AI Education Matters: Model AI Assignment "ScalarFlow: Implementing Reverse Mode Automatic Differentiation"

Nathan Sprague (James Madison University; spragunr@jmu.edu)

DOI: 10.1145/3516418.3516422

Introduction

Many of the most significant breakthroughs in artificial intelligence over the past decade have been based on progress in deep neural networks. That progress has been facilitated by deep-learning libraries like Theano (Al-Rfou et al., 2016), TensorFlow (Abadi et al., 2015) and PyTorch (Paszke et al., 2019) that allow rapid prototyping and efficient execution. The key algorithm at the heart of all of these libraries is *reverse-mode automatic differentiation*. This column introduces the Model AI Assignment ScalarFlow: Implementing Reverse Mode Automatic Differentiation. This assignment gives students the opportunity to gain a deeper understanding of modern deep-learning frameworks by building their own automatic differentiation engine and using it to experiment with some important concepts in deep learning.

In this column we will review some basic background on training neural networks, provide a brief overview of the reverse-mode automatic differentiation algorithm, describe the model assignment and provide some pointers to additional resources.

Traditional Backpropagation

In a multi-layer neural network, an input vector is passed through one or more hidden layers composed of simple computational units before being passed to a final output layer. Each internal unit in the network performs the same basic operations: inputs are multiplied by corresponding adjustable weights, the results are summed, and that sum is passed though a nonlinearity. Outputs from each unit then become the inputs for the units in the subsequent layer. Figure 1 illustrates the structure of a traditional three-layer network.

The goal in training a network of this sort is to find a set of weight values that minimizes a loss function describing how dissimilar the network outputs are from the provided labels in a set of training data. Since there is no closed-form solution for finding an optimal set of weights, neural networks are trained iteratively using gradient descent.

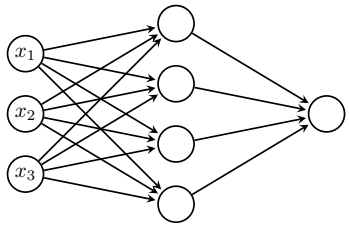

Figure 1: Three-layer network.

An important step in the history of neural networks was the invention of the backpropagation algorithm for efficiently calculating gradients in a multi-layer neural network. The backpropagation algorithm was independently invented several times, but gained popularity after its publication in the Parallel and Distributed Processing (PDP) anthology (McClelland & Rumelhart, 1986). The backpropagation algorithm is the traditional approach to training neural networks that has been presented in classrooms and textbooks for decades: an explicit set of weight update rules based on an analytical derivation of the gradient of the loss function in a multi-layer neural network.

Automatic Differentiation

Theano was the first widely-used software package to popularize an alternative approach to building and training neural networks. In this alternative paradigm, network components are represented as a computation graph built up from low-level mathematical operators. As a simple example, the formula $L(w, x, y) = (wx - y)^2 + w^2$ can be represented using the graph illustrated in Figure 2.

The output of the network is calculated by setting the value of the input nodes and

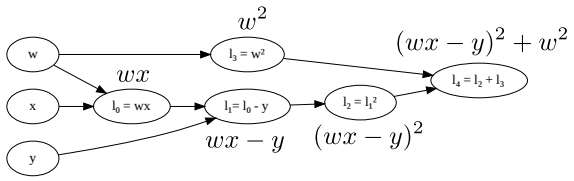

Figure 2: Example computation graph.

then iterating forward through the graph, performing the appropriate computation at each node. Since the individual nodes represent differentiable operations, it turns out to be straightforward to calculate partial derivatives at each node by iterating backwards through the graph. At each node we calculate the derivative with respect to the operands. By following the chain rule, we can determine the derivative at each node by simply multiplying these local derivatives backward through the graph. The result is a generalization of the traditional backpropagation algorithm that can be applied to *any* machine learning model that is built from differentiable operators.

Figure 3 illustrates one step in the the backward pass when calculating $L(1, 2, 3)$:

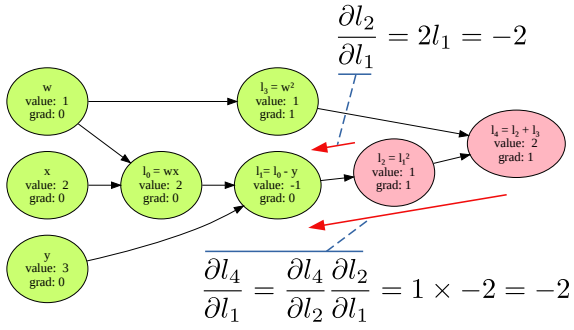

Figure 3: One step in the backward pass.

Here, the `value` term at each node represents the results stored during the forward pass. In this example, we have already calculated the derivatives for l_4 and l_2 (in red) and are working on the derivative for node l_1. The arrow labeled $\frac{\partial l_2}{\partial l_1}$ represents the "local" partial derivative of l_2 with respect to l_1. The arrow labeled $\frac{\partial l_4}{\partial l_1}$ represents the derivative value that will actually be stored at node l_1. The process is the same for each step of the backward pass: at each node we calculate the local partial derivative of the node with respect to its operands, then "pass back" those derivative values multiplied by the previously calculated derivative of the current node.

The appeal of building machine learning models within this framework is that the programmer only needs to specify the structure of the model. All of the logic for calculating the necessary gradients is handled automatically and efficiently by the library. This makes it possible to quickly prototype new models without the need to re-derive and re-implement rules for weight updates.

Modern libraries address significant complexities that don't arise in the example above. For one thing, they don't just handle scalar-valued computations, but tensor-valued computations where the values passed between nodes are multi-dimensional tensors, representing, for example, batches of multi-channel image data. Another important feature of modern libraries is that they take advantage of GPU operations and highly optimized CPU libraries when available. This makes it possible to write a machine learning model in Python and have it transparently take advantage of parallelized operations on a GPU.

