Tangent: Automatic Differentiation Using Source Code Transformation in Python

Bart van Merriënboer
Google Inc.
bartvm@google.com

Alexander B Wiltschko
Google Inc.
alexbw@google.com

Dan Moldovan
Google Inc.
mdan@google.com

Abstract

Automatic differentiation (AD) is an essential primitive for machine learning programming systems. Tangent is a new library that performs AD using *source code transformation* (SCT) in Python. It takes numeric functions written in a syntactic subset of Python and NumPy as input, and generates new Python functions which calculate a derivative. This approach to automatic differentiation is different from existing packages popular in machine learning, such as TensorFlow[1] and Autograd[1]. Advantages are that Tangent generates gradient code in Python which is readable by the user, easy to understand and debug, and has no runtime overhead. Tangent also introduces abstractions for easily injecting logic into the generated gradient code, further improving usability.

1 Introduction

Modern machine learning and deep learning relies heavily on gradient-based optimization algorithms. These methods require the efficient calculation of derivatives of potentially complex, high-dimensional models. AD is a set of techniques to evaluate these derivatives. It is based on the insight that the chain rule can be applied to the elementary arithmetic operations (primitives) performed by a program, and nearly every machine learning library implements it. Note that AD is different from symbolic differentiation (which applies to mathematical expressions instead of programs) and numerical differentiation (where the gradient is approximated using finite differences).

Two approaches to automatic differentiation are common: *tracing* and *source code transformation* (SCT). In the tracing approach, primitives are overloaded so that each operation is logged onto a tape (a linear trace) at runtime. The chain rule can then be applied by walking this tape backward. Source code transformation, on the other hand, explicitly rewrites the code prior to execution to produce a separate gradient version. Both approaches have different implementation, performance, and usability trade-offs [5].

Automatic differentiation packages using both approaches have long existed for, e.g., C, C++, Fortran, (see [3] for an overview) and have been used in fields such as computational fluid dynamics, atmospheric sciences, and astronomy. The machine learning community's different needs, which include a heavy focus on linear algebra, performance through heterogenous (GPGPU) computing, and a pervasive use of Python, led to the development of separate tools.

Theano [2] and TensorFlow [1] are two popular machine learning frameworks with support for AD. Although Python-based, they do not perform AD on the Python code. Instead, Python is used as a metaprogramming language to define a dataflow graph (computation graph) on which SCT is performed. Since these dataflow graphs do not have function calls or lexical scoping, the AD logic is simplified. However, the introduction of a separate programming paradigm which requires its own runtime can be confusing to the user.

[1] `https://github.com/HIPS/autograd`

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

AD has been implemented for Python and NumPy using tracing in the Autograd and ad[2] packages[3].

2 Benefits of Source Code Transformation for Automatic Differentiation

Because Tangent is (to our knowledge) the first SCT-Based AD system for Python, it occupies a unique point in the space of tradeoffs among usability, flexibilty, debuggability, and computational performance. It allows different tradeoffs in usability, and ease of debugging than prior systems.

2.1 Usability

The metaprogramming approach used by Theano and Tensorflow often results in more verbose and less idiomatic code (see Listing 1, left).

```
# TensorFlow
with tf.Graph().as_default():
  x = tf.get_variable('x', shape=())
  y = x * x
  dx = tf.gradients(y, x)

  with tf.Session() as sess:
    sess.run(x.initializer)
    sess.run(dx)
```

```
# Tangent / Autograd
def f(x):
    return x * x
df = grad(f)
```

Listing 1: Left: Gradient of $x \cdot x$ in TensorFlow. Right: Gradient of $x \cdot x$ in Tangent and Autograd, which both use a similar API: `grad(f)`

