# Lab1 第一部分

16302010029 谢东方

目录

[Lab1 第一部分 1](#_Toc526693699)

[感悟 2](#_Toc526693701)

[代码设计与原理说明 2](#_Toc526693702)

[network\_elements.py 3](#_Toc526693703)

[sin有关的代码 7](#_Toc526693704)

[image有关的代码 7](#_Toc526693705)

[Sin函数的拟合 8](#_Toc526693706)

[实验内容 8](#_Toc526693707)

[综上 13](#_Toc526693708)

[图片识别 13](#_Toc526693709)

[batch size 13](#_Toc526693710)

[bias和weight的调整 16](#_Toc526693711)

[不同的learning rate 17](#_Toc526693712)

## 感悟

这几天终于把BP网络写完，非常有成就感！以前一直对它有畏惧心理，觉得自己做不好，因为求导求了多次，总发现有些地方对不上，有些细节的地方没能了解透彻。再一次梳理过后，我开始重拾对神经网络的一点点信心。为什么说是一点点呢？因为超参数调整真是有点玄学。严谨地说，玄学的地方主要在于weight和bias比较没有规律可循，其他超参数和激活函数的选择是有一定规律的。

第一个部分的神经网络写下来，体会比较深的三个部分是神经网络的设计（面向对象）、原理的理解和超参数的调整。首先，是神经网络的设计，python有自己的一套类机制，好好用，好好设计会有很大好处。因为神经网络的复杂度太高，把它拆分之后，能很好降低问题的复杂度。我是将每个层（聚合层、激活层和softmax层）写成一个类，神经网络内部数据由这些类组成，便于灵活调整和替换。我做sin拟合的时候，没有写softmax层，之后，做下一个的时候，发现要加softmax，“虎躯一震”，回过神来，发现事情其实并不难，加一个softmax类就可以了，实现了和老版本的兼容。感觉挺开心的！

其次，原理的理解相当关键，毕竟我们不是搞玄学的嘛，哈哈。我之前一直没有明白X\*W和W\*x为什么不一样，我当时觉得原理和代码实现怎么可以不一样，就是这个小问题困惑了我很久。直到我看完了第二周吴恩达的视频，X是m\*n矩阵，m是样本数，n是feature数，代码这样写是为了批量处理，而原理这样写，W\*x是为了方便大家理解原理，因为每次都要分析m个样本的话，的确增加了理解难度。另外一个重要的发现是求导是用递归的方式（PS: 我很菜的）。以前，我虽然可以很艰难地证明出来求梯度的公式是对的，但是并不太懂它这样做的目的。现在明白目的就是递归，把导数一层一层传回去。感觉之前不懂的很大原因是，我的确就是没有动手实现过，所以比较模糊。

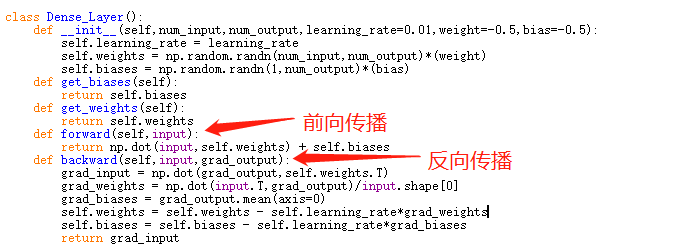
最后就是超参数的调整了。开始调的时候，真是感觉特别摸不着头脑。比方说，bias明明只差零点几，正确率却可以差个10%。后来逐渐找到一些技巧，首先learning rate，learning rate 太大会不收敛，太小迭代次数太慢也不行，所以最好取**最大**一点又不会发散的那个点。激活函数的选取，各有各的优点，下面有实验详细说明，这里就不细说了。层数，令我不解的是，层数少些反而效果好，这个可能还要学到后面才有办法理解。之后就是mini batch灰常好用，尤其是在第二个实验中，是以10个为batch训练的，它的好处是在接受相同数量的样本情况下，可以迅速调整权重。其他超参数调整还是要做对比实验才能发现哪个参数更好，并且，面对不同的实验，对应最佳的权重和偏差也不一样。

