神经网络编程框架

南京大学人工智能学院

申富饶，徐百乐

目录

CONTENTS

01 神经网络编程框架

神经网络编程框架

- TensorFlow：由Google开发的一个开源机器学习框架，广泛用于各类深度学习任务。它支持多种编程语言，包括Python、C++、Java等。
- PyTorch：由Facebook开发的开源机器学习库，特别适合于科研领域，支持动态图计算。
- Keras：最初是一个独立的深度学习框架，以其用户友好和易扩展性著称。现在Keras已经集成到TensorFlow中，成为TensorFlow的默认高级API。
- Caffe：由BVLC维护的深度学习框架，支持C++和Python语言，适合于图像处理任务。
- Theano：由蒙特利尔大学开发，是一个高性能的符号计算及深度学习框架，完全基于Python。
- CNTK (Microsoft Cognitive Toolkit)：由微软开发，通过有向图描述神经网络，支持多种神经网络模型。
- MXNet：由DMLC维护，支持混合符号和命令式编程，具有灵活的编程模型和多语言支持。
- PaddlePaddle：由百度开发的深度学习平台，支持C++和Python语言。
- Deeplearning4j：由Eclipse维护，是一个基于Java和Scala的深度学习库。
- ONNX (Open Neural Network eXchange)：由微软和Facebook等公司共同开发的项目，旨在提供一个开放格式的深度学习模型，简化模型在不同框架间的传递。

神经网络程序的一般结构

1. 准备训练、验证数据，进行数据预处理；
2. 定义神经网络结构，设置超参数；
3. 初始化网络模型、优化器、学习率规划器；
4. 迭代训练：
   - 以mini-batch的形式读取数据，送入网络进行前向传播，计算输出值；
   - 根据输出值与学习目标，用损失函数计算出此次迭代的loss；
   - 将loss进行反向传播，逐层计算梯度、更新参数；
   - 重复此步骤直到达到结束条件，存储模型参数；
5. 读取模型参数和测试数据，将数据送入网络，输出预测值。

神经网络编程框架的功能

- 读取数据与预处理功能；
- 定义张量数据结构，实现基本运算函数库；
- 模块化的神经网络库，实现常见的神经网络模型；
- 实现常见的损失函数；
- 计算反向传播梯度；
- 实现优化器；
- GPU并行计算；
- 储存、读取神经网络。

02 链式求导与计算图

数值微分

函数的导数定义如下：

$$\frac{df(x)}{dx}\Big|_{x=a} = f'(a) = \lim_{x\to a}\frac{f(x)-f(a)}{x-a}$$

数值计算中，常用差商法计算导数：

$$f'(x) = \frac{f(x+h)-f(x)}{h} + O(h) = \frac{f(x-h)-f(x)}{h} + O(h) = \frac{f(x+h)-f(x-h)}{2h} + O(h^2)$$

这三种近似分别称为数值微分的向前差商、 向后差商和中心差商公式, 中心差商公式有更好的精度。

链式求导

神经网络程序中的函数通常是由简单函数复合而来的，因此可以使用“自动微分”技术获得微分的精确表示。 这种方法的基础是微分的链式法则：

$$\frac{dg(f(x))}{dx} = \frac{dg}{df} \cdot \frac{df}{dx}$$

以及函数相加与相乘的导数

$$\frac{d(f(x)+g(x))}{dx} = \frac{df(x)}{dx} + \frac{dg(x)}{dx}, \quad \frac{d(f(x)\cdot g(x))}{dx} = g(x)\cdot\frac{df(x)}{dx} + f(x)\cdot\frac{dg(x)}{dx}$$

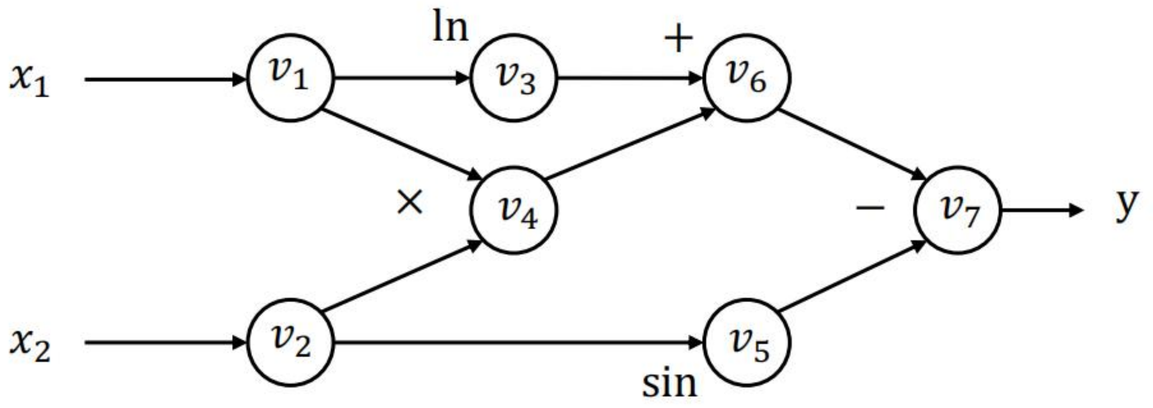
但如果不规划导数计算顺序，可能会导致重复计算，造成计算资源浪费

计算图

计算图是表示复合函数的有向无环图。图中的每个节点都表示一次运算和其运算结果，图中的边表示输入输出关系。比如：

$y = f(x_1, x_2) = ln(x_1) + x_1 x_2 - sin\ x_2$

用计算图的形式表示如下：



当 $x_1 = 2, x_2 = 5$ 时，前向运算：

$v_1 = x_1 = 2$

$v_2 = x_2 = 5$

$v_3 = lnv_1 = ln2 = 0.693$

$v_4 = v_1 v_2 = 10$

$v_5 = sin\ v_2 = \sin 5 = -0.959$

$v_6 = v_3 + v_4 = 10.693$

$v_7 = v_6 - v_5 = 11.652$

$y = v_7 = 11.652$

静态计算图

- 定义：静态计算图是一种在开始计算之前就完全定义好的计算图。它在执行前构建，并且图中的每个节点和边都代表一个具体的操作或数据。
- 执行：在会话（Session）中启动计算图来执行具体的计算任务，根据输入的数据按照图中定义的流程进行计算。
- 优点：

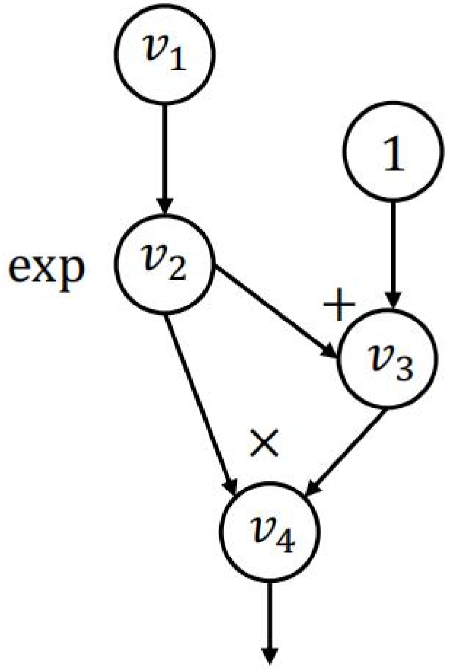
  优化：由于计算图是静态的，编译器和硬件可以对其进行优化，提高执行效率。
- 缺点：

  不灵活：对于需要动态决定网络结构的场景，静态图不够灵活。

  内存占用：由于整个图需要在内存中构建，非常大的图可能会占用大量内存。

  调试困难：需要通过工具或者日志来分析问题所在。

静态计算图




```
import tensorflow as tf

v1 = tf.Variable()
v2 = tf.exp(v1)
v3 = v2 + 1
v4 = v2 * v3

sess = tf.Session()
value4 = sess.run(v4, feed_dict={v1:
numpy.array([1]})
```

