神经网络 Project 1

1830029007 加兴华

0. 实验环境及文件说明

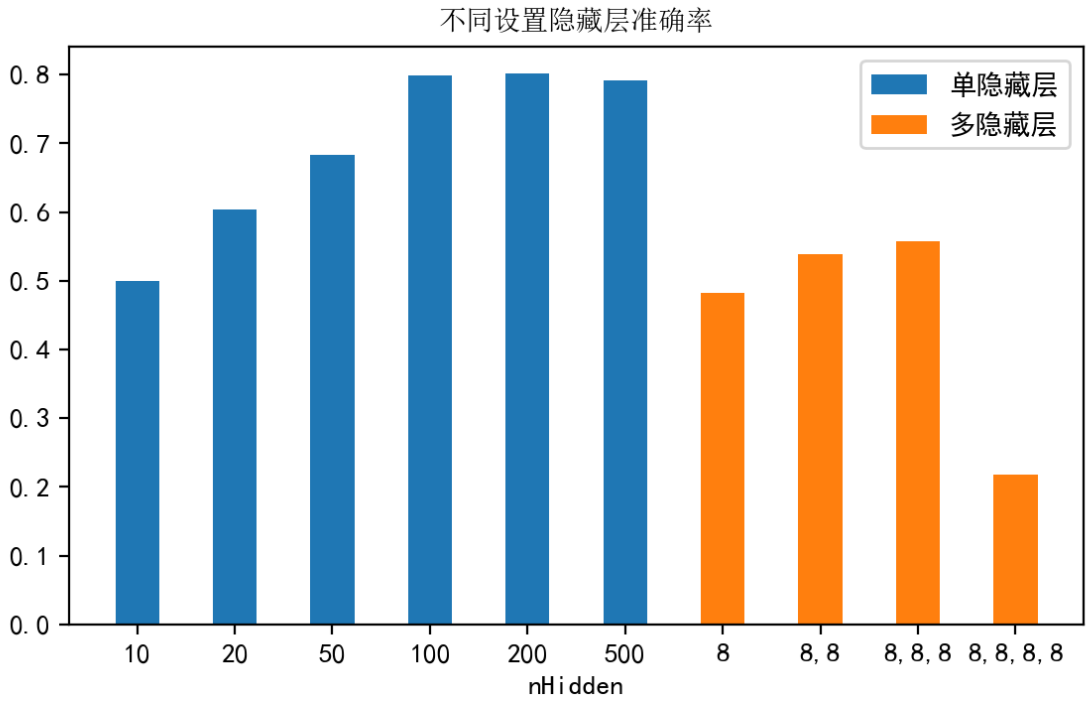
本次项目, 我的实验环境如下:

- 系统: windows 11
- 编程环境: python 3.8.8, Ipython 8.1.1
- CPU: i5-8265U

我的文件说明如下:

文件名	文件说明
project_1.ipynb	用于进行各任务实验的pyhton笔记本,可以找到每个任务的实现
DNNbasic_module.py	含有到Task5 为止的梯度函数和预测函数
Dropout_module.py	含有对DNNbasic进行了dropout修改的梯度函数和预测函数
CNN_module.py	含有对DNNbasic进行了模型首层卷积化修改的梯度函数和预测函数
Draw.py	用于对实验结果进行可视化的文件

1. 改变网络结构



通过多组实验，可以发现，在固定隐含层数量后，改变每层的单元数会对准确度产生影响，大致为精度先增加后减小，减少的原因是模型的过拟合；另一方面，改变隐含层数量也会对准确度产生影响，也大致符合先增后减的趋势。

