1. 矩阵求导基础

1.1 矩阵和运算的分量表示

我们在写矩阵的时候通常会这样写：

$$W$$

这是表示矩阵的形式，当然推荐大家使用分量的形式表示即：

$$w_{ij}$$

看起来顺眼多了。在来看看矩阵乘法如何表示的：

$$Y = X \cdot W \rightarrow y_{ij} = \sum x_{ik} \cdot w_{kj}$$

上式对 k 求和就是一个矩阵的乘法了，可以称为点乘。加法符号看着别扭直接去掉：

$$y_{ij} = x_{ik} \cdot w_{kj}$$

这便是约定求和，用在张量分析里挺好用的。写点公式不要太方便，同样的也可以用在机器学习的矩阵求导之中。

1.2 补充两个算子

在开始求导前来介绍两个简单的算子： δ_{ij} 和 ε_{ijk} 。这两个符号代表不同的数学含义，对于 δ_{ij} 仅有 ij 相等时等于 1 ，其他情况等于 0 。这是单位矩阵 I 的分量表示。举个例子：

$$A = I \cdot A \rightarrow a_{ij} = \delta_{ik} \cdot a_{kj}$$

实际上其仅起到指标替换的作用。 ε_{ijk} 当 ijk 有相等的数字时为0，为奇排列时为1，偶排列时为-1。这是向量的叉乘：

$$C = A \times B \rightarrow c_i = \varepsilon_{ijk} a_j b_k$$

其还有着矩阵行列式的用处：

$$|A| = \varepsilon_{ijk} a_{1i} a_{2j} a_{3k}$$

别忘了相同指标代表求和。看不懂么有关系，**这个符号我们应该会用得比较少**。

2. 该上导数了

2.1 从一个矩阵和向量的乘法开始

矩阵乘法可以表示为：

$$Y = AX$$

写成熟悉的分量形式：

$$y_{ij} = a_{ik}x_{kj}$$

好的，接下来就是求 $\frac{\partial y}{\partial x}$ 了，很简单：

$$\frac{\partial (a_{ik}x_{kj})}{\partial x_{kj}} = a_{ik} \to A$$

这是非常轻松就可以完成的计算。**完全不用死记硬背**。 来个 BSDN 那个公式看起来非常难记的：

$$\frac{\partial X^\top A}{\partial X} = A^\top$$

来分量形式

$$y_{ij} = x_{ik}a_{kj}$$

偏导数来了：

$$\frac{\partial (x_{ik}a_{kj})}{\partial x_{ki}}$$

似乎卡主了，因为 x_{ik} 和 x_{ki} 似乎不相同，无法消去了。实际上可以将指标替换一下，即可：

$$\frac{\partial (x_{ki}a_{jk})}{\partial x_{ki}} = a_{jk} \to A^\top$$

2.2 损失函数其实不好理解

记住那些公式没用的一点在于实际的情况可能更加复杂。比如损失函数 L 喜欢这样写：

$$L = \text{sum}\left((Y - D)_2\right)$$

损失函数 L 需要对 Y 求导。有一句话，当我们不知道怎么做的时候实际上是我们对任务拆分的还不够细。实际上上面的公式我们应当拆分为一个减法，一个乘法操作。像这样：

$$A = Y - D$$
$$B = A \odot A$$
$$L = \text{sum}(B)$$

求和的确不好做，但是别忘了，求和中矩阵中每个元素是独立的，因此当求和结果对于每个参数进行求导后，得到的是元素全为 1 的矩阵。如果这个不懂，我们来使用一个矩阵的思想。假设两个向量 pq 中元素全为 1 。此时计算方式为：

$$l_{ij} = p_i b_{ij} q_j$$

求导

$$\frac{\partial l_{ij}}{\partial b_{ij}} = p_i q_j = 1$$

B对A求导为：

$$\frac{\partial a_{ij} a_{ij}}{\partial a_{ij}} = 2a_{ij}$$

A 对 Y 求导全为 1，这里就不写了，有兴趣的可以自行推演。由此我们发现了这个损失函数对于 Y 的偏导数就是：

$$\frac{\partial L}{\partial Y} = 2(y - d)_{ij}$$

2.3 加上一个模型怎么办

这里我们定义的模型是

$$Y = X \cdot W + b$$

损失函数就是 2.1 节所说的。求导怎么办？一步一步来。
首先需要求的是 L 对于 W 的导数。首先写出分量形式：

$$y_{ij} = x_{ik} w_{kj} + b_j$$

对 w 的导数写出来：

$$\frac{\partial L}{\partial Y} \frac{\partial Y}{\partial W} = 2(y-d)_{ij} \frac{\partial (x_{ik} w_{kj})}{\partial w_{kj}} = 2(y-d)_{ij} x_{ik} = 2((Y-D)^\top X)^\top$$

这里看到因为最终计算结果中指标 y_{ij} 与 x_{ik} 是对相同指标 i 进行的求和，因此需要加上一个转置。
如果想求对于 X 的导数怎么办？，也好说：

$$\frac{\partial L}{\partial Y} \frac{\partial Y}{\partial X} = 2(y-d)_{ij} \frac{\partial (x_{ik} w_{kj})}{\partial x_{ik}} = 2(y-d)_{ij} w_{kj} = 2(Y-D)W^\top$$

这就无需多解释了。还有一个小的问题，b 别忘了。这个就更简单了

$$\frac{\partial L}{\partial Y} \frac{\partial Y}{\partial b} = 2(y-d)_{ij} \frac{\partial (x_{ik} w_{kj} + 1_i b_j)}{\partial b_j} = 2(y-d)_{ij} 1_i$$

