Lab 1 实验报告

PB20000296 郑滕飞

准备工作：

1、实验要求

用前馈神经网络拟合给定函数，并观察超参数影响。

2、数据生成

我采用了 gen_input 函数进行了数据生成，总体逻辑是在[0,2π)间等距选取 N 个点，并在计算出对应的值后将它们按照 5:2:3 的比例随机划分为数据集、验证集、测试集，从而保证互不相交与随机抽样的效果。实际实验中，N 取 1000，这意味着数据集大小为 500、验证集 200、测试集 300。

3、网络模型

由要求，需要对比不同参数下的网络模型，因此我采用的网络如下：假设深度为 d，宽度为 w，输入层为 1 输入 w 输出，之后每个隐藏层 w 输入 w 输出，输出层 2 输入 1 输出，所有相邻层之间全连接，隐藏层总个数为 d-1。

初始化与训练如下：

```
class FunctionNet(nn.Module):
    def __init__(self, width, depth):
        super(FunctionNet, self).__init__()
        self.input = nn.Linear(1, width)
        self.hidden = [nn.Linear(width, width)] * (depth - 1)
        self.output = nn.Linear(width, 1)

    def train(self, X_train, y_train, X_cr, y_cr, epoches, lr):
        optimizer = opt.SGD(self.parameters(), lr)
        loss_func = nn.MSELoss()
        out1 = []
        out2 = []
        for _ in range(epoches):
            y_p = self(X_train)
            loss = loss_func(y_p, y_train)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
            out1.append(loss.data.numpy())
            out2.append(loss_func(self(X_cr), y_cr).data.numpy())
        return out1, out2
```

输入的 train 与 cr 分别代表训练集与验证集，在每次训练后计算训练集与验证集上的 MSE，并输出两个数组，方便绘图、调试。

对于输出过程，forward 步由于要测试不同的激活函数的影响，会有不同的选择，以 relu 激活函数为例：

```
    def forward(self, x):
        out = nn.functional.relu(self.input(x))
        for layer in self.hidden:
            out = nn.functional.relu(layer(out))
        return self.output(out)
```

在通过输入层与每个隐藏层时通过激活函数，最后通过输出层。

若用 sigmoid 函数则变为：

```
    def forward(self, x):
        out = torch.sigmoid(self.input(x))
        for layer in self.hidden:
            out = torch.sigmoid(self.layer(out))
        return self.output(out)
```

参数测试与讨论：

1、测试函数

以下先讨论 relu 为激活函数的情况。在给定的激活函数下，模型考虑了四个超参数：深度、宽度、学习率与迭代次数。

为了测试参数的性能，我构建了如下的测试函数：

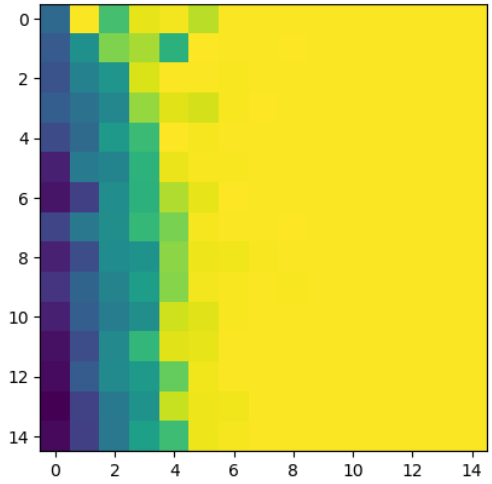
```
def test_c(width : int, depth : int, epoches : int, lr : float, if_plot : int):
    a = FunctionNet(width, depth)
    x_train = torch.load("x_train").float()
    y_train = torch.load("y_train").float()
    x_cri = torch.load("x_cri").float()
    y_cri = torch.load("y_cri").float()
    out1, out2 = a.train(x_train, y_train, x_cri, y_cri, epoches, lr)
    if (if_plot):
        pl.figure()
        pl.plot(np.log(out1))
        pl.legend("train")
        pl.plot(np.log(out2))
        pl.legend("cri")
        pl.show()
    return np.min(out2), np.argmin(out2)
```

在按照给定参数进行训练后，绘制训练集和验证集上的损失曲线(由于损失量级变化较大，这里采取 log 后绘制的办法，可以更清楚地看出变化趋势)，并给出训练中验证集上最小的误差及出现的位置(于是可以判定过拟合)。

2、深度与宽度

由于所拟合的是单输入单输出的简单函数，直觉上来看网络理应不太复杂，因此，我做了一组测试，深度从 1 取到 15，宽度从 5 取到 75(步长为 5)，在同样的 0.01 学习率(根据简单测试，这个学习率比较方便共用)，500 次迭代下，观察收敛效率。

最终，采用 relu 激活函数时得到的误差如下：

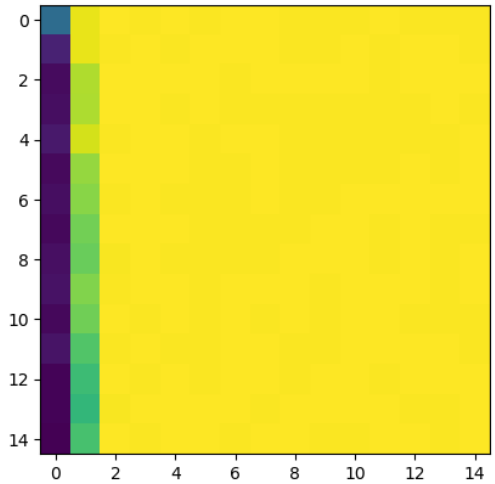


图中，横轴表示深度提升，纵轴表示宽度提升。可以直观看到，宽度提升对结果是有一定的正面影响的，而深度提升几乎只有负面的影响。收敛性态在只有一层隐藏层时普遍比右侧要好。事实上，根据后续测试，深度提升后调低学习率也能有一定正面影响，但收敛速度已经不如此时的结果。

测试集最低误差出现的位置如下(499 代表最后一次，也即未收敛)：

```
[[499 499 499 499 499 499  98 485   0 499  98 496  89   7   0]
 [499 499 499 499 499 499 499  59 499  58 498 448  54  65  71]
 [499 499 499 499 499  50  60  58  73 491  49 452  86 458  51]
 [499 499 499 499 499 499 499 499 487 499   0 474   0  73   0]
 [499 499 499 499 499 499 499 499 493  54 499 444 469  11  17]
 [499 499 499 499 499 499  44 499 495 490  27 473 465 459  46]
 [499 499 499 499 499 499 499 499 497  45 494 414  27  10   0]
 [499 499 499 499 499 499 499  48 499  17 499   0 458 424  43]
 [499 499 499 499 499 499 499   0  39  17 489 483  46 494   0]
 [499 499 499 499 499 499 499 499 499 497  23  18  34 438  16]
 [499 499 499 499 499 499 499  13 499 494  41   0   0 453   6]
 [499 499 499 499 499 499  30  16 499   0  34   8  39  48 443]
 [499 499 499 499 499 499  30 499 499 499 472  24 440  19 412]
 [499 499 499 499 499 499 499 499   5   9   2 464 426  38   0]
 [499 499 499 499 499 499 499  33  35 499   0 474 448  21  17]]
```

可发现，深度提高时更早进入了收敛(或者说“震荡”)状态，因此导致性态不够好。对 sigmoid 函数作为激活函数时，由于其含指数的性质，这一变化更加明显：



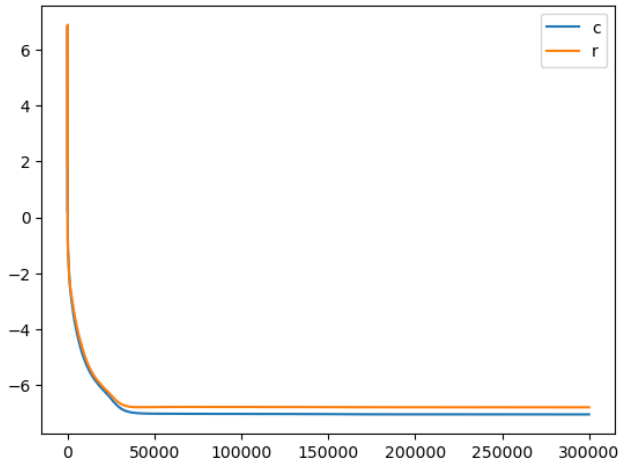
总的来说，想要达到良好的收敛性态，对我们的简单例子而言，单个隐藏层就有足够好的效果了，无需更高深度进行近似。不仅如此，低深度的反向传播容易，还有着更好的收敛效率。

3、学习率与收敛速度

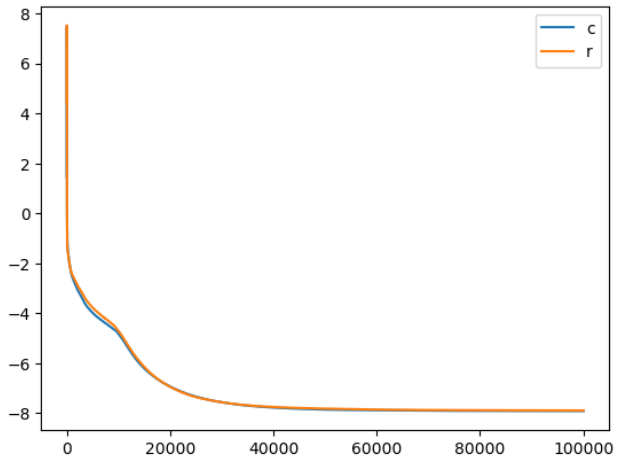
仍考虑 relu 函数，通过对损失曲线的观察，0.01 时基本不会出现震荡，因此直接以此学习率进行之后的计算。而由讨论，深度取单隐藏层，宽度取至少 75 时，已经可以有较好的效果。

上方的 500 次收敛只是测试不同参数下的效率，想要达到更好的结果，可以进行远比它更多的迭代。因此，下面需要对不同宽度、学习率进行更多检验。

首先，在宽度 75，迭代 300000 次下的一个结果是这样的：验证集最小损失 0.0011，在第 299972 次迭代后取到，对数损失曲线如下，其中 c 为训练集，r 为验证集：

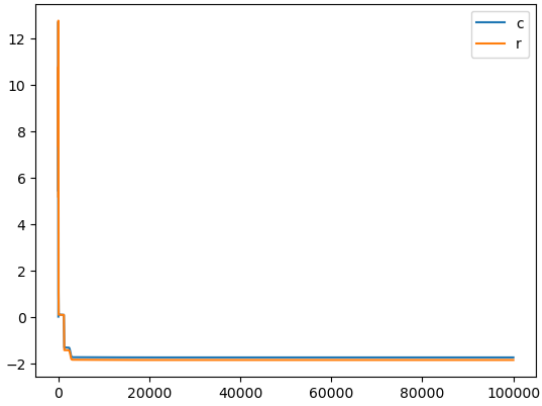


此结果意味着，在 300000 次之前基本达到了收敛，而最终误差量级在 1e-3。宽度 125，迭代 100000 次可以得到：验证集最小损失 0.00037，在第 99995 次迭代后取到，对数损失曲线为：



从这两次的结果来看，似乎网络宽度更大时效果能更好，但事实上，其中有很大的随机性。在多次测试中，平均的误差基本在 1e-3 左右，较好的结果能达到 1e-4 量级，但是，其中偶然会有 MSE 在 0.1 左右的结果，根据损失曲线判断，很可能是参数收敛到了某一并

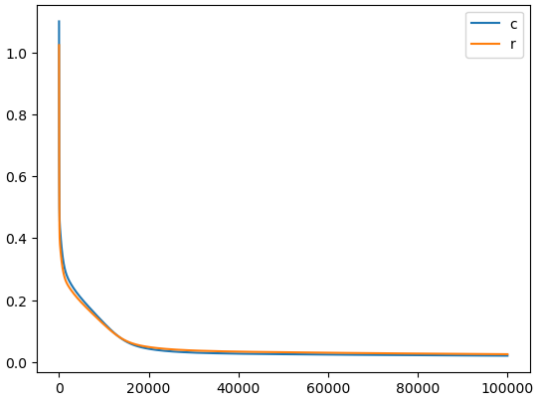
不好的局部极小点，这是前馈神经网络进行梯度下降所无法避免的。并且，哪怕更大的宽度也无法解决这一问题，宽度为 500 时，可以调高学习率至 0.1，一次结果如下：迭代 100000 次，验证集最小损失 0.1574，在第 33046 次迭代达到，损失曲线为：



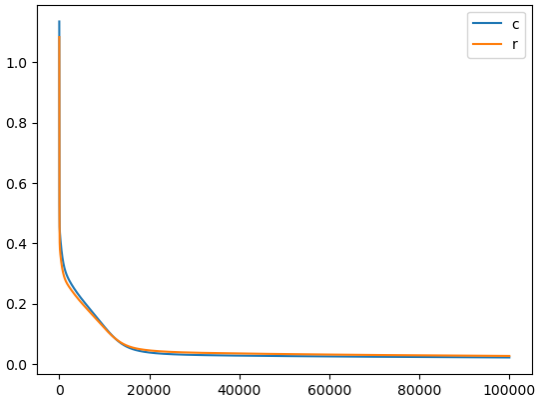
可以发现，稳定收敛的结果并不需要过多迭代达到，但想要较好的结果，更多取决于初始随机的情况。

4、sigmoid 损失函数的情况

sigmoid 损失函数时的参数热力图已在之前给过，由此，仍然选择一层隐藏层观察。由于其值域特性，损失曲线无需采用 log 表示，以宽度 75，迭代 100000 次下的一个结果是这样的：验证集最小损失 0.0026，在最后达到，损失曲线为：



事实上损失曲线基本形态都如上，例如宽度变为 100 时：

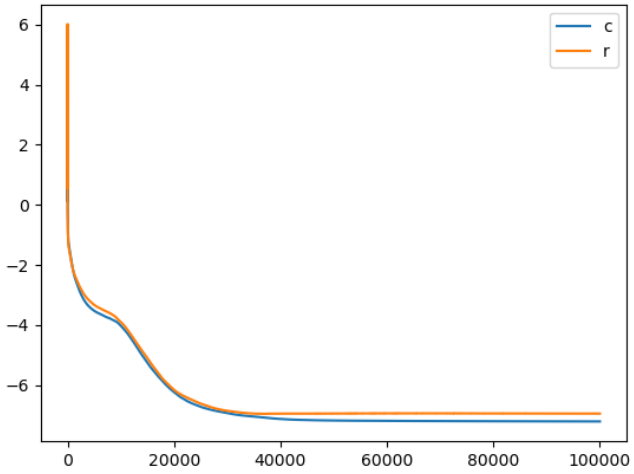


但是，在类似上方的测试中，误差始终落在 1e-2 量级，这可能是由于着我们给定的函数有较好连续性，并不满足 sigmoid 所更擅长的分类情况。因此，最终选用的激活函数仍是 relu。

