

Lab1 第一部分

16302010029 谢东方

目录

Lab1 第一部分	1
感悟.....	2
代码设计与原理说明.....	2
network_elements.py	3
sin 有关的代码	7
image 有关的代码.....	7
Sin 函数的拟合.....	8
实验内容	8
综上.....	13
图片识别.....	13
batch size	13
bias 和 weight 的调整.....	16
不同的 learning rate.....	17

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 $X \cdot W$ 和 $W \cdot x$ 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，就是这个小问题困扰了我很久。直到我看完了第二周吴恩达的视频， X 是 $m \times n$ 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， $W \cdot x$ 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

```
class Dense_Layer():
    def __init__(self, num_input, num_output, learning_rate=0.01, weight=-0.5, bias=-0.5):
        self.learning_rate = learning_rate
        self.weights = np.random.randn(num_input, num_output)*(weight)
        self.biases = np.random.randn(1, num_output)*(bias)
    def get_biases(self):
        return self.biases
    def get_weights(self):
        return self.weights
    def forward(self, input):
        return np.dot(input, self.weights) + self.biases
    def backward(self, input, grad_output):
        grad_input = np.dot(grad_output, self.weights.T)
        grad_weights = np.dot(input.T, grad_output)/input.shape[0]
        grad_biases = grad_output.mean(axis=0)
        self.weights = self.weights - self.learning_rate*grad_weights
        self.biases = self.biases - self.learning_rate*grad_biases
        return grad_input
```

前向传播

反向传播

Dense 公式: 前向公式

$$z^{(l)} = \underset{m \times n^{(l-1)}}{x^{(l-1)}} \times \underset{n^{(l-1)} \times n^{(l)}}{W^{(l)}} + \underset{1 \times n^{(l)}}{b^{(l)}} \Rightarrow m \times n^{(l)}$$

$$\frac{\partial J(\theta)}{\partial \alpha^{(l)}} = \frac{\partial J(\theta)}{\partial z^{(l+1)}} \cdot \frac{\partial z^{(l+1)}}{\partial \alpha^{(l)}}$$

\downarrow $m \times n^{(l+1)}$ \downarrow $n^{(l+1)} \times n^{(l)}$

$$= \frac{\partial J(\theta)}{\partial z^{(l+1)}} \cdot W^{(l)T}$$

\downarrow grad.output.

Dense 层的反向公式.

$$\frac{\partial J(\theta)}{\partial w_{jk}^{(l)}} = \frac{\partial J(\theta)}{\partial z_k^{(l)}} \alpha_j^{(l)} = \alpha_j^{(l)} \frac{\partial J(\theta)}{\partial z_k^{(l)}}$$

$$\frac{\partial J(\theta)}{\partial w^{(l)}} = \alpha^{(l-1)} \cdot \frac{\partial J(\theta)}{\partial z^{(l)}} \quad \text{但由于用了 } m \text{ 个样本.}$$

\downarrow $n^{(l-1)} \times 1$ \downarrow $1 \times n^{(l)}$

$$\therefore \frac{\partial J(\theta)}{\partial w^{(l)}} = \alpha^{(l-1)T} \cdot \frac{\partial J(\theta)}{\partial z^{(l)}} / m$$

\downarrow $m \times n^{(l-1)} \times m$ \downarrow $m \times n^{(l)}$

$$\frac{\partial J(\theta)}{\partial b^{(l)}} = \frac{\partial J(\theta)}{\partial z^{(l)}} \quad m \times n^{(l)}$$

需要求和取平均(对 m)

Dense-Layer

属性

① weights $n^{(l-1)} \times n^{(l)}$ input 行 \times output 列.
 \uparrow \downarrow
 输入层神经元数量 当前层神经元数量 shape: $(n^{(l-1)}, n^{(l)})$

② biases $1 \times n^{(l)}$ shape: $(n^{(l)},)$

```
class Tanh_Layer():
    def __init__(self):
        pass
    def _tanh(self, input):
        return np.tanh(input)
    def forward(self, input):
        return self._tanh(input)
    def backward(self, input, grad_output):
        tanh_grad = 1 - (self._tanh(input))**2
        return grad_output*tanh_grad
```

$$\frac{\partial J_0(x)}{\partial z^{(l)}} = \frac{\partial J_0(x)}{\partial \alpha^{(l)}} \odot \frac{\partial \alpha^{(l)}}{\partial z^{(l)}} = \frac{\partial J_0(x)}{\partial \alpha^{(l)}} \odot \alpha'^{(l)}$$

\downarrow $m \times n^{(l)}$ \downarrow $m \times n^{(l)}$

tanh-layer 反向传播公式.

Sigmoid_Layer 和 ReLU_Layer 与此相似!

```
class Softmax_Layer():
    def __init__(self):
        pass
    def forward(self, input):
        m = input.shape[0]
        input = input - (np.max(input, axis=1)).reshape(m, 1)
        e_op = np.exp(input)
        sum_op = (e_op.sum(axis=1)).reshape(m, 1)
        output = e_op / sum_op
        return output
    def backward(self, input, targets):
        output = self.forward(input)
        grad_output = output - targets
        return grad_output
```

Softmax Layer

输入 $z^{(L)}$
输出 $\alpha^{(L)}$

前向: $D = \max(z^{(L)}, \text{axis}=1) \rightarrow m \times 1$ 优化防止 overflow

$$\alpha^{(L)} = \frac{e^{z^{(L)} - D}}{\sum e^{z^{(L)} - D}}$$

反向:

$$J(\theta) = - \sum \text{targets} \ln \alpha^{(L)}$$
$$\frac{\partial J(\theta)}{\partial z^{(L)}} = \alpha^{(L)} - \text{targets}$$

