知乎 首发于 计算主义

计算图反向传播的原理及实现

张觉非
All IS ONE

关注他

Lyon 等 88 人赞同了该文章

前言

本文节选自即将出版的《深入理解神经网络——从逻辑回归到CNN》（暂定名）的第八章“计算图”。阅读本文前，可先阅读本专栏之前的文章“神经网络反向传播算法”和“梯度下降”。本文介绍了计算图以及计算图上的自动求导（通用的反向传播）的原理，用Python+Numpy实现了计算图以及计算图上的反向传播，用其搭建了一个多层全连接神经网络，建模 MNIST 手写数字识别。本文的样例代码见于：

张觉非/计算图框架
gitee.com

在下一篇博文中：

张觉非：运用计算图搭建 LR、NN、Wide & Deep、FM、FFM 和...
zhuanlan.zhihu.com

知乎 首发于 计算主义

推出下一本书，教读者实现一个基于计算图的机器学习框架。

神经网络的结构并不仅限于多层全连接，在深度学习领域，存在局部连接、权值共享、跳跃连接等丰富多样的神经元连接方式，多层全连接仅仅是其中的一种。在打开更广阔的新世界的大门之前，我们首先需要掌握描述和训练任意神经网络的方法。

计算图是一个强大的工具，绝大部分神经网络都可以用计算图描述。计算图用节点表示变量，用有向边表示计算。自动求导应用链式法则求某节点对其他节点的雅可比矩阵，它从结果节点开始，沿着计算路径向前追溯，逐节点计算雅可比。将神经网络和损失函数连接成一个计算图，则它的输入、输出和参数都是节点，可利用自动求导求损失值对网络参数的雅可比，从而得到梯度。

本文首先介绍计算图，并以多层全连接神经网络和一种非全连接网络为例，展示计算图的表达能力，之后，我们介绍自动求导的原理和实现。具备了这些知识，就能理解如何构建和训练任意神经网络，为进入深度学习领域做好准备。

1 计算图模型

我们需要一种灵活通用的方法描述各种神经网络，计算图就是一种合适的工具。本节首先介绍计算图的基本概念，之后阐述如何用计算图描述多层全连接神经网络以及一个简单的卷积神经网络。

1.1 简介

计算图（computational graph）是一种有向无环图（directed acyclic graph，DAG）。计算图用节点表示变量，用有向边（directed edge）表示计算。有向边的目的节点称为子节点，源节点称为父节点，计算图定义如何用父节点计算子节点，如图1所示。

