Automatic differentiation

Bruno Gavranović
Faculty of Electrical Engineering and Computing
University of Zagreb, Croatia

December 12, 2017

Abstract

Reverse-mode automatic differentiation - or - backpropagation - is the backbone of any deep learning library today. Most introductory backpropagation materials focus on two things: (1) dense calculations of gradients of *specific* computational graphs and (2) application of *standard* linear algebra operations on tensors of various shapes.

We claim that such introduction presents an obstacle to understanding of the essence of reverse-mode automatic differentiation: backpropagation is a *higher-order* function which transforms a computational graph of a function f into the computational graph of its derivative. Viewed through the lens of functional programming, many of the known properties can be deduced without ever lowering the level of abstraction. Conincidentaly, functional approach it is exactly the approach one needs to take to implement a general-purpose automatic differentiation framework, such as TensorFlow or PyTorch. Backpropagation can be demystified further by expanding the matrix multiplication operator as the `einsum` operator: a generization of many tensor contraction operations. This replacement results in less unnecessary notation, less error-prone calculation and better generalization capabilities to higher-rank tensor manipulation.

1 Introduction

Neural networks have shown remarkable performance in information processing on a high level of abstraction. In this paper, we will not analyze their structure on the level of abstraction of layers, activation functions and optimizers. We'll raise the abstraction level and view neural networks through the lens of functional programming. Neural networks are computational graphs, side-effect free, glued together in a programming language of our choice. We'll talk about how those computational graphs are defined, modified and executed. It is our hope that some high level patterns will emerge which will shed some light on core principles of learning mechanisms. It will turn out that many seemingly complex manipulations of such a graph are invariant to the type of nodes it is contained of.

2 Computational graphs

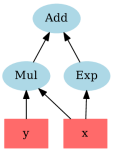
Every neural network can be represented by a computational graph G composed of nodes N and edges E. Every node $n \in N$ represents an operation, a function $f : A_0, ..., A_n \to B$, where A_i represents the input to the function and B represents its output. A_i can be input of an arbitrary type: scalar, vector, matrix or a tensor. B is output also of an arbitrary type, which can then represent input to another function. Types of A_i and B need not match.

An example of such a function $f : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$, $f(x, y) = xy + e^x$is given in the figure 1a. x and y are denoted in red and represent constants: our input variables. In the blue are operations `Add`, `Exp` and `Mul`. Edges of the graph only depict data flow and nothing else.

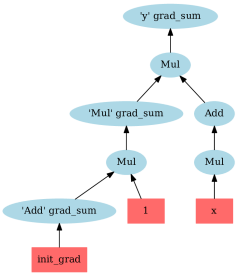
A curious thing to note is that it is also possible to represent $\frac{\partial}{\partial y} f(x, y)$ and $\frac{\partial}{\partial x} f(x, y)$ as a computational graph! In the figure 1b the computational graph of $\frac{\partial}{\partial y} f(x, y)$ is shown. In mathematical notation, it corresponds to $\frac{\partial}{\partial y} f(x, y) = \sum((\sum 1) \cdot 1) \cdot \sum \prod x$ which can be trivially simplified to $\frac{\partial}{\partial y} f(x, y) = x$.

The reason for such a form of the derivative will be clear later, but for now, let's just say that it does not present any obstacle to efficient calulation of gradients; as a matter of fact, it's exactly what enables the whole process.

Backpropagation is an automatic way to transform any computational graph $g \in G$ into another graph $h \in G$ such that the evaluation of graph h yields a partial derivative of graph g with respect to some variable.



(a) $f(x, y)$



(b) $\frac{\partial}{\partial y} f(x, y)$

Figure 1: $f(x, y) = xy + e^x$ and $\frac{\partial}{\partial y} f(x, y)$ in graph form

It's important to note that this formulation of graph transformations yields higher-order derivatives for free! As it also turns out, the result definition actually has an extremely elegant form.

