Tangent: Automatic differentiation using source-code transformation for dynamically typed array programming

Bart van Merriënboer
MILA, Google Brain
bartvm@google.com

Dan Moldovan
Google Brain
mdan@google.com

Alexander B Wiltschko
Google Brain
alexbw@google.com

Abstract

The need to efficiently calculate first- and higher-order derivatives of increasingly complex models expressed in Python has stressed or exceeded the capabilities of available tools. In this work, we explore techniques from the field of *automatic differentiation* (AD) that can give researchers expressive power, performance and strong usability. These include source-code transformation (SCT), flexible gradient surgery, efficient in-place array operations, and higher-order derivatives. We implement and demonstrate these ideas in the Tangent software library for Python, the first AD framework for a dynamic language that uses SCT.

1 Introduction

Many applications in machine learning rely on gradient-based optimization, or at least the efficient calculation of derivatives of models expressed as computer programs. Researchers have a wide variety of tools from which they can choose, particularly if they are using the Python language [21, 16, 24, 2, 1]. These tools can generally be characterized as trading off *research* or *production* use cases, and can be divided along these lines by whether they implement automatic differentiation using operator overloading (OO) or SCT. SCT affords more opportunities for whole-program optimization, while OO makes it easier to support convenient syntax in Python, like data-dependent control flow, or advanced features such as custom partial derivatives. We show here that it is possible to offer the programming flexibility usually thought to be exclusive to OO-based tools in an SCT framework.

Tangent is the first AD framework using SCT in a dynamically typed language. We produce efficient derivatives using a novel combination of multiple dispatch, lazy evaluation, and static optimizations. Further, Tangent has mutable multidimensional arrays as first class objects, implemented using persistent data structures for performance in the context of reverse mode AD. By operating directly on Python source code, Tangent is able to achieve a separation of concerns that other AD libraries do not. Specifically, we achieve compositionality with tools in the Python ecosystem, such as debuggers, profilers and other compilers. Tangent makes it easy and efficient to express machine learning models, and is open source [1].

2 Background

Automatic differentiation (AD) is a set of techniques to evaluate derivatives of mathematical functions defined as programs [10], and is heavily used in machine learning [3]. It is based on the insight that the chain rule can be applied to the elementary arithmetic operations (primitives) performed by the program. This allows derivatives to be calculated up to machine precision [17] with only a constant

---

[1]Source code and documentation available at https://github.com/google/tangent

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

overhead per operation. 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).

For multidimensional functions, $f : \mathbb{R}^n \to \mathbb{R}^m$, where f is a composition of primitives with known derivatives, the application of the chain rule results in a series of matrix-vector multiplications involving the primitives' Jacobians and partial derivatives of intermediate values. The order in which these multiplications are evaluated determines the runtime complexity. Forward-mode AD evaluates the chain rule from inside to outside and is efficient for functions where $m > n$. The implementation of forward mode is relatively straightforward, since the partial derivatives are evaluated in step with the primitives. Forward mode is commonly implemented by replacing numbers with *dual numbers*, which can be interpreted as a variable's value along with its partial derivative with respect to one of the inputs. Reverse-mode AD, where the chain rule is evaluated from outside to inside, is more efficient in the case where $n > m$. Reverse mode is more complex to implement because evaluation of the partial derivatives requires reversing the execution order of the original program. This reversal gives rise to a non-local program transformation where the beginning of the original program interacts with the generated derivative program.

Two methods of implementing reverse-mode AD are commonly distinguished: operator overloading (OO) and source code transformation (SCT). In the OO approach primitives are overloaded so that at runtime each numerical operation is logged onto a tape (a linear trace) along with its inputs. The chain rule can then be evaluated by walking this tape backward. SCT, on the other hand, explicitly transforms the original program (primal) prior to execution to produce a separate derivative function (adjoint) whose control flow is the reverse of the original program. Both approaches have different implementation, performance, and usability trade-offs [5].

OO is easier to implement and since it only requires tracing, it naturally supports all the features of the host language such as higher-order functions, recursion, and classes. If the control flow of the program is data dependent, the function must be retraced for each function call, which can cause significant overhead when the runtime of the primitives is small compared to the cost of tracing. Since the adjoint program is run by a separate 'derivative interpreter' (the algorithm that walks the tape in reverse), there is no adjoint program that can be inspected, optimized or compiled.

