The goals of this presentation:

• Capture a deeper understanding of Design-And-Run vs. Design-By-Run
• Introduce Chainer and recognize its differences with past frameworks
• Understand the performance advantages and structure of MXNet
A Powerful, Flexible, and Intuitive Framework for Neural Networks

Seiya Tokui, Kenta Oono, Shohei Hido, Justin Clayton: Preferred Networks
ABSTRACT

“… as new types of deep learning models are developed, existing frameworks designed for convolutional neural networks are becoming less useful … Chainer provides a flexible, intuitive, and high performance means of implementing a full range of deep learning models, including state-of-the-art models such as recurrent neural networks and variational autoencoders.”

- Chainer: A Next-Generation Open Source Framework for Deep Learning, 2015*
INTRODUCTION

Reviewing AI Research from 2015
INTRODUCTION: FEED FORWARD NEURAL NETWORK

- Deep Learning requires tools such as multiple dimension arrays, activation functions, and autonomous gradient computation
- Caffe & Torch: First frameworks designed to prevent duplicating these tools (Tensorflow 1.0 just released)
- Feed Forward Neural Networks:
  - i.e. convolutional neural networks
  - Great for computer vision and speech recognition
  - Effectively analyzes data samples of fixed length

Courtesy Wikipedia
INTRODUCTION: RNNS AND LSTM MEMORY

- In 2013, Google DeepMind showed groundbreaking results at the NIPS Deep Learning Workshop using Deep Reinforcement Learning
- Recurrent Neural Network (RNN): Has connections forming a directed graph along a temporal sequence and can exhibit temporal dynamic behavior
- Long Short-Term Memory (LSTM): Has feedback connections to process both single data points and full data sequences
- Promising results on variable-length data and use in Machine Translation & Conversation models

Courtesy Wikipedia
DEEP LEARNING FRAMEWORKS

From Define-And-Run to Define-By-Run
EXISTING FRAMEWORK LIMITATIONS

• Deep learning architectures moved beyond the frameworks of the time
• Focused on CNNs and not newer deep learning models like RNNs & LSTM
• Framework Setup:
  • Domain-Specific language to represent models
  • Interpreter translated model into data structure
• Existing frameworks were Define-And-Run
Inefficient memory usage

- Computational graph built before model training
- All NN layers must stay in memory (inefficient for backpropagation through time)

Limited extensibility: What if you need new functionality?

- Fork the repository: Divides contributions and efforts
- Hack the existing code base: Decreases development efficiency and creates maintenance problems

Inner workings not accessible to the user

- To debug and tune a model, you need to see what’s happening inside
- The computational graph becomes a black box in define-and-run
DEFINE-AND-RUN VS. DEFINE-BY-RUN

(a) Define-and-Run: existing approach

(b) Define-by-Run: new approach
CHAINER OVERVIEW

Basics, Examples, and Benchmarks
• Second-generation deep learning framework using Define-By-Run approach
• Open-source, written in Python
• Implements CuPy for GPU accelerated computation
• Supports popular optimization methods:
  • SGD, AdaGrad, RMSprop, Adam
• Automatic gradient computation for backpropagation
• Numerical operations including convolutions, losses, and activation functions implemented as Function
EXAMPLE: MLP

```python
# (1) Function Set definition
model = FunctionSet(
    l1=F.Linear(784, 100),
    l2=F.Linear(100, 100),
    l3=F.Linear(100, 10)).to_gpu()

# (2) Optimizer Setup
opt = optimizers.SGD()
opt.setup(model)

# (3) Forward computation
def forward(x, t):
    h1 = F.relu(model.l1(x))
    h2 = F.relu(model.l2(h1))
    y = model.l3(h2)
    return F.softmax_cross_entropy(y, t)

# (4) Training loop
for epoch in xrange(n_epoch):
    for i in xrange(0, N, b_size):
        x = Variable(to_gpu(...))
        t = Variable(to_gpu(...))
        opt.zero_grads()
        loss = forward(x, t)
        loss.backward()
        opt.update()
```
MLP: STEP 1

