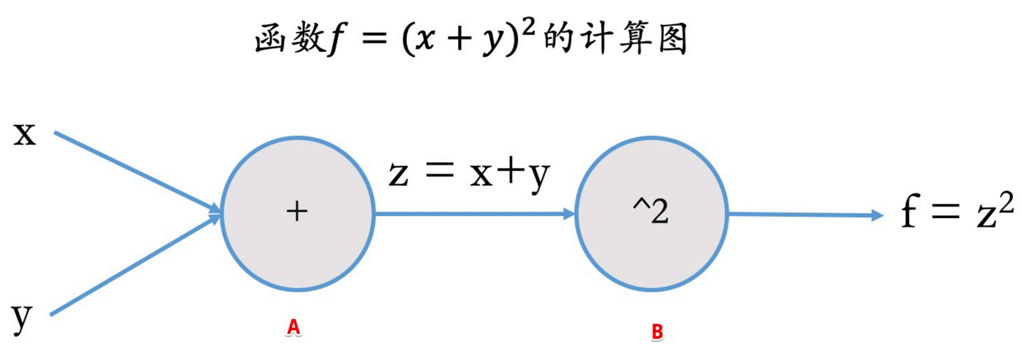
Lab 6：计算图和反向计算

第一部分：计算图的实现（面向过程编程、数值法求解）

本次Lab的主题为计算图的编程实现。首先我们用面向过程编程的方式实现一个简单的计算图。以下面的计算图为例：



步骤1：定义运算门节点

定义需要使用到的运算门。在这个例子中，我们需要用到加法和乘方两种运算。

In [ ]:
```
# 加法：
def add(x,y):
    return x+y

# 乘方：
def square(x):
    return x**2
```

步骤2：前向计算

假设x=1, y=2, 进行前向计算。

In [ ]:
```
x = 1
y = 2
# 输入x和y，做加法后得到中间状态z：

z = add(x,y)
print("中间状态z:", z)

# z经过乘方后，得到输出f：

f = square(z)
print("输出值f:", f)
```

In [ ]:
```
# 将上一个cell结联后：
f = square(add(x,y))
print("输出值f:", f)
```

步骤3：B点的梯度计算

正向计算完成后，我们开始做反向的梯度计算。

首先，用**数值法**计算B点的局部梯度。我们需要定义一个梯度计算函数。

任务1：完成gradient( )函数

传入参数

- func: 当前运算门函数
- value: 前向计算时传入当前运算门的值
- h: 一个极小的数h。

输出为当前位置的局部梯度值。

In [ ]:
```
# 数值法计算当前梯度的函数
# 传入参数：当前运算门函数、前向计算时传入当前运算门的值、一个极小的数h
# 替换下方的Pass，完成函数的设计

def gradient(func, value, h=0.00001):
    pass
```

In [ ]:
```
# 计算B点的局部梯度
value = z
gradient_b = gradient(square, value, 0.00001)
print(gradient_b)
```

计算B点的下游梯度：

根据链式法则，下游梯度 = 上游梯度 * 局部梯度，

B点的上游梯度为base case = 1

In [ ]:
```
# 计算B点的下游梯度，即A点的上游梯度：
base_case = 1
a_upstream = gradient_b * base_case
print(a_upstream)
```

步骤4： A点的梯度计算（分叉）

接下来，用数值法计算A点的局部梯度。

在A点的下游方向，计算出现了**分叉**，因此我们需要另一个求梯度的函数。

它的传入参数包括：

- func: 当前的运算门函数
- values: 前向计算时传入的两个值组成的array（两条路径各一个传入值，一般会把计算图中位置**靠上**线路的传入值放在前面）
- h: 一个极小的数h。

输出是一个长度为2的array，分别为两个方向的局部梯度，同样，一般会按照计算图的绘制，将图中靠上线路的梯度值放在前面。

任务2：完成gradient_bifurcated( )函数

按上面的提示，替换掉下面函数中的None，将其完成。

In [ ]:
```
# 替换掉下面的None，完成函数的编写
def gradient_bifurcated(func, values, h=0.00001):
    # 靠上的线路：
    gradient1 = None
    # 靠下的线路
    gradient2 = None

    return [gradient1,gradient2]
```

