Deep Learning Frameworks: CNTK & PyTorch

CSE 5194: Intro to High-Performance Deep Learning (Au ‘18)
Presented By: Subhashis Hazarika
Overview

- Microsoft Cognitive Toolkit (CNTK)
- Benefits & Unique Features
- Example
- Parallel Training
  - 1-bit Stochastic Gradient Descent[1]
  - Other in-house strategies
- Conclusion

- PyTorch (Facebook)
- Special Features
- Computational Graphs (dynamic)
- Automatic Differentiation[2]
- Conclusion

Microsoft Cognitive Toolkit (CNTK)

- Microsoft’s open-source deep-learning toolkit
  - [https://github.com/Microsoft/CNTK](https://github.com/Microsoft/CNTK)
  - Created MS speech researchers (Dong Yu et al.) in 2012, “Computational Network Toolkit”
  - Open sourced on CodePlex in Apr 2015
  - On GitHub since Jan 2016 under MIT license, and renamed to “Cognitive Toolkit”
  - Community contributions from MIT, Stanford, Nvidia and many others
Microsoft Cognitive Toolkit (CNTK)

• Runs over 80% of Microsoft Internal deep-learning workload
• 70-90% Microsoft employees use CNTK
• Some popular products:
  - Cortana
  - Bing
  - Bing ads
  - Microsoft HoloLens
  - Skype
  - Microsoft Research
  - Xbox
• 1st-class on Linux, Windows, and docker support
• New features in the latest version:
  • Keras backend ONNX, Java, Spark support
  • State-of-the-art model compression
Microsoft Cognitive Toolkit (CNTK)

Figure 1. Historical progress of speech recognition word error rate on more and more difficult tasks. The latest system for the switchboard task is marked with the green dot.

2015 System

Human Error rate ~ 4%
CNTK: The Fastest Toolkit!!

Benchmarking by **HKBU**, Version 8
http://dlbench.comp.hkbu.edu.hk/
Single Tesla K80 GPU, CUDA: 8.0 CUDNN: v5.1

<table>
<thead>
<tr>
<th>Model</th>
<th>Caffe</th>
<th>Cognitive Toolkit</th>
<th>MxNet</th>
<th>TensorFlow</th>
<th>Torch</th>
</tr>
</thead>
<tbody>
<tr>
<td>FCN5 (1024)</td>
<td>55.329ms</td>
<td><strong>51.038ms</strong></td>
<td>60.448ms</td>
<td>62.044ms</td>
<td>52.154ms</td>
</tr>
<tr>
<td>AlexNet (256)</td>
<td>36.815ms</td>
<td><strong>27.215ms</strong></td>
<td>28.994ms</td>
<td>103.960ms</td>
<td>37.462ms</td>
</tr>
<tr>
<td>ResNet (32)</td>
<td>143.987ms</td>
<td><strong>81.470ms</strong></td>
<td>84.545ms</td>
<td>181.404ms</td>
<td>90.935ms</td>
</tr>
<tr>
<td>LSTM (256)</td>
<td>-</td>
<td><strong>43.581ms</strong></td>
<td>288.142ms</td>
<td>-</td>
<td>1130.606ms</td>
</tr>
<tr>
<td>(v7 benchmark)</td>
<td></td>
<td><strong>(44.917ms)</strong></td>
<td>(284.898ms)</td>
<td>(223.547ms)</td>
<td>(906.958ms)</td>
</tr>
</tbody>
</table>

**Caffe**: 1.0rc5(39f28e4)
**CNTK**: 2.0 Beta10(1ae666d)
**MXNet**: 0.93(32dc3a2)
**TensorFlow**: 1.0(4ac9c09)
**Torch**: 7(748f5e3)
Microsoft Cognitive Toolkit (CNTK)

- CNTK is “production-ready” : State-of-the-art accuracy and efficiency and scales to multi-GPU/multi-server.

[note: December 2015]
CNTK: Other Benefits

• Accuracy
  • Verified training scripts for common networks (AlexNet, ResNet, Inception V3, Faster RCNN, etc.)

