Automatic differentiation in PyTorch

Adam Paszke
University of Warsaw
adam.paszke@gmail.com

Sam Gross
Facebook AI Research

Soumith Chintala
Facebook AI Research

Gregory Chanan
Facebook AI Research

Edward Yang
Facebook AI Research

Zachary DeVito
Facebook AI Research

Zeming Lin
Facebook AI Research

Alban Desmaison
University of Oxford

Luca Antiga
OROBIX Srl

Adam Lerer
Facebook AI Research

Abstract

In this article, we describe an automatic differentiation module of PyTorch — a library designed to enable rapid research on machine learning models. It builds upon a few projects, most notably Lua Torch, Chainer, and HIPS Autograd [4], and provides a high performance environment with easy access to automatic differentiation of models executed on different devices (CPU and GPU). To make prototyping easier, PyTorch does not follow the symbolic approach used in many other deep learning frameworks, but focuses on differentiation of purely imperative programs, with a focus on extensibility and low overhead. Note that this preprint is a draft of certain sections from an upcoming paper covering all PyTorch features.

1 Background

PyTorch, like most other deep learning libraries, supports reverse-mode [6] automatic differentiation [2] of scalar functions (or vector-Jacobian products of functions with multiple outputs), the most important form of automatic differentiation for deep learning applications which usually differentiate a single scalar loss.

Within this domain, PyTorch's support for automatic differentiation follows in the steps of Chainer, HIPS autograd [4] and twitter-autograd (twitter-autograd was, itself, a port of HIPS autograd to Lua). These libraries have two distinguishing features:

- **Dynamic, define-by-run execution.** A dynamic framework defines the function to be differentiated simply by running the desired computation, as opposed to specifying a static graph structure which is differentiated symbolically ahead of time and then run many times. This permits users to use any host-language features they want (e.g., arbitrary control flow constructs), at the cost of requiring differentiation to be carried out every iteration. Dynamic execution distinguishes PyTorch from static frameworks like TensorFlow [1], Caffe, etc.
- **Immediate, eager execution.** An eager framework runs tensor computations as it encounters them; it avoids ever materializing a "forward graph", recording only what is necessary to differentiate the computation. This stands in contrast to DyNet [5], which employs lazy evaluation, literally rebuilding the forwards and backwards graph every training iteration.[1] Immediate execution allows CPU and GPU computation to be pipelined, but gives up the opportunity for whole-network optimization and batching.

[1] DyNet does have an immediate execution mode for debugging, but it is not enabled by default.

31st Conference on Neural Information Processing Systems (NIPS 2017), Long Beach, CA, USA.

Although PyTorch's support for automatic differentiation was heavily inspired by its predecessors (especially twitter-autograd and Chainer), it introduces some novel design and implementation choices, which make it the one of the fastest implementations among automatic differentiation libraries supporting this kind of dynamic eager execution:

- **In-place operations**. In-place operations pose a hazard for automatic differentiation, because an in-place operation can invalidate data that would be needed in the differentiation phase. Additionally, they require nontrivial tape transformations to be performed. PyTorch implements simple but effective mechanisms that address both of these problems. They are described in more detail in Section 3.1.
- **No tape.** Traditional reverse-mode differentiation records a tape (also known as a Wengert list) describing the order in which operations were originally executed; this optimization allows implementations to avoid a topological sort. PyTorch (and Chainer) eschew this tape; instead, every intermediate result records only the subset of the computation graph that was relevant to their computation. This means PyTorch users can mix and match independent graphs however they like, in whatever threads they like (without explicit synchronization). An added benefit of structuring graphs this way is that when a portion of the graph becomes dead, it is automatically freed; an important consideration when we want to free large memory chunks as quickly as possible.
- **Core logic in C++.** PyTorch started its life as a Python library; however, it quickly became clear that interpreter overhead is too high for core AD logic. Today, most of it is written in C++, and we are in the process of moving core operator definitions to C++. Carefully tuned C++ code is one of the primary reasons PyTorch can achieve much lower overhead compared to other frameworks.

```
from torch.autograd import Variable
x, prev_h = Variable(torch.randn(1, 10)), Variable(torch.randn(1, 20))
W_h, W_x = Variable(torch.randn(20, 20)), Variable(torch.randn(20, 10))

i2h = torch.matmul(W_x, x.t())
h2h = torch.matmul(W_h, prev_h.t())
(i2h + h2h).tanh().sum().backward()
```