- Initial operation for defining three linear (fully-connected) layers.
- (Not shown):
  ```python
  import chainer.functions as F
  ```
- Consists only of layer definitions
- Stored on the GPU

```python
# (1) Function Set definition
model = FunctionSet(  
  11=F.Linear(784, 100),
  12=F.Linear(100, 100),
  13=F.Linear(100, 10)), to_gpu()

# (2) Optimizer Setup
opt = optimizers.SGD()
opt.setup(model)

# (3) Forward computation
def forward(x, t):
  h1 = F.relu(model.11(x))
  h2 = F.relu(model.12(h1))
  y = model.13(h2)
  return F.softmax_cross_entropy(y, t)

# (4) Training loop
for epoch in xrange(n_epoch):
  for i in xrange(0, N, b_size):
    x = Variable(to_gpu(...))
    t = Variable(to_gpu(...))
    opt.zero_grads()
    loss = forward(x, t)
    loss.backward()
    opt.update()
```
MLP: STEP 2

- Initialize the optimizer
- Can use optimizers.Adam(), optimizers.AdaDelta(), etc.
- Reference the MLP model

```python
# (1) Function Set definition
model = FunctionSet()
    11=F.Linear(784, 100),
    12=F.Linear(100, 100),
    13=F.Linear(100, 10)).to_gpu()

# (2) Optimizer Setup
opt = optimizers.SGD()
opt.setup(model)

# (3) Forward computation
def forward(x, t):
    h1 = F.relu(model.11(x))
    h2 = F.relu(model.12(h1))
    y = model.13(h2)
    return F.softmax_cross_entropy(y, t)

# (4) Training loop
for epoch in xrange(n_epoch):
    for i in xrange(0, N, b_size):
        x = Variable(to_gpu(...))
        t = Variable(to_gpu(...))
        opt.zero_grads()
        loss = forward(x, t)
        loss.backward()
        opt.update()
```
MLP: STEP 3

- Uses the forward() method
- Defines inter-layer connects
- Use ReLu activation function
- Softmax Entropy
- Return y

```python
# (1) Function Set definition
model = FunctionSet(
    11=F.Linear(784, 100),
    12=F.Linear(100, 100),
    13=F.Linear(100, 10)).to_gpu()
# (2) Optimizer Setup
opt = optimizers.SGD()
opt_setup(model)

# (3) Forward computation
def forward(x, t):
    h1 = F.relu(model.11(x))
    h2 = F.relu(model.12(h1))
    y = model.13(h2)
    return F.softmax_cross_entropy(y, t)

# (4) Training loop
for epoch in xrange(n_epoch):
    for i in xrange(0, N, b_size):
        x = Variable(to_gpu(...))
        t = Variable(to_gpu(...))
        opt.zero_grads()
        loss = forward(x, t)
        loss.backward()
        opt.update()
```
MLP: STEP 4

- Training loop
- Nested for-loops for batch training
- Save variables on GPU

```python
# (1) Function Set definition
model = FunctionSet(
    11=F.Linear(784, 100),
    12=F.Linear(100, 100),
    13=F.Linear(100, 10)).to_gpu()

# (2) Optimizer Setup
opt = optimizers.SGD()
opt.setup(model)

# (3) Forward computation
def forward(x, t):
    h1 = F.relu(model.11(x))
    h2 = F.relu(model.12(h1))
    y = model.13(h2)
    return F.softmax_cross_entropy(y, t)

# (4) Training loop
for epoch in xrange(n_epoch):
    for i in xrange(0, N, b_size):
        x = Variable(to_gpu(...))
        t = Variable(to_gpu(...))
        opt.zero_grads()
        loss = forward(x, t)
        loss.backward()
        opt.update()
```
EXAMPLE: SIMPLE RNN

# (1) Function Set definition
model = FunctionSet(
    emb=F.EmbedID(1000, 100),
    x2h=F.Linear(100, 50),
    h2h=F.Linear(50, 50),
    h2y=F.Linear(50, 1000)).to_gpu()