结论：

1、最终结果

最终激活函数 relu，参数：隐藏层数 1，宽度 100，迭代次数 100000，学习率 0.01，生成的对数损失曲线如下：



验证集 MSE 为 0.00095047726，测试集 MSE 为 0.0010445806，属于平均情况的结果。

2、总结

这次实验的难度不高，主要是手写神经网络的部分要正确实现。对于简单函数，并不需要过于复杂的深度与层数，而在深度较低时一般能有更好的收敛速率。但是，实验中前馈 神经网络也暴露出了问题：落于局部最优的随机性。对此，一方面可以考虑采取一定程度的随机梯度下降，或更多考虑全局的优化方式进行误差传播，另一方面也可以改进网络的结构使结果的稳定性更强。

Lab 2 实验报告

PB20000296 郑滕飞

准备工作：

1、实验要求

利用卷积神经网络进行图像分类。

2、网络模型

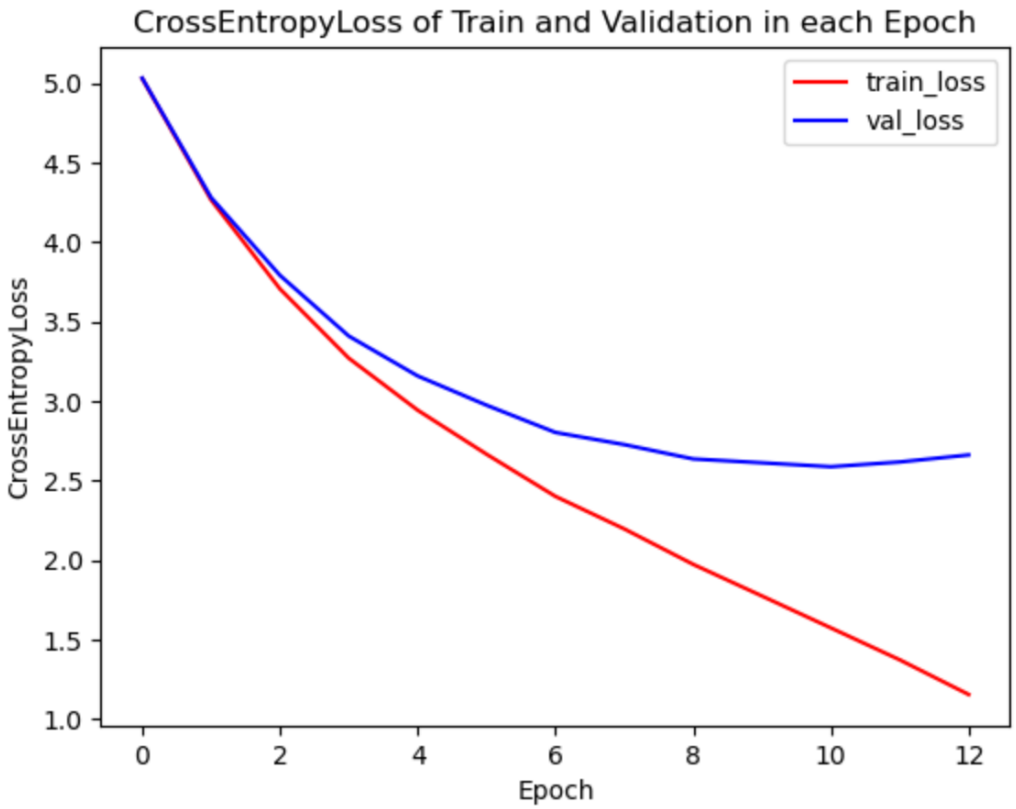
助教框架的网络由六组卷积网络-两层残差网络-池化层构成与最后线性化生成输出构成，其中卷积网络可以自行调整层数。每组网络的宽度分别为 64、128、256、512、256、128，由卷积层转换宽度。

参数测试与讨论：

1、迭代次数的影响

实验中，epoch 与 wait 联合控制迭代次数。达到最大 epoch 或验证集最优误差连续 wait + 1 次未下降后即结束迭代。

将 wait 设置为 1，直接运行框架得到的损失曲线如下：

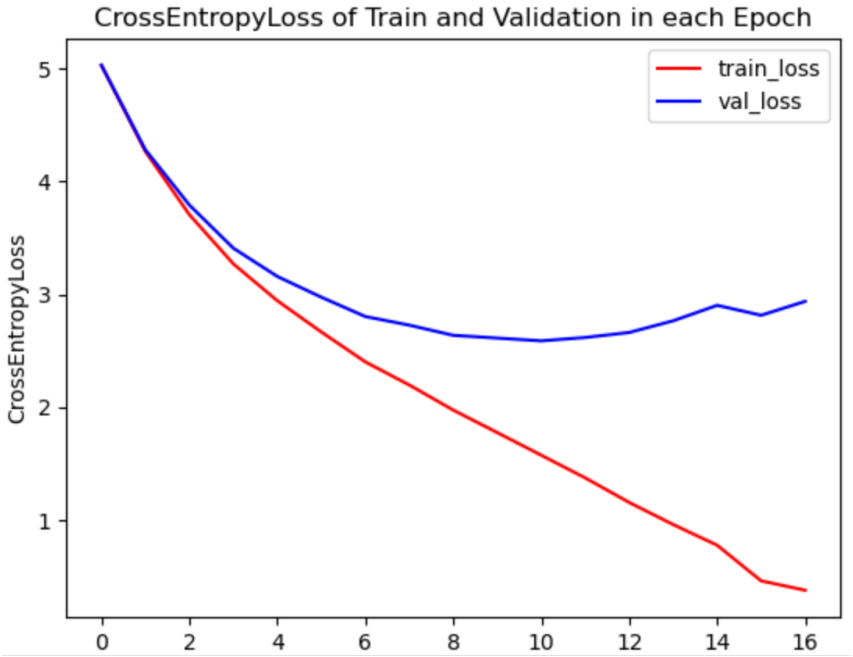


最后一次迭代后训练集损失 1.154265，验证集损失 2.661564。

在此学习率下，曲线不存在波动，也即测试集误差最低点基本是最低点，而再迭代则面临过拟合。

不过，为了避免过拟合，直接将迭代次数设置为 10 次后，实际测试集准确率为 38.1%，而 12 次迭代的测试集准确率为 39.5%。虽然存在一定的偶然因素，不过这意味着，一定的过拟合对 top 1 acc 的结果未必是坏事。

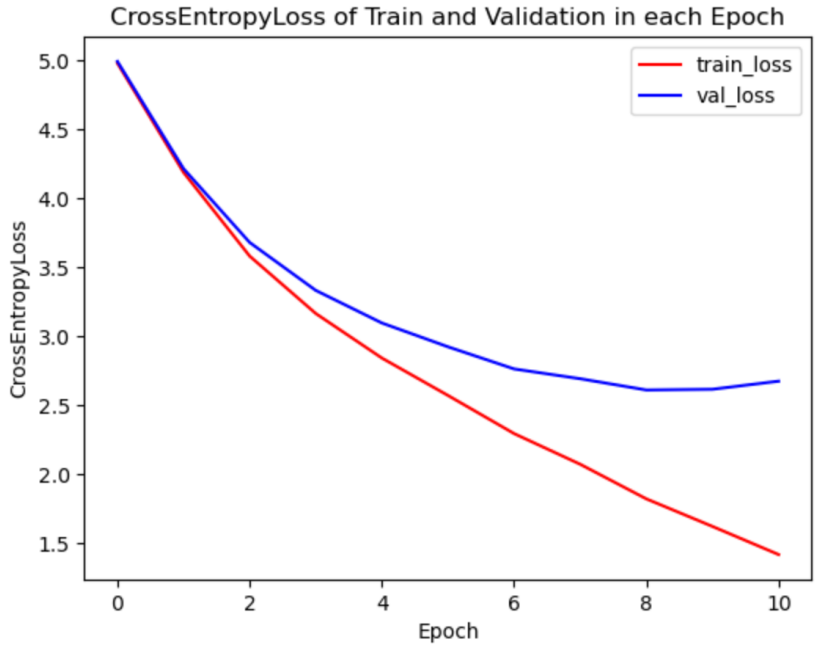
将 wait 设置成 5 的损失曲线如下：



最后一次迭代后训练集损失 0.375982，验证集损失 2.937525，而验证集误差结果仍是 39.5%。

2、学习率与 lrd

将 lr 设置为 0.003，并取消 lrd，wait 设置为 2，效果如下：

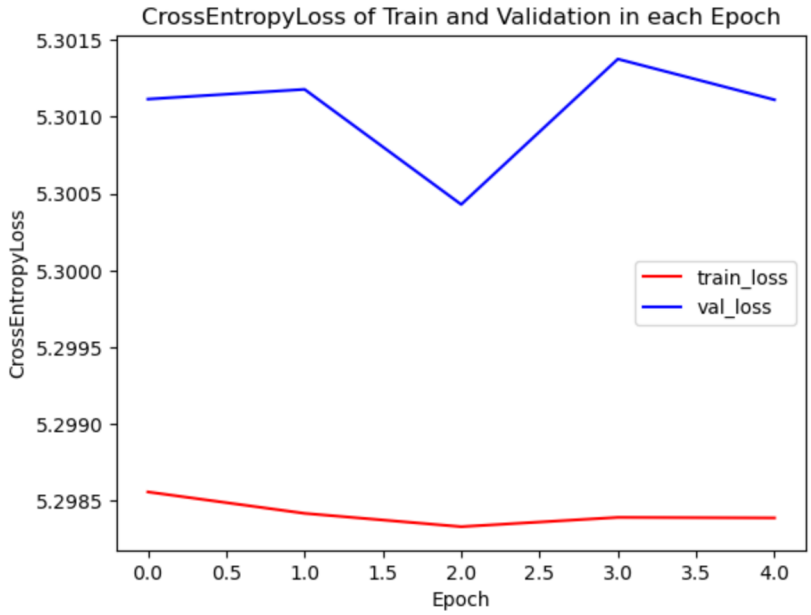


最优的验证集损失为 2.611793，而根据结果可知收敛是在第八次迭代达到的。

事实上，根据后续测试的结果，不取消 lrd 时，lr 对收敛次数的影响并不太大，而 lrd 可以保证收敛的位置更加精确，若其不存在，更大的 lr 很容易导致收敛结果附近的振荡。

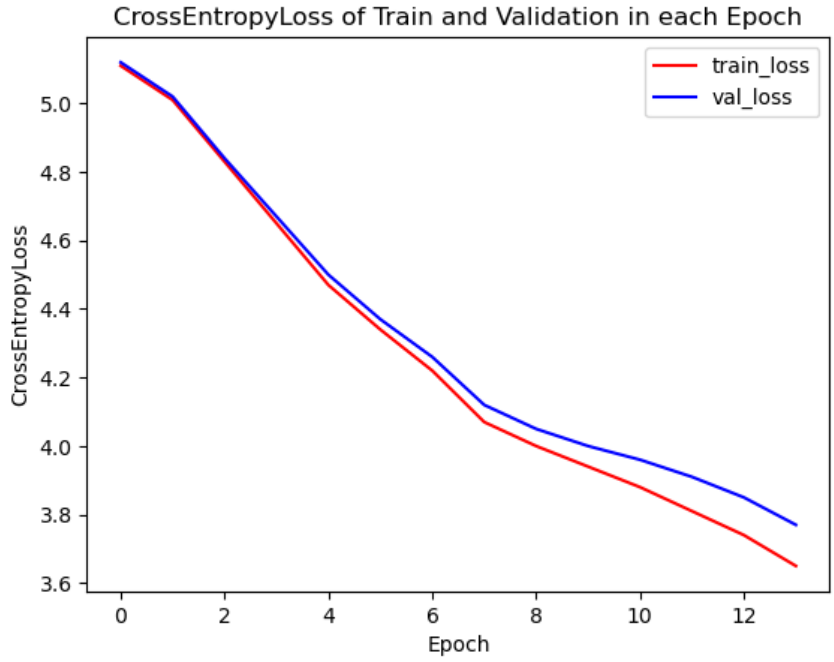
3、正则化

框架网络中，在卷积层与残差网络都提供了正则化与 dropout 的选项。当舍弃所有正则化项后，默认参数的测试结果如下：



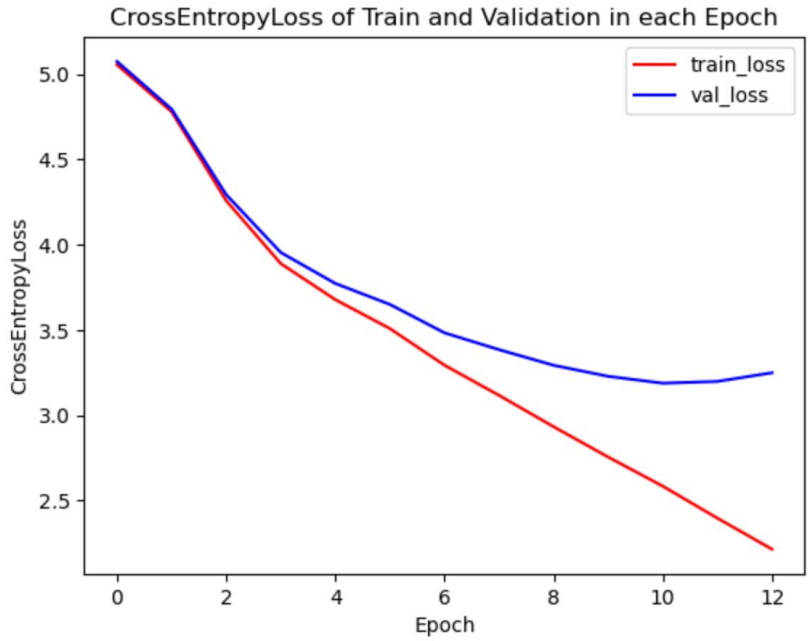
对于大网络，在不添加正则化时，容易发现几乎没有收敛效果。而后续测试也发现，卷积层几乎必须添加正则化来保证正常的验证集误差下降。

4、网络深度



实验发现，当层数很大时，收敛速度会显著减慢，且即使 `lr` 放大很多也无效。例如，上方是在每个卷积层后加两个卷积层后，将 `lr` 设置为 `0.015`，`15` 次迭代的误差曲线。