这里隐含这对于 i 求和的意思。由于 b 是一个向量，其与矩阵相加实际上相当于乘了一个长度为 1 的向量。

3. 程序来了

这里演示一个代码：

```python
import numpy as np

# 定义数据
X = np.random.normal(1, 1, [600, 2])
D = X ** 2 + np.random.normal(0, 0.3, [600, 1])

def model(x, w, b):
    """这是我们建立的模型"""
    y = x @ w + b
    return y
def backward(x, d, w, b):
    """以 MSE 为 loss 的求导过程"""
    y = model(x, w, b)
    # dloss/dy
    dloss = 2 * (y-d) / len(x)
    # dw = dloss/dy * dy/dw
    dw = x.T @ dloss
    db = np.sum(dloss, axis=0)
    return dw, db
```

没什么可以解释的了。

4. 一个基础的自动微分算法

4.1 构建一个计算图

在聊自动求导之前来看下如何计算偏微分。这里自动求导应该是自动微分（Auto Differentiation, AD）。比如一个计算过程：

$$h = f(x)$$
$$y = g(h)$$

（1.1）

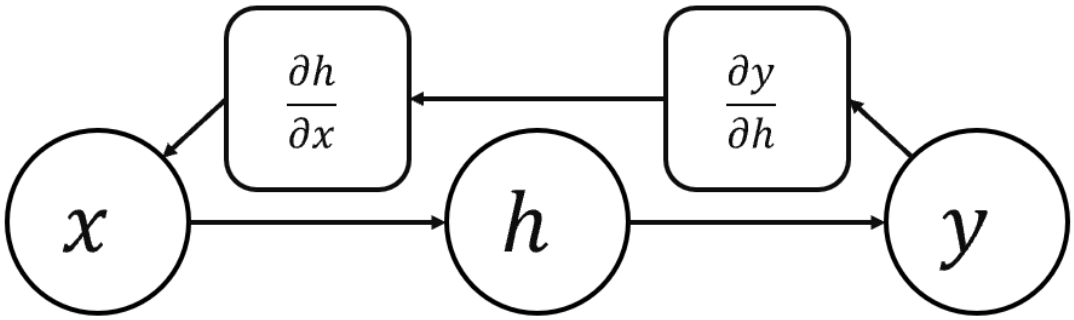
当要计算 y 关于 x 的偏导数过程中，有几个方式可以选择

- 第一符号计算：这种计算方式是很多计算机代数系统采用的方式，即将每个运算均作为符号计算。这需要复杂的代码实现。目前有两个可以推荐的符号计算工具： Mathematica 和 SymPy 。理科生离开这两个很难活，但是机器学习的自动微分用它们有点大材小用了，而且太慢。
- 第二数值求导：将变量取一个小的扰动 δx 并观察变量变化 δy，二者相除即是变量的偏导数。 这种方式并不适合于机器学习中大量的矩阵运算。
  机器学习中的自动微分综合了以上两种思路。即向前计算出 y 后，再反向传播计算偏导数。正向计算的过程中计算过程为：

$$x \to h \to y$$

（1.2）

在此将此正向计算过程称为计算图。实际上构建计算图的过程就是构建变量之间依赖关系的过程。计算图表明了在计算过程中各个变量应对哪些自变量求偏导。比如在上面的关系中应当计算的偏导数为 $\frac{\partial y}{\partial h}$ 和 $\frac{\partial h}{\partial x}$ 这是由计算图的边所决定的。可以看到所谓计算图就是由变量所组成的结点和变量关系所构成的边所组成的有向图。而求 y 关于 x 的偏导数就是求取结点 x 到结点 y 之间所有路径的链式求导。如图：



在这里插入图片描述最终计算的偏导数为：

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial h}\frac{\partial h}{\partial x}$$

（1.3）

所以在实现此类自动微分中仅需实现一个反向传播的过程即可。以一个更长依赖的作为例子，其有向图为：

$$x \to \cdots \to h^t \to h^{t+1} \to \cdots \to y$$

（1.4）

偏微分计算过程为

$$\frac{\partial y}{\partial x}=\frac{\partial y}{\partial h^l}\cdots\frac{\partial h^{t+1}}{\partial h^t}\cdots\frac{\partial h^1}{\partial x}$$

(1.5)

在计算的过程中需要按照有向图反向传播计算。以其中的一个局部 $\cdots\rightarrow h^t\rightarrow h^{t+1}\rightarrow\cdots$ 来看，需要的计算过程为：

- 正向计算即从 h^l 到 h^{l+1}
- 本节点偏导数 $\frac{\partial h^{t+1}}{\partial h^t}$, 称为grad_fn， 这编程需要用到。
- 本层反向传播输入的偏导数 $\frac{\partial y}{\partial h^t}$，称为反向传播误差。

