Standization Benchmarking, OpenAI, ONNX
Topics

1. Standardization Benchmarking
   a. DLBench http://dlbench.comp.hkbu.edu.hk/
   b. Convnet https://github.com/soumith/convnet-benchmarks
2. Open Neural Network Exchange (ONNX) https://github.com/onnx/onnx
3. OpenAI
Benchmarking State-of-the-Art Deep Learning Software Tools

Shaohuai Shi, Qiang Wang, Pengfei Xu, Xiaowen Chu

Department of Computer Science, Hong Kong Baptist University

2016 7th International Conference on Cloud Computing and Big Data (CCBD)
https://arxiv.org/abs/1608.07249

This paper is on version 7: http://dlbench.comp.hkbu.edu.hk/?v=v7
The on-going one is version 8: http://dlbench.comp.hkbu.edu.hk/?v=v8
# Open-source deep learning toolkits

<table>
<thead>
<tr>
<th>Software</th>
<th>Major Version in v7</th>
<th>Major Version in v8</th>
<th>Accelerated Matrix Operation Library</th>
<th>cuDNN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Caffe</td>
<td>1.0.0</td>
<td><strong>1.0rc5</strong> <em>Feb. 20, 2017</em></td>
<td>(latest: 1.0 or Caffe2)</td>
<td></td>
</tr>
<tr>
<td>CNTK</td>
<td>1.72</td>
<td><strong>2.0beta10</strong> <em>Feb. 1, 2017</em></td>
<td>(latest: 2.6)</td>
<td><strong>v5.1</strong> in both v7 and v8</td>
</tr>
<tr>
<td>MXNet</td>
<td>0.7.0</td>
<td><strong>0.9.3</strong> <em>Jan. 22, 2017</em></td>
<td>(latest: 1.3.1)</td>
<td>(latest: v7.4)</td>
</tr>
<tr>
<td>Torch</td>
<td>7</td>
<td>7 <em>Feb. 27, 2017</em></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TensorFlow</td>
<td>0.11</td>
<td><strong>1.0.0</strong> <em>Feb. 15, 2017</em></td>
<td></td>
<td>Eigen</td>
</tr>
</tbody>
</table>

 Eigen
Deep neural Networks

- Fully Connected Network (FCN)
- Convolutional Neural Network (CNN)
  - AlexNet, ResNet
- Recurrent Neural Network (RNN)
  - 2 LSTM (Long Short-Term Memory) layers, with input length of 32

(a) Fully Connected Network

(b) Convolutional Neural Network (AlexNet [16])

(c) Recurrent Neural Network.
LSTM (Long Short-Term Memory)

- LSTM cell

- Applications:
  - Language Modeling, speech recognition, machine translation, image caption generation...

Datasets

- Synthetic data sets (data and labels are generated randomly)

<table>
<thead>
<tr>
<th>Networks</th>
<th>Input</th>
<th>Output</th>
<th>Layers</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>FCN</td>
<td>FCN-S</td>
<td>26752</td>
<td>26752</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>~55 millions</td>
</tr>
<tr>
<td>CNN</td>
<td>AlexNet-S</td>
<td>150528</td>
<td>1000</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>~61 millions</td>
</tr>
<tr>
<td></td>
<td>ResNet-50</td>
<td>150528</td>
<td>1000</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>~3.8 billions</td>
</tr>
</tbody>
</table>

- 150528 = 224 * 224 * 3: imitate the imagenet dataset (also 1000 outputs)
- 26752?
- FCN-S has 4 hidden layers with 2048 nodes each layer
- Batch normalization and dropout are excluded in AlexNet-S
- Cheat? When benchmarking CNN, Tensorflow generates fake mini-batches --- data and labels are generated randomly in each batch, while order frameworks load data from file
Datasets (cont’d)

- Real-world data sets

<table>
<thead>
<tr>
<th>Networks</th>
<th>Input</th>
<th>Output</th>
<th>Layers</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>FCN</td>
<td>FCN-R</td>
<td>784</td>
<td>10</td>
<td>~13 millions</td>
</tr>
<tr>
<td>CNN</td>
<td>AlexNet-R</td>
<td>3072</td>
<td>10</td>
<td>~81 thousands</td>
</tr>
<tr>
<td></td>
<td>ResNet-56</td>
<td>3072</td>
<td>10</td>
<td>~0.85 millions</td>
</tr>
<tr>
<td>RNN</td>
<td>LSTM</td>
<td>10000</td>
<td>10000</td>
<td>~13 millions</td>
</tr>
</tbody>
</table>

