Tiny Machine Learner

Zane.F

1 Tiny Grader

想要从头实现一个神经网络，就得先解决自动微分的问题，其中最简单的办法就是用有限差分做近似。

```
# 手动获取当前能处理的最小的数字
eps = 1
while 1 < 1 + eps:
    eps /= 2
eps *= 2

def d(f, x):
    return (f(x + eps) - f(x)) / eps
```

不过这样做有两个缺点，一是它的精度比较低，二就是效率差。每求一次梯度都要进行两次正向运算。

要想进行真正意义上的自动微分，我们就得保持一个计算式的结构。普通的Python程序是没办法得到有关计算式结构的信息的，你唯一能做的就是用它作正向计算。

为此，我们需要撇弃普通的计算式，从最基本的常数和变量开始，重新构建一个具有结构的表达式。

```
class Const:
    def __init__(self, v: float):
        self.v = v
    def value(self) -> float:
        return self.v
    def grad(self, x) -> float:
        return 0

class Var:
    def __init__(self, v: float):
        self.v = v
    def value(self) -> float:
        return self.v
    def set_value(self, v: float) -> None:
        self.v = v
    def grad(self, x) -> float:
        return 1 if x == self else 0
```

我们首先传入一个数值作为常数和变量的值，这个值可以用value方法获取，区别是变量的值是可以通过set_value变动的，常数和变量的微分很好说，常数的导恒等于0，而变量的微分则要看是不是它自己，如果是，那么就返回1；如果不是，则按常数处理。

我们接着定义一个最基本的运算-加法。

```
class Add:
```

```
    def __init__(self, a, b):
        self.a = a
        self.b = b
    def value(self) -> float:
        return self.a.value() + self.b.value()
    def grad(self, x) -> float:
        return self.a.grad(x) + self.b.grad(x)
```

加法对象的值等于两个加数的和，微分等于两个加数的微分之和，有了这三样东西，我们就可以构建出最基本的有结构计算式，并且能够自动求值和微分。

```
x = Var(1)
y = Var(2)
z = Add(Add(x, Const(2)), y)  # z = (x + 2) + y
z.grad(x)  # z在x=1,y=2处对于x的偏导
x.set_value(2)
z.value()  # z在x=2,y=2处的值
```

我们构建出来的计算式很像是数据结构中的二叉树，常量和变量就像叶子节点：

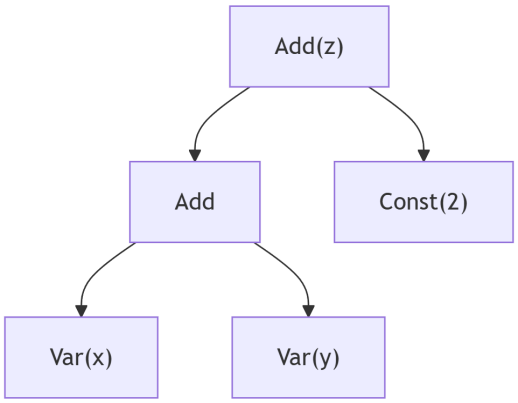

图 1. 树状表达式

仿照着上面的例子，逐步定义出减法、乘法、平方、绝对值的对象，我们就可以得到一个完善的自动微分库，不过这样的程序在实际应用中会遇到大麻烦，在讲述这个问题之前，我们先来看一个例子。

```
def fib1(n):
    if n == 0 or n == 1:
        return n
    else:
        return fib1(n - 1) + fib1(n - 2)

def fib2(n):
    def helper(n1, n2, n):
        if n == 0:
            return n1
        else:
            return helper(n2, n1 + n2, n - 1)
    return helper(0, 1, n)
```

这是两段经典的斐波那契数列的递归程序，`fib1`和`fib2`产出相同的结果，但是`fib1`的效率要比`fib2`差很多，首先`fib1`做了大量的重复计算：

