Theano: A CPU and GPU Math Compiler in Python

Theano is a compiler for mathematical expressions in Python. It combines the convenience of NumPy with the speed of optimized native machine language. The user writes mathematical expressions in a high-level description that mimics NumPy’s syntax and semantics, while being statically typed and purely functional. Theano optimizes the expression, translates the optimized expression into C/C++, and runs an optimizing compiler on the C/C++ code, all automatically. Theano can also generate CUDA code to produce GPU implementations for those high-level descriptions. Common machine learning algorithms are from 1.6× to 7.5× faster than competitive alternatives (including those in C/C++, NumPy, SciPy, and MATLAB) when compiled for the CPU and between 6.5× and 44× faster when compiled for the GPU. Theano’s speed comes from optimizing at a high level of granularity, namely over the symbolic graph representing a complicated mathematical expression. Theano’s speed on GPUs also comes from its ability to generate custom-made CUDA kernels for many important mathematical operations. Theano uses a library of graph transformation heuristics to optimize expression graphs for fast and numerically stable computation. Theano has been designed to hide the implementation details of a variety of backend technologies. With a single line of code, expressions in Theano can be automatically differentiated. These derivative expressions are also compiled by Theano. This paper illustrates how to use Theano, outlines the scope of the compiler, provides benchmarks on both CPU and GPU processors, and explains its overall design. Theano development and use began in January 2008.

Introduction

Python is a great language for describing large-scale mathematical calculations in a high level way, but the Python interpreter is not a good engine for executing them. Python numbers are full-fledged objects on the heap, but large-scale mathematical calculations should be carried out using a processor’s native types with minimal overhead. There are several options available to a Python programmer to get the speed of native machine-language code for numerical calculations, including [Cython], [numexpr], [Cython], and [scipy.weave]. [NumPy] provides an N-dimensional array data type, and many functions for indexing, reshaping, and performing elementary computations (exp, log, sin, etc.) on entire arrays at once. These functions are implemented in C for use within Python programs. However, the composition of many such NumPy functions inevitably leads to suboptimal performance, because there is no support for optimizations such as loop fusion or the amalgamation of several operations into an equivalent but more performant one. As a partial solution, [numexpr] provides faster implementations than NumPy when several elementwise computations are composed, by implementing a loop fusion optimization. However, numexpr requires writing the expression as a string, which can be cumbersome for complicated expressions, and it is limited to elementwise computations. In many use-cases, hand-written native machine language implementations are much faster than these common alternatives. [Cython] and [scipy.weave] make it easier to write just the crucial bottleneck code in C, and keep the rest of the program in Python. However, if the bottleneck is a large mathematical expression comprising hundreds of operations, manual optimization of the math can be time-consuming and suboptimal compared to automatic optimization.

Theano combines the convenience of NumPy with the speed of hand-optimized C/C++ code by generating and compiling native implementations from a
complete, high-level description of the computation graph. Theano optimizes this computation graph before translating it into C/C++ code. In our benchmarks, Theano’s optimization of this computation graph can lead to significant performance boosts over competing math libraries. Theano is capable of generating CPU as well as GPU implementations (the latter using CUDA), without requiring changes to user code. The end user is offered an interface similar to NumPy’s to declare variables and expressions involving those variables, creating a symbolic representation of the computations to perform. Any number of expressions can then be compiled into a highly optimized function taking numpy.ndarray instances as arguments and yielding the result(s) of the expression(s). The general principle of Theano is to describe calculations as a high-level mathematical graph. This symbolic description is separate from execution. Before executing the mathematical graph, the user must once compile it into a callable function. This callable function can then be executed repeatedly on a variety of inputs. There are several advantages to this approach: Theano lets user code stay clean, easy to read and familiar to NumPy users, while the ugly performance-oriented hacks are inserted automatically at compile time. For example, large expressions involving addition, subtraction, multiplication, and matrix products are rearranged to make the best use of low-level BLAS subroutines [BLAS]. Besides this user-facing advantage, there are backend advantages to separating the description of the calculations from their execution. Theano supports full program transformations and macros, which allows useful features such as automatic differentiation. Theano recognizes many unnecessary, sub-optimal, and numerically unstable expression patterns, and replaces them with superior alternatives. Theano’s expression graphs can contain information about the shape, number of dimensions as well as “broadcastability” of most variables, so Theano can specialize many operations. Theano is similar to [SymPy], insofar as both manipulate symbolic mathematical graphs. SymPy implements a more extensive set of mathematical operations, but it does not offer as efficient numerical evaluation.

