Deep learning paper

MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems

参考：https://github.com/dmlc/mxnet/issues/797
http://blog.csdn.net/daslab/article/details/50434145#深度学习编程模型

MXNet is a machine learning library combining symbolic expression with tensor computation to maximize efficiency and flexibility. It is lightweight and embeds in multiple host languages, and can be run in a distributed setting.

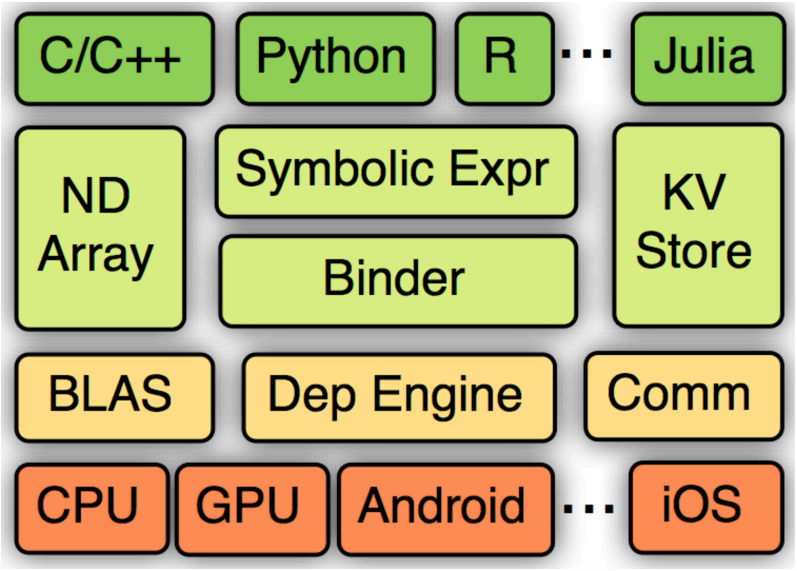
	命令式编程 Imperative Program	声明式编程 Declarative Program
Excute a = b + 1	立即执行加法，将结果保存在a中，a的类型与b相同	返回对应的计算图(computation graph)，我们可以之后对b进行赋值，然后再执行加法运算
优点	语义上容易理解灵活，可以精确控制行为。通常可以无缝的和主语言交互，方便的利用主语言的各类算法，工具包，bug及性能调试器	在真正开始计算的时候已经拿到了整个计算图，所以我们可以做一系列优化来提升性能。实现辅助函数也容易，例如对任何计算图都提供forward和backward函数，对计算图进行可视化，将图保存到硬盘和从硬盘读取。
缺点	实现统一的辅助函数困和提供整体优化都很困难	很多主语言的特性都用不上，某些在主语言中实现简单，但在这里却经常麻烦，例如 if-else 语句，debug 也不容易，例如监视一个复杂的计算图中的某个节点的中间结果并不简单.

MXNet 尝试将两种模式无缝的结合起来。在命令式编程上 MXNet 提供张量运算，而声明式编程中 MXNet 支持符号表达式。用户可以自由的混合它们来快速实现自己的想法。例如我们可以用声明式编程来描述神经网络，并利用系统提供的自动求导来训练模型。另一方便，模型的迭代训练和更新模型法则中可能涉及大量的控制逻辑，因此我们可以用命令式编程来实现。同时我们用它来进行方便的调式和与主语言交互数据。

MXNet 和其他流行的深度学习系统比较

	主语言	从语言	硬件	分布式	命令式	声明式
Caffe	C++	Python/Matlab	CPU/GPU	x	x	v
Torch	Lua	-	CPU/GPU/FPGA	x	v	x
Theano	Python	-	CPU/GPU	x	x	v
TensorFlow	C++	Python	CPU/GPU/Mobile	v	x	v
MXNet	C++	Python/R/Julia/Go	CPU/GPU/Mobile	v	v	v

