

从感知器到人工神经网络

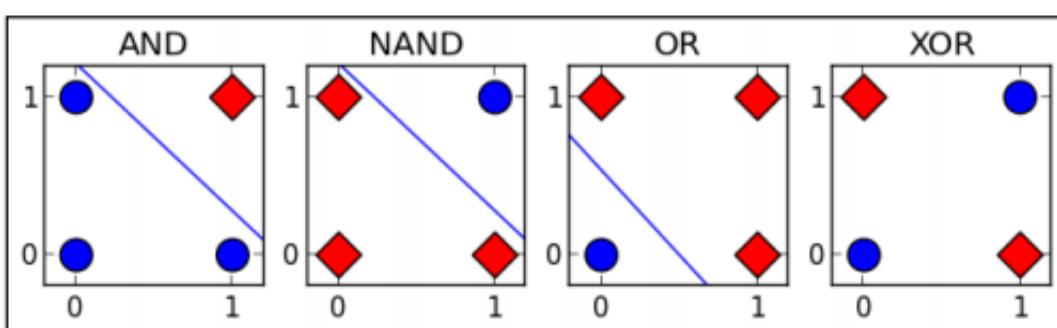
在第8章，感知器里，我们介绍了感知器，一种线性模型用来做二元分类。感知器不是一个通用函数近似器；它的决策边界必须是一个超平面。上一章里面介绍的支持向量机，用核函数修正了感知器的不足，将特征向量有效的映射到更高维的空间使得样本成为线性可分的数据集。本章，我们将介绍人工神经网络（artificial neural networks, ANN），一种用于强大的非线性回归和分类模型，用新的策略来克服感知器的缺点。

如果把感知器比喻成一个神经元，那么人工神经网络，即神经网，就是一个大脑。人脑就是由十几亿的神经元和上千亿的突触组成，人工神经网络是一种感知器或其他人工神经的有向图。这个图的边带有权重，这些权重是模型需要学习的参数。

有许多著作整本书描述人工神经网络；本章主要是对它的结构和训练方法进行介绍。目前scikit-learn的版本是0.16.1，在2014年Google Summer的项目中，多层感知器已经被作者实现，并提交在scikit-learn 0.15.1版本中，只是还没有被合并到scikit-learn。在未来的scikit-learn新版本中可能会原封不动的合并多层感知器的实现。也有一些神经网络模型的Python库，比如PyBrain (<https://github.com/pybrain/pybrain>)，Pylearn2 (<https://github.com/lisa-lab/pylearn2>)和scikit-neuralnetwork (<https://github.com/aigamedev/scikit-neuralnetwork>)等。

非线性决策边界

在第8章，感知器里，我们介绍过布尔函数如AND（与），OR（或）和NAND（与非）可以用感知器近似，而XOR（异或）作为线性不可分函数不能被近似，如下图所示：



让我们深入研究XOR来感受一下人工神经网络的强大。AND是两个输入均为1结果才为1，OR是两个输入至少有1个1结果即为1。XOR与它们不同，XOR是当两个输入中有且仅有1个1结果才为1。我们把XOR输出为1的两个输入看出是两个条件均为真。第一个条件是至少有1个输入为1，这与OR的条件相同。第二个条件是两个输入不都为1，这与NAND的条件相同。我们可以通过处理OR和NAND的输入，生成同样输出的XOR，然后用AND验证两个函数的输出是否均为1。也就是说，OR，NAND和AND可以组合生成同样结果的XOR。

下面是XOR，OR，NAND和AND四种函数有两个输入A和B时的输出真值表。从这个表我们可以检验OR，NAND和AND组合函数的输出，与同样输入的XOR输出相同：

A	B	A AND B	A NAND B	A OR B	A XOR B
0	0	0	1	0	0
0	1	0	1	1	1
1	0	0	1	1	1
1	1	1	0	1	0

A	B	A OR B	A NAND B	(A OR B) AND (A NAND B)
0	0	0	1	0
0	1	1	1	1
1	0	1	1	1
1	1	1	0	0

我们不使用单个感知器来表示XOR，而将建立一个具有多个人工神经元的人工神经网络，每个神经元都近似一个线性函数。每个样本的特征表述都被输入到两个神经元：一个NAND神经元和一个OR神经元。这些神经元的输出将连接到第三个AND神经元上，测试XOR的条件是否为真。

前馈与反馈人工神经网络

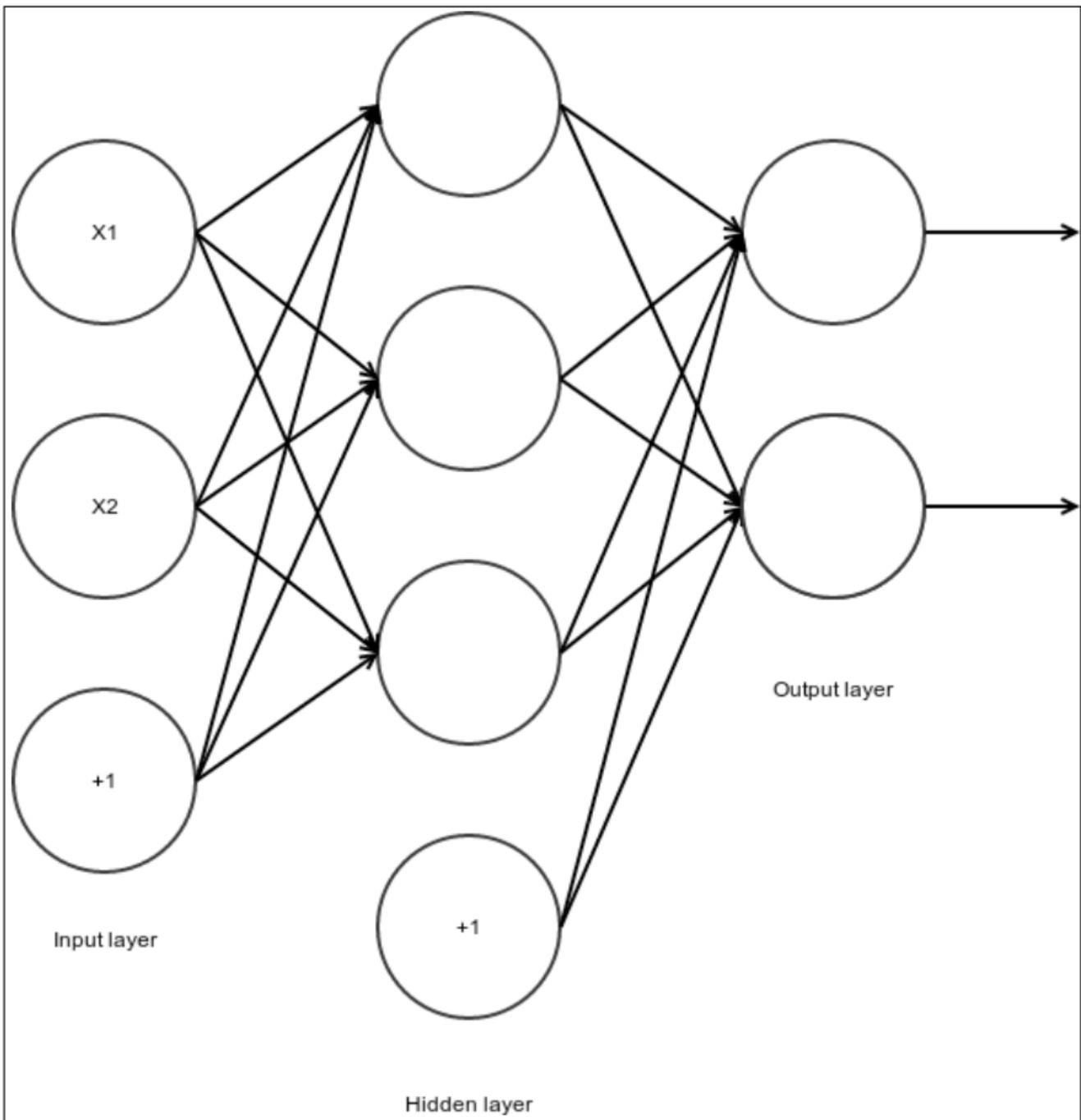
人工神经网络由三个组成部分。第一个组成部分是架构（architecture），或称为拓扑结构（topology），描述神经元的层次与连接神经元的结构。第二个组成部分是神经网络使用的激励函数。第三个组成部分是找出最优权重值的学习算法。

人工神经网络主要分为两种类型。前馈人工神经网络（Feedforward neural networks）是最常用的神经网络类型，一般定义为有向无环图。信号只能沿着最终输入的那个方向传播。另一种类型是反馈人工神经网络（feedback neural networks），也称递归神经网络（recurrent neural networks），网络图中有环。反馈环表示网络的一种内部状态，随着不同时间内输入条件的改变，网络的行为也会发生变化。反馈人工神经网络的临时状态让它们适合处理涉及连续输入的问题。因为目前scikit-learn没有实现反馈人工神经网络，本文只介绍前馈人工神经网络。

多层感知器

多层感知器（multilayer perceptron, MLP）是最流行的人工神经网络之一。它的名称不太恰当，多层感知器并非指单个带有多个层次的感知器，而是指可以是感知器的人工神经元组成的多个层次。MLP的层次结构是一个有向无环图。通常，每一层都全连接到下一层，某一层上的每个人工神经元的输出成为下一层若干人工神经元的输入。MLP至少有三层人工神经元。

输入层（input layer）由简单的输入人工神经元构成。每个输入神经元至少连接一个隐藏层（hidden layer）的人工神经元。隐藏层表示潜在的变量；层的输入和输出都不会出现在训练集中。隐藏层后面连接的是输出层（output layer）。下图所示的三层架构的多层感知器。带有+1标签的是常误差项神经元，大多数结构图中都不会画出来。



