转 自动微分(…n)简介——tensorflow核心原理

2018年08月14日 21:…

现代深度学习系统中…了一种技术——自动微分。在此之前，机器学习社区中很少发挥…器，一般都是用Backpropagation…新。手动实现过backprop算法的同学应该可以体会到其中的复杂…易错性，一个好的很好地将这部分难点…以优雅解决这个问题。接下来我们将一起学习这个优雅的技术:-)。…主要来源于陈天奇教的课程CSE599G… differentiation in machine learning: a survey》。

什么是自动…

微分求解大致可以分…

- 手动求解法(M…
- 数值微分法(Nu…
- 符号微分法(Sy…
- 自动微分法(Au…

为了讲明白什么是自…有区分有对比，从而更加深入理解自动微分技术。

手动求解法

手动求解其实就对应…的梯度公式，然后编写代码，代入实际数值，得出真实的梯度。在这样的方式下，每一次我们…型，都要修改对应的…用户手动编写梯度求解的代码，这也是为什么我们需要自动微分技术的原因。

数值微分法

数值微分法是根据导…

$f'(x)=\lim_{h\to 0} f(x+h…$

那么只要hh取很小…便求解导数，并且可以对用户隐藏求解过程，用户只要给出目标函数和要求解的梯度的变量…自动给出相应的梯度…不幸的是，数值微分法计算量太大，求解速度是这四种方法中最慢的，更加雪上加霜的是…roundoff error和t…场景，为了弥补缺点，便有如下center difference approximation：

$f'(x)=\lim_{h\to 0} f(x+h…$ …h

可惜并不能完全消除…然数值微分法有如上缺点，但是由于它实在是太简单实现了，于是很多时候，我们利用它来…法的正确性，比如在…ent check”就是利用数值微分法。

符号微分法

符号微分是代替我们…，实现微分的一些公式比如：

$ddx(f(x)+g(x))=dd…$ …$f(x)(ddxg(x))ddxf(x)g(x)=f'(x)g(x)-f(x)g'(x)g(x)2ddx(f(x)+g(x))=ddxf(x)+ddxg(x)ddx$

$(ddxf(x))g(x)+f(x)(…$ …$x)2$

然后对用户提供的表…数分”求解，什么是具有closed form的呢？也就是必须能写成完整数学表达式的，不能有…的循环结构，条件结…学符号问题，我们能利用现有的代数软件进行符号微分求解，这种程度意义上的“自动微分”很完美了。然而缺点…ed form的数学表达式，另一个有名的缺点是“表达式膨胀”（expression swell）问题，…心就会使得问题符号…终求解速度变慢，对于这个问题请看如下图：

Table 1 Iterations of the logistic map $l_{n+1} = 4l_n(1 - l_n)$, $l_1 = x$ an ne corresp derivatives of l_n with respect to x, illustrating expression swell.

n	l_n	$\frac{d}{dx} l_n$	$\frac{d}{dx} l_n$ (Op ized)
1	x	1	1
2	$4x(1-x)$	$4(1-x)-4x$	$4-8x$
3	$16x(1-x)(1-2x)^2$	$16(1-x)(1-2x)^2-16x(1-2x)^2-64x(1-x)(1-2x)$	$16(1-10x+24x^2-1$
4	$64x(1-x)(1-2x)^2(1-8x+8x^2)^2$	$128x(1-x)(-8+16x)(1-2x)^2(1-8x+8x^2)+64(1-x)(1-2x)^2(1-8x+8x^2)^2-64x(1-2x)^2(1-8x+8x^2)^2-256x(1-x)(1-2x)(1-8x+8x^2)^2$	$64(1-42x+504x^2-26$ $7040x^4-9984x^5+71$ $2048x^7)$

