A Flexible Framework for Complex Neural Networks

Shohei Hido  Orion Wolfe
Preferred Networks, Inc.
Simple neural networks: “Just deep” convolutional neural network

AlexNet, Kryzyevsky+, 2012 ImageNet winner

GoogLeNet, Szegedy+, 2014 ImageNet winner

Residual Network, He+, 2015 ImageNet winner
Complex neural networks: Stochastic, densely-connected, hierarchical, recurrent

- Stochastic Residual Net, Huang+, 2016
- FractalNet, Larsson+, 2016
- Dense CNN, Huang+, 2016
- RoR, Zhang+, 2016
- Recurrent NN (LSTM)
Chainer is one of the NVIDIA-supported DL frameworks

- Chainer uses a unique scheme named *Define-by-Run*

Chainer fits very well to complex neural networks
Preferred Networks (PFN)
A startup that applies deep learning to industrial IoT

- Founded: March 2014
- Headquarter: Tokyo, Japan
- US subsidiary: SF bay area, California
- Company size: 50 engineers & researchers
- Investors: Toyota, FANUC, NTT
What we do with Chainer: Partnering with world-leading companies

- Applying deep learning to industrial problems with real-world data
  - Specific requirements, modification to algorithms, many trial-and-errors
  - Completely different from making general-purpose recognition systems
- A proof: PFN won the 2nd prize @ Amazon Picking Challenge 2016

Toyota    NTT    FANUC
Panasonic  Cisco  NVIDIA
Two types of background behind DL frameworks

<table>
<thead>
<tr>
<th>1. Scalability-oriented</th>
<th>2. Flexibility-oriented</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Use-cases in mind</strong></td>
<td><strong>Use-cases in mind</strong></td>
</tr>
<tr>
<td>– Image/speech recognition system</td>
<td>– New algorithm research</td>
</tr>
<tr>
<td>– Fast DL as a service in cloud</td>
<td>– R&amp;D projects for AI products</td>
</tr>
<tr>
<td><strong>Problem type</strong></td>
<td><strong>Problem type</strong></td>
</tr>
<tr>
<td>– A few general applications</td>
<td>– Various specific applications</td>
</tr>
<tr>
<td>– 10+ million training samples</td>
<td>– 10+ k training samples</td>
</tr>
<tr>
<td>– 10+ nodes cluster w/ fast network</td>
<td>– 1 node with multiple GPUs</td>
</tr>
<tr>
<td><strong>Possible bottleneck</strong></td>
<td><strong>Possible bottleneck</strong></td>
</tr>
<tr>
<td>– Tuning of well-known algorithms</td>
<td>– Trial-and-error in prototyping</td>
</tr>
<tr>
<td>– Distributed computation for model/data-parallel training</td>
<td>– Debugging, profiling &amp; refactoring</td>
</tr>
<tr>
<td></td>
<td>– (wait time during compilation)</td>
</tr>
</tbody>
</table>
Designed for efficient research & development

- Flexible: new kinds of complex models for various applications
- Intuitive: rapid prototyping and efficient trial-and-error
- Powerful: comparable performance for 1 node & multi-GPUs

TensorFlow    Caffe    Theano    Torch

Scalability-oriented    Flexibility-oriented
Agenda

- Deep learning framework basics
- Basics of Chainer
- Chainer features
- Performance and applications
Neural network and computation

Forward computation

Input

Hidden units

Output

Backward computation (backpropagation)

Object: Tulip
Anomaly score: 0.35
Category: Sports

Image
Sensor
Text
Building and training neural networks: Computational graph construction is the key

1. Construct a computational graph
   - Based on network definition given by users
   - Chains of functions and operations on input variables

2. Compute loss and gradients
   - Forward computation to calculate loss for a minibatch
   - Backpropagation gives gradients to all of parameters

3. Optimize model
   - Update each parameter with the gradient
   - Repeat until convergence

Step 1. is the most important and there are many approaches
Building blocks

- These functionalities are very similar between frameworks
- But the structure, abstraction level, and interface are different
- It comes to the design of domain-specific language for NN

<table>
<thead>
<tr>
<th>Array data structure (vector/matrix/tensor)</th>
<th>Network (computational graph)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operations &amp; functions</td>
<td>Optimizer (SGD/AdaGrad/Adam)</td>
</tr>
</tbody>
</table>
3 types of domain-specific language: Network definition to computational graph construction

- **Text DSL**
  - Ex. Caffe (prototxt)
  - Ex. CNTK (NDL)

- **Symbolic program**
  - Operations on symbols
  - Ex. Theano
  - Ex. TensorFlow

- **Imperative program**
  - Direct computations on raw data arrays
  - Ex. Torch.nn
  - Ex. Chainer