动态计算图

- 定义&执行：动态计算图是在运行时逐步构建的。每执行到一个操作代码，就会即时创建对应的节点，并动态地构建出计算图的结构。
- 优点：

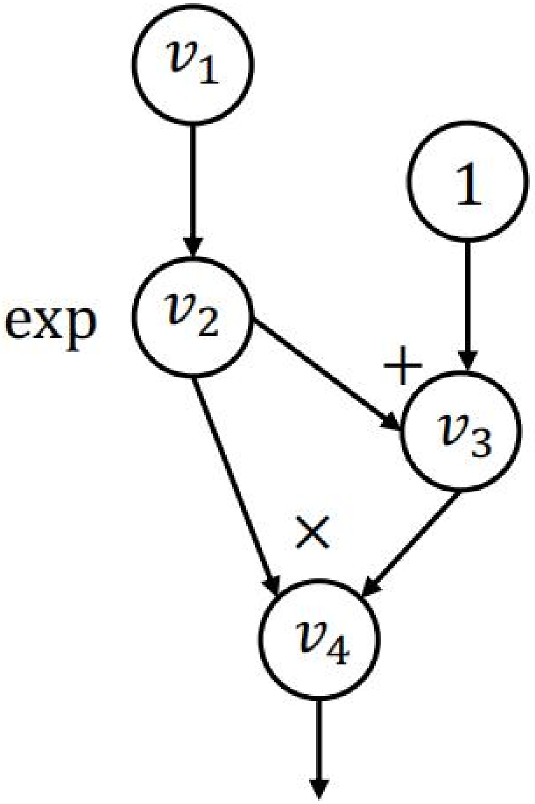
  灵活性：可以轻松处理复杂的控制流和动态图结构。

  内存效率：由于计算图是动态构建的，不需要一次性将整个图加载到内存中。

  Debug：计算图逐步构建与执行，可以在Debug模式下观察变量。
- 缺点：

  性能：相比静态图，动态图可能无法充分利用编译器和硬件优化。

动态计算图




```
import pytorch

v1 = torch.tensor([1])
v2 = torch.exp(v1)
v3 = v2 + 1
v4 = v2 * v3

if v4.numpy() > 0.5:
    v5 = v4 * 2
else:
    v5 = v4
v5.backward()
```

03 自动微分

前向求导

按拓扑顺序，前向计算每个中间变量对输入变量的导数：

$$v_i' = \frac{\partial v_i}{\partial x_1}$$

$$v_1' = 1, v_2' = 0$$

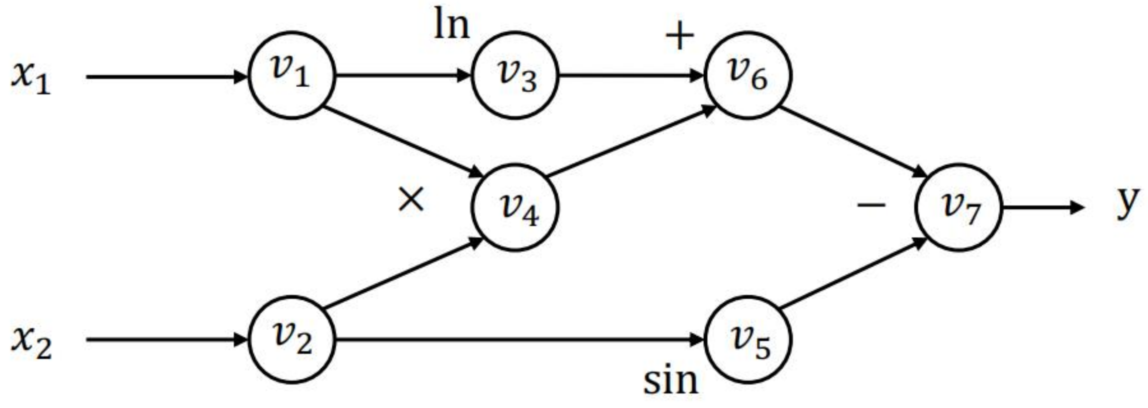
$$v_3' = \frac{1}{v_1} v_1' = 0.5$$

$$v_4' = v_1' v_2 + v_2' v_1 = 5$$

$$v_5' = v_2' \cos v_2 = 0$$

$$v_6' = v_3' + v_4' = 5.5$$

$$v_7' = v_6' - v_5' = 5.5$$



$$v_1 = x_1 = 2$$

$$v_2 = x_2 = 5$$

$$v_3 = ln v_1 = ln 2 = 0.693$$

$$v_4 = v_1 v_2 = 10$$

$$v_5 = sin\ v_2 = \sin 5 = -0.959$$

$$v_6 = v_3 + v_4 = 10.693$$

$$v_7 = v_6 - v_5 = 11.652$$

$$y = v_7 = 11.652$$

前向求导

- 对于函数$f: R^n \to R^k$而言，前向求导需要经过n次前向计算，而神经网络程序中n的数量大，k的数量小

- 为了更有效率地实现自动求导，需要寻找另一种求导方式

反向求导

按逆向拓扑序列，反向计算输出变量对中间变量的导数，即节点v_i的**伴随值**(adjoint)：

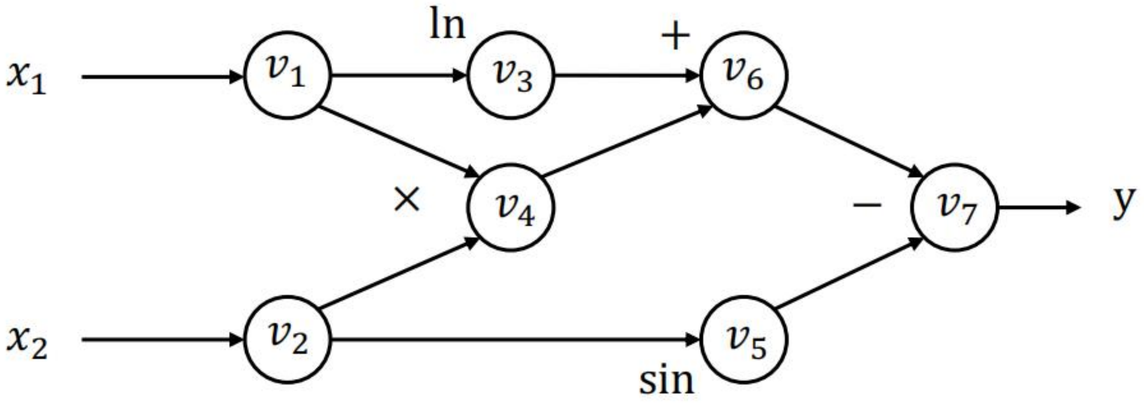
$$\overline{v_i} = \frac{\partial y}{\partial v_i}, \quad \overline{v_7} = 1$$

$$\overline{v_6} = \overline{v_7}\frac{\partial v_7}{\partial v_6} = 1, \quad \overline{v_5} = \overline{v_7}\frac{\partial v_7}{\partial v_5} = -1$$

$$\overline{v_4} = \overline{v_6}\frac{\partial v_6}{\partial v_4} = 1, \quad \overline{v_3} = \overline{v_6}\frac{\partial v_6}{\partial v_3} = 1$$

$$\overline{v_2} = \overline{v_5}\frac{\partial v_5}{\partial v_2} + \overline{v_4}\frac{\partial v_4}{\partial v_2} = \overline{v_5}\, cos v_2 + \overline{v_4} v_1 = 1.716$$

$$\overline{v_1} = \overline{v_4}\frac{\partial v_4}{\partial v_1} + \overline{v_3}\frac{\partial v_3}{\partial v_1} = \overline{v_4}\, v_2 + \overline{v_3}\frac{1}{v_1} = 5.5$$