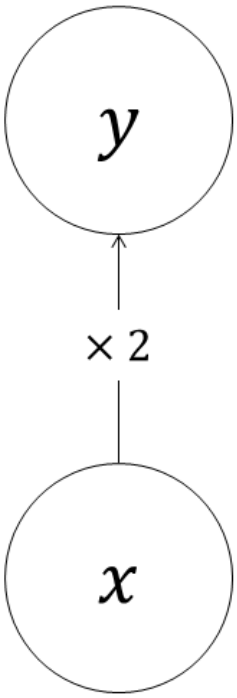


图1 计算图的节点和有向边

图1中的计算图描述了计算：

$$y = 2x$$

一个子节点可以有两个父节点，表示该子节点由两个父节点计算而得，如图2所示。

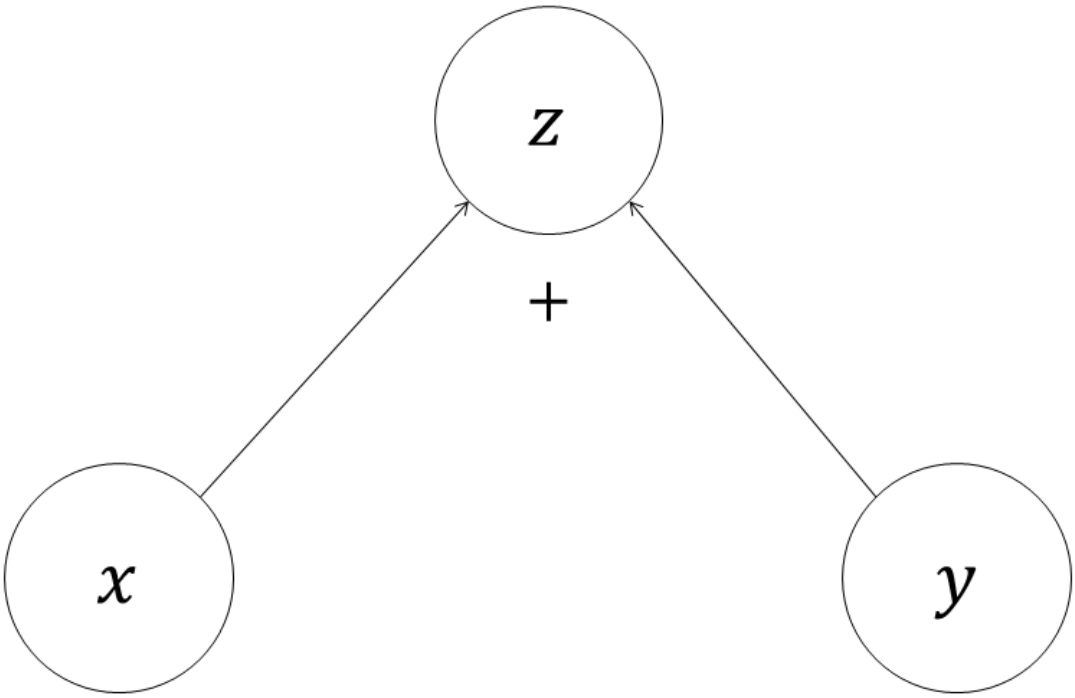


图2 有两个父节点的子节点

图2中的计算图描述的计算是：

$$z = x + y$$

一个子节点也可以有两个以上的父节点，但是这种情况可以通过添加中间节点而转化成每个子节点只有两个父节点的情况。图3中的两个计算图是等价的。

知乎 首发于 计算主义

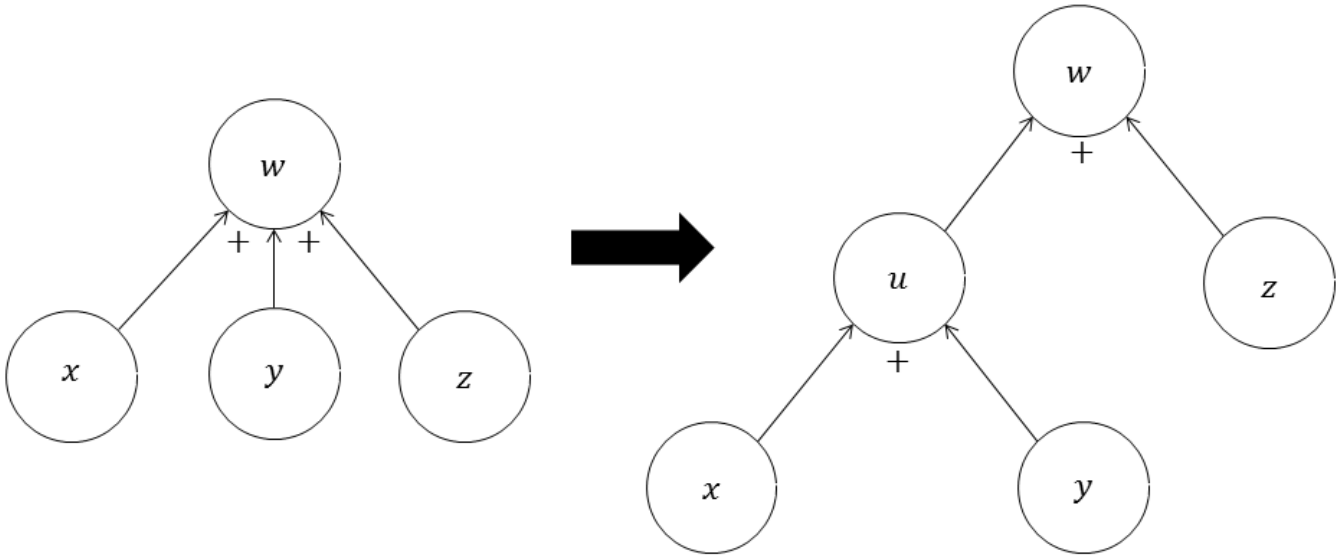


图3 两个以上父节点可以转化成两个父节点

图3中的两个计算图描述的计算都是：

$$w = x + y + z$$

所以本文中，每个子节点至多有两个父节点。用上述简单的组件可以表达复杂的计算，例如图4所示。

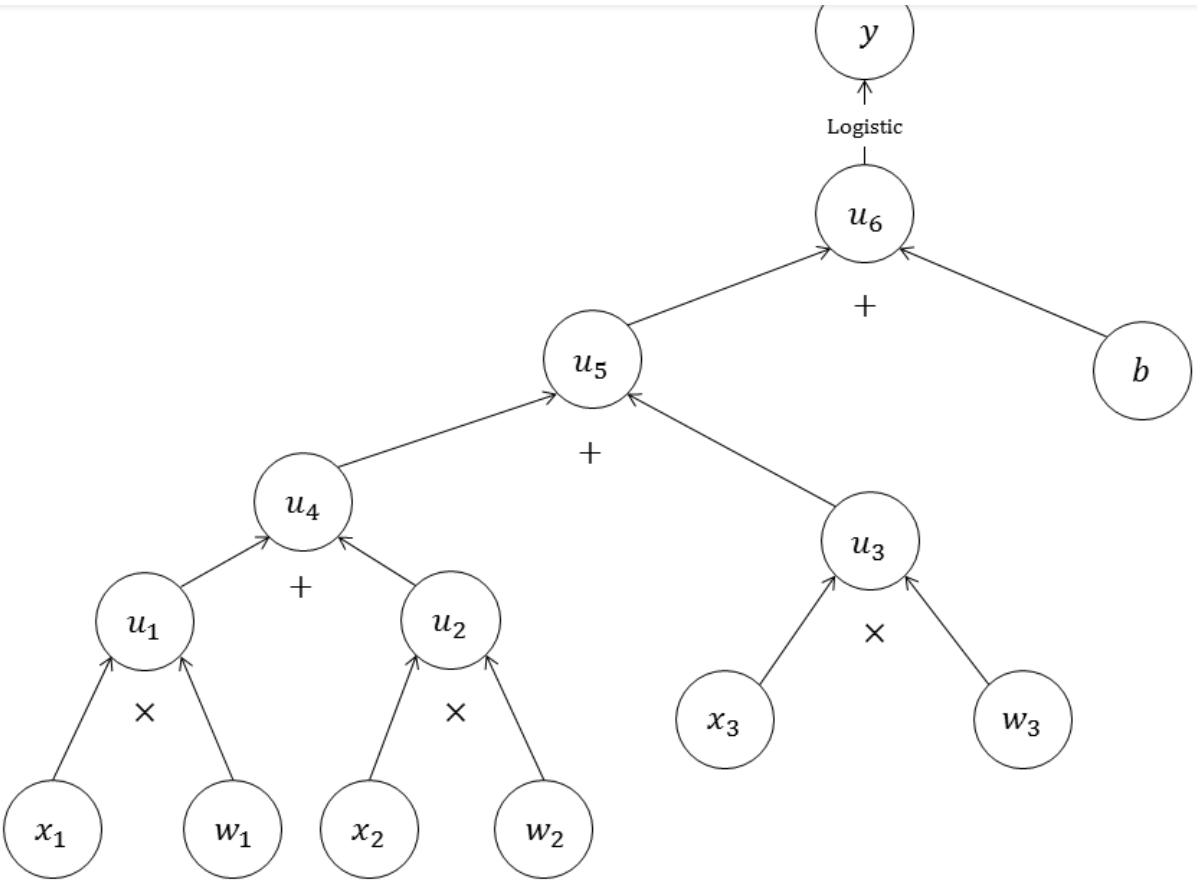


图4 逻辑回归的计算图

图4中的计算图表示逻辑回归模型：

$$y = \frac{1}{1 + e^{-(w_1 x_1 + w_2 x_2 + w_3 x_3 + b)}}$$

同一个计算可以用不同的计算图描述。计算式可以代数变形这自不必说，即便是同一个式子也可以用不同的计算图描述，这取决于表示计算的“粒度”。例如图4中，从节点得到节点的计算是Logistic函数，但也可以将Logistic函数拆解成更基础的计算，如图5所示。

知乎 首发于 计算主义

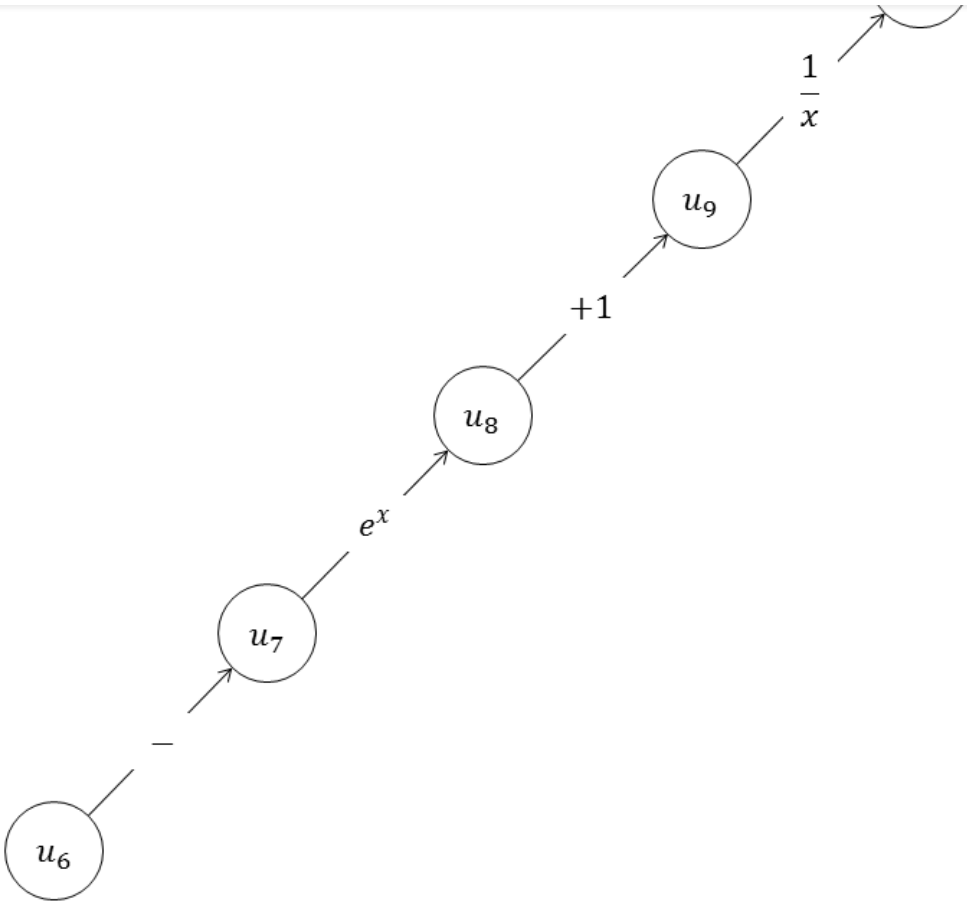


图5 将Logistic函数拆解成更基础的计算

图5中的计算图只包含取反、指数、增1和取倒数这四种基础运算，它表示的是Logistic函数。用更基础的运算构建计算图，会使计算图的规模更大。以上介绍的计算图的节点都是标量，节点也可以是向量、矩阵乃至张量（tensor）。可将矩阵或张量的元素重新排列为向量，例如矩阵：

$$\boldsymbol{X} = \begin{pmatrix} x_{1,1} & \cdots & x_{1,n} \\ \vdots & \ddots & \vdots \\ x_{m,1} & \cdots & x_{m,n} \end{pmatrix}$$

将的元素重新排列，可以得到向量：

$$x = \begin{pmatrix} x_{1,n} \\ \vdots \\ x_{m,1} \\ \vdots \\ x_{m,n} \end{pmatrix}$$

向量 x 是将 X 的各行排成一列，它的维数是 $m \times n$ 。对矩阵或张量的计算无非是对其元素的计算，所以它们都可以转化成对向量的计算。若计算的结果是矩阵或张量，也可以将其排列成向量。所以本文讨论的计算图的节点都是向量或标量。使用向量节点，逻辑回归模型可以表示为图6中的计算图。

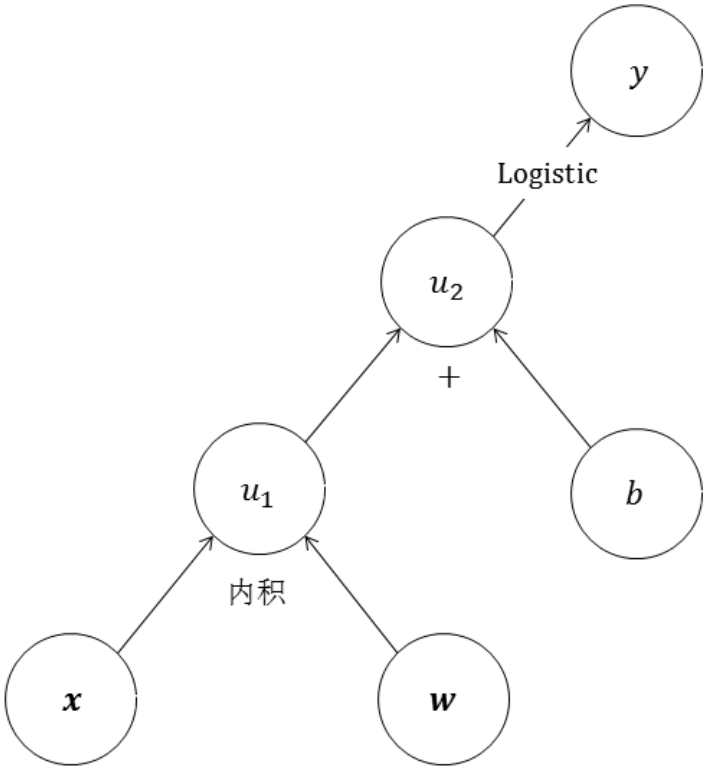


图6 节点为向量的逻辑回归模型计算图

图6中的 x 和 w 是向量，用它们得到标量节点 u_1 的运算是内积。若计算图有多个输入节点，即该计算图接受多个输入向量，可将这些向量连在一起视为一个输入向量。整个计算图本质上是一个映射：由输入向量得到输出向量。每个节点和它的父节点构成计算图的一个子计算图，它也是一个映射。