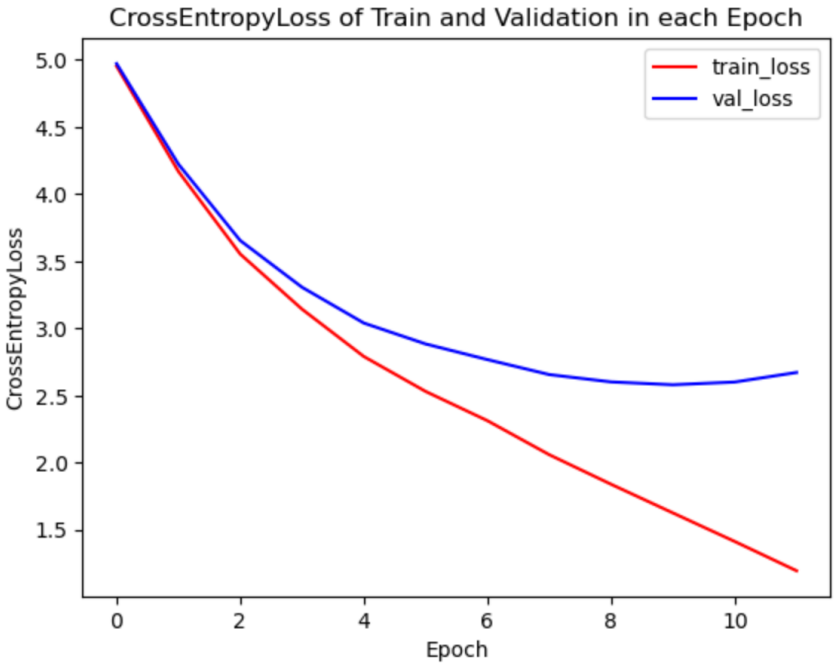
由于收敛速度太慢，减少接近输出的双重卷积层后，误差曲线成为：



可以发现，深度变高更容易收敛于局部极小，因此不宜使用过多的卷积层。框架里的六个部分各一层卷积已经是较好的选择。

结论：

1、最终结果



最低验证集损失 `2.580324`, 测试集 `top1acc` 为 `0.402667`, 各项参数选择见 `main.py`。

2、总结

到底为什么要做这种黑盒实验呢……

最重要的实现部分已经用框架给好了，除了调整结构调整参数以外干不了什么，而且一次要等一个多小时，如果说收获就是累积调参经验的话，好像也不那么符合课上那么多原理性学习的定位(悲)

[或者说，如果是感受各种结构和选择的影响，大概应该选择些更加易于训练的例子，而不是把主要时间放在等待上。]

Lab 3 实验报告

PB20000296 郑滕飞

准备工作：

1、实验要求

利用不同的语言模型进行文本情感分类，并对比效果

2、网络模型

本次实验中尝试了四种不同的网络模型：框架中有的 RNN、LSTM、带预训练的 BERT，与自己实现的 Transformer。

RNN 网络是最基本的循环神经网络结构，每个时刻向后递进，在实验中的定义方式为

```
self.rnn = nn.RNN(input_size=embed_dim, hidden_size=hidden_size,
    num_layers=num_layers, nonlinearity='tanh', batch_first=True)
```

给定输入的词向量维度与隐层的大小、层数进行迭代。

LSTM 在 RNN 网络的基础上增添了处理长期记忆的 LSTM 区块，决定是否保持记忆，在实验中的定义方式为：

```
self.rnn = nn.LSTM(input_size=embed_dim, hidden_size=hidden_size,
    num_layers=num_layers, batch_first=True)
```

Transformer 在实验中由于执行分类任务，没有用到生成的 decoder 部分，只利用 encoder 进行处理，定义方式为：

```
layer = nn.TransformerEncoderLayer(d_model=embed_dim,
    batch_first=True, nhead=8)
self.rnn = nn.TransformerEncoder(encoder_layer=layer,
    num_layers=num_layers)
hidden_size = embed_dim
```

由于编码器仍然是对词向量进行，这里必须指定最后输出的 hidden size 与原本词向量的大小相同。三者之后都需要添加全连接层：

```
self.fc = nn.Linear(hidden_size, num_class)
```

达到最终进行分类的效果。

三者的数据处理是类似的，都需要进行填充使得张量长度一致，然后进行各自的前向传播过程。

BERT 模型采用了预训练的版本：

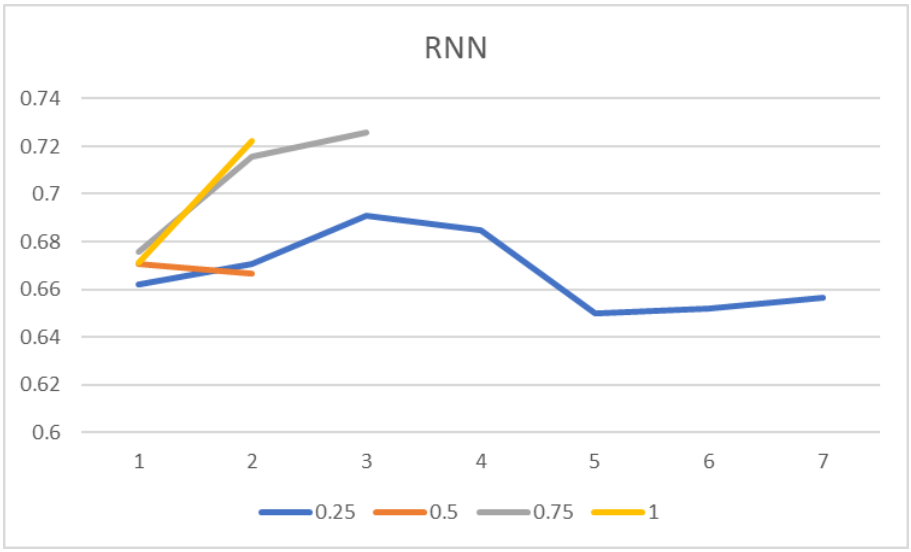
```
net = BertForSequenceClassification.from_pretrained(
        pretrained_bert_model_name,
        num_labels=2,
        output_attentions=False,
        output_hidden_states=False
    )
```

实际的训练事实上是在预训练版本进行微调。

参数测试与讨论：

1、RNN 模型

对不同的 train rate，RNN 网络在达到最大验证集 acc 前的验证集损失图如下(横轴为迭代次数)：



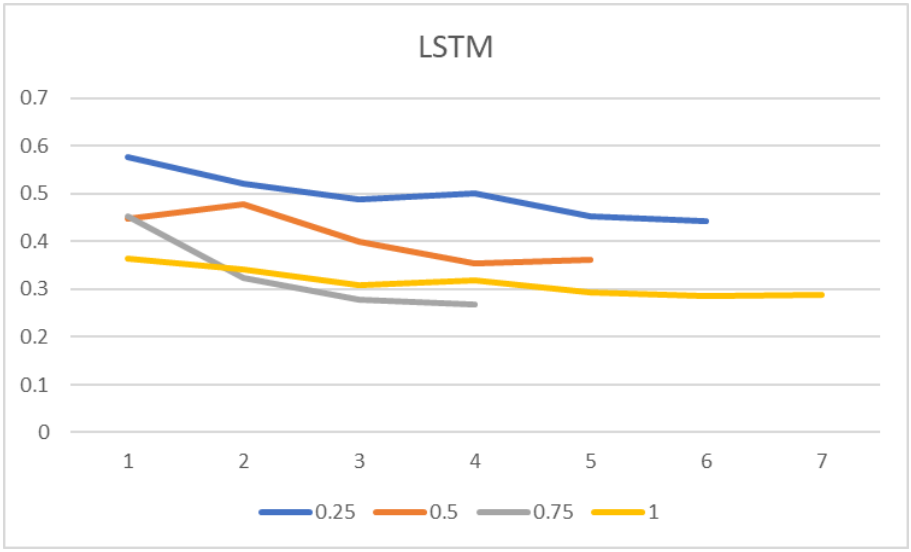
*train rate = 1 时事实上一次迭代就已经达到了最大 acc，不过由于一个点没法画出图像，这里把第二次迭代后的损失也画了上去

总体来看，哪怕是只用 0.25 的训练集，验证集损失在迭代过程中也很快就达到了上升。不过，在训练比例更低时，总体来说能迭代得更加稳定，结果也稍好(具体结果将在“结果展示”部分说明)。

但是，RNN 哪怕最好的验证集损失也在 0.64 以上，验证准确率也未超过 62%，总体来看只是比随机划分好一点。虽然其参数少、训练速度快(不到一分钟)，仍然无法弥补准确率的损失。

2、LSTM 模型

对不同的 train rate，LSTM 网络在达到最大验证集 acc 前的验证集损失图如下(横轴为迭代次数)：



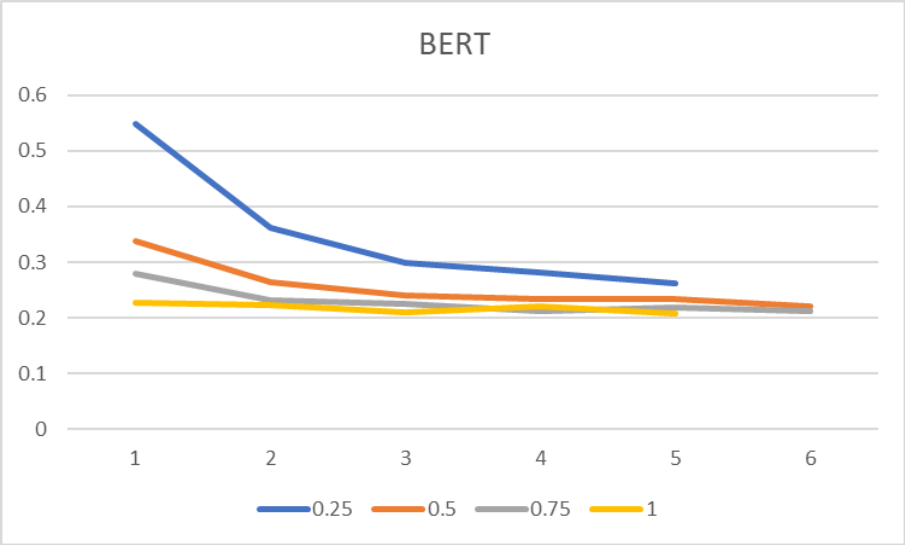
这个结果基本与预期相符。随着 train rate 的增加，一轮迭代改进的程度在逐渐增

大，收敛也更快，收敛结果一般更好。不过，增加到最大时，由于学习率的设定，反而容易变得更加不稳定，减速收敛。

LSTM 的总体结果都在 80%以上，接近 90%，且一轮训练时间基本在一分钟左右，兼具速度与质量。比起 RNN，结果改进了非常多。

3、BERT 模型

对不同的 train rate，BERT 模型在达到最大验证集 acc 前的验证集损失图如下(横轴为迭代次数)：



可以看出，train rate 对迭代次数的改变其实并不多，不过显著影响了迭代后的损失，每次的损失基本随着 train rate 单调降低。由于 BERT 采用了现成模型，当输入数据更全面时，微调也更加准确；且在现成模型的基础下，并不太容易落入奇怪的局部最优情况，结果较为稳定。

BERT 模型的所有结果都在 90%以上，且往往第一次迭代就能取得不错的结果，但遗憾的是，它的训练时间极长。当 train rate 为 1 时，一次迭代需要消耗 47 分钟的时间，几乎无法接受。 考虑到现实情况，LSTM 的总体效益更高。

4、Transformer 模型

这里先给出 Transformer 的 forwarding 代码：

```
l = offsets.size()[0]
la = embeds.size()[1]
mask = torch.ones((l, la), dtype=torch.bool).to("cuda:0")
for i in range(0,l):
   mask[i,0:offsets[i]] = False
x = self.rnn(embeds, src_key_padding_mask=mask)
x = self.fc(x[:,0,:])
# re = random.randrange(0, embeds.size()[1])
# x = self.rnn(embeds[:, re, :])
# x = self.fc(x)
```

未注释掉的部分是真正的前馈代码。在词向量处理后，通过每个句子的实际长度生成 mask，并且通过 encoder 编码后的第一个分量进行分类、对应优化。不过，起初这个代码

在实际运行时，直接报出了显存不足：

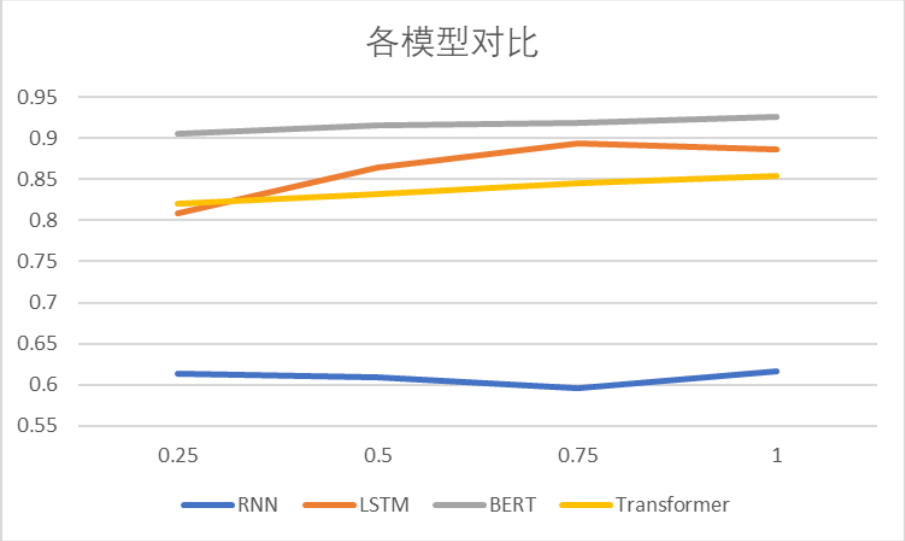
```
OutOfMemoryError: CUDA out of memory. Tried to allocate 1.76 GiB (GPU 0; 14.76 GiB total capacity; 11.90 GiB already allocated; 441.75 MiB free; 13.28 GiB reserved in total by PyTorch) If reserved memory is >> allocated memory try setting max_split_size_mb to avoid fragmentation. See documentation for Memory Management and PYTORCH_CUDA_ALLOC_CONF
```

因此，当时选择了注释掉的简单方案，从词向量中抽取部分进行训练，实际效果与 train rate 无关，准确率基本在 50%，相当于训练并无成果。不过，将 batch size 调低后，发现注释方案可以正常采用，但由于 forward 的设计，无法正常看到 val acc 的更新，因此直接在下方展示最终结果：

结果展示：

1、模型对比

各模型在各个 train rate 下最终的测试集准确率如下：



可以发现，总体来说微调 BERT 是远好于其他模型的，这是预训练的影响，非预训练情况的 BERT 行为应基本与 transformer 类似，因为本质就是 TransformerEncoder 模型。不过，考虑时间成本的情况下，效果稳定最好的还是 LSTM。在后续测试中，其基本可以稳定达到 87.5%以上的准确率，比普通的 RNN 与 Transformer 效果都要好(当然，不排除是参数合适程度的影响)。

2、总结

挺有意思的 Lab，尤其是自己写 transformer 分类研究 mask 设置等效果的时候。不过跑 BERT 的时间着实很吓人，四个合起来跑了近 20h，因为其中还有过一些输出无法正常显示的情况。

Lab 4 实验报告

PB20000296 郑滕飞

准备工作：

1、实验要求

利用图卷积神经网络(GCN)对不同数据集完成结点分类与链路预测任务，并分析各种参数对模型与预测性能的影响。

2、网络模型

助教提供的 demo 中，网络是由深度为 num_layers，宽度为 hidden_features 的一系列图卷积层构成的，此外引入了可选的 pair norm 与 drop edge 正则化。

