神经网络第一次作业报告

苟琪 MG21370009

January 13, 2022

1 作业内容

本次神经网络作业是用多层感知机实现MNIST手写数字分类，MNIST是一个很经典的数据集，也是很多深度学习入门数据集，其本身不是很难，因此本次作业的主要任务在于手写MLP网络和一些其它改进实现。我全程没有使用框架，仅仅使用numpy库完成了一个小型的BP网络。经实验发现，其在MNIST训练集上能达到99%的正确率，因此可以说明该BP网络是能够成功运行的。其次，对于任务要求中的每一项改进，我也分别找了一种方法与baseline进行比较（尽管有些优化是负优化）。在下面的具体实现中，我会按照baseline的实现，各种优化及对比，优缺点，实现代码说明以及小结等顺序进行说明。

2 baseline

对于baseline的实现，我简要挑几个值得说明的细节进行介绍，其余部分在代码里面都有较为完整的注释。

2.1 数据读取及加载

MNIST数据集给出的数据是ubyte文件，因此需要按照它所定义的方式进行读取，即先读取4个int字节，magic, numImages, numRows, numColumns分别代表，magic number，图片数量，图片宽度，图片长度等。MNIST图片大小是28*28的，然后依次读取numImages个28*28的图片像素点即可。

其次，在读取数据集后，我们可以模仿pytorch的写法，来自己实现一个Dataset和Dataloader，其具体实现也较为简单，仅仅需要实现`__getitem__`，`__len__`，`__next__`等方法即可。可参照我的代码。

2.2 模型搭建

此处仅仅搭建较为简单的MLP，即线性层加上激活函数的形式，此处激活函数我选择使用Relu而不是sigmoid，以防止梯度消失等情况。

2.2.1 线性层

对于线性层，其需要实现三个函数，一个是forward函数，此函数用来前向计算线性层的输出值。即 `y = x * w + b`即可。一个是backward函数，此函数用来方向传播时计算梯度，其接受一个下游传回的grad值，然后计算本层的梯度值，对于线性层来说，其梯度也较为简单，w的梯度为$x^T * grad$,b的梯度为$grad^T * \mathbf{1}$(其中**1**为其元素全为1的向量)。最后一个是update函数，用于更新其参数值,即 $w = w - grad_w * lr, b = b - grad_b * lr$。

2.2.2 激活函数

激活函数为RELU，其前向计算为$np.where(x < 0, 0, x)$，即大于0的部分保持不变，小于0的部分怎么变为0，其后向传播时梯度为$np.where(x > 0, 1, 0) * grad$，即大于0的梯度为1，小于0的部分梯度为0。

2.2.3 损失函数

此处损失函数我选择使用交叉熵损失函数，因为该问题本质上是一个多分类（10）问题，选择用交叉熵损失函数较好。

对于交叉熵损失函数的实现，需要先进行softmax操作，值得一提的是尽管从公式上来看softmax操作很容易实现，但是倘若直接这样去实现是不现实的，因为你很快就会发现程序会溢出，导致算得的loss为nan。其本质原因在于exp函数，因为神经网络的输出中很容易算的一个较为大的值如大于1000等，这时候再取exp就很容易溢出了。对于这一问题的解决方法有两点如下：

1. 直接对softmax取对数，然后直接计算logp(x)，因为后续交叉熵也直接使用lopp(x)，从而不计算p(x)，具体是 $p(x_i) = \frac{e^{xi}}{\sum_j e^{xj}}$，则$logp(x_i) = xi - \log \sum_j e^{xj}$，其中后面的部分成为log_sum_exp函数，得益于其计算方式。这样可以直接计算logp(x)。