量，将 w 和 b 视为变量。

1.2 多层全连接神经网络的计算图

常见的多层全连接神经网络示意图就是一个计算图，它的节点都是标量，表示计算的粒度较细。现在我们可以用向量节点更简洁地表示多层全连接神经网络，如图7所示。

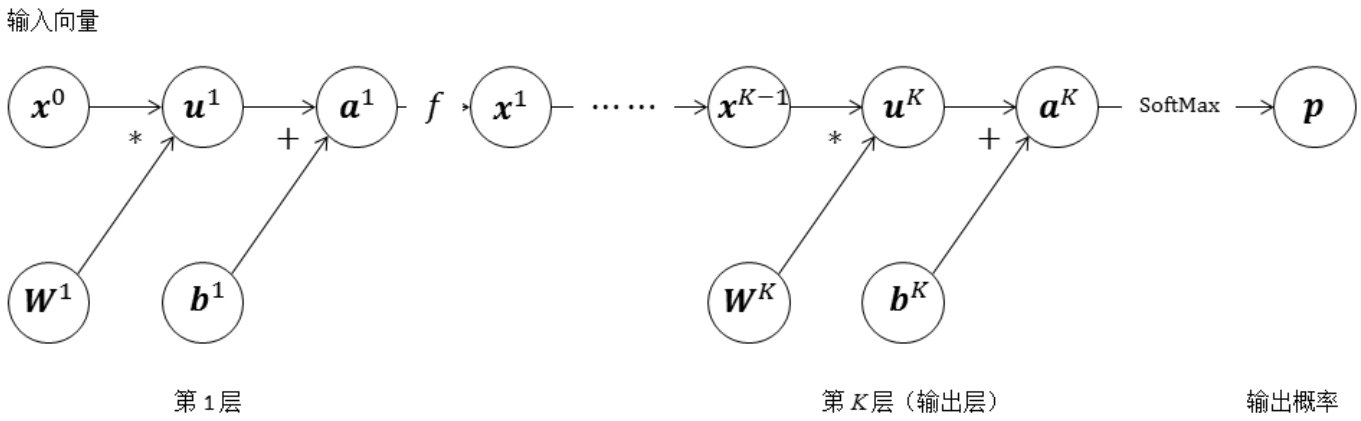


图7 用向量节点计算图表示多层全连接神经网络

W^k 是权值矩阵， b^k 是偏置向量。将输入 x^{k-1} 与权值矩阵相乘，得到向量 u^k ，将 u^k 与 b^k 相加得到仿射值向量 a^k ，对 a^k 的每个分量施加激活函数，得到该神经元层的输出 x^k 。输出层不对仿射值向量 a^k 施加激活函数，而是施加SoftMax函数，得到多分类概率向量 p 。

1.3 其他神经网络结构的计算图

计算图可以灵活地表达各种复杂的神

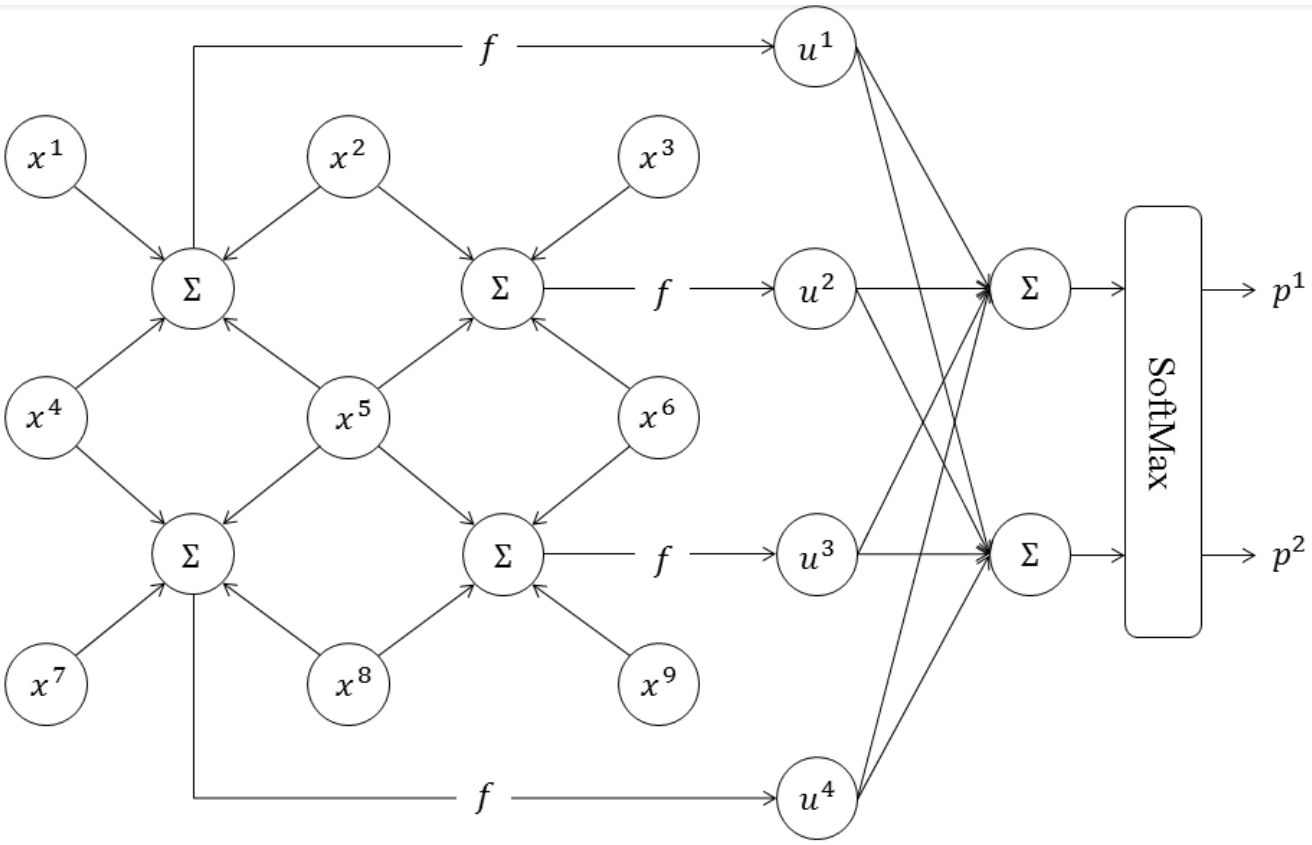


图8 非全连接结构的神经网络

这个神经网络的输入是 $x_1, x_2, \ldots, x_9$ 。将输入排成 3×3 的阵列，该阵列包含4个的 2×2 子阵列，将每个子阵列的输入加权求和再加偏置得到仿射值。这4个仿射单元使用同一套权值和偏置（图中没有画出）。对4个仿射值施加激活函数 f ，然后连接到两个仿射单元。对这两个仿射单元的输出施加SoftMax函数，得到两个概率 p^1 和 p^2 。后文会知道，这个神经网络就是一个卷积层加一个全连接层的卷积神经网络。可以用矩阵运算来表示该神经网络，计算图如图9所示（本文中不给出细节，细节请见即将出版的《深入理解神经网络》）。