- 3072 = 32 * 32 * 3 Cifar10
- PTB (Penn Tree Bank) dataset (929k training words, 73k validation words, 82k test words. 10k words in its vocabulary)
- The local response normalization(LRN) is excluded in AlexNet-R (not supported by CNTK)

Hardware platforms

TABLE 4. THE EXPERIMENTAL SETUP OF HARDWARE.

<table>
<thead>
<tr>
<th>Computational Unit</th>
<th>Cores</th>
<th>Memory</th>
<th>OS</th>
<th>CUDA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel CPU i7-3820</td>
<td>4</td>
<td>64 GB</td>
<td>Ubuntu 14.04</td>
<td>-</td>
</tr>
<tr>
<td>Intel CPU E5-2630x2</td>
<td>16</td>
<td>128 GB</td>
<td>CentOS 7.2</td>
<td>-</td>
</tr>
<tr>
<td>GTX 980</td>
<td>2048</td>
<td>4 GB</td>
<td>Ubuntu 14.04</td>
<td>8.0</td>
</tr>
<tr>
<td>GTX 1080</td>
<td>2560</td>
<td>8 GB</td>
<td>Ubuntu 14.04</td>
<td>8.0</td>
</tr>
<tr>
<td>Telsa K80 GK210</td>
<td>2496</td>
<td>12 GB</td>
<td>CentOS 7.2</td>
<td>8.0</td>
</tr>
</tbody>
</table>

Note: There are 2 GK210 GPUs on a K80 card, but only one GPU is used for testing on single GPU performance comparison.

TABLE 5. THE EXPERIMENTAL HARDWARE SETTING FOR DATA PARALLELIZATION.

<table>
<thead>
<tr>
<th>GPUs</th>
<th>CPU</th>
<th>Memory</th>
<th>PCIe</th>
<th>OS</th>
<th>CUDA</th>
</tr>
</thead>
<tbody>
<tr>
<td>K80x2</td>
<td>E5-2630v4</td>
<td>128GB</td>
<td>PCIe 3.0</td>
<td>CentOS 7.2</td>
<td>8.0</td>
</tr>
</tbody>
</table>

Note: There are two GK210 GPUs on a K80 card, so one K80 card can be used to conduct two-GPU parallelization experiments and two K80 cards are used to conduct four-GPU parallelization experiments.

- Dual E5-2630v3 (16 cores) in v7
- Dual E5-2630v4 (20 cores) in v8
- GPU autoboost feature is disabled to make the results repeatable

- The highest throughput of PCIe 3.0 with Telsa K80 is about 8GB/s
Metrics (Performance)

● Processing time
  ○ The time duration of an iteration that processes a mini-batch of input data
    ▪ Using a range of mini-batch sizes for different types of network
    ● For each mini-batch size, run numerous iterations and take the average

● Convergence rate (the decay of the cost function)
  ○ Data parallelization may affect the convergence rate
    ▪ Compare the convergence rates for the case of multi-GPUs

Synthetic data sets --- evaluate the running time performance

Real-world data sets --- measure the convergence rates
The methods of time measurement for each tool

- **Caffe:**
  - Use “caffe train” command, and then calculate the average time difference between two consecutive iterations

- **CNTK:**
  - Similar to Caffe, but exclude the first epoch which includes the time of disk I/O

- **MXNet:**
  - Use **internal timing function**

- **TensorFlow:**
  - Use timing function **in source scripts**

- **Torch:**
  - The same as TensorFlow
CLAIM!!!

- All the tools provide very flexible programming APIs or configuration options for performance optimization.

“The performance results reported in our experiments are based on our understanding of usage of these tools and are not necessarily the best that can be achieved.”
Scalability

- **Multicore**
  - Does the performance scale very well on many-core CPUs?