Network 类

```
class Network():
    def __init__(self, net_config, learning_rate=0.01, weight=-0.5, bias=-0.5):
    def forward(self, input):

    def train(self, X, y):
    def predict(self, X):
    def square_error_function(self, output, targets):

    def cross_entropy_function(self, output, targets):

    def get_weights(self):
    def get_biases(self):
```

```
def _square_error_function(self, output, targets):
    loss = np.square(output - targets).sum()
    loss_grad = 2.0 * (output - targets)
    return [loss, loss_grad]

def _cross_entropy_function(self, output, targets):
    loss = -((targets * np.log(output)).sum())
    loss_grad = targets
    return [loss, loss_grad]
```

Square Error function

$$J(\theta) = \sum (\alpha^{(L)} - \text{targets})^2$$

$$\frac{\partial J(\theta)}{\partial \alpha^{(L)}} = 2(\alpha^{(L)} - \text{targets})$$

CE function

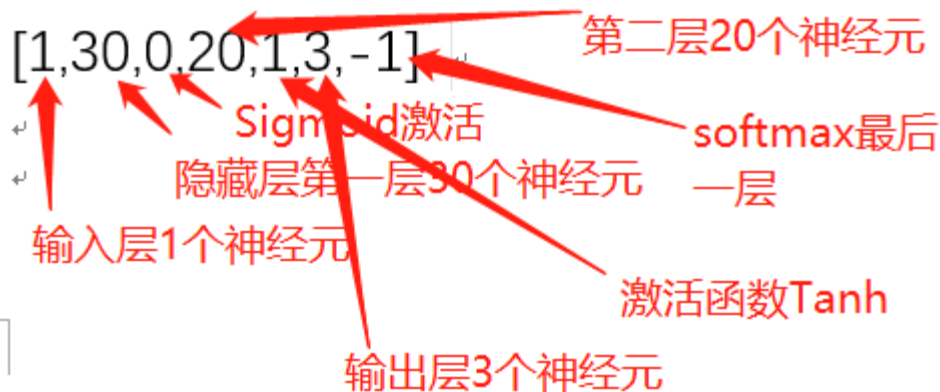
$$J(\theta) = - \sum \text{targets} \ln \alpha^{(L)}$$

$$\frac{\partial J(\theta)}{\partial \alpha^{(L)}} = - \frac{\text{targets}}{\alpha^{(L)}} \rightarrow \text{这一步运算不需要, 直接放入 softmax 就好.}$$






net_config 是约定的初始化 network 的参数

```
def get_net_config():
    net_config = []
    num_layer = 0
    while num_layer < 2:
        num_layer = int(input("需要几层? \n"))
        if num_layer < 2:
            print("至少两层!")
    num_layer = int(num_layer) - 1
    num_input = input("输入层几个神经元? \n")
    net_config.append(int(num_input))
    for i in range(num_layer-1):
        hint_mes = "隐藏层第" + str(i+1) + "层需要几个神经元? \n"
        num_hidden = input(hint_mes)
        net_config.append(int(num_hidden))
        hint_mes = "这一个激活层用什么激活函数? (sigmoid 0, tanh 1, relu 2, softmax 3) \n"
        activation_type = input(hint_mes)
        net_config.append(int(activation_type))
    num_output = input("输出层需要几个神经元? \n")
    net_config.append(int(num_output))
    has_softmax = input("是否要加上一层softmax层? -2 否, -1是\n")
    net_config.append(int(has_softmax))
    #print(net_config)
    return net_config
```

可以调用此函数获取。









sin 有关的代码

 sin_regression_activation	2018/10/7 9:46	Python File	2 KB
 sin_regression_learning_rate	2018/10/7 9:37	Python File	2 KB
 sin_regression_num_of_hidden	2018/10/7 9:45	Python File	2 KB
 sin_regression_size_of_hidden	2018/10/7 9:48	Python File	2 KB
 sin_regression_standard	2018/10/7 9:54	Python File	1 KB

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	2018/10/6 14:08	Python File	3 KB
 image_network_batch_size	2018/10/7 15:57	Python File	2 KB
 image_network_bias	2018/10/7 15:39	Python File	2 KB
 image_network_learning_rate	2018/10/7 16:24	Python File	2 KB
 image_network_weight	2018/10/7 15:38	Python File	2 KB
 image_tools	2018/10/7 15:37	Python File	2 KB

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。

```
def get_hit_rate(network, x_test, y_test):
    y_predict = network.predict(x_test)
    y_maximum = (np.max(y_predict, axis=1)).reshape(y_predict.shape[0], 1)
    y_predict = y_predict == y_maximum
    result = y_predict != y_test
    miss = np.sum(result) / 2
    miss_rate = miss / y_predict.shape[0]
    return 1-miss_rate
```

得到样本的命中率！

得到路径对应的图像数组

```
def get_image_array(path):
    im = Image.open(path)
    im_array = (np.array(im)).reshape(1, 784)
    im_array = im_array*1.0
    return im_array
```

陷入循环，手动测试

```
def one_by_one_test(best_network):
    l_char = ["苟", "利", "国", "家", "生", "死", "以", "岂", "因", "祸", "福", "避", "趋", ""]
    while True:
        tag = int(input("选择第几个字？（输入1到14数字）\n"))
        number = int(input("第几张图片？（输入0到255数字）\n"))
        path = "..\\TRAIN\\" + str(tag) + "\\\" + str(number) + ".bmp"
        im_array = get_image_array(path)
        predict = best_network.predict(im_array)
        for i in range(predict.shape[1]):
            print(l_char[i], '的概率是：', predict[0][i])
```

```

## 挑出最佳的神经网络
def get_best_network(networks):
    | print('将会挑出'+str(len(networks))+ '中最好的一个!')
    [x_test,y_test] = get_test_data()
    max_hit = 0.0
    best_network = networks[0]
    for i in range(len(networks)):
        hit_rate = get_hit_rate(networks[i],x_test,y_test)
        if hit_rate > max_hit:
            max_hit = hit_rate
            best_network = networks[i]
    print('最佳命中率是',max_hit)
    return best_network

