

目录

1	自己编写神经网络库 ANNbox【高仿 tensorflow】，框架结构简介，并实现 MLP 对 MNIST 数据进行分类.....	1
1.1	抽象节点类---Node.....	1
1.2	Node 的子类.....	1
1.2.1	placeholder(占位符节点,功能仿造 tensorflow 中的 placeholder,用于存放输入数据和标签数据)	2
1.2.2	variable(变量节点,功能仿造 tensorflow 中的 variable,用于存放输入权重系数及偏执系数)	2
1.2.3	Add 子类(对 Input 实例进行加法运算)	2
1.3	前向传播.....	2
1.3.1	topological_sort(), 拓扑排序方法: Kahn 算法	2
1.3.2	forward_pass: 按拓扑排序前向计算节点 value	5
1.3.3	简单前向计算实现算例.....	5
1.4	线性加权类 Linear	6
1.4.1	简单算例.....	6
1.5	sigmoid 激活函数类.....	6
1.6	损失函数.....	7
1.6.1	简单算例.....	8
1.7	反向传播.....	8
1.7.1	为抽象节点类 Node 添加 backward 方法	8
1.7.2	为 placeholder 添加 backward 方法	8
1.7.3	为 variable 添加 backward 方法	8
1.7.4	为 Linear 添加 backward 方法.....	9
1.7.5	为 Sigmoid 添加 backward 方法	10
1.7.6	为 MSE 添加 backward 方法	11
1.7.7	为 forward_pass 添加 backward 计算功能	11
1.8	参数更新_小批量梯度下降法.....	11
1.9	算例实现-MLP 实现 MNIST 数据分类.....	12
2	自己编写神经网络库 ANNbox【高仿 tensorflow】__01 实现 MLP.....	15

2.1	线性加权输入类 Linear	16
2.1.1	输出层权重系数矩阵初始化为 0 的情况.....	17
2.2	矩阵相乘类 matmul	18
2.3	矩阵相加类 Add 【用于运算符重载】	20
2.4	矩阵相减类 Sub 【用于运算符重载】	22
2.5	平方运算 Squared 类.....	24
2.6	reduce_mean 类【这里只给出了 reduce_mean 后结果维度为 1*1 的情况】	25
2.7	Node 类添加运算符重载.....	26
2.7.1	加号.....	26
2.7.2	减号.....	26
2.8	激活函数类.....	27
2.8.1	激活函数为 sigmoid.....	27
2.8.2	激活函数为 ReLU	28
2.9	哪些项是用来传递梯度项的.....	28
2.10	损失函数类.....	29
2.10.1	softmax_cross_entropy_with_logits 损失函数类	29
2.10.2	均方误差损失函数类.....	33
2.11	编程实现.....	34
2.11.1	验证算例.....	34
2.11.2	算例 1.....	35
2.11.3	算例 2.....	38
3	自己编写神经网络库 ANNbox 【高仿 tensorflow】_02 实现不同的优化方法	41
3.1	实现 tf.train.GradientDescentOptimizer 类__SGD.....	41
3.1.1	仿真算例.....	42
3.2	实现 tf.train.MomentumOptimizer 类__SGD with Momentum(SGDM).....	43
3.2.1	仿真算例.....	44
3.3	NAG(Nesterov Accelerated Gradient).....	46
3.4	实现 tf.train.AdagradOptimizer 类__AdaGrad	47
3.5	实现 tf.train.AdadeltaOptimizer 类__Adaptive Delta	48
4	自己编写神经网络库 ANNbox 【高仿 tensorflow】_03 文本情感分析	49
4.1.1	方法 1：基于词汇计数的方法实现输入语句数据向量化.....	49

4.1.2	方法 2: 提升准确率-减少输入噪声	52
4.1.3	方法 3: 提速并提升准确率—减少输入向量中的中性词及出现次数较少的单词	57
4.1.4	寻找意思最相似的词	61
5	自己编写神经网络库 ANNbox【高仿 tensorflow】_04 实现卷积神经网络, 并用简化版本的 AlexNet(无 pool, 无 dropout)	62
5.1	conv2d 类	62
5.1.1	对于 padding='same' 的模式, 有	62
5.1.2	对于 padding='valid' 的模式, 有	65
5.1.3	对应的编程实现如下:	67
5.2	bias_add 类	71
5.3	max_pool 类	73
5.3.1	对于 padding='same' 的模式, 有	73
5.3.2	对于 padding='valid' 的模式, 有	74
5.4	avg_pool 类	74
5.4.1	对于 padding='same' 的模式, 有	74
5.4.2	对于 padding='valid' 的模式, 有	75
5.5	droupout 类	76
5.6	编程实现	76
5.6.1	验证算例	76
6	自己编写神经网络库 ANNbox【高仿 tensorflow】_05 实现 RNN (01)	79
6.1	variable_scope 类 (利用上下文管理器实现变量的作用域限定)	79
6.2	get_variable 类 (创建共享变量)	79
6.3	concat 类 (多个张量按指定维度合并为一个张量)	82
6.4	unstack 类 (一个 R 维张量按指定维度拆分为多个 R-1 维张量)	84
6.5	transmit_node 类 (用于 unstack 分解操作的中间暂存变量, 不放入 feed_dict_inside 【非常重要】)	86
6.6	验证算例	86

1 自己编写神经网络库 ANNbox 【高仿 tensorflow】，框架结构简介，并实现 MLP 对 MNIST 数据进行分类

本节将介绍如何开发一个小型的神经网络库 ANNbox，通过这个过程的学习，我们会理解方向图及反向传播的机理。

1.1 抽象节点类---Node

首先建立一个 Node 类来抽象表示所有普通节点。将来我们将以这个节点为基类（或者抽象类），扩展出所有的操作类，类似于 tensorflow 中所有的操作，比如 sigmoid/relu/softmax_cross_entropy_with_logits/GradientDescentOptimizer/MomentumOptimizer 等。不难知道，神经网络是通过节点前后项链雕刻，每个节点都有输入节点及输出节点，在 Node 类的构造函数中添加了两个列表：inbound_nodes 表示对用于存储传入节点的列表的引用；outbound_nodes 表示对用于存储输出节点的列表的引用。虽然输出节点个数可能很多，但每个节点的输出值只有一个，用 value 表示，初始化为 None，表示该值存在，但具体值尚未设定。

对神经网络了解的话就不难知道，每个节点都应该有前向计算和反向计算这两个过程。这里暂时为这两个过程添加一个占位符方法。

```
class Node(object):
```

```
    """Node 类表示普通节点，定义了每个节点都具有的基本属性"""
```

```
    def __init__(self, inbound_nodes=[]):
```

```
        """
```

```
        每个节点可以从其他多个节点那接收输入          --> 添加列表：用于存储对传入节点的引用
```

```
        每个节点都会有一个输出【用于传递给其他多个节点】 --> 添加列表：用于存储对传出节点的引用。
```

```
        每个节点将最终计算出一个表示输出的值。我们将 value 初始化为 None，表示该值存在，但是尚未设定。
```

```
        """
```

```
        #从构造函数的形参中获得当前节点的传入节点
```

```
        self.inbound_nodes= inbound_nodes
```

```
        #设定当前节点的输出节点，输出节点为空列表[]，表示该值存在，但具体节点有哪些还没确定
```

```
        #具体的输出节点会在 其输出节点的初始化函数中 被添加到输出节点列表中， 实现如下
```

```
        self.outbound_nodes = []
```

```
        # 对每个传入节点的属性 outbound_nodes 添加当前节点，表示当前节点是传入节点的输出节点之一
```

```
        for n in self.inbound_nodes:
```

```
            n.outbound_nodes.append(self)
```

```
        # A calculated value
```

```
        self.value = None
```

```
    def forward(self):
```

```
        """
```

```
        前向传播函数【如果没有实现，称为占位符方法】
```

```
        """
```

```
    #         return NotImplemented
```

```
        raise NotImplemented
```

```
    def backward(self):
```

```
        """
```

```
        反向传播函数【如果没有实现，称为占位符方法，子类如果调用该函数的前提是必须实现它】
```

```
        """
```

```
        raise NotImplementedError
```

1.2 Node 的子类

Node 类是一个抽象类，定义了每个节点都具有的基本属性，但是只有 Node 的特殊子类会出现在图表中。下面将构建可进行计算和存储值的 Node 子类。例如，考虑 Node 的 Input 子类。

1.2.1 placeholder（占位符节点,功能仿造 tensorflow 中的 placeholder,用于存放输入数据和标签数据）

```
class placeholder(Node):
```

```
    """占位符节点,功能仿造 tensorflow 中的 placeholder,用于存放输入数据"""
```

```
    def __init__(self,name=[]):
```

```
        """输入层的节点没有传入节点(inbound nodes),所有无需向构造函数中传递任何 inbound nodes"""
```

```
        Node.__init__(self,name=name)
```

```
        feed_dict_inside[self] = None    #在字典 feed_dict_inside 中添加占位符节点的键值,并初始化为 None
```

```
    def forward(self, value=None):
```

```
        if value is not None:
```

```
            self.value = value
```

成员变量 value 可以被明确地设置，也可以用 forward() 方法进行设置。该值然后会传递给神经网络的其他节点。

1.2.2 variable（变量节点,功能仿造 tensorflow 中的 variable,用于存放输入权重系数及偏执系数）

```
class variable(Node):
```

```
    """变量节点,功能仿造 tensorflow 中的 variable,用于存放输入权重系数及偏执系数"""
```

```
    def __init__(self, value, name=[]):
```

```
        """输入层的节点没有传入节点(inbound nodes),所有无需向构造函数中传递任何 inbound nodes"""
```

```
        self.value = value
```

```
        Node.__init__(self,name=name,value=value)
```

```
        L.append(self)    #将 variable 存入节点列表 L 中
```

```
        trainables.append(self) #将 variable 添加进 trainables 列表中
```

```
    def forward(self, value=None):
```

```
        if value is not None:
```

```
            self.value = value
```

成员变量 value 可以被明确地设置，也可以用 forward() 方法进行设置。该值然后会传递给神经网络的其他节点。

1.2.3 Add 子类（对 Input 实例进行加法运算）

Add 是对多个输入节点进行相加，因此 Add 类必须有传入节点，且传入节点需要封装成列表传入基类 Node 的构造函数中。

```
class Add(Node):
```

```
    """add 是 Node 的另一个子类，实际上可以进行计算（加法）。"""
```

```
    def __init__(self, x, y):
```

```
        """Input 类没有传入节点，而 Add 类具有 2 个传入节点 x 和 y，并将这两个节点的值相加。"""
```

```
        Node.__init__(self, [x, y]) # 调用 Node 的构造函数
```

```
    def forward(self):
```

```
        x_value = self.inbound_nodes[0].value
```

```
        y_value = self.inbound_nodes[1].value
```

```
        self.value = x_value + y_value
```

1.3 前向传播

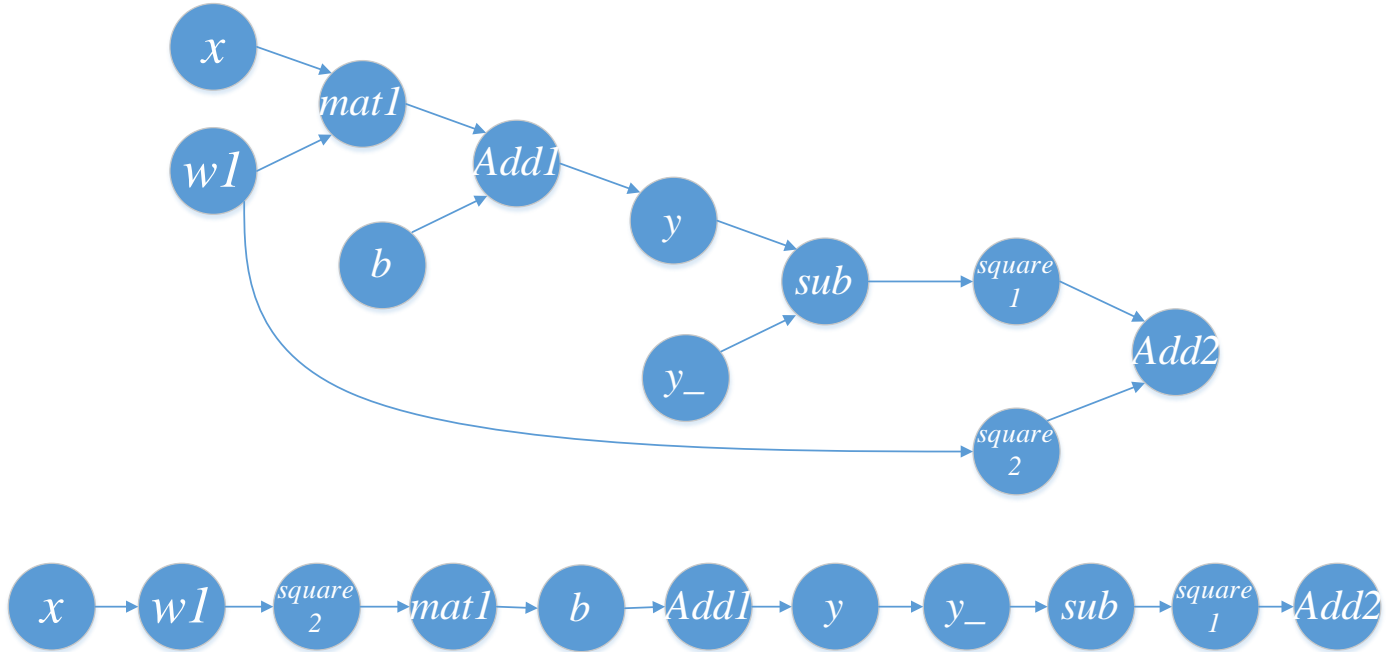
定义好节点后，需要定义节点的操作顺序。因为部分节点的输入取决于其他节点的输出，需要按拓扑排序的方法扁平化图表（示例如下）。

1.3.1 topological_sort()，拓扑排序方法：Kahn 算法

如图中有向图所示，箭头的起点（前驱）必须在箭头终点（后继）之前完成，这种有向图称为活动网络，记做 AOV 网络。不含有向回路的有向图称为有向无环图。将有向无环图全部顶点排列成一个线性有序的序列、使得 AOV

网络中所有存在前驱和后继关系都得到满足的过程，称为**拓扑排序**。AOV 网络的拓扑有序序列可能不唯一。常用的 Kahn 算法拓扑排序算法为 Kahn 算法：

- (1) 从有向图中选择一个没有前驱(即入度为 0)的顶点并且输出它；
- (2) 从网中删去该顶点,并且删去从该顶点发出的全部有向边；
- (3) 重复上述两步,直到剩余的网中不再存在没有前趋的顶点为止。



程序中 `topological_sort()` 函数用 Kahn 算法实现拓扑排序。该方法返回一个排好序的节点列表，所有计算都可排列列表进行。

```
def topological_sort(feed_dict, L_initial_with_variable_and_constant):
    """
    用 Kahn 算法进行拓扑排序
    (1) 从有向图中选择一个没有前驱(即入度为 0)的顶点并且输出它；
    (2) 从网中删去该顶点,并且删去从该顶点发出的全部有向边；
    (3) 重复上述两步,直到剩余的网中不再存在没有前趋的顶点为止。
    """
    input_nodes = [n for n in feed_dict.keys()]

    G = {} #用来存放节点输入与输出关系的字典
    nodes = [n for n in input_nodes] + [n for n in L_initial_with_variable_and_constant] #
    将输入节点 (placeholder) 及变量和常量 (variable 和 constant) 作为初始节点，原来只有输入节点，结果
    正则化项未能加入到计算图中

    #下面这个循环用于计算所有节点的输入输出关系，即完善 G
    while len(nodes) > 0:
        n = nodes.pop(0)
        if n not in G:
            G[n] = {'in': set(), 'out': set()}
        for m in n.outbound_nodes:
            if m not in G:
                G[m] = {'in': set(), 'out': set()}
            G[n]['out'].add(m)
```

自己编写神经网络库 ANNbox 【高仿 tensorflow】，框架结构简介，并实现 MLP 对 MNIST 数据进行分类

```

G[m]['in'].add(n)
nodes.append(m)
for m in n.inbound_nodes:
    if m not in G:
        G[m] = {'in': set(), 'out': set()}
    if m not in G[n]['in']:
        G[n]['in'].add(m)
    if n not in G[m]['out']:
        G[m]['out'].add(n)

#开始获取计算图
L = []
S = set(input_nodes + L_initial_with_variable_and_constant)
while len(S) > 0:
    n = S.pop()
    if isinstance(n, placeholder):
        n.value = feed_dict[n]
    if len(G[n]['in']) == 0:
        L.append(n)
    else:
        S.add(n)
    for m in n.outbound_nodes:
        G[n]['out'].remove(m)
        G[m]['in'].remove(n)
        if len(G[m]['in']) == 0:
            S.add(m)
return L

```

```
def topological_sort(feed_dict):
```

```
    """
```

用 Kahn 算法进行拓扑排序

- (1) 从有向图中选择一个没有前驱(即入度为 0)的顶点并且输出它；
- (2) 从网中删去该顶点,并且删去从该顶点发出的全部有向边；
- (3) 重复上述两步,直到剩余的网中不再存在没有前趋的顶点为止。

```
    """
```

```
input_nodes = [n for n in feed_dict.keys()]
```

```
G = {}
```

```
nodes = [n for n in input_nodes]
```

```
while len(nodes) > 0:
```

```
    n = nodes.pop(0)
```

```
    if n not in G:
```

```
        G[n] = {'in': set(), 'out': set()}
```

```

for m in n.outbound_nodes:
    if m not in G:
        G[m] = {'in': set(), 'out': set()}
    G[n]['out'].add(m)
    G[m]['in'].add(n)
    nodes.append(m)

L = []
S = set(input_nodes)
while len(S) > 0:
    n = S.pop()

    if isinstance(n, Input):
        n.value = feed_dict[n]

    L.append(n)
    for m in n.outbound_nodes:
        G[n]['out'].remove(m)
        G[m]['in'].remove(n)
        # if no other incoming edges add to S
        if len(G[m]['in']) == 0:
            S.add(m)

return L

```

`topological_sort()` 传入 `feed_dict`，我们按此方法为 `Input` 节点设置初始值。
`feed_dict` 由 Python 字典数据结构表示。

1.3.2 forward_pass: 按拓扑排序前向计算节点 value

该方法按拓扑排序计算网络中每个节点的值，并返回输出节点的 value 值

```

def forward_pass(output_node, sorted_nodes):
    for n in sorted_nodes:
        n.forward()
    output_node.forward()
    return output_node.value

```

1.3.3 简单前向计算实现算例

```

# -*- coding: utf-8 -*-
from miniFlow import *
x, y = placeholder(), placeholder()
f = Add(x, y)
feed_dict = {x: 10, y: 20}
global_variables_initializer()
sorted_nodes = topological_sort (feed_dict=feed_dict)
output = forward_pass(f,sorted_nodes)
print("{}+{}={} (according to miniflow)".format(feed_dict[x], feed_dict[y], output))

```

先用 `x, y = Input(), Input()` 构建图表中的输入节点，用 `f = Add(x,y)` 构建了对输入节点相加的节点，但此时的输入节点 `x` 和 `y` 的 value 还没有被初始化。然后用 `topological_sort` 对所有节点进行拓扑排序，并用形参 `feed_dict` 向图表中灌入数据（各个节点中的 value 值是在这个函数里依据 `feed_dict` 进行赋值的），最后再用 `forward_pass` 按节点排序顺序依次计算各个节点中的 `forward` 函数，并返回输出节点的 value。

1.4 线性加权类 Linear

首先是线性加权输入类，主要实现下式的运算操作【公式中参数详细的说明请参考深度学习基础模型算法原理及编程实现-03.全链接，全文同】

$$net_m^k = \sum_{i=1}^{N_k} x_i^k w_{im}^{k+1} + b_m^{k+1} \quad (1.1)$$

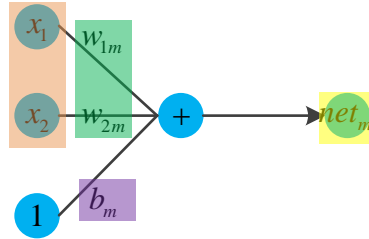
为了方便编程实现，将上式整理成矩阵运算的形式，有：

$$[net^{k+1}]_{batchsize \times N^{k+1}} = [x^k]_{batchsize \times N^k} [w^{k+1}]_{N^k \times N^{k+1}} + [b^{k+1}]_{1 \times N^{k+1}} \quad (1.2)$$

忽略层数 k 的影响，上式可整理为：

$$net_m = \sum_{i=1}^N x_i w_{im} + b_m \quad (1.3)$$

下面再通过简单的图示对线性加权进行说明：



相应的代码实现如下：

```
class Linear(Node):
```

```
    """线性加权神经元类，实现节点加权输入"""
```

```
    def __init__(self, X, W, b, name=[]):
```

```
        Node.__init__(self, inbound_nodes=[X,W,b], name=name)
```

```
    def forward(self):
```

```
        X = self.inbound_nodes[0].value
```

```
        W = self.inbound_nodes[1].value
```

```
        b = self.inbound_nodes[2].value
```

```
        self.value = np.dot(X, W) + b
```

1.4.1 简单算例

```
# -*- coding: utf-8 -*-
```

```
import numpy as np
```

```
from miniFlow import *
```

```
inputs = variable(value=np.array([1,2]),name='X')
```

```
weight, bias = variable(value=np.array([[1,2,3],[4,5,6]]), name='W1'), variable(value=np.array([1,2,3]), name='b1')
```

```
L1 = Linear(inputs, weight, bias, name = 'L1')
```

```
L1.forward()
```

```
print(L1.value)
```

1.5 sigmoid 激活函数类

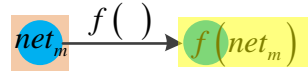
激活函数主要实现下式中的运算：

$$y_m^k = f(net_m^k) \quad (1.4)$$

忽略层数 k 的影响，上式可整理为：

$$y_m = f(\text{net}_m) \quad (1.5)$$

下面再通过简单的图示对激活函数对输入的变换进行说明：



相应的实现代码如下：

```
class Sigmoid(Node):
    """sigmoid 激活函数节点类型，实现对节点加输入的非线性变换"""
    def __init__(self, node, name=[]):
        Node.__init__(self, inbound_nodes=[node], name=name)

    def _sigmoid(self, x):
        return (1./(1+np.exp(-x)))

    def forward(self):
        self.value=self._sigmoid(self.inbound_nodes[0].value)
```

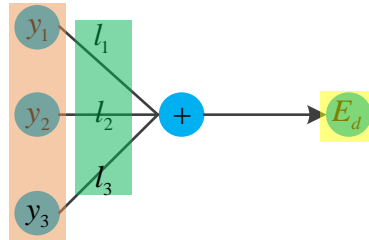
1.6 损失函数

前面使用了权重和偏置来计算加权输出，并用激活函数来学习高阶特征。但这只是神经网络计算中的一部分，大家知道，神经网络主要是通过修改权重和偏置（根据标签化的数据集进行训练）改善输出的精确度。精确度有多种衡量方式，都是为了衡量神经网络输出是否与已知正确的值接近。精确度通常称之为损失或代价。这里用均方差（MSE）来计算损失。

均方误差损失函数主要实现下式中的运算：

$$E_d = \frac{1}{2} \sum_{i=1}^{N_K} (l_i - y_i^K)^2 = \frac{1}{2} (\|L - Y^K\|_2)^2 \quad (1.6)$$

下面再通过简单的图示对激活函数对输入的变换进行说明：



相应的实现代码如下：

```
class MSE(Node):
    def __init__(self, labels, logits, name = []):
        """均方误差损失函数"""
        Node.__init__(self, inbound_nodes=[labels, logits], name=name)

    def forward(self):
        labels = self.inbound_nodes[0].value
        logits = self.inbound_nodes[1].value
        self.m = self.inbound_nodes[0].value.shape[0]
        self.diff = labels - logits
        # self.value = np.mean(self.diff**2,1)
        self.value = np.sum(self.diff**2,1)
```

1.6.1 简单算例

```
# -*- coding: utf-8 -*-
from miniFlow import *
y, a = Input(), Input()
cost = MSE(y,a)
y_ = np.array([1,2,3])
a_ = np.array([4.5,5,10])
feed_dict = {y:y_, a:a_}
graph = topological_sort(feed_dict)
output = forward_pass(cost, graph)
print(output)
print(cost.value)
结果:
70.25
70.25
```

1.7 反向传播

神经网络训练的目标就是调整参数使得神经网络的输出逼近标签数据，或者说就是减小损失函数，常用的方法就是梯度下降，梯度实际上是指上坡的方向，梯度下降就是指最陡的下降的方向。学习率就是控制梯度下降步伐的。为了实现反向传播，原来程序也需要做相应的修改。需要为相关类添加 **backward** 方法，并且添加了新的属性 **self.gradients**，用于在反向传递过程中存储和缓存梯度。

1.7.1 为抽象节点类 Node 添加 backward 方法

Node 类中将 **backward** 方法定义为占位符方法

```
class Node(object):
    .....
    def backward(self):
        """
        反向传播函数【如果没有实现，称为占位符方法，子类如果调用该函数的前提是必须实现它】
        """
        raise NotImplementedError
```

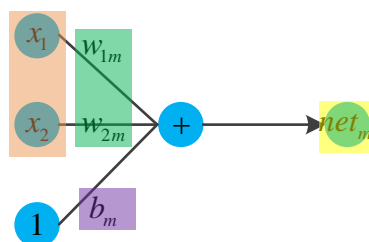
1.7.2 为 placeholder 添加 backward 方法

由于 **placeholder** 存放的是输入数据和标签数据，因此无需对其进行更新。所以其 **backward** 方法的函数体为 **pass**

```
class placeholder(Node):
    .....
    def backward(self):
        pass
```

1.7.3 为 variable 添加 backward 方法

由于 **Variable** 中存放的是权重系数和偏执系数，而反向传播算法就是通过迭代来逐步更新权重及偏执系数，因此需对其梯度进行计算。但是单由 **variable** 类中的信息是无法获得权重及偏执系数偏导数的详细信息的。在计算过程中，单个 **variable** 属于线性加权类【线性加权类结构示意图如下图所示，在下面的线性加权类中给出了损失函数对权重系数的偏导数】，因此 **variable** 类中的梯度项需访问线性加权类中的梯度项得到。



```
class variable(Node):
```

```

.....
def backward(self):
    """variable 类中的梯度项需访问线性加权类中的梯度项得到"""
#     self.gradients={self:0}
    self.gradients={self:np.zeros_like(self.value)}
    for n in self.outbound_nodes:
        grad_cost=n.gradients[self]
        self.gradients[self]+=grad_cost*1

```

1.7.4 为 Linear 添加 backward 方法

由前文知，Linear 类中同样有 inbound_nodes 和 outbound_nodes，详细如下图所示，为了方便对比，其中 inbound_nodes 和 outbound_nodes 中不同项所对应的方框颜色与上图中的方框颜色一致。且正如下图所示，outbound_nodes 的导数不要求，因为在当前类中没有可以用来求其导数的信息，并且当前节点的 outbound_nodes 是下一个节点的 inbound_nodes，所以 outbound_nodes 的导数在下一个节点中被计算。

此外，inbound_nodes 的导数被表示为当前类中的输入项与损失函数对输出项偏导数【如下图所示， $\frac{\partial E_d}{\partial net_j^k}$ 】的

乘积，其中当前类中的输入项是已知的，而 $\frac{\partial E_d}{\partial net_j^k}$ 并不知，但 $\frac{\partial E_d}{\partial net_j^k}$ 具体的值可通过访问下一个神经元对输入项的偏导数得到。

Linear 类中的输入项有 3 个，对不同的输入项的偏导数有不同的作用，如下图所示， $\frac{\partial E_d}{\partial x_i^k}$ 用来传播梯度项【用

表示，这里先给出结论，具体原因后面再说】， $\frac{\partial E_d}{\partial w_{ij}^{k+1}}$ 及 $\frac{\partial E_d}{\partial b_j^{k+1}}$ 用来更新系数。

inbound_nodes		inbound_nodes的导数	
<div></div> 输入项 x_i^k	$\frac{\partial E_d}{\partial x_i^k} = \sum_{j=1}^N \frac{\partial net_j^k}{\partial x_i^k} \frac{\partial E_d}{\partial net_j^k} = \sum_{j=1}^N w_{ij}^{k+1} \frac{\partial E_d}{\partial net_j^k}$	用来传播 梯度项	
<div></div> 权重系数 w_{im}^{k+1}	$\frac{\partial E_d}{\partial w_{ij}^{k+1}} = \sum_{j=1}^N \frac{\partial net_j^k}{\partial w_{ij}^{k+1}} \frac{\partial E_d}{\partial net_j^k} = \sum_{j=1}^N x_i^k \frac{\partial E_d}{\partial net_j^k}$	用来系数 更新	
<div></div> 偏执系数 b_m^{k+1}	$\frac{\partial E_d}{\partial b_j^{k+1}} = \sum_{j=1}^N \frac{\partial net_j^k}{\partial b_j^{k+1}} \frac{\partial E_d}{\partial net_j^k} = \sum_{j=1}^N \frac{\partial E_d}{\partial net_j^k}$		
outbound_nodes			
<div></div> 加权输出 net_j^k			

outbound_nodes 的导数不要求，因为在当前类中没有可以用来求其导数的信息，并且当前节点的 outbound_nodes 是下一个节点的 inbound_nodes，所以 outbound_nodes 的导数在下一个节点中被计算

为了方便编程实现，将上图中 3 个偏导数写成矩阵运算的形式，有：

$$\begin{aligned}
\left[\frac{\partial E_d}{\partial \mathbf{x}^k} \right]_{batchsize \times N^k} &= \left[\frac{\partial E_d}{\partial \mathbf{net}^k} \right]_{batchsize \times N^k} \left[\mathbf{w}^{k+1} \right]_{N^k \times N^{k+1}} \\
\left[\frac{\partial E_d}{\partial \mathbf{w}^{k+1}} \right]_{N^k \times N^{k+1}} &= \left(\left[\mathbf{x}^k \right]_{batchsize \times N^k} \right)^T \left[\frac{\partial E_d}{\partial \mathbf{net}^k} \right]_{batchsize \times N^k} \\
\left[\frac{\partial E_d}{\partial \mathbf{b}^{k+1}} \right]_{1 \times N^{k+1}} &= \sum_{i=1}^{batchsize} \left[\frac{\partial E_d}{\partial \mathbf{net}^k} \right]_{i \times N^k}
\end{aligned} \tag{1.7}$$

上图中详细的符号定义及代码推导请参考【深度学习基础模型算法原理及编程实现--03.全链接[3]】。代码修改如下：
class Linear(Node):

```

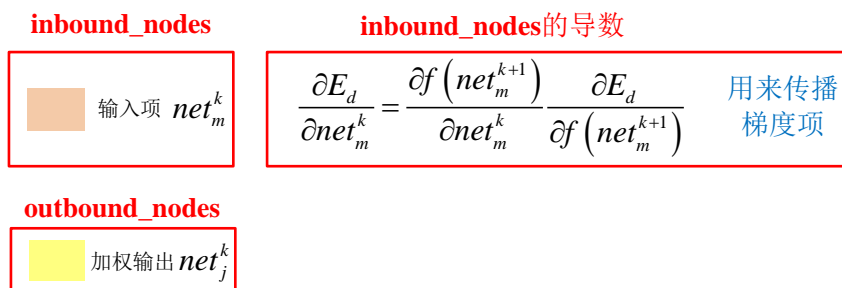
.....
def backward(self):
    self.gradients = {n: np.zeros_like(n.value) for n in self.inbound_nodes}
    for n in self.outbound_nodes:
        grad_cost = n.gradients[self]
        self.gradients[self.inbound_nodes[0]] += np.dot(grad_cost, self.inbound_nodes[1].value.T)
        self.gradients[self.inbound_nodes[1]] += np.dot(self.inbound_nodes[0].value.T, grad_cost)
        self.gradients[self.inbound_nodes[2]] += np.sum(grad_cost, axis=0, keepdims=False) #axis=0 表示对列求

```

和

1.7.5 为 Sigmoid 添加 backward 方法

由前文知，激活函数类中同样有 inbound_nodes 和 outbound_nodes，详细如下图所示，为了方便对比，其中 inbound_nodes 和 outbound_nodes 中不同项所对应的方框颜色与上图中的方框颜色一致。



上图中详细的符号定义及代码推导请参考【深度学习基础模型算法原理及编程实现--03.全链接[3]】。对于 sigmoid 激活函数，有：

$$\frac{\partial f(\mathbf{net}_m^k)}{\partial \mathbf{net}_m^k} = f(\mathbf{net}_m^k) (1 - f(\mathbf{net}_m^k)) \tag{1.8}$$

为了方便编程实现，将上图中对输入项的偏导数写成矩阵运算的形式，有：

$$\left[\frac{\partial E_d}{\partial \mathbf{net}^k} \right]_{batchsize \times N^k} = \left(\left[\frac{\partial f(\mathbf{net}^k)}{\partial \mathbf{net}^k} \right]_{batchsize \times N^k} \right) \otimes \left(\left[f(\mathbf{net}^k) \right]_{batchsize \times N^k} \right) \otimes \left(\left[1 - f(\mathbf{net}^k) \right]_{batchsize \times N^k} \right) \tag{1.9}$$

上图中详细的符号定义及代码推导请参考【深度学习基础模型算法原理及编程实现--03.全链接[3]】。代码修改如下：
class Sigmoid(Node): #激活函数节点类型

```

.....
def backward(self):
    self.gradients = {n: np.zeros_like(n.value) for n in self.inbound_nodes}
    for n in self.outbound_nodes:
        grad_cost = n.gradients[self]

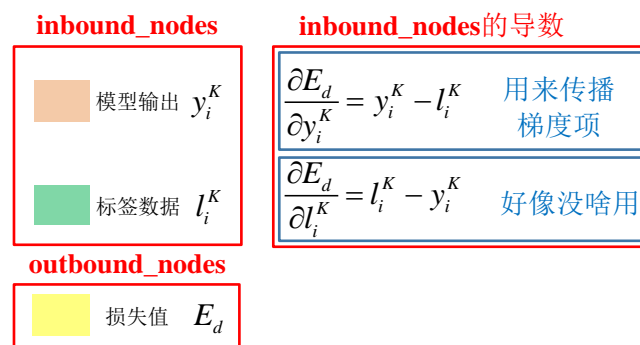
```

```
sigmoid = self.value
```

```
self.gradients[self.inbound_nodes[0]] += sigmoid * (1 - sigmoid) * grad_cost
```

1.7.6 为 MSE 添加 backward 方法

由前文知，损失函数类中同样有 inbound_nodes 和 outbound_nodes，详细如下图所示，为了方便对比，其中 inbound_nodes 和 outbound_nodes 中不同项所对应的方框颜色与上图中的方框颜色一致。需要注意的是，由于损失函数类提供了误差项的原始动力，所以损失函数类中对输入项的偏导数无需后面节点的梯度信息【一般而言，损失函数后也没有其他节点了】。



为了方便编程实现，将上图中对输入项的偏导数写成矩阵运算的形式，有：

$$\left[\frac{\partial E_d}{\partial \mathbf{net}^k} \right]_{batchsize \times N^k} = \left(\left[\frac{\partial f(\mathbf{net}^k)}{\partial \mathbf{net}^k} \right]_{batchsize \times N^k} \right) \otimes \left(\left[f(\mathbf{net}^k) \right]_{batchsize \times N^k} \right) \otimes \left(\left[1 - f(\mathbf{net}^k) \right]_{batchsize \times N^k} \right) \quad (1.10)$$

上图中详细的符号定义及代码推导请参考【深度学习基础模型算法原理及编程实现--03.全链接】[3]。代码修改如下：

```
class MSE(Node):
```

```
.....
```

```
def backward(self):
    self.gradients = {n: np.zeros_like(n.value) for n in self.inbound_nodes}
    self.gradients[self.inbound_nodes[0]] = 2./self.m*self.diff
    self.gradients[self.inbound_nodes[1]] = -2./self.m*self.diff
```

1.7.7 为 forward_pass 添加 backward 计算功能

第二项更改是辅助函数 forward_pass()。该函数被替换成了 forward_and_backward()。

```
def forward_and_backward(graph):
```

```
    for n in graph:
        n.forward()
```

```
    for n in graph[::-1]:
        n.backward()
```

1.8 参数更新_小批量梯度下降法

这里采用最简单的梯度下降法：

$$\begin{aligned} w_{ij}^k &\leftarrow w_{ij}^k - \eta \frac{\partial E_d}{\partial w_{ij}^k} \\ b_i^k &\leftarrow b_i^k - \eta \frac{\partial E_d}{\partial b_i^k} \end{aligned} \quad k = 1, \dots, K, i = 1, \dots, N_{k-1}, j = 1, \dots, N_k \quad (1.11)$$

整理成向量形式：

$$\begin{aligned} W^k &\leftarrow W^k - \eta \frac{\partial E_d}{\partial W^k} \\ b^k &\leftarrow b^k - \eta \frac{\partial E_d}{\partial b^k} \end{aligned} \quad k = 1, \dots, K \quad (1.12)$$

由前文的式(1.7)知，对全连接而言，不论批采样大小是多少， $\frac{\partial E_d}{\partial W^k}$ 的大小恒为 $[N^k \times N^{k+1}]$ ， $\frac{\partial E_d}{\partial b^k}$ 的大小恒为