Figure 1: An example use of PyTorch's automatic differentation module (`torch.autograd`).

2 Interface

Figure 1 gives a simple example of automatic differentiation in PyTorch. You write code as if you were executing tensor operations directly; however, instead of operating on Tensors (PyTorch's equivalent of Numpy's nd-arrays), the user manipulates Variables, which store extra metadata necessary for AD. Variables support a `backward()` method, which computes the gradient of all input Variables involved in computation of this quantity.

By default, these gradients are accumulated in the `grad` field of input variables, a design inherited from Chainer. However, PyTorch also provides a HIPS autograd-style functional interface for computing gradients: the function `torch.autograd.grad(f(x, y, z), (x, y))` computes the derivative of `f` w.r.t. `x` and `y` only (no gradient is computed for `z`). Unlike the Chainer-style API, this call doesn't mutate `.grad` attributes; instead, it returns a tuple containing gradient w.r.t. each of the inputs requested in the call.

Variable flags It is important to not compute derivatives when they are not needed. PyTorch, following Chainer, provides two facilities for excluding subgraphs from derivative computation: the "requires grad" and "volatile" flags. These flags obey the following rules: If *any* input Variable is volatile, the output is volatile. Otherwise, if *any* input Variable requires grad, the output requires grad. Additionally, having volatile set, implies that "requires grad" is unset. If an operation wouldn't require grad, the derivative closure is not even instantiated. The volatile flag makes it easy to disable differentiation (e.g., when running a model for inference, ones does not have to set "requires grad" to false on each parameter).

Hooks One downside of automatic differentiation is that the differentiation is relatively opaque to users: unlike the forward pass, which is invoked by user-written Python code, the differentiation is carried out from library code, which users have little visibility into. To allow users to inspect gradients, we provide a hooks mechanism to allow users to observe when the backwards of a function is invoked: `x.register_hook(lambda grad: print(grad))`. This code registers a hook on `x`, which prints the gradient of `x` whenever it is computed. A hook callback can also return a new gradient which is used in place of the original gradient; this capability has proven to be useful for metalearning and reinforcement learning.

Extensions PyTorch users can create custom differentiable operations by specifying a pair of forward and backward functions in Python. The forward function computes the operation, while the backward method extends the vector-Jacobian product. This can be used to make arbitrary Python libraries (e.g., Scipy [3]) differentiable (critically taking advantage of PyTorch's zero-copy NumPy conversion).

3 Implementation

Internally, a Variable is simply a wrapper around a Tensor that also holds a reference to a graph of Function objects. This graph is an immutable, purely functional representation of the derivative of computed function; Variables are simply mutable pointers to this graph (they are mutated when an in-place operation occurs; see Section 3.1).

A Function can be thought of as a closure that has all context necessary to compute vector-Jacobian products. They accept the gradients of the outputs, and return the gradients of the inputs (formally, the left product including the term for their respective operation.) A graph of Functions is a single argument closure that takes in a left product and multiplies it by the derivatives of all operations it contains. The left products passed around are themselves Variables, making the evaluation of the graph differentiable.

Memory management The main use case for PyTorch is training machine learning models on GPU. As one of the biggest limitations of GPUs is low memory capacity, PyTorch takes great care to make sure that all intermediate values are freed as soon as they become unneeded. Indeed, Python is well-suited for this purpose, because it is reference counted by default (using a garbage collector only to break cycles).

