Function minimization using conjugate gradients: `conj`

Carl Edward Rasmussen

May 15 1996

This document describes the utility function `conj` which optimizes network weights using a conjugent gradient method. The function has a private function `lns` for doing line-searches.

Optimization Procedure

The conjugent gradient method for minimizing a function of many variables works by iteratively computing search directions, along which a line search procedure minimizes the function, producing a new approximation to the (local) minimum of the objective function.

An *iteration* is defined as a computation of a search direction and the following line search. An *epoch* is defined as the computation of the function value and gradient, $f(\mathbf{x})$ and $\mathbf{g}(\mathbf{x}) = \nabla f(\mathbf{x})$. The function and gradients are always computed as a pair. An iteration might involve many epochs (because of the line search). The number of epochs is thus indicative of the computational effort.

Conjugate Gradients

At iteration number i the position in weight space is $\mathbf{x}_i$, the value of the objective function is $f(\mathbf{x}_i)$ and the gradient (vector of partial derivatives) is $\mathbf{g}_i = \nabla f(\mathbf{x}_i)$. I use what is known as the Polack-Ribiere version of conjugent gradients (Fletcher, 1987), which computes the new search direction $\mathbf{s}_i$ from the old direction $\mathbf{s}_{i-1}$ and the current and old gradients $\mathbf{g}_i$ and $\mathbf{g}_{i-1}$:

$$\mathbf{s}_i = \mathbf{g}_i + \frac{(\mathbf{g}_i - \mathbf{g}_{i-1})^{\mathrm{T}}\mathbf{g}_i}{(\mathbf{g}_{i-1})^{\mathrm{T}}\mathbf{g}_{i-1}}\mathbf{s}_{i-1}, \qquad i = 1, 2, 3, \ldots \quad \text{with} \quad \mathbf{s}_0 = \mathbf{g}_0 = \mathbf{0}. \tag{1}$$

The slope of $f(\mathbf{x})$ in the search direction is denoted by $f'(x)$. It may happen that the function has a positive slope (i.e. there is no guarantee that the slope is negative), in which case I use the direction of steepest descent (i.e. $\mathbf{s}_i = -\mathbf{g}_i$) for that single iteration. The gradient of the cost function is computed using the backpropagation rule.

The conjugent gradient method can be shown to have quadratic convergence properties, which is much better than the originally proposed method of steepest descent. Also, a quadratic optimisation problem can be solved in n (the dimensionality of the problem) iterations if the line searches are exact. However, the cost function of a neural network is typically not quadratic in the weights, so the procedure will not converge in n steps. Rather, it will be used in an iterative manner. Sometimes, the algorithm is used with *restarts* (i.e. starting again with steepest descent) every n iterations, however this is not done in the present implementation, since typically n is very large and the Polack-Ribiere version which is used here, has a natural tendency to automatically restart whenever progress is slow, since in that case $\mathbf{g}_{i-1} \approx \mathbf{g}_i$ and consequently $\mathbf{s}_i \approx \mathbf{g}_i$ in eq. (1).

Line search

The object of the line search is to minimize the objective function along the direction $\mathbf{s}$. One dimensional minimization problems are more tractable than multidimensional ones, but it is still not in general possible to find the global minimum and many epochs may be necessary to achieve close approximations to the minimum. Since in our case the optimisation problem is usually non-quadratic, it is not sensible to expend huge computational resources in doing very accurate line searches (remember that one function evaluation requires a full sweep through the entire training set). Consequently, the goal of the line search will merely be: to get significantly closer to a local minimum.

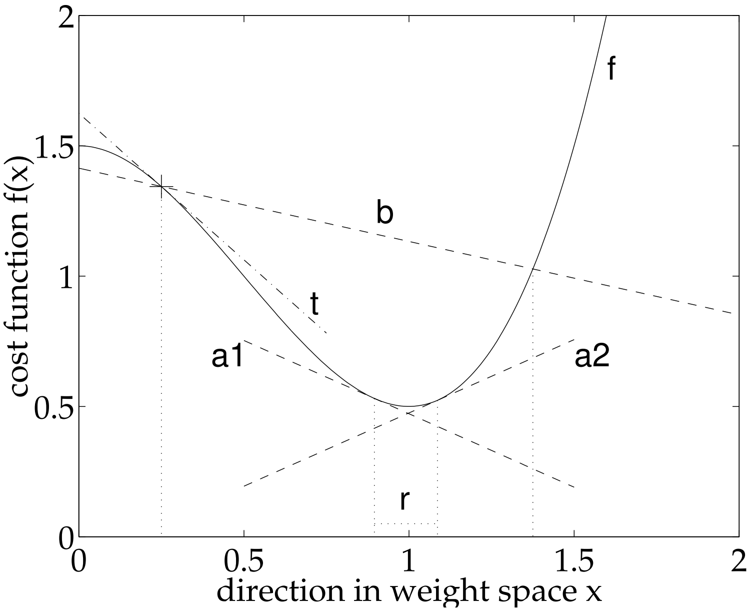

Figure 1: Illustration of the Wolfe-Powell conditions. The current point is marked by '+' at $x = 0.25$. The cost function is f and a tangent t is drawn at the current point. The intersections of the lines $a1$, $a2$ and b with the function f defines the region r of acceptable points.