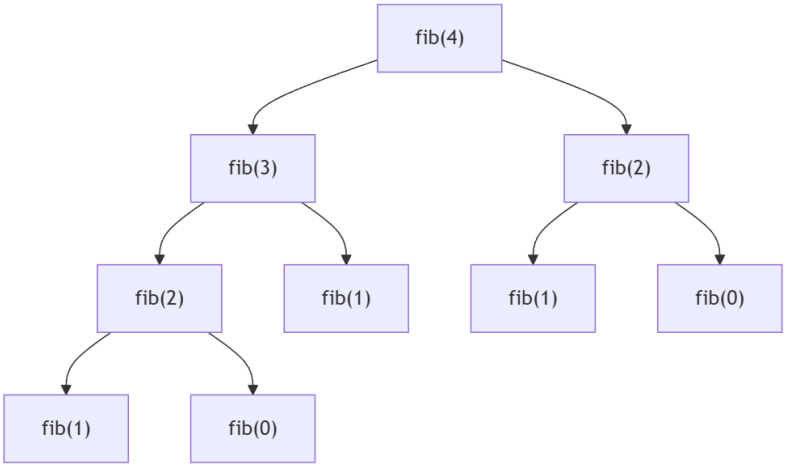

图 2. fib1函数的计算形状

不难看出，在计算`fib(4)`的值时，`fib(2)`被计算了两次，像这样的重复计算在n增大时会不断增多，拖累整个程序的效率。

我们上面定义的结构化的计算式也是一样，我们在更新神经网络的权重时，每一次计算梯度都要对整个神经网络进行一次反向运算来得到误差函数对某一个权重的梯度，尽管一些中间值已经被计算过了，简单来说就像这样：

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial g}\frac{\partial g}{\partial x}$$
$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial g}\frac{\partial g}{\partial y}$$

尽管$\frac{\partial f}{\partial g}$在求取$\frac{\partial f}{\partial x}$时已经计算出来，并且值没有发生变化，但是我们在求取$\frac{\partial f}{\partial y}$仍然需要重新计算该值，当我们的神经网络层数和神经元数量增加时，这样的重复计算量是大到可怕的，所以我们必须改造我们的程序，让它像`fib2`那样高效。

首先我们实际上只需要求取损失函数f对于单个权重w的梯度，它的计算是这样子的：

$$\frac{\partial f}{\partial w} = \frac{\partial f}{\partial f_1}\frac{\partial f_1}{\partial f_2}\frac{\partial f_2}{\partial f_3}\cdots\frac{\partial f_n}{\partial w}\frac{\partial w}{\partial w}$$

假设我们已经计算完了$\frac{\partial f}{\partial f_1}\frac{\partial f_1}{\partial f_2}\frac{\partial f_2}{\partial f_3}\cdots\frac{\partial f_n}{\partial w}\frac{\partial w}{\partial w}$的值了，$\frac{\partial w}{\partial w}$毫无疑问是1，因为我们只需要获取$f$对$w$的梯度，对于中间结果我们并不关心，所以我们直接把最终值保存在w的对象里：

```
class Var:
    # ... ...
    def grad(self, computed_value):
        self.g = computed_value
```

需要注意的是，像这样的链条可能不止一个，比方说f是关于f_1和f_2的函数，而f_1和f_2都是关于w的函数，那么就会出现分支的情况：

$$\frac{\partial f}{\partial w} = \frac{\partial f}{\partial f_1}\frac{\partial f_1}{\partial w} + \frac{\partial f}{\partial f_2}\frac{\partial f_2}{\partial w}$$

也就是说不同的computed_value可能会多次进入到同一个权重的grad方法中，为此我们需要把他们累加起来：

```
class Var:
    # ... ...
    def grad(self, computed_value):
        self.g += computed_value
```

我们前面定义的Add对象也需要做改造，其实Add也可以看作是一个函数：$y = w1 + w2$，w1和w2的权重都需要计算出来，而且Add这个函数没有改变两者的梯度，所以我们直接把前面累积的结果交给他们两个：

```
class Add:
    # ... ...
    def grad(self, computed_value):
        self.a.grad(computed_value)
        self.b.grad(computed_value)
```

Sub对象和Add类似，不过它会让b的梯度为负，所以需要做一步处理：

```
class Sub:
    # ... ...
    def grad(self, computed_value):
        self.a.grad(computed_value)
        self.b.grad(-computed_value)
```