2. 在做了上面的变换后，还是不能保证不能溢出，还需要用一个小技巧，找出$M = max(x_1, x_2, ..., x_n)$，由于 $p(x_i) = \frac{e^{xi}}{\sum_j e^{xj}}$，则$p(x_i) = \frac{e^{xi-M}}{\sum_j e^{xj-M}}$,然后根据上式的变换可得：$logp(x_i) = xi - M - \log \sum_j e^{xj-M}$，由于M是x向量中的元素最大值了，这时候xi-M就很小了，可以进行log_sum_exp计算了。具体可以看代码中交叉熵损失函数的前向传播部分。

对于交叉熵损失函数，其值得注意的还有一点那就是其梯度的计算，对于其梯度，如果只看其结论的话是很简洁的，实际上，其推导过程也不复杂，此处简单推导一下。由于多分类交叉熵损失函数只选择奖励正样本，没有惩罚负样本，因此其梯度可以分为两类计算，一类是对于正样本i，由之前前向计算时可知，$logp(x_i) = xi - M - \log \sum_j e^{xj-M}$，那么$\bigtriangledown x_i logp(x_i) = 1 - \frac{e^{xi}}{\sum_j e^{xj}}$，实际上就是$\bigtriangledown x_i logp(x_i) = 1 - softmax(xi)$,而对于负样本j（j != i），其梯度为$\bigtriangledown x_i logp(x_j) = -\frac{e^{xj}}{\sum_j e^{xj}}$,即为 $\bigtriangledown x_j logp(x_j) = -softmax(xj)$。其实现过程可参照代码中交叉熵损失函数反向传播部分。

2.2.4 其它组件

我还添加了优化器optimizer，可以直接调用其step方法实现参数更新。由于我们没有pytorch类似的计算图机制，我们只能手动保存所使用的神经网络层和其运行顺序在optimizer中，这样方便我们更新参数。还可以添加其它组件，如scheduler，我放在下一部分进行说明

3 优化方法说明及对比

需要提前说明的是，我认为本次作业重在考查实现bp及其优化的能力，为了方便对比，此处我没有分验证集，直接在训练集和测试集上进行对比了。而且在对比时其它参数均相同，如batch_size,epochs等，均可在代码中找到说明。

3.1 参数初始化优化

对于参数初始化问题，baseline选择的是正态分布初始化，当然这里也可以选择其它分布如均匀分布，指数分布等，显然，在这些之间，正态分布应该是最好的，因此我没有去实现这些分布，而是采用了He initilization[1]的方法，之所以叫He initilization是因为这种方法是He KaiMing提出来的,有的地方也称为kaiming initilization。其思想是让神经网络各层在初始化后，正向传播时，状态值的方差保持不变；反向传播时，关于激活值的梯度的方差保持不变。其具体推导我没有看懂，仅仅是记住了其结论，即在$W\ N(0, \sqrt{\frac{2}{n_i}})$，其中$n_i$表示上一层的神经元个数（隐层维度）。其实验结果如表1所示：其训练过程图如图1所示:

可以看到，He initilization方式显著提升了准确率，猜想神经网络各层的方差会因为初始化不好而放大导致后续收敛情况不好，而He initilization减缓了这种情况，因此loss可以快速下降，在前面几个epoch的时候已经超过了baseline的最优值了。

method	train loss	train acc	test acc
baseline	0.00608	0.94333	0.92492
he initilization	0.00059	0.99597	0.95966