- **Multi-GPU**
  - Do all the multi-GPU versions have a considerable higher throughput?
  - Is the convergent speed accelerated?
Implementation of distributed SGD

- Caffe: tree reduction strategy
- CNTK: uses MPI
  - Supports four types of parallel SGD
    - DataParallelSGD, BlockMomentumSGD, ModelAveragingSGD, DataParallelASGD
    - DataParallelSGD: distributes each mini-batch over N workers, run 1-bit SGD
- MXNet
  - Each GPU performs Forward and Backward on a mini-batch size of M/N
  - The gradients are then summed over all GPUs before updating the model (can be performed on both CPU and GPU)
- TensorFlow: places an individual model replica on each GPU
  - There is a duplicated model in the CPU side, which is updated synchronously after all GPUs finish processing a mini-batch of data
- Torch: Similar to MXNet
  - But puts gradients aggregation on the GPU side to reduce the overhead of PCIe data transfer
In general, the performance does not scale very well on many-core CPUs. TensorFlow has a relatively better scalability compared with other tools.
Results (cont’d)

With a single GPU platform, Caffe, CNTK and Torch perform better than MXNet and TensorFlow on FCN.

Figure 9. The performance comparison of FCN-S on GPU platforms.

With a single GPU platform, Caffe, CNTK and Torch perform better than MXNet and TensorFlow on FCN.
Results (cont’d)

With a single GPU platform, MXNet is outstanding in CNNs, especially the larger size of networks, while Caffe and CNTK also achieve good performance on smaller CNNs.

Figure 10. The performance comparison of AlexNet-S on GPU platforms.

With a single GPU platform, MXNet is outstanding in CNNs, especially the larger size of networks, while Caffe and CNTK also achieve good performance on smaller CNNs.
Results (cont’d)

- With a single GPU platform, CNTK obtains excellent time efficiency in RNN of LSTM
- All the multi-GPU versions have considerable higher throughput and the convergent speed is also accelerated.
Results (cont’d)

TABLE 8. COMPARATIVE EXPERIMENT RESULTS BETWEEN SINGLE GPU AND MULTIPLE GPUs (TIME PER MINI-BATCH IN SECOND)

<table>
<thead>
<tr>
<th></th>
<th># of GK210</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>FCN-R</td>
<td>Caffe</td>
<td>0.239</td>
<td>0.131</td>
<td><strong>0.094</strong></td>
</tr>
<tr>
<td></td>
<td>CNTK</td>
<td>0.181</td>
<td>0.111</td>
<td><strong>0.072</strong></td>
</tr>
<tr>
<td></td>
<td>TF</td>
<td>0.208</td>
<td>0.144</td>
<td><strong>0.121</strong></td>
</tr>
<tr>
<td></td>
<td>MXNet</td>
<td>0.184</td>
<td>0.104</td>
<td><strong>0.086</strong></td>
</tr>
<tr>
<td></td>
<td>Torch</td>
<td>0.165</td>
<td><strong>0.110</strong></td>
<td>0.112</td>
</tr>
<tr>
<td>AlexNet-R</td>
<td>Caffe</td>
<td>0.137</td>
<td>0.085</td>
<td><strong>0.047</strong></td>
</tr>
<tr>
<td></td>
<td>CNTK</td>
<td>0.108</td>
<td>0.062</td>
<td><strong>0.037</strong></td>
</tr>
<tr>
<td></td>
<td>TF</td>
<td>0.385</td>
<td>0.332</td>
<td><strong>0.321</strong></td>
</tr>
<tr>
<td></td>
<td>MXNet</td>
<td>0.122</td>
<td>0.070</td>
<td><strong>0.041</strong></td>
</tr>
<tr>
<td></td>
<td>Torch</td>
<td>0.141</td>
<td>0.077</td>
<td><strong>0.046</strong></td>
</tr>
<tr>
<td>RenNet-56</td>
<td>Caffe</td>
<td>0.378</td>
<td>0.254</td>
<td><strong>0.177</strong></td>
</tr>
<tr>
<td></td>
<td>CNTK</td>
<td>0.562</td>
<td>0.351</td>
<td><strong>0.170</strong></td>
</tr>
<tr>
<td></td>
<td>TF</td>
<td>0.523</td>
<td>0.291</td>
<td><strong>0.197</strong></td>
</tr>
<tr>
<td></td>
<td>MXNet</td>
<td>0.270</td>
<td>0.167</td>
<td><strong>0.101</strong></td>
</tr>
<tr>
<td></td>
<td>Torch</td>
<td>0.301</td>
<td>0.182</td>
<td><strong>0.096</strong></td>
</tr>
</tbody>
</table>

Note: The mini-batch sizes for FCN-R, AlexNet-R and ResNet-56 are 4096, 1024 and 128 respectively.