$[1 \times N^{k+1}]$ ，与 batchsize 无关，但 $\frac{\partial E_d}{\partial W^k}$ 及 $\frac{\partial E_d}{\partial b^k}$ 具体的值是与 batchsize 相关的，batchsize 个训练数据对应的偏导

数的和。所以这里的优化方法由 batchsize 决定，batchsize 为 1 的话就是随机梯度下降，batchsize 为所有训练数据样本个数的话就是批量梯度下降，batchsize 介于两者之间的话就是小批量梯度下降。

```
def sgd_update(trainables, learning_rate=1e-2):
```

```
    for t in trainables:
```

```
        partial = t.gradients[t]
```

```
        t.value -= learning_rate * partial
```

1.9 算例实现-MLP 实现 MNIST 数据分类

```
# -*- coding: utf-8 -*-
```

```
import matplotlib.pyplot as plt
```

```
import numpy as np
```

```
import random
```

```
from sklearn.datasets import load_boston
```

```
from sklearn.utils import shuffle, resample
```

```
import sys
```

```
from miniFlow import *
```

```
from tensorflow.examples.tutorials.mnist import input_data
```

```
# Load data
```

```
mnist = input_data.read_data_sets("./MNISTData", one_hot=True)
```

```
in_units = 784
```

```
h1_units = 300
```

```
o_units = 10
```

```
random.seed(1)
```

```
#####
```

```
#W1_ = 1.*np.random.randn(in_units, h1_units)
```

```
#b1_ = np.zeros(h1_units)
```

```
#W2_ = 0.0*np.random.randn(h1_units, o_units)
```

```
##W2_ = 1*np.zeros([h1_units, o_units])
```

```
#b2_ = np.zeros(o_units)
```

```
## Neural network
```

```
#X, y = placeholder(name='X'), placeholder(name='y')
```

```
#W1, b1 = variable(value=W1_, name='W1'), variable(value=b1_, name='b1')
```

```
#W2, b2 = variable(value=W2_, name='W2'), variable(value=b2_, name='b2')
```

```
#hidden1 = ReLU(Linear(X, W1, b1))
```

```
#out = Linear(hidden1, W2, b2) #out = ReLU(Linear(hidden1, W2, b2))
```

```
#cost = softmax_cross_entropy_with_logits(labels=y, logits=out, name='cost') #loss = tf.reduce_mean(cost)
```

```

#
#epochs = 10
#m = 50000
#batch_size = 128*2
#learning_rate=1e-3
#steps_per_epoch = m // batch_size
#####
W1_ = 1./np.sqrt(in_units*h1_units)*np.random.randn(in_units, h1_units)
b1_ = np.zeros(h1_units)
W2_ = 0.1*np.random.randn(h1_units, o_units)
b2_ = np.zeros(o_units)
# Neural network
X, y = placeholder(name='X'), placeholder(name='y')
W1, b1 = variable(value=W1_, name='W1'), variable(value=b1_, name='b1')
W2, b2 = variable(value=W2_, name='W2'), variable(value=b2_, name='b2')
#hidden1 = Sigmoid(Linear(X, W1, b1))
#out = Sigmoid(Linear(hidden1, W2, b2))
hidden1 = ReLU(Linear(X, W1, b1))
out = ReLU(Linear(hidden1, W2, b2),name='out')
cost = MSE(y, out)
epochs = 2
m = 50000
batch_size = 64
learning_rate=2e-2
steps_per_epoch = m // batch_size
#####

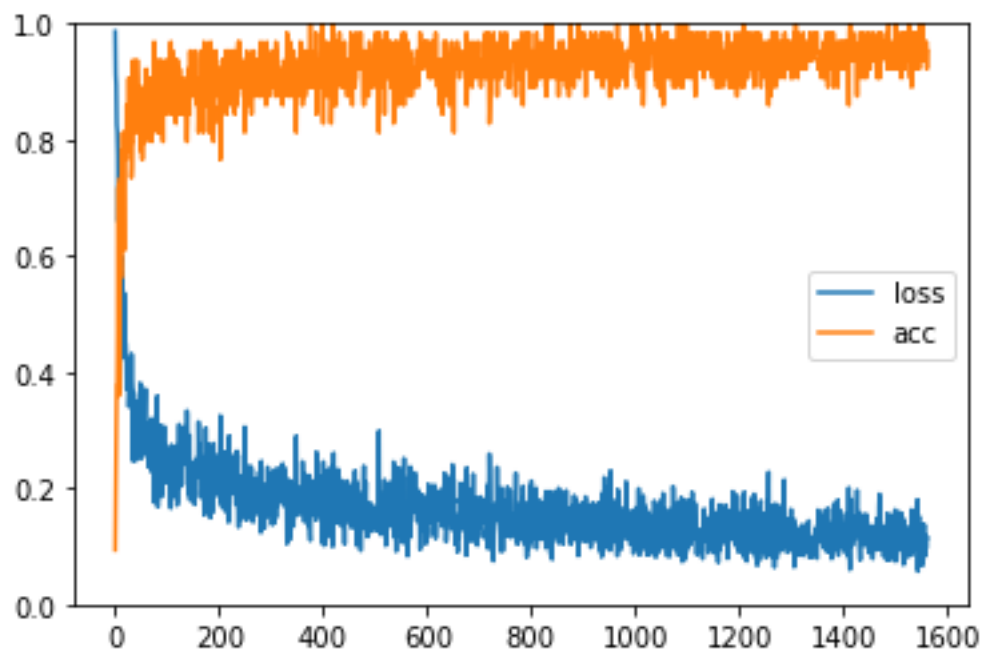
print("Total number of examples = {}".format(m))

global_variables_initializer()
loss_list = []
acc_list = []
for i in range(epochs):
    loss = 0
    for j in range(steps_per_epoch):
        X_batch, y_batch = mnist.train.next_batch(batch_size)
        feed_dict = {X: X_batch,y: y_batch.reshape(batch_size,-1)}
        graph = forward_and_backward(feed_dict)
        sgd_update(learning_rate)
    #    print('graph[-11].value[5,5:8]:',graph[-8].value[5,5:8],'loss:',np.mean(graph[-1].value))
    loss = np.mean(graph[-1].value)
    acc = np.mean((np.argmax(out.value,1) == np.argmax(y_batch,1)).astype(int))
    loss_list.append(loss)
    acc_list.append(acc)
    sys.stdout.write("\rprocess: {}/{}, loss:{}, acc:{}".format(j, steps_per_epoch, loss, acc))
plt.figure()
#    plt.subplot(211)

```



```
plt.plot(range(len(loss_list)),loss_list,label=u'loss')  
# plt.subplot(212)  
plt.plot(range(len(loss_list)),acc_list,label=u'acc')  
plt.ylim([0,1])  
plt.legend()  
plt.show()
```



2 自己编写神经网络库 ANNbox【高仿 tensorflow】__01 实现 MLP

为什么要自己动手编写神经网络呢？原因有 2 个：

1.前几天学习权重及偏执初始化对神经网络的影响时，尝试观察每次迭代过程中的系数变化。但发现在训练迭代过程中使用 `sess.run()` 只能得到权重的初始值，训练更新的权重是看不到的，只能通过 `tensorboard` 可视化权重的方式实现中间变量的观察（`tf.histogram_summary(layer_name + '/weights', Weights)`），于是我就想如果我自己编写的话就让程序在迭代的过程中可以观察到中间变量。

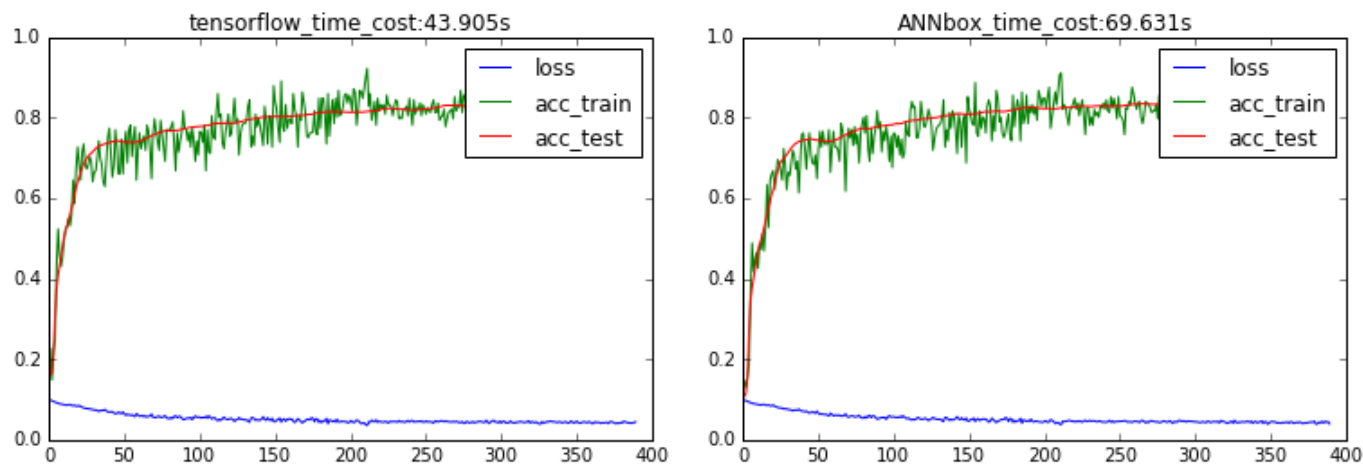
2.很久以前也手动编过全连接神经网络，当时是按照以单个层为类进行编写的【详参深度学习基础模型算法原理及编程实现系列文章】，但后来通过学习发现 `MiniFlow`【详参深度学习基础模型算法原理及编程实现 - 11.构建简化版的 tensorflow—MiniFlow】，觉得很厉害，主要的启发点如下：

(a).拓扑排序：利用 Kahn 算法将有向无环图排列成为一个线性有序序列，从而实现程序自身自动进行前向计算及后向计算

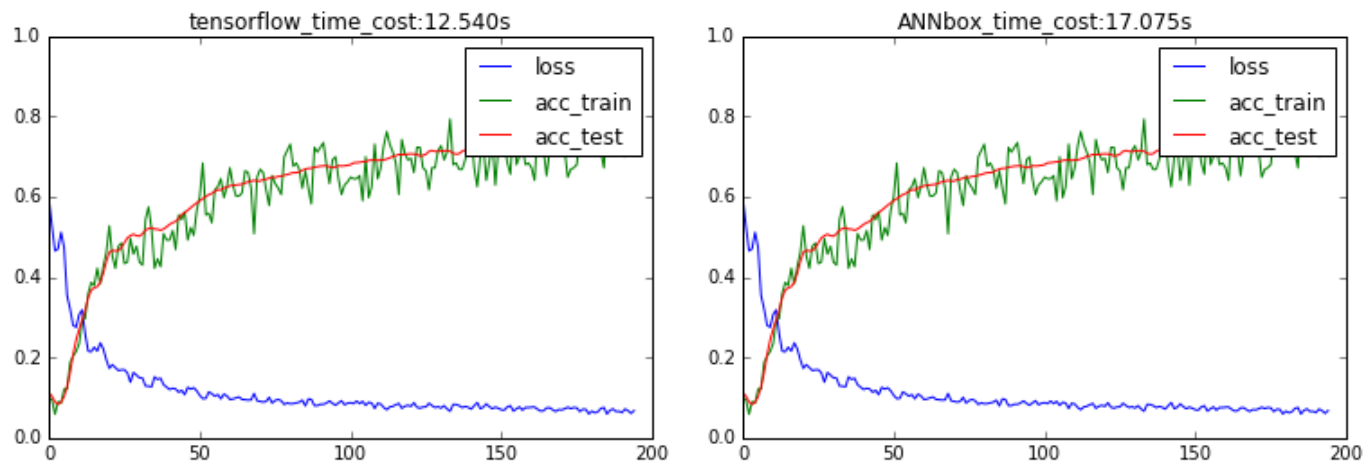
(b).类的拆分更彻底：以线性加权、激活函数变换、损失函数等单个操作为类进行编写，

这两点使得我可以编出移植性更好以及操作更傻瓜式的程序，就想 `tensorflow` 一样，所以我又从新整理了一下思路，尝试自己编写神经网络库 `ANNbox`，`ANNbox` 的使用方式趋近于 `tensorflow` 的使用方式，唯一的区别是，你无需在电脑上安装 `tensorflow`，并且对于原来在 `tensorflow` 框架下运行的代码，只需将其中的 `import tensorflow as tf` 改为 `import ANNbox as tf` 即可运行【目前还需要再改几项】。在看本文之前请先看下【深度学习基础模型算法原理及编程实现 - 11.构建简化版的 tensorflow—MiniFlow】及【深度学习基础模型算法原理及编程实现 - 03.全链接】以便对本文内容有足够的知识准备。

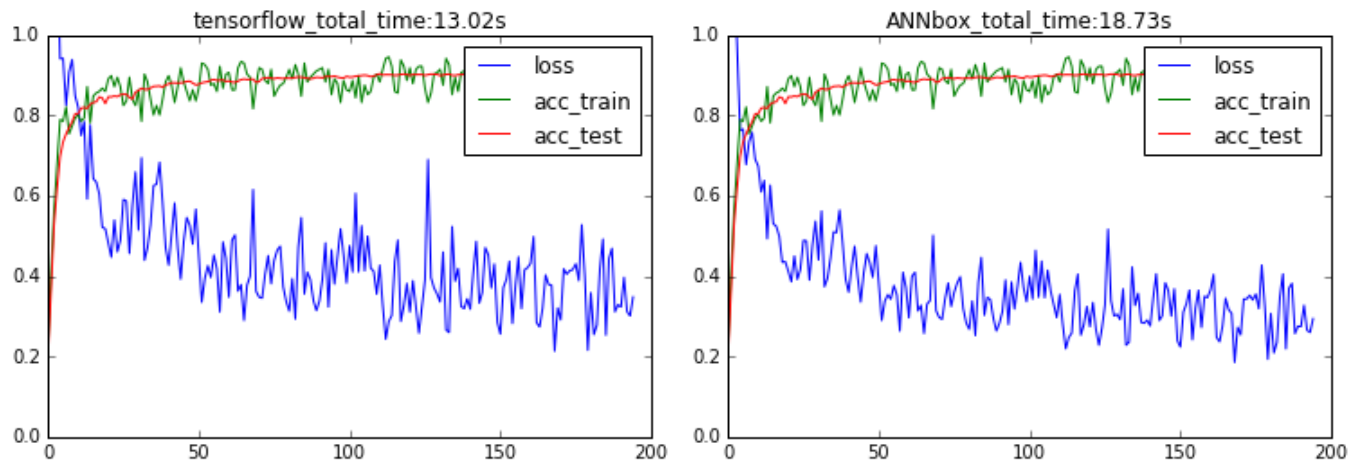
在本小节的学习结束之后，我们通过 2 层 MLP 实现 MNIST 数据集分类，只需要 `activator.py`、`ANNbox.py`、`base.py`、`optimization.py` 这 4 个文件，就可以实现对一段与 `tensorflow` 构架下极其相似的程序的编译，运行结果对比如下【几乎】，在最终的测试集上准确率达到 95%【可惜 `ANNbox` 计算耗时较长，后面慢慢改进】：



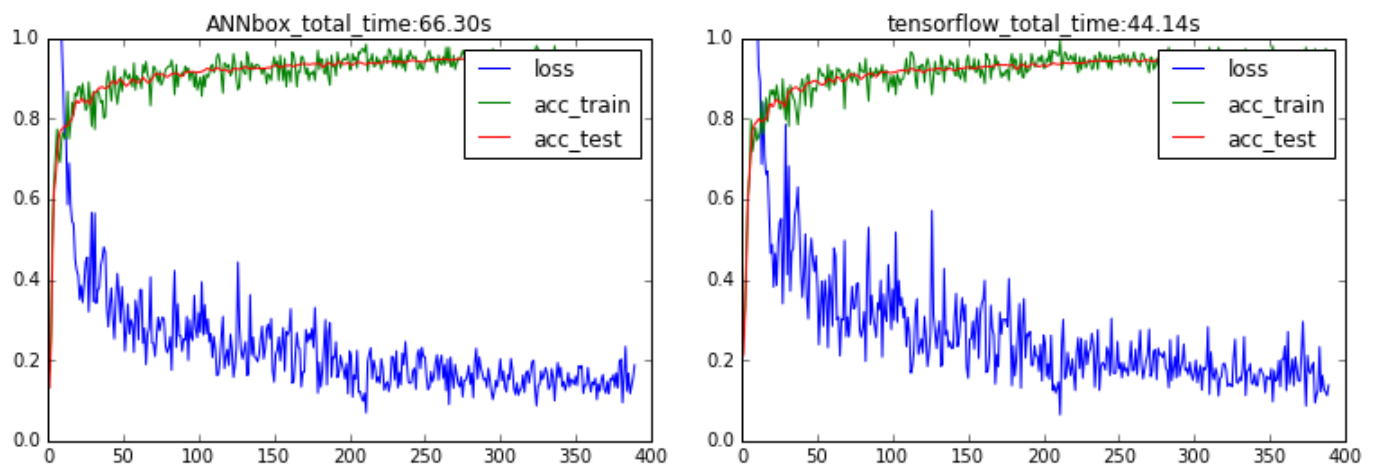
损失函数为均方差损失函数的情况【输入 784，隐层 300，输出 10】



损失函数为均方差损失函数的情况【输入 784，输出 10】



损失函数为 softmax_cross_entropy_with_logits 的情况【输入 784，输出 10】



损失函数为 softmax_cross_entropy_with_logits 的情况【输入 784，隐藏层 300，输出 10】

从上面 4 个图可以看出，在 cpu 上运行的情况下，我自己编的 ANNbox 库函数运行时间大概是 tensorflow 运行时间的 1.3-1.5 倍左右，应该有很大可以改进的地方【欢迎大家交流】

2.1 线性加权输入类 Linear

首先是线性加权输入类，主要实现下式的运算操作【公式中参数详细的说明请参考深度学习基础模型算法原理及编程实现--03.全链接，全文同】

$$net_m^{k+1} = \sum_{i=1}^{N_k} x_i^k w_{im}^{k+1} + b_m^{k+1} \quad (1.13)$$

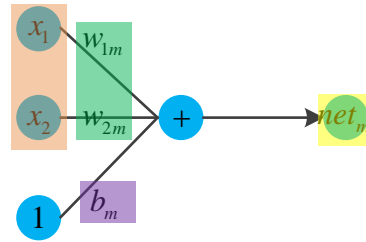
为了方便编程实现，将上式整理成矩阵运算的形式，有：

$$[net^{k+1}]_{batchsize \times N^{k+1}} = [x^k]_{batchsize \times N^k} [w^{k+1}]_{N^k \times N^{k+1}} + [1]_{batchsize \times 1} [b^{k+1}]_{1 \times N^{k+1}} \quad (1.14)$$

忽略层数 k 的影响，上式可整理为：

$$net_m = \sum_{i=1}^N x_i w_{im} + b_m \quad (1.15)$$

下面再通过简单的图示对线性加权进行说明：



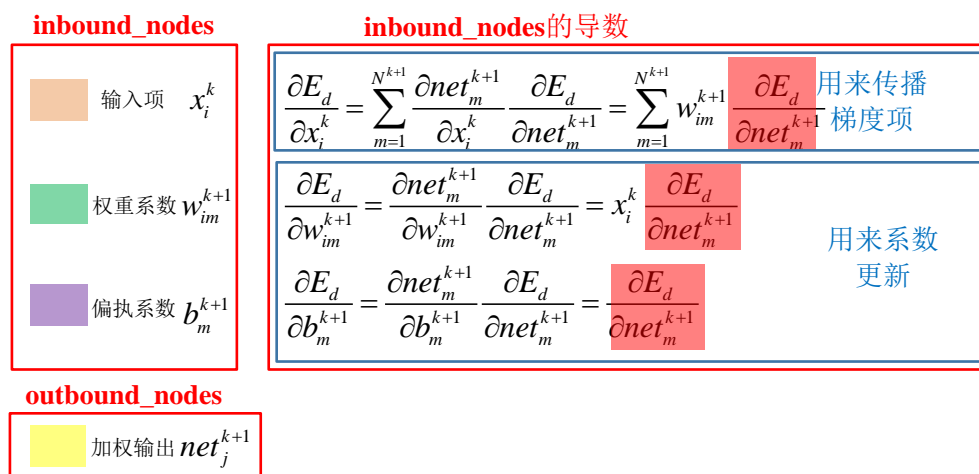
由前文知，Linear类中同样有inbound_nodes和outbound_nodes，详细如下图所示，为了方便对比，其中inbound_nodes和outbound_nodes中不同项所对应的方框颜色与上图中的方框颜色一致。且正如下图所示，outbound_nodes的导数不要求，因为在当前类中没有可以用来求其导数的信息，并且当前节点的outbound_nodes是下一个节点的inbound_nodes，所以outbound_nodes的导数在下一个节点中被计算。

此外，inbound_nodes的导数被表示为当前类中的输入项与损失函数对输出项偏导数【如下图所示， $\frac{\partial E_d}{\partial net_j^k}$ 】

的乘积，其中当前类中的输入项是已知的，而 $\frac{\partial E_d}{\partial net_j^k}$ 并不知，但 $\frac{\partial E_d}{\partial net_j^k}$ 具体的值可通过访问下一个神经元对输入项的偏导数得到。

Linear类中的输入项有3个，对不同的输入项的偏导数有不同的作用，如下图所示， $\frac{\partial E_d}{\partial x_i^k}$ 用来传播梯度项【用

表示，这里先给出结论，具体原因后面再说】， $\frac{\partial E_d}{\partial w_{ij}^{k+1}}$ 及 $\frac{\partial E_d}{\partial b_j^{k+1}}$ 用来更新系数。



outbound_nodes的导数不要求，因为在当前类中没有可以用来求其导数的信息，并且当前节点的outbound_nodes是下一个节点的inbound_nodes,所以outbound_nodes的导数在下一个节点中被计算

2.1.1 输出层权重系数矩阵初始化为0的情况

说到这里我想补充说明一下一个现象，就是好多人在设置网络权重系数的时候喜欢将最后一层的权重系数全部初始化为0，这样有时能取得不错的结果（其实这应该是一种巧合），但是如果有连续2层以上的神经网络的权重系数初始化为0，那么该网络将无法收敛，甚至无法迭代。这是为什么呢？

假设第k层以及k+1层的权重系数全部初始化为0，从上图中可以看出，如果k层的偏执项也为0，那么k+1层的输入也为0，从而导致k+1层的权重系数梯度为0，从而在当前迭代中，参数更新后的权重系数仍然为0；而由于k+1层权重初始值为0，所以对输入的偏导数也为0，这样直接导致第k层用来更新系数的梯度全为0，即迭代结

束后 k 层的权重及偏执系数全为 0。所以该神经网络的输出随着迭代的进行没有变化。

但对于只有单层神经网络的权重系数为 0 的情况而言，虽然其初始值为 0，但经过一次迭代后其权重系数就发生了变化。

为了方便编程实现，将上图中 3 个偏导数写成矩阵运算的形式，有：

$$\begin{aligned}
 \left[\frac{\partial E_d}{\partial \mathbf{x}^k} \right]_{batchsize \times N^k} &= \left[\frac{\partial E_d}{\partial \mathbf{net}^{k+1}} \right]_{batchsize \times N^{k+1}} \left(\left[\mathbf{w}^{k+1} \right]_{N^k \times N^{k+1}} \right)^T \\
 \left[\frac{\partial E_d}{\partial \mathbf{w}^{k+1}} \right]_{N^k \times N^{k+1}} &= \left(\left[\mathbf{x}^k \right]_{batchsize \times N^k} \right)^T \left[\frac{\partial E_d}{\partial \mathbf{net}^{k+1}} \right]_{batchsize \times N^{k+1}} \\
 \left[\frac{\partial E_d}{\partial \mathbf{b}^{k+1}} \right]_{1 \times N^{k+1}} &= \left(\left[\mathbf{1} \right]_{batchsize \times 1} \right)^T \left[\frac{\partial E_d}{\partial \mathbf{net}^{k+1}} \right]_{batchsize \times N^{k+1}} = \sum_{i=1}^{batchsize} \left[\frac{\partial E_d}{\partial \mathbf{net}^{k+1} [i,:]} \right]_{1 \times N^{k+1}}
 \end{aligned} \tag{1.16}$$

上图中详细的符号定义及代码推导请参考【深度学习基础模型算法原理及编程实现--03.全链接[3]】。相应的代码实现如下：

```
class Linear(Node):
```

```
    """线性加权神经元类，实现节点加权输入"""
```

```
    def __init__(self, X, W, b, name=[]):
```

```
        Node.__init__(self, inbound_nodes=[X, W, b], name=name)
```

```
    def forward(self):
```

```
        X = self.inbound_nodes[0].value
```

```
        W = self.inbound_nodes[1].value
```

```
        b = self.inbound_nodes[2].value
```

```
        self.value = np.dot(X, W) + b
```

```
    def backward(self):
```

```
        self.gradients = {n: np.zeros_like(n.value) for n in self.inbound_nodes}
```

```
#         self.gradients = {}
```

```
#         self.gradients[self.inbound_nodes[0]] = np.zeros_like(self.inbound_nodes[0].value)
```

```
#         self.gradients[self.inbound_nodes[1]] = np.zeros_like(self.inbound_nodes[1].value)
```

```
#         self.gradients[self.inbound_nodes[2]] = np.zeros_like(self.inbound_nodes[2].value) #axis=0表示对列求和
```

```
        for n in self.outbound_nodes:
```

```
            grad_cost = n.gradients[self]
```

```
            self.gradients[self.inbound_nodes[0]] += np.dot(grad_cost, self.inbound_nodes[1].value.T)
```

```
            self.gradients[self.inbound_nodes[1]] += np.dot(self.inbound_nodes[0].value.T, grad_cost)
```

```
            self.gradients[self.inbound_nodes[2]] += np.sum(grad_cost, axis=0, keepdims=False) #axis=0表示对列求
```

和

2.2 矩阵相乘类 matmul

首先是线性加权输入类，主要实现下式的运算操作【公式中参数详细的说明请参考深度学习基础模型算法原理及编程实现--03.全链接，全文同】

$$net_m^{k+1} = \sum_{i=1}^{N_k} x_i^k w_{im}^{k+1} \quad (1.17)$$

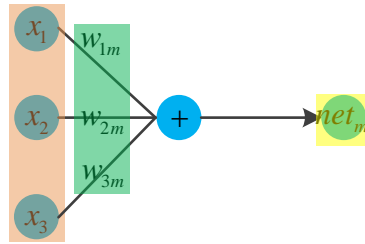
为了方便编程实现，将上式整理成矩阵运算的形式，有：

$$[net^{k+1}]_{batchsize \times N^{k+1}} = [x^k]_{batchsize \times N^k} [w^{k+1}]_{N^k \times N^{k+1}} \quad (1.18)$$

忽略层数 k 的影响，上式可整理为：

$$net_m = \sum_{i=1}^N x_i w_{im} \quad (1.19)$$

下面再通过简单的图示对线性加权进行说明：



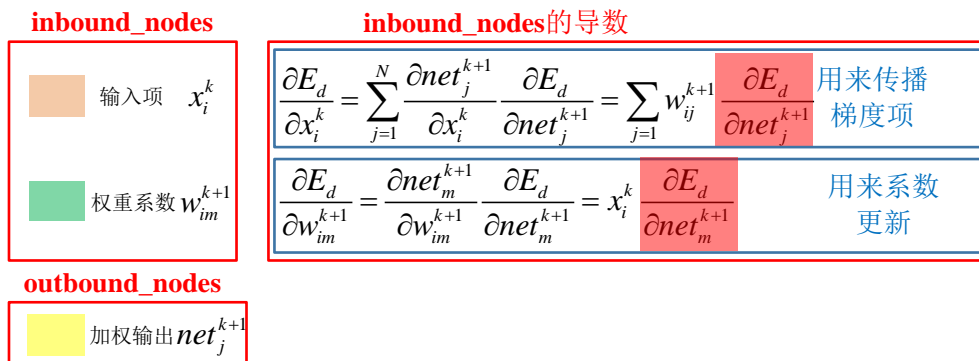
由前文知，`matmul`类中同样有`inbound_nodes`和`outbound_nodes`，详细如下图所示，为了方便对比，其中`inbound_nodes`和`outbound_nodes`中不同项所对应的方框颜色与上图中的方框颜色一致。且正如下图所示，`outbound_nodes`的导数不要求，因为在当前类中没有可以用来求其导数的信息，并且当前节点的`outbound_nodes`是下一个节点的`inbound_nodes`，所以`outbound_nodes`的导数在下一个节点中被计算。

此外，`inbound_nodes`的导数被表示为当前类中的输入项与损失函数对输出项偏导数【如下图所示， $\frac{\partial E_d}{\partial net_j^k}$ 】

的乘积，其中当前类中的输入项是已知的，而 $\frac{\partial E_d}{\partial net_j^k}$ 并不知，但 $\frac{\partial E_d}{\partial net_j^k}$ 具体的值可通过访问下一个神经元对输入项的偏导数得到。

`Linear`类中的输入项有3个，对不同的输入项的偏导数有不同的作用，如下图所示， $\frac{\partial E_d}{\partial x_i^k}$ 用来传播梯度项【用

表示，这里先给出结论，具体原因后面再说】， $\frac{\partial E_d}{\partial w_{ij}^{k+1}}$ 及 $\frac{\partial E_d}{\partial b_j^{k+1}}$ 用来更新系数。



为了方便编程实现，将上图中 3 个偏导数写成矩阵运算的形式，有：

$$\begin{aligned} \left[\frac{\partial E_d}{\partial \mathbf{x}^k} \right]_{batchsize \times N^k} &= \left[\frac{\partial E_d}{\partial \mathbf{net}^{k+1}} \right]_{batchsize \times N^{k+1}} \left(\left[\mathbf{w}^{k+1} \right]_{N^k \times N^{k+1}} \right)^T \\ \left[\frac{\partial E_d}{\partial \mathbf{w}^{k+1}} \right]_{N^k \times N^{k+1}} &= \left(\left[\mathbf{x}^k \right]_{batchsize \times N^k} \right)^T \left[\frac{\partial E_d}{\partial \mathbf{net}^{k+1}} \right]_{batchsize \times N^{k+1}} \end{aligned} \quad (1.20)$$

上图中详细的符号定义及代码推导请参考【[深度学习基础模型算法原理及编程实现--03.全链接](#)】[3]。相应的代码实现如下：

```
class matmul(Node):
    """矩阵相乘神经元类，实现节点加权输入"""
    def __init__(self, X, W, name=[]):
        Node.__init__(self, inbound_nodes=[X,W], name=name)

    def forward(self):
        X = self.inbound_nodes[0].value
        W = self.inbound_nodes[1].value
        self.value = np.dot(X, W)

    def backward(self):
        self.gradients = {n: np.zeros_like(n.value) for n in self.inbound_nodes}
        for n in self.outbound_nodes:
            grad_cost = n.gradients[self]
            self.gradients[self.inbound_nodes[0]] += np.dot(grad_cost, self.inbound_nodes[1].value.T)
            self.gradients[self.inbound_nodes[1]] += np.dot(self.inbound_nodes[0].value.T, grad_cost)
```

2.3 矩阵相加类 Add 【用于运算符重载】

首先是线性加权输入类，主要实现下式的运算操作【公式中参数详细的说明请参考深度学习基础模型算法原理及编程实现--03.全链接，全文同】

$$\mathbf{net}_m^{k+1} = \mathbf{y}_m^{k+1} + \mathbf{b}_m^{k+1} \quad (1.21)$$

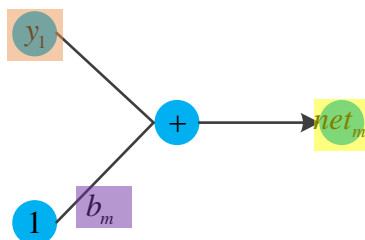
为了方便编程实现，将上式整理成矩阵运算的形式，有：

$$\left[\mathbf{net}^{k+1} \right]_{batchsize \times N^{k+1}} = \left[\mathbf{y}^{k+1} \right]_{batchsize \times N^{k+1}} + \left[\mathbf{1} \right]_{batchsize \times 1} \left[\mathbf{b}^{k+1} \right]_{1 \times N^{k+1}} \quad (1.22)$$

忽略层数 k 的影响，上式可整理为：

$$\mathbf{net}_m = \mathbf{y}_m + \mathbf{b}_m \quad (1.23)$$

下面再通过简单的图示对线性加权进行说明：



由前文知，Add类中同样有inbound_nodes和outbound_nodes，详细如下图所示，为了方便对比，其中inbound_nodes和outbound_nodes中不同项所对应的方框颜色与上图中的方框颜色一致。且正如下图所示，

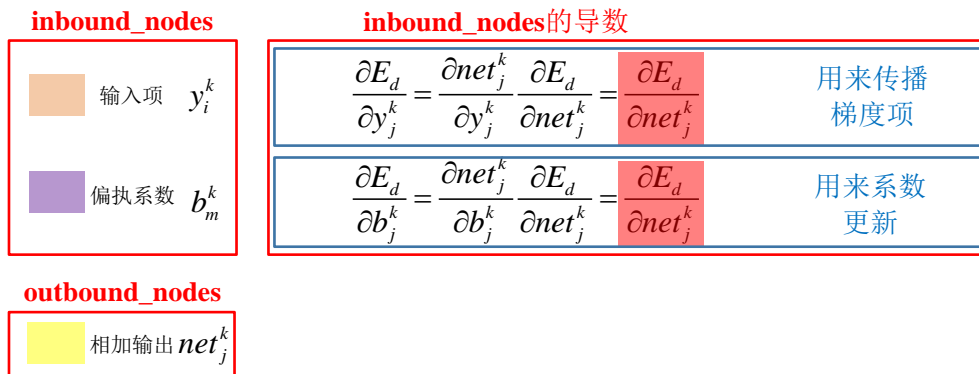
outbound_nodes的导数不要求，因为在当前类中没有可以用来求其导数的信息，并且当前节点的outbound_nodes是下一个节点的inbound_nodes，所以outbound_nodes的导数在下一个节点中被计算。

此外，inbound_nodes的导数被表示为当前类中的输入项与损失函数对输出项偏导数【如下图所示， $\frac{\partial E_d}{\partial net_j^k}$ 】

的乘积，其中当前类中的输入项是已知的，而 $\frac{\partial E_d}{\partial net_j^k}$ 并不知，但 $\frac{\partial E_d}{\partial net_j^k}$ 具体的值可通过访问下一个神经元对输入项的偏导数得到。

Linear类中的输入项有3个，对不同的输入项的偏导数有不同的作用，如下图所示， $\frac{\partial E_d}{\partial x_i^k}$ 用来传播梯度项【用

表示，这里先给出结论，具体原因后面再说】， $\frac{\partial E_d}{\partial w_{ij}^{k+1}}$ 及 $\frac{\partial E_d}{\partial b_j^{k+1}}$ 用来更新系数。



为了方便编程实现，将上图中 3 个偏导数写成矩阵运算的形式，有：

$$\begin{aligned} \left[\frac{\partial E_d}{\partial \mathbf{y}^k} \right]_{batchsize \times N^k} &= \left[\frac{\partial E_d}{\partial \mathbf{net}^k} \right]_{batchsize \times N^k} \\ \left[\frac{\partial E_d}{\partial \mathbf{b}^k} \right]_{1 \times N^k} &= \sum_{i=1}^{batchsize} \left[\frac{\partial E_d}{\partial \mathbf{net}^k [i,:]} \right]_{1 \times N^k} \end{aligned} \quad (1.24)$$

上图中详细的符号定义及代码推导请参考【[深度学习基础模型算法原理及编程实现--03.全链接][3]】。相应的代码实现如下：

```
class Add(Node):
    """线性加权神经元类，实现节点加权输入"""
    def __init__(self, Y, b, name=[]):
        Node.__init__(self, inbound_nodes=[Y, b], name=name)

    def forward(self):
        Y = self.inbound_nodes[0].value
        b = self.inbound_nodes[1].value
        self.value = Y + b
```


"""原来的版本"""

```
# def backward(self):
#     self.gradients = {n: np.zeros_like(n.value) for n in self.inbound_nodes}
#     for n in self.outbound_nodes:
#         grad_cost = n.gradients[self]
#         self.gradients[self.inbound_nodes[0]] += grad_cost #axis=0表示对列求和
#         self.gradients[self.inbound_nodes[1]] += np.sum(grad_cost, axis=0, keepdims=False) #axis=0表示对列求和
```

```
def backward(self):
    self.gradients = {n: np.zeros_like(n.value) for n in self.inbound_nodes}

    if(self.outbound_nodes==[]):
        grad_cost = 1./list_product(self.inbound_nodes[0].value.shape) *
np.ones_like(self.inbound_nodes[0].value)
        self.gradients[self.inbound_nodes[0]] += grad_cost #axis=0表示对列求和
        grad_cost = 1./list_product(self.inbound_nodes[1].value.shape) *
np.ones_like(self.inbound_nodes[1].value)
        self.gradients[self.inbound_nodes[1]] += grad_cost #axis=0表示对列求和
    else:
        for n in self.outbound_nodes:
            grad_cost = n.gradients[self]
            self.gradients[self.inbound_nodes[0]] += grad_cost #axis=0表示对列求和
            self.gradients[self.inbound_nodes[1]] += np.sum(grad_cost, axis=0, keepdims=False) #axis=0表示对列求和
```