稍不注意，符号微分求解就会如上中间列所示，表达式急剧膨胀，导致问题求解也随着变慢。

自动微分法

终于轮到我们的主角登场，自动微分的存在依赖于它识破如下事实：

> 所有数值计算归根结底是一系列有限的可微算子的组合

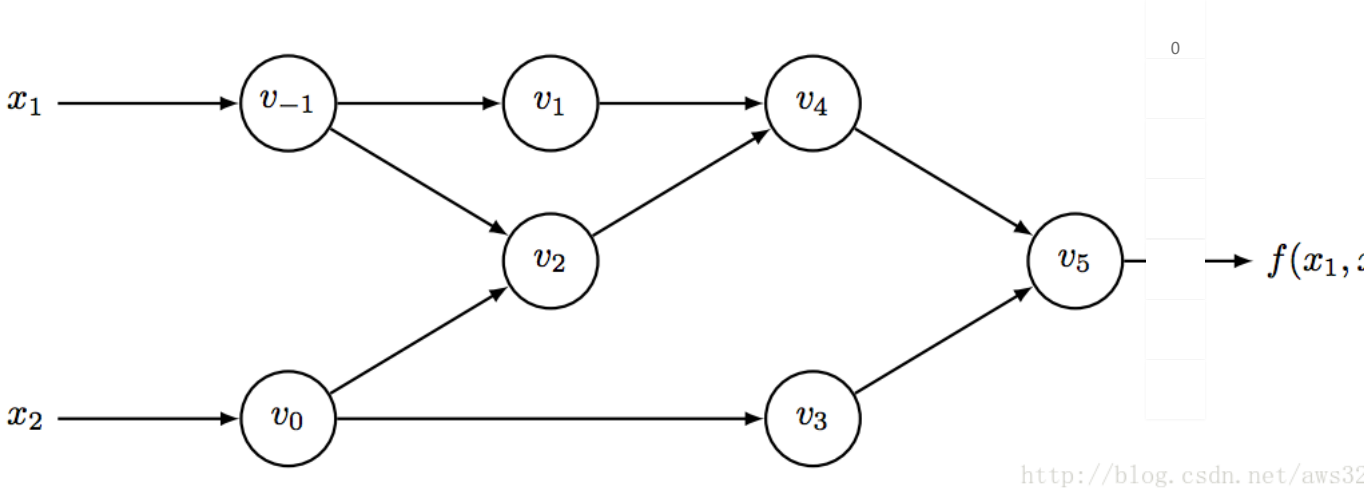
自动微分法是一种介于符号微分和数值微分的方法：数值微分强调一开始直接代入数值近似求解；符号微分强调直接对代数进行求解，最后才代入问题分将符号微分法应用于最基本的算子，比如常数，幂函数，指数函数，对数函数，三角函数等，然后代入数值，保留中间结果，最后再应用于整个函数相当灵活，可以做到完全向用户隐藏微分求解过程，由于它只对基本函数或常数运用符号微分法则，所以它可以灵活结合编程语言的循环结构，条件结动微分和不使用自动微分对代码总体改动非常小，并且由于它的计算实际是一种图计算，可以对其做很多优化，这也是为什么该方法在现代深度学习系应用。

自动微分Forward Mode

考察如下函数：

f(x1,x2)=ln(x1)+x1x2−sin(x2)f(x1,x2)=ln(x1)+x1x2−sin(x2)

我们可以将其转化为如下计算图：



转化成如上DAG（有向无环图）结构之后，我们可以很容易分步计算函数的值，并求取它每一步的导数值：

Forward Evaluation Trace			Forward Derivative Trace		
v_{-1}	$= x_1$	$= 2$	$\dot{v}_{-1}$	$= \dot{x}_1$	$= 1$
v_0	$= x_2$	$= 5$	$\dot{v}_0$	$= \dot{x}_2$	$= 0$
v_1	$= \ln v_{-1}$	$= \ln 2$	$\dot{v}_1$	$= \dot{v}_{-1}/v_{-1}$	$= 1/2$
v_2	$= v_{-1} \times v_0$	$= 2 \times 5$	$\dot{v}_2$	$= \dot{v}_{-1} \times v_0 + \dot{v}_0 \times v_{-1}$	$= 1 \times 5 +$
v_3	$= \sin v_0$	$= \sin 5$	$\dot{v}_3$	$= \dot{v}_0 \times \cos v_0$	$= 0 \times c$
v_4	$= v_1 + v_2$	$= 0.693 + 10$	$\dot{v}_4$	$= \dot{v}_1 + \dot{v}_2$	$= 0.5 +$
v_5	$= v_4 - v_3$	$= 10.693 + 0.959$	$\dot{v}_5$	$= \dot{v}_4 - \dot{v}_3$	$= 5.5 -$
y	$= v_5$	$= 11.652$	$\dot{y}$	$= \dot{v}_5$	$= \mathbf{5.5}$

上表中左半部分是从左往右每个图节点的求值结果，右半部分是每个节点对于x1x1的求导结果，比如v1˙=dvdx1v1˙=dvdx1，注意到每一步的求导都环的求导结果，这样不至于重复计算，因此也不会产生像符号微分法的”expression swell”问题。
自动微分的forward mode非常符合我们高数里面学习的求导过程，只要您对求导法则还有印象，理解forward mode自不在话下。如果函数输入输出

R→RmR→Rm

那么利用forward mode只需计算一次如上表右边过程即可，非常高效。对于输入输出映射为如下的：

Rn→RmRn→Rm

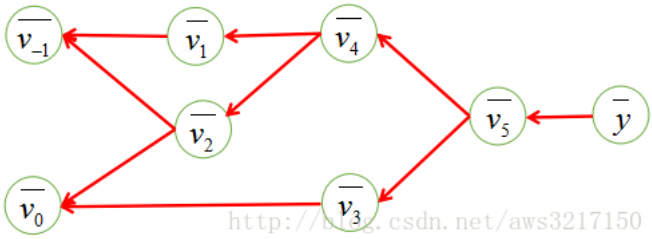
这样一个有nn个输入的函数，求解函数梯度需要nn遍如上计算过程。然而实际算法模型中，比如神经网络，通常输入输出是极其不成比例的，也就是

n>>mn>>m

那么利用forward mode进行自动微分就太低效了，因此便有下面要介绍的reverse mode。

自动微分Reverse Mode

如果您理解神经网络的backprop算法，那么恭喜你，自动微分的backward mode其实就是一种通用的backprop算法，也就是backprop是reverse m的一种特殊形式。从名字可以看出，reverse mode和forward mode是一对相反过程，reverse mode从最终结果开始求导，利用最终输出对每一个节其过程如下红色箭头所示：