```

其他的 python 文件对应相应的对比实验。

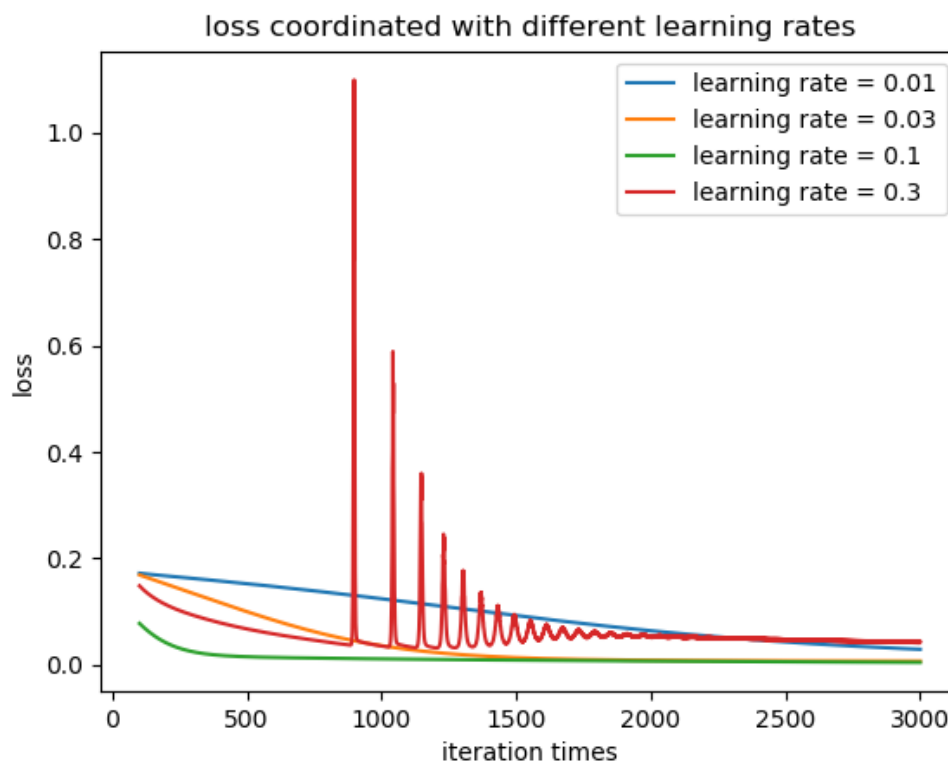
Sin 函数的拟合

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 $\text{randn} * (-0.5)$ ，biases 初始化为 $\text{randn} * (-0.5)$ 。




```
test loss is 0.055839549043404955 with learning rate = 0.01
test loss is 0.0140455703139848 with learning rate = 0.03
test loss is 0.009788715409165788 with learning rate = 0.1
test loss is 0.04767739783563317 with learning rate = 0.3
```

