Automatic Reverse-Mode Differentiation

William Cohen
Modified by xxx

Out xxx
Due xxx via Blackboard

1 Background: Automatic Differentiation

1.1 Why automatic differentiation is important

Most neural network packages (e.g., Torch or Theano) don't require a user to actually derive the gradient updates for a neural model. Instead they allow the user to define a *model* in a "little language", which supports common neural-network operations like matrix multiplication, "soft max", etc, and automatically derive the gradients. Typically, the user will define a *loss function* L in this sublanguage: the loss function takes as inputs the training data, current parameter values, and hyperparameters, and outputs a scalar value. From the definition of L, the system will compute the partial derivative of the loss function with respect to every parameter, using these gradients, it's straightforward bto implement gradient-based learning methods.

Going from the definition of L to its partial derivatives is called *automatic differentiation*. In this assignment you will start with a simple automatic differentiation system written in Python, and use it to implement a neural-network package.

1.2 Wengert lists

Automatic differentiation proceeds in two stages. First, function definitions $f(x_1, \ldots, x_k)$ in the sublanguage are converted to a format called a *Wengert list*. The second stage is to evaluate the function and its gradients using the Wengert list.

A *Wengert list* defines a function $f(x_1, \ldots, x_k)$ of some inputs $x_1, \ldots, x_k$. Syntactically, it is just a list of assignment statements, where the right-hand-side (RHS) of each statement is very simple: a call to a function g (where g is one of a set of primitive functions $G = \{g_1, \ldots, g_\ell\}$ supported by the sublanguage), where the arguments to g either inputs $x_1, \ldots, x_k$, or the left-hand-side (LHS) of a previous assignment. (The functions in G will be called *operators* here.) The output of the function is the LHS of the last item in the list. For example, a Wengert list for

$$f(x_1, x_2) \equiv (2x_1 + x_2)^2 \tag{1}$$

with the functions $G = \{\texttt{add}, \texttt{multiply}, \texttt{square}\}$ might be

$$\begin{aligned} z_1 &= \texttt{add}(x_1, x_1) \\ z_2 &= \texttt{add}(z_1, x_2) \\ f &= \texttt{square}(z_2) \end{aligned}$$

A Wengert list for

$$f(x) \equiv x^3$$

might be

$$\begin{aligned} z_1 &= \texttt{multiply}(x, x) \\ f &= \texttt{multiply}(z_1, x) \end{aligned}$$

The set of functions G defines the sublanguage. It's convenient if they have a small fixed number of arguments, and are differentiable with respect to all their arguments. But there's no reason that they have to be scalar functions! For instance, if G contains the appropriate matrix and vector operations, the loss for logistic regression (for example $\mathbf{x}$ with label y and weight matrix W) could be written as

$$f(\mathbf{x}, \mathbf{y}, W) \equiv \texttt{crossEntropy}(\texttt{softmax}(\mathbf{x} \cdot W), \mathbf{y}) + \texttt{frobeniusNorm}(W)$$

and be compiled to the Wengert list

$$\begin{aligned} \mathbf{z}_1 &= \texttt{dot}(\mathbf{x}, W) \\ \mathbf{z}_2 &= \texttt{softmax}(\mathbf{z}_1) \\ z_3 &= \texttt{crossEntropy}(\mathbf{z}_2, \mathbf{y}) \\ z_4 &= \texttt{frobeniusNorm}(W) \\ f &= \texttt{add}(z_3, z_4) \end{aligned}$$

This implements k-class logistic regression if $\mathbf{x}$ is a d-dimensional row vector, `dot` is matrix product, W is a $d \times k$ weight matrix, and

$$\begin{aligned}
\texttt{softmax}(\langle a_1, \ldots, a_d \rangle) &\equiv \langle \frac{e^{a_1}}{\sum_{i=1}^{d} e_i^a}, \ldots, \frac{e^{a_d}}{\sum_{i=1}^{d} e_i^a} \rangle \\
\texttt{crossEntropy}(\langle a_1, \ldots, a_d \rangle, \langle b_1, \ldots, b_d \rangle) &\equiv -\sum_{i=1}^{d} a_i \log b_i \\
\texttt{frobeniusNorm}(A) &\equiv \sqrt{\sum_{i=1}^{d} \sum_{j=1}^{k} a_{i,j}^2}
\end{aligned}$$

1.3 Backpropagation through Wengert list

We'll first discuss how to use a Wengert list, and then below, discuss how to construct one.

