最通俗的方式解释神经网络的数学过程

深度学习算法与计算机视觉 2019-07-14

选自Medium
转载自机器之心，未经允许禁止二次转载
作者：Omar U. Florez
参与：Nurhachu Null、张倩

【导读】模型的训练、调参是一项非常费时费力的工作，了解神经网络内部的数学原理有利于快速找出问题所在。本文作者从零开始，一步一步讲解了训练神经网络时所用到的数学过程。

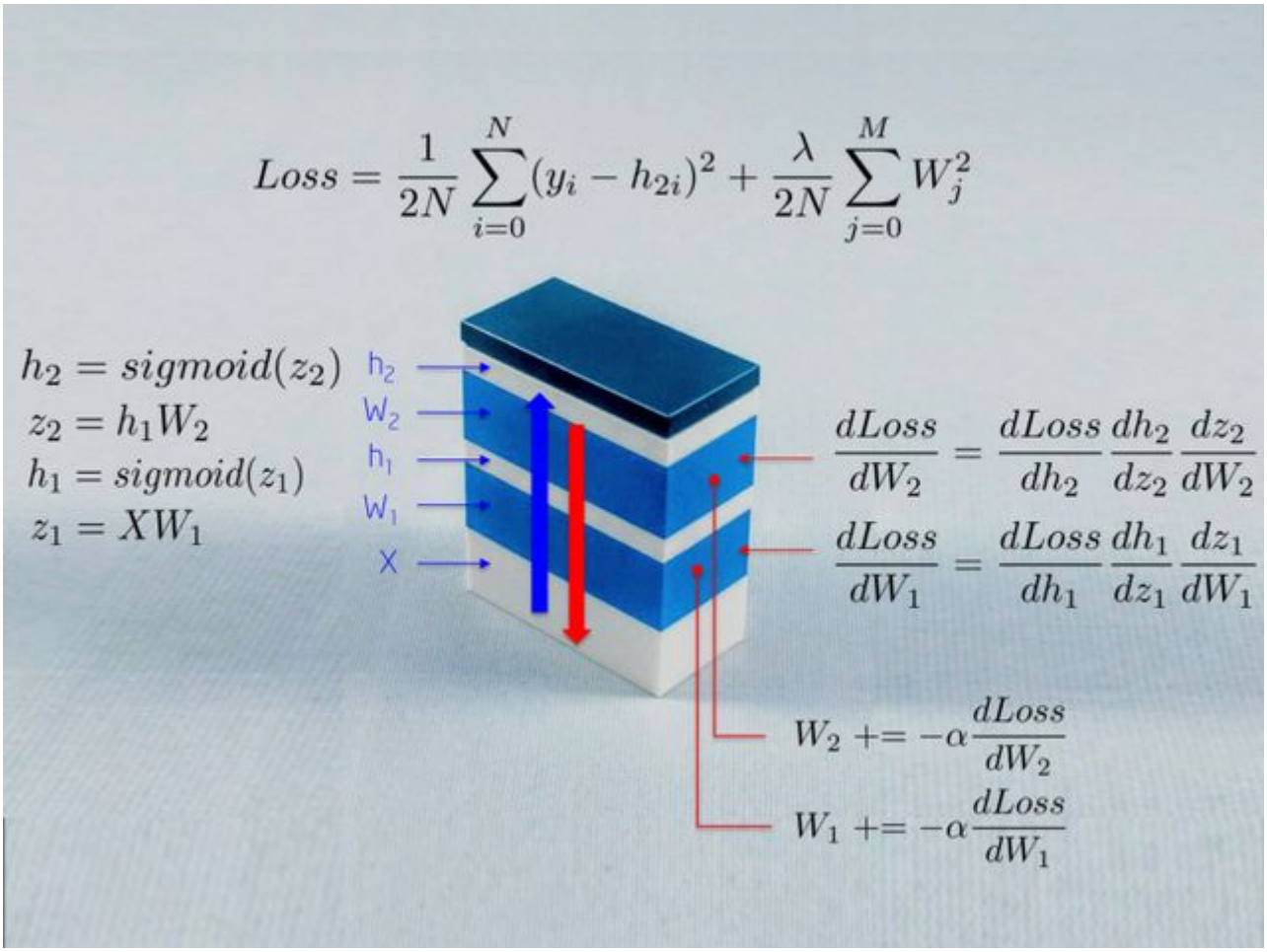
神经网络是线性模块和非线性模块的巧妙排列。当聪明地选择并连接这些模块时，我们就得到了一个强大的工具来逼近任何一个数学函数，如一个能够借助非线性决策边界进行分类的神经网络。

运行代码的步骤如下：

```
git clone https://github.com/omar-florez/scratch_mlp/
python scratch_mlp/scratch_mlp.py
```