I use the two sided Wolfe-Powell conditions (Fletcher, 1987) to determine whether a new point is significantly better than the current point. The Wolfe-Powell conditions consists of 2 inequalities. The first inequality requires that the absolute slope at the new point is smaller (in a absolute sense) than some fraction of the magnitude of the slope at the current point. This will in general guarantee that we have moved closer to a local extremum by a non-vanishing amount. The condition is $|f'(\mathbf{x}_i)| \leq \sigma |f'(\mathbf{x}_{i-1})|$, where I use $\sigma = 0.5$. In fig. 1 this condition implies that acceptable points must lie between the intersections of lines $a1$ and $a2$ with f. The second inequality requires that a substantial fraction of the decrease expected from the current slope is being achieved, thus guaranteeing a decrease in the function value and avoiding huge steps with negligible function decrease. The condition is $f(\mathbf{x}_i) \leq f(\mathbf{x}_{i-1}) + \rho(\mathbf{x}_i - \mathbf{x}_{i-1})f'(\mathbf{x}_{i-1})$, where I use $\rho = 0.25$. In fig. 1, only smaller values of x than where the line labeled b intersects with f are acceptable. In the figure, the final region of acceptable points is marked by r. It can be shown that acceptable points always exist for a continuous f, which is bounded from below, when $0 < \rho < \sigma < 1$.

The line search iteratively improves on the guesses for the minimum until an acceptable point is found. The pre-conditions (which are guaranteed to be satisfied by the conjugent gradient routine) are: the current point is `x0`, `f(x0)` and `g(x0)` are computed, $\texttt{f}'\texttt{(x0)} \leq \texttt{0}$ is guaranteed, and `max` is undefined. In pseudo-code, the algorithm is the following:

```
x1 := x0 and x2 := initial guess
compute f(x2) and g(x2)
loop {
    while { a2(x2) or b(x2) are violated } do {
        max := x2
        x2 := interpolate(x1,x2)
        compute f(x2) and g(x2)
    }
    if { a1(x2) is satisfied } do { return x2 }               [SUCCESS!]
    x3 := extrapolate(x1, x2, max)
    x1 := x2 and x2 := x3
    compute f(x2) and g(x2)
}
```

with the additional constraint, that a maximum of 20 epochs are allowed, to avoid infinite loops because of numerical inaccuracies; in this case the line search fails and the best approximation so far is returned. The notation `a2(x2)`, refers to the condition implied by the line $a2$ in fig. 1, at the point `x2`, and it returns either `violated` or `satisfied`. The solution is always bracketed between `x1` and `max` (whenever `max` is defined).

The initial step-size guess for the current iteration is computed in the step size from the previous iteration using the *slope ratio* method. The length of the previous step, δ_{i-1}, is multiplied by the ratio of previous and current slopes (but a maximum of 100):

$$\delta_i = \min\left(\frac{f'(x_{i-1})}{f'(x_i)}, 100\right)\delta_{i-1} \qquad \texttt{x2} = \texttt{x0} + \delta_i \qquad \text{with} \;\; \delta_0 = 1/(1 + |f'(x_0)|). \tag{2}$$

The limit of a factor of 100 for the relative step size is introduced to avoid errors when $f'(x_i)$ is numerically close to zero. This simple heuristic slope ratio method seems to work very well in practice. Typically only about 1.3 epochs per iteration are needed, indicating that the initial guess often lies in the region of acceptable points.

Interpolation between `x1` and `x2` is done by finding the minimum of a quadratic or cubic polynomial, fit at `f(x1)`, `f(x2)`, `f'(x1)` and `f'(x2)`. A cubic is used unless `f(x2)` > `f(x0)`, in which case `f'(x2)` is ignored and a quadratic is used, since in this case `x2` can be very far from the minimizer, and it is feared that the derivative out there may be misleading. If the interpolating polynomial does not have a (local) minimum inside the interval, bisection is used. In addition, if the point is with 10% of the interval length from an interval endpoint, then the new guess is moved to this distance in order to ensure exploration and avoid stagnation by repeated evaluation at essentially identical points.

Extrapolation from the `x1` to `x2` interval is done using a cubic, fit at `f(x1)`, `f(x2)`, `f'(x1)` and `f'(x2)`. If the cubic does not have a minimum, if the minimum does not correspond to an extrapolation or if the extrapolation is further than 3 times the interval length, then this point at 3 times the interval length is used is order to prevent uncontrolled extrapolation. If the new point is too close to the current point (within 10% of the interval length), this 10% length is used, to ensure exploration.