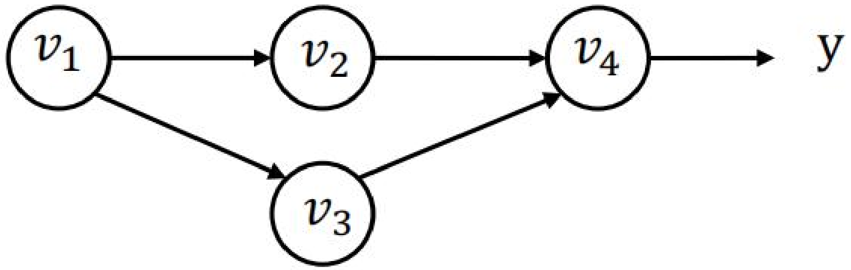


$$v_1 = x_1 = 2$$
$$v_2 = x_2 = 5$$
$$v_3 = ln v_1 = ln 2 = 0.693$$
$$v_4 = v_1 v_2 = 10$$
$$v_5 = sin\ v_2 = \sin 5 = -0.959$$
$$v_6 = v_3 + v_4 = 10.693$$
$$v_7 = v_6 - v_5 = 11.652$$
$$y = v_7 = 11.652$$

局部伴随值

需要额外注意的是多输出的节点：

$$\overline{v_1} = \overline{v_2}\frac{\partial v_2}{\partial v_1} + \overline{v_3}\frac{\partial v_3}{\partial v_1}$$



我们定义**局部伴随值**(partial adjoint)，对于计算图中每个连接$i \to j$：

$$\overline{v_{i\to j}} = \overline{v_j}\frac{\partial v_j}{\partial v_i}, \quad \overline{v_i} = \sum_{j\in out(i)} \overline{v_{i\to j}}$$

计算图的反向求导算法

def gradient(out):

    node_to_grad = {out: [1]} # 声明一个dict结构，用于存储partial adjoint

    for i in reverse_topo_order(out): # 从out节点开始进行反向拓扑排序

        $\overline{v_i} = \sum_{j \in out(i)} \overline{v_{i \to j}}$ = sum(node_to_grad[i]) # 求节点i输出边partial adjoint之和

        for $k \in in(i)$:

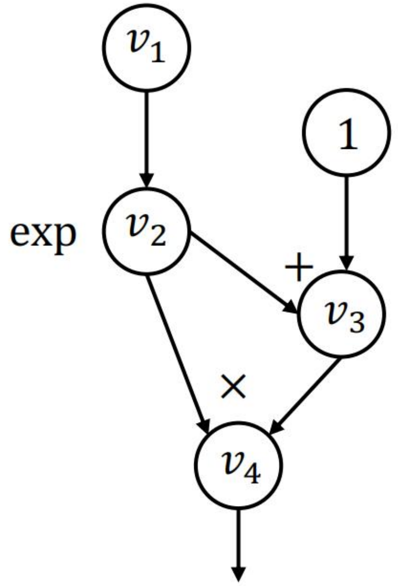
            compute $\overline{v_{k \to i}} = \overline{v_i} \frac{\partial v_i}{\partial v_k}$ # 计算节点i输入边对应的partial adjoint

            append $\overline{v_{k \to i}}$ to node_to_grad[k] # 将计算出的partial adjoint存入词典

return adjoint of input $\overline{v_{input}}$

反向求导的扩展计算图

def gradient(out):
  node_to_grad = {out: [1]}
  for i in reverse_topo_order(out):
    $\overline{v_i} = \sum_{j \in out(i)} \overline{v_{i \to j}}$ = sum(node_to_grad[i])
    for $k \in in(i)$:
      compute $\overline{v_{k \to i}} = \overline{v_i} \frac{\partial v_i}{\partial v_k}$
      append $\overline{v_{k \to i}}$ to node_to_grad[k]
return adjoint of input $\overline{v_{input}}$

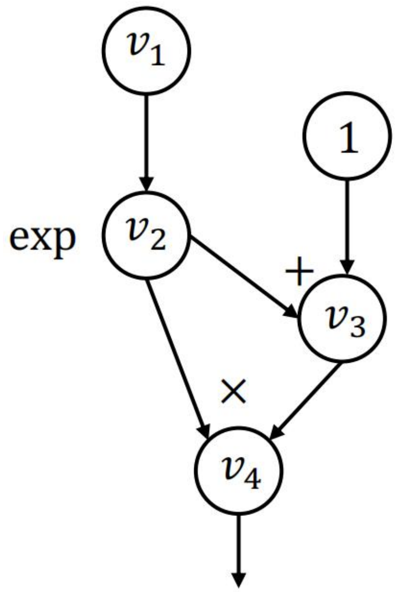


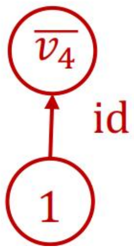
$y = e^x(e^x + 1)$

反向求导的扩展计算图

def gradient(out):
  node_to_grad = {out: [1]}
  for i in reverse_topo_order(out): # i=4
    $\overline{v_i} = \sum_{j \in out(i)} \overline{v_{i \to j}}$ = sum(node_to_grad[i])
    for $k \in in(i)$:
      compute $\overline{v_{k \to i}} = \overline{v_i} \frac{\partial v_i}{\partial v_k}$
      append $\overline{v_{k \to i}}$ to node_to_grad[k]
return adjoint of input $\overline{v_{input}}$

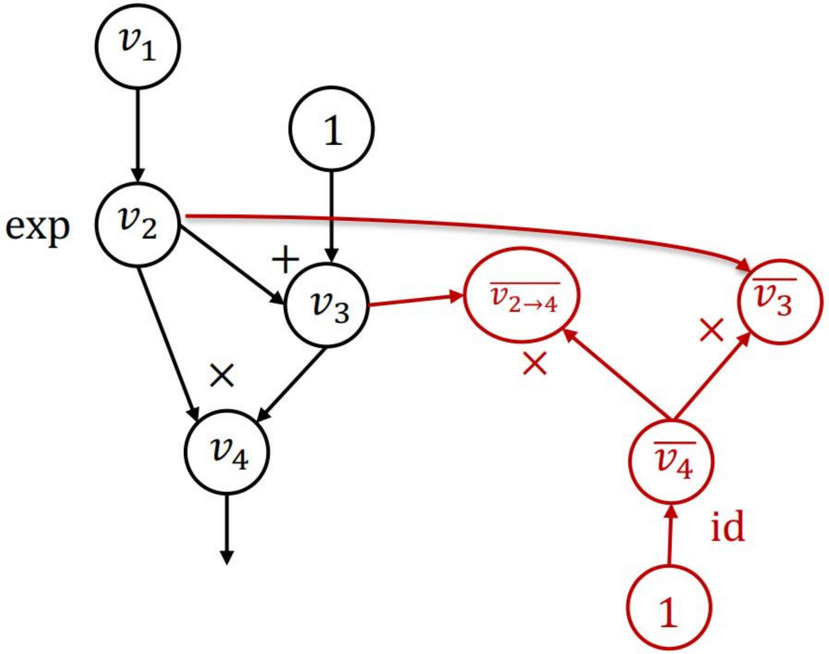
node_to_grad:{
  out:[1]
  4:[$\overline{v_4} = 1$]
}





