**MXNet学习**

**NDArray多维阵列**，如(batch, channel, height, width)，类似于blobs

示例：

a=mx.nd.array([[1,2,3],[4,5,6]])

a.size

a.asnumpy()

b=a\*a #element-wise

c=mx.nd.dot(a,b) #矩阵乘积

#产生[0,1]之间的10\*10的均匀分布的数，存在gpu#0中

c=mx.nd.uniform(low=0,high=1,shape=(10,10),ctx=”gpu(0)”)

#产生均值为1，标准差为2的正态分布，存在cpu中

d=mx.nd.normal(loc=1,scale=2,shape=(10,10),ctx=”cpu()”)

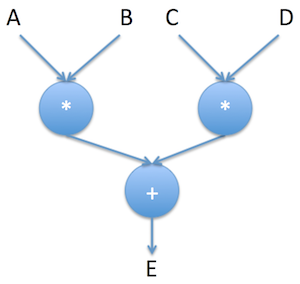
e=mx.nd.doc(c,d)

**symbol**

然而神经网络从本质上来说是一种并行的结构：在特定的技术层中，所有输出都可同步计算。每个层也可以并行运行。因此为了获得最佳性能，我们必须使用多线程或其他类似机制自行实现并行处理。具体做法大概都知道，而就算写出了恰当的代码，如果数据规模或网络布局不断变化，如何确保能可复用地进行？

**数据流编程（Dataflow programming）**是一种定义并行运算的灵活方法，这种方法中，数据可通过图（Graph）的方式流动。Graph定义了运算顺序，即数据是要按顺序运算或并行运算。每个运算都是一种黑匣子：我们只需要为其定义输入和输出，无需制定具体的行为。

如要计算E=A\*B+C\*D,如图所示：



代码示例：

>>> import mxnet as mx  
>>> a = mx.symbol.Variable('A')  
>>> b = mx.symbol.Variable('B')  
>>> c = mx.symbol.Variable('C')  
>>> d = mx.symbol.Variable('D')  
>>> e = (a\*b)+(c\*d)

(a,b,c,d)是符号，e是+运算的结果，也是符号，如：

>>> e.list\_arguments()  
['A', 'B', 'C', 'D']  
>>> e.list\_outputs()  
['\_plus1\_output']  
>>> e.get\_internals().list\_outputs()  
['A', 'B', '\_mul0\_output', 'C', 'D', '\_mul1\_output', '\_plus1\_output']

**binding(绑定)**

将符号和数据流绑定

>>> a\_data = mx.nd.array([1], dtype=np.int32)  
>>> b\_data = mx.nd.array([2], dtype=np.int32)  
>>> c\_data = mx.nd.array([3], dtype=np.int32)  
>>> d\_data = mx.nd.array([4], dtype=np.int32)

>>> executor=e.bind(mx.cpu(), {'A':a\_data, 'B':b\_data, 'C':c\_data, 'D':d\_data})

通过forward()函数实现数据流的计算

e\_data=executor.forward()

**remark**：(a,b,c,d)可以是一个数，也可以是1000\*1000的矩阵。数据和计算之间明确地区分

**Module**

定义网络：

net = mx.sym.Variable('data')

net = mx.sym.FullyConnected(net, name='fc1', num\_hidden=64)

net = mx.sym.Activation(net, name='relu1', act\_type="relu")

net = mx.sym.FullyConnected(net, name='fc2', num\_hidden=26)

net = mx.sym.SoftmaxOutput(net, name='softmax')

Module类4个常用参数：symbol，context，data\_names，label\_names

mod = mx.mod.Module(symbol=net,

context=mx.cpu(),

data\_names=['data'],

label\_names=['softmax\_label'])

训练一个模型：

**b**ind：通过内存分配为计算搭建环境

* init\_params ：初始化参数
* init\_optimizer：初始化优化器，默认是sgd
* metric.create：从输入指标名称创建评估指标
* forward ：前向计算
* update\_metric：对最后一次前向计算的输出进行评估和累加评估度量
* backward ：后向计算
* update ：根据安装的优化程序和先前的前向后向批次中计算的梯度更新参数

例子：

# allocate memory given the input data and label shapesmod.bind(data\_shapes=train\_iter.provide\_data, label\_shapes=train\_iter.provide\_label)

# initialize parameters by uniform random numbersmod.init\_params(initializer=mx.init.Uniform(scale=.1))

# use SGD with learning rate 0.1 to trainmod.init\_optimizer(optimizer='sgd', optimizer\_params=(('learning\_rate', 0.1), ))

# use accuracy as the metric

metric = mx.metric.create('acc')

# train 5 epochs, i.e. going over the data iter one pass

for epoch in range(5):

train\_iter.reset()

metric.reset()

for batch in train\_iter:

mod.forward(batch, is\_train=True) # compute predictions

mod.update\_metric(metric, batch.label) # accumulate prediction accuracy

mod.backward() # compute gradients

mod.update() # update parameters

print('Epoch %d, Training %s' % (epoch, metric.get()))

或者用fit函数：

mod.fit(train\_iter,

eval\_data=val\_iter,

optimizer='sgd',

optimizer\_params={'learning\_rate':0.1},

eval\_metric='acc',

num\_epoch=8)

用load\_checkpoint来载入保存好的参数，并设置模型：

sym, arg\_params, aux\_params = mx.model.load\_checkpoint(model\_prefix, 0)

mod = mx.mod.Module(symbol=sym)

mod.bind(for\_training=False, data\_shapes=[('data', (1,3,224,224))])

mod.set\_params(arg\_params, aux\_params)

mod.forward(Batch([array]))

prob = mod.get\_outputs()[0].asnumpy()