Barrista – Caffe Well-Served

Christoph Lassner[1,3]
classner@tue.mpg.de

Daniel Kappler[2]
dkappler@tue.mpg.de

Martin Kiefel[1,3]
mkiefel@tue.mpg.de

Peter Gehler[1,3]
pgehler@tue.mpg.de

Bernstein Center for
Comp. Neuroscience[1]
Otfried-Müller-Str. 25
Tübingen, Germany

MPI for Intelligent Systems,
Autonomous Motion Dep.[2]
Paul-Ehrlich-Str. 15
Tübingen, Germany

MPI for Intelligent Systems,
Perceiving Systems Dep.[3]
Spemannstr. 41
Tübingen, Germany

SUBMITTED to ACM MULTIMEDIA 2016 OPEN SOURCE SOFTWARE COMPETITION

ABSTRACT

The caffe framework is one of the leading deep learning toolboxes in the machine learning and computer vision community. While it offers efficiency and configurability, it falls short of a full interface to Python. With increasingly involved procedures for training deep networks and reaching depths of hundreds of layers, creating configuration files and keeping them consistent becomes an error prone process.

We introduce the barrista framework, offering full, pythonic control over caffe. It separates responsibilities and offers code to solve frequently occurring tasks for pre-processing, training and model inspection. It is compatible to all caffe versions since mid 2015 and can import and export .prototxt files.

Examples are included, e.g., a deep residual network implemented in only 172 lines (for arbitrary depths), comparing to 2320 lines in the official implementation for the equivalent model.

Categories and Subject Descriptors

I.5.1 [**Pattern Recognition**]: Applications—*Computer Vision*; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Software libraries*; I.5.1 [**Pattern Recognition**]: Models—*Neural Nets*

Keywords

Open Source; Computer Vision; Machine Learning; Neural Networks; Deep learning; caffe

MM '16, October 15 - 19, 2016, Amsterdam, Netherlands

ISBN 978-1-4503-3603-1/16/10…$15.00

DOI: http://dx.doi.org/10.1145/2964284.2973803

1. INTRODUCTION

Deep learning is one of the biggest success cases of machine learning in the last years. For problems like image classifications, previously considered one of the most challenging problems, deep learning has led to solutions that surpassed human performance. The tremendous commercial and academic success spurred development of several software packages. For the computer vision community, the *caffe* [7] framework is popular and frequently used. It is well-tested with many custom layer implementations and published models.

caffe does provide Python bindings for the core features, however the main workflow relies on *Google protocol buffers*[1] (which we will abbreviate to 'protobuf') for configuration, network design and serialization. This has been a pragmatic solution for network layouts so far, but new, increasingly complex, training strategies and networks with multiple hundreds of layers [4, 5] push this architecture to its limits.

We present the *barrista* framework, which provides a more powerful, programmatic interface, and propose solutions and insights that may be of general interest. Furthermore, we strongly advocate the concept of callbacks (which we refer to as 'monitors' throughout the paper, due to the *monitor* software design pattern). We use monitors as an easy-to-write and easy-to-combine tool to create data loaders, pre-processing, hyperparameter modification, logging, post-processing and visualization.

The features of *barrista* can be summarized as follows:

- The *barrista* framework provides full Python access to all *caffe* functionalities. Multi-GPU training is not yet available, but in the final development stage. Networks can be programmatically created and edited in a convenient way.
- Protobuf object introspection is at the core of the *barrista* library. It enables almost automatic parsing of the basic *caffe* interface and guarantees full compatibility and consistency with the used version of *caffe*. This includes custom-written layers and their parameters, which need to be referenced just by their name. All layer implementation is then inferred automatically.
- We maintain full compatibility with *caffe* models. By using the internal protobuf representation, it is possible to load and save all models and prototxt files *caffe* can read and write.

[1] https://developers.google.com/protocol-buffers/

- *barrista* provides a *monitor* interface with many existing monitors for pre-processing, post-processing, data augmentation, visualization, training and model inspection. Many of them can be executed in parallel to the network forward and backward pass to make the execution efficient.
- This provides a principled separation of responsibilities for steps such as data-preparation, data-feedback (i.e., incorporating training results to influence further training for, e.g., active learning) and visualization.

2. RELATED WORK

A full overview of deep learning software is out of the scope of this paper. We restrict the discussion to the most prominent frameworks and interface packages targeting the Python language.