SCT is harder to implement, since it requires tooling to transform intermediate representations of computer programs. Further, the AD tool must explicitly support all of the features of the host language, including function calls, loops, classes, etc. If a language feature is not explicitly handled by the AD system, the user cannot take derivatives of code using those features. For some languages like C and C++ this requires a separate toolchain, but reflective languages such as Lisp and Python contain the necessary tools to capture, transform, and output program representations. The advantage of SCT is that there is no runtime overhead, and that generated derivative code can be statically analyzed and optimized.

3   Prior work

AD packages using either approach 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. In the machine learning community different needs have led to the development of a separate set of tools. In particular, the community has a strong attachment to Python and its models rely heavily on multidimensional arrays.

Theano [2] and TensorFlow [1] are two popular machine learning frameworks with support for SCT 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 only operate on immutable values and do not have function calls or lexical scoping, the AD logic is simplified. The same graph representation is then used for static analysis, optimizations, and code generation.

OO has been used to implement AD in Python in packages such as Autograd [16], Chainer [24], and PyTorch [21].

Although OO frameworks are easier to implement, their runtime performance falls short of that of frameworks using SCT for workloads that do not spend most of their time in hand-optimized compute

2

primitives. On the other hand, existing frameworks that use SCT require the user to metaprogram computation graphs, significantly complicating the definition of ML models. Tangent applies SCT directly on the Python language in order to combine the performance achieved by SCT with the usability of programming directly in Python.

4 Features

Tangent supports reverse mode and forward mode, as well as function calls, loops, and conditionals. Higher-order derivatives are supported, and reverse and forward mode can readily be combined. To our knowledge, Tangent is the first SCT-based AD system for Python and moreover, it is the first SCT-based AD system for a dynamically typed language. As a consequence of performing SCT directly on the Python source code, the generated programs can be run, inspected, profiled, and debugged with standard Python tools. Tangent supports array programming on both CPU and GPU through the NumPy [19] and TensorFlow Eager libraries. A modular design makes it possible to extend Tangent to support other numeric libraries.

The ability to write code directly in Python makes Tangent less verbose and more idiomatic than the metaprogramming approach used by Theano and Tensorflow (see Listing 1a). Moreover, the metaprogrammed code requires a separate compiler and/or runtime, separate debugging tools, etc.

```
x = tf.placeholder(tf.float32)
y = x * x
dx, = tf.gradients(y, x)

with tf.Session() as sess:
  dx_ = sess.run(dx, feed_dict={x: 3})
```

```
def f(x):
    return x * x

df = grad(f)
dx = df(3)
```

(a) TensorFlow requires the programmer to define the variable x as part of the dataflow graph. After the program (dataflow graph) has been constructed, its evaluation must be triggered by creating a session and providing values for the arguments.

(b) Tangent and libraries such as Autograd allow the user to write pure Python.

Listing 1: Comparison between metaprogramming and direct programming approaches.

The OO approach can be problematic for debugging and usability as well as performance (see Listing 2). When an adjoint function grad(f) is called, the function f is executed with non-standard semantics, since each function and operator has been overloaded to log onto a tape, after which the tape is walked in reverse using a loop that is internal to the framework. This means that each function call incurs tracing overhead, and errors that occur during execution will potentially have tracebacks involving tracing logic that can be hard for a user to decipher.

```
def f(x):
    while x < 10000:
        x = x + 1
    return x
```