2.1 Simple rules

Nodes in our computational graph are going to be represented with the class `Node`. Every function, such as $f(x) = e^x$ will need to extend the class `Node` and implement two methods: `_eval` and `_partial_derivative`. Graph is evaluted lazily; outputs of functions are not known until the method `_eval` is called.

```
class Node:
    def __init__(self, children, name="Node"):
        # wraps normal numbers into Variables
        self.children = [child if isinstance(child, Node)
                               else Variable(child)
                         for child in children]
        self.name = name
        self.shape = None

    def _eval(self):
        raise NotImplementedError()

    def _partial_derivative(self, wrt, previous_grad):
        raise NotImplementedError()

class Variable(Node):
    def __init__(self, value, name=None):
        if name is None:
            name = str(value)
        super().__init__([], name)

        self._value = value
        self.shape = self._value.shape

    @property
    def value(self):
        return self._value

    @value.setter
    def value(self, val):
        self.cached = self._value = val

    def _eval(self):
        return self._value
```

```
    def _partial_derivative(self, wrt, previous_grad):
        if self == wrt:
            return previous_grad
        return 0

class Exp(Node):
    def __init__(self, node, name="Exp"):
        super().__init__([node], name)
        self.node = self.children[0]
        self.shape = self.node.shape

    def _eval(self):
        return np.exp(self.node())

    def _partial_derivative(self, wrt, previous_grad):
        if self.node == wrt:
            return previous_grad * self
        return 0
```

As the docstring of `_partial_derivative` notes, that method if function **only of instances of Node** and **does not require evaluation of any of the Nodes**.

The listing above represents a function $f(x) = e^x$ whose manipulation in the Python REPL might look something like this.

```
>>> import autodiff as ad
>>> x = ad.Exp(3, name="x")
>>> x
<autodiff.core.ops.Exp object at 0x7f65166e6e10>
>>> x()
20.085536923187668
>>> y = x + 2
>>> y
<autodiff.core.ops.Add object at 0x7f65167bf5f8>
>>> y()
array(22.085536923187668)
>>>
```

2.2 Partial derivative

In the case that the `Primitive` represents a function $\mathbb{R}^n \mapsto \mathbb{R}$, partial derivative is a correct name for what the method `partial_derivative` is, but in the case of arbitrary functions $\mathbb{R}^n \mapsto \mathbb{R}^m$, the name might be a bit of a misnomer. In the arbitrary setting where `Primitive` represents a function $f : \mathbb{R}^n \to \mathbb{R}^m$, the `partial_derivative` returns the sum of the jacobian of $\mathbf{J}^f$ over its columns, denoted with ψ , for the lack of a better notation:

$$f : \mathbb{R}^n \to \mathbb{R}^m, \qquad \mathbf{J}^f = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix} \qquad \psi = \sum_i \mathbf{J}^f_{ij} \tag{1}$$

Note that $f : \mathbb{R}^n \to \mathbb{R} \implies \psi_y = \frac{\partial}{\partial y} f$.

Now, why would we want to return that specific sum of the jacobian matrix? The answer: to satisfy the "API" for computational graphs. Just as we usually sum all the contributing gradients of a node before calculating its partial derivative w.r.t. its input, in the same way we sum the partial deriatives of all the outputs of the jacobian *for specific element of the input vector*.

In the case of functions that transforms tensors of arbitrary rank

$$\mathbf{f} : \mathbb{R}^{n_1 \times n_2 \times \ldots n_i} \to \mathbb{R}^{m_1 \times m_2 \times \ldots m_k} \tag{2}$$

the generalization of its jacobian would be a tensor $\mathbf{J}^f \in \mathbb{R}^{n_1 \times n_2 \times \ldots n_i \ \times \ m_1 \times m_2 \times \ldots m_k}$ In such a scenario, ψ corresponds to a tensor with all the "extra" axes summed up. In other words the shape of ψ_y always corresponds to the shape of y. The same approach holds for binary functions and, generally, all functions that take more than argument. This is generally much more easier and natural to implement

using indicial notation [4]. In standard numerical libraries this is usually implemented as a function `einsum`.

2.3 Backpropagation

Backpropagation is a *higher-order* function which transforms any computational graph into the graph of its gradient with respect to some node in it. Formally, it's a function $B : G \to G$ such that $B(g) = \psi_y g, \quad \forall g \in G$, where G is the set of all computational graphs. Again, note that it does not require the evaluation of the function at any point.