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

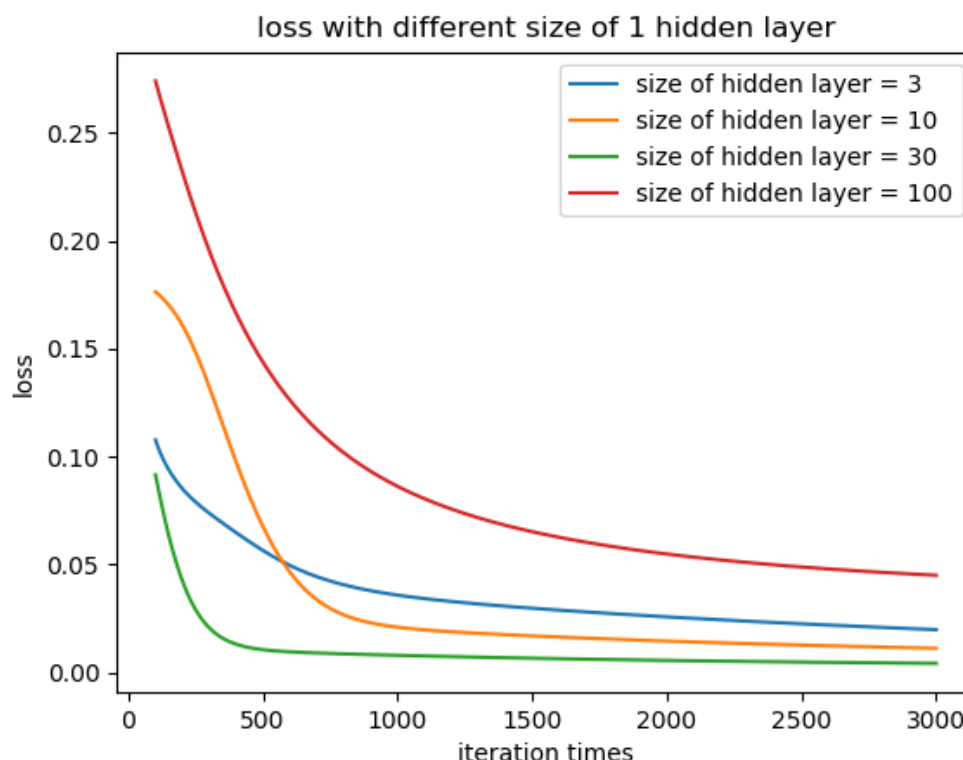
1. 设置不同的中间层大小

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。

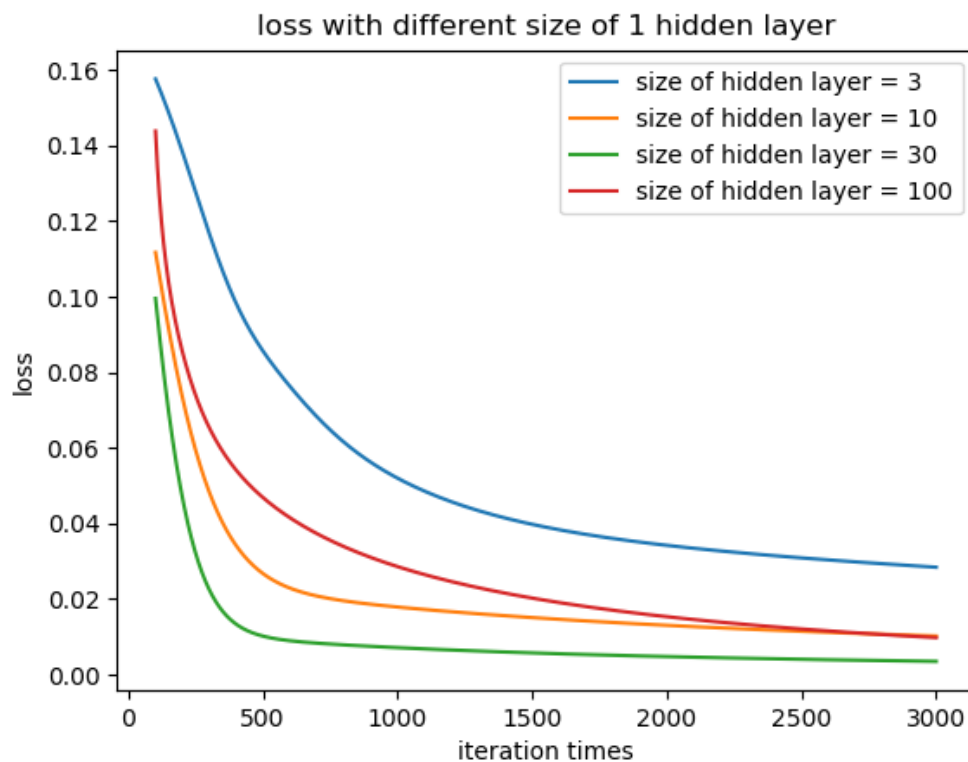
中间层的激活函数是 Sigmoid 函数。样本数 1000, 迭代次数 3000 次, 测试样本 100 个。

聚合层 weights 初始化为 $\text{randn} * (-0.5)$, biases 初始化为 $\text{randn} * (-0.5)$ 。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,

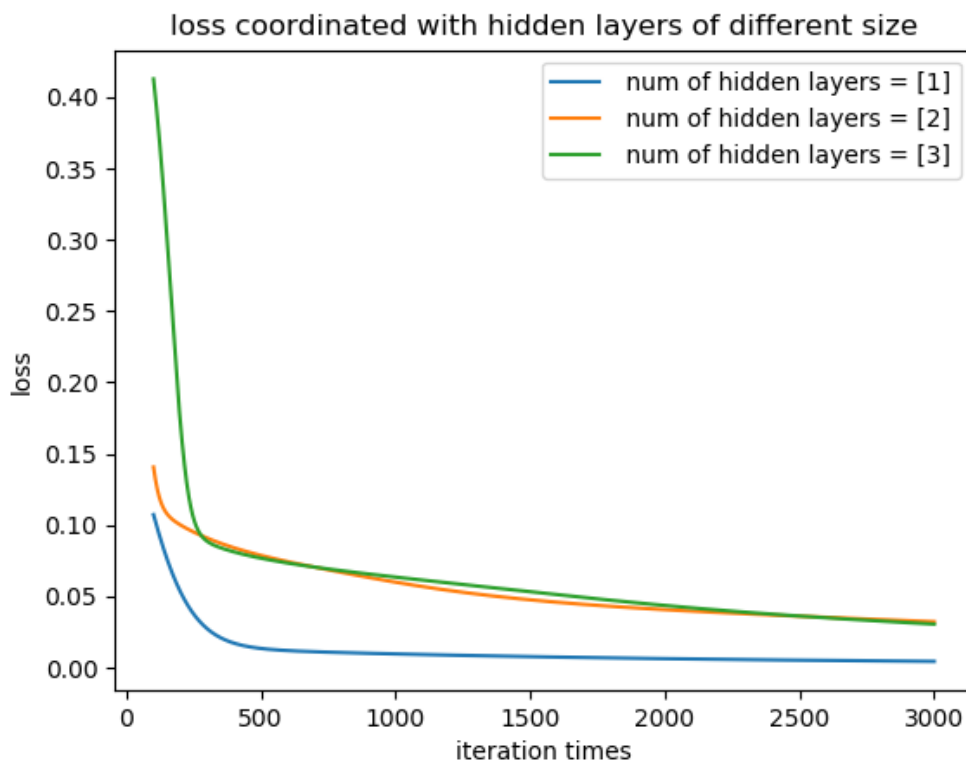


可见 size 为 30 的时候是样本太少，欠拟合！

2. 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 $\text{randn} * (-0.5)$ ，biases 初始化为 $\text{randn} * (-0.5)$ 。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 个神经元。