- CNTK performs better scaling on FCN and AlexNet, while MXNet and Torch are outstanding in scaling CNNs.
- GPU platform has a much better efficiency than many-core CPUs.
- Among the three GPU platforms, GTX1080 performs the best in most cases.
- The performance is also affected by the design of configuration files.

http://dlbench.comp.hkbu.edu.hk/?v=v8
Discussion

- 26752
  - In feed-forward phase, matrix multiplications are the most time-consuming operations
    - cuBLAS API: cublasSgemm
      - If we want to multiply matrix A and the transpose of matrix B
        - In-place matrix transpose --- CUBLAS_OP_T
          - It results in up to 3 times slower performance as compared to matrix multiply without transpose
            - E.g. C=A*B’, where size(A)=[1024, 26752] and size(B)=[2048,26752]

- Scaling is heavily influenced by process of gradient aggregation
  - The highest throughput of PCIe3.0 with Telsa K80 is about 8GB/s in our testing platform
    - It means that it takes 0.0256 seconds to transfer the gradient from GPU to CPU
    - Cut down the overhead of transferring data!
convnet-benchmarks

Soumith Chintala (Facebook AI Research)

One of the original authors of PyTorch
One of the authors of the paper: Wasserstein GAN
One of the maintainers of Torch 7

Convnet-benchmarks [https://github.com/soumith/convnet-benchmarks](https://github.com/soumith/convnet-benchmarks)
Libraries

- CuDNNr4-fp16
- CuDNNr4-fp32
- Fbfft (fbDNN)
- CuDNNr2
- CL-nn
- Cudaconvnet2

Benchmarks are done by using Torch bindings

- Nervana-neon-fp16
- Nervana-neon-fp32
- TensorFlow
- Chainer
- Caffe (native)
- Torch-7 (native)
- Caffe-CL Green Tea

Benchmarks are done by using timing function in source scripts or using Caffe bindings
Types of CNN, Hardware and Metrics

- **CNNs**
  - AlexNet OWT (One Weird Trick)
  - Overfeat [fast]
  - OxfordNet [Model-A]
  - GoogleNet V1

- **Hardware and OS**
  - 6-core Intel Core i7-5930 @ 3.50GHz
  - NVIDIA Titan X
  - Ubuntu 14.04 x86_64

- **Metrics**
  - Clock the time for a full forward + backward pass
    - Elapsed time of forward
    - Elapsed time of backward propagation
  - Layer-wise Benchmarking (Last Updated April 2015)
    - Build a CNN with 5 spatial convolution layers
    - In each layer
      - Elapsed time of forward
      - Elapsed time of backward

We cannot measure GPU execution time as CPU because launch of kernels is asynchronous.
Measurement of Single Execution

- Measure the kernel execution time by inserting two events at the start and the end of the launch.

```c
float elapsed=0;
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start, 0);
// launch the kernel
cudaEventRecord(stop, 0);
cudaEventSynchronize (stop);
cudaEventElapsedTime(
    &elapsed, start, stop);
cudaEventDestroy(start);
cudaEventDestroy(stop);
```

