# PyCarrot 机器学习框架

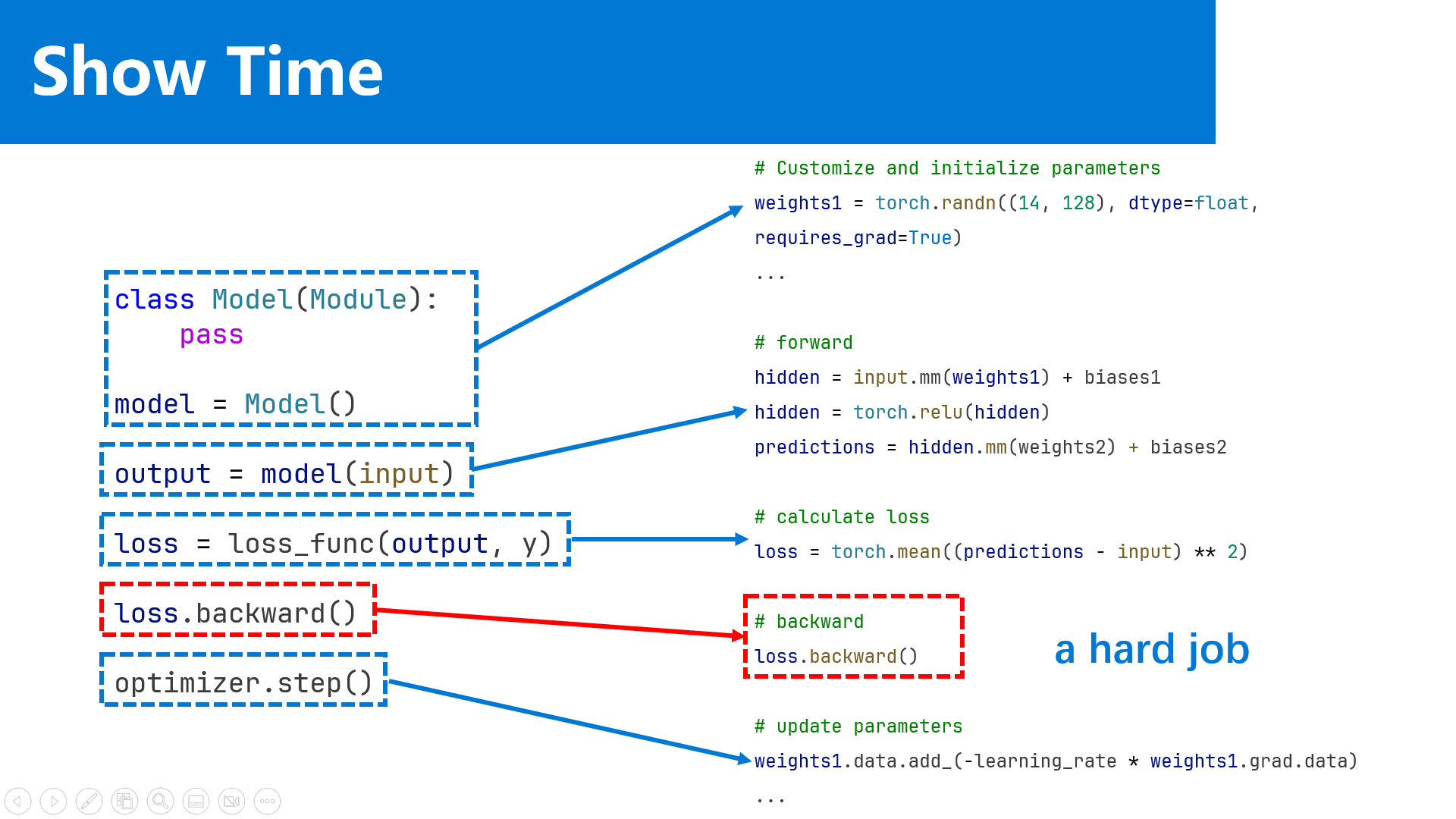
项目地址：<https://github.com/CarrotWuDev/PyCarrot>

## 前言

笔者的研究方向是人工智能方向。笔者在研究生第一学期学习了《机器学习》这门课程，在学习本门课程的过程中，笔者接触了PyTorch这个深度学习框架，PyTorch让整个深度学习过程变得非常简单，开发者不需要关注框架本身的细节，只需要关注数据科学本身。

