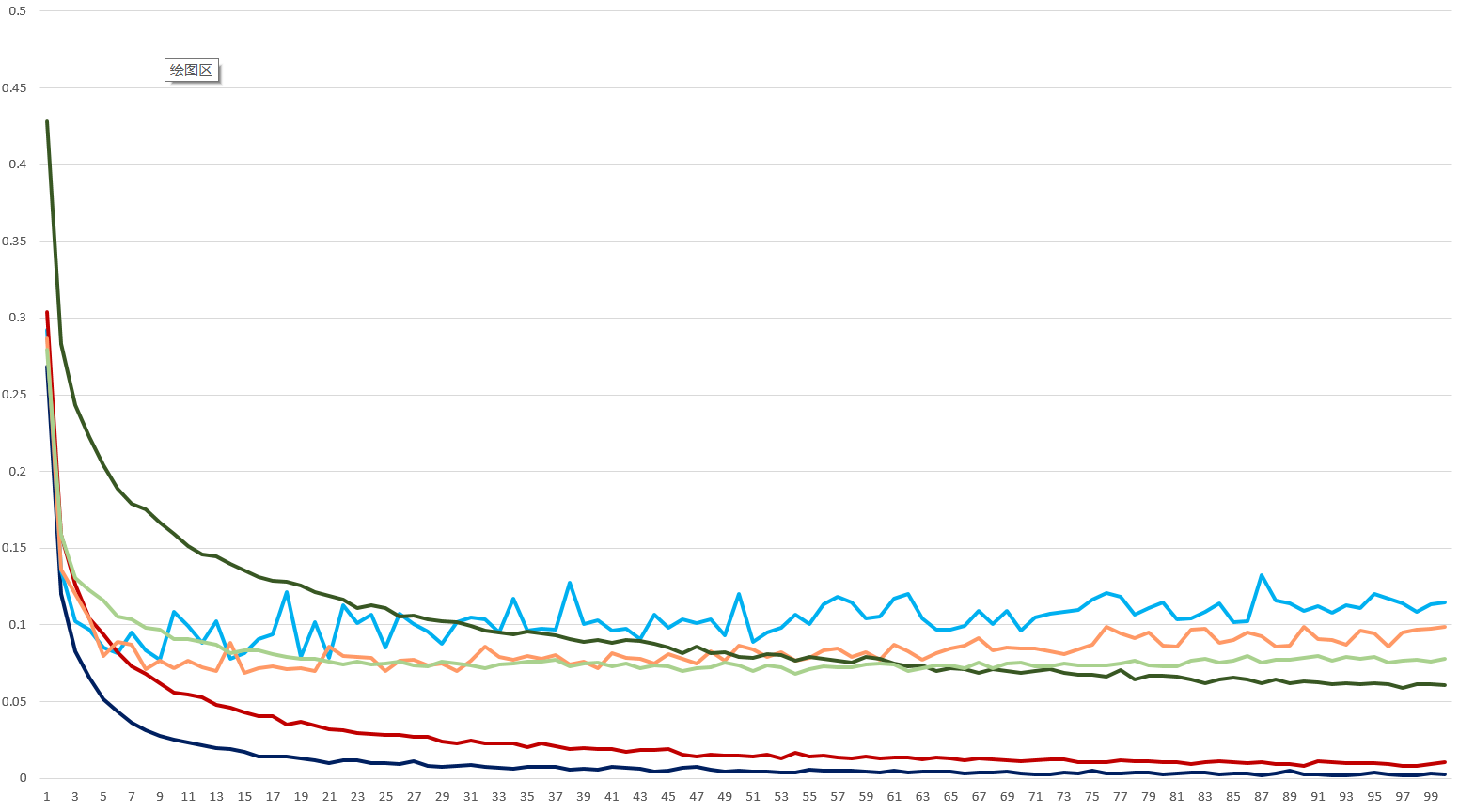
类似的，CNN在使用和不使用BN层点的情况下，loss函数值如下图所示。颜色同上，蓝色和橙色是使用BN层的训练集和测试集loss函数值，灰色和黄色是不使用BN层的训练集合测试集的loss函数值。