- Example (Nervana)

```c
start.record()
flops = 0

#fprop
propagation = None
for layer in layers:
    propagation = layer.fprop(propagation)

    flops += layer.flops
    if print_stats:
        layer.fprop_stats()

end.record()
end.synchronize()
msecs = end.time_since(start)
print("fprop(%2d): %8.3f msecs %8.3f gflops %")
    (loop, msecs, flops / (msecs * 1000000.0))
```
<table>
<thead>
<tr>
<th>Library</th>
<th>Class</th>
<th>Time (ms)</th>
<th>forward (ms)</th>
<th>backward (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CuDNN[R4]-fp16 (Torch)</td>
<td>cudnn.SpatialConvolution</td>
<td>71</td>
<td>25</td>
<td>46</td>
</tr>
<tr>
<td>Nervana-neon-fp16</td>
<td>ConvLayer</td>
<td>78</td>
<td>25</td>
<td>52</td>
</tr>
<tr>
<td>CuDNN[R4]-fp32 (Torch)</td>
<td>cudnn.SpatialConvolution</td>
<td>81</td>
<td>27</td>
<td>53</td>
</tr>
<tr>
<td>TensorFlow</td>
<td>conv2d</td>
<td>81</td>
<td>26</td>
<td>55</td>
</tr>
<tr>
<td>Nervana-neon-fp32</td>
<td>ConvLayer</td>
<td>87</td>
<td>28</td>
<td>58</td>
</tr>
<tr>
<td>fbfft (Torch)</td>
<td>fbnn.SpatialConvolution</td>
<td>104</td>
<td>31</td>
<td>72</td>
</tr>
<tr>
<td>Chainer</td>
<td>Convolution2D</td>
<td>177</td>
<td>40</td>
<td>136</td>
</tr>
<tr>
<td>cudaconvnet2*</td>
<td>ConvLayer</td>
<td>177</td>
<td>42</td>
<td>135</td>
</tr>
<tr>
<td>CuDNN[R2] *</td>
<td>cudnn.SpatialConvolution</td>
<td>231</td>
<td>70</td>
<td>161</td>
</tr>
<tr>
<td>Caffe (native)</td>
<td>ConvolutionLayer</td>
<td>324</td>
<td>121</td>
<td>203</td>
</tr>
<tr>
<td>Torch-7 (native)</td>
<td>SpatialConvolutionMM</td>
<td>342</td>
<td>132</td>
<td>210</td>
</tr>
<tr>
<td>CL-nn (Torch)</td>
<td>SpatialConvolutionMM</td>
<td>963</td>
<td>388</td>
<td>574</td>
</tr>
<tr>
<td>Caffe-CLGreenTea</td>
<td>ConvolutionLayer</td>
<td>1442</td>
<td>210</td>
<td>1232</td>
</tr>
</tbody>
</table>
Columns L1, L2, L3, L4, L5, Total are times in **milliseconds**

<table>
<thead>
<tr>
<th>Original Library</th>
<th>Class/Function Benchmarked</th>
<th>L1</th>
<th>L2</th>
<th>L3</th>
<th>L4</th>
<th>L5</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>fbfft</td>
<td>SpatialConvolutionCuFFT</td>
<td>57</td>
<td>27</td>
<td>6</td>
<td>2</td>
<td>9</td>
<td>101</td>
</tr>
<tr>
<td>cuda-convnet2 *</td>
<td>ConvLayer</td>
<td>36</td>
<td>113</td>
<td>40</td>
<td>4</td>
<td>8</td>
<td>201</td>
</tr>
<tr>
<td>cuda-convnet**</td>
<td>pylearn2.cuda_convnet</td>
<td>38</td>
<td>183</td>
<td>68</td>
<td>7</td>
<td>16</td>
<td>312</td>
</tr>
<tr>
<td>CuDNN R2</td>
<td>cudnn.SpatialConvolution</td>
<td>56</td>
<td>143</td>
<td>53</td>
<td>6</td>
<td>11</td>
<td>269</td>
</tr>
<tr>
<td>Theano</td>
<td>CorrMM</td>
<td>91</td>
<td>143</td>
<td>121</td>
<td>24</td>
<td>28</td>
<td>407</td>
</tr>
<tr>
<td>Caffe</td>
<td>ConvolutionLayer&lt;Dtype&gt;</td>
<td>93</td>
<td>136</td>
<td>116</td>
<td>24</td>
<td>27</td>
<td>396</td>
</tr>
<tr>
<td>Torch-7</td>
<td>nn.SpatialConvolutionMM</td>
<td>94</td>
<td>149</td>
<td>123</td>
<td>24</td>
<td>28</td>
<td>418</td>
</tr>
<tr>
<td>DeepCL</td>
<td>ConvolutionLayer</td>
<td>738</td>
<td>1241</td>
<td>518</td>
<td>47</td>
<td>104</td>
<td>2648</td>
</tr>
<tr>
<td>cherry-picking****</td>
<td><em>best per layer</em></td>
<td>36</td>
<td>27</td>
<td>6</td>
<td>2</td>
<td>8</td>
<td>79</td>
</tr>
</tbody>
</table>