即沿有向图反向传播回去即可。

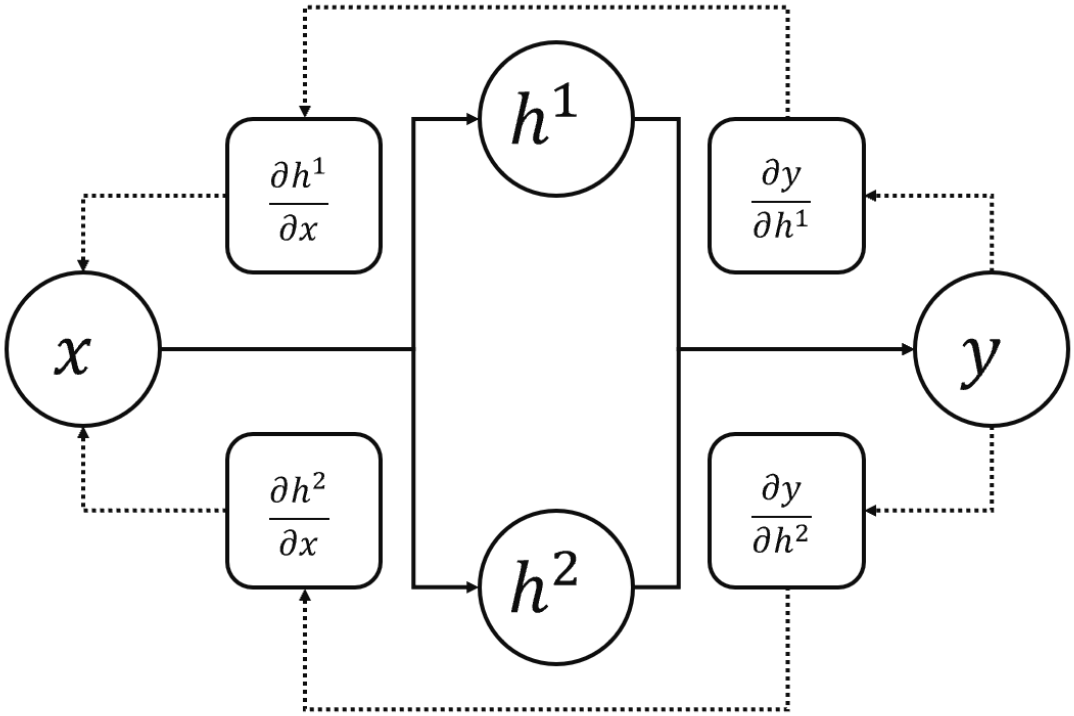
4.2 当有支路时怎么办

这里定义一个更加复杂的情景即有向图中是有支路的：

$$\begin{aligned}h^1&=f^1(x)\\h^2&=f^2(x)\\y&=g(h^1,h^2)\end{aligned}$$

（1.6）

此时绘制的有向图如图所示：

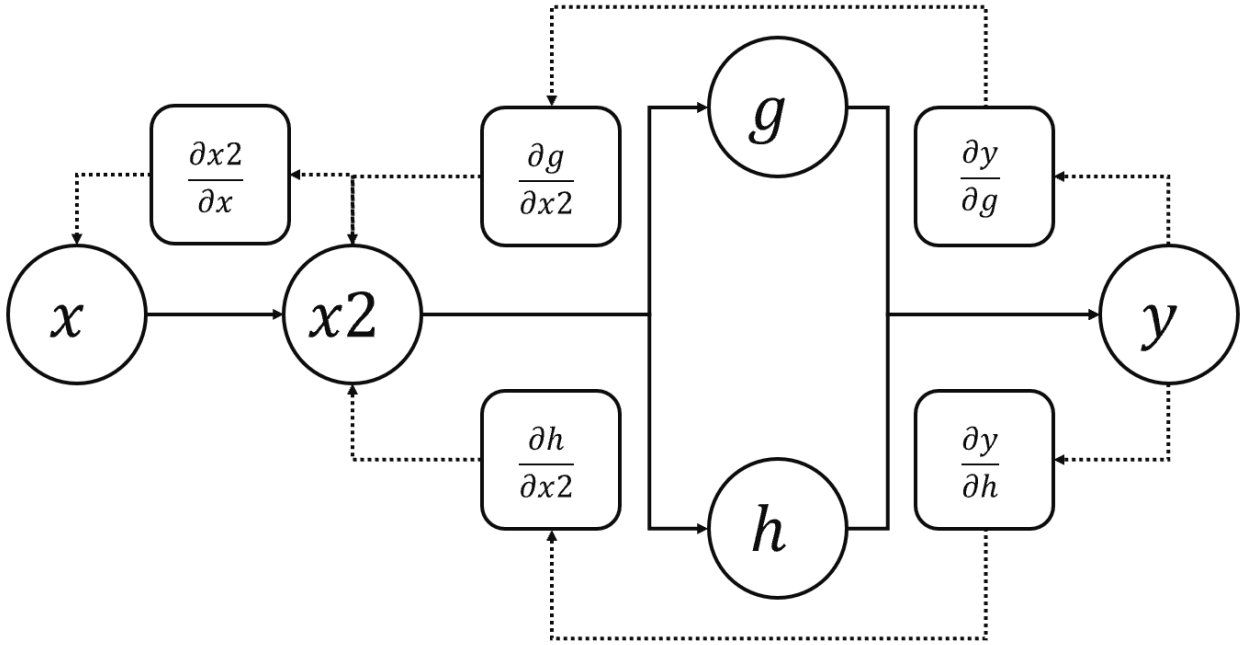


在这里插入图片描述计算图看起来稍微复杂一些了。此时第一个问题是 y 的自变量有两个，此时需要而求解关于 x 的导数过程中所有路径均需要计算，因此需要对此进行求导。另外一个问题是有向图在 x 处出现了分支，此时有两个边在反向求导过程中均需要对 x 求偏导。此时在求偏导过程中需要对于到达此处的两个导数求和。最终计算的导数需要对两个分支均进行计算。这需要计算 x 到达 y 的所有路径，这有些困难。因此本文使用一种代价更高的方式，将大部分的计算都可以拆分为一元或二元计算，这里列举一个更加复杂的例子：