MXNet 的系统架构如下图所示：



从上到下分别为各种主语言的嵌入，编程接口（矩阵运算，符号表达式，分布式通讯），两种编程模式的统一系统实现，以及各硬件的支持。

Symbol： 声明式的符号表达式

MXNet 使用多值输出的符号表达式来声明计算图。符号是由操作子构建而来。一个操作子可以是一个简单的矩阵运算“+”，也可以是一个复杂的神经网络里面的层，例如卷积层。一个操作子可以有多个输入变量和多个输出变量，还可以有内部状态变量。一个变量既可以是自由的，我们可以之后对其赋值；也可以是某个操作子的输出。例如下面的代码中我们使用 Julia 来定义一个多层感知机，它由一个代表输入数据的自由变量，和多个神经网络层串联而成。

```
using MXNet
mlp = @mx.chain mx.Variable(:data) =>
  mx.FullyConnected(num_hidden=64) =>
  mx.Activation(act_type=:relu)    =>
  mx.FullyConnected(num_hidden=10) =>
  mx.Softmax()
```

在执行一个符号表达式前，我们需要对所有的自由变量进行赋值。上例中，我们需要给定数据，和各个层里隐式定义的输入，例如全连接层的权重和偏值。我们同时要申明所需要的输出，例如 softmax 的输出。除了执行获得 softmax 输出外（通常也叫 forward），符号表达式也支持自动求导来获取各权重和偏值对应的梯度（也称之为 backward）。此外，我们还可以提前估计计算时需要的内存，符号表达式的可视化，读入和输出等。

NDArray：命令式的张量计算

MXNet 提供命令式的张量计算来桥接主语言的和符号表达式。下面代码中，我们在 GPU 上计算矩阵和常量的乘法，并使用 numpy 来打印结果

```
>>> import MXNet as mx
>>> a = mx.nd.ones((2, 3),
... mx.gpu())
>>> print (a * 2).asnumpy()
[[ 2.  2.  2.]
 [ 2.  2.  2.]]
```

另一方面，NDArray 可以无缝和符号表达式进行对接。假设我们使用 Symbol 定义了一个神经网络，那么我们可以如下实现一个梯度下降算法

```
for (int i = 0; i < max_iter; ++i) {
  network.forward();
  network.backward();
  network.weight -= eta * network.gradient
}
```

这里梯度由 Symbol 计算而得。Symbol 的输出结果均表示成 NDArray，我们可以通过 NDArray 提供的张量计算来更新权重。此外，我们还利用了主语言的 for 循环来进行迭代，学习率 eta 也是在主语言中进行修改。

上面的混合实现跟使用纯符号表达式实现的性能相差无二，然后后者在表达控制逻辑时会更加复杂。其原因是 NDArray 的执行会和 Symbol 类似的构建一个计算图，并与其他运算一同交由后台引擎执行。对于运算-=由于我们只是将其结果交给另一个 Symbol 的 forward 作为输入，因此我们不需要立即得到结果。当上面的 for 循环结束时，我们只是将数个 Symbol 和 NDarray 对应的计算图提交给了后台引擎。当我们最终需要结果的时候，例如将 weight 复制到主语言中或者保存到磁盘时，程序才会被阻塞直到所有计算完成。

KVStore：多设备间的数据交互

MXNet 提供一个分布式的 key-value 存储来进行数据交换。它主要有两个函数，

1. push： 将 key-value 对从一个设备 push 进存储

2. pull：将某个 key 上的值从存储中 pull 出来此外，KVStore 还接受自定义的更新函数来控制收到的值如何写入到存储中。最后 KVStore 提供数种包含最终一致性模型和顺序一致性模型在内的数据一致性模型。在下面例子中，我们将前面的梯度下降算法改成分布式梯度下降。

```
KVStore kvstore("dist_async");
kvstore.set_updater([](NDArray weight, NDArray gradient) {
    weight -= eta * gradient;
  });
for (int i = 0; i < max_iter; ++i) {
   kvstore.pull(network.weight);
   network.forward();
   network.backward();
   kvstore.push(network.gradient);
}
```

在这里我们先使用最终一致性模型创建一个 kvstore，然后将更新函数注册进去。在每轮迭代前，每个计算节点先将最新的权重 pull 回来，之后将计算的得到的梯度 push 出去。kvstore 将会利用更新函数来使用收到的梯度更新其所存储的权重。这里 push 和 pull 跟 NDArray 一样使用了延后计算的技术。它们只是将对应的操作提交给后台引擎，而引擎则调度实际的数据交互。所以上述的实现跟我们使用纯符号实现的性能相差无几。

