MXNet Overview

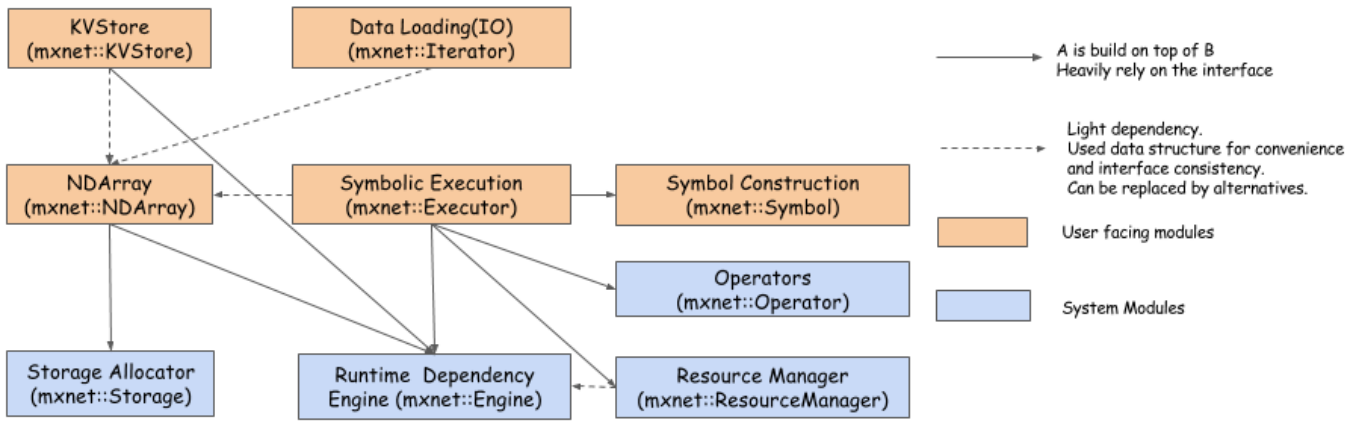
Intro

- MX表示Mix, 混合了符号式编程和指令式编程风格
- 示例

In [ ]:

```
import mxnet as mx

# Symbolic programming
# Create Symbol Graph
num_classes = 10
net = mx.sym.Variable('data')
net = mx.sym.FullyConnected(data=net, name='fc1', num_hidden=128)
net = mx.sym.Activation(data=net, name='relu1', act_type="relu")
net = mx.sym.FullyConnected(data=net, name='fc2', num_hidden=num_classes)
net = mx.sym.SoftmaxOutput(data=net, name='out')
mx.viz.plot_network(net)

# Bind current symbol to get an executor, allocate all the ndarrays needed.
# Allows specifying data types.
num_features = 100
batch_size = 100
ex = net.simple_bind(ctx=mx.cpu(), data=(batch_size, num_features))
args = dict(zip(net.list_arguments(), ex.arg_arrays))
for name in args:
    print(name, args[name].shape)

# Start training
learning_rate = 0.1
final_acc = 0
for i in range(100):
    x, y = toy_data.get(batch_size)
    args['data'][:] = x
    args['out_label'][:] = y
    ex.forward(is_train=True)
    ex.backward()
    # Imperative programming
    for weight, grad in zip(ex.arg_arrays, ex.grad_arrays):
        weight[:] -= learning_rate * (grad / batch_size)
    if i % 10 == 0:
        acc = (mx.nd.argmax_channel(ex.outputs[0]).asnumpy() == y).sum()
        final_acc = acc
        print('iteration %d, accuracy %f' % (i, float(acc)/y.shape[0]))
assert final_acc > 0.95, "Low training accuracy."
```

Architecture



- **Runtime Dependency Engine**: 根据依赖调度和执行的引擎，包括tensor计算、symbol执行、数据通信在内的所有操作都会提交给Engine来调度
- Storage Allocator: 为GPU、CPU分配和回收memory
- Resource Manager: 管理全局资源，包括random generator和临时存储空间
- **Operator**: 定义一个操作的前向计算和梯度计算
- NDArray: 动态异步N维数组, 使用上和numpy.ndarray基本没有区别，但是支持直接分配在GPU上，而且支持异步，自动并行化（依赖Engine）。为MXNet提供灵活的命令式编程。依赖mshadow项目
- **Symbolic Execution**: 静态符号graph的执行，提供了symbolic graph执行
- **Symbol Construction**: Graph的构建和优化，剥离出来nnvm项目
- **KVStore**: 为参数同步提供的Key-value存储接口，剥离出来ps-lite项目
- Data Loading(IO): 数据加载Iterator