尽管 反向传播技术具有直观、模块化的特质，但是它负责更新可训练的参数，这是一个一直未被深入解释的主题。让我们以乐高积木为喻，一次增加一块，从零构建一个神经网络来一探其内部功能。

神经网络就像是由乐高积木组成的



上图描述了训练一个神经网络时所用到的部分数学过程。我们将在本文中解释这个。读者可能感到有趣的一点是：一个神经网络就是很多模块以不同的目标堆叠起来。

- 输入变量 X 向神经网络馈送原始数据，它被存储在一个矩阵中，矩阵的行是观察值，列是维度。
- 权重 W_1 将输入 X 映射到第一个隐藏层 h_1。然后权重 W_1 充当一个线性核。
- Sigmoid 函数防止隐藏层中的数字落到 0-1 的范围之外。结果就是一个神经激活的数组，h_1 = Sigmoid(WX)。

此时，这些运算只是组成了一个一般线性系统，无法对非线性交互建模。当我们再叠加一层，给模块的结构增加深度的时候这一点就会改变。网络越深，我们就会学到越多微妙的非线性交互，能解决的问题也就越复杂，或许这也是深度神经模型兴起的原因之一。

为什么我要读这篇文章?

如果你理解一个神经网络的内部部分，你就能够在遇到问题的时候快速知道先去改变哪里，并且能够制定策略来测试你所知道的这个算法的部分不变量和预期的行为。

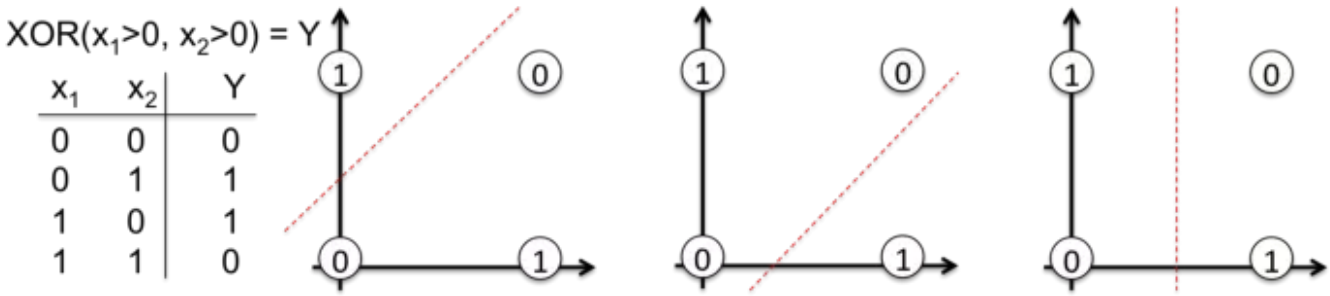
因为调试机器学习模型是一项复杂的任务。根据经验，数学模型在首次尝试的时候不会奏效。它们可能会对新数据给出较低的准确率，会耗费很长的训练时间或者太多的内存，返回一个很大的错误负数值或者 NAN 的预测……在有些情况下，了解算法的运行机制可以让我们的任务变得更加便利：

- 如果训练花费了太多的时间，那增加 minibatch 的大小或许是一个好主意，这能够减小观察值的方差，从而有助于算法收敛。
- 如果你看到了 NAN 的预测值，算法可能接收到了大梯度，产生了内存溢出。可以将这个视为在很多次迭代之后发生爆炸的矩阵乘法。减小学习率可以缩小这些数值。减少层数能够减少乘法的数量。剪切梯度也能够明显地控制这个问题。