Listing 2: In the case that x is a scalar, this trivial program and its derivative contain a tight loop. Since it does not require tracing, Tangent's derivative of this function is approximately 30% faster than PyTorch's, even though PyTorch is given type information about x whereas Tangent's derivative is dynamically typed.

```
# Generated gradient function
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 the gradient of def f(x): return x * x in Tangent. The unbroadcast function is responsible for reversing the broadcasting performed by NumPy when performing element-wise operations on differently-sized multidimensional arrays.

The adjoint code generated by Tangent is regular Python (see Listing 3), which means that it can be debugged using standard debuggers such as pdb, profiled using, e.g., line_profiler, optimized by JIT compilers such as Numba [14] and Pythran [11]. The adjoint code can readily be inspected

3

by users, and Tangent tries to ensure that is human-readable and commented, which is useful for debugging as well as for didactic purposes.

Unlike most existing ML frameworks, arrays in Tangent are mutable without incurring unnecessary performance loss (see Section 5.4 for implementation details).

4.1 Backward pass inlining

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 [26] (TBPTT) and/or gradient clipping [20]. In other cases, custom gradients are used to train models with discontinuous functions (e.g. straight-through estimators) or for many other applications [4, 9, 25, 12, 13, 18, 15]. A user might also be interested in accessing the values of gradients for logging or debugging.

Existing AD frameworks support this functionality by allowing the user to define custom adjoints for functions. Tangent provides this functionality as well, but uses Python's context manager syntax to introduce a second, novel way of allowing the user to inject arbitrary code into the gradient computation (see Listing 4). We believe this syntax provides a more succinct and readable way of modifying the adjoint code in many cases.

```python
# Original function
def f(x):
  with insert_grad_of(x) as dx:
    if dx > 10:
      print('Clipping', dx)
      dx = 10
  return x * x
```

```python
# Generated gradient function
def dfdx(x, bx_times_x=1.0):
  x_times_x = x * x
  # Grad of: dx = 10
  _bx = tangent.unbroadcast(bx_times_x * x, x)
  _bx2 = tangent.unbroadcast(bx_times_x * x, x)
  bx = _bx
  bx = tangent.add_grad(bx, _bx2)
  # Inserted code
  if bx > 10:
    print('Clipping', bx)
    bx = 10
  return bx
```

Listing 4: Gradient clipping implemented using Tangent. The code inside of the context manager is inserted directly into the derivative function.

5 Implementation

Tangent uses Python's built-in machinery to inspect and transform the abstract syntax tree (AST) of parsed source code. AD can be performed line by line [10, Proposition 4.2]. Hence, for each piece of supported Python syntax we have implemented a rule indicating how to rewrite an AST node into its primal and adjoint. We have defined adjoints for e.g. mathematical operators, function calls to NumPy methods, and constructs such as if-statements and for-loops. The adjoints are defined using a custom template programming syntax (see Listing 5) which makes it easy for users to add new or custom derivatives.

```python
# Templates are Python functions
@adjoint(numpy.multiply)
def adjoint_multiply(z, x, y):
  d[x] = y * d[z]
  d[y] = x * d[z]
# If the primal contains...
c = numpy.multiply(a, b)
```

