LBANN: Livermore Big Artificial Neural Network HPC Toolkit

Brian Van Essen, Hyojin Kim, Roger Pearce, Kofi Boakye, Barry Chen
Lawrence Livermore National Laboratory
MLHPC 2015

Presented by: Sourav Chakraborty
Who is LLNL?

Office of Science Laboratories
1. Ames Laboratory
   Ames, Iowa
2. Argonne National Laboratory
   Argonne, Illinois
3. Brookhaven National Laboratory
   Upton, New York
4. Fermi National Accelerator Laboratory
   Batavia, Illinois
5. Lawrence Berkeley National Laboratory
   Berkeley, California
6. Oak Ridge National Laboratory
   Oak Ridge, Tennessee
7. Pacific Northwest National Laboratory
   Richland, Washington
8. Princeton Plasma Physics Laboratory
   Princeton, New Jersey
9. SLAC National Accelerator Laboratory
   Menlo Park, California
10. Thomas Jefferson National Accelerator Facility
    Newport News, Virginia

Other DOE Laboratories
1. Idaho National Laboratory
   Idaho Falls, Idaho
2. National Energy Technology Laboratory
   Morgantown, West Virginia
   Pittsburgh, Pennsylvania
   Albany, Oregon
3. National Renewable Energy Laboratory
   Golden, Colorado
4. Savannah River National Laboratory
   Aiken, South Carolina
5. Lawrence Livermore National Laboratory
   Livermore, California
6. Los Alamos National Laboratory
   Los Alamos, New Mexico
7. Sandia National Laboratory
   Albuquerque, New Mexico
   Livermore, California

NNSA Laboratories
What do they do?

“LLNL’s mission is applying world-class science, technology, and engineering to national & global problems.”
Why LBANN?

Workload
- Novel fields of application
- Lack of labeled datasets
- Focus on unsupervised learning

Scale
- Lack of scalable and distributed training
- Many CPUs/GPUs across multiple nodes

Platform
- Utilize HPC resources
- Low latency communication
- Parallel filesystems

Software
- Take advantage of existing runtimes (MPI)
- Work well with other software stacks (scheduler)
Traditional Deep Learning Applications

- **COMPUTER VISION**
  - Image Classification
  - Object Detection

- **SPEECH & AUDIO**
  - Voice Recognition
  - Language Translation

- **NATURAL LANGUAGE PROCESSING**
  - Recommendation Engines
  - Sentiment Analysis
Upcoming Deep Learning Applications
Supervised vs Unsupervised Learning

**Supervised**
- Input data is labelled
- Classification, Regression
- Computer vision, Speech processing

**Unsupervised**
- Input data not labelled
- Clustering, Density estimation, Dimensionality reduction
- Insurance underwriting, Fraud detection
Why Unsupervised Learning?

• Large labeled datasets available for images
• Not available for many other domains
• Manual labeling is time consuming and expensive
  • sometimes impossible due to access restrictions
• Unsupervised learning assigns labels automatically to input data

MNIST dataset clustered and visualized with TensorBoard
ILSVRC winners from 2010 to 2017

- Error rates are decreasing
  - 28% in 2010 (Traditional Machine Learning)
  - 16% in 2012 (AlexNet)
  - Surpass Human accuracy in 2015 (Resnet)
  - 2.3% in 2017 (SENet)

Safety-critical applications (Medical, Self-driving cars) demand even more accuracy!
The cost of increased accuracy

- Larger and Deeper networks
- More complex models
- 30-fold increase from AlexNet (8 layers) to 256 layers (CUImages)
- Prone to overfitting
Bias-Variance Tradeoff

Smaller Models -> High Bias
Larger Models -> Low Bias, but High variance
How to get the “Best Fit”?
Solution: More Training Data!

Accuracy scales with Data

- 40% error reduction for each 10x increase in dataset size
Andrew Ng’s Deep Learning Rocket Analogy

• Powerful engine: Use large Low Bias models

• Rocket fuel: Minimize Variance with vast training data

Need large computational horsepower!
HPC to the rescue!

