Torchnet: An Open-Source Platform for (Deep) Learning Research

Ronan Collobert LOCRONAN@FB.COM
Laurens van der Maaten LVDMAATEN@FB.COM
Armand Joulin AJOULIN@FB.COM
Facebook AI Research, 1 Hacker Way, Menlo Park CA 94025 / 770 Broadway, New York NY 10003, USA

Abstract

Torch 7 is a scientific computing platform that supports both CPU and GPU computation, has a light-weight wrapper in a simple scripting language, and provides fast implementations of common algebraic operations. It has become one of the main frameworks for research in (deep) machine learning. Torch does, however, not provide abstractions and boilerplate code for machine-learning experiments. As a result, researchers repeatedly re-implement experimentation logics that are not interoperable. We introduce Torchnet: an open-source framework that provides abstractions and boilerplate logic for machine learning. It encourages modular programming and code re-use, which reduces the chance of bugs, and it makes it straightforward to use asynchronous data loading and efficient multi-GPU computations. Torchnet is written in pure Lua, which makes it easy to install on any architecture with a Torch installation. We envision Torchnet to become a platform to which the community contributes via plugins.

1. Introduction

Torch 7 is a versatile library for scientific computing framework that contains efficient low-level implementations of major algebraic operations on both CPU (via OpenMP/SSE) and GPU (via CUDA), coupled with a very lightweight wrapper in the Lua scripting language (Collobert et al., 2011). Torch has a very active developer community, which has developed packages for among others, optimization, manifold learning, metric learning, and neural networks. The Torch neural networks package is currently a popular framework for deep learning because it combines flexibility and computational efficiency.

Proceedings of the 33^{rd} International Conference on Machine Learning, New York, NY, USA, 2016. JMLR: W&CP volume 48.

The Torch neural network package makes it easy to specify models, evaluate the output of these models, and compute the derivatives of the model output with respect to its parameters or input. However, implementing a complete learning experiment still requires a substantial amount of development for which `torch/nn` does not provide support: researchers need to develop an efficient data loader, partition the available data into a training and test set, connect the model (gradient) evaluation with an optimizer of choice, implement performance measures to monitor the training and assess the quality of the final model, and set up logging. The development of such boilerplate code tends to involve much code replication and is error-prone, which may lead to incorrect research outcomes. In particular, the lack of pre-defined abstractions and reference implementations makes it hard for researchers to write code that can easily be adapted and re-used by others.

This paper presents Torchnet: a new open-source framework that facilitates rapid development of (deep) machine-learning experiments. Torchnet provides a collection of key abstractions, boilerplate code, and reference implementations that are aimed at making code both re-usable and efficient. In particular, Torchnet encourages a modular design that clearly separates the dataset, the data loading process, the model, the optimization, and the performance measures. The different components are connected in an `Engine` that implements model training and evaluation. The modular design makes it easy to re-use code and develop a series of experiments: for instance, running the same experiments on a different dataset amounts to plugging in a different dataloader, and changing the evaluation criterion amounts to plugging in a different performance meter. Torchnet does not compromise on efficiency: it provides out-of-the-box support for asynchronous data loading and supports training on multiple GPUs.

We envision Torchnet to become a platform to which the research community can contribute via plugins that implement machine-learning experiments or tools. This will make it easier to verify the details (and correctness) of experimental setups, to replicate results, and to re-use code.

Dataset	Description
`BatchDataset`	Merges samples into batches.
`ConcatDataset`	Concatenates K `Dataset`s into one.
`CoroutineBatchDataset`	`BatchDataset` that provides more control via coroutines.
`IndexedDataset`	Key-value store `Dataset`.
`ListDataset`	Load samples in list via closure.
`ResampleDataset`	Arbitrarily resampling of a `Dataset`.
`ShuffleDataset`	Randomly shuffle samples in `Dataset`.
`SplitDataset`	Splits `dataset` into disjoint sets.
`TableDataset`	Load samples from a Lua table.
`TransformDataset`	Transform samples via closure.

Table 1. Overview of all `Dataset`s implemented in Torchnet.

2. Abstractions

Torchnet implements five main types of abstractions, which draw inspiration from earlier Lush[1] frameworks similar to Torchnet: (1) `Dataset`s, (2) `DatasetIterator`s, (3) `Engine`s, (4) `Meter`s, and (5) `Log`s. The five main abstractions are presented separately below.