• Python and C++ API
  • Mostly implemented in C++ (train and test)
  • Low level + high level Python API

• Extensibility
  • User functions and learners in Python or C++

• Readers
  • Distributed, highly efficient built-in data readers
CNTK: Other Benefits

• Keras interoperability
  • Switching your backend to CNTK and your LSTM will be 2-3x faster immediately

• Binary evaluation
  • 10x speed-up in model execution

• Internal == External
CNTK: Basic Building Block

- CNTK expresses (nearly) arbitrary neural networks by composing simple building blocks into complex computational networks, supporting relevant network types and applications.
CNTK: Basic Building Block

- **Example:** 2-hidden layer feed-forward NN

\[
\begin{align*}
    h_1 &= \sigma(W_1 x + b_1) \\
    h_2 &= \sigma(W_2 h_1 + b_2) \\
    P &= \text{softmax}(W_{out} h_2 + b_{out})
\end{align*}
\]

with input \( x \in \mathbb{R}^M \) and one-hot label \( y \in \mathbb{R}^I \) and cross-entropy training criterion

\[
\begin{align*}
    ce &= y^T \log P \\
    \sum_{\text{corpus}} ce &= \text{max}
\end{align*}
\]

\[
\begin{align*}
    h_1 &= \text{sigmoid} \ (x @ w_1 + b_1) \\
    h_2 &= \text{sigmoid} \ (h_1 @ w_2 + b_2) \\
    P &= \text{softmax} \ (h_2 @ w_{out} + b_{out})
\end{align*}
\]

\[
\begin{align*}
    ce &= \text{cross_entropy} \ (P, y)
\end{align*}
\]
CNTK: Basic Building Block

\[
\begin{align*}
\text{h1} &= \text{sigmoid}\ (x \ @ \ w1 \ + \ b1) \\
\text{h2} &= \text{sigmoid}\ (\text{h1} \ @ \ w2 \ + \ b2) \\
\text{P} &= \text{softmax}\ (\text{h2} \ @ \ wout + \ bout) \\
\text{ce} &= \text{cross_entropy}\ (\text{P}, \ y)
\end{align*}
\]
CNTK: Basic Building Block

- Nodes: functions (primitives)
  - Can be composed into reusable composites
- Edges: values
  - Include tensors, sparse
- Automatic differentiation
  - $\frac{\partial F}{\partial \text{in}} = \frac{\partial F}{\partial \text{out}} \cdot \frac{\partial \text{out}}{\partial \text{in}}$
- Deferred computation
- Editable, clonable
- LEGO-like composability allows CNTK to support wide range of networks and applications.
CNTK: Workflow

- Prepare data
- Configure reader, network, learner (Python)
- Train:
  
  python my_cntk_script.py
from cntk import *

# reader
def create_reader(path, is_training):
    ...

# network
def create_model_function():
    ...
def create_criterion_function(model):
    ...

# trainer (and evaluator)
def train(reader, model):
    ...
def evaluate(reader, model):
    ...

# main function
model = create_model_function()
reader = create_reader(..., is_training=True)
train(reader, model)
reader = create_reader(..., is_training=False)
evaluate(reader, model)
CNTK: Distributed Training Workflow

• Prepare data
• Configure reader, network, learner (Python)
• Train: mpiexec --np 16 --hosts server1,server2,server3,server4 python my_cntk_script.py
CNTK: Unique Features

• Symbolic loops over sequences data

• Batch-Scheduling of Variable-Length Sequences

• Unique parallel training algorithms (1-bit SGD, Block Momentum)
CNTK: Symbolic loops over sequences data

• Extend our example to recurrent network (RNN)

\[
\begin{align*}
    h_1 &= \sigma(W_1 x + b_1) \\
    h_2 &= \sigma(W_2 h_1 + b_2) \\
    P &= \text{softmax}(W_{out} h_2 + b_{out}) \\
    ce &= L^T \log P \\
    \sum_{\text{corpus}} ce &= \text{max}
\end{align*}
\]
CNTK: Symbolic loops over sequences data