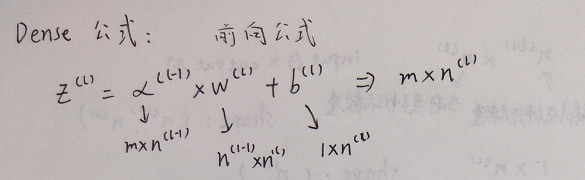
以上就是第一部分的感悟了，未完待续！

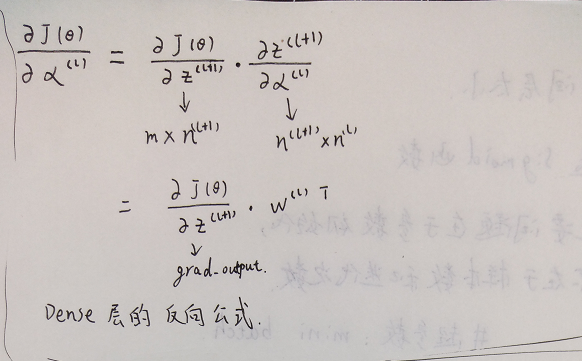
## 代码设计与原理说明

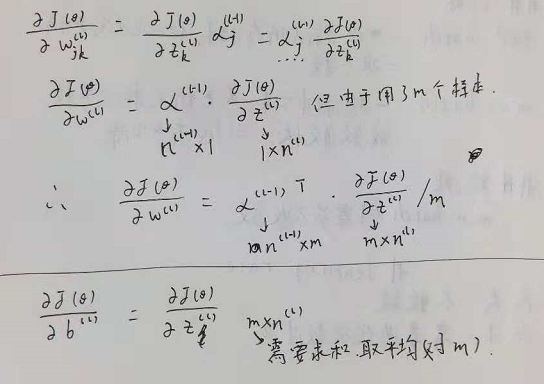
主要的工具类放在network\_elements.py中，sin的拟合放在以sin命名的py文件里面，图片识别放在以get\_im\_array.py和image\_network里面。