参数分析：

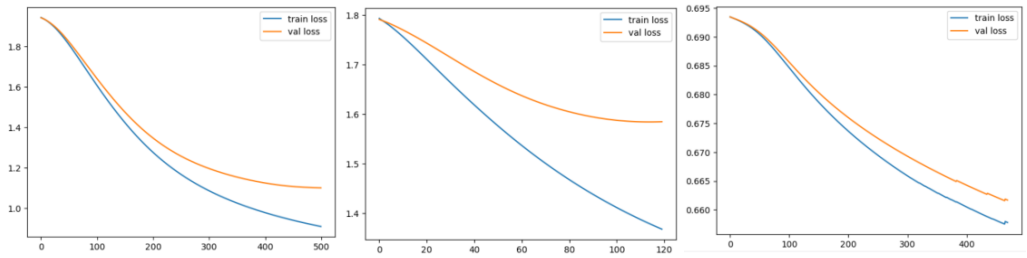
1、正则化超参数

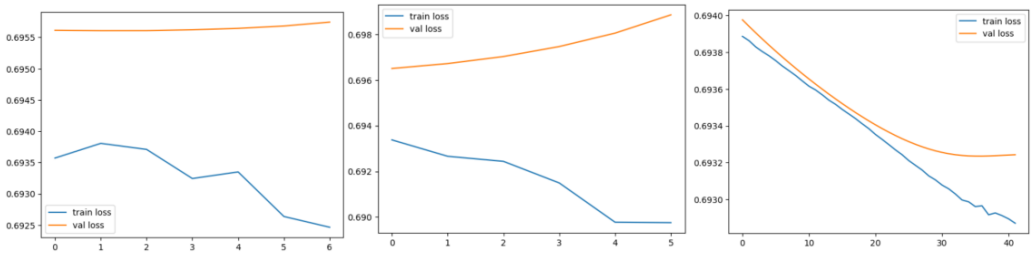
实验中，自环、pair norm 与 drop edge 可以起到相当于正则化的作用，而实际上，在其他参数为默认(num_layers = 2, act_fn = 'prelu')时，改变此三个参数的结果为（下方 T 与 F 为这三个参数依次取 True 或 False，N 与 L 表示结点分类与链路预测，表格中数值代表 best val score，绿色、黄色为每列最高、次高）：

参数	cora-N	cite-N	ppi-N	cora-L	cite-L	ppi-L
FFF	0.6790	0.4381	0.0135	0.4750	0.4872	0.4578
FFT	0.5627	0.3837	0.0064	0.4624	0.5110	0.5458
FTF	0.6328	0.4290	0.0121	0.5688	0.5446	0.6448
FTT	0.5846	0.3792	0.0042	0.5689	0.5356	0.6222
TFF	0.8395	0.7341	0.0250	0.6637	0.7502	0.3592
TFT	0.7878	0.7553	0.0223	0.5604	0.7837	0.3788
TTF	0.7878	0.7054	0.0042	0.6082	0.7422	0.7241
TTT	0.7860	0.7190	0.0001	0.6071	0.7580	0.7333

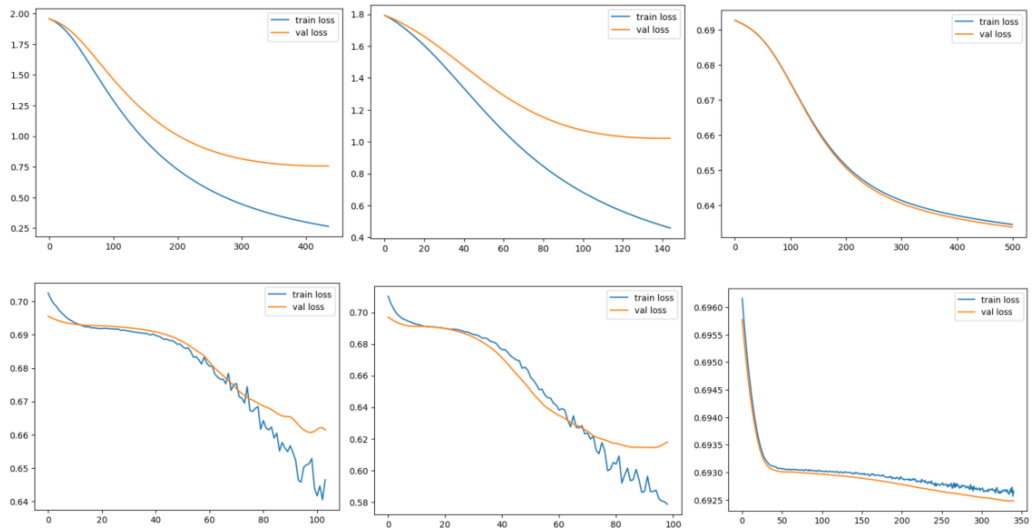
首先，总体来看，添加自环对效果的提升是显著的，它提升了稳定性，避免了多层叠加引起的奇异情况梯度爆炸。

如下为 FFF 时六条损失曲线，可以看出，对 cora 与 cite 的链路预测，在不添加自环时验证集误差从没有下降过，几乎没有效果，哪怕对 ppi，也很快由于不稳定而结束了迭代。而对结点分类问题，训练集的误差很快趋于平缓并收敛。对 ppi 的误差还出现了一些鲜明的不稳定波动。



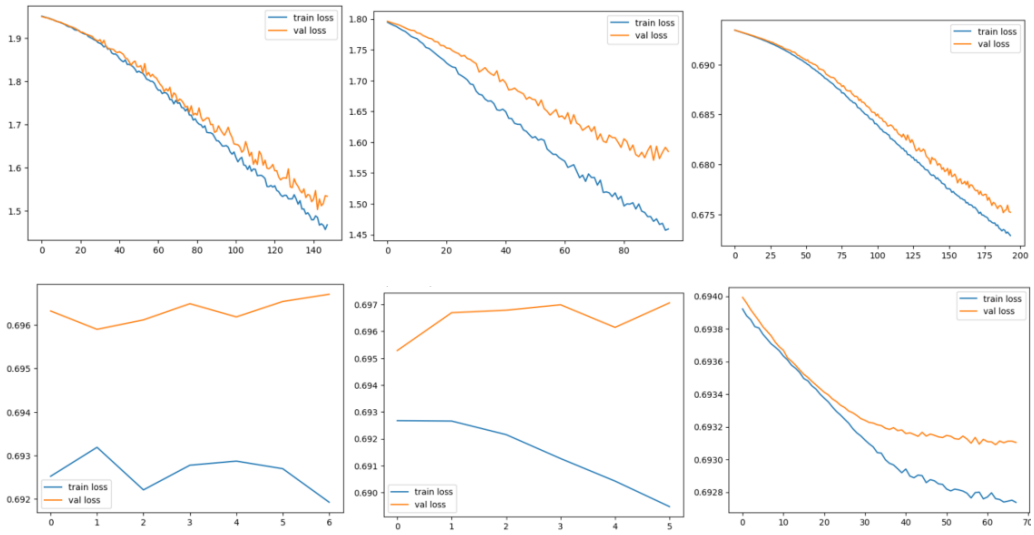


而添加自环后，参数 TFF 时图像如下：



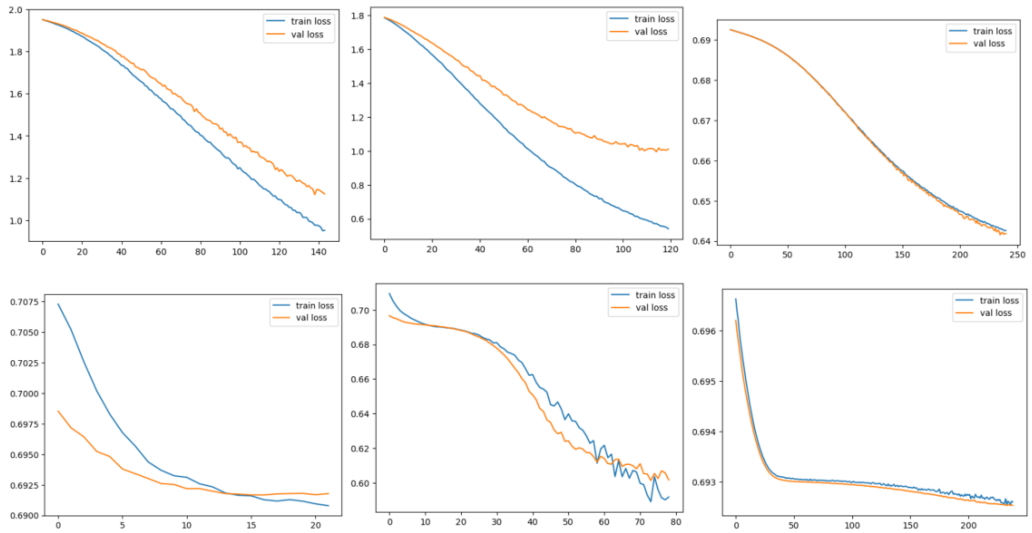
对链路预测问题，添加自环显著避免了迭代过程的过平滑，因此能进行更充分的迭代，而对结点分类问题，则加快了损失减小的速度（注意坐标轴尺度与之前不同）。

其次，对于 drop edge 参数，在不添加自环时基本会让结果变差，而添加自环后则可能起到效果。直观来说，这是因为删边在边较多时才能起到好的正则化作用。参数 FFT 时图像如下：



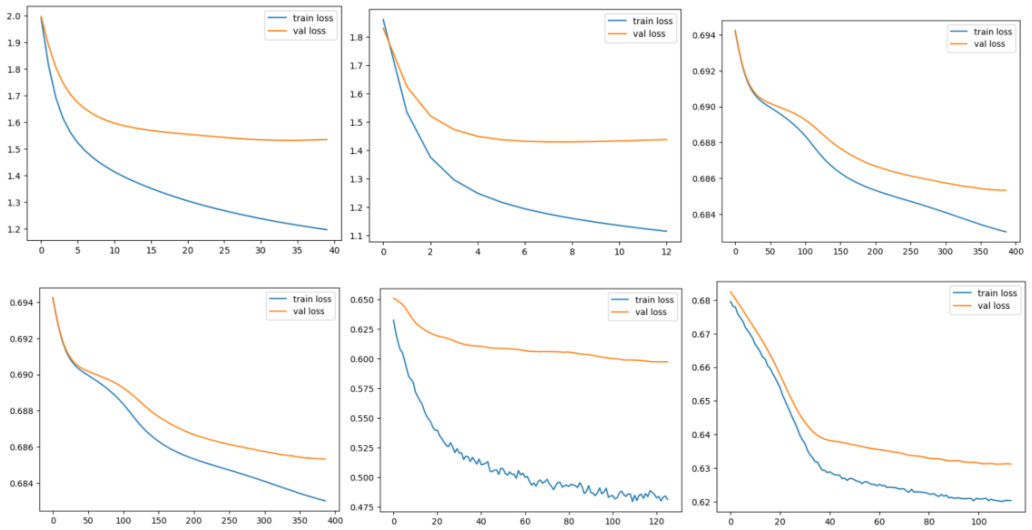
由于 drop edge 的正则化方式过于激进，其导致验证集误差与测试集误差一起波动很大，难以充分收敛。

引入自环后，参数 TFT 时则为：



由于自环带来的边数增多，删边时测试集的波动不再像上方例子一样大，因此可以起到正常的正则化效果，一定程度改进结果。

最后是关于 pair norm 参数。从结果可以发现，此参数对结点分类效果不好，但对 ppi 链路预测有着明显的效果。如下为参数 TTF 的情况：



链路预测中，其同样避免了迭代过程的过光滑，但不像 drop edge 那样激进。由于 ppi 涉及的结点较多，保证特征总距离不变的效果较好。‘

2、激活函数

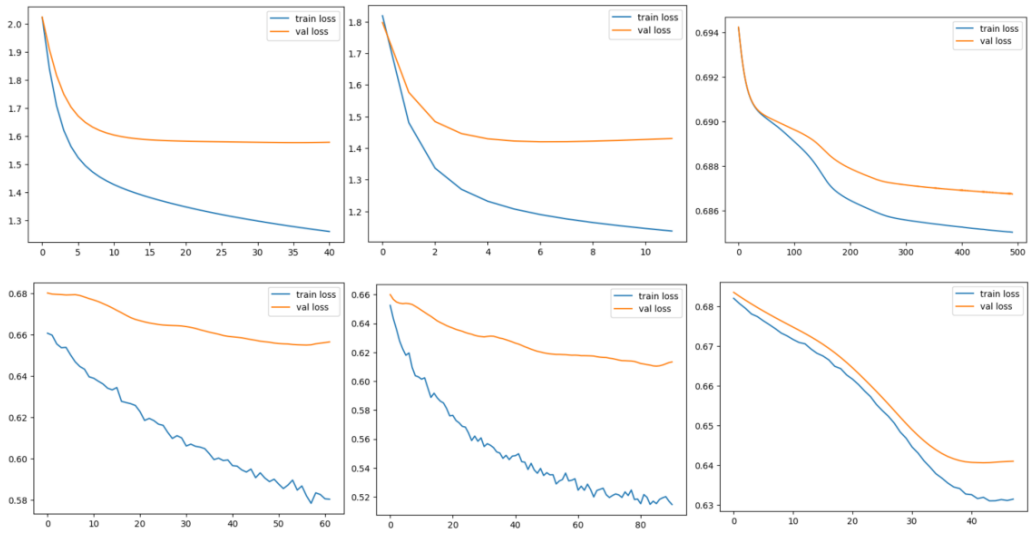
框架代码中提供了 prelu 与 tanh 作为激活函数的选择，对 tanh 激活函数，同样作出之前的表格：

参数	cora-N	cite-N	ppi-N	cora-L	cite-L	ppi-L
FFF	0.6439	0.4199	0.0144	0.4688	0.4918	0.4636
FFT	0.4225	0.3625	0.0059	0.4921	0.5032	0.5271
FTF	0.5775	0.3852	0.0111	0.6114	0.5916	0.6430
FTT	0.5314	0.3988	0.0008	0.5531	0.5434	0.6293
TFF	0.8764	0.7492	0.0266	0.7045	0.7940	0.3561

TFT	0.8118	0.7674	0.0233	0.7389	0.7738	0.3799
TTF	0.7786	0.7236	0.0036	0.6813	0.7340	0.7271
TTT	0.7860	0.7251	0.0000	0.6341	0.7023	0.7327

从结果上来看，pair norm 对 tanh 激活函数结果的正面影响更少，除了 ppi 链路预测外几乎没有正面效果。而在不需要 pair norm 时，tanh 作为激活函数的结果普遍比 prelu 更好，因此，除了在 ppi 链路预测选择 prelu 外，其他均选择 tanh，并且按照 tanh 时最优超参数进行选择。