Theano is designed for applications where the same computations will be repeated many times on different inputs, so that the time invested in the initial optimization and compilation (typically on the order of seconds) is negligible in comparison to time saved on the repeated calculations.

Theano is free open source software, licensed under the New (3-clause) BSD license. It depends upon NumPy, and can optionally use SciPy, as well as custom C and CUDA code generators which are able to specialize for particular types, sizes, and shapes of inputs. Theano can be extended to use scipy.weave, PyCUDA, Cython, and other numerical libraries and compilation technologies. Theano has been actively and continuously developed and used since January 2008. It has been used in several scientific papers. It was used to teach machine learning in graduate course IFT6266 at the Université de Montréal. We provide documentation and downloads on Theano’s website [theano]. All Theano users should subscribe to the announce mailing list (low traffic). There are medium traffic mailing lists for developer discussion [theano-dev] and user support [theano-users].

This paper is divided as follows: Section Using Theano walks through a simple example of how to use Theano, staying at a high level description. Section Case Study: Logistic Regression uses a less-trivial example to demonstrate automatic differentiation, GPU use, and some of the expression transformations Theano performs. Section Benchmarking Results presents some results of performance benchmarking on problems related to machine learning and expression evaluation. Section What’s in Theano gives an overview of the design of Theano. Section Limitations and Future Work outlines current limitations outlines planned future work.

Using Theano

There are four conceptual steps to using Theano: 1) declaring variables, 2) using these variables in expressions, 3) compiling these expressions into functions, and 4) calling these functions to perform the numerical computations of interest. These four steps are illustrated in the following simplistic example.

```
1: import theano
2: a = theano.tensor.vector('a')  # declare variable
3: b = a + a**10                 # build expression
4: f = theano.function([a], b)  # compile function
5: print f([0,1,2])             # call function
6: # prints 'array([0,2,1026])'
```

Listing 1: A simple Theano program.

Line 2 declares a Theano variable. Unlike Python’s variables, Theano’s variables and expressions are statically typed. For example, tensor type information includes the data type (int32, float32, float64, etc.), the number of dimensions (scalar=0, vector=1, etc.), and - for each dimension - whether it is broadcastable or not. Broadcasting in Theano is similar to that in NumPy, except that broadcasting decisions are made based on the type information rather than the shape of the actual ndarrays. A variable declared as theano.tensor.vector has one non-broadcastable dimension. In the example, `a` stands for a 1-dimensional ndarray of any number of elements of the default data type (float64), and even if that length is just one element, `a` will not be broadcasted to behave like a longer vector. Variable names are optional, but we have given our variable the name ‘`a`’. Theano variable names are purely decorative, they are used only by printing and visualization routines.
Line 3 builds the expression graph shown in Figure 1. The green rectangles in Figure 1 are the inputs: on the left is the a variable, and on the right is a constant (val=[10] in the label) that Theano has interpreted as a tensor (TensorType) of data type 'int8' that has one broadcastable dimension ([(True,)]). The integers on the arcs indicate the argument position (in the destination).

