Blocks and Fuel

Bart van Merriënboer bart.van.vanmerrienboer@umontreal.ca
Montreal Institute for Learning Algorithms, University of Montreal, Montreal, Canada

Dzmitry Bahdanau d.bahdanau@jacobs-university.de
Jacobs University, Bremen, Germany

Jan Chorowski jan.chorowski@ii.uni.wroc.pl
University of Wrocław, Wrocław, Poland

Vincent Dumoulin dumouliv@iro.umontreal.ca
Dmitriy Serdyuk serdyuk.dmitriy@gmail.com
David Warde-Farley wardefar@iro.umontreal.ca
Yoshua Bengio yoshua.bengio@umontreal.ca
Montreal Institute for Learning Algorithms, University of Montreal, Montreal, Canada

Abstract

We introduce two Python frameworks to train neural networks on large datasets: *Blocks* and *Fuel*. *Blocks* is based on the Theano, a linear algebra compiler with CUDA-support (Bastien et al., 2012; Bergstra et al., 2010). It facilitates the training of complex neural network models by providing parametrized Theano operations, attaching metadata to Theano's symbolic computation graph, and providing an extensive set of utilities to assist training the networks, e.g. training algorithms, logging, monitoring, visualization, and serialization. *Fuel* provides a standard format for machine learning datasets. It allows the user to easily iterate over large datasets, performing many types of pre-processing on the fly.

Keywords: Neural networks, GPGPU, large-scale machine learning

1. Introduction

Blocks and *Fuel* are being developed by the Montreal Institute of Learning Algorithms (MILA) at the University of Montreal. Their focus lies on quick prototyping of complex neural network models. The intended target audience is researchers.

Several other libraries built on top of Theano exist, including Pylearn2 (also developed by MILA), Lasagne, Keras and GroundHog. Blocks differentiates itself with a strong focus on research, and through its relationship with Theano. Instead of introducing new abstract objects representing 'models' or 'layers', Blocks annotates the Theano computation graph, maintaining the flexibility of Theano while making large models manageable.

Data processing is an integral part of training neural networks, which is not addressed by many of the aforementioned frameworks. *Fuel* aims to fill this gap. It provides tools to download datasets and iterate/preprocess them efficiently.

2. Blocks

Blocks comprises several components, which can be used independently from each other.

2.1 Bricks

Theano is a popular choice for the implementation of neural networks (see e.g. Goodfellow et al. (2013b); Pascanu et al. (2013)). Blocks and many other libraries, such as Pylearn2 (Goodfellow et al., 2013a), build on Theano by providing reusable components that are common in neural networks, such as linear transformations followed by non-linear activations, or more complicated components such as LSTM units. In Blocks these components are referred to as *bricks* or "parametrized Theano operations".

Bricks consist of a set of Theano shared variables, for example the weight matrix of a linear transformation or the filters of a convolutional layer. Bricks use these parameters to transform symbolic Theano variables.

Bricks can contain other bricks within them. This introduces a hierarchy on top of the flat computation graph defined by Theano, which makes it easier to address and configure complex models programmatically.

The parameters of bricks can be initialized using a variety of schemes that are popular in the neural network literature, such as sparse initialization, orthogonal initialization for recurrent weights, etc.

Blocks comes with a large number of 'bricks'. Besides standard activations and transformations used in feedforward networks (maxout, convolutional layers, table lookups) these also include a variety of more advanced recurrent neural network components (LSTM, GRU, support for attention mechanisms).

2.2 Graph management

Large neural networks can often result in Theano computation graphs containing hundreds of variables and operations. Blocks does not attempt to abstract away this complex graph, but to make it manageable by annotating variables in the graph. Each input, output, and parameter of a brick is annotated as such. Variables can also be annotated with the role they play in a model, such as *weights*, *biases*, *filters*, etc.

A series of convenience tools were written that allow users to filter the symbolic computation graph based on these annotations, and apply transformations to the graph. Methods such as weight noise and dropout can be implemented this way, resulting in an approach that allows for complex queries such as "apply weight noise to all weights that belong to an LSTM unit whose parent is a brick with the name *foo*".

2.3 Training algorithms

The implementation of SGD in Blocks is modular, allowing the user to combine different 'step rules' that modify the descent direction (learning rate scaling, momentum, gradient clipping, weight norm clipping, etc.) with a variety of algorithms such as steepest descent, AdaGrad, ADADELTA, Adam, RMSProp, etc.

2.4 Training

Experiment management is performed using a 'main loop', which combines a Theano graph with a training algorithm. The main loop has a flexible extension interface, which is used

to perform tasks such as monitoring validation sets, serialization, learning rate scheduling, plotting, printing and saving logs, etc.