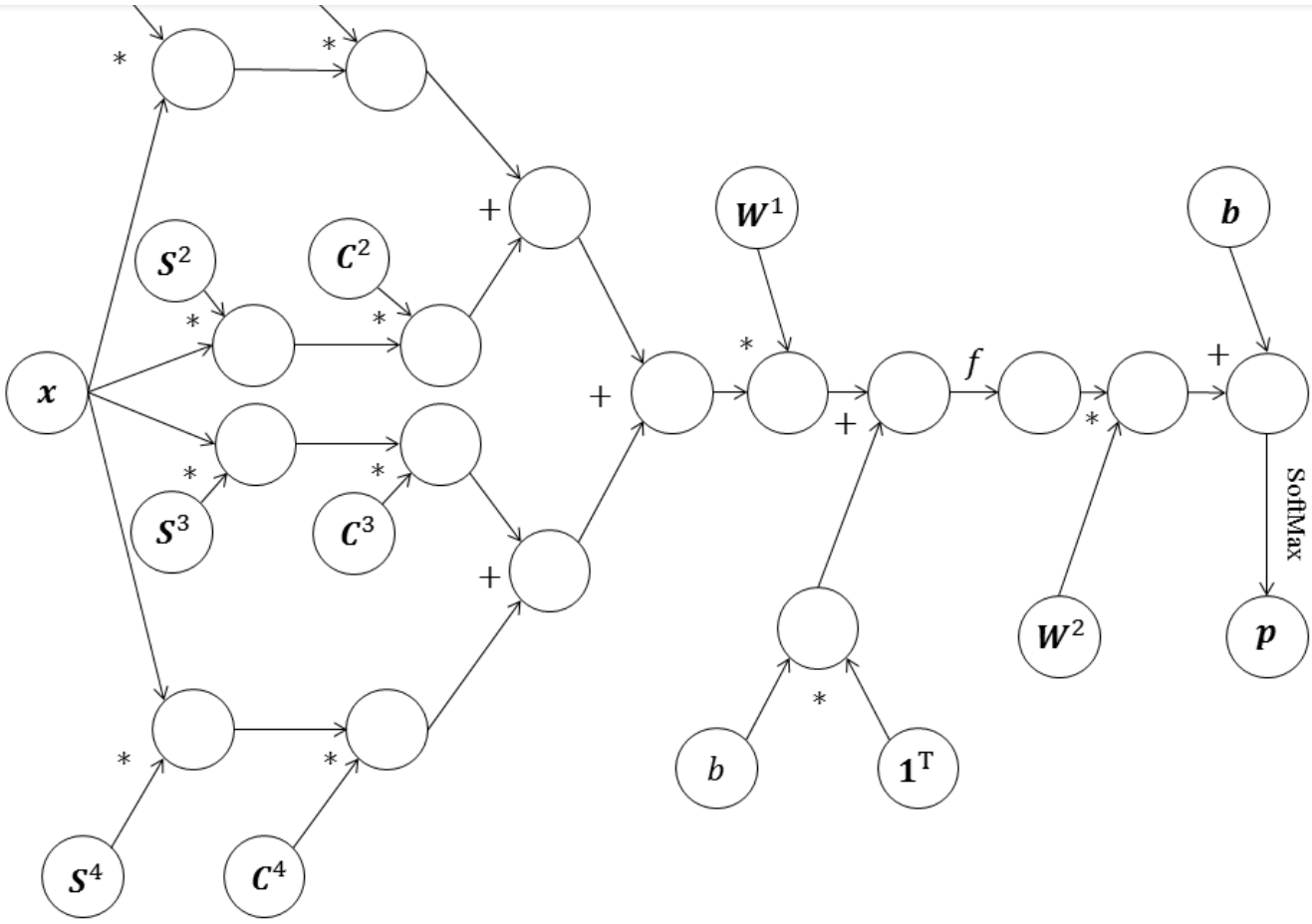


图9 卷积神经网络的计算图

由于我们将计算限制为矩阵/向量的加法和乘法以及激活函数，而没有其他类型的计算，故该计算图稍复杂。如果加入更多的计算类型，则可以更简洁地表示这个卷积神经网络，但是这个例子说明计算图足以表达复杂的神经网络。

2 自动求导

对于计算图中的任意节点 x 和 y ，如果存在一条从 x 到 y 的有向路径，其他节点都看作常量，则 y 可以视为 x 的映射，如图10所示。

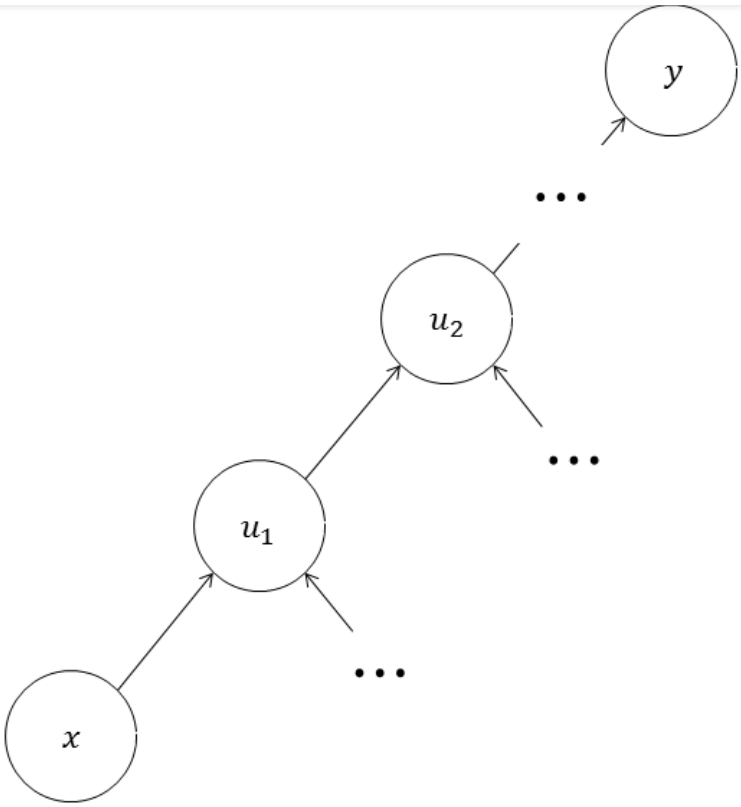


图10 计算图中的一条有向路径表示一个映射

图10省略了计算图的其他部分。由 x 计算 y 是一个多重复合映射。如果 x 是 n 维向量， y 是 m 维向量，则该计算图表示的计算是一个 $R^n \to R^m$ 的映射。这个映射的“导数”是一个 $m \times n$ 的矩阵，即雅可比矩阵。根据链式法则， y 对 x 的雅可比矩阵是：

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial u^k}\frac{\partial u^k}{\partial u^{k-1}} \cdots\cdots \frac{\partial u^2}{\partial u^1}\frac{\partial u^1}{\partial x}$$

本文用偏导数相同的符号表示雅可比矩阵，例如 $\frac{\partial y}{\partial x}$ ，注意 y 和 x 是向量， $\frac{\partial y}{\partial x}$ 是矩阵。则该计算图如果要计算 y 对 x 在 x^* 的雅可比矩阵，则需要计算 u^1 对 x 在 x^* 的雅可比矩阵 $\frac{\partial u^1}{\partial x}$ ， u^2 对 u^1 在 $u^1(x^*)$ 的雅可比矩阵 $\frac{\partial u^2}{\partial u^1}$ ，以此类推。如果结果 y 是标量，则雅可比矩阵 $\frac{\partial y}{\partial x}$ 是一个行向量，是 y 对 x 在 x^* 的梯度的转置。

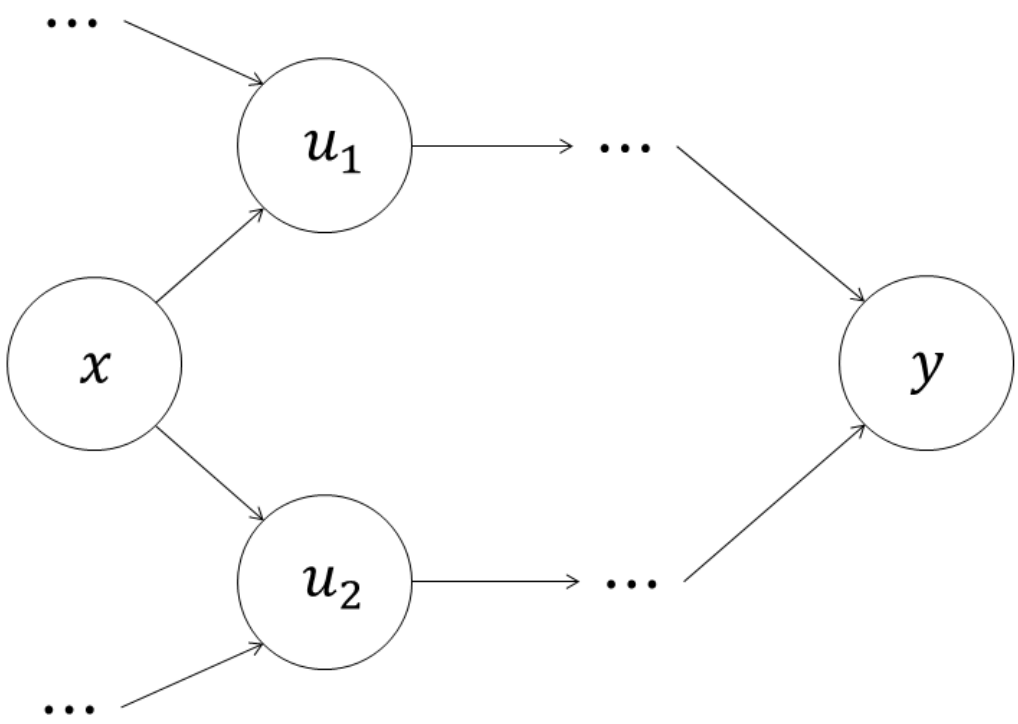


图11 一个节点有多个子节点

如果一个节点有多个子节点，如图11所示， x 节点经过两个子节点 u_1 和 u_2 计算出最终结果。假设现在 $\frac{\partial y}{\partial u_1}$ 和 $\frac{\partial y}{\partial u_2}$ 已知，该如何计算 $\frac{\partial y}{\partial x}$ 呢？如果 u_1 是 m 维向量， u_2 是 k 维向量，可将它们连在一起构成 $m+k$ 维向量：

$$u = \begin{pmatrix} u^1 \\ u^2 \end{pmatrix}$$

y 对 u 的雅可比是 $1 \times (m+k)$ 矩阵：

$$\frac{\partial y}{\partial u} = \left(\frac{\partial y}{\partial u_1^1} \cdots \frac{\partial y}{\partial u_m^1}, \frac{\partial y}{\partial u_1^2} \cdots \frac{\partial y}{\partial u_k^2} \right) = \left(\frac{\partial y}{\partial u^1}, \frac{\partial y}{\partial u^2} \right)$$

u 对 x 的雅可比是 $(m+k) \times n$ 矩阵：

$$\frac{\partial u}{\partial x} = \begin{pmatrix} \vdots & \ddots & \vdots \\ \frac{\partial u_m^1}{\partial x_1} & \cdots & \frac{\partial u_m^1}{\partial x_n} \\ \frac{\partial u_1^2}{\partial x_1} & \cdots & \frac{\partial u_1^2}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial u_k^2}{\partial x_1} & \cdots & \frac{\partial u_k^2}{\partial x_n} \end{pmatrix} = \begin{pmatrix} \frac{\partial u^1}{\partial x} \\ \frac{\partial u^2}{\partial x} \end{pmatrix}$$

