A Practitioner's Guide to MXNet

Xingjian Shi

Hong Kong University of Science and Technology (HKUST)

HKUST CSE Seminar, March 31st, 2017

Outline

Outline for section 1

Overview of Deep Learning

- Key of Deep Learning
  - Hierarchical Model Structure
  - End-to-end Model (Input $\rightarrow$ Model $\rightarrow$ Output)

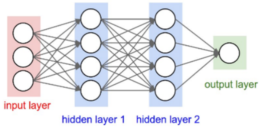

Figure 1: Example of a FNN

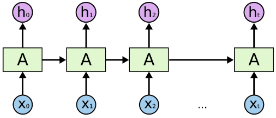

Figure 2: Example of a RNN

- State-of-the-art results in many areas:
  - Object Detection
  - Machine Translation
  - Speech Synthesis
  - ......

Computational Challenges

- Models are becoming more and more complicated!

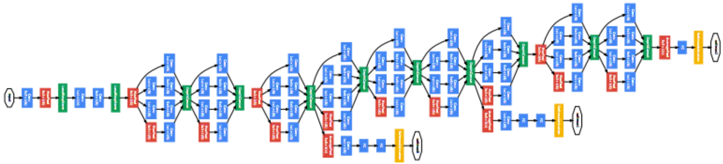

Figure 3: The first version of GoogLeNet (Szegedy et al., 2015)

- Datasets are becoming larger and larger!
  - ImageNet, MS-COCO, WMT...
- Nowadays we rely on Deep Learning Libraries
  - Theano, Caffe, MatConvNet, Torch, CNTK, TensorFlow and MXNet
  - All have their own advantages and disadvantages. None of them is the best!

MXNet Highlights – Popularity

- MXNet is becoming more and more popular!
- Stars: > 9000, Rank 5th
- Fork: > 3300, Rank 4th
- We've joined Apache Incubator.

MXNet Highlights – Efficiency

- Efficient
  - Fast on single machine (C++ back-end)
  - Support automatic parallelization
  - Linear scaling w.r.t No. machines and No. GPUs

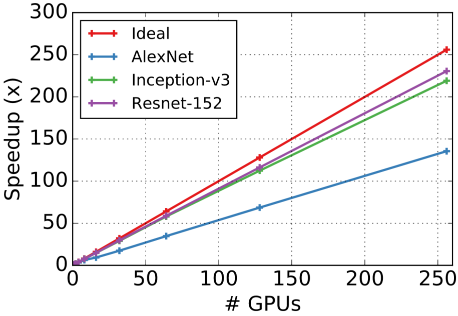

Figure 4: Scalability experiments on 16x AWS P2.16xlarges. 256 GPUs are used in total. CUDA 7.5 + CUDNN 5.1.

MXNet Highlights – Portability

- Portable
  - Front-end in multiple languages (Common back-end)
  - Support multiple operating systems

Figure 5: Part of the languages that are supported.

MXNet Highlights – Flexibility

- Flexible
  - Support both imperative programming and declarative programming
  - Imperative Programming → Numpy, Matlab, Torch
  - Declarative Programming → Tensorflow, Theano, Caffe
  - Mix the flavor: “Mix-Net”

Example 1: Imperative Programming

```
import mxnet.ndarray as nd
a = nd.ones((4, 4))
b = nd.ones((4, 4))
c = a + b
print(c.asnumpy())
```

Example 2: Declarative Programming

```
import mxnet.sym as sym
import numpy as np
a = sym.Variable('a', shape=(4, 4))
b = sym.Variable('b', shape=(4, 4))
c = a + b
# Compile the executor
exe = c.simple_bind(ctx=mx.cpu())
# Run the executor
exe.forward(a=np.ones((4, 4)))
print(exe.outputs[0].asnumpy())
```

Imperative Programming V.S Declarative Programming

- Imperative Programming
  - Straight-forward. Easy to view the middle level results.
  - Example: L-BFGS, Beam Search...
- Declarative Programming
  - Easier to optimize.
  - After getting the computational graph (logic), we could apply rules to simplify the graph. We can also choose the most efficient implementation to do the real computation.