---

%% Definition in text
f: {
  "A": "Variable",
  "B": "Variable",
  "C": ["B", "*", "A"],
  "ret": ["C", "+", 1]
}

# Compile
f = compile("f.txt")
d = f(A=np.ones(10),
     B=np.ones(10) * 2)

# Symbolic definition
A = Variable('A')
B = Variable('B')
C = B * A
D = C + Constant(1)
# Compile
f = compile(D)
d = f(A=np.ones(10),
     B=np.ones(10) * 2)

# Imperative declaration
a = np.ones(10)
b = np.ones(10) * 2
c = b * a
d = c + 1

---

Ex. MXNet

---
# Comparison of DSL type

<table>
<thead>
<tr>
<th>DSL type</th>
<th>Pros.</th>
<th>Cons.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Text DSL</td>
<td>• Human-readable definition</td>
<td>• Users must study the format</td>
</tr>
<tr>
<td></td>
<td>• Non-programmer can easily edit the network</td>
<td>• Format might have to be extended for new algorithms</td>
</tr>
<tr>
<td>Symbolic</td>
<td>• Static analysis at compile</td>
<td>• Users must study special syntax</td>
</tr>
<tr>
<td></td>
<td>• Optimization before training</td>
<td>• May need more efforts to implement new algorithms</td>
</tr>
<tr>
<td></td>
<td>• Easy to parallelize</td>
<td></td>
</tr>
<tr>
<td>Imperative</td>
<td>• Less efforts to learn syntax</td>
<td>• Hard to optimize in advance</td>
</tr>
<tr>
<td></td>
<td>• Easy debugging and profiling</td>
<td>• Less efficient in memory allocation and parallelization</td>
</tr>
<tr>
<td></td>
<td>• Suitable for new algorithms with complex logic</td>
<td></td>
</tr>
</tbody>
</table>

Chainer is at the extreme end of imperative program for high flexibility
Agenda

- Deep learning framework basics
- Basics of Chainer
- Chainer features
- Performance and applications
Chainer as an open-source project

- [https://github.com/pfnet/chainer](https://github.com/pfnet/chainer)
- 78 contributors
- 1,669 stars & 418 forks
- 5,805 commits
- Actively developed
  - 32 releases from v1.0.0 (June 2015) to v1.16.0 (September 2016)
  - Average #commits per week > 70
Chainer software stack

- Chainer is built on top of NumPy and CUDA
- CuPy is also introduced as an equivalent of NumPy on GPU
Graph build scheme (1/2) - Define-and-Run: Most of frameworks use this (Chainer does not)

- **Define**: build a computational graph based on definition
- **Run**: update the model (parameters) using training dataset
Graph build scheme (2/2) - Define-by-Run: Computational graph construction on the fly

- No graph is constructed before training
- Instead, the graph is built at each forward computation
- Computational graph can be modified dynamically for each iteration/sample or depending on some conditions

Define-by-Run

- Model definition
- Training data
- Parameters
- Computational graph
- Dynamic change
- Conditions
- Update
- Gradient function
Define-by-Run example: MLP for MNIST

- Only transformations between units are set before training
  
  \[
  l_1 = \text{Linear}(784, n\_units) \\
  l_2 = \text{Linear}(n\_units, 10)
  \]

- Connection is given as forward computation

  ```python
  def forward(x):
    h1 = \text{ReLU}(l1(x))
    return l2(h1)
  ```
Define-by-Run:
An interpreted language for neural network

- **Advantage**
  - Flexibility for new algorithms with complex components
    - Ex. recurrent, recursive, densely-connected, dilated, etc
  - Intuitive coding with highly imperative network definition
    - Ex. stochastic network of which graph changes for each iteration

- **Current drawbacks**
  - Graph is generated every time also for fixed networks
  - No optimization even for static part of graphs
    - JIT-like analysis and subgraph cache might be useful
Consider an objective (Link.Linear): \( L = f(x \ast w + b) \)

This computes the value of \( L \) in forward computation, and simultaneously builds the following computational graph

The gradient of \( L \) can be computed with respect to any variables by backpropagation