* indicates that the library was tested with Torch bindings of the specific kernels.
** indicates that the library was tested with Pylearn2 bindings.
ONNX

Sarah Bird, Dmytro Dzhulgakov (Facebook)

Dmytro was one of the core developers of the Caffe2 framework

http://learningsys.org/nips17/assets/slides/ONNX-workshop.pdf
https://github.com/onnx/onnx
facebook
ONNX

Sarah Bird, Dmytro Dzhulgakov
Facebook
Deep Learning Frameworks
Tensors and Dynamic neural networks in Python with strong GPU acceleration

Flexible Development
• Research-oriented imperative model
• Python flow-control constructs
• Dynamic graph support with autograd

http://pytorch.org
Released Jan 18th
500,000+ downloads
2700+ community repos
17,200+ user posts
351 contributors
A New Lightweight, Modular, and Scalable Deep Learning Framework

**RUN ANYWHERE, FAST**
Your favorite deep learning technology, now from zero to scale, cloud to mobile.

Train ImageNet in 1 hour

Production Powerhouse
- Scalable from small devices to large GPUs in DC
- Strong distributed training support
- Highly optimized mobile device support
- Based on ahead-of-time static graph – no interpreter needed in prod
Research to Production

Reimplementation takes weeks or months
Merge Frameworks?

- Model transfer is important, but less common
- Difficult to optimize the tools for all cases
- Separate but interoperable tools is more efficient
Shared Model Format

PYTORCH ← ONNX → Caffe2
Deep Learning Frameworks Zoo

- Caffe2
- PyTorch
- TensorFlow
- mxnet
- CNTK
- ...
Open Neural Network Exchange

Shared model and operator representation
From $O(n^2)$ to $O(n)$ pairs

Vendor and numeric libraries
Apple CoreML, Nvidia TensorRT, Intel/Nervana ngraph, Qualcomm SNPE, ...
Standard?

How Standards Proliferate:
(See: A/C chargers, character encodings, instant messaging, etc.)

**SITUATION:**
There are 14 competing standards.

14?! Ridiculous!
We need to develop one universal standard that covers everyone's use cases.

Yeah!

**SOON:**

**SITUATION:**
There are 15 competing standards.
Open community

- Framework agnostic
- GitHub from the beginning
- Close partnerships and OSS contributions
Unframeworks

**Vision: Interoperable Tools**
- Accelerate research to production
- Developers can use the best combination of tools for them
- Enables more people to contribute

**Approach:**
- Split toolchain into smaller components
UNIX philosophy for deep learning frameworks

Build reusable components that work well together (across frameworks)
ONNX high-level IR

- Initial focus on exchange for inference
- **SSA graph structure**, serializable
  - Support for structured control flow
- **Standard operator definitions**
  - Striking balance on granularity
  - Codified semantics in tests/ref
- **Common optimization passes**
Current status

• ONNX IR spec is V1.0
• Good coverage for vision models
• Iterating on:
  • Optimization-friendly RNNs
  • Control Flow
  • More hardware backends
Beyond static graphs: Capturing dynamic behavior
Declarative mode vs Eager mode

Python script

Building IR in Python

Python-independent execution

Framework’s VM

Operator implementations

Execution engine

Regular python extension

Python interpreter Code

Operator implementations

Caffe2

PyTorch
Tracing for static graph

Record which operators were invoked

def foo(x):
    y = x.mm(x)
    print(y) # still works!
    return y + 1

x = torch.Tensor([[1,2],[3,4]])
foo(x)

Enough to cover CNNs and static sections
Tracing for dynamic graphs

def foo(x, w):
    y = torch.zeros(1, 2)
    for t in x:
        y = y.mm(w) + t
    return y

w = torch.Tensor([[0.5, 0.2], [0.1, 0.4]])
x = torch.Tensor([[1, 2], [3, 4], [5, 6]])
foo(x, w)
x2 = torch.Tensor([[7, 8], [9, 10]])
foo(x2, w)

