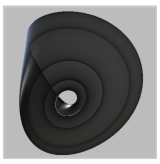


原 自动求导的二三事

标签： 神经网络 算法 博客 递归

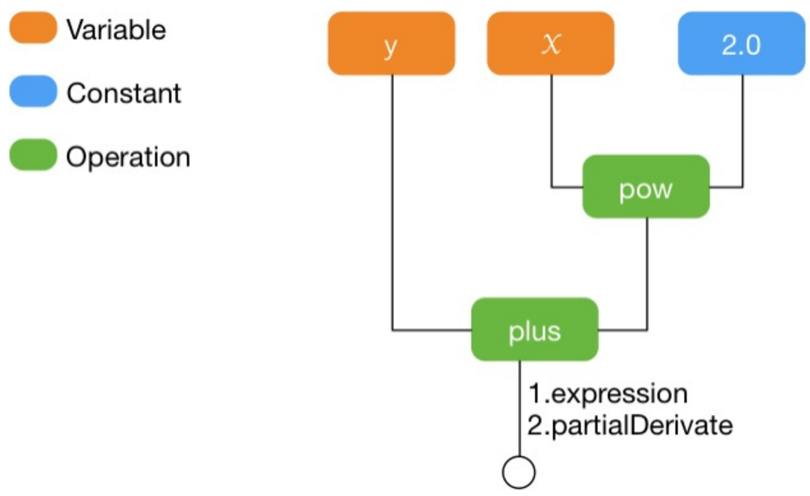
2017-03-17 16:33 2023人阅读 评论(0)

分类： 代码（13） 机器学习（45）

目录(?) [+]

知乎上看到一个回答，说是自己学习神经网络的时候都是自己对公式求导，现在常见的DL库都可以自动求导了。这个想必实现过神经网络的同学都有体会，因为神经网络的back-propagation**算法**本质上就是求导链式法则的堆叠，所以学习这部分的时候就是推来推去，推导对了，那算法你也就掌握了。

粗粗一想，只要能把所有操作用有向图构建出来，通过递归去实现自动求导似乎很简单，一时兴起写了一些代码，整理成博客记录一下。



[tips]完整代码见这里just.dark的代码库

动手

首先我们需要一个基础类，所有有向图的节点都会有下面两个方法`partialGradient()`是对传入的变量求偏导，返回的同样是一个图。
`expression()`是用于将整个式子打印出来

```
class GBaseClass():
    def __init__(self, name, value, type):
        self.name = name
        self.type = type
        self.value = value
        pass

    def partialGradient(self, partial):
        pass

    def expression(self):
        pass
```

从图1可以看出来，我们主要有三种Class，常量Constant，变量Variable以及算子Operation，最简单的常量：

```
G_STATIC_VARIABLE = {}
class GConstant(GBaseClass):
    def __init__(self, value):
        global G_STATIC_VARIABLE
        try:
            G_STATIC_VARIABLE['counter'] += 1
        except:
            G_STATIC_VARIABLE.setdefault('counter', 0);
        self.value = value
        self.name = "CONSTANT_" + str(G_STATIC_VARIABLE['counter']) 变量的名字以及类型
        self.type = "CONSTANT"

    def partialGradient(self, partial):
        return GConstant(0)

    def expression(self):
        return str(self.value)
```


而常量对一个变量求导

可以看到，我们为常量设置了自增的name，只需要传入value即可定义一个常量。而常量对一个变 高中数学告诉我们结果当然是0，所以我们返回一个新的常量 `GConstant(0)` ,而它的 `expression` 也很简单，就是返回本身的值。

接下来是Variable

```
class GVariable(GBaseClass):
    def __init__(self, name, value=None):
        self.name = name
        self.value = value
        self.type = "VARIABLE"

    def partialGradient(self, partial):
        if partial.name == self.name:
            return GConstant(1)
        return GConstant(0)

    def expression(self):
        return str(self.name)
```

甚至比常量还简单一些，因为是变量，所以它的值可能是不确定的，所以构造的时候默认为None，一个变量它对自身的导数是1，对其它变量是0，所以我们可以看到在 `partialGradient()` 也正是这样操作的。变量本身的 `expression` 也就是它本身的标识符。