读入数据模块

数据读取在整体系统性能上占重要地位。MXNet 提供工具能将任意大小的样本压缩打包成单个或者数个文件来加速顺序和随机读取。通常数据存在本地磁盘或者远端的分布式文件系统上（例如 HDFS 或者 Amazon S3），每次我们只需要将当前需要的数据读进内存。MXNet 提供迭代器可以按块读取不同格式的文件。迭代器使用多线程来解码数据，并使用多线程预读取来隐藏文件读取的开销。

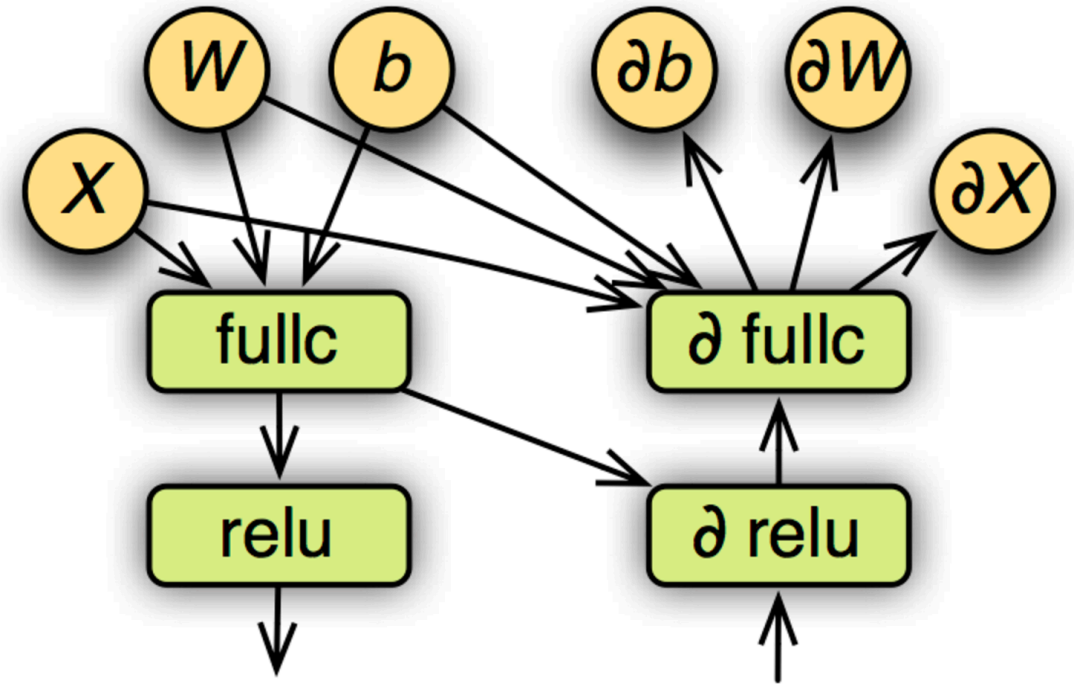
训练模块

MXNet 实现了常用的优化算法来训练模型。用户只需要提供数据数据迭代器和神经网络的 Symbol 便可。此外，用户可以提供额外的 KVStore 来进行分布式的训练。例如下面代码使用分布式异步 SGD 来训练一个模型，其中每个计算节点使用两快 GPU。

```
import MXNet as mx
model = mx.model.FeedForward(
    ctx                = [mx.gpu(0), mx.gpu(1)],
    symbol             = network,
    num_epoch          = 100,
    learning_rate      = 0.01,
    momentum           = 0.9,
    wd                 = 0.00001,
    initializer        = mx.init.Xavier(factor_type="in", magnitude=2.34))
model.fit(
    X                  = train_iter,
    eval_data          = val_iter,
    kvstore            = mx.kvstore.create('dist_async'),
    epoch_end_callback = mx.callback.do_checkpoint('model_'))
```