基于上述结果，在本次项目中，后续的任务中无特殊说明都默认隐含层设为1层，且含有100个单元。

2. 权重下降时带动量

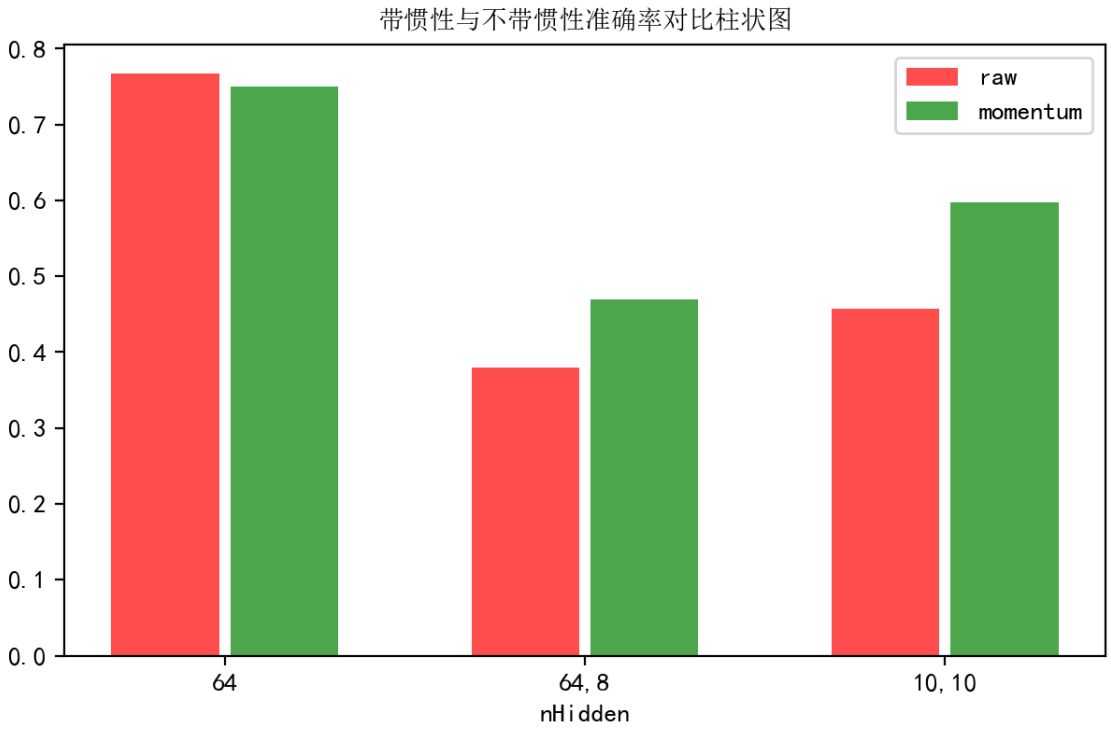
基于理论：

$$w^{t+1}=w^{t}-\alpha_{t}\nabla f\left(w^{t}\right)+\beta_{t}\left(w^{t}-w^{t-1}\right)$$

只要对源代码中权重更新处稍作修改即可让权重下降时带动量：

```
if iter==0:
      w - stepSize*g
elif iter!=0 :
      w = w - stepSize*g+momentum*(w-last)
last=w
```

随后我对不同隐含层组合进行了实验,结果如图:



实验说明带动量权重下降不一定效果会优于原先, 但在模型结构更加复杂时能够保证有更好的效果.

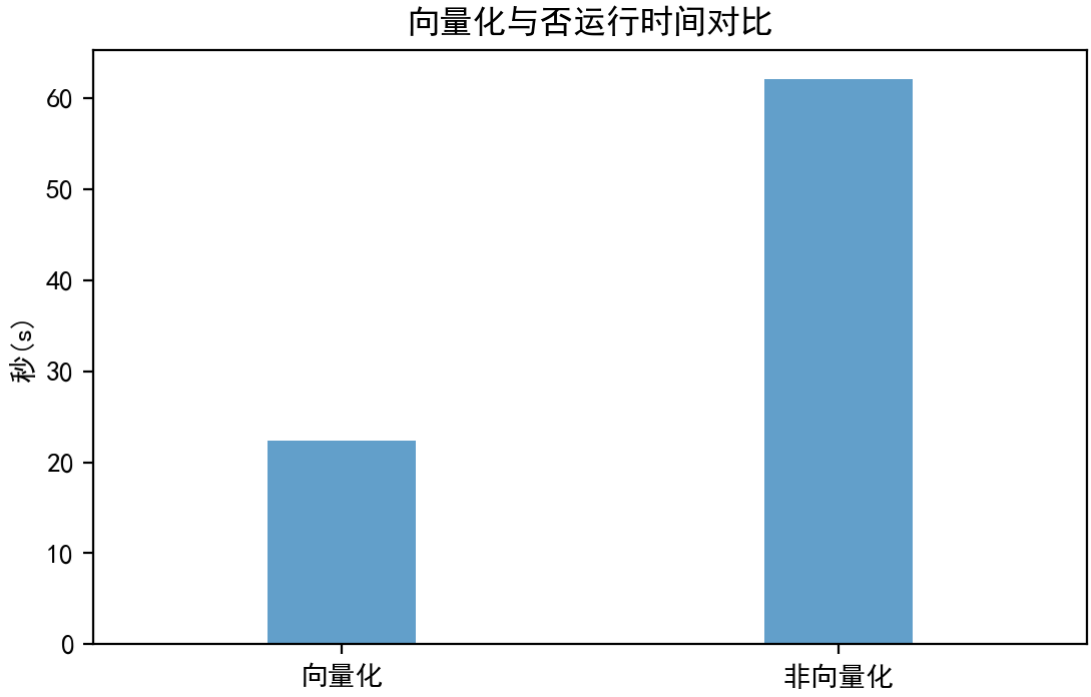
3. 向量化

在将梯度函数和预测函数中可以看作矩阵分块的地方主要为:

- 可以将实例可以打包在一起参与前向传播
- 反向传播时参数梯度可以以向量形式计算

将它们改为矩阵计算后, 记模型名为 'vec'，并将原模型记为 'novec'. 在相同条件下, 通过函数统计运行时长如下:

```
# vec cost 22.40316840000014 second
# novec cost 62.16510610000114 second
```



结果说明在让模型尽可能向量化后大约能让模型速度提升至1/3. 因此后续的实现都将基于向量化的模型展开.

4. 正则化和早停法

4.1 正则化

对于每一块权重区，在反向传播时使用正则化有以下关系：

$$\begin{aligned} Loss &= Err + \lambda \|W\|_2^2 \\ \frac{\partial Loss}{\partial W} &= g + 2\lambda W \end{aligned}$$

基于理论,可以将原代码中权重更新部分修改为:

```
if iter!=0 :w = w - stepSize*(g+penalty*w)+momentum*(w-last)
      last=w
```

随后我对使用正则化和不不使用在相同条件下进行了实验,输入如下:

```
# 带正则
# Training Iterations = 0, validation error = 0.907600
# Training Iterations = 8000, validation error = 0.257000
# Training Iterations = 16000, validation error = 0.238200
# Training Iterations = 24000, validation error = 0.250600
# Training Iterations = 32000, validation error = 0.223000
# Training Iterations = 40000, validation error = 0.231200
# Test error with final model = 0.231000

# 不带正则
# Training Iterations = 0, validation error = 0.897800
# Training Iterations = 8000, validation error = 0.263800
# Training Iterations = 16000, validation error = 0.240400
# Training Iterations = 24000, validation error = 0.235000
# Training Iterations = 32000, validation error = 0.242400
# Training Iterations = 40000, validation error = 0.237400
# Test error with final model = 0.246000
```

