神经网络，BP 算法，计算图模型，代码实现与详解

折 射

南加州大学 计算机科学硕士在读

用 pytorch 跟 tensorflow 实现神经网络固然爽，但是想要深入学习神经网络，光学会调包是不够的，还是得亲自动手去实现一个神经网络，才能更好去理解。

总之，废话少说，放码过来。

我这里只用了 numpy 这一个库，实现了一个简单的 2 层神经网络，解决经典的异或分类问题。

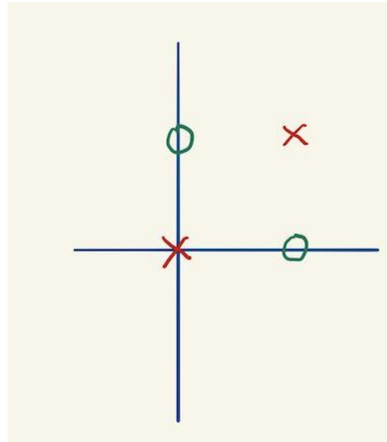
代码放在本人的 GitHub 上面：DeepLearningStudy。

这里默认大家对于神经网络有一定基础，假如完全不懂神经网络的原理以及 BP 算法的话，建议看一下我之前写过一篇 BP 算法纯理论推导的文章，虽然不看也没事。

一、问题介绍

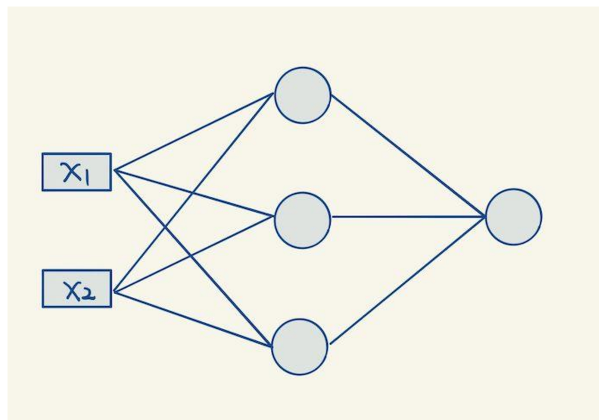
传说中线性分类器无法解决的异或分类问题。我们就拿它来作为我们神经网络的迷你训练数据。把输入数据拼成一个矩阵 X：

```
#训练数据：经典的异或分类问题
train_X = np.array([[0,0],[0,1],[1,0],[1,1]])
train_y = np.array([0,1,1,0])
```



异或分类的问题

我们定义一个简单的 2 层神经网络：



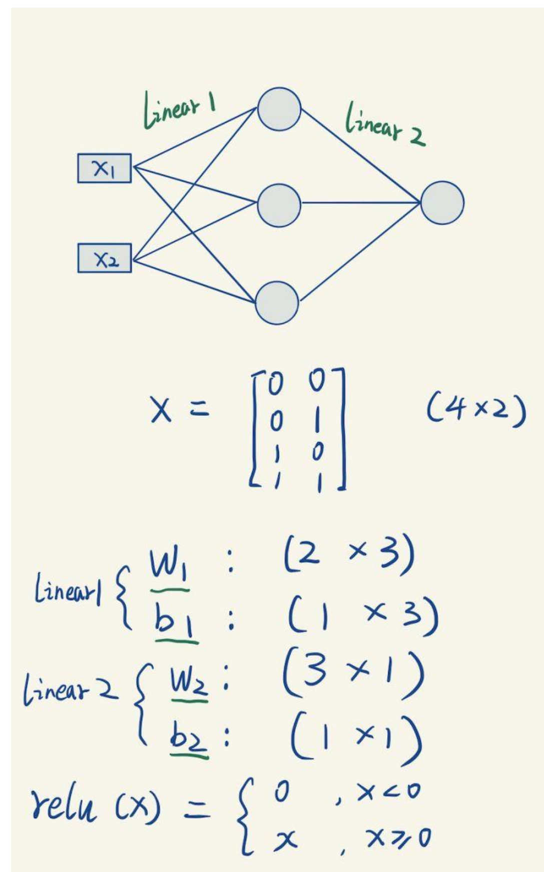
神经网络的结构

对应的代码

```
linear1 = LinearLayer(2,3)

relu1 = Relu()

linear2 = LinearLayer(3,1)
```