紧接着就是大头了, `Operation` ,比如图1所示，我们将一个变量和一个常量通过二元算子 `plus` 连接起来，本身它就构成了一个函数式了。

```
class GOperation(GBaseClass):
def __init__(self, a, b, operType):
    self.operatorType = operType;
    self.left = a
    self.right = b
```

几乎所有计算都是二元的，所以我们可以传入两个算子，operType是一个字符串，指示用什么计算项连接两个算子。对于特殊的比如 `exp` 等单元计算项，可以默认传入的右算子为None。
接下来我们需要求偏导和写expression了。

```
def partialGradient(self, partial):
        # partial must be a variable
    if partial.type != "VARIABLE":
        return None
    if self.operatorType == "plus"
        return GOperationWrapper(self.left.partialGradient(partial), self.right.partialGradient(partia

def expression(self):
    if self.operatorType == "plus":
        return self.left.expression() + "+" + self.right.expression()
```

比如我们先看看最简单的「加法」，`GOperationWrapper` 是对GOperation的外层封装，后面一些优化可以在里面完成，现在你可以直接认为：

```
def GOperationWrapper(left, right, operType):
    return GOperation(left, right, operType)
```

求导

我们来看看 `partialGradient` 做了什么，回忆一下高中数学，对一个加式的求导，就是左右两边算子分别求导再相加，所以我们在 `partialGradient` 就翻译了这个操作而已，复杂的事情交给递归去解决，`exp` 同理，更加简单。

当然此时我们只有 `plus` 这一个计算项，肯定无法处理复杂的情况，所以我们添加更多的计算项就可


```
def partialGradient(self, partial):
    # partial must be a variable
    if partial.type != "VARIABLE":
        return None
    if self.operatorType == "plus" or self.operatorType == "minus":
        return GOperationWrapper(self.left.partialGradient(partial), self.right.partialGradient(partia
                                 self.operatorType)

    if self.operatorType == "multiple":
        part1 = GOperationWrapper(self.left.partialGradient(partial), self.right, "multiple")
        part2 = GOperationWrapper(self.left, self.right.partialGradient(partial), "multiple")
        return GOperationWrapper(part1, part2, "plus")

    if self.operatorType == "division":
        part1 = GOperationWrapper(self.left.partialGradient(partial), self.right, "multiple")
        part2 = GOperationWrapper(self.left, self.right.partialGradient(partial), "multiple")
        part3 = GOperationWrapper(part1, part2, "minus")
        part4 = GOperationWrapper(self.right, GConstant(2), 'pow')
        part5 = GOperationWrapper(part3, part4, 'division')
        return part5

    # pow should be g^a,a is a constant.
    if self.operatorType == "pow":
        c = GConstant(self.right.value - 1)
        part2 = GOperationWrapper(self.left, c, "pow")
        part3 = GOperationWrapper(self.right, part2, "multiple")
        return GOperationWrapper(self.left.partialGradient(partial), part3, "multiple")

    if self.operatorType == "exp":
        return GOperationWrapper(self.left.partialGradient(partial), self, "multiple")

    if self.operatorType == "ln":
        part1 = GOperationWrapper(GConstant(1),self.left,"division")
        rst = GOperationWrapper(self.left.partialGradient(partial), part1, "multiple")
        return rst

    return None
```

咱一个一个看：`minus` 和 `plus` 类似，你也可以把高中课本的求导公式翻出来一个一个对照：

$$\begin{gathered} y=u+v, y'=u'+v' \\ y=u-v, y'=u'-v' \\ y=u*v, y'=u'v+uv' \\ y=u/v, y'=(u'v-uv')/v^2 \\ y=x^n, y'=nx^{n-1} \\ y=e^x, y'=e^x \\ y=ln(x), y'=1/x. \end{gathered}$$

当然还有最最重要的链式法则

$$y=f[g(x)], y'=f'[g(x)]\bullet g'(x)$$

比如就拿稍显复杂的 `division` 计算项来说：

```
    if self.operatorType == "division":
        part1 = GOperationWrapper(self.left.partialGradient(partial), self.right, "multiple")
        part2 = GOperationWrapper(self.left, self.right.partialGradient(partial), "multiple")
        part3 = GOperationWrapper(part1, part2, "minus")
        part4 = GOperationWrapper(self.right, GConstant(2), 'pow')
        part5 = GOperationWrapper(part3, part4, 'division')
        return part5
```