The ***Theano*** package [11] with its three interface add-ons *Lasagne* [3], *Keras*[2] and *blocks* [13] is similar to *caffe* and *barrista*. *Theano* is a more general machine learning software compared to *caffe*. Therefore, the wrappers are convenient for the specific use case of deep learning. Whereas *Lasagne* keeps a focus on being a lightweight and close wrapper around *Theano*, *Keras* aims for more generality and offers a backend for *Tensorflow* [1] as well, which will be discussed in the next paragraph. ***Keras*** includes a convenient fitting method, but does not separate responsibilities as clearly as the framework we propose. It offers the possibility to use callbacks, but these can not influence the training or the provided data as deeply as *barrista* and are not executed in parallel. ***Lasagne*** is a lightweight wrapper that focuses solely on Theano's deep learning components. It requires the user to implement a training loop including preprocessing and monitoring. ***blocks*** is very similar to *barrista* in its aims and offers a callback concept with its *Extensions*.

Tensorflow [1] is Google's open source deep learning framework and subject to rapid changes. This framework may emerge as a successor to *caffe*. Currently (v0.9), does not offer high level training and pre-processing routines. There is a thriving and competitive infrastructure ecosystem growing around it with no mature and clearly leading framework yet.

Mxnet [2] provides APIs for several languages, including Python. It offers high-level training methods and two separate concepts for callbacks. Plain callbacks do not have access to the network's gradient in contrast to a specific monitor object. However, only one such object can be used for training. Both callback types are not executed in parallel and can not be used for data-preprocessing.

Chainer [12] fully leverages the Python software stack and is designed for easy modification. This does not extend over the course of training methods, probably because it is relatively easy to implement these methods. However, reimplementation leads to creating the same pre-processing and optimization code repeatedly, which is what we address with our callback stack.

Neon[3] relies on the Python stack with the focus on runtime. It provides a comprehensive callback system, but not for parallel execution. While *Neon* is fast, its commercial distribution model offers parts as premium content. With open source alternatives available, this software has not found widespread use in the research community.

[2] http://keras.io/

[3] https://github.com/NervanaSystems/neon

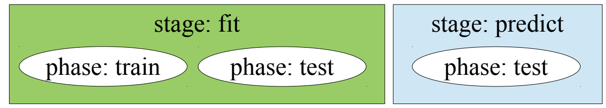

Figure 1: Usage of stages and phases by the 'Net' object. If available, the 'predict' stage is used automatically by the 'predict' function.

3. CONCEPTS AND DESIGN

3.1 Separation of responsibilities

One of the core ideas of the library design is the separation of responsibilities:

The user is responsible for preparing the data. This action is highly dataset specific and can hardly be generalized.

The library offers data augmentation, solver setup, logging, training and prediction. Data augmentation methods (like rotation, flipping) can be shared across learning tasks, therefore this responsibility can be moved to the deep learning library.

We believe that a library should solve repetitive tasks for the user out-of-the-box, but remain configurable. As in the popular *scikit-learn*[4] package, *barrista* offers many default options with sensible values that can be overridden. For example, input data can be passed as a list to the 'fit' or 'predict' methods of a model, but if the data does not fit into memory, it is straightforward to extend the *CyclingDataMonitor* to process a dataset chunkwise.

3.2 Representation

The description of a network is captured in a *NetSpecification* object. It can be programmatically constructed, altered, converted to and from a protobuf, the lingua franca for *caffe*. It describes a network layout in all possible stages and phases and is independent of a trained model.

To use a network model, its description can be instantiated to obtain a *Net* object. A *Net* has parameters and can be fitted to data or used to make predictions. If the network description includes a distinction between the 'fit' and 'predict' phases, then the corresponding architectures are automatically generated and used (see Fig. 1).

In contrast to other frameworks, the *Solver* is represented as a stateful entity, which reflects the underlying *caffe* structure for optimization methods such as Adam [8]. Furthermore, this has the advantage that the object can be serialized correctly. All solvers implemented in *caffe* are encapsulated with parameter checks.

3.3 Monitoring

We propose a powerful monitoring system with nearly full control over the training process. Our experience is that this offers an easy way to experiment with different training strategies. The monitors can access and modify both, the network and solver. Monitors for data loading and augmentation can be combined arbitrarily, to quickly adapt to the specific training needs without having to rewrite and test the code.