我们还需要定义一个损失函数 Loss，用来衡量我们的输出结果与实际结果的误差，这里用的是均方误差 MSE：

$$loss = \frac{1}{2} \cdot (y - \hat{y})^2$$

二、BP 算法 和 计算图（Computing Graph）理论

既然要自己实现神经网络。里面的线性层，Relu 层也得自己动手实现。我们需要首先知道计算图理论。

计算图理论是所有神经网络框架的核心理论基础。在理解了计算图理论以后，感觉 BP 算法突然之间变得非常容易了。

我们直接对着例子来讲解比较好。

上面的神经网络用纯数学函数来表达如下：

$$\hat{y} = relu(XW_1 + b_1) \cdot W_2 + b_2$$

计算图模型把一个复合运算拆分成为多个子运算，因此，我们需要引入很多中间变量。

定义：

a_i： 表示第 i 层网络的输入

O_i： 表示第 i 层网络的输出

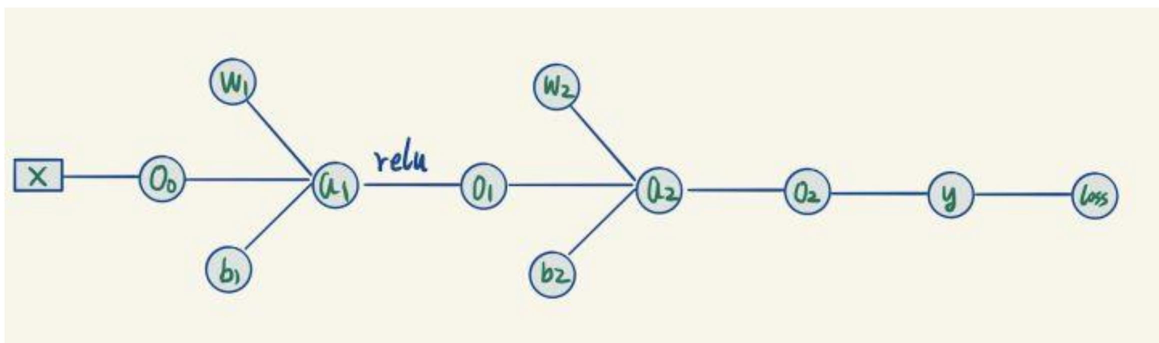
$$\begin{cases} O_0 = X \\ a_1 = O_0 W_1 + b_1, \quad O_1 = relu(a_1) \\ a_2 = O_1 W_2 + b_2, \quad O_2 = a_2 \\ y = O_2 \\ loss = \frac{1}{2}(y - \hat{y})^2 \end{cases}$$

a_i：第i层的输入

O_i：第i层的输出

这一堆公式等价于上面的神经网络的那一个公式

根据这些公式，我们就可以用全新的图来重新画之前神经网络图：



神经网络的计算图等价表示

这个图就是上面那一堆公式的等价结构表示。

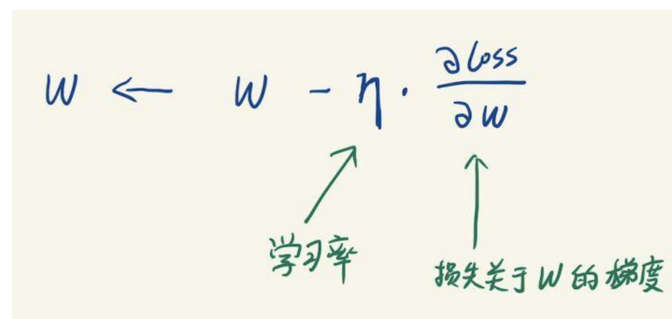
公式里面里面的每个出现过的变量，都视为一个节点。变量之间的连线描述了变量之间的计算的关系。你可以对照着图跟公式一个一个看。

这样的计算图有什么好处呢？下面我们基于这个计算图来用 BP 算法进行模型的训练。

对模型进行训练，就是找到一组模型的参数，使得我们的网络模型能够准确预测我们的训练数据。在我们这个例子里面，需要训练参数其实只有线性层的矩阵跟 bias 项： W_1 ，b_1 ，W_2 ，b_2 。

