Experiment 4 Neural Networks and Back Propagation

一、 Principle and Theory

The goal of any supervised learning algorithm is to find a function that best maps a set of inputs to its correct output. An example would be a simple classification task, where the input is an image of an animal, and the correct output would be the name of the animal. For an intuitive example, the first layer of a Neural Network may be responsible for learning the orientations of lines using the inputs from the individual pixels in the image. The second layer may combine the features learned in the first layer and learn to identify simple shapes such as circles. Each higher layer learns more and more abstract features such as those mentioned above that can be used to classify the image. Each layer finds patterns in the layer below it and it is this ability to create internal representations that are independent of outside input that gives multi-layered networks their power. The goal and motivation for developing the back-propagation algorithm was to find a way to train a multi-layered neural network such that it can learn the appropriate internal representations to allow it to learn any arbitrary mapping of input to output. Mathematically, a neuron's network function $f(x)$ is defined as a composition of other functions $gi(x)$ which can further be defined as a composition of other functions. This can be conveniently represented as a network structure, with arrows depicting the dependencies between

variables. A widely used type of composition is the nonlinear weighted sum, where:

$$f(x) = \left(\sum_i w_i g_i(x)\right)$$

Where *K* (commonly referred to as the activation function) is some predefined function, such as the hyperbolic tangent. It will be convenient for the following to refer to a collection of functions *gi* as simply a vector *g = (g1, g2, ..., gn)*. Back-propagation requires a known, desired output for each input value in order to calculate the loss function gradient. It is therefore usually considered to be a supervised learning method The squared error function is:

$$E = \frac{1}{2}(t - y)^2$$

where *E* is the squared error, *t* is the target output for a training sample, and *y* is the actual output of the output neuron. For each neuron *j*, its output *oj* is defined as

$$o_j = \varphi(net_j) = \varphi\left(\sum_{k=1}^{n} w_{kj} x_k\right)$$

The input net to a neuron is the weighted sum of outputs *ok* of previous neurons. If the neuron is in the first layer after the input layer, the *ok* of the input layer are simply the inputs *xk* to the network. The number of input units to the neuron is *n*. The variable *wij* denotes the weight between neurons *i* and *j*. The activation function *φ* is in general non-linear

and differentiable. A commonly used activation function is the logistic function, e.g.:

$$\varphi(z) = \frac{1}{1 + e^{-z}}$$

which has a nice derivative of:

$$\frac{\partial \varphi}{\partial z} = \varphi(1 - \varphi)$$

Calculating the partial derivative of the error with respect to a weight *wij* is done using the chain rule twice:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial o_j} \frac{\partial o_j}{\partial net_j} \frac{\partial net_j}{\partial w_{ij}}$$

We can finally yield:

$$\frac{\partial E}{\partial w_{ij}} = \delta_j x_i$$

With

$$\delta_j = \frac{\partial E}{\partial o_j} \frac{\partial o_j}{\partial net_j} = \begin{cases} (o_j - t_j)\varphi(net_j)\left(1 - \varphi(net_j)\right) & if\ j\ is\ an\ output\ neuron \\ (\sum_{l \in L} \delta_l w_{jl})\varphi(net_j)\left(1 - \varphi(net_j)\right) & if\ j\ is\ an\ inner\ neuron \end{cases}$$

二、**Objective**

The goals of the experiment are as follows:

(1) To understand how to build a neural network for a classification problem.

(2) To understand how the back-propagation algorithm is used for training a given a neural network.

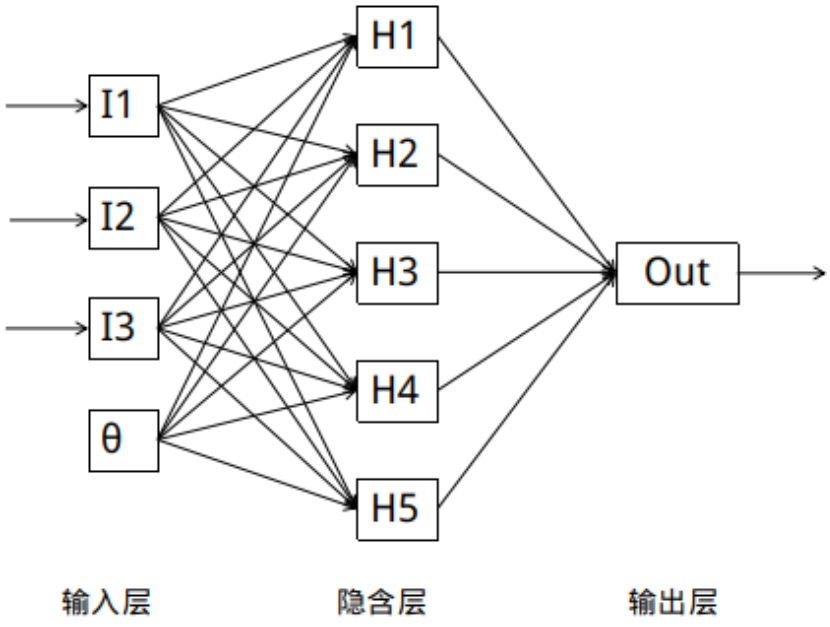
(3) To understand the limitation of the neural network model (e.g., the local minimum).

(4) To understand how to use back-propagation in Autoencoder

三、**Contents and Procedure**

(1) Given a dataset for classification, (E.g., Iris, Pima Indian and Wisconsin Cancer from the UCI ML Repository). Build a multi-layer neural network (NN) and train the network using the BP algorithm. We can start with constructing a NN with only one hidden layer.

三层 BP 神经网络由输入层，隐含层与输出层组成。通常输入层神经元的个数与特征数相关，输出层的个数与类别数相同，隐含层的层数与神经元数可以自定义。



每个隐含层和输出层神经元输出与输入的函数关系为:

$$I_j = \sum_i W_{ij} O_i$$

$$O_j = sigmod(I_l) = \frac{1}{1 + e^{-I_l}}$$

其中 Wij 表示神经元 i 与神经元 j 之间连接的权重，Oj 代表神经元 j 的输出，sigmod 函数是神经元的激励函数(activation function)。

```
#隐含层的输入
        for j in range(self.hidden_n):
            total = 0.0
            for i in range(self.input_n):
                total += self.input_cells[i] * self.input_weights[i][j]
            self.hidden_cells[j] = sigmoid(total)
```

我们将 n 个特征依次送入输入神经元，隐含层神经元获得输入层的输出并计算自己输出值，输出层的神经元根据隐含层输出计算出回归值。

BP 神经网络的训练过程即是根据前馈得到的预测值和参考值比较， 根据误差调整连接权重 Wij 的过程，训练过程称为反向传播过程(BackPropagation)。

首先计算输出层的误差：

$$E_j = sigmod'(O_j) * (T_j - O_j) = O_j(1 - O_j)(T_j - O_j)$$

其中 Ej 代表神经元 j 的误差，Oj 表示神经元 j 的输出，Tj 表示当前训练样本的参考输出，sigmod′(x)是 sigmod 函数的一阶导数。

计算隐含层误差：

$$E_j = sigmod'(O_j) * \sum_k E_k W_{jk} = O_j(1 - O_j) \sum_k E_k W_{jk}$$

```
# hidden layer 误差
hidden_deltas = [0.0] * self.hidden_n
for h in range(self.hidden_n):
 error = 0.0
 for o in range(self.output_n):
        error += output_deltas[o] * self.output_weights[h][o]
 hidden_deltas[h] = sigmod_derivate(self.hidden_cells[h]) * error
```

隐含层输出不存在参考值， 使用下一层误差的加权和代替(Tj−Oj)

计算完误差后就可以更新 Wij 和 θ j：

$$W_{ij} = W_{ij} + \lambda E_j O_i$$

其中 λ 称为学习率，一般在(0，0.1)区间上取值。

为了加快学习的效率，我们引入矫正矩阵，矫正矩阵记录上一次反向传播过程中的 EjOi 值， 这样 Wj 更新公式变为:

$$W_{ij} = W_{ij} + \lambda E_j O_i + \mu C_{ij}$$

μ 是矫正率。随后更新矫正矩阵：

$$C_{ij} = E_j O_i$$

```
        # 更新 output weights
        for h in range(self.hidden_n):
            for o in range(self.output_n):
                change = output_deltas[o] * self.hidden_cells[h]
                self.output_weights[h][o] += learn * change + correct *
self.output_correction[h][o]
                self.output_correction[h][o] = change
```