隐藏层中的人工神经元，也称单元（units）通常用非线性激励函数，如双曲正切函数（hyperbolic tangent function）和逻辑函数（logistic function），公式如下所示：

$$f(x) = \tanh(x)$$

$$f(x) = \frac{1}{1 + e^{-x}}$$

和其他的监督模型一样，我们的目标是找到成本函数最小化的权重值。通常，MLP的成本函数是残差平方和的均值，计算公式如下所示，其中的 m 表示训练样本的数量：

$$MSE = \frac{1}{m} \sum_{i=1}^m (y_i - f(x_i))^2$$

成本函数最小化

反向传播（backpropagation）算法经常用来连接优化算法求解成本函数最小化问题，比如梯度下降

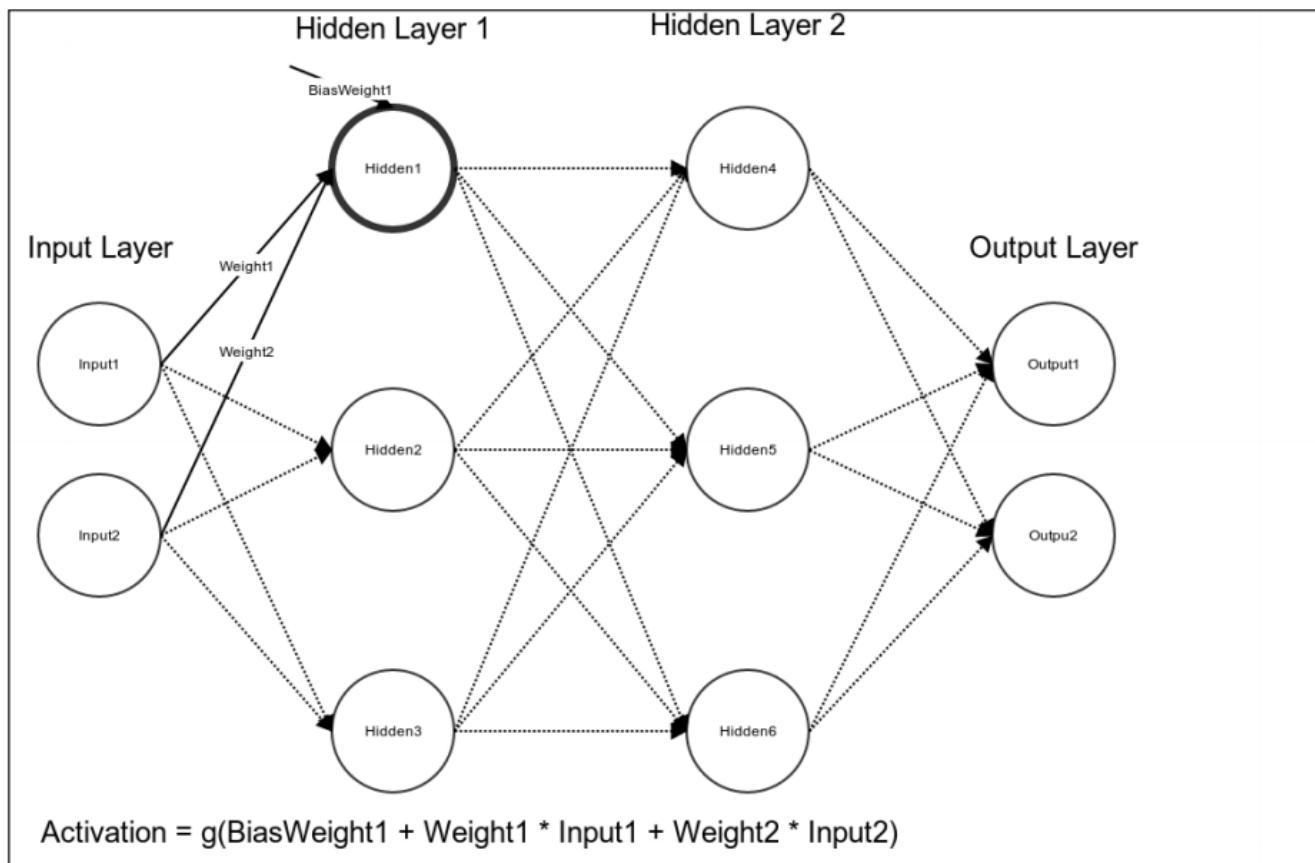
法。这个算法名称是反向（back）和传播（propagation）的合成词，是指误差在网络层的流向。理论上，反向传播可以用于训练具有任意层、任意数量隐藏单元的前馈人工神经网络，但是计算能力的实际限制会约束反向传播的能力。

反向传播与梯度下降法类似，根据成本函数的梯度变化不断更新模型参数。与我们前面介绍过的线性模型不同，神经网络包含不可见的隐藏单元；我们不能从训练集中找到它们。如果我们找不到这些隐藏单元，我们也就不能计算它们的误差，不能计算成本函数的梯度，进而无法求出权重值。如果一个随机变化是某个权重降低了成本函数值，那么我们保留这个变化，就可能同时改变另一个权重的值。这种做法有个明显的问题，就是其计算成本过高。而反向传播算法提供了一种有效的解决方法。

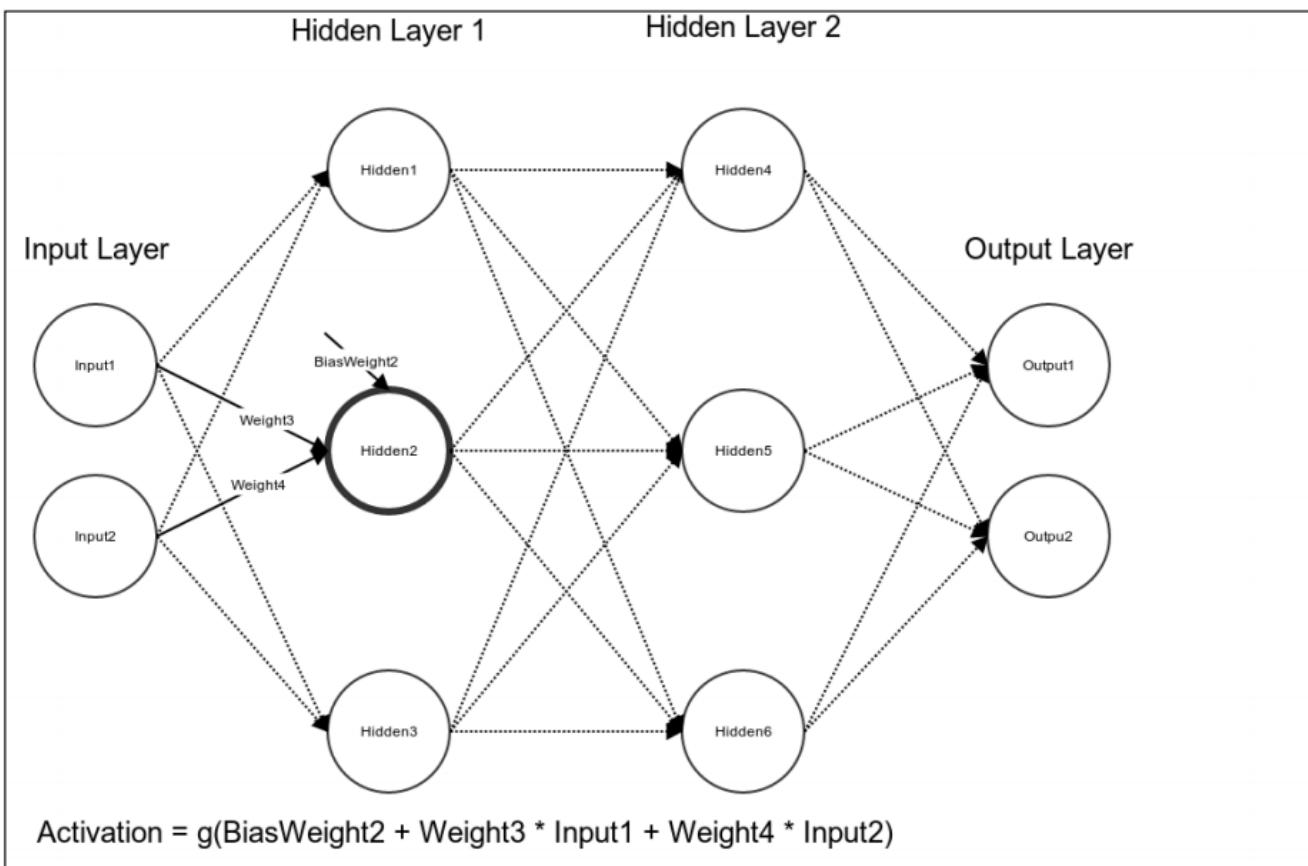
我们将用反向传播逐步来训练一个前馈人工神经网络。这个网络与两个输入单元，两个隐藏层分别有三个隐藏单元，两个输出单元。输入单元与第一个隐藏层的三个隐藏单元Hidden1, Hidden2和Hidden3全连接。单元之间连接的边开始用很小的随机数表示权重。

前向传播

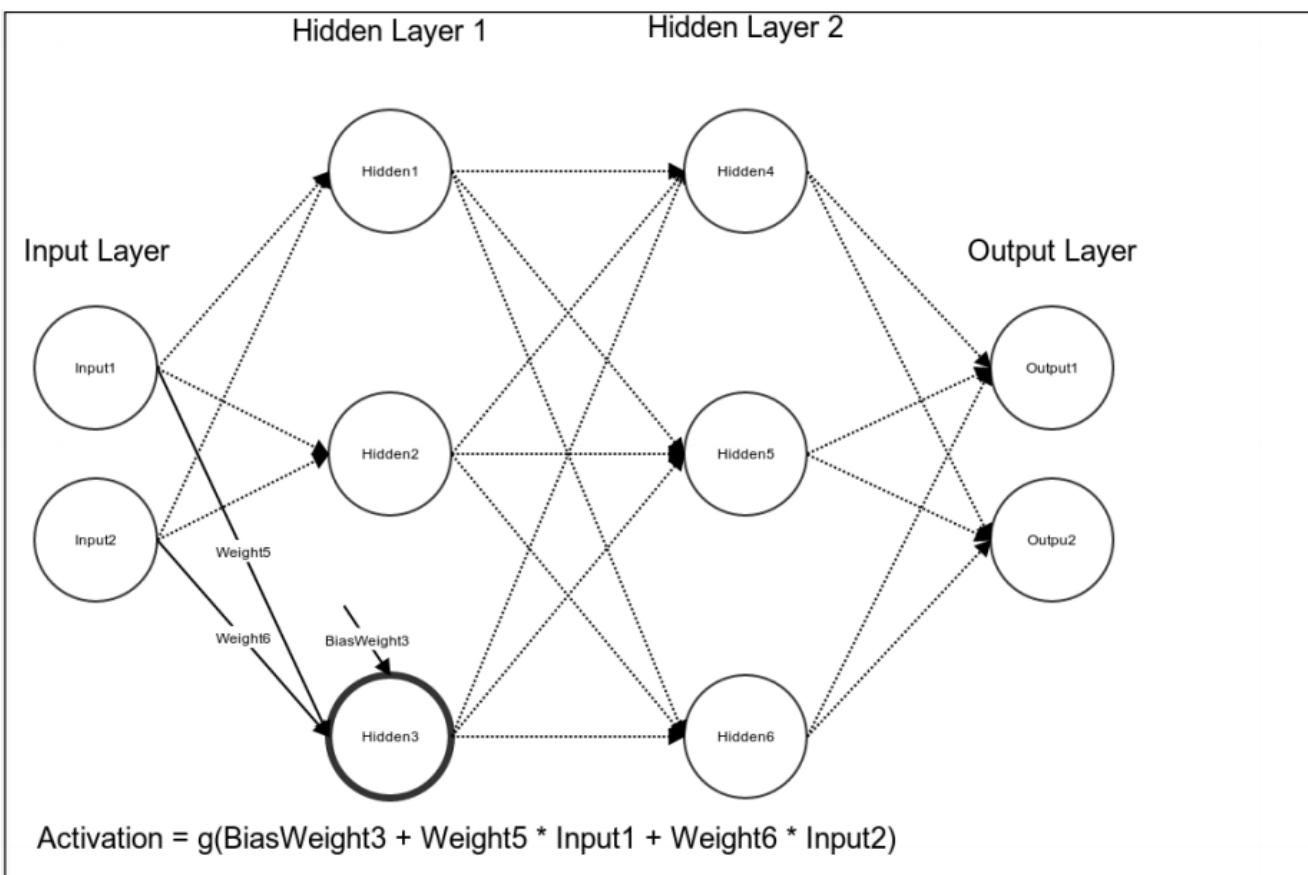
在前向传播（forward propagation）阶段，特征变量被输入到网络，然后传播到下一层产生输出激励（activation）。首先，我们计算Hidden1单元的激励。我们找到Hidden1单元的加权和，然后用激励函数处理输入的加权和。注意Hidden1单元会收到一个常误差项输入单元，并没有在下图中画出。其中 $g(x)$ 是一个激励函数：