具体的例子：学习异或函数

让我们打开黑盒子。我们现在要从零开始构建一个学习 异或函数的神经网络。选择这个非线性函数可绝对不是随机的。没有反向传播的话，就很难学会用一条直线分类。

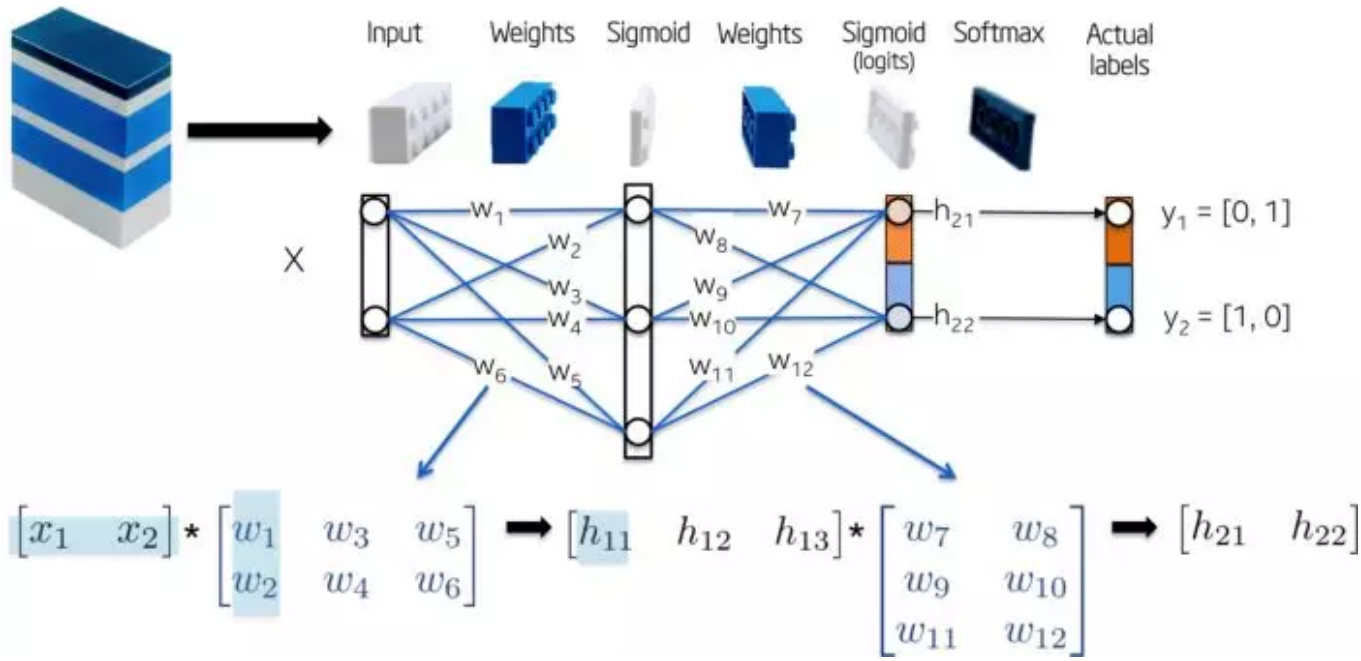
为了描述这个重要的概念，请注意下图中，一条直线是为何不能对异或函数输出中的 0 和 1 进行分类。现实生活中的问题也是非线性可分的。



这个网络的拓扑结构非常简单：

- 输入变量 X 是二维向量
- 权重 W_1 是具有随机初始化数值的 2x3 的矩阵
- 隐藏层 h_1 包含 3 个神经元。每个神经元接受观察值的加权和作为输入，这就是下图中绿色高亮的内积：z_1 = [x_1, x_2][w_1, w_2]
- 权重 W_2 是具有随机初始化值的 3x2 的矩阵
- 输出层 h_2 包含两个神经元，因为异或函数的输出要么是 0(y_1=[0,1])，要么是 1(y_2 = [1,0])

下图更加直观：

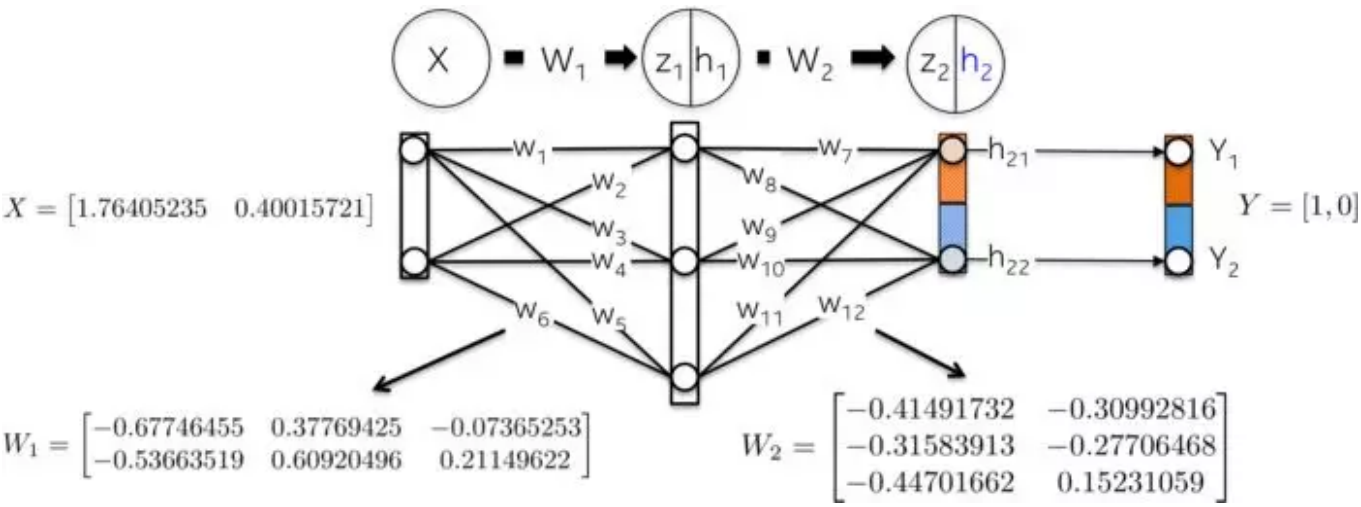


我们现在来训练这个模型。在我们这个简单的例子中，可训练的参数就是权重，但是应该知道的是，目前的研究正在探索更多可以被优化的参数类型。例如层之间的快捷连接、 正则化分布、拓扑结构、残差、学习率等等。