定义其他基本运算或者函数的方法类似，你并不需要关心要对谁求微分，你只需要对你的参数求微分并且乘上前面传过来的累积值然后传递给你的参数继续进行微分计算就可以了，下面看一个具体的例子：

```
x = Var(1)
y = Var(2)
z = Add(Add(x, Const(2)), y)  # z = (x + 2) + y
z.grad(1)  # 由于z已经是“最前端”的计算式了，所以累计值记为1
print(x.g, y.g)  # 获取z对x和y的梯度
```

不过目前还有一个问题，当你通过`z.grad()`进行过一次梯度计算后，再次计算梯度时，x和y中的g已经不是从0开始累加的了，这里我们需要进行一个梯度清零的操作：

```
x = Var(1)
y = Var(2)
z = Add(Add(x, Const(2)), y)  # z = (x + 2) + y
z.grad(1)
x.zero_grad()  # 清零梯度
y.zero_grad()
z.grad(1)
```

学过PyTorch的人可能有一种恍然大悟的感觉：

- 为什么在loss.backward()之前要进行optimizer.zero_grad()的操作?

  为了清除上一次计算梯度时留存的结果

- 为什么zero_grad()这项操作要由optimizer来完成?

`optimizer`对象在构造时就传入了神经网络的全部权重的引用，它知道哪些权重留存了结果需要清零。而`loss`对象对这点一无所知，它可能就像我们定义的`Add`对象一样，只知道自己有两个加数，但是连这两个加数到底是变量还是一个新的计算式都不知道。

- 为什么`optimizer`对象不依赖于`loss`对象?

  梯度计算的结果直接保存在对应的权重对象中，`optimizer`直接读取就可以了，毕竟它只需要梯度的值来更新权重。

像这样逐步改造我们的结构化表达式，无论是运算速度还是内存占用都得到了极大的改善，我们的工作也终于迈入了实用的第一步。

Alpha包内是经过改善的框架，Beta包内是我们最前面讲述的存在性能问题的框架。

2 Tiny Learner

这个文件里主要是矩阵有关的函数以及优化算法。

矩阵部分无需多言，一个是创建随机矩阵的函数，另一个是进行矩阵运算的函数。文件里描述的矩阵就是一个二维的Python列表，矩阵乘法也无非是简单的嵌套循环而已。

优化算法里有普通的梯度下降优化器、动量梯度下降优化器、RMS梯度下降优化器和Adam优化器。后面三个和普通的梯度下降算法不同的地方就是参考了以前的梯度值，以此来调节参数更新的速度。