反向求导的扩展计算图

def gradient(out):
  node_to_grad = {out: [1]}
  for i in reverse_topo_order(out): # i=4
    $\overline{v_i} = \sum_{j \in out(i)} \overline{v_{i\to j}}$ = sum(node_to_grad[i])
    for $k \in in(i)$:
      compute $\overline{v_{k\to i}} = \overline{v_i} \frac{\partial v_i}{\partial v_k}$
      append $\overline{v_{k\to i}}$ to node_to_grad[k]
return adjoint of input $\overline{v_{input}}$



node_to_grad:{
  out:[1]
  4:[$\overline{v_4} = 1$]
  3:[$\overline{v_{3\to 4}} = \overline{v_4} \times v_2 = v_2$]
  2:[$\overline{v_{2\to 4}} = \overline{v_4} \times v_3 = v_3$]
}

反向求导的扩展计算图

```
def gradient(out):
  node_to_grad = {out: [1]}
  for i in reverse_topo_order(out):  # i=3
     v̄_i = Σ_{j∈out(i)} v̄_{i→j} = sum(node_to_grad[i])
     for k ∈ in(i) :
        compute v̄_{k→i} = v̄_i ∂v_i/∂v_k
        append v̄_{k→i} to node_to_grad[k]
return adjoint of input v̄_input
```

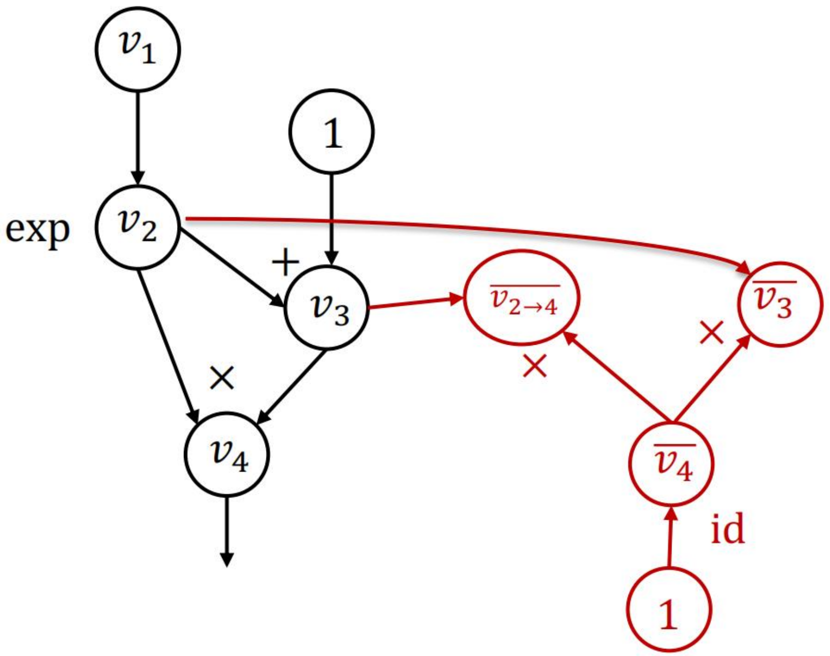
$\overline{v_i} = \sum_{j \in out(i)} \overline{v_{i\to j}} = \mathrm{sum}(\mathrm{node\_to\_grad}[i])$

compute $\overline{v_{k\to i}} = \overline{v_i}\frac{\partial v_i}{\partial v_k}$

append $\overline{v_{k\to i}}$ to node_to_grad$[k]$

return adjoint of input $\overline{v_{input}}$

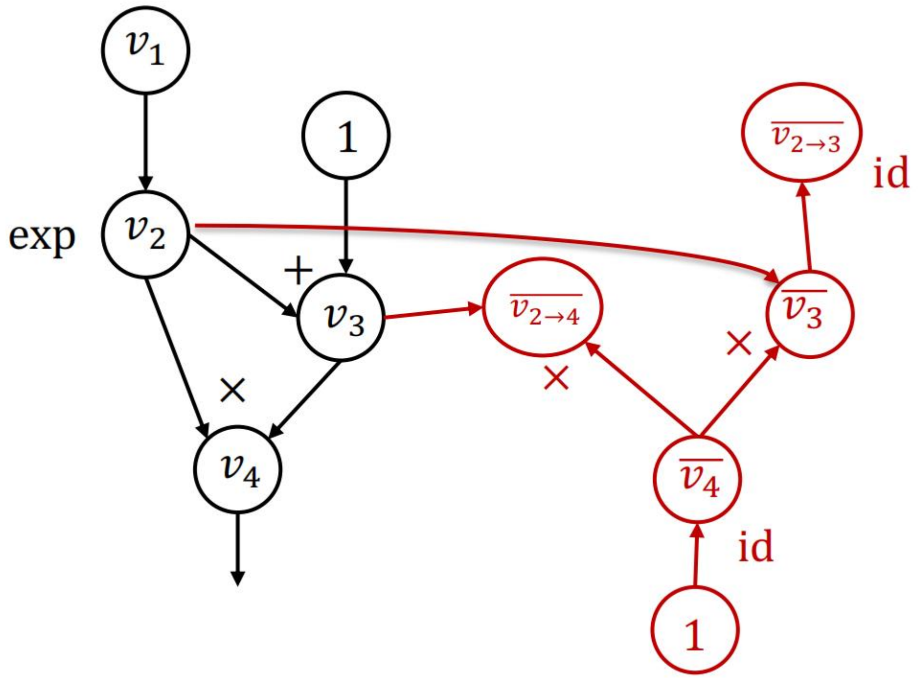
node_to_grad:{
  out:[1]
  4:[$\overline{v_4} = 1$]
  3:[$\overline{v_3} = \overline{v_{3\to 4}} = v_2$]
  2:[$\overline{v_{2\to 4}} = v_3$]
}



反向求导的扩展计算图

```
def gradient(out):
  node_to_grad = {out: [1]}
  for i in reverse_topo_order(out):  # i=3
    v̄_i = Σ_{j∈out(i)} v̄_{i→j} = sum(node_to_grad[i])
    for k ∈ in(i) :
      compute v̄_{k→i} = v̄_i ∂v_i/∂v_k
      append v̄_{k→i} to node_to_grad[k]
return adjoint of input v̄_input
```

node_to_grad:{
  out:[1]
  4:[$\overline{v_4} = 1$]
  3:[$\overline{v_3} = v_2$]
  2:[$\overline{v_{2\to4}} = v_3$, $\overline{v_{2\to3}} = \overline{v_3} \times 1 = v_2$]
}



反向求导的扩展计算图

```
def gradient(out):
  node_to_grad = {out: [1]}
  for i in reverse_topo_order(out):  # i=2
```