笔者也是在学习搭建神经网络的过程中了解到深度学习最重要的算法——**反向传播算法**，该算法也是深度学习的核心，正是该算法推动了深度学习的发展。

后面在《高级算法设计》这门课程中，笔者也是实现了反向传播算法，并在课程上分享了该算法。



高级算法设计讲解

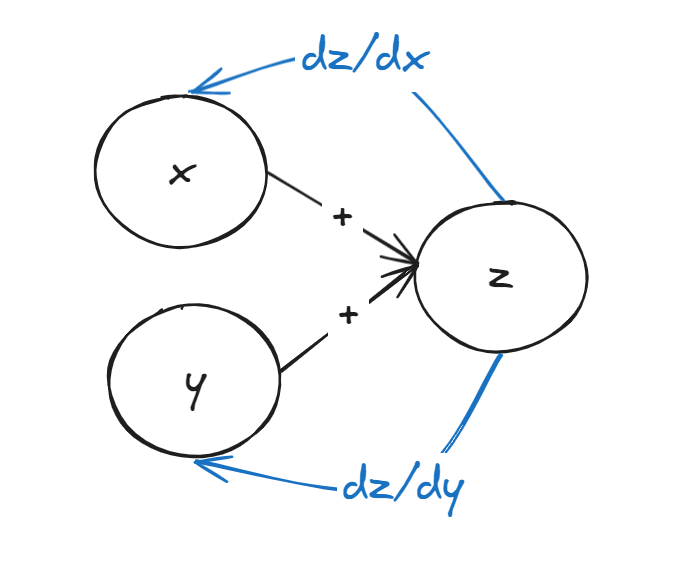
既然笔者已经实现了深度学习的核心——BP算法，那么笔者就在BP算法的基础上构建了一个机器学习框架PyCarrot. > 笔者的英文名是CarrotWu，所以就将该机器学习框架命名为PyCarrot。

## 反向传播算法（自动微分）

### 反向传播算法的介绍

反向传播的是在动微分在工程上的实现。其本质就是利用链式求导法则，从计算结果开始，将微分反向传播到各个计算节点（如PyTorch中的Tensor数据类型）。接着使用梯度对各个参数节点进行更新（优化算法），经过这样的过程，我们的模型在特定数据上会呈现optimum。

以表达式z = x + y为例：



alt text

### 反向传播算法的实现

### 常量版本（不涉及矩阵运算）

#### 知识铺垫

在Pytorch中Tensor数据类型构建了反向传播算法的基础。因为Tensor class不仅存储着数据本身，还存储着**如何传递梯度？**的信息。

主要就是一下两个字段：

1. self.grad：节点的梯度信息。
2. self.grad\_func：节点的梯度该如何传递到子节点，即计算子节点的梯度的函数（数学上的链式法则）。

class Tensor(object):  
 self.data  
 self.grad  
 self.grad\_func  
 ...

以表达式z = x \* y为例，我们来说明一下上面两个字段。

假设计算节点z的grad=a，而z由计算节点x和y进行乘法运算得到，那么z的grad\_func就存储着z的grad如何传递给x和y传递方式。所以grad\_func=[(x, grad\_wrt\_x), (y, grad\_wrt\_y)]。

grad\_wrt\_x用数学表示是这样：

grad\_wrt\_y用数学表示是这样：

用Python代码实现是下面这样：

def grad\_wrt\_x(grad):  
 return grad\*y  
  
def grad\_wrt\_y(grad):  
 return grad\*x

#### 代码实现

下面是Carrot类的实现：

首先是Carrot类的一些属性

class Carrot():  
 def \_\_init(self, data, requires\_grad=False, child\_nodes=[], name=None)  
 self.data = data # Carrot实例中所存储的数据  
 self.child\_nodes = child\_nodes # 子节点信息，相当于Tensor中的grad\_func，其值默认为空  
 self.name = name # 节点的标识信息  
 self.grad = None # 节点的梯度  
 self.requires\_grad = requires\_grad # 当前节点是否需要计算梯度  
 pass

在我们了解了节点的重要信息之后，我们需要用节点来构建表达式。

下面我以乘法为例

x = Carrot(data=2.0, requires\_grad=True, name="x")  
y = Carrot(data=4.0, requires\_grad=True, name="y")

