Ch.7 Nonlinear optimization [Book, Chap. 5]

To appreciate the vast difference between linear optimization and nonlinear optimization, consider the relation

$$y = w_0 + \sum_{l=1}^{L} w_l f_l \,, \tag{1}$$

where $f_l = f_l(x_1, \ldots, x_m)$, and the polynomial fit is a special case.

Although the response variable y is nonlinearly related to the predictor variables $x_1, \ldots, x_m$ (as f_l is in general a nonlinear function), y is a linear function of the parameters $\{w_l\}$, and the objective function

$$J = \sum (y - y_{\rm d})^2 \,, \tag{2}$$

is a quadratic function of the $\{w_l\}$. This means that the objective function $J(w_0, \ldots, w_L)$ is a parabolic surface, which has a single minimum, the global minimum.

When y is a *nonlinear* function of $\{w_l\}$, the objective function surface is in general filled with numerous hills and valleys, i.e. many local minima besides the global minimum. (If there are symmetries among the parameters, there can even be multiple global minima).

Thus nonlinear optimization involves finding a global minimum among many local minima.

Nonlinear optimization is vastly more tricky than linear optimization, with no guarantee that the algorithm actually finds the global minimum, as it may become trapped by a local minimum.

For NN models, find the optimal parameters $\mathbf{w}$ which minimize J.

Common to solve the minimization problem using an iterative procedure.
Suppose the current approximation of the solution is $\mathbf{w}_0$. A Taylor series expansion of $J(\mathbf{w})$ around $\mathbf{w}_0$ yields

$$J(\mathbf{w}) = J(\mathbf{w}_0) + (\mathbf{w}-\mathbf{w}_0)^{\mathrm{T}} \nabla J(\mathbf{w}_0) + \frac{1}{2}(\mathbf{w}-\mathbf{w}_0)^{\mathrm{T}} \mathbf{H} (\mathbf{w}-\mathbf{w}_0) + \cdots , \tag{3}$$

where ∇J has components $\partial J/\partial w_i$, and $\mathbf{H}$ is the *Hessian matrix*, with elements

$$(\mathbf{H})_{ij} \equiv \left. \frac{\partial^2 J}{\partial w_i \partial w_j} \right|_{\mathbf{w}_0} . \tag{4}$$

If $\mathbf{w}_0$ is a minimum, then $\nabla J(\mathbf{w}_0) = 0$, and (3) (with the higher order terms ignored) reduces to an equation describing a parabolic surface.

Hence, near a minimum, assuming the Hessian matrix is nonzero, the objective function has an approximately parabolic surface.

Applying the gradient operator to (3), we obtain

$$\nabla J(\mathbf{w}) = \nabla J(\mathbf{w}_0) + \mathbf{H}\,(\mathbf{w} - \mathbf{w}_0) + \cdots \tag{5}$$

Next, derive an iterative scheme for finding the optimal $\mathbf{w}$, with $\mathbf{w}_0$ the current approximation of the optimal solution.

At the optimal $\mathbf{w}$, $\nabla J(\mathbf{w}) = 0$, $=>$ (5), with higher order terms ignored, yields

$$\mathbf{H}\,(\mathbf{w} - \mathbf{w}_0) = -\nabla J(\mathbf{w}_0)\,, \quad \text{i.e.} \;\; \mathbf{w} = \mathbf{w}_0 - \mathbf{H}^{-1}\nabla J(\mathbf{w}_0)\,. \tag{6}$$

This suggests the following iterative scheme for proceding from step k to step $k+1$:

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \mathbf{H}_k^{-1} \nabla J(\mathbf{w}_k)\,. \tag{7}$$

This is *Newton's method*. In the 1-dimensional case, (7) reduces to the familiar form

$$w_{k+1} = w_k - \frac{J'(w_k)}{J''(w_k)}\,, \tag{8}$$

for finding a root of $J'(w) = 0$, where the prime and double prime denote respectively the first and second derivatives.

In the multi-dimensional case, if $\mathbf{w}$ is of dimension L, then the Hessian matrix $\mathbf{H}_k$ is of dimension $L \times L$. Computing $\mathbf{H}_k^{-1}$, the inverse of an $L \times L$ matrix, may be computational too costly. Simplification is needed, $=>$ quasi-Newton methods.

7.1 Gradient descent methods [Book, Sect. 5.1]

The *gradient descent* or *steepest descent* method was used in the original back-propagation algorithm, where the parameters are updated at the kth step by

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \eta \nabla J(\mathbf{w}_k)\,, \tag{9}$$

with η (a positive scalar) being the learning rate.

Clearly (9) is a major simplification of Newton's method (7), with the learning rate η replacing $\mathbf{H}^{-1}$, the inverse of the Hessian matrix.

One also tries to reach the optimal $\mathbf{w}$ by descending along the negative gradient of J in (9), hence the name gradient descent or

steepest descent, as the negative gradient gives the direction of steepest descent.