Given a Wengert list for f, it's obvious how to evaluate f: just step through the assignment statements in order, computing each value as needed. To be perfectly clear about this, the procedure is as follows. We will encode each assignment in Python as a nested tuple

$$(z, g, (y_1, \ldots, y_k))$$

where z is a string that names the LHS variable, g is a string that names the operator, and the y_i's are strings that name the arguments to g. So the list for the function of Equation 1 would be encoded in python as

```
[ ("z1", "add", ("x1","x2")),
  ("z2", "add", ("z1","x2")),
  ("f", "square", ("z2")) ]
```

We also store functions for each operator in a Python dictionary `G`:

```
G = { "add" : lambda a,b: a+b,
      "square": lambda a:a*a }
```

Before we evaluate the function, we will store the parameters in a dictionary `val`: e.g., to evalute f and $x_1 = 3$, $x_2 = 7$ we will initialize `val` to

```
val = { "x1" : 3, "x2" : 7 }
```

The pseudo-code to evaluate f is:

`def eval(`f`)`

initialize `val` to the inputs at which f should be evaluated

`for` $(z, g, (y_1, \ldots, y_k))$ in the list:

`op = G[g]`

`val[z] = op(val[`y_1`],...,val[`y_k`])`

`return` the last entry stored in `val`.

Some Python hints: (1) to convert $(y_1, \ldots, y_k)$ to `(val[`y_1`],...,val[`y_k`])` you might use Python's `map` function. (2) If `args` is a length-2 Python list and `g` is a function that takes two arguments (like `G[''add'']` above) then `g(*args)` will call `g` with the elements of that list as the two arguments to `g`.

To differentiate, we will use a generalization of backpropagation (backprop). We'll assume that `eval` has already been run and `val` has been populated, and we will compute, in *reverse* order, a value `delta`(z_i) for each variable z_i that appears in the list. We initialize `delta` by setting `delta`$(f) = 1$, (where f is the string that names the function output).

Informally you can think of `delta`(z_i) as the "sensitivity" of f to the variable z_i, , at the point we're evaluating f (i.e., the point a that corresponds to the initial dictionary entries we stored in `val`). Here z_i can be intermediate variable or an input. If it's an input x that we're treating as a parameter, `delta` is the gradient of the cost function, evaluated at a: i.e., `delta`$(x) = \frac{df}{dx}(a)$.

To compute these sensitivities we need to "backpropagate" through the list: when we encounter the assignment $(z, g, (y_1, \ldots, y_k))$ we will use `delta`(z) and the derivatives of g with respect to its inputs to compute the sensitivities of the y's. We will store derivatives for each operator in a Python dictionary `DG`.

Note that if g has k inputs, then we need k partial derivatives, one for each input, so the entries in `DG` are *lists* of functions. For the functions used in this example, we'll need these entries in DG.

```
DG = { "add" : [ (lambda a,b: 1),  (lambda a,b: 1) ],
       "square": [ lambda a:2*a ] }
```

To figure out these functions we used some high school calculus rules: $\frac{d}{dx}(x + y) = 1$, $\frac{d}{dy}(x + y) = 1$ and $\frac{d}{dx}(x^2) = 2x$.

