Note that this version very likely has various bugs — mostly showing up only if you use automatic differentiation, I hope.

You can get derivatives (gradients) using the new with gradient construct. Here’s a simple example:

> with gradient (abc=7) abc^2+5

[1] 54

attr(,"gradient")

[1] 14

The with gradient construct creates a new local environment with a variable abc, which is initialized to 7. The expression abc^2+5 is evaluated in this enviroment, and the result is returned along with its derivative with respect to abc, attached as a gradient attribute.

The expression whose gradient is found needn’t be so simple. Here’s another example:

> f <- function (x) {

+ s <- 0

+ for (i in 1:length(x)) s <- s + i\*x[i]

+ s

+ }

> vec <- c(3,1,9)

> with gradient (vec) f(vec)

[1] 32

attr(,"gradient")

[,1] [,2] [,3]

[1,] 1 2 3

> with gradient (vec) f(vec^2)

[1] 254

attr(,"gradient")

[,1] [,2] [,3]

[1,] 6 4 54

Here, with gradient (vec) initializes the vec variable to the value of the existing variable of that name (same as if (vec=vec) had been written).

The output of with gradient matches what the existing numericDeriv function gives:

> numericDeriv (quote (f(vec^2)), "vec")

[1] 254

attr(,"gradient")

[,1] [,2] [,3]

[1,] 6 4 54

However, with gradient computes the derivatives exactly (apart from roundoff error), whereas numericDeriv computes an approximation using finite differences.

The output of both with gradient and numericDeriv in this example is a “Jacobian” matrix, with one row in this case, since the value of f(vec^2) is a scalar, and three columns, giving the derivatives with respect to the three elements of vec.

When finding the gradient with respect to many variables (or vector elements), numericDeriv may be very slow, since getting numerical derivatives with respect to N variables requires at least N+1 evaluations of the expression. In contrast, with gradient evaluates the expression once, either computing gradients as it goes (called “forward mode” differentiation) or recording how the value was computed and later finding the gradient by “backpropagation” (also called “reverse mode” diffferentiation). The pqR implementation tries to choose which of these modes is best, though at present forward mode is always used for some operations for which reverse mode hasn’t been implemented yet.

Neural network training with multiple hidden layers is one application where reverse mode differentiation is crucial. Here’s a function to compute the output of a neural network with two hidden layers, given an input vector x and network parameters in L:

nnet <- function (x, L)

{

h1 <- tanh (L$b1 + x %\*% L$W1)

h2 <- tanh (L$b2 + h1 %\*% L$W2)

as.vector (L$b3 + h2 %\*% L$W3)

}

If the two hidden layers both have 10 units, so there are 100 weights in L$W2, naive forward mode differentiation would record a 10-by-100 element Jacobian matrix associated with h2. But reverse mode differentiation starting from the final scalar output value only needs to compute Jacobian matrices with a single row (and up to 100 columns), as it works backwards.

The current automatic differentiation implementation in pqR manages to achieve this automatically (though it doesn’t yet when some other operations are done).

Here’s how this network function could be used for gradient descent training to minimize squared error when predicting responses in a training set, using a network with n1 and n2 units in the hidden layers:

train <- function (X, y, n1, n2, iters, step, sd=0.1)

{

# Initialize parameters randomly.

L <- list()

n0 <- ncol(X)

L$b1 <- rnorm (n1,sd=sd)

L$W1 <- matrix (rnorm(n0\*n1,sd=sd), n0, n1)

L$b2 <- rnorm (n2,sd=sd)

L$W2 <- matrix (rnorm(n1\*n2,sd=sd), n1, n2)

L$b3 <- rnorm (1,sd=sd)

L$W3 <- rnorm (n2,sd=sd)