其具体计算过程如下表所示：

Forward Evaluation Trace

$$
\begin{array}{lll}
v_{-1} = x_1 & = 2 \\
v_0 = x_2 & = 5 \\
\hline
v_1 = \ln v_{-1} & = \ln 2 \\
v_2 = v_{-1} \times v_0 & = 2 \times 5 \\
v_3 = \sin v_0 & = \sin 5 \\
v_4 = v_1 + v_2 & = 0.693 + 10 \\
v_5 = v_4 - v_3 & = 10.693 + 0.959 \\
\hline
y = v_5 & = 11.652
\end{array}
$$

Reverse Adjoint Trace

$$
\begin{array}{lll}
\bar{x}_1 = \bar{v}_{-1} & & = 5. \\
\bar{x}_2 = \bar{v}_0 & & = 1. \\
\hline
\bar{v}_{-1} = \bar{v}_{-1} + \bar{v}_1 \frac{\partial v_1}{\partial v_{-1}} & = \bar{v}_{-1} + \bar{v}_1 / v_{-1} & = 5.5 \\
\bar{v}_0 = \bar{v}_0 + \bar{v}_2 \frac{\partial v_2}{\partial v_0} & = \bar{v}_0 + \bar{v}_2 \times v_{-1} & = 1.7 \\
\bar{v}_{-1} = \bar{v}_2 \frac{\partial v_2}{\partial v_{-1}} & = \bar{v}_2 \times v_0 & = 5 \\
\bar{v}_0 = \bar{v}_3 \frac{\partial v_3}{\partial v_0} & = \bar{v}_3 \times \cos v_0 & = -0 \\
\bar{v}_2 = \bar{v}_4 \frac{\partial v_4}{\partial v_2} & = \bar{v}_4 \times 1 & = 1 \\
\bar{v}_1 = \bar{v}_4 \frac{\partial v_4}{\partial v_1} & = \bar{v}_4 \times 1 & = 1 \\
\bar{v}_3 = \bar{v}_5 \frac{\partial v_5}{\partial v_3} & = \bar{v}_5 \times (-1) & = -1 \\
\bar{v}_4 = \bar{v}_5 \frac{\partial v_5}{\partial v_4} & = \bar{v}_5 \times 1 & = 1 \\
\hline
\bar{v}_5 = \bar{y} & & = 1
\end{array}
$$

上表左边和之前的forward mode一致，用于求解函数值，右边则是reverse mode的计算过程，注意必须从下网上看，也就是一开始先计算输出yy对于
导数，用v¯¯5v¯5表示dydv5dydv5，这样的记号可以强调我们对当前计算结果进行缓存，以便用于后续计算，而不必重复计算。由链式法则我们可以
每个节点的导数。
比如对于节点v3v3:

dydv3=dydv5dv5dv3dydv3=dydv5dv5dv3

用另一种记法变得到：

dydv3=v5¯¯¯¯dv5dv3dydv3=v5¯dv5dv3

比如对于节点v0v0：

dydv0=dydv2dv2dv0+dydv3dv3dv0dydv0=dydv2dv2dv0+dydv3dv3dv0

如果用另一种记法，便可得出：

dydv0=v¯¯2dv2dv0+v¯¯3dv3dv0dydv0=v¯2dv2dv0+v¯3dv3dv0

和backprop算法一样，我们必须记住前向时当前节点发出的边，然后在后向传播的时候，可以搜集所有受到当前节点影响节点。
如上的计算过程，对于像神经网络这种模型，通常输入是上万到上百万维，而输出损失函数是1维的模型，只需要一遍reverse mode的计算过程，便可
对于各个输入的导数，从而轻松求取梯度用于后续优化更新。

自动微分的实现

这里主要讲解reverse mode的实现方式，forward mode的实现基本和reverse mode一致，但是由于机器学习算法中大部分用reverse mode才可以高
以它是我们理解的重心。代码设计轮廓来源于CSE599G1的作业，通过分析完成作业，可以展示自动微分的简洁性和灵活可用性。

首先自动微分会将问题转化成一种有向无环图，因此我们必须构造基本的图部件，包括节点和边。可以先看看节点是如何实现的

```
class Node(object):
    """Node in a computation graph."""
    def __init__(self):
        """Constructor, new node is indirectly created by Op object __call__ met

            Instance variables
            ------------------
            self.inputs: the list of input nodes.
            self.op: the associated op object,
                e.g. add_op object if this node is created by adding two other n
            self.const_attr: the add or multiply constant,
                e.g. self.const_attr=5 if this node is created by x+5.
            self.name: node name for debugging purposes.
        """
        self.inputs = []
        self.op = None
        self.const_attr = None
        self.name = ""
```

