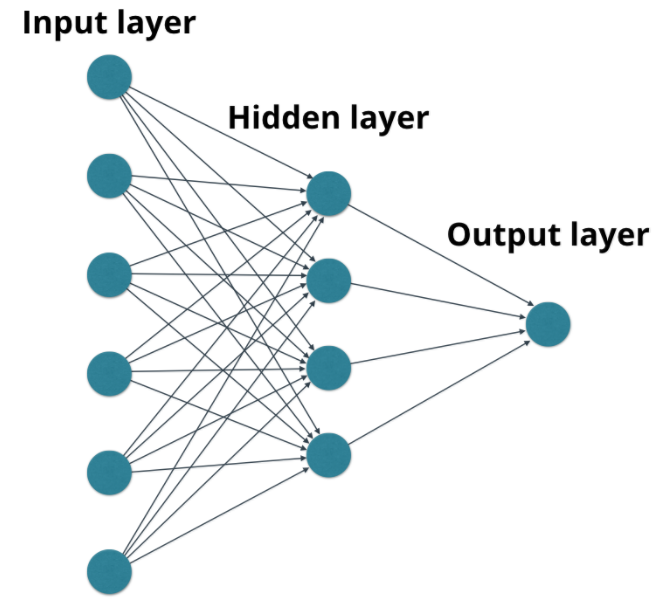
### 材料整理来自优达学城

### 什么是神经网络



神经网络是一个数学函数图表，例如[线性组合](https://en.wikipedia.org/wiki/Linear_combination" \t "https://classroom.udacity.com/nanodegrees/nd101/parts/b59b8899-07e9-46db-a665-6d432caabd01/modules/7aed441d-1f4c-47d7-9904-0264d77f6053/lessons/b6deebe4-7f78-4947-b2c6-fc660ca942fb/concepts/_blank) 和激活函数。该图表包含节点和边。

每层的节点（输入层的节点除外）都会使用上一层的节点中的输入进行数学函数计算。例如，节点可以表示 f(x,y)=x+y，其中 x 和 y 是上一层的节点中的输入值。

类似地，每个节点会创建一个输出值，并且可能会传递给下一层的节点。输出层的输出值不会传递到下一层（最后一层了！）

输入层和输出层之间的层叫做隐藏层。

#### 前向传播

将值从第一层（输入层）通过每个节点表示的所有数学函数进行传播，网络会输出一个值。这种流程叫做前向传递。

图表、节点和边形成了图表结构。虽然上面的示例很简单，但是不难理解，越来越复杂的图表可以计算几乎任何内容。

通常，创建神经网络需要两个步骤：

定义节点和边图表。

通过该图表传播值。

MiniFlow 的流程也是相同的。你将用一种方法定义网络的节点和边，并用另一个方法在图表中传播值。

### MiniFlow 架构

我们看看如何用 MiniFlow 实现这一图表结构。我们将使用一个 Python 类来表示普通节点。

class Node(object):

def \_\_init\_\_(self):

pass

我们知道，每个节点可能会从其他多个节点那接收输入。我们还知道，每个节点都会创建一个输出，这些输出有可能会传递给其他节点。我们添加以下两个列表：一个用于存储对传入节点的引用，另一个用于存储对传出节点的引用。

class Node(object):

def \_\_init\_\_(self,inbound\_nodes=[]):

# Node(s) from which this Node receives values

self.inbound\_nodes = inbound\_nodes

# Node(s) to which this Node passes values

self.outbound\_nodes = []

# For each inbound Node here,add this Node as an

# outbound Node to \_that\_ Node

for n in self.inbound\_nodes:

n.outbound\_nodes.append(self)

每个节点将最终计算出一个表示输出的值。我们将 value 初始化为 None，表示该值存在，但是尚未设定。

class Node(object):

def \_\_init\_\_(self,inbound\_nodes=[]):

# Node(s) from which this Node receives values

self.inbound\_nodes = inbound\_nodes

# Node(s) to which this Node passes values

self.outbound\_nodes = []

# For each inbound Node here,add this Node as an

# outbound Node to \_that\_ Node

for n in self.inbound\_nodes:

n.outbound\_nodes.append(self)

self.value = None

每个节点都必须能够将值向前传递，并进行反向传播（稍后会详细介绍）。暂时，我们为前向传播添加一个占位符方法。我们将稍后处理反向传播。

### 可以进行计算的节点

虽然 Node 定义了每个节点都具有的基本属性，但是只有 Node 的特殊[子类](https://docs.python.org/3/tutorial/classes.html" \l "inheritance" \t "https://classroom.udacity.com/nanodegrees/nd101/parts/b59b8899-07e9-46db-a665-6d432caabd01/modules/7aed441d-1f4c-47d7-9904-0264d77f6053/lessons/b6deebe4-7f78-4947-b2c6-fc660ca942fb/concepts/_blank)会出现在图表中。在本次实验练习中，你将构建可以进行计算和存储值的 Node 子类。例如，考虑 Node 的 Input 子类。

class Input(Node):

def \_\_init\_\_(self):

# An Input node has no inbound nodes,

# so no need to pass anything to the Node instantiator.

Node.\_\_init\_\_(self)

# NOTE: Input node is the only node where the value

# may be passed as an argument to forward().

#

# All other node implementations should get the value

# of the previous node from self.inbound\_nodes

#

# Example:

# val0 = self.inbound\_nodes[0].value

def forward(self, value=None):

# Overwrite the value if one is passed in.

if value is not None:

self.value = value

与 Node 的其他子类不同，Input 子类实际上并不计算任何内容。Input 子类仅仅存储了一个 value，例如数据特征或模型参数（权重/偏置）。

你可以明确地设置 value，或者用 forward() 方法进行设置。该值然后会传递给神经网络的其他节点。

### Add 子类

Add 是 Node 的另一个子类，实际上可以进行计算（加法）。

class Add(Node):

def \_\_init\_\_(self, x, y):