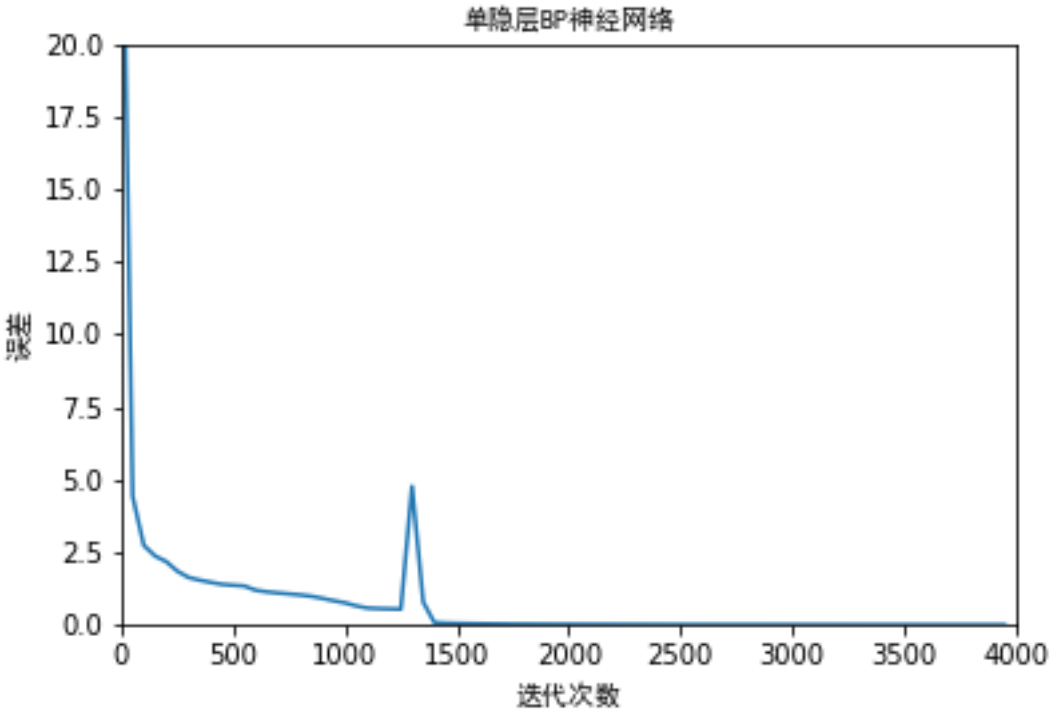
最简单的训练终止条件为设置最大迭代次数，我们将数据集迭代 4000 次后终止训练。

self.setup(9, 20, 2)　#输入层神经元 9，隐层神经元 20，输出层神经元 2

self.train(cases[:400], labels, 4000, 0.05, 0.1)　　#迭代次数 4000 ，学习率 0.05 ，矫正率 0.1

我们使用的数据集是 UCI 数据库中的 Breast-Cancer 数据集。数据集中共有 699 条信息，数据集中有 16 处缺失值，缺失值使用"?"表示。因为缺失的数据不多，所以我们暂时先采用丢弃带有“？”的数据。

处理后的数据集 cancer.csv 中共有 683 条信息，文件有 11 个列，第 1 个列为 id 号，第 2-10 列为特征，11 列为标签（0 为良性、1 为恶性）。



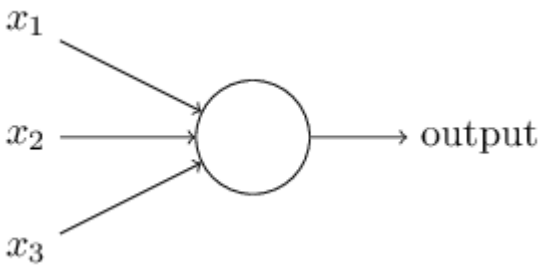
预测：result = [264, 19]

error = 0.06713780918727916

随着迭代次数的增加，训练集错误率逐渐下降，4000 次迭代后，错误率小于 1%。而预测集的错误率在 6.7%左右。

(2) Compare the NN results to the results of Perceptron, which model is better in terms of efficiency and accuracy?

在“感知器”中，有两个层次。分别是输入层和输出层。输入层里的“输入单元”只负责传输数据，不做计算。输出层里的“输出单元”则需要对前面一层的输入进行计算。



感知器中的权值是通过训练得到的。

我们建立一个感知器模型对 breast-cancer 数据集分类：

per = Perceptron()

per.fit(X_train, y_train)

预测结果：

[4 2 4 4 2 2 2 4 2 2 2 2 4 2 4 4 4 4 2 2 4 4 2 4 4 2 2 4 4 4 4 4 4 4 2
4 4 4 4 4 2 4 2 2 4 2 2 4 4 2 2 2 4 2 2 2 2 2 4 4 2 2 2 4 2 2 2 4 2 2 4
2 2 2 4 4 2 2 2 4 2 2 2 4 2 4 2 4 4 2 2 2 2 4 4 2 2 2 4 2 2 4 2 2 2 2 2 4
2 2 2 2 2 2 4 2 2 4 4 2 4 2 2 2 4 2 2 4 4 2 4 4 2 2 2 2 4 2 4 2 4 2 2 2 2
2 4 4 2 4 4 2 4 2 2 2 2 4 4 4 2 4 2 2 4 2 4 4]

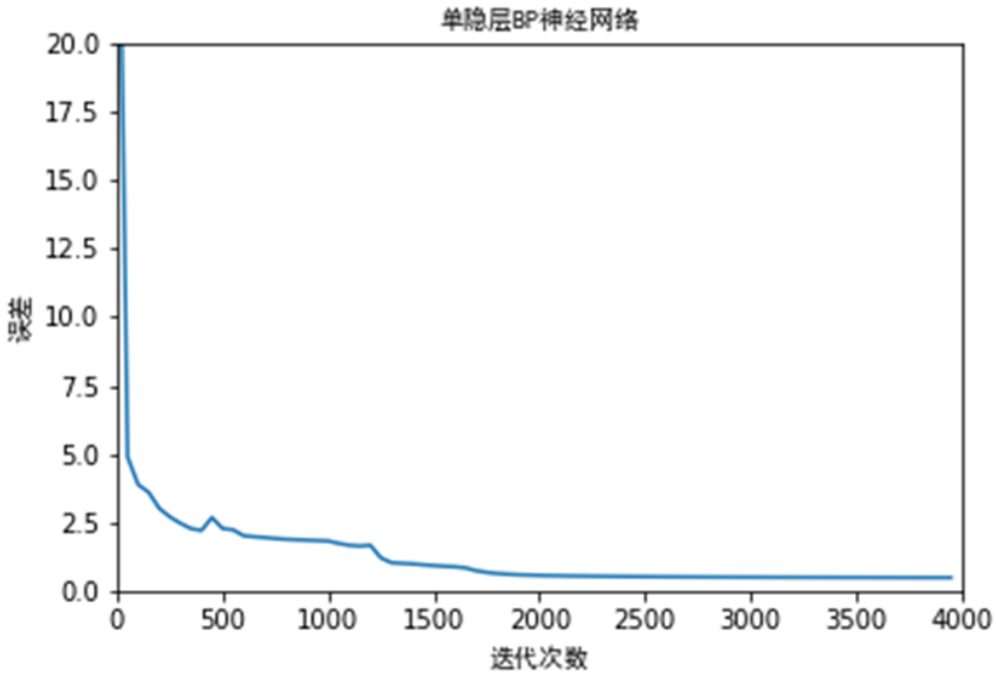
预测准确率：

Accuracy of Perceptron Classifier: 0.9883040935672515

由此可见对于线性可分的问题，即线性超平面能将它们分开，感知器算法在 efficiency 和 accuracy 方面都有非常好的效果。神经网络对于此类问题在达到同样的正确率需要更多的训练时间。但是对于非线性可分的问题，用线性超平面无法划分，则感知器算法无法解决，需要神经网络算法。因此对于 NN 和 Perceptron 的优越性，要根据具体的划分问题来选择。

(3) Whether the performance of the model is heavily influenced by different parameters settings (e.g., the learning rate α)?