通过重复比较带正则与不带正则的模型训练结果，发现虽然使用正则化不会对训练中的误差带来很大的好处，但是却可以让最终的验证误差和测试误差更小，说明正则化对防止模型过拟合有效果。

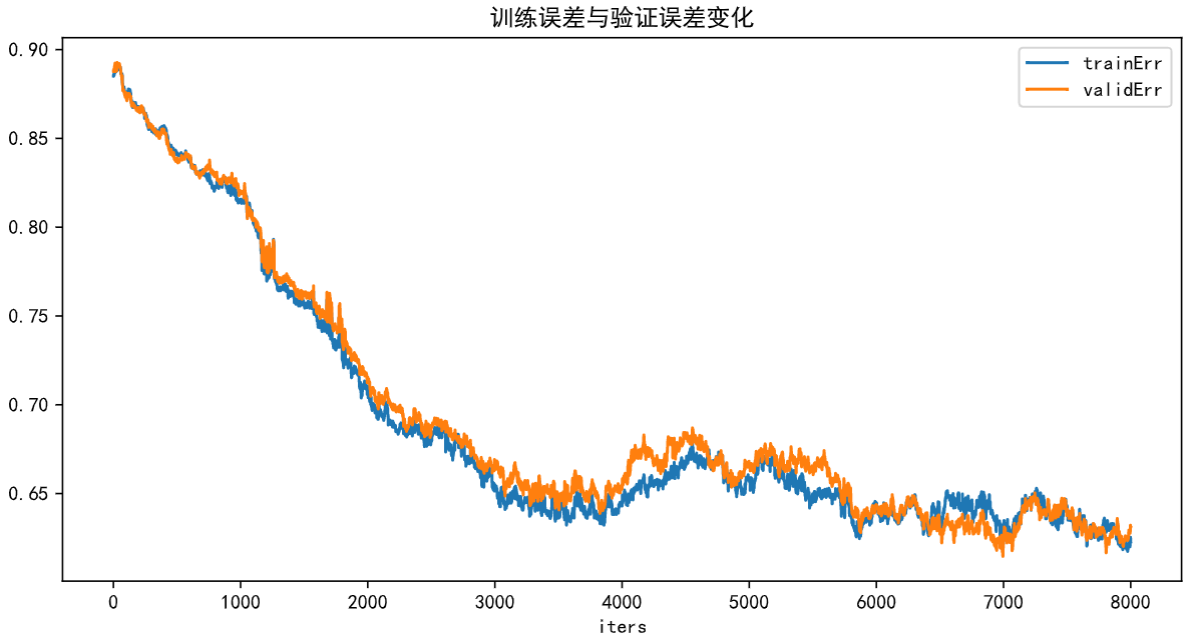
4.2 早停法

除了正则化, 还可以让模型在某次迭代与历史中验证集精度最低点的相对精度差超过阈值就停下:

```
if (verr-min_verr) / min_verr > 5e-2 and flag==0:
      print("\nstop at Iterations= ",iter,', validation error =',min_verr,'\n')
      break
```

随后我对使用早停法的模型在训练中的误差进行了收集,结果如下:

```
# Training Iterations = 0, validation error = 0.888000
# Training Iterations = 1600, validation error = 0.753600
# Training Iterations = 3200, validation error = 0.653200

# stop at Iterations  4071 , validation error = 0.6388

# Training Iterations = 4800, validation error = 0.662000
# Training Iterations = 6400, validation error = 0.637800
# Training Iterations = 8000, validation error = 0.631000

# Test error with final model = 0.624000
```



可以看到如果使用早停法，将会在4000多次迭代时就停下，后续模型误差在训练集和验证集上都是先上升后下降，且下降的低点与停止点差异不大，因此可以认为早停法是有效的。

5. softmax

对于softmax损失函数 ,有以下理论:

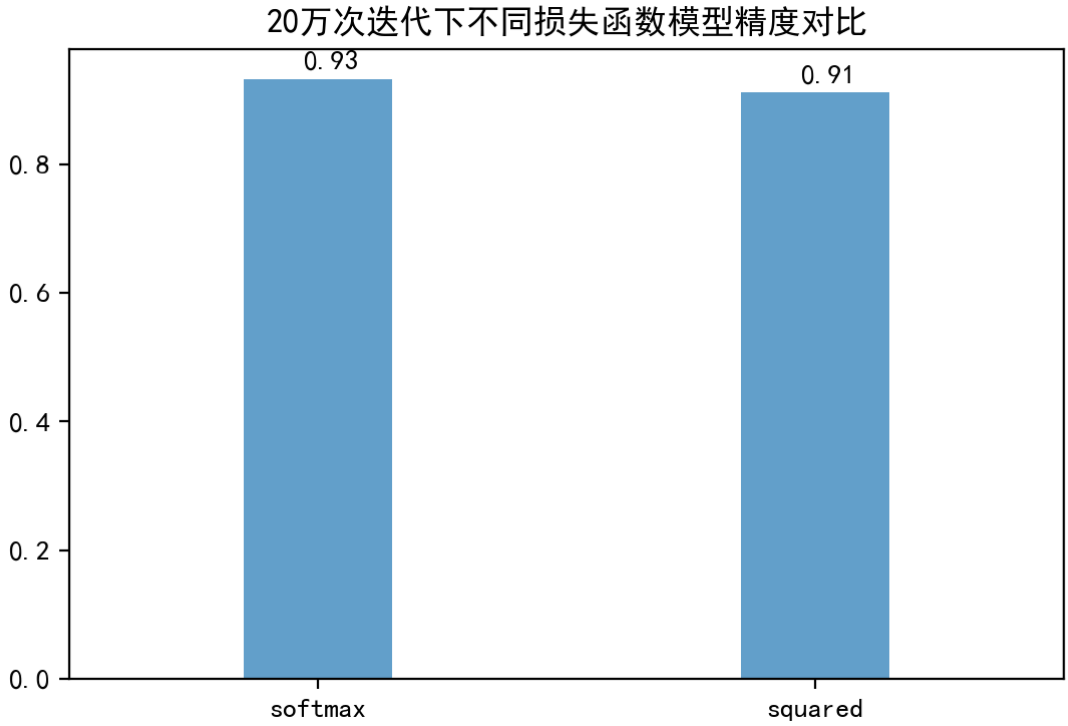
$$
\begin{aligned}
p\left(y_i\right) &= \frac{\exp(z_i)}{\sum_{j=1}^{c} \exp(z_j)} \\
L &= -\ln\left(p\left(y_i\right)\right), \\
\frac{\mathrm{d}L}{\mathrm{d}y_i} &= \begin{cases} p\left(y_i\right) - 1 & i = j \\ p\left(y_j\right) & i \neq j \end{cases}
\end{aligned}
$$