- Extend our example to recurrent network (RNN)

\[
\begin{align*}
h_1(t) &= \sigma(W_1 x(t) + R_1 h_1(t-1) + b_1) \\
h_2(t) &= \sigma(W_2 h_1(t) + R_2 h_2(t-1) + b_2) \\
P(t) &= \text{softmax}(W_{out} h_2(t) + b_{out}) \\
ce(t) &= L^T(t) \log P(t) \\
\sum_{\text{corpus}} ce(t) &= \text{max}
\end{align*}
\]

\[
\begin{align*}
h1 &= \text{sigmoid}(x @ W1 + \text{past_value}(h1) @ R1 + b1) \\
h2 &= \text{sigmoid}(h1 @ W2 + \text{past_value}(h2) @ R2 + b2) \\
P &= \text{softmax}(h2 @ W_{out} + b_{out}) \\
ce &= \text{cross_entropy}(P, L)
\end{align*}
\]
CNTK: Symbolic loops over sequences data

h1 = sigmoid(x @ w1 + past_value(h1) @ r1 + b1)

h2 = sigmoid(h1 @ w2 + past_value(h2) @ r2 + b2)

P = softmax(h2 @ wout + bout)

ce = cross_entropy(P, L)

- CNTK automatically unrolls cycles at execution time
  - cycles are detected with Tarjan’s algorithm
- Efficient and composable
CNTK: Batch-Scheduling of Variable-Length Sequences

• Minibatches containing sequences of different lengths are automatically packed and padded
CNTK: Batch-Scheduling of Variable-Length Sequences

- Minibatches containing sequences of different lengths are automatically packed and padded
**CNTK**: Batch-Scheduling of Variable-Length Sequences

- Minibatches containing sequences of different lengths are automatically packed and padded

- Fully transparent batching
  - Recurrent $\rightarrow$ CNTK unrolls, handles sequence boundaries
  - Non-recurrent operations $\rightarrow$ parallel
  - Sequence reductions $\rightarrow$ mask
CNTK: Data-parallel training

• Data-parallelism: distribute minibatch over workers, all-reduce partial gradients
CNTK: Data-parallel training

- Data-parallelism: distribute minibatch over workers, all-reduce partial gradients
CNTK: Data-parallel training

• Data-parallelism: distribute minibatch over workers, all-reduce partial gradients
CNTK: Data-parallel training

• But this strategy alone is not good enough
• Most speech network models use dense matrices in their DNNs
• Communicating this dense matrices is a big overhead in distributed execution
• Example: DNN, MB size 1024, 160M model parameters
  • compute per MB $\rightarrow$ 1/7 second
  • communication per MB $\rightarrow$ 1/9 second (640M over 6 GB/s)
  • can’t even parallelize to 2 GPUs: communication cost already dominates!
CNTK: Data-parallel training

• How to reduce communication cost:

• Communicate less each time
  • 1-bit SGD\(^1\)
    • quantize gradients to 1 bit per value
    • trick: carry over quantization error to next minibatch

• Communicate less often
  • Automatic minibatch (MB) sizing\(^2\)
  • Block momentum\(^3\)
    • Combines model averaging with error-residual idea

---

\(^1\) F. Seide, H. Fu, J. Droppo, G. Li, D. Yu: “1-Bit Stochastic Gradient Descent...Distributed Training of Speech DNNs”, Interspeech 2014.


\(^3\) K. Chen, Q. Huo: “Scalable training of deep learning machines by incremental block training...,” ICASSP 2016
CNTK: 1-bit SGD

• Calculation cost
  • Variable cost:
    • Gradient computation (scalable by # of nodes)
  • Fixed cost:
    • Gradient post-processing (accumulation, etc.)
    • Add gradient to model

• Communication cost

• Goal: Reduce #bits exchanged between servers
CNTK: 1-bit SGD

- Design the system to exchange gradients (as opposed to model parameters)
- Quantize those gradients (aggressively) during data exchange
- General back-propagation for mini-batch size $N$

\[
\lambda(t + N) = \lambda(t) + \epsilon(t) \cdot G(t)
\]

\[
G(t) = \sum_{\tau = t}^{t+N-1} \frac{\partial F_\lambda(o(\tau))}{\partial \lambda} \bigg|_{\lambda = \lambda(t)}
\]

- When quantizing a gradient $G(t)$, they keep the quantization error and add it to the respective next minibatch gradient before quantization.

\[
G_{ij\ell}^{quant}(t) = Q(G_{ij\ell}(t) + \Delta_{ij\ell}(t - N))
\]

\[
\Delta_{ij\ell}(t) = G_{ij\ell}(t) - Q^{-1}(G_{ij\ell}^{quant}(t))
\]
CNTK: 1-bit SGD

• This error feedback ensures that all the gradients are eventually added up into the model with close to full accuracy.