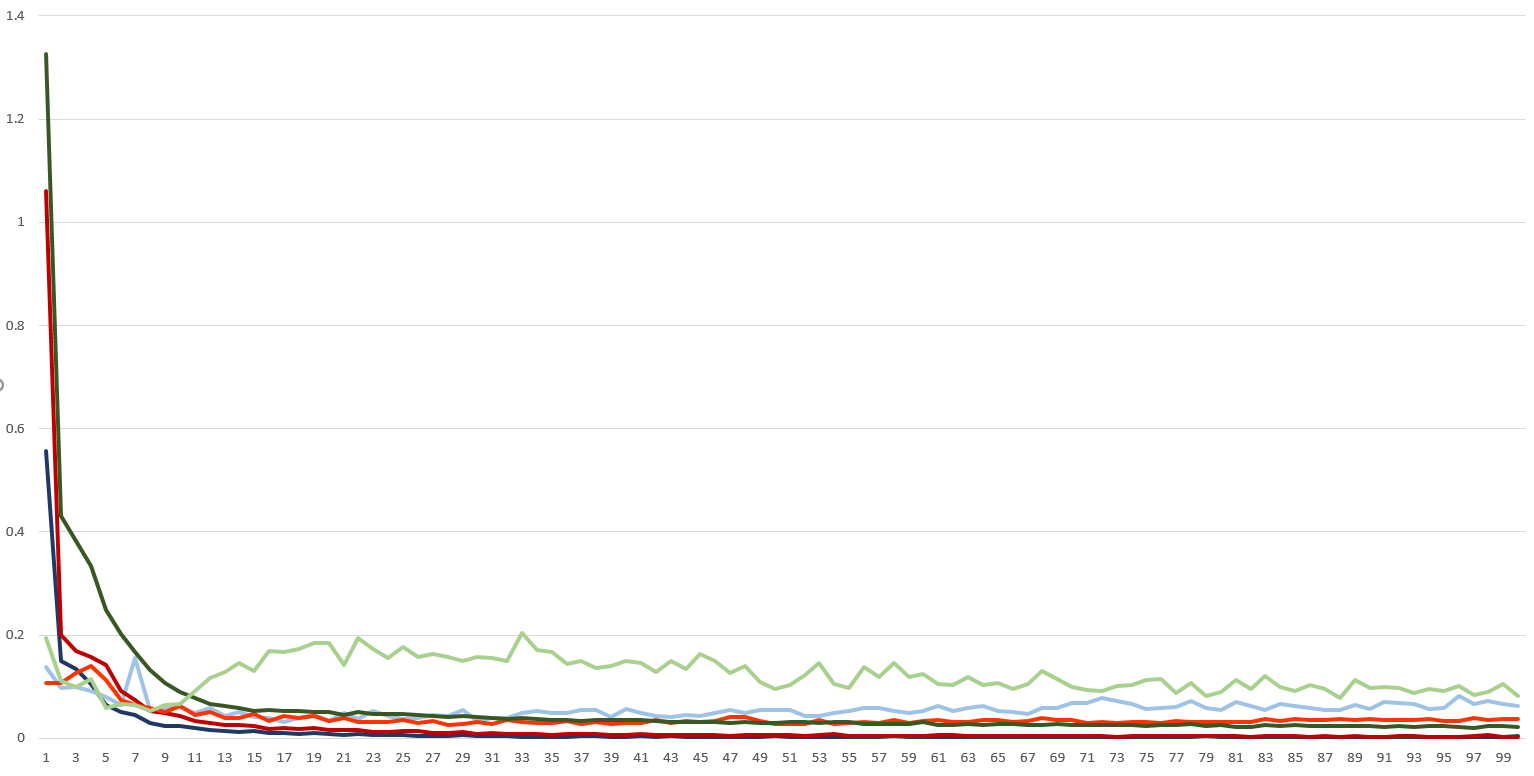
有趣的是，不使用CNN的BN层反而会收敛更快，而最终精度也没有什么区别。经过多次实验，修改参数实验，均为这个现象。

1. **Dropout层的drop\_rate对CNN和MLP的影响**

首先是MLP，分别取drop\_rate=0.2、0.5和0.8运行，训练集和测试集的loss函数值如下图，其中深蓝和浅蓝是0.2，棕红和橙红是0.5，墨绿和浅绿是0.8。



同理CNN如下图



观察MLP的实验结果可以比较容易的发现，drop\_rate越大，在训练集上的loss函数值越大，模型精度越低；但是在测试集上的loss函数值会相对较小，模型精度越高。这个规律在CNN网络上也有一定的体现，但不是特别明显。一方面因为CNN网络本身模型精度就很高，另一方面可能是因为drop\_rate=0.8这个值对于CNN来说过分的高了，以至于精度提高的效果有明显的反弹迹象。

考虑到训练网络模型的最终目标是让测试集上的模型精度越高越好，所以在一定范围内增大drop\_rate会有非常好的效果。

1. **为什么训练集合测试集的loss函数值会有如此大的偏差呢？**

神经网络这种东西嘛，过拟合在所难免，哪怕是用Dropout层降低过拟合带来的影响，也只是降低影响，并不能消除影响。

训练时是以训练集上的数据计算梯度实现梯度下降，而测试集中的数据从始至终都没有进行梯度下降，所以在网络趋于稳定之后就等价于随机调整，loss函数值也是比较随机的在波动。但是在训练集上，由于不断的梯度下降，loss函数值也就会一直有降低趋势，而在测试集上趋于稳定之后的训练过程，其实就变成了挖掘训练集自身的各种细微特征的过程，这种对于测试集并没有什么帮助，但确实能够使得模型在训练集上的精度继续的提高。

由于Minst数据集本质上是一个分类问题，而我们使用的Softmax+交叉熵的loss函数，通过一张图是0-9每个数字的概率计算出损失。假如要使得loss=0，必须是Softmax之后的数值为0或者1，但显然Softmax函数的值域是不包含0和1的，只能无限趋于0和1，而无限趋于0和1就意味着Softmax的输入数据出现正无穷或者负无穷，也就是每张图经过网络后最终数据出的10个数中存在正无穷或者负无穷。这其实是把已经获取到的特征再进行强化的过程，而这个过程只会影响在训练集上的loss函数值，并不会影响训练集上的分类模型的正确率，甚至对于测试集上的loss函数值和正确率都有可能起到相反的作用。这从另外一个角度解释了，为什么在模型趋于收敛之后继续训练，测试集上的loss函数值总是会在随机波动的过程中呈现出缓慢增大的趋势。