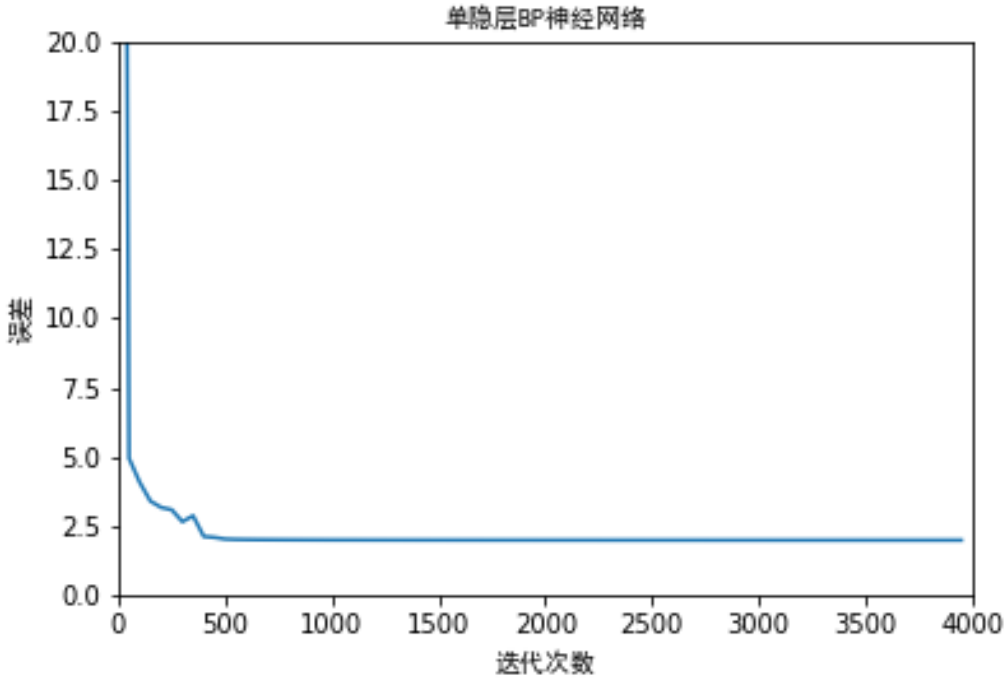
self.train(cases[:400], labels, 4000, 0.001, 0.1) #学习率α=0.001



result = [262, 21]

error = 0.07420494699646643 错误率：7.4%

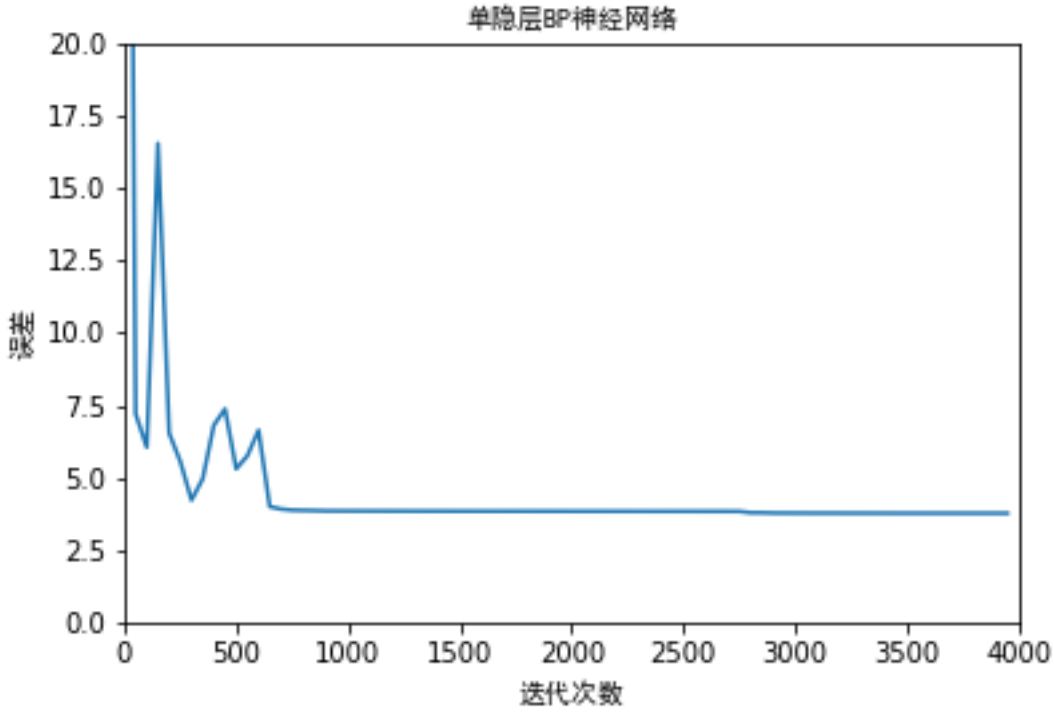
self.train(cases[:400], labels, 4000, 0.1, 0.1)　#学习率α=0.1



result = [271, 12]

error = 0.04240282685512368　　错误率：4.2%

self.train(cases[:400], labels, 4000, 0.3, 0.1)　　#学习率α=0.3

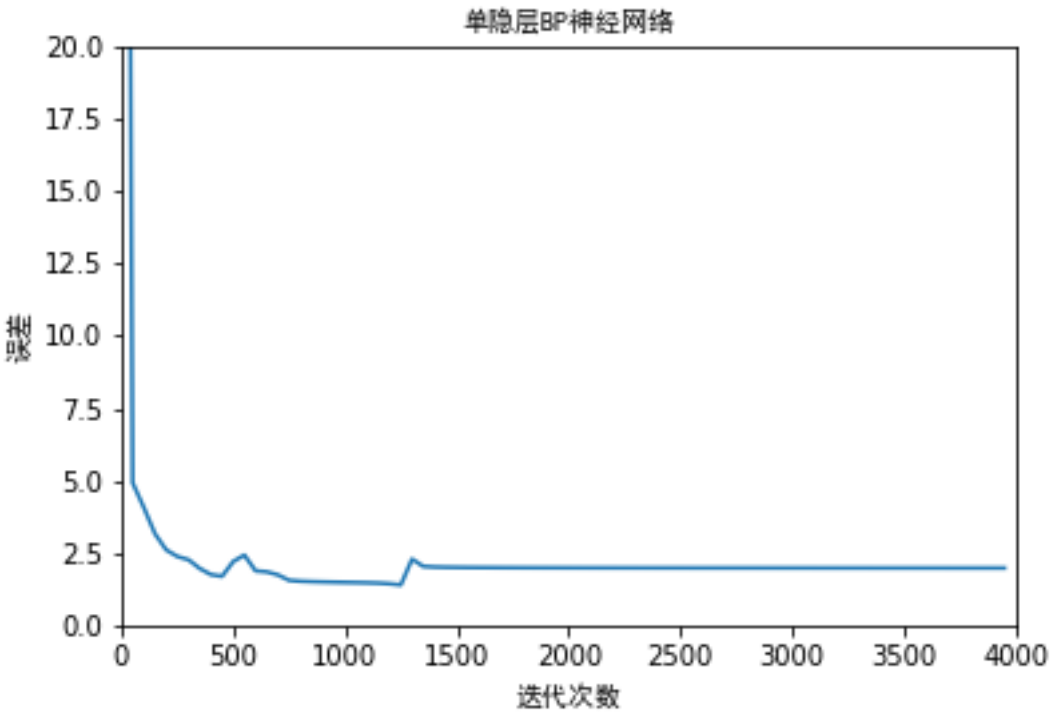


result = [275, 8]

error = 0.028268551236749116　　错误率:2.8%

由此可见，如果学习速率太小，则会使收敛过慢，如果学习速率太大，则会导致代价函数振荡。

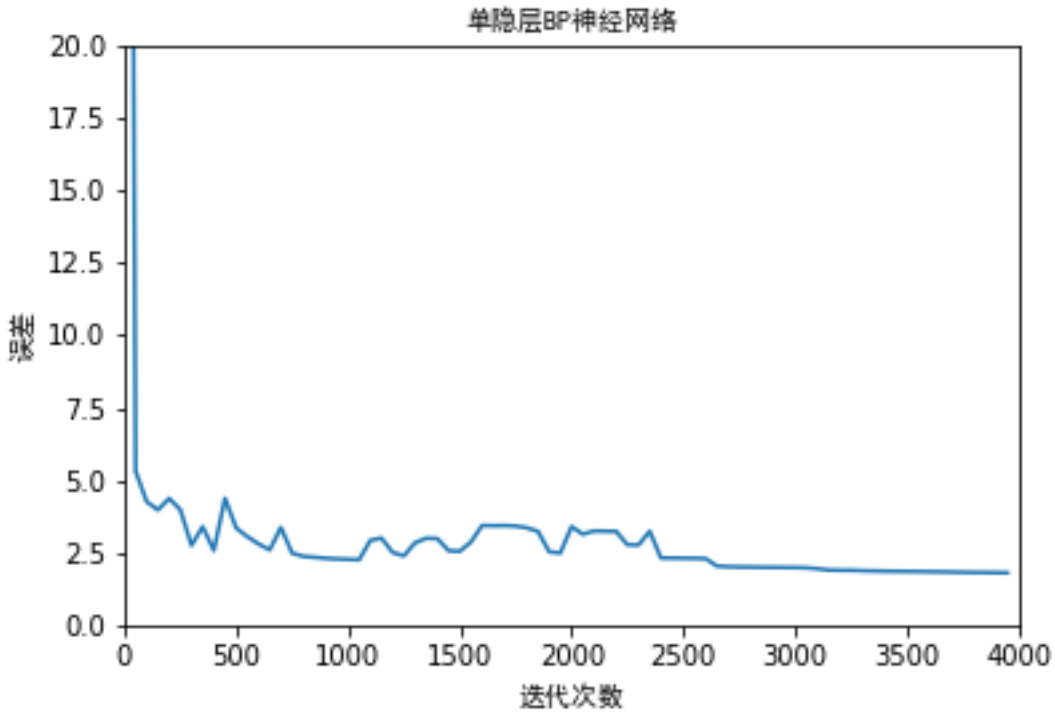
self.train(cases[:400], labels, 4000, 0.05, 0.1)　　矫正率：μ=0.1