# You could access `x` and `y` in forward with

# self.inbound\_nodes[0] (`x`) and self.inbound\_nodes[1] (`y`)

Node.\_\_init\_\_(self, [x, y])

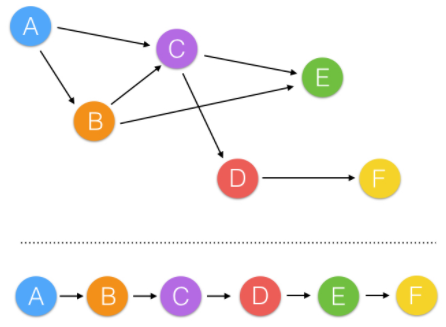
def forward(self):

self.value = self.inbound\_nodes[0].value+self.inbound\_nodes[1].value

注意 \_\_init\_\_ 方法 Add.\_\_init\_\_(self, [x, y]) 的不同之处。Input 类没有传入节点，而 Add 类具有 2 个传入节点 x和 y，并将这两个节点的值相加。

### 前向传播

MiniFlow 具有以下两个方法，可以帮助你定义和通过图表传播值：topological\_sort() 和 forward\_pass()。



为了定义你的网络，你需要定义节点的操作顺序。因为某些节点的输入取决于其他节点的输出，你需要按以下方式扁平化图表：在进行计算之前，每个节点的输入依赖项都已解决，这种技巧叫做[拓扑排序](https://en.wikipedia.org/wiki/Topological_sorting" \t "https://classroom.udacity.com/nanodegrees/nd101/parts/b59b8899-07e9-46db-a665-6d432caabd01/modules/7aed441d-1f4c-47d7-9904-0264d77f6053/lessons/b6deebe4-7f78-4947-b2c6-fc660ca942fb/concepts/_blank)。

topological\_sort() 函数使用 [Kahn 算法](https://en.wikipedia.org/wiki/Topological_sorting" \l "Kahn.27s_algorithm" \t "https://classroom.udacity.com/nanodegrees/nd101/parts/b59b8899-07e9-46db-a665-6d432caabd01/modules/7aed441d-1f4c-47d7-9904-0264d77f6053/lessons/b6deebe4-7f78-4947-b2c6-fc660ca942fb/concepts/_blank)进行拓扑排序。该方法的细节方面并不重要，结果很重要；topological\_sort()返回一个排好序的节点列表，所有计算都可以按序进行。topological\_sort() 传入 feed\_dict，我们按此方法为 Input 节点设置初始值。feed\_dict 由 Python 字典数据结构表示。

你还可以使用方法 forward\_pass()，该方法会实际地运行网络并输出一个值。

def forward\_pass(output\_node, sorted\_nodes):

for n in sorted\_nodes:

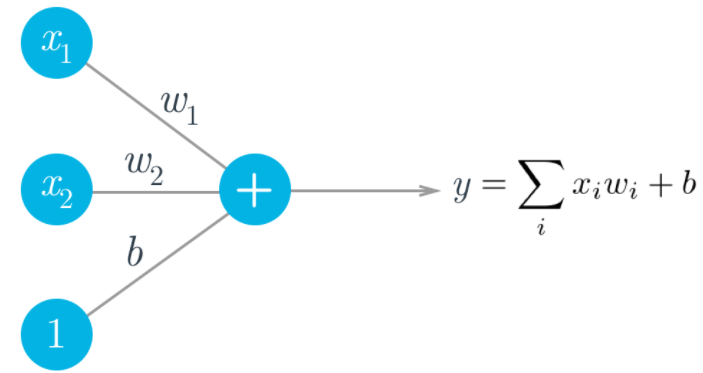
n.forward()

return output\_node.value

### 学习和损失

就像当前状态的 MiniFlow 一样，神经网络传入输入并产生输出。但是与当前状态的 MiniFlow 不一样，神经网络可以逐渐改善其输出的准确性（很难想象 Add 会逐渐提高准确性！）。要理解为何准确性很重要，请首先实现一个比 Add 更难（也更实用）的节点。

线性方程



回忆下神经网络入门那部分课程。简单的人工神经元取决于以下三个组件：

输入，xi​​

权重，wi​​

偏置，b

输出 y 就是输入加上偏置的加权和。

注意，通过更改权重，你可以更改任何给定输入对输出带来的影响。神经网络的学习流程发生在反向传播过程中。在反向传播中，网络会修改权重，以改善网络的输出准确性。miniflow.py:

class Linear(Node):

def \_\_init\_\_(self, inputs, weights, bias):

Node.\_\_init\_\_(self, [inputs, weights, bias])

# NOTE: The weights and bias properties here are not

# numbers, but rather references to other nodes.

# The weight and bias values are stored within the

# respective nodes.

def forward(self):