$\overline{v_i} = \sum_{j \in out(i)} \overline{v_{i \to j}}$ = sum(node_to_grad[i])

for $k \in in(i)$:

compute $\overline{v_{k \to i}} = \overline{v_i} \frac{\partial v_i}{\partial v_k}$

append $\overline{v_{k \to i}}$ to node_to_grad[k]

return adjoint of input $\overline{v_{input}}$

node_to_grad:{

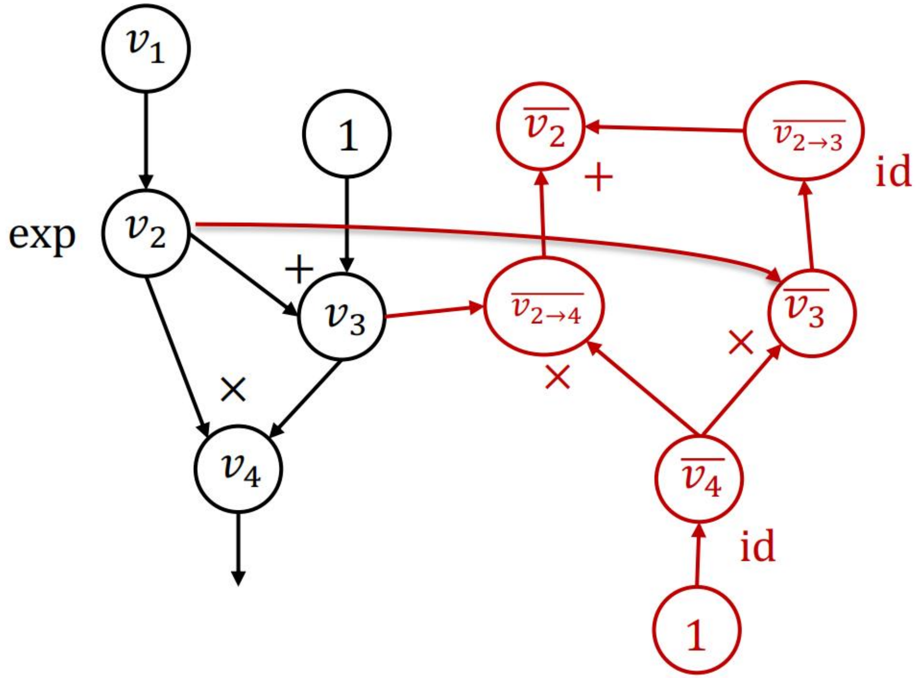
out:[1]

4:[$\overline{v_4} = 1$]

3:[$\overline{v_3} = v_2$]

2:[$\overline{v_2} = \overline{v_{2\to4}} + \overline{v_{2\to3}} = v_3 + v_2$]

}



反向求导的扩展计算图

```
def gradient(out):
  node_to_grad = {out: [1]}
  for i in reverse_topo_order(out):  # i=2
    v̄_i = Σ_{j∈out(i)} v̄_{i→j} = sum(node_to_grad[i])
    for k ∈ in(i) :
      compute v̄_{k→i} = v̄_i ∂v_i/∂v_k
      append v̄_{k→i} to node_to_grad[k]
return adjoint of input v̄_input
```

$\overline{v_i} = \sum_{j \in out(i)} \overline{v_{i \to j}} =$ sum(node_to_grad[i])

compute $\overline{v_{k \to i}} = \overline{v_i} \frac{\partial v_i}{\partial v_k}$

append $\overline{v_{k \to i}}$ to node_to_grad[k]

return adjoint of input $\overline{v_{input}}$

node_to_grad:{

out:[1]

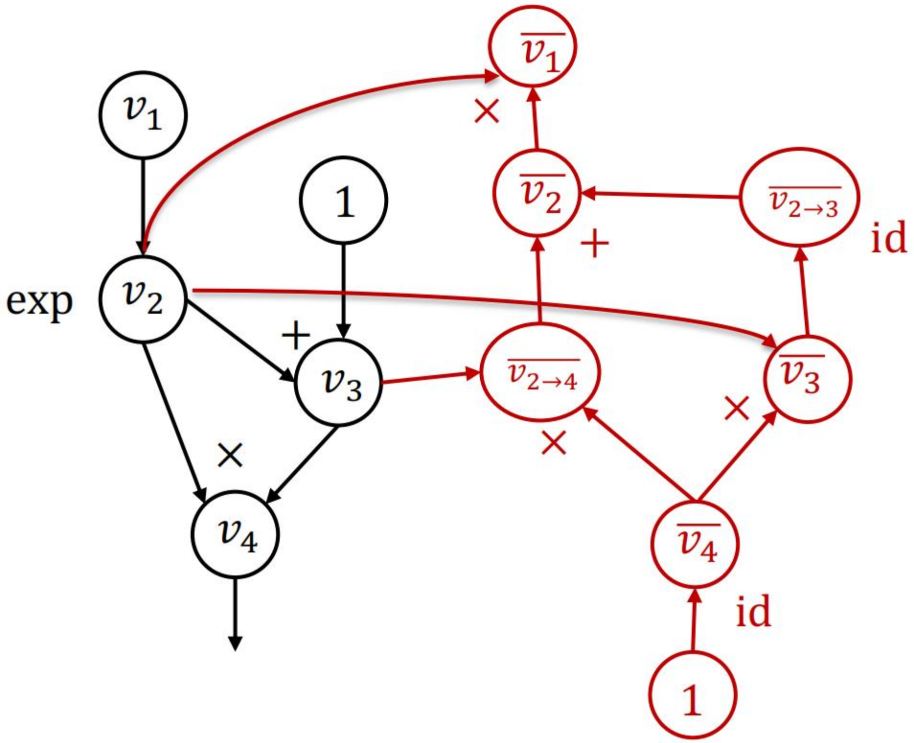
4:[$\overline{v_4} = 1$]

3:[$\overline{v_3} = v_2$]

2:[$\overline{v_2} = v_3 + v_2$]

1:[$\overline{v_{1 \to 2}} = \overline{v_2} \times v_2 = (v_3 + v_2) \times v_2$]

}



反向求导的扩展计算图

```
def gradient(out):
    node_to_grad = {out: [1]}
    for i in reverse_topo_order(out):  # i=1
        v̄_i = Σ_{j∈out(i)} v̄_{i→j} = sum(node_to_grad[i])
        for k ∈ in(i) :
            compute v̄_{k→i} = v̄_i ∂v_i/∂v_k
            append v̄_{k→i} to node_to_grad[k]
return adjoint of input v̄_input
```

node_to_grad:{

out:[1]

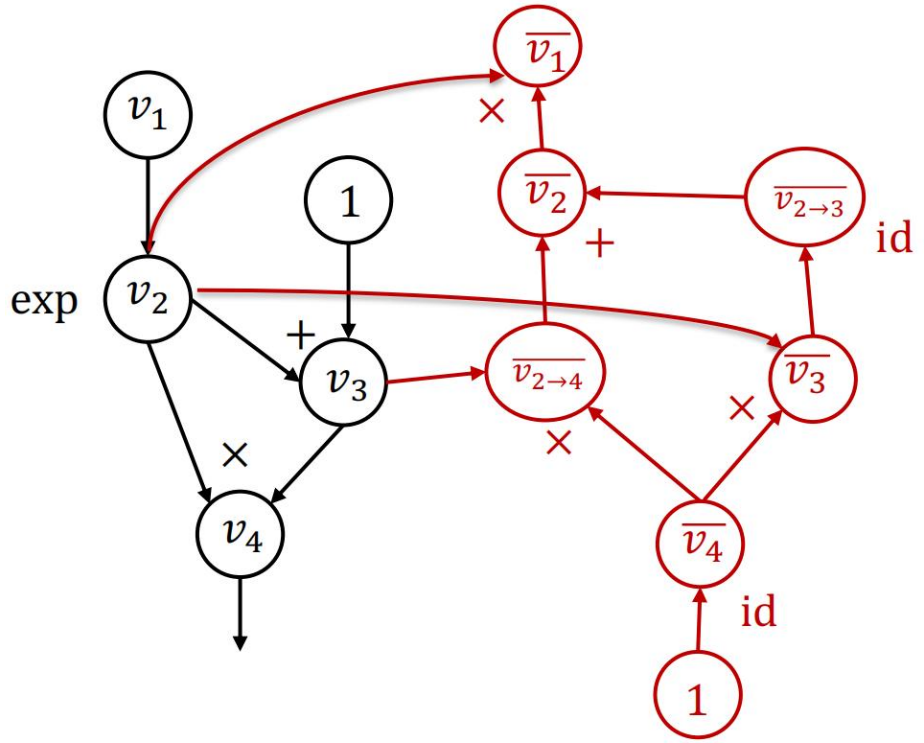
4:[$\overline{v_4} = 1$]