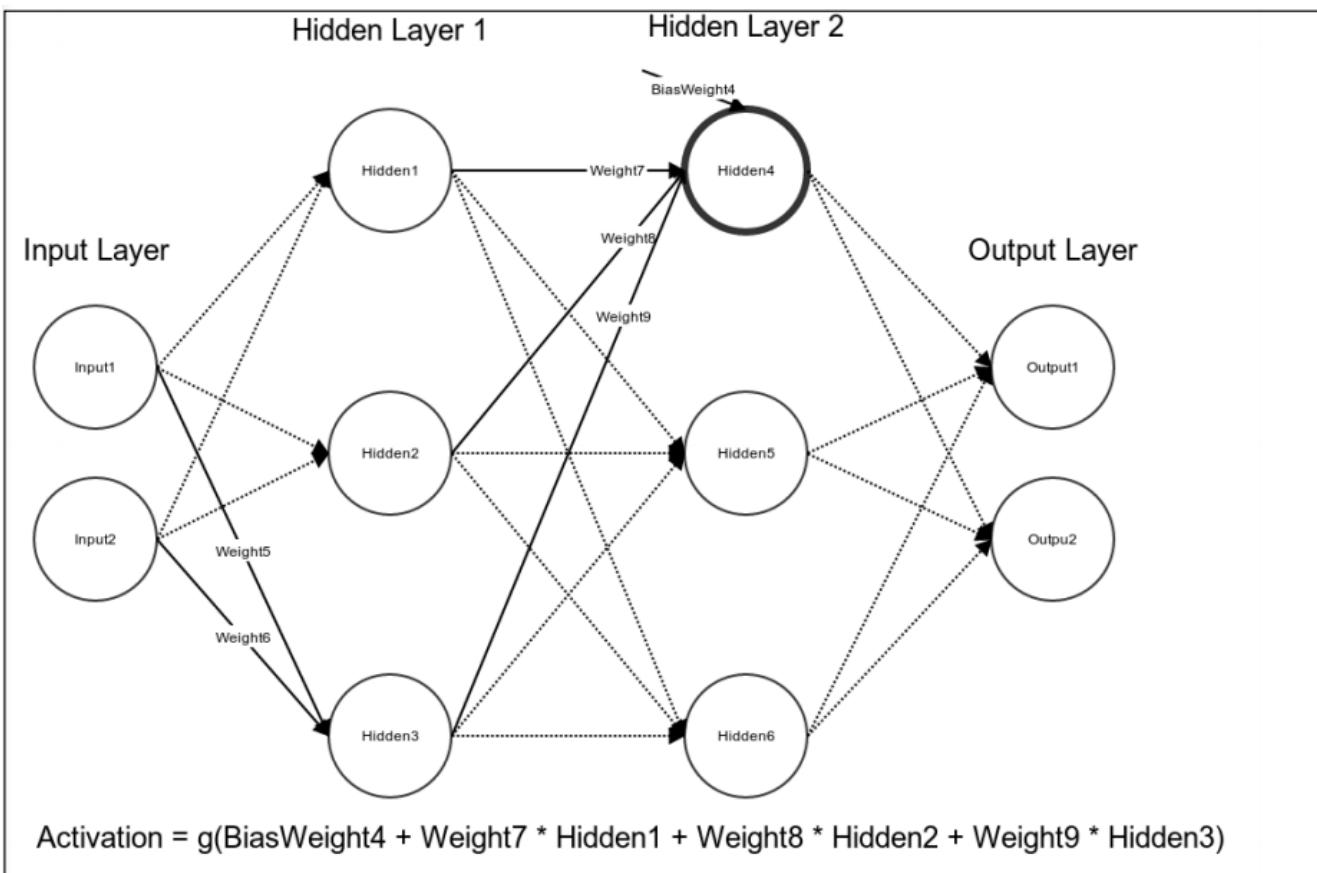
然后，我们计算第二个隐藏单元Hidden2。和Hidden1类似，它也会收到一个常误差项输入单元。我们计算输入单元的加权和，或成为预激励，经过激励函数处理的激励如下图所示：



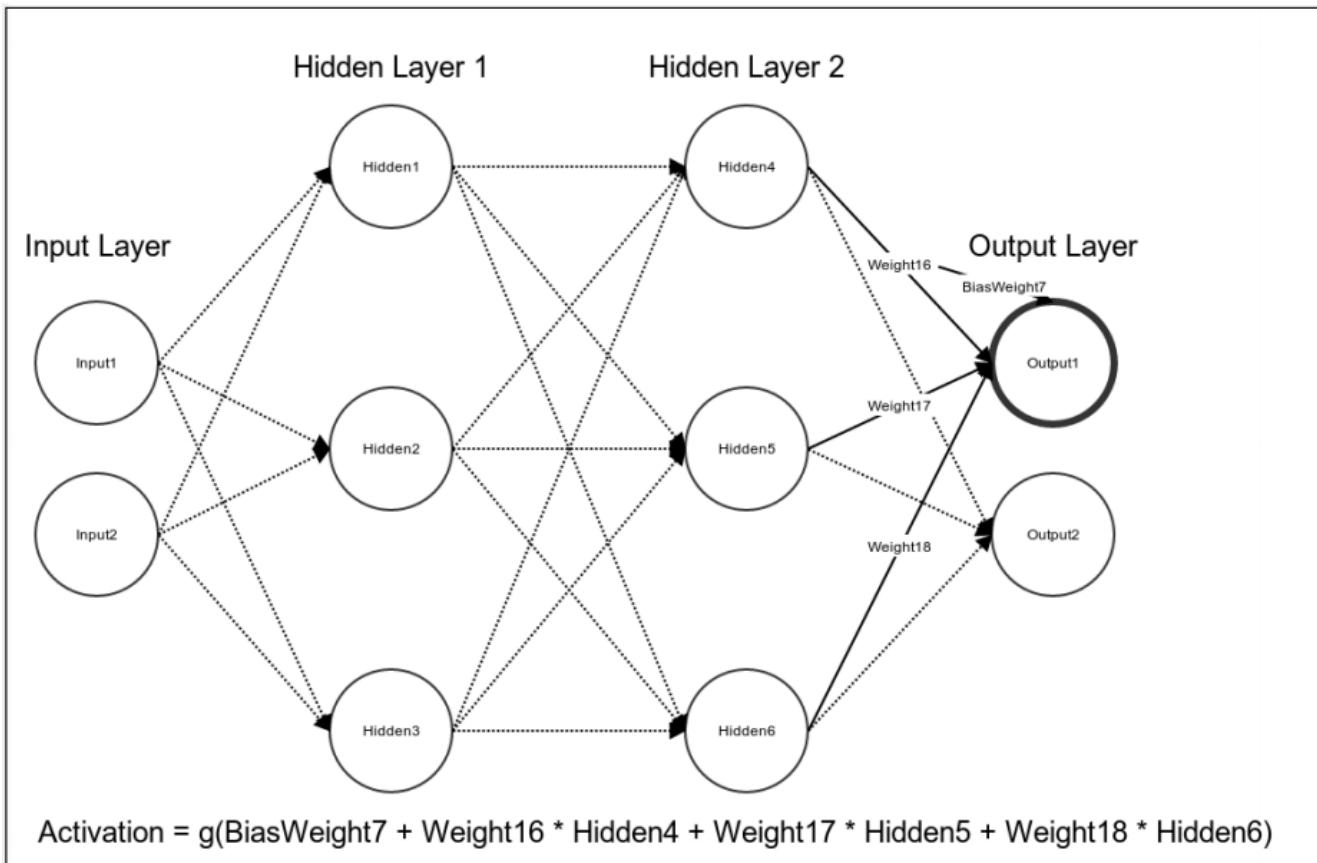
同理，我们计算第三个隐藏单元Hidden3的激励：



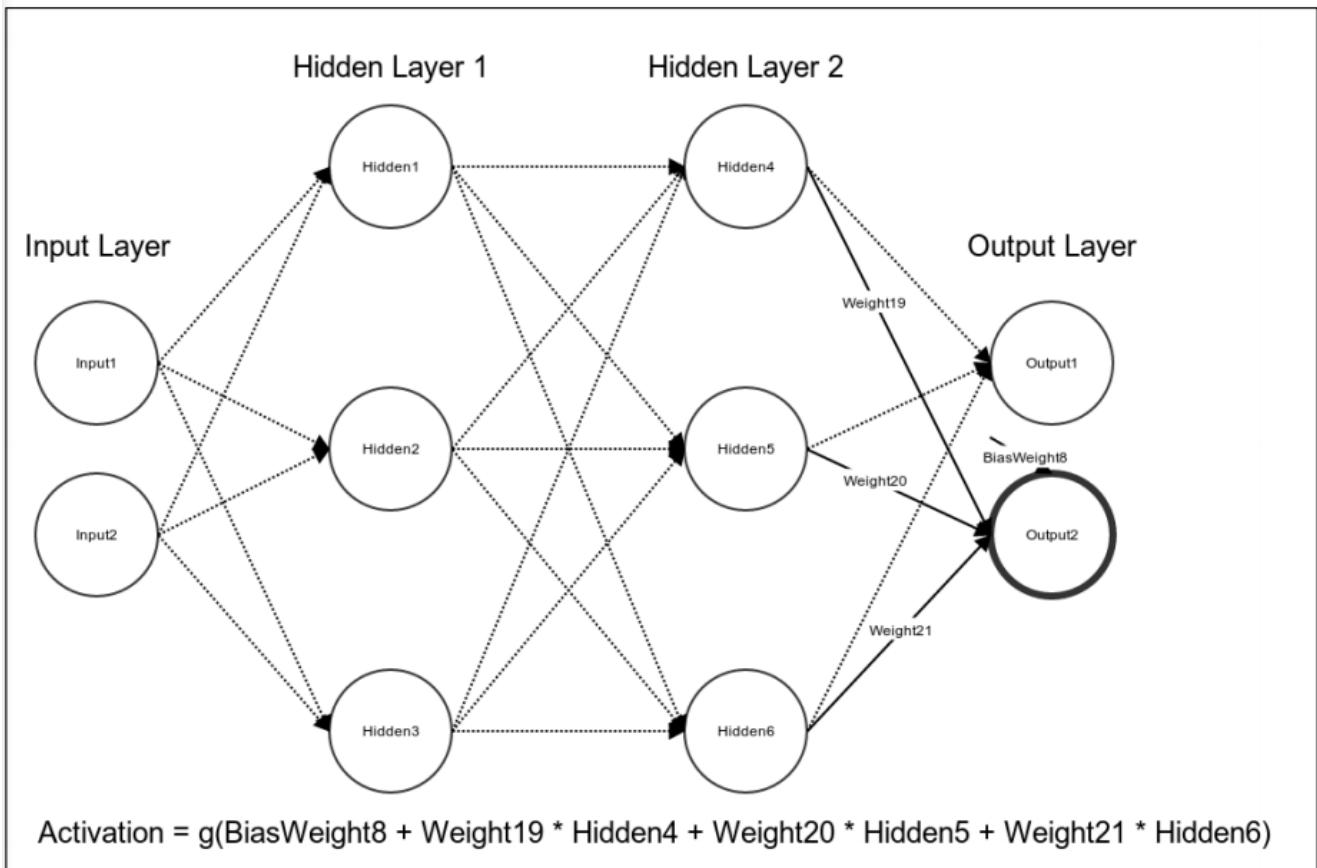
第一个隐藏层三个隐藏单元的激励计算完之后，我们再处理第二个隐藏层。本例中，第一个隐藏层第一个隐藏层到第二个隐藏层。与第一个隐藏层三个隐藏单元计算过程类似，都有一个常误差项输入单元，并没有在图中画出，我们计算Hidden4的激励如下图所示：