result = [264, 19]

error = 0.06713780918727916　错误率：6.7%

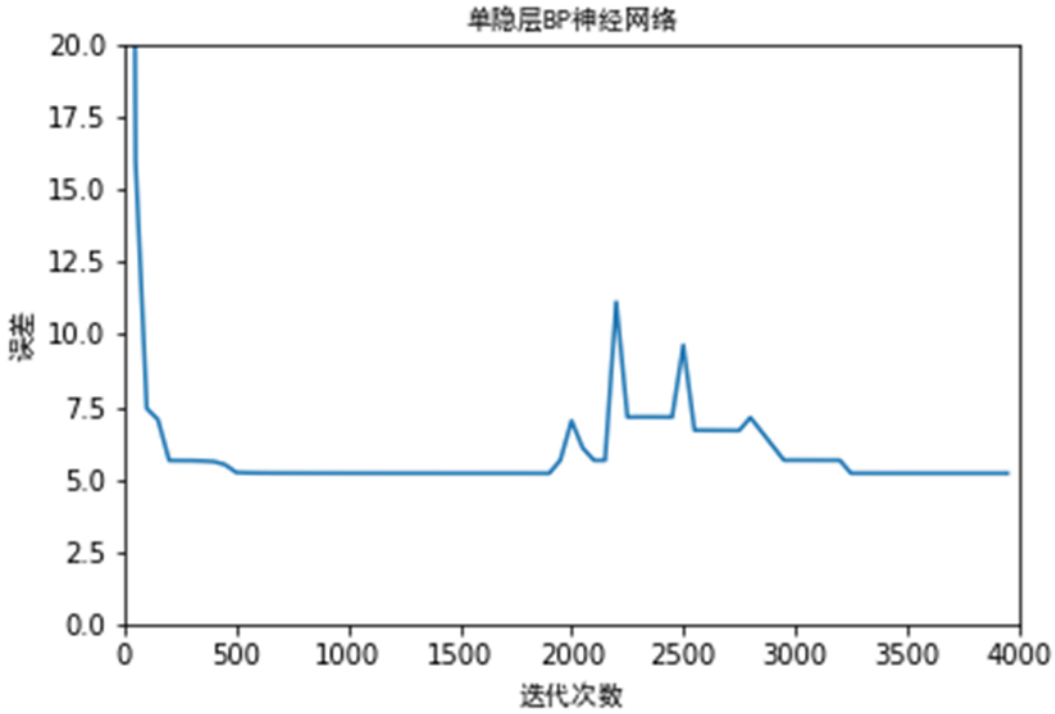
self.train(cases[:400], labels, 4000, 0.05, 0.2)　　矫正率： μ=0.2



result = [262, 21]

error = 0.07420494699646643　　错误率：7.4%

self.train(cases[:400], labels, 4000, 0.05, 0.5)　　矫正率： μ=0.5



result = [274, 9]

error = 0.03180212014134275 错误率：3.2%

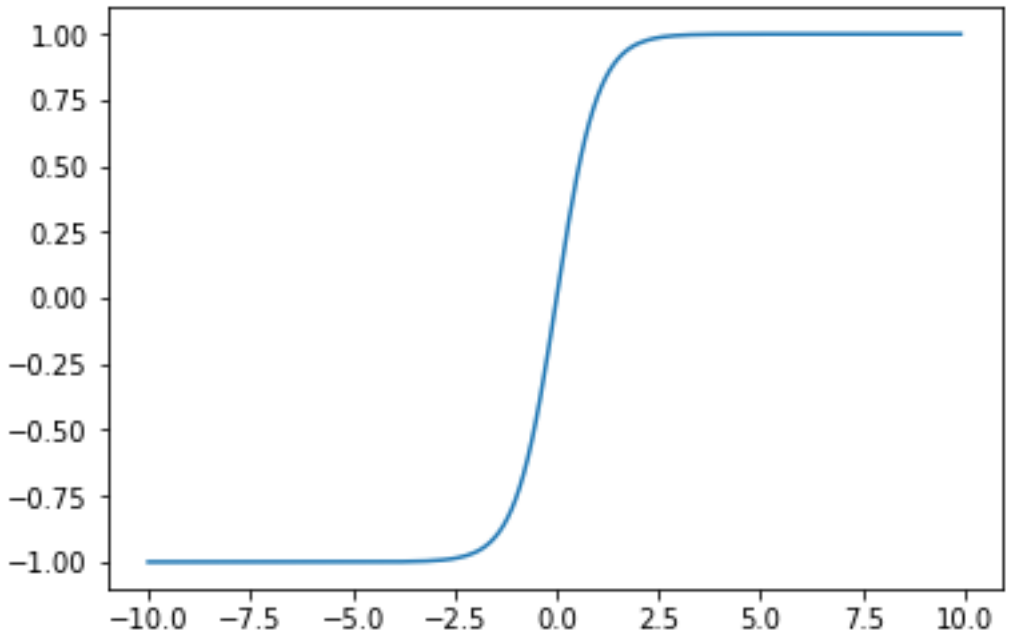
由此可见，矫正率很大的话会引起振荡而且收敛速度过慢的问题，应根据实际情况来选择合适的最佳学习率与矫正率。

(4) How the efficiency and accuracy will be influenced by different activation functions and more hidden layers.

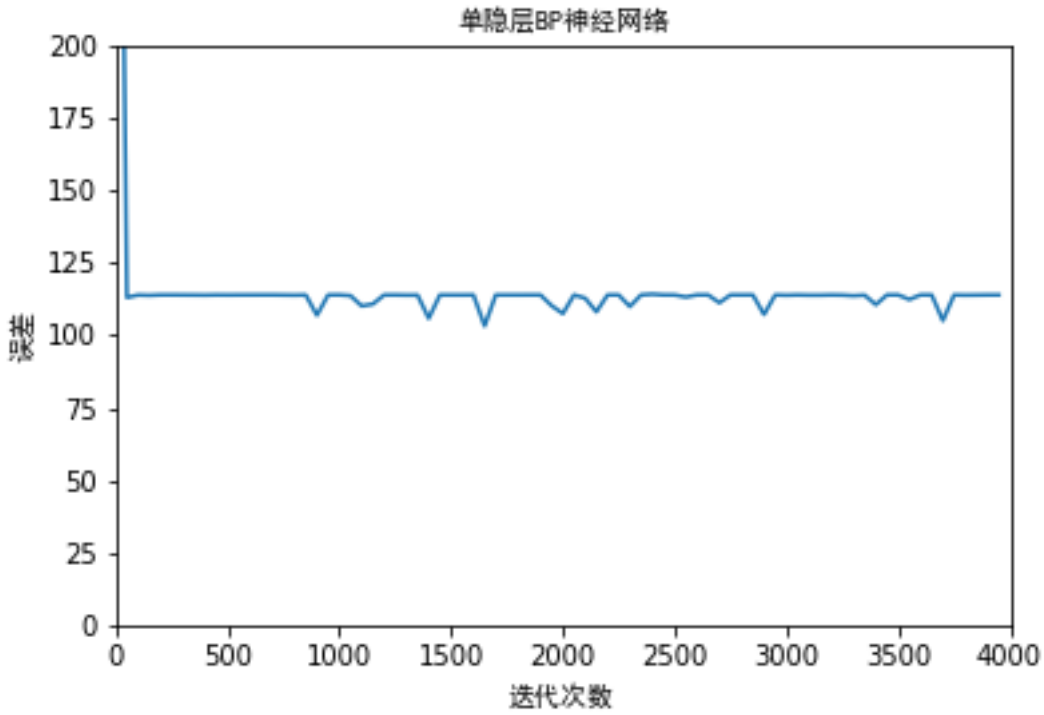
tanh 函数：f(z) = tanh(z)

tanh 导数：f(z)’ = 1 − (f(z))^2

```
def tanh(x):                #tanh 函数
    return (math.exp(x) - math.exp(-x)) / (math.exp(x) + math.exp(-x))
def tanh_derivate(x):    #tanh 导数
        return    1 - x*x
```