Autograd and Tangent allow users to write models in more idiomatic code (see Listing 1, right). This is particularly useful for code that contains control flow constructs (see Listing 2)

```
# TensorFlow
def loop_tensorflow(x, num_steps):
  def cond(i, _):
    return i < num_steps

  def body(i, x):
    result = tf.cond(
        tf.reduce_sum(x) > 1,
        lambda: x / 2, lambda: x)
    return tf.add(i, 1), result

  i = tf.constant(0)
  _, ret_x = tf.while_loop(
    cond, body, [i, x])
  return tf.reduce_sum(ret_x)
```

```
# Tangent / Autograd
def loop_tangent(x, num_steps):
  for _ in range(num_steps):
    if np.sum(x) > 1:
      x /= 2
  return np.sum(x)
```

Listing 2: Left: Implementation of a differentiable nested loop and conditional in TensorFlow. Right: Implementation of the same program in Tangent. Much less code is required, due to the use of native Python syntax.

However, the tracing approach can be problematic for debugging and usability. When the function `df` is called, the function `f` is executed with non-standard semantics (logging to the tape), after which the tape is walked in reverse using a loop that is internal to Autograd. Errors that occur anywhere during execution will potentially have tracebacks that are hard to understand for the user, because they are buried inside the Autograd implementation.

[2] `http://pythonhosted.org/ad/`

[3] A more complete list of AD tools in several languages: `http://www.autodiff.org/?module=Tools`

Because Tangent uses source code transformation, the function `df` that Tangent generates is a new Python function, with standard semantics, whose source code can be directly inspected (see Listing 3). This simplifies both user understanding and debugging.

```
def dfdx(x, by=1.0):
    # Grad of: y = x * x
    _bx = tangent.unbroadcast(by * x, x)
    _bx2 = tangent.unbroadcast(by * x, x)
    bx = _bx
    bx = tangent.add_grad(bx, _bx2)
    return bx
```

Listing 3: Source code of gradient of $x \cdot x$ in Tangent. The `unbroadcast` is responsible for reversing the broadcasting done by NumPy when performing element-wise operations on differently-sized multidimensional arrays.

2.2 Injecting Custom Logic Into Gradients

There are several cases in which it can be useful for the user to inject custom code into the gradient computation.

Many algorithms use approximations or modifications of the gradient. For example, for performance reasons, recurrent neural networks (RNNs) are often trained using truncated backpropagation through time [8] (TBPTT). This algorithm performs fewer loop iterations in the gradient than in the original function. In other cases, custom gradients are used to train models with discontinuous functions (e.g. using straight-through estimators [4]).

Second, debugging errors in the gradient computation (e.g., overflow/underflow) can greatly benefit from the ability to insert arbitrary code into the generated gradient, which allows the user to add logging statements or insert breakpoints.

Traditional AD frameworks have little support for this kind of code injection. Theano and TensorFlow allow the user to manipulate the dataflow graph of the gradient directly to accomplish some of these changes, but this can be cumbersome. Tangent overloads Python's context manager syntax to introduce a novel way of allowing the user to inject arbitrary code into the gradient computation. We have found this syntax to be a very succinct way of implementing these cases.

```
def f(x):
    with grad_of(x) as dx:
        if dx > 10:
            print('Warning, large gradient of x', dx)
        dx /= 2
    return x * x
```

Listing 4: Gradient manipulation; in this case `df(2)` will return 2, because the gradient is halved. Logging statements for large gradient values (or `NaN` gradient values) are also easily inserted.

3 Implementation

Tangent uses Python's built-in machinery to introspect and transform the *abstract syntax tree* (AST) of parsed source code at runtime. For each piece of supported Python syntax, we have implemented a rule indicating how to rewrite an AST node into its backward pass equivalent, or "adjoint". We have defined adjoints for function calls to NumPy methods, as well as larger pieces of syntax, such as if-statements and for-loops. The adjoints are stored in function definitions that serve as "templates", or code macros [6]. Another alternative, which we found too cumbersome, would be to use a templating engine like Mustache [4] and store adjoints as plain strings. Our templates also use a special syntax `d[x]` to refer to the derivative of a variable `x` (see Listing 5).