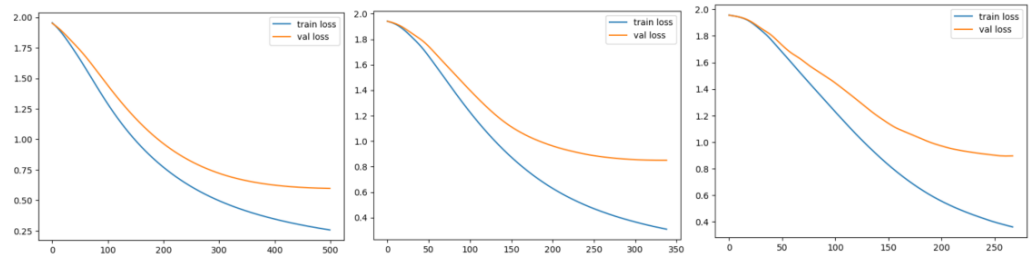
观察 tanh 在参数 TTF 时的效果：



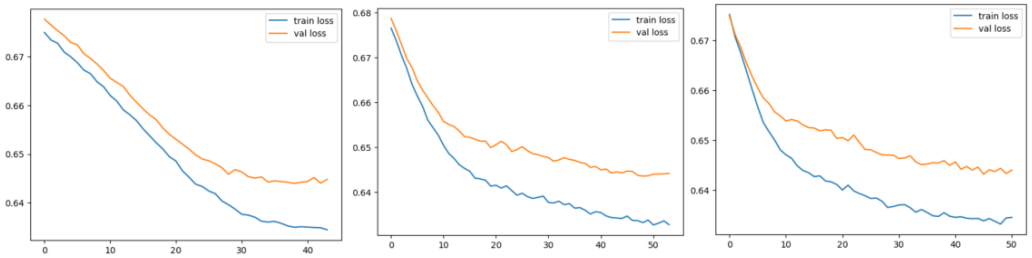
比起 prelu，tanh 的波动更加剧烈，不过也做到了更优的避免过光滑。

3、层数影响

对于 cora 这样的简单例子，层数增加只会引起结果变差，上方最优参数选择后，层数 2 到 4 的结点分类损失曲线如下：

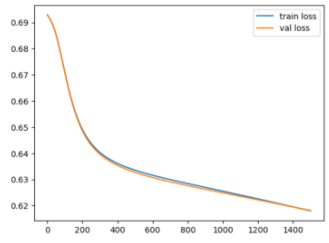
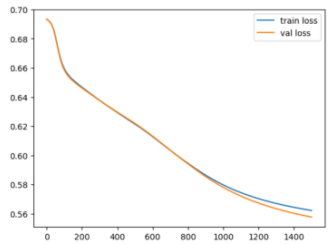
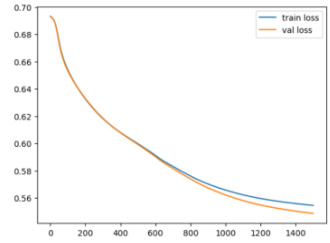
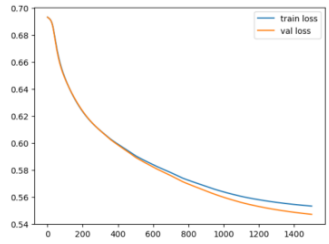
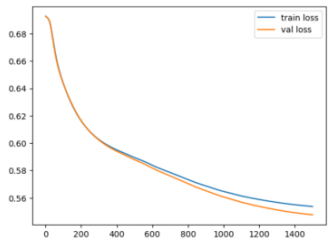
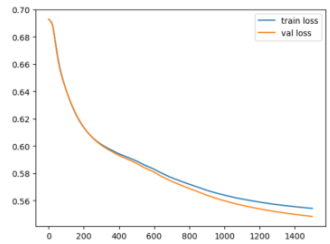
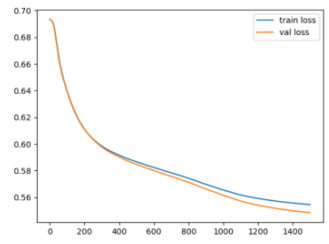
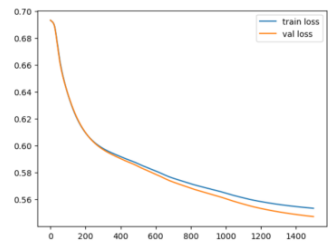
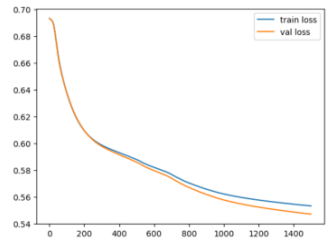


准确率分别为 0.8579、0.8450、0.8266。对 cite 结点分类也是同理，层数高时更容易陷入不准确的局部最优，2 到 4 层准确率分别为 0.7462、0.7100、0.7160。cora 与 cite 的链路预测结果类似。对 ppi 的链路预测，虽然数据集较大，但提升层数效果并不好：



如上为 2 到 4 层时损失曲线，准确率分别为 0.7329、0.7253、0.7215。

不过，最复杂的、之前效果最差的例子，ppi 的结点分类，则是在层数增加时效果显著提升，由于还未收敛，将最大迭代次数提升为 1500，结果列表如下：

num_layers = 2	num_layers = 3	num_layers = 4
val_score = 0.0445	val_score = 0.1695	val_score = 0.1970
		
num_layers = 5	num_layers = 6	num_layers = 7
val_score = 0.2029	val_score = 0.2009	val_score = 0.2028
		
num_layers = 8	num_layers = 9	num_layers = 10
val_score = 0.1975	val_score = 0.2031	val_score = 0.2080
		

可以发现，由于数据集较为复杂，层数在 5 到 7 层时的收敛速度较快，且时间性能相对较好，因此选取 7 层，并将迭代次数提升至充分大（对其他五个任务也提升迭代次数至能够收敛）。

结论：

1、最终结果

具体参数选取见上一部分。最终结果分别为：

cora-N:

best_val_loss: 0.5855, best_val_score: 0.8819

test_score 0.8561

cite-N

best_val_loss: 0.9549, best_val_score: 0.7795

test_score 0.7768

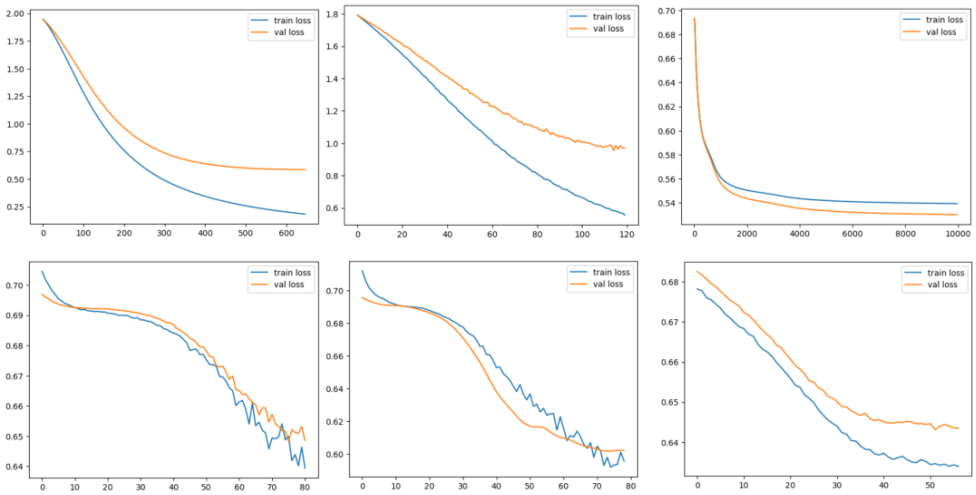
ppi-N:
best_val_loss: 0.5300, best_val_score: 0.2897
test_score 0.2877

cora-L:
best_val_loss: 0.6483, best_val_score: 0.7054
test_score 0.6908

cite-L:
best_val_loss: 0.6016, best_val_score: 0.7576
test_score 0.7700

ppi-L:
best_val_loss: 0.6431, best_val_score: 0.7322
test_score 0.7317

损失曲线分别为:



2、总结

这次实验的设计比之前有趣了很多，也观察到了更多内容。由于迭代时间较短，可以充分观察各个参数对迭代产生的影响，而且可以发现不同数据集的特征对参数选取造成的差异。

Lab 5 实验报告

PB20000296 郑滕飞

阅读笔记：

1、风格迁移的基本模型

从数学角度来说，风格迁移的含义是：在保证目标图像 T 语义信息完整的同时，尽可能地使其风格接近源图像 S 的效果。因此，如果有函数 s(P)将图片 P 的风格映射到某空间向量，函数 c(P)将图片 P 的语义映射到某空间向量，事实上需要求解优化问题

$$\min_X(\alpha||c(X)-c(T)||^2+\beta||s(X)-s(S)||^2)$$

并将结果 X 作为生成的图像。这里α,β为内容保留与风格迁移的权重系数，此函数记为$L(X,T,S)$。

对于函数 c 与函数 s 的不同选取诞生了不同的算法。例如，如果认为图像的低频成分对应语义信息，高频成分对应风格信息，就可以通过小波变换分解后直接替换来达到风格迁移的效果；如果认为语义信息与边缘密切相关，就需要结合边缘定位进行迁移。

但是，这些传统方法在给出了语义信息的具体表达后，也就意味着注定只能使用低阶的语义信息，难以刻画对图像的直观理解—而这恰恰是深度神经网络所擅长的部分。由于深度神经网络在语义识别方面的良好表现，如果我们利用训练得到的网络生成代表语义信息、风格信息的向量，能够得到更好的结果。

2、VGG 网络

为得到语义信息，论文采用了已经预训练完成的用于目标识别与定位的网络 VGG19。

此网络的整体架构如下[此处命名与代码中一致，方便后续说明]：

```
conv_1 relu_1 conv_2 relu_2
pool_3
conv_3 relu_3 conv_4 relu_4
pool_5
conv_5 relu_5 conv_6 relu_6 conv_7 relu_7 conv_8 relu_8
pool_9
conv_9 relu_9 conv_10 relu_10 conv_11 relu_11 conv_12 relu_12
pool_13
conv_13 relu_13 conv_14 relu_14 conv_15 relu_15 conv_16 relu_16
pool_17
fc_17 fc_18 fc_19
softmax
```

其中 conv 为 3×3 卷积层，pool 为 2×2 池化层，fc 为全连接层，最后一个全连接层后通过 softmax 输出结果。

论文中对标准的 VGG19 网络做了三项调整：首先，论文认为无论是语义还是风格特征都足以通过卷积层的输出确定，因此只保留卷积层的计算结果即可，可以舍弃计算复杂度很高的全连接层；其次，论文将所有的输出进行了归一化以方便在后续控制内容保留与风格迁移的权重，这相当于对所有结果进行了统一的线性变换，而 relu 与池化会保留这样全局的线性变换，因此不会影响输出；最后，图像合成任务中，平均池化(即划分为 2×2 网格取平均值)的表现会比最大池化效果更好，因此将最大池化替换为平均池化。

3、函数 c 与 s 的表示

对于卷积神经网络，每层的输出事实上是一个矩阵，矩阵的每个位置对应关于原图片的非线性函数。我们记第 l 个卷积层对图像 X 的输出为$F^l(X)$，其大小为$n_l \times m_l$，由于假设卷积层的输出包含了所有特征信息，c 与 s 可以写为

$$c(X) = (a^1 c^1(X), \dots, a^l c^l(X), \dots), s(X) = (b^1 s^1(X), \dots, b^l s^l(X), \dots)$$

这里a^l, b^l为第 l 层的结果在语义或特征刻画中占据的权重。这样，损失函数就有表示

$$L(X,T,S) = \alpha \sum_l ||a^l\, c^l(X) - a^l c^l(T)||^2 + \beta \sum_l ||b^l\, s^l(X) - b^l s^l(S)||^2$$

于是，我们只需要得到c^l, s^l的表示，结合经验调整权重系数，即可得到损失函数的表示。

对内容的刻画较为简单：我们认为$c^l(X) = \texttt{flatten}\left(F^l(X)\right)$，这里 flatten 表示将矩阵展平成向量，这样即有

$$||c^l(X) - c^l(T)||^2 = \sum_{i,j} \left(F_{ij}^l(X) - F_{ij}^l(T)\right)^2$$

此外，取系数$a^l = 2^{-1/2}$，即第 l 层的结果直接对应平方误差

$$\frac{1}{2}\sum_{i,j} \left(F_{ij}^l(X) - F_{ij}^l(T)\right)^2$$

由于卷积层后还要经历 ReLU 层，采用梯度下降迭代时，第 l 层结果对$F_{ij}^l(X)$的梯度当$F_{ij}^l(X) > 0$时方为$F_{ij}^l(X) - F_{ij}^l(T)$，否则为 0。(论文中记号令$F_{ij}^l(T) = P_{ij}^l$)。根据 VGG 网络的作用，越高的 l 代表着越高层的语义信息，但根据越高层次重建出的结果对图像的具体像素的约束则越来越小。

对于风格，我们认为它可以通过**质量矩阵**来刻画，也即定义

$$s^l(X) = \texttt{flatten}\left(G^l(X)\right), G^l(X) = F^l(X)\left(F^l(X)\right)^T$$

上标的 T 表示转置。

而对权重系数b^l，记为$\frac{\sqrt{w_l}}{2n_l m_l}$，单层的风格函数即有表示(论文中记号令$G_{ij}^l(S) = A_{ij}^l$)

$$||b^l s^l(X) - b^l s^l(S)||^2 = \frac{w_l}{4n_l^2 m_l^2} \sum_{ij} \left(G_{ij}^l(X) - G_{ij}^l(S)\right)^2$$

当$F_{ij}^l(X) < 0$时对其导数仍然为 0，否则可计算出其为$\frac{w_l}{n_l^2 m_l^2}\left(\left(F^l(X)\right)^T\left(G^l(X) - G^l(S)\right)\right)_{ji}$。

4、结果生成

由于已经得到了损失函数的表达，从初始的某个X_0开始，对$L(X,T,S)$梯度下降进行迭代即可得到结果。值得注意的是

$$\frac{\partial L}{\partial X} = \sum_{ijl} \frac{\partial L}{\partial F_{ij}^l}\frac{\partial F_{ij}^l}{\partial X}$$

左侧的$\frac{\partial L}{\partial F_{ij}^l}$我们已经通过理论推导计算出了结果，而$\frac{\partial F_{ij}^l}{\partial X}$则依赖每个卷积层的具体表达，需要通过数值计算得到。

代码实现：

[参考 github.com/enomotokenji/pytorch-Neural-Style-Transfer]