```python
# ...Tangent will expand the template...
new_ast = tangent.template.replace(
    adjoint_multiply,
    z='c', x='a', y='b')
# ...generating the following adjoint
b_a = b * b_c
b_b = a * b_c
```

Listing 5: Tangent's source generation uses templating. The template takes the form of a Python function which is parsed into its AST. The variable names in the AST are substituted and variables for the partial derivatives are constructed, before the AST is inserted into the code of the adjoint function.

Generated derivative code is constructed using the built-in Python AST. The alternative program representations are Python bytecode, which changes across Python versions, and a formatting-aware

4

AST used in the Python 2-to-3 conversion tool, `2to3`, which has little tooling and is more cumbersome to use. We acquire and manipulate the Python AST with the `inspect` and `ast` modules from the standard library, and standardize small differences between the Python 2 and Python 3 AST with `gast` and use `astor` to invert ASTs into readable source code.

To support dynamic typing and array programming while maintaining efficiency, Tangent relies on a novel combination of multiple dispatch, lazy evaluation, persistent data structures, and static optimizations.

5.1 Multiple dispatch

Python is a dynamic language which uses dynamic typing, late binding and operator overloading. These fundamental features of the language make it impossible to determine ahead of time how a statement will be executed, which means it is impossible to determine ahead of time what the adjoint program should be. Instead of enforcing static types (for example by using type annotations and MyPy[2]), Tangent embraces late binding and generates adjoints that will use the runtime types to determine what derivative computation to execute.

For example, `x * y` where `x` and `y` are scalars at runtime results in a scalar multiplication. However, if either of the two variables is a NumPy `ndarray` object, the multiplication operator is dispatched to perform broadcasting followed by element-wise multiplication. The adjoint of this operation requires summing over the broadcasted axes. Tangent will generate code that uses type checking to ensure that the correct adjoint calculation is performed based on the runtime types.

Similarly, the initialization and addition of gradients cannot be generated statically. We introduce `add_grad` and `init_grad` operators which use multiple dispatch. For example, `init_grad(x)` will return `0` if x is a scalar, but will return `numpy.zeros_like(x)` if x is an `ndarray`.

5.2 Lazy evaluation

A common performance bottleneck in the context of AD and array programming is that initializing the gradient of a large array results in allocating a large zero array. When gradients are accumulated later on this large array of zeros is added to a partial gradient, which is effectively a no-op. In general, the gradient initialization and addition might happen in different functions, making it non-trivial to statically optimize this case. To address this issue, Tangent lazily initializes gradients: Instead of allocating an array of zeros, Tangent returns a special `ZeroGrad` object. The `add_grad` operator uses multiple dispatch to return the other argument when either argument is of the type `ZeroGrad`.

5.3 Static optimizations

When constructing the adjoint of a function, some of the code of the forward pass might become dead code. The opportunity for removing unused code only grows when taking higher order derivatives. One of the advantages of SCT is that the resulting code can be optimized by an optimizing compiler whose dead code elimination (DCE) pass would address this problem. However, Python is an interpreted language, and very few optimizations are applied before its execution. For this reason, Tangent includes a small Python optimizing compiler toolchain which constructs a control-flow graph (CFG) on which forward dataflow analysis is performed. Tangent uses this to perform dead code elimination on generated adjoints. The same machinery is used to perform algebraic simplifications and constant propagation. Note that although these optimizations are hard to perform on Python in general, we can exploit the fact that Tangent operates on a more limited subset of Python which is more amenable to analysis (see Section 6 for details).

Note that these optimizations are aimed at removing dead code or simplifying trivial expressions (such as multiplication by 1) generated by the AD algorithm. Unlike frameworks such as XLA and TVM [7], we expressly do not attempt to optimize the numerical kernels themselves. Since Tangent outputs regular Python code, functions can be passed to an optimizing Python compiler such as Numba for this purpose.

A central problem in reverse mode AD is that intermediate values are required to be kept alive after they go out of scope since they might be needed by their adjoint. For example, if a function contains

---

[2]`http://mypy-lang.org/`

5

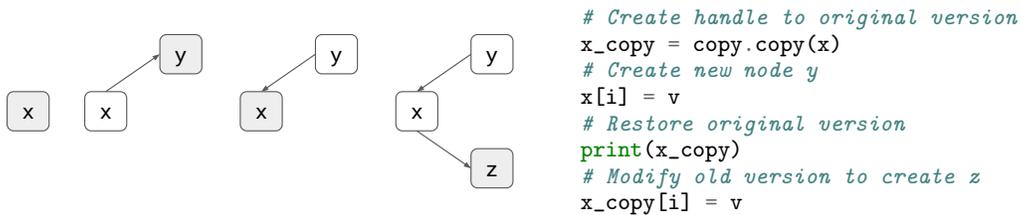
```
# Raw generated code                    # Optimized generated code
def dfdx(x, by=1.0):                     def dfdx(x, by=1.0):
  # Initialize the tape                    y = x
  _stack = tangent.Stack()                 # Grad of: y = x
  y = None                                 _bx = tangent.copy(by)
  # Beginning of forward pass              bx = _bx
  tangent.push(_stack, y, '_19429e9f')     return bx
  y = x
  # Beginning of backward pass
  _y = y
  # Grad of: y = x
  y = tangent.pop(_stack, '_19429e9f')
  _bx = tangent.copy(by)
  by = tangent.init_grad(y)
  bx = _bx
  return bx
```

Listing 6: A simple example of Tangent's optimization capabilities as applied to the gradient function of `def f(x): y = x; return y`. Note that the original transformation includes the writing and reading of y to and from the tape, and contains dead code in initializing the gradient of y which is never returned. Tangent's dataflow analysis is able to match the tape reads and writes and understands that the value of y is the same, allowing it to aggressively optimize the function.

`z = x * y` the variables x and y cannot be deleted after the function returns since the backward pass requires their values to calculate `dx = dz * y` and `dy = dz * x`. Tangent, like most SCT frameworks, uses a global stack (tape) to store intermediate variables on in order to ensure they are kept alive. Hence, before the function returns, x and y are pushed onto this stack and they will be popped off the stack right before the adjoint calculation. Note that the trace used in OO is also referred to as a tape, the difference being that the tape in OO stores not only the intermediate variables, but also the operations performed.

In order to perform DCE effectively on the generated code, our dataflow analysis follows variables uses through their respective pushes (reads) and pops (writes) in the primal and adjoint code. This highlights the close interaction required between the optimizing compiler and the AD machinery for maximum performance. To enable the dataflow analysis to match reads and writes they are augmented in the source code with unique hashes (see Listing 6).

5.4 Persistent data structures