按照同样方法计算Hidden5和Hidden6的激励。当第二个隐藏层三个隐藏单元的激励计算完成后，我们计算输出层。Output1的激励是第二个隐藏层三个隐藏单元的激励的加权和经过激励函数处理的结果。类似与隐藏单元，有一个常误差项的输入单元，如下图所示：



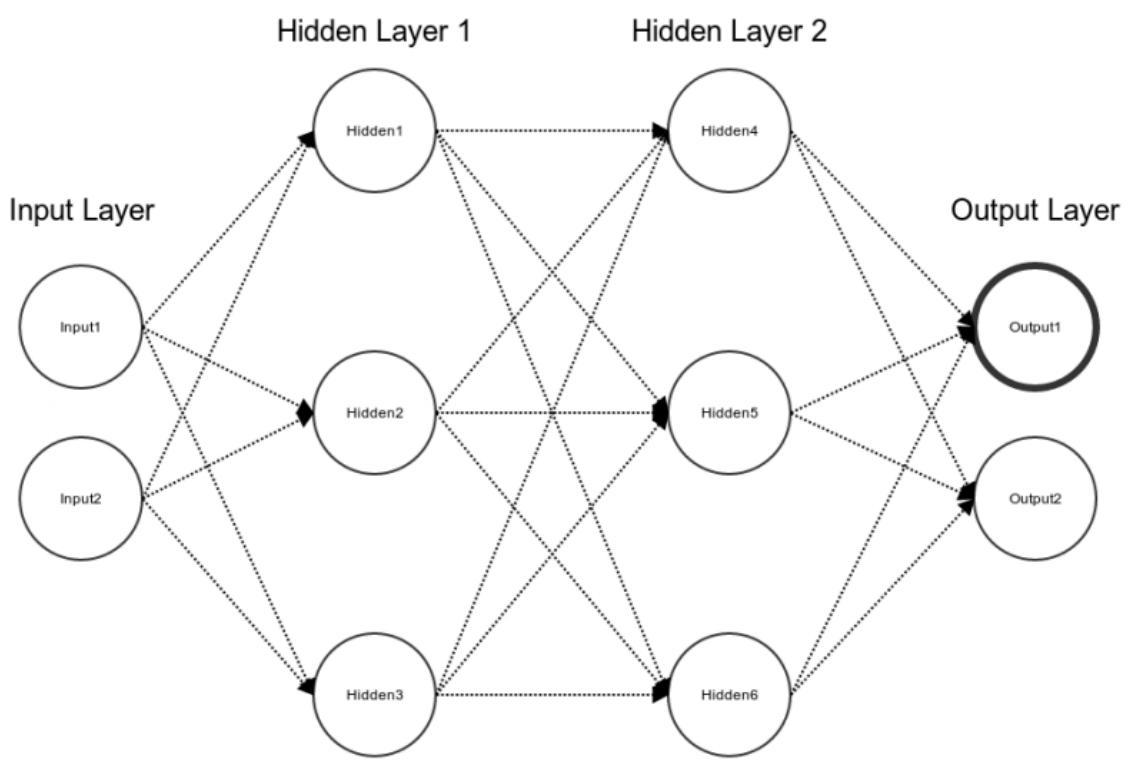
同理，我们计算第二个输出单元Output2的激励：



计算完神经网络中所有单元的激励之后，我们就完成了前向传播过程。用这些随机生成的权重值是不可能很好的近似网络的真实函数的。我们必须更新权重值来生成一个更好的近似函数。

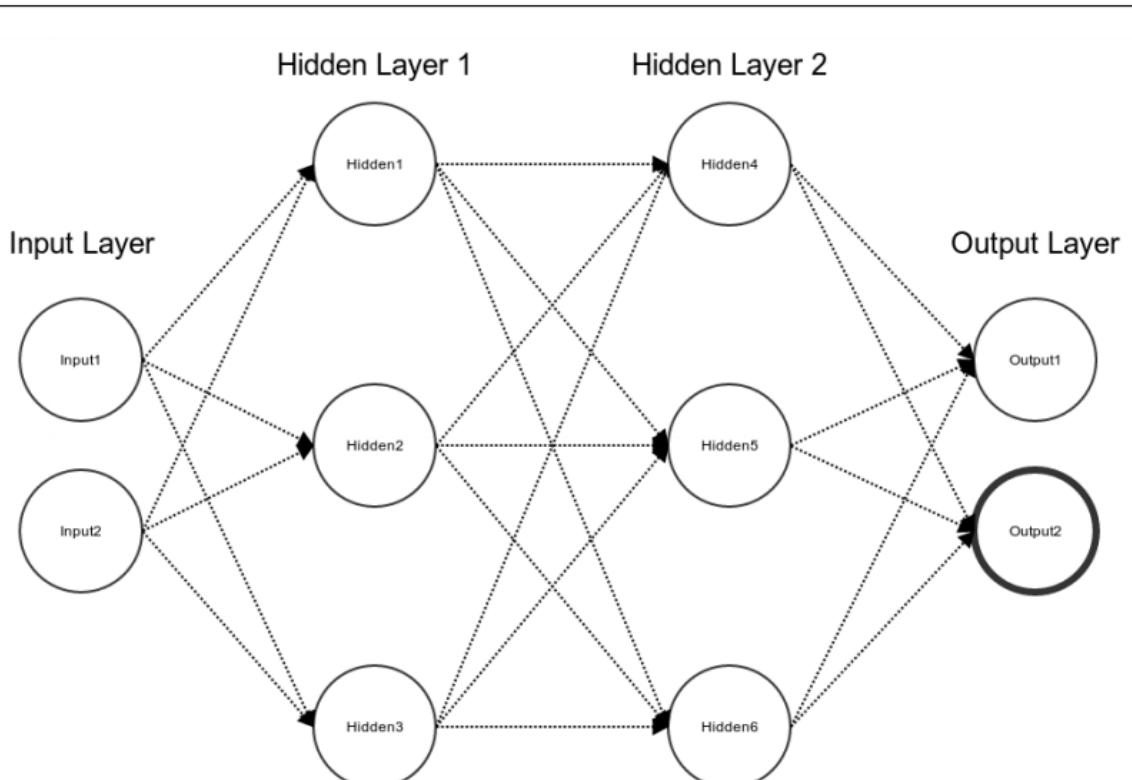
反向传播

我们把输出单元计算的误差作为网络误差。隐藏单元表示不可见的变量；没有数据可以进行对照，因此我们无法度量隐藏单元。为了更新权重，我们必须把网络的误差反向传回。于是，我们先从Output1输出单元开始。其误差等于真实值与预测值的差，乘以激励函数对Output1输出单元的偏导数：



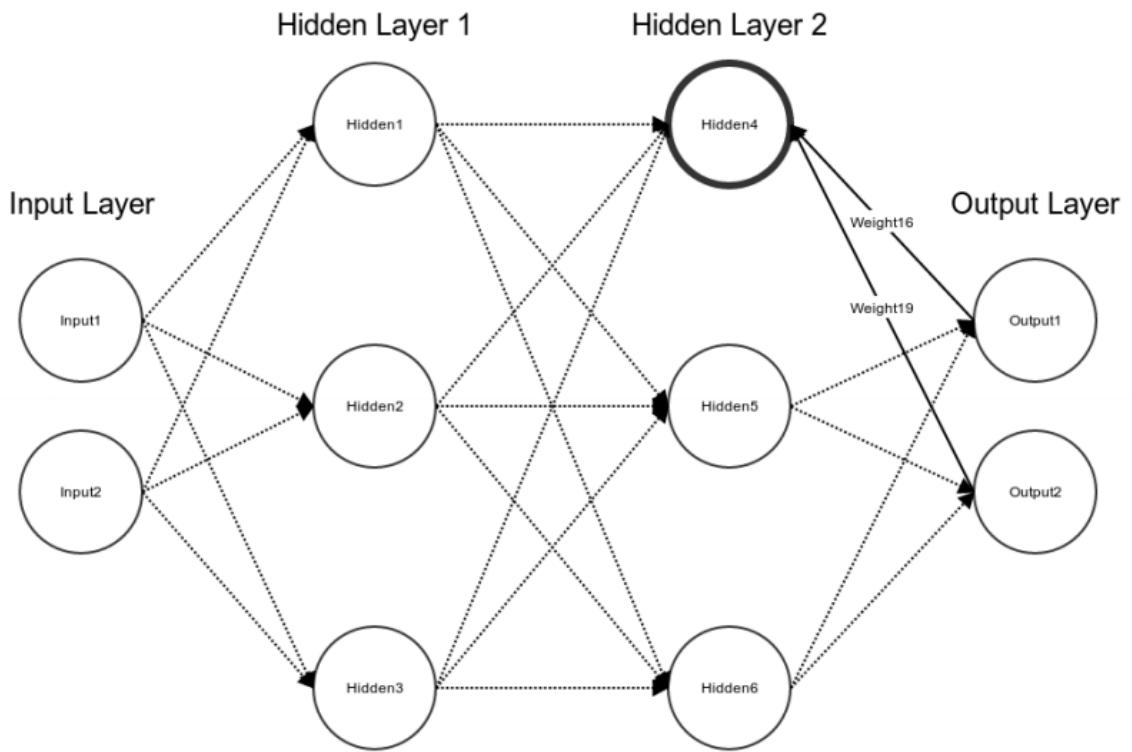
$$\text{Error}(\text{Output1}) = g'(\text{Output1}) * (\text{True1} - \text{Output1})$$

同理，我们计算第二个输出单元Output2的误差：



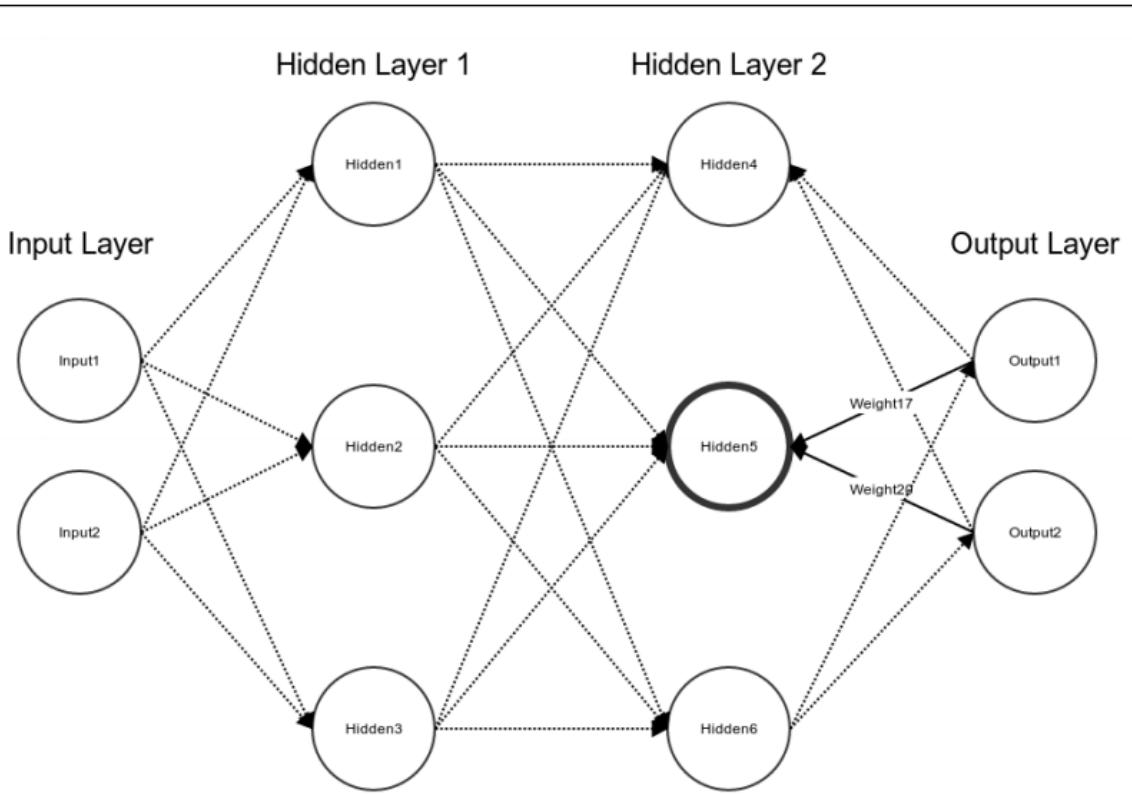
$$\text{Error}(\text{Output2}) = g'(\text{Output2}) * (\text{True2} - \text{Output2})$$