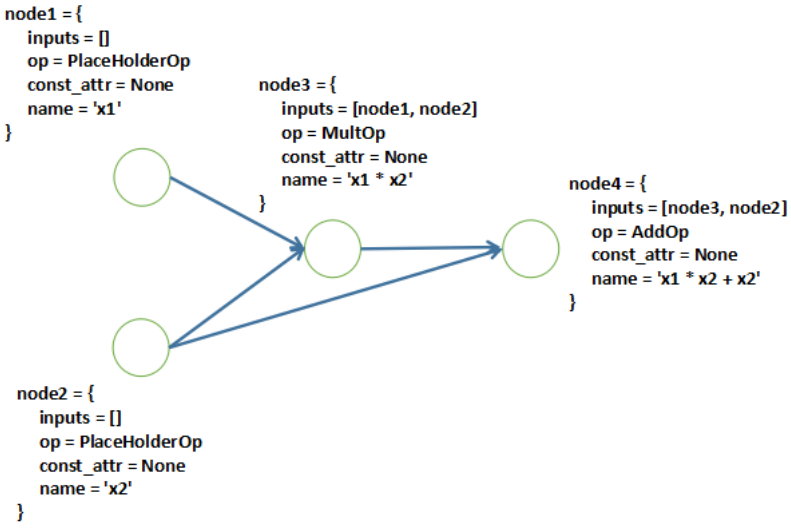

首先节点可以分为三种：

- 常数节点
- 变量节点
- 带操作算子节点

因此Node类中定义了op成员用于存储节点的操作算子，const_attr代表节点的常数值，name是节点的标识，主要用于调试。
对于边的实现则简单的多，每个节点只要知道本身的输入节点即可，因此用inputs来描述节点的关系。
有了如上的定义，利用操作符重载，我们可以很简单构造一个计算图，举一个简单的例子：

f(x1,x2)=x1x2+x2f(x1,x2)=x1x2+x2

对于如上函数，只要重载加法和乘法操作符，我们可以轻松得到如下计算图：



操作算子是自动微分最重要的组成部分，接下来我们重点介绍，先上代码：

```
class Op(object):

    def __call__(self):
        """ 操作符被调用时 会产生一个新节点，然后将本身绑定到节点中"""
        new_node = Node()
        new_node.op = self
        return new_node

    def compute(self, node, input_vals):
        """ 通过提供的输入input_vals,计算当前节点node的输出值 """
        assert False, "Implemented in subclass"

    def gradient(self, node, output_grad):
        """ 通过子节点返回的梯度值:output_grad，结合当前node的操作算子，返回inputs相应的梯度列表"""
        assert False, "Implemented in subclass"
```


从定义可以看出，所有实际计算都落在各个操作算子中，上面代码应该抽象一些，我们来举一个乘法算子的例子加以说明：

```
class MulOp(Op):
    def __call__(self, node_A, node_B):
        """ 乘法操作符 因为它是二元操作符，必须接受两个节点
            分别是node_A， node_B，然后保存在新生成的节点
            对应的inputs中
        """
        new_node = Op.__call__(self)
        new_node.inputs = [node_A, node_B]
        new_node.name = "(%s*%s)" % (node_A.name, node_B.name)
        return new_node

    def compute(self, node, input_vals):
        """ 乘法算子计算值的时候，实际提供值 input_vals必须
            有两个值，然后不出意外，直接相乘返回即可
        """
        assert len(input_vals) == 2
        return input_vals[0] * input_vals[1]

    def gradient(self, node, output_grad):
        """ 计算梯度， 乘法算子所在的节点node的inputs必然是两个，
            那么返回两个输入的实际计算梯度
        """
        return [node.inputs[1] * output_grad, node.inputs[0] * output_grad]
```

我们重点讲解一下gradient方法，它接收两个参数，一个是node，也就是当前要计算的节点，而output_grad则是后面节点传来的，我们来看看它到底

对于如下例子：

$y=f(x1*x2)y=f(x1*x2)$

那么要求yy关于x1x1的导数，那么根据链式法则可得：

$\partial y\partial x1=\partial y\partial f\partial f\partial x1=\partial y\partial x1x2\partial x1x2\partial x1=output\_grad*x2\partial y\partial x1=\partial y\partial f\partial f\partial x1=\partial y\partial x1x2\partial x1x2\partial x1=output\_grad*x2$

则output_grad就是上面的$\partial y\partial f\partial y\partial f$，计算yy对于x2x2类似。因此在程序中我们会返回如下：

```
return [node.inputs[1] * output_grad, node.inputs[0] * output_grad]
```

- 1

再来介绍一个特殊的op——PlaceHolderOp，它的作用就如同名字，起到占位符的作用，也就是自动微分中的变量，它不会参与实际计算，只等待用

际值，因此他的实现如下：

```python
class PlaceholderOp(Op):
    def __call__(self):
        """Creates a variable node."""
        new_node = Op.__call__(self)
        return new_node

    def compute(self, node, input_vals):
        """No compute function since node value is fed directly in Executor."""
        assert False, "placeholder values provided by feed_dict"

    def gradient(self, node, output_grad):
        """No gradient function since node has no inputs."""
        return None
```


了解了节点和操作算子的定义，接下来我们考虑如何协调执行运算。首先是如何计算函数值，对于一幅计算图，由于节点与节点之间的计算有一定的依
必须先计算node1之后才可以计算node2，那么如何能正确处理好计算关系呢？一个简单的方式是对图节点进行拓扑排序，这样可以保证需要先计算的
计算。这部分代码由Executor掌控：

```python
class Executor:
    def __init__(self, eval_node_list):
        # 需要运算的节点列表
        self.eval_node_list = eval_node_list

    def run(self, feed_dict):
        node_to_val_map = dict(feed_dict)
        topo_order = find_topo_sort(self.eval_node_list)  # 需要计算的节点先进行拓扑排序
        for node in topo_order:  # 遍历图，计算非变量节点的值
            if isinstance(node.op, PlaceholderOp):  # 如果是普通变量，不用计算，实际值由feed_dict提供
                continue
            vals = [node_to_val_map[n] for n in node.inputs]  # 取出将节点输入相应的实际值
            compute_val = node.op.compute(node, vals)  # 根据节点附带的op实施计算
            # 将计算结果存在node_to_val_map中，为了简单性，计算引擎假设所有计算类型都是numpy的ndarray，所以对
            # 计算结果为标量值的，需要包装成ndarray
            node_to_val_map[node] = compute_val if isinstance(compute_val, np.ndarray) else np.array(co
        # 将结果返回给用户
        node_val_results = [node_to_val_map[node] for node in self.eval_node_list]
        return node_val_results
```