HPC systems @ LLNL

<table>
<thead>
<tr>
<th>System</th>
<th>Top500 Rank</th>
<th>Program</th>
<th>Manufacture / Model</th>
<th>OS</th>
<th>Interconnect</th>
<th>Serial Nodes</th>
<th>Serial Cores</th>
<th>Memory (GB)</th>
<th>Peak TFLOP/s</th>
<th>TFLOP/s (GFlops)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unclassified Network (OCF)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vulcan</td>
<td>23</td>
<td>ASC+M&amp;C+HPIC</td>
<td>IBM BGQ</td>
<td>RH/EN/CRN</td>
<td>SD Torus</td>
<td>24,576</td>
<td>393,216</td>
<td>393,216</td>
<td>5,033.2</td>
<td></td>
</tr>
<tr>
<td>Cab (TLC2)</td>
<td></td>
<td>ASC+M&amp;C+HPIC</td>
<td>Appro</td>
<td>TOSS</td>
<td>IB QDR</td>
<td>1,296</td>
<td>20,736</td>
<td>41,472</td>
<td>426.0</td>
<td></td>
</tr>
<tr>
<td>Quartz</td>
<td>46</td>
<td>ASC+M&amp;C</td>
<td>Penguin</td>
<td>TOSS</td>
<td>Om-Pen-Path</td>
<td>2,688</td>
<td>96,768</td>
<td>344,064</td>
<td>3251.4</td>
<td></td>
</tr>
<tr>
<td>RZTopaz</td>
<td></td>
<td>ASC</td>
<td>Penguin</td>
<td>TOSS</td>
<td>Om-Pen-Path</td>
<td>768</td>
<td>27,648</td>
<td>98,304</td>
<td>929.0</td>
<td></td>
</tr>
<tr>
<td>RZManta</td>
<td></td>
<td>ASC</td>
<td>IBM RHEL</td>
<td>IB EDR</td>
<td></td>
<td>36</td>
<td>720</td>
<td>11,520</td>
<td>597.6</td>
<td>676.8</td>
</tr>
<tr>
<td>Ray</td>
<td></td>
<td>ASC+M&amp;C</td>
<td>IBM RHEL</td>
<td>IB EDR</td>
<td></td>
<td>34</td>
<td>1,080</td>
<td>17,280</td>
<td>896.4</td>
<td>1015.2</td>
</tr>
<tr>
<td>Catalyst</td>
<td></td>
<td>ASC+M&amp;C</td>
<td>Cray</td>
<td>TOSS</td>
<td>IB QDR</td>
<td>324</td>
<td>7,776</td>
<td>41,472</td>
<td>149.3</td>
<td></td>
</tr>
<tr>
<td>Syrah</td>
<td></td>
<td>ASC+M&amp;C</td>
<td>Cray</td>
<td>TOSS</td>
<td>IB QDR</td>
<td>324</td>
<td>7,184</td>
<td>20,736</td>
<td>107.8</td>
<td></td>
</tr>
<tr>
<td>Surface</td>
<td></td>
<td>ASC+M&amp;C</td>
<td>Cray</td>
<td>TOSS</td>
<td>IB EDR</td>
<td>162</td>
<td>2,592</td>
<td>41,500</td>
<td>451.9</td>
<td>451.9</td>
</tr>
<tr>
<td>Borax</td>
<td></td>
<td>ASC+M&amp;C</td>
<td>Penguin</td>
<td>TOSS</td>
<td>N/A</td>
<td>48</td>