以 tanh 函数为激活函数的 BP 神经网络：



result = [67, 216]

error = 0.7632508833922261　　错误率：76%

tanh 函数也称为双切正切函数，取值范围为 [-1,1]。tanh 在特征相差明显时的效果会很好，在循环过程中会不断扩大特征效果，对于本数据集的分类，tanh 函数的表现不好，测试集的错误率很高。

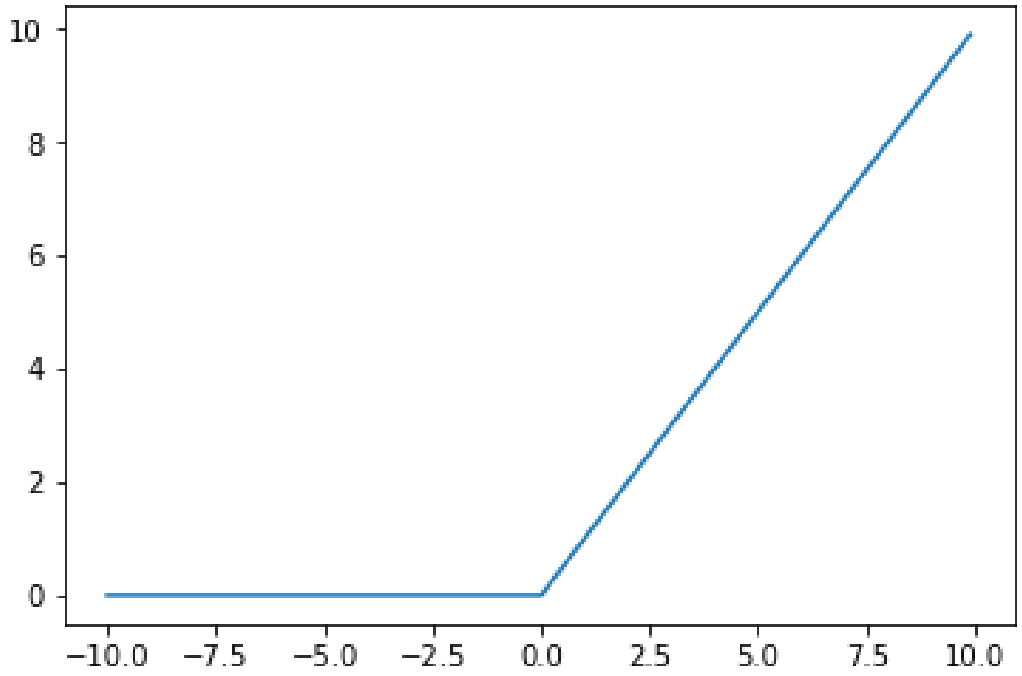
ReLU 函数计算如下：

$$\phi(x) = \max(0, x)$$

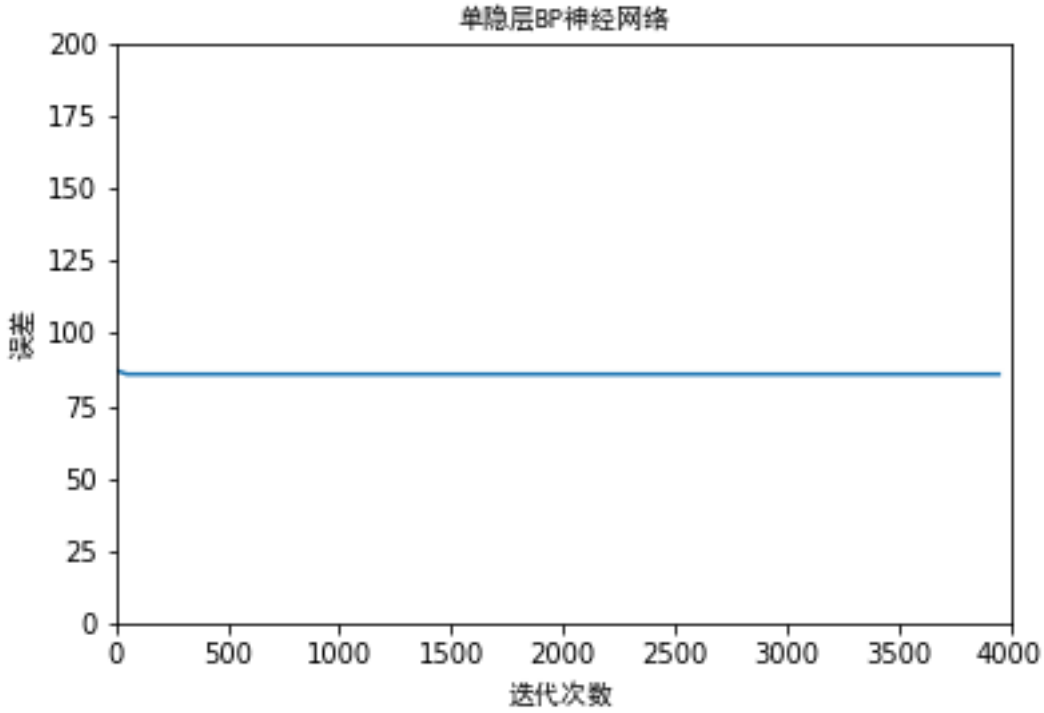
```
def relu(x):            #ReLU 函数
    return np.where(x<=0,0,x)

def relu_derivate(x): # ReLU 导数
    return  np.where(x<=0,0,1)
```

ReLU 函数：



以 ReLU 函数为激活函数的 BP 神经网络：



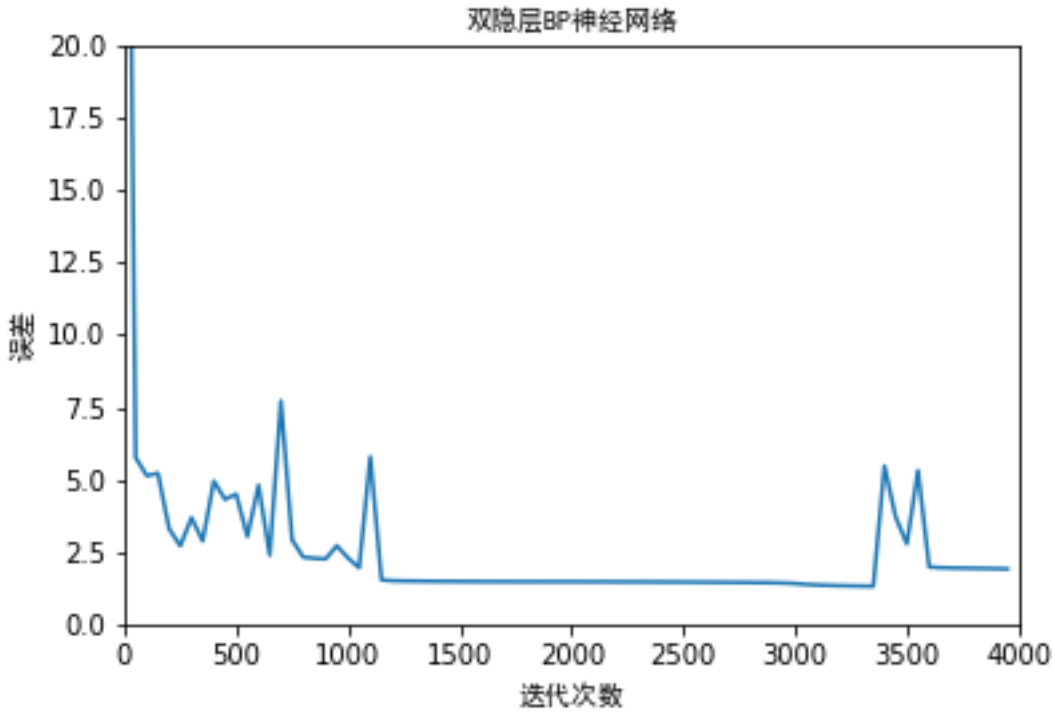
result = [216, 67]

error = 0.23674911660777384　　错误率：23%

ReLU 函数计算量小，但是在训练的时候，ReLU 单元比较脆弱并且可能“死掉”。我们要通过合理设置学习率，使这种情况的发生概率降低。对于本数据集的分类，ReLU 函数的表现不错，测试集错误率较大。