3 Application

在经典的石蕊数据集和sklearn库里自带的一些数据集上测试了一下，没想到真的跑通了。

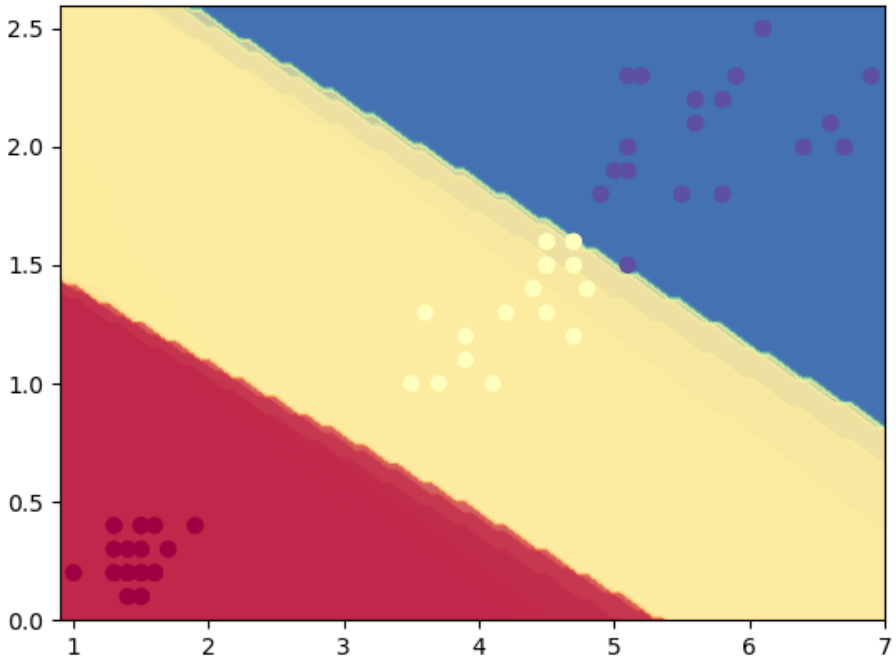

图 3. 石蕊数据集测试结果

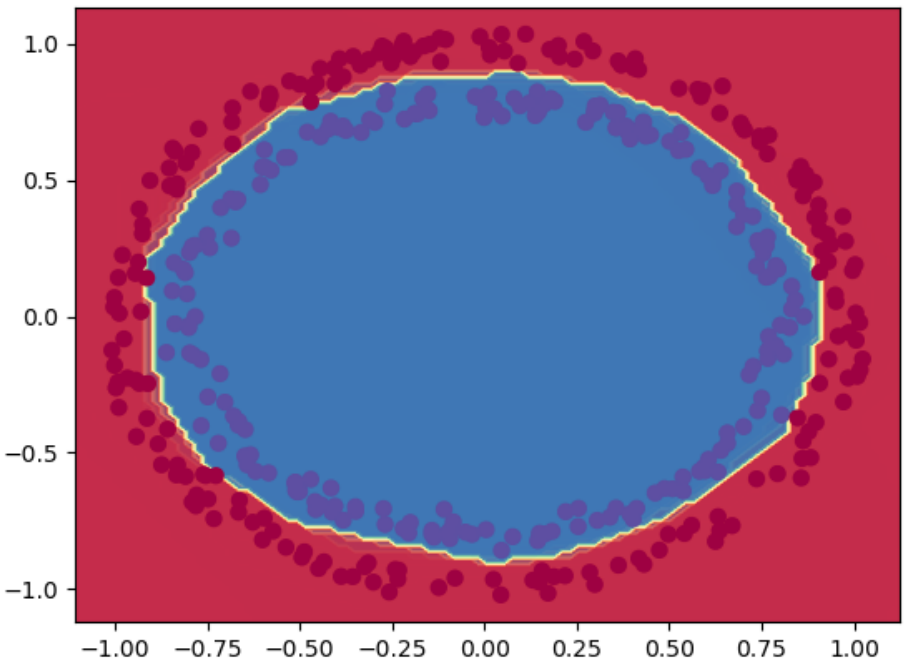

图 4. Circle数据集测试结果

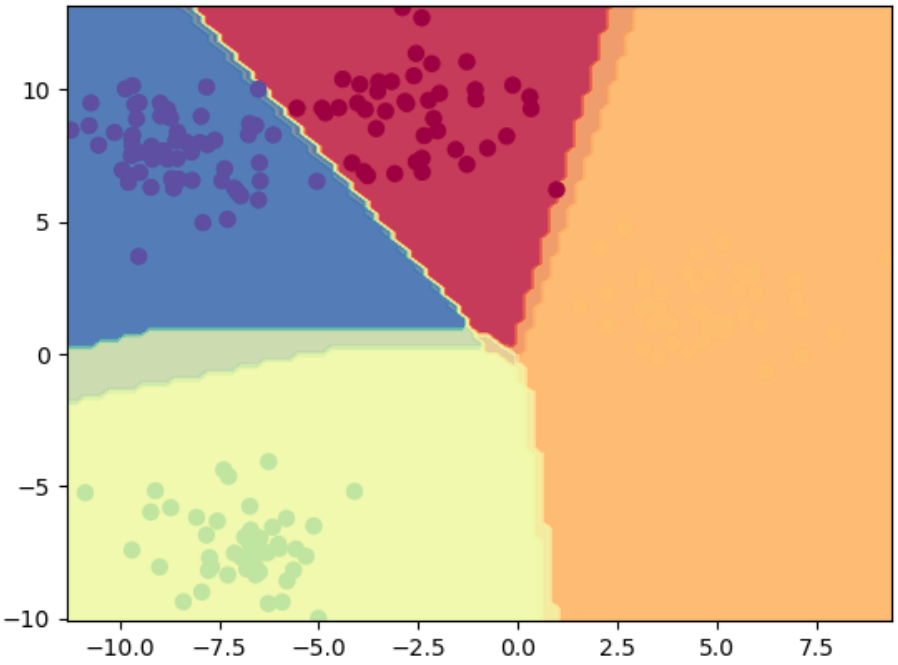

图 5. Blob数据集测试结果