2.4 矩阵相减类 Sub 【用于运算符重载】

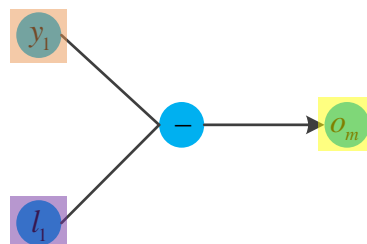
该类主要是用于均方误差损失函数，所以用于相减的两个输入节点参数维数应该是**，主要实现了下式的运算操作【公式中参数详细的说明请参考深度学习基础模型算法原理及编程实现--03.全链接，全文同】

$$o_m^k = y_m^k - l_m^k \quad (1.25)$$

为了方便编程实现，将上式整理成矩阵运算的形式，有：

$$\begin{bmatrix} o^k \end{bmatrix}_{batchsize \times N^{k+1}} = \begin{bmatrix} y^k \end{bmatrix}_{batchsize \times N^{k+1}} - \begin{bmatrix} l^k \end{bmatrix}_{batchsize \times N^{k+1}} \quad (1.26)$$

下面再通过简单的图示对线性加权进行说明：

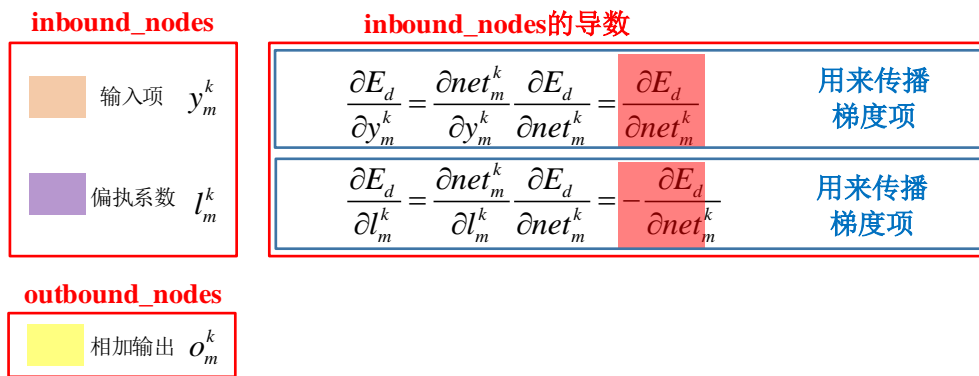


由前文知，Sub类中同样有inbound_nodes和outbound_nodes，详细如下图所示，为了方便对比，其中

inbound_nodes和outbound_nodes中不同项所对应的方框颜色与上图中的方框颜色一致。且正如下图所述，outbound_nodes的导数不要求，因为在当前类中没有可以用来求其导数的信息，并且当前节点的outbound_nodes是下一个节点的inbound_nodes，所以outbound_nodes的导数在下一个节点中被计算。

此外，inbound_nodes的导数被表示为当前类中的输入项对inbound_nodes的导数与损失函数对输出项偏导数【如下图所示， $\frac{\partial E_d}{\partial net_j^k}$ 】的乘积，其中当前类中的输入项是已知的，而 $\frac{\partial E_d}{\partial net_j^k}$ 并不知，但 $\frac{\partial E_d}{\partial net_j^k}$ 具体的值可通过访问下一个神经元对输入项的偏导数得到。

Sub类中的输入项有2个，对不同的输入项的偏导数有不同的作用，如下图所示， $\frac{\partial E_d}{\partial x_i^k}$ 用来传播梯度项， $\frac{\partial E_d}{\partial w_{ij}^{k+1}}$ 及 $\frac{\partial E_d}{\partial b_j^{k+1}}$ 用来更新系数。



为了方便编程实现，将上图中 2 个偏导数写成矩阵运算的形式，有：

$$\begin{aligned} \left[\frac{\partial E_d}{\partial \mathbf{y}^k} \right]_{batchsize \times N^k} &= \left[\frac{\partial E_d}{\partial \mathbf{net}^k} \right]_{batchsize \times N^k} \\ \left[\frac{\partial E_d}{\partial \mathbf{l}^k} \right]_{batchsize \times N^k} &= - \left[\frac{\partial E_d}{\partial \mathbf{net}^k} \right]_{batchsize \times N^k} \end{aligned} \quad (1.27)$$

上图中详细的符号定义及代码推导请参考【[深度学习基础模型算法原理及编程实现-03.全链接][3]】。相应的代码实现如下：

```
class Sub(Node):
    """线性加权神经元类，实现节点加权输入"""
    def __init__(self, Y, label, name=[]):
        Node.__init__(self, inbound_nodes=[Y, label], name=name)

    def forward(self):
        Y = self.inbound_nodes[0].value
        label = self.inbound_nodes[1].value
        self.value = Y - label

    def backward(self):
```

```

self.gradients = {n: np.zeros_like(n.value) for n in self.inbound_nodes}
for n in self.outbound_nodes:
    grad_cost = n.gradients[self]
    self.gradients[self.inbound_nodes[0]] += grad_cost
    self.gradients[self.inbound_nodes[1]] -= grad_cost

```

2.5 平方运算 Squared 类

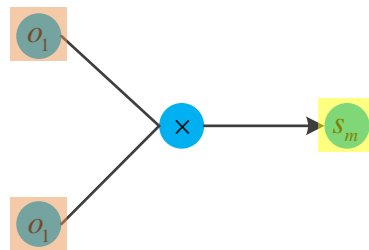
这里假设只是用于均方差损失函数中的平方运算，那么输入节点 o 的维度是 $batchsize \times N^k$ 的，主要实现以下运算：

$$s_m^k = (o_m^k)^2 \quad (1.28)$$

为了方便编程实现，将上式整理成矩阵运算的形式，有：

$$\begin{bmatrix} s^k \end{bmatrix}_{batchsize \times N^{k+1}} = \begin{bmatrix} o^k \end{bmatrix}_{batchsize \times N^{k+1}} \cdot \begin{bmatrix} o^k \end{bmatrix}_{batchsize \times N^{k+1}} \quad (1.29)$$

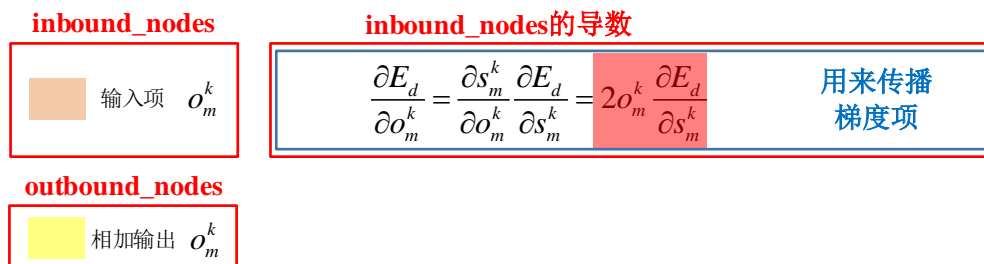
下面再通过简单的图示对线性加权进行说明：



由前文知，Squared类中同样有inbound_nodes和outbound_nodes，详细如下图所示，为了方便对比，其中inbound_nodes和outbound_nodes中不同项所对应的方框颜色与上图中的方框颜色一致。且正如下图所示，outbound_nodes的导数不要求，因为在当前类中没有可以用来求其导数的信息，并且当前节点的outbound_nodes是下一个节点的inbound_nodes，所以outbound_nodes的导数在下一个节点中被计算。

此外，inbound_nodes的导数被表示为当前类中的输入项对inbound_nodes的导数与损失函数对输出项偏导数【如下图所示， $\frac{\partial E_d}{\partial s_m^k}$ 】的乘积，其中当前类中的输入项是已知的，而 $\frac{\partial E_d}{\partial s_m^k}$ 并不知，但 $\frac{\partial E_d}{\partial s_m^k}$ 具体的值可通过访问下一个神经元对输入项的偏导数得到。

Squared类中的输入项有1个，对输入项的偏导数如下图所示，且 $\frac{\partial E_d}{\partial o_m^k}$ 是用来传播梯度项的。



为了方便编程实现，将上图中的偏导数写成矩阵运算的形式，有：

$$\left[\frac{\partial E_d}{\partial \mathbf{o}^k} \right]_{batchsize \times N^k} = 2 \left[\mathbf{o}^k \right]_{batchsize \times N^k} \cdot \left[\frac{\partial E_d}{\partial \mathbf{s}^k} \right]_{batchsize \times N^k} \quad (1.30)$$

上图中详细的符号定义及代码推导请参考【[深度学习基础模型算法原理及编程实现--03.全链接][3]】。相应的代码实现如下：

```
class Squared(Node):
    """线性加权神经元类，实现节点加权输入"""
    def __init__(self, o, name=[]):
        Node.__init__(self, inbound_nodes=[o], name=name)

    def forward(self):
        self.value = o**2

    def backward(self):
        self.gradients = {n: np.zeros_like(n.value) for n in self.inbound_nodes}
        for n in self.outbound_nodes:
            grad_cost = n.gradients[self]
            self.gradients[self.inbound_nodes[0]] += 2*o*grad_cost
```

2.6 reduce_mean 类【这里只给出了 reduce_mean 后结果维度为 1*1 的情况】

reduce_mean 类是用来求平均值的，TensorFlow 中的定义如下，tf.reduce_mean(input_tensor, reduction_indices=None, keep_dims=False, name=None)，这里我们先只考虑 input_tensor 和 name 这两个参数，其他参数暂且不考虑，将来用到再说。

假设输入节点 \mathbf{o} 的维度为 N 维，且第 n 个维度的长度用 D_n 表示，那么输入节点 \mathbf{o} 的维度可表示为 $D_1 \times \dots \times D_N$ ，主要实现以下运算：

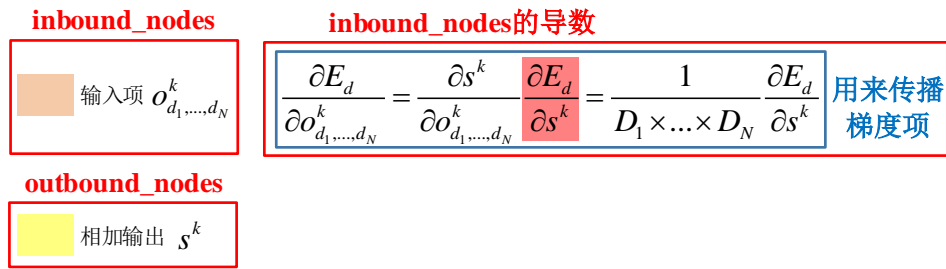
$$s_{1 \times 1}^k = \frac{\text{sum} \left(\left[\mathbf{o}^k \right]_{D_1 \times \dots \times D_N} \right)}{D_1 \times \dots \times D_N} \quad (1.31)$$

由前文知，Squared 类中同样有 inbound_nodes 和 outbound_nodes，详细如下图所示，为了方便对比，其中 inbound_nodes 和 outbound_nodes 中不同项所对应的方框颜色与上图中的方框颜色一致。且正如下图所示，outbound_nodes 的导数不要求，因为在当前类中没有可以用来求其导数的信息，并且当前节点的 outbound_nodes 是下一个节点的 inbound_nodes，所以 outbound_nodes 的导数在下一个节点中被计算。

此外，inbound_nodes 的导数被表示为当前类中的输入项对 inbound_nodes 的导数与 **损失函数对输出项偏导数**【如下图所示， $\frac{\partial E_d}{\partial \mathbf{s}^k}$ 】的乘积，其中当前类中的输入项是已知的，而 $\frac{\partial E_d}{\partial \mathbf{s}^k}$ 并不知，但 $\frac{\partial E_d}{\partial \mathbf{s}^k}$ 具体的值可通过访问下一个神经元对输入项的偏导数得到。

reduce_mean 类中的输入项有 1 个，对输入项的偏导数如下图所示，且 $\frac{\partial E_d}{\partial \mathbf{o}_{index}^k}$ 是用来传播梯度项的，其中的 index

表示任意维度下的下标。



为了方便编程实现，将上图中的偏导数写成矩阵运算的形式，对于 reduce_mean 输出为 1*1 的情况，有：

$$\left[\frac{\partial E_d}{\partial o^k} \right]_{D_1 \times \dots \times D_N} = \frac{1}{D_1 \times \dots \times D_N} [1]_{D_1 \times \dots \times D_N} \frac{\partial E_d}{\partial s^k} \quad (1.32)$$

上式中的total_num为维数的总大小。上图中详细的符号定义及代码推导请参考【[深度学习基础模型算法原理及编程实现--03.全链接][3]】。相应的代码实现如下：

```
def list_product(input_):
```

```
    o=1
```

```
    for i in input_:
```

```
        o*=i
```

```
    return o
```

```
class reduce_mean (Node):
```

```
    """线性加权神经元类，实现节点加权输入"""
```

```
    def __init__(self, o, name=[]):
```

```
        Node.__init__(self, inbound_nodes=[o], name=name)
```

```
    def forward(self):
```

```
        self.value=np.mean(self.inbound_nodes[0].value)
```

```
    def backward(self):
```

```
        self.gradients = {n: np.zeros_like(n.value) for n in self.inbound_nodes}
```

```
        if(self.outbound_nodes==[]):
```

```
            grad_cost = 1.
```

```
            self.gradients[self.inbound_nodes[0]] = np.ones_like(self.inbound_nodes[0].value)*grad_cost/list_product
```

```
(self.inbound_nodes[0].value.shape)
```

```
        else:
```

```
            for n in self.outbound_nodes:
```

```
                grad_cost = n.gradients[self]
```

```
                self.gradients[self.inbound_nodes[0]]
```

```
                np.ones_like(self.inbound_nodes[0].value)*grad_cost/list_product (self.inbound_nodes[0].value.shape)
```

+=

2.7 Node 类添加运算符重载

2.7.1 加号

```
class Node(object):
```

```
...
```

```
    def __add__(self, other):
```

```
        return Add(self, other)
```

2.7.2 减号

```
class Node(object):
```

...

```
def __sub__(self, other):
    return Sub(self, other)
```

2.8 激活函数类

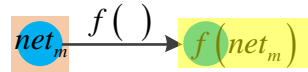
激活函数主要实现下式中的运算：

$$y_m^k = f(\text{net}_m^k) \quad (1.33)$$

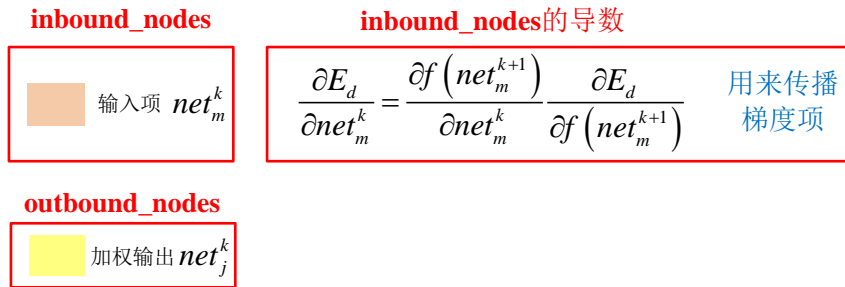
忽略层数 k 的影响，上式可整理为：

$$y_m = f(\text{net}_m) \quad (1.34)$$

下面再通过简单的图示对激活函数对输入的变换进行说明：



由前文知，激活函数类中同样有 `inbound_nodes` 和 `outbound_nodes`，详细如下图所示，为了方便对比，其中 `inbound_nodes` 和 `outbound_nodes` 中不同项所对应的方框颜色与上图中的方框颜色一致。



上图中详细的符号定义及代码推导请参考【深度学习基础模型算法原理及编程实现--03.全链接[3]】。

2.8.1 激活函数为 sigmoid

$$\frac{\partial f(\text{net}_m^k)}{\partial \text{net}_m^k} = f(\text{net}_m^k)(1 - f(\text{net}_m^k)) \quad (1.35)$$

为了方便编程实现，将上图中对输入项的偏导数写成矩阵运算的形式，有：

$$\left[\frac{\partial E_d}{\partial \text{net}^k} \right]_{\text{batchsize} \times N^k} = \left(\left[\frac{\partial f(\text{net}^k)}{\partial \text{net}^k} \right]_{\text{batchsize} \times N^k} \right) \otimes \left(\left[f(\text{net}^k) \right]_{\text{batchsize} \times N^k} \right) \otimes \left(\left[1 - f(\text{net}^k) \right]_{\text{batchsize} \times N^k} \right) \quad (1.36)$$

相应的实现代码如下：

```
class Sigmoid(Node):
    """sigmoid 激活函数节点类型，实现对节点加输入的非线性变换"""
    def __init__(self, node, name=[]):
        Node.__init__(self, inbound_nodes=[node], name=name)

    def _sigmoid(self, x):
        return (1./(1+np.exp(-x)))

    def forward(self):
        self.value=self._sigmoid(self.inbound_nodes[0].value)

    def backward(self):
```

```

self.gradients = {n: np.zeros_like(n.value) for n in self.inbound_nodes}
for n in self.outbound_nodes:
    grad_cost = n.gradients[self]
    sigmoid = self.value
    self.gradients[self.inbound_nodes[0]] += sigmoid * (1 - sigmoid) * grad_cost

```

2.8.2 激活函数为 ReLU

$$\frac{\partial f(\mathbf{net}_m^k)}{\partial \mathbf{net}_m^k} = \begin{cases} 1 & \mathbf{net}_m^k > 0 \\ 0 & \mathbf{net}_m^k \leq 0 \end{cases} \quad (1.37)$$

为了方便编程实现，将上图中对输入项的偏导数写成矩阵运算的形式，有：

$$\left[\frac{\partial E_d}{\partial \mathbf{net}^k} \right]_{batchsize \times N^k} = \left(\left[\frac{\partial f(\mathbf{net}^k)}{\partial \mathbf{net}^k} \right]_{batchsize \times N^k} \right) \otimes \left(\left[\frac{\partial E_d}{\partial f(\mathbf{net}^k)} \right]_{batchsize \times N^k} \right) \quad (1.38)$$

相应的实现代码如下：

```

class ReLU(Node):
    """激活函数节点类型，实现对节点加输入的非线性变换"""
    def __init__(self, node, name=[]):
        Node.__init__(self, inbound_nodes=[node], name=name)

    def _ReLU(self, x):
        return np.array([list(map(lambda x: x if x > 0.0 else 0.0, row)) for row in x])

    def _ReLU_Derivative(self, x):
        return np.array([list(map(lambda x: 1.0 if x > 0.0 else 0.0, row)) for row in x])

    def forward(self):
        self.value = self._ReLU(self.inbound_nodes[0].value)

    def backward(self):
        self.gradients = {n: np.zeros_like(n.value) for n in self.inbound_nodes}
        for n in self.outbound_nodes:
            grad_cost = n.gradients[self]
            relu = self._ReLU_Derivative(self.inbound_nodes[0].value)
            self.gradients[self.inbound_nodes[0]] += relu * grad_cost
        # print('grad_ReLU', self.gradients[self.inbound_nodes[0]][0])

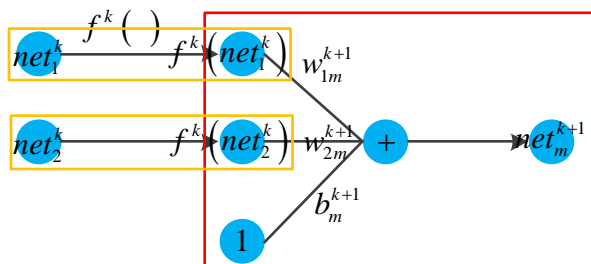
```

2.9 哪些项是用来传递梯度项的

由【深度学习基础模型算法原理及编程实现-03.全链接】中式(3.10)可得梯度项计算公式如下：

$$\delta_j^k = \sum_{m=1}^{N_{k+1}} \delta_m^{k+1} \frac{\partial f^k(\mathbf{net}_j^k)}{\partial \mathbf{net}_j^k} w_{jm}^{k+1} \quad (1.39)$$

以下图为例，图中有两个激活函数实例及一个线性加权实例构成了局部神经网络。



通过链式法则可以算出 k 层第 2 个加权输入的偏导数为：

$$\delta_2^k = \frac{\partial E_d}{\partial net_2^k} = \frac{\partial E_d}{\partial net_m^{k+1}} \frac{\partial net_m^{k+1}}{\partial net_2^k} = \delta_m^{k+1} \frac{\partial \left(\sum_{i=1}^{N_k} f^k(net_i^k) w_{im}^{k+1} + b_m^{k+1} \right)}{\partial net_2^k} = \frac{\partial f^k(net_2^k)}{\partial net_2^k} w_{jm}^{k+1} \delta_m^{k+1} \quad (1.40)$$

由于下一层中的节点可能有多个，即 m 不唯一，并将上式中求对 k 层第 2 个加权输入的偏导数一般化，有：

$$\delta_j^k = \sum_{m=1}^{N_m^{k+1}} \frac{\partial f^k(net_j^k)}{\partial net_j^k} w_{jm}^{k+1} \delta_m^{k+1} \quad (1.41)$$

这与【深度学习基础模型算法原理及编程实现-03.全链接】中式(3.10)一样。其中 $\frac{\partial f^k(net_j^k)}{\partial net_j^k}$ 由激活函数层提供， w_{jm}^{k+1} 由线性加权层对输入的偏导数提供。

由线性加权层对输入的偏导数提供。

2.10 损失函数类

2.10.1 softmax_cross_entropy_with_logits 损失函数类

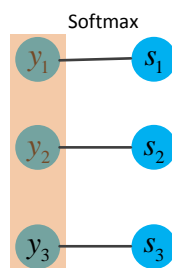
softmax_cross_entropy_with_logits 损失函数可进一步分解为 softmax 操作及交叉熵损失函数，下面分别对其过程进行描述。

2.10.1.1 softmax 操作

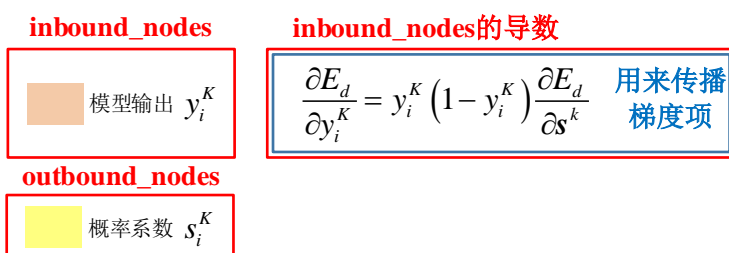
softmax 操作主要实现下式中的运算：

$$s_i^K = \exp(y_i^K) / \sum_{i=1}^{N^K} \exp(y_i^K) \quad (1.42)$$

下面再通过简单的图示对激活函数对输入的变换进行说明：



由前文知，softmax操作类中同样有inbound_nodes和outbound_nodes，详细如下图所示，为了方便对比，其中inbound_nodes和outbound_nodes中不同项所对应的方框颜色与上图中的方框颜色一致。



为了方便编程实现，将上图中对输入项的偏导数写成矩阵运算的形式，有：

$$\left[\frac{\partial E_d}{\partial \mathbf{y}^k} \right]_{batchsize \times N^k} = \left(\left[\frac{\partial E_d}{\partial \mathbf{s}^k} \right]_{batchsize \times N^k} \right) \otimes \left(\left[\mathbf{y}^k \right]_{batchsize \times N^k} \right) \otimes \left(\left[1 - \mathbf{y}^k \right]_{batchsize \times N^k} \right) \quad (1.43)$$

上图中详细的符号定义及代码推导请参考【[深度学习基础模型算法原理及编程实现--03.全链接][3]】。相应的实现代码如下：

```
class softmax(Node):
    """softmax 是 Node 的另一个子类，实现对 softmax 操作"""
    def __init__(self, x, name=[]):
        Node.__init__(self, inbound_nodes=[x], name=name) # 调用 Node 的构造函数
        self.is_eff = True

    # def forward(self):
    #     logits = self.inbound_nodes[0].value
    #     logits = np.exp(logits - np.max(logits,1).reshape(-1,1))
    #     self.value=np.zeros_like(logits)
    #     for i in range(logits.shape[0]):
    #         logsum = np.sum(logits[i,:])
    #         self.value[i,:] = logits[i,:]/logsum

    def forward(self):
        logits = self.inbound_nodes[0].value
        logits=np.exp(logits - np.max(logits,1,keepdims=1))
        logsum=np.sum(logits,1,keepdims=1)
        self.value=logits/logsum

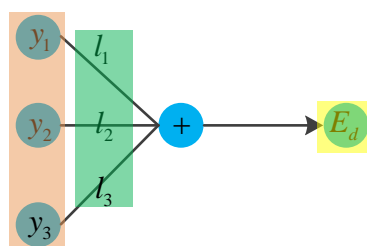
    def backward(self):
        self.gradients = {n: np.zeros_like(n.value) for n in self.inbound_nodes}
        for n in self.outbound_nodes:
            grad_cost = n.gradients[self]
            self.gradients[self.inbound_nodes[0]] += self.value*(1. - self.value)*grad_cost
```

2.10.1.2 交叉熵损失函数

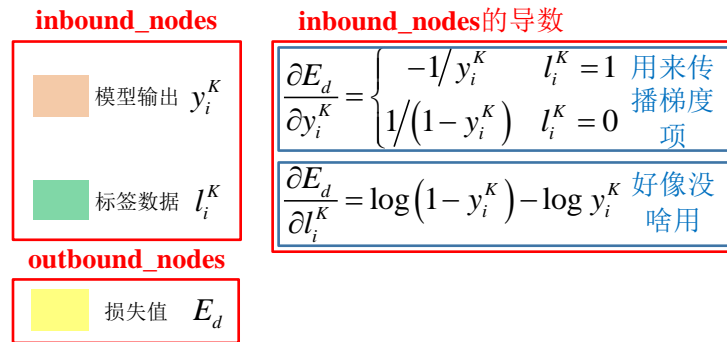
softmax_cross_entropy_with_logits 损失函数主要实现下式中的运算：

$$E_d = - \sum_{i=1}^{N_k} \left[l_i \log y_i^K + (1 - l_i) \log (1 - y_i^K) \right] \quad (1.44)$$

下面再通过简单的图示对激活函数对输入的变换进行说明：



由前文知，损失函数类中同样有 inbound_nodes 和 outbound_nodes，详细如下图所示，为了方便对比，其中 inbound_nodes 和 outbound_nodes 中不同项所对应的方框颜色与上图中的方框颜色一致。



为了方便编程实现，将上图中对输入项的偏导数写成矩阵运算的形式，有：

$$\begin{aligned} \left[\frac{\partial E_d}{\partial \mathbf{y}^k} \right]_{batchsize \times N^k} &= \left[\frac{1 - \mathbf{l}^k}{1 - \mathbf{y}^k} - \frac{\mathbf{l}^k}{\mathbf{y}^k} \right]_{batchsize \times N^k} = \left[\frac{\mathbf{y}^k - \mathbf{l}^k}{(1 - \mathbf{y}^k) \mathbf{y}^k} \right]_{batchsize \times N^k} \\ \left[\frac{\partial E_d}{\partial \mathbf{l}^k} \right]_{batchsize \times N^k} &= \left[\log(\mathbf{I} - \mathbf{y}^k) - \log \mathbf{y}^k \right]_{batchsize \times N^k} \end{aligned} \quad (1.45)$$

上图中详细的符号定义及代码推导请参考【[深度学习基础模型算法原理及编程实现-03.全链接](#)][3]】。相应的实现代码如下：

```
class cross_entropy_with_logits(Node):
```

```
    """cross_entropy_with_logits 是 Node 的另一个子类，实现对交叉熵损失函数"""
```

```
    def __init__(self, labels, logits, name=[]):
```

```
        Node.__init__(self, inbound_nodes=[labels, logits], name=name) # 调用 Node 的构造函数
```

```
        self.is_eff = True
```

```
#     def forward(self):
```

```
#         labels=self.inbound_nodes[0].value
```

```
#         logits=self.inbound_nodes[1].value
```

```
#         self.m = labels.shape[0]
```

```
#         self.value = np.zeros(labels.shape[0])
```

```
#         self.diff = np.zeros(labels.shape)
```

```
#         for i in range(self.m):
```

```
#             ind = np.argmax(labels[i,:])
```

```
#             self.value[i] = -1.*np.log(logits[i][ind]+1e-5)-np.sum(np.log(1-logits[i][:]+1e-5))+np.log(1-logits[i][ind]+1e-5)
```

```
#             self.diff[i,:] = 1/(1-logits[i])
```

```
#             self.diff[i,ind] = -1/logits[i][ind]
```

```
def forward(self):
```

```
    labels=self.inbound_nodes[0].value
```

```
    logits=self.inbound_nodes[1].value
```

```
    self.m = labels.shape[0]
```

```
    self.value = np.zeros(labels.shape[0])
```

```
    self.diff = np.zeros(labels.shape)
```

```
    self.value=np.sum(-labels*np.log(logits)-(1-labels)*np.log(1-logits),1)
```

```
    self.diff=(logits-labels)/(logits*(1-logits))
```

```
def backward(self):
```

```
    self.gradients = {}
```

```
self.gradients[self.inbound_nodes[1]]=self.diff
```

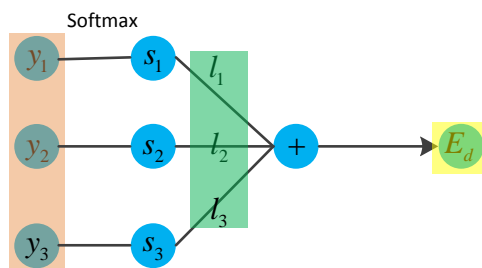
2.10.1.3 softmax_cross_entropy_with_logits 损失函数

softmax_cross_entropy_with_logits 损失函数主要实现下式中的运算：

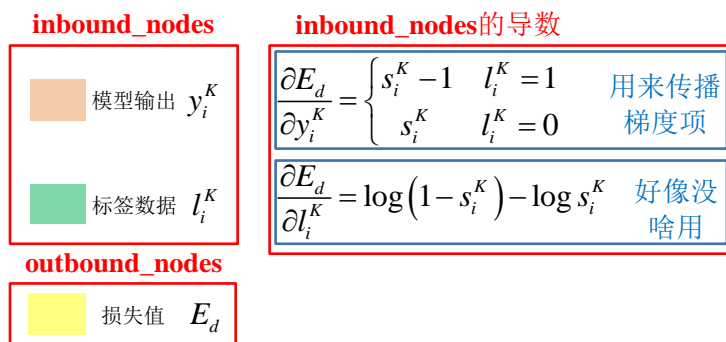
$$s_i^K = \exp(y_i^K) / \sum_{i=1}^{N^K} \exp(y_i^K)$$

$$E_d = -\sum_{i=1}^{N^K} (l_i \log s_i^K + (1-l_i) \log(1-s_i^K))$$
(1.46)

下面再通过简单的图示对激活函数对输入的变换进行说明：



由前文知，损失函数类中同样有 inbound_nodes 和 outbound_nodes，详细如下图所示，为了方便对比，其中 inbound_nodes 和 outbound_nodes 中不同项所对应的方框颜色与上图中的方框颜色一致。



为了方便编程实现，将上图中对输入项的偏导数写成矩阵运算的形式，有：

$$\left[\frac{\partial E_d}{\partial \mathbf{y}^k} \right]_{batchsize \times N^k} = \left[\mathbf{s}^k - \mathbf{I}^k \right]_{batchsize \times N^k}$$

$$\left[\frac{\partial E_d}{\partial \mathbf{l}^k} \right]_{batchsize \times N^k} = - \left[\log \mathbf{s}^k \right]_{batchsize \times N^k}$$
(1.47)

上图中详细的符号定义及代码推导请参考【深度学习基础模型算法原理及编程实现--03.全链接】[3]】。相应的实现代码如下：

```
class softmax_cross_entropy_with_logits(Node):
    def __init__(self, labels, logits, name = []):
        """softmax 交叉熵损失函数"""
        Node.__init__(self, inbound_nodes=[labels, logits], name=name)
        self.is_eff = True

    def softmax(self, logits):
        logits = np.exp(logits)
        for i in range(logits.shape[0]):
            logits[i, :] = logits[i, :] / np.sum(logits[i, :])
        return logits
```

```

# def forward(self):
#     logits = self.inbound_nodes[1].value
#     logits = np.exp(logits - np.max(logits,1).reshape(-1,1))
#     logits_ = np.zeros(logits.shape)
#     self.m = self.inbound_nodes[0].value.shape[0]
#     self.value = np.zeros(self.inbound_nodes[0].value.shape[0])
#     self.diff = np.zeros(self.inbound_nodes[0].value.shape)
#     for i in range(logits.shape[0]):
#         logsum = np.sum(logits[i,:])
#         logits_[i,:] = logits[i,:]/(logsum+1e-5)
#         ind = np.argmax(self.inbound_nodes[0].value[i])
#         self.value[i] =
-1.*np.log(logits_[i][ind]+1e-5)-np.sum(np.log(1-logits_[i][:]+1e-5))+np.log(1-logits_[i][ind]+1e-5)
#         self.diff[i,:] = logits_[i]-self.inbound_nodes[0].value[i]

```

```

def forward(self):
    labels=self.inbound_nodes[0].value
    logits=self.inbound_nodes[1].value
    logits=np.exp(logits - np.max(logits,1,keepdims=1))
    logits_=np.zeros(logits.shape)
    self.m=self.inbound_nodes[0].value.shape[0]
    self.value=np.zeros(labels.shape[0])
    logsum=np.sum(logits,1,keepdims=1)
    logits_=logits/logsum
    self.value=np.sum(-labels*np.log(logits_)-(1-labels)*np.log(1-logits_),1)
    self.diff=logits_-self.inbound_nodes[0].value

```

```

def backward(self):
    self.gradients = {}
    self.gradients[self.inbound_nodes[1]]=self.diff

```

2.10.2 均方误差损失函数类

2.10.2.1 实现方式 01

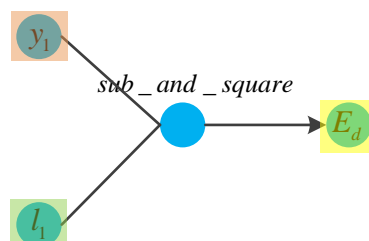
与 tensorflow 一样，即 `tf.reduce_mean(tf.squared(y_ - y))`，`reduce_mean` 及 `squared` 的定义如前文所示。

2.10.2.2 实现方式 02

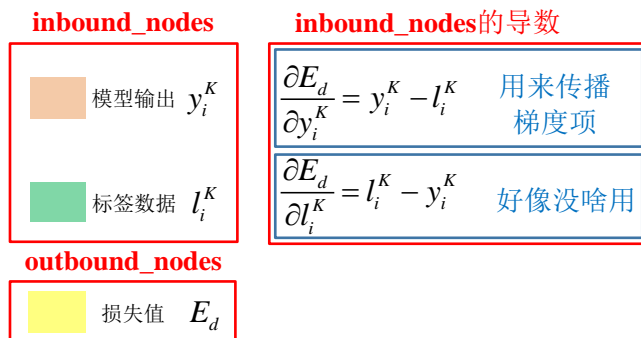
均方误差损失函数主要实现下式中的运算：

$$E_d = \frac{1}{2} \sum_{i=1}^{N_k} (l_i - y_i^K)^2 = \frac{1}{2} (\|L - Y^K\|_2)^2 \quad (1.48)$$

下面再通过简单的图示对激活函数对输入的变换进行说明：



由前文知，损失函数类中同样有 `inbound_nodes` 和 `outbound_nodes`，详细如下图所示，为了方便对比，其中 `inbound_nodes` 和 `outbound_nodes` 中不同项所对应的方框颜色与上图中的方框颜色一致。



为了方便编程实现，将上图中对输入项的偏导数写成矩阵运算的形式，有：

$$\begin{aligned} \left[\frac{\partial E_d}{\partial \mathbf{y}^k} \right]_{batchsize \times N^k} &= \frac{1}{batchsize} \left([\mathbf{y}^k]_{batchsize \times N^k} - [\mathbf{l}^k]_{batchsize \times N^k} \right) \\ \left[\frac{\partial E_d}{\partial \mathbf{l}^k} \right]_{batchsize \times N^k} &= \frac{1}{batchsize} \left([\mathbf{l}^k]_{batchsize \times N^k} - [\mathbf{y}^k]_{batchsize \times N^k} \right) \end{aligned} \quad (1.49)$$

上图中详细的符号定义及代码推导请参考【[深度学习基础模型算法原理及编程实现--03.全链接](#)][3]】。相应的实现代码如下：

```
class MSE(Node):
    def __init__(self, labels, logits, name = []):
        """均方误差损失函数"""
        Node.__init__(self, inbound_nodes=[labels, logits], name=name)

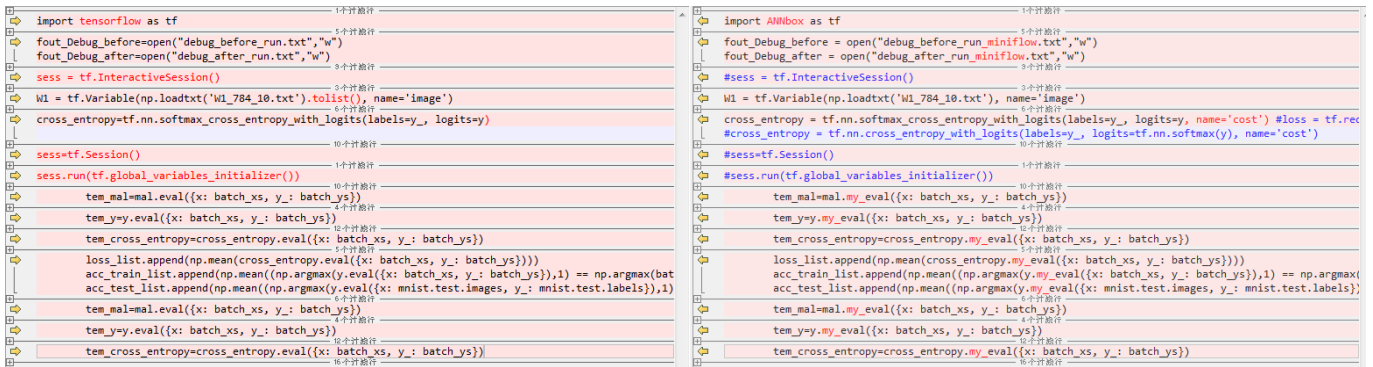
    def forward(self):
        labels = self.inbound_nodes[0].value
        logits = self.inbound_nodes[1].value
        self.m = self.inbound_nodes[0].value.shape[0]
        self.diff = labels - logits
        # self.value = np.mean(self.diff**2,1)
        self.value = np.sum(self.diff**2,1)

    def backward(self):
        self.gradients = {}
        # self.gradients[self.inbound_nodes[0]]=2./self.m*self.diff
        self.gradients[self.inbound_nodes[1]]=-2./self.m*self.diff
```

2.11 编程实现

2.11.1 验证算例

为了对比分别调用 `tensorflow` 和 `ANNbox` 库函数计算结果的差别，分别用同一段程序调用 `tensorflow` 库及 `ANNbox` 库函数对比计算结果。详细代码文件在：中，`beyond compare` 对比两个版本的计算程序如下：



01.其中第一个差异是调用库函数的差异:

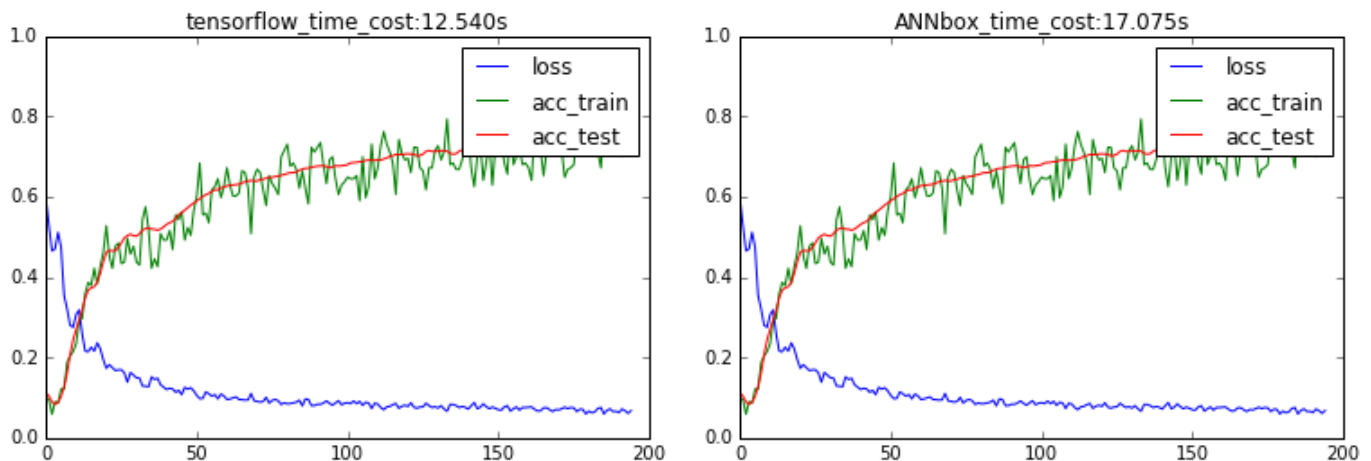
import **tensorflow** as tf

import **ANNbox** as tf

02.第二个差异是 tensorflow 中需建立会话 `sess = tf.InteractiveSession()`, 而 ANNbox 则无需建立【其实我还没有弄明白会话的作用, 说以在 ANNbox 中并没有建立此功能, 但并不影响最终的结果】

03.第三个差异是全局变量初始化, 我在 ANNbox 中用 `tf.global_variables_initializer().run()` 来实现。

04.第 4 个差异是值评估函数的差异。Tensorflow 中用的函数名是 `eval`, 而 ANNbox 中用的是 `my_eval`
最终的运行结果如下:



从过程记录文件中的数据对比分别调用 ANNbox 库函数及 tensorflow 库函数的程序在测试数据集上的准确率:

ANNbox,acc_test_list:[0.1993, 0.3639, 0.5266, 0.6176, 0.6968, 0.7338, 0.7534, 0.7698, 0.7836, 0.8034,...]

tensorflow,acc_test_list:[0.1993, 0.3639, 0.5266, 0.6176, 0.6968, 0.7338, 0.7534, 0.7698, 0.7836, 0.8034,...]