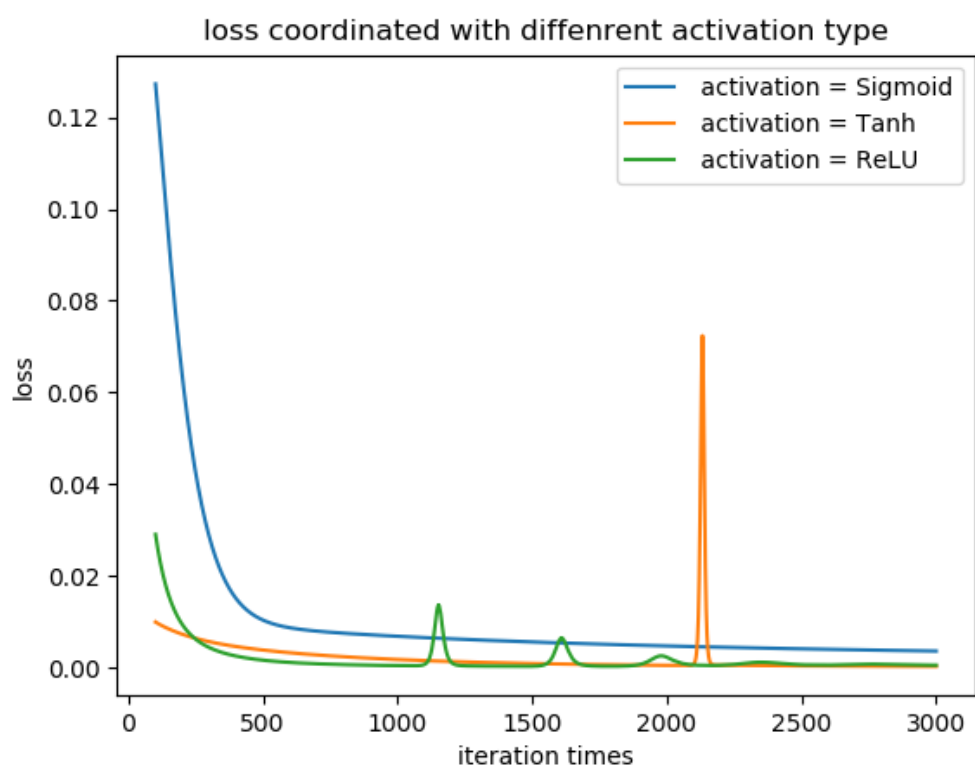
看来还是一层的中间层比较合适。

不同激活函数，loss 的下降速度

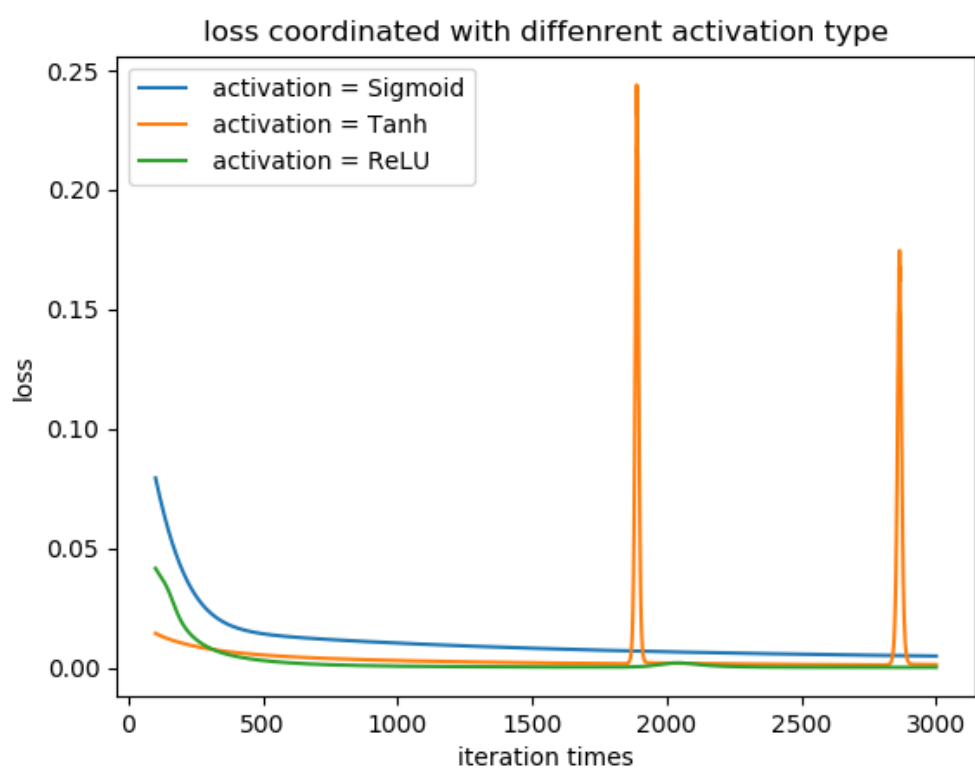
代码：sin_regression_activation.py

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 $\text{randn} * (-0.5)$ ，biases 初始化为 $\text{randn} * (-0.5)$ 。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
2. Tanh 函数
解决了 Sigmoid 函数不是以 0 为中心的问题。
3. 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