BP 算法采用梯度下降法来逐渐迭代更新权值。

梯度下降法的原理很简单，就是不断进行迭代，每次迭代用损失函数关于参数的梯度来更新当前的参数。



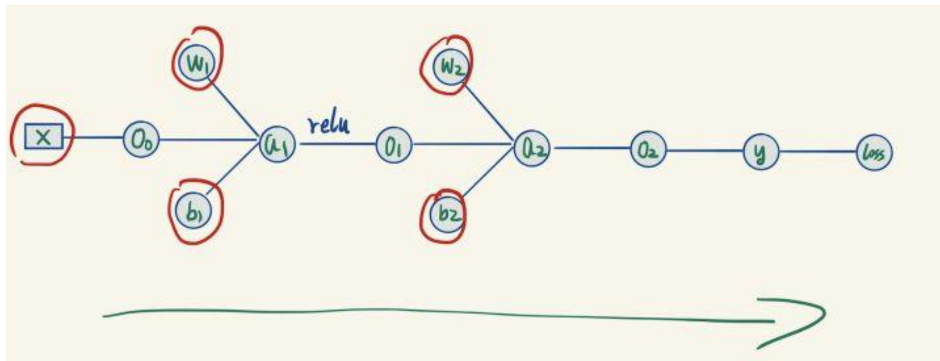
关于梯度我要多说几句。梯度表示 Y 关于 x 的变化率，可以理解成 x 的速度。由于 W1 是一个 2X3 的矩阵，那么 loss 关于 W1 的梯度可以理解 W1 的每个元素的瞬时速度。W1 的梯度的形状，必然是严格跟参数本身的形状是一样的（每个点都有对应的速度）。也就是说损失函数关于 W1 的梯度也必然是一个 2X3 的矩阵（不然更新公式里面无法做加减）。

下面开始正式训练

首先给定输入 X，同时初始化 W_1 ，b_1 ，W_2 ，b_2 （记住不能初始为全 0）。

正向传播(forward pass)

BP 算法首先在计算图上面进行正向传播(forward pass)，即从左到右计算所有未知量：



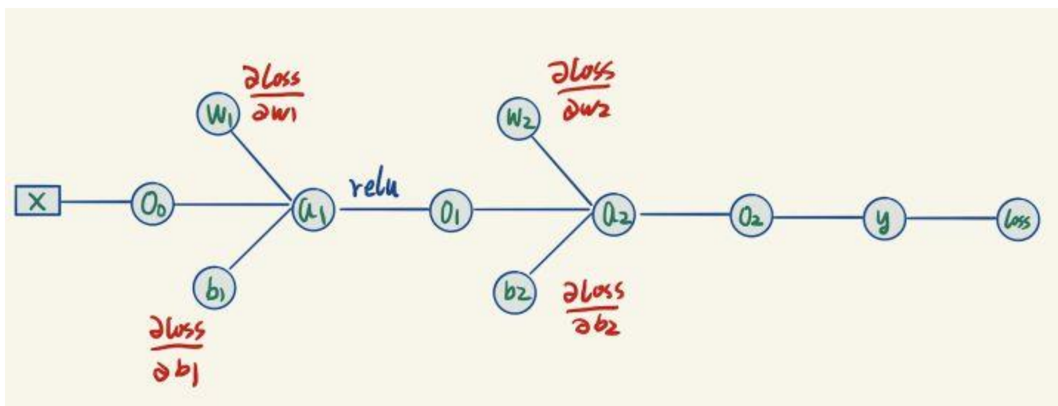
正向传播

只要严格按照顺序计算，里面所有的 O_i,a_i 都能先后求出，右边的 y 就是网络的预测结果。

反向传播(backward pass)