算完输出层的误差之后，我们把误差传回第二个隐藏层。首先，我们计算**Hidden4**的误差。把**Output1**的误差乘以连接**Hidden4**与**Output1**的权重，再**Output2**的误差乘以连接**Hidden4**与**Output2**的权重，再把它们相加就得到了**Hidden4**的误差：



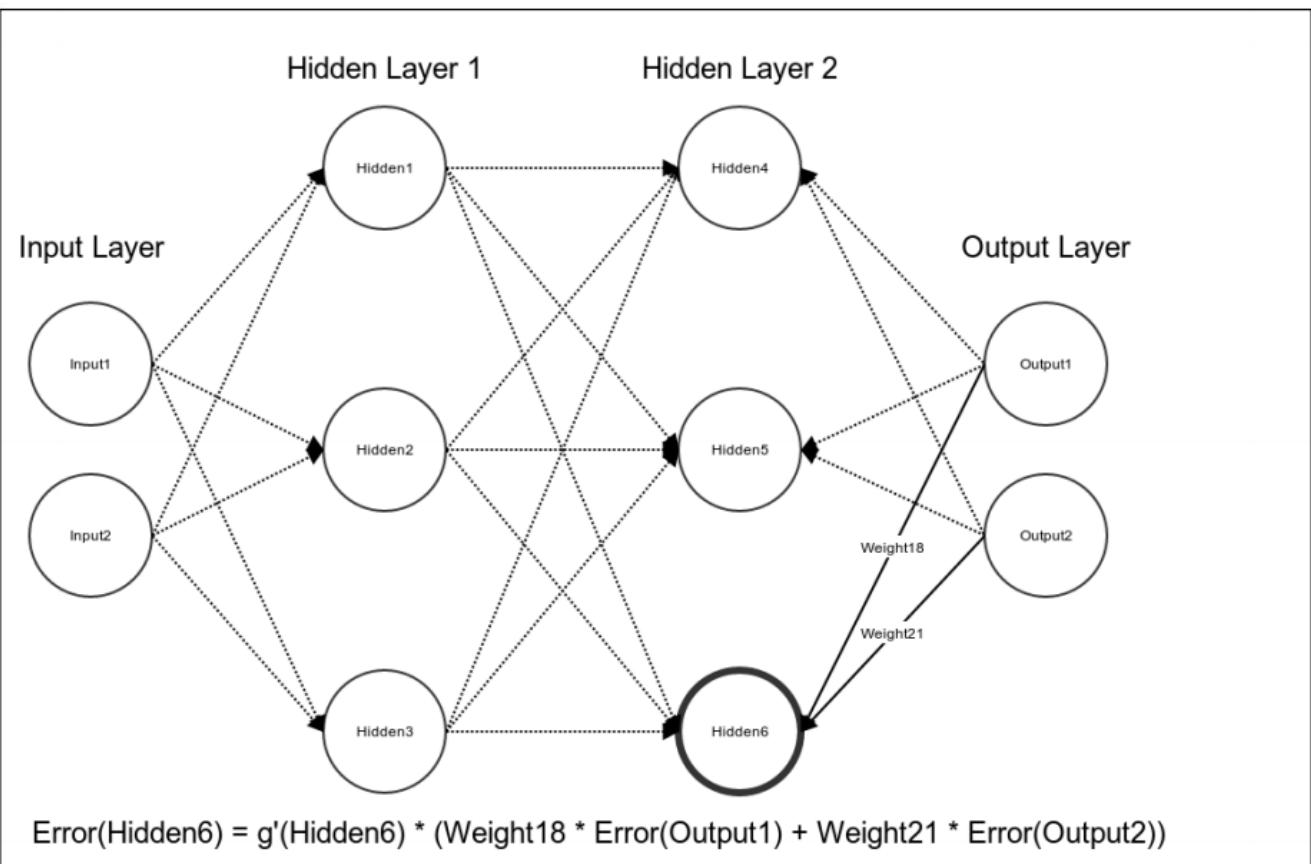
$$\text{Error}(\text{Hidden4}) = g'(\text{Hidden4}) * (\text{Weight16} * \text{Error}(\text{Output1}) + \text{Weight19} * \text{Error}(\text{Output2}))$$

同理，我们计算隐藏单元Hidden5的误差：

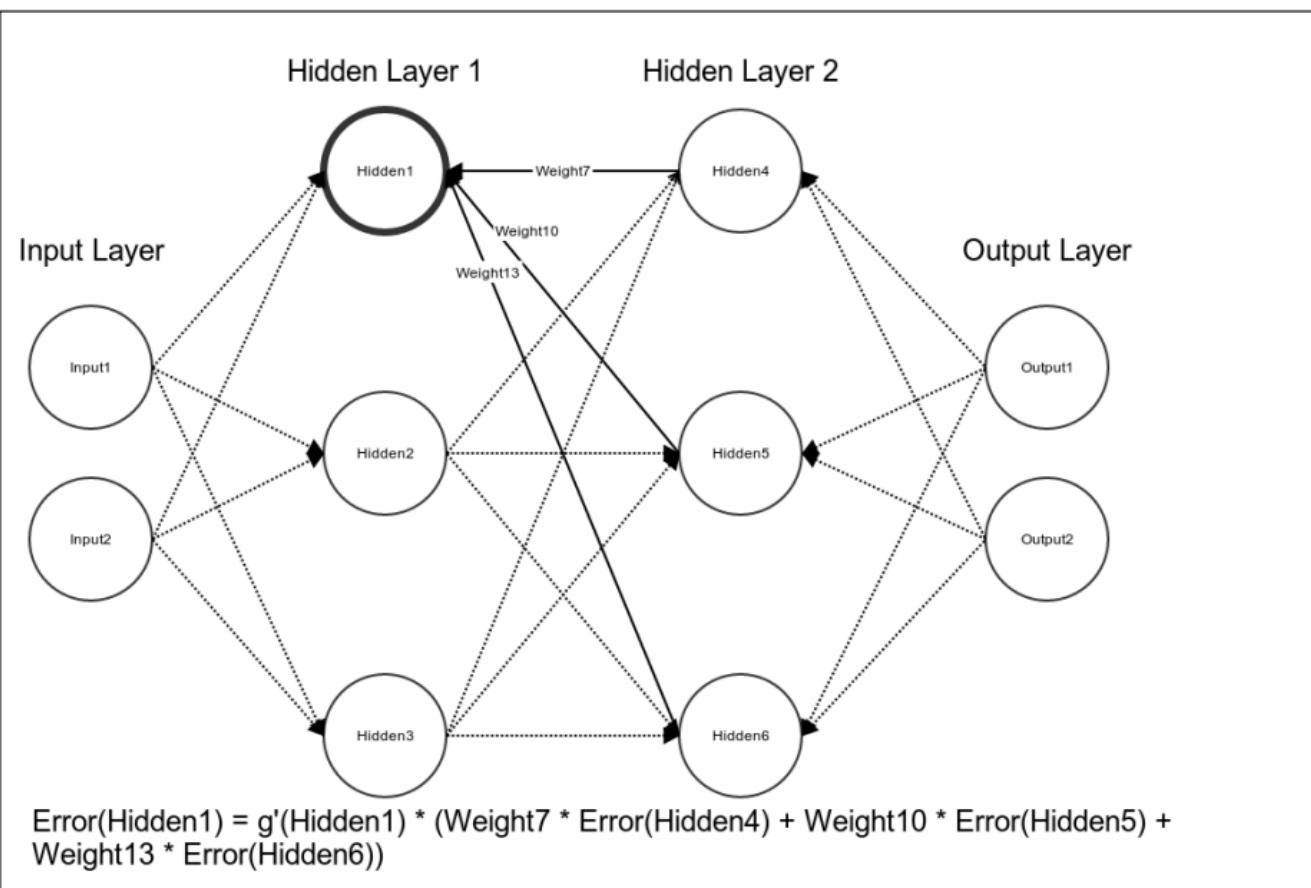


$$\text{Error}(\text{Hidden5}) = g'(\text{Hidden5}) * (\text{Weight17} * \text{Error}(\text{Output1}) + \text{Weight20} * \text{Error}(\text{Output2}))$$