1、整体结构

完整的风格迁移过程的代码实现可以分为以下步骤:

a. 加载预训练模型

b. 读取源图像与目标图像，并通过预训练模型计算出需要的 F 与 G

c. 初始化结果图像，构造对应的损失函数

d. 在每次迭代中，对结果图像进行梯度下降，直至达到优化目标

由于 pytorch 中 optimizer 可以进行自动微分，梯度下降的过程并不需要手动完成，只要构造合适的损失层就可以自动传播。因此，代码的核心在于从 VGG 中进行调整并正确增添损失。

2、c 与 s 的定义

虽然在上一部分中我们认为c^l, s^l是针对该卷积层的损失函数，在实际操作时，需要在卷积层后增添一层，这层对应的 loss 为用$F^l(X)$计算出的损失，而输出则直接为输入，不用改变。这样，对所有层的 loss 进行反向传播就相当于对总损失函数进行了一次梯度下降。

对 c，也即 ContentLoss 层，定义如下:

```
    def forward(self, input):
        self.loss = self.criterion(input * self.weight, self.target)
        self.output = input
        return self.output

    def backward(self, retain_graph=True):
        self.loss.backward(retain_graph=retain_graph)
        return self.loss
```

此处 criterion 选取均方误差，而 weight 则可以控制本层的传播强度，也即代表系数的权重。target 则在建立时直接传入$F_{ij}^l(T)$，此后不进行修改。

值得注意的是，由于一次迭代需要对多层 loss 进行传播，需要设置 retain_graph 为 True。

而 s，即 StyleLoss 层，定义为:

```
    def forward(self, input):
        self.output = input.clone()
        self.G = self.gram(input)
        self.G.mul_(self.weight)
        self.loss = self.criterion(self.G, self.target)
        return self.output

    def backward(self, retain_graph=True):
        self.loss.backward(retain_graph=retain_graph)
        return self.loss
```

这里的 gram 即为归一化的质量矩阵 GramMatrix:

```
class GramMatrix(nn.Module):
```

```
    def forward(self, input):
        a, b, c, d = input.size()  # a=batch size(=1)
        # b=number of feature maps
        # (c,d)=dimensions of a f. map (N=c*d)
        features = input.view(a * b, c * d)  # resise F_XL into \hat F_XL
        G = torch.mm(features, features.t())  # compute the gram product
        # we 'normalize' the values of the gram matrix
        # by dividing by the number of element in each feature maps.
        return G.div(a * b * c * d)
```

值得注意的是，这里由于 batch 的存在与特征映射的构建，输入事实上是一个四维张量，需要合并成矩阵形式后再进行运算。

3、模型初始化

先给出初始化部分的代码：

```
    i = 1
    for layer in list(cnn):
        if isinstance(layer, nn.Conv2d):
            name = "conv_" + str(i)
            model.add_module(name, layer)
        elif isinstance(layer, nn.ReLU):
            name = "relu_" + str(i)
            model.add_module(name, layer)
            i += 1
        elif isinstance(layer, nn.MaxPool2d):
            name = "pool_" + str(i)
            model.add_module(name, layer)
        else:
            break

        if name in content_layers:
            # add content loss:
            target = model(content_img).clone()
            content_loss = ContentLoss(target, content_weight)
            model.add_module("content_loss_" + str(i), content_loss)
            content_losses.append(content_loss)

        if name in style_layers:
            # add style loss:
            target_feature = model(style_img).clone()
            target_feature_gram = gram(target_feature)
            style_loss = StyleLoss(target_feature_gram, style_weight)
            model.add_module("style_loss_" + str(i), style_loss)
            style_losses.append(style_loss)
```

这里 cnn 是已经预训练完成的 VGG19 模型，content_layers 与 style_layers 均是一些层的名字。

具体来说，我们复制 VGG19 全连接层前的每层，若这层要用作 ContentLoss 的计算，就额外在这层后插入一个 ContentLoss 层，对 StyleLoss 同理。值得一提的是，由于模型在构建到当前层时恰好能得到当前层的输出，$F_{ij}^{l}(T)$与$G_{ij}^{l}(S)$可以直接在构建 ContentLoss 或 StyleLoss 层前计算并传入。

4、自动微分方法

迭代的主函数如下：

```
s_loss = []
c_loss = []
run = [0]
while run[0] <= num_steps:

    def closure():
        # correct the values of updated input image
        input_param.data.clamp_(0, 1)
        optimizer.zero_grad()
        model(input_param)
        style_score = 0
        content_score = 0
        for sl in style_losses:
            style_score += sl.backward().item()
        for cl in content_losses:
            content_score += cl.backward().item()
        s_loss.append(style_score)
        c_loss.append(content_score)
        run[0] += 1
        return style_score + content_score

    optimizer.step(closure)

plt.plot(s_loss)
plt.plot(c_loss)
plt.plot(np.array(s_loss) + np.array(c_loss))
plt.legend(["style", "content", "total"])
plt.show()
# a last correction...
input_param.data.clamp_(0, 1)
```

由于 optimizer 被初始化为了以 input_param 为参数的 LBFGS 优化器，需要通过闭包的方式进行调用。这里我增添了绘制两类损失与总损失的功能，从而能直接看出损失。

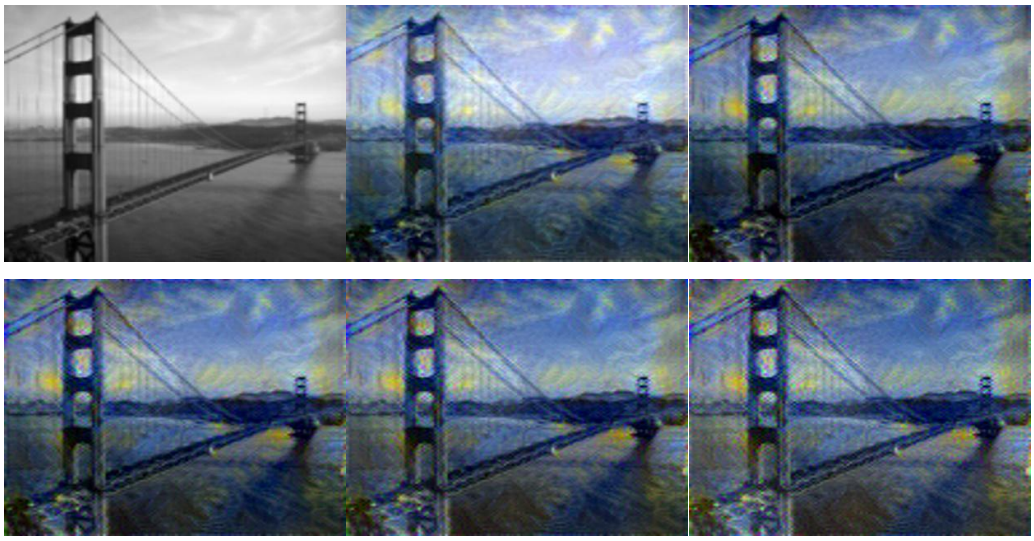
此处，通过每次迭代中对所有损失层进行 backward，optimizer 可以自动对参数进行迭代，省略了复杂的微分过程。

算法调整：

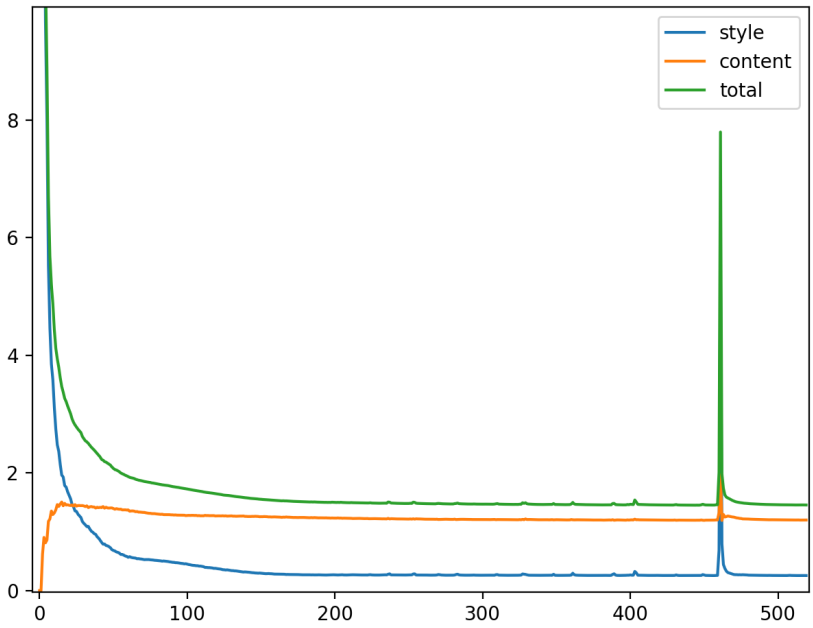
1、基本效果

前一部分中，我对参考代码的修改仅限于结构和输出形式的调整，没有加入任何功能上的改变，因此输出与直接使用应当一致。

基于默认参数 content 权重 1，style 权重 500，content 层为 conv-4，style 层为 conv-1 到 conv-5，迭代 0、100、200、300、400、500 次得到的图片如下：

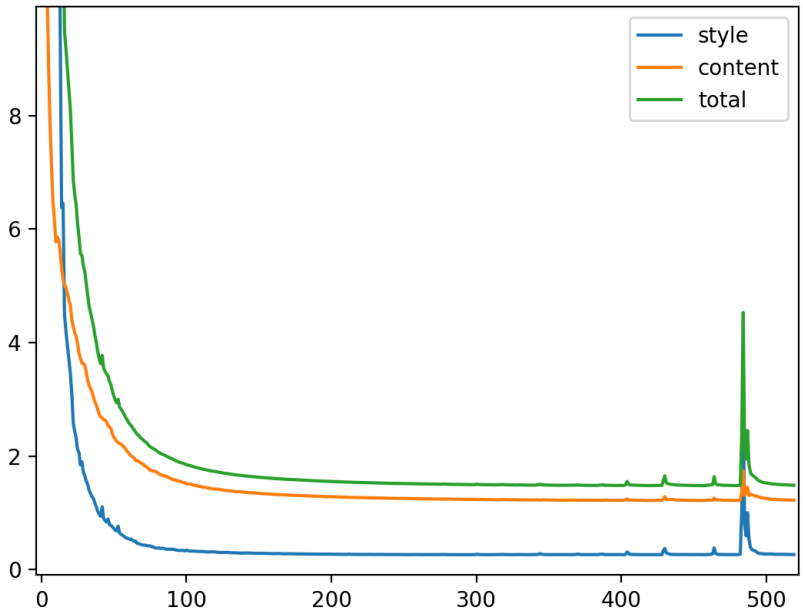


可以看到，以 content 图片灰度化作为起始的迭代中，很快就表现出了风格的特征，而在后续迭代中特征得到逐渐加强。但当迭代次数不断升高时，图片的细节变得更加不稳定。绘制出损失曲线如下：

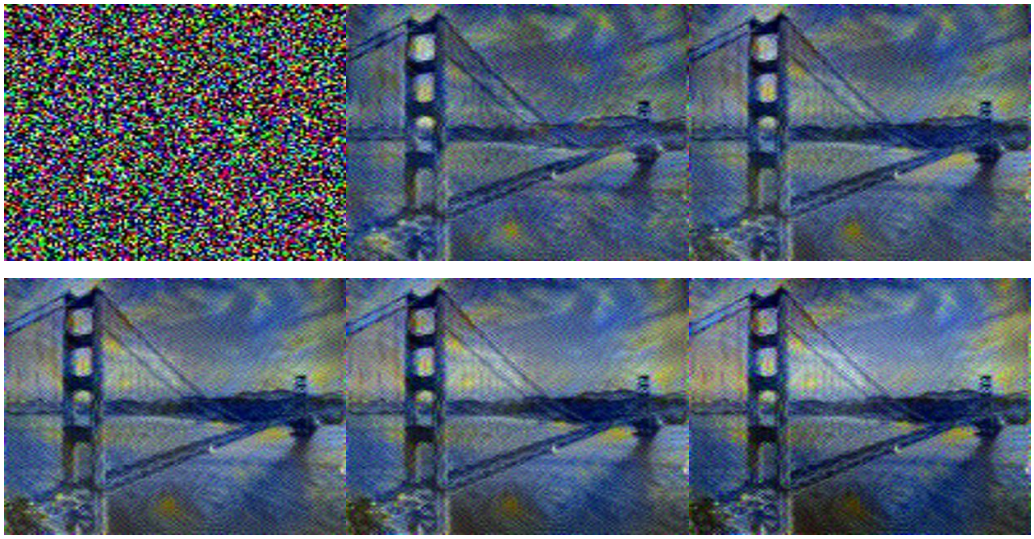


从开始迭代后，style 损失不断降低，而 content 损失则有一定升高，从而总误差很快降低至稳定。值得注意的是，在 400 到 500 次间有一个损失突然提升的过程，这与拟牛顿法的优化器有关，会在后续分析。

而基于同一组参数，另一个选择是以随机生成的白噪声作为初始化。可以想到，这样初始化后，`content` 与 `style` 的损失都将从最高逐渐降低，直到趋于稳定：



用白噪音进行初始化后，在到达最优附近的波动相对更小，直到迭代次数过多时重新开始升高。`0`、`100`、`200`、`300`、`400`、`500` 次迭代得到图片如下：