Line 4 transforms the expression graph to the one shown in Figure 2, compiles C++ code for the remaining expressions, and returns the result as a callable function f. The add and pow expressions are transformed into a single compound expression labelled `Elemwise{Composite{Composite{sqr,sqr,mul,sqr},add}}` that computes our expression with the single loop over a and no calls to pow. Notice that we used an optional mode argument to function to produce Figure 1. The mode argument controls which graph transformations to use, whether to use C, and how Theano will actually execute the computations. Figure 1 was produced with the 'FAST_COMPILE' mode which performs minimal graph transformations and ignores C++ implementations. Figure 2 was produced with the default 'FAST_RUN' mode which attempts many graph transformations and uses C++ implementations. There are other modes too: 'PROFILE_MODE' measures where Theano's compiled functions spend their time and 'DEBUG_MODE' runs many redundant calculations and verifications to detect potential errors introduced by graph transformations. Theano is verified nightly by thousands of unit tests, but it is still advised to run your functions in 'DEBUG_MODE' periodically (on small data because it is slow!) to be sure that your results are correct.

The last line of the example (print f([0,1,2])) calls our function f and computes our expression with the numerical values [0,1,2] provided as vector a. The types of the arguments and the return value of f are defined by the input variables and the output variables we provided to theano.function on Line 3. In our example, all computations are done on numpy.ndarray instances, one of which is returned to the user. Type conversions are performed automatically as necessary - the Python list of Python integers given in the example is internally converted to a numpy.ndarray of doubles, to match the declaration of variable a.

Case Study: Logistic Regression

The previous section introduces the basic approach to using Theano. This section describes a more involved example to illustrate some useful Theano features: shared variables, which make code more concise and make it easy to take advantage of an available GPU; automatic differentiation; and some graph transformations that bring numerical stability and speed improvements, even on a CPU.

Our example implements binary logistic regression, a probabilistic linear classifier often used in statistics and machine learning. This model is parametrized by a weight matrix W and bias vector b. For some choice of W and b, the model estimates the probability

\[ P(Y = 1|x) \]

(which we will denote with shorthand p) that the input x belongs to class y=1 as:

\[
\frac{e^{Wx^{(i)}+b}}{1+e^{Wx^{(i)}+b}}
\]  

We are given a set of N training examples, \( D = \{ (x^{(i)}, y^{(i)}), 0 < i \leq N \} \). Our goal is to pick W and b such that the model predicts \( p^{(i)} = 1 \) for examples with \( y^{(i)} = 1 \), and predicts 0 when \( y^{(i)} = 0 \). More precisely, we wish to minimize the average cross-entropy over D with an \( \ell_2 \) penalty on W, giving a cost function defined as:

\[
\text{cost} = 0.01 \cdot W^2 - \frac{1}{N} \sum_i (y^{(i)} \cdot p^{(i)} + (1 - y^{(i)}) \cdot (1 - p^{(i)}))
\]

Tuning parameters W and b to minimize this cost can be performed by more sophisticated algorithms, but for our example we will use stochastic gradient descent (SGD).

The code in Listing 2 implements this minimization with Theano. Since Theano functions involve some overhead, treating many examples in parallel is important for getting good performance from the model.
x is thus defined as a matrix, where each row is a training example, (Line 7) and the labels y as a vector (Line 8).

The shared() function (Lines 9+10 of Listing 2) creates shared variables for W and b and assigns them initial values. Shared variables are similar to standard Theano variables, but are stateful. In a sense, they behave like global variables which any Theano function may use without having to declare them in its inputs list. A shared variable’s value is maintained throughout the execution of the program and can be accessed for reading or writing using the .value attribute, as shown in Line 12. Theano manages the storage of these values. In particular, it stores single-precision dense shared tensors on the GPU by default when a GPU is available. In such cases it uses a different Theano-specific data type for internal storage in place of the NumPy ndarray.