Example 3: Optimization on the graph–1

```
import numpy as np
a = np.random((1000000,))
b = np.exp(a)
c = np.log(b)
d = np.exp(c)
print(d)
# Optimized
d = np.exp(a)
```

Example 4: Optimization on the graph–2

```
import numpy as np
a = np.random((100, 1))
c = np.random((100, 100))
d = np.dot(a, a.T) + c
# We could use a single GER call.
```

Outline for section 2

Installation on Python

- Using pre-compiled packages
  - Linux, MacOS
    ```
    pip install mxnet        # CPU
    pip install mxnet-mkl    # CPU with MKL-DNN
    pip install mxnet-cu75   # GPU with CUDA 7.5
    pip install mxnet-cu80   # GPU with CUDA 8.0
    ```
  - Windows: will support soon
- Compile from source
  - Clone the latest version
    ```
    git clone https://github.com/dmlc/mxnet.git
    ```
  - Need compiler that supports C++11
  - CUDA v8.0 + CUDNN v5.1 is the best combination
  - Use Make or CMake to compile
  - Install by running setup
    ```
    cd mxnet/python
    python setup.py develop --user
    ```

Validate the installation

- Quick testing

```
cd mxnet
# GPU
nosetests tests/python/gpu/test_operator_gpu.py
# Only CPU
nosetests tests/python/unittest/test_operator.py
```

- Import the package

```
>>> import mxnet as mx
```

- Try the examples

```
cd mxnet/example/image-classification
python train_cifar10.py --gpus 0
```

Overview of Low-level APIs

- NDArray API
  - Imperative programming
- Symbol + Executor API
  - Declarative programming
- KVStore API
  - Key to distributed learning

NDArray

- **mxnet.ndarray**
- Container similar to numpy.ndarray. Support multiple running contexts.

```
>>> import mxnet as mx
>>> import mxnet.ndarray as nd
>>> x = nd.array([[1, 2, 3], [4, 5, 6]])
>>> x.asnumpy()
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.]], dtype=float32)
>>> y = nd.array([[4, 5, 6], [1, 2, 3]], ctx=mx.gpu(0))
>>> z = nd.array([[1, 2, 1], [1, 2, 1]], ctx=mx.gpu(1))
>>> x[:] = y.copyto(mx.cpu())
>>> x.asnumpy()
array([[ 4.,  5.,  6.],
       [ 1.,  2.,  3.]], dtype=float32)
```

Example 5: First glance at NDArray

- Need to use x[:] to make sure that we've changed the content of x instead of creating a new variable.

NDArray

- Support most features (auto-broadcasting, axis) in Numpy

```
>>> import mxnet as mx
>>> import mxnet.ndarray as nd
>>> x = nd.array([[1, 3, 2], [7, 2, 1]])
>>> y = nd.array([4, 5, 6])
>>> z = x + y
>>> z.asnumpy()
array([[  5.,   8.,   8.],
       [ 11.,   7.,   7.]], dtype=float32)
>>> nd.argsort(z, axis=0).asnumpy()
array([[ 0.,  1.,  2.],
       [ 1.,  2.,  0.]], dtype=float32)
```

Example 6: Auto-broadcasing and axis support

- All OPs will be asynchronous! The engine will take care of the dependency and try to run them in parallel. We need synchronization before getting the results.
- Lots of OPs, http://mxnet.io/api/python/ndarray.html

Symbol + Executor

- **mxnet.symbol**
- Use symbol to construct the logic. We can suggest the shape of the variable, 0 indicates missing value.

```
>>> import mxnet as mx
>>> a = mx.sym.Variable('a', shape=(3, 2))
>>> b = mx.sym.Variable('b', shape=(3, 0))
>>> c = 2 * a + b
>>> c.list_arguments()
['a', 'b']
>>> c.infer_shape()
([(3L, 2L), (3L, 2L)], [(3L, 2L)], [])
>>> c.eval(a=nd.ones((3, 2)), b=nd.ones((3, 2)))[0].asnumpy()
array([[ 3.,  3.],
       [ 3.,  3.],
       [ 3.,  3.]], dtype=float32)
```

Example 7: Automatic shape inference + Eval

Symbol + Executor

- Bind NDArrays to a symbol to construct the executor, which is the main object for computation.

```
>>> a = mx.sym.Variable('a')
>>> b = mx.sym.Variable('b')
>>> c = 2 * a + b
>>> exe = c.simple_bind(mx.cpu(), a=(2,), b=(2,))
>>> exe.forward(is_train=True)
>>> exe.backward(out_grads=nd.array([-1, 1]))
>>> exe.grad_dict['a'].asnumpy()
array([ -2.,  2.], dtype=float32)
>>> exe.grad_dict['b'].asnumpy()
array([ -1.,  1.], dtype=float32)
```