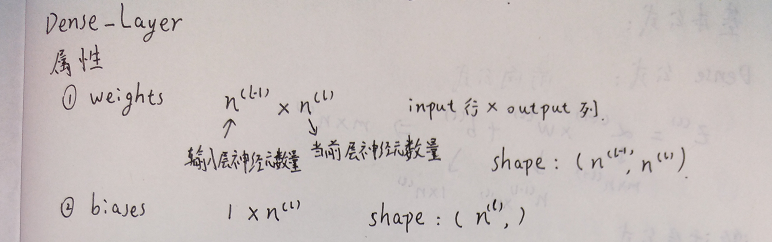
### network\_elements.py

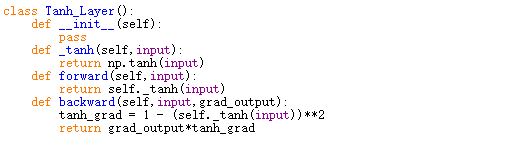


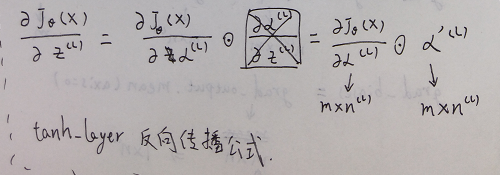




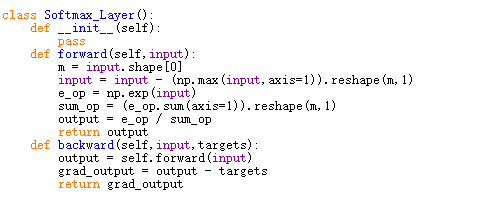


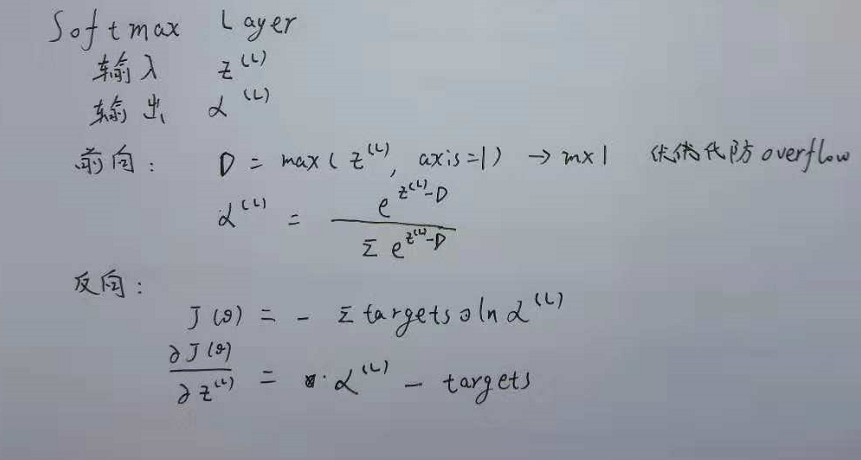




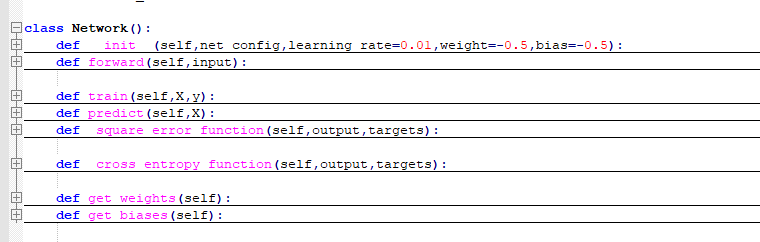


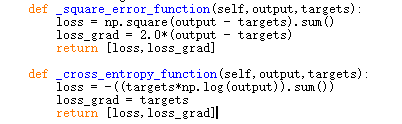
Sigmoid\_Layer和ReLU\_Layer与此相似！

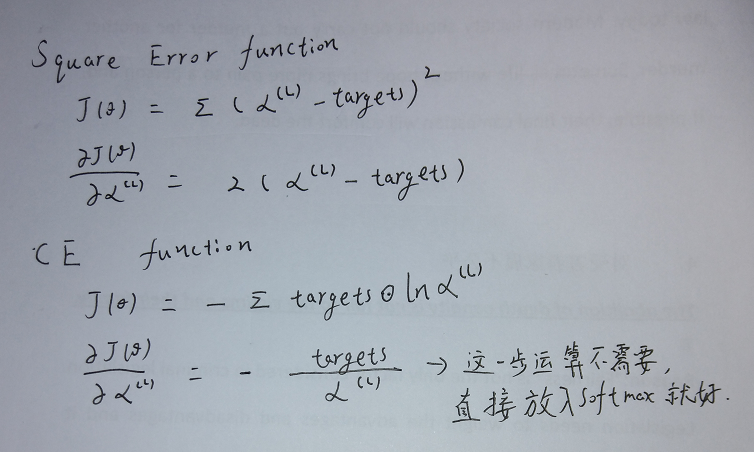




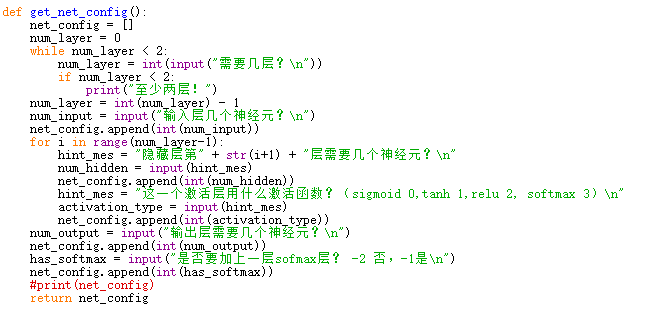
Network类



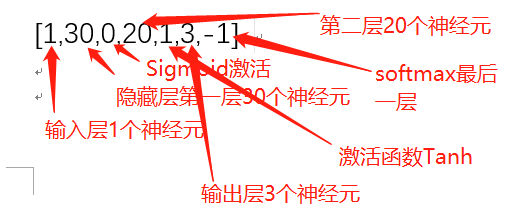




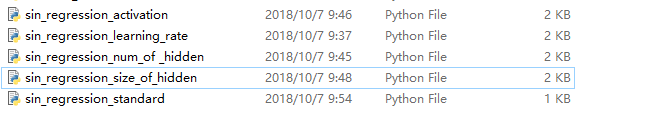
net\_config 是约定的初始化network的参数



可以调用此函数获取。

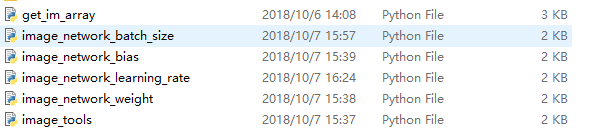


### sin有关的代码



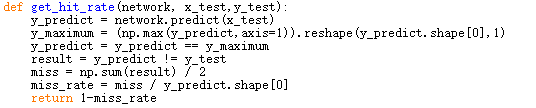
activation， learning rate， num of hidden ，size of hidden 分别对应相应的对比实验。standard表示我觉得最好的版本。

### image有关的代码

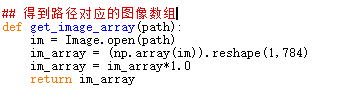


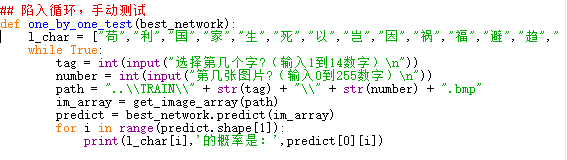
get\_im\_array.py 是数据的初始化，我对初始数据进行了shuffle处理，也许是因为这个效果会好一点。

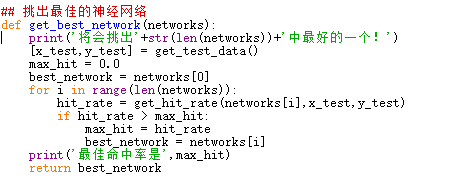
image\_tools.py 含有对应的工具。



得到样本的命中率！







其他的python文件对应相应的对比实验。

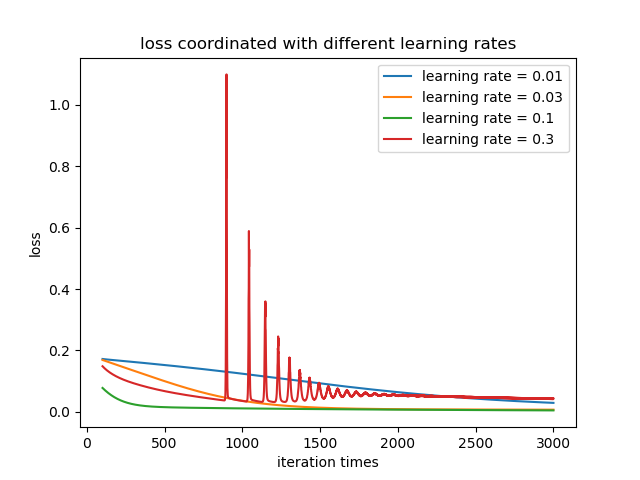
## Sin函数的拟合

### 实验内容

#### 不同learning rate下，loss的下降速度

代码：sin\_regression\_learning\_rate.py

参数设置：3层神经网络，输入层一个神经元，中间层3个神经元，输出层1个神经元。中间层的激活函数是Sigmoid函数。样本数1000，迭代次数3000次，测试样本100个。聚合层weights初始化为randn \* (-0.5)，biases初始化为randn \* (-0.5)。





