一、内存泄漏原因

之前用nnengine-pro训练模型时出现内存泄漏，导致训练到一半内存不够分配而停止，分析之后发现原因在于：

每次前向计算输入的张量和中间计算结果张量的对象在内存中没有被释放，导致程序在模型训练的过程内存越来越少，最终爆内存

二、 张量引用来源

在模型训练过程中对张量的来源主要有三种：计算图的引用、张量运算符的引用、模型的引用

1、计算图的引用

问题

在`tensor.py`中有一个表示计算图的类`TcGraph`，用于保存张量计算图，在每次新建一个张量`Tensor`或者用张量运算符`Op`计算出一个结果`Tensor`时，`TcGraph`都会将这些张量加入到其中列表中，并且不会删除，因此在程序运行过程中这些引用将会一直存在，导致这些张量无法被释放

```
class TcGraph:
    instance = None

    qvlehao
    def __init__(self):
        self.tmap = dict()
        # (op_name, (input1, input2, ...), (output1, output2, ...))
        self.graph = list()
```

```
    def addTensor(self, t):
        '''
        alloc a internal repr id for given tensor.
        '''
        tmap = self.tmap
        if t not in tmap:
            tmap[t] = len(tmap)
        return self.getTensor(t)
```

解决

`TcGraph`主要用于后端`T-Lang`，跟咱们前端没有关系，所以我在每次前向传播结束之后都把`TcGraph`清空一次，消除`TcGraph`对所有张量的引用

2、张量运算符的引用

问题

张量之间的运算是由`tensor.py`中的张量运算符类`Op`来实现的，每次进行张量计算时，都会创建一个张量运算符对象，这个对象会产生对输入张量的引用和对输出张量的引用，此外，该张量计算的结果（也是一个张量`Tensor`）还会产生一个对张量运算符的引用，如下所示

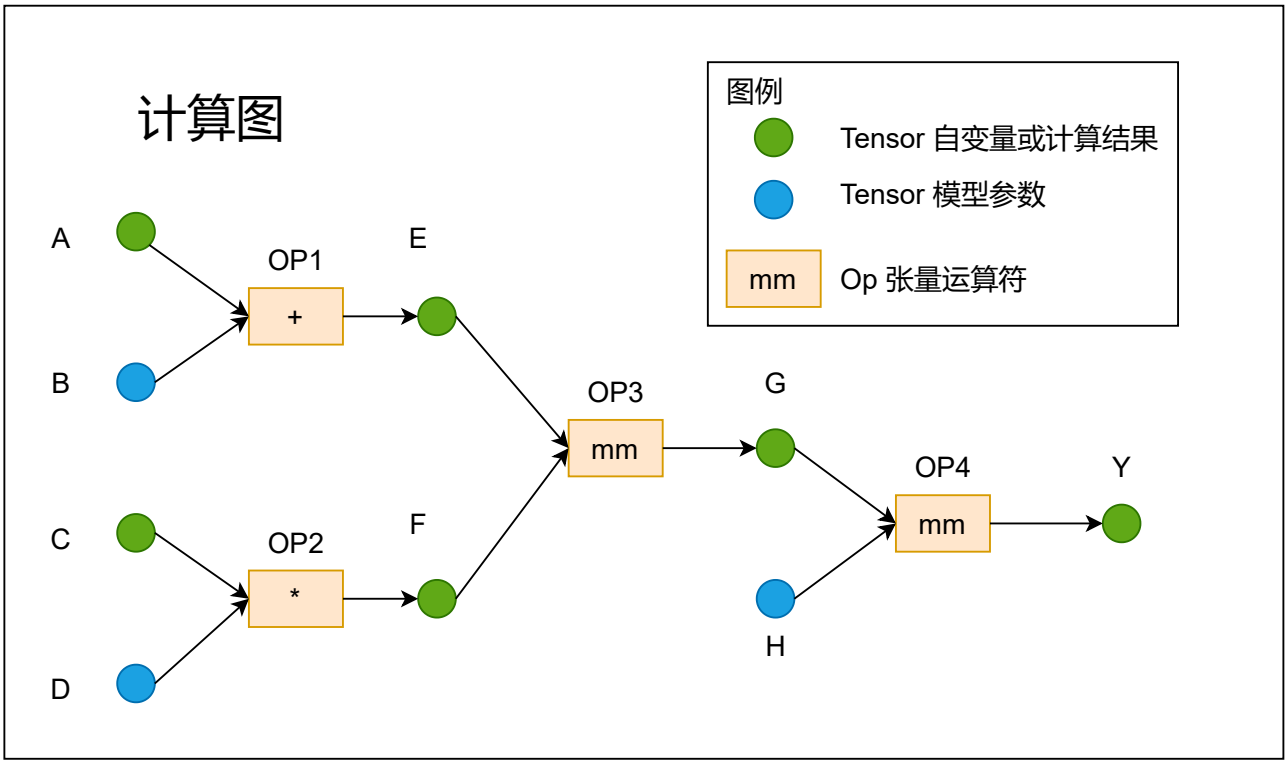
```
class Op:

    qvlehao
    def __init__(self, args, tcc_opname='unsupported'):
        self.input: List[Tensor] = args
        self.output: [Tensor, None] = None
        self.grad_fn = []
```

```
class Tensor(object):

    qvlehao
    def __init__(self, data,
                 autograd: bool = False,
                 creation_op=None):

        self.data = np.array(data, dtype=np.float64)
        self.shape = self.data.shape
        self.autograd = autograd
        if autograd:
            self.grad = np.zeros_like(self.data)
        else:
            self.grad = None

        self.creation_op = creation_op
        self.dependents = {}

        self.tcg_id = TcGraph.AddTensor(self)
```