```
# Create handle to original version
x_copy = copy.copy(x)
# Create new node y
x[i] = v
# Restore original version
print(x_copy)
# Modify old version to create z
x_copy[i] = v
```

Listing 7: Illustration of the persistent array data structure. Root nodes are gray and edges represent deltas.

AD is problematic in the context of mutability. If x and y from the previous example are mutable arrays, their value could have been changed by an in-place operation, resulting in an incorrect adjoint calculation. For this reason, arrays are in principle immutable in existing AD frameworks for ML such as TensorFlow, Autograd, and Theano. PyTorch allows users to mutate arrays if they can guarantee that the previous version will not be needed by the backward pass, otherwise an error will be thrown. This makes algorithms which rely on mutating arrays in place inefficient and difficult to express.

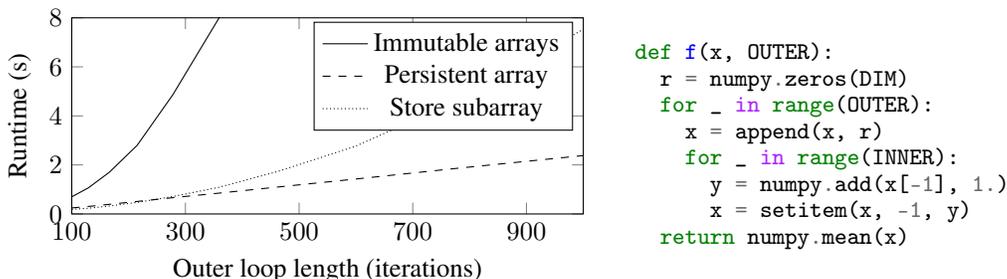
Persistent data structures [8] are data structures that are effectively immutable: They are mutable data structures where all previous versions can be accessed. Unlike truly immutable data structures,

6

different versions of persistent data structures may share memory and hence can be more memory-efficient, although accessing previous versions might require extra work. Functional languages often use persistent data structures for implementing, e.g., linked lists, trees, stacks. We note that persistent array data structures can be used to support mutability efficiently in the context of AD.

By default, Tangent handles index assignments (`x[i] = y`) efficiently by copying only the affected subarray `x[i]` onto the tape. To deal with mutability in full generality Tangent also introduces a persistent array data structure with support for index assignment as well as inserting and deleting rows at the end. Each time the array is modified, the delta with the previous version is stored. Since previous versions can be modified as well, this results in a version tree where the root contains the current array in memory and other versions of the array are represented by leaf nodes (see Listing 7). If the user attempts to read a specific version of the array, the deltas on the path from the leaf to the root of the version tree are applied in order to reconstruct the array in memory. When the handle to a specific array version is destroyed, the deltas are garbage collected. We note that in the context of reverse mode AD the most common access pattern is a linear series of mutations during the forward pass, followed by accessing the arrays in reverse order during the backward pass. In this case, our persistent array results in optimal memory and time complexity.

As an example, consider the double loop from Listing 8, which is a simplification of a neural lattice language model from [6]. Given an outer loop with n iterations, an inner loop with m iterations, and a vector dimensionality of d, the complexity of this algorithm is $O(n^2md)$ for immutable arrays. When using regular NumPy arrays, Tangent will intelligently handle index assignments and only copy the affected subarray onto the tape, bringing the complexity down to $O(n^2d + ndm)$. When a persistent array is used, the complexity goes down to $O(ndm)$. When using persistent arrays, Tangent's runtime and memory complexity is determined only by the amount of data that is inserted, deleted or modified. In contrast, most libraries will have the gradient's runtime and memory complexity grow linearly with the number of times an array is modified. The technique described in [22] for memory-augmented networks is also a special case of using persistent arrays.