<td>1,728</td>
<td>6,144</td>
<td>58.1</td>
<td></td>
</tr>
<tr>
<td>RZTanna</td>
<td></td>
<td>ASC</td>
<td>Penguin</td>
<td>TOSS</td>
<td>N/A</td>
<td>48</td>
<td>1,728</td>
<td>6,144</td>
<td>58.1</td>
<td></td>
</tr>
<tr>
<td>Herd</td>
<td></td>
<td>M&amp;C</td>
<td>Appro</td>
<td>TOSS</td>
<td>IB EDR</td>
<td>9</td>
<td>256</td>
<td>1,088</td>
<td>1.6</td>
<td></td>
</tr>
<tr>
<td>OCF Totals</td>
<td>Systems 12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Classified Network (SCF)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pinot (TLC2, SNS)</td>
<td>M&amp;C</td>
<td>Appro</td>
<td>TOSS</td>
<td>IB QDR</td>
<td></td>
<td>162</td>
<td>2,592</td>
<td>10,368</td>
<td>53.9</td>
<td></td>
</tr>
<tr>
<td>Sequoia</td>
<td>5</td>
<td>ASC</td>
<td>IBM BGQ</td>
<td>RH/EN/CRN</td>
<td>SD Torus</td>
<td>98,304</td>
<td>1,572,864</td>
<td>1,572,864</td>
<td>20132.7</td>
<td></td>
</tr>
<tr>
<td>Zin (TLC2)</td>
<td>217</td>
<td>ASC</td>
<td>Appro</td>
<td>TOSS</td>
<td>IB QDR</td>
<td>2,916</td>
<td>46,656</td>
<td>93,312</td>
<td>961.1</td>
<td></td>
</tr>
<tr>
<td>Jade+Jadeita</td>
<td>45</td>
<td>ASC</td>
<td>Penguin</td>
<td>Om-Pen-Path</td>
<td></td>
<td>2,688</td>
<td>96,768</td>
<td>344,064</td>
<td>3251.4</td>
<td></td>
</tr>
<tr>
<td>Mica</td>
<td></td>
<td>ASC</td>
<td>Penguin</td>
<td>Om-Pen-Path</td>
<td></td>
<td>384</td>
<td>13,824</td>
<td>49,152</td>
<td>444.5</td>
<td></td>
</tr>
<tr>
<td>Shark</td>
<td></td>
<td>ASC</td>
<td>IBM RHEL</td>
<td>IB EDR</td>
<td></td>
<td>36</td>
<td>720</td>
<td>11,520</td>
<td>597.6</td>
<td>676.9</td>
</tr>
<tr>
<td>Max</td>
<td></td>
<td>ASC</td>
<td>Appro</td>
<td>TOSS</td>
<td>IB EDR</td>
<td>324</td>
<td>5,184</td>
<td>82,944</td>
<td>107.8</td>
<td>52.4</td>
</tr>
<tr>
<td>Agate</td>
<td></td>
<td>ASC</td>
<td>Penguin</td>
<td>TOSS</td>
<td>N/A</td>
<td>48</td>
<td>1,728</td>
<td>6,144</td>
<td>58.1</td>
<td></td>
</tr>
<tr>
<td>SCF Totals</td>
<td>Systems 8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Combined Totals</td>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Top 500 performance trends