发现结果是一致的。所以证明了 ANNbox 库函数的有效性。

2.11.2 算例 1

上面的程序只有一层隐藏层, 这里我们使用两层隐藏层, 且第一层隐层的激活函数 `relu`, 第二层为全等映射, 且损失函数为 `softmax_cross_entropy_with_logits`, 在测试数据集上的准确率达到 95%。两个版本程序的差别也就是上一算例中列出的 4 类差别。

2.11.2.1 ANNbox 版本

```
from tensorflow.examples.tutorials.mnist import input_data
import ANNbox as tf
import matplotlib.pyplot as plt
import numpy as np
import sys
import time
#loaddata
mnist = input_data.read_data_sets("./MNISTDat", one_hot=True)
#sess = tf.InteractiveSession()
```

```

# Create the model
in_units = 784
h1_units = 300
o_units = 10
W1=tf.Variable(tf.truncated_normal([in_units, h1_units], stddev=0.1))
b1=tf.Variable(tf.zeros([h1_units]))
x=tf.placeholder(tf.float32,[None,in_units])
hidden1=tf.nn.relu(tf.matmul(x,W1)+b1)
W2=tf.Variable(tf.zeros([h1_units, o_units]))
b2=tf.Variable(tf.zeros([o_units]))
y=tf.matmul(hidden1,W2)+b2
y_ = tf.placeholder(tf.float32, [None, o_units])
#cross_entropy = tf.reduce_sum(tf.nn.softmax_cross_entropy_with_logits(labels=y_, logits=y))
cross_entropy = tf.nn.softmax_cross_entropy_with_logits(labels=y_, logits=y, name='cost') #loss = tf.reduce_mean(cost)
#cross_entropy = tf.nn.cross_entropy_with_logits(labels=y_, logits=tf.nn.softmax(y), name='cost')
epochs = 2
m = 50000
batch_size = 64*2*2
learning_rate=2e-4
Momentum_rate=0.9
steps_per_epoch = m // batch_size
#train_step = tf.train.GradientDescentOptimizer(learning_rate).minimize(cross_entropy)
train_step = tf.train.MomentumOptimizer(learning_rate,Momentum_rate).minimize(cross_entropy)
#train_step = tf.train.AdagradOptimizer(0.3).minimize(cross_entropy)
start_time=time.time()
#sess=tf.Session()
tf.global_variables_initializer().run()
#sess.run(tf.global_variables_initializer())
loss_list,acc_train_list,acc_test_list=[],[],[]
for _ in range(epochs):
    for i in range(steps_per_epoch):
        # batch_xs=mnist.train.images[i*batch_size:(i+1)*batch_size]
        # batch_ys=mnist.train.labels[i*batch_size:(i+1)*batch_size]
        batch_xs,batch_ys=mnist.train.next_batch(batch_size)
        '''train'''
        train_step.run({x: batch_xs, y_: batch_ys})
        loss_list.append(np.mean(cross_entropy.my_eval({x: batch_xs, y_: batch_ys})))
        acc_train_list.append(np.mean((np.argmax(y.my_eval({x: batch_xs, y_: batch_ys}),1) ==
np.argmax(batch_ys,1)).astype(int)))
        acc_test_list.append(np.mean((np.argmax(y.my_eval({x: mnist.test.images, y_: mnist.test.labels}),1) ==
np.argmax(mnist.test.labels,1)).astype(int)))
        sys.stdout.write("\rprocess: {}/ {}, loss:{:.5f}, acc_train:{:.2f}, acc_test:{:.2f}".format(i, steps_per_epoch,
loss_list[-1], acc_train_list[-1], acc_test_list[-1]))
    plt.figure()
    # plt.subplot(211)
    plt.plot(range(len(loss_list)),loss_list,label=u'loss')
    # plt.subplot(212)

```

```
plt.plot(range(len(loss_list)),acc_train_list,label=u'acc_train')
plt.plot(range(len(loss_list)),acc_test_list,label=u'acc_test')
plt.ylim([0,1])
plt.title('ANNbox')
plt.legend()
plt.show()
end_time = time.time()
print('total_time:',end_time-start_time)
```

2.11.2.2 Tensorflow 版本

```
from tensorflow.examples.tutorials.mnist import input_data
import tensorflow as tf
import matplotlib.pyplot as plt
import numpy as np
import sys
import time
#loaddata
mnist = input_data.read_data_sets("./MNISTDat", one_hot=True)
sess = tf.InteractiveSession()
# Create the model
in_units = 784
h1_units = 300
o_units = 10
W1 = tf.Variable(tf.truncated_normal([in_units, h1_units], stddev=0.1))
b1 = tf.Variable(tf.zeros([h1_units]))
x = tf.placeholder(tf.float32, [None, in_units])
hidden1=tf.nn.relu(tf.matmul(x,W1)+b1)
W2=tf.Variable(tf.zeros([h1_units, o_units]))
b2=tf.Variable(tf.zeros([o_units]))
y=tf.matmul(hidden1,W2)+b2
y_ = tf.placeholder(tf.float32, [None, o_units])
#cross_entropy = tf.reduce_sum(tf.nn.softmax_cross_entropy_with_logits(labels=y_, logits=y))
cross_entropy=tf.nn.softmax_cross_entropy_with_logits(labels=y_, logits=y)

epochs = 2
m = 50000
batch_size = 64*2*2
learning_rate=2e-4
Momentum_rate=0.9
steps_per_epoch = m // batch_size
#train_step = tf.train.GradientDescentOptimizer(learning_rate).minimize(cross_entropy)
train_step = tf.train.MomentumOptimizer(learning_rate,Momentum_rate).minimize(cross_entropy)
#train_step = tf.train.AdagradOptimizer(0.3).minimize(cross_entropy)
start_time=time.time()
sess=tf.Session()
tf.global_variables_initializer().run()
sess.run(tf.global_variables_initializer())
loss_list,acc_train_list,acc_test_list=[],[],[]
```



```

for _ in range(epochs):
    for i in range(steps_per_epoch):
        # batch_xs=mnist.train.images[i*batch_size:(i+1)*batch_size]
        # batch_ys=mnist.train.labels[i*batch_size:(i+1)*batch_size]
        batch_xs,batch_ys=mnist.train.next_batch(batch_size)
        '''train'''
        train_step.run({x: batch_xs, y_: batch_ys})
        loss_list.append(np.mean(cross_entropy.eval({x: batch_xs, y_: batch_ys})))
        acc_train_list.append(np.mean((np.argmax(y.eval({x: batch_xs, y_: batch_ys}),1) ==
np.argmax(batch_ys,1)).astype(int)))
        acc_test_list.append(np.mean((np.argmax(y.eval({x: mnist.test.images, y_: mnist.test.labels}),1) ==
np.argmax(mnist.test.labels,1)).astype(int)))
        sys.stdout.write("\rprocess: {}/ {}, loss:{:.5f}, acc_train:{:.2f}, acc_test:{:.2f}".format(i, steps_per_epoch,
loss_list[-1], acc_train_list[-1], acc_test_list[-1]))
plt.figure()
# plt.subplot(211)
plt.plot(range(len(loss_list)),loss_list,label=u'loss')
# plt.subplot(212)
plt.plot(range(len(loss_list)),acc_train_list,label=u'acc_train')
plt.plot(range(len(loss_list)),acc_test_list,label=u'acc_test')
plt.ylim([0,1])
plt.title('tensorflow')
plt.title('ANNbox')
plt.legend()
plt.show()
end_time = time.time()
print('total_time:',end_time-start_time)

```

2.11.3 算例 2

```

# -*- coding: utf-8 -*-
import matplotlib.pyplot as plt
import numpy as np
import random
from sklearn.datasets import load_boston
from sklearn.utils import shuffle, resample
import sys
from miniFlow import *
from tensorflow.examples.tutorials.mnist import input_data
# Load data
mnist = input_data.read_data_sets("./MNISTDat", one_hot=True)

in_units = 784
h1_units = 300
o_units = 10
random.seed(1)
#####
W1_ = 1.*np.random.randn(in_units, h1_units)
b1_ = np.zeros(h1_units)

```

```

W2_ = 0.0*np.random.randn(h1_units, o_units)
#W2_ = 1*np.zeros([h1_units, o_units])
b2_ = np.zeros(o_units)
# Neural network
X, y = placeholder(name='X'), placeholder(name='y')
W1, b1 = variable(value=W1_, name='W1'), variable(value=b1_, name='b1')
W2, b2 = variable(value=W2_, name='W2'), variable(value=b2_, name='b2')
hidden1 = ReLU(Linear(X, W1, b1))
out = Linear(hidden1, W2, b2) #out = ReLU(Linear(hidden1, W2, b2))
cost = softmax_cross_entropy_with_logits(labels=y, logits=out, name='cost') #loss = tf.reduce_mean(cost)

epochs = 1
m = 50000
batch_size = 64*2
learning_rate=1e-3
steps_per_epoch = m // batch_size
#####
#W1_ = 1./np.sqrt(in_units*h1_units)*np.random.randn(in_units, h1_units)
#b1_ = np.zeros(h1_units)
#W2_ = 0.1*np.random.randn(h1_units, o_units)
#b2_ = np.zeros(o_units)
## Neural network
#X, y = placeholder(name='X'), placeholder(name='y')
#W1, b1 = variable(value=W1_, name='W1'), variable(value=b1_, name='b1')
#W2, b2 = variable(value=W2_, name='W2'), variable(value=b2_, name='b2')
##hidden1 = Sigmoid(Linear(X, W1, b1))
##out = Sigmoid(Linear(hidden1, W2, b2))
#hidden1 = ReLU(Linear(X, W1, b1))
#out = ReLU(Linear(hidden1, W2, b2),name='out')
#cost = MSE(y, out)
#epochs = 2
#m = 50000
#batch_size = 64
#learning_rate=2e-2
#steps_per_epoch = m // batch_size
#####

print("Total number of examples = {}".format(m))

global_variables_initializer()
loss_list = []
acc_list = []
for i in range(epochs):
    loss = 0
    for j in range(steps_per_epoch):
        X_batch, y_batch = mnist.train.next_batch(batch_size)
        feed_dict = {X: X_batch,y: y_batch.reshape(batch_size,-1)}

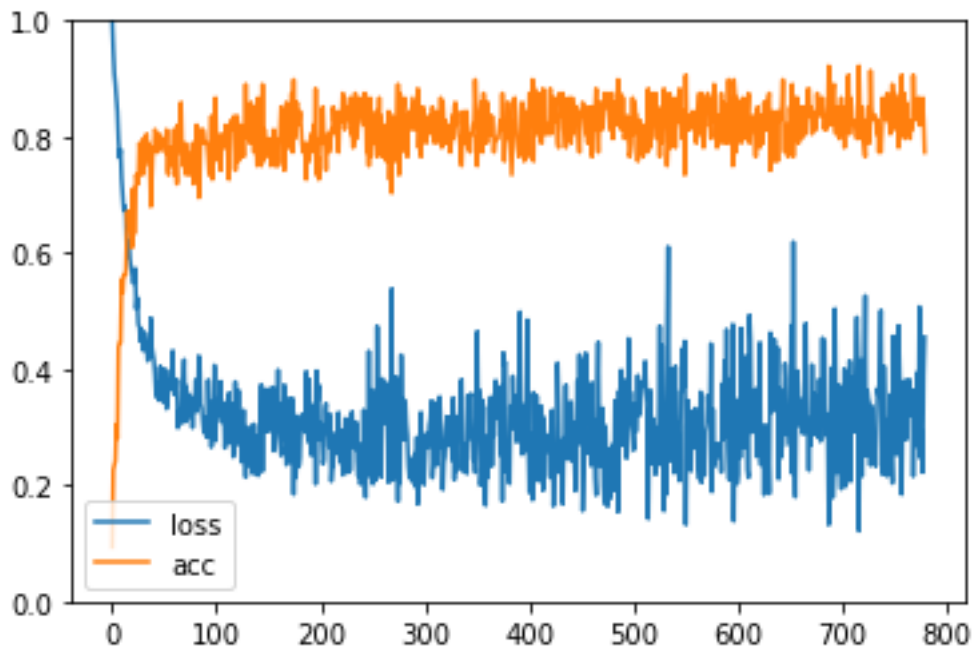
```

```

graph = forward_and_backward(feed_dict)
sgd_update(learning_rate)
#     print('graph[-11].value[5,5:8]:',graph[-8].value[5,5:8],'loss:',np.mean(graph[-1].value))
loss = np.mean(graph[-1].value)
acc = np.mean((np.argmax(out.value,1) == np.argmax(y_batch,1)).astype(int))
loss_list.append(loss)
acc_list.append(acc)
sys.stdout.write("\rprocess: {}/ {}, loss:{}, acc:{}".format(j, steps_per_epoch, loss, acc))

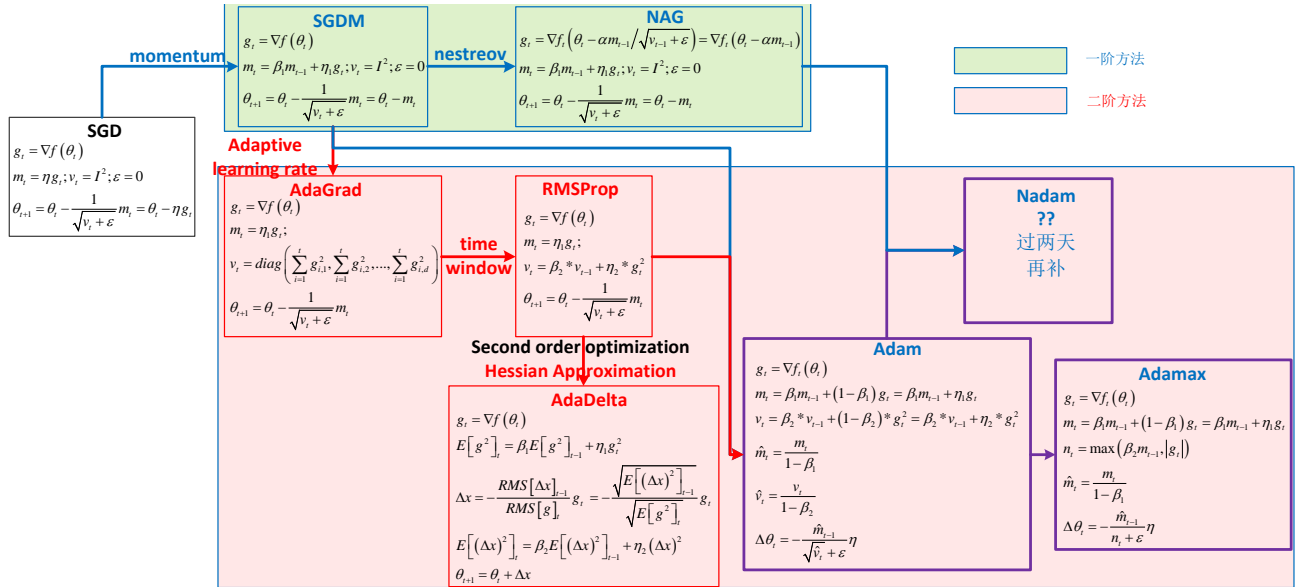
plt.figure()
#     plt.subplot(211)
plt.plot(range(len(loss_list)),loss_list,label=u'loss')
#     plt.subplot(212)
plt.plot(range(len(loss_list)),acc_list,label=u'acc')
plt.ylim([0,1])
plt.legend()
plt.show()

```



3 自己编写神经网络库 ANNbox【高仿 tensorflow】_02 实现不同的优化方法

本节主要描述不同优化方法在 MiniFlow 中的实现方法，具体的公式及数学原理可详参【深度学习基础模型算法原理及编程实现--10.优化方法：从梯度下降到 NAdam】。将不同优化方法类单独写在 optimization.py 文件中。优化算法的出现顺序从前往后依次为：SGD、SGDM、NAG、AdaGrad、AdaDelta、Adam、Nadam，本文会依次实现这些优化方法，这些优化算法的演变其实都来自一个模型，其演化路径如下图所示：



3.1 实现 tf.train.GradientDescentOptimizer 类_SGD

在【构建简化版的 tensorflow---MiniFlow，并实现 MLP 对 MNIST 数据进行分类】中通过设置超参数 batchsize 可分别实现：批量梯度下降法（Batch Gradient Descent）、随机梯度下降法（Stochastic Gradient Descent）、小批量梯度下降法（Mini-batch Gradient Descent）。这里再将实现公式及程序描述如下。最简单的梯度下降法：

$$w_{ij}^k \leftarrow w_{ij}^k - \eta \frac{\partial E_d}{\partial w_{ij}^k} \quad k=1, \dots, K, i=1, \dots, N_{k-1}, j=1, \dots, N_k \quad (1.50)$$

$$b_i^k \leftarrow b_i^k - \eta \frac{\partial E_d}{\partial b_i^k}$$

整理成向量形式：

$$\mathbf{W}^k \leftarrow \mathbf{W}^k - \eta \frac{\partial E_d}{\partial \mathbf{W}^k} \quad (1.51)$$

$$\mathbf{b}^k \leftarrow \mathbf{b}^k - \eta \frac{\partial E_d}{\partial \mathbf{b}^k}$$

由前文分析知，对全连接而言，不论批采样大小是多少， $\frac{\partial E_d}{\partial \mathbf{W}^k}$ 的大小恒为 $[N^k \times N^{k+1}]$ ， $\frac{\partial E_d}{\partial \mathbf{b}^k}$ 的大小恒为 $[1 \times N^{k+1}]$ ，

与 batchsize 无关，但 $\frac{\partial E_d}{\partial \mathbf{W}^k}$ 及 $\frac{\partial E_d}{\partial \mathbf{b}^k}$ 具体的值是与 batchsize 相关的，batchsize 个训练数据对应的偏导数的和。所以

这里的优化方法由 batchsize 决定，batchsize 为 1 的话就是随机梯度下降，batchsize 为所有训练数据样本个数的话就是批量梯度下降，batchsize 介于两者之间的话就是小批量梯度下降。

```
class GradientDescentOptimizer(Node):
```

```
    def __init__(self, learning_rate=1e-2):
        self.learning_rate=learning_rate
```

```

def minimize(self,loss):
    """暂时不知道怎么联系起来"""
    self.loss=loss
    return self

def run(self,feed_dict):
#         print('optimization_forward_and_backward')
    self.loss.check_graph(feed_dict)
    graph = forward_and_backward(feed_dict,self.loss.L)

    """SGD_update"""
    for t in trainables:
        partial = t.gradients[t]
        t.value -= self.learning_rate * partial
    return graph

```

3.1.1 仿真算例

```

from tensorflow.examples.tutorials.mnist import input_data
import ANNbox as tf
import matplotlib.pyplot as plt
import numpy as np
import sys
import time

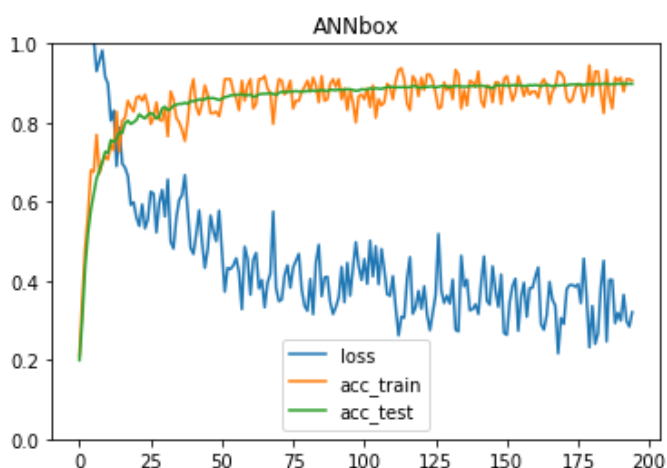
#loaddata
mnist = input_data.read_data_sets("./MNISTData", one_hot=True)
#sess = tf.InteractiveSession()
# Create the model
in_units = 784
o_units = 10
W1 = tf.Variable(np.loadtxt('W1_784_10.txt'), name='image')
b1 = tf.Variable(tf.zeros([o_units]))
x = tf.placeholder(tf.float32, [None, in_units])
mal = tf.matmul(x, W1)
y = mal + b1
y_ = tf.placeholder(tf.float32, [None, o_units])
#cross_entropy = tf.reduce_sum(tf.nn.softmax_cross_entropy_with_logits(labels=y_, logits=y))
cross_entropy = tf.nn.softmax_cross_entropy_with_logits(labels=y_, logits=y, name='cost') #loss = tf.reduce_mean(cost)
#cross_entropy = tf.nn.cross_entropy_with_logits(labels=y_, logits=tf.nn.softmax(y), name='cost')
epochs = 1
m = 50000
batch_size = 64*2*2
learning_rate=1e-3
Momentum_rate=0.5
steps_per_epoch = m // batch_size
train_step = tf.train.GradientDescentOptimizer(learning_rate).minimize(cross_entropy)
#train_step = tf.train.MomentumOptimizer(learning_rate,Momentum_rate).minimize(cross_entropy)
#train_step = tf.train.AdagradOptimizer(0.3).minimize(cross_entropy)
start_time=time.time()

```

```

#sess=tf.Session()
tf.global_variables_initializer().run()
#sess.run(tf.global_variables_initializer())
loss_list,acc_train_list,acc_test_list=[],[],[]
for _ in range(epochs):
    for i in range(steps_per_epoch):
        batch_xs=mnist.train.images[i*batch_size:(i+1)*batch_size]
        batch_ys=mnist.train.labels[i*batch_size:(i+1)*batch_size]
        "train"
        train_step.run({x: batch_xs, y_: batch_ys})
        loss_list.append(np.mean(cross_entropy.my_eval({x: batch_xs, y_: batch_ys})))
        acc_train_list.append(np.mean((np.argmax(y.my_eval({x: batch_xs, y_: batch_ys}),1) ==
np.argmax(batch_ys,1)).astype(int)))
        acc_test_list.append(np.mean((np.argmax(y.my_eval({x: mnist.test.images, y_: mnist.test.labels}),1) ==
np.argmax(mnist.test.labels,1)).astype(int)))
        sys.stdout.write("\rprocess: {}/ {}, loss:{:.5f}, acc_train:{:.2f}, acc_test:{:.2f}".format(i, steps_per_epoch,
loss_list[-1], acc_train_list[-1], acc_test_list[-1]))
plt.figure()
# plt.subplot(211)
plt.plot(range(len(loss_list)),loss_list,label=u'loss')
# plt.subplot(212)
plt.plot(range(len(loss_list)),acc_train_list,label=u'acc_train')
plt.plot(range(len(loss_list)),acc_test_list,label=u'acc_test')
plt.ylim([0,1])
plt.title('ANNbox')
plt.legend()
plt.show()
end_time = time.time()
print('total_time:',end_time-start_time)

```



3.2 实现 tf.train.MomentumOptimizer 类_SGD with Momentum(SGDM)

引入前一时刻动量项来对 SGD 方法进行加速, 通过对前一时刻动量项和当前梯度线性加权, 可以产生加速收敛和减小震荡的效果 (详细的数学原理可以参考一阶 RC 低通滤波器), 这个方法就是 SGD with Momentum(SGDM)。相应的计算公式如下:

$$\begin{aligned}
 M_{w^k} &= \beta_1 M_{w^{k-1}} + \eta_1 \frac{\partial E_d}{\partial W^k} \\
 M_{b^k} &= \beta_1 M_{b^{k-1}} + \eta_1 \frac{\partial E_d}{\partial b^k} \\
 W^k &\leftarrow W^k - M_{w^k} \\
 b^k &\leftarrow b^k - M_{b^k}
 \end{aligned} \tag{1.52}$$

代码实现如下:

```
class MomentumOptimizer(Node):
    def __init__(self, learning_rate=1e-2, momentum_rate=0.9):
        self.learning_rate=learning_rate
        self.momentum_rate=momentum_rate
        self.isMomentumInitial=False

    def minimize(self, loss):
        """暂时不知道怎么联系起来"""
        isLEnd[0]=True
        self.loss=loss
        # print('47_isLEnd:', isLEnd[0])
        return self

    def run(self, feed_dict):
        # graph = forward_and_backward(feed_dict, self.L)
        self.loss.check_graph(feed_dict)
        # print('MomentumOptimizer 中 self.loss.L 中所含的节点')
        # for l_ in self.loss.L:
        #     print(l_.name)
        graph = forward_and_backward(feed_dict, self.loss.L)

        if(self.isMomentumInitial==False):
            self.momentum = {t: np.zeros_like(t.gradients[t]) for t in trainables}
            self.isMomentumInitial=True

        """SGD_update"""
        for t in trainables:
            self.momentum[t] *= self.momentum_rate
            self.momentum[t] += self.learning_rate*t.gradients[t]
            t.value -= self.momentum[t]
        # t.value -= self.learning_rate * t.gradients[t]

        return graph
```

3.2.1 仿真算例

```
from tensorflow.examples.tutorials.mnist import input_data
import ANNbox as tf
import matplotlib.pyplot as plt
import numpy as np
import sys
```

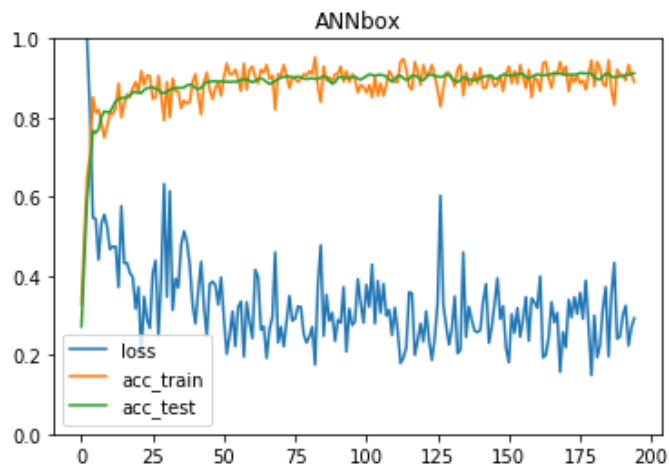
```

import time
#loaddata
mnist = input_data.read_data_sets("../MNISTDat", one_hot=True)
#sess = tf.InteractiveSession()
# Create the model
in_units = 784
o_units = 10
W1=tf.Variable(tf.truncated_normal([in_units, o_units], stddev=0.1))
b1 = tf.Variable(tf.zeros([o_units]))
x = tf.placeholder(tf.float32, [None, in_units])
mal = tf.matmul(x, W1)
y = mal + b1
y_ = tf.placeholder(tf.float32, [None, o_units])
#cross_entropy = tf.reduce_sum(tf.nn.softmax_cross_entropy_with_logits(labels=y_, logits=y))
cross_entropy = tf.nn.softmax_cross_entropy_with_logits(labels=y_, logits=y, name='cost') #loss = tf.reduce_mean(cost)
#cross_entropy = tf.nn.cross_entropy_with_logits(labels=y_, logits=tf.nn.softmax(y), name='cost')
epochs = 1
m = 50000
batch_size = 64*2*2
learning_rate=1e-3
Momentum_rate=0.9
steps_per_epoch = m // batch_size
#train_step = tf.train.GradientDescentOptimizer(learning_rate).minimize(cross_entropy)
train_step = tf.train.MomentumOptimizer(learning_rate,Momentum_rate).minimize(cross_entropy)
#train_step = tf.train.AdagradOptimizer(0.3).minimize(cross_entropy)
start_time=time.time()
#sess=tf.Session()
tf.global_variables_initializer().run()
#sess.run(tf.global_variables_initializer())
loss_list,acc_train_list,acc_test_list=[],[],[]
for _ in range(epochs):
    for i in range(steps_per_epoch):
        batch_xs=mnist.train.images[i*batch_size:(i+1)*batch_size]
        batch_ys=mnist.train.labels[i*batch_size:(i+1)*batch_size]
        ""train""
        train_step.run({x: batch_xs, y_: batch_ys})
        loss_list.append(np.mean(cross_entropy.my_eval({x: batch_xs, y_: batch_ys})))
        acc_train_list.append(np.mean((np.argmax(y.my_eval({x: batch_xs, y_: batch_ys}),1) ==
np.argmax(batch_ys,1)).astype(int)))
        acc_test_list.append(np.mean((np.argmax(y.my_eval({x: mnist.test.images, y_: mnist.test.labels}),1) ==
np.argmax(mnist.test.labels,1)).astype(int)))
        sys.stdout.write("\rprocess: {}/ {}, loss:{:.5f}, acc_train:{:.2f}, acc_test:{:.2f}".format(i, steps_per_epoch,
loss_list[-1], acc_train_list[-1], acc_test_list[-1]))
    plt.figure()
    # plt.subplot(211)
    plt.plot(range(len(loss_list)),loss_list,label=u'loss')
    # plt.subplot(212)

```



```
plt.plot(range(len(loss_list)),acc_train_list,label=u'acc_train')
plt.plot(range(len(loss_list)),acc_test_list,label=u'acc_test')
plt.ylim([0,1])
plt.title('ANNbox')
plt.legend()
plt.show()
end_time = time.time()
print('total_time:',end_time-start_time)
```



对比上一小结中的 SGD 方法，不难发现 SGDM 方法的收敛速度更快，但是收敛过程震荡小并不是很明显，这是因为每次梯度更新中用到的样本数 `batchsize` 较多，使得迭代过程原本就很稳定，从而 SGDM 并没有太明显的减震效果，如果减少样本数，那 SGDM 的减震效果还是很明显的。

3.3 NAG(Nesterov Accelerated Gradient)

$$\mathbf{W}^k \leftarrow \mathbf{W}^k - \eta \frac{\partial E_d}{\partial \mathbf{W}^k}$$

$$\mathbf{b}^k \leftarrow \mathbf{b}^k - \eta \frac{\partial E_d}{\partial \mathbf{b}^k}$$

$$\mathbf{M}_{\mathbf{W}^k} = \beta_1 \mathbf{M}_{\mathbf{W}^{k-1}} + \eta_1 \frac{\partial E_d}{\partial \mathbf{W}^k}$$

$$\mathbf{M}_{\mathbf{b}^k} = \beta_1 \mathbf{M}_{\mathbf{b}^{k-1}} + \eta_1 \frac{\partial E_d}{\partial \mathbf{b}^k}$$

$$\mathbf{W}^k \leftarrow \mathbf{W}^k - \mathbf{M}_{\mathbf{W}^k}$$

$$\mathbf{b}^k \leftarrow \mathbf{b}^k - \mathbf{M}_{\mathbf{b}^k}$$

由 SGD 的参数更新公式知： $\theta_{t+1} = \theta_t - \frac{1}{\sqrt{v_t + \varepsilon}} (\beta_1 m_{t-1} + (1 - \beta_1) g_t)$ 知，之前的动量项 m_{t-1} 并没有直接影响当

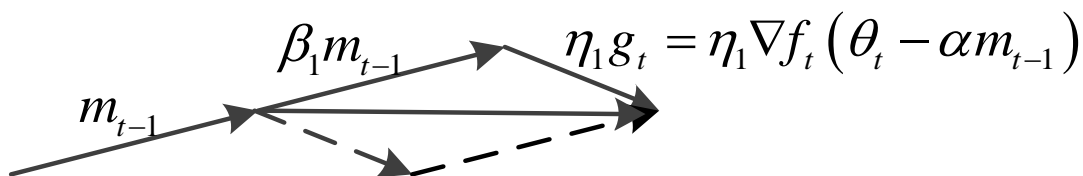
前梯度 g_t 的计算。而 NAG 是在 SGD、SGD-M 的基础上进一步改进得到的，改进点在于步骤 1：我们知道在时刻 t 的主要下降方向是由累积动量决定的，自己的梯度方向说了也不算，那与其看当前梯度方向，不如先看看如果跟着累积动量走了一步，那个时候再怎么走。所以 Nesterov 的改进就是让之前的动量直接影响当前的动量 $g_t = \nabla f_t(\theta_t - \alpha m_{t-1} / \sqrt{v_{t-1} + \varepsilon})$ 。其实 momentum 项和 nestreov 项都是为了使梯度更新更加灵活。

则 NAG 的优化流程为：

步骤 1：计算目标函数关于当前参数的梯度： $g_t = \nabla f_t(\theta_t - \alpha m_{t-1} / \sqrt{v_{t-1} + \varepsilon}) = \nabla f_t(\theta_t - \alpha m_{t-1})$

步骤 2：根据历史梯度计算一阶动量和二阶动量： $m_t = \beta_1 m_{t-1} + \eta_1 g_t; v_t = I^2; \varepsilon = 0$

步骤 3：根据下降梯度进行更新： $\theta_{t+1} = \theta_t - \frac{1}{\sqrt{v_t + \varepsilon}} m_t = \theta_t - m_t$ 。