既然我们想要用参数的梯度来更新参数，那么我们需要求出最后的节点 loss 关于每个参数的梯度，求梯度的方法是进行反向传播。

现在我们已经进行过一次正向传播，因此图里面所有的量都变成已知的了。



我们现在要求的是图里面标为红色的这 4 个梯度，它们距离 loss 有点儿远。

但是不急，有了这个计算图，我们可以慢慢从右往左推出这 4 个值。

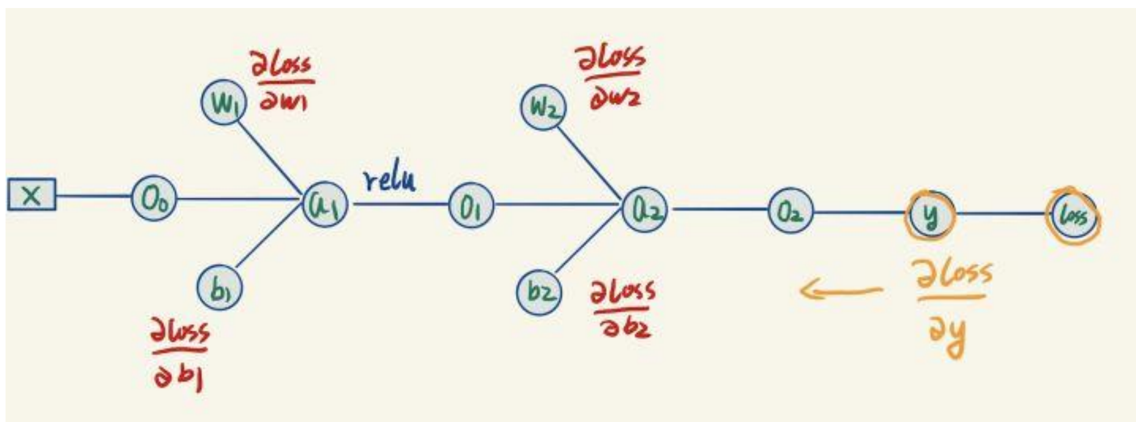
先从最右边开始，观察到 Loss 节点只有一条边跟 y 连着，首先计算 loss 关于 y 的导数（这个求导只有一个变量 y，怎么求不用我解释了吧）：

$$loss = \frac{1}{2}(y - \hat{y})^2$$

$$\frac{\partial loss}{\partial y} = y - \hat{y}$$

我们就求得了损失函数关于输出 y 的导数，然后继续往左边计算。

（已经求出的梯度我们用橙色来标记）



y 是通过 O2 计算出来的，我们可以计算 y 关于 O2 的梯度：

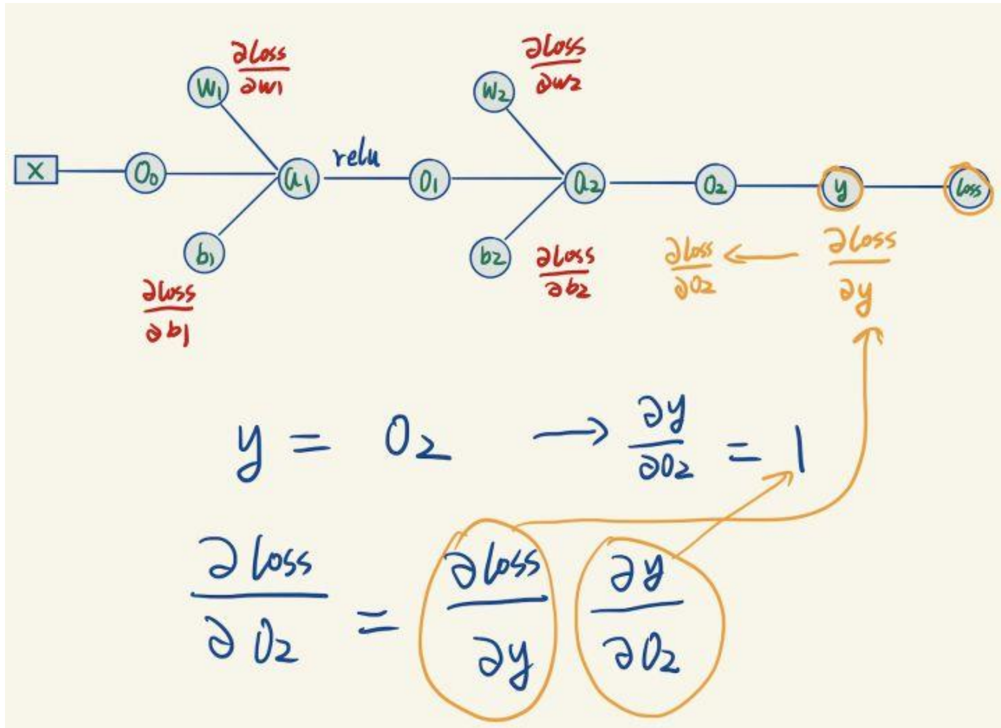
$$y = O_2 \longrightarrow \frac{\partial y}{\partial O_2} = 1$$