可是Carrot节点之间是如何实现乘法z = x \* y的呢？

先给出结论：

1. 实现Carrot实例内部数据之间的乘法。
2. 构建z节点的依赖信息child\_nodes
3. 根据运算得到的内容构建新的Carrrot实例z

下面是伪代码：

result\_data = x.data \* y.data  
child\_nodes = [(x, grad\_wrt\_x), (y, grad\_wrt\_y)]  
requires\_grad = x.requires\_grad or y.requires\_grad  
  
return Carrot(result\_data, child\_nodes, requires\_grad, name="add")

注意：使用运算符重载实现

def \_\_mul\_\_(self, other: "Carrot") -> "Carrot":  
 """  
 left mul: self \* other  
 """  
 data = self.data \* other.data  
 requires\_grad = self.requires\_grad or other.requires\_grad  
 child\_nodes = []  
 if self.requires\_grad:  
  
 def grad\_wrt\_self(grad):  
 return grad \* other.data  
  
 child\_nodes.append((self, grad\_wrt\_self))  
 if other.requires\_grad:  
  
 def grad\_wrt\_other(grad):  
 return grad \* self.data  
  
 child\_nodes.append((other, grad\_wrt\_other))  
  
 result\_node = Carrot(  
 data=data, child\_nodes=child\_nodes, name="mul", requires\_grad=requires\_grad  
 )  
 return result\_node

我们测试一下

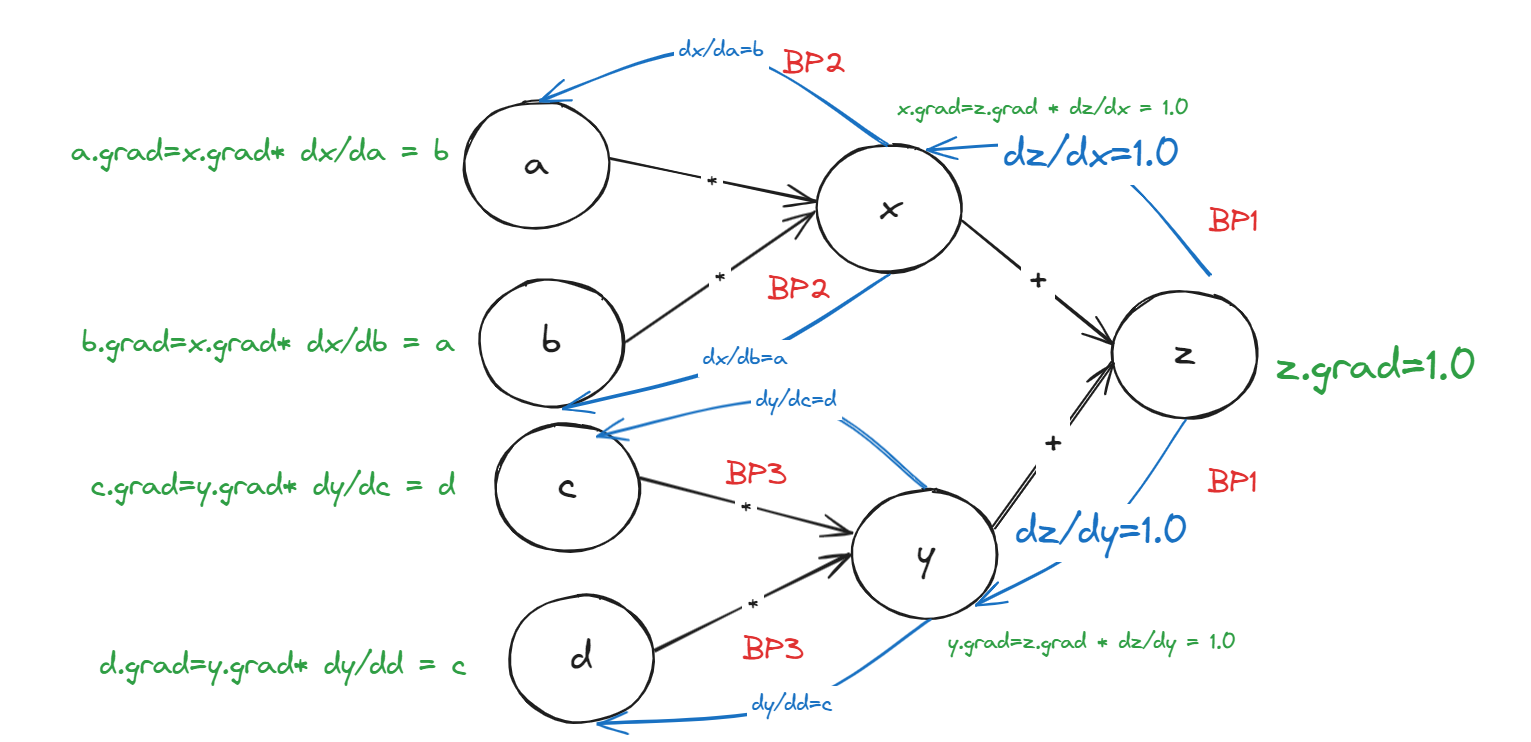
x = Carrot(data=2.0, requires\_grad=True, name="x")  
y = Carrot(data=4.0, requires\_grad=True, name="y")  
z = x + y  
print(z.data)  
print(z.requires\_grad)  
print(z.grad)  
print(z.name)  
print(z.child\_nodes[0][0])  
print(z.child\_nodes[0][1])  
print(z.child\_nodes[1][0])  
print(z.child\_nodes[1][1])  
---  
8.0  
True  
0  
mul  
<\_\_main\_\_.Carrot object at 0x000001B5F0A44AA0>  
<function Carrot.\_\_mul\_\_.<locals>.grad\_wrt\_self at 0x000001B5F0DB6FC0>  
<\_\_main\_\_.Carrot object at 0x000001B5F0E0EE10>  
<function Carrot.\_\_mul\_\_.<locals>.grad\_wrt\_other at 0x000001B5F0DB7380>