Executor是实际计算图的引擎，用户提供需要计算的图和实际输入，Executor计算相应的值和梯度。

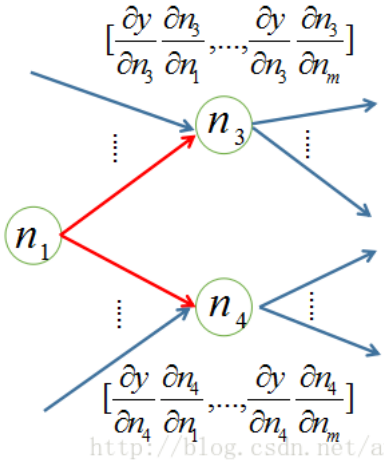
如何从计算图中计算函数的值，上面我们已经介绍了，接下来是如何自动计算梯度。reverse mode的自动微分，要求从输出到输入节点，按照先后依
个节点求取输出对于当前节点的梯度，那么和我们上面介绍的刚好相反，为了得到正确计算节点顺序，我们可以将图节点的拓扑排序倒序即可。代码也
所示：

```python
def gradients(output_node, node_list):
    """ output_node 输出节点， node_list是需要求取输出对于当前节点梯度的列表
    """
    # 该字典用于储存当前节点发射出边的节点列表对应的关于输出节点导数
    node_to_output_grads_list = {output_node: [oneslike_op(output_node)]}
    # 该字典用于储存输出节点对当前节点的导数
    node_to_output_grad = {}
    # 倒序的拓扑排序
    reverse_topo_order = reversed(find_topo_sort([output_node]))

    for node in reverse_topo_order:
        # 根据链式法则，当前节点输出边对应节点的导数相加即是当前节点的导数
        grad = sum_node_list(node_to_output_grads_list[node])
        node_to_output_grad[node] = grad
        # 根据输出关于当前节点的梯度，计算输出关于当前节点输入的梯度
        grads = node.op.gradient(node, grad)
        for i in range(len(node.inputs)):  # 遍历节点的输入
            ch = node.inputs[i]
            grads_list = node_to_output_grads_list.get(ch, [])
            grads_list.append(grads[i])  # 存储对应发射边的梯度
            node_to_output_grads_list[ch] = grads_list

    grad_node_list = [node_to_output_grad[node] for node in node_list]
    return grad_node_list
```


这里先介绍一个新的算子——oneslike_op。他是一个和numpy自带的oneslike函数一样的算子，作用是构造reverse梯度图的起点，因为最终输出关于
就是一个和输出shape一样的全1数组，引入oneslike_op可以使得真实计算得以延后，因此gradients方法最终返回的不是真实的梯度，而是梯度计算图
复用Executor，计算实际的梯度值。
紧接着是根据输出节点，获得倒序的拓扑排序序列，然后遍历序列，构造实际的梯度计算图。我们重点来介绍node_to_output_grad和node_to_output
这两个字典的意义。
先关注node_to_output_grads_list，他key是节点，value是一个梯度列表，代表什么含义呢？先看如下部分计算图：



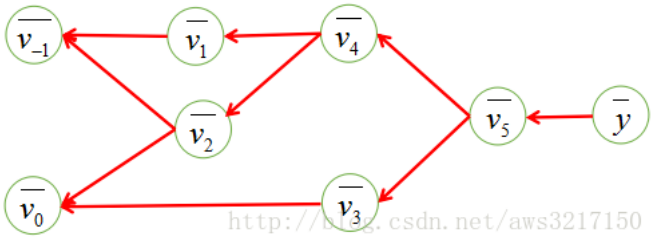
此时我们要计算输出yy关于节点n1n1的导数，那么我们观察到他的发射边连接的节点有n3,n4n3,n4，而对应n3,n4n3,n4节点调用相应op的gradient
输出yy关于各个输入节点的导数。此时为了准确计算输出yy关于节点n1n1的导数，我们需要将其发射边关联节点的计算梯度搜集起来，比如上面的例
搜集：

node_to_output_grads_list={n1:[∂y∂n3∂n3∂n1,∂y∂n4∂n4∂n1]}node_to_output_grads_list={n1:[∂y∂n3∂n3∂n1,∂y∂n4∂n4∂n1]}