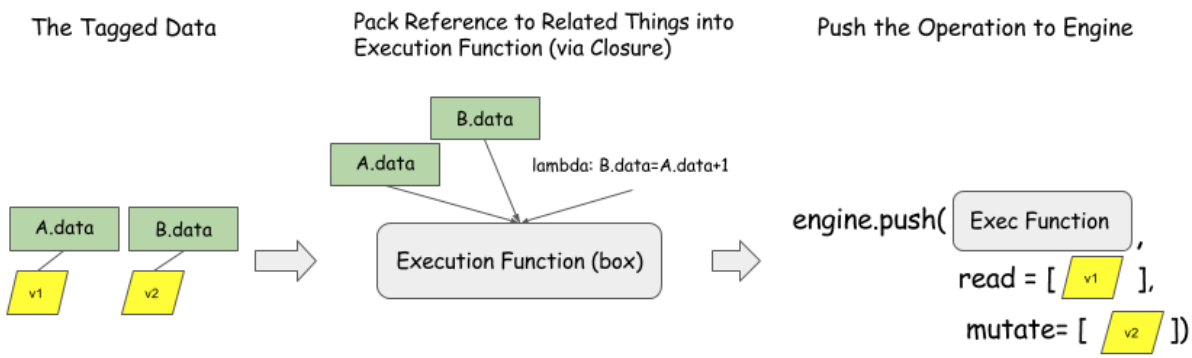
Components

mxnet::Engine

- 目标：在MXNet中，所有的任务，包括tensor计算，symbol执行，数据通讯，都会交由引擎来执行。引擎实现多设备、多线程的依赖调度
- 设计思路：
  - 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**VarHandle。
  - 每个提交给引擎的**任务都需要申明它所需要的资源标签**。依赖包括读依赖和写依赖。
  - 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。
  - 任何两个没有资源依赖冲突的任务都可能会被**并行**执行
  - 设计上通过虚拟的标签和操作Closure保证引擎的**通用性**

```
class MXNET_API Engine {
 public:
  // 申请资源标签
  virtual VarHandle NewVariable() = 0;

  // AsyncFn: 执行的操作
  // const_vars: 只读资源
  // mutable_vars: 修改资源
  virtual void PushAsync(AsyncFn exec_fun, Context exec_ctx,
                         std::vector<VarHandle> const& const_vars,
                         std::vector<VarHandle> const& mutable_vars,
                         FnProperty prop = FnProperty::kNormal,
                         int priority = 0,
                         const char* opr_name = nullptr) = 0;

  void PushSync(SyncFn exec_fn, Context exec_ctx,
                std::vector<VarHandle> const& const_vars,
                std::vector<VarHandle> const& mutable_vars,
                FnProperty prop = FnProperty::kNormal,
                int priority = 0,
                const char* opr_name = nullptr)
  // ...
};
```

- 示例



- 更多设计思参考 http://mxnet.io/architecture/note_engine.html (http://mxnet.io/architecture/note_engine.html)

mxnet::Operator

- Operator包含了实际的计算逻辑，是实现类的接口，类似于tensorflow的OpKernel，所有的操作都是对mshadow::TBlob结构的操作
- mshadow是个轻量的CPU/GPU Matrix/Tensor模版库. 类比Eigen

```
class Operator {
 public:
  virtual void Forward(const OpContext &ctx,
                       const std::vector<TBlob> &in_data,
                       const std::vector<OpReqType> &req,
                       const std::vector<TBlob> &out_data,
                       const std::vector<TBlob> &aux_states);
  virtual void Backward(const OpContext &ctx,
                        const std::vector<TBlob> &out_grad,
                        const std::vector<TBlob> &in_data,
                        const std::vector<TBlob> &out_data,
                        const std::vector<OpReqType> &req,
                        const std::vector<TBlob> &in_grad,
                        const std::vector<TBlob> &aux_states);
  // ...
};
```