对应的求导公式是

$$y=\frac{u}{v},y'=\frac{u'v-uv'}{v^2}$$

代码里的`part1`就是$u'v$,`part2`是uv',`part3`是$u'v-uv'$，`part4`是v^2,最后的结果`part5`则是除法计算项将$u'v-uv'$和v^2连接起来。**代码做的不过是如实翻译公式**而已。

另一个很重要的就是链式法则：

$$y=f[g(x)],y'=f'[g(x)]\bullet g'(x)$$

比如我们在对`power`计算项求导的时候，（这里限制了指数位置必须是常数），除了翻译公式$y=x^n,y'=nx^{n-1}$外，**还要考虑底数部分可能是一个函数，所以还需要乘上这个函数的偏导**：

```
    if self.operatorType == "pow":
        c = GConstant(self.right.value - 1)
        part2 = GOperationWrapper(self.left, c, "pow")
        part3 = GOperationWrapper(self.right, part2, "multiple")
        return GOperationWrapper(self.left.partialGradient(partial), part3, "multiple")
```

至此我们已经完成了主要的部分，我们可以在这些基础计算项的基础上**封装出更复杂的计算逻辑**，比如神经网络中常用的Sigmoid

$$sigmoid(x)=\frac{1}{1+e^{-x}}$$

```
def sigmoid(X):
    a = GConstant(1.0)
    b = GOperationWrapper(GConstant(0), X, 'minus')
    c = GOperationWrapper(b, None, 'exp')
    d = GOperationWrapper(a, c, 'plus')
    rst = GOperationWrapper(a, d, 'division')
    return rst
```

你完全不用关系如果对sigmoid求导，因为你只需要对它返回的结果调用`partialGradient()`就可以了，递归会自动去梳理其中的拓扑序，完成导数求解。

验证

我们试着构建一个计算式然后运行一下（完整代码见代码1）：

```
# case 3
X = GVariable("x")
y = GVariable("y")
beta = GVariable("beta")
xb = GOperationWrapper(X, beta, 'multiple')
s_xb = sigmoid(xb)
m = GOperationWrapper(s_xb, y, 'minus')
f = GOperationWrapper(m, GConstant(2), 'pow')

print "F:\n\t", f.expression()
print "F partial gradient of B:\n\t", f.partialGradient(x).expression()
上面我们构造了如下公式
```

$$f=(sigmoid(x*\beta)-y)^2$$

程序输出为：

```
F:
    ((1.0)/(1.0+exp(0-(x)*(beta)))-y)^(2)
F partial gradient of x:
    (((0)*(1.0+exp(0-(x)*(beta)))-(1.0)*(0+(0-(1)*(beta)+(x)*(0))*(exp(0-(x)*(beta)))))/((1.0+exp(0-(x)
```

天啦噜!!(╯' - ')╯︵ ┻━┻，怎么是这么复杂的一堆，如何验证结果是对的呢，你可以把上面的式子拷贝到wolframe alpha上，第一个式子的结果里我们发现wolframe alpha已经自动帮我们对x求了一次导：

Derivative

Derivative: Approximate form Step-by-step soluti

$$\frac{\partial}{\partial x}\left(\left(\frac{1}{1+\exp(0-x\,\beta)}-y\right)^2\right) = -\frac{2\,\beta\, e^{\beta x}\left((y-1)\,e^{\beta x}+y\right)}{\left(e^{\beta x}+1\right)^3}$$

第二个求导结果放进去，发现它在「alternate form」里有一个形态稍加转化就是上面这个求导结
提取一个-2出来）：

Alternate form

$$\frac{\beta\, e^{\beta x}\left((2.-2.\,y)\,e^{\beta x}-2.\,y\right)}{\left(1.+e^{\beta x}\right)^3}$$

所以我们的求导结果是对的。

接下来有个问题，我们打印出来的东西太复杂了，明细有很多地方可以简化，比如 `0*a=0`、`1*a=a` 这样的小学知识就可以帮到我们，可以明显帮我们简化公式，这个时候就到了我们的 `GOperationWrapper` 了，加入一些简单的逻辑：

```
def GOperationWrapper(left, right, operType): 妙!!
    if operType == "multiple":
        if left.type == "CONSTANT" and right.type == "CONSTANT":
            return GConstant(left.value * right.value)

        if left.type == "CONSTANT" and left.value == 1:
            return right

        if left.type == "CONSTANT" and left.value == 0:
            return GConstant(0)

        if right.type == "CONSTANT" and right.value == 1:
            return left

        if right.type == "CONSTANT" and right.value == 0:
            return GConstant(0)

    if operType == "plus":
        if left.type == "CONSTANT" and left.value == 0:
            return right

        if right.type == "CONSTANT" and right.value == 0:
            return left

    if operType == "minus":
        if right.type == "CONSTANT" and right.value == 0:
            return left

    return GOperation(left, right, operType)
```