2.1. Datasets

The `Dataset` abstraction is an abstraction that provides just two functions: (1) a `size()` function that returns the number of samples in the dataset, and (2) a `get(idx)` function that returns the `idx`-th sample in the dataset. In line with Torchnet's emphasis on modular programming, complex data loaders can be constructed by plugging a dataset into another dataset that performs operations such as dataset concatenation, dataset splitting, batching of data, resampling of data, filtering of data, and sample transformations. An overview of all operations that can be performed on a dataset using Torchnet is provided in Table 1. The main advantage of this modular approach is that it facilitates the construction of complex data loaders in a small number of lines of code: when one wants to train a model on a new dataset, all that needs to implemented is a function that returns the number of samples in that dataset and a function that returns a specific sample. The datasets in Table 1 can subsequently be used to rebalance the classes according to a particular distribution, construct mini-batches for training, split data into training and test data, *etc.* Moreover, the engine that is performing the training or evaluation of the model is mostly agnostic to the exact dataset it is trained on: given data loaders for, say the Imagenet (Deng et al., 2009) and MS COCO (Lin et al., 2014b) datasets, retraining (or testing) an Imagenet convolutional network (He et al., 2016) on the MS COCO dataset amounts to simply plugging the core data loader for the MS COCO dataset into the existing code.

[1]See `http://lush.sourceforge.net/`.

2.2. Dataset Iterators

When performing training or testing, one iterates over all samples in a dataset and performs operations such as parameter updates or accumulation of performance measures (which is what `Engine`s and `Meter`s do, respectively). In its simplest form, such a dataset iterator is a simple for loop that runs from one to the dataset size and calls the `get()` function with loop value as input; the `DatasetIterator` implements exactly this iterator with an optional data-dependent filtering (that can be implemented via a `filter()` closure). In practical scenarios in which efficiency is important, one would rather perform the data loading asynchronously in multiple threads. The `ParallelDatasetIterator` provides this functionality: it has a predefined number of threads that all load data from the underlying dataset, and when a sample is requested from the iterator, the first available sample is returned. Given sufficient threads, the resulting data iterator will always have a sample available for immediate return, which allows one to hide the loading and preprocessing of data for training or testing altogether. This is particularly important when complex transforms are performed on the data before it is fed to the model, *e.g.*, the affine and color transformations that are commonly applied on images when training computer-vision models (Howard, 2013).

2.3. Engines

When experimenting with different models and datasets, the underlying training procedure is often the same. The `Engine` abstraction provides the boilerplate logic necessary for the training and testing of models. In particular, it implements the interaction between the model (which is assumed to be a `nn.Module`), the `DatasetIterator` and the loss function (which is assumed to be a `nn.Criterion`). An instance of an `Engine` implements two functions that specify these interactions: (1) a `train()` function that samples data, propagates this data through the model, computes the value of the loss, propagates the loss gradient through the model, and performs parameter updates; and (2) a `test()` function that samples data, propagates the data through the model, and measures the quality of the resulting predictions.

An `Engine` provides a collection of hooks that allow the user to plugin experiment-specific code such as performance `Meter`s without editing the core logic of the `Engine`. This encourages re-use of code and potentially prevents bugs, whilst still providing complete flexibility in writing training and testing code. A particularly nice feature of hooks is that they are implemented as closures, making it easy to share logic (such as the copying of data samples to the GPU) between the code that is used for training and the code that is used for testing models.

Dataset	Description
AUCMeter	Area under ROC curve.
AverageValueMeter	Average value of variable.
ClassErrorMeter	Classification error.
ConfusionMeter	Confusion matrix.
MultiLabel ConfusionMeter	Confusion matrix for multi-label problems.
NDCGMeter	Normalized discounted cumulative gain.
PrecisionAtKMeter	Precision at K.
PrecisionMeter	Precision at threshold.
RecallMeter	Recall at threshold.
TimeMeter	Elapsed time.

Table 2. Overview of all Meters implemented in Torchnet.

The current Torchnet code contains two Engine implementations: (1) a SGDEngine that implements training of models via SGD and (2) an OptimEngine that implements training of models via any of the optimizers implemented in the torch/optim package, which includes AdaGrad (Duchi et al., 2011), Adam (Kingma & Ba, 2015), conjugate gradients, and L-BFGS.

2.4. Meters

During both training and testing of learning models, one typically wants to measure properties such as the time needed to perform a training epoch, the value of the loss function averaged over all examples, the area under the ROC curve of a binary classifier, the classification error of a multi-class classifier, the precision and recall of a retrieval model, or the normalized discounted cumulative gain of a ranking algorithm. Torchnet provides a wide variety of Meters that prevent researchers from re-implementing such performance measurements over and over again (and possibly introducing bugs when doing so). Table 2 provides an overview of all meters currently supported. Most meters (with the exception of TimeMeter and AverageValueMeter) implement two main functions: (1) an add(output, target) function that adds the values of model outputs and corresponding targets for one or more samples to the meter, and (2) a value() function that returns the current value of the meter.