# Train for 'iters' iterations to minimize squared

# error predicting y.

for (i in 1:iters) {

# Find gradient of squared error (summed over all

# training examples) with respect to the parameters.

r <- with gradient (L) {

e <- 0

for (i in 1:nrow(X)) {

o <- nnet (X[i,], L)

e <- e + (y[i]-o)^2

}

e

}

g <- attr(r,"gradient")

if (i %% 100 == 0)

cat ("Iteration", i, ": Error", round(r,4), "\n")

# Update parameters to reduce squared error.

L$b1 <- L$b1 - step \* as.vector (g$b1)

L$W1 <- L$W1 - step \* as.vector (g$W1)

L$b2 <- L$b2 - step \* as.vector (g$b2)

L$W2 <- L$W2 - step \* as.vector (g$W2)

L$b3 <- L$b3 - step \* as.vector (g$b3)

L$W3 <- L$W3 - step \* as.vector (g$W3)

}

L

}

Note that when the gradient is with respect to a list (L here), the gradient is a list of the corresponding Jacobian matrices (which here have just one row).

Here’s code to use this train function to learn an example function of two inputs, based on 100 slightly noisy examples:

set.seed(1)

truef <- function (X) cos (2\*sqrt(X[,1]\*X[,2])) - 2\*(0.4-X[,1])^2

N <- 100

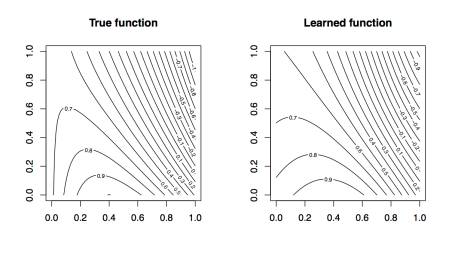
X <- cbind (runif(N), runif(N))

y <- truef (X) + rnorm(N,sd=0.01)

print (system.time (L <- train (X, y, 10, 10, 30000, 0.001)))

The 30000 training iterations (each looking at all 100 training cases) take 30 seconds on my computer.

The result is pretty good, as seen from the contour plots below:



Relevant R Code of pqR Package

|  |
| --- |
| # Compute the output of a network with two hidden layers |
|  | # (with tanh activation function), when given 'x' as input. |
|  | # Parameters are in the list 'L'. |
|  |  |
|  | nnet <- function (x, L) |
|  | { |
|  | h1 <- tanh (L$b1 + x %\*% L$W1) |
|  | h2 <- tanh (L$b2 + h1 %\*% L$W2) |
|  | as.vector (L$b3 + h2 %\*% L$W3) |
|  | } |
|  |  |
|  |  |
|  | # Train network with hidden layers of n1 and n2 units on |
|  | # data in 'X' and 'y'. Initialize with std. dev. 'sd'. Do |
|  | # 'iters' gradient descent iterations with stepsize 'step'. |
|  |  |
|  | train <- function (X, y, n1, n2, iters, step, sd=0.1) |
|  | { |
|  | # Initialize parameters randomly. |
|  |  |
|  | L <- list() |
|  | n0 <- ncol(X) |
|  | L$b1 <- rnorm (n1,sd=sd) |
|  | L$W1 <- matrix (rnorm(n0\*n1,sd=sd), n0, n1) |
|  | L$b2 <- rnorm (n2,sd=sd) |
|  | L$W2 <- matrix (rnorm(n1\*n2,sd=sd), n1, n2) |
|  | L$b3 <- rnorm (1,sd=sd) |
|  | L$W3 <- rnorm (n2,sd=sd) |
|  |  |
|  | # Train for 'iters' iterations to minimize squared |
|  | # error predicting 'y'. |
|  |  |
|  | for (i in 1:iters) { |
|  |  |
|  | # Find gradient of squared error (summed over all |
|  | # training examples) with respect to the parameters. |
|  |  |
|  | r <- with gradient (L) { |
|  | e <- 0 |
|  | for (i in 1:nrow(X)) { |
|  | o <- nnet (X[i,], L) |
|  | e <- e + (y[i]-o)^2 |
|  | } |
|  | e |
|  | } |
|  |  |
|  | g <- attr(r,"gradient") |
|  |  |
|  | if (i %% 100 == 0) |
|  | cat ("Iteration", i, ": Error", round(r,4), "\n") |
|  |  |
|  | # Update parameters to reduce squared error. |
|  |  |
|  | L$b1 <- L$b1 - step \* as.vector (g$b1) |
|  | L$W1 <- L$W1 - step \* as.vector (g$W1) |
|  | L$b2 <- L$b2 - step \* as.vector (g$b2) |
|  | L$W2 <- L$W2 - step \* as.vector (g$W2) |
|  | L$b3 <- L$b3 - step \* as.vector (g$b3) |
|  | L$W3 <- L$W3 - step \* as.vector (g$W3) |
|  | } |
|  |  |
|  | L |
|  | } |
|  |  |
|  |  |
|  | # Example of learning a 2D function. |
|  |  |
|  | set.seed(1) |
|  |  |
|  | pdf("smnn.pdf",width=8,height=4.6) |
|  | par(mfrow=c(1,2)) |
|  |  |
|  | truef <- function (X) cos (2\*sqrt(X[,1]\*X[,2])) - 2\*(0.4-X[,1])^2 |
|  |  |
|  | grid <- seq(0,1,length=101) |
|  | Xgrid <- cbind (rep(grid,times=101), rep(grid,each=101)) |
|  |  |
|  | contour (matrix(truef(Xgrid),101,101), levels=seq(-1,1,by=0.1)) |
|  | title("True function") |
|  |  |
|  | N <- 100 |
|  | X <- cbind (runif(N), runif(N)) |
|  | y <- truef (X) + rnorm(N,sd=0.01) |
|  |  |
|  | print (system.time (L <- train (X, y, 10, 10, 30000, 0.001))) |
|  |  |
|  | print(L) |
|  |  |
|  | contour ( |
|  | matrix (apply (Xgrid, 1, function (x) nnet (x, L)), 101, 101), |
|  | levels=seq(-1,1,by=0.1)) |
|  | title("Learned function") |
|  |  |
|  |  |
|  | # Confirm that there's no allocation of large (eg, 10 x 100) |
|  | # intermediate Jacobian matrices when computing gradients of |
|  | # network parameters. |
|  |  |
|  | Rprofmemt(nelem=8) |
|  | with gradient (L) nnet (c(0.3,0.6), L) |