一旦搜集好对应输出边节点关于当前节点导数，那么当前节点的导数便可以由链式法则计算得出，也就是：

∂y∂n1=∂y∂n3∂n3∂n1+∂y∂n4∂n4∂n1∂y∂n1=∂y∂n3∂n3∂n1+∂y∂n4∂n4∂n1

因此node_to_output_grad字典存储的就是节点对应的输出关于节点的导数。经过gradients函数执行后，会返回需要求取输出关于某节点的梯度计算



而对于Executor而言，它并不知道此时的图是否被反转，它只关注用户实际输入，还有计算相应的值而已。

自动梯度的应用

有了上面的大篇幅介绍，我们其实已经实现了一个简单的自动微分引擎了，接下来看如何使用：

```
def test_mul_two_vars():
    x2 = ad.Variable(name="x2")
    x3 = ad.Variable(name="x3")
    y = x2 * x3

    grad_x2, grad_x3 = ad.gradients(y, [x2, x3])

    executor = ad.Executor([y, grad_x2, grad_x3])
    x2_val = 2 * np.ones(3)
    x3_val = 3 * np.ones(3)
    y_val, grad_x2_val, grad_x3_val = executor.run(feed_dict={x2: x2_val, x3: x3_val})

    assert isinstance(y, ad.Node)
    assert np.array_equal(y_val, x2_val * x3_val)
    assert np.array_equal(grad_x2_val, x3_val)
    assert np.array_equal(grad_x3_val, x2_val)
```


使用相当简单，我们像编写普通程序一样，对变量进行各种操作，只要提供要求导数的变量，还有提供实际输入，引擎可以正确给出相应的梯度值。下面给出一个根据自动微分训练Logistic Regression的例子：

```
import autodiff as ad
import numpy as np


def logistic_prob(_w):
    def wrapper(_x):
        return 1 / (1 + np.exp(-np.sum(_x * _w)))
    return wrapper


def test_accuracy(_w, _X, _Y):
    prob = logistic_prob(_w)
    correct = 0
    total = len(_Y)
    for i in range(len(_Y)):
        x = _X[i]
        y = _Y[i]
        p = prob(x)
        if p >= 0.5 and y == 1.0:
            correct += 1
        elif p < 0.5 and y == 0.0:
            correct += 1
    print("总数：%d, 预测正确：%d" % (total, correct))


def plot(N, X_val, Y_val, w, with_boundary=False):
    import matplotlib.pyplot as plt
    for i in range(N):
        __x = X_val[i]
        if Y_val[i] == 1:
            plt.plot(__x[1], __x[2], marker='x')
        else:
```

```
            plt.plot(__x[1], __x[2], marker='o')
    if with_boundary:
        min_x1 = min(X_val[:, 1])
        max_x1 = max(X_val[:, 1])
        min_x2 = float(-w[0] - w[1] * min_x1) / w[2]
        max_x2 = float(-w[0] - w[1] * max_x1) / w[2]
        plt.plot([min_x1, max_x1], [min_x2, max_x2], '-r')

    plt.show()


def gen_2d_data(n):
    x_data = np.random.random([n, 2])
    y_data = np.ones(n)
    for i in range(n):
        d = x_data[i]
        if d[0] + d[1] < 1:
            y_data[i] = 0
    x_data_with_bias = np.ones([n, 3])
    x_data_with_bias[:, 1:] = x_data
    return x_data_with_bias, y_data


def auto_diff_lr():
    x = ad.Variable(name='x')
    w = ad.Variable(name='w')
    y = ad.Variable(name='y')

    # 注意，以下实现某些情况会有很大的数值误差，
    # 所以一般真实系统实现会提供高阶算子，从而减少数值误差

    h = 1 / (1 + ad.exp(-ad.reduce_sum(w * x)))
    L = y * ad.log(h) + (1 - y) * ad.log(1 - h)
    w_grad, = ad.gradients(L, [w])
    executor = ad.Executor([L, w_grad])

    N = 100
    X_val, Y_val = gen_2d_data(N)
    w_val = np.ones(3)

    plot(N, X_val, Y_val, w_val)
    executor = ad.Executor([L, w_grad])
    test_accuracy(w_val, X_val, Y_val)
    alpha = 0.01
    max_iters = 300
    for iteration in range(max_iters):
        acc_L_val = 0
        for i in range(N):
            x_val = X_val[i]
            y_val = np.array(Y_val[i])
            L_val, w_grad_val = executor.run(feed_dict={w: w_val, x: x_val, y: y_val})
            w_val += alpha * w_grad_val
            acc_L_val += L_val
        print("iter = %d, likelihood = %s, w = %s" % (iteration, acc_L_val, w_val))
    test_accuracy(w_val, X_val, Y_val)
    plot(N, X_val, Y_val, w_val, True)


if __name__ == '__main__':
    auto_diff_lr()
```

- 转载：https://blog.csdn.net/aws3217150/article/details/70214422

自动求导机制 阅读数 7206
本说明将概述Autograd如何工作并记录操作。了解这些并不是绝对必要的，但我们建议您熟悉它，因... 博文 来自： 人工智能

tensorflow求导和梯度计算 阅读数 767
1.函数求一阶导importtensorflowastftf.enable_eager_execution()tfe=tf.contrib.eagerfrommathim... 博文 来自： 大雄没有叮当...

tensorFlow使用的求导方法原理 阅读数 1130
原理：TensorFlow使用的求导方法称为自动微分（AutomaticDifferentiation），它既不是符号求导也... 博文 来自： H_hei的博客

TensorFlow可微编程实践1---自动微分简介 阅读数 1227
在程序中求导数和微分一般有四种方式：1.手动求微分：采用纯人工方式，与计算机无关，这种我们不... 博文 来自： Yt7589的专栏

TensorFlow - 求导 阅读数 1763
函数flyfish定义域是关于输入的上域是关于输出的值域实际上是上域的一个子集上域是可能输出的集合... 博文 来自： flyfish的专栏

【Autograd】深入理解BP与自动求导 阅读数 2725
“所有数值计算归根结底是一系列有限的可微算子的组合”——《Anintroductiontoautomaticdiffere... 博文 来自： CD's Coding

TensorFlow自动求导原理 阅读数 2381
原理：TensorFlow使用的求导方法称为自动微分（AutomaticDifferentiation），它既不是符号求导也... 博文 来自： qq_25094489...