但是我们想要的是 loss 关于 O2 的梯度，这里应用到了链式求导法则：

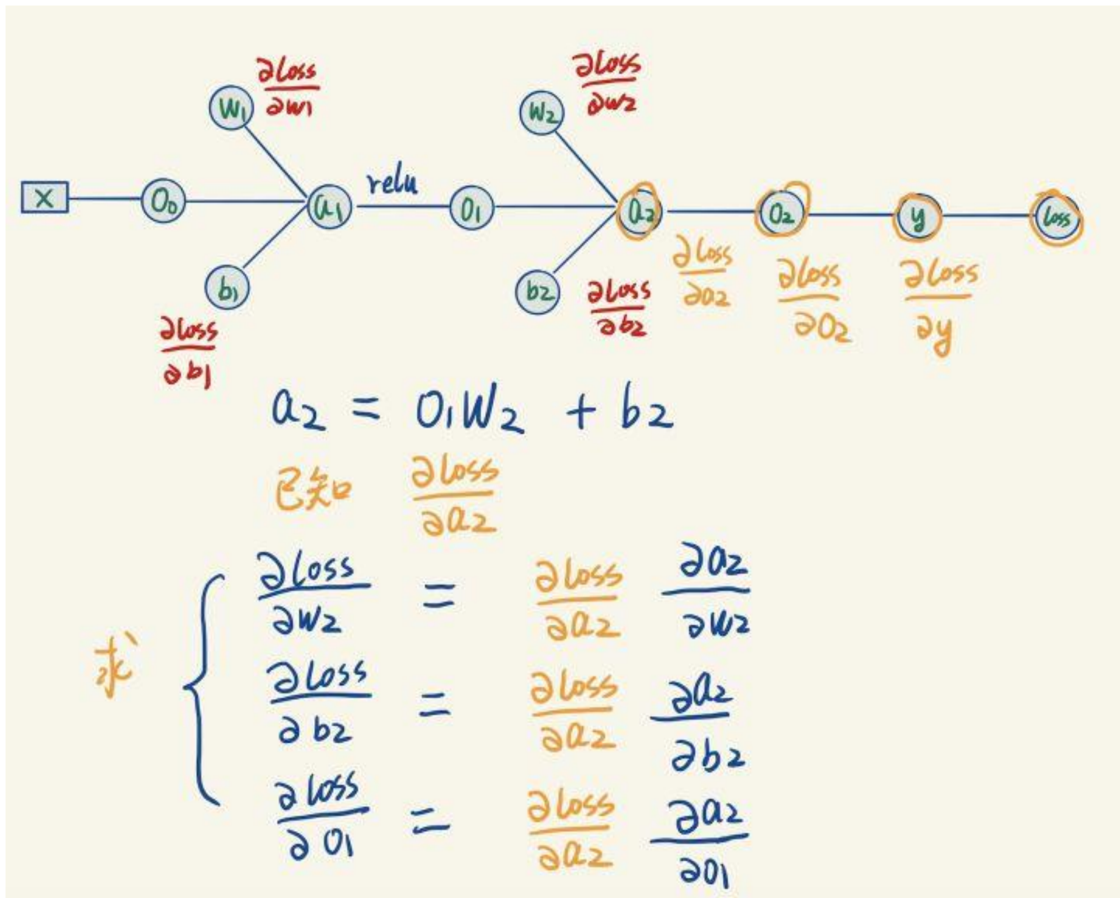
$$\frac{\partial loss}{\partial O_2} = \frac{\partial loss}{\partial y} \frac{\partial y}{\partial O_2}$$

Loss 关于 y 的梯度已经求过了，因此可以求出 loss 关于 O2 的梯度。

继续往左计算梯度：



$O_2=a_2$ 跟 $y=a_2$ 一样的道理。往左可以一路算到 loss 关于 a_2 的梯度。



上图中 a_2 关于它每个变量的梯度，可以直接根据 a_2 与它左边 3 个变量的表达式来算出，如下：

$$a_2 = o_1 w_2 + b_2$$

$$\begin{cases} \frac{\partial a_2}{\partial w_2} = o_1 \\ \frac{\partial a_2}{\partial o_1} = w_2 \\ \frac{\partial a_2}{\partial b_2} = 1 \end{cases}$$