The conjugent gradient algorithm calls the above line search procedure for each of the directions that are computed. The entire minimization procedure terminates when a pre-specified number of function evaluations have been performed, or when two subsequent line searches have failed (this will usually happen for reasons of numerical inaccuracies when we are very close to a local minimum). All computations are done using double precision arithmetic.

Discussion

Other implementations of conjugent gradient methods have been proposed. Unfortunately it is quite rare that implementational details are given or discussed. It is also extremely rare to see good empirical evidence of the performance of optimisation procedures.

One paper which does address some of these issues is (Møller, 1990). He presents an implementation called Scaled Conjugate Gradients (SCG), which avoids the line search by using a finite differences approximation to the Hessian and computing a step size with a Newton like method. The SCG always uses 2 epochs per iteration, since the function and gradients are needed at the current point, and the gradient (which cannot be computed without first computing the function value) somewhere in the vicinity of the current point is needed for the finite differences approximation to the Hessian. The SCG method is thus very like a version of the present method modified to require at least one interpolation. The advantages of the present method are:

- a good guess (by the slope ratio algorithm) is used for the extra point needing evaluation, instead of a very local point for the finite differences method, thus giving more relevant information about the function.
- the SCG method ignores half of the function values which are computed, thus discarding useful information.
- the initial guess may be accepted, thus avoiding extra epochs (although it is conceivable that SCG might benefit from perhaps more accurate new points). Typically only 1.3 epochs per iteration are needed as opposed to CGS's 2.0.
- interpolation is (typically) done using a cubic (rather than SCG's quadratic) polynomial.
- rigorous criteria are used to ensure progress at each iteration.

For these reasons I prefer the present implementation, but it is conceivable that there are no significant differences between the methods in practice.