双隐层结构：

self.setup(8,20,5,2)　　#输入层 8，一隐层 20，二隐层 5，输出层 2



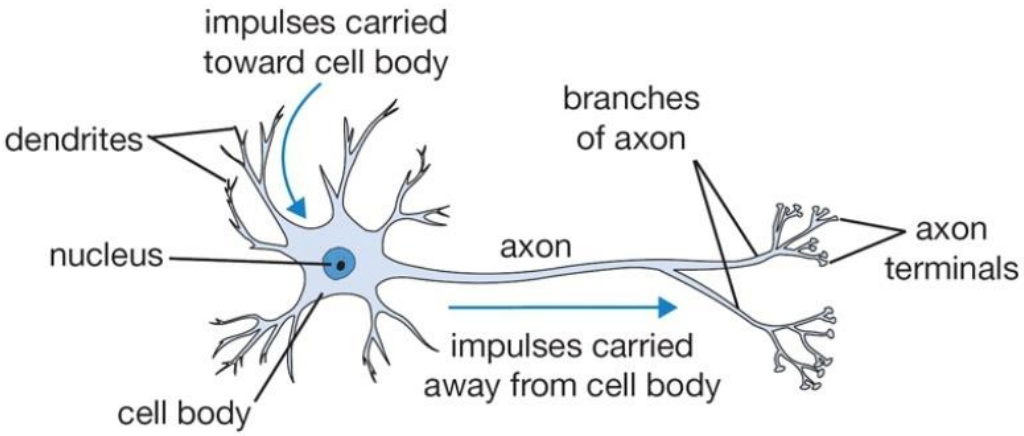
result = [268, 15]

error = 0.053003533568904596　　错误率：5.3%

4 层的 BP 神经网络由输入层，2 层隐藏层，输出层组成。4 层 BP 神经网络比 3 层 BP 神经网络，分类效果要好一些，但是训练时间会更长些。实验证明，隐层层数与神经元的数目对分类结果有很大影响，一般隐层层数与神经元的数目较多，训练效果会比较好，但也并不是越多越好，层数与神经元的数目过多的复杂模型训练效率低，容易陷入过拟合。

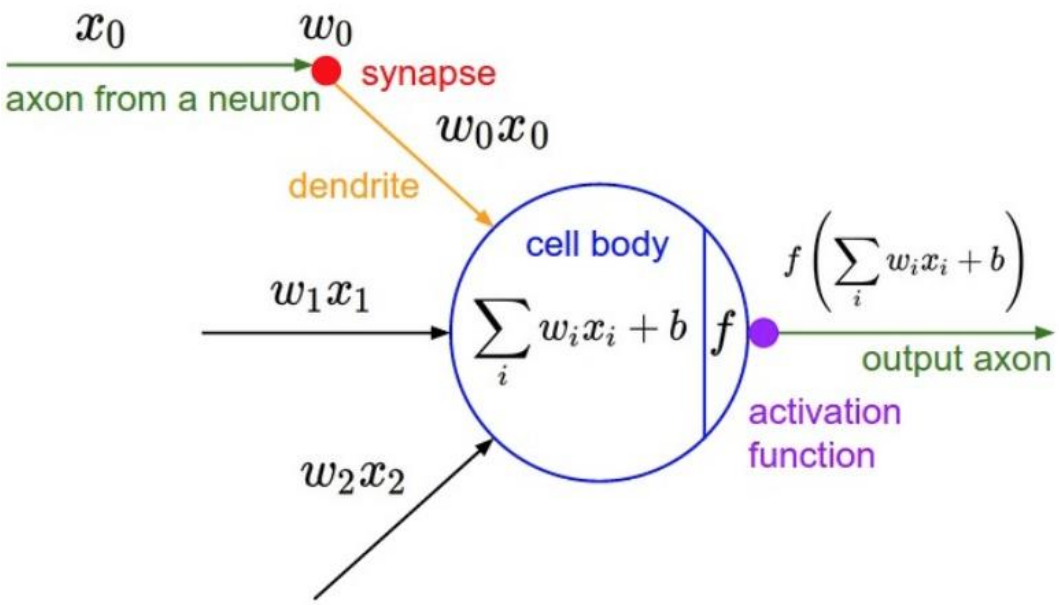
(5) Do you think that biological neural networks work in the same way as our NN model? Can you provide any discussions to support your opinions?

大脑的基本计算单位是神经元（neuron）。人类的神经系统中大约有 860 亿个神经元，它们被大约 10^14-10^15 个突触（synapses）连接起来。下图展示了一个生物学的神经元。每个神经元都从它的树突获得输入信号，然后沿着它唯一的轴突（axon）产生输出信号。轴突在末端会逐渐分枝，通过突触和其他神经元的树突相连。



在神经元的计算模型中，沿着轴突传播的信号将基于突触的突触强度，与其他神经元的树突进行乘法交互。其观点是，突触的强度（也就是权重 w），是可学习的且可以控制一个神经元对于另一个神经元的影响强度（还可以控制影响方向：使其兴奋（正权重）或使其抑制（负权重））。下图展示了常用的数学模型。

在基本模型中，树突将信号传递到细胞体，信号在细胞体中相加。如果最终之和高于某个阈值，那么神经元将会激活，向其轴突输出一个峰值信号。在计算模型中，我们假设峰值信号的准确时间点不重要，是激活信号的频率在交流信息。基于这个速率编码的观点，将神经元的激活率建模为激活函数（activation function），它表达了轴突上激活信号的频率。

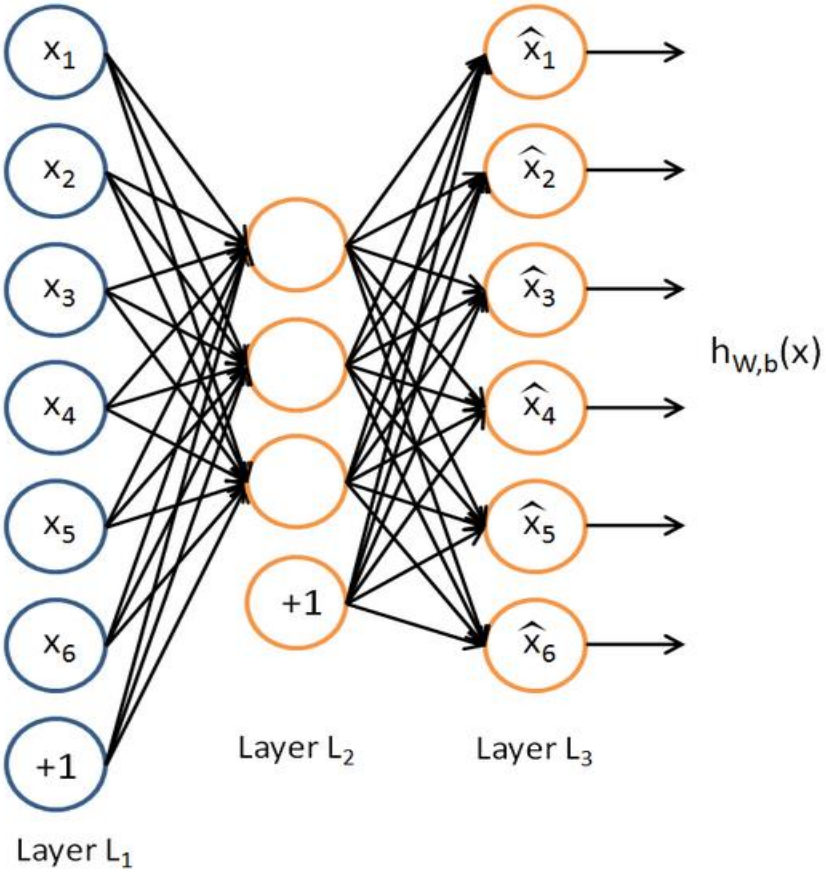


使用生物神经网络后，大脑中无数的神经元之间实现了互连，之后便可进行学习。随着大脑不断暴露在新的刺激之下，这些神经元改变了它们的互连结构，建立了新的连接、加强了现有连接、删除了那些未使用的连接。越多的重复已给定任务，神经性的连接就越强，直至该任务重复学习无数次。

人工神经网络旨在以一种抽象的方式模拟神经元处理新刺激的过程，但是规模变得更小、更简单。人工神经网络由互连神经元的层构成，这种互连神经元可以接收一组输入和一组权重。然后进行数学操作，并将结果以“激活码”的方式输出，这与生物神经元中的突触十分相似。

(6) Using the BP algorithm to train an autoencoder. (We will train the simplest autoencoder which is a feedforward, non-recurrent neural net with an input layer, an output layer and one or more hidden layers connecting them. The difference is that, for an autoencoder, the output layer has equally many nodes as the input layer, and instead of training it to predict some target value y given inputs *x*, an autoencoder is trained to reconstruct its own inputs *x*.