Line 15 defines \( P(Y = 1|x^{(i)}) = 1 \) as \( p_1 \). Line 16 defines the cross-entropy term in cost as \( \text{xent} \). Line 17 defines the predictor by thresholding over \( P(Y = 1|x^{(i)}) = 1 \) as \( \text{prediction} \). Line 18 defines cost as \( \text{cost} \), by adding the cross-entropy term to the \( \ell_2 \) penalty. Line 19 \((gw, gb = T\text{.grad}(cost, [w,b]))\) performs automatic differentiation of scalar-valued cost with respect to variables \( w \) and \( b \). It works like a macro, iterating backward over the expression graph, applying the chain rule of differentiation and building expressions for the gradients on \( w \) and \( b \).

Lines 22-25 (\( \text{train} = \text{function}... \)) introduce the updates argument to function. An update is an expression that will be computed by the function, like a return value, but the computed result is stored in a shared variable instead of returned to the caller. On a GPU, this means that a shared variable and its updated value can all reside on the device. Having both on the device can be important for performance, because it is slow to copy between the host and the GPU. Here we adjust \( w \) and \( b \) by their gradients, the direction that causes the cost to drop most sharply. This update step implements stochastic gradient descent.

Line 26 compiles a second function (\( \text{predict} = \text{function}... \)) from the same expression graph. This is a standard pattern when using Theano - we define one big expression graph that corresponds to some application domain, and then compile several functions from it to compute various sub-regions of the graph. Note that all these functions may read and write the states of the various shared variables, hence their name.

Lines 28-30 randomly generate four training examples, each with 100 feature values. (In practice, training examples would be inputs to the program.) Line 31-33 runs the \( \text{train} \) gradient update step, ten times. Lines 34-41 print some debug output.

Theano applies some graph transformations to optimize the \( \text{train} \) and \( \text{predict} \) functions for speed and numerical stability, when compiling them in Lines 22-25 and 26, respectively. For example, in the \( \text{predict} \) function, \( P((1+exp(-u)) \) is recognized as the logistic sigmoid function and replaced with an implementation that is faster for large positive and negative values of \( u \). All the element-wise operations are fused together after the vector-matrix multiplication and compiled as a specialized C function with a single loop over the data. In the \( \text{train} \) function, Theano additionally recognizes \( \log(\text{sigmoid}(u)) \) and \( \log(1-\text{sigmoid}(u)) \) as instances of the softplus function: \( \text{log1p}(\exp(u)) \), for which Theano has an implementation that avoids a dangerous potential overflow. When updating \( w \) with its new value, Theano also recognizes that a single call to the BLAS \( \text{dgemv} \) routine can implement the \( \ell_2 \) regularization of \( w \), scale its gradient, and decrement \( w \) by its scaled gradient.

**Benchmarking Results**

Theano was developed to allow the rapid development of algorithms in machine learning. This section presents performance in two tasks from that domain: the training of a multi-layer perceptron (MLP) and a convolutional network. More extensive benchmarks are forthcoming, and will be posted on our website.

We chose these architectures because of their popularity in machine learning and their different computational demands. Large matrix-matrix multiplications dominate in the MLP example, and two-dimensional image convolutions with small kernels dominate computations in the convolutional network. More information about these models and their learning algorithms is available from the Deep Learning Tutorials.
The implementations used in these benchmarks are available online [dlb]. CPU timing was carried out on an Intel(R) Core(TM)2 Duo CPU E8500 @ 3.16GHz with 2 GB of RAM. All implementations were linked against the BLAS implemented in the Intel Math Kernel Library, version 10.2.4.032 and allowed to use only one thread. GPU timing was done on a GForce GTX 285. CPU computations were done at double-precision. GPU computations were done at single-precision.

Our first benchmark is training a single layer MLP by mini-batch gradient descent. Each implementation multiplied 60 784-element input vectors by a 784 \times 500 weight matrix, compressed by a tanh function, then multiplied by a 500 \times 10 matrix, and finally classified using a multi-class generalization of logistic regression. The gradient was calculated by performing similar calculations, but in reverse.