这一步我们算出了两个需要计算的梯度，似乎并没有遇到困难，继续往左传播。

在计算 a_1 梯度的时候，我们遇到了 relu 激活函数，不要怕，relu 函数的梯度好求得很：

$$o_1 = relu(a_1)$$

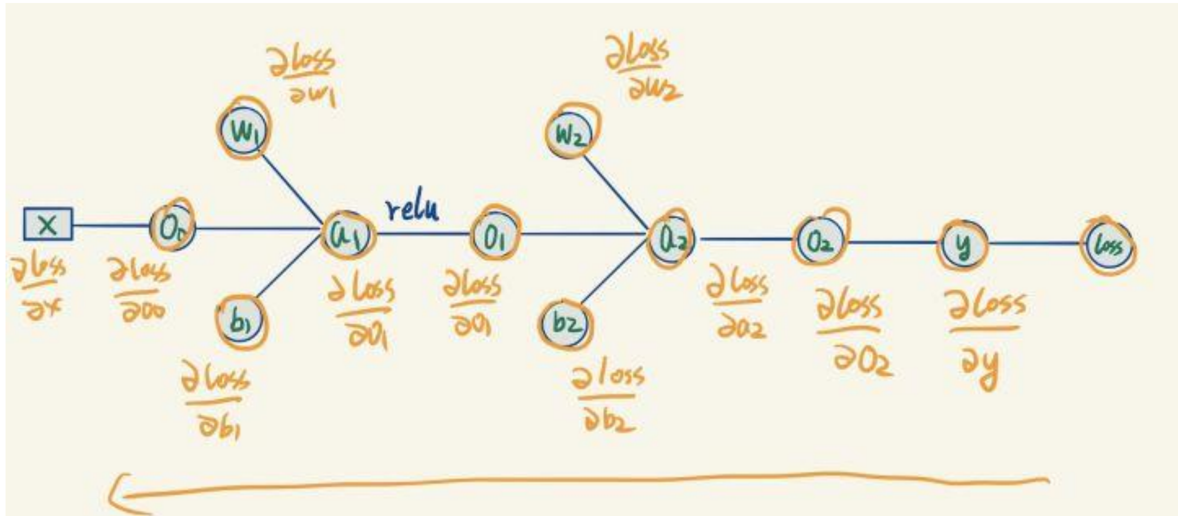
$$relu(x) = \begin{cases} 0, & x<0 \\ x, & x\geq 0 \end{cases}$$

$$\frac{\partial relu(x)}{\partial x} = \begin{cases} 0, & x<0 \\ 1, & x\geq 0 \end{cases}$$

它的导数就是，所有大于 0 的位置导数都是 1，其他位置导数都是 0，比如：

$$a_1 \begin{bmatrix} 1 & -1 \\ 2 & -5 \\ -6 & 4 \end{bmatrix} \qquad \frac{\partial relu(a_1)}{\partial a_1} = \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}$$