2.5. Logs

Torchnet provides two Logs for logging experiments: a simple Log and a RemoteLog. Both can output log information as raw text (to a file or stdout) and as JSON.

3. Example

This section presents a simple, working example of how to train a logistic regressor on the MNIST dataset using Torchnet[2]. The code first includes necessary dependencies:

[2]The example is included in the Torchnet repository.

```
require 'nn'
local tnt   = require 'torchnet'
local mnist = require 'mnist'
```

Subsequently, we define a function that constructs an asynchronous dataset iterator over the MNIST training or test set. The dataset iterator receives as input a closure that constructs the Torchnet Dataset object. Here, the dataset is a ListDataset that simply returns the relevant row from tensors that contain the images and the targets; in practice, you would replace this ListDataset with your own dataset definition. The core Dataset is wrapped in a BatchDataset to construct mini-batches of size 128:

```
local function getIterator(mode)
  return tnt.ParallelDatasetIterator{
    nthread = 1,
    init    = function() require 'torchnet' end,
    closure = function()
      local dataset = mnist[mode .. 'dataset']()
      return tnt.BatchDataset{
         batchsize = 128,
         dataset = tnt.ListDataset{
           list = torch.range(
             1, dataset.data:size(1)
           ),
           load = function(idx)
             return {
               input  = dataset.data[idx],
               target = torch.LongTensor{
                 dataset.label[idx]
               },
             } -- sample contains input and target
           end,
        }
      }
    end,
  }
end
```

Subsequently, we set up a simple linear model:

```
local net = nn.Sequential():add(nn.Linear(784,10))
```

Next, we initialize the Torchnet engine and implement hooks that reset, update, and print the average loss and the average classification error. The hook that updates the average loss and classification error is called after the forward() call on the training criterion:

```
local engine = tnt.SGDEngine()
local meter  = tnt.AverageValueMeter()
local clerr  = tnt.ClassErrorMeter{topk = {1}}
engine.hooks.onStartEpoch = function(state)
  meter:reset()
  clerr:reset()
end
engine.hooks.onForwardCriterion =
function(state)
  meter:add(state.criterion.output)
  clerr:add(
    state.network.output, state.sample.target)
```

```
  print(string.format(
    'avg. loss: %2.4f; avg. error: %2.4f',
    meter:value(), clerr:value{k = 1}))
end
```

Next, we minimize the logistic loss using SGD:

```
local criterion = nn.CrossEntropyCriterion()
engine:train{
  network   = net,
  iterator  = getIterator('train'),
  criterion = criterion,
  lr        = 0.1,
  maxepoch  = 10,
}
```

After the model is trained, we measure the average loss and the classification error on the test set:

```
engine:test{
  network   = net,
  iterator  = getIterator('test'),
  criterion = criterion,
}
```

More advanced examples would likely implement additional `hooks` in the `engine`. For instance, if one wants to measure the test error after each training epoch, this may be implemented in the `engine.hooks.onEndEpoch` hook. Making the same example run a GPU requires a few simple additions to the code, in particular, to copy both the model and the data to the GPU. Copying data samples to a buffer on the GPU[3] can be performed by implementing a hook that is executed after the samples become available:

```
require 'cunn'
net       = net:cuda()
criterion = criterion:cuda()
local input  = torch.CudaTensor()
local target = torch.CudaTensor()
engine.hooks.onSample = function(state)
  input:resize(
      state.sample.input:size()
  ):copy(state.sample.input)
  target:resize(
      state.sample.target:size()
  ):copy(state.sample.target)
  state.sample.input  = input
  state.sample.target = target
end
```

4. Comparison with Other Frameworks

Torchnet is substantially different from frameworks for deep learning such as Caffe (Jia et al., 2014), Chainer[4], TensorFlow (Abadi et al., 2016), and Theano (Bergstra et al., 2011) in that it does not focus on performing efficient inference and gradient computations in deep networks. Instead, Torchnet provides a framework on top of a deep-learning framework (in this case, `torch/nn`) that makes rapid experimentation easier by providing boilerplate code and by encouraging a modular design that makes it easy to re-use code. Torchnet makes few assumptions about the underlying learning framework: Torchnet abstractions can readily be implemented for, *e.g.*, Caffe or TensorFlow.