Figure 3: Fitting a multi-layer perceptron to simulated data with various implementations of stochastic gradient descent. These models have 784 inputs, 500 hidden units, a 10-way classification, and are trained 60 examples at a time.

Figure 4: Fitting a convolutional network using different software. The benchmark stresses convolutions of medium-sized (256 by 256) images with small (7 by 7) filters.

Figure 3 compares the number of examples processed per second by different implementations. We compared Theano (revision #ec057beb6c), NumPy 1.4.1, Matlab 7.9.0.529, and Torch 5 (a machine learning library written in C/C++) [torch5]. On the GPU we compared Theano with GPUMat 0.25 for Matlab ([gpumat]). As shown in Figure 3, on the CPU Theano is 1.8x faster than NumPy, 1.6x faster than Matlab, and 7.5x faster than Torch 5. Torch was written for flexibility, not speed (Ronan Collobert, p.c.). Theano’s speed increases 5.8x on the GPU from the CPU, a total increase of 11x over NumPy on the CPU and 44x over Torch 5 on the CPU. GPUMat increases the Matlab speed on the GPU only 1.4x from the CPU, far less than the 5.8x increase Theano achieves through CUDA specializations.

What’s in Theano?

This section gives an overview the design of Theano. A Theano expression graph is a bi-partite directed acyclic graph. It is bi-partite because there are two kinds of nodes: variable nodes are the inputs to and outputs from apply nodes. A variable node represents input or an intermediate mathematical result. It has a Type (.type) that signals the sort of value the variable might take at runtime. An apply node represents the application of the Op (.op) to some input.
variables (.inputs) producing some output variables (.outputs). Figures 1 and 2 have been simplified for clarity. Technically, there is an intermediate result for the output of the Elemwise(pov, no_inplace), and the variable nodes (box) and apply nodes (ellipse) are distinct from the Type and Op instances respectively (not shown) that give them meaning.

Variables

Theano supports three kinds of variable nodes: Variables, Constants, and Shared variables. Variable nodes (with a capital V) are the most common kind—a Variable is either found as a leaf of the graph (if it was created explicitly with a call like theano.tensor.vector()), or as the output of an apply node (if it was defined by the application of an Op). In the latter case, the Variable will have a .owner attribute pointing to the apply node. a and b in Listing 1 are Variables (without .owner). p_1 in Listing 2 is also a Variable (with .owner). theano.function takes two arguments: the input list, which is a list of Variables; and the output value or list, which is a Variable or list of Variables. Constant nodes each have a .value attribute, which is the immutable (read-only) value of this variable. 10 in Listing 1 was converted to a Constant node. Shared Variable nodes each have a mutable (read and write) .value attribute. This value can be modified by calling a Theano function that was defined with updates, like train in Listing 2. When a Shared variable is stored on the GPU, reading and writing its .value attribute transfers data to and from the GPU device.

Types

The important variable Types in Theano are:

- **TensorType** - denotes a numpy.ndarray with specific number of dimensions, a record of which of these dimensions are broadcastable, and dtype. The dtype is the data types, e.g. int32, float64, etc.
- **SparseType** - denotes one of the csr or csc formats in scipy.sparse.
- **RandomStateType** - denotes a NumPy RandomState object. They are rarely used directly by Theano user code. They are storage containers for the random number generator.
- **Generic** - denotes any Python value. They are rarely used directly by Theano user code. Generic Variables exist mainly for Ops to be able to allocate workspace outputs.

Theano types are often stricter than their NumPy/SciPy equivalents. For example, there are different versions of SparseType in Theano, which are specific to different encodings like csr or csc. The Theano TensorType that corresponds to a numpy.ndarray also specifies the number of dimensions (scalar=0, vector=1, etc.), which of them are broadcastable, and what dtype should be used. This information is used when performing graph transformations.

For Shared Variables and Constants, the type is inferred automatically based on the value given during initialization.
Ops & Functionality

*Ops* are objects that define computations. Most of the ops (e.g. *add*, *exp*) behave like NumPy counterparts. Table 1 lists the core functionality offered by Theano's Ops. More extensive reference documentation is available online [theano].