3.4 实现 tf.train.AdagradOptimizer 类_AdaGrad

AdaGrad 利用 L2 正则化来调节梯度更新速率，通过计算迄今为止所有梯度值的平方和来度量当前维度上的学习速率。相应的计算公式如下：

$$\begin{aligned}
 M_{w^k} &= M_{w^{k-1}} + \left[\frac{\partial E_d}{\partial w^k} \right] \cdot \left[\frac{\partial E_d}{\partial w^k} \right] \\
 M_{b^k} &= M_{b^{k-1}} + \left[\frac{\partial E_d}{\partial b^k} \right] \cdot \left[\frac{\partial E_d}{\partial b^k} \right] \\
 W^k &\leftarrow W^k - \eta \frac{\partial E_d}{\partial W^k} / \sqrt{M_{w^k}} \\
 b^k &\leftarrow b^k - \eta \frac{\partial E_d}{\partial b^k} / \sqrt{M_{b^k}}
 \end{aligned} \tag{1.53}$$

其中参数的累计量全部初始化为 0.1。

代码实现如下：

```
class AdagradOptimizer(Node):
    def __init__(self, learning_rate=0.001, initial_accumulator_value=0.1, use_locking=False, name='Adagrad'):
        Node.__init__(self, inbound_nodes=[], name=name)
        self.learning_rate=learning_rate
        self.initial_accumulator_value=initial_accumulator_value
        self.is_accumulator_value_initial=False

    def minimize(self, loss):
        """暂时不知道怎么联系起来"""
        isLEnd[0]=True
        self.loss=loss
```

```
return self
```

```
def run(self, feed_dict):
    self.loss.check_graph(feed_dict)
    graph=forward_and_backward(feed_dict,self.loss.L)

    if(self.is_accumulator_value_initial==False):
        self.accumulator_value={t:self.initial_accumulator_value*np.ones_like(t.gradients[t]) for t in trainables}
        self.is_accumulator_value_initial=True

    '''Adagrad_update'''
    for t in trainables:
        partial=t.gradients[t]
        self.accumulator_value[t]+=partial**2
        t.value-=self.learning_rate*partial/(self.accumulator_value[t])**0.5
    return graph
```

3.5 实现 `tf.train.AdadeltaOptimizer` 类_Adaptive Delta

AdaDelta 是二阶优化算法，采用上面“近似 Hessian”方法来算得参数更新量，用历史参数更新量的均方根来近似 jacob 矩阵，用梯度的均方根近似 Hessian 矩阵。

```
tf.train.AdadeltaOptimizer.init(learning_rate=0.001, rho=0.95, epsilon=1e-08, use_locking=False, name='Adadelta')
```

4 自己编写神经网络库 ANNbox【高仿 tensorflow】_03 文本情感分析

通过整理文本数据训练出一个可以预测文本正面或负面情感的预测模型。

4.1.1 方法 1：基于词汇计数的方法实现输入语句数据向量化

4.1.1.1 文本输入处理

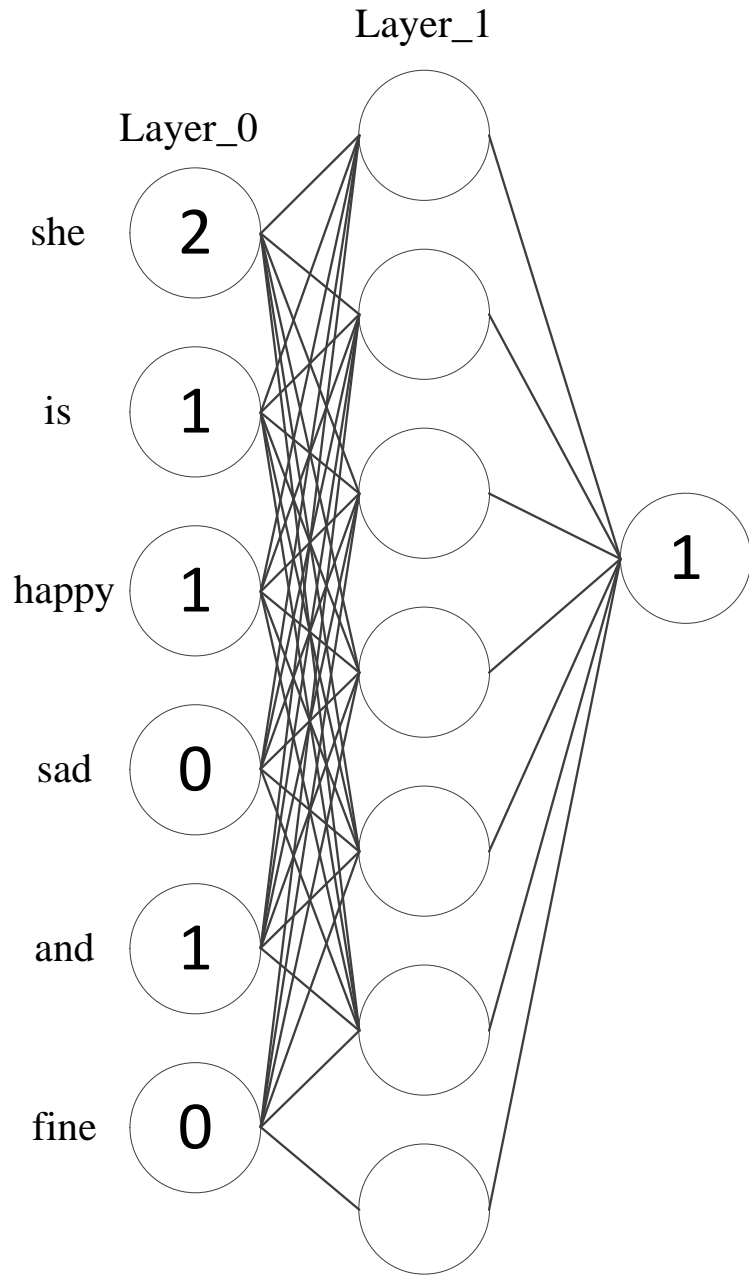
首先统计所有句子中出现的词汇组成一个集合，从而得到“单词-序列号”这一映射关系。然后对于每个句子，统计该句子中每个单词出现的次数并放入对应的“序列号”处，没有出现的单词用 0 代替。如假设文本库为“*She is happy and she is fine*”、“*She is sad*”，那么该语库中所有句子中出现的词汇组合成的集合为{‘she’, ‘is’, ‘happy’, ‘sad’, ‘and’, ‘fine’}, 句子“*She is happy and she is fine*”的输入向量为[2,1,1,0,1,0]

4.1.1.2 标签处理

Positive:+1 , Negative:-1

4.1.1.3 神经网络模型

隐藏层仅有 1 层，且激活函数为全等映射函数，输出层激活函数为 sigmoid，损失函数为平方误差损失函数，以“*She is happy and she is fine*”为输入向量的模型为：



4.1.1.4 程序实现

-*- coding: utf-8 -*-

```

import matplotlib.pyplot as plt
import numpy as np
import sys
import random
import miniFlow as mf
import pandas as pd
class data_input():
    def __init__(self, reviews, labels, hidden_nodes=10):
        """参数: reviews(dataFrame), 用于训练 | labels(dataFrame), 用于训练"""
        np.random.seed(1)
        self.pre_process_data(reviews, labels)

    def pre_process_data(self, reviews, labels):
        """预处理数据, 统计 reviews 中出现的所有单词, 并且生成 word2index"""
        # 统计 reviews 中出现的所有单词,
        review_vocab = set()
        for review in reviews.values:
            word = review[0].split(' ')
            review_vocab.update(word)

        self.review_vocab = list(review_vocab)

        # 统计 labels 中所有出现的 label(其实在这里, 就+1 和-1 两种)
        label_vocab = set()
        for label in labels.values:
            label_vocab.add(label[0])
        self.label_vocab = list(label_vocab)

        # 构建 word2idx, 给每个单词安排一个"门牌号"
        self.word2idx = dict()
        for idx, word in enumerate(self.review_vocab):
            self.word2idx[word] = idx

    def update_input_layer(self, reviews, labels):
        """对 review 进行数字化处理,统计其中单词出现次数, 并将结果存放到 self.layer_0 中, 也就是输入层"""
        inputs = np.zeros((len(reviews), len(self.review_vocab)))
        for ind in range(len(reviews)):
            for word in reviews.iloc[ind,0].split(' '):
                if word.lower() in self.word2idx:
                    idx = self.word2idx[word.lower()]
                    # 统计单词出现的次数, 作为输入
                    inputs[ind,idx] += 1
        #
            inputs[ind,idx] = 1

        labels_ = np.zeros((len(labels), 1))
        for ind in range(len(labels)):
            if(labels.iloc[ind,0]=='positive'):

```

```

        labels_[ind] = 1
    return inputs, labels_

# load data
reviews = pd.read_csv('reviews.txt', header=None)
labels = pd.read_csv('labels.txt', header=None)
data = data_input(reviews, labels)
inputs_test, labels_test = data.update_input_layer(reviews[-5000:-1], labels[-5000:-1])

## create the model
in_units = len(data.review_vocab)
h1_units = 10
o_units = 1
random.seed(1)
W1 = mf.variable(value=np.random.normal(0.0, in_units**-0.5, (in_units, h1_units)), name='W1')
b1 = mf.variable(value=np.zeros(h1_units), name='b1')
W2 = mf.variable(value=np.random.normal(0.0, h1_units**-0.5, (h1_units, o_units)), name='W2')
b2 = mf.variable(value=np.zeros(o_units), name='b2')
X, y = mf.placeholder(name='X'), mf.placeholder(name='y')
hidden1 = mf.Linear(X, W1, b1, name='linear1')
out = mf.Sigmoid(mf.Linear(hidden1, W2, b2, name='linear2'), name='out')

# Define loss and optimizer
cost = mf.MSE(y, out, name='cost')
epochs = 2
m = len(reviews) - 5000
batch_size = 250
# learning_rate = 2e-4
learning_rate = 2e-3
Momentum_rate = 0.95
steps_per_epoch = m // batch_size
# train_step = mf.train.GradientDescentOptimizer(learning_rate).minimize(cost)
train_step = mf.train.MomentumOptimizer(learning_rate, Momentum_rate).minimize(cost)
print("Total number of examples = {}".format(m))
mf.global_variables_initializer().run()
loss_list = []
acc_train_list = []
acc_test_list = []

# Train
for i in range(epochs):
    loss = 0
    for j in range(steps_per_epoch):
        inputs_train, labels_train =
data.update_input_layer(reviews[j*batch_size:(j+1)*batch_size], labels[j*batch_size:(j+1)*batch_size])
        feed_dict = {X: inputs_train, y: labels_train}
        graph = train_step.run(feed_dict)
        loss = np.mean(graph[-1].value)
        loss_list.append(loss)
        acc_train = np.mean(((graph[-2].value > 0.5).astype(int)).reshape(-1, 1) == labels_train.astype(int))
        acc_train_list.append(acc_train)

```

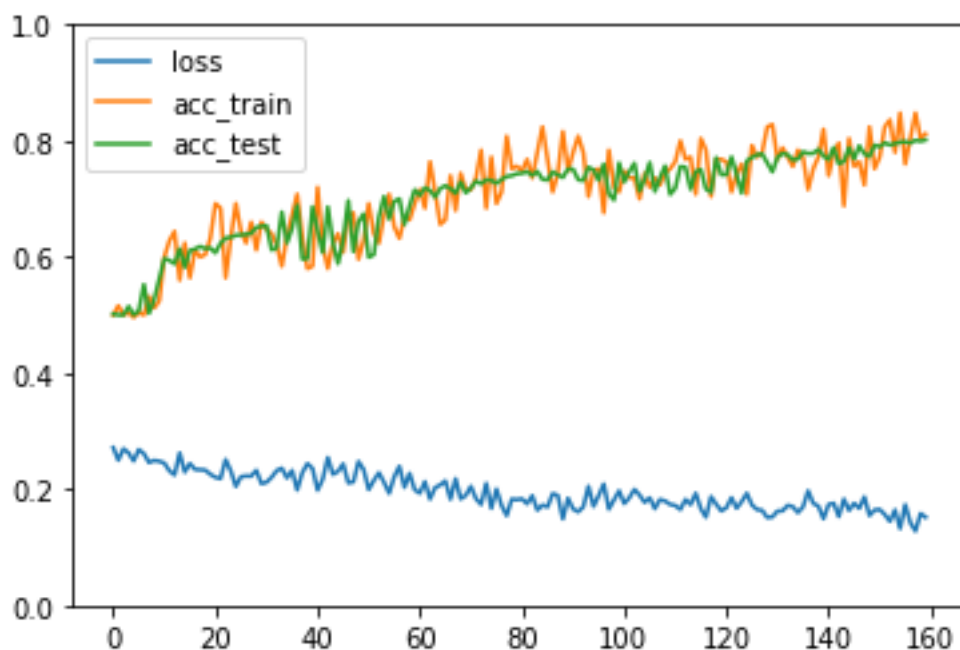
```

acc_test = np.mean(((out.run({X: inputs_test, y: labels_test})>0.5).astype(int).reshape(-1,1) ==
labels_test).astype(int))
acc_test_list.append(acc_test)
sys.stdout.write("\rprocess: {}/ {}, loss:{:.5f}, acc_train:{:.2f}, acc_test:{:.2f}".format(j, steps_per_epoch, loss,
acc_train, acc_test))
plt.figure()
# plt.subplot(211)
plt.plot(range(len(loss_list)),loss_list,label=u'loss')
# plt.subplot(212)
plt.plot(range(len(loss_list)),acc_train_list,label=u'acc_train')
plt.plot(range(len(loss_list)),acc_test_list,label=u'acc_test')
plt.ylim([0,1])
plt.legend()
plt.show()

```

4.1.1.5 分类结果

process: 79/80, loss:0.15171, acc_train:0.81, acc_test:0.80



准确率有 80%，比随机猜测要好一些。

4.1.2 方法 2：提升准确率-减少输入噪声

4.1.2.1 方法 1 的问题

方法 1 中的输入数据中包含了许多对于输出没有任何价值的单词，将其称为噪声，比如 the, an, of, on 等等。这些单词属于中性词，对于判断 Positive 或者 Negative 没有任何帮助，并且这些词出现频率非常高，在统计个数时，这些词肯定比带有情绪的词个数要多。当这些中性的词占据了大部分的输入时，即 label=1(positive)的时候，这些词出现了很多次，label=0(negative)的时候，这些词也出现了很多次，那这些词到底是正向词还是负向词呢？神经网络在训练的时候就会感到疑惑。一下这些噪声的出现频率有多高。

```

import numpy as np
import sys
import time
import pandas as pd
# 读取数据

```

```

reviews = pd.read_csv('reviews.txt', header=None)
labels = pd.read_csv('labels.txt', header=None)
from collections import Counter
positive_counter = Counter()
negative_counter = Counter()
total_counter = Counter()
for review, label in zip( reviews.values, labels.values ):
    word = review[0].split(' ')
    if label == 'positive':
        positive_counter.update(word)
    elif label == 'negative':
        negative_counter.update(word)
    total_counter.update(word)
positive_counter.most_common()[:30]
negative_counter.most_common()[:30]

```

结果:

```
positive_counter.most_common()[:30]
```

Out[5]:

```

[(',', 550468),
 ('the', 173324),
 ('.', 159654),
 ('and', 89722),
 ('a', 83688),
 ('of', 76855),
 ('to', 66746),
 ('is', 57245),
 ('in', 50215),
 ('br', 49235),
 ('it', 48025),
 ('i', 40743),
 ('that', 35630),
 ('this', 35080),
 ('s', 33815),
 ('as', 26308),
 ('with', 23247),
 ('for', 22416),
 ('was', 21917),
 ('film', 20937),
 ('but', 20822),
 ('movie', 19074),
 ('his', 17227),
 ('on', 17008),
 ('you', 16681),
 ('he', 16282),
 ('are', 14807),
 ('not', 14272),

```



```
('t', 13720),
('one', 13655)]
```

```
negative_counter.most_common()[:30]
```

```
Out[6]:
```

```
[(' ', 561462),
 ('.', 167538),
 ('the', 163389),
 ('a', 79321),
 ('and', 74385),
 ('of', 69009),
 ('to', 68974),
 ('br', 52637),
 ('is', 50083),
 ('it', 48327),
 ('i', 46880),
 ('in', 43753),
 ('this', 40920),
 ('that', 37615),
 ('s', 31546),
 ('was', 26291),
 ('movie', 24965),
 ('for', 21927),
 ('but', 21781),
 ('with', 20878),
 ('as', 20625),
 ('t', 20361),
 ('film', 19218),
 ('you', 17549),
 ('on', 17192),
 ('not', 16354),
 ('have', 15144),
 ('are', 14623),
 ('be', 14541),
 ('he', 13856)]
```

从 `positive_counter` 和 `negative_counter` 的统计结果来看，那些出现最多次的词都是噪声。真正有用词，例如 `happy`, `funny`, `horrible` 等出现次数并不多，但是却是非常重要的

4.1.2.2 语句输入预处理

由于中性词数量太多，可以尝试先降低中性词的权重，一个简单的方法是：

不在以单词个数作为输入，而是以单词是否出现作为输入（只输入 0 或者 1, 0 表示这个单词在 `review` 中没有出现，1 表示出现）。那么修改就非常简单了，将 `update_input_layer()` 中 `+=` 修改为 `=`

4.1.2.3 程序实现

```
# -*- coding: utf-8 -*-
import matplotlib.pyplot as plt
import numpy as np
import sys
```

```

import random
import miniFlow as mf
import pandas as pd
class data_input():
    def __init__(self, reviews, labels, hidden_nodes=10):
        """参数: reviews(dataFrame), 用于训练 | labels(dataFrame), 用于训练"""
        np.random.seed(1)
        self.pre_process_data(reviews, labels)

    def pre_process_data(self, reviews, labels):
        """预处理数据, 统计 reviews 中出现的所有单词, 并且生成 word2index"""
        # 统计 reviews 中出现的所有单词,
        review_vocab = set()
        for review in reviews.values:
            word = review[0].split(' ')
            review_vocab.update(word)

        self.review_vocab = list(review_vocab)

        # 统计 labels 中所有出现的 label(其实在这里, 就+1 和-1 两种)
        label_vocab = set()
        for label in labels.values:
            label_vocab.add(label[0])
        self.label_vocab = list(label_vocab)

        # 构建 word2idx, 给每个单词安排一个"门牌号"
        self.word2idx = dict()
        for idx, word in enumerate(self.review_vocab):
            self.word2idx[word] = idx

    def update_input_layer(self, reviews, labels):
        """对 review 进行数字化处理,统计其中单词出现次数, 并将结果存放到 self.layer_0 中, 也就是输入层"""
        inputs = np.zeros((len(reviews), len(self.review_vocab)))
        for ind in range(len(reviews)):
            for word in reviews.iloc[ind,0].split(' '):
                if word.lower() in self.word2idx:
                    idx = self.word2idx[word.lower()]
                    # 统计单词出现的次数, 作为输入
                    inputs[ind,idx] += 1
                    inputs[ind,idx] = 1

        labels_ = np.zeros((len(labels), 1))
        for ind in range(len(labels)):
            if(labels.iloc[ind,0]=='positive'):
                labels_[ind] = 1
        return inputs,labels_

# loaddata

```

```

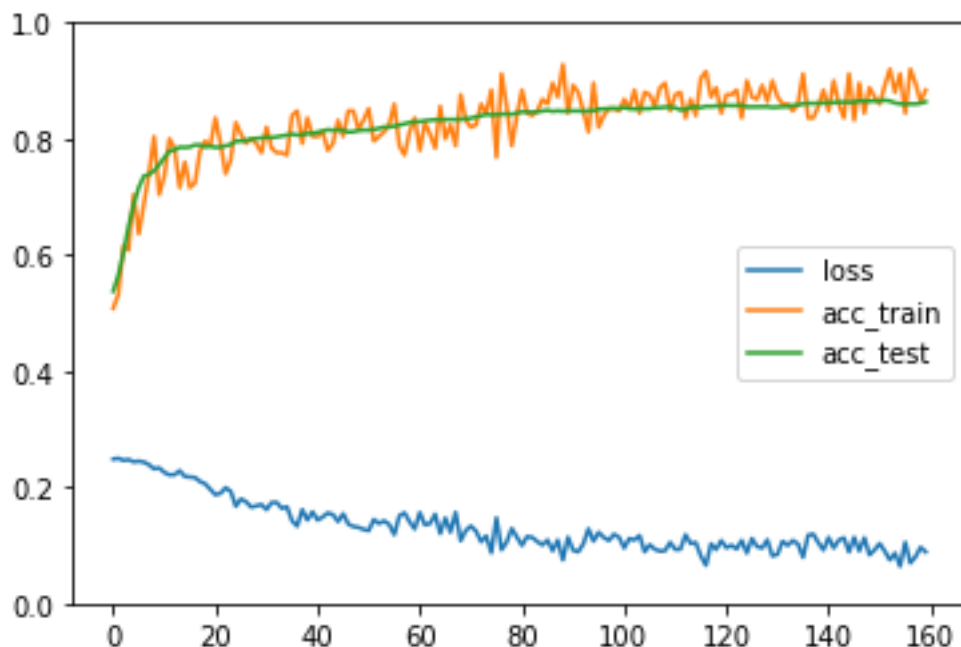
reviews = pd.read_csv('reviews.txt', header=None)
labels = pd.read_csv('labels.txt', header=None)
data = data_input(reviews, labels)
inputs_test, labels_test = data.update_input_layer(reviews[-5000:-1], labels[-5000:-1])
## create the model
in_units = len(data.review_vocab)
h1_units = 10
o_units = 1
random.seed(1)
W1=mf.variable(value=np.random.normal( 0.0, in_units**-0.5, (in_units, h1_units)), name='W1')
b1=mf.variable(value=np.zeros(h1_units), name='b1')
W2=mf.variable(value=np.random.normal( 0.0, h1_units**-0.5, (h1_units, o_units)), name='W2')
b2=mf.variable(value=np.zeros(o_units), name='b2')
X, y = mf.placeholder(name='X'), mf.placeholder(name='y')
hidden1=mf.Linear(X, W1, b1, name='linear1')
out=mf.Sigmoid(mf.Linear(hidden1, W2, b2, name='linear2'), name='out')
# Define loss and optimizer
cost = mf.MSE(y, out, name='cost')
epochs = 2
m = len(reviews) - 5000
batch_size = 250
learning_rate=2e-2
Momentum_rate=0.95
steps_per_epoch = m // batch_size
#train_step = mf.train.GradientDescentOptimizer(learning_rate).minimize(cost)
train_step = mf.train.MomentumOptimizer(learning_rate, Momentum_rate).minimize(cost)
print("Total number of examples = {}".format(m))
mf.global_variables_initializer().run()
loss_list = []
acc_train_list = []
acc_test_list = []
# Train
for i in range(epochs):
    loss = 0
    for j in range(steps_per_epoch):
        inputs_train, labels_train =
data.update_input_layer(reviews[j*batch_size:(j+1)*batch_size], labels[j*batch_size:(j+1)*batch_size])
        feed_dict = {X: inputs_train, y: labels_train}
        graph = train_step.run(feed_dict)
        loss = np.mean(graph[-1].value)
        loss_list.append(loss)
        acc_train = np.mean((((graph[-2].value>0.5).astype(int)).reshape(-1,1) == labels_train).astype(int))
        acc_train_list.append(acc_train)
        acc_test = np.mean((((out.run({X: inputs_test, y: labels_test})>0.5).astype(int)).reshape(-1,1) ==
labels_test).astype(int))
        acc_test_list.append(acc_test)
    sys.stdout.write("\rprocess: {}/{}", loss:{:.5f}, acc_train:{:.2f}, acc_test:{:.2f}".format(j, steps_per_epoch, loss,

```

```
acc_train, acc_test))
plt.figure()
# plt.subplot(211)
plt.plot(range(len(loss_list)), loss_list, label=u'loss')
# plt.subplot(212)
plt.plot(range(len(loss_list)), acc_train_list, label=u'acc_train')
plt.plot(range(len(loss_list)), acc_test_list, label=u'acc_test')
plt.ylim([0,1])
plt.legend()
plt.show()
```

4.1.2.4 分类结果

process: 159/160, loss:0.08832, acc_train:0.88, acc_test:0.86



准确率超过 80%了

4.1.3 方法 3：提速并提升准确率—减少输入向量中的中性词及出现次数较少的单词

4.1.3.1 方法思路

统计词汇在正负情感上的分布，减少对分类无用的中性词汇，从而在减少词汇表大小（从而输入向量维度减小，加速计算）的同时，减少了对神经网络判别产生迷惑的项（提升准确率）。

首先统计每个单词在正负样本中分别出现的次数 $num_pos[word]$ 及 $num_neg[word]$ ，然后针对出现次数大于

100(可以自己设定)的单词，计算每个单词的正负比例对数： $pos_neg_ratio[word] = \log\left(\frac{num_pos[word]}{num_neg[word]}\right)$ ，

不难发现，中性词对正负样本分类不明显，当样本数量较大时， $pos_neg_ratio[word]$ 应该趋于 1；相应的，如

果 $pos_neg_ratio[word]$ 离 1 越远，那么该词对正负样本的分类效果应该越好。最后将出现次数大于 100 且正负比例对数小于 -0.05 或大于 +0.05 的词汇加入到词汇表中（这里的 100 及 0.05 同样可以自己设定），计算时仅仅考虑词汇表中出现的单词。

4.1.3.2 程序实现

```
# -*- coding: utf-8 -*-
import matplotlib.pyplot as plt
import numpy as np
import sys
import random
import miniFlow as mf
import pandas as pd
from collections import Counter

class data_input():
    def __init__(self, reviews, labels, hidden_nodes=10):
        """参数: reviews(dataFrame), 用于训练 | labels(dataFrame), 用于训练"""
        np.random.seed(1)
        self.pre_process_data(reviews, labels)

    def pre_process_data(self, reviews, labels):
        """预处理数据, 统计 reviews 中出现的所有单词, 并且生成 word2index"""
        # 统计 reviews 中出现的所有单词,
        # review_vocab = set()
        # for review in reviews.values:
        #     word = review[0].split(' ')
        #     review_vocab.update(word)
        # self.review_vocab = list(review_vocab)
        # 统计 reviews 中出现的所有单词, 减去中性词的影响
        positive_counts = Counter()
        negative_counts = Counter()
        total_counts = Counter()
        for review, label in zip( reviews.values, labels.values ):
            word = review[0].split(' ')
            if label == 'positive':
                positive_counts.update(word)
            elif label == 'negative':
                negative_counts.update(word)
            total_counts.update(word)
        positive_counts.most_common()[:30]
        negative_counts.most_common()[:30]

        #正负情感中各个词汇出现的比例
        pos_neg_ratios = Counter()
        for term,cnt in list(total_counts.most_common()):
            if(cnt > 100):
                pos_neg_ratio = positive_counts[term] / float(negative_counts[term]+1)
                pos_neg_ratios[term] = pos_neg_ratio

        for word,ratio in pos_neg_ratios.most_common():
            if(ratio > 1):
                pos_neg_ratios[word] = np.log(ratio)
```

```

else:
    pos_neg_ratios[word] = -np.log((1 / (ratio + 0.01)))

min_count = 50
polarity_cutoff = 0.1
review_vocab = set()
for review in reviews.values:
    for word in review[0].split(" "):
        ## New for Project 6: only add words that occur at least min_count times
        #                                and for words with pos/neg ratios, only add words
        #                                that meet the polarity_cutoff
        if(total_counts[word] > min_count):
            if(word in pos_neg_ratios.keys()):
                if((pos_neg_ratios[word] >= polarity_cutoff) or (pos_neg_ratios[word] <= -polarity_cutoff)):
                    review_vocab.add(word)
            else:
                review_vocab.add(word)
self.review_vocab = list(review_vocab)

# 统计 labels 中所有出现的 label(其实在这里，就+1 和-1 两种)
label_vocab = set()
for label in labels.values:
    label_vocab.add(label[0])
self.label_vocab = list(label_vocab)

# 构建 word2idx，给每个单词安排一个"门牌号"
self.word2idx = dict()
for idx, word in enumerate(self.review_vocab):
    self.word2idx[word] = idx

def update_input_layer(self, reviews, labels):
    """对 review 进行数字化处理,统计其中单词出现次数，并将结果存放到 self.layer_0 中，也就是输入层"""
    inputs = np.zeros((len(reviews), len(self.review_vocab)))
    for ind in range(len(reviews)):
        for word in reviews.iloc[ind,0].split(' '):
            if word.lower() in self.word2idx:
                idx = self.word2idx[word.lower()]
                # 统计单词出现的次数，作为输入
                inputs[ind,idx] += 1
            inputs[ind,idx] = 1

    labels_ = np.zeros((len(labels), 1))
    for ind in range(len(labels)):
        if(labels.iloc[ind,0]=='positive'):
            labels_[ind] = 1
    return inputs,labels_

# loaddata

```

```

reviews = pd.read_csv('reviews.txt', header=None)
labels = pd.read_csv('labels.txt', header=None)
data = data_input(reviews, labels)
inputs_test, labels_test = data.update_input_layer(reviews[-5000:-1], labels[-5000:-1])
## create the model
in_units = len(data.review_vocab)
h1_units = 10
o_units = 1
random.seed(1)
W1=mf.variable(value=np.random.normal( 0.0, in_units**-0.5, (in_units, h1_units)), name='W1')
b1=mf.variable(value=np.zeros(h1_units), name='b1')
W2=mf.variable(value=np.random.normal( 0.0, h1_units**-0.5, (h1_units, o_units)), name='W2')
b2=mf.variable(value=np.zeros(o_units), name='b2')
X, y = mf.placeholder(name='X'), mf.placeholder(name='y')
hidden1=mf.Linear(X, W1, b1, name='linear1')
out=mf.Sigmoid(mf.Linear(hidden1, W2, b2, name='linear2'), name='out')
# Define loss and optimizer
cost = mf.MSE(y, out, name='cost')
epochs = 2
m = len(reviews) - 5000
batch_size = 250
learning_rate=2e-2
Momentum_rate=0.95
steps_per_epoch = m // batch_size
#train_step = mf.train.GradientDescentOptimizer(learning_rate).minimize(cost)
train_step = mf.train.MomentumOptimizer(learning_rate, Momentum_rate).minimize(cost)
print("Total number of examples = {}".format(m))
mf.global_variables_initializer().run()
loss_list = []
acc_train_list = []
acc_test_list = []
# Train
for i in range(epochs):
    loss = 0
    for j in range(steps_per_epoch):
        inputs_train, labels_train =
data.update_input_layer(reviews[j*batch_size:(j+1)*batch_size], labels[j*batch_size:(j+1)*batch_size])
        feed_dict = {X: inputs_train, y: labels_train}
        graph = train_step.run(feed_dict)
        loss = np.mean(graph[-1].value)
        loss_list.append(loss)
        acc_train = np.mean((((graph[-2].value>0.5).astype(int)).reshape(-1,1) == labels_train).astype(int))
        acc_train_list.append(acc_train)
        acc_test = np.mean((((out.run({X: inputs_test, y: labels_test})>0.5).astype(int)).reshape(-1,1) ==
labels_test).astype(int))
        acc_test_list.append(acc_test)
    sys.stdout.write("\rprocess: {}/{}", loss:{:.5f}, acc_train:{:.2f}, acc_test:{:.2f}".format(j, steps_per_epoch, loss,

```

```

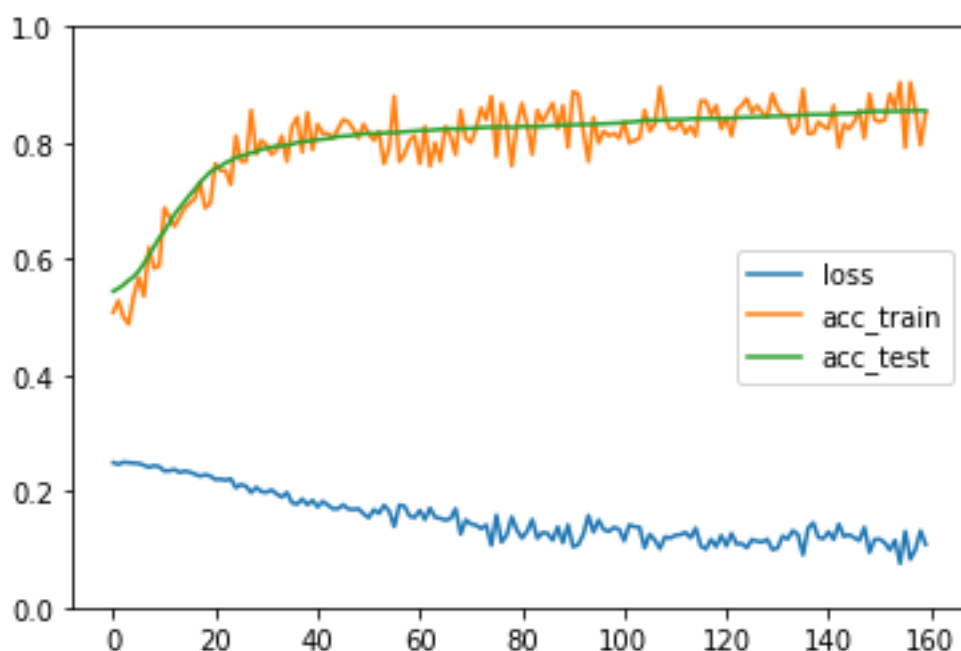
acc_train, acc_test))
plt.figure()
# plt.subplot(211)
plt.plot(range(len(loss_list)), loss_list, label=u'loss')
# plt.subplot(212)
plt.plot(range(len(loss_list)), acc_train_list, label=u'acc_train')
plt.plot(range(len(loss_list)), acc_test_list, label=u'acc_test')
plt.ylim([0,1])
plt.legend()
plt.show()

```

4.1.3.3 分类结果

好像准确率提升的并不多，但是计算速度快了许多

process: 79/80, loss:0.10845, acc_train:0.85, acc_test:0.86



4.1.4 寻找意思最相似的词

其实就是第一个隐藏层中，每个单词所对应的权重向量投影最大的那些词。

```

from collections import Counter
from collections import Counter
def get_most_similar_words(focus):
    """获取与 focus 意思相近的词"""
    most_similar = Counter()
    for word in data.word2idx.keys():
        most_similar[word] = np.dot(W1.value[data.word2idx[word]], W1.value[data.word2idx[focus]])
    return most_similar.most_common()
print(get_most_similar_words("excellent")[:10])

```

Out[39]:

```

[('great', 0.1152234330418324), ('excellent', 0.080198714194045187), ('best', 0.073144738228254028), ('love',
0.066772701417992214), ('wonderful', 0.060141522399788996), ('well', 0.059674210384868261), ('perfect',
0.056961287292946916), ('loved', 0.054479527232679389), ('amazing', 0.049861357145510987), ('still',
0.046883315279415624)]

```


5 自己编写神经网络库 ANNbox 【高仿 tensorflow】_04 实现卷积神经网络，并用简化版本的 AlexNet(无 pool,无 dropout)

5.1 conv2d 类

首先是线性加权输入类，主要实现下式的运算操作【[公式中参数详细的说明请参考深度学习基础模型算法原理及编程实现--03.卷积神经网络[]，全文同】

5.1.1 对于 padding='same'的模式，有

$$net_{d_{k+1}}^k(m, n) = \sum_{d_k=1}^{D_k} \sum_{i=1}^{KF_k} \sum_{j=1}^{KH_k} zerospadding(A_{d_k}^k, KF_k - 1, KH_k - 1, 0)(i + m - 1, j + n - 1)W_{d_k, d_{k+1}}^k(i, j) \quad (1.54)$$

$$, 1 \leq m \leq F_k, 1 \leq n \leq H_k$$

为了方便编程实现，将上式整理成矩阵运算的形式，有：

$$\begin{aligned} & [net^k]_{1 \times F_{k+1} \times H_{k+1} \times D_{k+1}} \\ &= \left[\begin{aligned} & \text{zerospadding} \left([A^k]_{1 \times F_k \times H_k \times D_k}, KF_k - 1, KH_k - 1, 0 \right)_{1 \times (F_k + KF_k - 1) \times (H_k + KH_k - 1) \times D_k} \\ & * [W^k]_{KF_k \times KH_k \times D_k \times D_{k+1}} \end{aligned} \right]_{1 \times F_{k+1} \times H_{k+1} \times D_{k+1}} \quad (1.55) \end{aligned}$$

这里之所以成立是因为对于 padding='same' 的情况，有： $F_{k+1} = F_k, H_{k+1} = H_k$

对于批数为 batchsize 的小批量处理数据，有：

$$\begin{aligned} & [net^k]_{batchsize \times F_{k+1} \times H_{k+1} \times D_{k+1}} \\ &= \left[\begin{aligned} & \text{zerospadding} \left(A^k, KF_k - 1, KH_k - 1, 0 \right)_{batchsize \times (F_k + KF_k - 1) \times (H_k + KH_k - 1) \times D_k} \\ & * [W^k]_{KF_k \times KH_k \times D_k \times D_{k+1}} \end{aligned} \right]_{batchsize \times F_{k+1} \times H_{k+1} \times D_{k+1}} \quad (1.56) \end{aligned}$$