Then the optimizer updates the value of parameters
Code sample (1/3): Convolutional neural network

```python
class AlexNet(Chain):
    def __init__(self):
        super(AlexNet, self).__init__(
            conv1=L.Convolution2D(3, 96, 11, stride=4),
            conv2=L.Convolution2D(96, 256, 5, pad=2),
            conv3=L.Convolution2D(256, 384, 3, pad=1),
            conv4=L.Convolution2D(384, 384, 3, pad=1),
            conv5=L.Convolution2D(384, 256, 3, pad=1),
            fc6=L.Linear(9216, 4096),
            fc7=L.Linear(4096, 4096),
            fc8=L.Linear(4096, 1000),
        )

    def __call__(self, x, t):
        h = F.max_pooling_2d(F.relu(F.local_response_normalization(self.conv1(x))), 3, stride=2)
        h = F.max_pooling_2d(F.relu(F.local_response_normalization(self.conv2(h))), 3, stride=2)
        h = F.relu(self.conv3(h))
        h = F.relu(self.conv4(h))
        h = F.max_pooling_2d(F.relu(self.conv5(h)), 3, stride=2)
        h = F.dropout(F.relu(self.fc6(h)), train=self.train)
        h = F.dropout(F.relu(self.fc7(h)), train=self.train)
        y = self.fc8(h)
        return y

* ImageNet Classification with Deep Convolutional Neural Networks
```
class SimpleRNN(Chain):
    def __init__(self, n_vocab, n_units):
        super(SimpleRNN, self).__init__(
            embed=L.EmbedID(n_vocab, n_units),
            x2h=L.Linear(n_units, n_units),
            h2h=L.Linear(n_units, n_units),
            h2y=L.Linear(n_units, n_vocab),)
        self.h = None

    def __call__(self, x):
        y, h_new = self.fwd_one_step(x, self.h)
        self.h = h_new
        return y

    def fwd_one_step(self, x, h):
        x = F.tanh(self.embed(x))
        if h is None:
            h = F.tanh(self.x2h(x))
        else:
            h = F.tanh(self.x2h(x) + self.h2h(h))
        y = F.softmax(self.h2y(h))
        return y, h

# Truncated BPTT (length=3)
for i in range(0, datasize, batchsize):
    ... accum_loss += model(x, t)
    if i % bptt_length == 0:
        model.zerograds()
        accum_loss.backward()
        accum_loss.unchain_backward()
        optimizer.update()
A variant of Residual Net that skips connections stochastically

- Outperformed the original Residual Net (ImageNet 2015 winner, MSR)
- Stochastic skip: \( H_\ell = \begin{cases} 
\text{ReLU}(f_\ell(H_{\ell-1}) + \text{id}(H_{\ell-1})) \\
\text{ReLU}(\text{id}(H_{\ell-1})) 
\end{cases} \)

w/ survival probability: \( p_\ell = \Pr(b_\ell = 1) \)

# Mock code in Chainer

class StochasticResNet(Chain):
    def __init__(self, prob, size, ...):
        super(StochasticResNet, size, ...).__init__(
            ## Define f[i] as same for Residual Net
            self.p = prob # Survival probabilities
        )

    def __call__(self, h):
        for i in range(self.size):
            b = numpy.random.binomial(1, self.p[i])
            c = self.f[i](h) + h if b == 1 else h
            h = F.relu(c)
        return h

Taken from http://arxiv.org/abs/1603.09382v2
G. Huang et al.
Agenda

- Deep learning framework basics
- Basics of Chainer
- Chainer features
- Performance and applications
Easy to debug during forward computation

# MNIST input has 784 dimensions, not 748

```python
def __init__(self):
    super(MLP3Wrong, self).__init__(
        l1=L.Linear(748, 100),
        l2=L.Linear(100, 100),
        l3=L.Linear(100, 10)
    )
```

Error stack trace in IPython
(Will be more clean soon)

Where the error actually happened in forward computation

```python
<ipython-input-28-b52a3b9f26b2> in __call__(self, x)
    10 11  def __call__(self, x):
    --> 12      h1 = F.tanh(self.l1(x))
    13  h2 = F.tanh(self.l2(x))
    14  y = self.l3(h3)
```

Type is checked at the numpy array level

```
TypeError: Invalid operation is performed in: LinearFunction (Forward)
Expect: prod(in_types[0].shape[1:]) == in_types[1].shape[1]
Actual: 784 != 748
```
**CuPy**: (partially-)NumPy-compatible GPU library

- **Motivation**: NumPy + CUDA = CuPy
  - NumPy is the standard library in Python for numerical computation
  - Unfortunately NumPy does NOT work with CUDA, instead CuPy does it

- **Ex. CPU/GPU-agnostic logsumexp function**
  ```python
def logsumexp(x, axis=None):
    xp = cuda.get_array_module(x)  # Get CuPy or NumPy
    x_max = x.max(axis)
    exp_sum = xp.exp(x - x_max).sum(axis)
    return x_max + xp.log(exp_sum)
  ```

- **CuPy supports**:
  - NVIDIA cuBLAS and cuDNN for faster computation
  - User-defined functions and CUDA kernels
  - Array indexing, slicing, transpose, and reshape
  - All dtypes, broadcasting, memory pool, etc
Use CuPy just like NumPy

- **Conversion between** `numpy.ndarray` and `cupy.ndarray`
  ```python
  w_c = cupy.asarray(numpy.ones(10))  # cupy.ndarray
  w_n = cupy.asnumpy(cupy.ones(10))   # numpy.ndarray
  ```

