Paragraph

A Backpropagation engine
in C++

Parakram Majumdar

August 4, 2018

1 Introduction

The *training* of Artificial Neural Networks involves calibrating the *weights* of each neuron so that the *output* generated by the neural network matches the expected output of a *training set*. Most training algorithms employ some form of *gradient descent*, wherein each weight is adjusted in the opposite direction of the partial derivative of the error with respect to that weight.

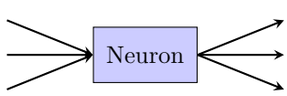
$$\Delta w = -c \times \frac{\partial o}{\partial w}$$

The *backpropagation* algorithm employs *algorithmic differentiation* to compute the gradient, and is especially common for training *deep* neural networks.

This document describes the anatomy of *Paragraph*, a backpropagation engine written in C++.

2 Basic Concepts

A neural network consists of interconnected neurons, where each neuron, based on the input signals received from input connections, sends an output signal to output connections.



Thus, the network can be considered a directed computational `Graph`, where each `Node` represents a neuron. Each node is one of two types :

- `Operation`: Like neurons, operations react to their inputs to produce an output.
- `Variable`: Variables act as "input nodes", and do not have any inputs of their own. They allow the graph to be manipulated by external systems.

The nodes communicate to each other via `Tensor`s, or multi-dimensional arrays of real numbers (`double`s). The tensors also generalize quite well to represent gradients of multi-dimensional functions. These gradients can then be used to calibrate the variables in a graph.

3 Tensors

`Tensor`s are multi-dimensional arrays of real numbers. An n-dimensional tensor has fixed dimensionalities $\{d_1, d_2, ..., d_n\}$, which drive the coordinates of each number stored in the array.

Common operations on tensors are described in the TensorFunctions document. A notable operation is the *tensor multiplication*, a generalization of matrix multiplication, which is key in determining the gradient with respect to a node, given the gradients of its outputs. For example, suppose we have a node g which takes another node f as input, then

$$\nabla(g \circ f) = \nabla g \cdot_n \nabla f$$

Where $\cdot_n$ is the tensor multiplication operator on n dimensions, n being the dimesionality of f.

Many useful computations on tensors can work in an iterative fashion, traversing the elements of the tensor in a row-major ordering. This means that storing the elements in contiguous memory in row-major ordering allows for noticable speed-ups from the co-location of data. With most modern operating systems providing virtual memory mappings, simply `alloc`ating large contiguous chunks of memory is a viable option.

Furthermore, it is also critical that tensors allow for sparse representation of the data. A typical real life deep learning network will consist of many sparse but otherwise large tensors, rendering the computations intractable unless the sparsity is exploited.

4 Node

A Node represents a tensor that is computed as part of a Graph. As mentioned earlier, nodes can be of two types: Variables and Operations.

Variables, do not have any input nodes, and therefore, for the purposes of a single computation traversal, are contant tensors. However the graph allows for changing the values of these Variables across multiple

computation traversals. Thus, effectively, if the graph can be considered a single function that outputs a tensor, its Variable nodes represent its inputs.

Operations represent functions that go from multiple input tensors to a single output tensor. The backpropagation algorithm requires each operation to have a well defined first derivative with respect to its inputs.

Paragraph represents a node as a thin `struct` with an `enum` representing the node type, and an `int` index uniquely identifying the node in a graph.

Furthermore, the indices of the nodes happen to be in a topologically sorted order of the graph, as the Graph Builder assigns indices to the outputs of a node only after the node itself has been assigned. This fact is useful in determining the order in which nodes must be traversed in order to complete a traversal in the graph. However, in general, a graph may choose to ignore this ordering during computation, for example to reduce peak memory usage. The Graph does, however, use the node indices for storing and retrieving properties of each node, such as a node's name, its inputs and outputs, and for operation nodes, the tensor functor.

5 Graph

5.1 Forward propagation

Given the input values of Variable nodes, the Graph can compute the values for a set of output nodes. The input values for Variables which are not dependencies of the target output nodes need not be provided.

5.2 Backward propagation

The Graph also allows for the computation of the gradients of a given set of output nodes with respect to another given set of variable nodes. This is done in two traversals.

First, a forward propagation step computes and stores the values of all intermediate nodes.

Then, using a generalized chain rule for multi-argument tensor functions, a backward propagation step computes the derivative of the outputs with respect to each node.

5.3 Order of computation

It is somewhat easy to ensure that only nodes which are necessary are traversed, and are traversed only once (per propagation). This ensures that the computation time is minimized.

Furthermore, it is also easy to ensure that the computed values of each node is dropped out of storage as soon as they become unnecessary. This allows us to reduce the peak memory consumption, which is key for making large graphs and/or large inputs tractable. However, the *order* of computation of nodes, which may have further impact on key memory utilization, is currently not minimized. This requires future work.

6 Graph Builder

While on one hand, it quite useful to have immutable graphs in a distributed processing environment, immutable graphs are quite cumbersome to set up. Thus, it is useful to have a separate construct, a Graph Builder, which is mutable and therefore convenient for building large graphs, and which can generate an immutable Graph on demand.

The Graph Builder object allows the user to add nodes to the graph, along with their names, types, and in the case of operation nodes, their inputs and corresponding tensor functions. It automatically assigns indices to the nodes.