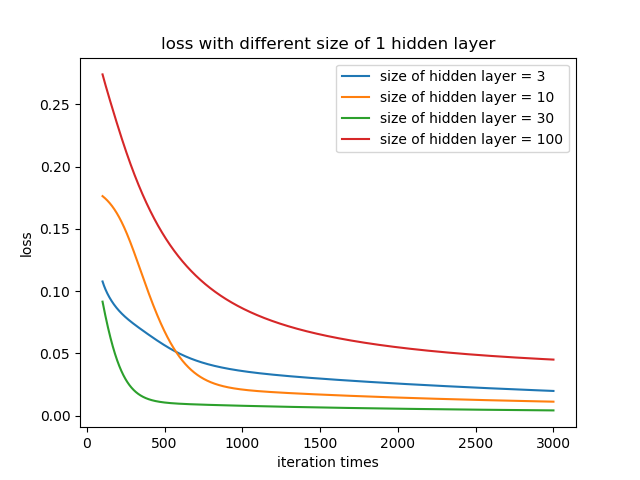
可以看出，learning rate 为0.01是loss下降得较为缓慢，0.03时下降较快，0.1时更快，0.3时出现了大幅度振动的情况，应该是因为learning rate 太大，出现了overshoot的情况。

#### 不同网络结构，loss的下降速度

1. 设置不同的中间层大小

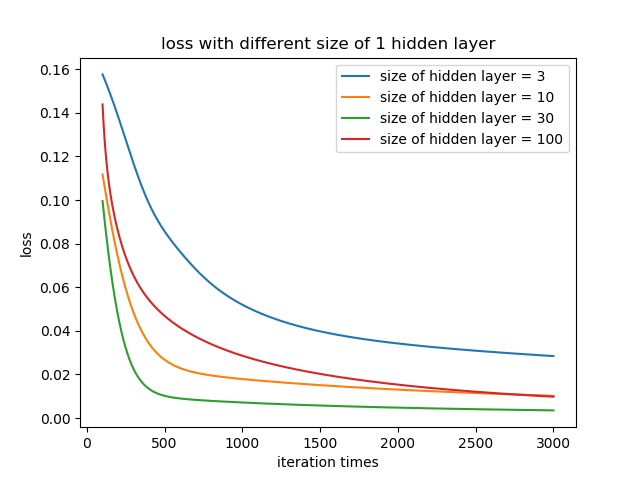
代码：sin\_regression\_size\_of\_hidden.py

参数设置：3层神经网络，输入层一个神经元，中间层个数不确定，输出层1个神经元。中间层的激活函数是Sigmoid函数。样本数1000，迭代次数3000次，测试样本100个。聚合层weights初始化为randn \* (-0.5)，biases初始化为randn \* (-0.5)。learning rate 设置为0.1。



size为10和30的时候，都下降得比较快，但是只有30的时候能够降到最低的loss，因此认为30大小的神经网络能够handle问题的复杂度。size为100的时候，**样本数量可能太少了或者过拟合**，3的时候可能欠拟合了，不能handle问题的复杂度。

当样本数量增加到3000时，

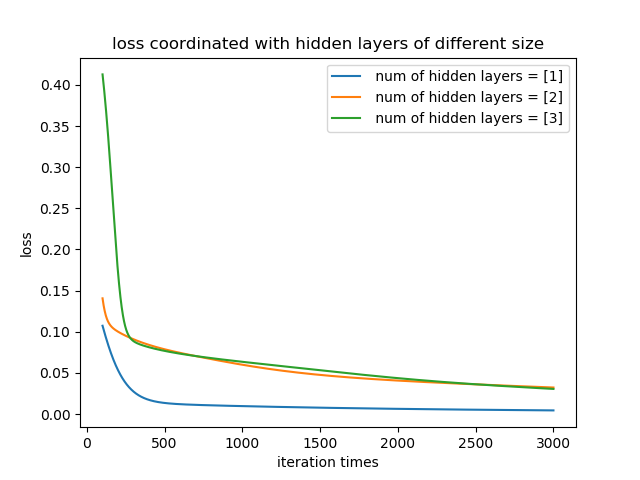


**可见size为30的时候是样本太少，欠拟合！**

1. 不同的中间层数量

代码：sin\_regression\_num\_of\_hidden.py

参数设置：不同层数神经网络，输入层一个神经元，中间层数不确定，输出层1个神经元。中间层的激活函数是Sigmoid函数。样本数1000，迭代次数3000次，测试样本100个。聚合层weights初始化为randn \* (-0.5)，biases初始化为randn \* (-0.5)。learning rate 设置为0.1。中间层，1层的是30个神经元，2层的是3个神经元和10个神经元，3层是3个神经元，6个神经元和12神经元。