Sufficient python code is given in the figure below:

```
def grad(top_node, wrt_list, previous_grad=None):
  """
  Transforms the computational graph of top_node into a list of
  computational graphs corresponoding to partial derivatives of
  top_node with respect to all variables in wrt_list.

  It delegates the actual implementation of partial derivatives
  to nodes in the computational graph and doesn't care
  how they're implemented.
  It can be elegantly implemented using foldl.
  Essentially, grad is structural transformation that is
  a function *only* of the topology of the computational graph.

  :param top_node: node in the graph whose gradient
          will be taken with respect to all variables in wrt_list
  :param wrt_list: list of objects, instances of Node,
          whose gradient we're looking for
  :param previous_grad: incoming gradient to top node,
          by default np.ones(top_node.shape)
  :return: returns a list of gradients corresponding to
  variables in wrt_list
  """
  assert isinstance(wrt_list, list) or isinstance(wrt_list, tuple)
  if previous_grad is None:
      previous_grad = Variable(np.ones(top_node.shape),
                               name=add_sum_name(top_node))

  dct = collections.defaultdict(list)
  # add the incoming gradient for the top node
  dct[top_node] += [previous_grad]

  def add_partials(dct, node):
      # sum all the incoming partial derivatives
      dct[node] = Add(*dct[node], name=add_sum_name(node))
      # calculate all partial derivs w.r.t. each child and
      # add them to child's list
      for child in set(node.children):
          pd = node.partial_derivative(wrt=child, previous_grad=dct[node])
          dct[child] += [pd]
      return dct

  # basically a foldl
  dct = functools.reduce(add_partials, reverse_topo_sort(top_node), dct)

  # if a node is not a part of the graph, return Variable(0) instead of []
  return [dct[wrt] if dct[wrt] != [] else Variable(0) for wrt in wrt_list]
```

The algorithm above starts at the top node of the graph and traverses all the nodes in it in the order of a reversed topological sort. It computes the $\psi_y f$ for every of its children and adds it to the `{node: [previous_grad]}` dictionary. Before a specific node is traversed using the reverse topo sort, all the list of `previous_grads` is added together, using the `Add` operation, which is also a node.

By defining the `add_partials` function, the whole process can be efficiently implemented using the reduce function, which corresponds to `foldl` in functional programming.

It might be clear now why figure 1b depicts such a complicated graph: all those products and sums are needed in the general case where we could've supplied the `previous_grad`, or where the input to `ad.Exp` could've been a function `x` or many other cases.

Backpropagation does not care about the specific way partial derivatives are implemented. The algorithm above incrementally creates the new graph only based on the structure of the given computational graph. There seems to be work indicating that `grad` function actually preseves the structure of the given function [1] [3].

2.4 Sufficient conditions for backpropagation

Let `partial_derivative(self, wrt, previous_grad)` be its method which returns $\psi_y f$, where y is the `wrt` argument, shortened of "with respect to". In the case of a computational graph with a valid forward pass, we claim that the following conditions are **sufficient** for the backpropagation algorithm to work.

1. Method `evaluate` returns *the value* of f
2. Method `partial_derivative` returns *the node* `n` whose `eval` corresponds to $\psi_y f$
3. `n.shape == wrt.shape`
4. `self.shape == previous_grad.shape`

The first condition signifies that computational graphs are lazily evaluated. We're abstracting the process of computation over the actual implementation. Second condition signifies that computation of graph of ψ doesn't imply its evaluation. In other words, `partial_derivative` only deals with nodes in the computational graph and *doesn't actually evaluate the gradients*. `partial_derivative` simply defines the $\psi_y f$ in terms of other nodes. The last two conditions are trivially satisfied in case of scalar functions.

2.5 Einstein summation convention

Let's compare standard matrix multiplication operations with Einstein notation, or indicial notation. Let

$$\mathbf{C} = \mathbf{AB} \tag{3}$$

where $\mathbf{A} \in \mathbb{R}^{i \times j}$ and $\mathbf{B} \in \mathbb{R}^{j \times k}$.