# (2) Optimizer Setup
opt = optimizers.SGD()
opt.setup(model)

# (3) One step forward
def fwdlstep(h, w, t):
    x = F.tanh(model.emb(w))
    h = F.tanh(model.x2h(x) + model.h2h(h))
    y = model.h2y(h2)
    return h, F.softmax_cross_entropy(y, t)

# (4) Full RNN forward computation
def forward(seq)
    h = Variable() # init state
    loss = 0
    for curw, nextw in zip(seq, seq[1:]):
        w = Variable(curw)
        t = Variable(nextw)
        h, new_loss = fwdlstep(h, w, t)
        loss += new_loss
    return loss
DIFFERENCES

SIMPLE MLP

```python
# (1) Function Set definition
definition model = FunctionSet

11=F.Linear(784, 100),
12=F.Linear(100, 100),
13=F.Linear(100, 10)).to_gpu()

# (2) Optimizer Setup
opt = optimizers.SGD()

# (3) Forward computation
def forward(x, t):
    h1 = F.relu(model.11(x))
    h2 = F.relu(model.12(h1))
    y = model.13(h2)
    return F.softmax_cross_entropy(y, t)

# (4) Training loop
for epoch in range(n_epoch):
    for i in range(0, N, b_size):
        x = Variable(to_gpu(...))
        t = Variable(to_gpu(...))
        opt.zero_grads()
        loss = forward(x, t)
        loss.backward()
        opt.update()
```

SIMPLE RNN

```python
# (1) Function Set definition
model = FunctionSet

emb=F.EmbedID(1000, 100),
12=F.Linear(100, 50),
13=F.Linear(50, 50),
14=F.Linear(50, 1000)).to_gpu()

# (2) Optimizer Setup
opt = optimizers.SGD()

# (3) One step forward
def forward_step(h, w, t):
    x = F.tanh(model.emb(w))
    h = F.tanh(model.12h(x) + model.12h(h))
    y = model.13h2(y(h))
    return h, F.softmax_cross_entropy(y, t)

# (4) Full RNN forward computation
def forward(seq):
    h = Variable() # init state
    loss = 0
    for curw, nextw in zip(seq, seq[1:]):
        w = Variable(curw)
        t = Variable(nextw)
        h, new_loss = forward_step(h, w, t)
        loss += new_loss
    return loss
```
• The model now uses EmbedID, a word2vec type of layer

• Function fwd1step:
  • Use recurrent layer $h$ as the input and output
  • Layer $h$ updated using current input $x$ and prev. state $h$

• This representation of an RNN was not available in earlier deep learning frameworks (Caffe)

• Repeat fwd1step and collect losses at each step for use when updating the RNN using BPTT

• Python debugging and profiling available!
BENCHMARKS

- Comparison to Caffe
  - Train on ImageNet dataset
  - 5 Standard CNNs, AlexNet, Overfeat, & VGG
- Only the first batch requires memory pool allocation & GPU kernel transfer
- 1st batch is up to 7.0 times slower than subsequent batches
- *Chainer doesn’t need to recompile after making changes, saving time in trial-and-error*
<table>
<thead>
<tr>
<th></th>
<th>Basic convolutional nets</th>
<th>ImageNet</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>conv1</td>
<td>conv2</td>
<td>conv3</td>
</tr>
<tr>
<td>batchsize</td>
<td>64</td>
<td>32</td>
<td>64</td>
</tr>
<tr>
<td>forward</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chainer 1st</td>
<td>245.62</td>
<td>253.08</td>
<td>266.21</td>
</tr>
<tr>
<td>Chainer 2-11th</td>
<td>72.31</td>
<td>69.00</td>
<td>85.17</td>
</tr>
<tr>
<td>Caffe 1-10th</td>
<td>47.14</td>
<td>34.66</td>
<td>59.54</td>
</tr>
<tr>
<td>backward</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chainer 1st</td>
<td>103.01</td>
<td>101.14</td>
<td>90.44</td>
</tr>
<tr>
<td>Chainer 2-11th</td>
<td>85.48</td>
<td>92.33</td>
<td>82.58</td>
</tr>
<tr>
<td>Caffe 1-10th</td>
<td>101.44</td>
<td>105.76</td>
<td>86.46</td>
</tr>
</tbody>
</table>

