README.md

Updates

27/11/2017

- Added feature-extraction example
- Re-ran all notebooks on latest VM version

Notes

The notebooks are not specifically written for speed, instead they aim to create an easy comparison between the frameworks. However, any suggestions on improving the training-time are welcome!

The rankings are almost for fun and aren't meant to suggest anything about the overall performance of the framework since they omit important comparisons such as: help&support, custom layers (can you create a capsule net?), data-loaders, debugging, different platform-support, distributed training, and much more! They are mean to show how to create the same networks across different frameworks

For example, create a CNN in Python using Caffe2 and then replicate it in Julia using KNet, also try it in PyTorch,and then create an RNN and replicate it in Tensorflow. You can do some feature-extraction in Chainer, and then replicate that in CNTK.

Notebooks are run on (half) an Nvidia K80 GPU, on Microsoft Azure Deep Learning Virtual Machine, NC6, where frameworks have been updated to the latest version

	NC6
Cores	6 (E5-2690v3)
GPU	1 x K80 GPU (1/2 Physical Card)
Memory	56 GB
Disk	380 GB SSD

Goal

Create a Rosetta Stone of deep-learning frameworks to allow data-scientists to easily leverage their expertise from one framework to another (by translating, rather than learning from scratch). Also, to make the models more transparent to comparisons in terms of training-time and default-options.

A lot of online tutorials use very-low level APIs, which are very verbose, and don't make much sense (given higher-level helpers being available) for most use-cases unless one plans to create new layers. Here we try to apply the highest-level API possible, conditional on being to override conflicting defaults, to allow an easier comparison between frameworks. It will demonstrated that the code structure becomes very similar once higher-level APIs are used and can be roughly represented as:

- Load data; x_train, x_test, y_train, y_test = cifar_for_library(channel_first=?, one_hot=?)
- Generate CNN/RNN symbol (usually no activation on final dense-layer)
- Specify loss (cross-entropy comes bundles with softmax), optimiser and initialise weights + sessions
- Train on mini-batches from train-set using custom iterator (common data-source for all frameworks)
- Predict on fresh mini-batches from test-set
- Evaluate accuracy

Since we are essentially comparing a series of deterministic mathematical operations (albeit with a random initialization), it does not make sense to me to compare the accuracy across frameworks and instead they are reported as **checks we want to match**, to make sure we are comparing the same model architecture.

Results (24 Nov 2017)

Training CNN (VGG-style) on CIFAR-10 - Image Recognition

DL Library	Test Accuracy (%)	Training Time (s)
MXNet	77	145
Caffe2	79	148
Gluon	76	152
Knet(Julia)	78	153
Chainer	79	162
CNTK	78	163
PyTorch	78	169
Tensorflow	78	173
Keras(CNTK)	77	194
Keras(TF)	77	241
Lasagne(Theano)	77	253
Keras(Theano)	78	269

Input for this model is the standard CIFAR-10 dataset containing 50k training images and 10k test images, uniformly split across 10 classes. Each 32 by 32 px image is supplied as a tensor of shape (3, 32, 32) with pixel intensity re-scaled from 0-255 to 0-1. For example: automobile with corresponding y=(0, 1, 0, 0, 0, 0, 0, 0, 0, 0) where labels=[airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck]

Training RNN (GRU) on IMDB - Natural Language Processing (Sentiment Analysis)

DL Library	Test Accuracy (%)	Training Time (s)	Using CuDNN?
MXNet	86	29	Yes
Pytorch	86	31	Yes
Knet(Julia)	85	30	Yes
Tensorflow	86	30	Yes
CNTK	85	32	Yes
Keras(TF)	86	35	Yes
Keras(CNTK)	86	86	No Available

Input for this model is the standard IMDB movie review dataset containing 25k training reviews and 25k test reviews, uniformly split across 2 classes (positive/negative). Reviews are already downloaded as a tensor of word indexes e.g. (If you like adult comedy cartoons, like South Park) is received as (1 2 3 4 5 6 3 7 8). Processing follows Keras approach where start-character is set as 1, out-of-vocab (vocab size of 30k is used) represented as 2 and thus word-index starts from 3. Zero-padded / truncated to fixed axis of 150 words per review.

Where possible I try to use the cudnn-optimised RNN (noted by the CUDNN=True switch), since we have a vanilla RNN that can be easily reduced to the CuDNN level. For example with CNTK we use optimized_rnnstack instead of Recurrence(LSTM()). This is much faster but less flexible and, for example, with CNTK we can no longer use more complicated variants like Layer Normalisation, etc. It appears in PyTorch this is enabled by default. For MXNet I could not find this and instead use the slightly slower Fused RNN. Keras has just very recently received cudnn support, however only for the Tensorflow backend (not CNTK). Tensorflow has many RNN variants (including their own custom kernel) and there is a nice benchmark here, I will try to update the example to use CudnnLSTM instead of the current method.

Note: CNTK supports dynamic axes which means we don't need to pad the input to 150 words and can consume as-is, however since I could not find a way to do this with other frameworks I have fallen back to padding - which is a bit unfair on CNTK and understates its capabilities

The classification model creates an embedding matrix of size (150x125) and then applies 100 gated recurrent units and takes as output the final output (not sequence of outputs and not hidden state). Any suggestions on alterations to this are welcome.

Inference ResNet-50 (Feature Extraction)

DL Library	Images/s GPU	Images/s CPU
Tensorflow	155	11
MXNet(w/mkl)	129	25
MXNet	130	8
PyTorch	130	6
CNTK	117	8
Chainer	107	3
Keras(TF)	98	5
Caffe2	71	6
Keras(CNTK)	46	4
ONNX_Caffe2		
ONNX_MXNet		

A pre-trained ResNet50 model is loaded and chopped just after the avg_pooling at the end (7, 7), which outputs a 2048D dimensional vector. This can be plugged into a softmax layer or another classifier such as a boosted tree to perform transfer learning. Allowing for a warm start; this forward-only pass to the avg_pool layer is timed on both CPU and GPU.