3:[$\overline{v_3} = v_2$]

2:[$\overline{v_2} = v_3 + v_2$]

1:[$\overline{v_1} = \overline{v_{1\to 2}} = (v_3 + v_2) \times v_2$]

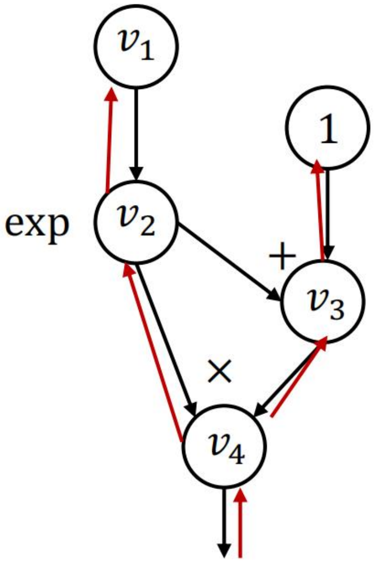
}

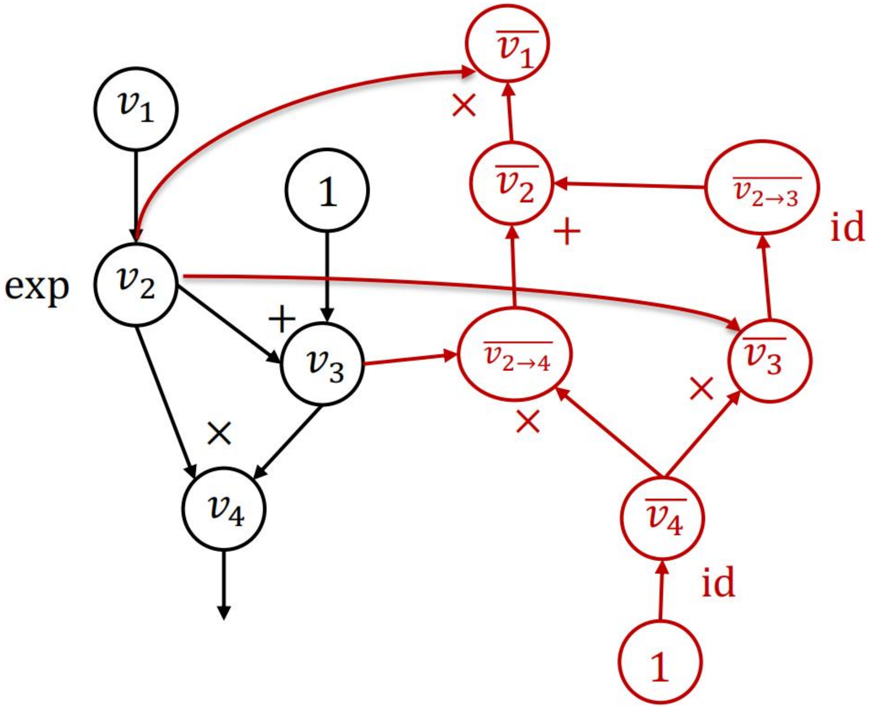


反向求导的扩展计算图

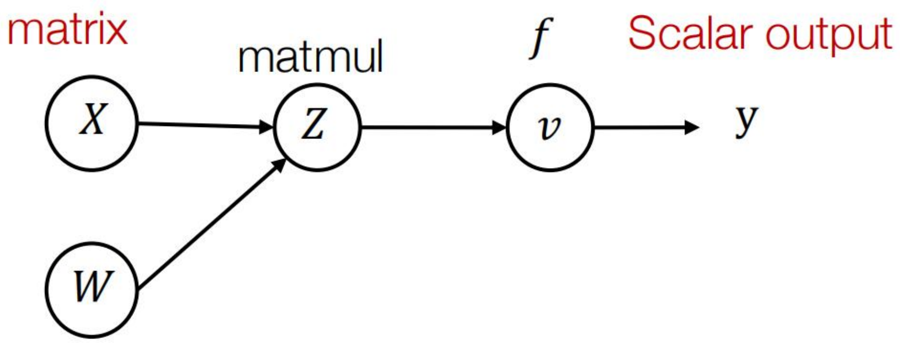
反向求导能够被建模为扩展计算图，也属于计算图的一种，可以用来进一步计算梯度的梯度。

早期框架（caffe等）采用反向传播，而现代框架（PyTorch, TensorFlow）主要使用扩展计算图。





张量的反向求导



$$Z_{ij} = \sum_k X_{ik} W_{kj}, \quad Z = XW$$

$$v = f(Z)$$

节点Z的伴随值:

$$\overline{Z} = \begin{pmatrix} \frac{\partial y}{\partial Z_{11}} & \cdots & \frac{\partial y}{\partial Z_{1n}} \\ \vdots & \ddots & \vdots \\ \frac{\partial y}{\partial Z_{m1}} & \cdots & \frac{\partial y}{\partial Z_{mn}} \end{pmatrix}$$

$$\overline{X_{ik}} = \sum_j \frac{\partial Z_{ij}}{\partial X_{ik}} \overline{Z_{ij}} = \sum_j W_{kj} \overline{Z_{ij}}$$

$$\overline{X} = \overline{Z} W$$

代码实现：计算图节点类

```
class Value:
    op: Optional[Op] # 节点对应的计算操作， Op是自定义的计算操作类
    inputs: List["Value"]
    cached_data: NDArray
    requires_grad: bool

    def realize_cached_data(self): # 进行计算得到节点对应的变量，存储在cached_data里
    def is_leaf(self):
    def _init(self, op: Optional[Op], inputs: List["Value"], *, num_outputs: int = 1,
cached_data: List[object] = None, requires_grad: Optional[bool] = None):

    @classmethod
    def make_const(cls, data, *, requires_grad=False): # 建立一个用data生成的独立节点
    def make_from_op(cls, op: Op, inputs: List["Value"]): # 用op和inputs计算，生成节点
```

代码实现：计算操作类

```
class Op(ABC):

    @abstractmethod
    def compute(self, *args: Tuple["NDArray"]) -> NDArray:
        # 前向计算. 参数args是由NDArray组成的序列Tuple，输出计算的结果NDArray
        pass

    @abstractmethod
    def gradient(self, out_grad: "Value", node: "Value") -> Tuple["Value"]:
        # 后向求导. 计算每个输入变量对应的局部伴随值(partial adjoint)
        # 参数out_grad是输出变量对应的伴随值，node是计算操作所在的计算图节点
        pass
```

代码实现：节点类成员函数

```
def Value.realize_cached_data(self):
    if self.cached_data is not None:
        return self.cached_data

    self.cached_data = self.op.compute(
        *[x.realize_cached_data() for x in self.inputs]
    )
    return self.cached_data

def Value.is_leaf(self):
    return self.op is None
```

代码实现：张量类

```
class Tensor (Value):
    grad: "Tensor"
    def __init__(self, array, *,
device:Optinal[Device]=None, dtype=None,
requires_grad=True, **kwargs):

    @staticmethod
    def _array_from_numpy(numpy_array, device,dtype):
    @staticmethod
    def make_from_op(op: Op, inputs: List["Value"]):
    @staticmethod
    def make_const(data, requires_grad=False):
```

```
@ property
def data (self):  #封装Value.cached_data
@ data.setter
def data (self, value):
@ property
def shape (self):
@ property
def dtype (self):

def backward (self, out_grad=None):
def detach (self):
def __add__ (self, other):
def __sub__ (self, other):
……
```