- We use Reverse-mode Automatic Differentiation. Also known as Back-propagation. Compute vector-Jacobian product.
  - $\frac{\partial g(f(x))}{\partial x} = \frac{\partial g(f(x))}{\partial f(x)} \frac{\partial f(x)}{\partial x}$

Symbol + Executor

- We have symbols that are commonly used in neural networks.

```
>>> data = mx.sym.Variable('data')
>>> conv1 = mx.sym.Convolution(data=data,
                               num_filter=16,
                               kernel=(3, 3),
                               name="conv1")
>>> fc1 = mx.sym.FullyConnected(data=conv1,
                                num_hidden=16,
                                name="fc1")
>>> fc1.list_arguments()
['data', 'conv1_weight', 'conv1_bias',
 'fc1_weight', 'fc1_bias']
```

- The parameters will be automatically created. We can also explicitly create the parameter symbols.

Symbol + Executor

```
>>> data = mx.sym.Variable('data')
>>> weight = mx.sym.Variable('weight')
>>> bias = mx.sym.Variable('bias')
>>> conv1 = mx.sym.Convolution(data=data,
                               weight=weight,
                               bias=bias,
                               num_filter=16,
                               kernel=(3, 3),
                               name="conv1")
>>> conv1.list_arguments()
['data', 'weight', 'bias']
```

Symbol + Executor

- We could construct loss symbols by make_loss

```
>>> data = mx.sym.Variable('data')
>>> label = mx.sym.Variable('label')
>>> loss = mx.sym.mean(
              mx.sym.softmax_cross_entropy(data=data,
                                           label=label))
>>> loss = mx.sym.make_loss(loss, name="cross_entropy")
```

- We can group multiple symbols

```
>>> data = mx.sym.Variable('data')
>>> target = mx.sym.Variable('target')
>>> l2 = mx.sym.mean(mx.sym.square(data - target))
>>> l2 = mx.sym.make_loss(l2, name="l2")
>>> out = mx.sym.Group([l2, data])
>>> out.list_outputs()
['l2_output', 'data_output']
```

- Same set of operations as in NDArray are supported!
  Symbol API

Symbol + Executor

- Straight-forward SGD with Low-level API

```
>>> data = mx.sym.Variable('data')
>>> target = mx.sym.Variable('target')
>>> weight = mx.sym.Variable('weight')
>>> bias = mx.sym.Variable('bias')
>>> conv1 = mx.sym.Convolution(data=data,
                                weight=weight,
                                bias=bias,
                                num_filter=3,
                                kernel=(3, 3),
                                pad=(1, 1),
                                name="conv1")
>>> l2 = mx.sym.mean(mx.sym.square(conv1 - target))
>>> l2 = mx.sym.make_loss(l2, name="l2")
>>> exe = l2.simple_bind(ctx=mx.gpu(), data=(10, 3, 5, 5),
                         target=(10, 3, 5, 5))
>>> for i in range(10):
        exe.foward(is_train=True, data=..., target=...)
        exe.backward()
        exe.arg_dict['weight'] -= lr * exe.grad_dict['weight']
        exe.arg_dict['bias'] -= lr * exe.grad_dict['bias']
```

KVStore

- **mxnet.kvstore**
- Implementation of Parameter Server (PS)
- Pull, Push and Update
- Example: Downpour SGD
    - Client pull the parameter from the server
    - Client compute the gradient
    - Client push the gradient to the server
    - Server will update the stored parameter once receiving gradient
- Use ‘kv.pull()’ and ‘kv.push()’ in MXNet

Overview of High-level APIs

- Low-level APIs are good if you want to implement some brand new algorithms. E.g, implement new distributed machine learning algorithms.
- Just some standard training/testing scheme?
- Use high-level API $\rightarrow$ mx.mod.Module

Module

- **mxnet.module**
- First, use symbol API to create your model.

```
data=mx.sym.Variable('data')
fc1=mx.sym.FullyConnected(data,name='fc1',num_hidden=128)
act1=mx.sym.Activation(fc1,name='relu1',act_type='relu')
fc2=mx.sym.FullyConnected(act1,name='fc2',num_hidden=10)
out=mx.sym.SoftmaxOutput(fc2,name='softmax')
```

- Next, feed a symbol into Module.

```
# create a module by given a Symbol
mod = mx.mod.Module(out)
```

- Now you can use Module APIs.

