Automatic Batching as a Compiler Pass in PyTorch

James Bradbury
Google Brain*
jekbradbury@google.com

Chunli Fu
Columbia University†
cf2710@columbia.edu

Abstract

Define-by-run deep learning frameworks like PyTorch provide increased flexibility and convenience, but still require researchers building dynamic models to manually vectorize their networks in order to achieve the performance benefits of minibatching. We describe an automatic batching framework that keeps track of batch-level masking and padding and rewrites data-dependent control flow, and present two implementations: a prototype in an external package, and the second integrated with PyTorch's recently introduced compilation infrastructure. These tools let users implement their models as imperative code that applies to individual data samples, then efficiently train and validate them on batched data; they simplify model code and eliminate a class of implementation bugs by allowing programmers to work directly at a more natural level of abstraction. When the batching transformation runs ahead of time as a part of the PyTorch compiler, our approach has less overhead than other autobatching strategies for define-by-run frameworks.

1 Introduction

One commonly-cited advantage of imperative frameworks like DyNet [Neubig et al., 2017a], Chainer [Tokui et al., 2015], PyTorch [Paszke et al., 2017], TensorFlow Eager, and Flux.jl over graph-based frameworks like Theano [Al-Rfou et al.] and TensorFlow [Abadi et al., 2016] is that they make it easier to implement highly "dynamic" neural networks [Tokui et al., 2015]. These are networks which execute different sequences of operations depending on the structure or content of input data (e.g., as in Socher et al. [2013]) or the results of intermediate computations (as in Dyer et al. [2016]).

This is true in the sense that every dynamic network can be implemented in an imperative framework using language-native control flow like Python's `for` and `if`—but such an implementation will typically only work correctly at batch size one. Since training with larger batch sizes is essential to take advantage of modern parallel hardware, the next step is often to manually convert all or part of the network into a "vectorized" form that operates at once on entire minibatches.

When implementing models for sequence data with varying length, for instance, this process requires adding padding to align different-sized arrays and masking the resulting batches to avoid performing computations with invalid data. Programmers must ensure that mask metadata is correctly propagated through the operations in their neural networks. Many uses of language-native control flow must also be rewritten, including example-dependent branches and loops over variable-length dimensions.

We describe a methodology for automating these transformations, and provide first a prototype implementation as a PyTorch library called Matchbox[3] and, subsequently, a more optimized implementation integrated with the PyTorch compiler as a pass[4] over PyTorch "Torch Script" IR.

*Work done while the author was at Salesforce Research

†Work done while the author was an intern at Facebook AI Research

[3] `https://github.com/salesforce/matchbox`

[4] `https://github.com/pytorch/pytorch/tree/master/torch/csrc/jit/passes/to_batch.cpp`

32nd Conference on Neural Information Processing Systems (NIPS 2018), Montréal, Canada.

2 Approach

Our proposal consists of two relatively independent sets of functionality: automatic mask propagation and control flow rewriting.

2.1 `Batch` type and mask propagation semantics

A minibatch comprises a number of data or activation samples, each potentially with a different shape. These shapes may be fixed in some dimensions, corresponding to model constants like the number of neurons in a particular layer, while varying among examples in the batch in other dimensions, like the number of timesteps in a recurrent encoding. Our approach overloads array and neural network operations on a `Batch` type that holds such a minibatch.

The data values in a `Batch` are stored in a padded array whose size is the minibatch size in the first dimension, the common size of all examples in static dimensions, and at least as large as the largest example in the batch in dynamic dimensions. Shape information is stored in a mask array whose dimensions are the same size as the data array except in static dimensions, where the mask is of size one to avoid redundant storage. Each entry in the mask corresponds to one or more entries in the data array (singleton—i.e., static—dimensions are broadcasted), with a one in the mask denoting that the corresponding data entries represent valid, meaningful data and a zero denoting that they do not.