下面再通过简单的图示对卷积操作进行说明：


待补充

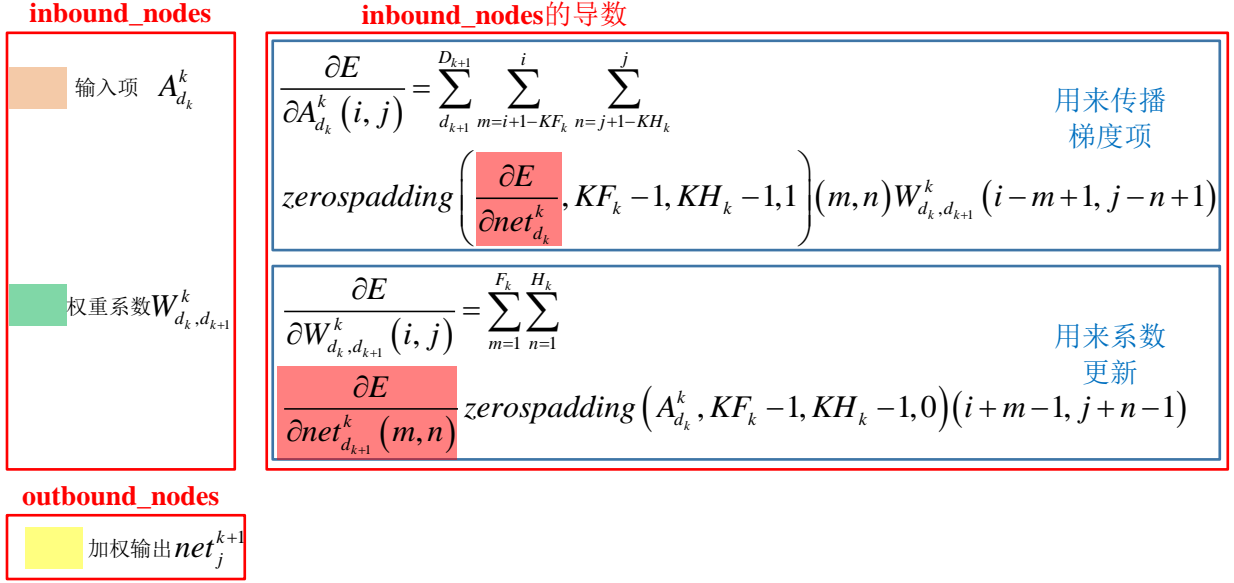
由前文知，matmul类中同样有inbound_nodes和outbound_nodes，详细如下图所示，为了方便对比，其中inbound_nodes和outbound_nodes中不同项所对应的方框颜色与上图中的方框颜色一致。且正如下图所示，outbound_nodes的导数不要求，因为在当前类中没有可以用来求其导数的信息，并且当前节点的outbound_nodes是下一个节点的inbound_nodes，所以outbound_nodes的导数在下一个节点中被计算。

此外，inbound_nodes的导数被表示为当前类中的输入项与损失函数对输出项偏导数【如下图所示， $\frac{\partial E_d}{\partial net_j^k}$ 】

的乘积，其中当前类中的输入项是已知的，而 $\frac{\partial E_d}{\partial net_j^k}$ 并不知，但 $\frac{\partial E_d}{\partial net_j^k}$ 具体的值可通过访问下一个神经元对输入项的偏导数得到。

Conv2d类中的输入项有2个，对不同的输入项的偏导数有不同的作用，如下图所示， $\frac{\partial E_d}{\partial A_{d_k}^k(i, j)}$ 用来传播梯度

项【用  表示，这里先给出结论，具体原因后面再说】， $\frac{\partial E_d}{\partial W_{d_k, d_{k+1}}^k(i, j)}$ 用来更新系数。



为了方便编程实现，将上图中 2 个偏导数写成矩阵运算的形式，有：

$$\begin{aligned} \left[\frac{\partial E}{\partial A^k} \right]_{1 \times F_k \times H_k \times D_k} &= \left[\text{zerospadding}\left(\left[\delta^k\right]_{1 \times F_{k+1} \times H_{k+1} \times D_{k+1}}, KF_k - 1, KH_k - 1, 1\right) \right]_{1 \times (F_{k+1} + KF_k - 1) \times (H_{k+1} + KH_k - 1) \times D_{k+1}} \\ &\quad * \text{rot180}\left(\left[W^k\right]_{KF_k \times KH_k \times D_k \times D_{k+1}}\right). \text{reshape}(KF_k \times KH_k \times D_{k+1} \times D_k) \Big]_{1 \times F_k \times H_k \times D_k} \\ \left[\frac{\partial E}{\partial W^k} \right]_{D_k \times KF_k \times KH_k \times D_{k+1}} &= \left[\left(\left[\text{zerospadding}\left(A_{d_k}^k, KF_k - 1, KH_k - 1, 0\right) \right]_{1 \times (F_k + KF_k - 1) \times (H_k + KH_k - 1) \times D_k} \right). \text{reshape}(D_k, F_k + KF_k - 1, H_k + KH_k - 1, 1) \right. \\ &\quad \left. * \left(\left[\delta^k\right]_{1 \times F_{k+1} \times H_{k+1} \times D_{k+1}} \right). \text{reshape}(F_{k+1}, H_{k+1}, 1, D_{k+1}) \right]_{D_k \times KF_k \times KH_k \times D_{k+1}} \end{aligned}$$

这里之所以成立是因为对于 padding='same' 的情况，有： $F_{k+1} = F_k, H_{k+1} = H_k$

(1.57)

对于批数为 batchsize 的小批量处理数据，有：

$$\begin{aligned} \left[\frac{\partial E}{\partial A^k} \right]_{\text{batchsize} \times F_k \times H_k \times D_k} &= \left[\text{zerospadding}\left(\left[\delta^k\right]_{\text{batchsize} \times F_{k+1} \times H_{k+1} \times D_{k+1}}, KF_k - 1, KH_k - 1, 1\right) \right]_{\text{batchsize} \times (F_{k+1} + KF_k - 1) \times (H_{k+1} + KH_k - 1) \times D_{k+1}} \\ &\quad * \text{rot180}\left(\left[W^k\right]_{KF_k \times KH_k \times D_k \times D_{k+1}}\right). \text{reshape}(KF_k \times KH_k \times D_{k+1} \times D_k) \Big]_{\text{batchsize} \times F_k \times H_k \times D_k} \\ \left[\frac{\partial E}{\partial W^k} \right]_{KF_k \times KH_k \times D_k \times D_{k+1}} &= \left[\left(\left[\text{zerospadding}\left(A^k, KF_k - 1, KH_k - 1, 0\right) \right]_{\text{batchsize} \times (F_k + KF_k - 1) \times (H_k + KH_k - 1) \times D_k} \right). \text{reshape}(D_k, F_k + KF_k - 1, H_k + KH_k - 1, \text{batchsize}) \right. \\ &\quad \left. * \left(\left[\delta^k\right]_{\text{batchsize} \times F_{k+1} \times H_{k+1} \times D_{k+1}} \right). \text{reshape}(F_{k+1}, H_{k+1}, \text{batchsize}, D_{k+1}) \right]_{D_k \times KF_k \times KH_k \times D_{k+1}} \end{aligned}$$

(1.58)

5.1.1.1 Tensorflow 中，padding='SAME'的情况下，不同 Stride 步长下的取值方式

我们将“卷积输出_under_原始尺寸”的维度定义为 stride=[1,1,1,1]模式下的输出维度，在 padding='SAME'的条件下，下图中的红色数字都是在不同 stride 下，“卷积输出”映射到“卷积输出_under_原始尺寸”中的位置。

strides: [1, 1, 1, 1]
padding: SAME

$$\begin{bmatrix} 4. & 3. & 1. & 0. & 9. \\ 2. & 1. & 0. & 1. & 2. \\ 1. & 2. & 4. & 1. & 6. \\ 3. & 1. & 0. & 2. & 5. \\ -3. & 0. & 8. & 7. & 0. \end{bmatrix} * \begin{bmatrix} 1. & 0. & 1. \\ 2. & 1. & 0. \\ 0. & 0. & 1. \end{bmatrix} =$$

卷积输出:

$$\begin{bmatrix} 5. & 11. & 8. & 4. & 9. \\ 7. & 14. & 6. & 17. & 4. \\ 3. & 6. & 12. & 16. & 9. \\ 5. & 20. & 12. & 12. & 10. \\ -2. & -3. & 11. & 28. & 16. \end{bmatrix}$$

卷积输出_under_原始尺寸:

$$\begin{bmatrix} 5. & 11. & 8. & 4. & 9. \\ 7. & 14. & 6. & 17. & 4. \\ 3. & 6. & 12. & 16. & 9. \\ 5. & 20. & 12. & 12. & 10. \\ -2. & -3. & 11. & 28. & 16. \end{bmatrix}$$

strides: [1, 2, 2, 1]
padding: SAME

$$\begin{bmatrix} 4. & 3. & 1. & 0. & 9. \\ 2. & 1. & 0. & 1. & 2. \\ 1. & 2. & 4. & 1. & 6. \\ 3. & 1. & 0. & 2. & 5. \\ -3. & 0. & 8. & 7. & 0. \end{bmatrix} * \begin{bmatrix} 1. & 0. & 1. \\ 2. & 1. & 0. \\ 0. & 0. & 1. \end{bmatrix} =$$

卷积输出:

$$\begin{bmatrix} 5. & 8. & 9. \\ 3. & 12. & 9. \\ -2. & 11. & 16. \end{bmatrix}$$

卷积输出_under_原始尺寸:

$$\begin{bmatrix} 5. & 0. & 8. & 0. & 9. \\ 0. & 0. & 0. & 0. & 0. \\ 3. & 0. & 12. & 0. & 9. \\ 0. & 0. & 0. & 0. & 0. \\ -2. & 0. & 11. & 0. & 16. \end{bmatrix}$$

strides: [1, 2, 2, 1]
padding: SAME

$$\begin{bmatrix} 4. & 3. & 1. & 0. & 9. \\ 2. & 1. & 0. & 1. & 2. \\ 1. & 2. & 4. & 1. & 6. \\ 3. & 1. & 0. & 2. & 5. \\ -3. & 0. & 8. & 7. & 0. \end{bmatrix} * \begin{bmatrix} 1. & 0. & 1. \\ 2. & 1. & 0. \\ 0. & 0. & 1. \end{bmatrix} =$$

卷积输出:

$$\begin{bmatrix} 5. & 8. & 9. \\ 3. & 12. & 9. \\ -2. & 11. & 16. \end{bmatrix}$$

卷积输出_under_原始尺寸:

$$\begin{bmatrix} 5. & 0. & 8. & 0. & 9. \\ 0. & 0. & 0. & 0. & 0. \\ 3. & 0. & 12. & 0. & 9. \\ 0. & 0. & 0. & 0. & 0. \\ -2. & 0. & 11. & 0. & 16. \end{bmatrix}$$

卷积输出:

卷积输出_under_原始尺寸:

strides: [1, 3, 3, 1]
padding: SAME

$$\begin{bmatrix} 4. & 3. & 1. & 0. & 9. \\ 2. & 1. & 0. & 1. & 2. \\ 1. & 2. & 4. & 1. & 6. \\ 3. & 1. & 0. & 2. & 5. \\ -3. & 0. & 8. & 7. & 0. \end{bmatrix} * \begin{bmatrix} 1. & 0. & 1. \\ 2. & 1. & 0. \\ 0. & 0. & 1. \end{bmatrix} =$$

卷积输出:

$$\begin{bmatrix} 14. & 4. \\ -3. & 16. \end{bmatrix}$$

卷积输出_under_原始尺寸:

$$\begin{bmatrix} 0. & 0. & 0. & 0. & 0. \\ 0. & 14. & 0. & 0. & 4. \\ 0. & 0. & 0. & 0. & 0. \\ 0. & 0. & 0. & 0. & 0. \\ 0. & -3. & 0. & 0. & 16. \end{bmatrix}$$

strides: [1, 4, 4, 1]
padding: SAME

$$\begin{bmatrix} 4. & 3. & 1. & 0. & 9. \\ 2. & 1. & 0. & 1. & 2. \\ 1. & 2. & 4. & 1. & 6. \\ 3. & 1. & 0. & 2. & 5. \\ -3. & 0. & 8. & 7. & 0. \end{bmatrix} * \begin{bmatrix} 1. & 0. & 1. \\ 2. & 1. & 0. \\ 0. & 0. & 1. \end{bmatrix} =$$

卷积输出:

$$\begin{bmatrix} 5. & 9. \\ -2. & 16. \end{bmatrix}$$

卷积输出_under_原始尺寸:

$$\begin{bmatrix} 5. & 0. & 0. & 0. & 9. \\ 0. & 0. & 0. & 0. & 0. \\ 0. & 0. & 0. & 0. & 0. \\ 0. & 0. & 0. & 0. & 0. \\ -2. & 0. & 0. & 0. & 16. \end{bmatrix}$$

卷积输出:

卷积输出_under_原始尺寸:

strides: [1, 5, 5, 1]
padding: SAME

$$\begin{bmatrix} 4. & 3. & 1. & 0. & 9. \\ 2. & 1. & 0. & 1. & 2. \\ 1. & 2. & 4. & 1. & 6. \\ 3. & 1. & 0. & 2. & 5. \\ -3. & 0. & 8. & 7. & 0. \end{bmatrix} * \begin{bmatrix} 1. & 0. & 1. \\ 2. & 1. & 0. \\ 0. & 0. & 1. \end{bmatrix} =$$

$$[[12.]]$$

$$\begin{bmatrix} 0. & 0. & 0. & 0. & 0. \\ 0. & 0. & 0. & 0. & 0. \\ 0. & 0. & 12. & 0. & 0. \\ 0. & 0. & 0. & 0. & 0. \\ 0. & 0. & 0. & 0. & 0. \end{bmatrix}$$

5.1.2 对于 padding='valid'的模式，有

$$net_{d_{k+1}}^k(m, n) = \sum_{d_k=1}^{D_k} \sum_{i=1}^{KF_k} \sum_{j=1}^{KH_k} A_{d_k}^k(i+m-1, j+n-1) W_{d_k, d_{k+1}}^k(i, j) \quad (1.59)$$

$$1 \leq m \leq F_k - KF_k + 1, 1 \leq n \leq H_k - KH_k + 1$$

为了方便编程实现，将上式整理成矩阵运算的形式，有：

$$\left[net^k \right]_{(F_k - KF_k + 1) \times (H_k - KH_k + 1) \times D_{k+1}} = \sum_{d_k=1}^{D_k} \left[A^k \right]_{F_k \times H_k \times D_k} * \left[W^k \right]_{KF_k \times KH_k \times D_k \times D_{k+1}} \quad (1.60)$$

对于批数为 batchsize 的小批量处理数据，有：

$$\left[net^k \right]_{batchsize \times (F_k - KF_k + 1) \times (H_k - KH_k + 1) \times D_{k+1}} = \sum_{d_k=1}^{D_k} \left[A^k \right]_{batchsize \times F_k \times H_k \times D_k} * \left[W^k \right]_{KF_k \times KH_k \times D_k \times D_{k+1}} \quad (1.61)$$

下面再通过简单的图示对卷积操作进行说明：

待补充

由前文知，matmul类中同样有inbound_nodes和outbound_nodes，详细如下图所示，为了方便对比，其中inbound_nodes和outbound_nodes中不同项所对应的方框颜色与上图中的方框颜色一致。且正如下图所示所述，outbound_nodes的导数不要求，因为在当前类中没有可以用来求其导数的信息，并且当前节点的outbound_nodes是下一个节点的inbound_nodes，所以outbound_nodes的导数在下一个节点中被计算。

此外，inbound_nodes的导数被表示为当前类中的输入项与损失函数对输出项偏导数【如下图所示， $\frac{\partial E_d}{\partial net_j^k}$ 】

的乘积，其中当前类中的输入项是已知的，而 $\frac{\partial E_d}{\partial net_j^k}$ 并不知，但 $\frac{\partial E_d}{\partial net_j^k}$ 具体的值可通过访问下一个神经元对输入项的偏导数得到。

Conv2d类中的输入项有2个，对不同的输入项的偏导数有不同的作用，如下图所示， $\frac{\partial E_d}{\partial A_{d_k}^k(i, j)}$ 用来传播梯度

项， $\frac{\partial E_d}{\partial W_{d_k, d_{k+1}}^k(i, j)}$ 用来更新系数。

inbound_nodes



inbound_nodes的导数

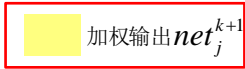
$$\frac{\partial E}{\partial A_{d_k}^k(i, j)} = \sum_{d_{k+1}}^{D_{k+1}} \sum_{m=i+1-KF_k}^i \sum_{n=j+1-KH_k}^j \text{zerospadding} \left(\frac{\partial E}{\partial \text{net}_{d_{k+1}}^k}, 2(KF_k-1), 2(KH_k-1), 3 \right) (m, n) W_{d_k, d_{k+1}}^k (i-m+1, j-n+1)$$

用来传播
梯度项

$$\frac{\partial E}{\partial W_{d_k, d_{k+1}}^k(i, j)} = \sum_{m=1}^{F_k-KF_k+1} \sum_{n=1}^{H_k-KH_k+1} \frac{\partial E}{\partial \text{net}_{d_{k+1}}^k(m, n)} A_{d_k}^k(i+m-1, j+n-1)$$

用来系数
更新

outbound_nodes



为了方便编程实现，将上图中 2 个偏导数写成矩阵运算的形式，有：

$$\left[\frac{\partial E}{\partial A^k} \right]_{1 \times F_k \times H_k \times D_k} = \left[\text{zerospadding} \left(\left[\delta^k \right]_{1 \times (F_k-KF_k+1) \times (H_k-KH_k+1) \times D_{k+1}}, 2(KF_k-1), 2(KH_k-1), 3 \right) \right]_{1 \times (F_k+KF_k-1) \times (H_k+KH_k-1) \times D_{k+1}} \left[\left[W^k \right]_{KF_k \times KH_k \times D_k \times D_{k+1}} \right]_{D_k \times KF_k \times KH_k \times D_{k+1}} \text{.reshape}(KF_k, KH_k, D_{k+1}, D_k)$$

$$\left[\frac{\partial E}{\partial W^k} \right]_{KF_k \times KH_k \times D_k \times D_{k+1}} = \left[\left(\left[A_{d_k}^k \right]_{1 \times F_k \times H_k \times D_k} \right) \text{.reshape}(D_k, F_k, H_k, 1) \right]_{D_k \times F_k \times H_k \times D_k} \left[\left(\left[\delta_{d_{k+1}}^k \right]_{1 \times (F_k-(KF_k-1)) \times (H_k-(KH_k-1)) \times D_{k+1}} \right) \text{.reshape}(F_k-(KF_k-1), H_k-(KH_k-1), 1, D_{k+1}) \right]_{D_k \times (F_k-KF_k+1) \times (H_k-KH_k+1) \times D_{k+1}} \text{.reshape}(KF_k, KH_k, D_k, D_{k+1})$$

(1.62)

对于批数为 batchsize 的小批量处理数据，有：

$$\left[\frac{\partial E}{\partial A^k} \right]_{\text{batchsize} \times F_k \times H_k \times D_k} = \left[\text{zerospadding} \left(\left[\delta^k \right]_{\text{batchsize} \times (F_k-KF_k+1) \times (H_k-KH_k+1) \times D_{k+1}}, 2(KF_k-1), 2(KH_k-1), 3 \right) \right]_{\text{batchsize} \times (F_k+KF_k-1) \times (H_k+KH_k-1) \times D_{k+1}} \left[\left[W^k \right]_{KF_k \times KH_k \times D_k \times D_{k+1}} \right]_{D_k \times KF_k \times KH_k \times D_{k+1}} \text{.reshape}(KF_k, KH_k, D_{k+1}, D_k)$$

$$\left[\frac{\partial E}{\partial W^k} \right]_{KF_k \times KH_k \times D_k \times D_{k+1}} = \left[\left(\left[A_{d_k}^k \right]_{\text{batchsize} \times F_k \times H_k \times D_k} \right) \text{.reshape}(D_k, F_k, H_k, \text{batchsize}) \right]_{D_k \times F_k \times H_k \times \text{batchsize}} \left[\left(\left[\delta_{d_{k+1}}^k \right]_{\text{batchsize} \times (F_k-(KF_k-1)) \times (H_k-(KH_k-1)) \times D_{k+1}} \right) \text{.reshape}(F_k-(KF_k-1), H_k-(KH_k-1), \text{batchsize}, D_{k+1}) \right]_{D_k \times (F_k-KF_k+1) \times (H_k-KH_k+1) \times D_{k+1}} \text{.reshape}(KF_k, KH_k, D_k, D_{k+1})$$

(1.63)

5.1.2.1 Tensorflow 中，padding='VALID'的情况下，不同 Stride 步长下的取值方式

我们将“卷积输出_under_原始尺寸”的维度定义为 stride=[1,1,1,1]模式下的输出维度，在 padding='SAME'的条件下，下图中的红色数字都是在不同 stride 下，“卷积输出”映射到“卷积输出_under_原始尺寸”中的位置。

<div style="background-color: #4a7ebb; color: white; padding: 5px; margin-bottom: 10px;"> strides: [1, 1, 1, 1] padding: VALID </div> <div style="display: flex; align-items: center;"> <div style="margin-right: 10px;"> $[[4. \ 3. \ 1. \ 0. \ 9.]$ $[2. \ 1. \ 0. \ 1. \ 2.]$ $[1. \ 2. \ 4. \ 1. \ 6.]$ $[3. \ 1. \ 0. \ 2. \ 5.]$ $[-3. \ 0. \ 8. \ 7. \ 0.]$ </div> <div style="margin-right: 10px;"> $[[1. \ 0. \ 1.]$ $[2. \ 1. \ 0.]$ $[0. \ 0. \ 1.]$ </div> <div> $=$ </div> </div>	<div style="background-color: #4a7ebb; color: white; padding: 5px; margin-bottom: 10px;">卷积输出:</div> <div> $[[14. \ 6. \ 17.]$ $[6. \ 12. \ 16.]$ $[20. \ 12. \ 12.]]$ </div>	<div style="background-color: #4a7ebb; color: white; padding: 5px; margin-bottom: 10px;">卷积输出_under_原始尺寸:</div> <div> $[[14. \ 6. \ 17.]$ $[6. \ 12. \ 16.]$ $[20. \ 12. \ 12.]]$ </div>
---	--	---

<div>strides: [1, 2, 2, 1] padding: VALID</div>	$\begin{bmatrix} 4. & 3. & 1. & 0. & 9. \\ 2. & 1. & 0. & 1. & 2. \\ 1. & 2. & 4. & 1. & 6. \\ 3. & 1. & 0. & 2. & 5. \\ -3. & 0. & 8. & 7. & 0. \end{bmatrix} * \begin{bmatrix} 1. & 0. & 1. \\ 2. & 1. & 0. \\ 0. & 0. & 1. \end{bmatrix} =$	<div>卷积输出:</div> $\begin{bmatrix} 14. & 17. \\ 20. & 12. \end{bmatrix}$	<div>卷积输出_under_原始尺寸:</div> $\begin{bmatrix} 14. & 0. & 17. \\ 0. & 0. & 0. \\ 20. & 0. & 12. \end{bmatrix}$
<div>strides: [1, 3, 3, 1] padding: VALID</div>	$\begin{bmatrix} 4. & 3. & 1. & 0. & 9. \\ 2. & 1. & 0. & 1. & 2. \\ 1. & 2. & 4. & 1. & 6. \\ 3. & 1. & 0. & 2. & 5. \\ -3. & 0. & 8. & 7. & 0. \end{bmatrix} * \begin{bmatrix} 1. & 0. & 1. \\ 2. & 1. & 0. \\ 0. & 0. & 1. \end{bmatrix} =$	<div>卷积输出:</div> $\begin{bmatrix} 14. \end{bmatrix}$	<div>卷积输出_under_原始尺寸:</div> $\begin{bmatrix} 14. & 0. & 0. \\ 0. & 0. & 0. \\ 0. & 0. & 0. \end{bmatrix}$
<div>strides: [1, 4, 4, 1] padding: VALID</div>	$\begin{bmatrix} 4. & 3. & 1. & 0. & 9. \\ 2. & 1. & 0. & 1. & 2. \\ 1. & 2. & 4. & 1. & 6. \\ 3. & 1. & 0. & 2. & 5. \\ -3. & 0. & 8. & 7. & 0. \end{bmatrix} * \begin{bmatrix} 1. & 0. & 1. \\ 2. & 1. & 0. \\ 0. & 0. & 1. \end{bmatrix} =$	<div>卷积输出:</div> $\begin{bmatrix} 14. \end{bmatrix}$	<div>卷积输出_under_原始尺寸:</div> $\begin{bmatrix} 14. & 0. & 0. \\ 0. & 0. & 0. \\ 0. & 0. & 0. \end{bmatrix}$
<div>strides: [1, 5, 5, 1] padding: VALID</div>	$\begin{bmatrix} 4. & 3. & 1. & 0. & 9. \\ 2. & 1. & 0. & 1. & 2. \\ 1. & 2. & 4. & 1. & 6. \\ 3. & 1. & 0. & 2. & 5. \\ -3. & 0. & 8. & 7. & 0. \end{bmatrix} * \begin{bmatrix} 1. & 0. & 1. \\ 2. & 1. & 0. \\ 0. & 0. & 1. \end{bmatrix} =$	<div>卷积输出:</div> $\begin{bmatrix} 14. \end{bmatrix}$	<div>卷积输出_under_原始尺寸:</div> $\begin{bmatrix} 14. & 0. & 0. \\ 0. & 0. & 0. \\ 0. & 0. & 0. \end{bmatrix}$

5.1.3 对应的编程实现如下:

5.1.3.1 主函数

class conv2d(Node):

"""二维卷积类，实现对矩阵的卷积操作"""

def __init__(self,X,conv_weights,strides=[1,1,1,1],padding='SAME',name=[]):

Node.__init__(self, inbound_nodes=[X,conv_weights], name=name)

self.strides=strides

self.padding=padding

def forward(self):

self.kF_KH_for_zerospadding=self.inbound_nodes[1].value.shape[0:2]

self.imagesize=self.inbound_nodes[0].value.shape

image=self.inbound_nodes[0].value

kernal=self.inbound_nodes[1].value

if(self.padding=='SAME'):

image_after_zerospadding = bs.zerospadding(image,kernal_size=[kernal.shape[0],kernal.shape[1]],mode=0)

if(self.padding=='VALID'):

image_after_zerospadding = bs.zerospadding(image,kernal_size=[kernal.shape[0],kernal.shape[1]],mode=2)

conv_origin_size_label_one_zero=np.zeros((image.shape[0],image_after_zerospadding.shape[1]-kernal.shape[0]+1,image_after_zerospadding.shape[2]-kernal.shape[1]+1,kernal.shape[-1]))

conv_out_in_orignal_size=np.zeros_like(conv_origin_size_label_one_zero)

```

conv_out=np.zeros((image.shape[0],int((image_after_zeropadding.shape[1]-kernel.shape[0])/self.strides[1])+1,int((image
_after_zeropadding.shape[2]-kernel.shape[1])/self.strides[2])+1,kernel.shape[-1]))
    ind_i=0
    kernel_hsize=conv_origin_size_label_one_zero.shape[1]
    kernel_wsize=conv_origin_size_label_one_zero.shape[2]
    for i in range(0,kernel_hsize,self.strides[1]):
        i=kernel_hsize-1-i
        ind_i=int(i/self.strides[1])
        for j in range(0,kernel_wsize,self.strides[2]):
            j=kernel_wsize-1-j
            ind_j=int(j/self.strides[2])
            for patch in range(image.shape[0]):
                for channel in range(kernel.shape[-1]):
                    conv_out[patch,ind_i,ind_j,channel]=np.sum(image_after_zeropadding[patch,i:i+kernel.shape[0],j:j+kernel.shape[1],:]*ke
rnal[:, :, :, channel])

                    conv_origin_size_label_one_zero[patch,i,j,channel]=1
                    conv_out_in_orignal_size[patch,i,j,channel]=conv_out[patch,ind_i,ind_j,channel]
self.value=conv_out
self.conv_origin_size_label_one_zero=conv_origin_size_label_one_zero
self.conv_out_in_orignal_size=conv_out_in_orignal_size
self.image_after_zeropadding=image_after_zeropadding

def backward(self):
    rw.SAFE_FWRITE(IsWriteDebug, fout_Debug_miniflow,"后向计算:\n")
    self.gradients = {n: np.zeros_like(n.value) for n in self.inbound_nodes}
    kernal=self.inbound_nodes[1].value
    weight_gradient=np.zeros_like(kernal).transpose(2,0,1,3)
    image_after_zeropadding_transpose=self.image_after_zeropadding.transpose(3,1,2,0)
    conv_origin_size_label_one_zero_transposes=self.conv_origin_size_label_one_zero.transpose(1,2,0,3)
    for n in self.outbound_nodes:
        grad_cost = n.gradients[self] if(self in n.gradients.keys()) else np.ones_like(self.value)
        grad_cost_transpose_to_orignal_size=np.zeros_like(self.conv_origin_size_label_one_zero)
        kind=0
        for ki in range(grad_cost_transpose_to_orignal_size.shape[1]):
            kjnd=0
            for kj in range(grad_cost_transpose_to_orignal_size.shape[2]):
                if(self.conv_origin_size_label_one_zero[0,ki,kj,0]!=0):
                    grad_cost_transpose_to_orignal_size[:,ki,kj,:]=grad_cost[:,kind,kjnd,:]
                    kjnd+=1
                if(kjnd%grad_cost.shape[1]==0):
                    kjnd=0
                    kind+=1
            grad_cost_transpose=grad_cost_transpose_to_orignal_size.transpose(1,2,0,3)

        if(self.padding=='SAME'):

```

```

delta_padding=bs.zerospadding(self.conv_origin_size_label_one_zero*grad_cost_transpose_to_original_size,kernal_size=[
kernal.shape[0],kernal.shape[1]],mode=1)
    if(self.padding=='VALID'):

delta_padding=bs.zerospadding(self.conv_origin_size_label_one_zero*grad_cost_transpose_to_original_size,kernal_size=[
kernal.shape[0],kernal.shape[1]],mode=3)

    image_gradient=np.zeros_like(self.inbound_nodes[0].value)
    kernal180=bs.mat_rotat90(kernal,2)
    kernal180_transpose=kernal180.transpose(0,1,3,2)
    for i in range(image_gradient.shape[1]):
        for j in range(image_gradient.shape[2]):
            for batchsize in range(delta_padding.shape[0]):
                for dk in range(kernal180_transpose.shape[-1]):

image_gradient[batchsize,i,j,dk]=np.sum(delta_padding[batchsize,i:i+kernal180_transpose.shape[0],j:j+kernal180_transpo
se.shape[1],:]*kernal180_transpose[:, :, :, dk])

    self.gradients[self.inbound_nodes[0]] += image_gradient

    for i in range(weight_gradient.shape[1]):
        for j in range(weight_gradient.shape[2]):
            for dk in range(image_after_zerospadding_transpose.shape[0]):
                for dk_1 in range(conv_origin_size_label_one_zero_transpoes.shape[-1]):
                    weight_gradient[dk,i,j,dk_1]=np.sum(\

image_after_zerospadding_transpose[dk,i:i+conv_origin_size_label_one_zero_transpoes.shape[0],j:j+conv_origin_size_lab
el_one_zero_transpoes.shape[1],:]*\

(conv_origin_size_label_one_zero_transpoes[:, :, :, dk_1]*grad_cost_transpose[:, :, :, dk_1]))
        '''上式中最后一个参数 conv_origin_size_label_one_zero 还需要乘上 梯度项'''
    self.gradients[self.inbound_nodes[1]] += weight_gradient.transpose(1,2,0,3)

```

5.1.3.2 辅助函数

矩阵旋转

```

def mat_rotat90(filter_, i): #将矩阵 input_array 旋转 i 次
    temp = np.zeros_like(filter_)
    for ind1 in range(filter_.shape[-1]):
        for ind2 in range(filter_.shape[-2]):
            temp2 = filter_[ :, :, ind2, ind1].tolist()
            for _ in range(i):
                temp2 = list(map(list,zip(*temp2[::-1])))
            temp[:, :, ind2, ind1] = np.array(temp2)
    return temp

```

#补 0

```

def zerospadding(input_,kernal_size=[0,0],mode=1):

```



```

if(mode==0):
    output_size_h=input_.shape[1]+kernal_size[0]-1
    output_size_w=input_.shape[2]+kernal_size[1]-1
    err_height=kernal_size[0]-1
    zero_before_height=int(err_height/2)
    zero_after_height=err_height-zero_before_height
    err_width=kernal_size[1]-1
    zero_before_width=int(err_width/2)
    zero_after_width=err_width-zero_before_width
    output_=np.zeros((input_.shape[0],output_size_h,output_size_w,input_.shape[3]))
    if(zero_after_height!=0 and zero_after_width!=0):
        output_[zero_before_height:-zero_after_height,zero_before_width:-zero_after_width,:]=input_
    elif(zero_after_height==0 and zero_after_width!=0):
        output_[zero_before_height:,zero_before_width:-zero_after_width,:]=input_
    elif(zero_after_height!=0 and zero_after_width==0):
        output_[zero_before_height:-zero_after_height,zero_before_width,:]=input_
    elif(zero_after_height==0 and zero_after_width==0):
        output_[zero_before_height:,zero_before_width,:]=input_
    return output_
elif(mode==1):
    output_size_h=input_.shape[1]+kernal_size[0]-1
    output_size_w=input_.shape[2]+kernal_size[1]-1
    err_height=kernal_size[0]-1
    zero_after_height=int(err_height/2)
    zero_before_height=err_height-zero_after_height
    err_width=kernal_size[1]-1
    zero_after_width=int(err_width/2)
    zero_before_width=err_width-zero_after_width
    output_=np.zeros((input_.shape[0],output_size_h,output_size_w,input_.shape[3]))
    if(zero_after_height!=0 and zero_after_width!=0):
        output_[zero_before_height:-zero_after_height,zero_before_width:-zero_after_width,:]=input_
    elif(zero_after_height==0 and zero_after_width!=0):
        output_[zero_before_height:,zero_before_width:-zero_after_width,:]=input_
    elif(zero_after_height!=0 and zero_after_width==0):
        output_[zero_before_height:-zero_after_height,zero_before_width,:]=input_
    elif(zero_after_height==0 and zero_after_width==0):
        output_[zero_before_height:,zero_before_width,:]=input_
    return output_
elif(mode==2):
    output_size_h=input_.shape[0]
    output_size_w=input_.shape[1]
    output_=input_
    return output_
elif(mode==3):
    output_size_h=input_.shape[1]+2*(kernal_size[0]-1)
    output_size_w=input_.shape[2]+2*(kernal_size[1]-1)
    zero_before_height=kernal_size[0]-1

```

```

zero_after_height=kernal_size[0]-1
zero_before_width=kernal_size[1]-1
zero_after_width=kernal_size[1]-1
output_=np.zeros((input_.shape[0],output_size_h,output_size_w,input_.shape[3]))
if(zero_after_height!=0 and zero_after_width!=0):
    output_[zero_before_height:zero_after_height,zero_before_width:zero_after_width,:]=input_
elif(zero_after_height==0 and zero_after_width!=0):
    output_[zero_before_height:,zero_before_width:zero_after_width,:]=input_
elif(zero_after_height!=0 and zero_after_width==0):
    output_[zero_before_height:zero_after_height,zero_before_width,:]=input_
elif(zero_after_height==0 and zero_after_width==0):
    output_[zero_before_height:,zero_before_width,:]=input_
return output_

```

5.2 bias_add 类

bias_add 类是为了实现卷积输出与偏置项线性相加类，用来实现下式的运算操作【公式中参数详细的说明请参考深度学习基础模型算法原理及编程实现--05.卷积神经网络，全文同】

$$net_{d_{k+1}}^k(m, n) = conv_{d_{k+1}}^k(m, n) + b_{d_{k+1}}^k, 1 \leq m \leq F_{k+1}, 1 \leq n \leq H_{k+1} \quad (1.64)$$

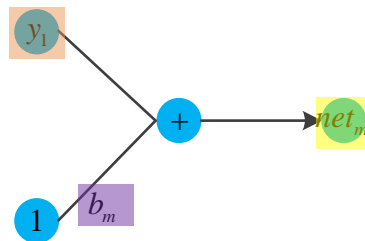
为了方便编程实现，将上式整理成矩阵运算的形式，有：

$$\begin{aligned} [net_{d_{k+1}}^k]_{F_{k+1} \times H_{k+1}} &= [conv_{d_{k+1}}^k]_{F_{k+1} \times H_{k+1}} + b_{d_{k+1}}^k [ones(F_{k+1}, H_{k+1})]_{F_{k+1} \times H_{k+1}} \\ [net^k]_{F_{k+1} \times H_{k+1} \times D_{k+1}} &= [conv^k]_{F_{k+1} \times H_{k+1} \times D_{k+1}} + b_{d_{k+1}}^k [ones(F_{k+1}, H_{k+1})]_{F_{k+1} \times H_{k+1}} \end{aligned} \quad (1.65)$$

忽略层数 k 的影响，上式可整理为：

$$net_{d_{k+1}}(m, n) = conv_{d_{k+1}}(m, n) + b_{d_{k+1}}, 1 \leq m \leq F_{k+1}, 1 \leq n \leq H_{k+1} \quad (1.66)$$