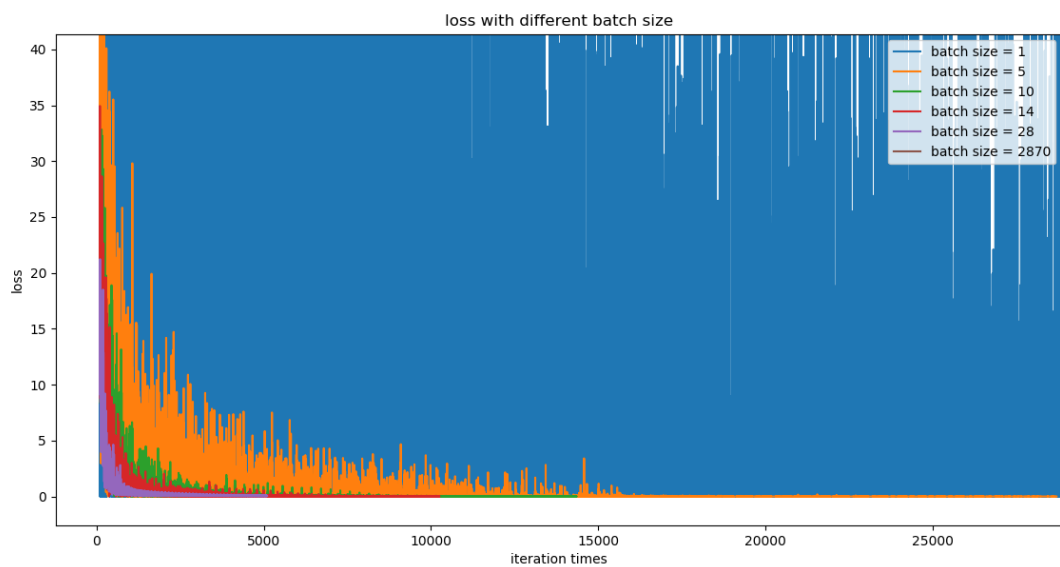
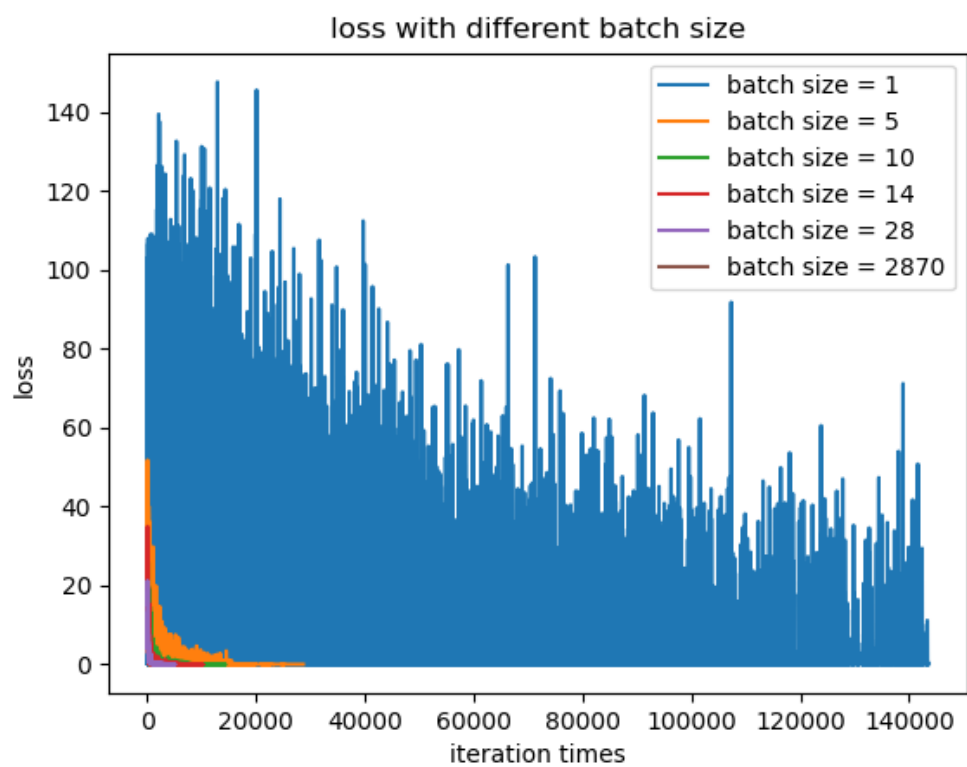
参数设置：**3 层数神经网络**，输入层一个神经元，中间层 30 个，输出层 1 个神经元。中间层的激活函数是 **Sigmoid 函数**（如果本人愿意多次 Run，可以用 **ReLU** 效果更加）。样本数 1000，迭代次数 3000 次，测试样本 100 个。聚合层 weights 初始化为 $\text{randn} * (-0.5)$ ，biases 初始化为 $\text{randn} * (-0.5)$ 。learning rate 设置为 0.1。