都是小学课本如实翻译，就可以把结果简化掉,可以看到已经减少了一截了，而且对于计算也有一些优化。完整代码见代码2：

```
F partial gradient of x:
    ((0-(0-beta)*(exp(0-(x)*(beta))))/((1.0+exp(0-(x)*(beta)))^(2)))*((2)*(((1.0)/(1.0+exp(0-(x)*(beta)
```

还能做什么，优化！

接下来我们还能做什么呢？在写一个类似的递归函数传入Variable的值然后计算函数式的结果，这个就不在这写了，大同小异。

我们梳理下刚才调用的逻辑，你会发现对x求导到最底层的时候做了很多**重复计算**，大家回忆一下递归的好处，其中有一个就是「记忆化搜索」，可以大幅提高运行效率。也就是在第一次运行的时候记录下结果，以后再调用的时候就直接返回存好的结果。

所以我们可以在 求导/求表达式 的时候把结果存下来：

比如对expression进行改造：代码见xxx

```
def expression(self):
    if self.expressionRst != None:
        return self.expressionRst

    ....
    ....
    ....

    self.expressionRst = rst
    return rst
```


除此之外还可以做更多的优化，比如在不同地方可能会出现相同的计算式，其实完全可以根据计算式的expression，进一步记忆化，保证每一个式子只在程序里出现一次，**比如我们在过程中多次使用到了 `GConstant(0)` ，其实这个完全只声明并使用一次。**

通过打印 `G_STATIC_VARIABLE` 我们发现程序运行一次创建了13个常量，而对 `GConstant` 进行一层记忆化封装之后：

```
G_CONSTANT_DICT = {}
def GConstantWrapper(value):
    global G_CONSTANT_DICT
    if G_CONSTANT_DICT.has_key(value):
        return G_CONSTANT_DICT[value]

    rst = GConstant(value)
    G_CONSTANT_DICT.setdefault(value,rst)
    return rst
```

最后一共只创建了3个常量，(0)，(1)和(2)，这些东西都可以重复利用，不需要浪费空间和CPU去声明新实例，这也符合函数式编程的思想，这这里推荐大家读一下《SICP》，会有帮助的。**我们甚至可以将这个思路推广到所有出现的计算式**，可以在后续计算和求导的时候节省大量的时间，不过在此就不做实现了。

尾巴

花了一个小时写代码，N个碎片时间写博客，但真心觉得求导链式法则和递归简直就是天作之合，不记录一下于心难忍。当然真实tf和mxnet使用的自动求导肯定还有更多优化的，不过就不深钻下去了，这个状态~味道刚刚好。

上一篇 序列的算法（一·a）马尔可夫模型
下一篇 序列的算法（一·b）隐马尔可夫模型

相关文章推荐

- 最优化方法：范数和规则化regularization
- 自动求导程序的设计与实现（Python)
- 三角函数公式合集——从诱导公式到求导公式
- liferay关于ldap配置的二三事
- 关于乘积求导法则在考研数学中的应用
- SICP学习笔记 2.3.2 实例：符号求导

- 自动求导机制
- pytorch学习笔记（三）：自动求导
- 广州买房二三事
- 二三事。老师说标题长才霸气。长长长...

猜你在找

- 机器学习之概率与统计推断
- 机器学习之数学基础
- 机器学习之凸优化
- 机器学习之矩阵
- 响应式布局全新探索
- 探究Linux的总线、设备、驱动模型
- 深度学习基础与TensorFlow实践
- 深度学习之神经网络原理与实战技巧
- 前端开发在线峰会
- TensorFlow实战进阶：手把手教你做图像识别应用

查看评论

暂无评论

发表评论

用 户 名： hukexin0000

评论内容：

提交

* 以上用户言论只代表其个人观点，不代表CSDN网站的观点或立场