In [ ]:
```
# 计算A点在两条路径上的局部梯度：

gradient_a = gradient_bifurcated(add, [x,y], 0.00001)
print(gradient_a)
```

In [ ]:
```
# 利用链式法则，计算A在靠上路径的下游梯度，即x到f的总梯度：
x_gradient = gradient_a[0] * a_upstream
print(x_gradient)
```

In [ ]:
```
# 利用链式法则，计算A在靠下路径的下游梯度，即y到f的总梯度：
y_gradient = gradient_a[1] * a_upstream
print(y_gradient)
```

步骤5/任务3：计算总梯度

可以利用gradient( )及gradient_bifurcated( )两个函数，根据计算图，求取x、y到f的总梯度。

替换掉下面cell中的None，计算x和y到f的总梯度。

In [ ]:
```
base_case = 1
x = 1
y = 2
h = 0.00001

#x到f的总梯度
x_gradient = None
print("x到f的总梯度:",x_gradient)

#y到f的总梯度
y_gradient = None
print("y到f的总梯度:",y_gradient)
```

至此，我们算出了x、y到f的总梯度的数值解（近似值）。

这种面向过程的思路可以帮我们完成简单计算图的构建和求解，但它只能计算近似值，且当计算图变得复杂时，此方法将难以胜任，因此我们要采用面向对象编程的思路。

第二部分：计算图的实现（面向对象编程、解析法求解）

本次练习只针对“单主线”的计算图，分叉现象仅限于输入值相加或相乘的情况。如果计算图为多路径，需要在本练习的基础上，用递归的思路解决。不熟悉Python面向对象编程的同学，可以参考：[https://www.w3ccoo.com/python/python_classes.html (https://www.w3ccoo.com/python/python_classes.html)](https://www.w3ccoo.com/python/python_classes.html) 。

步骤1：定义运算门节点

将运算门节点作为一个“类”（class）。在初始化时，做以下定义：

4个attributes，包括：

- 该节点前向计算的输入值
- 该节点前向计算的输出值
- 该节点的局部梯度
- 该节点反向计算时输出的下游梯度

前向计算的函数：forward( )，该函数将调动其子类别中具体的compute_forward( )函数。compute_forward( )将在子类别中定义，子类别按计算的类别划分，比如加、减、乘、除等，从而进行前向计算。

反向计算的函数：backward( )，需要传入上游梯度和选择的路径（默认为0，即没分叉的情况，或跟随靠上的路径），该函数将调动其子类别中具体的compute_backward( )函数。

In [ ]:
```
class Node:
    def __init__(self):
        self.inputs = None #该节点前向计算的输入值
        self.outputs = None #该节点前向计算的输出值
        self.derivative = None #该节点的局部梯度
        self.downstream = [] #该节点反向计算时输出的下游梯度，因为下游梯度可能出现分叉的情况，因此用一个空list表示

    def forward(self):
        if not self.outputs:
            self.compute_forward() #调动子类别中具体的compute_forward()函数
        return self.outputs

    def backward(self,upstream, path=0): #分叉时，Path = 0为计算图中靠上的路径；Path = 1为计算图中靠下的路径
        self.compute_backward(upstream, path=0) #调动子类别中具体的compute_backward()函数
```

步骤2：定义加法节点

该节点为运算门节点的子类别，它需要传入一个list，该list包含两个节点。初始化时，这两个传入节点的输出值会记录在self.inputs中。

该节点的compute_forward()函数将把self.outputs定义为list中两数之和

在compute_backward()函数中，首先要计算当前节点的局部梯度。f=x+y，无论是对x求导，还是对y求导，当前局部梯度都为1，因此需要把self.derivative设为1。然后，通过让局部梯度和上游梯度相乘，计算下游梯度。注意，加法节点需要分别计算两个方向的下游梯度，虽然它们数值相同，但需要将下游节点self.downstream设置为一个长度为2的数组。

In [ ]:
```
class AdditionNode(Node):
    def __init__(self, node_in):
        Node.__init__(self)
        self.inputs = [node_in[0].outputs,node_in[1].outputs]

    def compute_forward(self):
        self.outputs = self.inputs[0] + self.inputs[1]

    def compute_backward(self, upstream, path=0):
        self.derivative = 1
        self.downstream = [1 * upstream[path], 1 * upstream[path]]
```