既然实现了乘法z = x \* y，那么我们对z应用反向传播算法进行梯度计算。

笔者实现的反向传播主要做了以下几件事：

1. 计算当前节点z的梯度，即设置为1。
2. 遍历当前节点z的child\_nodes:list，通过当前节点的grad和grad\_wrt\_operand()计算子姐弟那的梯度。接着调用子节点bp()将其grad递归传递到叶子节点。
3. 做的是深度优先遍历。

可能比较难理解，给出图示加深理解



alt text

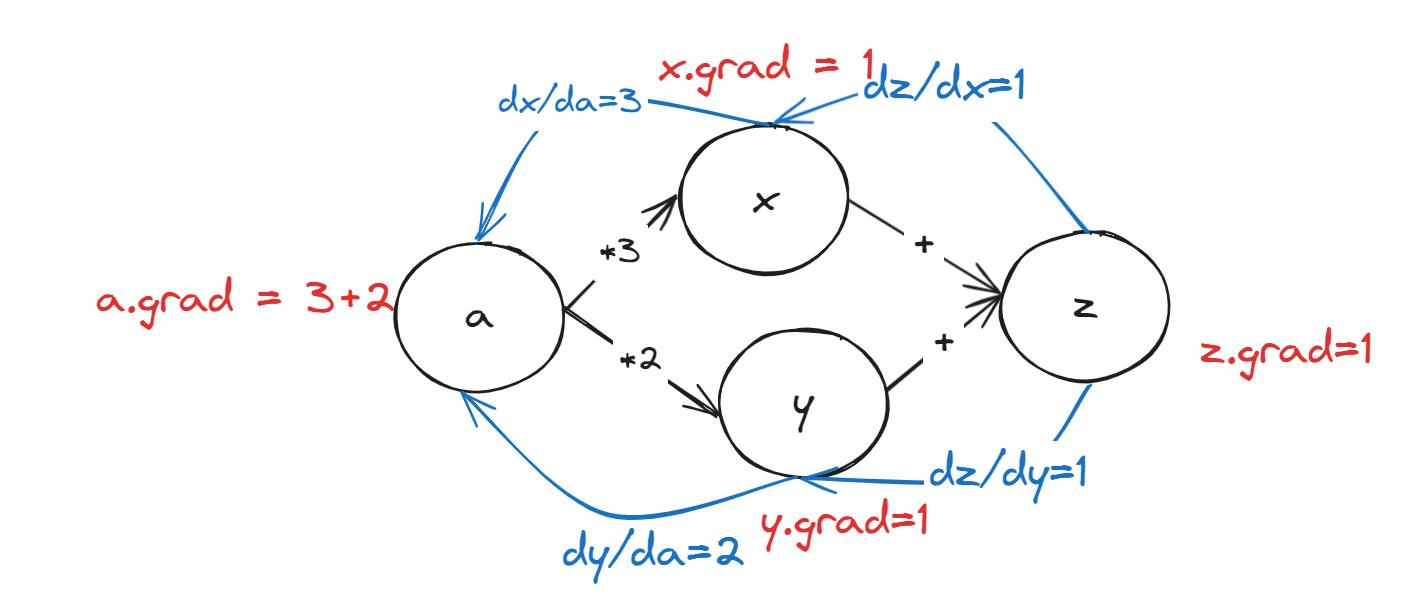
下面是具体的代码实现：

def backward(self, grad=None):  
 """  
 backward pass  
 """  
 if grad is None:  
 self.grad = 1.0  
 grad = 1.0  
 else:  
 self.grad += grad  
 pass  
 # recursion for bp  
 for child\_node, grad\_wrt\_func in self.child\_nodes:  
 child\_node: Carrot  
 child\_node\_grad = grad\_wrt\_func(  
 grad  
 ) # calculate partial grad, in the following bp, need to add partial grad  
 child\_node.backward(child\_node\_grad)  
 pass  
 pass

我详细解释一下下面一段代码：

if grad is None:  
 self.grad = 1.0  
 grad = 1.0  
else:  
 self.grad += grad  
 pass

既然当前节点能够运行BP算法，并且if grad is None:说明是从当前当前节点开始反向传播的，那么要设置当前节点的梯度为1.0，即self.grad=1.0，还要将**要传递的梯度**设置为1.0，即grad=1.0。如果要传递的梯度不为None,则要进行self.grad+=grad，为什么？



alt text

从数学的角度来理解，**即某一个变量可以分不同的路径对结果变量造成影响/改变**，而反向传播每一次是按照一条路径传播梯度的，那么要计算某一个节点的梯度/影响，则要将该变量在每一条路径梯度/影响加和来计算总的梯度/梯度。

我们实现了Carrot类以及对该类的运算和反向传播，下面进行测试。

测试1：

$$
z = x\times y\\
\text{Get:} \frac{\partial z}{\partial x}, \frac{\partial z}{\partial y}
$$

x = Carrot(data=2.0, requires\_grad=True, name="x")  
y = Carrot(data=4.0, requires\_grad=True, name="y")  
z = x \* y  
z.backward()  
print(f"z.grad={z.grad}")  
print(f"y.grad={y.grad}")  
