Automatic Differentiation for R

Radford M. Neal, University of Toronto

Dept. of Statistical Sciences and Dept. of Computer Science

Vector Institute Affiliate

`http://www.cs.utoronto.ca/~radford`

`http://radfordneal.wordpress.com`

`http://pqR-project.org`

Work in Progress

Vector Research Symposium, 22 February 2019

Aims of this project

Given R code written only to evaluate a function (e.g., a log likelihood), allow the derivatives of this function with respect to any real-valued inputs (e.g., model parameters) to be computed as well.

Do this:

- Universally — for R functions written using most common language features, in any reasonable style, without thought of gradients.
- Conveniently — with only the essential additions to the program, e.g., specifying w.r.t. what variables gradients should be taken.
- Efficiently — speed of resulting value + gradient evaluation should be comparable to what would be obtained with manually-written code.

I expect to mostly achieve these aims, though with a few compromises (e.g., good performance may sometimes require tweaking code a bit).

The pqR implementation of R

This project is part of my pqR implementation of R (`pqR-project.org`). This is a fork of the R Core implementation with many improvements. Some changes relevant to automatic differentiation are:

- A new parser makes introducing new language constructs easier.
- Interpretive speed has been greatly improved, eliminating any need to use the "byte-code compiler". Implementation by direct interpretation makes extending the language (e.g., for AD) easier.
- Multiple processor cores can be automatically used to parallelize numerical computations. This may be especially useful when gradients can be computed in parallel with the function value.
- An internal mechanism allows expressions to be evaluated asking for a "variant" result — crucial for how AD is implemented in this project.

Summary of the approach

- New language constructs are introduced to allow convenient use of automatic differentiation.
- The tracking of gradients associated with values is invisible at the R level, ensuring that functions behave the same when gradients are requested as when not.
- Forward-mode differentiation is used, but with two modifications:
  - Jacobian matrices (giving derivatives of some set of values with respect to some set of variables) may be stored in a compact form, when the Jacobian is block diagonal and/or has repetitive portions.
  - The programmer can manually specify points where backpropagation (reverse-mode differentiation) should be done.

  The strategy used to integrate AD in the interpreter could support pure reverse-mode differentiation if this turns out to be desirable.

The `with gradient` construct

A new `with gradient` language construct delivers both the value of an expression, and its gradient with respect to a specified set of variables:

```
> with gradient (a=1.2) sin(3*a) # gradient is 3*cos(3*1.2)
[1] -0.4425204
attr(,"gradient")
[1] -2.690275
```

The gradient is attached as a `"gradient"` attribute – an existing convention used by the standard `nlm` (non-linear minimization) function.

Gradients can be with respect to several variables, giving a list:

```
> r <- with gradient (a=3, b=8) a*b + a^2
> attr(r,"gradient") $ a
[1] 14
> attr(r,"gradient") $ b
[1] 3
```

An example function

```
# Find the distance travelled by a projectile launched on level
# ground with initial velocity (vx,vy), with no air resistance.

distance_travelled <- function (vx, vy, dt=0.0001, g=9.8) {
    x <- y <- 0
    repeat {
        last_x <- x
        last_y <- y
        x <- x + vx*dt
        y <- y + vy*dt
        if (y < 0)  # return impact x location, interpolating
            return ((x*last_y - last_x*y) / (last_y-y))
        vy <- vy - g*dt
    }
}
```

Using automatic differentiation with this example

Let's use this function to find the angle of launch maximizing distance (for fixed initial speed), using `nlm`, with derivatives found by `with gradient`.

```
> nlm (function (a)
+       with gradient (a) -distance_travelled (cos(a), sin(a)),
+     0) $ estimate * 180/pi
[1] 44.99701
```

Note that derivatives are automatically tracked through the function call, assignments, loop, and if statement.

The `distance_travelled` function might have been written with no thought of differentiation — though it is essential to this example that it interpolates the impact position; not doing so makes it piecewise constant in `vy`, with a zero derivative w.r.t. `vy` for any value of `dt`.

When is this useful?

For this example, `nlm` actually works fine when the function does not attach a gradient — `nlm` then uses numerical differentiation, which is not bad when differentiating w.r.t. only one variable.

But in general...

Numerical differentiation: For an n-argument function, obtaining a numerical gradient requires $n + 1$ function evaluations.

Forward-mode automatic differentiation: Potentially takes $n + 1$ times as long to do each operation, computing both the value and n derivatives. But in practice, not all operations need to track derivatives, and many that do involve only a subset of derivatives.

So forward-mode AD may provide a large speedup over numerical differentiation. But when the computation involves many intermediate values, reverse-mode AD may do better.

An example of forward-mode inefficiency

```
> V <- as.list(seq(0,1,length=3))
> with gradient (V) {
+     p <- 0
+     for (i in 1:length(V))
+         p <- p + i*V[[i]]
+     p^2 + p^3 + p^4 + p^5
+ }
[1] 1360
attr(,"gradient")
attr(,"gradient")[[1]]
[1] 1592
attr(,"gradient")[[2]]
[1] 3184
attr(,"gradient")[[3]]
[1] 4776
```

Note: Since `V` is a list, the gradient is also a list, whose elements are derivatives w.r.t. elements of `V`.