[4] `https://mustache.github.io/`

```
# Code quote specifying the gradient of np.multiply.
# This function serves only as a container for code that will be
# expanded and in-lined in generated code.
def adjoint_multiply(result, arg1, arg2):
  d[arg1] = arg2 * d[result]
  d[arg2] = arg1 * d[result]


# To generate the adjoint for this line...
var3 = np.multiply(var1,var2)

#  ... we use macro expansion
new_ast = tangent.template.replace(adjoint_multiply,
                                   result='var3',
                                   arg1='var1',
                                   arg2='var2')

# If the code specified in the AST 'new_ast' were converted into a string:
b_var1 = var2 * b_var3
b_var2 = var1 * b_var3
```

Listing 5: Underlying implementation of gradient code construction. The user will not routinely write code in this style, unless implementing custom gradients. The gradient of `np.multiply` is specified as a code quote. This function will never be run as-is — it only contains a template that will be used to generate contextually-correct gradient code. Tangent internally discovers the names of the arguments and results in the code snippet and looks up the appropriate template (that process is omitted here). Then, Tangent combines the template and the variable names to create a new AST.

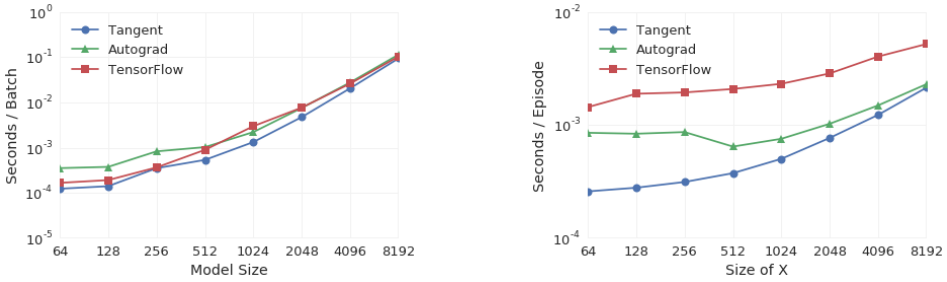
Tangent is restricted to a subset of Python where functions have no side effects. Mutating arrays is allowed, but only through index assignment syntax (`a[i] = b`). This requirement prevents us from having to copy large multi-dimensional arrays each time they are used. Tangent also does not support closures, because closures with free variable references lead to a problem sometimes referred to as 'perturbation confusion', which is non-trivial to address [7]. These restrictions only apply to statements that involve active variables i.e., variables which affect the output of the function whose derivative we are computing.

Two other limitations on the supported subset of Python are worth mentioning. First, if the name of a function cannot be tracked to its source code ahead-of-time, we will generate an error. This situation may arise if functions are being passed as variables, or if they are being renamed in the user's code. A second case is the use of classes, and class member functions. This is an important use case, as many large neural network models are coded using an object-oriented style. Tangent does not currently support taking derivatives through classes, although we are actively working on this feature.

We optimize generated code for both readability and performance. We do this by constructing a control flow graph (CFG) from the AST in order to determine which variables are active (a form of forward dataflow analysis). To improve readability of the final code and performance, we use Tangent's ability to perform dataflow analysis on Python code to perform several simplifications on the transformed AST (similar to an optimizing compiler). We perform algebraic simplifications e.g., instead of explicitly initializing a gradient to zero and accumulating into it (`dx = 0; dx += 2`) we simply assign (`dx = 2`) where possible. We also perform dead code elimination (cf. Listing 3 where the original statement, `y = x * x`, was removed).

4 Performance

Because Tangent performs AD ahead-of-time, it has no runtime overhead. Its performance then depends largely on the CPython interpreter and implementations of the underlying numeric kernels, such as matrix multiplication and convolution. Here, we compare the performance of Tangent,



(a) Benchmark results for the MLP from Listing 6.

(b) Benchmark for a simple loop from Listing 2.