Torchnet is similar to Blocks and Fuel for Theano[5]. In particular, Torchnet `Datasets` are similar to Fuel `Transformers`, but Torchnet `Datasets` are more flexible because they also implement batching, splitting, and resampling of data. Torchnet `DatasetIterators` are similar to Fuel `DataStreams`, but Torchnet has better support for asynchronous, multi-threaded data loading: Fuel provides a `ServerDataStream` that runs separate data-loading process, which communicates with the trainer via TCP sockets — multi-threaded data loading has to be implemented manually. By contrast, Torchnet provides a plug-in `ParallelDatasetIterator` that makes asynchronous, multi-threaded data loading trivial to use. A potential advantage of Fuel's `ServerDataStream` over Torchnet's `ParallelDatasetIterator` is that the data-loading process can run on a different machine than the training code. In Blocks, `Bricks` are similar to `Modules` of `torch/nn`, and `MainLoop` in Blocks is similar to Torchnet's `Engine`. Blocks implements some basic performance measures such as classification errors but, at present, Blocks does not provide a rich set of measures as implemented by Torchnet `Meters`.

5. Outlook

We envision Torchnet to become a community-owned platform that, next to the core implementation of Torchnet, provides a collection of subpackages in the same way that Torch does. The most important subpackages we foresee will provide implementations of boilerplate code that is relevant to machine-learning problems in, for instance, computer vision (`vision`), natural language processing (`text`), and speech processing (`speech`). However, other subpackages may be smaller and focus on more specific problems or even specific datasets. For instance, we envision having small subpackages that wrap vision datasets such as the Imagenet and COCO datasets (Deng et al., 2009; Lin et al., 2014a), speech datasets such as the TIMIT and LibriSpeech datasets (Garofalo et al., 1993; Panayotov et al., 2015), and text datasets such as the One Billion Word Benchmark and WMT-14[6] datasets (Chelba et al., 2013) into a Torchnet `Dataset`.

[3] It is faster to copy data to a pre-allocated buffer on the GPU than to do a copy via a call to `cuda()` because allocating and freeing memory on the GPU is computationally expensive.

[4] See `http://chainer.org` for details.

[5] See `https://github.com/mila-udem` for details.

[6] See `http://statmt.org/wmt14` for details.

References

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G.S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mane, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viegas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. In *arXiv:1603.04467*, 2016.

Bergstra, J., Bastien, F., Breuleux, O., Lamblin, P., Pascanu, R., Delalleau, O., Desjardins, G., Warde-Farley, D., Goodfellow, I., Bergeron, A., and Bengio, Y. Theano: Deep learning on gpus with python. In *Proceedings of the NIPS BigLearn Workshop*, 2011.

Chelba, C., Mikolov, T., Schuster, M., Ge, Q., Brants, T., Koehn, P., and Robinson, T. One billion word benchmark for measuring progress in statistical language modeling. In *arXiv:1312.3005*, 2013.

Chintala, S. https://github.com/soumith/convnet-benchmarks, 2016.

Collobert, R., Kavukcuoglu, K., and Farabet, C. Torch7: A matlab-like environment for machine learning. In *Proceedings of the NIPS BigLearn Workshop*, 2011.

Deng, J., Dong, W., Socher, R., Li, L. J., Li, K., and Fei-Fei, L. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, (CVPR)*, 2009.

Duchi, J., Hazan, E., and Singer, Y. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

Garofalo, J.S., Lamel, L.F., Fisher, W.M., Fiscus, J.G., Pallett, D.S., and Dahlgren, N.L. Timit acoustic-phonetic continuous speech corpus ldc93s1, 1993.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016.

Howard, A.G. Some improvements on deep convolutional neural network based image classification. In *arXiv:1312.5402*, 2013.

Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., and Darrell, T. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the ACM International Conference on Multimedia*, pp. 675–678, 2014.

Kingma, D.P. and Ba, J. Adam: A method for stochastic optimization. In *Proceedings of the International Conference on Learning Representations*, 2015.

Lin, T., Maire, M., Belongie, S., Hays, J., Perona, P., Ramanan, D., Dollár, P., and Zitnick, L. Microsoft COCO: Common objects in context. In *ECCV 2014*, 2014a.

Lin, T., Maire, M., Belongie, S., Hays, J., Perona, P., Ramanan, D., Dollár, P., and Zitnick, L. Microsoft COCO: Common objects in context. In *ECCV 2014*, 2014b.

Panayotov, V., Chen, G., Povey, D., and Khudanpur, S. Librispeech: An asr corpus based on public domain audio books. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, pp. 5206–5210, 2015.