根据链式法则， y 对 x 的雅可比是 $1 \times n$ 矩阵：

$$\frac{\partial y}{\partial x} = \left(\frac{\partial y}{\partial u^1}, \frac{\partial y}{\partial u^2} \right) \begin{pmatrix} \frac{\partial u^1}{\partial x} \\ \frac{\partial u^2}{\partial x} \end{pmatrix} = \frac{\partial y}{\partial u^1} \frac{\partial u^1}{\partial x} + \frac{\partial y}{\partial u^2} \frac{\partial u^2}{\partial x}$$

如果有两个以上子节点，同样可以证明：

$$\frac{\partial y}{\partial x} = \sum_{i=1}^{s} \frac{\partial y}{\partial u^i} \frac{\partial u^i}{\partial x}$$

这对于自动求导非常有利：如果一个节点有多个子节点，将结果节点对这些子节点的雅可比与这些子节点对该节点的雅可比相乘再求和，就得到了结果节点对该节点的雅可比。有时需要计算 y 对多个节点的梯度，如图12所示。

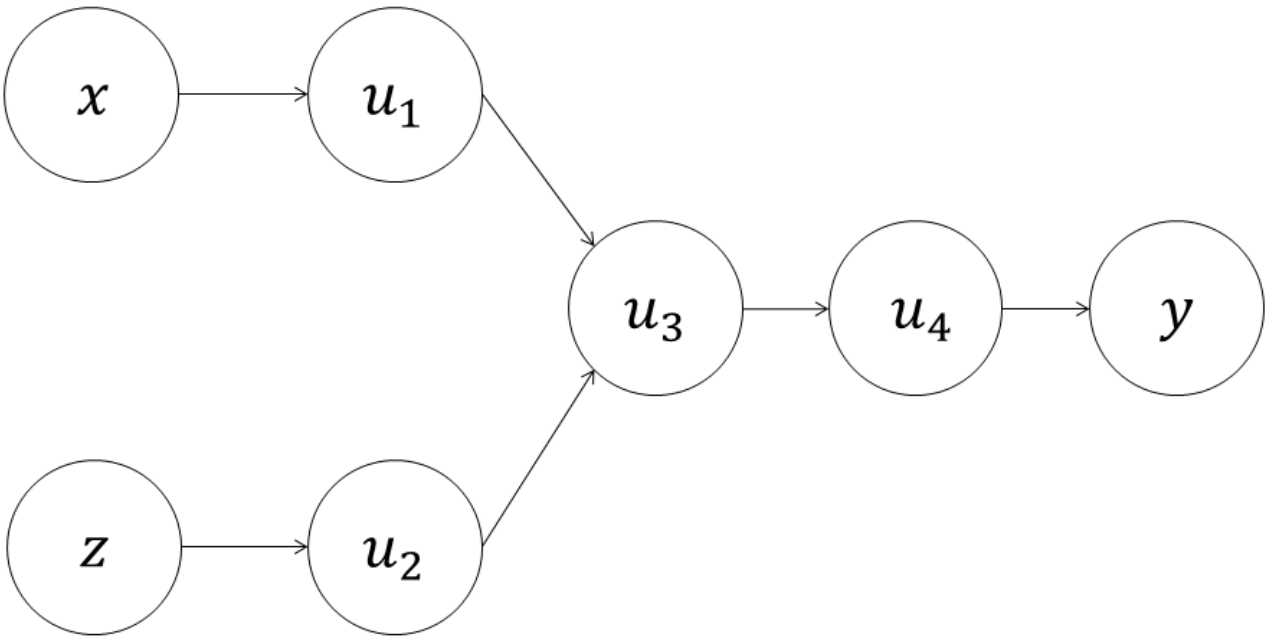


图12 计算节点对多个上游节点的梯度

假设图12中的 x 是 n 维向量， z 是 m 维向量， y 是标量，可将该计算图视作映射 $f: R^{n+m} \rightarrow R$ 。可以两次应用上式分别计算 f 对 x 和 z 的梯度，但是注意下式：

$$\frac{\partial y}{\partial u^3} = \frac{\partial y}{\partial u^4}\frac{\partial u^4}{\partial u^3}$$

该式会被计算两次，如果将其结果保存起来，则可以节省计算量。这就是自动求导的核心所在：保存结果节点对计算路径上各个节点的雅可比，并用它们计算结果节点对更上游节点的雅可比。中间节点的雅可比就是被“反向传播”的对象，计算图自动求导是广义的反向传播。

3 自动求导的实现

本节讨论计算图自动求导的实现，我
两个属性：value 和 jacobi。value

- $evaluate()$ 计算节点的值，如果有父节点的值尚未计算，则抛出异常；
- $get\_children()$ 返回所有子节点，若无子节点则返回空集；
- $get\_parents()$ 返回所有父节点，若无父节点则返回空集；
- $get\_jacobi()$ 接受一个父节点，计算本节点对这个父节点的雅可比。注意本方法与属性的区别，是结果节点对本节点的雅可比，是计算并返回本节点对某个父节点的雅可比。

若要计算某个节点的值，则它的所有父节点必须先被计算。信息沿着计算图路径从前向后传播，这就相当于神经网络的前向传播。以下 forward_propagation 函数实现了计算某节点值的前向传播过程。

```
function forward_propagation(v):
    for p in v.get_parents():
        if p.value is NULL:
            forward_propagation(p)
    v.evaluate()
    return v.value
```

节点的 evaluate() 执行的计算可以是标量计算，矩阵/向量计算或者其他更复杂的计算，忽略各种计算的差异，将它们的时间复杂度都视为 $O(1)$ ，若计算图有个节点，则它的时间复杂度是 $O(n)$ 。

若要计算某个节点对它的某个上游节点的雅可比，则沿着计算图路径从后向前，逐节点计算结果节点对它们的雅可比。在所有子节点的雅可比计算完成后，则父节点的雅可比可以计算。中间节点的雅可比可能会被使用多次，将它们保存在对象属性（jacobi）中，可避免重复计算。以下 back_propagation 函数计算节点对某个上游节点的雅可比。

```
function back_propagation(y,v):
    if v.jacobi is NULL:
        if  v == y :
            v.jacobi = I
        else:
            v.jacobi = O # O 为
            for c in v.get_chil
```

```
    return v.jacobi
```

get_jacobi 对计算路径上的每条边执行一次， n 个节点的计算图最多有 $C_n^2 = \frac{n(n-1)}{2!}$ 条边，如果认为所有的时间复杂度都是 $O(1)$ ，则自动求导的时间复杂度是 $O(n^2)$ 。试想如果粗暴地直接应用链式法则，则中间节点的雅可比有可能被重复计算多次。反向传播的本质是以空间换时间，将中间节点的雅可比保存起来，重复使用。父节点的雅可比根据其子节点的雅可比计算，信息沿着计算路径向前传播，这就是反向传播的含义。

现在我们用原生Python和Numpy库实现计算图以及自动求导，并用计算图搭建多层全连接神经网络。与TensorFlow不同，我们的节点不是三维乃至更高维度的张量（Tensor），而是矩阵（包括向量和标量）。根据之前的讨论，原则上只用矩阵就可以实现任何计算。首先，我们实现计算图节点的基类，代码如下：

```
import numpy as np

from graph import Graph, default_graph
from util import *


class Node:
    """
    计算图节点类基类
    """

    def __init__(self, *parents):
        self.parents = parents  # 父节点列表
        self.children = []  # 子节点列表
        self.value = None  # 本节点的值
        self.jacobi = None  # 结果节点对本节点的雅可比矩阵
        self.graph = default_graph  # 计算图对象，默认为全局对象default_graph

        # 将本节点添加到父节点的子节点列表中
        for parent in self.parents:
            parent.children.append(self)

        # 将本节点添加到计算图中
        self.graph.add_node(self)

    def set_graph(self, graph):
        """
        设置计算图
```

```
        self.graph = graph

    def get_parents(self):
        """
        获取本节点的父节点
        """
        return self.parents

    def get_children(self):
        """
        获取本节点的子节点
        """
        return self.children

    def forward(self):
        """
        前向传播计算本节点的值，若父节点的值未被计算，则递归调用父节点的forward方法
        """
        for node in self.parents:
            if node.value is None:
                node.forward()

        self.compute()

    def compute(self):
        """
        抽象方法，根据父节点的值计算本节点的值
        """
        pass

    def get_jacobi(self, parent):
        """
        抽象方法，计算本节点对某个父节点的雅可比矩阵
        """
        pass

    def backward(self, result):
        """
        反向传播，计算结果节点对本节点的雅可比矩阵
        """
        if self.jacobi is None:
            if self is result:
                self.jacobi = r
```

```
                for child in self.get_children():
                    if child.value is not None:
                        self.jacobi += child.backward(result) * child.get_jacobi(self)

        return self.jacobi

    def clear_jacobi(self):
        """
        清空结果节点对本节点的雅可比矩阵
        """
        self.jacobi = None

    def dimension(self):
        """
        返回本节点的值展平成向量后的维数。展平方式固定式按行排列成一列。
        """
        return self.value.shape[0] * self.value.shape[1]

    def shape(self):
        """
        返回本节点的值作为矩阵的形状：（行数，列数）
        """
        return self.value.shape

    def reset_value(self, recursive=True):
        """
        重置本节点的值，并递归重置本节点的下游节点的值
        """

        self.value = None

        if recursive:
            for child in self.children:
                child.reset_value()
```