Projected Performance Development

- Performance
- Lists
- Sum
- #1
- #500

Graph showing trends from 1995 to 2020.
What’s so different about HPC systems?

- Low Latency interconnects
  - InfiniBand, OmniPath
- Many CPU cores
  - Xeon, OpenPower, ARM
- Node local storage
  - NVRAM, NVMe
- Tightly Coupled GPUs,
  - PCI Gen 4, NVLink
- Optimized Communication Libraries
  - MPI, OpenShmem

Overview of the “Summit” system @ ORNL
Similar to the “Sierra” system @ LLNL
Anatomy of an HPC node

GPU Brawn
Training AI algorithms requires processors that perform a mathematical workout. Each of Summit’s 4,614 nodes contains six deep-learning-optimized GPUs packed with more than 21 billion transistors. And because deep learning requires less precision than traditional computing requires, Summit holds the potential for exascale-level performance for AI algorithms that can operate at extreme speed.

Memory Where It Matters
Like real estate, the value of a supercomputer’s memory is closely tied to location. Summit’s sizable local memory, including high-bandwidth memory on each GPU, gives AI researchers a convenient launching point for data-intensive tasks. Minimal data movement means researchers can run deep-learning networks faster and achieve greater accuracy.
Typical HPC Deployment

• Storage far from Compute
• Throughput of PFS much slower
• How to bridge the gap?
LBANN Architecture

- Vision
- Physics
- Biology

LBANN

Elemental (Distributed Linear Algebra)

BLAS (Intra-node)
OpenMP (Intra-node)
MPI (Inter-node)

HPC Hardware
Autoencoders
Single Hidden Layer Autoencoder

ISLVRC 2012 Dataset
256x256x3 = 196608

<table>
<thead>
<tr>
<th>Neurons</th>
<th>Parameters</th>
<th>Weight Matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>50K</td>
<td>9.8 B</td>
<td>73 GB</td>
</tr>
<tr>
<td>100K</td>
<td>19.7 B</td>
<td>147 GB</td>
</tr>
<tr>
<td>400K</td>
<td>78.6 B</td>
<td>293 GB</td>
</tr>
</tbody>
</table>

Figure 3: Reconstructed images using a single-layer autoencoder with X neurons after 100 epochs.
Model Parallelism

• Each layer of model is distributed across nodes
  • Distributed matrix library (Elemental) provides dense matrix operations
• Input data is staged into node-local NVRAM
  • Each node stages a separate mini-batch
Data Parallelism
Example Model - MNIST

```yaml
model {
  name: "sequential_model"
  data_layout: "data_parallel"
  mini_batch_size: 64
  block_size: 256
  num_epochs: 20
  num_parallel_readers: 0
  procs_per_model: 0

  # Objective function#
  objective_function {
    cross_entropy {}
    l2_weight_regularization {
      scale_factor: 1e-4
    }
  }

  # Metrics#
  metric {
    categorical_accuracy {}
  }

  callback {
    summary {
      dir: ".";
      mat_interval: 25
    }
  }
}

# Layers#
layer {
  name: "data"
  children: "conv1 target"
  data_layout: "data_parallel"
  input {
    io_buffer: "partitioned"
  }
}
layer {
  name: "conv1"
  data_layout: "data_parallel"
  convolution {
    num_dims: 2
    num_output_channels: 20
    conv_dims_i: 5
    conv_pads_i: 0
    conv_strides_i: 1
    has_bias: true
  }
}
layer {
  name: "pool1"
  data_layout: "data_parallel"
  pooling {
    num_dims: 2
    pool_dims_i: 2
    pool_pads_i: 0
    pool_strides_i: 2
    pool_mode: "max"
  }
}
```
Example Data Reader

```yaml
data_reader {
  reader {
    name: "mnist"
    role: "train"
    shuffle: true
    data_filedir: "/p/lscratchf/brainusr/datasets/MNIST"
    data_filename: "train-images-idx3-ubyte"
    label_filename: "train-labels-idx1-ubyte"
    validation_percent: 0.1
    absolute_sample_count: 0
    percent_of_data_to_use: 1.0
    image_preprocessor {
      normalizer {
        scale: true
        subtract_mean: false
        unit_variance: false
        z_score: false
      }
      augmenter {
        horizontal_flip: false
        vertical_flip: false
        rotation: 0
        horizontal_shift: 0
        vertical_shift: 0
        shear_range: 0
      }
      noiser {
        disable: true
        factor: 0.0
      }
    }
  }
  reader {
    name: "mnist"
    role: "test"
    shuffle: true
    data_filedir: "/p/lscratchf/brainusr/datasets/MNIST"
    data_filename: "t10k-images-idx3-ubyte"
    label_filename: "t10k-labels-idx1-ubyte"
    absolute_sample_count: 0
    percent_of_data_to_use: 1.0
    image_preprocessor {
      normalizer {
        scale: true
        subtract_mean: false
        unit_variance: false
        z_score: false
      }
      augmenter {
        horizontal_flip: false
        vertical_flip: false
        rotation: 0
        horizontal_shift: 0
        vertical_shift: 0
        shear_range: 0
      }
      noiser {
        disable: true
        factor: 0.0
      }
    }
  }
}
```
Sample Run and Output