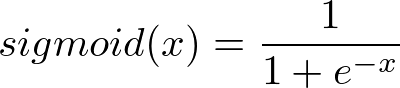
X = self.inbound\_nodes[0].value

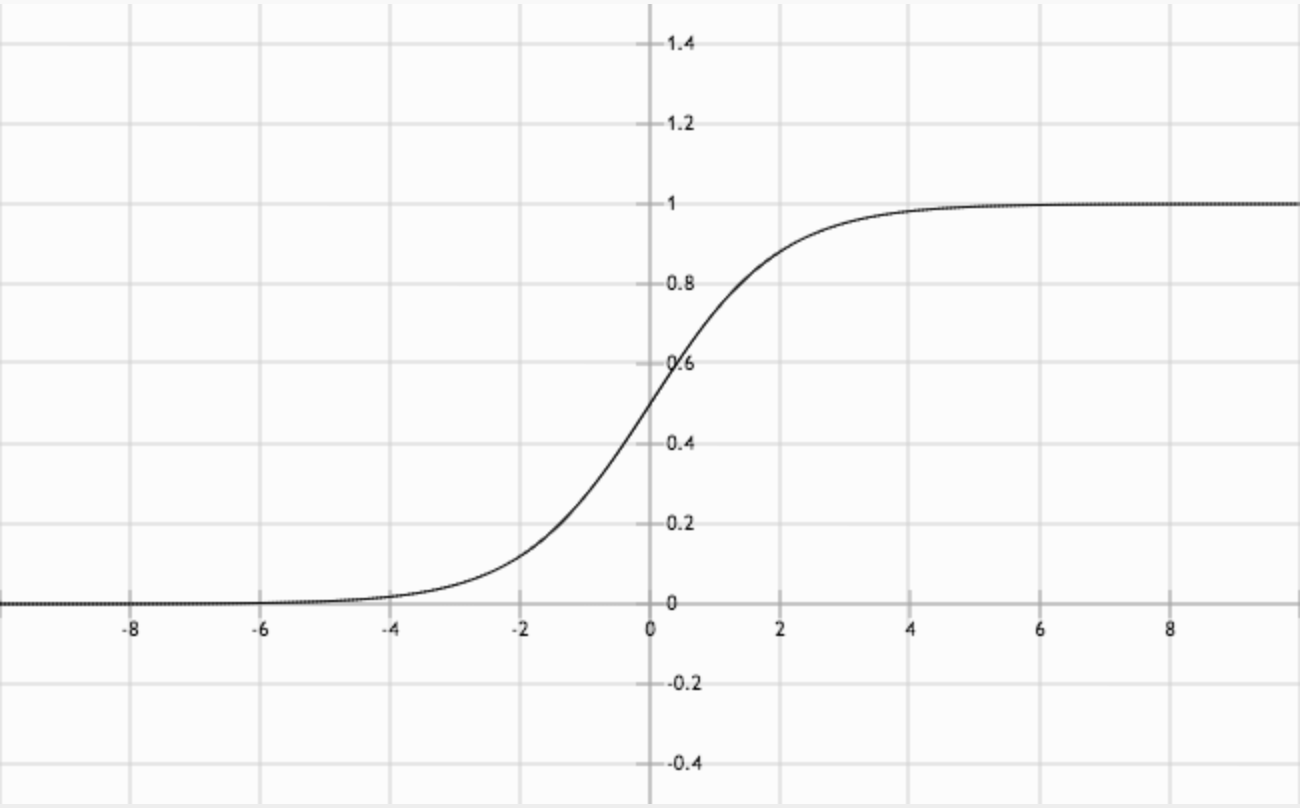
W = self.inbound\_nodes[1].value

b = self.inbound\_nodes[2].value

self.value = np.dot(X, W) + b

### Sigmoid 函数

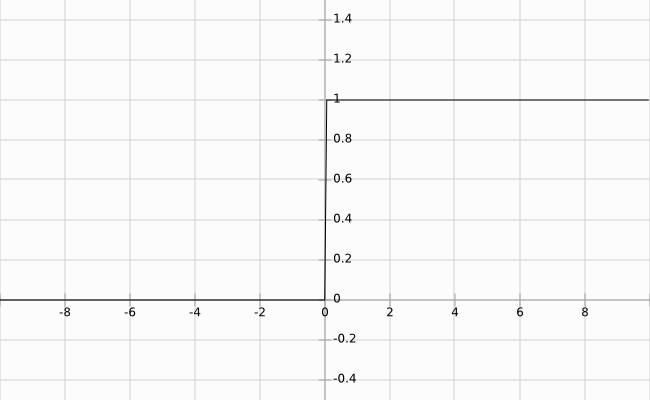




S 型函数图表。注意“S”型形状。

线性变换可以进行简单的值变换，但是神经网络通常需要更细致的变换。例如，人工神经元的原始设计之一——[感知机](https://en.wikipedia.org/wiki/Perceptron" \t "https://classroom.udacity.com/nanodegrees/nd101/parts/b59b8899-07e9-46db-a665-6d432caabd01/modules/7aed441d-1f4c-47d7-9904-0264d77f6053/lessons/b6deebe4-7f78-4947-b2c6-fc660ca942fb/concepts/_blank)就会出现二级制输出行为。感知机将加权输入与阈值对比。当加权输入超过该阈值，感知机就会被激活并输出 1，否则输出 0。

你可以用阶梯函数表示感知机的行为。



阶梯函数示例 （y = 0 和 y = 1 之间的跳转应该瞬间发生）。

激活（二进制输出行为这一概念）通常可以解决分类问题。例如，如果你要神经网络推测某个手写图片是否为 ‘9’，就相当于寻求二进制输出——是，是 ’9’，或者否，不是 ‘9’。阶梯函数是最纯粹的二进制输出形式，这很棒。但是，阶梯函数不是连续的，不可微，这很糟糕。可微分的函数才能产生梯度下降。

S 型函数（上述等式 (3)）将阈值替换成美观的 S 型曲线（如上所示），模仿出感知机的激活行为，并且保持连续性，因此可微分。此外，S 型函数具有非常简单的导数，和 S 型曲线本身非常相似。