再往左继续传，我就不写每个步骤了。

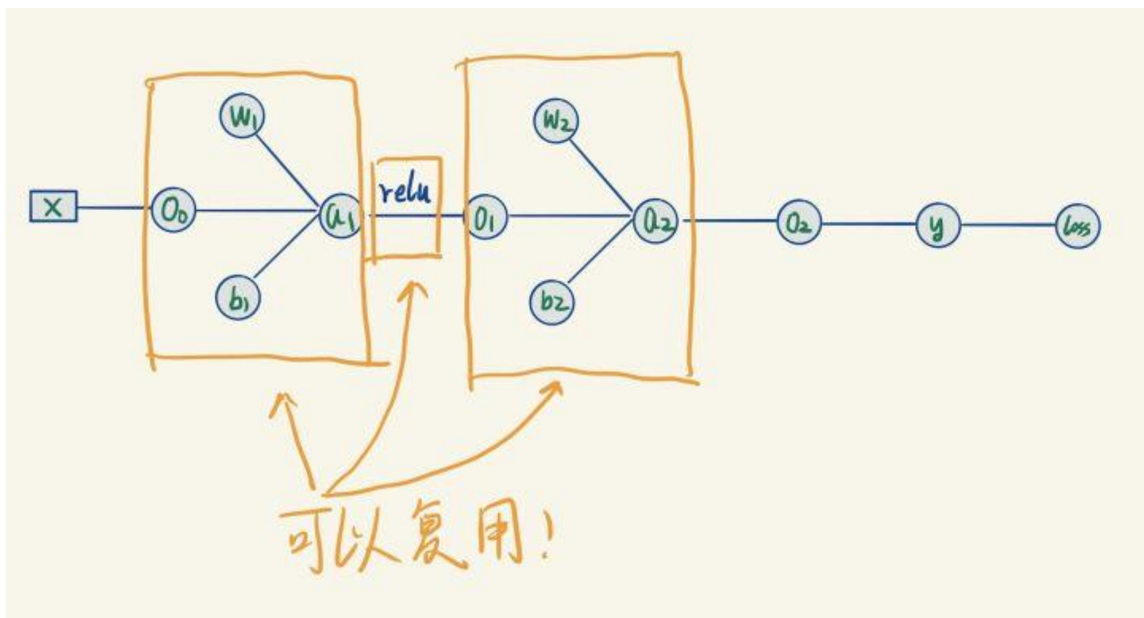


总之可以一直传到所有梯度都求出来为止。

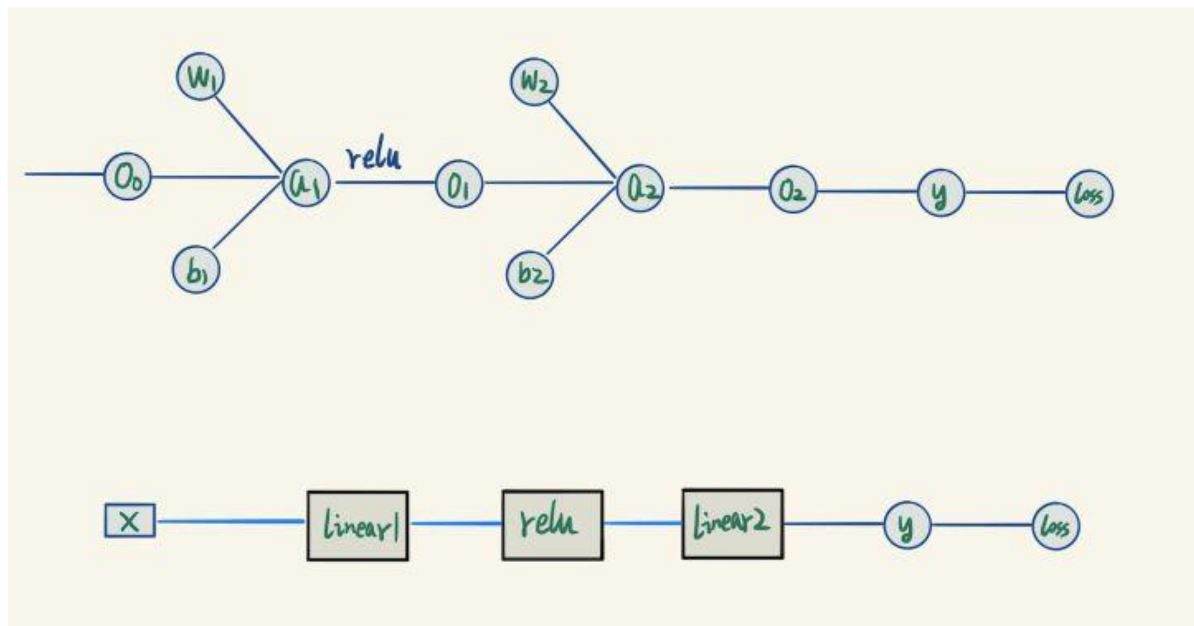
接下来一步就是愉快地进行随机梯度下降法的更新操作了。

三、模块化各种 Layer

观察我们的网络，发现里面的几个模块之间其实大部分干的事情都是相似的，无非就是层数不一样。那么我们就可以复用，我们完全可以把它们抽象成不同的 Layer：



于是，我们可以把这些类似模块看成一个小黑盒子，我们的模型等价于下面这个：



于是上面那个复杂的网状结构，被我们简化成了线性结构。

下面我对照代码实现每个小黑盒子吧，实现代码在这个文件里面：

Layers.py

首先介绍线性全连接层，先看代码吧：

```
class LinearLayer:
    def __init__(self, input_D, output_D):
        self._W = np.random.normal(0, 0.1, (input_D, output_D)) #
初始化不能为全0
        self._b = np.random.normal(0, 0.1, (1, output_D))
        self._grad_W = np.zeros((input_D, output_D))
        self._grad_b = np.zeros((1, output_D))

    def forward(self, X):
        return np.matmul(X, self._W) + self._b

    def backward(self, X, grad):
        self._grad_W = np.matmul( X.T, grad)
        self._grad_b = np.matmul(grad.T, np.ones(X.shape[0]))
        return np.matmul(grad, self._W.T)
```

```
    def update(self, learn_rate):
        self._W = self._W - self._grad_W * learn_rate
        self._b = self._b - self._grad_b * learn_rate
```