PyTorch's Variable and Function must be designed to work well in a reference counted regime. For example, a Function records pointers to the Function which *consumes* its result, so that a Function subgraph is freed when its retaining output Variable becomes dead. This is opposite of the conventional ownership for closures, where a closure retains the closures it invokes (a pointer to the Function which *produces* its result.)

Another challenge is avoiding reference cycles. A naıve implementation of automatic differentiation can easily introduce such cycles (e.g. when a differentiable function would like to save a reference to its output). PyTorch breaks them by recording not a full-fledged variable, but instead a "saved variable", which omits a pointer to the Function in such cases.

C++ operators We have found that even though it is possible to express all operations in Python using the extension API, they suffer from high interpreter overhead. Moving operators to C++ reduces the overhead and lowers the latency a single differentiable operation dispatched form Python to as few as 3.4us, compared to 1.7us for tensor operation. An additional benefit is that one can have multiple threads executing them in parallel (in contrast to Python, which limits parallelism due to the GIL). This is especially important in context of multiple GPUs, which cannot be saturated by a single CPU thread.

3.1 Supporting in-place operations

Frequently, users of PyTorch wish to perform operations *in-place* on a tensor, so as to avoid allocating a new tensor when it's known to be unnecessary. Intuitively, an in-place operation is equivalent to its corresponding non-inplace operation, except that the Variable which is modified in-place has its computation history "rebased" to point to the derivative of the in-place operator, instead of its

previous Function (the computation history always remains purely functional). However, these in-place operations interact with autograd in a subtle way.

Invalidation An in-place operation can invalidate data that is needed to compute derivatives. Consider the following example:

```
y = x.tanh()
y.add_(3)
y.backward()
```

This program instructs PyTorch to perform an in-place operation on y; however, this is unsound if y, whose value is $\tanh(x)$, was saved so that it could be used in the backwards computation (recall that $\tanh'(x) = 1 - \tanh^2(x)$).

Making a copy of y when saving it would be inefficient, PyTorch instead fails at runtime when differentiating this program. Every underlying storage of a variable is associated with a *version counter*, which tracks how many in-place operations have been applied to the storage. When a variable is saved, we record the version counter at that time. When an attempt to use the saved variable is made, an error is raised if the saved value doesn't match the current one.

Aliasing PyTorch supports nontrivial aliasing between variables; operations like transpose and narrow produce new tensors with new sizes and strides which share storage with the original tensors. The problem with aliasing is that it can require nontrivial transformations on computation histories of many variables. Consider the following example:

```
y = x[:2]
x.add_(3)
y.backward()
```

Ordinarily, an in-place operation on x would only affect the computational history of x. However, in this case, the in-place addition to x also causes some elements of y to be updated; thus, y's computational history has changed as well. Supporting this case is fairly nontrivial, so PyTorch rejects this program, using an additional field in the version counter (see Invalidation paragraph) to determine that the data is shared.

In future work, we are looking to relax this restriction. The challenge is that there may be arbitrarily many aliases of a variable, so it is not feasible to go through each one-by-one and update their computation histories. However, it may be possible to *lazily* initialize a computation history, materializing it only when a computation produces a result that is not aliased to the original variable.

References

[1] M. Abadi, A. Agarwal, P. Barham, et al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] A. Griewank. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

[3] E. Jones, T. Oliphant, P. Peterson, et al. SciPy: Open source scientific tools for Python, 2001.

[4] D. Maclaurin. *Modeling, Inference and Optimization with Composable Differentiable Procedures*. PhD thesis, 2016.

[5] G. Neubig, C. Dyer, Y. Goldberg, and o. Matthews. DyNet: The Dynamic Neural Network Toolkit. *ArXiv e-prints*, Jan. 2017.

[6] B. Speelpenning. *Compiling Fast Partial Derivatives of Functions Given by Algorithms*. PhD thesis, Champaign, IL, USA, 1980. AAI8017989.