Doesn’t do what you want!
Tracing for dynamic graphs

def foo(x, w):
    y = torch.zeros(1, 2)
    for t in x:
        y = y.mm(w) + t
    return y

w = torch.Tensor([[0.5, 0.2], [0.1, 0.4]])
x = torch.Tensor([[1, 2], [3, 4], [5, 6]])
foo(x, w)
x2 = torch.Tensor([[7, 8], [9, 10]])
foo(x2, w)

Capture control flow from python?
Approaches for dynamic graphs

- Parse or compile Python (tricky)
- Use special primitives (annoying)
  
  ```
  for t in x:
      y = y.mm(w) + t
  ```

  ```
  lib.For(x, y, lambda y, t:
      y.mm(w) + t)
  ```

- Capture common patterns like RNN
- Build DSL for subset of Python
- Make it easy to embed C++ calling back to framework
Putting it together

Capturing dynamic behavior

• Trace static portions
• Minimum rewrites for dynamic parts
• Establish tooling for step-by-step code migration
ONNX is a community project.

https://onnx.ai
https://github.com/onnx
<table>
<thead>
<tr>
<th>Framework / tool</th>
<th>Installation</th>
<th>Exporting to ONNX (frontend)</th>
<th>Importing ONNX models (backend)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Caffe2</td>
<td>part of caffe2 package</td>
<td>Exporting</td>
<td>Importing</td>
</tr>
<tr>
<td>PyTorch</td>
<td>part of pytorch package</td>
<td>Exporting, Extending support</td>
<td>coming soon</td>
</tr>
<tr>
<td>Cognitive Toolkit (CNTK)</td>
<td>built-in</td>
<td>Exporting</td>
<td>Importing</td>
</tr>
<tr>
<td>Apache MXNet</td>
<td>part of mxnet package docs github</td>
<td>Exporting</td>
<td>Importing</td>
</tr>
<tr>
<td>Chainer</td>
<td>chainer/onnx-chainer</td>
<td>Exporting</td>
<td>coming soon</td>
</tr>
<tr>
<td>TensorFlow</td>
<td>onnx/onnx-tensorflow</td>
<td>Exporting</td>
<td>Importing [experimental]</td>
</tr>
<tr>
<td>Apple CoreML</td>
<td>onnx/onnx-coreml and onnx/onnxmltools</td>
<td>Exporting</td>
<td>Importing</td>
</tr>
<tr>
<td>SciKit-Learn</td>
<td>onnx/onnxmltools</td>
<td>Exporting</td>
<td>n/a</td>
</tr>
<tr>
<td>ML.NET</td>
<td>built-in</td>
<td>Exporting</td>
<td>Importing</td>
</tr>
<tr>
<td>Menoh</td>
<td>pfnet-research/menoh</td>
<td>n/a</td>
<td>Importing</td>
</tr>
<tr>
<td>MATLAB</td>
<td>onnx converter on matlab central file exchange</td>
<td>Exporting</td>
<td>Importing</td>
</tr>
</tbody>
</table>
Founders: Elon Musk, Sam Altman

Key person: Ilya Sutskever (University of Toronto -> Stanford University -> DNNResearch (Google Brain) -> OpenAI, co-inventor of AlexNet, AlphaGo and TensorFlow)
Products

- **Gym**
  - A toolkit for developing and comparing reinforcement learning algorithms
  - Gives you access to a standardized set of environments
  - [https://github.com/openai/gym](https://github.com/openai/gym)

- **RoboSumo**
  - Competitive self-play
  - [https://blog.openai.com/competitive-self-play/](https://blog.openai.com/competitive-self-play/)

- **Debate Game**
  - Teaches machines to debate toy problems in front of a human judge

- **OpenAI Five**
  - Learn to play Dota2 against human players at a high skill level entirely through trial-and-error algorithms

- **Dactyl**
  - Train a robot Shadow Hand
  - The robot hand is trained entirely in physically inaccurate simulation
  - [https://blog.openai.com/learning-dexterity/](https://blog.openai.com/learning-dexterity/)
Deep Reinforcement Learning

A Beginner's Guide to Deep Reinforcement Learning

https://skymind.ai/wiki/deep-reinforcement-learning#neural