forward 太简单了，就不讲了，看一下 backward。

backward 里面其实要计算 3 个值，W, b 的梯度算完以后要存起来，前一层的梯度算完以后直接作为返回值传出去，推导的公式如下：

注意矩阵求导应用链式法则的时候，顺序非常重要。要严格按照指定顺序来乘，不然形状对不上。具体什么顺序，可以自己想办法慢慢拼凑出来。

还有一个 update 函数，调用此函数这一层会按照梯度下降法来更新它的 W 跟 b 的值，这个实现也很简单直接看代码就明白了。

然后实现 Relu 层：

```
class Relu:
    def __init__(self):
        pass
```

```
    def forward(self, X):
        return np.where(X < 0, 0, X)

    def backward(self, X, grad):
        return np.where(X > 0, X, 0) * grad
```

由于这一层没有需要保存参数，只需要实现以下 forward 跟 backward 方法就行了，非常简单。

接下来开始实现神经网络训练。

四、搭建神经网络

训练部分的代码在 nn.py 里面，里面的代码哪里看不懂可以翻回去看之前的解释，命名都是跟上面说的一样的。

```
#训练数据：经典的异或分类问题
train_X = np.array([[0,0],[0,1],[1,0],[1,1]])
train_y = np.array([0,1,1,0])

#初始化网络，总共 2 层，输入数据是 2 维，第一层 3 个节点，第二层 1 个节点作为输出层，激活函数使用 Relu
linear1 = LinearLayer(2,3)
relu1 = Relu()
linear2 = LinearLayer(3,1)

#训练网络
for i in range(10000):

    #前向传播 Forward，获取网络输出
    o0 = train_X
    a1 = linear1.forward(o0)
    o1 = relu1.forward(a1)
    a2 = linear2.forward(o1)
    o2 = a2
```

```
    #获得网络当前输出，计算损失 loss
    y = o2.reshape(o2.shape[0])
    loss = MSELoss(train_y, y) # MSE 损失函数


    #反向传播，获取梯度
    grad = (y - train_y).reshape(result.shape[0],1)
    grad = linear2.backward(o1, grad)
    grad = relu1.backward(a1, grad)
    grad = linear1.backward(o0, grad)


    learn_rate = 0.01  #学习率

    #更新网络中线性层的参数
    linear1.update(learn_rate)
    linear2.update(learn_rate)

    #判断学习是否完成
    if i % 200 == 0:
        print(loss)
    if loss < 0.001:
        print("训练完成！ 第%d 次迭代" %(i))
        break
```

我觉得没啥好讲的，就直接对着我们的计算图，一步一步来。

注意一下中间过程几个向量的形状。列向量跟行向量是不一样的，一不小心把列向量跟行向量做运算，numpy 不会报错，而是会广播成一个矩阵。所以运算的之前，记得该转置得转置。

```
#将训练好的层打包成一个 model
model = [linear1, relu1, linear2]

#用训练好的模型去预测
def predict(model, X):
    tmp = X
    for layer in model:
        tmp = layer.forward(tmp)
    return np.where(tmp > 0.5, 1, 0)
```

把模型打包然后用上面的 predict 函数来预测。也没啥好说的，就直接往后一直 forward 就完儿事。

```
#开始预测
print("-----")
X = np.array([[0,0],[0,1],[1,0],[1,1]])
result = predict(model, X)
print("预测数据 1")
print(X)
print("预测结果 1")
print(result)
```

预测训练完的网络就能拿去搞预测了，我这里设置学习率为 0.01 的情况下，在第 3315 次迭代时候完成训练。

最后我们成功地预测了训练数据。

```
0.0025150371478791156
训练完成!  第3315次迭代
-----
预测数据1
[[0 0]
 [0 1]
 [1 0]
 [1 1]]
预测结果1
[[0]
 [1]
 [1]
 [0]]
```

五、总结

本文出现过的所有图片笔记的 PDF、还有代码，全都在我的 github 上：

denggaoshan/DeepLearningStudy

这个里面代码欢迎各位拿去作学习用。以后有空可能会往里面加上卷积层，RNN，attention 之类的比较经典的代码。如果有关于文章或者代码有任何不懂的地方，欢迎评论。