反向传播是这样的一种方法：在给定的一批具有标签的观察值上，朝着将预定义的错误指标（就是损失函数）最小化的方向（梯度）更新权重。该算法已经多次被重复发现，这是另一种更通用的被称为自动微分的技术在反向积累模式下的特例。

网络初始化

让我们用随机数来初始化网络权重



前向步骤：

这一步的目标就是把输入变量 X 向前传递到网络的每一层，直至计算出输出层 h_2 的向量。

这就是其中发生的计算过程：

以权重 W_1 为线性核对输入数据 X 做线性变换：

$$z_1 = XW_1$$

$$z_1 = \begin{bmatrix} 1.76405235 & 0.40015721 \end{bmatrix} \begin{bmatrix} -0.67746455 & 0.37769425 & -0.07365253 \\ -0.53663519 & 0.60920496 & 0.21149622 \end{bmatrix}$$

$$z_1 = \begin{bmatrix} -1.40982136 & 0.91005018 & -0.04529517 \end{bmatrix}$$

使用 Sigmoid 激活函数对加权和进行缩放，得到了第一个隐藏层 h_1 的值。请注意，原始的 2D 向量现在映射到了 3D 空间。

$$h_1 = sigmoid(z_1)$$

$$h_1 = \begin{bmatrix} 0.19626223 & 0.71301043 & 0.48867814 \end{bmatrix}$$

第 2 层 h_2 中发生了类似的过程。让我们首先来计算第一个隐藏层的加权和 z_2，它现在是输入数据。

$$z_2 = h_1 W_2$$

$$z_2 = \begin{bmatrix} 0.19626223 & 0.71301043 & 0.48867814 \end{bmatrix} \begin{bmatrix} -0.41491732 & -0.30992816 \\ -0.31583913 & -0.27706468 \\ -0.44701662 & 0.15231059 \end{bmatrix}$$

$$z_2 = \begin{bmatrix} -0.52507645 & -0.18394635 \end{bmatrix}$$

然后计算它们的 Sigmoid 激活函数。向量 [0.37166596 0.45414264] 代表的是网络对给定的输入 X 计算出的对数概率或者预测向量。

$$h_2 = sigmoid(z_2)$$

$$h_2 = \begin{bmatrix} 0.37166596 & 0.45414264 \end{bmatrix}$$

计算整体损失

也被称为「实际值减去预测值」，这个损失函数的目标就是量化预测向量 h_2 和人工标签 y 之间的距离。

请注意，这个损失函数包括一个正则项，它以岭回归的形式惩罚较大的权重。换言之，平方值比较大的权重会增大损失函数，而这正是我们希望最小化的指标。

$$Loss = \frac{1}{2N}\sum_{i=0}^{N}(y_i - h_{2i})^2 + \frac{\lambda}{2N}\sum_{j=0}^{M} W_j^2$$

反向步骤：

这一步的目标就是沿着最小化损失函数的方向更新神经网络的权重。正如我们将要看到的，这是一个递归算法，它可以重用之前计算出来的梯度，而且严重依赖微分函数。因为这些更新减小了损失函数，所以一个神经网络便「学会了」去逼近具有已知类别的观察值的标签。这就是被称作泛化的一种属性。

与前向步骤不同的是，这个步骤沿着反向的顺序进行。它首先计算出输出层中损失函数对每个权重的偏导数 (dLoss/dW_2)，然后计算隐藏层的偏导数 (dLoss/dW1)。让我们详细地解释每个导数吧。

dLoss/dW_2:

链式法则表明，我们可以将一个神经网络的梯度计算分解成好多个微分部分：

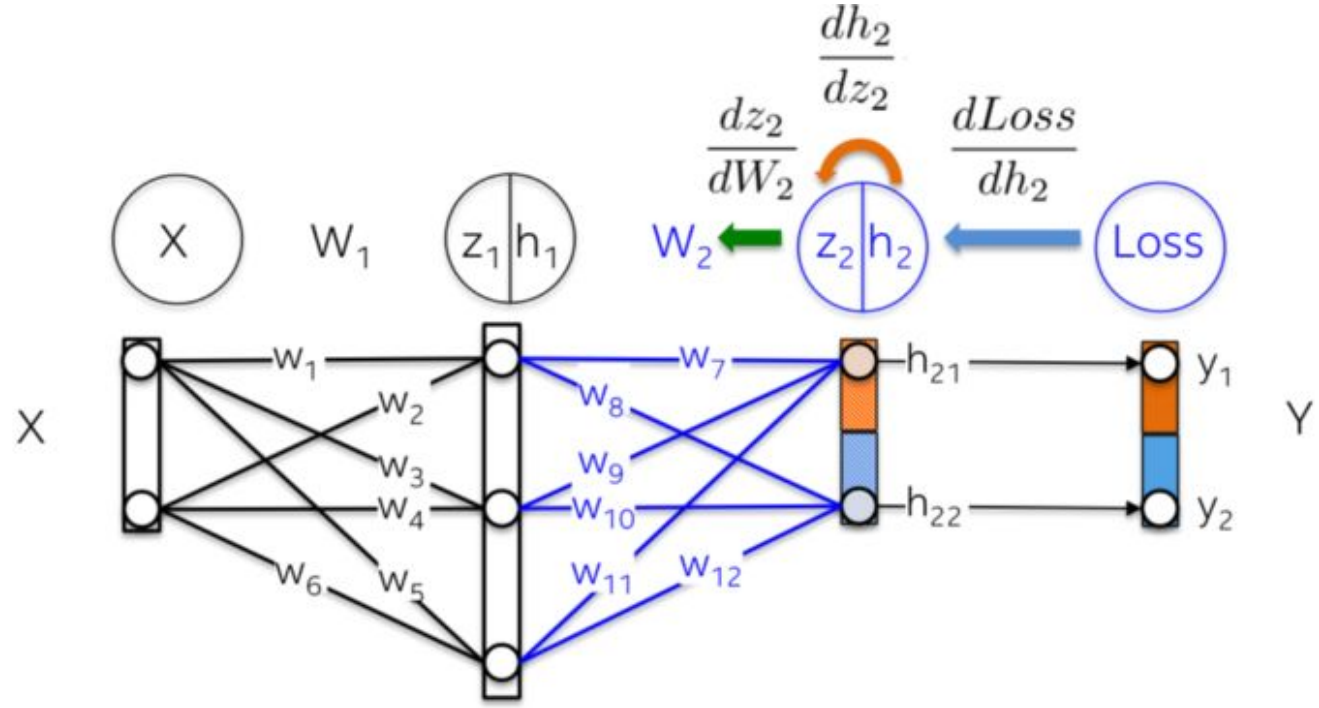
$$\frac{dLoss}{dW_2} = \frac{dLoss}{dh_2}\frac{dh_2}{dz_2}\frac{dz_2}{dW_2}$$

$$\frac{dLoss}{dh_2} = -(y - h_2) \qquad \frac{dh_2}{dz_2} = h_2(1 - h_2) \qquad \frac{dz_2}{dW_2} = h_1$$

为了帮助记忆，下表列出了上面用到的一些函数定义以及它们的一阶导数：

Function	First derivative
Loss = (y-h2)^2	dLoss/dW2 = -(y-h2)
h2 = Sigmoid(z2)	dh2/dz2 = h2(1-h2)
z2 = h1W2	dz2/dW2 = h1
z2 = h1W2	dz2/dh1 = W2

更直观地，我们在下图中要更新权重 W_2（蓝色部分）。为了做到这件事，我们需要沿着导数链计算三个偏导数。



将数值代入到这些偏导数中，我们就能够计算出 W_2 的偏导数，如下所示：

$$\frac{dLoss}{dW_2} = \frac{dLoss}{dh_2}\frac{dh_2}{dz_2}\frac{dz_2}{dW_2}$$

$$\frac{dLoss}{dh_2} = -(y - h_2) \qquad \frac{dh_2}{dz_2} = h_2(1 - h_2) \qquad \frac{dz_2}{dW_2} = h_1$$

$$\begin{aligned}\frac{dLoss}{dh_2} &= -(y - h_2) \\ &= -(\begin{bmatrix}1 & 0\end{bmatrix} - \begin{bmatrix}0.37166596 & 0.45414264\end{bmatrix}) \\ &= \begin{bmatrix}-0.62833404 & 0.45414264\end{bmatrix}\end{aligned}$$

$$\begin{aligned}\frac{dh_2}{dz_2} &= h_2(1 - h_2) \\ &= \begin{bmatrix}0.37166596 & 0.454142644\end{bmatrix}(1 - \begin{bmatrix}0.37166596 & 0.454142644\end{bmatrix}) \\ &= \begin{bmatrix}-0.23353037 & 0.2478971\end{bmatrix}\end{aligned}$$

$$\begin{aligned}\frac{dz_2}{dW_2} &= h_1 \\ &= \begin{bmatrix}0.19626223 & 0.71301043 & 0.48867814\end{bmatrix}\end{aligned}$$