Finally, the pseudo-code to compute the deltas is below. Note that we don't just store values in `delta`: we accumulate them additively.

```
def backprop(f,val)
    initialize delta: delta[f] = 1
    for (z,g,(y_1,...,y_k)) in the list, in reverse order:
        for i = 1,...,k:
            op_i = DG[g][i]
            if delta[y_i] is not defined set delta[y_i] = 0
            delta[y_i] = delta[y_i] + delta[z] * op_i(val[y_1],...,val[y_k])
```

1.4 Examples

Let's look at Equation 1. In freshman calculus you'd just probably just do this:

$$\begin{aligned} f(x_1, x_2) &= (2x_1 + x_2)^2 = 4x_1^2 + 4x_1x_2 + x_2^2 \\ \frac{df}{dx_1} &= 8x_1 + 4x_2 \\ \frac{df}{dx_2} &= 4x_1 + 2x_2 \end{aligned}$$

Here we'll use the Wengert list, in reverse, and the chain rule. This list is

$$\begin{aligned} z_1 &= \texttt{add}(x_1, x_1) \\ z_2 &= \texttt{add}(z_1, x_2) \\ f &= \texttt{square}(z_2) \end{aligned}$$

Table 1 contains a detailed derivation of $\frac{df}{dx_1}$, where in each step we either plug in a definition of a variable in the list, or use the derivative of one of the operators (`square` or `add`). Table 2 contains an analogous derivation of $\frac{df}{dx_1}$. Notice that these derivations are nearly identical. In fact, they are very analogous to the computations carried out by the `backprop` algorithm: can you see how?

todo: the other example

1.5 Discussion

What's going on here? Let's simplify for a minute and assume that the list is of the form

$$z_1 = f_1(z_0)$$

Derivation Step	Reason
$\frac{df}{dx_1} = \frac{dz_2^2}{dz_2} \cdot \frac{dz_2}{dx_1}$	$f = z_2^2$
$\frac{df}{dx_1} = 2z_2 \cdot \frac{dz_2}{dx_1}$	$\frac{d(a^2)}{da} = 2a$
$\frac{df}{dx_1} = 2z_2 \cdot \frac{d(z_1+x_2)}{dx_1}$	$z_2 = z_1 + x_2$
$\frac{df}{dx_1} = 2z_2 \cdot \left(1 \cdot \frac{dz_1}{dx_1} + 1 \cdot \frac{dx_2}{dx_1}\right)$	$\frac{d(a+b)}{da} = \frac{d(a+b)}{db} = 1$
$\frac{df}{dx_1} = 2z_2 \cdot \left(1 \cdot \frac{d(x_1+x_1)}{dx_1} + 1 \cdot \frac{dx_2}{dx_1}\right)$	$z_1 = x_1 + x_1$
$\frac{df}{dx_1} = 2z_2 \cdot \left(1 \cdot \left(1 \cdot \frac{dx_1}{dx_1} + 1 \cdot \frac{dx_1}{dx_1}\right) + 1 \cdot \frac{dx_2}{dx_1}\right)$	$\frac{d(a+b)}{da} = \frac{d(a+b)}{db} = 1$
$\frac{df}{dx_1} = 2z_2 \cdot (1 \cdot (1 \cdot 1 + 1 \cdot 1) + 1 \cdot 0)$	$\frac{da}{da} = 1$ and $\frac{da}{db} = 0$ for inputs a, b
$\frac{df}{dx_1} = 2z_2 \cdot 2 = 8x_1 + 4x_2$	simplify

Table 1: A detailed derivation of $\frac{df}{dx_1}$

Derivation Step	Reason
$\frac{df}{dx_2} = \frac{dz_2^2}{dz_2} \cdot \frac{dz_2}{dx_2}$	$f = z_2^2$
$\frac{df}{dx_2} = 2z_2 \cdot \frac{dz_2}{dx_2}$	$\frac{d(a^2)}{da} = 2a$
$\frac{df}{dx_2} = 2z_2 \cdot \frac{d(z_1+x_2)}{dx_2}$	$z_2 = z_1 + x_2$
$\frac{df}{dx_2} = 2z_2 \cdot \left(1 \cdot \frac{dz_1}{dx_2} + 1 \cdot \frac{dx_2}{dx_2}\right)$	$\frac{d(a+b)}{da} = \frac{d(a+b)}{db} = 1$
$\frac{df}{dx_2} = 2z_2 \cdot \left(1 \cdot \frac{d(x_1+x_1)}{dx_2} + 1 \cdot \frac{dx_2}{dx_2}\right)$	$z_1 = x_1 + x_1$
$\frac{df}{dx_2} = 2z_2 \cdot \left(1 \cdot \left(1 \cdot \frac{dx_2}{dx_2} + 1 \cdot \frac{dx_1}{dx_2}\right) + 1 \cdot \frac{dx_2}{dx_2}\right)$	$\frac{d(a+b)}{da} = \frac{d(a+b)}{db} = 1$
$\frac{df}{dx_1} = 2z_2 \cdot (1 \cdot (1 \cdot 1 + 1 \cdot 0) + 1 \cdot 1)$	$\frac{da}{da} = 1$ and $\frac{da}{db} = 0$ for inputs a, b
$\frac{df}{dx_1} = 2z_2 \cdot 2 = 4x_1 + 2x_2$	simplify