计算图

一个已经赋值的符号表达式可以表示成一个计算图。下图是之前定义的多层感知机的部分计算图，包含 forward 和 backward。



其中圆表示变量，方框表示操作子，箭头表示数据依赖关系。在执行之前，MXNet 会对计算图进行优化，以及为所有变量提前申请空间。

计算图优化

计算图优化已经在数据库等领域被研究多年，我们目前只探索了数个简单的方法。
1. 注意到我们提前申明了哪些输出变量是需要的，这样我们只需要计算这些输出需要的操作。例如，在预测时我们不需要计算梯度，所以整个 backforward 图都可以忽略。而在特征抽取中，我们可能只需要某些中间层的输出，从而可以忽略掉后面的计算。
2. 可以合并某些操作，例如 *a* * b+1 只需要一个 blas 或者 cuda 函数即可，而不需要将其表示成两个操作。
3. 实现了一些“大”操作，例如一个卷积层就只需要一个操作子。这样我们可以大大减小计算图的大小，并且方便手动的对这个操作进行优化。

内存申请

内存通常是一个重要的瓶颈，尤其是对 GPU 和智能设备而言。而神经网络计算时通常需要大量的临时空间，例如每个层的输入和输出变量。对每个变量都申请一段独立的空间会带来高额的内存开销。幸运的是，我们可以从计算图推断出所有变量的生存期，就是这个变量从创建到最后被使用的时间段，从而可以对两个不交叉的变量重复使用同一内存空间。这个问题在诸多领域，例如编译器的寄存器分配上，有过研究。然而最优的分配算法需要 $O(n^2)$ 时

间复杂度，这里 n 是图中变量的个数。
MXNet 提供了两个启发式的策略，每个策略都是线性的复杂度。
1. inplace。在这个策略里，我们模拟图的遍历过程，并为每个变量维护一个还有多少其他变量需要它的计数。当我们发现某个变量的计数变成 0 时，我们便回收其内存空间。
2. co-share。我们允许两个变量使用同一段内存空间。这么做当然会使得这两个变量不能同时在写这段空间。所以我们只考虑对不能并行的变量进行 co-share。每一次我们考虑图中的一条路（path），路上所有变量都有依赖关系所以不能被并行，然后我们对其进行内存分配并将它们从图中删掉。

引擎

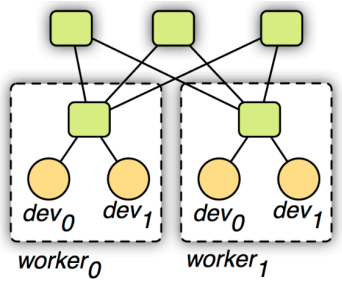
在 MXNet 中，所有的任务，包括张量计算，symbol 执行，数据通讯，都会交由引擎来执行。首先，所有的资源单元，例如 NDArray，随机数生成器，和临时空间，都会在引擎处注册一个唯一的标签。然后每个提交给引擎的任务都会标明它所需要的资源标签。引擎则会跟踪每个资源，如果某个任务所需要的资源到到位了，例如产生这个资源的上一个任务已经完成了，那么引擎会则调度和执行这个任务。

通常一个 MXNet 运行实例会使用多个硬件资源，包括 CPU，GPU，PCIe 通道，网络，和磁盘，所以引擎会使用多线程来调度，既任何两个没有资源依赖冲突的任务都可能会被并行执行，以求最大化资源利用。

与通常的数据流引擎不同的是，MXNet 的引擎允许一个任务修改现有的资源。为了保证调度正确性，提交任务时需要分开标明哪些资源是只读，哪些资源会被修改。这个附加的写依赖可以带来很多便利。例如我们可以方便实现在 numpy 以及其他张量库中常见的数组修改操作，同时也使得内存分配时更加容易，比如操作子可以修改其内部状态变量而不需要每次都重来内存。再次，假如我们要用同一个种子生成两个随机数，那么我们可以标注这两个操作会同时修改种子来使得引擎不会并行执行，从而使得代码的结果可以很好的被重复。