代码实现：张量类

```
def detach(self):
    # 创建一个新的张量，但不接入计算图
    return Tensor.make_const(self.realize_cached_data())

@staticmethod
def make_const(data, requires_grad=False):
    tensor = Tensor.__new__(Tensor)
    tensor._init(None, [], # 将前置节点置空
    cached_data = data if not isinstance(data, Tensor) else data.realize_cached_data()
    requires_grad = requires_grad)
    return tensor
```

代码实现：张量类

```
@staticmethod
def make_from_op(op: Op, inputs: List["Value"]):
    # 创建新张量并将其输入设为inputs，将新张量接入了计算图
    tensor = Tensor.__new__(Tensor)
    tensor._init(op, inputs)
    if not tensor.requires_grad:
        return tensor.detach()
    tensor.realize_cached_data()
    return tensor

class TensorOp(Op):
  # 继承计算操作类，以__call__方式进行计算操作的时候建立计算图节点
  def __call__(self, *args):
    return Tensor.make_from_op(self, args)
```

代码实现：张量计算

```
class EWiseAdd(TensorOp):          # 对应元素相加
class AddScalar(TensorOp):         # 加常数
class EWiseMul(TensorOp):          # 对应元素乘
class MulScalar(TensorOp):         # 乘常数
class PowerScalar(TensorOp):       # 常数幂
class EWiseDiv(TensorOp):          # 对应元素除
class DivScalar(TensorOp):         # 除以常数
class Transpose(TensorOp):         # 矩阵转置
class Reshape(TensorOp):           # 变形
class BroadcastTo(TensorOp):       # 广播
class Summation(TensorOp):         # 按维度求和
class MatMul(TensorOp):            # 矩阵相乘
class Negate(TensorOp):            # 求相反数
class Log(TensorOp):               # 求对数
class Exp(TensorOp):               # 求指数
```

代码实现：张量计算

```
class EWiseAdd(TensorOp):
    def compute(self, a: NDArray, b: NDArray):
        return a + b
    def gradient(self, out_grad: Tensor, node: Tensor):
        return out_grad, out_grad

class AddScalar(TensorOp):
    def __init__(self, scalar):
        self.scalar = scalar
    def compute(self, a: NDArray):
        return a + self.scalar
    def gradient(self, out_grad: Tensor, node: Tensor):
        return out_grad
```

```
def Tensor.__add__(self, other):
    if isinstance(other, Tensor):
        return needle.ops.EWiseAdd()(self, other)
    else:
        return needle.op.AddScalar(other)(self)
```

实现张量计算之后，在Tensor类中进行计算符重载

代码实现：自动求导

```
def Tensor.backward(self, out_grad=None):
    out_grad = (
        out_grad
        if out_grad
        else init.ones(*self.shape, dtype=self.dtype, device=self.device) # 最后一个节点时，out_grad为1
    )
    compute_gradient_of_variables(self, out_grad)
```

代码实现：自动求导

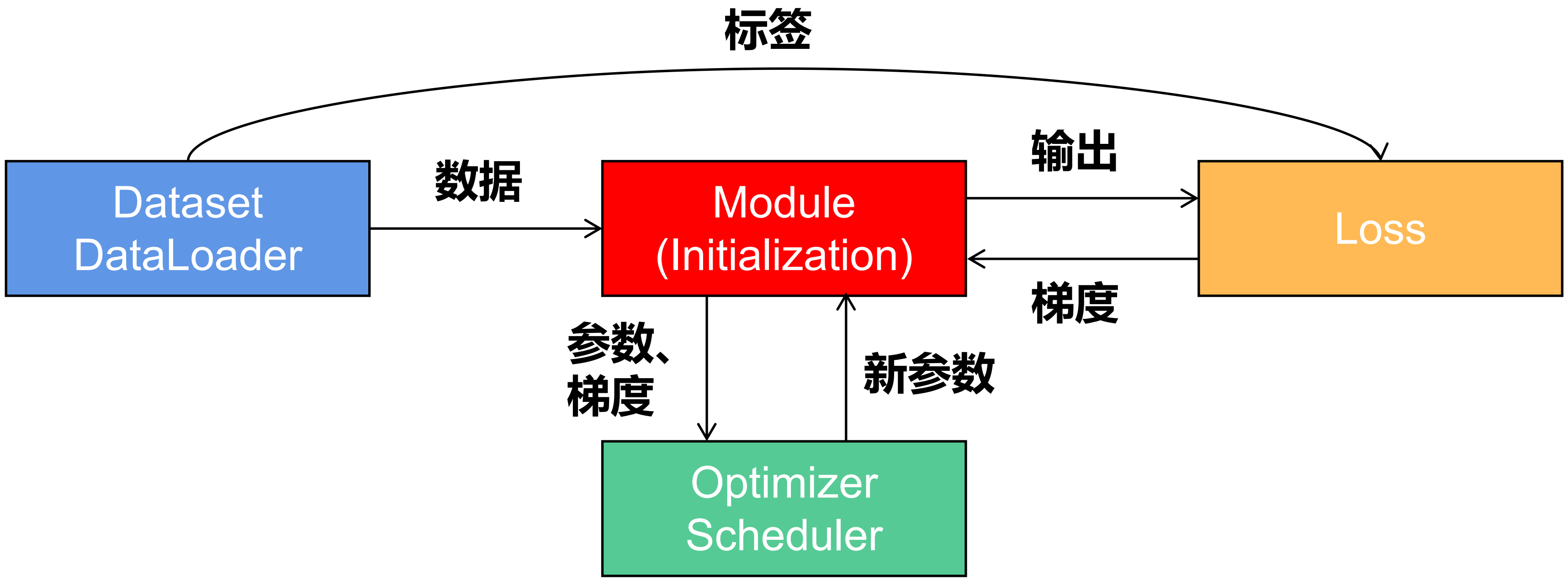
```
def compute_gradient_of_variables(output_tensor, out_grad):
    node_to_output_grads_list: Dict[Tensor, List[Tensor]] = {} # dict结构，用于存储partial adjoint
    node_to_output_grads_list[output_tensor] = [out_grad]
    reverse_topo_order = list(reversed(find_topo_sort([output_tensor]))) # 自行实现的拓扑排序函数
    for v in reverse_topo_order:
        node.grad = sum_node_list(node_to_output_grads_list[v]) # adjoint = sum(partial adjoint)
        if not v.op:
            continue
        adjoints = v.op.gradient_as_tuple(out_grad, v)
        for i, adj in enumerate(adjoints): # 计算node.inputs的partial adjoint
            k = v.inputs[i]
            if k not in node_to_output_grads_list:
                node_to_output_grads_list[k] = []
            node_to_output_grads_list[k].append(adj) # 将计算出的partial adjoint存入dict
    return node_to_output_grads_list
```