ScalarFlow Assignment Part I - Autodiff Library

The model assignment[1] involves adding functionality to **ScalarFlow**, a minimalist automatic differentiation library that (as the name suggests) only supports scalar-valued nodes. Avoiding vector-valued or tensor-valued quantities significantly simplifies the implementation while preserving the key ideas of the algorithm. The provided starter code includes all of the functionality necessary to build a computation graph, but none of the code required to perform the calculations for the forward or backward pass. The main requirement of the assignment is for students to add this missing functionality.

Figure 4 provides an example showing the Python code that would be used to build and execute the computation graph from Figure 2.

Automatic differentiation libraries are characterized as either *define-and-run* or *define-by-run*. In the former, the user explicitly builds

[1] http://modelai.gettysburg.edu/2021/scalarflow/

```
import scalarflow as sf

graph = sf.Graph()

with graph:
    # Input nodes
    w = sf.Variable(1.0, name='w')
    x = sf.Variable(2.0, name='x')
    y = sf.Variable(3.0, name='y')

    # Computation graph
    node0 = sf.Multiply(w, x)
    node1 = sf.Subtract(node0, y)
    node2 = sf.Pow(node1, 2)
    node3 = sf.Pow(w, 2)
    node4 = sf.Add(node2, node3)

# The lines below will only
# work *after* the student
# completes the assignment!
result = graph.run(node4,
    compute_derivatives=True)

print(node4.value) # Prints "2.0"
print(w.grad)      # Prints "-2.0"
```

Figure 4: Example Python code using the ScalarFlow library.

and executes a computation graph, as in the example in Figure 4. In the latter, the user develops and executes their model using library operations and the computation graph is implicitly constructed as a side effect. The define-by-run model tends to be more convenient for the programmer and is now the default for both TensorFlow and PyTorch. ScalarFlow follows the define-by-run model with the hope that the more explicit approach will be easier for students to understand and implement.

ScalarFlow Assignment Part II - Machine Learning

The starter code provided to students also includes a simple machine-learning library built on top of ScalarFlow. The library provides a logistic regression classifier and a simple, two-class, feed-forward neural network. In this second stage of the assignment students are asked to modify the neural network implementation to enable the use of rectified linear units, or ReLU's. The ReLU nonlinearity is defined as $f(x) = \max(0, x)$.

One of the historical stumbling blocks in the development of deep neural networks was the problem of *vanishing gradients*. It turns out that traditional sigmoid nonlinearities are particularly prone to the problem of vanishing gradients because their derivative is near zero across much of their domain. ReLU's tend to be much more resistant to vanishing gradients because they have a large area with non-zero gradients.

Once students have added functionality for ReLU nonlinearities they are required to write a short report investigating the relative performance of sigmoid vs. ReLU nonlinearities as the number of network layers increases in a small-scale synthetic classification task.

There are two main learning objectives for this second stage of the assignment. First, it helps students to understand the significant impact of nonlinearity selection on the problem of vanishing gradients. Second it provides an opportunity for students to actually engage with ScalarFlow as part of a complete machine learning pipeline. The hope is that this will allow them to gain an appreciation for the connection between a low-level automatic differentiation engine and high-level machine learning algorithms.

Additional Resources

A good academic survey of automatic differentiation in machine learning is provided by Baydin et. al. (Baydin, Pearlmutter, Radul, & Siskind, 2018). There also are a number of online sources that provide tutorial introductions to the reverse-mode automatic differentiation algorithm. These include Automatic Differentiation, Explained[2] by Chi-Feng Wang, this helpful stats.stackechange.com answer[3], and Reverse-mode automatic differentiation: a tutorial[4]

Andrew Ng has a short video[5] describing the basics of finding derivatives using a

[2] https://towardsdatascience.com/automatic-differentiation-explained-b4ba8e60c2ad

[3] https://stats.stackexchange.com/a/235758

[4] https://rufflewind.com/2016-12-30/reverse-mode-automatic-differentiation

[5] https://youtu.be/nJyUyKN-XBQ

computation graph. Mu Li has a longer video[6] introduction that extends the basic idea of reverse-mode automatic differentiation to vector-valued quantities. During previous iterations of this assignment I have shared a short video[7] that describes the algorithm and works through a complete forward and backward pass using the graph in Figure 2.

Another potentially useful resource is Autodidact: a pedagogical implementation of Autograd[8]. This is another minimalist version of automatic differentiation that is implemented as a thin wrapper around existing numpy operations.

References

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., ... Zheng, X. (2015). *TensorFlow: Large-scale machine learning on heterogeneous systems.* Retrieved from https://www.tensorflow.org/

Al-Rfou, R., Alain, G., Almahairi, A., Angermueller, C., Bahdanau, D., Ballas, N., ... Zhang, Y. (2016, May). Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, *abs/1605.02688*.

Baydin, A. G., Pearlmutter, B. A., Radul, A. A., & Siskind, J. M. (2018). Automatic differentiation in machine learning: a survey. *Journal of machine learning research*, *18*.

McClelland, J. L., & Rumelhart. (1986). *Parallel distributed processing* (Vol. 2). MIT press Cambridge, MA.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., ... Chintala, S. (2019). Pytorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems 32*.

[6] https://youtu.be/RP0JScZG6gA

[7] https://youtu.be/EEbnprb_YTU

[8] https://github.com/mattjj/autodidact

Nathan Sprague is an associate professor of Computer Science at James Madison University. His research and teaching interests are focused in the areas of machine learning and robotics.