• Observed that as long as the error feedback is used, they can quantize all the way to 1 bit at no or nearly no loss of accuracy.

• Empirical evaluation of the approach
local_learner = momentum_sgd(network['output'].parameters,
    lr_schedule, mm_schedule,
    l2_regularization_weight = l2_reg_weight)

learner = data_parallel_distributed_learner(local_learner,
    num_quantization_bits=num_quantization_bits,
    distributed_after=warm_up)

Trainer(network['output'], (network['ce'], network['pe']), learner, progress_printer)
CNTK: Automatic MB Sizing

• Observation: given a learning rate, there is a maximum MB size that ensures model convergence

• Learning rate shrinking during training

• At any learning rate change point:
  • Try a range of minibatch size on a small data block and pick the largest feasible one

• Tested against Caffe2 for ResNet50: After warm up, use large minibatch sizes.
CNTK: Conclusion

• Production ready

• Performance:
  • Speed: faster than others, 5-10x faster on recurrent networks
  • Scalability: few lines of change to scale to thousands of GPUs
  • Built-in readers: efficient distributed readers

• Programmability:
  • Powerful C++ library for enterprise users
  • Intuitive and performant Python APIs
  • C#/.NET/Java inference support
PyTorch

• Python-based scientific computing package targeted at two sets of audiences:
  • A replacement for NumPy to use the power of GPUs
  • A deep learning research platform that provides maximum flexibility and speed

• Why we need PyTorch?
  • More “Pythonic”: Imperative, Flexible, Easy to debug, Intuitive and cleaner code
  • More “Neural Networkic”: Write code as the network works, forward/backward

---

Andrei Karpathy
@karpathy

I've been using PyTorch a few months now and I've never felt better. I have more energy. My skin is clearer. My eyesight has improved.

2:56 PM - May 26, 2017

❤️ 1,527  💬 421 people are talking about this
PyTorch

• Facebook support plans:
  • Caffe2: main focus has been performance and cross-platform deployment
  • PyTorch: main focus has been to facilitate rapid prototyping and research

• Active work is going on (2018) to merge Caffe2 and PyTorch
  • Cross model support
  • Prefer more of PyTorch like abstraction (nn.Module) for the frontend
  • PyTorch 1.0 will be able to support ONNX natively and interface with other framework or accelerated libraries both for ingesting and emitting models
PyTorch: Three levels of Abstraction

• **Tensor:** Like array in Numpy, but runs on GPUs
• **Variable:** Node in computational graph; stores data and gradients
• **Module:** A neural network layer; may store state or learnable weights

• Important components:
  • `torch.nn` – Build & train neural network models
  • `torch.autograd` – Automatic differentiation **
  • `torch.optim` – Optimization algorithms
PyTorch: Tensors and Numpy