Table 1: 初始化优化及对比

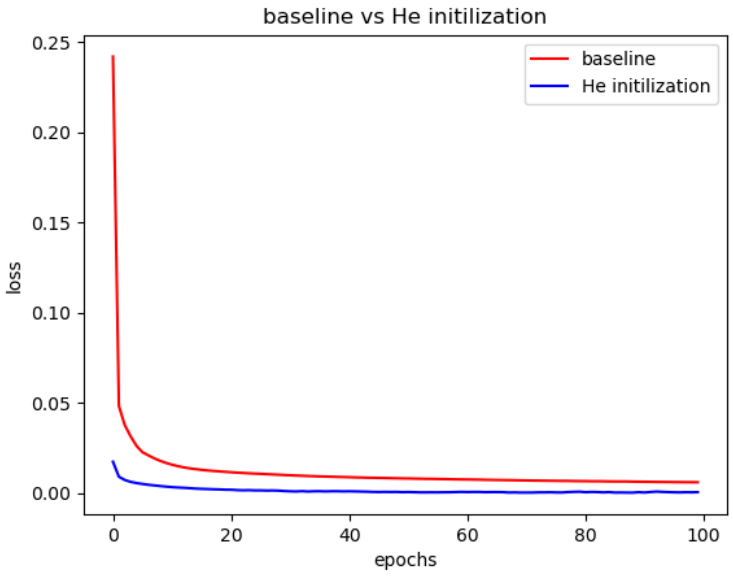

Figure 1: He init vs baseline

method	train loss	train acc	test acc
baseline	0.00608	0.94333	0.92492
weights cross entropy	0.00661	0.93670	0.92043

Table 2: 加权交叉熵损失函数对比

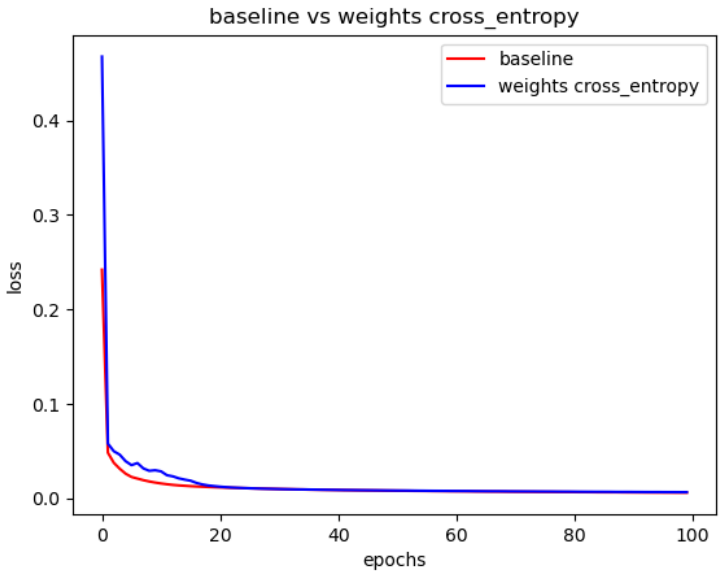

Figure 2: weigths cross entropy vs baseline

3.2 损失函数优化

该问题是个分类问题，因此显然用MSE,MAE等效果均没有交叉熵好，因此我这里选择优化交叉熵损失函数，具体是给交叉熵损失函数加上权重，该权重可以近似代表了各样本类别的比例。该权重损失函数本来是应对类别不平衡的问题的，这里我也使用了，需要变换的一个点是需要提前求出数据集中权重比例。具体是首先统计各类别数目n_i,6ffiM = max($\mathrm{n}_1, n_2, ..., n_m$)，然后取$weights[i] = M/n_i$。在对应反向传播时需要乘以对应的梯度。其实验结果如表2所示。
其训练过程对比图如图2所示。
从结果可以看出来，这种方式要弱于baseline，因为其类别已经较为均衡，故不太适合这种方式，反而回降低精度。

3.3 学习率优化

对于baseline，其学习率使用的是固定学习率，因此一个可以改进的方

method	train loss	train acc	test acc
baseline	0.00608	0.94333	0.92492
linear lr decay	0.00835	0.91863	0.90925