```python
def f(x, OUTER):
    r = numpy.zeros(DIM)
    for _ in range(OUTER):
        x = append(x, r)
        for _ in range(INNER):
            y = numpy.add(x[-1], 1.)
            x = setitem(x, -1, y)
    return numpy.mean(x)
```

Listing 8: Runtime for a simplified version of a lattice language model with dimension 2000 and inner loop of 15 iterations. Results are an average of 10 runs.

6 Limitations

SCT relies on the ability to perform dataflow analysis to determine which variables are 'active' i.e. which variables affect the output of the function whose derivative we are constructing. To this end, Tangent is restricted to a subset of Python where these analyses are feasible. Note that these restrictions only apply to statements involving active variables.

1. Functions that modify a variable in-place must also return that variable. Hence, `numpy.add(a, b, out=a)` is disallowed and should be written as `a = numpy.add(a, b)`. Likewise, a user-defined function that modifies x in-place using `x[i] = v`, must have x as a returned value.

2. Closures are not supported since closures with free variable references lead to a problem sometimes referred to as 'perturbation confusion' [23], which is non-trivial to address. Additionally, Python uses lexical, not dynamic scoping, so writing adjoint values into the same scope where primal values are read is not straightforward.

7

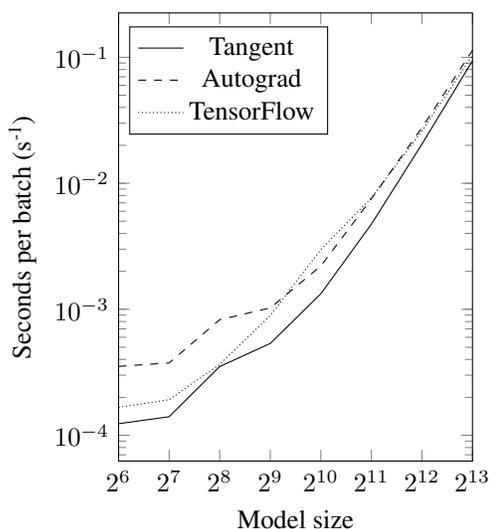
3. Object methods are not currently supported because it is non-obvious what the partial derivative with respect to a member variable is.

4. In order to perform AD, the function and its source code must be resolvable at the time that the AD transformation is applied. This means that higher-order functions and nested function definitions are not supported. Tangent could apply additional AD passes at runtime to avoid this limitation.

5. Some Python syntax is not (yet) supported[3] e.g. `try` and `except` statements, as well as `break` and `continue`.

If the return value of a function is not used, it is assumed that its inputs were unchanged. This allows statements such as `print(numpy.mean(x))` to be used without interfering with the AD transformation.

7   Performance

Tangent was not designed with raw performance in mind. Instead, it intends to strike a balance between usability and good software design practices, while exploring the feasibility and implementation details of applying SCT to dynamically typed languages. That said, Tangent's lack of runtime overhead combined with static optimizations and lazy gradient initialization means that its runtime performance is competitive with existing frameworks (see Listing 9).

```python
def logsumexp(x):
    return numpy.log(numpy.sum(numpy.exp(x),
                              axis=-1, keepdims=True))

def logsoftmax(logits):
    return logits - logsumexp(logits)