Module

- **mxnet.module**
- First, use symbol API to create your model.

```
data=mx.sym.Variable('data')
fc1=mx.sym.FullyConnected(data,name='fc1',num_hidden=128)
act1=mx.sym.Activation(fc1,name='relu1',act_type='relu')
fc2=mx.sym.FullyConnected(act1,name='fc2',num_hidden=10)
out=mx.sym.SoftmaxOutput(fc2,name='softmax')
```

- Next, feed a symbol into Module.
- Automatic data parallel with multiple GPUs in a single machine.

```
# create a module by given a Symbol
mod = mx.mod.Module(out, ctx=[mx.gpu(0), mx.gpu(1), ...])
```

- Now, you can use Module APIs.

Module

- Then, allocate memory by given input shapes and initialize the module:

```
mod.bind(data_shapes=data.provide_data,
         label_shapes=data.provide_label)
# initialize parameters with the default initializer
mod.init_params()
```

- Now, you can train and predict.
  - Call high-level API

    ```
    mod.fit(data,num_epoch=10, ...) # train
    mod.predict(new_data) # predict on new data
    ```

  - Perform step-by-step computations

    ```
    # forward on the provided data batch
    mod.forward(data_batch)
    # backward to calculate the gradients
    mod.backward()
    # update parameters using the default optimizer
    mod.update()
    ```

Standard Training/Testing Logic

- Training

```
sym = symbol_builder(ctx=[mx.gpu(0), mx.gpu(1), ...])
net = build_module(sym)
for i in range(TOTAL_TRIAN_BATCH):
  training_batch = draw_batch() # data + label
  net.forward_backward(data_batch=training_batch)
  net.update()
  logging.info(...) # Log the statistics
  if (i + 1) % SAVE_ITER == 0:
    net.save_checkpoint(prefix="model", epoch=i)
```

- Testing

```
net = mx.mod.Module.load(prefix="model", epoch=1000)
for i in range(TOTAL_TEST_BATCH):
  testing_batch = draw_batch() # data
  net.forward(is_train=False, data_batch=testing_batch)
  outputs = net.get_outputs()
  loss += loss_function(outputs, label)
logging.info(loss) # Log the loss
```

CNN and RNN

- CNN
  Use the given symbols to construct the loss.
  Sample AlexNet
- RNN
  The key is to share the parameter symbols. Following is RNN-tanh.

```
weight = mx.sym.Variable('weight')
bias = mx.sym.Variable('bias')
state = mx.sym.zeros(shape=(0, 0))
for i in range(10):
    state = mx.sym.FullyConnected(
                data=mx.sym.Concat(data[i], state, num_args=2),
                weight=weight,
                bias=bias,
                num_hidden=100)
    state = mx.sym.tanh(state)
```

Link to RNN Cells in MXNet

Outline for section 3

Write New Operators

- Use CustomOp in the front-end language (i.e., Python)
  - Can be very fast (use mx.nd)
  - Can also be relatively slow (use numpy)
- Use C++ (CUDA).
  - Gain best performance
- Operator testing
  - Use functions in **mx.test_utils** to automatically check the correctness of the forward and backward pass
  - We support automatic gradient checker using central difference.

```
from mxnet.test_utils import check_numeric_gradient
check_numeric_gradient(YOUR_SYMBOL, location=INPUT_VALUES)
```

Tricks to Debug the Program

- Use CustomOps to view the mid-level result
  - Create some special ops that works like an identity mapping
  - Use ‘asnumpy()‘ in the CustomOp to synchronize

```
sym1 = ...
# Insert our debugging OP
sym1 = custom_debug(sym1)
sym2 = ...sym1...
```

- Visualize Gradient Statistics
  - Gradient Norm, Uphill Steps, ...
  - Can be implemented in MXNet using Imperative APIs

Outline for section 4

Summary

- MXNet is efficient, portable and flexible
- NDArray for imperative programming, Symbol + Executor for declarative programming, KVStore for distributed learning
- Module is used as a high level wrapper of the network
- CustomOp can be implemented via Python/C++ and can be used for debugging