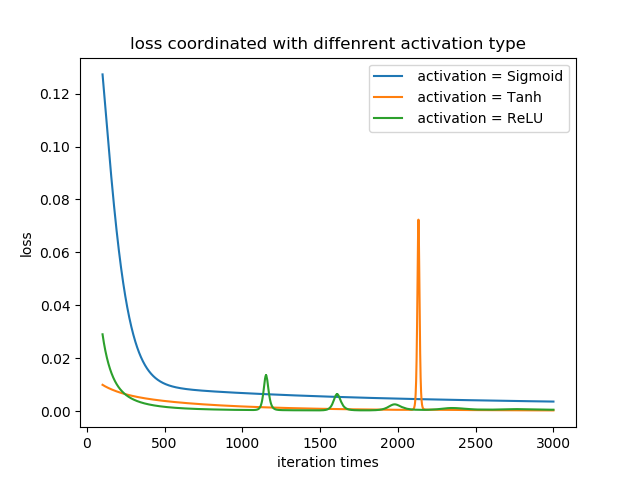
看来还是一层的中间层比较合适。

#### 不同激活函数，loss的下降速度

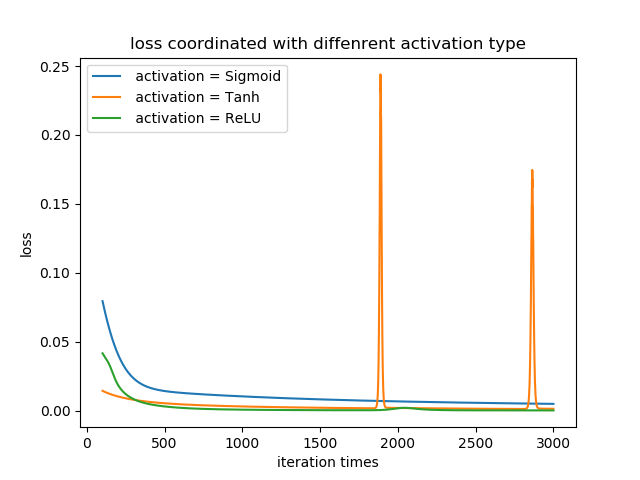
代码：sin\_regression\_activation.py

参数设置：3层数神经网络，输入层一个神经元，中间层数30个神经元，输出层1个神经元。中间层的激活函数是不同的函数。样本数1000，迭代次数3000次，测试样本100个。聚合层weights初始化为randn \* (-0.5)，biases初始化为randn \* (-0.5)。learning rate 设置为0.1。

第一次实验



第二次实验



似乎ReLU更有吸引力！！！但是！ReLU有时候会overflow，所以比较麻烦。

##### 查阅资料知：

1. Sigmoid 函数

缺点是容易出现梯度消失的情况，输出的值不是以0为中心的，幂运算比较耗时。

优点是比较平滑，易于求导

1. Tanh 函数

解决了Sigmoid函数不是以0为中心的问题。

1. ReLU 函数

优点是在正区间内解决了梯度消失的情况，容易计算，防止过拟合，收敛速度快。

缺点是输出值不是以0为中心的，会出现Dead ReLU problem。

### 综上

建议采用参数

代码：sin\_regression\_standard.py

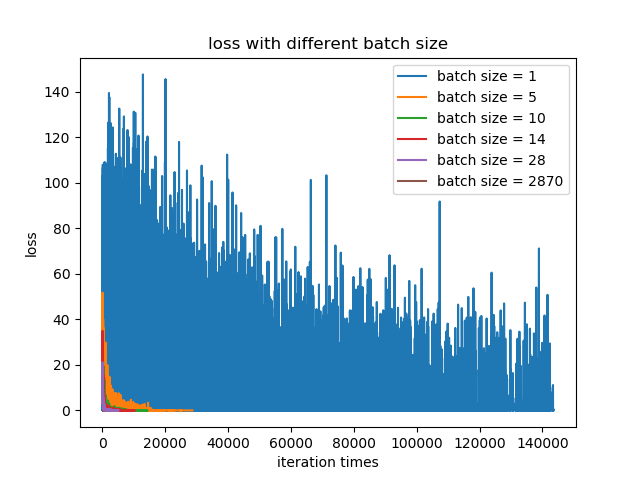
参数设置：**3层数神经网络，输入层一个神经元，中间层30个，输出层1个神经元。中间层的激活函数是Sigmoid函数（如果本人愿意多次Run，可以用ReLU效果更加）。**样本数1000，迭代次数3000次，测试样本100个。聚合层weights初始化为randn \* (-0.5)，biases初始化为randn \* (-0.5)。learning rate 设置为0.1。