Table 2: A detailed derivation of $\frac{df}{dx_2}$

$$\begin{aligned} z_2 &= f_2(z_1) \\ &\dots \\ z_m &= f_m(z_{m-1}) \end{aligned}$$

so $f = z_m = f_m(f_{m-1}(\dots f_1(z_0)\dots))$. We'll assume we can compute the f_i functions and their derivations f_i'. We know that one way to find $\frac{dz_m}{dz_0}$ would be to repeatedly use the chain rule:

$$\begin{aligned} \frac{dz_m}{dz_0} &= \frac{dz_m}{dz_{m-1}}\frac{dz_{m-1}}{dz_0} \\ &= \frac{dz_m}{dz_{m-1}}\frac{dz_{m-1}}{dz_{m-2}}\frac{dz_{m-2}}{dz_0} \\ &\dots \\ &= \frac{dz_m}{dz_{m-1}}\frac{dz_{m-1}}{dz_{m-2}}\cdots\frac{dz_1}{dz_0} \end{aligned}$$

Let's take some time to unpack what this means. When we do derivations by hand, we are working symbolically: we are constructing a *symbolic representation* of the derivative function. This is an interesting problem—it's called *symbolic differentiation*—but it's not the same task as automatic differentiation. In automatic differentiation, we want instead an *algorithm* for *evaluating* the derivative function.

To simplify notation, let $h_{i,j}$ be the function $\frac{dz_i}{dz_j}$. (I'm doing this so that I can use $h_{i,j}(a)$ to denote the result of evaluating the function $\frac{dz_i}{dz_j}$ at a: $\frac{dz_i}{dz_j}(a)$ is hard to read.) Notice that there are m^2 of these $h_{i,j}$ functions—quite a few!—but for machine learing applications we won't care about most of them: typically we just care about the partial derivative of the cost function (the final variable in the list) with respect to the parameters, so we only need $h_{m,i}$ for certain i's.

Let's look at evaluating $h_{m,0}$ as some point $z_0 = a$ (say $z_0 = 53.4$). Again to simplify, define

$$\begin{aligned} a_1 &= f_1(a) \\ a_2 &= f_2(f_1(a)) \\ &\dots \\ a_m &= f_m(f_{m-1}(f_{m-2}(\dots f_1(a)\dots))) \end{aligned}$$

When we write

$$\frac{dz_m}{dz_0} = \frac{dz_m}{dz_{m-1}}\frac{dz_{m-1}}{dz_0}$$

We mean that: for all a

$$h_{m,0}(a) = f'_m(a_m) \cdot h_{m-1,1}(a)$$

That's a useful step because we have assumed we have available a routine to evaluate $f'_m(a_m)$: in the code this would be the function `DG[`f_m`][1]`. Continuing, when we write

$$\frac{dz_m}{dz_0} = \frac{dz_m}{dz_{m-1}}\frac{dz_{m-1}}{dz_{m-2}} \cdots \frac{dz_1}{dz_0}$$

it means that

$$h_{m,0}(a) = f'_m(a_m) \cdot f'_{m-1}(a_{m-1}) \cdots f'_2(a_1) \cdot f'_1(a)$$