步骤3：定义二次方节点

该节点为运算门节点的子类别，它的传入节点只有1个，传入节点的输出值记录在self.inputs中。

该节点的compute_forward()函数将把self.outputs定义为传入实数的平方。

在compute_backward()函数中，首先要计算当前节点的局部梯度。f=x^2，对x求导为：f' = 2x，而此处的x为前向计算时该节点的传入数值，即self.inputs的值。然后，通过让局部梯度和上游梯度相乘，计算下游梯度。

In [ ]:
```
class SquareNode(Node):
    def __init__(self, node_in):
        Node.__init__(self)
        self.inputs = node_in.outputs

    def compute_forward(self):
        self.outputs = self.inputs ** 2

    def compute_backward(self,upstream,path=0):
        self.derivative = 2 * self.inputs
        self.downstream = [self.derivative * upstream[path]]
```

步骤4：定义终端节点

即输入值x和y，可视为特殊的节点，它们没有传入节点，只有手动定义的输入值；没有计算，只有输出值。

In [ ]:
```
class TermNode(Node):
    def __init__(self, value):
        Node.__init__(self)
        self.outputs = value
```

步骤5：定义计算图

将计算图单独定义为一种“类”（class），在初始化时，建立一个空的数组，用于按顺序存储节点。

该类class有3种函数。

首先是添加节点的函数，用于按顺序添加节点。

其次是前向计算的函数，它遍历图中的节点，并依次调用节点中的forward()函数。

第三是反向计算的函数，它通过反方向遍历途中的节点，并依次调用节点中的backward()函数。

注意：节点中的backward()函数需要传入上游梯度。对于图中的最后一个阶段，其上游梯度为base case(即为1)；对于其余节点，其上游梯度为（反方向）上一个节点的下游梯度。终端节点通常不被加入图中，只作为初始参数传入第一个图中的节点。

分叉：在ComputationalGraph的backward()中，需要传入一个包含0和1的list（即path_rt），该list长度和节点的个数相同。它用来列举每次分叉时需要选择的路径，其中0为计算图中靠上的路径，1为计算图中靠下的路径。如果该节点没有分岔，则在相应位置记为0 。比如，上一个例子中，如果计算x到f的路径，则该list为[0, 0]，如果计算y到f的路径，则该list为[1, 0]。

任务4：ComputationalGraph

根据以上提示，替换掉下面的pass，完成ComputationalGraph的编写。

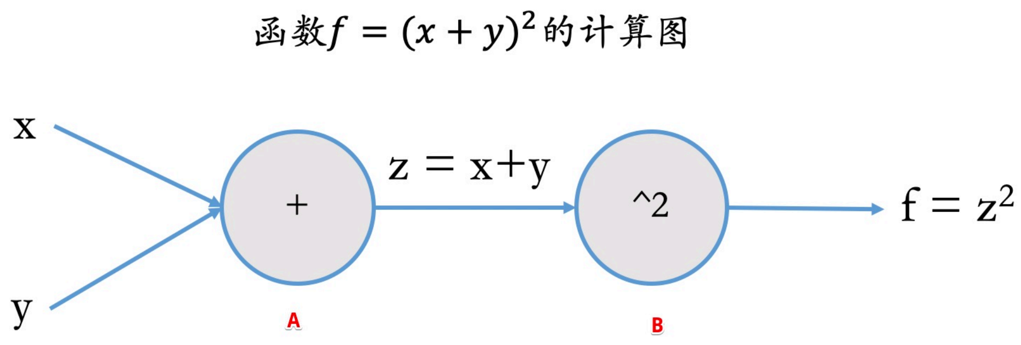
In [ ]:
```
# 替换掉Pass，完成ComputationalGraph的编写
class ComputationalGraph:
    def __init__(self):
        self.nodes = []

    def add_node(self,node):
        pass

    def forward(self):
        pass

    def backward(self,path_rt):
        pass
```