1. Allocate nodes:
$ salloc -N1 -t 60

2. Choose your model, data, and optimizer
$ srun -n12 build/gnu.catalyst.llnl.gov/install/bin/lbann
   --model=model_zoo/models/lenet_mnist/model_lenet_mnist.prototext
   --reader=model_zoo/data_readers/data_reader_mnist.prototext
   --optimizer=model_zoo/optimizers/opt_adagrad.prototext
   --num_epochs=5

3. Wait for output!

--------------------------------------------------------------------------------
   global MB = [  64/  64/  64] global last MB = [  48 /  48 /  16 ]
   local MB = [  64/  64/  64] local last MB = [  48+0/  48+0/  16+0]
--------------------------------------------------------------------------------
Model 0 training epoch 4 objective function : 0.0471567
Model 0 training epoch 4 categorical accuracy : 99.6241%
Model 0 training epoch 4 run time : 7.64182s
Model 0 training epoch 4 mini-batch time statistics : 0.00901458s mean, 0.0212693s max, 0.0078979s min, 0.000458463s stdev
Model 0 validation objective function : 0.0670221
Model 0 validation categorical accuracy : 98.9%
Model 0 validation run time : 0.25341s
Model 0 validation mini-batch time statistics : 0.00269454s mean, 0.00285273s max, 0.0020936s min, 6.65695e-05s stdev
Model 0 test objective function : 0.0600125
Model 0 test categorical accuracy : 99.02%
Model 0 test run time : 0.421912s
Model 0 test mini-batch time statistics : 0.00268631s mean, 0.00278771s max, 0.00131827s min, 0.00011085s stdev
Experimental Setup

- Catalyst @ LLNL
- 324 Nodes (~150 TFlops)
- 24 Xeon X5660 cores
- 128 GB DRAM
- 800 GB NVRAM
- 24-32 GB/s Lustre PFS
Autoencoder Performance

- ILSVRC 2012 Dataset
- Varying number of neurons in the Hidden layer (10K - 400K)
- 100 epochs
- MiniBatch Size = 40
- Learning Rate = 0.0001

Figure 4: Reconstruction error of a single-layer autoencoder with ImageNet.
Model Parallelism - Strong Scaling

- Fixed problem size, increasing number of nodes
- Scales well till 32 nodes
- I/O dominates at 64-128 nodes

Figure 5: Strong scaling 50K neuron hidden layer: per mini-batch training time vs. number of nodes.
Model Parallelism - Weak Scaling

- Problem size increases with resources
- Neuron count doubles as number of nodes increases

Ideal scaling = constant time
Actual = up to 13.3% difference
Data Parallelism - mini-batch size

- Larger mini-batches
  - More data processed in parallel
  - More efficient distributed matrix multiplication
  - Less synchronization overhead

Figure 7: Strong scaling mini-batch size to exploit data parallelism (50K neuron hidden layer).
Data Parallelism - Data movement

- Best case for PFS
  - Data cached in DRAM
- Worst case for NVRAM
  - Flush cache before each test
  - Include time to copy and untar data from PFS
- More parallel workers = More throughput
Further optimizations

- Larger block sizes improve efficiency
  - Smallest block size (32) is ~19% slower
  - Benefits stall after block size 256

- 24 physical cores, 48 hyperthreaded cores
  - 18 ranks / node leads to imbalance
  - Runtime is ~19% higher
Conclusion

• LBANN is a DL framework focusing on scale
  • Large and complex models
  • Very large data sets

• Scalable unsupervised learning
  • Single layer autoencoder

• Utilize HPC resources
  • NVRAM for Data staging
  • BLAS for fast linear algebra
  • MPI for inter-node communication
Chainer: a Next-Generation Open Source Framework for Deep Learning