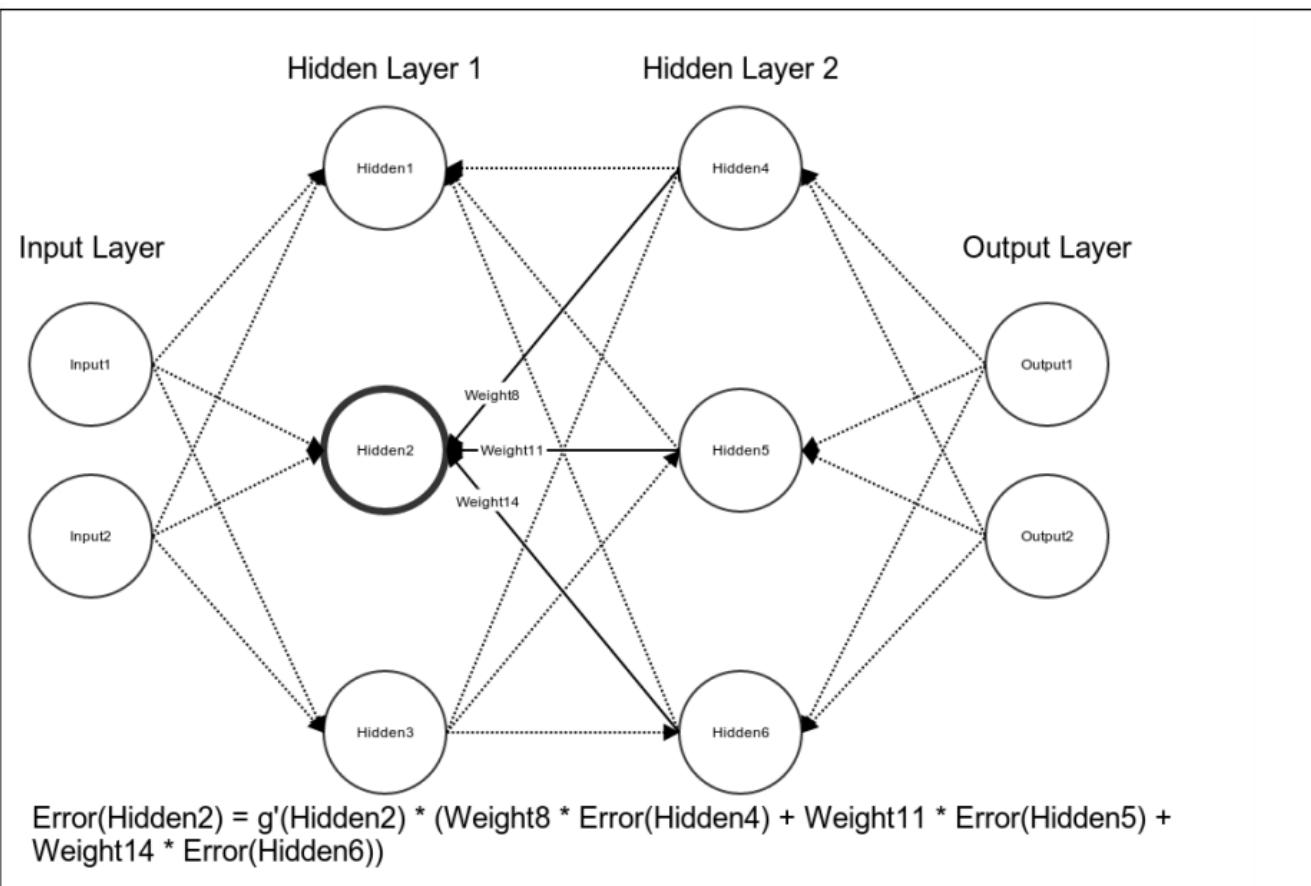
同理，我们计算隐藏单元Hidden6的误差：



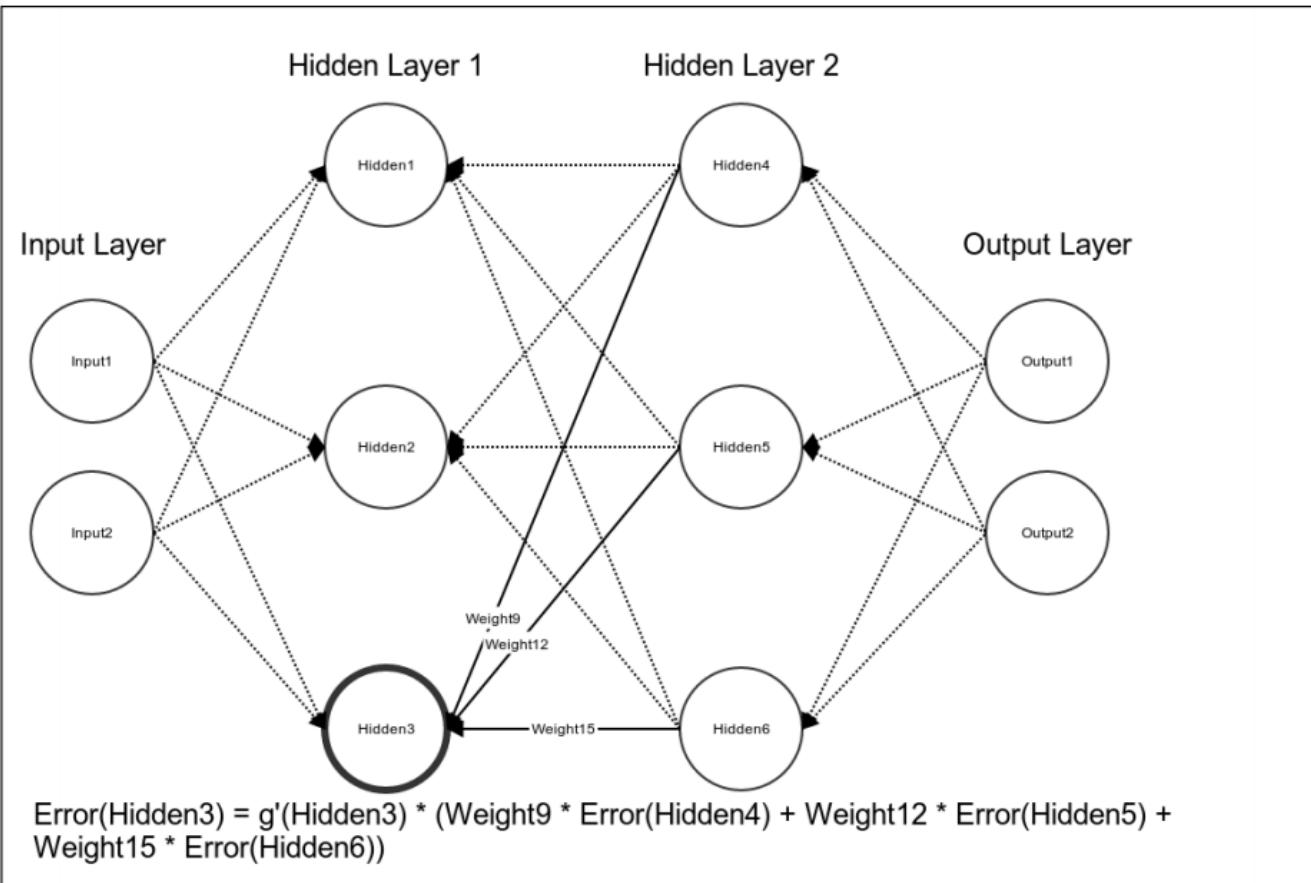
算完第二个隐藏层的误差之后，同理把误差传回第一个隐藏层。Hidden1的误差就是激励函数对Hidden1隐藏单元的偏导数乘以第二隐藏层加权误差和，如下图所示：



同理，我们计算隐藏单元Hidden2的误差：

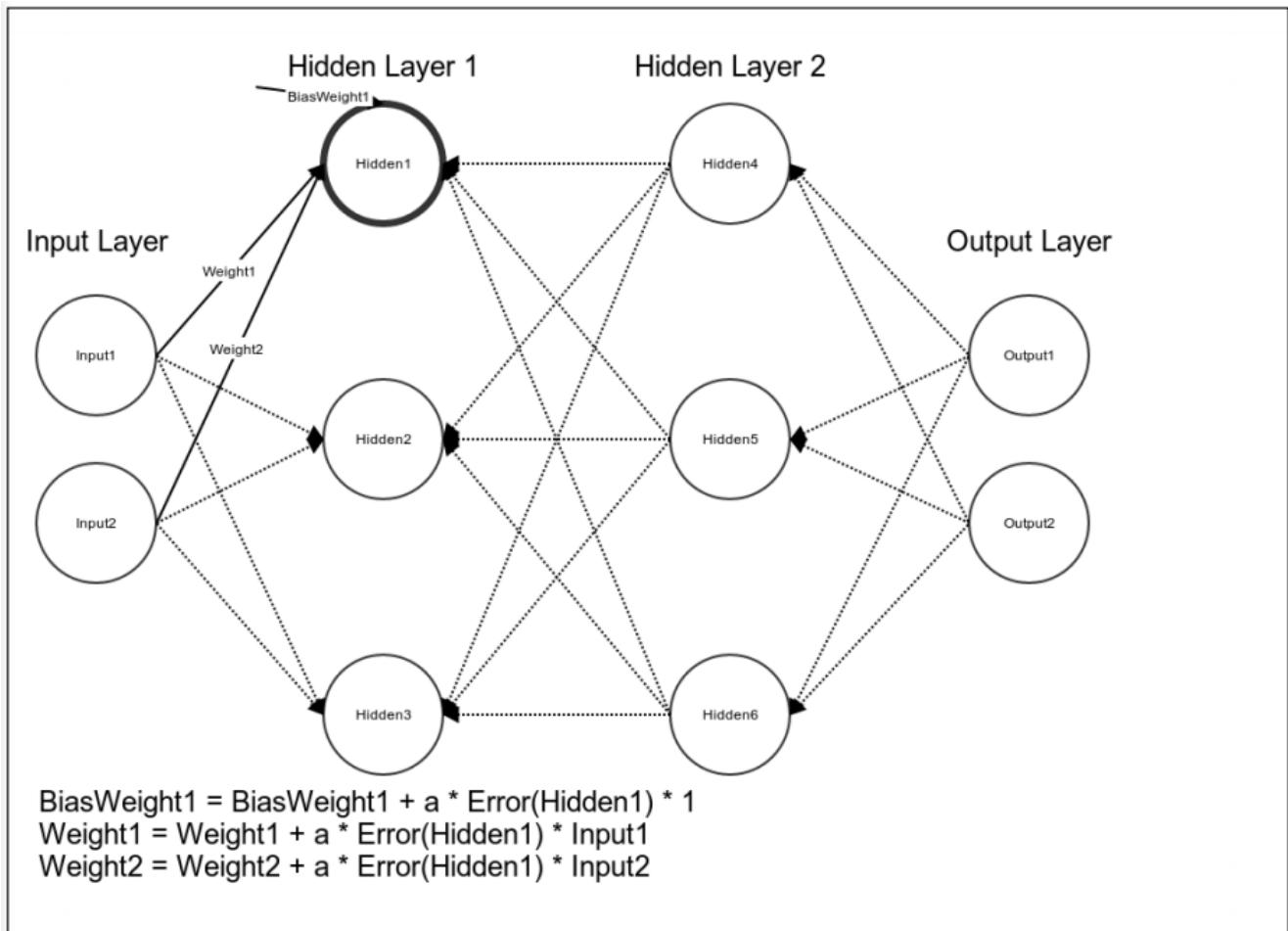


同理，我们计算隐藏单元Hidden3的误差：

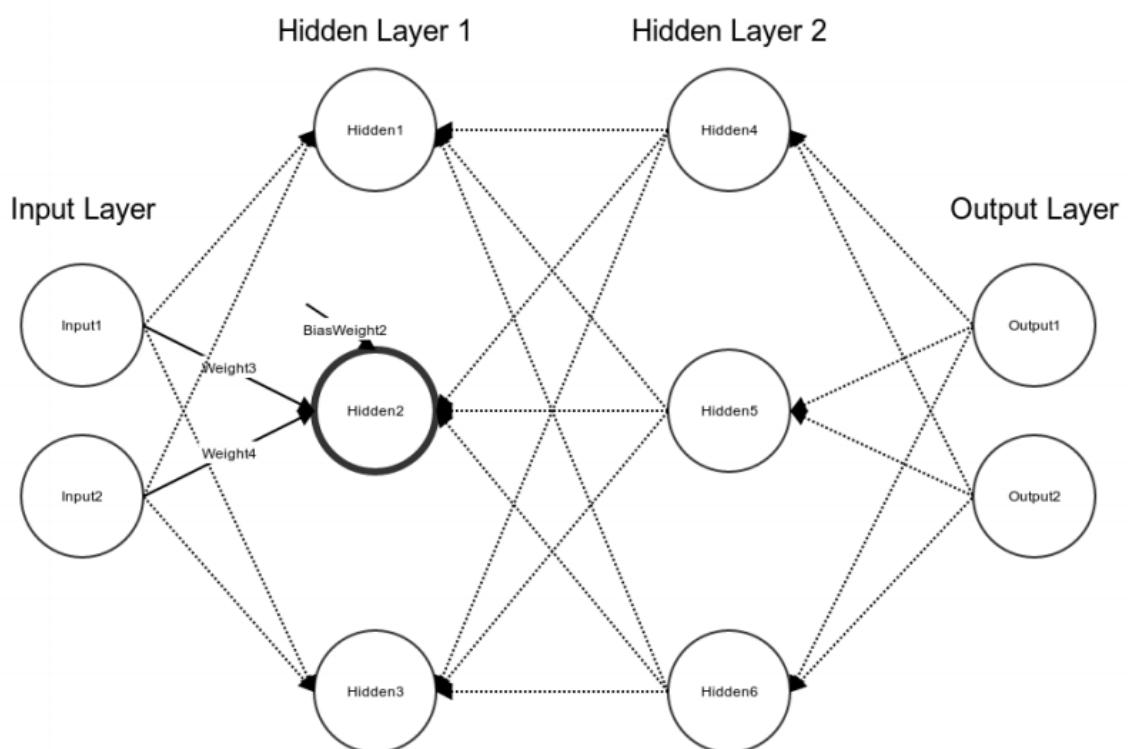


算完第一个隐藏层的误差之后，我们用这些误差来升级权重值。首先升级连接输入单元与Hidden1的边的权重，以及连接常误差项与Hidden1的边的权重。我们将连接Input1与Hidden1的边的权重Weight1增加学习速率，Hidden1的误差以及Input1的值的乘积。