生成一个具体的计算图对象，并完成计算



根据上述示意图，创建一个ComputationalGraph类的对象，命名为graph1

In [ ]:
```
# 初始化一个空的计算图对象
graph1 = ComputationalGraph()

# 定义两个输入值（终端节点）
X = TermNode(1)
Y = TermNode(2)

# 定义A节点，其输入为X和Y两个终端节点
NodeA = AdditionNode([X,Y])
# 将A节点加入计算图
graph1.add_node(NodeA)
# 让A节点做前向计算
graph1.forward()

# 打印当前A节点的状态。注意，由于尚未做反向计算，因此A的局部梯度和下游梯度应该为None
print("节点A的输出为：", NodeA.outputs)
print("节点A的局部梯度为：", NodeA.derivative)
print("节点A的下游梯度为：", NodeA.downstream)
```

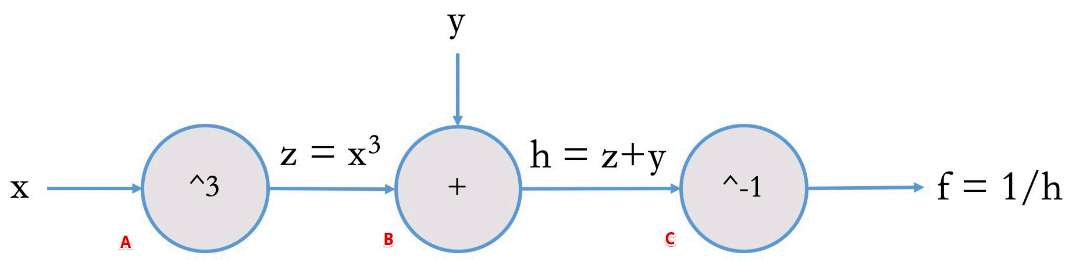
In [ ]:
```
# 定义B节点，其输入节点为A节点
NodeB = SquareNode(NodeA)
# 将B节点加入计算图
graph1.add_node(NodeB)
# 让B节点做前向计算
graph1.forward()

# 打印当前B节点的状态。注意，由于尚未做反向计算，因此B的局部梯度和下游梯度应该为None
print("节点B的输出为：", NodeB.outputs)
print("节点B的局部梯度为：", NodeB.derivative)
print("节点B的下游梯度为：", NodeB.downstream)
```

In [ ]:
```
# 现在，我们已经布置好了所有节点。我们可以通过调用backward()函数，让整个计算图做反向传递
graph1.backward([0,0]) #[0,0]表示，这两个节点如果遇到分叉，下一步都走靠上方的路径（即先被添加到计算图的路径）

# 打印节点A和B的状态，此时局部梯度和下游梯度已经被计算完毕。
print("节点B的输出为：", NodeB.outputs)
print("节点B的局部梯度为：", NodeB.derivative)
print("节点B的下游梯度为：", NodeB.downstream)
print("节点A的输出为：", NodeA.outputs)
print("节点A的局部梯度为：", NodeA.derivative)
print("节点A的下游梯度为：", NodeA.downstream)
```

任务5



根据上图，建立一个计算图对象，名为graph2，并依次添加节点（需定义额外的节点子类别）。设x和y都为2。

完成前向计算和反向计算，获得节点C的输出值f，以及从x到f的总梯度。

In [ ]:
```
# 在下方完成任务5，可使用额外的cell
```

In [ ]:
```
print("节点C的输出为：", NodeC.outputs)
print("节点C的局部梯度为：", NodeC.derivative)
print("节点C的下游梯度为：", NodeC.downstream)
print("节点B的输出为：", NodeB.outputs)
print("节点B的局部梯度为：", NodeB.derivative)
print("节点B的下游梯度为：", NodeB.downstream)
print("节点A的输出为：", NodeA.outputs)
print("节点A的局部梯度为：", NodeA.derivative)
print("节点A的下游梯度为：", NodeA.downstream)
```

提交方式

本次作业有5个任务。完成所有cell的运行后，保存为ipynb和PDF格式（保留所有输出）。将导出的ipynb命名为“Lab6+姓名+学号.ipynb”，将导出的PDF命名为“Lab6+姓名+学号.pdf”，并将上述两个文件提交到学习通作业模块的相应位置（如果ipynb无法单独上传，请打包成zip格式）。 请独立完成练习，参考答案将在截止时间后公布。 截止时间：2024年6月12日23:59。超时1天之内将扣除5%的分数，超时1天以上将扣除10%的分数。