Allocating random number variables and seeding generators is typically done via a `RandomStreams` instance, which replicates the `numpy.random.RandomState` interface and wraps `numpy.random.RandomState` functionality. Theano also provides an experimental new `MRG_RandomStreams` generator which provides a few distributions using an MRG algorithm with both a CPU and GPU implementation [Ecu].

There is a narrower range of Ops that work on Sparse-Type Variables: packing and unpacking of compressed sparse row/column sparse matrices into dense variables is supported, as is conversion between sparse and dense matrices. Transpose, negation, addition, and subtraction are supported. Scalar and elementwise multiplication with a dense matrix is supported, and matrix multiplication between sparse and dense is supported. Roughly 90% of Ops for tensors have implementations for the GPU, notable exceptions being advanced indexing, scan, summation over certain combinations of axes, and reductions max, min and prod. Our goal is extend coverage to all ops.

Theano does not currently have ops for sparse or dense matrix inversion, nor linear algebra decompositions. Ops for complex number dtypes are not as widely implemented or well-tested as those for integer and float dtypes. Object dtypes are not implemented in Theano.

Transformations

Theano uses graph transformations to implement a range of tasks from merging redundant calculations to transferring computations to the GPU. The optimization of expression graphs is carried out several stages.

The first stage removes duplicate expressions, and when several constants are actually equal, they are replaced with a single node. Theano treats two apply nodes with the same inputs and the same Op as being duplicates and only keeps one. The automatic gradient mechanism often introduces this sort of redundancy, so this phase is quite important. The 'FAST_COMPILE' mode includes only this stage.

The second stage transforms the graph into an equivalent, canonical form, so that subsequent patterns do not have to recognize as wide a variety of equivalent expressions. For example, expression subgraphs involving just multiplication and division are put into a standard fraction form (e.g. $a \div (b \times c \div d) \rightarrow (a \times d) / (b \times c)$), and terms in both numerator and denominator are cancelled.

The third stage replaces expressions to improve numerical stability. The logistic sigmoid substitution described at the end of Section Case Study: Logistic Regression is an example. After numerically unstable subgraphs have been replaced with more stable ones, Theano pre-calculates expressions involving only constants.

The fourth stage specializes generic expressions and subgraphs. Expressions like $\text{pow}(x, 2)$ become $\text{sqr}(x)$. Theano also performs more elaborate specializations: expressions involving scalar-multiplied matrix additions and multiplications may become BLAS General matrix multiply (GEMM) nodes, sums of incremented tensors become incremented sums, and *reshape*, *dimshuffle*, and *subtensor* Ops are replaced by constant-time versions that work by aliasing memory.

After this stage of specialization, Elementwise sub-graphs are fused into Compound ones that permit loop fusion (such as the `Elemwise(Composite(....))` Op in Figure 2). If Theano is using a GPU, Ops with corresponding GPU implementations are substituted in.

Lastly, Theano replaces Ops with equivalents that reuse the memory of their inputs and also invalidate those inputs by side-effect of running. Many Ops (e.g. GEMM and all elementwise Ops) have such equivalents. Reusing memory this way can improve speed by reducing CPU cache misses.

Code Generators

Many (roughly 80%) of Theano’s Ops generate and compile C or CUDA code during theano.function. The majority of Ops (such as all elementwise Ops and Sum) that generate C code specialize the code based on the dtype and number of dimensions of their arguments. Some Ops, such as the small-filter convolution (`conv2d`), further specialize code based on the size the arguments will have.