代码实现：自动求导

- 计算图在执行TensorOP的__call__函数时被隐式创建
- 每个节点只知道其输入节点，不知道输出节点
- 代码编写完成后，需结合数值微分方法进行测试

04 神经网络框架实现

神经网络编程框架：模块化



神经网络编程框架：模块化

Dataset & Data loader：加载数据集、划分mini-batch、数据预处理

Module：神经网络模块化结构

Initialization: 初始化网络参数

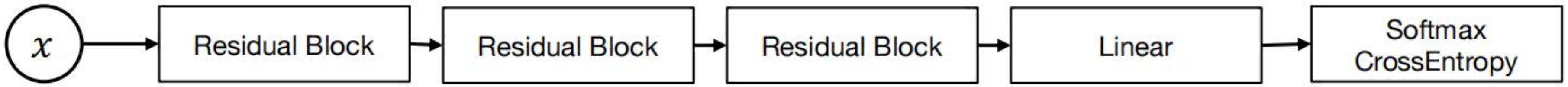
Loss：损失函数

Optimizer：接收 nn.Module 传入的模型参数与梯度，进行参数更新

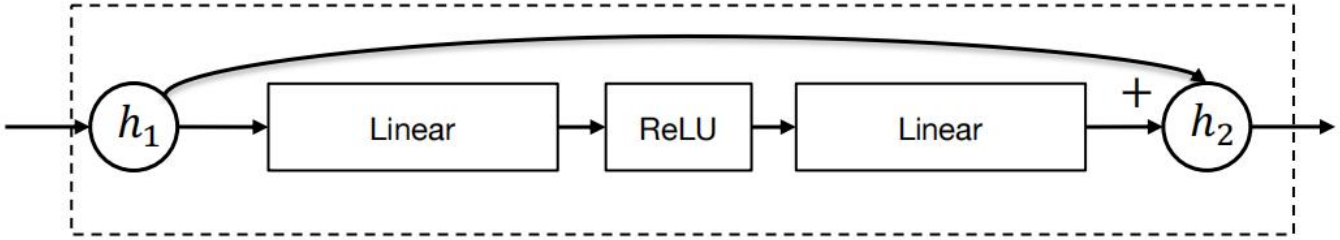
Scheduler: 规划学习率

神经网络编程框架：模块化

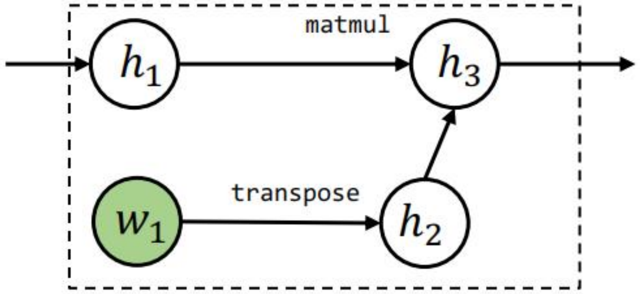
Multi-layer Residual Net



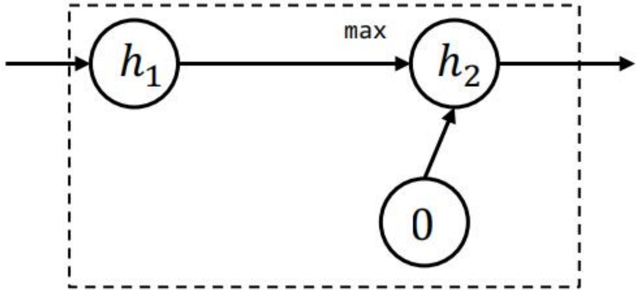
Residual block



Linear



ReLU



代码实现：神经网络模块

```python
class Module(ABC):
    def __init__(self):
        self.training = True

    def parameters(self) -> List["Tensor"]:
        return _unpack_params(self.__dict__)

    def _children(self) -> List["Module"]:
        return _child_modules(self.__dict__)

    def __call__(self, *args, **kwargs):
        return self.forward(*args, **kwargs)

    def eval(self):
        self.training = False
        for m in self._children():
            m.training = False

    def train(self):
        self.training = True
        for m in self._children():
            m.training = True
```

代码实现：神经网络模块

```
class Parameter(Tensor): # 声明一个类专门表示网络参数

def _unpack_params(value: object) -> List[Tensor]:
    if isinstance(value, Parameter): return [value]
    elif isinstance(value, Module): return value.parameters()
    elif isinstance(value, dict):
        params = []
        for k, v in value.items():
            params += _unpack_params(v)
        return params
    elif isinstance(value, (list, tuple)):
        params = []
        for v in value:
            params += _unpack_params(v)
        return params
    else: return []
```

代码实现：神经网络模块

```
def _child_modules(value: object) -> List["Module"]:
    if isinstance(value, Module):
        modules = [value]
        modules.extend(_child_modules(value.__dict__))
        return modules
    if isinstance(value, dict):
        modules = []
        for k, v in value.items():
            modules += _child_modules(v)
        return modules
    elif isinstance(value, (list, tuple)):
        modules = []
        for v in value:
            modules += _child_modules(v)
        return modules
    else:
        return []
```

代码实现：神经网络模块

```
class Linear(Module)                      # 线性层
class Flatten(Module)                     # 平铺层
class ReLU(Module)                        # ReLU激活函数
class Sigmoid(Module)                     # Sigmoid激活函数
class Softmax(Module)                     # Softmax层
class CrossEntrophyLoss(Module)           # 交叉熵损失
class BinaryCrossEntrophyLoss(Module)     # 二元交叉熵损失
class MSELoss(Module)                     # 均方损失
class BatchNorm1d(Module)                 # 一维批归一化 （选做）
class LayerNorm1d(Module)                 # 一维层归一化 （选做）
class DropOut(Module)                     # Dropout层 （选做）
class Sequential(Module)                  # 多层模型
class Residual(Module)                    # 残差连接
```

代码实现：神经网络模块

```
class Linear(Module):
    def __init__(self, in_features, out_features, bias=True, dtype="float32"):
        super().__init__()
        self.in_features = in_features
        self.out_features = out_features
        self.weight = ... #初始化算法
        self.bias = ...

    def forward(self, X: Tensor) -> Tensor:
        y = X @ self.weight
        if self.bias:
            return y + ops.broadcast_to(self.bias, (*X.shape[:-1], self.out_features))
        return y
```

代码实现：神经网络模块

```
class Sequential(Module):
    def __init__(self, *modules):
        super().__init__()
        self.modules = modules

    def forward(self, x: Tensor) -> Tensor:
        for module in self.modules:
            x = module(x)
        return x
```

代码实现：优化器与学习率规划器

```
class SGD(Optimizer)                          # 基本随机梯度下降
class SGD_WeightDecay(Optimizer)              # L2 (L1) 正则化
class Momentum(Optimizer)                     # 动量法 （Nestrov），以上三项可合并
class Adam(Optimizer)                         # Adam
class StepDecay(Scheduler)                    # 阶梯型衰减
class LinearWarmUp(Scheduler)                 # 线性热启动
class CosineDecayWithWarmRestarts(Scheduler) # 带热重启的Cosine衰减
```

代码实现：优化器

```
class Optimizer(ABC):
    def __init__(self, params):
        self.params = params

    @abstractmethod
    def step(self):
        raise NotImplementedError()

    def reset_grad(self):
        for p in self.params:
            p.grad = None
```

代码实现：优化器

```
class SGD(Optimizer): # 基本随机梯度下降
    def __init__(self, params, lr = 0.001):
        super().__init__(params)
        self.lr = lr

    def step(self):
        for i, param in enumerate(self.params):
            grad = Tensor(param.grad, dtype='float32').data
            param.data = param.data - grad * self.lr
```

代码实现：规划器

```
class Scheduler(ABC):
    def __init__(self, optimizer):
        self.optimizer= optimizer

    @abstractmethod
    def step(self):
        raise NotImplementedError()

    def get_lr(self):
        return self.optimizer.lr
```

代码实现：规划器

```
class StepDecay(Scheduler):
    def __init__(self, optimizer, step_size, gamma):
        super().__init__(optimizer)
        self.step_size = step_size
        self.gamma = gamma
        self.last_epoch = 0

    def step(self):
        self.last_epoch += 1
        if self.last_epoch % self.step_size == 0:
            self.optimizer.lr *= self.gamma
```

大作业：

1. 实现ppt中列举的TensorOp子类并测试；

2. 实现ppt中列举的Module、Optimizer、Scheduler子类；

3. 使用简易框架在MNIST\CIFAR-10数据集上进行ResMLP实验，实现模型存储和读取，测试阶段不计算梯度。

可以直接使用开源代码，但需完成测试

提交代码和实验报告，但不包含实验数据

DDL: 2025-6-30