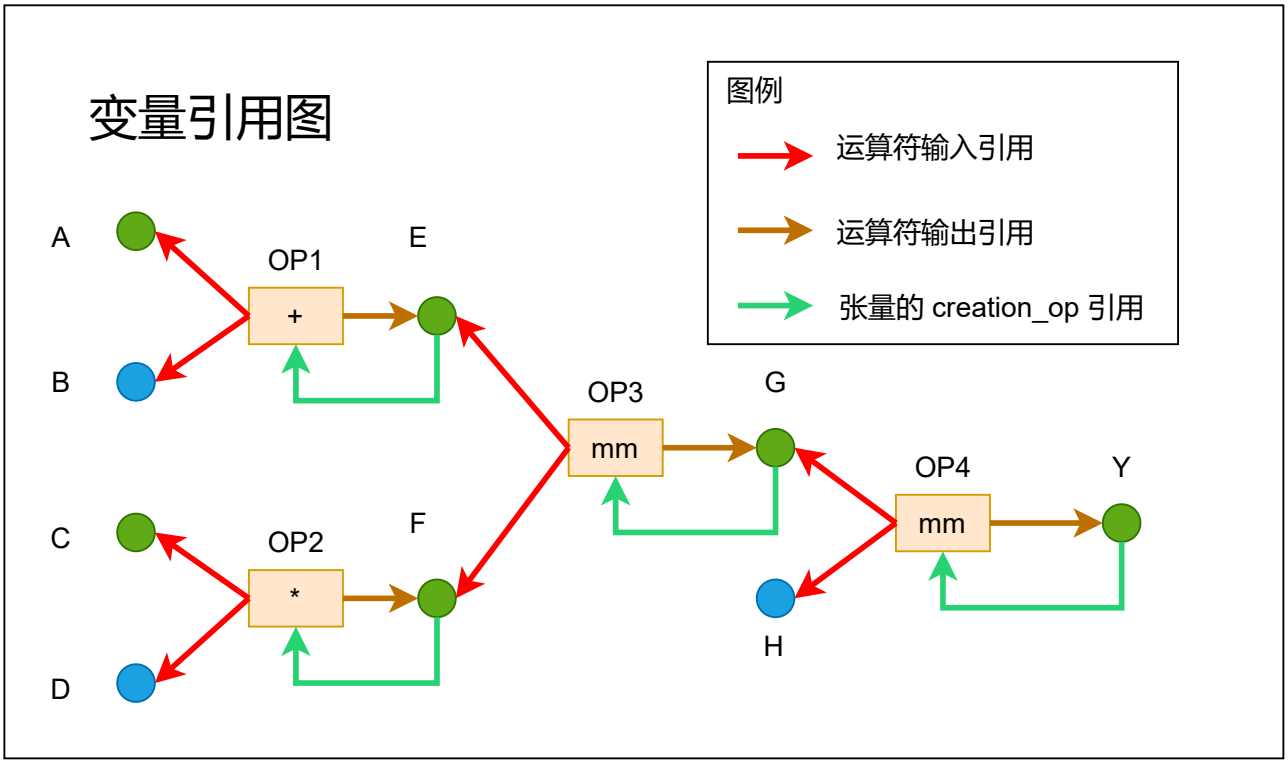
计算图和引用图

对于如下的计算例子，其对的计算图 如下图

```
E = A + B
F = C * D
G = mm(E, F) # mm 是矩阵乘法 mat multiply
H = mm(G, H)
```