3. Fuel

Fuel's goal is to provide a common interface to a variety of data formats and published datasets such as MNIST, CIFAR-10, ImageNet, etc. while making it easy for users to write an interface to new datasets. Fuel allows for different ways of iterating over these datasets, such as sequential or shuffled minibatches, support for in-memory and out-of-core datasets, and resampling (cross validation, bootstrapping). It also provides a variety of on-the-fly preprocessing methods such as random cropping of images, creating n-grams from text files, and the ability to implement many other methods easily.

To sidestep Python's global interpreter lock (GIL) and ensure optimal performance, Fuel can perform all operations in a separate process, transferring the processed data to the training process using TCP sockets.

Blocks relies on Fuel for its data interface, but Fuel can easily be used by other machine learning frameworks that interface with datasets.

Listing 1: Iterating over flattened MNIST features in shuffled batches

```
from fuel.datasets import MNIST
mnist = MNIST(which_set='train')
data_stream = DataStream(
    mnist,
    iteration_scheme=ShuffledScheme(mnist.num_examples, 512)
)
flattened = Flatten(data_stream, which_sources=('features',))
```

3.1 Standardized data format

Datasets are distributed in a wide range of formats. Fuel simplifies dataset storage by converting all datasets to annotated HDF5 files. In addition to being an efficient format for large datasets that don't fit into memory, HDF5 is easy to organize and document. All of the data is stored in a single HDF5 file, with the following metadata attached:

- What are the data sources available (e.g. features, targets, etc.)?
- How are these data sources officially split (e.g. training, validation, and test sets)?
- Are some data sources unavailable for some splits (e.g. test set only offers unlabeled examples)?
- What are the axes semantics for a given data source (e.g. batch, feature, width, height, channel, time, etc.)?

Integrating user data into Fuel is straightforward, and simply requires the data to be written to an HDF5 file with metadata according to the specifications.

3.2 Automated data management

Fuel offers built-in scripts that automate the task of downloading datasets, (similar to e.g. SKDATA[1]) and converting them to Fuel's HDF5 specification.

The fuel−download script is used to download raw data files. Downloading the raw MNIST data files is as easy as typing fuel−download mnist. The fuel−convert script is used to convert raw data files into HDF5-format.

Reproducibility being an important feature of both Fuel and Blocks, the fuel−convert script automatically tags all files it creates with relevant module and interface versions and the exact command that was used to generate these files. Inspection of this metadata is done with the fuel−info script.

4. Serialization and checkpointing

The training of large, deep neural networks can often take days or even weeks. Hence, regular checkpointing of training progress is important. Blocks aims to make the resumption of experiments entirely transparent, even across platforms, while ensuring the reproducibility of these experiments.

This goal is complicated by shortcomings in Python's PICKLE serialization module, which is unable to serialize many iterators, which Fuel heavily depends on in order to iterate over large datasets efficiently. To circumvent this we reimplemented the ITERTOOLS from the Python standard library to be serializable[2].

As a result, Blocks experiments are able to be interrupted in the middle of a pass over the dataset, serialized, and resumed later, without affecting the final training results.

5. Documentation and community

Blocks and Fuel are well documented, with both API documentation and tutorials available online. Two active mailing lists[3] support users of the libraries. A separate repository[4] is maintained for users to contribute non-trivial examples of the use of Blocks. Implementations of neural machine translation models (NMT, Bahdanau et al. (2015)) and the Deep Recurrent Attentive Writer (DRAW, Gregor et al. (2015)) model are publicly available examples of state-of-the-art models succesfully implemented using Blocks.

Acknowledgments

The authors would like to acknowledge the support of the following agencies for research funding and computing support: NSERC, Calcul Québec, Compute Canada, the Canada Research Chairs and CIFAR. We would also like to thank the developers of Theano.

1. https://jaberg.github.io/skdata/
2. https://github.com/dwf/picklable_itertools
3. https://groups.google.com/d/forum/blocks-users and https://groups.google.com/d/forum/fuel-users
4. https://github.com/mila-udem/blocks-examples

References

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2015.

Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, James Bergstra, Ian J. Goodfellow, Arnaud Bergeron, Nicolas Bouchard, and Yoshua Bengio. Theano: new features and speed improvements. NIPS Workshop: Deep Learning and Unsupervised Feature Learning, 2012.

James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*, June 2010.

Ian J. Goodfellow, David Warde-Farley, Pascal Lamblin, Vincent Dumoulin, Mehdi Mirza, Razvan Pascanu, James Bergstra, Frédéric Bastien, and Yoshua Bengio. Pylearn2: a machine learning research library. *arXiv preprint arXiv:1308.4214*, 2013a. URL `http://arxiv.org/abs/1308.4214`.

Ian J. Goodfellow, David Warde-Farley, Mehdi Mirza, Aaron Courville, and Yoshua Bengio. Maxout networks. pages 1319–1327, 2013b.

Karol Gregor, Ivo Danihelka, Alex Graves, and Daan Wierstra. Draw: A recurrent neural network for image generation. *arXiv preprint arXiv:1502.04623*, 2015.

Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. 2013.