代码中，Graph类是计算图类，default_graph对象是一个全局的计算图对象，它们的实现我们稍后呈现。Node类是计算图节点的基类，所有类型的节点都继承自Node类。Node类实现了计算图节点的一些公共方法，它的构造函数接受可变数量的Node类对象，作为本节点的父节点，本节点的值用这些父节点的值计算而得。

点对本节点的雅可比矩阵。若它们为空，则表示尚没有被计算。构造函数将通过参数传进来的节点加入本节点的父节点列表，再将本节点加入所有父节点的子节点列表，最后将本节点加入计算图对象的节点列表。

接下来是一些简单的设置和获取方法，不必赘述。forward是执行前向传播，计算本节点的值的方法，它是第3节的伪代码的实现。为了计算本节点的值，forward方法首先检查父节点的值是否为空，若某个父节点的值为空，则递归调用该父节点的forward方法。确保所有父节点的值都已被计算后，forward方法调用compute方法计算本节点的值。在基类中，compute方法是一个抽象方法，需要具体的节点子类去覆盖实现各种不同的计算。get_jacobi方法是另一个抽象方法，它接受一个父节点，计算当前本节点对这个父节点的雅可比矩阵。

backward方法是实现反向传播的关键，它接受一个被视为计算图计算结果的节点，计算当前该结果节点对本节点的雅可比矩阵。backward方法是第3节的伪代码的实现。若本节点的jacobi属性为空，则表示结果节点对本节点的雅可比矩阵尚未被计算。若本节点就是结果节点，则雅可比矩阵为单位矩阵，否则利用链式法则根据结果节点对各个子节点的雅可比矩阵计算结果节点对本节点的雅可比矩阵。

接下来的两个方法容易理解，不再赘述。reset_value方法将本节点的值置空。因为本节点的值影响下游节点的值，所以应该递归置空所有下游节点的值。是否递归取决于参数recursive。

有了基类，我们就可以实现各种不同的节点类，它们执行不同计算。我们首先实现Variable类，它保存一个变量。Variable对象没有父节点，它们是计算图的终端节点。可以随机初始化Variable对象的值，也可以为Variable对象赋值。Variable类的代码如下：

```
class Variable(Node):
    """
    变量节点
    """

    def __init__(self, dim, init=False, trainable=True):
        """
        变量节点没有父节点，构造函数接受变量的维数，以及变量是否参与训练的标识
        """
        Node.__init__(self)
        self.dim = dim

        # 如果需要初始化，则以正态分布随机初始化变量的值
        if init:
            self.value = np.mat(np.random.normal(0, 0.001, self.dim))

        # 变量节点是否参与训练
```

```
    def set_value(self, value):
        """
        为变量赋值
        """
        assert isinstance(value, np.matrix) and value.shape == self.dim

        # 本节点的值被改变，重置所有下游节点的值
        self.reset_value()
        self.value = value
```

Variable类的构造函数接受dim参数，确定变量的形状。init参数表示是否要随机初始化变量的值。trainable参数表示本变量节点是否参与训练。set_value方法为Variable类独有，它设置变量的值。若变量的值被改变，则计算图中所有下游节点的值都将作废，所以set_value方法调用reset_value方法递归清除本节点以及所有下游节点的值。Variable对象的值是被赋予或被随机初始化的，所以它不用实现compute方法。Variable对象没有父节点，它也不用实现get_jacobi方法。接下来我们实现向量加法节点，代码如下：

```
class Add(Node):
    """
    矩阵加法
    """

    def compute(self):
        assert len(self.parents) == 2 and self.parents[0].shape() == self.parents[1].s
        self.value = self.parents[0].value + self.parents[1].value

    def get_jacobi(self, parent):
        return np.mat(np.eye(self.dimension()))  # 矩阵之和对其中任一个矩阵的雅可比矩阵是单
```

Add 类的compute方法将两个父节点的值相加。get_jacobi方法求当前Add对象对某一个父节点的雅可比矩阵。向量加法是一个 $R^n \rightarrow R^n$ 的映射，它对其中某一个参与加和的向量的雅可比矩阵是 $n \times n$ 单位矩阵。向量内积（点积）节点的代码如下：

```
class Dot(Node):
    """
    向量内积
    """

    def compute(self):
        assert len(self.parents
```

```
    def get_jacobi(self, parent):
        if parent is self.parents[0]:
            return self.parents[1].value.T
        else:
            return self.parents[0].value.T
```

在我们的实现中，值一律采用numpy.matrix类型，即矩阵。 n 维向量就是 $n \times 1$ 矩阵，标量就是 1×1 矩阵。Dot类的compute方法计算两个父节点的内积。因为节点的值是numpy.matrix类型，经过重载的*运算执行的是矩阵相乘，对于一个行向量（列向量的转置）和一个列向量来说，计算的结果是一个 1×1 矩阵（标量），即这两个向量的内积。get_jacobi方法计算Dot节点对某一个父节点的雅可比矩阵，请看下式：

$$\frac{f\left(x^T y\right)}{x_i} = \frac{\partial}{\partial x_i}\left(\sum_{j=1}^{n} x_j y_j\right) = y_i$$

所以有：

$$\left(\frac{f\left(x^T y\right)}{x_1}, \frac{f\left(x^T y\right)}{x_2}, \ldots, \frac{f\left(x^T y\right)}{x_n}\right) = \left(y_1, y_2, \ldots, y_n\right)$$

内积对某个向量的雅可比矩阵是另一个向量的转置，这就是Dot类的get_jacobi方法所返回的值。矩阵相乘节点的代码如下：

```
class MatMul(Node):
    """
    矩阵乘法
    """

    def compute(self):
        assert len(self.parents) == 2 and self.parents[0].shape()[1] == self.parents[1
        self.value = self.parents[0].value * self.parents[1].value

    def get_jacobi(self, parent):
        """
        将矩阵乘法视作映射，求映射对参与计算的矩阵的雅克比矩阵
        """
```

```
        zeros = np.mat(np.zeros((self.dimension(), parent.dimension())))
        if parent is self.parents[0]:
            return fill_diagonal(zeros, self.parents[1].value.T)
        else:
            jacobi = fill_diagonal(zeros, self.parents[0].value)
            row_sort = np.arange(self.dimension()).reshape(self.shape()[::-1]).T.ravel
            col_sort = np.arange(parent.dimension()).reshape(parent.shape()[::-1]).T.r
            return jacobi[row_sort,:][:,col_sort]
```

之前讨论原理时我们提到过，不论输入输出的形状是什么，一个计算节点都可以视作向量到向量的映射，只不过我们需要将矩阵展平。例如一个 $m \times n$ 矩阵乘以一个 $n \times k$ 矩阵，得到一个 $m \times k$ 矩阵，若将第一个矩阵视作自变量，则矩阵乘法可以视作是一个 $R^{m \times n} \to R^{m \times k}$ 的映射，它的雅克比矩阵是一个 $(m \times k) \times (m \times n)$ 的矩阵。MatMul类节点的compute很简单，就是将两个父节点的value相乘（Numpy对矩阵类型冲够了*运算符，执行矩阵乘法），但它的get_jacobi较复杂，读者可以自己尝试推导一下。接下来我们实现Logistic节点，它对父节点的每个分量施加Logistic函数，代码如下：

```
class Logistic(Node):
    """
    对矩阵的元素施加Logistic函数
    """

    def compute(self):
        x = self.parents[0].value
        self.value = np.mat(1.0 / (1.0 + np.power(np.e, np.where(-x > 1e2, 1e2, -x))))

    def get_jacobi(self, parent):
        return np.diag(np.mat(np.multiply(self.value, 1 - self.value)).A1)
```

可以利用Logistic函数的值方便地求得其导数。Logistic类的get_jacobi方法利用已经计算好的value成员计算对父节点的雅可比矩阵。该雅可比矩阵是一个对角矩阵，对角线元素是Logistic函数对父节点某个元素的导数。类似地，ReLU节点的代码如下：

```
class ReLU(Node):
    """
    对矩阵的元素施加ReLU函数
    """

    def compute(self):
```

```
    def get_jacobi(self, parent):
        return np.diag(np.where(self.parents[0].value.A1 > 0.0, 1.0, 0.0))
```

ReLU节点的值和雅可比矩阵都很容易计算，代码自明，不再赘述。接下来，我们实现SoftMax节点，代码如下：

```
class SoftMax(Node):
    """
    SoftMax函数
    """

    @staticmethod
    def softmax(a):
        a[a > 1e2] = 1e2  # 防止指数过大
        ep = np.power(np.e, a)
        return ep / np.sum(ep)

    def compute(self):
        self.value = SoftMax.softmax(self.parents[0].value)

    def get_jacobi(self, parent):
        """
        我们不实现SoftMax节点的get_jacobi函数，训练时使用CrossEntropyWithSoftMax节点（见下）
        """
        return np.mat(np.eye(self.dimension()))  # 无用
```