[4] http://scikit-learn.org

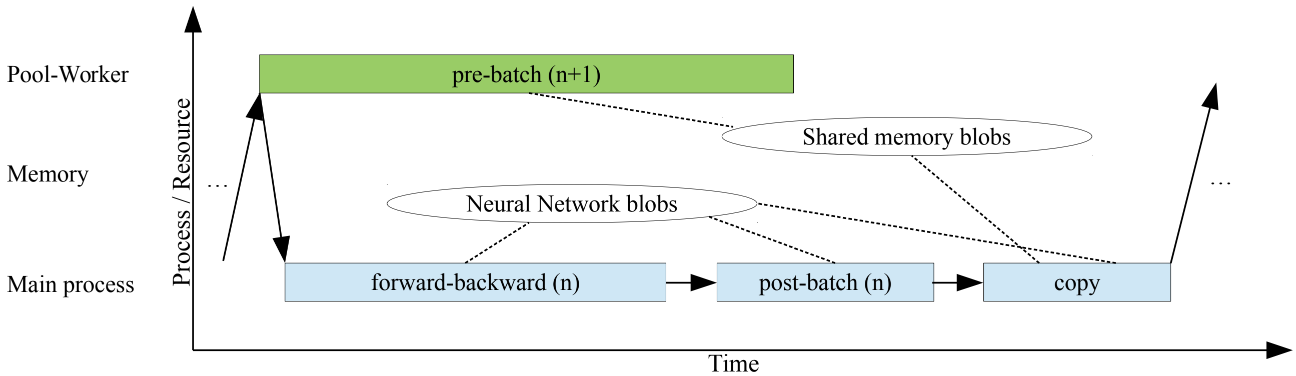

Figure 2: Parallel callback processing pipeline. Solid arrows indicate control flow, dashed lines resource access. The duration of the 'forward-backward' and 'pre-batch' operation strongly depends on network layout/pre-processing steps, hence the length of the boxes is not necessarily proportional to execution time. Memory is only allocated once at the beginning of training and does not change over time (the blobs have limited extent only for easier visualization).

4. IMPLEMENTATION

4.1 Protobuf object introspection

Python offers full object introspection abilities: all properties can be queried and their name and type examined. We use this feature to analyze the *caffe*-generated protobuf object. Exploiting the knowledge about *caffe*'s naming convention, we can infer significant parts of the interface automatically.

This allows us to work with upstream and custom defined layers. The only additional information which must be provided is the relationship between a layer and its parameters, since this is not encoded in the protobuf specification. It is sufficient to specify layer and parameter names, the rest is inferred from *barrista*.

With this architecture, we are able to elegantly avoid the maintenance overhead for wrapping an evolving library. Additionally, we keep the additional effort for new layer definitions as low as possible. In the core *caffe* library, adding new layers requires changes in several places.

4.2 Monitor implementation

We use an inheritance-based Signal/Slot design pattern implementation to create the monitors. This makes (1) creation of new signals as well as (2) creation of new monitors easy and preserves simplicity and separation of the code. The monitors receive their parameters solely through a dictionary. This keeps the interface flexible and extensible and enables state modification.

4.3 Parallel monitor execution

True parallel threaded execution in one process is inherently difficult in Python due to the global interpreter lock. However, data loading can be time-consuming and slow down the training. The *caffe* data loading facilities can be used like any other layer, additionally we provide parallel data layers that are executed in a true parallel context. To make this functionality available on all platforms, we rely on the Python *multiprocessing* module to create an additional worker process. To avoid interprocess communication overhead, we use shared memory with locks for data transfer. Any monitor implementing the *ParallelDataMonitor* interface will be executed in parallel, which is useful for expensive pre-processing tasks. A control flow visualization is given in Fig. 2.

Exploiting Python's duck typing, we create a dummy *Net* object that implements the basic interface of the *Net* with its memory pointing to shared memory with the main process. Thus, the data monitors executed in parallel can be used unchanged for serial execution, e.g., for debugging. After parallel execution, the main process copies the shared memory to the original *Net* object, an operation that usually takes less than 1ms.

4.4 Visualization

With visualization monitors it is possible to create plots in regular intervals and create movies of filter evolution. We include a basic set of monitors. This includes (1) activation, (2) gradient histogram, (3) gradient magnitude and (4) loss visualization; all of which allow to keep a close view of the network during training to diagnose obstacles for the optimizer, e.g., vanishing gradients in very deep architectures [14]. Adding more involved visualizations such as the 'guided backprop' visualization [10] can easily be integrated using a monitor.

4.5 Examples

We include three self-contained examples with the library: an overview over the most important functions, the classical MNIST training example with a three layer CNN, and an implementation of a state-of-the art residual network for CIFAR10 [9, 4]. The latter two are complete setups that can directly be adapted for new training scenarios. The folders come with 'data.py', 'train.py', 'test.py', 'visualize.py' and a model folder with the proposed model file. All the files are executable and highlight one aspect of the setup.