按同样的方法处理权重Weight2，我们把学习速率，Hidden1的误差以及Input2的值的乘积增加到Weight2。最后，我们计算常误差项输入的权重，把学习速率，Hidden1的误差以及常误差项的值1的乘积增加到常误差项输入的权重，如下图所示：



同理，我们升级连接输入单元与Hidden2的边的权重，以及连接常误差项与Hidden2的边的权重：

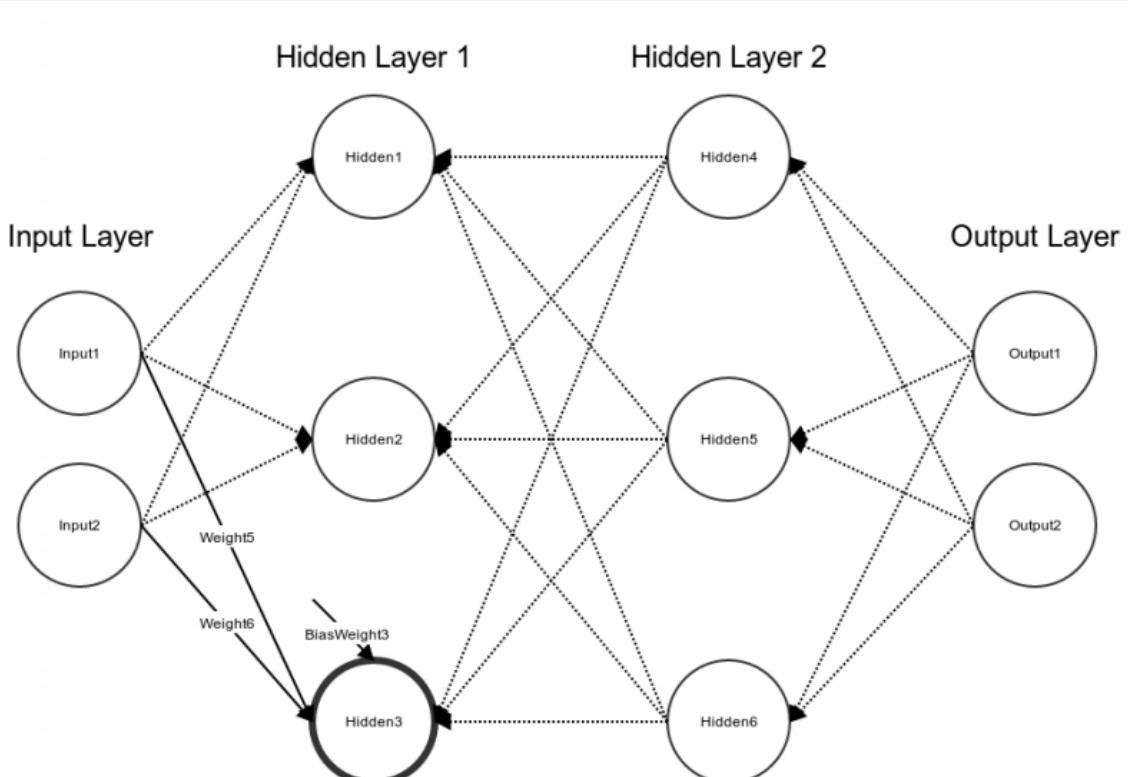


$$\text{BiasWeight2} = \text{BiasWeight2} + a * \text{Error}(\text{Hidden2}) * 1$$

$$\text{Weight3} = \text{Weight3} + a * \text{Error}(\text{Hidden2}) * \text{Input1}$$

$$\text{Weight4} = \text{Weight4} + a * \text{Error}(\text{Hidden2}) * \text{Input2}$$

同理，我们升级连接输入单元与Hidden3的边的权重，以及连接常误差项与Hidden3的边的权重：

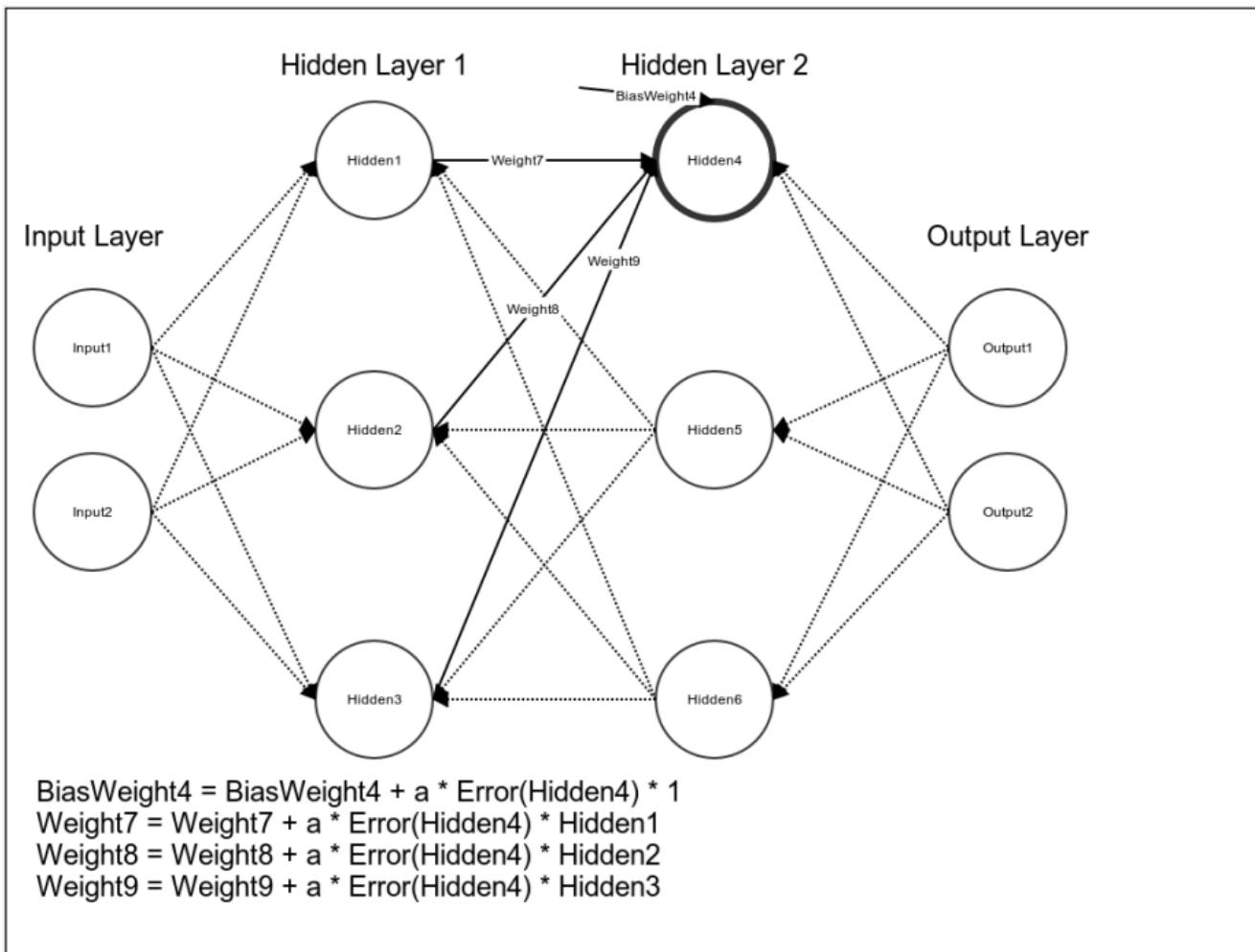


$$\text{BiasWeight3} = \text{BiasWeight3} + a * \text{Error}(\text{Hidden3}) * 1$$

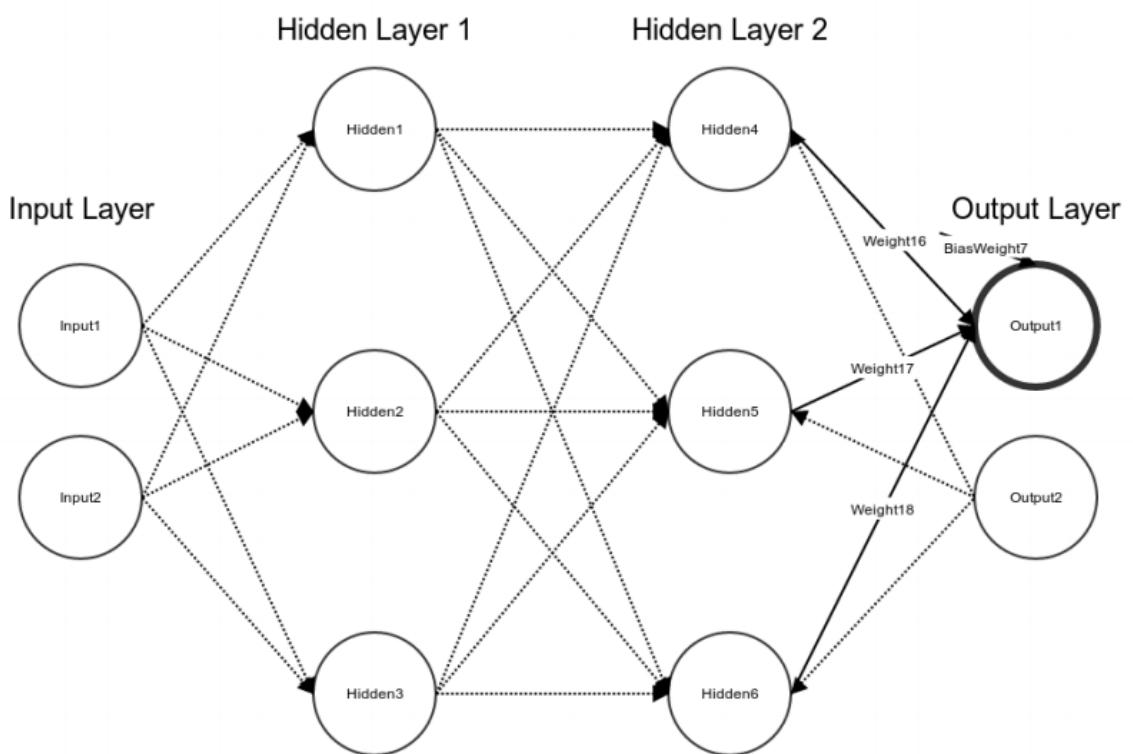
$$\text{Weight5} = \text{Weight5} + a * \text{Error}(\text{Hidden3}) * \text{Input1}$$

$$\text{Weight6} = \text{Weight6} + a * \text{Error}(\text{Hidden3}) * \text{Input2}$$