SoftMax节点执行的计算我们已经很熟悉了，但是我们不实现它的get_jacobi方法，因为计算SoftMax函数对输入向量的雅可比矩阵较复杂，但是如果将SoftMax函数的输出送给交叉熵，计算交叉熵损失对SoftMax函数的输入向量的雅可比矩阵是相当简单的，所以我们实现一个将SoftMax函数与交叉熵损失合二为一的节点类，代码如下：

```
class CrossEntropyWithSoftMax(Node):
    """
    对第一个父节点施加SoftMax之后，再以第二个父节点为标签One-Hot向量计算交叉熵
    """

    def compute(self):
        prob = SoftMax.softmax(self.parents[0].value)
        self.value = np.mat(-np
```

```
        # 这里存在重复计算，但为了代码清晰简洁，舍弃进一步优化
        prob = SoftMax.softmax(self.parents[0].value)
        if parent is self.parents[0]:
            return (prob - self.parents[1].value).T
        else:
            return (-np.log(prob)).T
```

CrossEntropyWithSoftMax节点的compute方法对第一个父节点的值施加SoftMax函数，再与第二个父节点的值计算交叉熵。第二个父节点的值是类别标签的One-Hot编码向量。get_jacobi方法对第一个父节点计算雅可比，对第二个父节点的雅可比矩阵不会被使用，但是也实现在此。

至此，我们实现了几种典型的计算图节点，它们对于我们接下来要做的事情已经足够。有兴趣的读者可以自己实现一些其他类型的节点。接下来我们实现Graph类，代码如下：

```
import os


class Graph:
    """
    计算图类
    """

    def __init__(self):
        self.nodes = []  # 计算图内的节点的列表

    def add_node(self, node):
        """
        添加节点
        """
        self.nodes.append(node)

    def clear_jacobi(self):
        """
        清除图中全部节点的雅可比矩阵
        """
        for node in self.nodes:
            node.clear_jacobi()

    def reset_value(self):
        """
        重置图中全部节点的值
```

```
            node.reset_value(False)  # 每个节点不递归清除自己的子节点的值

    def draw(self, ax=None):
        try:
            import networkx as nx
            import matplotlib.pyplot as plt
            from matplotlib.colors import ListedColormap
            import numpy as np
        except:
            raise Exception("Need Module networkx")

        G = nx.Graph()

        already = []
        labels = {}
        for node in self.nodes:
            G.add_node(node)
            labels[node] = node.__class__.__name__ + ("{:s}".format(str(node.dim)) if
                           + ("\n[{:.3f}]".format(np.linalg.norm(node.jacobi)) if node
            for c in node.get_children():
                if {node, c} not in already:
                    G.add_edge(node, c)
                    already.append({node, c})
            for p in node.get_parents():
                if {node, p} not in already:
                    G.add_edge(node, p)
                    already.append({node, c})

        savefig = False
        if ax is None:
            fig = plt.figure(figsize=(12, 12))
            ax = fig.add_subplot(111)
            savefig = True

        ax.clear()
        ax.axis("on")
        ax.grid(True)

        pos = nx.spring_layout(G, seed=42)

        # 有雅克比的变量节点
        cm = plt.cm.Reds
        nodelist = [n for n in
```

```
                        node_size=2000, alpha=1.0, ax=ax)

    # 无雅克比的变量节点
    nodelist = [n for n in self.nodes if n.__class__.__name__ == "Variable" and n.
    nx.draw_networkx_nodes(G, pos, nodelist=nodelist, node_color="#999999", cmap=c
                        node_size=2000, alpha=1.0, ax=ax)

    # 有雅克比的计算节点
    nodelist = [n for n in self.nodes if n.__class__.__name__ != "Variable" and n.
    colorlist = [np.linalg.norm(n.jacobi) for n in nodelist]
    nx.draw_networkx_nodes(G, pos, nodelist=nodelist, node_color=colorlist, cmap=c
                        node_size=2000, alpha=1.0, ax=ax)

    # 无雅克比的中间
    nodelist = [n for n in self.nodes if n.__class__.__name__ != "Variable" and n.
    nx.draw_networkx_nodes(G, pos, nodelist=nodelist, node_color="#999999", cmap=c
                        node_size=2000, alpha=1.0, ax=ax)

    # 边
    nx.draw_networkx_edges(G, pos, width=2, edge_color="#014b66", ax=ax)
    nx.draw_networkx_labels(G, pos, labels=labels, font_weight="bold", font_color=
                         font_family='arial', ax=ax)

    # 保存图像
    if savefig:
        file = "pic/computing_graph.png"
        if os.path.exists(file):
            os.remove(file)
        plt.savefig(file)  # save as png


# 全局默认计算图
default_graph = Graph()
```

我们的Graph类较简单，它只保留计算图的全部节点，实现清除所有节点的雅可比矩阵和值的方法。default_graph是一个全局的Graph对象，默认情况下所有节点都将被加入到default_graph中。最后，我们实现训练优化器类。所有优化器类都继承自一个基类，代码如下：

```
from graph import Graph, default_graph
from node import *
```

```
    """
    优化器基类
    """

    def __init__(self, graph, target, batch_size=12):
        assert isinstance(target, Node) and isinstance(graph, Graph)
        self.graph = graph
        self.target = target
        self.batch_size = batch_size

        # 为每个参与训练的节点累加一个Mini Batch的全部样本的梯度
        self.acc_gradient = dict()
        self.acc_no = 0

    def one_step(self):
        """
        计算并累加样本的梯度，一个Mini Batch结束后执行变量更新
        """
        self.forward_backward()

        self.acc_no += 1
        if self.acc_no >= self.batch_size:
            self.update()
            self.acc_gradient.clear()  # 清除梯度累加
            self.acc_no = 0  # 清除计数

    def get_gradient(self, node):
        """
        返回一个Mini Batch的样本的平均梯度
        """
        assert node in self.acc_gradient
        return self.acc_gradient[node] / self.batch_size

    def update(self):
        """
        抽象方法，利用梯度更新可训练变量
        """

        pass

    def forward_backward(self):
        """
        前向传播计算结果节点的值并
```

```
        # 清除计算图中所有节点的雅可比矩阵
        self.graph.clear_jacobi()

        # 前向传播计算结果节点
        self.target.forward()

        # 反向传播计算梯度
        for node in self.graph.nodes:
            if isinstance(node, Variable) and node.trainable:
                node.backward(self.target)

                # 最终结果（标量）对节点值（视作向量）的雅克比是一个行向量，将其转置是梯度（列)
                # 这里将梯度 reshape 成与节点值相同的形状，好对节点值进行更新。
                gradient = node.jacobi.reshape(node.shape())
                if node not in self.acc_gradient:
                    self.acc_gradient[node] = gradient
                else:
                    self.acc_gradient[node] += gradient
```

Optimizer类的构造函数接受一个Graph对象，一个作为优化目标的节点对象，以及Mini Batch的样本数量。因为我们的计算图节点只能包含一个向量，所以不能利用更高的维度在节点值中包含整个Mini Batch。于是，我们对Mini Batch的实现是这样的：对一个Mini Batch中的样本依次执行前向传播和反向传播，将参与训练的变量的梯度累加在acc_gradient中，一个Mini Batch计算完毕后执行变量更新，这时使用Mini Batch中多个样本的平均梯度。

one_step方法调用forward_backward方法对一个样本执行前向传播和反向传播，将目标节点对各个变量的梯度累加在acc_gradient中，最后清除所有节点的雅可比矩阵。one_step方法计数样本数，当样本数达到batch_size时，执行update方法更新变量，并清除累加梯度以及计数。get_gradient方法返回一个Mini Batch的所有样本的平均梯度。

update是抽象方法，利用Mini Batch的平均梯度以各种不同的方法更新变量值。update方法将在具体的优化器类中得到实现。forward_backward方法执行前向传播，计算目标节点的值，然后反向传播计算目标节点对每个参与训练的变量节点的雅可比矩阵。原始梯度下降的优化器实现如下：

```
class GradientDescent(Optimizer):
    """
    梯度下降优化器
    """

    def __init__(self, graph, t
```

```
    def update(self):

        for node in self.graph.nodes:
            if isinstance(node, Variable) and node.trainable:
                gradient = self.get_gradient(node)

                node.set_value(node.value - self.learning_rate * gradient)
```