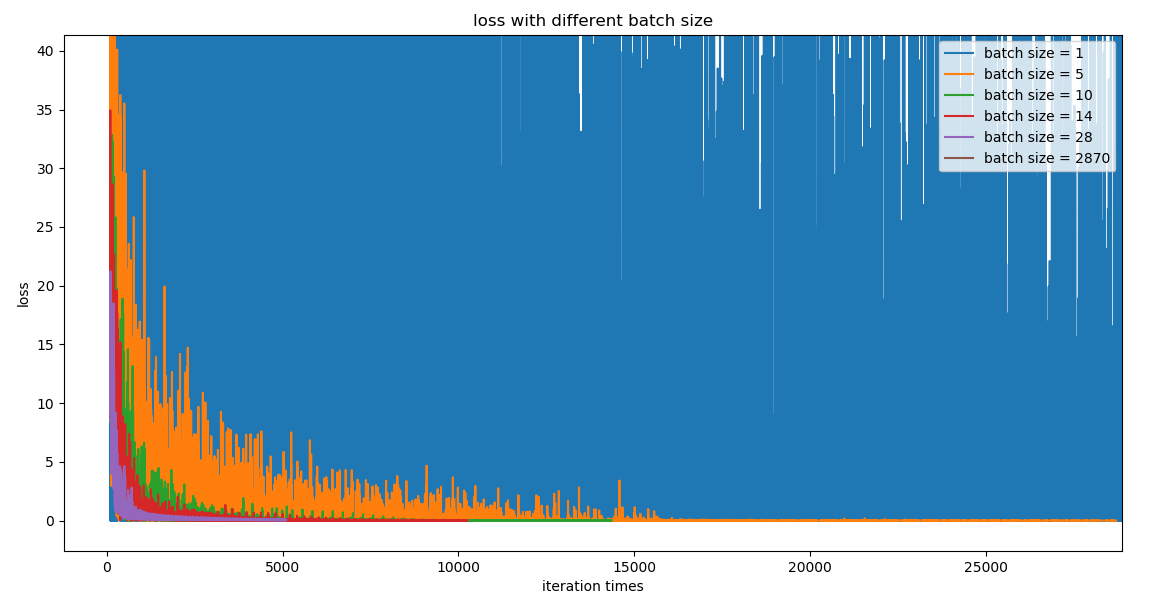
## 图片识别

### batch size

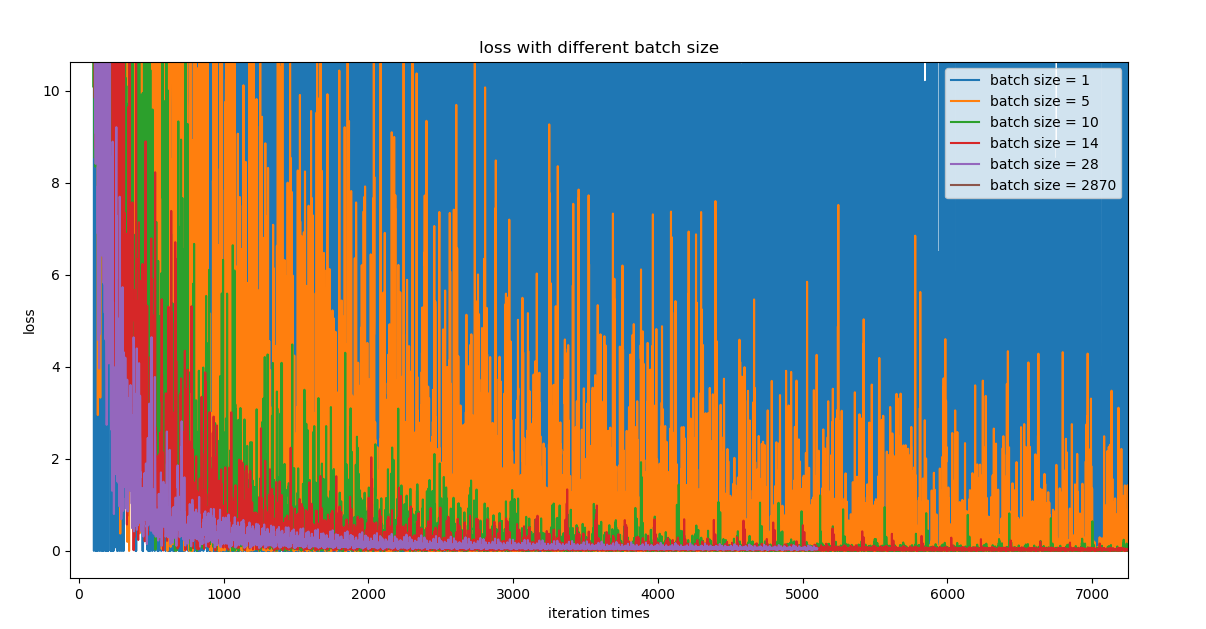
代码： image\_network\_batch\_size.py

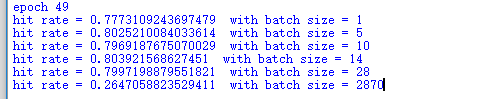
参数设置：一共两层，输入层784个神经元，输出层14个神经元，由一层Dense layer和softmax layer组成。训练样本数量2870个，测试样本数量714个，batch size 分别为1、5、10、14、28、2870（full size）。迭代次数50次。learning rate设置为0.1。weight 初始化为-0.1，bias初始化为0.5。