在实现上只需要对损失函数部分进行修改:

```
if Lossmethod=='softmax':
      yhat=yhat.T
      shift_x = yhat - np.max(yhat,axis=0)
      s=np.exp(shift_x)
      s=(s/np.sum(s,axis=0)).T
      f = np.multiply(-np.log(s),y).sum() # 相当于理论中的 L
      err=s
      err[y==1]-=1 # 相当于理论中的梯度
```

随后我对使用不同损失函数的模型进行了对比试验:

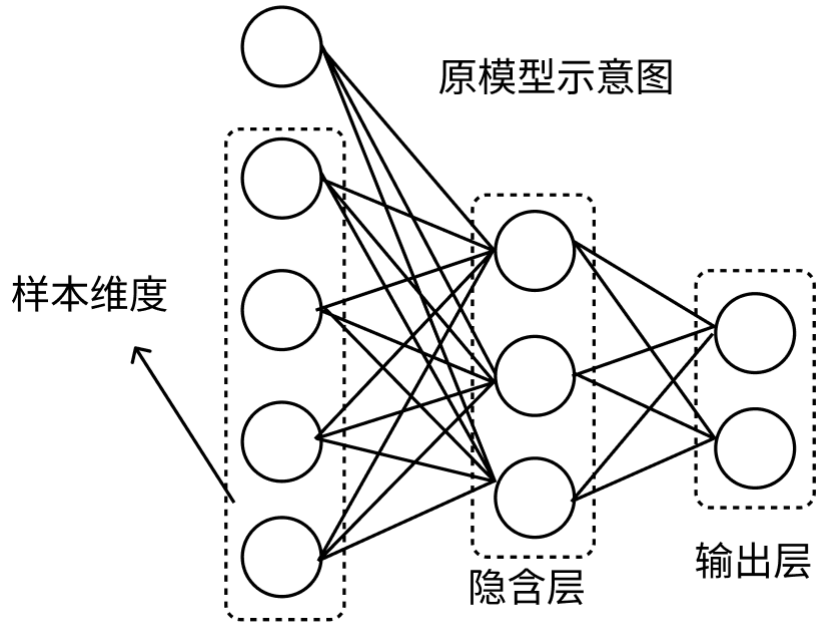
```
# softmax
# Training Iterations = 0, validation error = 0.899600
# Training Iterations = 40000, validation error = 0.271800
# Training Iterations = 80000, validation error = 0.180200
# Training Iterations = 120000, validation error = 0.125000
# Training Iterations = 160000, validation error = 0.091600
# Training Iterations = 200000, validation error = 0.070400
# Test error with final model = 0.068000

# squared
# Training Iterations = 0, validation error = 0.906200
# Training Iterations = 40000, validation error = 0.216800
# Training Iterations = 80000, validation error = 0.181600
# Training Iterations = 120000, validation error = 0.149800
# Training Iterations = 160000, validation error = 0.111400
# Training Iterations = 200000, validation error = 0.080000
# Test error with final model = 0.089000
```

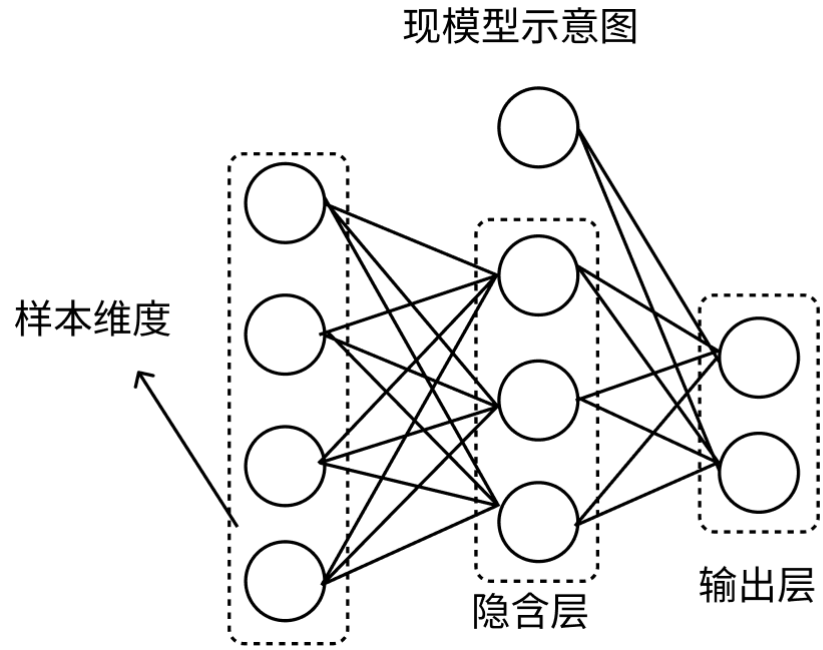


可以看到, 在模型精度本身就较高的情况下, 有时改用softmax损失函数能进一步提高模型精度. 这可能是由于softmax将最后输出解释为概率, 从而使模型有更好的鲁棒性.