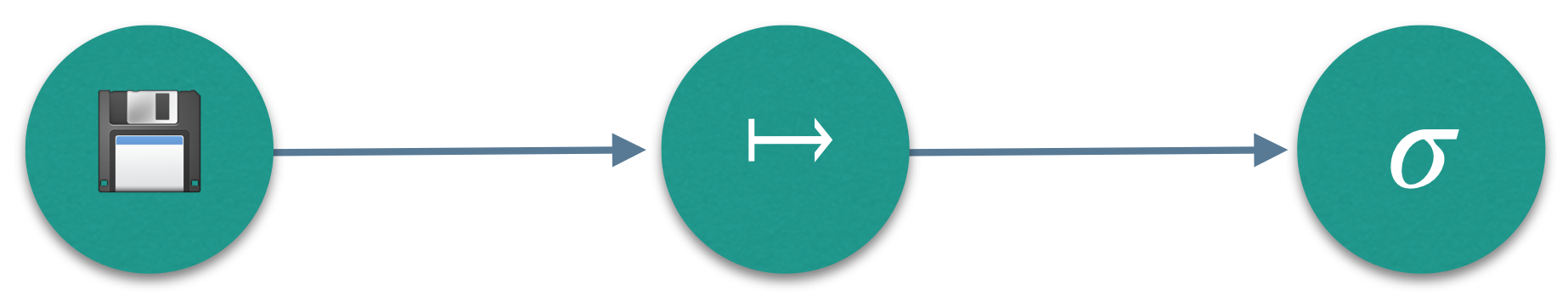
IMG_259

注意 S 型函数只有一个参数。注意，S 型函数是激活 函数（非线性），意味着它传入一个输入，并对其进行数学运算。

从概念上讲，S 型函数会做出决策。它会根据数据中的加权特征判断特征对某个分类产生影响。这样，S 型激活在线性变换后可以很好地运转。它现在具有随机权重和偏置，S 型节点的输出也是随机的。通过反向传播和梯度下降进行学习的流程（你很快就要实现这一点）会修改权重和偏置，使 S 型节点的激活开始于预期输出匹配。

注意，我已经提供了 S 型函数的等式。请将该等式添加到 MiniFlow 库中。为此，你需要使用 np.exp ([documentation](https://docs.scipy.org/doc/numpy/reference/generated/numpy.exp.html" \t "https://classroom.udacity.com/nanodegrees/nd101/parts/b59b8899-07e9-46db-a665-6d432caabd01/modules/7aed441d-1f4c-47d7-9904-0264d77f6053/lessons/b6deebe4-7f78-4947-b2c6-fc660ca942fb/concepts/_blank))，这样操作起来更轻松。

你将使用 Sigmoid 和 Linear，如下所示：



输入 > 线性变换 > S 型函数

class Sigmoid(Node):

def \_\_init\_\_(self, node):

Node.\_\_init\_\_(self, [node])

def \_sigmoid(self, x):

return 1 / (1+np.exp(-x))

def forward(self):

x = self.inbound\_nodes[0].value

self.value = self.\_sigmoid(x)

你可能觉得奇怪，为何 \_sigmoid 具有单独的方法。正如在 S 型函数（等式 (4)）的导数中看到的，S 型函数实际上是它自己的导数的一部分。将 \_sigmoid 分离出来意味着你不需要为前向传播和反向传播实现两次。

这很不错！此时，你已经使用了权重和偏置来计算输出。并且你使用了激活函数来对输出进行分类。你可能还记得，神经网络通过修改权重和偏置（根据标签化的数据集进行训练）改善输出的精确度。

我们可以采用多种技巧来定义神经网络的精确度，所有技巧围绕的都是神经网络是否能够生成与已知正确的值非常接近的值。人们用不同的名称来表示这一精确度测量者，通常称之为损失或代价。我将经常使用代价一词。

对于本测验，你将使用均方差 (MSE) 计算代价。如下所示：

C(w,b)=​1/m∑∣∣y(x)−a∣∣2​​

此处，w 表示网络中所有的权重集合，b 表示所有的偏置，m 表示训练示例的总数，a 是 y(x) 的近视值，a 和 y(x) 都是长度相同的向量。

权重集合是所有权重矩阵压平成的向量，串联成一个大的向量。偏置也相似，但是它们已经是向量，所以在串联前不需要压平。

以下是创建 w 的代码示例：

# 2 by 2 matrices

w1 = np.array([[1, 2], [3, 4]])

w2 = np.array([[5, 6], [7, 8]])

# flatten

w1\_flat = np.reshape(w1, -1)

w2\_flat = np.reshape(w2, -1)

w = np.concatenate((w1\_flat, w2\_flat))# array([1, 2, 3, 4, 5, 6, 7, 8])

这样可以轻松地将神经网络中使用的所有权重和偏置提取出来，从而更轻松地编写代码，我们将在接下来的梯度下降部分看到。

注意：你不需要在你的代码中实现！只是将权重和偏置看做集合比单独对待更容易处理。

代价 C 取决于正确输出 y(x) 和网络的输出 a 之间的差值。很容易看出 y(x) 和 a (对于 x 的所有值）) 之间的差始终不为 0。

这是理想情况，实际上学习流程就是为了尽量减小代价。

现在来计算代价。

你在上道测验的前向中实现了这一网络。

现在可以看出它输出的是无用数据。S 型节点的激活什么也没表示，因为该网络没有可以与之对比的带标签输出。此外，权重和偏置无法更改，没有代价的话，无法进行学习。

class MSE(Node):

def \_\_init\_\_(self, y, a):

Node.\_\_init\_\_(self, [y, a])

def forward(self):

y = self.inbound\_nodes[0].value.reshape(-1, 1)

a = self.inbound\_nodes[1].value.reshape(-1, 1)

self.m = self.inbound\_nodes[0].value.shape[0]

# Save the computed output for backward.

self.diff = y - a

self.value = np.mean(self.diff\*\*2)

现在，我将简单复习下梯度下降概念，使我们能够开始用 MiniFlow 训练我们的网络。注意，我们的目标是通过尽量减小代价，使我们的网络输出与目标值尽量接近。你可以将代价看做一座山，我们想要到达山底。

想象你的模型参数表示为一个停在山顶的球。直观地来说，我们希望将球推下山。这样可以明白，但是我们讨论的是代价函数，如何知道哪条路是下山呢？

幸运的是，梯度下降正好给出了这一信息。

严格来说，梯度实际上指的是上坡，是最陡上升方向。但是，如果我们在此值前面加个负号，就得出了最陡下降方向，这正是我们需要的。

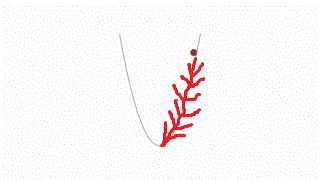
稍后你将详细了解梯度，但是暂时可以将其看做数字向量。每个数字表示我们应该对照着调整神经网络中相应的权重或偏置的数量。按照梯度值调整所有的权重和偏置降低了网络的代价（或误差）。