print(f"x.grad={x.grad}")  
  
---  
z.grad=1.0  
y.grad=2.0  
x.grad=4.0

测试2：

$$
a = 2\\
x = 2 \times a\\
y = 3 \times a\\
z = x + y\\
\text{Get}: \frac{\partial z}{\partial a}
$$

a = Carrot(data=1.0, requires\_grad=True, name="a")  
coefficient1 = Carrot(data=2.0, requires\_grad=False)  
coefficient2 = Carrot(data=3.0, requires\_grad=False)  
z = a\*coefficient1 + a \* coefficient2  
z.backward()  
print(f"z.grad={z.grad}")  
print(f"a.grad={a.grad}")  
z.zero\_grad()  
---  
z.grad=1.0  
a.grad=5.0

至此，常量版本的BP算法已经实现，下面我们会在常量版本的基础上构建张量版本的BP算法。

### 张量版本

#### **要点（非常重要）**

**要点1：使用Numpy来表示Carrot的data属性** 使用python中loop来实现的同样维度的矩阵乘法，速度在minute级别。

c = np.zeros((size,size))  
  
for i in range(size):  
 for j in range(size):  
 for k in range(size):  
 c[i,j] += a[i,k] \* b[k,j]  
 pass  
 pass

使用numpy来实现矩阵乘法，速度在ms级别。

import numpy as np  
  
size = 600  
a = np.random.rand(size, size)  
b = np.random.rand(size, size)  
x = numpy.dot(a,b)  
  
8.63 ms ± 2.2 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

**要点2： 某个张量的梯度的shape是和其张量的shape是一样的。**

举个例子

a = [1, 2, 3]  
grad\_a = [1, 1, 1]  
# a.shape = grad\_a.shape = (3,)  
  
b = [[1], [2], [3]]  
b = [[1], [1], [1]]  
# b.shape = grad\_b.shape = (3, 1)

**要点3：矩阵微分** 在本框架的实现中，我们使用了两个矩阵微分。

grad是父节点传播过来的梯度。

$$ S = AB\ = B^T\

= A^T $$

请查阅书籍《The Martrix Cookbook》获取更多矩阵微分相关的信息：<https://www.math.uwaterloo.ca/~hwolkowi/matrixcookbook.pdf>