Table 3: 线性衰减学习率对比

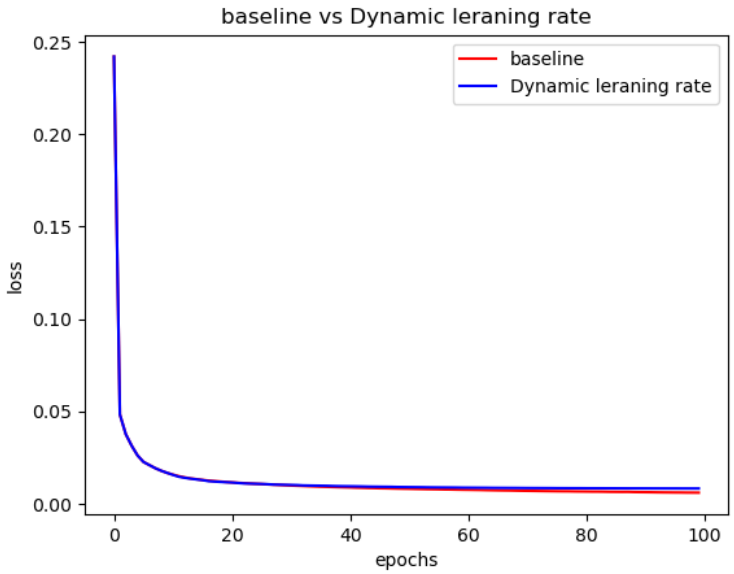

Figure 3: linear lr decay vs baseline

式是使用动态学习率。一种很自然的想法是当模型优化到最优点附近时，应该降低学习率，否则可能因为学习率过大，模型一直在最优点附近振荡，因此可以在一定步数后收敛学习率是一个很好的优化方式，这里我使用了一种较为简单的方式去更改学习率，线性更改方式，就通过一定步数后，让模型原有学习率乘以一个线性衰减权重。这里我设置的衰减系数为0.9，间隔步数为10000步。其对应的实验结果如表3所示。 其训练过程对比图如图3所示。

从结果来看，这个线性衰减学习率也是一个负优化，我大致分析了一下其原因，主要是因为我的学习率不能设置很大，不能超过3e-5，否则很容易溢出，因此导致一个很小的学习率，需要运行很多个epochs才能收敛，因此在还没有接近最优点的情况下，不能减低学习率，否则会使得学习过程变缓，收敛变缓，同样的epochs下其结果弱于baseline。正确的做法应该是先给一个较大的学习率（我这里不能这样做，会溢出），后面再进行衰减。

method	train loss	train acc	test acc
baseline	0.00608	0.94333	0.92492
L2 regular term	0.00842	0.92252	0.91244