In indicial notation the eq. 3 has a similar form:

$$C_{ik} = A_{ij} B_{jk} \tag{4}$$

A, B and C here represent the entire matrices, not just individual elements. That form just comes from writing out the value of a specific element c_{ik}:

$$c_{ik} = \sum_j a_{ij} b_{jk} \tag{5}$$

and leaving out the summation sign, as it's implicit in Einstein notation. Notice that all the following equations represent the same matrix multiplication:

$$C_{ik} = B_{jk} A_{ij}, \tag{6}$$
$$C_{xz} = A_{xy} B_{yz} \tag{7}$$
$$\tag{8}$$

Here are some more common operations expressed in indicial notation:

$$C_{bik} = A_{bij} B_{bjk} \quad \text{Batch matrix multiply} \tag{9}$$
$$B_{ji} = A_{ij}, \quad \text{Matrix transpose} \tag{10}$$
$$B_i = A_{ii}, \quad \text{Matrix diagonal} \tag{11}$$
$$B = A_{ii}, \quad \text{Matrix trace} \tag{12}$$
$$B = A_i A_i \quad \text{Vector inner product} \tag{13}$$
$$B_{ij} = A_i A_j \quad \text{Vector outer product} \tag{14}$$
$$\ldots \tag{15}$$

The general idea behind Einstein summation is to generalize all possible ways we can compose tensors: naming each one is not possible!

2.5.1 Partial derivatives using indicial notation

Calculating partial derivatives using indicial notation reduces down to just switching the operands around, as shown in the following Python code:

```
a = np.random.rand(2, 3)
b = np.random.rand(3, 5)
c = np.einsum("ij,jk->ik", a, b) # equivalent to matmul

a_grad = np.einsum("ik,jk->ij", np.ones_like(c), b)
```

To calculate the gradient of a, we just flip the opeands and the corresponding strings. In the place of `a` we put `np.ones_like(c)` and in the place of `ij` we put `ik`. By doing this we automatically ensure that the gradient of `a` has the exact same shape as `a` and we enable the previous gradient, the gradient of `c` to be multiplied elementwise by `c` (there isn't one in this case). Notice there wasn't a need for the use of transpose operaton and notice there isn't a need to define the gradients in a different way depending with respect to which input we're differentiating.

In indicial notation, it would look something like this:

$$c = a \cdot b, \qquad C_{ik} = A_{ij} B_{jk} \tag{16}$$

$$\frac{\partial c}{\partial a} = b = 1 \cdot b, \qquad \left\{ \frac{\partial C}{\partial A} \right\}_{ij} = \mathbf{1}_{ik} B_{ij} \tag{17}$$

$$\tag{18}$$

,

where $\mathbf{1}_{ik}$ represents a tensor of ones in the shape of (i, k).

Notice the structure of the operations is the same as in the scalar case: which is one of the strenghts of indicial notation: it's a natural generalization of multiplication operation to arbirary tensors.

2.6 Conclusion

The idea that *everything is a computational graph* is a powerful one and gives us a fresh perspective on machine learning. This paper showed that efficient implementation of automatic differentiation framework can be created when looking at neural networks through the lens of functional programming. There seems to be exciting new work which takes that even further and focuses on category theory approach to neural networks [2].

Generally, ignoring the low-level abstractions and linear algebra enables us to find high-level patterns when dealing with neural networks use them to, hopefully, gain insight into first principes of learning mechanisms.

References

[1] Conal Elliott. "Beautiful differentiation". In: *International Conference on Functional Programming (ICFP)*. 2009. URL: `http://conal.net/papers/beautiful-differentiation`.

[2] B. Fong, D. I. Spivak, and R. Tuyéras. "Backprop as Functor: A compositional perspective on supervised learning". In: *ArXiv e-prints* (Nov. 2017). arXiv: `1711.10455 [math.CT]`.

[3] "http://timvieira.github.io/blog/post/2017/08/18/backprop-is-not-just-the-chain-rule/lagrange-backprop-generalization". In: ().

[4] "http://www.ita.uni-heidelberg.de/ dullemond/lectures/tensor/tensor.pdf". In: ().