Figure 1: Average of 50 runs, batch size of 16, for varying number of parameters. Run on a Xeon E5-1650 v3 @ 3.5 GHz, 64GB of RAM, with Ubuntu 14.04 on Python 2.7, with MKL.

TensorFlow and Autograd, a tracing automatic differentiation system for NumPy. As a simple benchmark, we use a multi-layer perceptron, implemented in NumPy (see Listing 6).

Tangent's performance is superior to Autograd on a simple multi-layer perceptron benchmark (Figure 1a), particularly for smaller model sizes, where overhead dominates. Autograd[5] adds interpretive overhead at every gradient call, because it first traces user code, and then interprets the trace to calculate the derivative. Tangent largely matches the performance of TensorFlow, slightly exceeding it for some model sizes. Tangent is faster for all model sizes for the simple loop benchmark (Figure 1b).

```
def logsumexp(x, axis=None, keep_dims=False):
  return np.log(np.sum(np.exp(x), axis=axis, keepdims=keep_dims))

def logsoftmax(logits):
  return logits - logsumexp(logits, axis=-1, keep_dims=True)

def softmax_crossent(logits, y):
  return -np.sum(logsoftmax(logits) * y, axis=-1)

def mlp(x, w1, b1, wout, bout, label):
  h1 = np.tanh(np.dot(x, w1) + b1)
  out = np.dot(h1, wout) + bout
  loss = np.mean(softmax_crossent(out, label))
  return loss

# After generating these functions for calculating the derivative
# of 'mlp()', we timed their execution
autograd_dmlp = autograd.multigrad(mlp,argnums=(1,2,3,4))
tangent_dmlp = tangent.grad(mlp,wrt=(1,2,3,4))
```

Listing 6: Multilayer perceptron used for benchmarking Tangent and Autograd

5 Conclusion

We have introduced the AD library Tangent, highlighted several of its unique features, and compared its performance to existing AD libraries.

Tangent is unique in that both the original function and the generated gradient are pure Python, which allows for easy debugging and introspection. Gradient transformation is a first class operation in Tangent which does not require direct manipulation of the internal representation or the redefinition of primitives. We believe that this enables easier development of complex machine learning models in Python.

[5]Code checked out from: `https://github.com/HIPS/autograd/tree/7fa48ab4c`

We have only described the use of Tangent with NumPy, but it is agnostic to the numeric libraries used, as long as gradients for library functions are defined. We plan to extend support of Tangent to other numeric libraries, particularly those with GPU support. In future work, we hope to make it easier to express larger and more complicated models in Tangent, as well as increase the subset of Python that Tangent supports. We plan to release Tangent as a free and open-source library on GitHub in late November 2017.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[2] Rami Al-Rfou, Guillaume Alain, Amjad Almahairi, Christof Angermueller, Dzmitry Bahdanau, Nicolas Ballas, Frédéric Bastien, Justin Bayer, Anatoly Belikov, Alexander Belopolsky, et al. Theano: A python framework for fast computation of mathematical expressions. *arXiv preprint*, 2016.

[3] Atilim Gunes Baydin, Barak A Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *arXiv preprint arXiv:1502.05767*, 2015.

[4] Yoshua Bengio, Nicholas Léonard, and Aaron Courville. Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint arXiv:1308.3432*, 2013.

[5] Christian H Bischof and H Martin Bücker. Computing derivatives of computer programs. Technical report, Argonne National Lab., IL (US), 2000.

[6] Eugene Kohlbecker, Daniel P Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In *Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 151–161. ACM, 1986.

[7] Barak A. Pearlmutter and Jeffrey Mark Siskind. Reverse-mode ad in a functional framework: Lambda the ultimate backpropagator. *ACM Trans. Program. Lang. Syst.*, 30(2):7:1–7:36, March 2008.

[8] Ronald J Williams and Jing Peng. An efficient gradient-based algorithm for on-line training of recurrent network trajectories. *Neural computation*, 2(4):490–501, 1990.