**TABLE: MEAN TIME FOR COMPUTATIONS ON VARIOUS NETWORKS (MSEC)**
CHAINER EXTENSIONS

• ChainerMN: Multiple GPU integration
• ChainerRL: Additional state of the art deep reinforcement
• ChainerCV: Additional computer vision algorithms and functions
• ChainerUI: Management and visualization tools
CHAINER IN APPLICATIONS

- Trained ResNet-50 on ImageNet in 15 minutes, breaking Facebook’s record
  - 90 epochs, large minibatch size of 32k
  - Used 1024 Tesla P100 GPUs
  - Uses ChainerMN
- PaintsChainer: Line drawing colorizer
- Official AWS Support through CloudFormation
GETTING STARTED WITH CHAINER

- Go to https://chainer.org
- To use Chainer:
  - $pip install chainer
- Run MNIST example:
  - $wget https://github.com/chainer/chainer/archive/v6.3.0.tar.gz
  - $tar xzf v6.3.0.tar.gz
  - $python chainer-6.3.0/examples/mnist/train_mnist.py
- Official Documentation: https://docs.chainer.org
A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems

Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, Zheng Zhang
“…MXNet is a multi-language machine learning (ML) library to ease the development of ML algorithms, especially for deep neural networks … MXNet blends declarative symbolic expression with imperative tensor computation … MXNet is computation and memory efficient and runs on various heterogeneous systems, ranging from mobile devices to GPU clusters.”

- MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems *(2015)*
INTRODUCTION

Programming Paradigms:

- **Imperative**: how computation needs to be performed (numpy)
- **Declarative**: what needs to be done (Caffe, CXXNet)
- Dividing line can be muddy at times (Theano, Tensorflow)

Program Execution

- **Concrete**: Result returned right away on same thread
- **Asynchronous**: Statements gathered and transformed into dataflow graph first
MXNET: BLENDING APPROACHES

- Uses **declarative** programming to specify the computation structure
- Embed multiple host languages (C++, Python, R, Go, & Julia)
- Able to fuse different programming languages & paradigms into the same backend engine
- Aggressively reduces memory footprint, performing in-place update & memory space reuse whenever possible
- Provides superset programming interface to Torch7, Chainer, Theano, Caffe, Tensorflow with support for GPU clusters
<table>
<thead>
<tr>
<th>System</th>
<th>Core Lang</th>
<th>Binding Langs</th>
<th>Devices (beyond CPU)</th>
<th>Distributed</th>
<th>Imperative Program</th>
<th>Declarative Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>Caffe [7]</td>
<td>C++</td>
<td>Python/Matlab</td>
<td>GPU</td>
<td>×</td>
<td>×</td>
<td>√</td>
</tr>
<tr>
<td>Torch7 [3]</td>
<td>Lua</td>
<td>-</td>
<td>GPU/FPGA</td>
<td>×</td>
<td>√</td>
<td>×</td>
</tr>
<tr>
<td>Theano [1]</td>
<td>Python</td>
<td>-</td>
<td>GPU</td>
<td>×</td>
<td>×</td>
<td>√</td>
</tr>
<tr>
<td>MXNet</td>
<td>C++</td>
<td>Python/R/Julia/Go</td>
<td>GPU/Mobile</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
</tbody>
</table>

**MXNET COMPARISON TO ML LIBRARIES**
MXNET PROGRAMMING INTERFACE

Symbols, NDArrays, KVStore, and additional Modules
MXNet uses multi-output symbolic expressions, \textit{Symbol}, to declare the computational graph.

Composited by operators: matrix ops or complex neural net layer.

Variables can be free or an output of another symbol.

To evaluate a symbol: bind free variables with data, then declare outputs.

Symbol has “forward” evaluation and auto symbolic differentiation ("backwards").