Lessons Learned

CNN

The below offers some insights I gained after trying to match test-accuracy across frameworks and from all the GitHub issues/PRs raised.

1. The above examples (except for Keras), for ease of comparison, try to use the same level of API and so all use the same generator-function. For MXNet and CNTK I have experimented with a higher-level API, where I use the framework's training generator function. The speed improvement is negligible in this example because the whole dataset is loaded as NumPy array in RAM and the only processing done each epoch is a shuffle. I suspect the framework's generators perform the shuffle asynchronously. Curiously, it seems that the frameworks shuffle on a batch-level, rather than on an observation level, and thus ever so slightly decreases the test-accuracy (at least after 10 epochs). For scenarios where we have IO activity and perhaps pre-processing and data-augmentation on the fly, custom generators would have a much bigger impact on performance.

2. Enabling CuDNN's auto-tune/exhaustive search parameter (which selects the most efficient CNN algorithm for images of fixed-size) has a huge performance boost. This had to be manually enabled for Chainer, Caffe2, PyTorch and Theano. It appears CNTK, MXNet and Tensorflow have this enabled by default. Yangqing mentions that the performance boost between cudnnGet (default) and cudnnFind is, however, much smaller on the Titan X GPU; it seems that the K80 + new cudnn makes the problem more prominent in this case. Running cudnnFind for every combination of size in object detection has serious performance regressions, however, so exhaustive_search should be disabled for object detection

3. When using Keras it's important to choose the [NCHW] ordering that matches the back-end framework. CNTK operates with channels first and by mistake I had Keras configured to expect channels last. It then must have changed the order at each batch which degraded performance severely. Generally, [NHWC] is the default for most frameworks (like Tensorflow) and [NCHW] is the optimal format to use when training on NVIDIA GPUs using cuDNN.

4. Tensorflow, PyTorch, Caffe2 and Theano required a boolean supplied to the dropout-layer indicating whether we were training or not (this had a huge impact on test-accuracy, 72 vs 77%). Dropout should not be applied to test in this case.

5. Tensorflow required two more changes: speed was improved a lot by enabling TF_ENABLE_WINOGRAD_NONFUSED and also changing the dimensions supplied to channel first rather than last (data_format='channels_first'). Enabling the WINOGRAD for convolutions also, naturally, improved Keras with TF as a backend

6. Softmax is usually bundled with cross_entropy_loss() for most functions and it's worth checking if you need an activation on your final fully-connected layer to save time applying it twice

7. Kernel initializer for different frameworks can vary (I've found this to have +/- 1% effect on accuracy) and I try to specify xavier/glorot uniform whenever possible/not too verbose

8. Type of momentum implemented for SGD-momentum; I had to turn off unit_gain (which was on by default in CNTK) to match other frameworks' implementations

9. Caffe2 has an extra optimisation for the first layer of a network (no_gradient_to_input=1) that produces a small speed-boost by not computing gradients for input. It's possible that Tensorflow and MXNet already enable this by default. Computing this gradient could be useful for research purposes and for networks like deep-dream

10. Applying the ReLU activation after max-pooling (instead of before) means you perform a calculation after dimensionality-reduction and thus shave off a few seconds. This helped reduce MXNet time by 3 seconds

11. Some **further checks** which may be useful:

- specifying kernel as (3) becomes a symmetric tuple (3, 3) or 1D convolution (3, 1)?
- strides (for max-pooling) are (1, 1) by default or equal to kernel (Keras does this)?
- default padding is usually off (0, 0)/valid but useful to check it's not on/'same'
- is the default activation on a convolutional layer 'None' or 'ReLu' (Lasagne)
- the bias initializer may vary (sometimes no bias is included)
- gradient clipping and treatment of inifinty/NaNs may differ across frameworks
- some frameworks support sparse labels instead of one-hot (which I use if available, e.g. Tensorflow has f.nn.sparse_softmax_cross_entropy_with_logits)

- data-type assumptions may be different - I try to use float32 and int32 for X and y but, for example, torch needs double for y (to be coerced into torch.LongTensor(y).cuda)
- if the framework has a slightly lower-level API make sure during testing you don't compute the gradient by setting something like training=False

12. Installing Caffe2 for python 3.5 proved a bit difficult so I wanted to share the process:

```
# build as root
sudo -s
cd /opt/caffe2
make clean
git pull
git checkout v0.8.1
git submodule update
export CPLUS_INCLUDE_PATH=/anaconda/envs/py35/include/python3.5m
mkdir build
cd build
echo $PATH
# CONFIRM that Anaconda is not in the path
cmake .. -DBLAS=MKL -DPYTHON_INCLUDE_DIR=/anaconda/envs/py35/include/python3.5m -DPYTHON_LIBRARY=/anaconda/envs/py3
make -j$(nproc)
make install
```

RNN

1. There are multiple RNN implementations/kernels available for most frameworks (for example Tensorflow); once reduced down to the cudnnLSTM/GRU level the execution is the fastest, however this implementation is less flexible (e.g. maybe you want layer normalisation) and may become problematic if inference is run on the CPU at a later stage. At the cudDNN level most of the frameworks' runtimes are very similar. This Nvidia blog-post goes through several interesting cuDNN optimisations for recurrent neural nets e.g. fusing - "combining the computation of many small matrices into that of larger ones and streaming the computation whenever possible, the ratio of computation to memory I/O can be increased, which results in better performance on GPU".

Inference

Comming soon