An analogy is a hiker trying to descend in thick fog from a mountain to the bottom of a valley by taking the steepest descending path. Alas, this approach is surprisingly inefficient.

The learning rate η can be either (i) a fixed constant, or (ii) calculated by a line minimization algorithm.
In (i), one takes a step of fixed size along the direction of the negative gradient of J.

In (ii), proceed along the negative gradient of J until reaching the minimum of J along that direction.

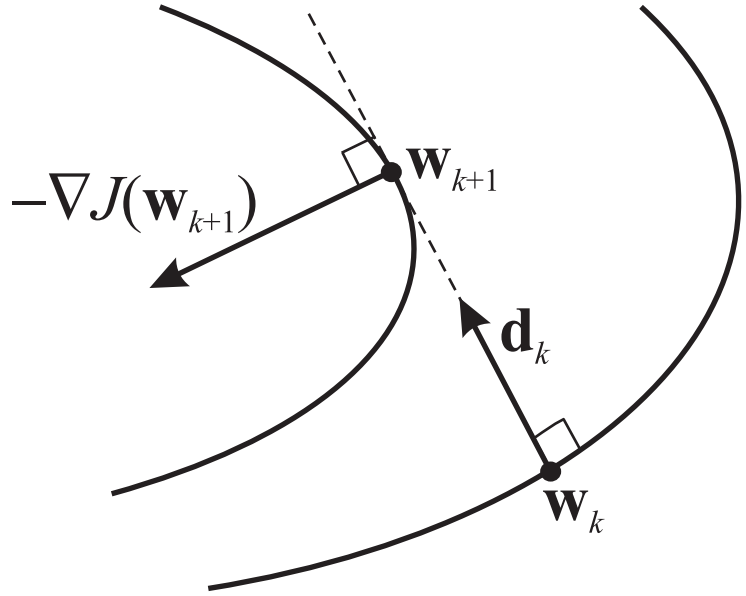

Figure : The gradient descent approach starts from the parameters $\mathbf{w}_k$ estimated at step k of an iterative optimization process. The descent path $\mathbf{d}_k$ is chosen along the negative gradient of the objective function J, which is the steepest descent direction. Note that $\mathbf{d}_k$ is perpendicular to the J contour where $\mathbf{w}_k$ lies. The descent along $\mathbf{d}_k$ proceeds until it is tangential to a second contour at $\mathbf{w}_{k+1}$, where the direction of steepest descent is given by $-\nabla J(\mathbf{w}_{k+1})$. The process is iterated.

More precisely, suppose at step k, we have estimated parameters $\mathbf{w}_k$. We then descend along the negative gradient of the objective function, i.e. travel along the direction

$$\mathbf{d}_k = -\nabla J(\mathbf{w}_k)\,. \tag{10}$$

We then travel along $\mathbf{d}_k$, with our path described by $\mathbf{w}_k + \eta \mathbf{d}_k$, until we reach the minimum of J along this direction.

Going further along this direction would ascending rather than descending, so we should stop at this minimum of J along $\mathbf{d}_k$, which occurs at

$$\frac{\partial}{\partial \eta} J(\mathbf{w}_k + \eta \mathbf{d}_k) = 0\,, \tag{11}$$

thereby yielding the optimal step size η. Differentiation by η gives

$$\mathbf{d}_k^{\mathrm{T}} \nabla J(\mathbf{w}_k + \eta \mathbf{d}_k) = 0\,. \tag{12}$$

With

$$\mathbf{w}_{k+1} = \mathbf{w}_k + \eta \mathbf{d}_k \,, \tag{13}$$

we can rewrite Eq. (12) as

$$\mathbf{d}_k^{\mathrm{T}} \nabla J(\mathbf{w}_{k+1}) = 0 \,, \quad \text{i.e.} \;\; \mathbf{d}_k \perp \nabla J(\mathbf{w}_{k+1}). \tag{14}$$

But since $\mathbf{d}_{k+1} = -\nabla J(\mathbf{w}_{k+1})$, we have

$$\mathbf{d}_k^{\mathrm{T}} \mathbf{d}_{k+1} = 0 \,, \quad \text{i.e.} \;\; \mathbf{d}_k \perp \mathbf{d}_{k+1} \,. \tag{15}$$

As the new direction $\mathbf{d}_{k+1}$ is orthogonal to the previous direction $\mathbf{d}_k$, $=>$ an inefficient zigzag path of descent.