|  |
| --- |
| # Compute the output of a network with two hidden layers |
|  | # (with tanh activation function), when given 'x' as input. |
|  | # Parameters are in the list 'L'. |
|  |  |
|  | nnet <- function (x, L) |
|  | { |
|  | h1 <- tanh (L$b1 + x %\*% L$W1) |
|  | h2 <- tanh (L$b2 + h1 %\*% L$W2) |
|  | as.vector (L$b3 + h2 %\*% L$W3) |
|  | } |
|  |  |
|  |  |
|  | # Train network with hidden layers of n1 and n2 units on |
|  | # data in 'X' and 'y'. Initialize with std. dev. 'sd'. Do |
|  | # 'iters' gradient descent iterations with stepsize 'step'. |
|  |  |
|  | train <- function (X, y, n1, n2, iters, step, sd=0.1) |
|  | { |
|  | # Initialize parameters randomly. |
|  |  |
|  | L <- list() |
|  | n0 <- ncol(X) |
|  | L$b1 <- rnorm (n1,sd=sd) |
|  | L$W1 <- matrix (rnorm(n0\*n1,sd=sd), n0, n1) |
|  | L$b2 <- rnorm (n2,sd=sd) |
|  | L$W2 <- matrix (rnorm(n1\*n2,sd=sd), n1, n2) |
|  | L$b3 <- rnorm (1,sd=sd) |
|  | L$W3 <- rnorm (n2,sd=sd) |
|  |  |
|  | # Train for 'iters' iterations to minimize squared |
|  | # error predicting 'y'. |
|  |  |
|  | for (i in 1:iters) { |
|  |  |
|  | # Find gradient of squared error (summed over all |
|  | # training examples) with respect to the parameters. |
|  |  |
|  | r <- with gradient (L) { |
|  | e <- 0 |
|  | for (i in 1:nrow(X)) { |
|  | o <- nnet (X[i,], L) |
|  | e <- e + (y[i]-o)^2 |
|  | } |
|  | e |
|  | } |
|  |  |
|  | g <- attr(r,"gradient") |
|  |  |
|  | if (i %% 100 == 0) |
|  | cat ("Iteration", i, ": Error", round(r,4), "\n") |
|  |  |
|  | # Update parameters to reduce squared error. |
|  |  |
|  | L$b1 <- L$b1 - step \* as.vector (g$b1) |
|  | L$W1 <- L$W1 - step \* as.vector (g$W1) |
|  | L$b2 <- L$b2 - step \* as.vector (g$b2) |
|  | L$W2 <- L$W2 - step \* as.vector (g$W2) |
|  | L$b3 <- L$b3 - step \* as.vector (g$b3) |
|  | L$W3 <- L$W3 - step \* as.vector (g$W3) |
|  | } |
|  |  |
|  | L |
|  | } |
|  |  |
|  |  |
|  | # Example of learning a 2D function. |
|  |  |
|  | set.seed(1) |
|  |  |
|  | pdf("smnn.pdf",width=8,height=4.6) |
|  | par(mfrow=c(1,2)) |
|  |  |
|  | truef <- function (X) cos (2\*sqrt(X[,1]\*X[,2])) - 2\*(0.4-X[,1])^2 |
|  |  |
|  | grid <- seq(0,1,length=101) |
|  | Xgrid <- cbind (rep(grid,times=101), rep(grid,each=101)) |
|  |  |
|  | contour (matrix(truef(Xgrid),101,101), levels=seq(-1,1,by=0.1)) |
|  | title("True function") |
|  |  |
|  | N <- 100 |
|  | X <- cbind (runif(N), runif(N)) |
|  | y <- truef (X) + rnorm(N,sd=0.01) |
|  |  |
|  | print (system.time (L <- train (X, y, 10, 10, 30000, 0.001))) |
|  |  |
|  | print(L) |
|  |  |
|  | contour ( |
|  | matrix (apply (Xgrid, 1, function (x) nnet (x, L)), 101, 101), |
|  | levels=seq(-1,1,by=0.1)) |
|  | title("Learned function") |
|  |  |
|  |  |
|  | # Confirm that there's no allocation of large (eg, 10 x 100) |
|  | # intermediate Jacobian matrices when computing gradients of |
|  | # network parameters. |
|  |  |
|  | Rprofmemt(nelem=8) |
|  | with gradient (L) nnet (c(0.3,0.6), L) |

|  |
| --- |
| # 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 |
|  | } |
|  | } |
|  |  |
|  |  |
|  | # See the derivative of distance with respect to angle of launch for |
|  | # angles of 40 and 60 degrees. |
|  |  |
|  | with gradient (a = 40 \* pi/180) distance\_travelled (cos(a), sin(a)) |
|  | with gradient (a = 60 \* pi/180) distance\_travelled (cos(a), sin(a)) |
|  |  |
|  |  |
|  | # Find angle of launch maximizing distance, for fixed initial speed. |
|  | # Note that nlm minimizes, so we negate the distance. |
|  |  |
|  | system.time(print( |
|  | nlm (function (a) |
|  | with gradient (a) -distance\_travelled (cos(a), sin(a)), |
|  | 0) $ estimate \* 180/pi |
|  | )) |
|  |  |
|  |  |
|  | # For this simple example with only one variable, optimizing without |
|  | # the gradient information is about as fast... |
|  |  |
|  | system.time(print( |
|  | nlm (function (a) |
|  | -distance\_travelled (cos(a), sin(a)), |
|  | 0) $ estimate \* 180/pi |
|  | )) |