Reverse-Mode Automatic Differentiation in Haskell Using the Accelerate Library

JAMES BRADBURY, Stanford University
FARHAN KATHAWALA, Stanford University

Automatic Differentiation is a method for applying differentiation strategies to source code, by taking a computer program and deriving from that program a separate program which calculates the derivatives of the output of the first program. Because of this, Automatic Differentiation is of vital importance to most deep learning tasks as it allows for the easy backpropogation of complex calculations in order to minimize the loss of a learning algorithm. Most of these calculations are some variant of composed matrix and vector operations (such as matrix multiplication or dot products), and because Haskell, and the functional programming paradigm as a whole, stresses the easy composition of pure functions, the task of providing a good Automatic Differentiation implementation for deep learning tasks becomes much easier when using Haskell. Our work provides a functioning Automatic Differentiation library for use with the Accelerate library which implements matrix operations.

Additional Key Words and Phrases: Deep Learning, NVIDIA, CUDA, CUBLAS, Abstract Syntax Trees

1. INTRODUCTION

Andreas Griewak in 1989 noted that [Griewank 1989]

> "Reverse Accumulation yields truncation error free gradient values at less than 5 / *n* times the computing time of divided differences.

Thus, we have known for a long time that Automatic Differentiation is by far a better approach than calculating out complex derivatives by hand and applying this derivative to each scalar function whenever our learning computation function changes because Automatic Differentiation is error-free and takes less time than manually computing the gradient might take.

Approaches to Automatic Differentiation exist in many forms, the most well known of which is the Theano library in the Python programming language [Bastien et al. 2012], but these implementations often require workarounds in order to do things not supported by the underlying programming language. For example, the lack of lazily evaluated functions (also called thunks) in Python, requires the Theano library to undergo extra compilation steps in order to have a sufficiently quick differentiation strategy.

Haskell provides a number of features suited to the task at hand such as

Lazy Evalutation. The ability to chain together many functions and delay their computation is critical as it necessitates the creation of and ability to traverse a computation graph. This graph of the computations of a given program is what we utilize to perform automatic differentiation on any program of any variation of matrix and vector operation composition.

Tools For Transforming a Computation Graph. Haskell provides Generalized Abstract Data Types (GADTs) as a language extension, which allow you to create more powerful data types, and as such this allows for the easy creation of Abstract Syntax Trees which can easily be made into computations.

Available Libraries. The Accelerate library implements an Abstract Syntax Tree type for all matrix operations which we can leverage to traverse the computation graph and from that easily compose a differentiated computation graph. Using other libraries in the Haskell ecosystem, Accelerate is able to then send this entire computation graph as one kernel to the GPU of a system for quick calculation. These atomic calculations defined by the GPU allow differentiation of matrix operations to be done cleanly and quickly.

Utilizing these features we are able to present a small step towards a unified deep-learning framwork in Haskell. Using the Accelerate library to leverage matrix and vector operations, we provide the backpropogation ability and future work may be able to combine these elments into a larger package for one-stop deep learning capabilities.

2. IMPLEMENTATION

Our implementation can be broken down into the following steps

2.1. Binding to CUBLAS

GPUs allow for atomic matrix and array operations which can be implemented in the Accelerate library by utilizing the GPU of a system with the CUDA architecture. To implement such functions as matrix multiplication (called "gemm") and vector addition (called "axpy"), it was necessary to interface with the CUDA version of the Basic Linear Algebra System (called CUBLAS). Using the Foreign Function interface of the Accelerate we were able to directly call CUBLAS functions as long as we implemented a pure, functional implementation of the function for use when a supported GPU is not present on a system [Clifton-Everest et al. 2014]. This also required keeping track of device pointers and GPU execution streams, so below we show a cleaned-up, with some details omitted for brevity, example of the gemv (matrix-vector multiplication) implementation we used

```
pureGemv :: (Matr,Vect) -> Vect
pureGemv (arr,brr)  = slice result (lift $ Z:.All:.0)
  where
    result          = (fold (+) 0 $ zipWith (*) arrRepl brrRepl)
    bLen            = length brr
    Z :. rowsA :. _ = unlift (shape arr) ::Z:.Exp Int:.Exp Int

    arrRepl         = replicate (lift $ Z:.All:.1:.All) arr
    brrMat          = reshape   (lift $ Z:.bLen:.1) brr
    brrMatT         = transpose brrMat
    brrRepl         = replicate (lift $ Z:.rowsA:. All:.All) brrMatT

cudaGemvF :: (Matr, Vect) -> CIO (Vect)
cudaGemvF (a,b) = do
  let Z :. ra :. ca = arrayShape a   -- m k
      Z :. rb       = arrayShape b   -- k n
  c <- allocateArray $ Z :. ra -- m n
  withDevicePtrs a ms $ \aptr -> do
    withDevicePtrs b ms $ \bptr -> do
      withDevicePtrs c ms $ \cptr -> do
        liftIO $ BL.gemm 1 ra rb 1 bptr0 1 aptr0 ca 0 cptr0 1
        return c

  where bptr0 = castDevPtr bptr
              aptr0 = castDevPtr aptr
              cptr0 = castDevPtr cptr
```

```
gemv :: (Matr, Vect) -> Vect
gemv (v1,v2) = foreignAcc cudaGemv pureGemv $ lift (v1,v2)
  where cudaGemv = CUDAForeignAcc "cudaGemvF"
```