对应的张量运算符的引用情况如下：



可以看到，最终的结果张量`Y`和张量运算符`OP4`之间存在循环引用，而其余所有变量的直接或间接被`OP4`所引用，因此在程序运行过程中，计算图中的所有张量和张量运算符的引用不为0，因而所占用的内存不会被释放

解决

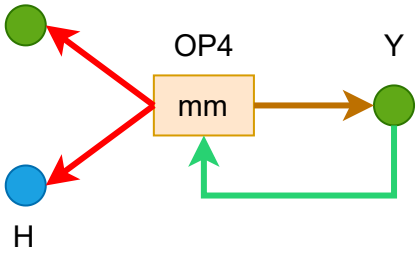
模型的训练是“前向传播-反向传播---前向传播-反向传播---前向传播-反向传播---....”这样一个循环过程。

每次前向传播都将构建一张计算图，对应的反向传播需要利用这张计算图进行梯度更新。

每次前向传播都将输入新的自变量（如上图中的`A`和`C`），根据这些自变量和模型参数（如上图中的`B`、`D`和`H`）构建计算图，因此在模型训练过程中，只需保留模型参数即可，因此每次反向传播结束后，这张动态图中除了模型参数以外其它变量（包括张量`Tensor`和张量运算符`Op`）都可以删掉。

我的解决方法如下：

每次反向传播结束后，以`loss`（最终模型的损失张量）为根节点，用广度优先搜索依次处理计算图的每个张量节点，对每个张量节点，如下图，消除它的张量运算符对输入输出的引用（如下图的红色箭头和棕色箭头），同时消除该张量对张量运算符的引用（如下图的绿色箭头）



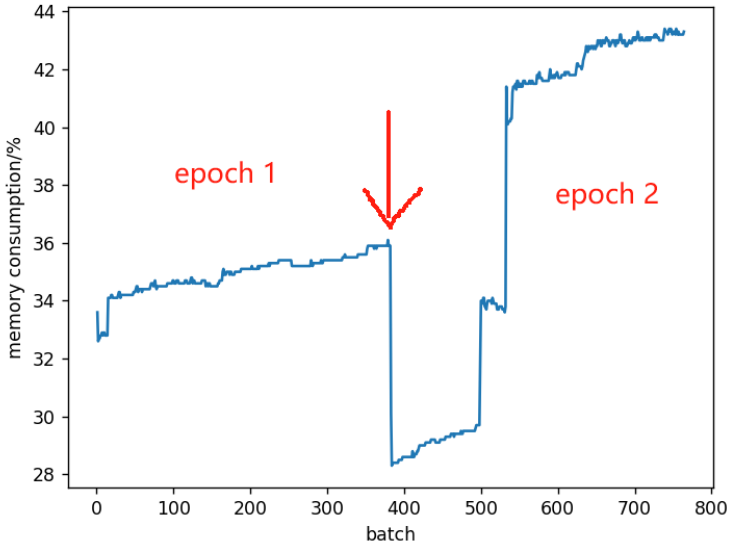

```
def clear_non_parameter_tensor(root: Tensor):
    tensor_queue: queue.Queue[Tensor] = queue.Queue()
    tensor_queue.put(root)
    while not tensor_queue.empty():
        t = tensor_queue.get()
        if t.creation_op is not None:
            for input_tensor in t.creation_op.input:
                tensor_queue.put(input_tensor)
            t.creation_op.input.clear()
            t.creation_op.output = None
            t.creation_op = None
```

3、 模型的引用

模型的参数也是张量，在构建模型时，会创建对应的模型参数张量，从而产生对这些模型参数张量的引用，由于模型在模型训练过程中一直存在，因此对这些模型参数的引用也一直存在，这些张量的内存不会被释放，也不能被释放，否则模型就无了

运行效果

利用上述方法消除引用后，在`minist`上运行了两个轮次，每个轮次训练382批次，运行过程中内存消耗的情况如下，横坐标为训练批次batch，箭头左边为第一轮次，箭头右边为第二轮次


memory consumption/%
batch
epoch 1
epoch 2
44
42
40
38
36
34
32
30
28
0
100
200
300
400
500
600
700
800