- Tensor is very similar to NumPy’s `ndarray`.
- But unlike the latter, **tensors can tap into the resources of a GPU to significantly speed up matrix operations.**

```python
In [3]: arr = np.random.randn((3,5))
In [4]: arr
Out[4]:

array([[-1.00343281, -0.07042071,  0.81870386],
       [-0.86401346, -1.4290267 , -1.12398822],
       [-1.14619856,  0.39963316, -1.11038695],
       [ 0.00215314,  0.68790149, -0.55967659]])

In [5]: tens = torch.from_numpy(arr)
In [6]: tens
Out[6]:

-1.0003  -0.0704   0.8187
-0.8640  -1.4290  -1.1240
-1.1462   0.3996  -1.1104
 0.0022   0.6879  -0.5597
[torch.DoubleTensor of size 4x3]

In [8]: random_tensor = torch.randn((4,3))
In [9]: random_tensor
Out[9]:

1.0070  -0.6404   1.2707
-0.7767   0.1075   0.4539
-0.1782  -0.0091  -1.0463
 0.4164  -1.1172  -0.2888
[torch.FloatTensor of size 4x3]
PyTorch: Concept of Computation Graphs

\[
\begin{align*}
  b &= w_1 \times a \\
  c &= w_2 \times a \\
  d &= (w_3 \times b) + (w_4 \times c) \\
  L &= f(d)
\end{align*}
\]
PyTorch: Concept of Computation Graphs

• The computation graph is simply a data structure that allows you to efficiently apply the chain rule to compute gradients for all of your parameters.
PyTorch: Variables and Autograd

- PyTorch accomplishes what we described using the *Autograd* package.
- Variables:
  - A thin wrapper class on top of Tensor
  - *Variables* are specifically tailored to hold values which change during training of a neural network, i.e. the learnable parameters of our network
  - Tensors on the other hand are used to store values that are not to be learned
  - Variables retain the history of the data they hold
  - Allows to perform Automatic differentiation on Variables. (can be accessed calling \texttt{.grad})
  - Turn on/off the gradient computation with “\texttt{requires_grad}” flag.
PyTorch: Variables and Autograd

• PyTorch abstracts the need to write two separate functions (for forward, and for backward pass), into two member of functions of a single class called `torch.autograd.Function`.

[Diagram showing variables and functions with data, grad, grad_fn, forward(), and backward().]
PyTorch: Variables and Autograd

```
In [1]:
``` from torch import FloatTensor
from torch.autograd import Variable

# Define the leaf nodes
a = Variable(FloatTensor([4]))
weights = [Variable(FloatTensor([i]), requires_grad=True) for i in (2, 5, 9, 7)]

# unpack the weights for nicer assignment
w1, w2, w3, w4 = weights

b = w1 * a

```
In [8]:
``` print(a)
print(w1)
print(b)
print(d)

tensor([4.])
tensor([2.], requires_grad=True)
tensor([9.], grad_fn=<ThMulBackward>)
tensor([212.], grad_fn=<ThAddBackward>)

In [9]:
``` L.backward()
```

In [10]:
``` for index, weight in enumerate(weights, start=1):
    gradient, _ = weight.grad.data
    print(f"Gradient of L w.r.t to w{index}: {gradient}"

Gradient of L w.r.t to w1: -36.0
Gradient of L w.r.t to w2: -28.0
Gradient of L w.r.t to w3: -8.0
Gradient of L w.r.t to w4: -20.0
```
PyTorch: Variables and Autograd

- Dynamic Computational Graphs
  - A dynamic framework defines the function to be differentiated simply by running the desired computation, as opposed to specifying a static graph structure which is differentiated symbolically ahead of time and then run many times.

- When you call `backward()`, as the gradients are computed, these buffers are essentially freed, and the graph is destroyed

- If you call `forward` again, an entirely new graph is generated with new memory allocated to it

- `requires_grad` and `volatile` flags

- hooks to inspect gradient updates
  ```python
  x.register_hook(lambda grad: print(grad))
  ```
PyTorch: nn Module

- Higher-level wrapper for working with neural networks

```python
import torch
from torch.autograd import Variable

N, D_in, H, D_out = 64, 1000, 100, 10
x = Variable(torch.randn(N, D_in))
y = Variable(torch.randn(N, D_out), requires_grad=False)

model = torch.nn.Sequential(
    torch.nn.Linear(D_in, H),
    torch.nn.ReLU(),
    torch.nn.Linear(H, D_out))
loss_fn = torch.nn.MSELoss(size_average=False)

learning_rate = 1e-4
for t in range(500):
    y_pred = model(x)
    loss = loss_fn(y_pred, y)

    model.zero_grad()
    loss.backward()

    for param in model.parameters():
        param.data -= learning_rate * param.grad.data
```
PyTorch: Conclusion

• Good framework for quick prototyping AI research ideas
• Easy debugging
• Still in heavy development phase
• Lots of changes coming this year!
Thank You!