数据通讯

KVStore 的实现是基于参数服务器。但它跟前面的工作有两个显著的区别。
1. 我们通过引擎来管理数据一致性，这使得参数服务器的实现变得相当简单，同时使得 KVStore 的运算可以无缝的与其他结合在一起。
2. 我们使用一个两层的通讯结构，原理如下图所示。第一层的服务器管理单机内部的多个设备之间的通讯。第二层服务器则管理机器之间通过网络的通讯。第一层的服务器在与第二层通讯前可能合并设备之间的数据来降低网络带宽消费。同时考虑到机器内和外通讯带宽和延时的不同性，我们可以对其使用不同的一致性模型。例如第一层我们用强的一致性模型，而第二层我们则使用弱的一致性模型来减少同步开销。



可移植性

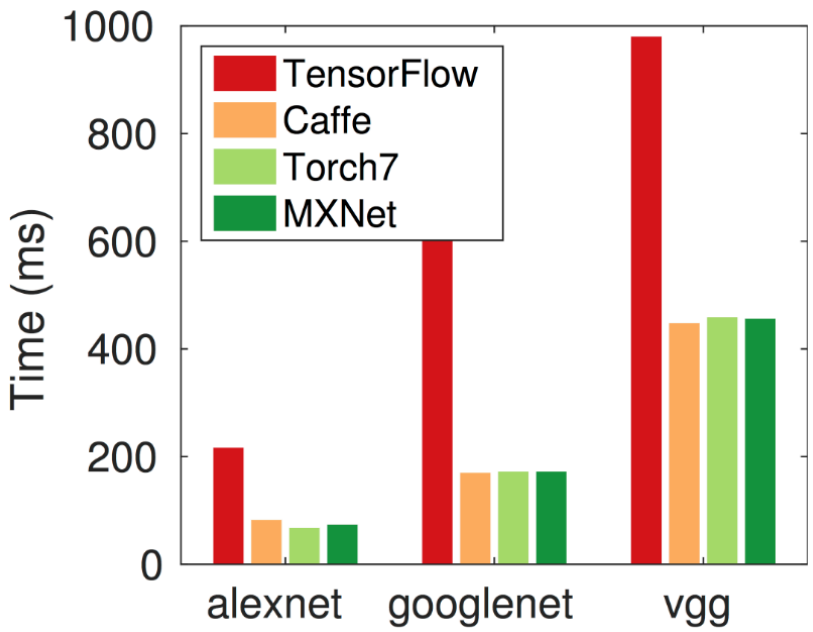
轻量和可移植性是 MXNet 的一个重要目标。MXNet 核心使用 C++实现，并提供 C 风格的头文件。因此方便系统移植，也使得其很容易被其他支持 C FFI (forigen language interface)的语言调用。此外，我们也提供一个脚本将 MXNet 核心功能的代码连同所有依赖打包成一个单一的只有数万行的 C++源文件，使得其在一些受限的平台，例如智能设备，方便编译和使用。

实验结果

这里我们提供一些早期的实验结果。

与其他系统相比

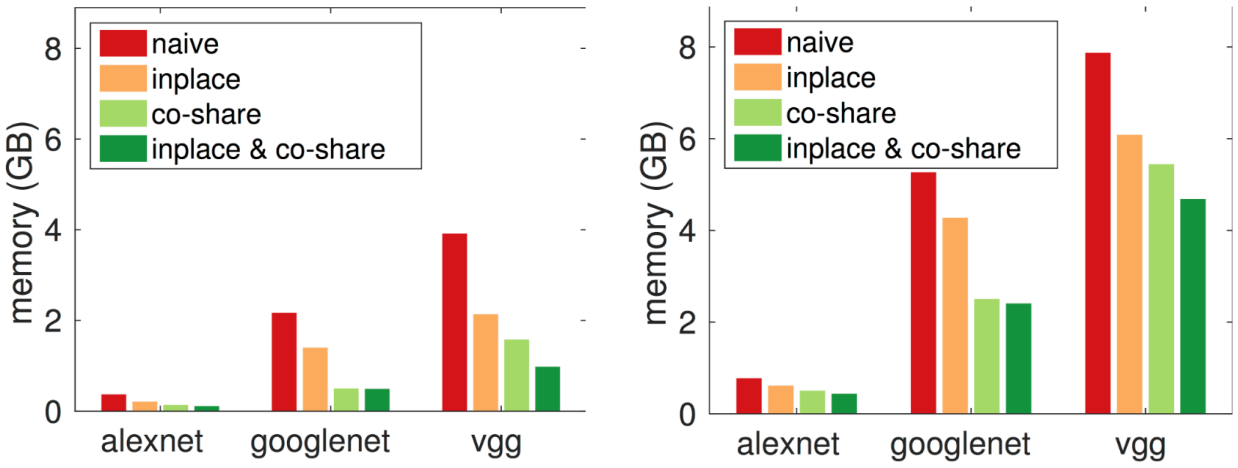
我们首先使用一个流行卷积网络测试方案来对比 MXNet 与 Torch，Caffe 和 TensorFlow 在过去几届 imagenet 竞赛冠军网络上的性能。每个系统使用同样的 CUDA 7.0 和 CUDNN 3，但 TensorFlow 使用其只支持的 CUDA 6.5 和 CUDNN 2。我们使用单块 GTX 980 并报告单个 forward 和 backward 的耗时。