可以看到，虽然损失函数的性态相对稳定，但实际图像上依然会有白噪音的痕迹。出现原因同样将在下一小节进行分析。

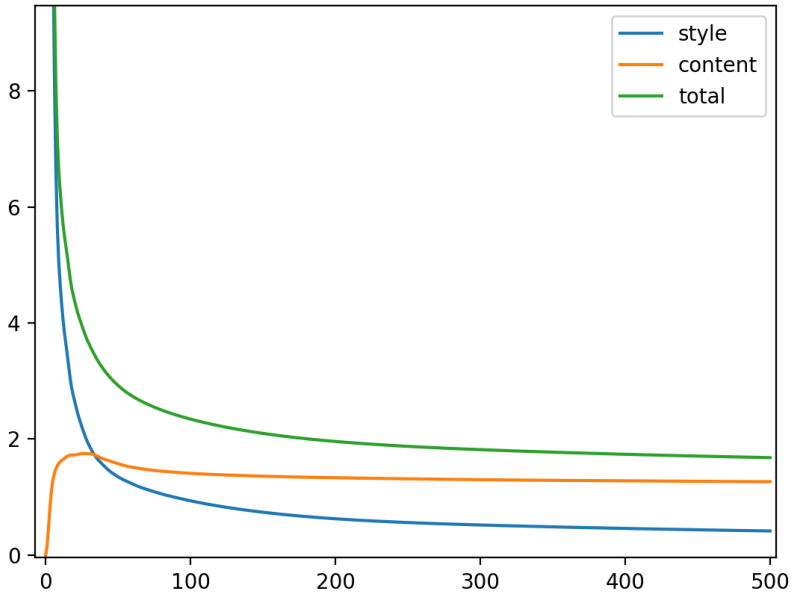
2、优化器与自适应停止

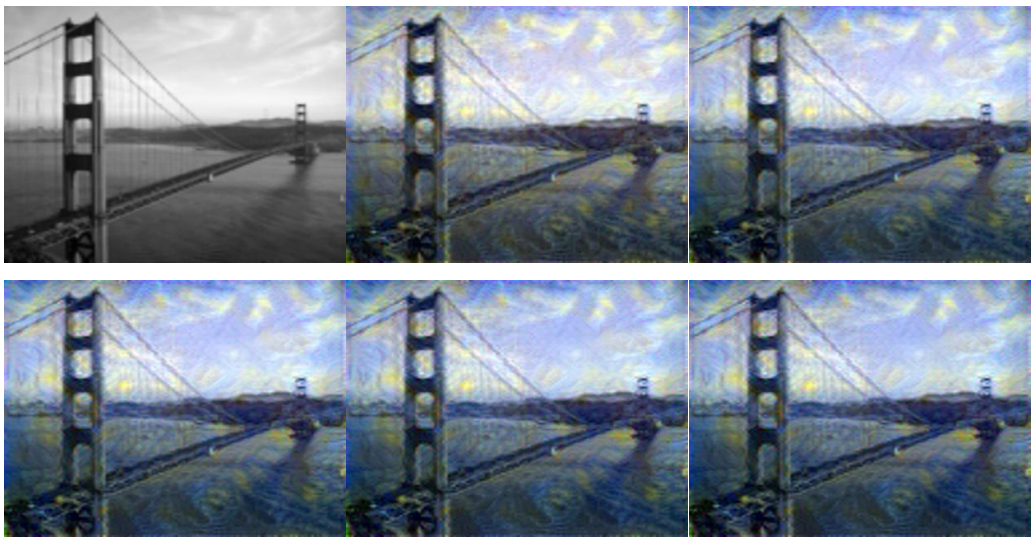
首先，值得注意的是，由于默认参数中 `style` 只用到了 1 到 5 层的 `Gram Matrix`，而 `Content` 的表示则只有第四层信息，在这些信息上的最优化问题不足以完全确定这张图。也即，即使迭代到了理论的最好结果，也并不能完全决定图片。在这样的欠定优化问题中，迭代结果一定会保持某些起始空间的信息，这也就是为什么白噪音作为起始后迭代的结果显著并不光滑。从这个角度来说，由目标图片作为起始是一个更好的选择，因为能在不影响优化

数值的情况下更多保留目标图片的语义特征

其次，参考代码的优化器选择为拟牛顿法 LBFGS 优化器，在这个问题中其实并不合适。由于 ReLU 函数并不具有二阶光滑性，拟牛顿法所模拟的二阶矩阵很可能奇异并累积误差。此外，由于 Gram Matrix 的 MSE 和系数矩阵的 MSE 都是对系数的凸函数，事实上落入局部最优的可能性并不多，不需要相对大幅度的重置（这也是引起 400 到 500 次迭代误差突然上升的原因）。因此，可以直接使用 Adam 优化器。

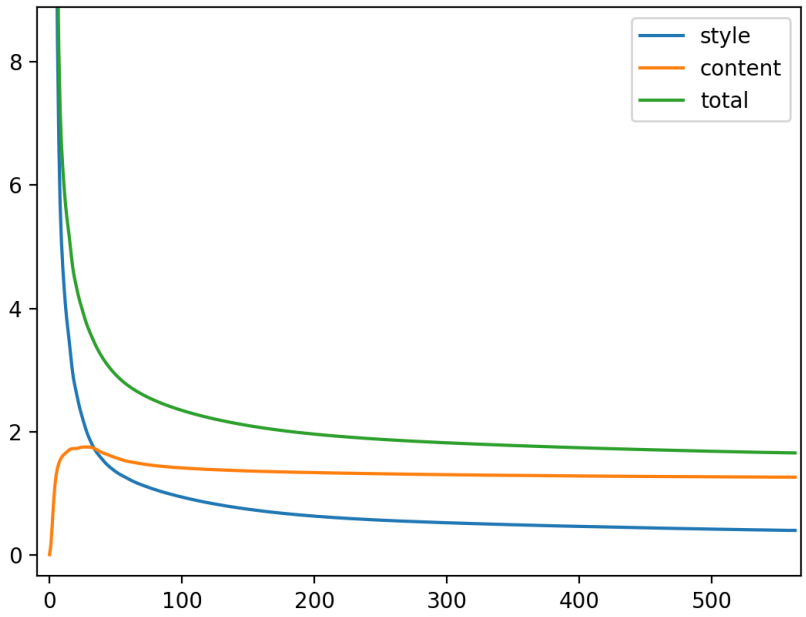
由于无需过于担心落入局部最优，lr 可以相对较大，这是以 1e-2 的学习率进行 500 次迭代的损失曲线与每 100 次的作图：





无论从损失函数还是图片中，都能看出 Adam 优化器的变化更加光滑。不过，从颜色上也可以直观看到，Adam 的收敛速度会比二阶方法慢不少，但在变得更加光滑后，更容易采用自适应停止策略。例如可以选择最好误差连续三次没有更新便停止的策略，采用充分大的迭代次数后，得到的停止次数为 563，最好误差 1.652。

损失曲线如下：

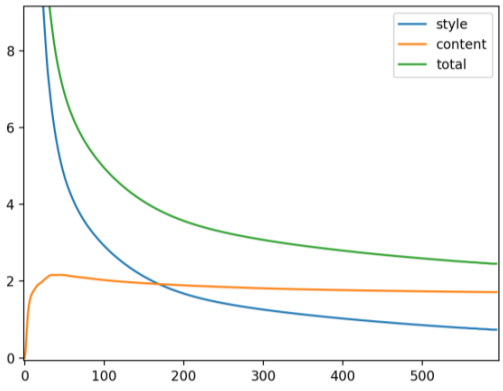


将 Adam 迭代的最终结果、LBFGS 迭代 100 次与 200 次的结果对比：

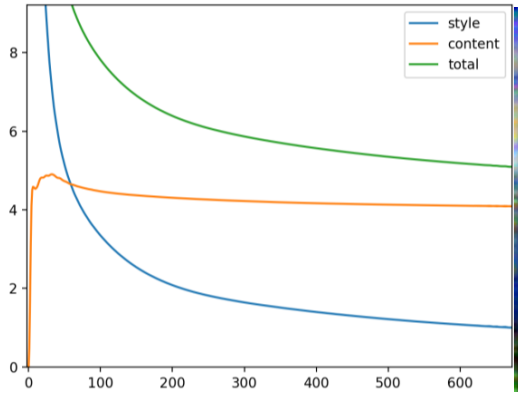
可以发现，Adam 优化出的结果更加倾向于保留 content，因此想形成与 LBFGS 的类似结果需要放大 style 对应的权重。

3、分层加权

在参考代码中，对 Content 与 Style 的权重只能进行整体的叠加，而根据之前 c 与 s 的构建，事实上应该能对各层分别控制。于是，将 content_layers 与 style_layers 设置为字典类型，键代表层，值代表权重，即可分层控制。由此，亦可研究不同层次所带来的影响。例如，为了达到更好的保持内容的效果，在 conv_1 上添加 0.1 的权重，conv_4 仍为 1，而风格的权重改为均 1000，自适应迭代共 593 次，损失曲线与最终结果为：



而若 conv_5 内容权重设置为 1，其他内容权重为 0，风格权重仍均 1000，自适应迭代 673 次后损失曲线与最终结果为：



由于本部分主要讲述对代码的改进以实现允许分层加权，实际上不同层权重设置的影响会在下一部分分析。

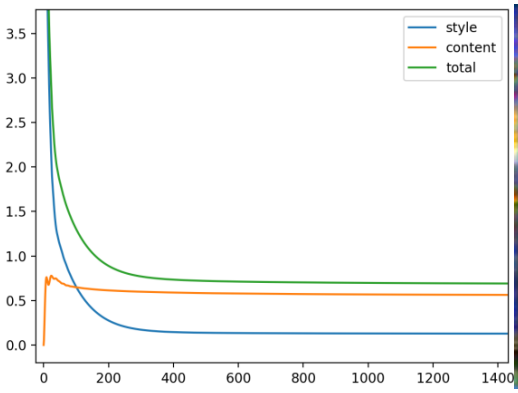
4、平均池化

根据论文的介绍，采用平均池化层会比简单的最大池化有更好的效果，不过，由于 VGG 是通过最大池化训练的，没有平均池化对应的卷积层参数，这里只能直接在构造时修改：

```
elif isinstance(layer, nn.MaxPool2d):
    name = "pool_" + str(i)
    model.add_module(name, nn.AvgPool2d(kernel_size=layer.kernel_size))
```

事实上，打印出 layer 的参数后可以发现，原本的 MaxPool2d 层除 kernel_size 设置为 2，代表 2×2 池化之外，其他参数均为默认，因此这里建立时也无需调整其他参数，直接构造 2×2 平均池化即可。

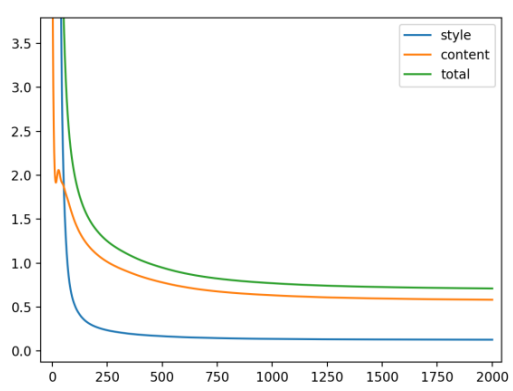
conv_5 内容权重设置为 1，其他内容权重为 0，风格权重仍均 1000，自适应迭代 1427 次后，可以达到 0.69 的最终损失（相比之下，同样参数最大池化的损失有 5.09，推测是由于最大值函数的光滑性远不如平均值，因此大幅影响了优化的效率），损失曲线与最终效果为：



从直观与损失曲线来看，这样的直接替换已经能起到优化结果的作用，也暗示了对图像处理来说，平均池化是更好的保存信息的方式，因此报告此后的结果都是在平均池化下完成的。不过，它导致了更大的迭代次数，由于光滑性，也可以考虑适当增大学习率以降低迭代次数。经实验发现学习率最高设置为 3e-2 较为合适，因此在本小节的对比之后，需要较为精细时采用 1e-2 学习率，否则采用 3e-2。

此外，仍在刚才参数下，考察平均池化对白噪声作为起始的迭代的影响。在学习率设置

为 1e-2 时，直接达到了预设的最大迭代次数 2000 仍未收敛，增加至 3e-2 后，损失曲线与结果为：



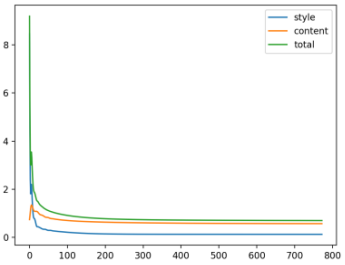
2000 次迭代最终损失 0.71，对比目标图像初始化的 0.69 可以认为基本收敛到了最终结果，不过如同之前的分析，仍然能在图片中明显看出白噪声的成分。

5、初始化的影响

采用白噪声初始化与采用目标图片初始化的对比已经在前文完成了，容易想到，如果能以更低的起始损失进行初始化，可以有效减少迭代的次数，这样仍然可以在低学习率时完成，保证精度。由于本就含有内容与风格两方面的损失，可以想到通过两者线性加权进行初始化，并对比效果。

例如，从以两者平均进行初始化，迭代效果与损失曲线如下：

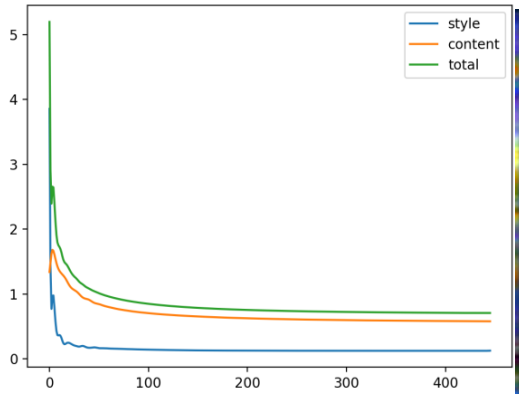


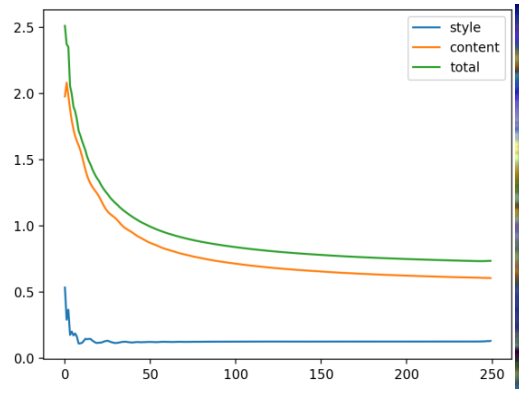



仅需 772 次迭代就收敛到了结果，约为直接以内容初始化的一半。

从损失曲线可以看出，起始风格产生的损失仍然相对较大，因此调整源图片占比为 0.7，

目标图片占比 0.3，得到仅需 400 次迭代。而若按 9:1 的比例加权，收敛次数仅为 249，损失曲线与最终结果分别如下：





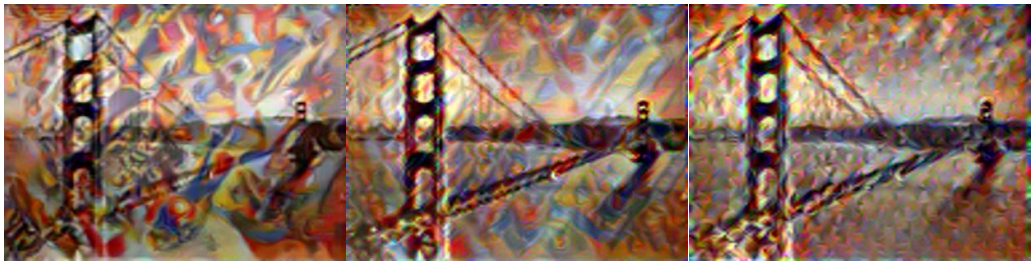
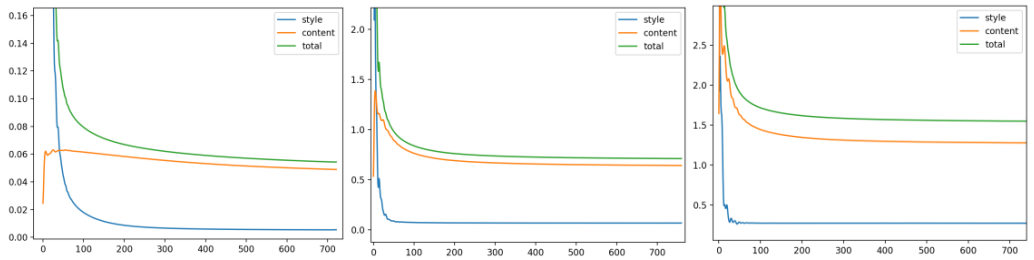
虽然看起来目标图片成分越大可以越快收敛，但这事实上导致了迭代的最终损失较大。目标图片占比 1、0.5、0.3、0.1 时最终损失分别为 0.691、0.699、0.708、0.733，精细程度不断下降。考虑到两者的权衡与对不同图片的适用性，以直接平均作为最终的选择。