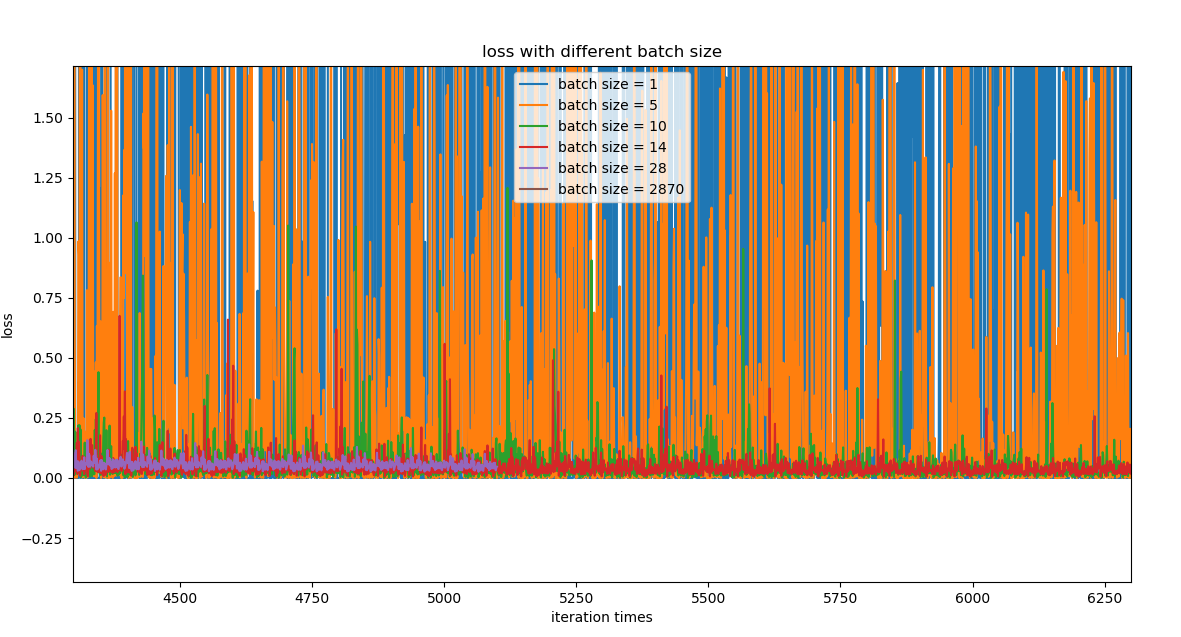


batch size 为1时更新得很快，却很难收敛。5的时候，虽然震荡快，但是最后可以收敛。

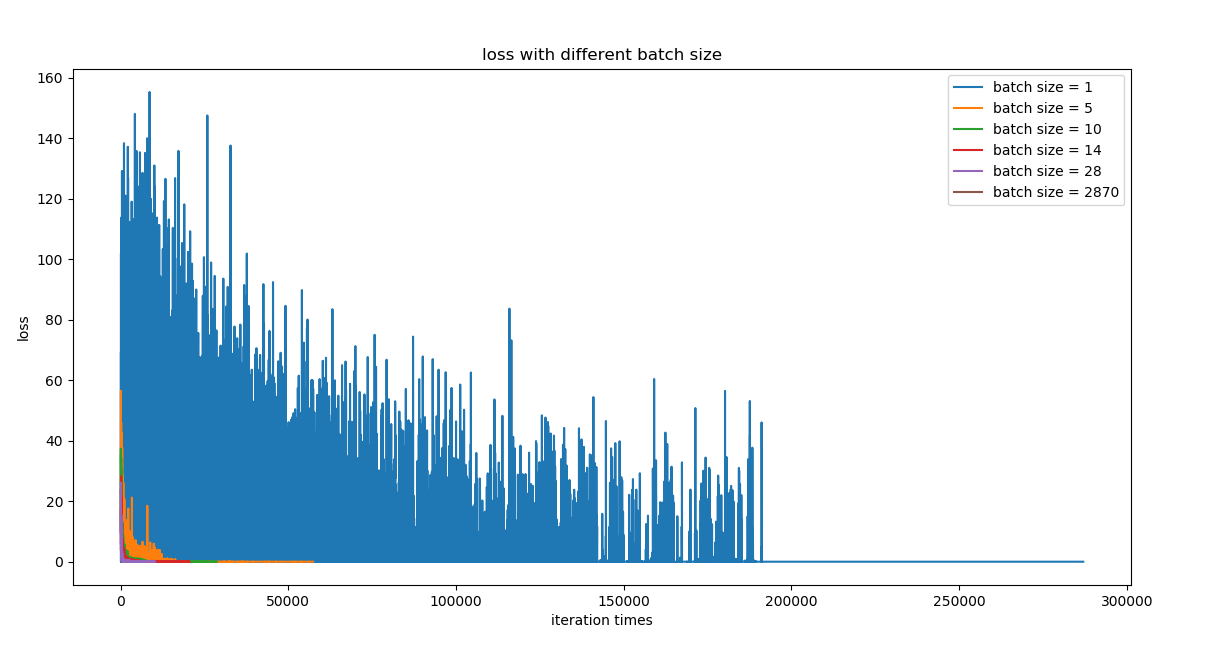


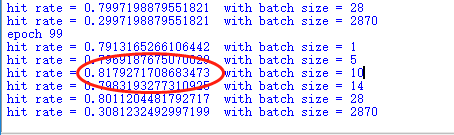


28的效果不错，就是迭代次数少了点，14的效果也不错。

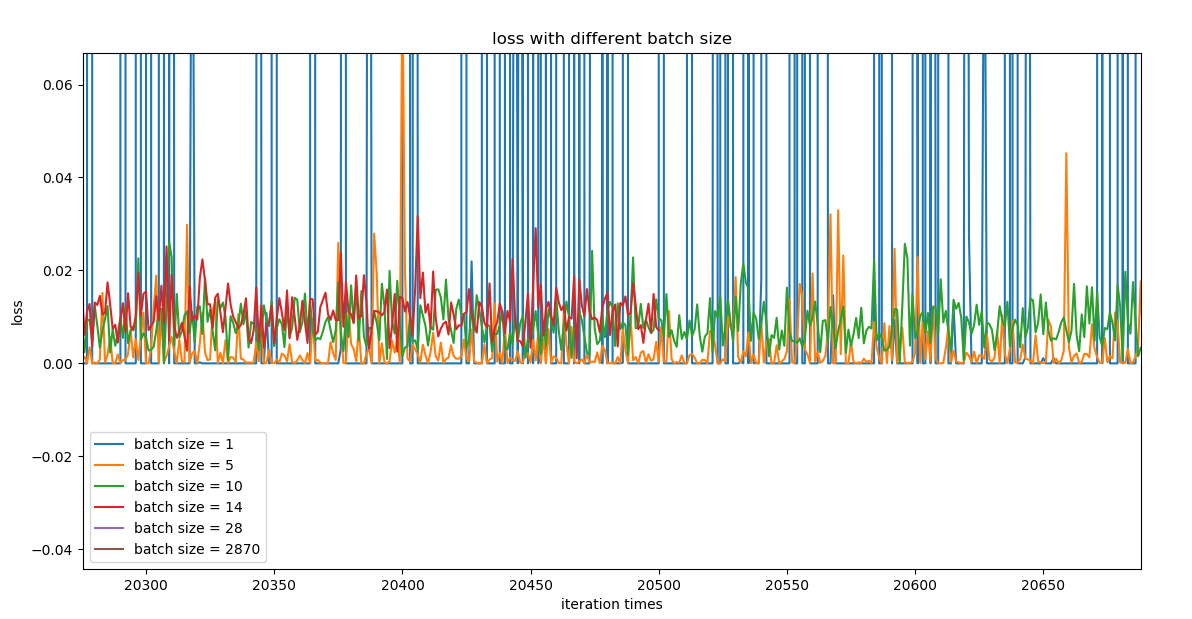


当我迭代100次之后，batch size为1的网络收敛了！





奇迹，居然上了81%！！！



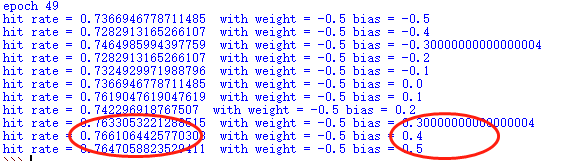
10，14和28的时候好像没有很大区别，都不错。

### bias和weight的调整

1. 不同的bias初始值（random.randn()\*bias）

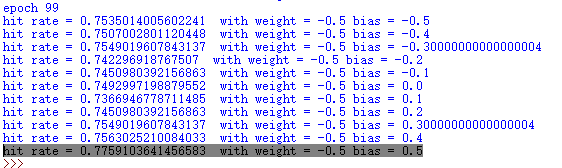
代码：image\_network\_bias.py

参数设置：一共两层，输入层784个神经元，输出层14个神经元，由一层Dense layer和softmax layer组成。训练样本数量2870个，测试样本数量714个，batch size 10个。迭代次数50次。learning rate设置为0.1。