李理：自动梯度求解——使用自动求导实现多层神经网络 阅读数 1408
本系列文章面向深度学习研究者，希望通过ImageCaptionGeneration，一个有意思的具体任务，深入... 博文 来自： qunnie_yi的博客

自动微分(Automatic Differentiation)简介 - CarlXie - CSDN博客
现代深度学习系统中(比如MXNet, TensorFlow等)都用到了一种技术——自动微分。...自动微分法(Automatic Differentiation)为了讲明白什么是自动微分,我们有必要了解

TensorFlow可微编程实践1---自动微分简介 - Yt7589的专..._CSDN博客
这里需要特别注意,TensorFlow Eager Execution API不能...这种方法的核心是通过人工求出函数的微分的解析式,然后...C++自动微分(Automatic differentiation)原理1 - ..

tensorflow 二阶导数计算 阅读数 2030
http://stackoverflow.com/questions/38200982/how-to-compute-all-second-derivatives-only-t... 博文 来自： go deep

机器学习之——自动求导 阅读数 1831
作者：叶虎小编：张欢随机梯度下降法（SGD）是训练深度学习模型最常用的优化方法。在前期文章中... 博文 来自： 燕哥带你学算法

转载-自动微分(Automatic Differentiation)入门教程 阅读数 124
转载自https://blog.csdn.net/aws3217150/article/details/70214422现代深度学习系统中（比如MX... 博文 来自： hyk_1996的博客

自动微分(Automatic Differentiation)简介 阅读数 1.2万
现代深度学习系统中（比如MXNet，TensorFlow等）都用到了一种技术——自动微分。在此之前，机... 博文 来自： CarlXie

Computational Graph（计算图） 阅读数 490
李宏毅深度学习笔记https://www.bilibili.com/video/av9770302是什么一种描述方程的“语言”既然... 博文 来自： Zhouxnli 的博客

白发千万不要染，饭后一件事，想要多黑就多黑

0

TensorFlow可微编程实践2---自动微分符号体系 - Yt7589..._CSDN博客

自动微分采用一套与常规机器学习和深度学习不同的符号体系,我们只有熟悉了这个符号体系,才能比较轻松的看懂...

自动微分方法简介 - 能找到答案的,只有自己 - CSDN博客

5. 反向自动微分 反向自动微分正是TensorFlow采用的自动微分方法。该方法共分为两个阶段完成: 依照forward direction (输入->输出,叶节点->) 计算图中每个...

TensorFlow中的Eager Execution和自动微分 阅读数 1756

在传统的TensorFlow开发中，我们需要首先通过变量和Placeholder来定义一个计算图，然后启动一个... 博文 来自： Yt7589的专栏

前向传播算法(Forward propagation)与反向传播算法(Back propagation) 阅读数 162

虽然学深度学习有一段时间了，但是对于一些算法的具体实现还是模糊不清，用了很久也不是很了解。... 博文 来自： LegenDavid's ...

《 60分钟快速入门学习pytorch笔记二》Autograd: 自动求导(automatic different... 阅读数 62

PyTorch中所有神经网络的核心是autograd包.我们首先简单介绍一下这个包,然后训练我们的第一个神... 博文 来自： weixin_40293...

详解反向传播算法(上) 阅读数 417

原文出处：知乎：详解反向传播算法(上)详解反向传播算法(上)晓雷1年前目录：1用计算图来解释几种... 博文 来自： counter_king...

图解微积分：反向传播 阅读数 804

原文链接戳此处IntroductionBackpropagationisthekeyalgorithmthatmakestrainingdeepmodelsc... 博文 来自： 亡城、的专栏

计算图上的微积分：反向传播算法 阅读数 1757

引言Backpropagation(BP)是使得训练深度模型在计算上可行的关键算法。对现代神经网络，这个算法... 博文 来自： Purplenigma...

TensorFlow核心知识 阅读数 1113

TensorFlow概要Google第一代分布式机器学习框架DistBelief 1 ，在内部大规模使用后并没有选择开... 博文 来自： lbg198808的...

Tensorflow一些常用基本概念与函数（4） 阅读数 8.3万

摘要：本系列主要对tf的一些常用概念与方法进行描述。本文主要针对tensorflow的模型训练Training... 博文 来自： lenbow的博客

tensorflow 反向传播求导 阅读数 543

X=tf.constant([-1,-2],dtype=tf.float32)w=tf.Variable([2.,3.])truth=[3.,3.]Y=w*X#cost=tf.reduce_s... 博文 来自： shuijiaobuzhu...