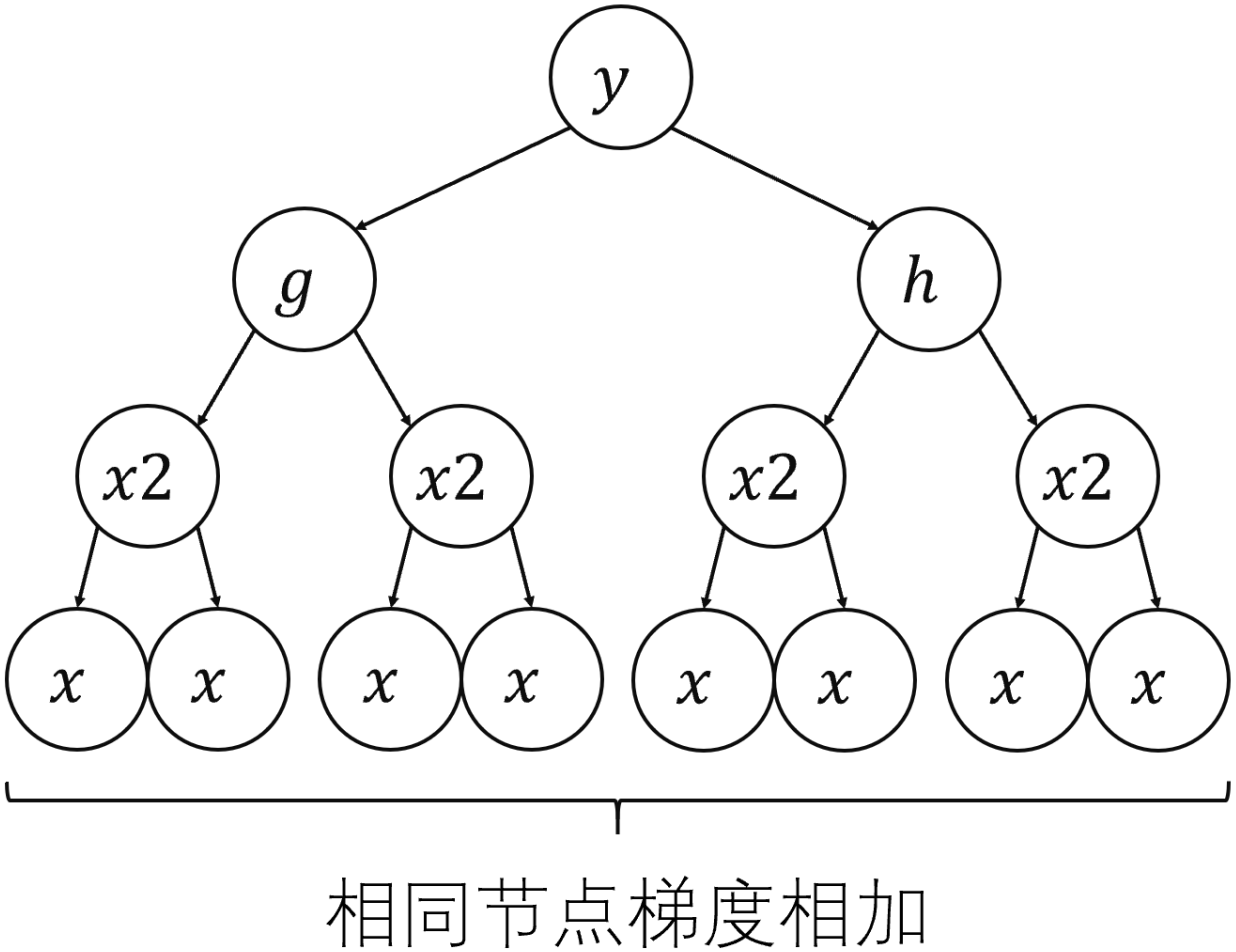
$$
\begin{aligned}
x &= 2 \\
x2 &= x^2 \\
g &= x2 \cdot x2 \\
h &= x2 \cdot x2 \\
y &= g + h
\end{aligned}
$$

（1.8）

求 $\frac{\partial y}{\partial x}$，在此绘制的有向图为：



在这里插入图片描述上图直接计算难以求解，因此将计算进行拆解为树状结构：



在这里插入图片描述进行这种拆解是为了更加方便的进行编程。在求导过程中仅需沿着 y 走到每个叶节点即可。这是适合于递归的方式进行编程的。而递归计算完成后每个节点的梯度需要进行相加。将上述递归过程写为程序。这个递归计算在有较多支路时是偏大的，有较多重复计算，可以使用图算法进行优化。

4.3 递归式计算

为1.8式构建一个自动微分算法。从一个简单的代码开始：

```
class Tensor:
    def __init__(self, data, depend=[], name=None):
        """初始化"""
        self.data = data
        self.depend = depend
        self.name = name
        self.grad = 0
    def __mul__(self, data):
        """乘法"""
        def grad_fn1(grad):
            return grad * data.data
        def grad_fn2(grad):
            return grad * self.data
        depend = [(self, grad_fn1), (data, grad_fn2)]
        new = Tensor(self.data * data.data, depend, name="mul")
        return new
    def __rmul__(self, data):
        def grad_fn1(grad):
            return grad * data.data
        def grad_fn2(grad):
            return grad * self.data
        depend = [(self, grad_fn1), (data, grad_fn2)]
        new = Tensor(self.data * data.data, depend, name="mul")
        return new
    def __add__(self, data):
        """加法"""
        def grad_fn(grad):
            return grad
        depend = [(self, grad_fn), (data, grad_fn)]
        new = Tensor(self.data + data.data, depend, name="add")
        return new
    def __radd__(self, data):
        def grad_fn(grad):
            return grad
        depend = [(self, grad_fn), (data, grad_fn)]
        new = Tensor(self.data + data.data, depend, name="add")
        return new
    def __pow__(self, n):
        """
        幂乘
        """
        def grad_fn(grad):
            return grad * n * self.data ** (n-1)
        depend = [(self, grad_fn)]
        new = Tensor(self.data ** n, depend, name="pow")
        return new
    def __repr__(self):
        return f"Tensor:{self.data}"
    def backward(self, grad=None):
        """
        反向传播，需要递归计算
```

```
        """

        if grad == None:
            self.grad = 1
            grad = 1
        else:
            # 这一步用于计算累积积分
            self.grad += grad
        # 这一步是递归计算每个节点
        for tensor, grad_fn in self.depend:
            # 传播的梯度不是累积梯度
            bw = grad_fn(grad)
            tensor.backward(bw)
x = Tensor(2)
x.name = "x"
x2 = x ** 2
x2.name = "x2"
f = x2 * x2
f.name = "f"
g = x2 * x2
g.name = "g"
y = f + g
y.name = "y"
y.backward()

print(f"y={y}, dy/dg={g.grad}, dy/dx={x.grad}")
```