结果是一个 3x2 的矩阵 dLoss/dW_2，它将会沿着最小化损失函数的方向更新 W_2 的数值。

$$\frac{dLoss}{dW_2} = \frac{dLoss}{dh_2}\frac{dh_2}{dz_2}\frac{dz_2}{dW_2}$$

$$\frac{dLoss}{dh_2} = -(y - h_2) \qquad \frac{dh_2}{dz_2} = h_2(1 - h_2) \qquad \frac{dz_2}{dW_2} = h_1$$

$$\frac{dLoss}{dW_2} = \begin{bmatrix} 0.19626223 & 0.71301043 & 0.48867814 \end{bmatrix}^T \begin{bmatrix} -0.23353037 & 0.2478971 \end{bmatrix} \begin{bmatrix} -0.62833404 & 0.45414264 \end{bmatrix}$$

$$= \begin{bmatrix} -0.02879856 & 0.02209533 \\ -0.10462365 & 0.08027117 \\ -0.07170623 & 0.0550157 \end{bmatrix}$$

$$W_2^* = W_2 - \alpha\frac{dLoss}{dW_2} = \begin{bmatrix} -0.41491732 & -0.30992816 \\ -0.31583913 & -0.27706468 \\ -0.44701662 & 0.15231059 \end{bmatrix} - 0.001 \begin{bmatrix} -0.02879856 & 0.02209533 \\ -0.10462365 & 0.08027117 \\ -0.07170623 & 0.0550157 \end{bmatrix}$$

$$W_2^* = \begin{bmatrix} -0.41488852 & -0.30995026 \\ -0.31573451 & -0.27714495 \\ -0.44694491 & 0.15225557 \end{bmatrix}$$

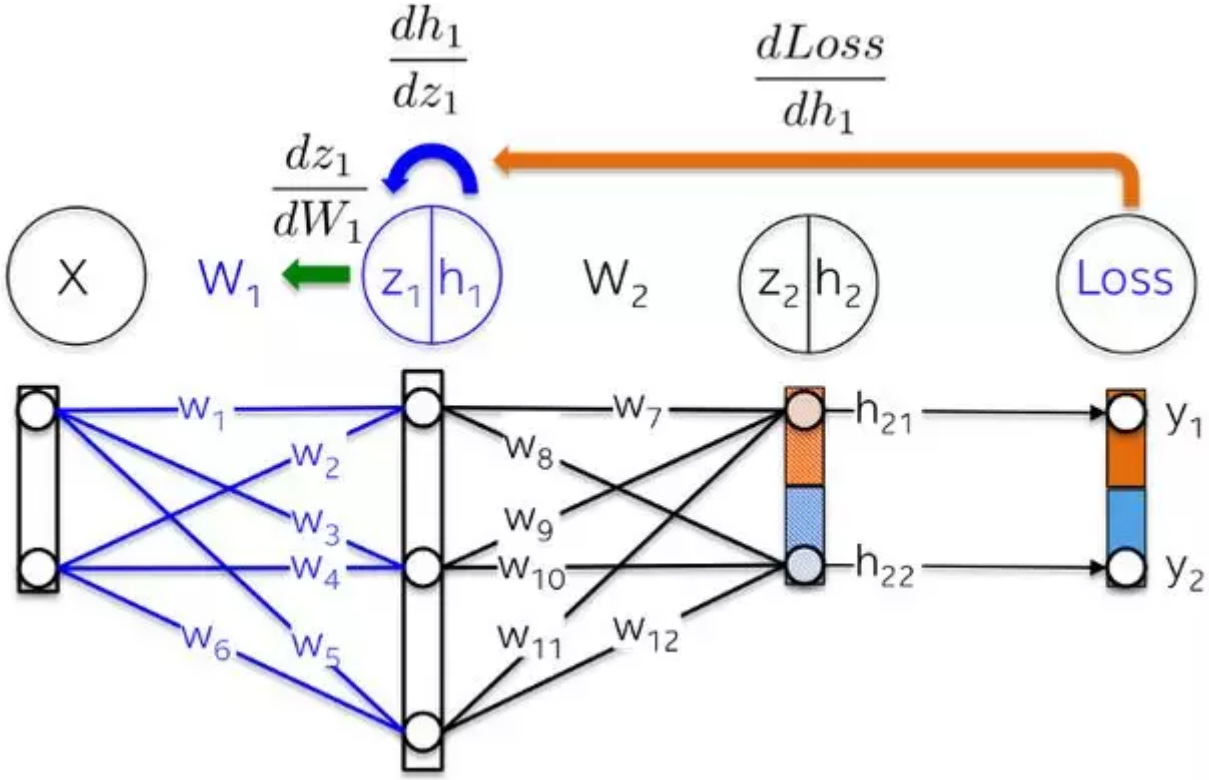
dLoss/dW_1:

计算用于更新第一个隐藏层 W_1 权重的链式规则就展现了重复使用已有计算结果的可能。

$$\frac{dLoss}{dW_1} = \frac{dLoss}{dh_1}\frac{dh_1}{dz_1}\frac{dz_1}{dW_1} \qquad \frac{dLoss}{dW_2} = \frac{dLoss}{dh_2}\frac{dh_2}{dz_2}\frac{dz_2}{dW_2}$$

$$\frac{dLoss}{dh_1} = \frac{dLoss}{dz2}\frac{dz2}{dh1}$$

更直观地，从输出层到权重 W_1 的路径会碰到在后面层中早已计算出来的偏导数。



例如，偏导数 dLoss/dh_2 和 dh_2/dz_2 在上一节中已经被计算为输出层 dLoss/dW_2 学习权值的依赖项。

$$\frac{dLoss}{dW_1} = \frac{dLoss}{dh_1}\frac{dh_1}{dz_1}\frac{dz_1}{dW_1} \qquad \frac{dLoss}{dW_2} = \frac{dLoss}{dh_2}\frac{dh_2}{dz_2}\frac{dz_2}{dW_2}$$

$$\frac{dLoss}{dh_1} = \frac{dLoss}{dz2}\frac{dz2}{dh1}$$

$$\begin{aligned}
\frac{dLoss}{dh_1} &= \frac{dLoss}{dz2}\frac{dz2}{dh1} \\
&= [-(y-h_2)h_2(1-h_2)][W_2] \\
&= \begin{bmatrix}0.23353037 & 0.2478971\end{bmatrix}\begin{bmatrix}-0.41491732 & -0.30992816 \\ -0.31583913 & -0.27706468 \\ -0.44701662 & 0.15231059\end{bmatrix} \\
&= \begin{bmatrix}0.02599101 & 0.01515256 & 0.08274025\end{bmatrix}
\end{aligned}$$

$$\begin{aligned}
\frac{dh_1}{dz_1} &= h_1(1-h_1) \\
&= \begin{bmatrix}0.19626223 & 0.71301043 & 0.48867814\end{bmatrix}(1-\begin{bmatrix}0.19626223 & 0.71301043 & 0.48867814\end{bmatrix}) \\
&= \begin{bmatrix}0.15774337 & 0.20462656 & 0.24987182\end{bmatrix}
\end{aligned}$$

$$\frac{dz_1}{dW_1} = X = \begin{bmatrix}1.76405235 & 0.40015721\end{bmatrix}$$

将所有的导数放在一起，我们就能够再一次执行链式法则，来为隐藏层的 W_1 更新权重。

$$\frac{dLoss}{dW_1} = \begin{bmatrix}1.76405235 & 0.40015721\end{bmatrix}^T \begin{bmatrix}0.02599101 & 0.01515256 & 0.08274025\end{bmatrix} \begin{bmatrix}0.15774337 & 0.20462656 & 0.24987182\end{bmatrix}$$

$$= \begin{bmatrix}0.00723246 & 0.00546965 & 0.03647082\\ 0.00164061 & 0.00124073 & 0.00827303\end{bmatrix}$$

$$W_1^* = W_1 - \alpha \frac{dLoss}{dW_1} = \begin{bmatrix}-0.67746455 & 0.37769425 & -0.07365253\\ -0.53663519 & 0.60920496 & 0.21149622\end{bmatrix} - 0.001 \begin{bmatrix}0.00723246 & 0.00546965 & 0.03647082\\ 0.00164061 & 0.00124073 & 0.00827303\end{bmatrix}$$

$$W_1^* = \begin{bmatrix}-0.67747178 & 0.37768878 & -0.073689\\ -0.53663683 & 0.60920372 & 0.21148795\end{bmatrix}$$

最后，我们给权重赋予新的数值，完成了对神经网络的一步训练。

$$W1 = W1^*$$

$$W2 = W2^*$$

实现

让我们仅使用 numpy 作为线性代数引擎来将上面的数学方程转换成代码。神经网络在一个循环中进行训练，其中每次迭代会给神经网络展示标准的输入数据。在这个小例子中，我们只考虑每次迭代中的整个数据集。前向步骤、损失函数和反向步骤的计算会得到比较好的泛化，因为我们在每一次循环中都用它们对应的梯度（矩阵 dL_dw1 和 dL_dw2）来更新可训练的参数。