The `back gradient` construct

When computing `p^2 + p^3 + p^4 + p^5` in this example, each operation tracks derivatives w.r.t. all elements of `V` — with increasing inefficiency as the length of `V` increases.

Re-writing the code using the `back gradient` construct avoids this:

```
with gradient (V) {
    p <- 0
    for (i in 1:length(V))
        p <- p + i*V[[i]]
    back gradient (p)
        p^2 + p^3 + p^4 + p^5
}
```

Within the `back gradient (p)` construct, expressions involving `p` have their derivatives tracked with respect to `p` only. When the `back gradient` construct finishes, these derivatives w.r.t. `p` are converted to derivatives w.r.t. the elements of `V`, using the chain rule.

The `compute gradient` construct

When desired, the method for computing the gradient of an expression can be specified explicitly. For example, rather than

```
sigmoid <- function (x) 1 / (1+exp(-x))
```

for which gradients will be computed automatically, one could instead write

```
sigmoid <- function (x)
    compute gradient (x) { v <- 1 / (1+exp(-x)); v }
    as (v * (1-v))
```

Using the already-computed function value, `v`, may be more efficient.

The `compute gradient` construct could be also useful in cases where the interpreter does not know how to compute the gradient — for example, when the function is computed by an external C or Fortran routine.

Both `compute gradient` and `back gradient` bypass gradient computations, with minimal overhead, when evaluated in a context where gradients are not needed.

Form of the gradient for structured values

The gradient attached to the result of

```
with gradient (V = list (x=5, y=7))
    list (a=V$x^2, b=V$x*V$y, c=V$y^2)
```

is a list with elements `x` and `y`, each of which is a list with elements `a`, `b`, and `c`, giving the derivative of `a`, `b`, or `c` with respect to `x` or `y`.

The gradient attached to the result of

```
with gradient (V = c(5,7))
    c (V[1]^2, V[1]*V[2], V[2]^2)
```

is a 3×2 Jacobian matrix, with element (i, j) giving the derivative of the i'th element of the result with respect to the j'th element of `V`.

For lists of vectors, as values or variables differentiated w.r.t., the gradient has the list structure above, with elements that are Jacobian matrices.

For lists of lists of matrices, etc. — well, things get more complicated...

Compact representations of Jacobian matrices

Consider a neural net with 1000 inputs, one hidden layer of 100 tanh units, and one output. Its output and gradient w.r.t. network parameters could be computed as follows, where `x` is the input vector:

```
o <- with gradient (W, b, v, a) a + v %*% tanh (b + x %*% W)
```

Implementing this naively with forward differentiation would be very inefficient. Even before starting the computation, the 1000×100 matrix `W` would be associated with a Jacobian matrix with 10^{10} elements.

But this Jacobian for `W` is an identity matrix, so can be represented quite compactly. The Jacobian for `W` associated with vector `x %*% W`, of length 100, will have 10^7 elements, but will be block-diagonal, all blocks equal to `x`, so can also be very compactly represented. A block-diagonal representation of the Jacobian for `W` of `tanh (b + x %*% W)` has 10^5 elements (same size as `W`).

So, using compact representations (internally), this one-hidden-layer net can be handled efficiently by forward differentiation. But `back gradient` is necessary for efficient handling of more hidden layers.

Some application areas

- Estimation of parameters of statistical models (e.g., by maximum likelihood), using gradient-based optimization. R is widely-used and convenient for specifying such models.

  Many such models have small to moderate (≈ 100) numbers of parameters. The AD methods described here should work well in this application.

- Training of neural networks, for supervised, unsupervised, and reinforcement learning.

  These networks often have quite large numbers of parameters, and quite large numbers of intermediate values (hidden units). This project aims to give acceptable performance in this application, though perhaps not as good as some other well-optimized frameworks.

- Markov chain Monte Carlo methods based on gradients, such as Hamiltonian Monte Carlo.

Implementation status

A test version currently available at `pqR-project.org` implements

- The language constructs for gradient computation
- The apparatus for recording and propagating gradients during program interpretation
- Gradients of and with respect to scalar reals, lists of scalar reals, lists of lists, etc., with access / replacement of elements using `$` and `[[.]]`.
- Evaluation of derivatives for nearly all relevant base and stats functions (e.g., `sin`, `dnorm`, `rexp`, ...)

It does not yet implement (but probably will in a month or two)

- Gradients of and with respect to vectors, matrices, and arrays of reals, and access / replacement of subvectors with `[.]`.
- Evaluation of derivatives for functions involving vectors and matrices, such as `sum`, `rep`, `colSums`, and `%*%` (matrix multiply).

Possible further extensions

- Allow the expression for a gradient in a `compute gradient` construct to be a function that computes a Jacobian-vector product. Useful when `compute gradient` is used with large objects, such as matrices.
- Allow computation of higher derivatives through use of nested gradient constructs. (Not currently possible, since currently gradients are not themselves associated with gradient information.)
- Allow gradients to be used more easily, by allowing arithmetic on lists of parallel structure. This would, for example, allow gradient descent to be done for parameters structured as lists without having to use R's unlist/relist functions.
- Could an implementation strategy somewhat analogous to that used for AD support incremental re-computation of a function after only a small part of its argument has changed? Would be very useful for MCMC.