计算的输出为

> y=Tensor:32, dy/dg=1, dy/dx=64

5. 神经网络中的自动求导

神经网络中有几个计算，其中基本单元是矩阵。由此需要定义一个 Tensor 。并且 Tensor 中的元素都是 ndarray 。此时需要实现的基本计算为：

- 加减乘除计算
- 矩阵乘法
- 卷积计算
- 矩阵切片和矩阵连接
- 激活函数

```
"""
自动求导功能的实现
参考github:autograd
"""

import numpy as np

def data_trans(data):
    """
    转换为array类型数据
    """
    if isinstance(data, np.ndarray):
        return data
    else:
        return np.array(data)
def tensor_trans(data):
    """
    转换为Tensor类型
    """
    if isinstance(data, Tensor):
        return data
    else:
        return Tensor(data)

class Tensor:
    def __init__(self, data, training=False, depends_on=[], name="input"):
        self._data = data_trans(data)
        self.training = training
        self.shape = self._data.shape
        self.grad = None
        self.depends_on = depends_on
        self.step = -1
        self.name = name
        if self.training:
            self.zero_grad()

    def zero_grad(self):
        self.grad = Tensor(np.zeros_like(self.data, dtype=np.float64))

    @property
    def data(self) -> np.ndarray:
        return self._data

    @data.setter
    def data(self, new_data: np.ndarray):
        self._data = new_data
        self.grad = None

    def __repr__(self):
        return f"Tensor({self._data}, training={self.training})"
```

```
def __add__(self, other):
    """加法"""
    return _add(self, tensor_trans(other))

def __radd__(self, other):
    """右加"""
    return _add(tensor_trans(other), self)

def __mul__(self, other):
    """乘法"""
    return _mul(self, tensor_trans(other))

def __rmul__(self, other):
    """右乘"""
    return _mul(tensor_trans(other), self)

def __matmul__(self, other):
    """矩阵点乘，运算符号@"""
    return _matmul(self, tensor_trans(other))
def __rmatmul__(self, other):
    """矩阵点乘，运算符号@"""
    return _matmul(tensor_trans(other), self)

def __sub__(self, other):
    """减法"""
    return _sub(self, tensor_trans(other))

def __rsub__(self, other):
    """右减"""
    return _sub(tensor_trans(other), self)

def __neg__(self):
    """取反"""
    return _neg(self)

def __getitem__(self, idxs):
    """矩阵分片"""
    return _slice(self, idxs)

def backward(self, grad=None):
    if grad is None:
        if self.shape == ():
            grad = Tensor(1.0)
    self.grad.data = self.grad.data + grad.data
    for tensor, grad_fn in self.depends_on:
        backward_grad = grad_fn(grad.data)
        tensor.backward(Tensor(backward_grad))
def export_graph(self, prev=0, point=[], edge=[], prevname="out"):
    """绘制计算图"""
    if prev == 0:
        point = [0]
```

```
            edge = []
            step = 0
        a = np.max(point)
        if self.step not in point:
            self.step = a + 1
            point.append(self.step)
            edge.append((f"{self.name}:{self.step}", f"{prevname}:{prev}"))
        else:
            edge.append((f"{self.name}:{self.step}", f"{prevname}:{prev}"))
        for tensor, grad_fn in self.depends_on:
            tensor.export_graph(self.step, point, edge, self.name)
        return point, edge
    def sum(self):
        return tensor_sum(self)


def tensor_sum(t: Tensor) -> Tensor:
    """
    将所有元素进行相加
    """
    data = t.data.sum()
    training = t.training

    if training:
        def grad_fn(grad):
            """
            本层梯度就是本身，前一层传播的梯度乘以1.
            """
            return grad * np.ones_like(t.data)
        depends_on = [(t, grad_fn)]

    else:
        depends_on = []

    return Tensor(data,
                  training,
                  depends_on, "sum")

def _add(t1: Tensor, t2: Tensor) -> Tensor:
    """加法"""
    data = t1.data + t2.data
    training = t1.training or t2.training
    depends_on = []
    if t1.training:
        def grad_fn1(grad):
            # 在梯度计算假设梯度维度为[N, H, W, C] 而加入的偏置b维度为[C]
            # 所以实际上应当对于 N, H, W 维度进行相加才是b的偏导数。
            ndims_added = grad.ndim - t1.data.ndim
            for _ in range(ndims_added):
                grad = grad.sum(axis=0)
            # 在维度为1的地方进行相加
```

```
            for i, dim in enumerate(t1.shape):
                if dim == 1:
                    grad = grad.sum(axis=i, keepdims=True)
            return grad

        depends_on.append((t1, grad_fn1))

    if t2.training:
        def grad_fn2(grad: np.ndarray) -> np.ndarray:
            # 在梯度计算假设梯度维度为[N, H, W, C] 而加入的偏置b维度为[C]
            # 所以实际上应当对于 N, H, W 维度进行相加才是b的偏导数。
            ndims_added = grad.ndim - t2.data.ndim
            for _ in range(ndims_added):
                grad = grad.sum(axis=0)
            # 在维度为1的地方进行相加
            for i, dim in enumerate(t2.shape):
                if dim == 1:
                    grad = grad.sum(axis=i, keepdims=True)
            return grad
        depends_on.append((t2, grad_fn2))

    return Tensor(data,
                  training,
                  depends_on,
                  "add")

def _mul(t1: Tensor, t2: Tensor) -> Tensor:
    """矩阵乘法"""
    data = t1.data * t2.data
    training = t1.training or t2.training
    depends_on = []
    if t1.training:
        def grad_fn1(grad: np.ndarray) -> np.ndarray:
            grad = grad * t2.data
            # 在梯度计算假设梯度维度为[N, H, W, C] 而加入的偏置b维度为[C]
            # 所以实际上应当对于 N, H, W 维度进行相加才是b的偏导数。
            ndims_added = grad.ndim - t1.data.ndim
            for _ in range(ndims_added):
                grad = grad.sum(axis=0)
            # 在维度为1的地方需要相加
            for i, dim in enumerate(t1.shape):
                if dim == 1:
                    grad = grad.sum(axis=i, keepdims=True)
            return grad
        depends_on.append((t1, grad_fn1))

    if t2.training:
        def grad_fn2(grad: np.ndarray) -> np.ndarray:
            grad = grad * t1.data
            # 在梯度计算假设梯度维度为[N, H, W, C] 而加入的偏置b维度为[C]
            # 所以实际上应当对于 N, H, W 维度进行相加才是b的偏导数。
```

```
            ndims_added = grad.ndim - t2.data.ndim
            for _ in range(ndims_added):
                grad = grad.sum(axis=0)
            # 在维度为1的地方需要相加
            for i, dim in enumerate(t2.shape):
                if dim == 1:
                    grad = grad.sum(axis=i, keepdims=True)
            return grad
        depends_on.append((t2, grad_fn2))
    return Tensor(data,
                  training,
                  depends_on, "mul")

def _neg(t: Tensor) -> Tensor:
    """取反"""
    data = -t.data
    training = t.training
    if training:
        depends_on = [(t, lambda x: -x)]
    else:
        depends_on = []

    return Tensor(data, training, depends_on, "neg")

def _sub(t1: Tensor, t2: Tensor) -> Tensor:
    """减法"""
    return t1 + -t2

def _matmul(t1: Tensor, t2: Tensor) -> Tensor:
    """
    矩阵点乘
    此部分内容可以参阅本人《无痛学习矩阵求导》
    """
    data = t1.data @ t2.data
    training = t1.training or t2.training
    depends_on = []
    if t1.training:
        def grad_fn1(grad: np.ndarray) -> np.ndarray:
            """梯度计算"""
            return grad @ t2.data.T
        depends_on.append((t1, grad_fn1))

    if t2.training:
        def grad_fn2(grad: np.ndarray) -> np.ndarray:
            """梯度计算"""
            return t1.data.T @ grad
        depends_on.append((t2, grad_fn2))
    return Tensor(data,
                  training,
                  depends_on, "matmul")
```

```
def _slice(t: Tensor, idxs) -> Tensor:
    """切片操作"""
    data = t.data[idxs]
    training = t.training

    if training:
        def grad_fn(grad: np.ndarray) -> np.ndarray:
            """梯度计算就是切片位置"""
            bigger_grad = np.zeros_like(data)
            bigger_grad[idxs] = grad
            return bigger_grad
        depends_on = [(t, grad_fn)]
    else:
        depends_on = []

    return Tensor(data, training, depends_on, "slice")
def conv2d(inputs: Tensor, filters: Tensor, stride: int, padding="SAME") -> Tensor:
    """卷积计算，后续会补充理论"""
    training = inputs.training or filters.training
    depends_on = []
    B, H, W, C = np.shape(inputs.data)
    K, K, C, C2 = np.shape(filters.data)
    if padding == "SAME":
        H2 = int((H-0.1)//stride + 1)
        W2 = int((W-0.1)//stride + 1)
        pad_h_2 = K + (H2 - 1) * stride - H
        pad_w_2 = K + (W2 - 1) * stride - W
        pad_h_left = int(pad_h_2//2)
        pad_h_right = int(pad_h_2 - pad_h_left)
        pad_w_left = int(pad_w_2//2)
        pad_w_right = int(pad_w_2 - pad_w_left)
        X = np.pad(inputs.data, ((0, 0),
                                 (pad_h_left, pad_h_right),
                                 (pad_w_left, pad_w_right),
                                 (0, 0)), 'constant', constant_values=0)
    elif padding == "VALID":
        H2 = int((H - K)//stride + 1)
        W2 = int((W - K)//stride + 1)
        X = inputs
    else:
        raise "parameter error"
    out = np.zeros([B, H2, W2, C2])
    for itr1 in range(B):
        for itr2 in range(H2):
            for itr3 in range(W2):
                for itrc in range(C2):
                    itrh = itr2 * stride
                    itrw = itr3 * stride
                    out[itr1, itr2, itr3, itrc] = np.sum(X[itr1, itrh:itrh+K, itrw:itrw+K, :] *
    if inputs.training:
        def grad_fn1(grad: np.ndarray) -> np.ndarray:
```

```
            """梯度计算"""
            error = np.zeros_like(X)
            for itr1 in range(B):
                for itr2 in range(H2):
                    for itr3 in range(W2):
                        for itrc in range(C2):
                            itrh = itr2 * stride
                            itrw = itr3 * stride
                            error[itr1, itrh:itrh+K, itrw:itrw+K, :] += grad[itr1, itr2, itr3, i
            return error
        depends_on.append((inputs, grad_fn1))

    if filters.training:
        """梯度计算"""
        def grad_fn2(grad: np.ndarray) -> np.ndarray:
            dw = np.zeros_like(filters.data)
            for itr1 in range(B):
                for itr2 in range(H2):
                    for itr3 in range(W2):
                        for itrc in range(C2):
                            itrh = itr2 * stride
                            itrw = itr3 * stride
                            dw[:, :, :, itrc] += grad[itr1, itr2, itr3, itrc] * X[itr1, itrh:itr
            return dw
        depends_on.append((filters, grad_fn2))
    return Tensor(out, training, depends_on, "conv2d")
def relu(t1: Tensor) -> Tensor:
    """通过激活函数"""
    training = t1.training
    depends_on = []
    if t1.training:
        def grad_fn(grad: np.ndarray) -> np.ndarray:
            """梯度计算"""
            grad2 = np.copy(grad)
            grad2 = grad * (t1.data>0).astype(np.float64)
            return grad2
        depends_on.append((t1, grad_fn))
    return Tensor(np.clip(t1.data, 0, np.inf), training, depends_on, "conv2d")
def concat(array_list, axis=-1) -> Tensor:
    """矩阵连接操作"""
    depends_on = []
    training = False
    datas = np.concatenate([itr.data for itr in array_list], axis)
    dims = [itr.data.shape[axis] for itr in array_list]
    for itr in array_list:
        if itr.training:
            training = itr.training
    for idx, itr in enumerate(array_list):
        if itr.training:
            def grad_fn(grad: np.ndarray) -> np.ndarray:
                return np.split(grad, dims, axis)[idx]
```

```
            depends_on.append((itr, grad_fn))
    return Tensor(datas, training, depends_on, "concat")
```