听明白了吗？

好的！现在我们知道朝着哪个方向推球了。下一步是考虑用多大的推力，称之为学习速度，该名称比较恰当，因为该值确定了神经网络学习的快慢速度。

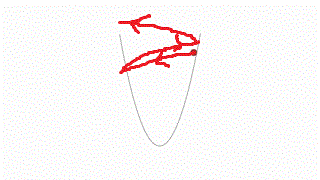
你可能希望设置非常大的学习速度，这一网络就能学的非常快，对吧？

要小心！如果该值太大，可能会迭代过度并最终偏离目标。呀！



**收敛**

这是理想的行为。



**发散**

当学习速度过高时会出现这种情况。

那么，什么样的学习速度合适呢？

我们只能做出猜测，根据以往经验，0.1 至 0.0001 范围的值效果最好。0.001 至 0.0001 范围的值很常见，因为 0.1 和 0.01 有时候过大。

下面是梯度下降的公式（伪代码）：

x = x - learning\_rate \* gradient\_of\_x

x 是神经网络使用的参数（即单个权重或偏置）。

我们将 gradient\_of\_x（上坡方向）与 learning\_rate（推力）相乘，然后将 x 减去相乘结果，从而向下推动。

梯度下降解决方案

def gradient\_descent\_update(x, gradx, learning\_rate):

"""

Performs a gradient descent update.

"""

x = x - learning\_rate \* gradx

# Return the new value for x

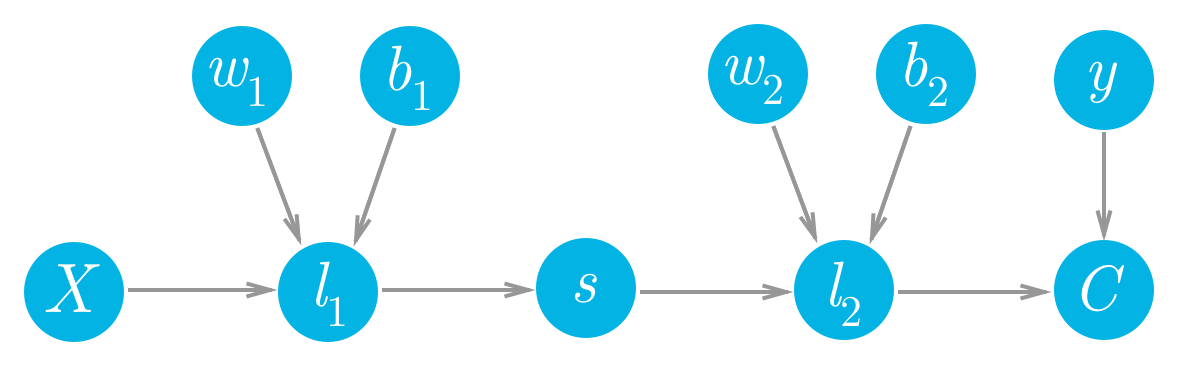
return x

我们调整了旧的 x，朝着 gradx 的方向推动，推力为 learning\_rate。减去 learning\_rate \* gradx。注意，梯度一开始朝着最陡上升方向，所以将 x 减去 learning\_rate \* gradx 使其变成最陡下降方向。你可以通过将减法替换为加法自己进行确定。

#### 梯度和反向传播

我们现在知道如何使用梯度更新我们的权重和偏置，我们还需要知道如何计算所有节点的梯度。对于每个节点，我们需要根据梯度更改代价的值（考虑到该节点的值）。这样，我们做出的梯度下降更新最终会实现最低代价。

我们来看一个网络，其中具有一个线性节点 l1，一个 S 型节点 s，以及另一个线性节点 l2，还有一个 MSE 节点，用于计算代价 C。



简单的两层网络的前向传递。

用 MiniFlow 编写的话，应该如下所示：

X, y = Input(), Input()

W1, b1 = Input(), Input()

W2, b2 = Input(), Input()

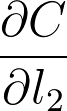
l1 = Linear(X, W1, b1)

s = Sigmoid(l1)

l2 = Linear(s, W2, b2)

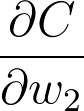
cost = MSE(l2, y)

我们可以看到这些节点的每个值都向前流动，最终生成代价 C。例如，第二个线性节点 l2的值进入代价节点并确定该节点的值。更改 l2将导致 C 相应地出现更改。我们可以将更改之间的这种关系写成梯度。

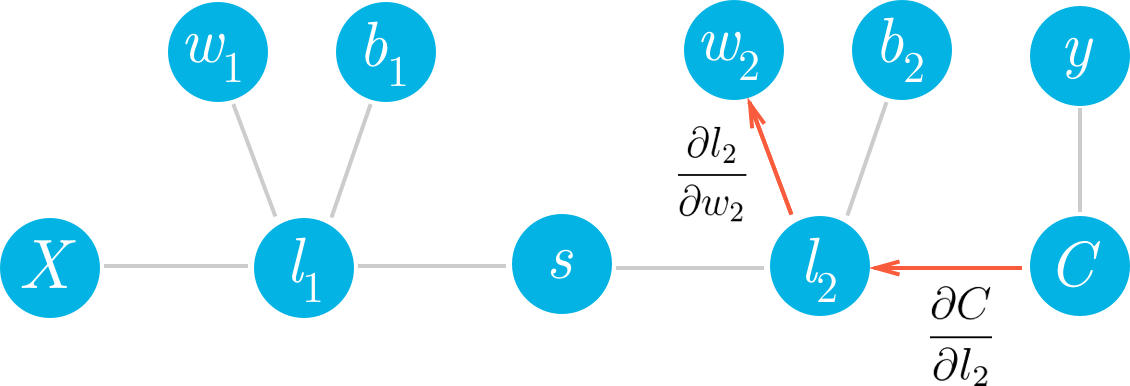


这就是梯度的含义，是一种斜率，表示给出 l2中的更改，你会对代价 ∂C 进行多大幅度的更改。所以，节点的代价梯度更大的话，代价就会改变更大。这样，我们可以对每个节点的代价带来的影响进行分配。节点的梯度越大，就会对最终的代价影响越大。节点的影响越大，我们就会在梯度下降步骤中更新幅度越大。