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

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

autograd_dmlp = autograd.multigrad(
    mlp, argnums=(1, 2, 3, 4))
tangent_dmlp = tangent.grad(
    mlp, wrt=(1, 2, 3, 4))
```

Listing 9: Runtime for a simple feedforward neural network with a single hidden layer. We vary the input size and hidden layer size, which are set to the same value. The reported runtime is averaged over 50 runs with a batch size of 16. Run on a Xeon E5-1650 v3 @ 3.5 GHz, 64GB of RAM, with Ubuntu 14.04 on Python 2.7 with MKL. Note that for sufficiently large models the runtime of the numerical kernels dominates, which means that the frameworks have similar runtimes irrespective of their AD implementation.

8   Conclusion

In this work we introduced the AD library Tangent. Tangent is the first application of source-code transformation on a dynamically typed language such as Python. It uses several novel approaches, such as persistent data structures and lazy evaluation to ensure good performance. Machine learning models are natural and easy to express and debug in Tangent using many features that are not available in other frameworks e.g. mutable arrays, inspectable derivative code, and modifying gradients by injecting arbitrary code in the backward pass.

---

[3]For an up to date overview of supported AST nodes please refer to the code in `tangent/fence.py`.

8

We believe Tangent is an important step on the path to fully general differentiable programming. Instead of an ML-framework, Tangent can be seen as the addition of the gradient operator to the Python language, without the need for metaprogramming or separate derivative interpreters (OO). This means that the user can write normal Python code while the entire Python ecosystem including debuggers, profilers, and introspection capabilities, become part of the ML toolkit. This allows users to express models more naturally and debug them more easily.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283, 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 arXiv:1605.02688*, 472:473, 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] Jacob Buckman and Graham Neubig. Neural lattice language models. *arXiv preprint arXiv:1803.05071*, 2018.

[7] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594. USENIX Association, 2018.

[8] James R Driscoll, Neil Sarnak, Daniel D Sleator, and Robert E Tarjan. Making data structures persistent. *Journal of computer and system sciences*, 38(1):86–124, 1989.

[9] Yaroslav Ganin, Evgeniya Ustinova, Hana Ajakan, Pascal Germain, Hugo Larochelle, François Laviolette, Mario Marchand, and Victor Lempitsky. Domain-adversarial training of neural networks. *The Journal of Machine Learning Research*, 17(1):2096–2030, 2016.

[10] Andreas Griewank and Andrea Walther. *Evaluating derivatives: principles and techniques of algorithmic differentiation*, volume 105. Siam, 2008.

[11] Serge Guelton, Pierrick Brunet, Mehdi Amini, Adrien Merlini, Xavier Corbillon, and Alan Raynaud. Pythran: Enabling static optimization of scientific python programs. *Computational Science & Discovery*, 8(1):014001, 2015.

[12] Nicolas Heess, Gregory Wayne, David Silver, Tim Lillicrap, Tom Erez, and Yuval Tassa. Learning continuous control policies by stochastic value gradients. In *Advances in Neural Information Processing Systems*, pages 2944–2952, 2015.

[13] Eric Jang, Shixiang Gu, and Ben Poole. Categorical reparameterization with gumbel-softmax. *arXiv preprint arXiv:1611.01144*, 2016.

[14] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: A LLVM-based Python JIT compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, page 7. ACM, 2015.

[15] Timothy P Lillicrap, Daniel Cownden, Douglas B Tweed, and Colin J Akerman. Random feedback weights support learning in deep neural networks. *arXiv preprint arXiv:1411.0247*, 2014.

9

[16] Dougal Maclaurin, David Duvenaud, and Ryan P Adams. Autograd: Effortless gradients in numpy. In *ICML 2015 AutoML Workshop*, 2015.

[17] Uwe Naumann. *The art of differentiating computer programs: an introduction to algorithmic differentiation*, volume 24. Siam, 2012.

[18] Arild Nøkland. Direct feedback alignment provides learning in deep neural networks. In *Advances in Neural Information Processing Systems*, pages 1037–1045, 2016.

[19] Travis E Oliphant. *A guide to NumPy*, volume 1. Trelgol Publishing USA, 2006.

[20] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *International Conference on Machine Learning*, pages 1310–1318, 2013.

[21] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. *NIPS 2017 Autodiff Workshop*, 2017.

[22] Jack Rae, Jonathan J Hunt, Ivo Danihelka, Timothy Harley, Andrew W Senior, Gregory Wayne, Alex Graves, and Tim Lillicrap. Scaling memory-augmented neural networks with sparse reads and writes. In *Advances in Neural Information Processing Systems*, pages 3621–3629, 2016.

[23] Jeffrey Mark Siskind and Barak A Pearlmutter. Perturbation confusion and referential transparency: Correct functional implementation of forward-mode ad. 2005.

[24] Seiya Tokui, Kenta Oono, Shohei Hido, and Justin Clayton. Chainer: a next-generation open source framework for deep learning. In *NIPS 2015 LearningSys Workshop*, volume 5, 2015.

[25] Aäron van den Oord, Oriol Vinyals, and Koray Kavukcuoglu. Neural discrete representation learning. *CoRR*, abs/1711.00937, 2017. URL http://arxiv.org/abs/1711.00937.

[26] 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.

10