首先来绘制一下计算图：

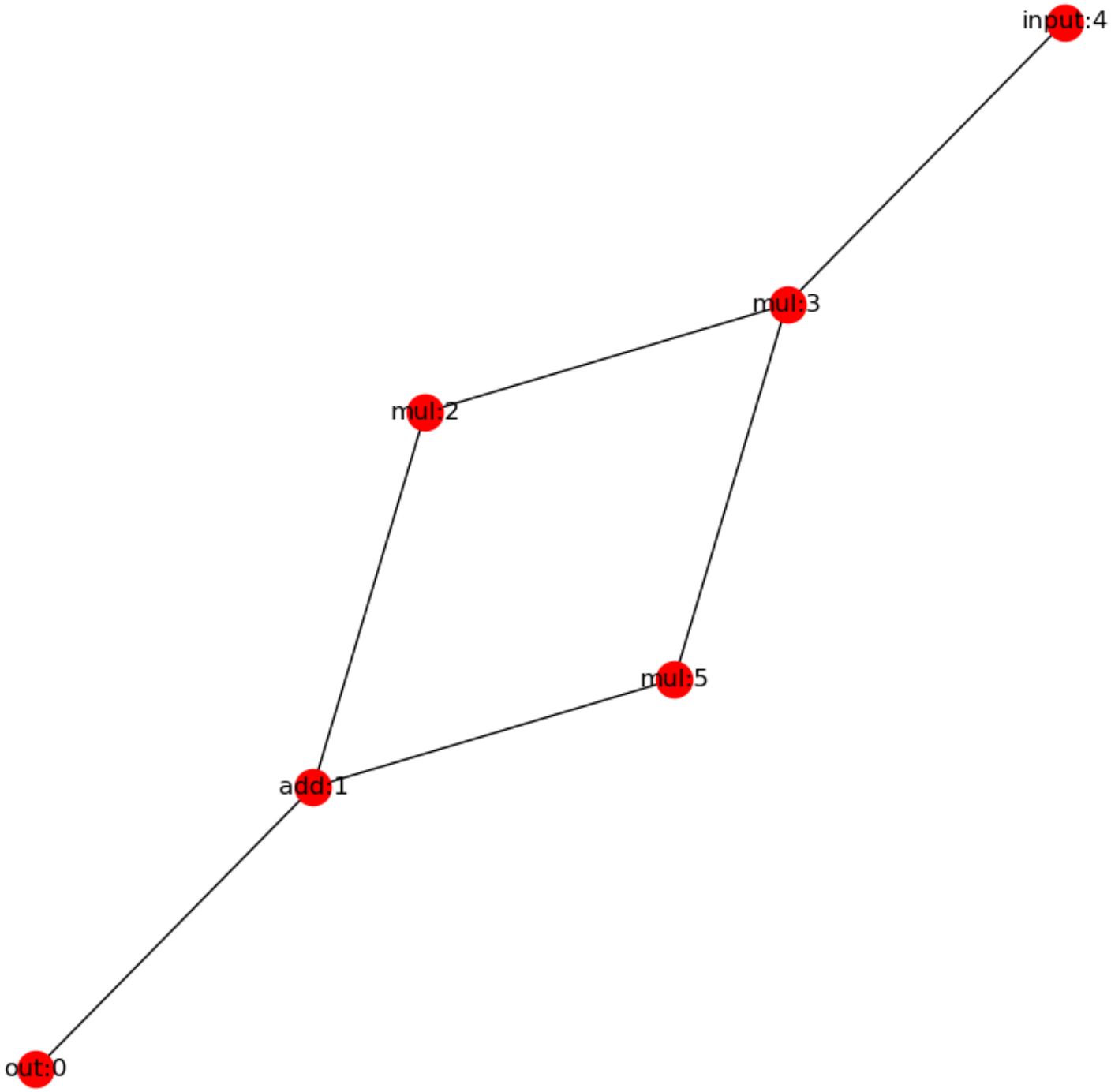
```
import matplotlib.pyplot as plt
import networkx as nx

G = nx.Graph()

x = Tensor(1, True)
x2 = x * x
g = x2 * x2
h = x2 * x2
y = g + h

a, b = y.export_graph()
G.add_edges_from(b)
nx.draw(G, with_labels=True, arrows=True)
plt.show()
```