代码保存在这个 repo 中：https://github.com/omar-florez/scratch_mlp

```
#
# Initialize weights:
np.random.seed(2017)
w1 = 2.0*np.random.random((input_dim, hidden_dim))-1.0      #w0=(2,hidden_dim)
w2 = 2.0*np.random.random((hidden_dim, output_dim))-1.0     #w1=(hidden_dim,2)

#Calibrating variances with 1/sqrt(fan_in)
w1 /= np.sqrt(input_dim)
w2 /= np.sqrt(hidden_dim)
for i in range(num_epochs):
    index = np.arange(X.shape[0])[:N]
    #
    # Forward step:
    h1 = sigmoid(np.matmul(X[index], w1))                   #(N, 3)
    logits = sigmoid(np.matmul(h1, w2))                     #(N, 2)
    probs = np.exp(logits)/np.sum(np.exp(logits), axis=1, keepdims=True)
    h2 = logits
    #
    # Definition of Loss function: mean squared error plus Ridge regularization
    L = np.square(y[index]-h2).sum()/(2*N) + reg_coeff*(np.square(w1).sum()+np.square(w2).sum())/(2*N)
    #
    # Backward step: Error = W_l e_l+1 f'_l
    #       dL/dw2 = dL/dh2 * dh2/dz2 * dz2/dw2
    dL_dh2 = -(y[index] - h2)                               #(N, 2)
    dh2_dz2 = sigmoid(h2, first_derivative=True)            #(N, 2)
    dz2_dw2 = h1                                            #(N, hidden_dim)
    #Gradient for W2:  (hidden_dim,N)x(N,2)=(N,2)
    dL_dw2 = dz2_dw2.T.dot(dL_dh2*dh2_dz2) + reg_coeff*np.square(w2).sum()

    #dL/dw1 = dL/dh1 * dh1/dz1 * dz1/dw1
    #       dL/dh1 = dL/dz2 * dz2/dh1
    #       dL/dz2 = dL/dh2 * dh2/dz2
    dL_dz2 = dL_dh2 * dh2_dz2                               #(N, 2)
    dz2_dh1 = w2                                            #z2 = h1*w2
    dL_dh1 =  dL_dz2.dot(dz2_dh1.T)                         #(N,2)x(2, hidden_dim)=(N, hidden_dim)
    dh1_dz1 = sigmoid(h1, first_derivative=True)            #(N,hidden_dim)
    dz1_dw1 = X[index]                                      #(N,2)
    #Gradient for W1:  (2,N)x((N,hidden_dim)*(N,hidden_dim))
    dL_dw1 = dz1_dw1.T.dot(dL_dh1*dh1_dz1) + reg_coeff*np.square(w1).sum()
    #
    #weight updates:
    w2 += -learning_rate*dL_dw2
    w1 += -learning_rate*dL_dw1
```

Input calibration

Forward

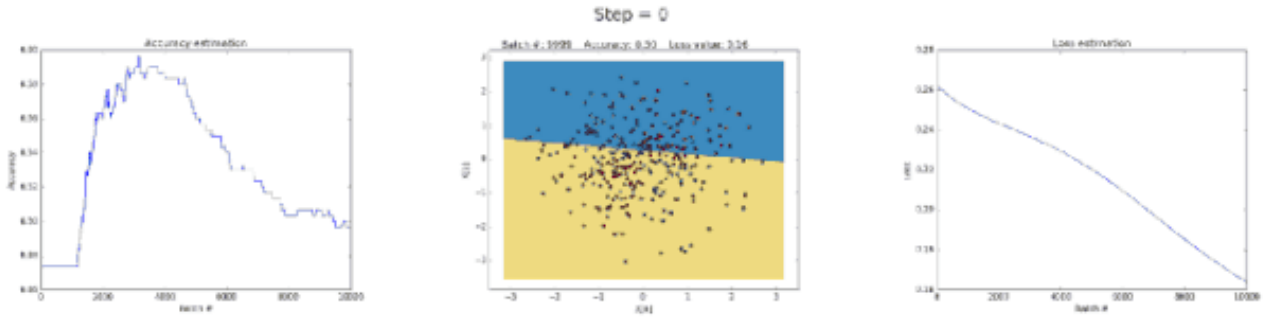
Loss function

Backward step

Weight updates

让我们来运行这份代码！

下面可以看到一些进行了好多次迭代训练得到的能够近似异或函数的神经网络。



左图：准确率；中间的图：学习到的决策边界；右图：损失函数

首先，我们来看一下隐藏层具有 3 个神经元的神经网络为何能力较弱。这个模型学会了用一个简单的决策边界来进行二分类，这个边界开始是一条直线，但是随后就表现出了非线性的行为。随着训练的持续，右图中的损失函数也明显地减小。

隐藏层拥有 50 个神经元的神经网络明显地增加了模型学习复杂决策边界的能力。这不仅仅能够得到更准确的结果，而且也使梯度发生了爆炸，这是训练神经网络时的一个显著问题。当梯度非常大的时候，反向传播中的连乘会产生很大的更新权重。这就是最后几步训练时损失函数突然增大的原因（step>90）。损失函数的正则项计算出了已经变得很大的权重的平方值（sum(W^2)/2N）。

正如你所看到的一样，这个问题可以通过减小学习率来避免。可以通过实现一个能够随着时间减小学习率的策略来实现。或者通过强制执行一个更强的正则化来实现，可能是 ‡^{5} 或者 ‡^{6} 。梯度消失和梯度爆炸是很有趣的现象，我们后续会做完整的分析。

原文链接：[illegible]