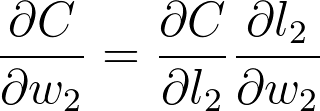
如果我们想更新某个梯度下降的权重，我们需要知道这些权重对应的代价的梯度。看看我们可以如何使用此框架算出第二层权重 w2 的梯度。我们想要计算 C 相对于 w2的梯度：



我们可以从图表中看出，w2与 l2相关联，所以更改 w2将导致 l2出现更改，从而导致 C 出现更改。我们可以通过在网络中将代价梯度发送回去，将影响分配给 w2。首先，你知道 l2对 C 的影响有多大，然后知道 w2对 l2的影响有多大。将这些梯度相乘可以得出归为 w2的总影响。



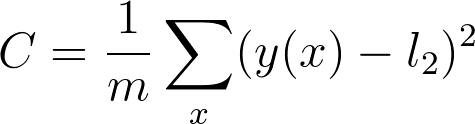
将这些梯度相乘只是链式法则的一种应用：



可以从图表中看出，w2、l2 和 C 相互链接在一起。如果 w2发生任何更改，将导致 l2出现更改，更改幅度由梯度 ∂l2/∂w2 决定。因为 l2更改了，这将导致代价 C 出现更改，更改幅度由梯度 ∂C/∂l2 决定。你可以将链式法则看做多米诺效应，网络中的某项更改将从网络中传播开来，并一路更改其他节点。

如果你将链式法则看做普通的分数，可以看到分母上的 ∂l2 和分子消掉了，获得 ∂C/∂w2​​（虽然和普通分子的计算过程并不完全一样，但是可以帮助你理解）。现在来算算 w2的梯度。首先，我们需要知道 l2的梯度。

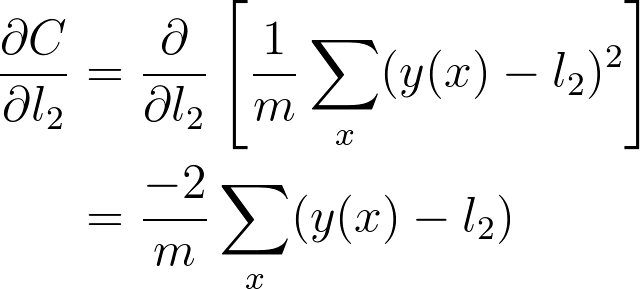
提醒下，代价是：

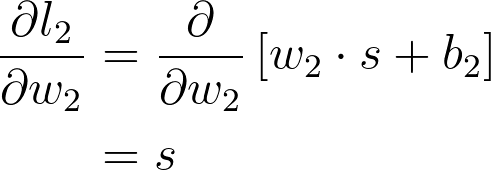


第二个线性节点的值为

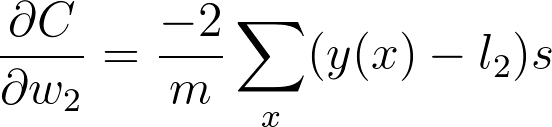
IMG_262

w2、s和 b2都是向量，w2s 表示 w2 和 s 的点积。



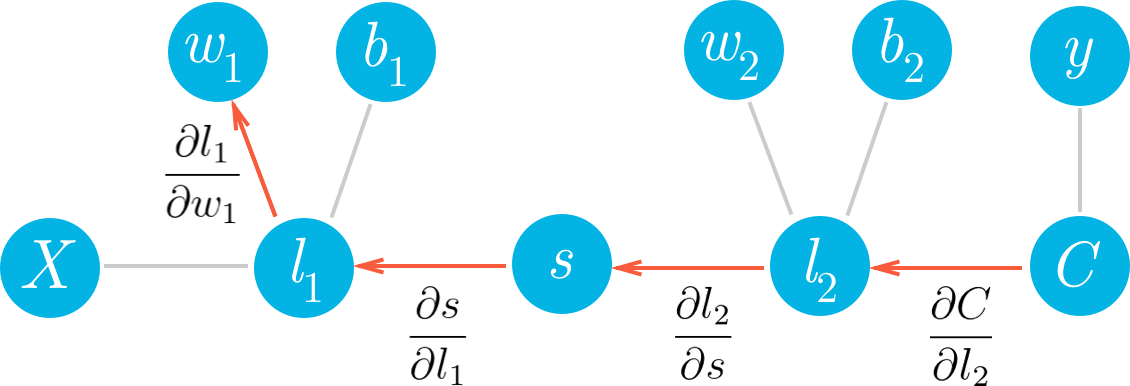


将这些放一起，可以得出 w2的梯度

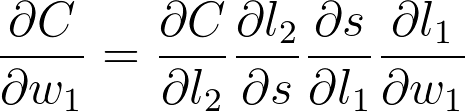


这是你在 w2的梯度下降更新中用到的梯度。可以看出，我们在图表上一直往回计算，将一路上发现的所有梯度相乘。

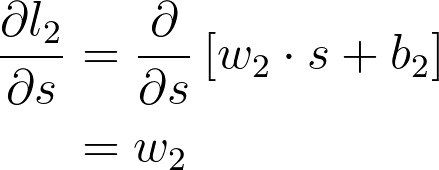
现在，我们再深入一步，计算 w1的梯度。和之前用到的方法一样，在图表上一直往回计算。



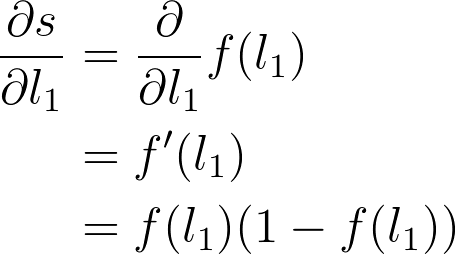
希望现在你能明白如何通过查看图表写出 w1的梯度。我们将使用链式法则在图表上一直往回计算，得出每个节点的梯度，直到算出 w1 的梯度。