As in the NumPy Masked Array library [Dubois, 2001], which implements similar semantics, operations overloaded for the `Batch` type satisfy the invariant that invalid values in their inputs never affect valid values in their outputs.

2.2 Control flow rewriting

The system rewrites control flow in a way closely inspired by the single-program multiple-data (SPMD) programming model used by Intel ISPC [Pharr and Mark, 2012] and (under the name SIMT) NVIDIA CUDA [Nickolls et al., 2008]. The control flow rewriting pass—operating on either a Python AST or PyTorch IR—creates an "execution mask" that identifies at runtime which side is taken of a conditional, and which iterations are traversed in a loop, for each example in a batch. Every operation that updates a loop-carried dependency, or sets a variable that will escape the context of a conditional body, is rewritten to have its effect gated by the execution mask.

3 Related Work

There are two existing toolkits for automatic batching: TensorFlow Fold [Looks et al., 2017], which introduces an embedded domain-specific language (DSL) with implicitly vectorized functional semantics, and DyNet autobatch [Neubig et al., 2017b], which lazily constructs computation graphs for each example before applying vectorization as a global graph optimization strategy.

Compared to these tools, our approach has the advantage of permitting true dynamic control flow—e.g., conditionals that switch on runtime data values—without requiring costly recompilation at runtime. The SIMT assumption that our system makes—namely, that computations performed by same line of code as applied to different examples are likely to be able to be batched together—enables it to achieve runtime overhead that is $O(1)$ in the batch size rather than $O(N)$ or worse, but this comes at the cost of reduced generality (e.g., our approach does not support recursion).

4 Conclusion

Much as automatic differentiation allows programmers to avoid having to derive gradient expressions by hand, automatic batching tools abstract away the details of vectorization and allow researchers to implement their networks at the level of individual examples. We have presented an alternative approach to this task that draws on ideas from the programming language and compiler communities and which we hope will help satisfy the research community's desire for more powerful and ergonomic abstractions for deep learning.

References

M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. TensorFlow: A system for large-scale machine learning. In *OSDI*, 2016.

R. Al-Rfou, G. Alain, A. Almahairi, C. Angermueller, D. Bahdanau, N. Ballas, F. Bastien, J. Bayer, A. Belikov, A. Belopolsky, et al. Theano: A Python framework for fast computation of mathematical expressions.

P. Dubois. Design of a Masked Array facility for Python. O'Reilly Open Source Convention, 2001. URL `ftp://ftp.oreilly.com/pub/conference/os2001/Dubois_P_1329.ppt`.

C. Dyer, A. Kuncoro, M. Ballesteros, and N. Smith. Recurrent neural network grammars. In *NAACL*, 2016.

M. Looks, M. Herreshoff, D. Hutchins, and P. Norvig. Deep learning with dynamic computation graphs. In *ICLR*, 2017.

G. Neubig, C. Dyer, Y. Goldberg, A. Matthews, W. Ammar, A. Anastasopoulos, M. Ballesteros, D. Chiang, D. Clothiaux, T. Cohn, et al. DyNet: The dynamic neural network toolkit. *arXiv preprint arXiv:1701.03980*, 2017a.

G. Neubig, Y. Goldberg, and C. Dyer. On-the-fly operation batching in dynamic computation graphs. In *NIPS*, 2017b.

J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. In *ACM SIGGRAPH 2008 classes*, page 16. ACM, 2008.

A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in PyTorch. In *NIPS Autodiff workshop*, 2017.

M. Pharr and W. R. Mark. ispc: A SPMD compiler for high-performance CPU programming. In *Innovative Parallel Computing (InPar), 2012*, pages 1–13. IEEE, 2012.

R. Socher, A. Perelygin, J. Wu, J. Chuang, C. Manning, A. Ng, and C. Potts. Recursive deep models for semantic compositionality over a sentiment treebank. In *EMNLP*, 2013.

S. Tokui, K. Oono, S. Hido, and J. Clayton. Chainer: A next-generation open source framework for deep learning. In *NIPS LearningSys workshop*, 2015.