> 注: 接下来的任务由于需要修改的代码较多就不再在报告中展示.

6. 隐含层偏置



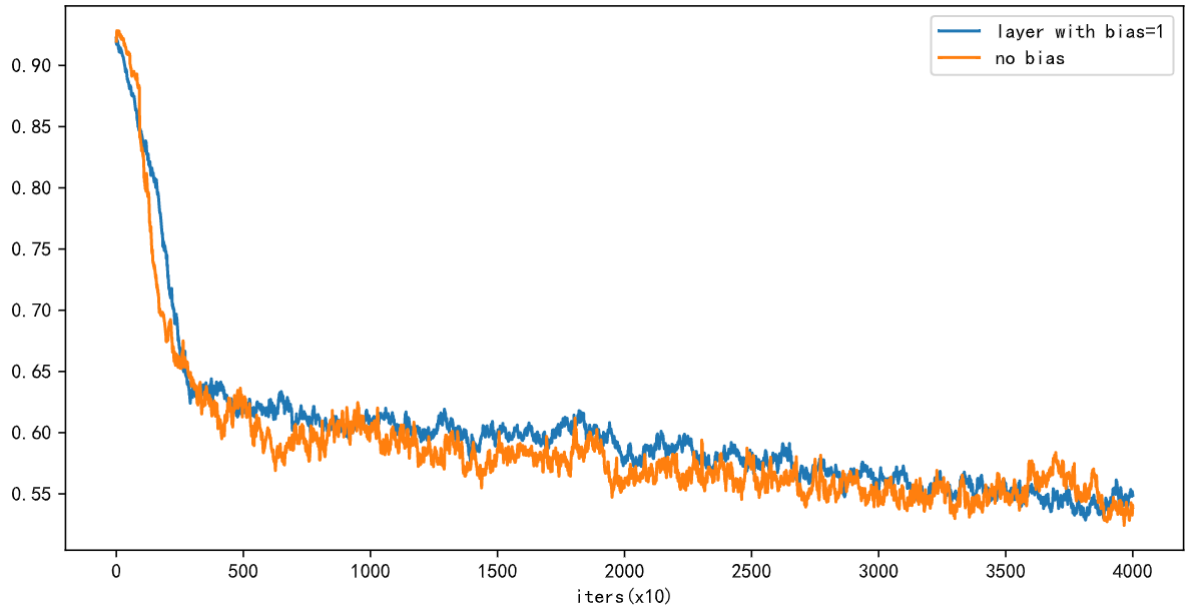


根据示意图，我们需要：

1. 去除加在输入样本上的偏置。
2. 给每个隐藏层都增加一个偏置，且该偏置与更前面的层无关。

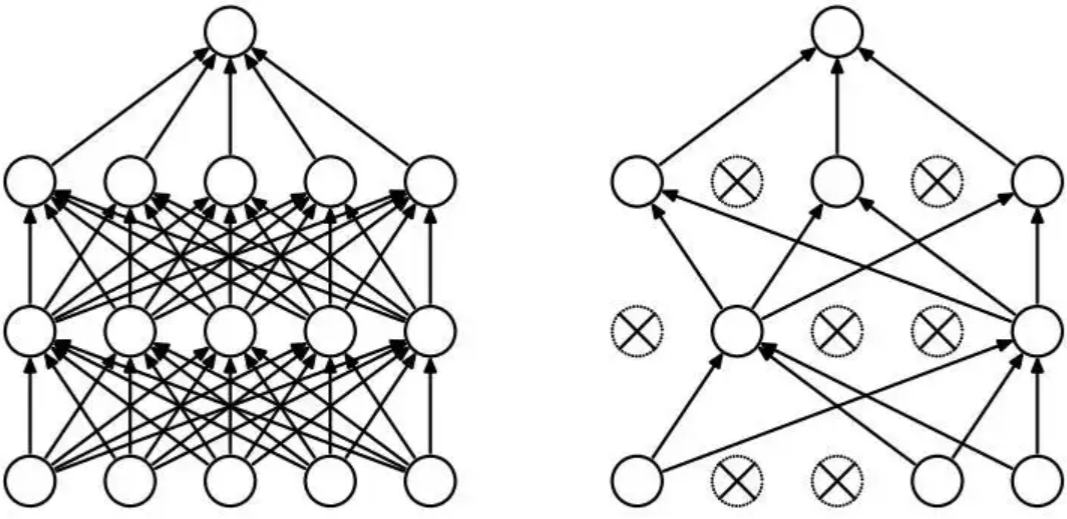
在代码中进行实现后，我收集了模型每迭代10次在验证集上的准确度，绘制如下：

隐含层有无偏置模型验证集准确度走势



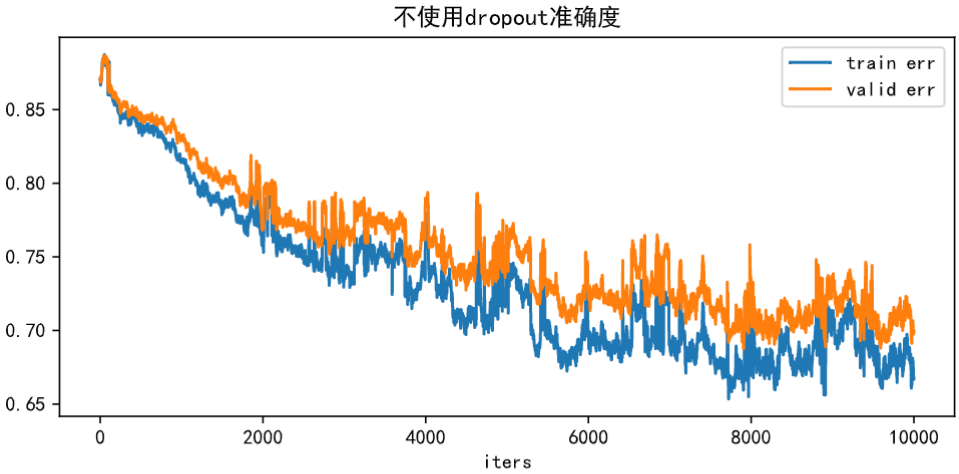
可以看到表现与原来基本一致。

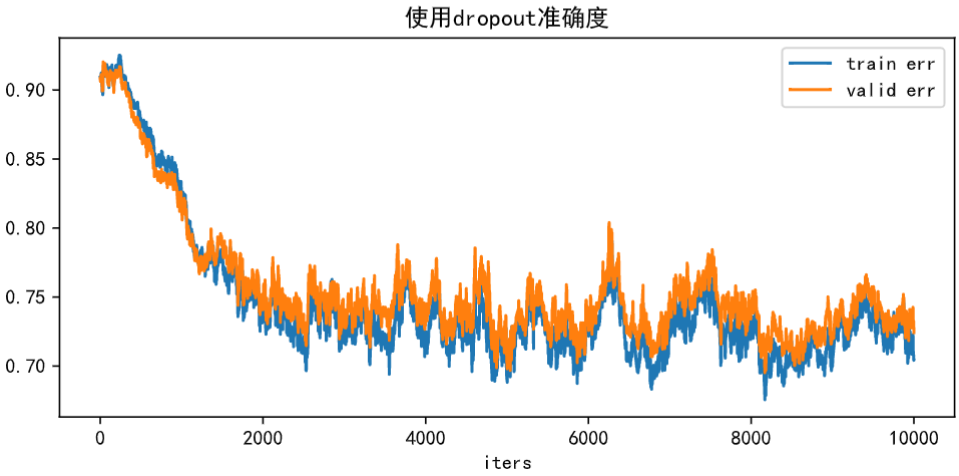
7. dropout



(a) Standard Neural Net (b) After applying dropout.

dropout即让每个隐含层中的神经元以概率 p 不参与到前向传播和反向传播的过程中,在这一次迭代中也不更新梯度. 在代码中实现好后, 我对使用dropout与不使用在相同条件下进行了实验对比:





可以看到，使用dropout的作用与正则化类似，都防止模型出现过拟合，使用前后迭代中训练误差与验证误差的差距明显地得到了改善。

8. fine-tuning

令最后一层权重为W，则问题为：

$$min_W||FW-y||_2^2$$

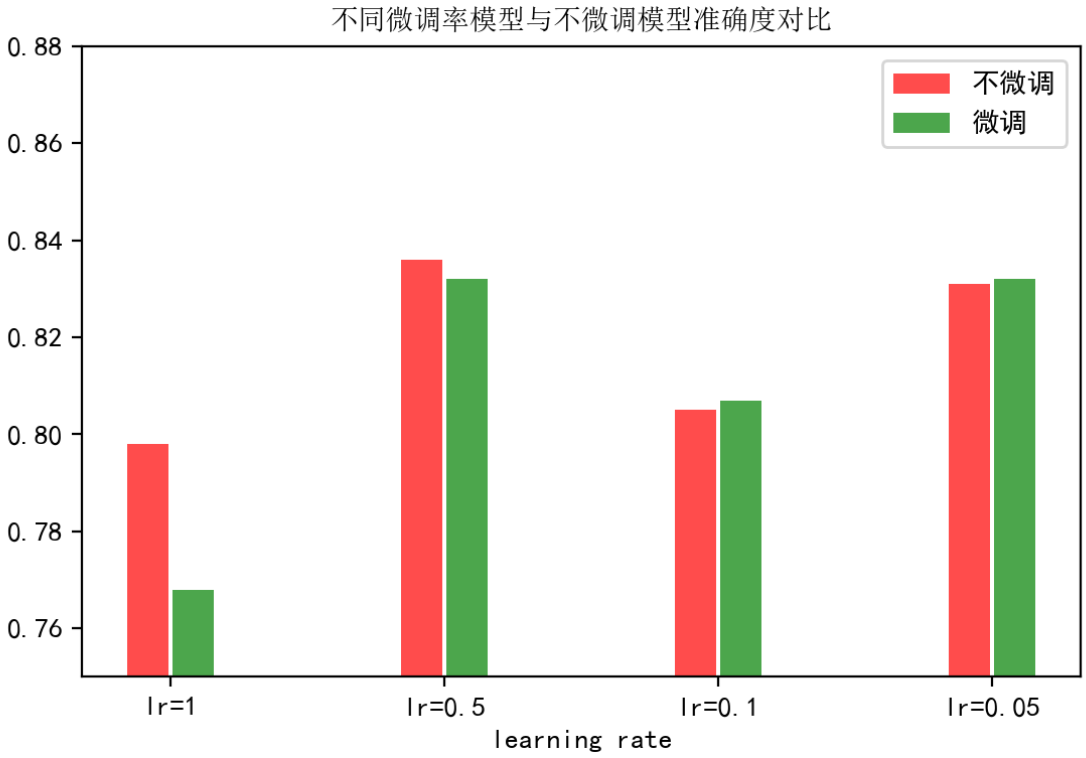
$$W^*=(F'F)^{-1}X'y$$

因此，再训练好模型后将最后一层的权重修改为上述就能实现微调。

在实验中，我发现如果单纯将W替换为W*，对于已经接近收敛的模型可能会起反作用，于是对下面的更新公式进行了实验：

$$W=(1-LearningRate)*W+LearningRate*W^*$$

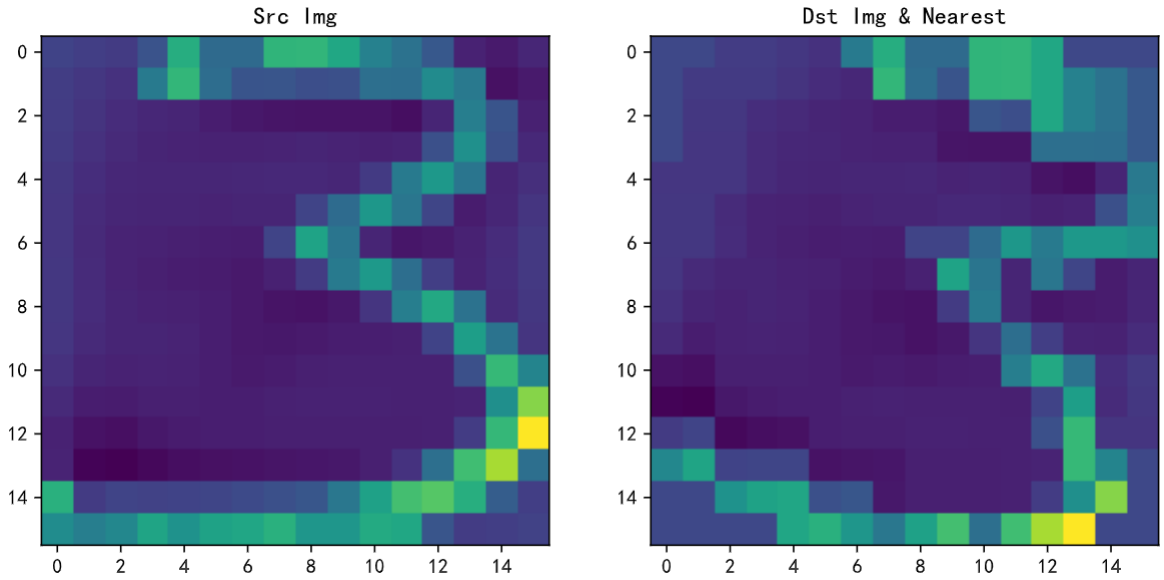