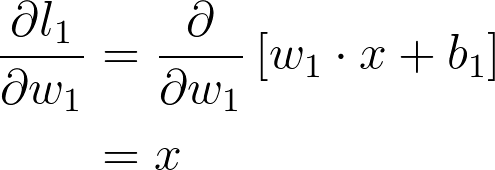


现在我们可以计算此表达式中的每个梯度，以便得出 w1的梯度

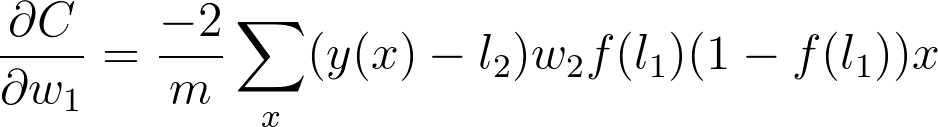


下一步是 S 型函数 s=f(l1) 的梯度。因为这里使用的是逻辑函数，所以导数可以写成 S 型函数本身





将所有这一切放在一起，得出



现在可以看到清晰的规律了。要算出梯度，只需将它前面所有节点（从代价那开始）的梯度相乘。这就是反向传播概念。梯度在网络上向后传播，并使用梯度下降来更新权重和偏置。如果某个节点具有多个向外的节点，则直接将每个节点的梯度相加即可。

#### 在 MiniFlow 中实现

我们来看看如何在 MiniFlow 中实现这一流程。在图表中可以看到每个节点从传出节点那获得代价梯度。例如，节点 l1通过 S 型节点 s 获取 ∂C/∂l1。然后，l1将代价梯度继续传递给权重节点 w1，但是会乘以 ∂l1/∂w1，即 l1的梯度除以其输入 w1。

所以，每个节点会将代价梯度传递给传入节点，每个节点将从其传出节点那获得代价梯度。然后，对于每个节点，我们需要算出一个梯度，即代价梯度乘以该节点的梯度除以其输出。下面我为 Linear 节点写出了这一流程。

# Initialize a partial for each of the inbound\_nodes.

self.gradients = {n: np.zeros\_like(n.value) for n in self.inbound\_nodes}

# Cycle through the outputs. The gradient will change depending# on each output,

# so the gradients are summed over all outputs.for n in self.outbound\_nodes:

# Get the partial of the cost with respect to this node.

grad\_cost = n.gradients[self]

# Set the partial of the loss with respect to this node's inputs.

self.gradients[self.inbound\_nodes[0]] += np.dot(grad\_cost, self.inbound\_nodes[1].value.T)

# Set the partial of the loss with respect to this node's weights.

self.gradients[self.inbound\_nodes[1]] += np.dot(self.inbound\_nodes[0].value.T, grad\_cost)

# Set the partial of the loss with respect to this node's bias.

self.gradients[self.inbound\_nodes[2]] += np.sum(grad\_cost, axis=0, keepdims=False)

新的代码

自上次查看 MiniFlow 起，MiniFlow 已经出现了几处更改：

首先是 Node 类现在具有一个 backward 方法，并且添加了新的属性 self.gradients，用于在反向传递过程中存储和缓存梯度。

class Node(object):

def \_\_init\_\_(self, inbound\_nodes=[]):

# A list of nodes with edges into this node.

self.inbound\_nodes = inbound\_nodes

# The eventual value of this node. Set by running

# the forward() method.

self.value = None

# A list of nodes that this node outputs to.

self.outbound\_nodes = []

# New property! Keys are the inputs to this node and

# their values are the partials of this node with

# respect to that input.

self.gradients = {}

# Sets this node as an outbound node for all of

# this node's inputs.

for node in inbound\_nodes:

node.outbound\_nodes.append(self)

def forward(self):

raise NotImplementedError

def backward(self):

"""

Every node that uses this class as a base class will

need to define its own `backward` method.

"""

raise NotImplementedError

第二项更改是辅助函数 forward\_pass()。该函数被替换成了 forward\_and\_backward()。

def forward\_and\_backward(graph):

# Forward pass

for n in graph:

n.forward()

for n in graph[::-1]:

n.backward()

设置

下面是 sigmoid 函数 w.r.t 的导数 x：

sigmoid(x)=1/(1+exp(−x))

∂sigmoid/∂x=sigmoid(x)∗(1−sigmoid(x))

nn.py:

import numpy as np

from miniflow import \*

X, W, b = Input(), Input(), Input()

y = Input()

f = Linear(X, W, b)

a = Sigmoid(f)

cost = MSE(y, a)

X\_ = np.array([[-1., -2.], [-1, -2]])

W\_ = np.array([[2.], [3.]])

b\_ = np.array([-3.])

y\_ = np.array([1, 2])

feed\_dict = {

X: X\_,

y: y\_,

W: W\_,

b: b\_,

}

graph = topological\_sort(feed\_dict)

forward\_and\_backward(graph)

# return the gradients for each Input

gradients = [t.gradients[t] for t in [X, y, W, b]]

"""

Expected output

[array([[ -3.34017280e-05, -5.01025919e-05],

[ -6.68040138e-05, -1.00206021e-04]]), array([[ 0.9999833],

[ 1.9999833]]), array([[ 5.01028709e-05],

[ 1.00205742e-04]]), array([ -5.01028709e-05])]

"""

print(gradients)

miniflow.py:

import numpy as np

class Node(object):

"""

Base class for nodes in the network.

Arguments:

`inbound\_nodes`: A list of nodes with edges into this node.

"""

def \_\_init\_\_(self, inbound\_nodes=[]):

"""

Node's constructor (runs when the object is instantiated). Sets

properties that all nodes need.