Table 4: L2正则项优化对比

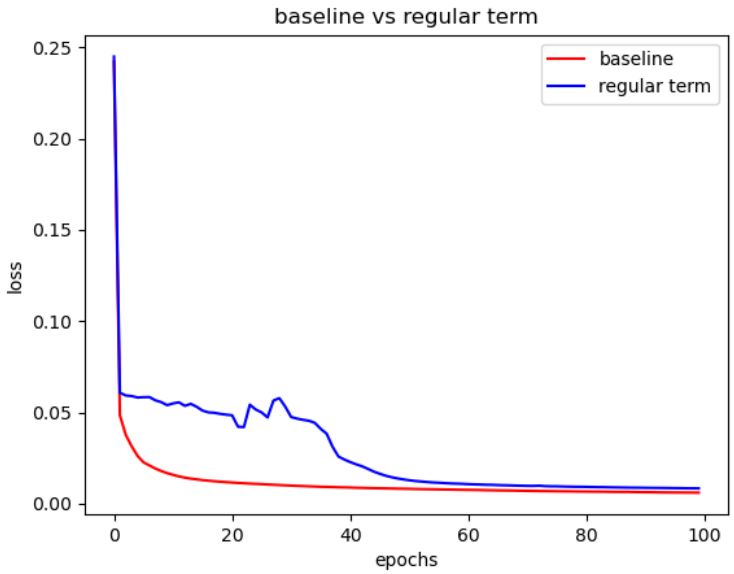

Figure 4: L2 regular term vs baseline

3.4 正则项

baseline是没加正则项的，正则项的作用是防止模型过拟合，即在训练集loss优化到 一定程度后，训练集loss下降，但是验证集loss上升。这是因为模型学习能力过强，因此需要加入正则项惩罚一下，我这里选择的是L2范数正则项，其用数学表达就是$L = \lambda/2\|w\|^2$，λ是一个超参数，一般设置为1e-5,因此这里实现的时候需要在前向传播加上该损失，后向传播加上其对应的梯度为$\lambda * w$。其对应的实验结果如表 4 其训练过程对比图如图4所示。

从实验结果来看，加入正则项的优化也是负优化，其原因与上面学习率衰减原因类似，模型并没有过拟合，相反其还有待优化空间，此时加入正则项或降低学习率会减缓其收敛速度。

method	train loss	train acc	test acc
baseline	0.00608	0.94333	0.92492
best(L2 regular term + linear lr decay)	0.00001	0.99978	0.96226

Table 5: best result

3.5 最好的参数

由于前面提到了学习率不能过大，否则会溢出，因此我加多了epochs数目。还有一个是使用了He(kaiming)intilization后模型能快速收敛到最优值附近，因此在使用了He intilization的基础上再添加正则项和线性衰减次数是比较好的结果。 经过多次实验，我找到了目前最好的参数：
`batch_size:32,epochs:200,learning_rate:3e-3,L2` 正则化 + 线性衰减（间隔步数：`50000,`衰减系数：`0.9`） 其最优实验结果如5所示：

4 优缺点

本次作业的优点是自己手写了MLP，对神经网络了解得更透彻了。但同时也存在很多不足，其中有一些是导致上面负优化的原因。

1. 模型不能搭得太深，我自己写的模型线性层不能搭得太深，否则很容易溢出，而pytorch等框架能够搭得很深，猜想是其在其中做了一些优化，如batch normalization之类的。

2. 学习率不能太大，导致线性衰减学习优化和正则项优化弱于baseline，倘若学习率能设置较大，或许可以改善这个问题。

3. 由于没有类似于框架的计算图，因此需要自己记录所使用的网络层和运行顺序等情况，从代码结构来看，显得很繁琐。

4. 不能用GPU，模型全程是用cpu跑的，尽管模型层数已经很少了，隐藏层维度也设置较小，但是其运行速度依然很慢。

5. 保存模型较为麻烦，如果要保存模型，需要自己实现其接口，保存各项参数，再加载的时候读取，这实在太过麻烦，因此我这里直接就训练好后立即预测，看起来不美观，对于有代码洁癖的人来说无疑很痛苦。

5 代码说明及复现

本次作业全程使用numpy实现,代码也传到Gitee上，对应链接为https://gitee.com/forbetterlife-gq/nn

关于代码结构，其中data 是训练和测试数据目录， output是对应的结果图像目录，dataset.py保存了dataset类和dataloader类，模型的各种实现都在model.py中，包括损失函数，激活函数，MLP,Optimizer，Scheduler 若需要运行代码，可以直接运行train.py，在train.py中有baseline和各种优化及其说明。 需要注意的是在test.py中仅仅实现了predict函数，但是需要在train中调用，因为保存模型很麻烦，所以不需要运行test.py文件。

6 小结

本次作业难度不大，但工作量极大，而且想要写得特别好特别完美是不可能的，因为你会发现你若想优化一下代码，会导致需要修改的地方特别多，甚至需要重新布局，因为你在对抗的是一个框架，要想它完美，那必然是朝着框架的方向去优化，这对于个人来说不太现实，这时候不得不佩服pytorch等框架的开发者们。然后就是找最优的参数时需要尝试很多组不同的参数，然后得到最优结果。我的最优结果是在训练集上准确率99.98%,测试集上准确率96.2%。

References

[1] K. He, X. Zhang, S. Ren, and J. Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.