- OperatorProperty是Operator的semantic接口，类似于tensorflow的OpDef

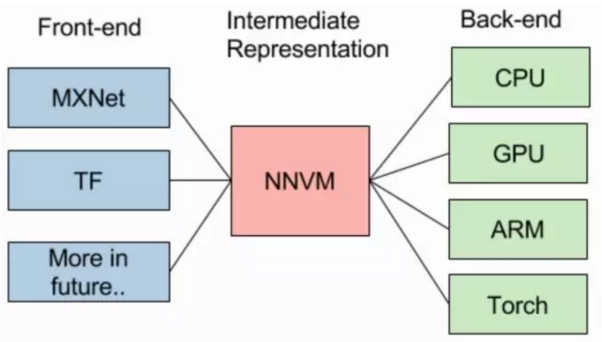
```
class OperatorProperty {
 public:
  virtual void Init(const std::vector<std::pair<std::string, std::string>
 >& kwargs);
  virtual std::map<std::string, std::string> GetParams() const;
  virtual std::vector<std::string> ListArguments() const;
  virtual std::vector<std::string> ListOutputs() const;
  virtual std::vector<std::string> ListAuxiliaryStates() const;
  virtual int NumOutputs() const;
  virtual int NumVisibleOutputs() const;
  virtual bool InferShape(std::vector<TShape> *in_shape,
                          std::vector<TShape> *out_shape,
                          std::vector<TShape> *aux_shape) const;
  virtual bool InferType(std::vector<int> *in_type,
                         std::vector<int> *out_type,
                         std::vector<int> *aux_type) const;
  virtual OperatorProperty* Copy() const;
  virtual Operator* CreateOperator(Context ctx) const;
  virtual Operator* CreateOperatorEx(Context ctx, std::vector<TShape> *in_
shape,
                                     std::vector<int> *in_type) const;

  // ...
};
```

Graph

Graph表示和优化

- 剥离出来一个单独的项目dmlc::nnvm来做Graph计算的中间表示层，提供计算图的优化，包括减少内存开销，设备分配，而不关心operator接口的定义以及op如何执行
  - nnvm中的Operator是通用的接口，并不是以基类的方式存在，而是提供属性注册接口，使用DSL来将前端的Operator映射到nnvm的Operator。
  - 完全独立的Graph\Op\Node定义
  - Pass: Graph的函数，输入一个Graph，产出一个Graph，比如符号求导, memory planning, shape/type推测等.
  - 不包括图的执行

```
// registeration of oeprators
// NOTE that the attr function can register any
// additional attributes to the operator
NNVM_REGISTER_OP(add)
.describe("add two inputs together")
.set_num_inputs(2)
.set_attr<OpKernel>("OpKernel<gpu>", AddKernel)
.include("ElementwiseOpAttr");
```

> “前端把计算表达成一个中间形式，通常我们称之为计算图，NNVM 则统一的对图做必要的操作和优化，然后再生成后端硬件代码。简单地说, NNVM 是一个神经网络的比较高级的中间表示模块，它包含了图的表示以及执行无关的各种优化（例如内存分配，数据类型和形状的推导）-- 陈天奇：NNVM打造模块化深度学习系统
>
> 
>

Graph的执行

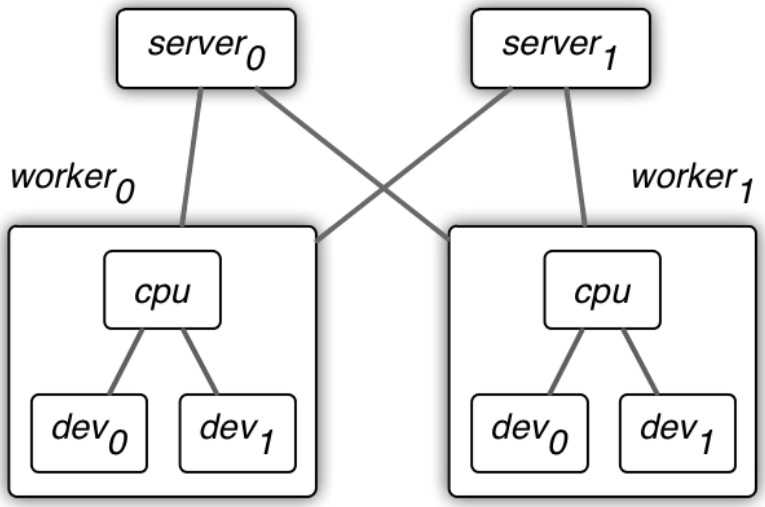
- 在nnvm和engine的基础之上，Graph的执行就非常简单了
- 绑定Graph和输入变量之后生成Executor，Executor执行Forward和Backward只要遍历Graph，将Graph上Op的kernel依次Push给Engine

```
class Executor {
 public:
  virtual void Forward(bool is_train) = 0;
  virtual void PartialForward(bool is_train, int step, int *step_left) =
 0;
  virtual void Backward(const std::vector<NDArray> &head_grads) = 0;
  virtual const std::vector<NDArray> &outputs() const = 0;
  static Executor *Bind(nnvm::Symbol symbol,
                        const Context& default_ctx,
                        const std::map<std::string, Context>& group2ctx,
                        const std::vector<NDArray> &in_args,
                        const std::vector<NDArray> &arg_grad_store,
                        const std::vector<OpReqType> &grad_req_type,
                        const std::vector<NDArray> &aux_states,
                        Executor* shared_exec = NULL);
  //
};
```