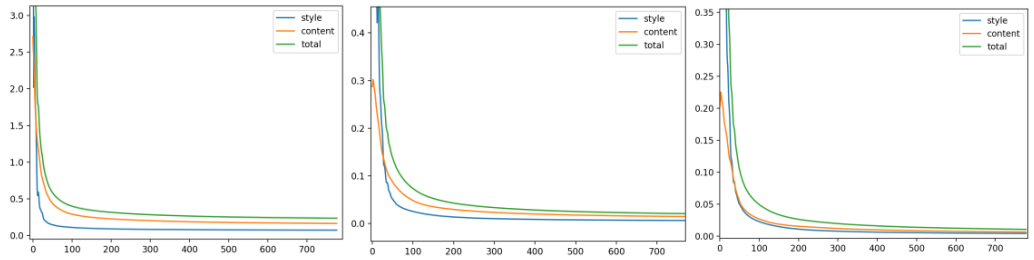
结果展示：

1、内容层次影响

之前的例子是将下方图片 1 风格迁移为图片 2，为了进行更充分的考察，本节将下方图片 1、2 风格迁移为图片 3。

令 1 到 5 层风格权重为 500，首先考察内容权重 1 放置在不同层的影响。以下分别为内容权重在卷积 1、3、5、9、13、16 层（基本为经历了不同池化层数的效果）图片 1 迁移为图片 3 的迭代结果与损失曲线，学习率为 3e-2：

每种情况的迭代次数基本一致，都是在 700 到 800 次，但最终损失差别很大，分别为 0.05、0.71、1.55、0.23、0.02、0.01。当层级较低时，内容权重的影响较大，因此迭代中对内容还原多，最终损失小，而层级较高时内容权重所包含的自由度不多，且与风格分离，容易迭代到误差更小的情况。当内容层数与风格最大层数同为第五层时，该层的耦合严重，因此该层参数产生了极大的损失。

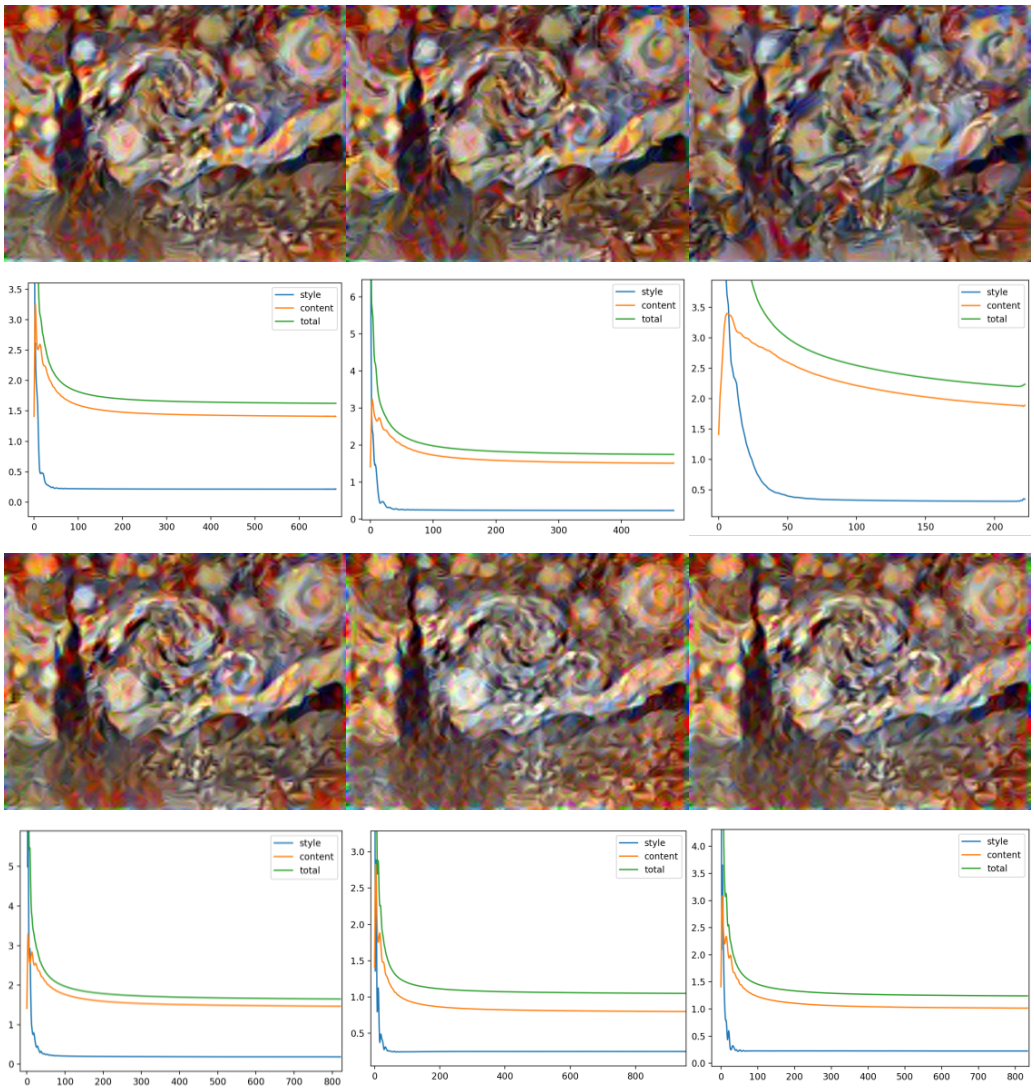
从结果的效果上来看，越高层次的内容对应越高层的语义信息，因此图片的基本信息保持越差。但第一层结果表明，在内容损失层次过低时，有可能难以达到整体的优化，而只有局部的扭曲，因此内容损失的层数应与风格损失的最大层数接近，方才可以保证在同一层进行更彻底的优化。

2、风格层次影响

考虑内容损失恒定在第五层为 1 时图片 2 迁移为图片 3。下方的六个例子中的风格损失依次为：

```
{'conv_2':500, 'conv_3':500, 'conv_4':500, 'conv_5':500, 'conv_6':500}
{'conv_3':500, 'conv_4':500, 'conv_5':500, 'conv_6':500, 'conv_7':500}
{'conv_4':500, 'conv_5':500, 'conv_6':500, 'conv_7':500, 'conv_8':500}
```

```
{'conv_1':200, 'conv_2':400, 'conv_3':600, 'conv_4':800, 'conv_5':1000}
{'conv_1':1000, 'conv_2':800, 'conv_3':600, 'conv_4':400, 'conv_5':200}
{'conv_1':300, 'conv_2':600, 'conv_3':900, 'conv_4':600, 'conv_5':300}
```



由于风格是通过质量矩阵判断，而质量矩阵事实上忽略了位置信息，直观上看，对风格的调整更加导致整体色彩的变化。六次的运行次数与损失分别为：

```
runs: 683 best score: 1.6245089753065258
runs: 483 best score: 1.747919482178986
runs: 222 best score: 2.199137344956398
runs: 821 best score: 1.6490540759987198
runs: 952 best score: 1.0495783939259127
runs: 833 best score: 1.2421233381901402
```

在将风格损失层后移的过程中，与内容损失所在的 conv_5 层耦合逐渐增多，该层的误差越来越大，也越来越难以保持内容的一致。这与更改内容损失层的结果一致，仍然意味着内容损失层应在风格损失层的最大层数附近。

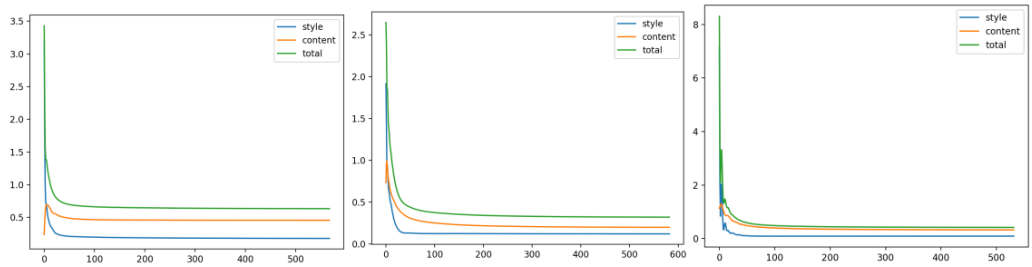
而在这一前提下，对权重的调整对结果的影响相对较小。总体来说，风格损失的中心越是早于内容损失层（如第五张图中心靠前），对内容的还原就越好，而越是迟于内容损失层

（如第四张图中心靠后），就越会向源图片扭曲。第六张图介于两者中间，结果也介于两者中间。由此，为使结果更好，在内容损失层接近风格损失的最大层外，可以将风格损失的中心一定程度前移。但若移到过前，又会发生上一小节所述的独立优化导致难以保持内容的情况。

3、图片影响

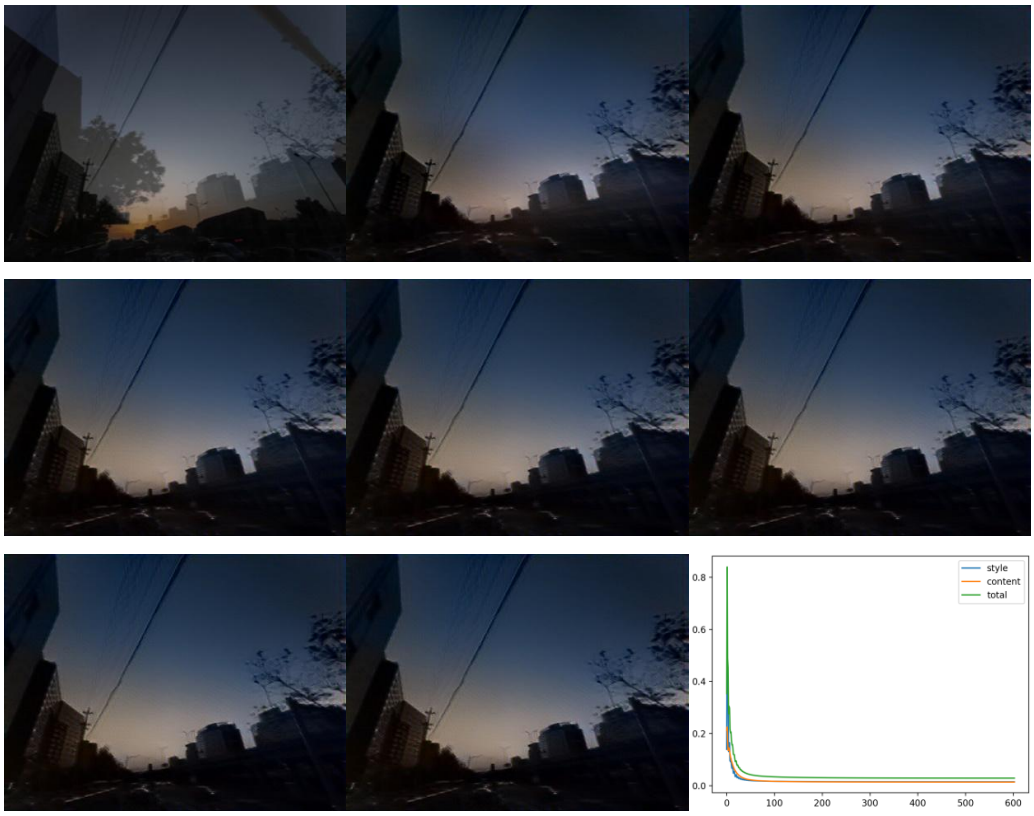
本小节的参数为，内容损失在 conv_5 为 1，其他为 0，风格损失在前五层依次为 1000、800、600、400、200，其他为 0。刚才的实验中，迁移到的图片都是具有明显风格化的，因此能看到十分明显的颜色特征，但是，当选取的图片较为一般（如平时照片时），还需要考察风格迁移的具体结果。本节采用的图片 4、5、6 如下：

由于照片本身像素较多，之前统一为 128×128 会造成大量信息损失，因此统一压缩为 256×256，并取充分大的最大迭代次数。以下是 4->5、5->6、6->4 的结果与损失。

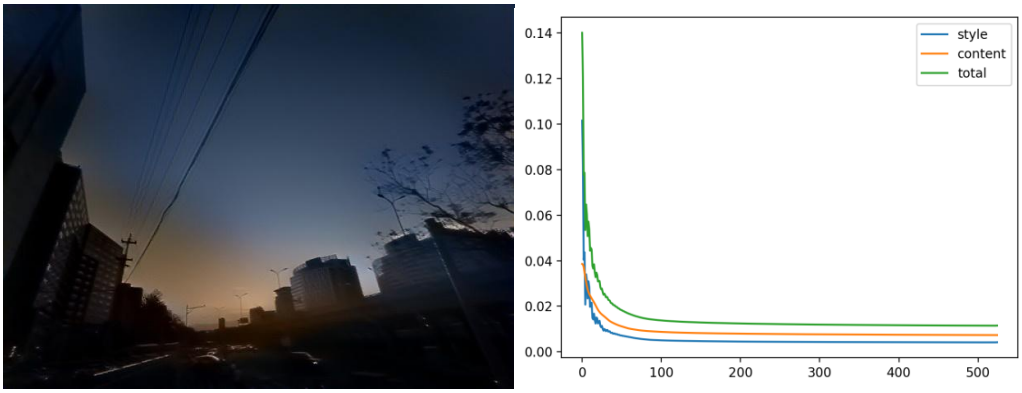


结果中可以看出，不同图片对误差和结果的影响是较大的，但总体来说，颜色风格能完成较好的融合。由于初始化的原因，事实上颜色融合也会包含一定的位置信息。而反过来，由于我们的目标图片是进行了灰度化的，如果对相似内容的图片进行风格迁移，事实上可以起到对灰度图重填色的效果。例如，我们对下面的图片 7 迁移到图片 8（图片 9 为原图）：

0、100、200、300、400、500、600 次迭代后与最终迭代结果及损失曲线为：



可以看出，迭代后基本还原了源图片的颜色。若将风格权重设置得更高，还可以得到更好的还原。减半内容权重后，压缩至 512×512，取学习率 1e-2 充分迭代，最终得到：



也就是说，如果能找到合适的匹配目标，通过匹配系数可以达到的灰度图重填色的效果，这是基于深度学习完成填色的一个重要思路。

总结：

最后一个 Lab 做完了，好耶！（由于自己在写一个灰度图重填色的课题，这篇论文给了我很大的启发，所以就多考察了一些细节）

比赛也已经做完等写报告了，这门课也算是基本结束了，感谢助教老师一学期的付出。