在这里插入图片描述



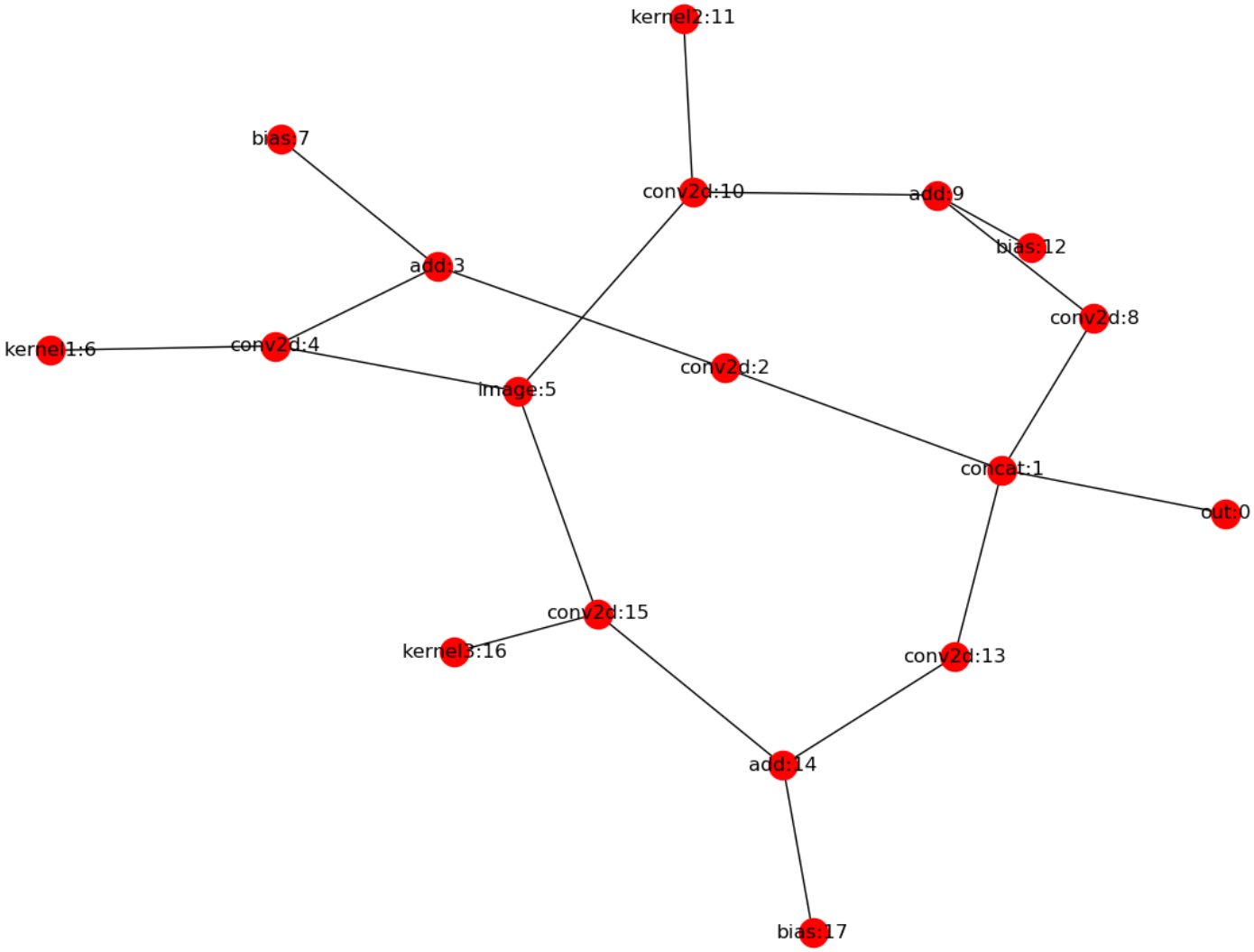
6. 尝试构建一个卷积神经网络

这里使用自动求导的方式构建一个残缺的 Inception 结构（GoogleNet）：

```
import matplotlib.pyplot as plt
import networkx as nx
# 假设的输入
x = Tensor(np.ones([4, 10, 10, 128]), True, [], "image")
# 可训练参数
b1w = Tensor(np.random.normal(0, 1/np.sqrt(1*1*128), [1, 1, 128, 128]), True, [], "kernel1")
b1b = Tensor(np.zeros([128]), True, [], "bias")
b2w = Tensor(np.random.normal(0, 1/np.sqrt(3*3*128), [3, 3, 128, 128]), True, [], "kernel2")
b2b = Tensor(np.zeros([128]), True, [], "bias")
b3w = Tensor(np.random.normal(0, 1/np.sqrt(5*5*128), [5, 5, 128, 128]), True, [], "kernel3")
b3b = Tensor(np.zeros([128]), True, [], "bias")
# Inception优化
# 当然这个是有问题的
# Inception在5*5的分支可以拆分为两个，同时还应该加入逐点卷积限制通道数。
branch1 = conv2d(x, b1w, 1) + b1b
branch1 = relu(branch1)
branch2 = conv2d(x, b2w, 1) + b2b
branch2 = relu(branch2)
branch3 = conv2d(x, b3w, 1) + b3b
branch3 = relu(branch3)
out = concat([branch1, branch2, branch3], axis=-1)

G = nx.Graph()
a, b = out.export_graph()
G.add_edges_from(b)
nx.draw(G, with_labels=True, arrows=True)
plt.show()
```

最终绘制的图为：



在这里插入图片描述看起来已经非常复杂了。 image 是输入，还是有点乱，这是画图的问题。

节选自《深度学习框架基础》，请勿公开转载。

注释：本文在书写过程中参考了 Github 上的 autograd 项目代码，如果算是抄袭也是可以的。有几个缺陷要说明：

- 有向图中以迭代方式进行计算实际上在支路较多的时候会出现微分计算代价过高的问题，需要后续优化
- 池化等操作尚未完成
- 代码并非原创