2.2. Retrieving the Abstract Syntax Tree

Once we can provide a few matrix and vector operations, the user will specify a program which composes these calculations on some set of inputs with a specific input to be optimized, in most cases of deep learning algorithms this input is the weight vector. Given this user-defined program, we need to retrieve the Abstract Syntax Tree representation of this program in order to perform Automatic Differentiation on the program. The Accelerate library uses the Generalized Abstract Data Type (GADT) extension to the Haskell language to create Abstract Syntax Trees in Higher-Order-Abstract-Syntax (HOAS) [Chakravarty et al. 2011]. This use of GADTs and Abstract Syntax Trees in HOAS makes it possible to traverse the tree using pattern matching for all the possible computations and expressions.

2.3. Calculating the Differentiation

To do this, we need to ask the question "What are we taking the derivative with respect to?". In any kind of machine learning algorithm, we are taking the derivative of the input which we want to improve, so as to improve the learned computation. This means that we want to take the derivative with respsect to the weight vector/matrix of the computation. With this in mind, we traverse the user-defined program's Abstract Syntax Tree and tag the weight parameters as we come across them. Then, we create out own Abstract Syntax Tree in a similar method to the one used by the Accelerate library, but our tree starts with the derivative of the root of the user-defined tree and reverses its way to the top. One can consider that if a computation in the original tree looks like

$$y = f(x) \tag{1}$$

then obviously the correlated computation in the derived tree would be

$$dx = f'(dy) \tag{2}$$

Similarly, we can think about the derivatives of matrix/vector operations easily, using Haskell as such. In this manner, the following function

$$z = \textbf{dot}\ x\ y \tag{3}$$

can be derived as

$$(dx, dy) = (\textbf{map}\ (dz\,*)\ y, \textbf{map}\ (dz\,*)\ x) \tag{4}$$

We go through the tree making conversions as such and composing them together until we run into a function which has as one of its inputs any of the weight inputs we tagged earlier. If we encounter such a function, then we store the entire graph up to and including that derivative to evaluate later because, just as the input for Eq. 1 was x and the output of Eq. 2 was dx, so too the output of the stored graph will be the derivative of the tagged weight input. Doing this for the whole graph completes the Automatic Differentiation.

3. SUMMARY

In short, the library we produced can be seen as the first stepping-stone towards bridging the similar paradigms of functional programming and deep learning. By taking advantage of function composability, GADTs, Abstract Syntax Trees, and the lazy evaluation offered by Haskell, we were able to provide a limited, albeit sufficient for some types of deep learning, library for Automatic Differentiation of any arbitrary, varied composition of matrix and vector operations such as matrix-vector multiplication, vector-matrix multiplication, outer product, dot product, and vector addition. Future work in cleaning up and fully implementing these functions may allow us to release this library as a package compatible with the Haskell Accelerate library.

REFERENCES

Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, James Bergstra, Ian Goodfellow, Arnaud Bergeron, Nicolas Bouchard, David Warde-Farley, and Yoshua Bengio. 2012. Theano: new features and speed improvements. *arXiv preprint arXiv:1211.5590* (2012).

Manuel MT Chakravarty, Gabriele Keller, Sean Lee, Trevor L McDonell, and Vinod Grover. 2011. Accelerating Haskell array codes with multicore GPUs. In *Proceedings of the sixth workshop on Declarative aspects of multicore programming*. ACM, 3–14.

Robert Clifton-Everest, Trevor L McDonell, Manuel MT Chakravarty, and Gabriele Keller. 2014. Embedding foreign code. In *Practical Aspects of Declarative Languages*. Springer, 136–151.

Andreas Griewank. 1989. On automatic differentiation. *Mathematical Programming: recent developments and applications* 6, 6 (1989), 83–107.