自编码器是只有一层或多层隐层，输入和输出具有相同节点数的神经网络。自编码器的目的是求函数$h_{W,b}(x) \approx x$。



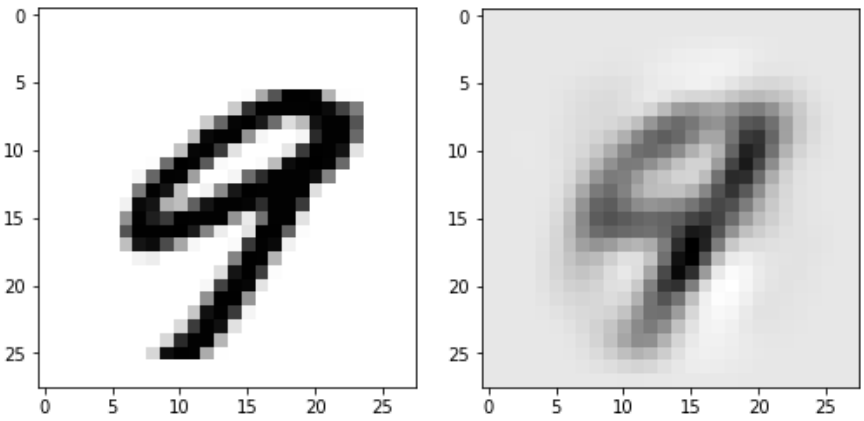
搭建一个自动编码器需要完成下面三样工作：搭建编码器，搭建解码器，设定一个损失函数，用以衡量由于压缩而损失掉的信息。

```
# 图片输入
input_layer = tf.placeholder('float', [None, 784])
# 隐层的输入
layer_1 = tf.nn.sigmoid (tf.add(tf.matmul(input_layer,
hidden_layer_vals['weights']),hidden_layer_vals['biases']))
# 输出层的输入
output_layer = tf.matmul(layer_1,output_layer_vals['weights']) +
output_layer_vals['biases']
# 定义 cost function
meansq = tf.reduce_mean(tf.square(output_layer - output_true))
# 定义 optimizer
learn_rate = 0.1
optimizer = tf.train.AdagradOptimizer(learn_rate).minimize(meansq)
```

我们选用 MNIST 数据集作为自编码器的学习资料，MNIST 数据集来自美国国家标准与技术研究所。训练集 (training set) 由来自 250 个不同人手写的数字构成。测试集(test set) 也是同样比例的手写数字数据。训练数据集包含 60,000 个样本，测试数据集包含 10,000 样本。在 MNIST 数据集中的每张图片由 28 x 28 个像素点构成，每个像素点用一个灰度值表示。在这里，我们将 28 x 28 的像素展开为一个一维的行向量，这些行向量就是图片数组里的行(每行 784 个值, 或者说每行就是代表了一张图片)。

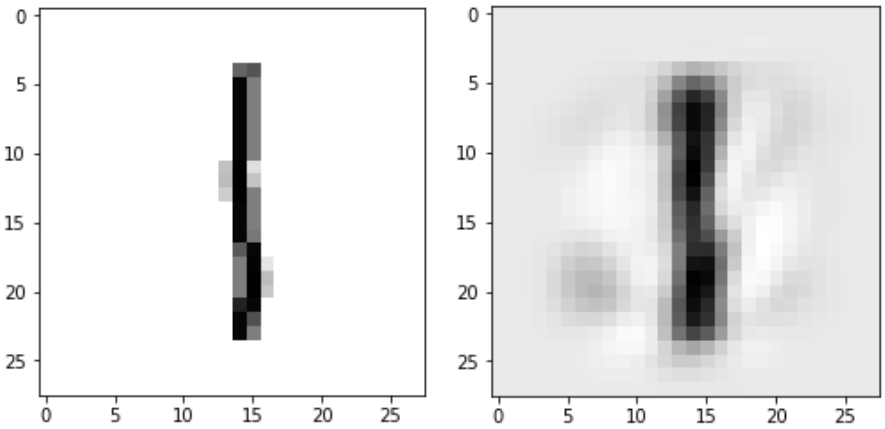
```
n_nodes_inpl = 784      #输入层节点数 784
n_nodes_hl   = 32       #隐层节点数 32
n_nodes_outl = 784      #输出层节点数 784
```

设定迭代次数 1000 次后的结果如下：（左图输入，右图输出）



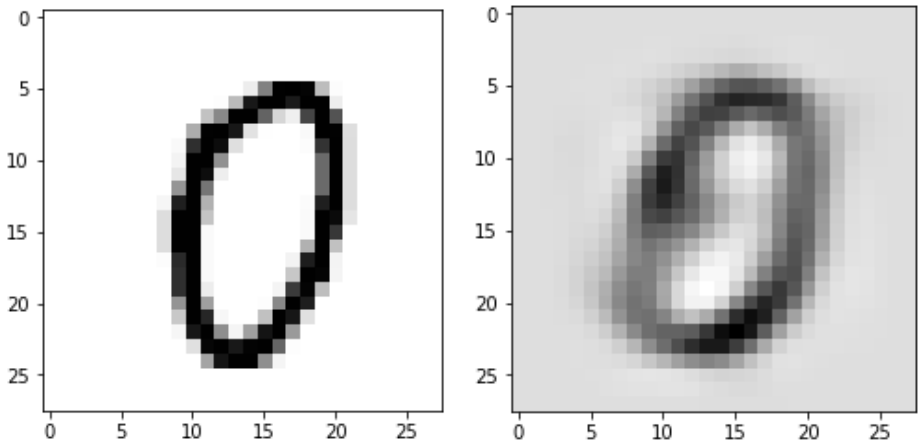
隐层编码如下：

[[1. 1. 1. 1. 1. 0. 0. 0. 1. 0. 1. 1. 0. 1. 1. 1. 1. 1.
   1. 1. 1. 1. 0. 0. 0. 0. 0. 1. 1. 1. 1. 1.]]



隐层编码如下：

[[1. 1. 1. 0. 1. 0. 0. 0. 0. 0. 0. 1. 1. 1. 1. 1. 0. 0.
   1. 0. 0. 1. 0. 1. 0. 1. 0. 0. 0. 0. 0. 0.]]



隐层编码如下：

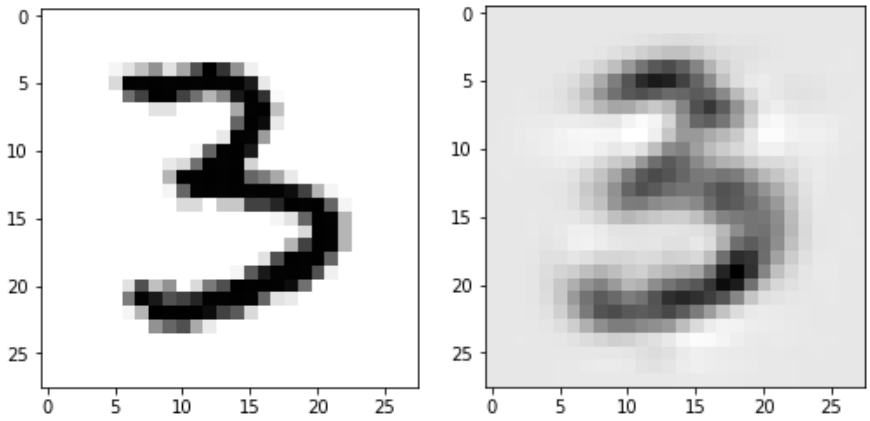
[[1. 1. 1. 1. 0. 0. 1. 1. 1. 1. 0. 1. 0. 1. 0. 0. 1. 0.
0. 1. 1. 0. 1. 1. 1. 0. 1. 1. 0. 0. 0. 0.]]

使用单隐层 32 节点的自编码器，将 784 个数据压缩为 32 个数据，我们可以看到，输出结果并不特别理想，我们可以通过增加节点数和隐层层数来获取更好的输出结果。

(7) For an autoencoder with only one hidden layer, how the number of nodes in the hidden layer influence the model performance?

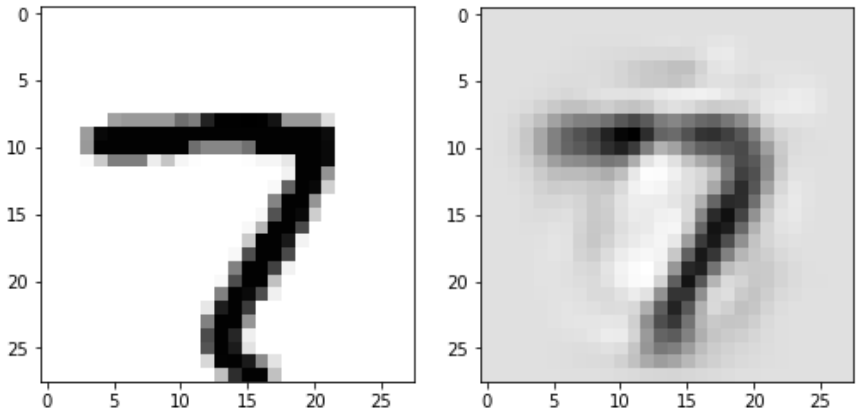
```
n_nodes_inpl = 784      #输入层节点数 784
n_nodes_hl   = 64       #隐层节点数 64
n_nodes_outl = 784      #输出层节点数 784
```