- **Speed-up by CuPy over NumPy: 5 to 25 times**
  ```python
  def test(xp):
      a = xp.arange(1000000).reshape(1000, -1)
      return a.T * 2
  
  for i in range(1000):
      test(numpy)
  
  for i in range(1000):
      test(cupy)
  ```

<table>
<thead>
<tr>
<th></th>
<th>msec</th>
<th>speed up</th>
</tr>
</thead>
<tbody>
<tr>
<td>NumPy</td>
<td>2,929</td>
<td>1.0</td>
</tr>
<tr>
<td>CuPy</td>
<td>585</td>
<td>5.0</td>
</tr>
<tr>
<td>CuPy + Memory Pool</td>
<td>123</td>
<td>23.8</td>
</tr>
</tbody>
</table>

Intel Core i7-4790 @3.60GHz, 32GB, GeForce GTX 970
Training loop abstraction (from v1.11.0): Manual training loop is not necessary

- Dataset & Iterator: dataset manipulation over iterations
- Trainer & Updater: execute training loops w/ settings
- Extension: do something else during training loops
  - Ex. dump_graph(), snapshot, LogReport, ProgressBar,

```
$ python train_mnist.py --gpu=0
GPU: 0
# unit: 1000
# Minibatch-size: 100
# epoch: 20

epoch | main/loss | validation/main/loss | main/accuracy | validation/main/accuracy
1     | 0.18805   | 0.0980894           | 0.942934     | 0.9673
2     | 0.0730052 | 0.0749994           | 0.976849     | 0.9761

progress: [############################] 14.17%
this epoch [###############################] 83.33%
1700 iter, 2 epoch / 20 epochs
174.79 iters/sec. Estimated time to finish: 0:00:58.926501.
```
Miscellaneous

- Other features
  - Install with pip in one line: `$ pip install chainer`
  - Multi-GPU support by explicitly selecting the ID to use
  - Pre-trained Caffe model import from Model Zoo
  - Model serialization & save & load: HDF5 or NumPy npz
  - Automatic initialization of layer input size when specified as `None`
Agenda

- Deep learning framework basics
- Basics of Chainer
- Chainer features
- Performance and applications
Benchmark results (CNN): Chainer shows comparable performance

- Forward computation is almost the same with Torch/TF
- Chainer’s backward computation has been accelerated from v1.7 to v1.15 to achieve comparable performance

Chainer v1.15* shows our unofficial result measured independently on September 2016
Other results are measured at [https://github.com/soumith/convnet-benchmarks](https://github.com/soumith/convnet-benchmarks) by April 2016
All frameworks except Caffe (native) used cuDNN
Chainer in industry: From demonstrations to commercialization

- Chainer-based collaborations are on-going with partners
  - computer vision, deep reinforcement learning, etc...
- FANUC will commercialize the outcome by the end of 2016
Recent achievement: PFN used Chainer at Amazon Picking Challenge 2016

- Chainer for image segmentation and object recognition
- PFN team won the 2nd prize for picking and 4th for stow tasks
  - Compete with other 15 established robotics labs and companies
  - Only 3 months to build the vision system and robot hand from scratch

<table>
<thead>
<tr>
<th>Stow task</th>
<th>Pick task</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. 214: Delft</td>
<td>1. 105: Delft</td>
</tr>
<tr>
<td>2. 186: NimbRo</td>
<td>2. 105: PFN</td>
</tr>
<tr>
<td>3. 164: MIT</td>
<td>3. 97: NimbRo</td>
</tr>
<tr>
<td>4. 161: PFN</td>
<td>4. 67: MIT</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>16. ARCV</td>
<td>16. Rutgers</td>
</tr>
</tbody>
</table>
Researcher’s perspective on Chainer: Easy to use and extend

- Fast Learning curve
  - Concise object model, modular libraries
- Straightforward GPU support
  - Numpy and CuPy interchangeable
  - Support for multiple GPUs
- Easily customizable
- Easy to debug / fix
- Latest deep learning algorithms
Researcher’s perspective on Chainer: Example case study

- Solving for systems of equations for a physical, thermodynamics system

- Several methods can be possible: 
  - Physical modeling
  - Empirical modeling / other machine learning techniques
  - Deep learning

- Prior knowledge about monotonic and coupling properties can be more easily incorporated into deep learning using Chainer

- In this situation, the problem was easily set-up using Chainer and the model was both expressive and concise

\[ x = f(p, u, T, c, v) \]
Summary

- Chainer is a Python-based deep learning framework with dynamic network construction scheme and CuPy.
- It is designed for efficient research and prototyping while keeping comparable performance and actively improving.
- Github: [https://github.com/pfnet/chainer](https://github.com/pfnet/chainer)

Your contributions will be appreciated & we are hiring!