实验结果如下：



可以发现，当学习率为0.1或小于0.1时，使用微调对模型准确率的增益较大，当学习率过高可能会加剧模型的过拟合从而使模型的准确度下降。

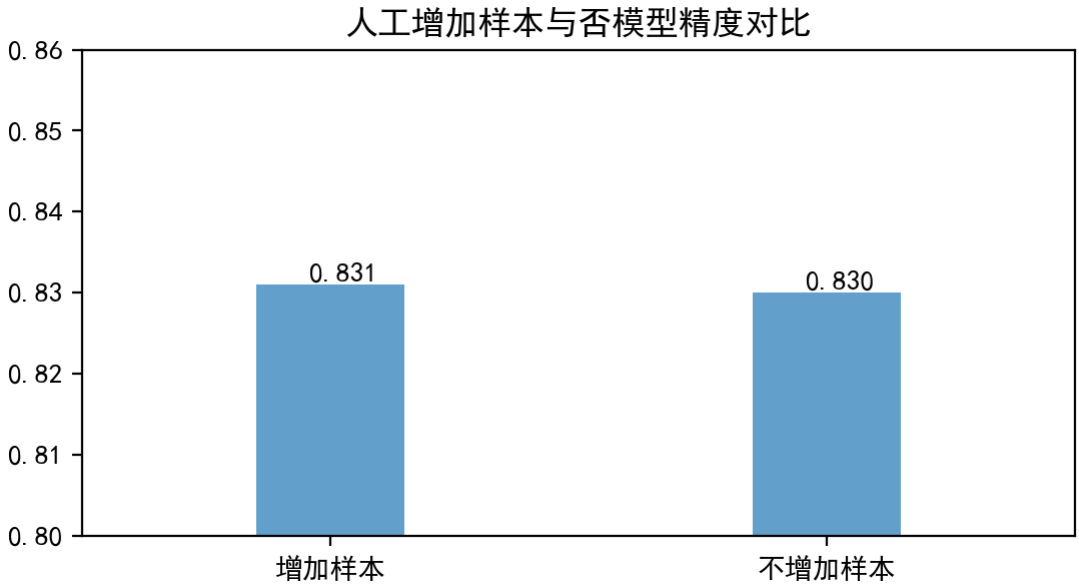
9. 人工增加样本

在实际中我们可以通过对图像数据进行变换来增加样本数，在本项目中，我通过了最近邻法实现了图像的等大小旋转，并应用该函数对样本进行旋转操作，效果如下：



可以看出，当旋转角度不大时，原本图像的信息得到了比较大的保留，理论上是模型可以学习的。实现时，我将该函数加入到模型中进行随机梯度下降前，对选取出的批样本按概率p进行旋转操作，旋转的角度为从-10°到-10°间均匀分布。