bias 为0.1、0.3、0.4、0.5的时候不错。

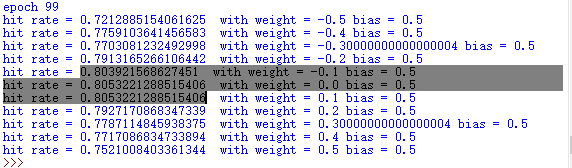
接下来，迭代100次。



发现bias取0.5效果很好。

1. bias取0.5，weight取不同的值(random.randn()\*weight)

代码：image\_network\_weight.py



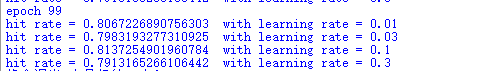
看样子，weight -0.1到0.1都合适。

### 不同的learning rate

代码： image\_network\_learning\_rate.py

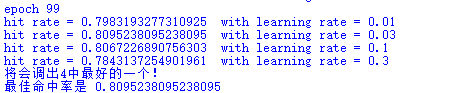
参数设置：一共两层，输入层784个神经元，输出层14个神经元，由一层Dense layer和softmax layer组成。训练样本数量2870个，测试样本数量714个，batch size 分别为10。迭代次数100次。learning rate设置为0.01，0.03, 0.1和0.3。weight 初始化为-0.1，bias初始化为0.5。

第一次试验的结果如下，



0.01和0.1都不错，感觉用0.1比较合适，我也比较喜欢用0.1，哈哈。

第二次试验的结果是这样的，



0.03好像也不错，而0.1还是比较稳定的。