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

Presented by: Sourav Chakraborty
Chainer – a deep learning framework

Chainer provides a set of features required for research and development using deep learning such as designing neural networks, training, and evaluation.
## Features and Characteristics of Chainer

<table>
<thead>
<tr>
<th>Powerful</th>
<th>Versatile</th>
</tr>
</thead>
<tbody>
<tr>
<td>☑ CUDA</td>
<td>Supports GPU calculation using CUDA</td>
</tr>
<tr>
<td>☑ cuDNN</td>
<td>High-speed training/inference by cuDNN</td>
</tr>
<tr>
<td>☑ NCCL</td>
<td>Supports a fast, multi-GPU learning using NCCL</td>
</tr>
<tr>
<td>☑ Convolutional Network</td>
<td>N-dimensional Convolution, Deconvolution, Pooling, BN, etc.</td>
</tr>
<tr>
<td>☑ Recurrent Network</td>
<td>RNN components such as LSTM, Bi-directional LSTM, GRU and Bi-directional GRU</td>
</tr>
<tr>
<td>☑ Many Other Components</td>
<td>Many layer definitions and various loss functions used in neural networks</td>
</tr>
<tr>
<td>☑ Various Optimizers</td>
<td>Various optimizers, e.g., SGD, MomentumSGD, AdaGrad, RMSProp, Adam, etc.</td>
</tr>
<tr>
<td>☑ Define-by-Run</td>
<td>Easy to write a complicated network</td>
</tr>
<tr>
<td>☑ High debuggability</td>
<td>User-friendly error messages. Easy to debug using pure Python debuggers.</td>
</tr>
<tr>
<td>☑ Simple APIs</td>
<td>Well-abstracted common tools for various NN learning, easy to write a set of learning flows</td>
</tr>
</tbody>
</table>
Popularity Growth of Chainer

Reactions on Github and Twitter

- watch
- star
- fork
- Twitter followers (EN) https://twitter.com/chainerofficial360
- JP https://twitter.com/chainerjp

Number of Open issues and Open PRs

- Open Issues
- Open PRs
- Closed Issues from last release
- Merged PRs from last release (stable)
- Merged PRs from last release (master)
Neural network = Computational graph

\[ l = \text{MSE}(\text{matmul}(x, W) + b, y) \]

NN can be interpreted as a computational graph that applies many linear and nonlinear functions to input vectors.
How to handle a computational graph

Static

A definition of computational graph exists apart from code that performs computation according to the definition

Dynamic

The actual code that performs computation is treated as a definition of computational graph
How about Chainer? → Dynamic

Chainer is the first deep-learning framework to adopt “Define-by-Run”*

* autograd adopted Define-by-Run but it was not a framework for deep learning.

- **Define-and-Run (static graph)**
  
  Consists of two steps: first to build a computational graph, then feed data to the computational graph (Caffe, theano, TensorFlow, etc.)

- **Define-by-Run (dynamic graph)**
  
  Describing a forward-pass computation means to construct a computational graph for the backward computation (Chainer, DyNet, PyTorch, etc.)
Define-and-Run and Define-by-Run

**Define-and-Run**

# Building
x = Variable('x')
y = Variable('y')
z = x + 2 * y

# Evaluation
for xi, yi in data:
    eval(z, (xi, yi))

**Define-by-Run**

# Build, evaluate at the same time
for xi, yi in data:
    x = Variable(xi)
y = Variable(yi)
z = x + 2 * y

You can make a branch to change the forward computation depending on the data.
How to write a Convolutional Network

```python
import chainer
import chainer.links as L
import chainer.functions as F

class LeNet5(chainer.Chain):
    def __init__(self):
        super(LeNet5, self).__init__()
        with self.init_scope():
            self.conv1 = L.Convolution2D(1, 6, 5, 1)
            self.conv2 = L.Convolution2D(6, 16, 5, 1)
            self.conv3 = L.Convolution2D(16, 120, 4, 1)
            self.fc4 = L.Linear(None, 84)
            self.fc5 = L.Linear(84, 10)

def __call__(self, x):
    h = F.sigmoid(self.conv1(x))
    h = F.max_pooling_2d(h, 2, 2)
    h = F.sigmoid(self.conv2(h))
    h = F.max_pooling_2d(h, 2, 2)
    h = F.sigmoid(self.conv3(h))
    h = F.sigmoid(self.fc4(h))
    return self.fc5(h)
```