图片识别

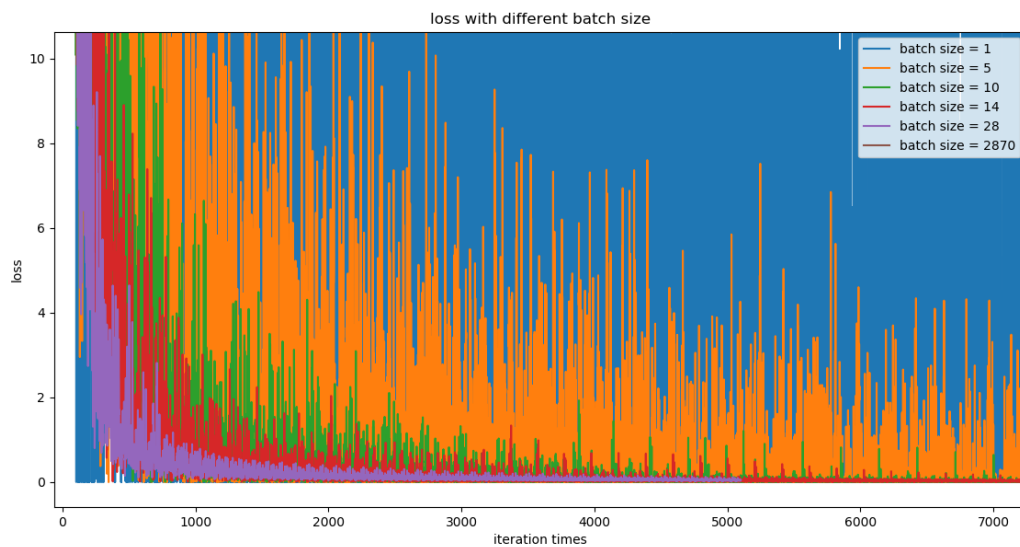
batch size

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。

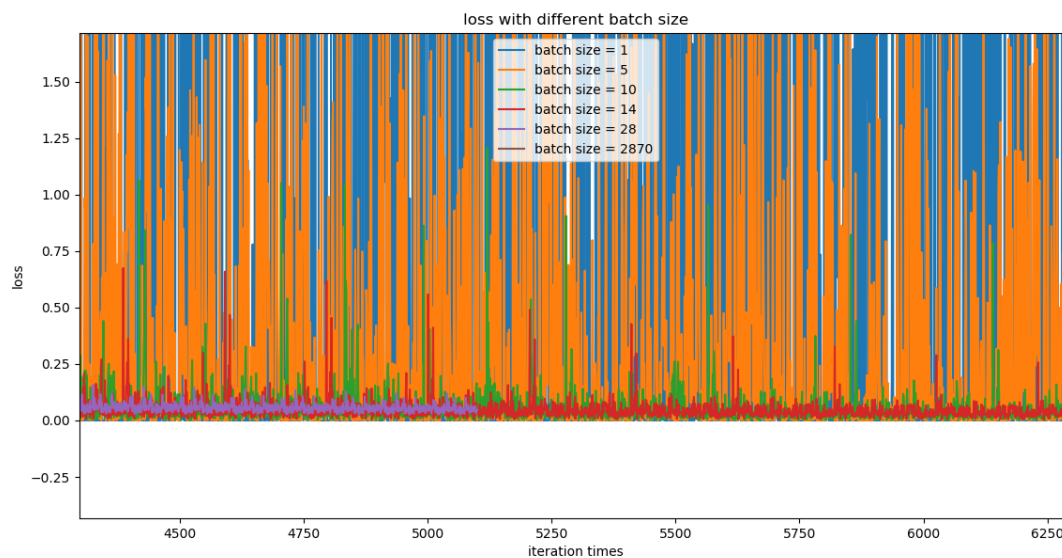


batch size 为 1 时更新得很快，却很难收敛。5 的时候，虽然震荡快，但是最后可以收敛。

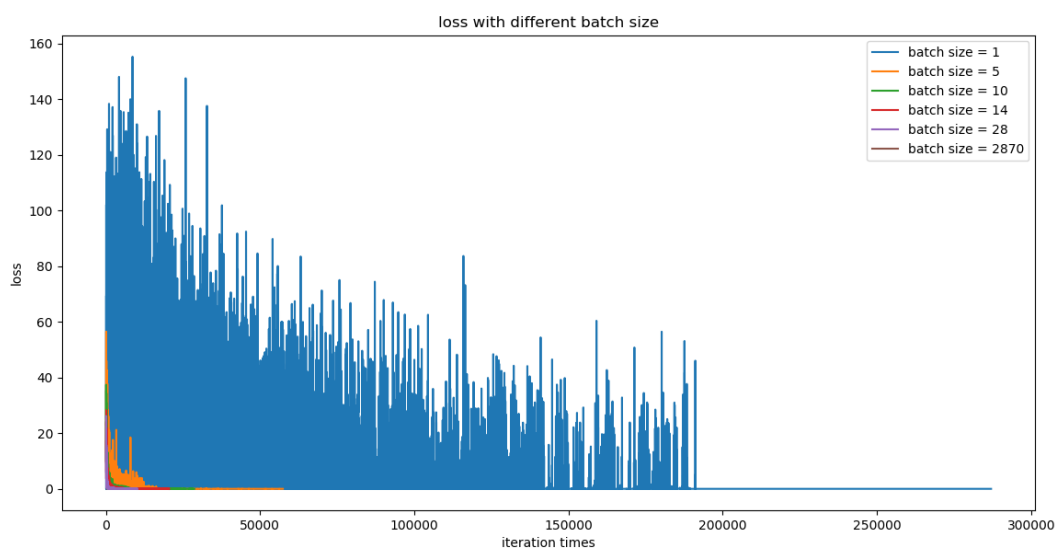


```
epoch 49
hit rate = 0.7773109243697479 with batch size = 1
hit rate = 0.8025210084033614 with batch size = 5
hit rate = 0.7969187675070029 with batch size = 10
hit rate = 0.803921568627451 with batch size = 14
hit rate = 0.7997198879551821 with batch size = 28
hit rate = 0.2647058823529411 with batch size = 2870
```