在单层100个神经元、100000次迭代、带有动量和正则化的设置下，模型在测试集上的精度如图：



可见，人工增加样本在相同的条件下对模型的精度提升不大，原因可能在于，原来的模型在测试集上的误差并不是由于字迹的方向导致的；也可能是旋转样本使得字迹变得更加模糊，从而使得模型的学习无效。

10. 卷积层

最一般情况的第一层卷积层公式为：

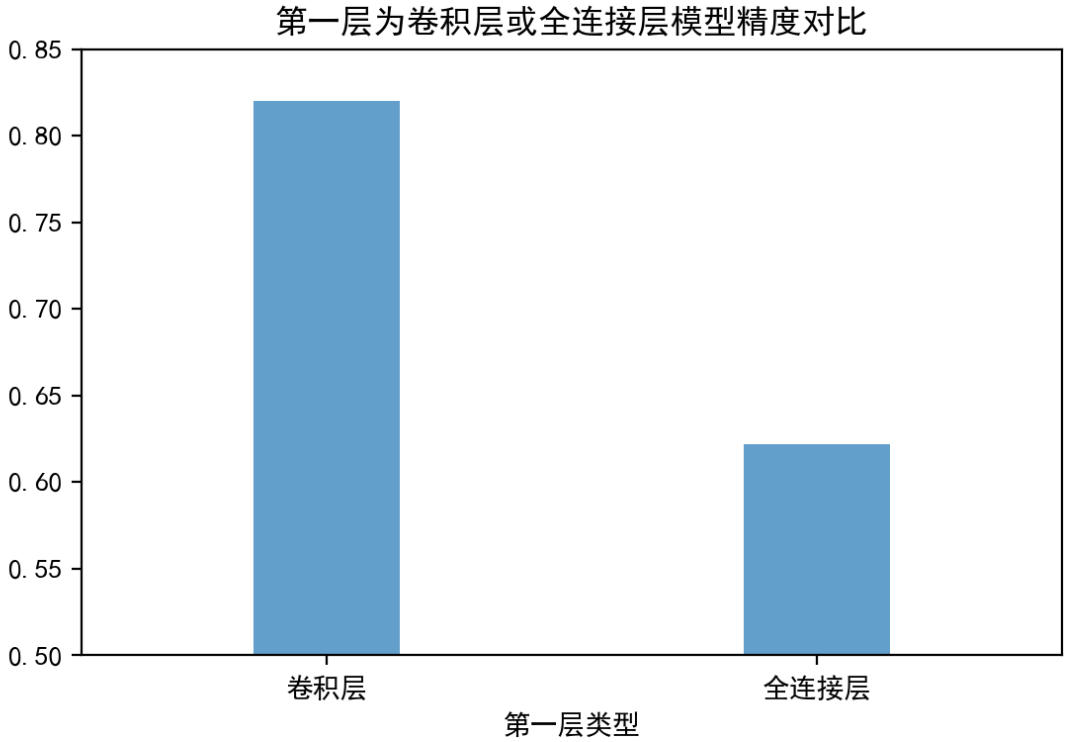
$$\boldsymbol{Z}^p = \boldsymbol{W}^p \otimes \boldsymbol{X} + b^p = \sum_{d=1}^{D} \boldsymbol{W}^{p,d} \otimes \boldsymbol{X}^d + b^p$$
$$\boldsymbol{Y}^p = f\left(\boldsymbol{Z}^p\right)$$

其中p为输出层的厚度，d为输入层的厚度，在本项目中都等于1，涉及参数的梯度为：

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}^{(l,p,d)}} &= \frac{\partial \mathcal{L}}{\partial Z^{(l,p)}} \otimes \boldsymbol{X}^{(l-1,d)} \\
&= \delta^{(l,p)} \otimes \boldsymbol{X}^{(l-1,d)}, \\
\frac{\partial \mathcal{L}}{\partial b^{(l,p)}} &= \sum_{i,j} \left[\delta^{(l,p)}\right]_{i,j} \\
\Rightarrow \delta^{l-1} &= \left(\delta^l * \operatorname{rot} 180\left(W^{l+1}\right)\right) \odot \sigma'\left(z^l\right)
\end{aligned}$$

其中 $\delta^{(l,p)}$ 为反向传播到卷积层 l 并退激活后的第 p 个梯度。

基于理论，我对代码进行修改，构建了一个偏置为0的简易单卷积层神经网络，并对【卷积层，32单元全连接层】的模型和【100单元全连接层，32单元全连接层】在用softmax为损失函数、进行10万次迭代的条件下进行了实验对比，结果如图：



可以看到，第一层为全连接层的模型准确率很低，而第一层为卷积层的模型准确度则能和先前用其他技术辅助的情况一样高，这是一位内卷积不仅具有很好的空间性质，而且使用了权重共享，能降低学习的参数量从而减少模型复杂度。

11. 总结

在本项目中，有一部分任务由于需要并没有将模型算到收敛，而经过实验，我发现采用动量系数=0.5，正则化系数=0.01，单个含有100个神经元的隐含层，损失函数为均方损失的模型收敛时能够得到最高的准确度 97% ，且因为模型经过向量化，即使迭代次数很大，在个人电脑上也能用cpu在很短的时间内计算结束。