输入层与第一隐藏层之间边的权重全部更新之后，我们可以用同样的方法计算第一隐藏层与第二隐藏层直接边的权重。我们计算权重Weight7，把学习速率，Hidden4的误差以及Hidden1的值的乘积增加到Weight7。同样的方法计算Weight8与Weight15：



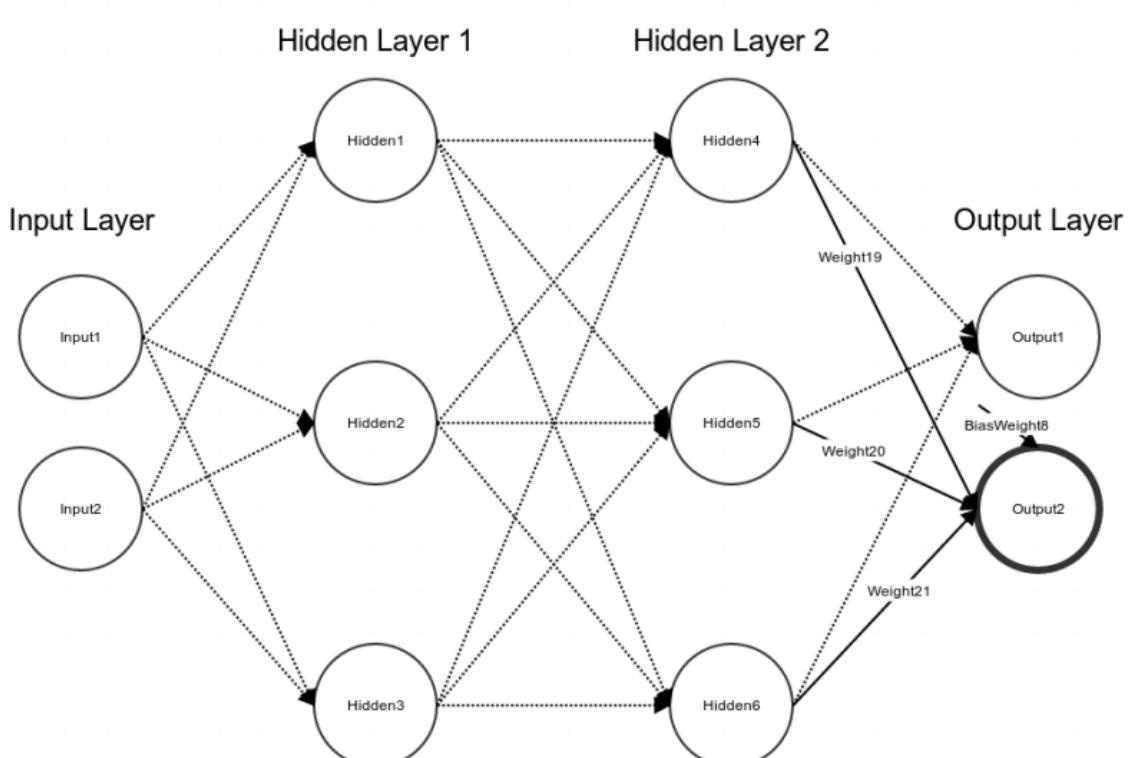
连接Hidden5与Hidden6的边的权重按同样的方法升级。升级完两个隐藏层之间的权重之后，我们就来升级第二隐藏层与输出层之间的权重。用同样的方法升级Weight16到Weight21的权重：



$$\text{BiasWeight7} = \text{BiasWeight7} + a * \text{Error}(\text{Output1}) * 1$$

$$\text{Weight16} = \text{Weight16} + a * \text{Error}(\text{Output1}) * \text{Hidden3}$$

$$\text{Weight17} = \text{Weight17} + a * \text{Error}(\text{Output1}) * \text{Hidden4}$$

$$\text{Weight18} = \text{Weight18} + a * \text{Error}(\text{Output1}) * \text{Hidden6}$$


$$\text{BiasWeight8} = \text{BiasWeight8} + a * \text{Error}(\text{Output2}) * 1$$

$$\text{Weight19} = \text{Weight19} + a * \text{Error}(\text{Output2}) * \text{Hidden3}$$

$$\text{Weight20} = \text{Weight20} + a * \text{Error}(\text{Output2}) * \text{Hidden4}$$

$$\text{Weight21} = \text{Weight21} + a * \text{Error}(\text{Output2}) * \text{Hidden6}$$

把学习速率，Output2的误差以及Hidden6的激励的乘积增加到weight21之后，这个阶段网络权重的更新工作就完成了。我们现在可以用心的权重再运行一遍前向传播，成本函数的值应该会减少。重复这个过程直到模型收敛或者停止条件得到了满足为止。与之前我们介绍过的线性模型不同，反向传播不能优化凸函数。反向传播可能用某个局部最小值的参数值达到收敛，而不是全局最小值。实际应用中，局部最小值通常可以解决问题。

用多层感知器近似XOR函数

让我们训练一个多层感知器来近似XOR函数。目前，scikit-learn的0.16.1版本还没有合并，作者在其github上提供了单独的MLP模块NeuralNetworks (<https://github.com/IssamLaradji/NeuralNetworks>)，我们在0.16.1版本基础上稍作修改，即可完成本书的例子。

请fork作者的NeuralNetworks，将multilayer_perceptron文件夹里的.py文件复制到sklearn/neural_network文件夹里，然后将对__init__.py文件做如下修改即可：

```
# __init__.py
from .rbm import BernoulliRBM

from .multilayer_perceptron import MultilayerPerceptronClassifier
from .multilayer_perceptron import MultilayerPerceptronRegressor

__all__ = ["BernoulliRBM",
           "MultilayerPerceptronClassifier",
           "MultilayerPerceptronRegressor"
       ]
```

另外，在Linux和Mac OS系统里，复制.py文件的权限记得改成chmod 644

首先，我们建议一个简单的二分分类数据集表示XOR，然后用交叉检验分割成训练集和测试集：

```
In [1]:
```

```
from sklearn.cross_validation import train_test_split
from sklearn.neural_network import MultilayerPerceptronClassifier
import numpy as np
y = [0, 1, 1, 0] * 1000
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]] * 1000)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=3)
```

然后，我们将MultilayerPerceptronClassifier类实例化。用n_hidden设置神经网络架构中隐藏层的层数。我们将隐藏层的层数设置为两层。MultilayerPerceptronClassifier类自动创建两个输入单元和一个输出单元。在多元分类问题中分类器会为每一个可能的类型创建一个输出。

选择神经网络架构是很复杂的事情。确定隐藏单元和隐藏层的数量有一些首要原则，但是都没有必然的依据。隐藏单元的数量由样本数量，训练数据的噪声，要被近似的函数复杂性，隐藏单元的激励函数，学习算法和使用的正则化方法决定。实际上，架构的效果只能通过交叉检验得出。

我们通过fit()函数训练模型：

```
In [3]:
```

```
clf = MultilayerPerceptronClassifier(hidden_layer_sizes=[2],
                                      activation='logistic',
                                      algorithm='sgd',
                                      random_state=3)
clf.fit(X_train, y_train)
```

```
Out[3]:
```

```
MultilayerPerceptronClassifier(activation='logistic', algorithm='sgd',
                                alpha=1e-05, batch_size=200, hidden_layer_sizes=[2],
                                learning_rate='constant', learning_rate_init=0.5,
                                max_iter=200, power_t=0.5, random_state=3, shuffle=False,
                                tol=1e-05, verbose=False, warm_start=False)
```

最后，我们打印估计模型对测试集预测的准确率和一些手工输入的预测结果。预测测试集的结果表明，这个人工神经网络可以完美的近似XOR函数：

In [5]:

```
print('层数: %s, 输出单元数量: %s' % (clf.n_layers_, clf.n_outputs_))
predictions = clf.predict(X_test)
print('准确率: %s' % clf.score(X_test, y_test))
for i, p in enumerate(predictions[:10]):
    print('真实值: %s, 预测值: %s' % (y_test[i], p))
```

层数: 3, 输出单元数量: 1

准确率: 1.0

真实值: 1, 预测值: 1
真实值: 1, 预测值: 1
真实值: 1, 预测值: 1
真实值: 0, 预测值: 0
真实值: 1, 预测值: 1
真实值: 0, 预测值: 0
真实值: 0, 预测值: 0
真实值: 1, 预测值: 1
真实值: 0, 预测值: 0
真实值: 1, 预测值: 1

手写数字识别

在上一章我们介绍过用支持向量机识别MNIST数据集里面的手写数字。下面我们用人工神经网络来识别：

In [6]:

```
from sklearn.datasets import load_digits
from sklearn.cross_validation import train_test_split, cross_val_score
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.neural_network.multilayer_perceptron import MultilayerPerceptronClassifier
```

首先我们用`load_digits`函数加载数据集。因为我们要在交叉验证期间拷贝（fork）进程，所以程序要放在`main()`函数里运行：

In [7]:

```
if __name__ == '__main__':
    digits = load_digits()
    X = digits.data
    y = digits.target
```

在人工神经网络里，放大特征非常重要，因为这样可以让算法更快的收敛。在用`MultilayerPerceptronClassifier`训练模型前，我们用`Pipeline`类放大数据。这个神经网络包括一个输入层和一个输出层，两个隐藏层，其中一个有150个单元，另一个有100个单元。我们还增加了正则化`alpha`超参数的值。最后，我们打印三个交叉验证组合的预测准确率。

In [8]:

```
pipeline = Pipeline([
    ('ss', StandardScaler()),
    ('mlp', MultilayerPerceptronClassifier(hidden_layer_sizes=[150, 100], alpha=0.1))
])
print('准确率: %s' % cross_val_score(pipeline, X, y, n_jobs=-1))
```

准确率: [0.8654485 0.87646077 0.87248322]

人工神经网络的平均准确率与支持向量机一致。增加隐藏单元和隐藏层，运用网格搜索，会进一步提供模型的准确率。

总结

本章我们介绍了人工神经网络，一种通过人工神经元的组合来表述复杂函数的强大的分类和回归模型。本文介绍的有向无环图称为前馈人工神经网络。多层感知器就是一种前馈人工神经网络，其每一次都完全连接后面一层。带一个隐藏层和若干隐藏单元的MLP是一种通用函数近似器。它可以表示任何连续函数，尽管它未必能够自动的学习到适当的权重。我们还介绍了网络的隐藏层如何表示不可见的变量，以及如何用反向传播算法学习权重。最后，我们用scikit-learn的多次感知器MultilayerPerceptronClassifier类完成了XOR函数近似和MNIST数据集的手写数字识别。

这是本书的最后一章。我们介绍许多模型，学习算法，效果评估方法，以及这些理论在scikit-learn中的实现。第一章，我们把机器学习描述成一种通过经验改善任务学习效果的过程。然后，我们通过一些例子演示常见的机器学习任务，学习经验和效果评估方法。我们介绍过匹萨价格与直接的回归案例和垃圾短信分类案例。我们通过颜色聚类压缩图片，聚类SURF描述器识别猫和狗的照片。我们用主成分分析做面部识别，用随机森林决策树拦截网页上的广告图片，还用支持向量机和人工神经网络识别手写数字。感谢你的阅读，希望你可以用scikit-learn和书中案例解决自己的问题。