mxnet::KVStore

- MXNet在一套接口下设计了两层的通讯结构。
  - 第一层的服务器管理单机内部的多个设备(CPU\GPU)之间的通讯。kvstore_local.h提供了三种模式，区别只是在哪去做设备上的梯度的平均计算和在哪更新参数
  - 第二层服务器则管理机器之间通过网络的通讯。通过ps同步参数，可以选择同步或异步模式。
  - 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。

```
class KVStore {
 public:
  static KVStore *Create(const char *type = "local");
  inline const std::string& type() { return type_; }

  virtual void Init(const std::vector<int>& keys,
                    const std::vector<NDArray>& values) = 0;
  virtual void Push(const std::vector<int>& keys,
                    const std::vector<NDArray>& values,
                    int priority = 0)  = 0;
  virtual void Pull(const std::vector<int>& keys,
                    const std::vector<NDArray*>& values,
                    int priority = 0) = 0;

  typedef std::function<void(int, const NDArray&, NDArray*)> Updater;
  virtual void set_updater(const Updater& updater);

  static bool IsWorkerNode();
  static bool IsServerNode();
  static bool IsSchedulerNode();
  virtual int get_rank() const;
  // ...

  // Run as server (or scheduler)
  //
  // The behavior of a server:
  // \code
  // while(receive(x)) {
  //   if (IsCommand(x)) controller(x)
  //   else if (IsKeyValue(x)) updater(x)
  // }
  // \endcode
  //
  virtual void RunServer(const Controller& controller) { }
};
```



- dist方式实现细节
  - 同步和异步的控制是在Server端。如果是同步模式，收到Worker Push的梯度后先缓存，等待所有Worker都发送Push之后汇总更新Server端的参数后，再返回ACK. 异步模式则收到梯度即更新参数并返回ACK
  - Server端存储的是`std::unordered_map<int, NDArray>`，所有参数都是必须预先定义好，并在启动式由第一个Worker上的参数值初始化
  - 对于稠密参数，key就是参数tensor的id，value就是这个tensor。稀疏参数情况下key为featureid, value为float
  - 所有Server平均划分Key的值域范围，一个Server只负责存储其中一块值域。为了保证负载均衡，用户使用的Key和Server上存储的key不一样，需要编码和解码, Encode过程: hash(user_key)%num_server确定一台ps，那么server_key = ps.range.begin+user_key，解码过程相反，减去那台ps的range起始值。Push会编码解码，Pull只有一次编码
  - 如果一个Key的value太大, 平均partition到所有的Server，key编码规则相同, 相当于将分片放在每个Server相对位置相同的地方。
  - 参数更新逻辑(optimizer)都实现在python中，通过pickle.dumps(optimizer)序列化之后传输给ps
- 示例

```
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);
}
```

- ps-lite框架(并不是一种ps实现)
  - 包括底层通信和上层的Key-value接口,以及消息格式，并不包含Server端如何存储和更新参数，提供Server side UDF。
  - 框架包括Server\Worker\Scheduler。Server存储参数，Work提供Push\Pull接口访问参数，Scheduler全局只有一个，用来同步Server和Worker之间的行为，比如Barrier
  - 轻量化，ODSI2014论文中提到到容错、max delay consistency都没有在此框架中实现

Reference

- mxnet source code (https://github.com/dmlc/mxnet)
- ps-lite source code (https://github.com/dmlc/ps-lite)
- nnvm source code (https://github.com/dmlc/nnvm)
- mshadow source code (https://github.com/dmlc/mshadow)
- MXNet System Architecture (http://mxnet.io/architecture/)
- MXNet设计和实现简介 (https://github.com/dmlc/mxnet/issues/797)
- Scaling Distributed Machine Learning with the Parameter Server (https://www.cs.cmu.edu/~muli/file/parameter_server_osdi14.pdf)
- NNVM打造模块化深度学习系统 (https://mp.weixin.qq.com/s?__biz=MzA3MzI4MjgzMw==&mid=2650719529&idx=3&sn=6992a6067c79349583762cb28eecda89&chks