"minimize.c" 5 ≡

```
#include <math.h>
#include <stdlib.h>
#include "util.h"

extern int  no_wts;                              /* length of weight vector */
extern real *w;                                         /* weight vector */
extern real fgeval(real *dw);   /* evaluate function and partial derivatives */

real *dw1, *dw2,                /* two arrays of derivatives for all weights */
     *s;                           /* search direction used for linesearches */
int  nfgeval;  /* number of func/grad evals. so far; incremented by fgeval() */

int lns(f1, z, d1)
  real *f1,                                       /* current function value */
       *z,                                        /* guess for initial step */
       d1;                                                         /* slope */
{
  real RHO = 0.25, SIG = 0.5, INT = 0.1, EXT = 3.0;
  int  MAX = 20;
  real d2, d3, f2, f3, z2, z3, A, B, max = -1.0;
  int  i, k;

  for (i=0; i<no_wts; i++) w[i] += *z*s[i];                /* update weights */
  f2 = fgeval(dw2);
  d2 = 0.0; for (i=0; i<no_wts; i++) d2 += dw2[i]*s[i];
  f3=*f1; d3=d1; z3=-*z;               /* initialize point 3 equal to point 1 */
  k = nfgeval + MAX; while (nfgeval < k) { /* allow limited amount of search */
    while (((f2 > *f1+*z*RHO*d1) || (d2 > -SIG*d1)) && (nfgeval < k)) {
      max=*z;                                       /* tighten the bracket */
      if (f2 > *f1) z2=z3-(0.5*d3*z3*z3)/(d3*z3+f2-f3);     /* quadratic fit */
      else {                                                    /* cubic fit */
        A = 6.0*(f2-f3)/z3+3.0*(d2+d3);
        B = 3.0*(f3-f2)-z3*(d3+2.0*d2);
        z2 = (sqrt(B*B-A*d2*z3*z3)-B)/A;   /* numerical error possible - ok! */
      }
      if (z2 != z2) z2 = z3/2;                 /* if z2 is NaN then bisection */
      if (z2 > INT*z3) z2 = INT*z3;      /* bound solution away from current */
      if (z2 < (1.0-INT)*z3) z2 = (1.0-INT)*z3;        /* bound away from z3 */
      *z += z2;                                 /* update absolute stepsize */
      for (i=0; i<no_wts; i++) w[i] += z2*s[i];            /* update weights */
      f2 = fgeval(dw2);
      d2 = 0.0; for (i=0; i<no_wts; i++) d2 += dw2[i]*s[i];
      z3 -= z2;                   /* z3 is now relative to the location of z2 */
    }
    if (d2 > SIG*d1) { *f1 = f2; return 1; }                      /* SUCCESS */
    A = 6.0*(f2-f3)/z3+3.0*(d2+d3);             /* make cubic extrapolation */
    B = 3.0*(f3-f2)-z3*(d3+2.0*d2);
    z2 = -d2*z3*z3/(B+sqrt(B*B-A*d2*z3*z3));    /* num. error possible - ok! */
```

```
    if (z2 != z2)                                               /* z2 is NaN? */
      z2 = (max < -0.5) ? *z*(EXT-1.0) : 0.5*(max-*z);          /* bisection */
    else if (z2 < 0.0)                /* minimum is to the left of current? */
      z2 = (max < -0.5) ? *z*(EXT-1.0) : 0.5*(max-*z);          /* bisection */
    else if ((max > -0.5) && (z2+*z > max))          /* extrap. beyond max? */
      z2 = 0.5*(max-*z);                                        /* bisection */
    else if ((max < -0.5) && (z2+*z > *z*EXT))       /* extrap. beyond EXT? */
      z2 = *z*(EXT-1.0);                             /* set to extrap. limit */
    else if (z2<-z3*INT)                     /* too close to current point? */
      z2 = -z3*INT;
    else if ((max > -0.5) && (z2 < (max-*z)*(1.0-INT))) /* too close to max? */
      z2 = (max-*z)*(1.0-INT);
    f3=f2; d3=d2; z3=-z2;                               /* swap point 2 and 3 */
    *z += z2;
    for (i=0; i<no_wts; i++) w[i] += z2*s[i];              /* update weights */
    f2 = fgeval(dw2);
    d2 = 0.0; for (i=0; i<no_wts; i++) d2 += dw2[i]*s[i];
  }
  *f1 = f2;
  return 0;                                                /* linesearch failed */
}

int conj(iter, epoch, restart, costvalue)
  int  *iter,       /* "iter" and "epoch" indicates the maximum number of... */
       *epoch,  /* iterations or epochs allowed. Actual numbers are returned */
       *restart;      /* if (*restart) then restart CG with steepest descent */
  real *costvalue;                  /* return the value of the costfunction */
{
  static int  ls_failed = 0;              /* set to 1 if line search failed */
  int         j,                                     /* miscellaneous counter */
              cur_iter = 0;                       /* counts current iteration */
  static real fun, slope1, step;
  real        *tmp, y, z, q, slope2;
  extern int  TIMEOUT;            /* is set to one when SIGVTALRM is caught */

  nfgeval = 0;   /* global int "number of function and gradient evaluations" */
  if (*restart) {            /* start by using direction of steepest descent */
    fun = fgeval(dw1);
    slope1 = 0.0;
    for (j=0; j<no_wts; j++) { s[j] = -dw1[j]; slope1 -= s[j]*s[j]; }
    step = -1.0/(slope1-1.0);           /* set initial step-size to 1/(|s|+1) */
    *restart = 0;          /* probably we won't want to restart on next call */
  }
  while ((!TIMEOUT) && ((*epoch == 0) || (nfgeval < *epoch)) &&
         ((*iter == 0) || (cur_iter < *iter))) {
    cur_iter++;
    if (lns(&fun, &step, slope1)) {              /* if line search succeeded */
      y = z = q = 0.0; for (j=0; j<no_wts; j++)
        { y += dw2[j] * dw2[j]; z += dw1[j] * dw2[j]; q += dw1[j] * dw1[j]; }
      y = (y-z)/q;
      for (j=0; j<no_wts; j++) s[j] = y*s[j]-dw2[j];         /* new direction */
      tmp = dw2; dw2 = dw1; dw1 = tmp;                    /* swap derivatives */
      slope2 = 0.0; for (j=0; j<no_wts; j++) slope2 += dw1[j]*s[j];
      if (slope2 > 0.0) {     /* must be negative, else use steepest descent */
        slope2 = 0.0;
```

```
        for (j=0; j<no_wts; j++) { s[j] = -dw1[j]; slope2 -= s[j]*s[j]; }
      }
      step *= (slope1/slope2 > 100.0) ? 100.0 : slope1/slope2; slope1 = slope2;
      ls_failed = 0;
    } else {                                           /* line search failed */
      if (ls_failed)          /* break if previous failed, else try steepest */
        { *epoch = nfgeval; *iter = cur_iter; *costvalue = fun; return 0; }
      dw1 = dw2;                                           /* swap derivatives */
      slope1 = 0.0;
      for (j=0; j<no_wts; j++) { s[j] = -dw1[j]; slope1 -= s[j]*s[j]; }
      step = -1.0/(slope1-1.0);     /* set new step-size guess to 1/(|s|+1) */
      ls_failed = 1;
    }
  }
  *epoch = nfgeval; *iter = cur_iter; *costvalue = fun; return 1;
}
```
◇

References

Fletcher, R. (1987). *Practical methods for optimization.* John Wiley & sons, 2nd edition.

Møller, M. F. (1990). A scaled conjugent gradient algorithm for fast supervised learning. *Neural Networks*, 6:525–533.