28 的效果不错，就是迭代次数少了点，14 的效果也不错。

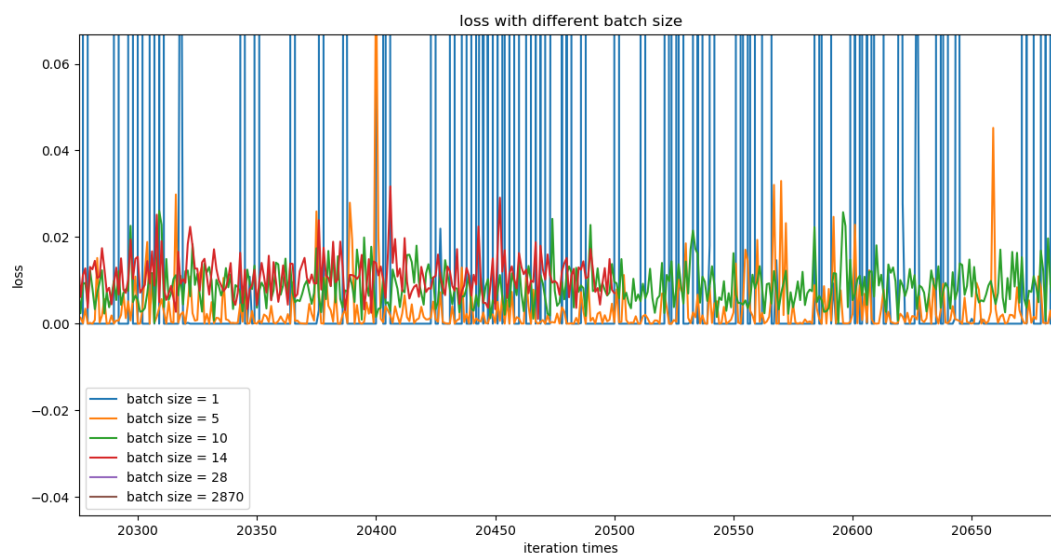


当我迭代 100 次之后，batch size 为 1 的网络收敛了！



```
hit rate = 0.7997198879551821 with batch size = 28
hit rate = 0.2997198879551821 with batch size = 2870
epoch 99
hit rate = 0.7913165266106442 with batch size = 1
hit rate = 0.7969187675070029 with batch size = 5
hit rate = 0.8179271708683473 with batch size = 10
hit rate = 0.7983193277310925 with batch size = 14
hit rate = 0.8011204481792717 with batch size = 28
hit rate = 0.3081232492997199 with batch size = 2870
```

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。

```
epoch 49
hit rate = 0.7366946778711485 with weight = -0.5 bias = -0.5
hit rate = 0.7282913165266107 with weight = -0.5 bias = -0.4
hit rate = 0.7464985994397759 with weight = -0.5 bias = -0.30000000000000004
hit rate = 0.7282913165266107 with weight = -0.5 bias = -0.2
hit rate = 0.7324929971988796 with weight = -0.5 bias = -0.1
hit rate = 0.7366946778711485 with weight = -0.5 bias = 0.0
hit rate = 0.7619047619047619 with weight = -0.5 bias = 0.1
hit rate = 0.742296918767507 with weight = -0.5 bias = 0.2
hit rate = 0.7633053221288515 with weight = -0.5 bias = 0.30000000000000004
hit rate = 0.7661064425770303 with weight = -0.5 bias = 0.4
hit rate = 0.7647058823529411 with weight = -0.5 bias = 0.5
>>>
```

bias 为 0.1、0.3、0.4、0.5 的时候不错。

接下来，迭代 100 次。

```
epoch 99
hit rate = 0.7535014005602241 with weight = -0.5 bias = -0.5
hit rate = 0.7507002801120448 with weight = -0.5 bias = -0.4
hit rate = 0.7549019607843137 with weight = -0.5 bias = -0.30000000000000004
hit rate = 0.742296918767507 with weight = -0.5 bias = -0.2
hit rate = 0.7450980392156863 with weight = -0.5 bias = -0.1
hit rate = 0.7492997198879552 with weight = -0.5 bias = 0.0
hit rate = 0.7366946778711485 with weight = -0.5 bias = 0.1
hit rate = 0.7450980392156863 with weight = -0.5 bias = 0.2
hit rate = 0.7549019607843137 with weight = -0.5 bias = 0.30000000000000004
hit rate = 0.7563025210084033 with weight = -0.5 bias = 0.4
hit rate = 0.7759103641456583 with weight = -0.5 bias = 0.5
>>>
```

发现 bias 取 0.5 效果很好。

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

代码：image_network_weight.py

```
epoch 99
hit rate = 0.7212885154061625 with weight = -0.5 bias = 0.5
hit rate = 0.7759103641456583 with weight = -0.4 bias = 0.5
hit rate = 0.7703081232492998 with weight = -0.30000000000000004 bias = 0.5
hit rate = 0.7913165266106442 with weight = -0.2 bias = 0.5
hit rate = 0.803921568627451 with weight = -0.1 bias = 0.5
hit rate = 0.8053221288515406 with weight = 0.0 bias = 0.5
hit rate = 0.8053221288515406 with weight = 0.1 bias = 0.5
hit rate = 0.7927170868347339 with weight = 0.2 bias = 0.5
hit rate = 0.7787114845938375 with weight = 0.30000000000000004 bias = 0.5
hit rate = 0.7717086834733894 with weight = 0.4 bias = 0.5
hit rate = 0.7521008403361344 with weight = 0.5 bias = 0.5
>>>
```

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，

```

epoch 99
hit rate = 0.8067226890756303 with learning rate = 0.01
hit rate = 0.7983193277310925 with learning rate = 0.03
hit rate = 0.8137254901960784 with learning rate = 0.1
hit rate = 0.7913165266106442 with learning rate = 0.3

```

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，

```

epoch 99
hit rate = 0.7983193277310925 with learning rate = 0.01
hit rate = 0.8095238095238095 with learning rate = 0.03
hit rate = 0.8067226890756303 with learning rate = 0.1
hit rate = 0.7843137254901961 with learning rate = 0.3
将会调出4中最好的一个！
最佳命中率是 0.8095238095238095

```

0.03 好像也不错，而 0.1 还是比较稳定的。