- Start writing a model by inheriting Chain class
- Register parametric layers inside the init_scope
- Write forward computation in __call__ method (no need to write backward computation)
Training models

model = LeNet5()
model = L.Classifier(model)

# Dataset is a list! ([] to access, having __len__)
dataset = [(x1, t1), (x2, t2), ...]

# iterator to return a mini-batch retrieved from dataset
it = iterators.SerialIterator(dataset, batchsize=32)

# Optimization methods (you can easily try various methods by changing SGD to
# MomentumSGD, Adam, RMSprop, AdaGrad, etc.)
opt = optimizers.SGD(lr=0.01)
opt.setup(model)

updater = training.StandardUpdater(it, opt, device=0)  # device=-1 if you use CPU
trainer = training.Trainer(updater, stop_trigger=(100, 'epoch'))
trainer.run()

For more details, refer to official examples: https://github.com/pfnet/chainer/tree/master/examples
Define-by-Run brings flexibility and intuitiveness

“Forward computation” becomes a definition of network

- Depending on data, it is easy to change a network structure
- You can define a network itself by Python code
  - The network structure can be treated as a program instead of data.

For Chainer, the “forward computation” can be written in Python

- Enables you to write a network structure freely using the syntax of Python
- Define-by-Run makes it easy to insert any process like putting a print statement between network computations (In case of define-and-run which compiles a network, this kind of debugging is difficult)
- Easy to reuse code of the same network for other purposes with few changes (e.g. by just adding a conditional branch partially)
- Easy to check intermediate values and the design of the network itself using external debugging tools etc.
Chainer v2.0.1

Significantly reduced memory consumption, organized API in response to the users feedback

Aggressive Buffer Release to reduce the memory consumption during training→

https://chainer.org

CuPy has been released as an independent library. This allows for array operations using GPU via an interface highly compatible with NumPy.

https://cupy.chainer.org
Add-on packages for Chainer

Distribute deep learning, deep reinforcement learning, computer vision

- **ChainerMN (Multi-Node):** additional package for distributed deep learning
  High scalability (100 times faster with 128GPU)

- **ChainerRL:** deep reinforcement learning library
  DQN, DDPG, A3C, ACER, NSQ, PCL, etc. OpenAI Gym support

- **ChainerCV:** provides image recognition algorithms, dataset wrappers
  Faster R-CNN, Single Shot Multibox Detector (SSD), SegNet, etc.
ChainerMN

Chainer + Multi-Node
ChainerMN: Multi-node

Keeping the easy-to-use characteristics of Chainer as is, ChainerMN enables to use multiple nodes which have multiple GPUs easily to make training faster
Distributed deep learning with ChainerMN

100x speed up with 128 GPUs

![Graph showing speedup with increasing number of GPUs](image)
Comparison with other frameworks

ChainerMN is the fastest at the comparison of elapsed time to train ResNet-50 on ImageNet dataset for 100 epochs (May 2017)
We confirmed that if we increase the number of nodes, the almost same accuracy can be achieved.
Scale-out test on Microsoft Azure

ChainerMN on Azure (K80, batchsize=32, InfiniBand)
Easy-to-use API of ChainerMN

You can start using ChainerMN just by wrapping one line!

```python
optimizer = chainer.optimizers.MomentumSGD()

optimizer = chainermn.DistributedOptimizer(
    chainer.optimizers.MomentumSGD())
```
Applications using Chainer
Object Detection

https://www.youtube.com/watch?v=yNc5N1MOOt4
Semantic Segmentation

https://www.youtube.com/watch?v=IGOjchGdVQs