设定迭代次数 1000 次后的结果如下：（左图输入，右图输出）



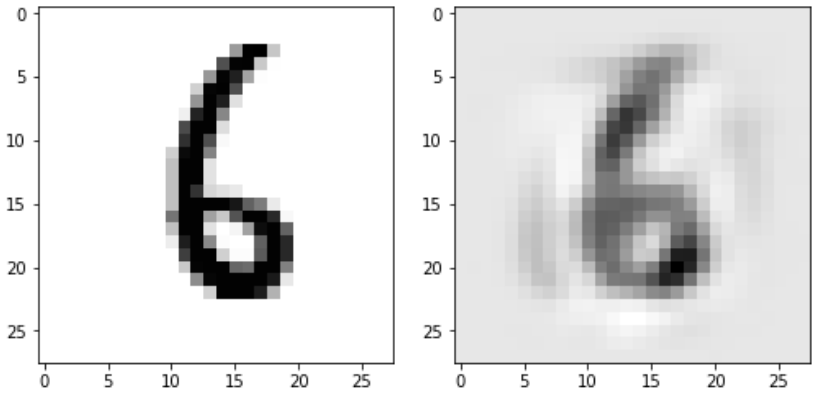
隐层编码如下：

```
[[ 0.   1.   1.   1.   1.   1.   0.   0.   1.   1.   1.   0.   0.   1.   0.   1.   1.   1.
   1.   1.   1.   1.   1.   0.   1.   0.   0.   0.   1.   1.   0.   1.   1.   0.   0.   0.
   0.   0.   1.   0.   0.   0.   0.   1.   0.   1.   0.   1.   1.   0.   1.   1.   1.   0.
   0.   1.   1.   0.   1.   0.   0.   1.   1.   1.]]
```



隐层编码如下：

```
[[ 1.  1.  0.  0.  1.  1.  1.  1.  1.  0.  0.  0.  0.  0.  0.  1.  0.  1.
   0.  0.  0.  0.  0.  1.  1.  0.  0.  1.  1.  0.  1.  0.  0.  0.  1.  0.
   0.  1.  1.  1.  0.  1.  1.  0.  1.  0.  0.  0.  0.  1.  1.  1.  1.  0.
   0.  0.  0.  0.  1.  1.  0.  0.  1.  0.]]
```



隐层编码如下：

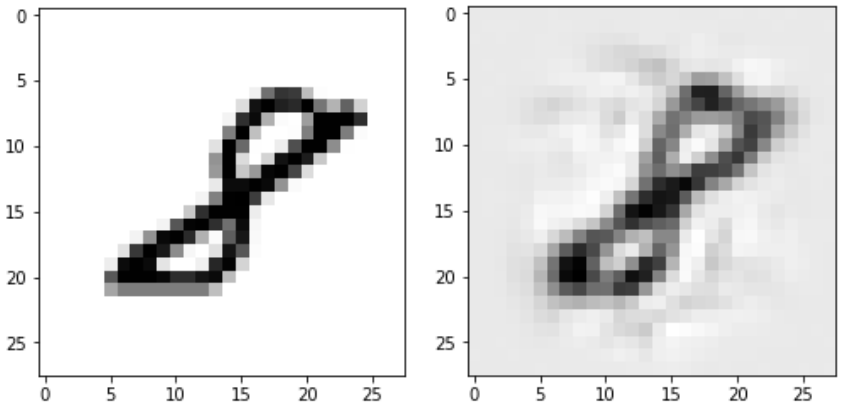
```
[[ 0.  0.  1.  1.  1.  1.  1.  0.  0.  1.  1.  1.  0.  0.  1.  1.  1.  0.
   1.  1.  1.  1.  1.  1.  1.  1.  0.  0.  0.  0.  1.  0.  1.  0.  0.  0.
   1.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  1.  0.  1.  0.  1.
   0.  1.  1.  0.  1.  0.  1.  1.  0.  0.]]
```

我们可以看到，相同迭代次数下，64 节点的单隐层和 32 节点的单隐层的自编码器的效果似乎区别不大。

n_nodes_inpl = 784　　#输入层节点数 784

n_nodes_hl　= 320　　#隐层节点数 320

n_nodes_outl = 784　　#输出层节点数 784



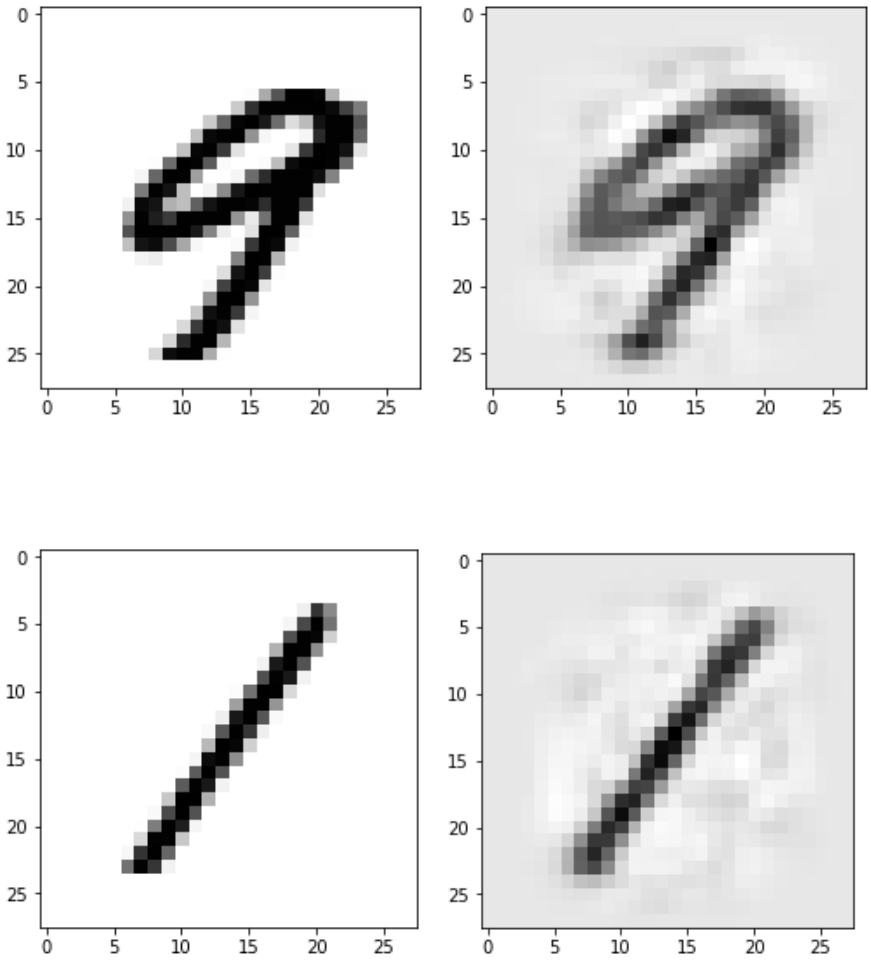
隐层编码如下：

1	1	0	1	1	0	0	1	0	0	1	1	1	0	1	1
1	0	0	1	0	1	0	1	1	1	1	1	1	0	0	0
1	0	1	0	0	0	1	1	1	1	0	0	1	1	1	1
1	0	0	1	1	0	0	0	1	1	0	0	1	0	0	0
1	1	0	1	1	1	1	0	1	1	1	0	1	0	1	1
1	1	1	1	1	0	0	0	1	1	1	0	1	1	1	0
1	0	0	0	1	0	1	1	1	0	1	0	0	0	1	1
1	0	0	0	1	0	0	1	1	1	1	1	0	0	1	1
1	0	0	0	0	0	0	1	1	0	0	0	1	1	1	1
0	0	0	1	1	0	1	1	1	1	0	1	1	1	0	0
1	1	1	0	1	0	1	1	1	1	0	0	1	1	0	1
1	1	1	1	0	1	0	1	1	1	0	0	0	0	0	1
1	0	0	1	1	0	1	1	1	1	0	1	1	0	1	1
1	0	1	1	0	1	1	1	1	0	1	0	0	1	1	1
0	1	0	0	1	0	0	1	1	0	1	0	1	0	0	0
0	1	0	1	0	1	0	1	1	1	1	0	0	1	0	0
1	1	1	0	1	1	1	1	0	1	1	0	0	0	0	1

0 1 1 0 0 1 1 0 1 0 0 0 0 0 0 1

1 1 0 1 1 1 1 1 0 1 1 0 0 1 1 0

0 1 0 0 1 1 0 1 1 1 0 1 1 0 0 0



我们可以看到，相同迭代次数下，320 节点的单隐层比 64 节点的单隐层的自编码器的效果好很多，已经非常接近原图像。我们可以得出结论，相同迭代次数下，隐层节点数越多，自编码器效果越好。