下面再通过简单的图示对线性加权进行说明：



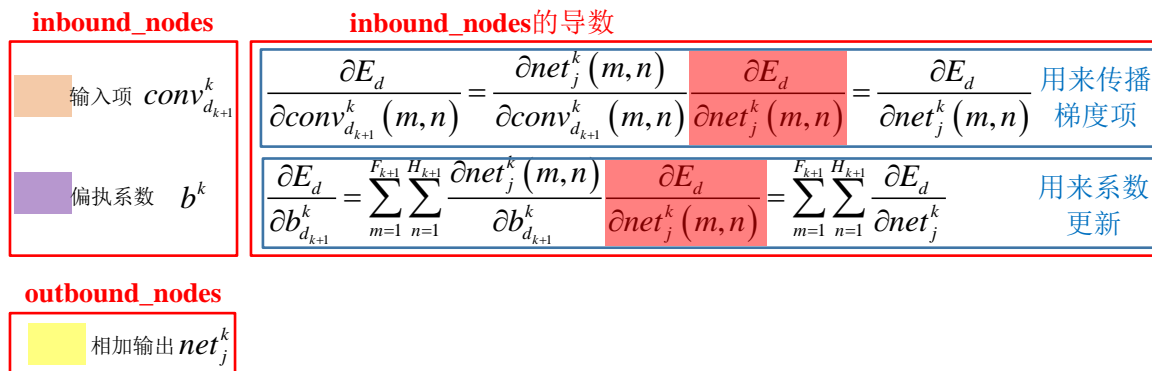
由前文知，Add类中同样有inbound_nodes和outbound_nodes，详细如下图所示，为了方便对比，其中inbound_nodes和outbound_nodes中不同项所对应的方框颜色与上图中的方框颜色一致。且正如下图所示，outbound_nodes的导数不要求，因为在当前类中没有可以用来求其导数的信息，并且当前节点的outbound_nodes是下一个节点的inbound_nodes，所以outbound_nodes的导数在下一个节点中被计算。

此外，inbound_nodes的导数被表示为当前类中的输入项与损失函数对输出项偏导数【如下图所示， $\frac{\partial E_d}{\partial net_j^k}$ 】

的乘积，其中当前类中的输入项是已知的，而 $\frac{\partial E_d}{\partial net_j^k}$ 并不知，但 $\frac{\partial E_d}{\partial net_j^k}$ 具体的值可通过访问下一个神经元对输入项的偏导数得到。

Bias_add类中的输入项有2个，对不同的输入项的偏导数有不同的作用，如下图所示， $\frac{\partial E_d}{\partial conv_{d_{k+1}}^k}$ 用来传播梯度

项， $\frac{\partial E_d}{\partial b^k}$ 用来更新系数。



为了方便编程实现，将上图中 3 个偏导数写成矩阵运算的形式，有：

$$\begin{aligned} \left[\frac{\partial E_d}{\partial conv^k} \right]_{batchsize \times F^k \times H^k \times D^{k+1}} &= \left[\frac{\partial E_d}{\partial net_{d_{k+1}}^k} \right]_{batchsize \times F^k \times H^k \times D^{k+1}} \\ \left[\frac{\partial E_d}{\partial b_{d_{k+1}}^k} \right]_{1 \times D^{k+1}} &= \sum_{\text{对 } batchsize / F^k / H^k \text{ 求和}} \left(\left[\frac{\partial E_d}{\partial net_{d_{k+1}}^k} \right]_{batchsize \times F^k \times H^k \times D^{k+1}} \right)_{1 \times D^{k+1}} \end{aligned} \quad (1.67)$$

上图中详细的符号定义及代码推导请参考【[深度学习基础模型算法原理及编程实现--03.全链接][3]】。相应的代码实现如下：

```
class bias_add(Node):
    """卷积输出与偏执项相加"""
    def __init__(self, conv, bias, name=[]):
        Node.__init__(self, inbound_nodes=[conv, bias], name=name)

    def forward(self):
        # self.value=np.zeros_like(self.inbound_nodes[0].value)
        # for i in range(self.inbound_nodes[0].value.shape[-1]):
        #     self.value[:, :, :, i]=self.inbound_nodes[0].value[:, :, :, i]+self.inbound_nodes[1].value[i]
        conv = self.inbound_nodes[0].value
        bias = self.inbound_nodes[1].value
        self.value = conv + bias

    def backward(self):
        self.gradients = {n: np.zeros_like(n.value) for n in self.inbound_nodes}
        for n in self.outbound_nodes:
            grad_cost = n.gradients[self]
            self.gradients[self.inbound_nodes[0]] += grad_cost #axis=0表示对列求和
```

```
self.gradients[self.inbound_nodes[1]] += np.sum(np.sum(np.sum(grad_cost,0),0),0)
```

5.3 max_pool 类

max_pool 用来实现最大下采样，一般接于卷积层之后，tensorflow 中的 max_pool 函数定义如下，tf.nn.max_pool(value,ksize,strides,padding,name=None)，参数和卷积很像，一共有 4 个：

第一个参数 value：需要池化的输入，一般是 feature map，维度是[batch, height, width, channels]

第二个参数 ksize：池化窗口的大小，是一个长度为四的向量，一般是[1, height, width, 1]

第三个参数 strides：和卷积类似，窗口在每一个维度上滑动的步长，一般也是[1, stride,stride, 1]

第四个参数 padding：和卷积类似，可以取'VALID' 或者'SAME'

返回一个 Tensor，类型不变，shape 仍然是[batch, height, width, channels]这种形式

主要实现下式的运算操作【[公式中参数详细的说明请参考深度学习基础模型算法原理及编程实现--03.卷积神经网络]，全文同】

5.3.1 对于 padding='same'的模式，有

前向计算实现下式的运算操作【公式中参数详细的说明请参考深度学习基础模型算法原理及编程实现--05.卷积神经网络，全文同】

$$net_{d_{k+1}}^k(m,n) = \max \left(\text{zerospadding} \left(A_{d_k}^k, KF_k - 1, KH_k - 1, 0 \right) [m : KF_k + m - 1, n : KH_k + n - 1] \right) \quad (1.68)$$

$$1 \leq m \leq F_k, 1 \leq n \leq H_k$$

下面再通过简单的图示对线性加权进行说明：

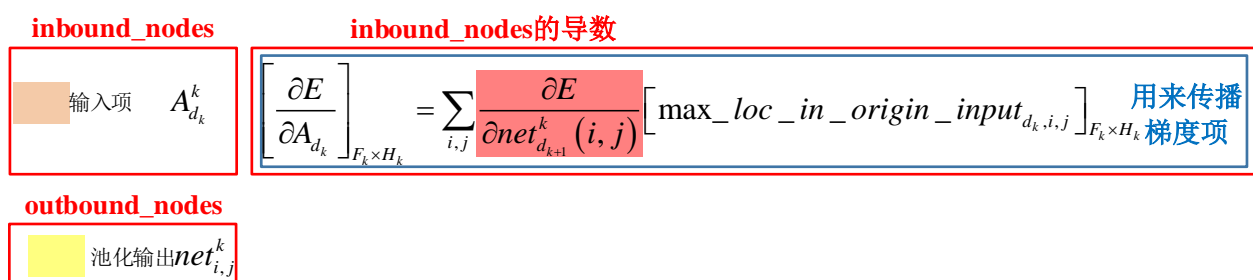
待补充

由前文知，max_pool类中同样有inbound_nodes和outbound_nodes，详细如下图所示，为了方便对比，其中inbound_nodes和outbound_nodes中不同项所对应的方框颜色与上图中的方框颜色一致。且正如下图所示，outbound_nodes的导数不要求，因为在当前类中没有可以用来求其导数的信息，并且当前节点的outbound_nodes是下一个节点的inbound_nodes，所以outbound_nodes的导数在下一个节点中被计算。

此外，inbound_nodes的导数被表示为当前类中的输入项与损失函数对输出项偏导数【如下图所示， $\frac{\partial E_d}{\partial net_j^k}$ 】

的乘积，其中当前类中的输入项是已知的，而 $\frac{\partial E_d}{\partial net_j^k}$ 并不知，但 $\frac{\partial E_d}{\partial net_j^k}$ 具体的值可通过访问下一个神经元对输入项的偏导数得到。

Max_pool类中的输入项有4个，分别为需要池化的输入、池化窗口的大小、窗口采样步长、padding模式，但只需对需要池化的输入求导数，如下图所示， $\frac{\partial E_d}{\partial A_{d_k}^k}$ 用来传播梯度项。



为了方便编程实现，将上图中的偏导数写成矩阵运算的形式，有：

$$\left[\frac{\partial E}{\partial A_{d_k}} \right]_{F_k \times H_k} = \sum_{i,j} \frac{\partial E}{\partial net_{d_{k+1}}^k(i,j)} \delta_{i,j}^k \left[\max_loc_in_origin_input_{d_k,i,j} \right]_{F_k \times H_k} \quad (1.69)$$

5.3.2 对于 padding='valid'的模式，有

前向计算实现下式的运算操作【公式中参数详细的说明请参考深度学习基础模型算法原理及编程实现--05.卷积神经网络，全文同】

$$net_{d_{k+1}}^k(m,n) = \max \left(A_{d_k}^k \left[m:KF_k+m-1, n:KH_k+n-1 \right] \right) \quad (1.70)$$

$$1 \leq m \leq F_k - KF_k + 1, 1 \leq n \leq H_k - KH_k + 1$$

下面再通过简单的图示对最大下采样进行说明：

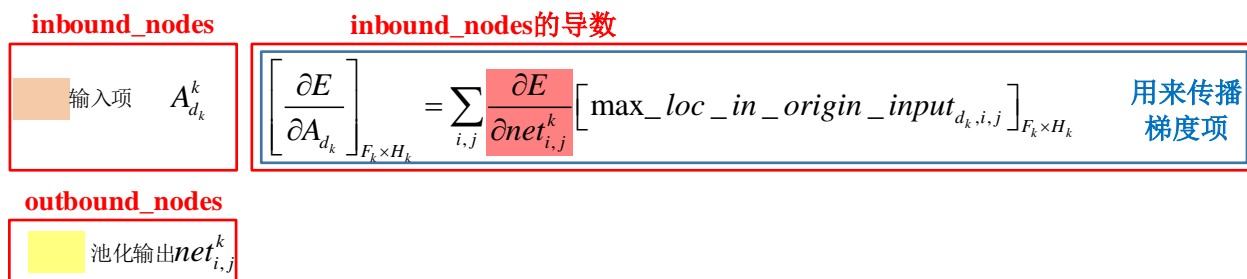
待补充

由前文知，max_pool类中同样有inbound_nodes和outbound_nodes，详细如下图所示，为了方便对比，其中inbound_nodes和outbound_nodes中不同项所对应的方框颜色与上图中的方框颜色一致。且正如下图所示，outbound_nodes的导数不要求，因为在当前类中没有可以用来求其导数的信息，并且当前节点的outbound_nodes是下一个节点的inbound_nodes，所以outbound_nodes的导数在下一个节点中被计算。

此外，inbound_nodes的导数被表示为当前类中的输入项与损失函数对输出项偏导数【如下图所示， $\frac{\partial E_d}{\partial net_j^k}$ 】

的乘积，其中当前类中的输入项是已知的，而 $\frac{\partial E_d}{\partial net_j^k}$ 并不知，但 $\frac{\partial E_d}{\partial net_j^k}$ 具体的值可通过访问下一个神经元对输入项的偏导数得到。

Max_pool类中的输入项有4个，分别为需要池化的输入、池化窗口的大小、窗口采样步长、padding模式，但只需对需要池化的输入求导数，如下图所示， $\frac{\partial E_d}{\partial A_{d_k}^k}$ 用来传播梯度项。



为了方便编程实现，将上图中的偏导数写成矩阵运算的形式，有：

$$\left[\frac{\partial E}{\partial A_{d_k}} \right]_{F_k \times H_k} = \sum_{i,j} \delta_{i,j}^k \left[\max_loc_in_origin_input_{d_k,i,j} \right]_{F_k \times H_k} \quad (1.71)$$

5.4 avg_pool 类

5.4.1 对于 padding='same'的模式，有

前向计算实现下式的运算操作【公式中参数详细的说明请参考深度学习基础模型算法原理及编程实现--05.卷积神经网络，全文同】

$$net_{d_{k+1}}^k(m,n) = \text{mean} \left(\text{zerospadding} \left(A_{d_k}^k, KF_k - 1, KH_k - 1, 0 \right) \left[m:KF_k+m-1, n:KH_k+n-1 \right] \right) \quad (1.72)$$

$$1 \leq m \leq F_k, 1 \leq n \leq H_k$$

下面再通过简单的图示对线性加权进行说明：

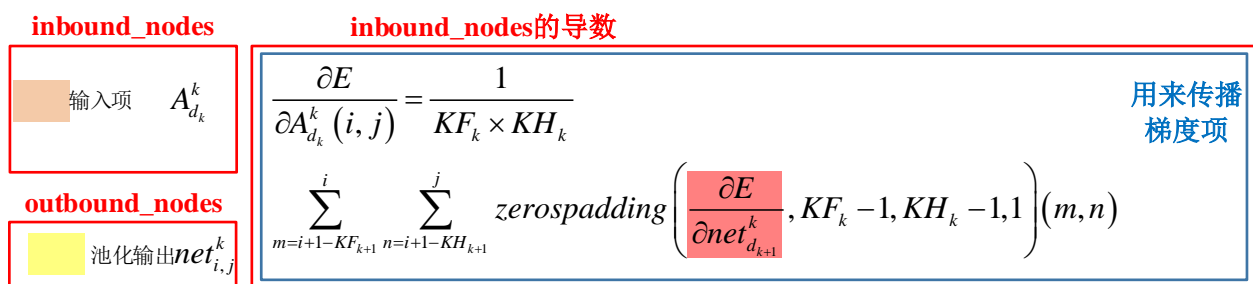
待补充

由前文知，max_pool类中同样有inbound_nodes和outbound_nodes，详细如下图所示，为了方便对比，其中inbound_nodes和outbound_nodes中不同项所对应的方框颜色与上图中的方框颜色一致。且正如下图所示所述，outbound_nodes的导数不要求，因为在当前类中没有可以用来求其导数的信息，并且当前节点的outbound_nodes是下一个节点的inbound_nodes，所以outbound_nodes的导数在下一个节点中被计算。

此外，inbound_nodes的导数被表示为当前类中的输入项与损失函数对输出项偏导数【如下图所示， $\frac{\partial E_d}{\partial net_j^k}$ 】

的乘积，其中当前类中的输入项是已知的，而 $\frac{\partial E_d}{\partial net_j^k}$ 并不知，但 $\frac{\partial E_d}{\partial net_j^k}$ 具体的值可通过访问下一个神经元对输入项的偏导数得到。

Max_pool类中的输入项有4个，分别为需要池化的输入、池化窗口的大小、窗口采样步长、padding模式，但只需对需要池化的输入求导数，如下图所示， $\frac{\partial E_d}{\partial A_{d_k}^k}$ 用来传播梯度项。



为了方便编程实现，将上图中的偏导数写成矩阵运算的形式，有：

$$\left[\frac{\partial E}{\partial A^k} \right]_{batchsize \times F_k \times H_k \times D_k} = \frac{1}{KF_k \times KH_k} \left[\text{zerospadding}(\delta^k, KF_k - 1, KH_k - 1, 1) \right]_{batchsize \times (F_{k+1} + KF_k - 1) \times (H_{k+1} + KH_k - 1) \times D_{k+1}} \quad (1.73)$$

$$* ([1]_{D_k \times KF_k \times KH_k \times D_{k+1}}) . \text{reshap}(KF_k, KH_k, D_{k+1}, D_k)$$

5.4.2 对于 padding='valid' 的模式，有

前向计算实现下式的运算操作【公式中参数详细的说明请参考深度学习基础模型算法原理及编程实现--05.卷积神经网络，全文同】

$$net_{d_{k+1}}^k(m, n) = \text{mean}\left(A_{d_k}^k[m : KF_k + m - 1, n : KH_k + n - 1]\right) \quad (1.74)$$

$$1 \leq m \leq F_k - KF_k + 1, 1 \leq n \leq H_k - KH_k + 1$$

下面再通过简单的图示对线性加权进行说明：

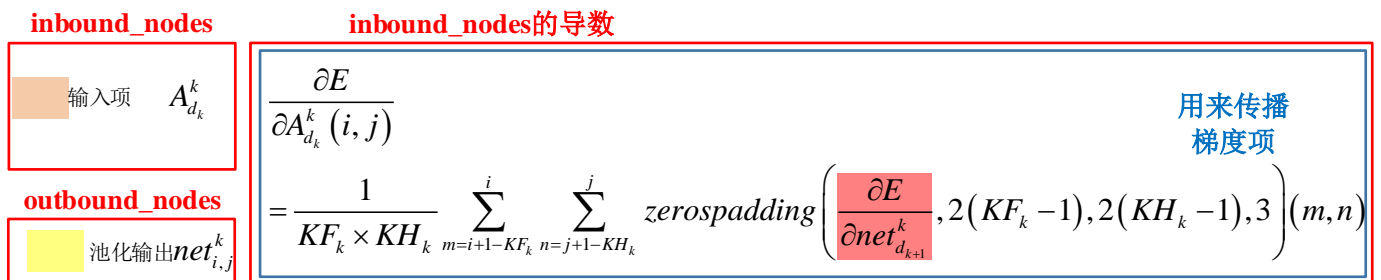
待补充

由前文知，max_pool类中同样有inbound_nodes和outbound_nodes，详细如下图所示，为了方便对比，其中inbound_nodes和outbound_nodes中不同项所对应的方框颜色与上图中的方框颜色一致。且正如下图所示所述，outbound_nodes的导数不要求，因为在当前类中没有可以用来求其导数的信息，并且当前节点的outbound_nodes是下一个节点的inbound_nodes，所以outbound_nodes的导数在下一个节点中被计算。

此外, `inbound_nodes`的导数被表示为当前类中的输入项与损失函数对输出项偏导数【如下图所示, $\frac{\partial E_d}{\partial net_j^k}$ 】

的乘积, 其中当前类中的输入项是已知的, 而 $\frac{\partial E_d}{\partial net_j^k}$ 并不知, 但 $\frac{\partial E_d}{\partial net_j^k}$ 具体的值可通过访问下一个神经元对输入项的偏导数得到。

`Max_pool`类中的输入项有4个, 分别为需要池化的输入、池化窗口的大小、窗口采样步长、`padding`模式, 但只需对需要池化的输入求导数, 如下图所示, $\frac{\partial E_d}{\partial A_{d_k}^k}$ 用来传播梯度项。



为了方便编程实现, 将上图中的偏导数写成矩阵运算的形式, 有:

$$\left[\frac{\partial E}{\partial A^k} \right]_{batchsize \times F_k \times H_k \times D_k} = \frac{1}{KF_k \times KH_k} \left[\text{zerospadding} \left(\left[\delta^k \right]_{batchsize \times (F_{k+1}-KF_k+1) \times (H_{k+1}-KH_k+1) \times D_{k+1}}, 2(KF_k - 1), 2(KH_k - 1), 3 \right) \right]_{batchsize \times (F_{k+1}+KF_k-1) \times (H_{k+1}+KH_k-1) \times D_{k+1}} * \left([1]_{D_k \times KF_k \times KH_k \times D_{k+1}} \right). \text{reshap}(KF_k, KH_k, D_{k+1}, D_k)$$

(1.75)

5.5 dropout 类

5.6 编程实现

5.6.1 验证算例

为了对比分别调用 `tensorflow` 和 `ANNbox` 库函数计算结果的差别, 分别用同一段程序调用 `tensorflow` 库及 `ANNbox` 库函数对比计算结果。详细代码文件在: 中, `beyond compare` 对比两个版本的计算程序如下:

```

import tensorflow as tf
fout_Debug_before=open("debug_before_run.txt","w")
fout_Debug_after=open("debug_after_run.txt","w")
sess = tf.InteractiveSession()
W1 = tf.Variable(np.loadtxt('W1_784_10.txt'), name='image')
cross_entropy=tf.nn.softmax_cross_entropy_with_logits(labels=y_, logits=y)
sess=tf.Session()
sess.run(tf.global_variables_initializer())
tem_mal=mal.eval({x: batch_xs, y_: batch_ys})
tem_yy=y.eval({x: batch_xs, y_: batch_ys})
tem_cross_entropy=cross_entropy.eval({x: batch_xs, y_: batch_ys})
loss_list.append(np.mean(cross_entropy.eval({x: batch_xs, y_: batch_ys})))
acc_train_list.append(np.mean((np.argmax(y_.eval({x: batch_xs, y_: batch_ys}),1) == np.argmax(bat
acc_test_list.append(np.mean((np.argmax(y_.eval({x: mnist.test.images, y_: mnist.test.labels}),1)
tem_mal=mal.eval({x: batch_xs, y_: batch_ys})
tem_yy=y.eval({x: batch_xs, y_: batch_ys})
tem_cross_entropy=cross_entropy.eval({x: batch_xs, y_: batch_ys})

import ANNbox as tf
fout_Debug_before = open("debug_before_run_miniflow.txt","w")
fout_Debug_after = open("debug_after_run_miniflow.txt","w")
#sess = tf.InteractiveSession()
W1 = tf.Variable(np.loadtxt('W1_784_10.txt'), name='image')
cross_entropy = tf.nn.softmax_cross_entropy_with_logits(labels=y_, logits=y, name='cost') #loss = tf.re
#cross_entropy = tf.nn.softmax_cross_entropy_with_logits(labels=y_, logits=tf.nn.softmax(y), name='cost')
#sess=tf.Session()
#sess.run(tf.global_variables_initializer())
tem_mal=mal.my_eval({x: batch_xs, y_: batch_ys})
tem_yy=y.my_eval({x: batch_xs, y_: batch_ys})
tem_cross_entropy=cross_entropy.my_eval({x: batch_xs, y_: batch_ys})
loss_list.append(np.mean(cross_entropy.my_eval({x: batch_xs, y_: batch_ys})))
acc_train_list.append(np.mean((np.argmax(y_.my_eval({x: batch_xs, y_: batch_ys}),1) == np.argmax(bat
acc_test_list.append(np.mean((np.argmax(y_.my_eval({x: mnist.test.images, y_: mnist.test.labels}),1)
tem_mal=mal.my_eval({x: batch_xs, y_: batch_ys})
tem_yy=y.my_eval({x: batch_xs, y_: batch_ys})
tem_cross_entropy=cross_entropy.my_eval({x: batch_xs, y_: batch_ys})
  
```

01.其中第一个差异是调用库函数的差异:

import **tensorflow** as tf

import **ANNbox** as tf

所以证明了 ANNbox 库函数的有效性。

6 自己编写神经网络库 ANNbox【高仿 tensorflow】_05 实现 RNN (01)

为了实现 RNN，真是费了老劲了，幸好工作氛围不错，业余还有精力来编 ANNbox，在这里要感谢单位。我想尽量用之前定义好的函数来实现最基本的 RNN。其实许多时候，在 tensorflow 中，我们可以不用调用诸如 rnn_cell.BasicRNNCell、tf.nn.rnn 这样的函数，而是通过自己定义 rnn 模块的前向计算函数，利用 tensorflow 强大的计算图结构管理就可以实现循环神经网络。而实现这些需要以下 4 点技术支持：

01. 拓扑结构管理
02. 域名控制 (variable_scope)
03. 各类逻辑操作 (concat/unstack)
04. 基本的数学运算 (对各类 tensor 的加减乘除/激活函数/优化器/.....)

通过前面几节工作的铺垫，到这里，第 1 点及第 4 点都已经基本实现，本节着重介绍第 2 点及第 3 点的原理实现，最后给出一个利用框架实现二进制加法运算的算例。

6.1 variable_scope 类 (利用上下文管理器实现变量的作用域限定)

变量名是在基类 Node 中定义的，所以用全局变量 scope_name 来传递作用域名称（默认值为''），scope_name 在 variable_scope 的初始化函数中被赋值，并在 variable_scope 上下文作用域范围结束的时候重新赋为空字符。用 global_resue(默认值为 FALSE)来传递同名 get_variable 变量是否重用，在 variable_scope 上下文作用域范围结束的时候重新赋为 False。相应的编程实现如下：

```
class variable_scope():
    def __init__(self, local_scope_name='', reuse=False):
        scope_name[0] = local_scope_name
        global_resue[0] = reuse

    def __enter__(self):
        '''enter() 返回一个对象。上下文管理器会使用这一对象作为 as 所指的变量'''
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        '''
        结束上下文作用域的范围, 并将 global_resue 设置为 0
        '''
        scope_name[0] = ''
        global_resue[0] = False
```

为了实现 name 传递，我在 Node 基类中对变量名称加上了 variable_scope 域名对其命名的管理

```
class Node(object):
    def __init__(self, inbound_nodes=[], name='', value=None, shape=[], is_get_variable=False):
        .....
        if(scope_name[0]!='' and name!=''):
            self.name=scope_name[0]+'/'+name
        if(scope_name[0]!='' and name==''):
            self.name=scope_name[0]
        if(scope_name[0]=='' and name!=''):
            self.name=name
        .....
```

6.2 get_variable 类 (创建共享变量)

get_variable 和 variable 的区别网上有许多，这里不再相信介绍。但我需要声明一下循环神经网络中为什么要用到 get_variable()，其实这里的 get_variable 主要是用来创建共享变量，由前面 深度学习基础模型算法原理及编程实现 --06. 循环神经网络[<https://blog.csdn.net/drillistbox/article/details/79720859>] 中不难发现，在 RNN 的每层神经网络中，沿时间轴及深度轴传递的权重矩阵都是一致的，即每个权重矩阵会用到许多次，但用的都是同一内存上的变量，且

还需分别更新，因此需要用的 `get_variable` 创建共享变量：即在 `reuse=true` 及 `get_variable` 变量拥有同样的名字的情况下，不同的 `get_variable` 实例对象拥有不同地址，但是其属性 `self.value` 是共享的，共享的 `value` 值用全局共用字典 `get_variable_dict` 来管理。

在编程实现的时候，我们用一个全局变量 `get_variable_dict` 来存放 `get_variable` 实例对象的 `value` 值，每次创建 `get_variable` 实例的时候需判定当前对象在 `get_variable_dict` 中是否存在同名变量，并且当前 `variable_scope` 中的 `reuse` 是否为 `True`，通过这两个因素来决定是重复调用 `get_variable_dict` 中存放的 `get_variable` 实例对象的值。

如果我们在计算图中某个节点定义了一个变量 `w`，后来又在其他节点用 `get_variable` 来同一作用域下定义同名变量 `w/w`，下面的 3 个图给出了大致流程：

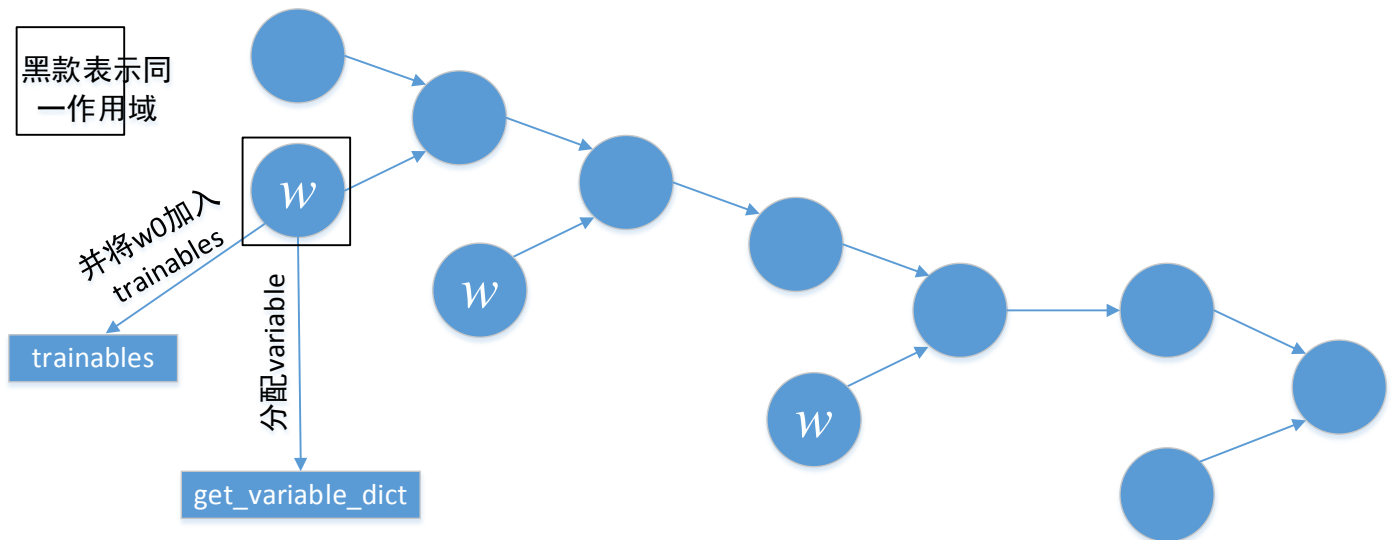


图 1 中首先定义了 `get_variable` 实例对象 `w`，由于当前 `get_variable_dict` 的键值中没有同名变量，所以在 `get_variable_dict` 中分配了权重系数的内存空间

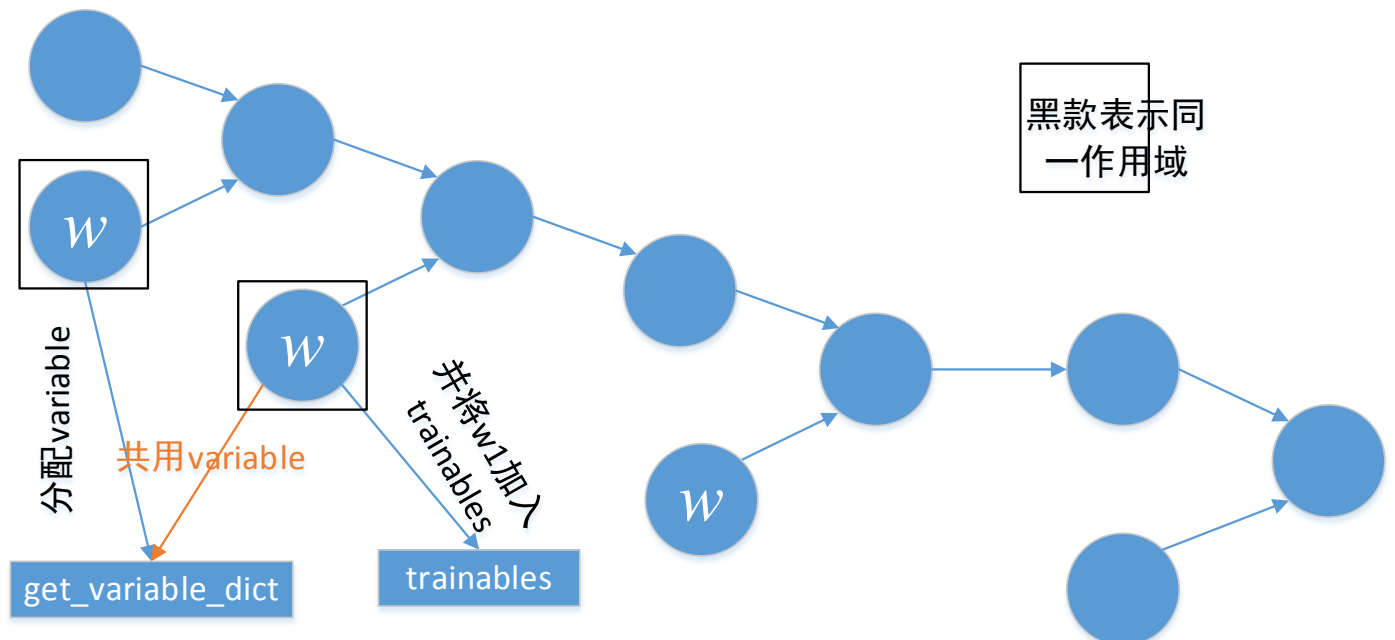


图 2 中定义了一个同名的 `get_variable` 实例对象 `w`，且当前作用域的 `reuse` 为 `true`，由于当前 `get_variable_dict` 的键值中有同名变量，所以共享在 `get_variable_dict` 已有的权重系数的内存空间

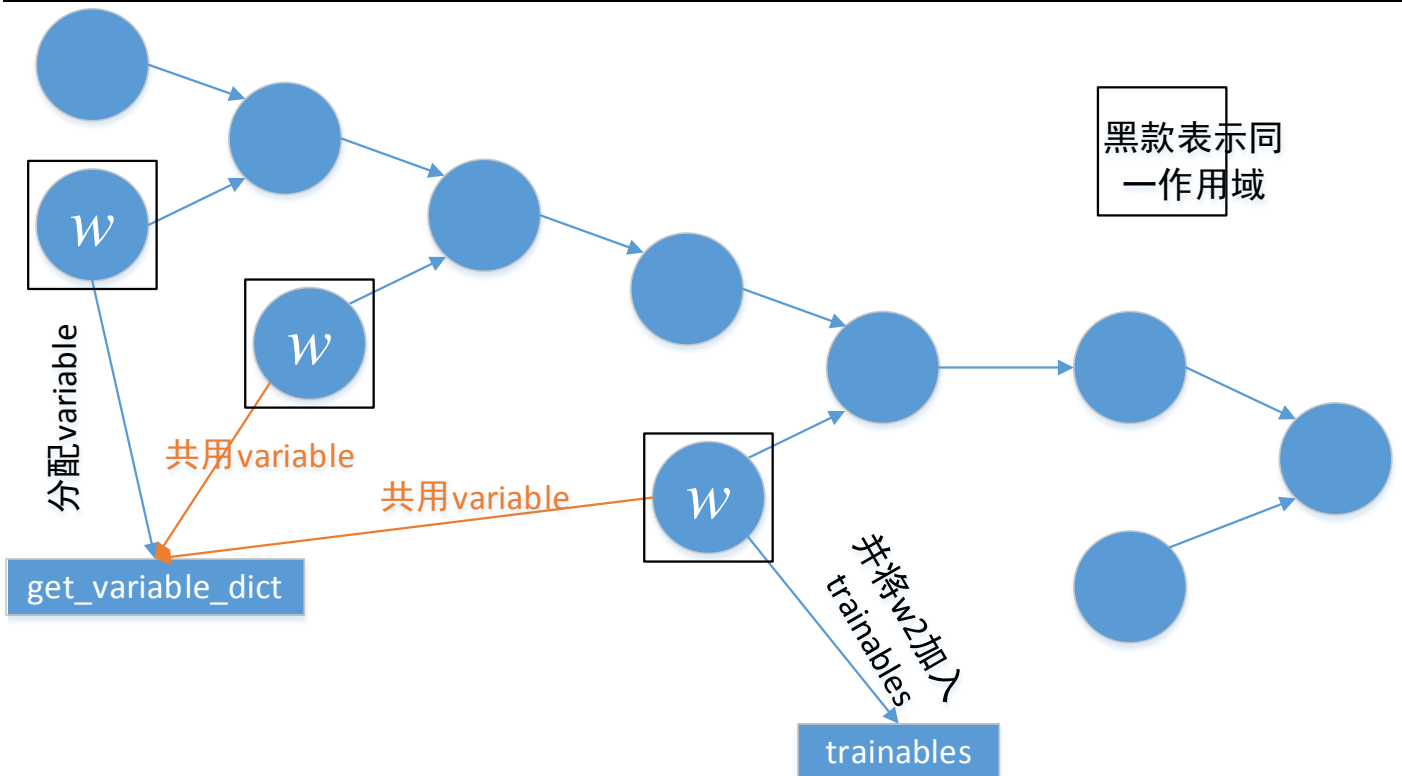


图 3 中定义了一个同名的 `get_variable` 实例对象 `w`，且当前作用域的 `reuse` 为 `true`，由于当前 `get_variable_dict` 的键值中有同名变量，所以共享在 `get_variable_dict` 已有的权重系数的内存空间。编程实现如下：

```
class get_variable(Node):
    '''变量节点,功能仿造 tensorflow 中的 get_variable,用于存放输入权重系数及偏执系数'''
    def __init__(self,name='',shape=[],initializer=None):
        '''输入层的节点没有传入节点(inbound nodes),所有无需向构造函数中传递任何 inbound nodes'''
        Node.__init__(self,name=name,value=None,is_get_variable=True)
        self.shape=shape
        if(self.name in get_variable_dict.keys() and global_reuse[0]==True):
            '''get_variable_dict 中存在同名变量,并且可以重用,因此无需分配内存'''
            self.value=get_variable_dict[self.name]
        if(self.name in get_variable_dict.keys() and global_reuse[0]==False):
            '''get_variable_dict 中存在同名变量,需要将 reuse 设为 True,否则为 False'''
            raise Exception(self,"already exist in current scope, do you want to set reuse to
True!")
        #         '''get_variable_dict 中存在同名变量,并且不可以重用,因此需管理名称,并分配内存'''
        #         '''名称管理'''
        #         ind=0
        #         while self.name+str(ind) in get_variable_dict.keys():
        #             ind+=1
        #         self.name=self.name+str(ind)
        #         '''重新分配内存'''
        #         get_variable_dict[self.name]=initializer(shape)
        #         self.value=get_variable_dict[self.name]
        #         '''暂时封存'''
        ##         trainables.append(self) #将 variable 添加进 trainables 列表中
        if(self.name not in get_variable_dict.keys()):
            '''get_variable_dict 中不存在同名变量,则不管能不能重用,均需分配内存'''
```

```

get_variable_dict[self.name]=initializer(shape)
self.value=get_variable_dict[self.name]
'''暂时封存'''
#         trainables.append(self) #将 variable 添加进 trainables 列表中
'''所有的 get_variable 变量都要加入 trainables'''
trainables.append(self) #将 variable 添加进 trainables 列表中
if(isLEnd[0]==False):L_initial_with_variable_and_constant.append(self) #将 variable
存入节点列表 L 中

def forward(self):
    pass
#         self.value = self.initializer(self.shape)

def backward(self):
    self.gradients={self:np.zeros_like(self.value)}
    for n in self.outbound_nodes:
        grad_cost=n.gradients[self]
        self.gradients[self]+=grad_cost*1