When we execute the `backprop` code above, this is what we do: in particular we group the operations as

$$h_{m,0}(a) = \Big(\Big(\Big((f'_m(a_m) \cdot f'_{m-1}(a_{m-1})\Big) \cdot f'_{m-2}(a_{m-2})\Big) \cdots f'_2(a_1))\Big) \cdot f'_1(a)$$

and the `delta`'s are partial products: specifically

$$\texttt{delta}[z_i] = f'_m(a_m) \cdots f'_i(a_i)$$

todo: discuss accumulation and distribution

1.6 Constructing Wengert lists

Wengert lists are useful but tedious to program in. Usually they are constructed using some sort programming language extension. You will be provided a package, `xman.py`, to construct Wengert lists from Python expressions: `xman` is short for "expression manager". Here is an example of using `xman`:

```
from xman import *
...
class f(XManFunctions):
    @staticmethod
    def half(a):
        ...
class Triangle(XMan):
```

```
    h = f.input()
    w = f.input()
    area = f.half(h*w)
...
xm = Triangle().setup()
print xm.operationSequence(xm.area)
```

In the definition of `Triangle`, the variables `h`, `w`, and `area` are called *registers*. Note that after creating an instance of a subclass of `xman.XMan`, you need to call `setup()`, which returns the newly-created instance. After the `setup` you can call the `operationSequence` method to construct a Wengert list, which will be encoded in Python as

```
[('z1', 'mul', ['h', 'w']),
 ('area', 'half', ['z1'])]
```

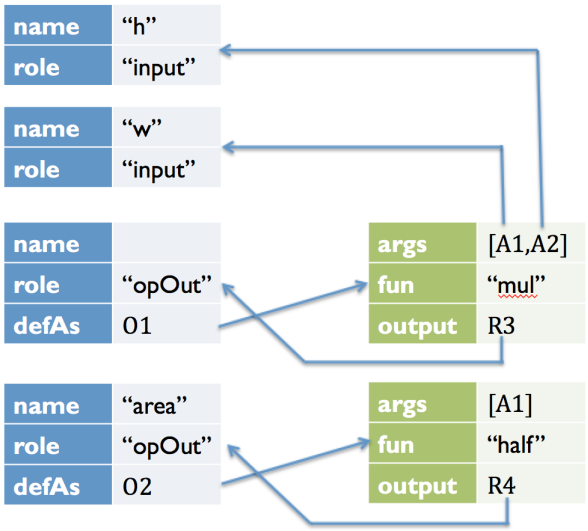
Internally this works as follows. There are two types of Python objects, called *Registers* and `Operations`. A `Register` corresponds to a variable, and an `Operation` corresponds to a function call.

The base `XManFunctions` class defines a method `input()` which creates a register object that is marked as an "input", meaning that it has no definition. (A similar method `param()` creates a register object that is marked as a "parameter", which like an input has no definition.) It also defines a few functions that correspond to operators, like `mul` and `add`. These return a register object that is cross-linked to an `Operation` object, as illustrated in Figure 1.6. (The `Register` class also uses Python's operator overloading to so that syntax like `h*w` is expanded to `XManFunctions.mul(h,w)`.)

To produce the Wengert list, the `setup()` command uses Python introspection methods to add names to each register, based on the Python variable that points to it, and generates new variable names for any reachable registers that cannot be named with Python variables. Finally, the `operationSequence` does a pre-order traversal of the data structure to create a Wengert list.

2 Assignment

*todo: more hints: return delta[z]*d, to you can do the matrix-vector thing; sometimes output is used in derivatives.*




```
class XManFunctions(object):
    @staticmethod
    def input(default=None):
        return Register(role='input',default=default)
    ...
    @staticmethod
    def mul(a,b):
        return XManFunctions.registerDefinedByOperator('mul',a,b)
    ...
    @staticmethod
    def registerDefinedByOperator(fun,*args):
        reg = Register(role='operationOutput')
        op = Operation(fun,*args)
        reg.definedAs = op
        op.outputReg = reg
        return reg
```

Figure 1: The Python data structures created by the `Triangle` class. Blue objects are `Registers`, and green ones are `Operations`. Arrows are pointers.

3 Data

4 Deliverables

5 Marking breakdown