可以看出 MXNet，Torch 和 Caffe 三者在性能上不相上下。这个符合预期，因为在单卡上我们评测的几个网络的绝大部分运算都由 CUDA 和 CUDNN 完成。TensorFlow 比其他三者都慢 2 倍以上，这可能由于是低版本的 CUDNN 和项目刚开源的缘故。

内存的使用

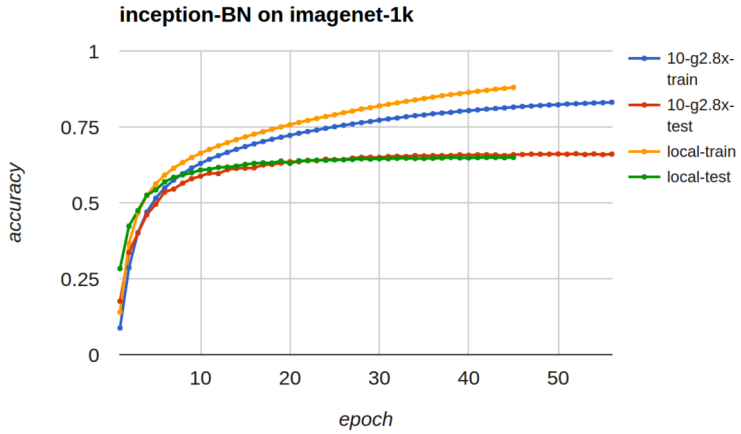
接下来我们考察不同的内存分配算法对内存占用的影响。下图分别表示使用 batch=128 时，在做预测时和做训练时的不同算法在内部变量（除去模型，最初输入和最终输出）上的内存开销。



可以看出，inplace 和 co-share 两者都可以极大的降低内存使用。将两者合起来可以在训练时减少 2 倍内存使用，在预测时则可以减小 4 倍内存使用。特别的，即使是最复杂的 vggnet，对单张图片进行预测时，MXNet 只需要 16MB 额外内存。

Scalability

最后我们报告在分布式训练下的性能。我们使用 imagenet 1k 数据（120 万 224x224x3 图片，1000 类），并用 googlenet 加上 batch normalization 来训练。我们使用 Amazon EC2 g2.8x，单机和多机均使用同样的参数，下图表示了使用单机和 10 台 g2.8x 时的收敛情况。



从训练精度来看，单机的收敛比多机快，这个符合预期，因为多机时有效的 batch 大小比单机要大，在处理同样多的数据上收敛通常会慢。但有意思的是两者在测试精度上非常相似。单机下每遍历一次数据需要 1 万 4 千秒，而在十台机器上，每次只需要 1 千 4 百秒。如果考虑运行时间对比测试精度，10 台机器带来了 10 倍的提升。

MXNet Python 库操作简介

MXNet 的 Python 库有三个基本的概念：

- NDArray：提供矩阵和张量计算，可在 CPU 和 GPU 上进行，并自动化的平行。
- Symbol：使定义神经网络变得非常简单，并提供自动化的微分。
- KVStore：使数据在多 GPU 和多机器之间的同步变得简单。

参考 :http://blog.csdn.net/daslab/article/details/50434145#深度学习编程模型