"""

# A list of nodes with edges into this node.

self.inbound\_nodes = inbound\_nodes

# The eventual value of this node. Set by running

# the forward() method.

self.value = None

# A list of nodes that this node outputs to.

self.outbound\_nodes = []

# New property! Keys are the inputs to this node and

# their values are the partials of this node with

# respect to that input.

self.gradients = {}

# Sets this node as an outbound node for all of

# this node's inputs.

for node in inbound\_nodes:

node.outbound\_nodes.append(self)

def forward(self):

"""

Every node that uses this class as a base class will

need to define its own `forward` method.

"""

raise NotImplementedError

def backward(self):

"""

Every node that uses this class as a base class will

need to define its own `backward` method.

"""

raise NotImplementedError

class Input(Node):

"""

A generic input into the network.

"""

def \_\_init\_\_(self):

# The base class constructor has to run to set all

# the properties here.

#

# The most important property on an Input is value.

# self.value is set during `topological\_sort` later.

Node.\_\_init\_\_(self)

def forward(self):

# Do nothing because nothing is calculated.

pass

def backward(self):

# An Input node has no inputs so the gradient (derivative)

# is zero.

# The key, `self`, is reference to this object.

self.gradients = {self: 0}

# Weights and bias may be inputs, so you need to sum

# the gradient from output gradients.

for n in self.outbound\_nodes:

grad\_cost = n.gradients[self]

self.gradients[self] += grad\_cost \* 1

class Linear(Node):

"""

Represents a node that performs a linear transform.

"""

def \_\_init\_\_(self, X, W, b):

# The base class (Node) constructor. Weights and bias

# are treated like inbound nodes.

Node.\_\_init\_\_(self, [X, W, b])

def forward(self):

"""

Performs the math behind a linear transform.

"""

X = self.inbound\_nodes[0].value

W = self.inbound\_nodes[1].value

b = self.inbound\_nodes[2].value

self.value = np.dot(X, W) + b

def backward(self):

"""

Calculates the gradient based on the output values.

"""

# Initialize a partial for each of the inbound\_nodes.

self.gradients = {n: np.zeros\_like(n.value) for n in self.inbound\_nodes}

# Cycle through the outputs. The gradient will change depending

# on each output, so the gradients are summed over all outputs.

for n in self.outbound\_nodes:

# Get the partial of the cost with respect to this node.

grad\_cost = n.gradients[self]

# Set the partial of the loss with respect to this node's inputs.

self.gradients[self.inbound\_nodes[0]]+=np.dot(grad\_cost, self.inbound\_nodes[1].value.T)

# Set the partial of the loss with respect to this node's weights.

self.gradients[self.inbound\_nodes[1]]+=np.dot(self.inbound\_nodes[0].value.T, grad\_cost)

# Set the partial of the loss with respect to this node's bias.

self.gradients[self.inbound\_nodes[2]]+=np.sum(grad\_cost,axis=0, keepdims=False)

class Sigmoid(Node):

"""

Represents a node that performs the sigmoid activation function.

"""

def \_\_init\_\_(self, node):

# The base class constructor.

Node.\_\_init\_\_(self, [node])

def \_sigmoid(self, x):

return 1. / (1. + np.exp(-x))

def forward(self):

input\_value = self.inbound\_nodes[0].value

self.value = self.\_sigmoid(input\_value)

def backward(self):

# Initialize the gradients to 0.

self.gradients = {n: np.zeros\_like(n.value) for n in self.inbound\_nodes}

# Cycle through the outputs. The gradient will change depending

# on each output, so the gradients are summed over all outputs.

for n in self.outbound\_nodes:

# Get the partial of the cost with respect to this node.

grad\_cost = n.gradients[self]

value = self.\_sigmoid(self.inbound\_nodes[0].value)

self.gradients[self.inbound\_nodes[0]] += value\*(1-value)\*grad\_cost

backward 方法会对导数求和（如果只有一个变量，则是法向导数），并且相对于所有输出节点的唯一输入。最后一行实现了导数 ∂sigmoid/∂x,∂cost/∂sigmoid。

将数学表达式替换为代码：

​​∂sigmoid/∂x​ is sigmoid \* (1 - sigmoid) and  ∂cost/∂sigmoid is grad\_cost.

现在你已经知道相对于每个输入（forward\_and\_backward() 的返回值）的代价梯度，你的网络可以开始学习了！为此，你将实现一个技巧，叫做随机梯度下降。

#### 随机梯度下降

随机梯度下降 (SGD) 是一种梯度下降形式，其中对于每次前向传递，都会从总的数据集中随机选择一批数据。还记得之前讨论的批量大小吗？即批次大小。理想情况下，每次前向传递时，都会将整个数据集提供给神经网络。但是实际操作中，这么做是不现实的，因为内存有限。随机梯度下降是梯度下降的近视值，神经网络处理的批次越多，近视值就越准确。 SGD 的实现包括：

从总的数据集中随机抽样一批数据。

前向和后向运行网络，计算梯度（根据第 (1) 步的数据）。

应用梯度下降更新。

重复第 1-3 步，直到出现收敛情况或者循环被其他机制暂停（即迭代次数）。

如果一切顺利，网络的损失应该通常会下降，表示权重和偏置越来越有用。

到目前为止，MiniFlow 已经可以完成第 2 步。在下面的测验中，第 1 和 4 步已执行。你需要执行第 3 步。

提醒下，下面是梯度下降更新方程，其中 α 表示学习速度：

x=x−α\*∂cost/∂x​​

对于这道练习，我们将使用实际的数据集——[波士顿房价数据集](https://archive.ics.uci.edu/ml/datasets/Housing" \t "https://classroom.udacity.com/nanodegrees/nd101/parts/b59b8899-07e9-46db-a665-6d432caabd01/modules/7aed441d-1f4c-47d7-9904-0264d77f6053/lessons/b6deebe4-7f78-4947-b2c6-fc660ca942fb/concepts/_blank)。经过训练后，该网络将能够预测波士顿的房价！

代码如下地址：

https://github.com/dongyangdaozi/Miniflow