Supports load, save, memory estimation, and visualization.

```
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()
```
**NDARRAY**

- NDArray (N-Dimensional Array) has imperative tensor computation
- Example: Gradient Descent with symbolic NN and weight fn. \((w = w - ng)\)

```python
while(1) { net.forward_backward(); net.w -= eta * net.g; }
```
- MXNet uses lazy evaluation of NDArray
- Backend correctly resolves data dependency between the two

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

- Distributed Key-Value store for data synchronization over multiple devices
- Two primitives: push and pull
- Example: Distributed gradient descent by data parallelization

```python
while(1){ kv.pull(net.w); net.forward_backward(); kv.push(net.g); }  
```
- Weight updating function registered to KVStore
- Each worker repeatedly pulls newest weight and pushed locally computed gradient
- Same performance comparing to single declarative program
OTHER MODULES

- MXNet has tools to pack arbitrary sized examples into single compact file:
  - Facilitate sequential and random seek
  - Data iterators
  - Multi-Threaded data pre-fetching and pre-processing
- Training module implements optimization algorithms (SGD)
  - Optionally distributed with KVStore provided
MXNET IMPLEMENTATION

The Computation Graph
• Binded symbolic expression
• MXNet transforms the graph to:
  • Optimize efficiency
  • Allocate memory to internal variables
• Only the subgraph required to obtain outputs specified during binding
• Examples:
  • Prediction only requires forward graph
  • Extracting features from internal layers does not require last layers
• Operators can be grouped
  • $a \times b + 1$ replaced by single BLAS/GPU call
• Manual implementation of operations on layers in neural network
• Variable lifetime is known for computation graph
• Ideal allocation strategy: $O(n^2), n = \text{number of variables}$
• 2 Strategies for linear time complexity:
  • *Inplace*: simulates graph traversal & keeps reference counter of unused nodes
  • *Co-Share*: Allow 2 nodes to share piece of memory iff cannot run parallel
    • Find longest path & perform needed memory allocation
• Each source unit registered to engine with unique, randomly generated tag
• Any operations pushed to engine with specified tags
• Engine tasks:
  • Schedules pushed operations if dependencies resolved
  • Uses multiple threads to schedule operations
  • Tracks mutation operations to schedule array mutations (as in numpy)
  • Has easier special operation scheduling
    • e.g. generating two random numbers same seed
DATA COMMUNICATION

- KVStore based on parameter server
- Use Engine to schedule KVStore operations & manage data consistency
- Adopt Two-Level Structure:
  - Level-1: Server manages data synchronization between devices in single machine
  - Level-2: Server manages inter-machine synchronization
- Outbound data from level-1 server aggregated to reduce bandwidth
- Inter-machine synch. Can use different consistency model
EVALUATION

Raw Performance & Memory Usage
RAW PERFORMANCE

- Compare MXNet with Torch7, Caffe & TensorFlow on “convnet-benchmarks”
- MXNet has similar performance to Torch7 and Caffe
- TensorFlow always 2x slower (using lower CUDNN version)
• “inplace” and “co-share” effectively reduce memory footprint
• Combining results in 2x reduction for all networks during training, up to 4x for model prediction
• VGG net: training needs less than 16MB extra
SCALABILITY

- Experiment on Amazon EC2 g2.8x instances
- Four Nvidia GK104 GPUs & 10G Ethernet
- Train googlenet with batch normalization on ILSVRC12 dataset
  - 1.3 million images, 1,000 classes
  - Learning Rate = 0.05, Momentum = 0.9,
  - Weight Decay = $10^{-4}$
  - Each GPU: 36 images per batch
  - Average cost of data pass = 14K on single machine; 1.4K sec on 10 machines
MXNET APPLICATIONS

- Available as Apache MXNet on AWS
- Designed to be distributed on dynamic cloud infrastructure
- Gluon: API for Deep Learning
  - GluonCV & GluonNLP Toolkits
GETTING STARTED WITH MXNET

- Website: [https://mxnet.incubator.apache.org](https://mxnet.incubator.apache.org)
- Installing:
  - $pip install mxnet
  - Or $pip install mxnet-mkl
SUMMARY

- Chainer: The first Define-By-Run framework specializing in constructing RNNs
  - https://docs.chainer.org
  - $pip install chainer
- MXNet: Scalable Deep Learning Framework supported by AWS
  - https://mxnet.incubator.apache.org
  - $pip install mxnet

Questions? Thank You!