Tensorflow & Caffe 对比 阅读数 8450

初学TF,参考了师兄的blog:http://blog.csdn.net/cham_3/article/details/71374444 TensorFlow的关... 博文 来自： yang_502的博客

自动求导机制 - 人工智能 - CSDN博客

它比任何其他自动求导的设置更有效——它将使用绝对最小的内存来评估模型。...自动微分(Automatic Differentiation)简介——tensorflow核心原理 08-14 48 ...

机器学习之——自动求导 - 燕哥带你学算法 - CSDN博客

自动微分(Automatic Differentiation)简介——tensorflow核心原理 08-14 56 现代深度学习系统中(比如MX...

【DeepLearning】深度学习第一课：使用autograd来自动求导 阅读数 1491

使用autograd来自动求导在机器学习中，我们通常使用梯度下降（gradientdescent）来更新模型参数... 博文 来自： zsWang9的博客

Evaluating Derivatives Principles and Techniques of AD-1 08-02

Evaluating Derivatives Principles and Techniques of Algorithmic Differentiation 自动微分算法的经典著作。作者是著名的AD软件包ADOL... 下载

矩阵微分（matrix derivatives） 阅读数 4879

关于矩阵求导，得到的导数则是矩阵形式；关于矢量求导，得到的导数则是矢量形式；关于标量求导，... 博文 来自： Zhang's Wikip...

tensorflow学习笔记：偏微分方程 阅读数 508

#-*-coding:utf-8-*-#偏微分方程importtensorflowastfimportnumpyasnpimportPIL.Imagefromioi... 博文 来自： dream_hunter

TensorFlow官方文档偏微分方程 阅读数 134

偏微分方程TensorFlow 不仅仅是用来机器学习，它更可以用来模拟仿真。在这里，我们将通过模拟仿... 博文 来自： macair123的...

Tensorflow教程-偏微分方程 阅读数 661

偏微分方程TensorFlow 不仅仅是用来机器学习，它更可以用来模拟仿真。在这里，我们将通过模拟仿... 博文 来自： qq_31215157...

TensorFlow 之 Automatic differentiation and gradien..._CSDN博客

所有相关操作都会记录下来用于automatic differentiation....TensorFlow可微编程实践1---自动微分简介 06-19 ...autograd包是PyTorch所有神经网络的核心,为Tensors上

强化学习精要：核心算法与TensorFlow实现 05-20

该代码是《强化学习精要：核心算法与TensorFlow实现》这本书的几个核心代码，需要代码知识的可以配合书本学习。 下载

组成 TensorFlow 核心的六篇论文 阅读数 1204

作者：chen_h微信号&QQ：862251340微信公众号：coderpai简书地址：http://www.jianshu.com/... 博文 来自： CoderPai的博客

强化学习-冯超 02-05

强化学习精要 核心算法与TensorFlow实现,原理细致，代码实现简洁 下载

springboot自动配置的核心原理 阅读数 1223

springboot的最重要特点除了帮助我们管理依赖外，还有自动配置，springboot把一个个技术模块封... 博文 来自： 码农的世界你...

TensorFlow可微分编程实践3---交叉熵与代价函数微分 阅读数 1077

在上篇博文中，我们讲述怎样处理第l−1l−1l-1层到第lll层的前向传输和反向求导，我们还没有讲述关于... 博文 来自： Yt7589的专栏

IT新装备 跑赢新起点

IT新装备 跑赢新起点

TensorFlow：偏微分方程 阅读数 393

TensorFlow：偏微分方程 博文 来自： 《好好先生》...

自动微分方法简介 阅读数 370

假设我们定义了一个方程：f(x,y)=x2y+y+2f(x,y)=x2y+y+2f(x,y)=x^2y+y+2，我们需要对xxx和yyy... 博文 来自： 能找到答案的...

图像缩放双线性插值算法 阅读数 11700

插值算法对于缩放比例较小的情况是完全可以接受的，令人信服的。一般的，缩小0.5倍以上或放大3.0...

【小程序】微信小程序开发实践 阅读数 92652

帐号相关流程注册范围 企业 政府 媒体 其他组织换句话讲就是不让个人开发者注册。 :)填写企业信息不... 博文 来自： 小雨同学的技...

反射获取枚举上的注解 阅读数 5939

关于反射获取枚举上的信息，javaSE的jdk相关API里面并没有提供方法，直接获取也只能获取到enum... 博文 来自： 逆风奔跑。。...

虚拟机Linux如何使用笔记本电脑的前置摄像头 阅读数 9296

一、Windows设置1.点击开始->运行，在对话框中输入”services.msc”，回车，打开windows服务... 博文 来自： fendoubasaon...

手把手教你整合最优雅SSM框架：SpringMVC + Spring + MyBatis 阅读数 163506

小疯手把手带你整合SpringMVC+Spring+MyBatis三大框架，俗称SSM，用它完全代替传统的SSH框... 博文 来自： 小疯的代码健...

POJ 2104 K-th Number【整体二分 + 树状数组】 阅读数 2088

本来只是想学一下CDQ，还是先把整体二分搞懂一点。这题窝几个月前分别用划分树，树套树，主席树... 博文 来自： Tuesday

关于SpringBoot bean无法注入的问题（与文件包位置有关） 阅读数 92093

问题场景描述整个项目通过Maven构建，大致结构如下： 核心Spring框架一个module spring-boot-b... 博文 来自： 开发随笔