4.6 Quality assurance

To provide high quality code, we follow best practices. With a public CI server that checks primary (tests) and secondary (style) quality of the code, we assure that the library is usable and bugs are detected before a code release.

Monitoring the test coverage at the same time (currently more than 94% for the non-user-interface functions of the library; the UI can not be tested on the headless CI server), we make sure that no important parts remain untested. All parts of the library are covered with documentation. Python 3 compatibility is implemented, but is currently not maintainable due to the lacking compatibility of the core *caffe* framework.

(a) (b) (c) (d)

Figure 3: Example visualizations. **(a)** Human pose estimation and semantic segmentation, **(b)** intrinsic image reflection component, **(c)** bilateral feature space with x, y, r, g, b features of an image [6] (non-standard caffe layer), **(d)** activations from the last layer of a deep residual network on CIFAR10 [9] using visualization monitors during training.

5. USAGE SCENARIOS

The library wraps the entire *caffe* functionality, currently with the exception of multi-GPU support (which can be added by a minimal C++ patch to *caffe*). *barrista* can be used for all applications of *caffe*. We used it successfully for several applications: human pose estimation, semantic segmentation, image filtering, image classification, character recognition, material classification, intrinsic image decomposition as well as robotic grasp pose prediction. Some example visualizations are shown in Fig. 3. With this wide array of applications, we are confident that *barrista* can deliver a high level of convenience for a large user base.

LSTM support in *caffe* is currently under development and discussion. It is likely to be integrated in the main framework. Since the underlying representation remains tied to protobuf, only minimal changes should be required to make these features available in *barrista*.

6. AVAILABILITY AND CONCLUSION

The source code is available under the MIT license from Github at https://github.com/classner/barrista. It is currently necessary to apply a C++ patch to *caffe* to make *barrista* work. A pull request for the required changes to the master branch of *caffe* is pending (#3629). The change is of general interest, and once accepted we will make *barrista* available as a PyPi project. In the meantime we provide a patched *caffe* version as a submodule of the project.

barrista is highly flexible and aims to bridge the gap between performance, convenience and continuity. We are confident that it prove useful in many applications and contribute to an easier usage of *caffe* by reducing development time for both, beginners and experts of deep learning.

We thank Yangqing Jia and the BVLC vision group for creating and maintaining *caffe*.

7. REFERENCES

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems. http://tensorflow.org/, 2015.

[2] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. http://mxnet.rtfd.org, 2015.

[3] S. Dieleman, J. Schlüter, C. Raffel, E. Olson, S. K. Sønderby, D. Nouri, D. Maturana, M. Thoma, E. Battenberg, J. Kelly, J. D. Fauw, M. Heilman, diogo149, B. McFee, H. Weideman, takacsg84, peterderivaz, Jon, instagibbs, K. Rasul, CongLiu, Britefury, and J. Degrave. Lasagne: First release. http://dx.doi.org/10.5281/zenodo.27878, 2015.

[4] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *CVPR*, 2016.

[5] K. He, X. Zhang, S. Ren, and J. Sun. Identity mappings in deep residual networks. *arXiv preprint arXiv:1603.05027*, 2016.

[6] V. Jampani, M. Kiefel, and P. V. Gehler. Learning sparse high dimensional filters: Image filtering, dense crfs and bilateral neural networks. In *CVPR*, 2016.

[7] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[8] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1312.6980*, abs/1412.6980, 2014.

[9] A. Krizhevsky. Learning Multiple Layers of Features from Tiny Images. Master's thesis.

[10] J. T. Springenberg, A. Dosovitskiy, T. Brox, and M. A. Riedmiller. Striving for simplicity: The all convolutional net. *arXiv preprint arXiv:1412.6806*, abs/1412.6806, 2014.

[11] Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv preprint arXiv:1605.02688*, 2016.

[12] S. Tokui, K. Oono, S. Hido, and J. Clayton. Chainer: a next-generation open source framework for deep learning. In *Proc. of the Workshop on Machine Learning Systems (LearningSys) of NIPS*, 2015.

[13] B. van Merriënboer, D. Bahdanau, V. Dumoulin, D. Serdyuk, D. Warde-Farley, J. Chorowski, and Y. Bengio. Blocks and fuel: Frameworks for deep learning. *arXiv preprint arXiv:1506.00619*, 2015.

[14] S.-E. Wei, V. Ramakrishna, T. Kanade, and Y. Sheikh. Convolutional pose machines. In *CVPR*, 2016.