除了Optimizer 类的参数，GradientDescent类的构造函数还接受learning_rate参数，即学习率。GradientDescent类的update方法相当简单，就是获得平均梯度，乘以学习率并取反后更新到变量节点的当前值上。RMSProp和Adam优化器的代码如下：

```
class RMSProp(Optimizer):
    """
    RMSProp优化器
    """

    def __init__(self, graph, target, learning_rate=0.01, beta=0.9, batch_size=32):
        Optimizer.__init__(self, graph, target, batch_size)
        self.learning_rate = learning_rate

        assert 0.0 < beta < 1.0
        self.beta = beta

        self.s = dict()

    def update(self):

        for node in self.graph.nodes:
            if isinstance(node, Variable) and node.trainable:
                gradient = self.get_gradient(node)

                if node not in self.s:
                    self.s[node] = np.power(gradient, 2)
                else:
                    self.s[node] = self.beta * self.s[node] + (1 - self.beta) * np.pow

                node.set_value(node.value - self.learning_rate * gradient / (np.sqrt(s
```



```
class Adam(Optimizer):
```

```
    """

    def __init__(self, graph, target, learning_rate=0.01, beta_1=0.9, beta_2=0.99, bat
        Optimizer.__init__(self, graph, target, batch_size)
        self.learning_rate = learning_rate

        assert 0.0 < beta_1 < 1.0
        self.beta_1 = beta_1

        assert 0.0 < beta_2 < 1.0
        self.beta_2 = beta_2

        self.s = dict()
        self.v = dict()

    def update(self):

        for node in self.graph.nodes:
            if isinstance(node, Variable) and node.trainable:
                gradient = self.get_gradient(node)

                if node not in self.s:
                    self.v[node] = gradient
                    self.s[node] = np.power(gradient, 2)
                else:
                    self.v[node] = self.beta_1 * self.v[node] + (1 - self.beta_1) * gr
                    self.s[node] = self.beta_2 * self.s[node] + (1 - self.beta_2) * np

                node.set_value(node.value - self.learning_rate * self.v[node] / np.sqr
```

关于RMSProp和Adam，专栏之前的文章已经有了详细论述和Python实现，这里只是把相同逻辑实现在我们的计算图优化器框架内，就不再详细解释了。有兴趣的读者可以自己实现其他优化器。最后，我们用这个计算图框架搭建一个多层全连接神经网络，并用它分类MNIST手写数字，代码如下：

```
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
from tensorflow.examples.tutori
```

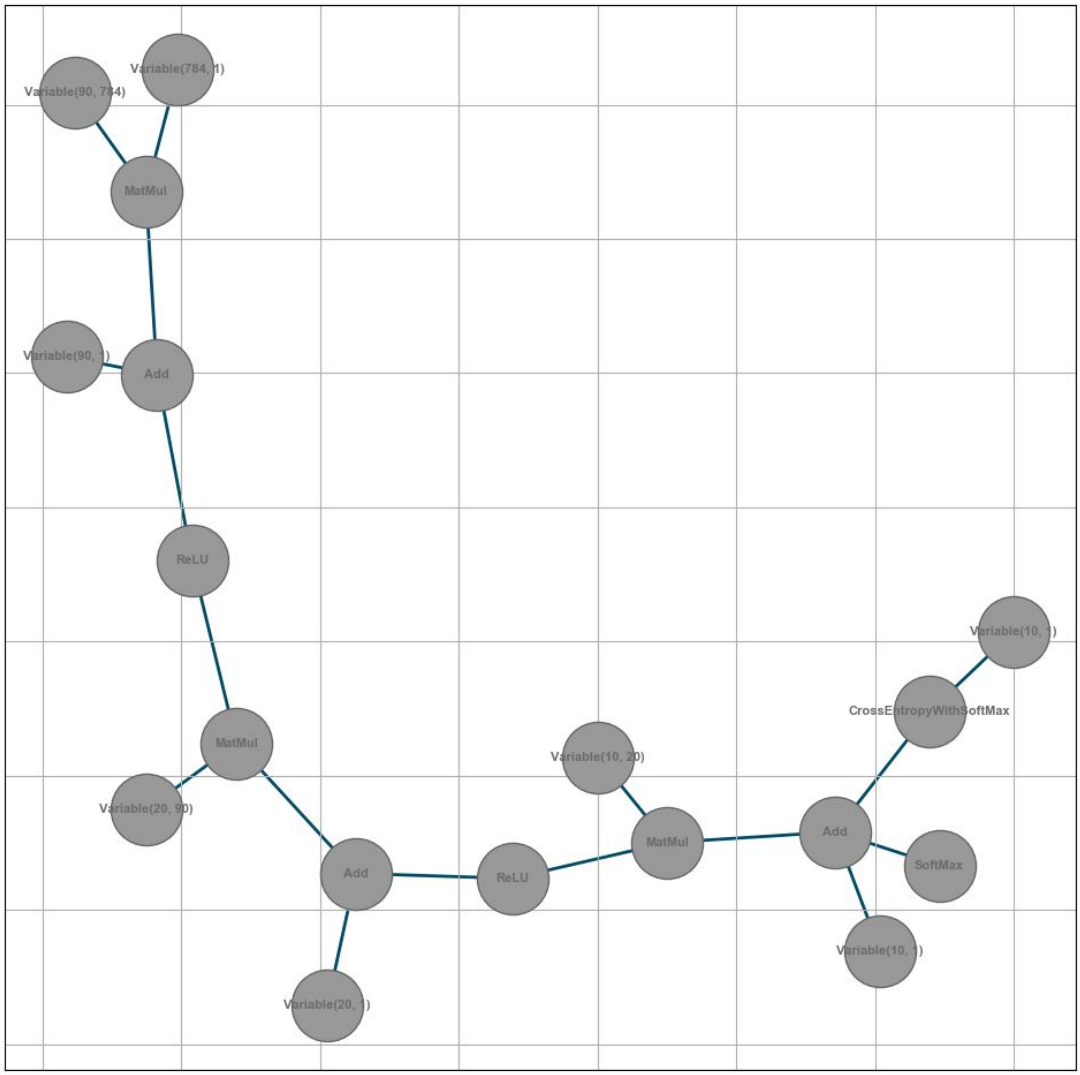
```
mnist = input_data.read_data_sets("E:/train_pic/mnist_dataset/", one_hot=True)
train_x = mnist.train.images
train_y = mnist.train.labels

test_x = mnist.test.images
test_y = mnist.test.labels


# 构造多分类逻辑回归计算图，输入变量
# 构造多层全连接神经网络的计算图
X = Variable((784, 1), trainable=False)  # 10维特征变量

# 第一隐藏层12个神经元
hidden_1 = ReLU(
    Add(
        MatMul(
            Variable((90, 784), True),  # 90x784权值矩阵
            X
        ),
        Variable((90, 1), True)  # 90维偏置向量
    )
)

# 第一隐藏层8个神经元
hidden_2 = ReLU(
    Add(
        MatMul(
            Variable((20, 90), True),  # 20x90权值矩阵
            hidden_1
        ),
        Variable((20, 1), True)  # 20维偏置向量
    )
)

# 输出层6个神经元
logits = Add(
    MatMul(
        Variable((10, 20), True),  # 10x20权值矩阵
        hidden_2
    ),
    Variable((10, 1), True)  #
)
```

```
# 训练标签
label = Variable((10, 1), trainable=False)

# 交叉熵损失
loss = CrossEntropyWithSoftMax(logits, label)

# 绘制计算图
default_graph.draw()

# Adam优化器
optimizer = Adam(default_graph, loss, 0.01, batch_size=32)

# 训练
for e in range(6):

    # 每个epoch在测试集上评估模型正确率
    probs = []
    losses = []
    for i in range(len(test_x)):
        X.set_value(np.mat(test_x[i, :]).T)
        label.set_value(np.mat(test_y[i, :]).T)

        # 前向传播计算概率
        prob.forward()
        probs.append(prob.value.A1)

        # 计算损失值
        loss.forward()
        losses.append(loss.value[0, 0])

    # 取概率最大的类别为预测类别
    pred = np.argmax(np.array(probs), axis=1)
    truth = np.argmax(test_y, axis=1)
    accuracy = accuracy_score(truth, pred)

    print("Epoch: {:d}，测试集损失值：{:.3f}，测试集正确率：{:.2f}%".format(e + 1, np.mean

    for i in range(len(train_x)):
        X.set_value(np.mat(train_x[i, :]).T)
        label.set_value(np.mat(train_y[i, :]).T)

        # 执行一步优化
        optimizer.one_step()
```

```
        if i % 500 == 0:
            loss.forward()
            print("Iteration: {:d}, Mini Batch损失值：{:.3f}".format(i + 1, loss.value[
```

```
# 训练结束后打印最终评价
print("验证集正确率：{:.3f}".format(accuracy_score(truth, pred)))
print(classification_report(truth, pred))
```

我们将这个神经网络的计算图绘制出来：



4 小结

本文介绍了计算图，大部分神经网络都可以用计算图表示。以计算图中的一个节点为最终结果，可以计算它对其他节点的雅可比，这就是计算图的自动求导。在神经网络语境下，自动求导可看作是广义的反向传播。

编辑于 02-14

神经网络 深度学习（Deep Learning） 人工智能

知乎 首发于 **计算主义**

5 条评论

切换为时间排序

写下你的评论...

sgxsfsf 2019-06-18

棒！ `假设现在和已知，该如何计算呢？` 感觉文中丢失了一些符号信息。

赞

张觉非 (作者) 回复 sgxsfsf 2019-06-18

从 word 导进来的，公式都得重新输入，就有漏的地方。多谢提醒，我修改一下。

1

匿名用户 2019-07-01

计算图和graphical model意思差不多啊

赞

readzw 2019-08-16

请问tensorflow就是这么实现的么?

赞

readzw 2019-08-16

我一个模型有上下两层模块，下层模块比较大，只能把batch_size设置的很小。如果我下层分多次算，再把结果一次输入上层，这样能不能节省显存?

赞