```

6.3 concat 类（多个张量按指定维度合并为一个张量）

concat 类是用来组合多个张量的，TensorFlow 中的定义如下：

```
tf.concat(concat_dim, values, name='concat')
```

concat_dim 是 tensor 连接的方向（维度），values 是要连接的 tensor 链表，name 是操作名。待连接的张量在 concat_dim 维度的尺寸可以不一样，其他维度的尺寸必须一样。下面举两个例子：两个二维 tensor 连接，两个三维 tensor 连接。

输入项 o 为多个待合并的节点的集合，用 list 或者元组表示，假设要在第 axis 维度对所有输入进行合并，那么所有输入节点的其它维度的长度应该是一样的，设第 i 个输入节点为 o_i ，相应的 shape 为 $[s_1, s_1, \dots, s_{axis-i}, \dots, s_N]$ ，

那么合并后的张量的 shape 为 $[s_1, s_1, \dots, \sum_{i=1}^{\text{输入节点数}} s_{axis-i}, \dots, s_N]$ 。主要实现以下运算：

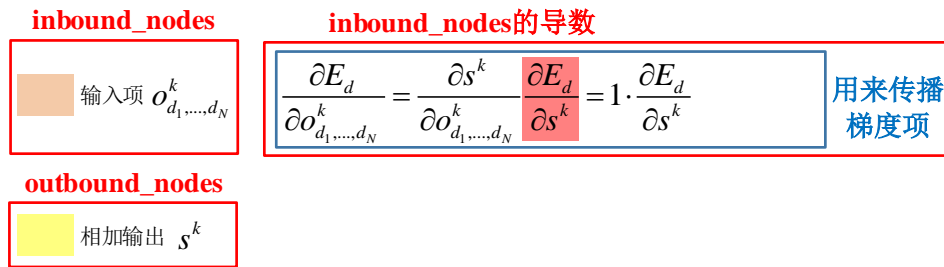
$$[s]_{\begin{matrix} s_1 \times s_2 \times \dots \times \\ \sum_{i=1}^{\text{输入节点数}} s_{axis-i} \times \dots \times s_N \end{matrix}} = \left[[o_1]_{s_1 \times s_2 \times \dots \times s_{axis-i} \times \dots \times s_N}, [o_2]_{s_1 \times s_2 \times \dots \times s_{axis-i} \times \dots \times s_N}, \dots, [o_m]_{s_1 \times s_2 \times \dots \times s_{axis-i} \times \dots \times s_N} \right] \quad (1.76)$$

由前文知，Squared 类中同样有 inbound_nodes 和 outbound_nodes，详细如下图所示，为了方便对比，其中 inbound_nodes 和 outbound_nodes 中不同项所对应的方框颜色与上图中的方框颜色一致。且正如下图所示，outbound_nodes 的导数不要求，因为在当前类中没有可以用来求其导数的信息，并且当前节点的 outbound_nodes 是下一个节点的 inbound_nodes，所以 outbound_nodes 的导数在下一个节点中被计算。

此外，inbound_nodes 的导数被表示为当前类中的输入项对 inbound_nodes 的导数与 **损失函数对输出项偏导数**【如下图所示， $\frac{\partial E_d}{\partial s^k}$ 】的乘积，其中当前类中的输入项是已知的，而 $\frac{\partial E_d}{\partial s^k}$ 并不知，但 $\frac{\partial E_d}{\partial s^k}$ 具体的值可通过访问下一个神经元对输入项的偏导数得到。

reduce_mean 类中的输入项有 1 个，对输入项的偏导数如下图所示，且 $\frac{\partial E_d}{\partial o_{index}^k}$ 是用来传播梯度项的，其中的 index

表示任意维度下的下标。



为了方便编程实现，将上图中的偏导数写成矩阵运算的形式，对于 redce_mean 输出为 1*1 的情况，有：

$$\left[\frac{\partial E_d}{\partial o^k} \right]_{[s_1 \times s_1 \times \dots \times s_{axis_ind} \times \dots \times s_N]} = \left[\frac{\partial E_d}{\partial s^k} \right] \left(\begin{matrix} \vdots, \vdots, \dots, \sum_{i=1}^{ind-1} s_{axis_i}, \sum_{i=1}^{ind} s_{axis_i}, \dots, \vdots \end{matrix} \right)$$

合并的维度

(1.77)

上图中详细的符号定义及代码推导请参考【深度学习基础模型算法原理及编程实现--03.全链接[3]】。相应的代码实现如下：

```
class concat(Node):
    ...

    tf.concat(concat_dim, values, name='concat')
    这里将多个 tensor 合并为一个 tensor,所以当前类 concat 自身就可以用来做作为合并后的 tensor,无需向
    unstack 中用中间传输节点 transmit_node
    ...

    def __init__(self, axis, *L, name=''):
        Node.__init__(self, inbound_nodes=L[0], name=name)
        self.axis=axis
        #缺少检查所有 inbound_nodes 节点在 除了 self.axis 这一维度上的 shape 是否相等,必须要有
        self.shape=L[0][0].shape[:]
        self.shape[self.axis]=0
        for L_ in L[0]:
            self.shape[self.axis]+=L_.shape[self.axis]

    def forward(self):
        len_shape=len(self.shape)
        # print(self.shape)
        self.value=np.zeros(self.shape)
        ind_start=0
        for L_ in self.inbound_nodes:
            exec('self.value['+'+', '*self.axis+str(ind_start)+' ':'+str(ind_start+L_.shape[self.axis])+',
            :'+str(len_shape-1-self.axis)+']=L_.value')
            ind_start+=L_.shape[self.axis]

    def backward(self):
        len_shape=len(self.shape)
        self.gradients={n: np.zeros_like(n.value) for n in self.inbound_nodes}
        grad_cost=self.outbound_nodes[0].gradients[self]
        ind_start=0
```

```

for in_node in self.inbound_nodes:
    exec('self.gradients[in_node]=grad_cost['+':','*self.axis+str(ind_start)+':'+str(ind_start+
in_node.shape[self.axis])+',':*(len_shape-1-self.axis)+']')
    ind_start+=in_node.shape[self.axis]

```

6.4 unstack 类（一个 R 维张量按指定维度拆分为多个 R-1 维张量）

tf.unstack(value,num=None,axis=0,name='unstack')

tf.unstack()将给定的 R 维张量拆分成 R-1 维张量，将 value 根据 axis 分解成 num 个张量，返回的值是 list 类型，如果没有指定 num 则根据 axis 推断出。

输入项 x 为待拆分的 R 维张量，相应的 shape 为 $[D_1, D_2, \dots, D_i, \dots, D_R]_{1 \times R}$ ，假设要在 $\text{axis}=i$ 维对输入的 R 维张量进行拆分，那么所有输出张量用一个列表表示，列表中有 $N=x.\text{shape}[i]$ 个元素，且所有元素的维度一样，均为 $[D_1, D_1, \dots, D_{i-1}, D_{i+1}, \dots, D_N]_{1 \times (R-1)}$ ，输出 tensor 一般用 list 或者元组表示： $[o_1, o_1, \dots, o_i, \dots, o_N]$ ，其中 o_i 表示第 i 个输出节点。主要实现以下运算：

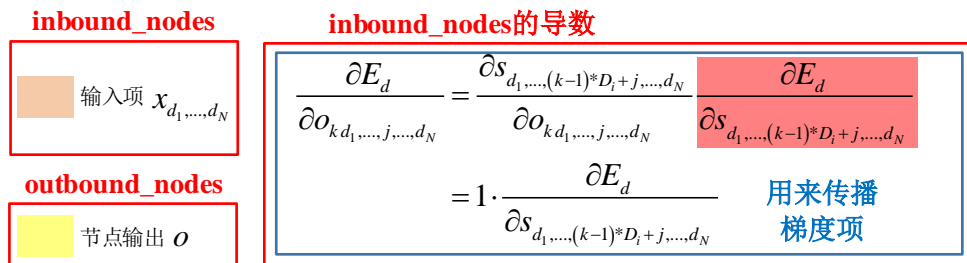
$$\left[[o_1]_{[D_1 \times D_2 \times \dots \times D_{i-1} \times D_{i+1} \times \dots \times D_R]}, [o_2]_{[D_1 \times D_2 \times \dots \times D_{i-1} \times D_{i+1} \times \dots \times D_R]}, \dots, [o_N]_{[D_1 \times D_2 \times \dots \times D_{i-1} \times D_{i+1} \times \dots \times D_R]} \right] = [x]_{[D_1 \times D_2 \times \dots \times D_i \times \dots \times D_R]} \quad (1.78)$$

由前文知，Squared类中同样有inbound_nodes和outbound_nodes，详细如下图所示，为了方便对比，其中inbound_nodes和outbound_nodes中不同项所对应的方框颜色与上图中的方框颜色一致。且正如下图所示，outbound_nodes的导数不要求，因为在当前类中没有可以用来求其导数的信息，并且当前节点的outbound_nodes是下一个节点的inbound_nodes，所以outbound_nodes的导数在下一个节点中被计算。

此外，inbound_nodes的导数被表示为当前类中的输入项对inbound_nodes的导数与损失函数对输出项偏导数【如下图所示， $\frac{\partial E_d}{\partial s}$ 】的乘积，其中当前类中的输入项是已知的，而 $\frac{\partial E_d}{\partial s}$ 并不知，但 $\frac{\partial E_d}{\partial s}$ 具体的值可通过访问下一个神经元对输入项的偏导数得到。

reduce_mean类中的输入项有1个，对输入项的偏导数如下图所示，且 $\frac{\partial E_d}{\partial o_{index}^k}$ 是用来传播梯度项的，其中的index

表示任意维度下的下标。



为了方便编程实现，将上图中的偏导数写成矩阵运算的形式，对于 redce_mean 输出为 1*1 的情况，有：

$$\left[\frac{\partial E_d}{\partial s^k} \right] [:, :, \dots, k:k+1, \dots, :] = \left[\frac{\partial E_d}{\partial o_k} \right] [D_1 \times D_2 \times \dots \times D_{i-1} \times D_{i+1} \times \dots \times D_R]$$

(1.79)

上图中详细的符号定义及代码推导请参考【[深度学习基础模型算法原理及编程实现--06.循环神经网络][6]】。相应的代码实现如下：

```
class unstack(Node):
    ...

    tf.concat(concat_dim, values, name='concat')
    这里将多个 tensor 合并为一个 tensor,所以当前类 concat 自身就可以用来做作为合并后的 tensor,无需向
    unstack 中用中间传输节点 transmit_node
    ...

    def __init__(self, axis, *L, name=''):
        Node.__init__(self, inbound_nodes=L[0], name=name)
        self.axis=axis
        #缺少检查所有 inbound_nodes 节点在 除了 self.axis 这一维度上的 shape 是否相等,必须要有
        self.shape=L[0][0].shape[:]
        self.shape[self.axis]=0
        for L_ in L[0]:
            self.shape[self.axis]+=L_.shape[self.axis]

    def forward(self):
        len_shape=len(self.shape)
        # print(self.shape)
        self.value=np.zeros(self.shape)
        ind_start=0
        for L_ in self.inbound_nodes:
            exec('self.value['+'+', '*self.axis+str(ind_start)+' ':''+str(ind_start+L_.shape[self.axis])+',
            :'+str(len_shape-1-self.axis)+']=L_.value')
            ind_start+=L_.shape[self.axis]

    def backward(self):
        len_shape=len(self.shape)
        self.gradients={n: np.zeros_like(n.value) for n in self.inbound_nodes}
        grad_cost=self.outbound_nodes[0].gradients[self]
        ind_start=0
        for in_node in self.inbound_nodes:
            exec('self.gradients[in_node]=grad_cost['+'+', '*self.axis+str(ind_start)+' ':''+str(ind_start+
            in_node.shape[self.axis])+', :'+str(len_shape-1-self.axis)+']')
            ind_start+=in_node.shape[self.axis]
```


6.5 transmit_node 类（用于 unstack 分解操作的中间暂存变量，不放入 feed_dict_inside【非常重要】）

6.6 验证算例

由于 get_variable 无法像 variable 那样将参数数值作为输入项传给 tensor 对象，并且在设定同样的 seed 情况下，我发现 tensorflow 和 numpy 产生的随机数是不一样的，为什么还不清楚，后续会继续查找缘由。所以我先在 tensorflow 中设定 seed 产生了初始化的权重系数，然后再在 ANNbox 中在 get_variable 初始化的时候读入 tensorflow 中保存的 get_variable 变量初始值【ANNbox 中的 get_variable 是我自己编的，想怎么弄就怎么弄】，以 8 位二进制加法运算为例，将算出的权重系数及偏执系数的中间结果对比打印如下：

从第 1 次迭代到第 130 多次迭代，ANNbox 和 tensorflow 的计算结果是完全一致的。

```
0.175 0.036 -0.021 0.286 0.314 -0.219 0.030 -0.027
epoch:132, W:
-0.119 -1.473 -0.641 -0.534 -0.535 0.177 3.424 0.556
iteration at:132.00,mean loss is:0.18, acc:0.22epoch:133, W:
0.541 3.800 -0.009 0.885 0.957 -0.673 -3.190 0.017
0.686 3.686 -0.352 0.918 1.021 -0.647 -3.091 -0.258
0.064 0.611 0.017 0.244 0.202 -0.058 -0.407 -0.152
-0.063 -0.902 -0.303 -0.057 -0.120 0.203 1.134 0.113
-0.405 -0.449 0.144 -0.411 -0.346 0.262 0.251 0.008
-0.099 0.338 0.159 -0.117 -0.117 -0.074 -0.414 -0.347
0.066 0.610 0.010 -0.059 0.161 0.131 -0.516 -0.430
-0.089 -0.701 -0.092 -0.046 -0.079 0.171 0.307 0.172
-0.194 -1.625 -0.018 -0.229 -0.279 0.354 1.734 0.209
0.176 0.031 -0.012 0.296 0.318 -0.214 0.060 -0.042
epoch:133, W:
-0.096 -1.543 -0.667 -0.523 -0.526 0.163 3.454 0.556
iteration at:133.00,mean loss is:0.07, acc:0.59epoch:134, W:
0.572 3.847 -0.021 0.920 0.991 -0.705 -3.423 0.007
0.716 3.731 -0.354 0.956 1.061 -0.678 -3.331 -0.288
0.078 0.623 0.004 0.253 0.217 -0.070 -0.497 -0.157
-0.058 -0.928 -0.325 -0.067 -0.123 0.204 1.063 0.117
-0.437 -0.452 0.162 -0.439 -0.382 0.287 0.371 0.010
-0.090 0.350 0.143 -0.124 -0.115 -0.081 -0.502 -0.348
0.074 0.624 -0.005 -0.061 0.166 0.124 -0.602 -0.434
-0.098 -0.716 -0.082 -0.049 -0.088 0.178 0.380 0.179
-0.201 -1.679 -0.004 -0.231 -0.293 0.368 1.825 0.222
0.175 0.027 -0.003 0.304 0.321 -0.209 0.091 -0.055
```

```
0.175 0.036 -0.021 0.286 0.314 -0.219 0.030 -0.027
epoch:132, W:
-0.119 -1.473 -0.641 -0.534 -0.535 0.177 3.424 0.556
iteration at:132.00,mean loss is:0.18, acc:0.22epoch:133, W:
0.541 3.800 -0.009 0.885 0.957 -0.673 -3.190 0.017
0.686 3.686 -0.352 0.918 1.021 -0.647 -3.091 -0.258
0.064 0.611 0.017 0.244 0.202 -0.058 -0.407 -0.152
-0.063 -0.902 -0.303 -0.057 -0.120 0.203 1.134 0.113
-0.405 -0.449 0.144 -0.411 -0.346 0.262 0.251 0.008
-0.099 0.338 0.159 -0.117 -0.117 -0.074 -0.414 -0.347
0.066 0.610 0.010 -0.059 0.161 0.131 -0.516 -0.430
-0.089 -0.701 -0.092 -0.046 -0.079 0.171 0.307 0.172
-0.194 -1.625 -0.018 -0.229 -0.279 0.354 1.734 0.209
0.176 0.031 -0.012 0.296 0.318 -0.214 0.060 -0.042
epoch:133, W:
-0.096 -1.543 -0.667 -0.523 -0.526 0.163 3.454 0.556
iteration at:133.00,mean loss is:0.07, acc:0.59epoch:134, W:
0.572 3.847 -0.021 0.920 0.991 -0.705 -3.423 0.007
0.716 3.731 -0.354 0.956 1.061 -0.678 -3.331 -0.288
0.078 0.623 0.004 0.253 0.217 -0.070 -0.497 -0.157
-0.058 -0.928 -0.325 -0.067 -0.123 0.204 1.063 0.117
-0.437 -0.452 0.162 -0.439 -0.382 0.287 0.371 0.010
-0.090 0.350 0.143 -0.124 -0.115 -0.081 -0.502 -0.348
0.074 0.624 -0.005 -0.061 0.166 0.124 -0.602 -0.434
-0.098 -0.716 -0.082 -0.049 -0.088 0.178 0.380 0.179
-0.201 -1.679 -0.004 -0.231 -0.293 0.368 1.825 0.222
0.175 0.027 -0.003 0.304 0.321 -0.209 0.091 -0.055
```

140 多步后，结果逐渐对不上。

```
0.025 0.360 -0.032 -0.274 -0.022 -0.117 -0.535 -0.266
0.193 0.670 -0.140 -0.166 0.295 0.097 -0.505 -0.389
-0.211 -0.816 0.020 0.002 -0.240 0.210 0.270 0.185
-0.340 -2.096 -0.035 -0.307 -0.594 0.612 2.241 0.726
0.065 0.044 0.033 0.275 0.196 -0.030 0.521 -0.043
epoch:148, W:
-0.034 -2.320 -1.026 -0.792 -0.636 0.279 6.315 0.915
iteration at:148.00,mean loss is:0.05, acc:0.50epoch:149, W:
0.719 4.333 -0.117 1.082 1.173 -0.865 -4.889 0.027
0.830 4.340 -0.371 1.072 1.229 -0.766 -5.109 -0.533
0.283 0.683 -0.210 0.235 0.455 -0.146 -0.591 -0.048
-0.025 -1.217 -0.465 -0.310 -0.127 0.318 1.485 0.311
-0.826 -0.452 0.509 -0.599 -0.818 0.474 0.775 -0.199
0.030 0.373 -0.040 -0.292 -0.017 -0.115 -0.551 -0.252
0.198 0.687 -0.145 -0.183 0.302 0.100 -0.515 -0.378
-0.216 -0.838 0.025 0.016 -0.248 0.208 0.278 0.177
-0.342 -2.105 -0.048 -0.296 -0.608 0.617 2.252 0.754
0.060 0.057 0.028 0.277 0.190 -0.024 0.481 -0.034
```

```
0.025 0.360 -0.032 -0.273 -0.022 -0.117 -0.535 -0.266
0.193 0.670 -0.140 -0.166 0.295 0.097 -0.505 -0.389
-0.211 -0.816 0.021 0.002 -0.241 0.211 0.270 0.185
-0.340 -2.096 -0.035 -0.307 -0.594 0.612 2.240 0.726
0.065 0.044 0.033 0.275 0.196 -0.030 0.522 -0.043
epoch:148, W:
-0.034 -2.321 -1.026 -0.792 -0.636 0.279 6.315 0.915
iteration at:148.00,mean loss is:0.05, acc:0.50epoch:149, W:
0.719 4.333 -0.117 1.082 1.172 -0.865 -4.888 0.027
0.830 4.340 -0.371 1.072 1.229 -0.766 -5.109 -0.533
0.283 0.683 -0.210 0.235 0.455 -0.146 -0.591 -0.048
-0.025 -1.217 -0.465 -0.310 -0.127 0.318 1.485 0.311
-0.826 -0.452 0.509 -0.599 -0.818 0.474 0.775 -0.199
0.030 0.372 -0.040 -0.292 -0.017 -0.115 -0.551 -0.252
0.199 0.687 -0.145 -0.183 0.302 0.100 -0.515 -0.378
-0.216 -0.838 0.025 0.016 -0.248 0.208 0.278 0.177
-0.342 -2.105 -0.048 -0.296 -0.608 0.618 2.252 0.754
0.060 0.057 0.028 0.277 0.190 -0.024 0.481 -0.034
```

随着迭代步数的增加，计算结果逐渐对不上，这是因为该算例中，ANNbox 中的 get_variable 初始化值是从 tensorflow 版本中的 get_variable 值截断产生的，由于截断误差较小，最开始的计算误差不足以体现，但随着迭代次数的变多，误差积累放大，导致后面的权重系数及偏执系数存在明显的差异。不过并不影响模型的收敛，因为二进制加法运算比较简单，最终准确率都是 100%。

详细的代码如下：

ANNbox 版本：

```
# -*- coding: utf-8 -*-
import sys
sys.path.append('./../ANNbox')
import ANNbox as tf
#import tensorflow as tf
```

```

import numpy as np

fp=open('annbox_febbug.txt',"w")

def binary2int(binary):
    out=0
    for index,num in enumerate(binary[::-1]):
        out += num*pow(2,index)
    return out

def binary2int2(binary):
    out=0
    for index,num in enumerate(binary):
        out += num*pow(2,index)
    return out

'''RNN 参数设置，计算 8 位二进制加法，因此序列长度为 8，每次向 place_holder 中喂入 batch_size 大小的
数据，包括两个相加的二进制序列和一个结果二进制序列,hidden_size 表示 RNN 的隐层节点维数'''
sequence_length=8
batch_size=100
input_size=2
hidden_size=8
output_size= 1
epochs=500
seed_=1

n_classes=output_size
state_size=hidden_size
##RNN 的初始化状态，全设为零。注意 state 是与 input 保持一致，接下来会有 concat 操作，所以这里要有 batch
的维度。即每个样本都要有隐层状态
#init_state=tf.zeros([batch_size,state_size],dtype=tf.float32)

#定义 rnn_cell 的权重参数，
with tf.variable_scope('rnn_cell'):

W=tf.get_variable('W',[input_size+state_size,state_size],initializer=tf.truncated_normal_i
nitializer(mean=0.,stddev=0.1))
    b=tf.get_variable('b',[state_size],initializer=tf.constant_initializer(0.0))

#使之定义为 reuse 模式，循环使用，保持参数相同
rnn_cell_ind=0
def rnn_cell(rnn_input,state):
    with tf.variable_scope('rnn_cell',reuse=True):

W=tf.get_variable('W',[input_size+state_size,state_size],initializer=tf.truncated_normal_i
nitializer(mean=0.,stddev=0.1))
    b=tf.get_variable('b',[state_size],initializer=tf.constant_initializer(0.0))

```

```

# 定义 rnn_cell 具体的操作, 这里使用的是最简单的 rnn, 不是 LSTM
global rnn_cell_ind
rnn_cell_ind+=1
cnt=tf.concat(1,(rnn_input,state),name='cnt'+str(rnn_cell_ind)+'_in_rnn_cell')
mtl=tf.matmul(cnt,W,name='mtl'+str(rnn_cell_ind)+'_in_rnn_cell')
add=tf.Add(mtl,b,name='add'+str(rnn_cell_ind)+'_in_rnn_cell')
return tf.nn.tanh(add,name='tanh'+str(rnn_cell_ind))
# return tf.tanh(tf.matmul(tf.concat((rnn_input,state),1),W)+b)

'''推断过程, 即是 RNN 模型的构建过程'''
def inference(x_input):
    '''将输入转换到 tensorflow 中 RNN 需要的格式'''
    rnn_inputs=tf.unstack(input_holder,axis=1,name='x_inputs')
    '''构建 rnn 单元和 rnn 网络'''
    '''方法 01, 简单两句搞定, 关键在与返回值上, 输出看了下, states 只有两个值, 应该一个是初始状态, 一个是末状态, 而其他的中间状态其实涵盖在 outputs 里了, 需要直接去取就好, 不知道 LSTM 的输出是怎样, states 会不会是 8 个, 得去试一下'''
    #rnn_cell=tf.nn.rnn_cell.BasicRNNCell(hidden_size)
    #rnn_outputs,final_state=tf.nn.rnn(rnn_cell,rnn_inputs,dtype=tf.float32)
    '''方法 02'''
    #RNN 的初始化状态, 全设为零。注意 state 是与 input 保持一致, 接下来会有 concat 操作, 所以这里要有 batch 的维度。即每个样本都要有隐层状态
    init_state=tf.zeros([batch_size,state_size],name='init_state')
    state=init_state
    rnn_outputs=[]
    #循环 num_steps 次, 即将一个序列输入 RNN 模型
    for rnn_input in rnn_inputs:
        state=rnn_cell(rnn_input,state)
        rnn_outputs.append(state)

    '''定义输出层权值和偏置, 用 variable_scope 和 get_variable 是为了变量的不重复, 否则测试和训练会构建重复的 variable 出来, Tensorboard 里面网络的 graph 会变的比较乱'''
    with tf.variable_scope('hidden_to_output_layer'):
        w2out=tf.get_variable('w2out',shape=[hidden_size,output_size],initializer=tf.truncated_normal_initializer(mean=0.,stddev=0.1))
        bias2out=tf.get_variable('bias2out',shape=[output_size],initializer=tf.constant_initializer(0.0))

        logits=[tf.Add(tf.matmul(rnn_output,w2out,name='matmul_'+str(ind)),bias2out,name='Add'+str(ind)) for ind,rnn_output in enumerate(rnn_outputs)]
        y_pred=tf.concat(1,[tf.nn.sigmoid(logit,name='sigmoid_'+str(ind)) for ind,logit in enumerate(logits)],name='concat_result')
        return y_pred

def generateBinaryDict():

```

```

largest_number=pow(2,8)
int2binary={}

binary=np.unpackbits(np.array([range(largest_number)],dtype=np.uint8).T,axis=1).astype('float32')
    for i in range(largest_number):
        int2binary[i]=binary[i]
    return int2binary

input_holder=tf.placeholder(tf.float32,[batch_size,8,2],name='input_placeholder')
label_holder=tf.placeholder(tf.float32,[batch_size,8],name='label_placeholder')
y_pred=inference(input_holder)

sub=tf.nn.Sub(y_pred,label_holder,name='sub')
loss=tf.reduce_mean(tf.square(sub,name='square'),name='reduce_mena')
#op=tf.train.AdagradOptimizer(1.0).minimize(loss)
learning_rate=1e0
Momentum_rate=0.95
op=tf.train.MomentumOptimizer(learning_rate,Momentum_rate).minimize(loss)
init=tf.global_variables_initializer()
np.random.seed(seed_)
#with tf.Session() as sess:
init.run()
binary_dict=generateBinaryDict()
for e in range(epochs):
    batch_in_data=[]
    batch_out_data=[]
    reshape_bi_list=[]
    '''生成 minibatch data, 其实没必要, 但是为了规范'''
    for i in range(batch_size):
        x_int1=np.random.randint(0,128)
        x_int2=np.random.randint(0,128)
        y_true_int=x_int1+x_int2

        x_bi1=binary_dict[x_int1]
        x_bi2=binary_dict[x_int2]
        y_true_bi=binary_dict[y_true_int]
        '''注意输入和输出都需要反向, 因为建的 RNN 是从前往后传信息的, 而二进制加法的进位规则是从后往前计算并进位的'''
        for j in range(8):
            reshape_bi_list.append(x_bi1[7-j])
            reshape_bi_list.append(x_bi2[7-j])
        batch_out_data.append(y_true_bi[::-1])

    batch_in_data=np.reshape(reshape_bi_list,[-1,8,2])
    batch_out_data=np.reshape(batch_out_data,[-1,8])

```

```

op.run(feed_dict={input_holder: batch_in_data,label_holder: batch_out_data})
loss_value=loss.my_eval(feed_dict={input_holder: batch_in_data,label_holder:
batch_out_data})
y_pred_value=y_pred.my_eval(feed_dict={input_holder: batch_in_data,label_holder:
batch_out_data})

# print('iteration at: % d,mean loss is: %f' % (e,loss_value))
x_left=np.array([binary2int2(1) for l in batch_in_data[:, :,0]])
x_right=np.array([binary2int2(1) for l in batch_in_data[:, :,1]])
pre=np.array([binary2int2(1) for l in np.round(y_pred_value)])
# print('acc:',(((x_left+x_right)==pre).astype('int')).mean())
print('iteration at: % d,mean loss is: %f acc:%f' %
(e,loss_value,(((x_left+x_right)==pre).astype('int')).mean()))

tem=W.value
fp.write('epoch:{}, W:\n'.format(e))
for ind in range(tem.shape[0]):
    for jnd in range(tem.shape[1]):
        fp.write('{:3.3f} '.format(tem[ind,jnd]))
    fp.write('\n')
tem=b.value
fp.write('epoch:{}, W:\n'.format(e))
for ind in range(len(tem)):
    fp.write('{:3.3f} '.format(tem[ind]))
fp.write('\n')
fp.write('iteration at:{:3.2f},mean loss is:{:3.2f},
acc:{:3.2f}'.format(e,loss_value,(((x_left+x_right)==pre).astype('int')).mean()))

fp.close()

```

tensorflow 版本:

```

# -*- coding:utf-8 -*-
#import sys
#sys.path.append('./../ANNbox')
#import ANNbox as tf
import tensorflow as tf
import numpy as np

fp=open('tensorflow_febbug.txt',"w")

def binary2int(binary):
    out=0
    for index,num in enumerate(binary[::-1]):
        out += num*pow(2,index)
    return out

```

```

def binary2int2(binary):
    out=0
    for index,num in enumerate(binary):
        out += num*pow(2,index)
    return out

'''RNN 参数设置，计算 8 位二进制加法，因此序列长度为 8，每次向 place_holder 中喂入 batch_size 大小的
数据，包括两个相加的二进制序列和一个结果二进制序列,hidden_size 表示 RNN 的隐层节点维数'''
sequence_length=8
batch_size=100
input_size=2
hidden_size=8
output_size= 1
epochs=500
seed_=1

n_classes=output_size
state_size=hidden_size
##RNN 的初始化状态，全设为零。注意 state 是与 input 保持一致，接下来会有 concat 操作，所以这里要有 batch
的维度。即每个样本都要有隐层状态
#init_state=tf.zeros([batch_size,state_size],dtype=tf.float32)

#定义 rnn_cell 的权重参数，
with tf.variable_scope('rnn_cell'):
    W=tf.get_variable('W',[input_size
+state_size,state_size],initializer=tf.truncated_normal_initializer(mean=0.,stddev=0.1,seed=seed_))
    b=tf.get_variable('b',[state_size],initializer=tf.constant_initializer(0.0))

#使之定义为 reuse 模式，循环使用，保持参数相同
rnn_cell_ind=0
def rnn_cell(rnn_input,state):
    with tf.variable_scope('rnn_cell',reuse=True):

W=tf.get_variable('W',[input_size+state_size,state_size],initializer=tf.truncated_normal_initializer(mean=0.,stddev=0.1,seed=seed_))
    b=tf.get_variable('b',[state_size],initializer=tf.constant_initializer(0.0))
    # 定义 rnn_cell 具体的操作，这里使用的是最简单的 rnn，不是 LSTM
    global rnn_cell_ind
    rnn_cell_ind+=1
    cnt=tf.concat(1,(rnn_input,state))
    mtl=tf.matmul(cnt,W)+b

    return tf.nn.tanh(mtl)
#    return tf.tanh(tf.matmul(tf.concat((rnn_input,state),1),W)+b)

```

'''推断过程，即是 RNN 模型的构建过程'''

```
def inference(x_input):
```

'''将输入转换到 tensorflow 中 RNN 需要的格式'''

```
rnn_inputs=tf.unstack(x_input,axis=1)
```

'''构建 rnn 单元和 rnn 网络'''

'''方法 01，简单两句搞定，关键在与返回值上，输出看了下，states 只有两个值，应该一个是初始状态，一个是末状态，而其他的中间状态其实涵盖在 outputs 里了，需要直接去取就好，不知道 LSTM 的输出是怎样，states 会不会是 8 个，得去试一下'''

```
# rnn_cell=tf.nn.rnn_cell.BasicRNNCell(hidden_size)
```

```
# rnn_outputs,final_state=tf.nn.rnn(rnn_cell,rnn_inputs,dtype=tf.float32)
```

'''方法 02'''

#RNN 的初始化状态，全设为零。注意 state 是与 input 保持一致，接下来会有 concat 操作，所以这里要有 batch 的维度。即每个样本都要有隐层状态

```
init_state=tf.zeros([batch_size,state_size])
```

```
state=init_state
```

```
rnn_outputs=[]
```

#循环 num_steps 次，即将一个序列输入 RNN 模型

```
for rnn_input in rnn_inputs:
```

```
    state=rnn_cell(rnn_input,state)
```

```
    rnn_outputs.append(state)
```

'''定义输出层权值和偏置，用 variable_scope 和 get_variable 是为了变量的不重复，否则测试和训练会构建重复的 variable 出来，Tensorboard 里面网络的 graph 会变的比较乱'''

```
with tf.variable_scope('hidden_to_output_layer'):
```

```
w2out=tf.get_variable('w2out',shape=[hidden_size,output_size],initializer=tf.truncated_normal_initializer(mean=0.,stddev=0.1,seed=seed_))
```

```
bias2out=tf.get_variable('bias2oput',shape=[output_size],initializer=tf.constant_initializer(0.0))
```

```
logits=[tf.matmul(rnn_output,w2out)+bias2out for ind,rnn_output in enumerate(rnn_outputs)]
```

```
y_pred=tf.concat(1,[tf.nn.sigmoid(logit) for ind,logit in enumerate(logits)])
```

```
return y_pred
```

```
def generateBinaryDict():
```

```
    largest_number=pow(2,8)
```

```
    int2binary={}
```

```
    binary=np.unpackbits(np.array([range(largest_number)],dtype=np.uint8).T,axis=1)
```

```
    for i in range(largest_number):
```

```
        int2binary[i]=binary[i]
```

```
    return int2binary
```

```
input_holder=tf.placeholder(tf.float32,[None,8,2])
```

```
label_holder=tf.placeholder(tf.float32,[None,8])
```

```
y_pred=inference(input_holder)
```



```

loss=tf.reduce_mean(tf.square(y_pred-label_holder))
#op=tf.train.AdagradOptimizer(1.0).minimize(loss)
learning_rate=1e0
Momentum_rate=0.95
op=tf.train.MomentumOptimizer(learning_rate,Momentum_rate).minimize(loss)
init=tf.initialize_all_variables()
np.random.seed(seed_)
with tf.Session() as sess:
    sess.run(init)
    binary_dict=generateBinaryDict()
    for e in range(epochs):
        batch_in_data=[]
        batch_out_data=[]
        reshape_bi_list=[]
        '''生成 minibatch data, 其实没必要, 但是为了规范'''
        for i in range(batch_size):
            x_int1=np.random.randint(0,128)
            x_int2=np.random.randint(0,128)
            y_true_int=x_int1+x_int2

            x_bi1=binary_dict[x_int1]
            x_bi2=binary_dict[x_int2]
            y_true_bi=binary_dict[y_true_int]
            '''注意输入和输出都需要反向, 因为建的 RNN 是从往后传信息的, 而二进制加法的进位规则是从
            后往前计算并进位的'''
            for j in range(8):
                reshape_bi_list.append(x_bi1[7-j])
                reshape_bi_list.append(x_bi2[7-j])
                batch_out_data.append(y_true_bi[::-1])

        batch_in_data=np.reshape(reshape_bi_list,[-1,8,2])
        batch_out_data=np.reshape(batch_out_data,[-1,8])

        sess.run(op,feed_dict={input_holder:batch_in_data,label_holder:batch_out_data})

loss_value=sess.run(loss,feed_dict={input_holder:batch_in_data,label_holder:batch_out_data
})

y_pred_value=sess.run(y_pred,feed_dict={input_holder:batch_in_data,label_holder:batch_out_
data})
    x_left=np.array([binary2int2(1) for l in batch_in_data[:, :, 0]])
    x_right=np.array([binary2int2(1) for l in batch_in_data[:, :, 1]])
    pre=np.array([binary2int2(1) for l in np.round(y_pred_value)])
    print('iteration at:% d,mean loss is:%f acc:%f' %
(e,loss_value,(((x_left+x_right)==pre).astype('int')).mean()))

    tem=sess.run(W)

```



```
fp.write('epoch:{}, W:\n'.format(e))
for ind in range(tem.shape[0]):
    for jnd in range(tem.shape[1]):
        fp.write('{:3.3f} '.format(tem[ind,jnd]))
    fp.write('\n')
tem=sess.run(b)
fp.write('epoch:{}, W:\n'.format(e))
for ind in range(len(tem)):
    fp.write('{:3.3f} '.format(tem[ind]))
fp.write('\n')
fp.write('iteration at:{:3.2f},mean loss is:{:3.2f},
acc:{:3.2f}'.format(e,loss_value,(((x_left+x_right)==pre).astype('int')).mean()))
fp.close()
```