Modern x86 architectures are relatively forgiving if code is not perfectly specialized to the input dimensions, and only the `conv2d` Op goes to any great length to generate many special case implementations for the CPU. By comparison, GPU architectures are much less forgiving of code that is not carefully specialized for the size and physical layout of function arguments. Theano’s code generators for `GpuSum`, `GpuElementwise`, and `GpuConv2d` generate a wider variety of implementations than their respective CPU-targeting Ops. The difference in speed on a GPU between a naive and an optimal implementation of even a simple algorithm like row/column summation in a matrix can be an order of magnitude or more. Theano’s ability to generate custom-made CUDA kernels for many important mathematical operations accounts for the good GPU performance in our benchmarks.

Limitations and Future Work

Theano does not make significant efforts to optimize the compilation process itself. Theano can take up to a few seconds to construct a Theano function (especially when it must compile freshly-generated C code),
even when a naïve implementation of the function’s expression would require only a fraction of a second. So Theano takes time when creating Theano functions, which is not the case for libraries such as NumPy and SciPy whose functions have already been compiled. Theano is therefore suited to applications where a function will be called enough times that the time spent on the initial compilation is negligible. Theano has been tested primarily with graphs from 10-1000 nodes, which is sufficient for many algorithms. The time spent on applying graph transformations tends to grow super-linearly with the size of the expression graph. Beyond a few thousand nodes, Theano’s optimization algorithm can be impractically slow, unless you disable some of the more expensive optimizations, or compile pieces of the graph separately.

A Theano function call also requires more overhead (on the order of microseconds) than a native Python function call. For this reason, Theano is suited to applications where functions correspond to expressions that are not too small (see Figure 5).

The set of Types and Ops that Theano provides continues to grow, but it does not cover all the functionality of NumPy and covers only a few features of SciPy. Wrapping functions from these and other libraries is often straightforward, but implementing related graph transformations and implementing Ops for gradients can be more difficult. We expect to improve support for advanced indexing and linear algebra in the coming months. Documentation online describes how to add new Ops, Types, and transformations.

Theano’s graph transformations give good results for expressions related to machine learning with neural networks, but they are not as well tested outside that domain. Theano is not a powerful computer algebra system, and it is an important area of future work to improve its ability to recognize numerical instability in complicated elementwise expression graphs.

Debugging Theano functions can require non-standard techniques and Theano-specific tools. The reason is two-fold: 1) definition of Theano expressions is separate from their execution, and 2) optimizations can introduce many changes to the computation graph.

Conclusion

Theano is a mathematical expression compiler for Python that translates high level NumPy-like code into machine language for efficient CPU and GPU computation. Theano achieves good performance by minimizing the use of temporary variables, minimizing pressure on fast memory caches, making full use of gemm and gemv BLAS subroutines, and generating fast C code that is specialized to sizes and constants in the expression graph. Theano implementations of machine learning algorithms related to neural networks on one core of an E8500 CPU are up to 1.8 times faster than implementations in NumPy, 1.6 times faster than MATLAB, and 7.6 times faster than a related C++ library. Using a Nvidia GTX285 GPU, Theano is 5.8 times faster again. One of Theano’s greatest strengths is its ability to generate custom-made CUDA kernels, which can not only significantly outperform CPU implementations but alternative GPU implementations as well.

Acknowledgements

Theano has benefited from the contributions of many members of Yoshua Bengio’s machine learning group in the computer science department (Informatique) at the University of Montreal, especially: Arnaud Bergeron, Thierry Bertin-Mahieux, Olivier Delalleau, Douglas Eck, Dumitru Erhan, Philippe Hanel, Simon Lemieux, Pierre-Antoine Manzagol, and François Savard. David Warde-Farley contributed to the preparation of this paper. The authors acknowledge the support of the following agencies for research funding and computing support: NSERC, RQCHP, CIFAR, SHARCNET and CLUMEQ.

References

Theano, http://www deeplearning net/software/theano
Cython S. Behnel, R. Bradshaw, and D. S. Seljebotn, Cython C-Extensions for Python, http://www.cython.org/
SymPy http://code.google.com/p/sympy/
EILearn: Energy Based Learning, http://eblearn.sourceforge.net/