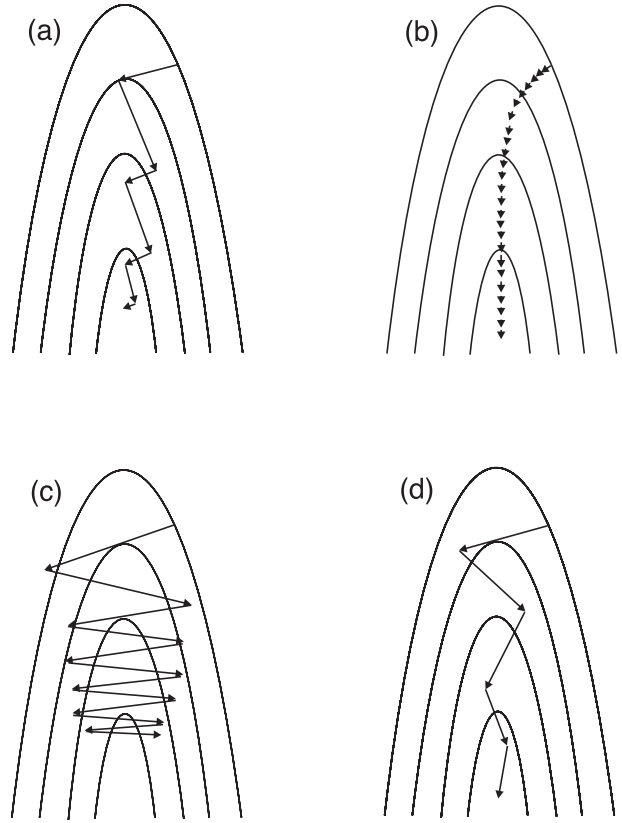

Figure : The gradient descent method with (a) line minimization (i.e. optimal step size η), (b) fixed step size which is too small, (c) fixed step size which is too large, and (d) momentum, which reduces the zigzag behaviour during descent.

The other alternative of using fixed step size is also inefficient, as a small step size results in taking too many steps, while a large step size results in an even more severely zigzagged path of descent.

One way to reduce the zigzag in the gradient descent scheme is to add '*momentum*' to the descent direction, so

$$\mathbf{d}_k = -\nabla J(\mathbf{w}_k) + \mu \mathbf{d}_{k-1}\,, \tag{16}$$

with μ the momentum parameter.
The momentum or memory of $\mathbf{d}_{k-1}$ prevents the new direction $\mathbf{d}_k$ to be orthogonal to $\mathbf{d}_{k-1}$, thereby reducing the zigzag.
The next estimate for the parameters in the momentum method is also given by (13).

The *conjugate gradient method* [Book, Sect. 5.2], automatically chooses the momentum parameter μ.

Other nonlinear optimization methods: Quasi-Newton method [Book, Sect. 5.3] and Levenberg-Marquardt method [Book, Sect. 5.4].

All 3 are better than the steepest descent method.

7.2 Evolutionary computation and genetic algorithms [Book, Sect. 5.5]

All optimization methods presented so far belong to the class of methods known as *deterministic optimization*, in that each step of the optimization process is determined by explicit formulas.
While such methods tend to converge to a minimum efficiently, they often converge to a nearby local minimum.

To find a global minimum, one usually has to introduce some stochastic element into the search.

A simple way is to repeat the optimization process many times, each starting from different random initial weights. These multiple runs will find multiple minima, and one hopes that the lowest minimum among them is the desired global minimum.

There is no guarantee that the global minimum has been found. Nevertheless by using a enough large number of runs and broadly distributed random initial weights, the global minimum can usually be found with such an approach.

Unlike deterministic optimization, *stochastic optimization* methods repeatedly introduce randomness during the search process to avoid

getting trapped in a local minimum. Such methods include *simulated annealing* and *evolutionary computation*.

Intelligence has emerged in Nature via biological evolution, so it is not surprising that *evolutionary computation* (EC) (Fogel, 2005) has become a significant branch of Computational Intelligence/Artificial Intelligence.

Among EC methods, genetic algorithms (GA) (Haupt and Haupt, 2004) were inspired by biological evolution where cross-over of genes from parents and genetic mutations results in a stochastic process which can lead to superior descendants after many generations.

The weight vector $\mathbf{w}$ of a model can be treated as a long strand of DNA, and an ensemble of solutions is treated like a population of organisms.

A part of the weight vector of one solution can be exchanged with a part from another solution to form a new solution, analogous to the *cross-over* of DNA material from two parents.

E.g., two parents have weight vectors $\mathbf{w}$ and $\mathbf{w}'$. A random position is chosen (in this example just before the third weight parameter) for an incision, and the second part of $\mathbf{w}'$ is connected to the first part of $\mathbf{w}$ and vice versa in the offsprings, i.e.

$$\begin{matrix} [w_1, w_2, w_3, w_4, w_5, w_6] \\ \\ [w_1', w_2', w_3', w_4', w_5', w_6'] \end{matrix} \quad \begin{matrix} \\ -\text{cross-over} \rightarrow \\ \\ \end{matrix} \quad \begin{matrix} [w_1', w_2', w_3, w_4, w_5, w_6] \\ \\ [w_1, w_2, w_3', w_4', w_5', w_6'] \end{matrix} \tag{17}$$

Genetic *mutation* can be simulated by randomly perturbing one of the weights w_j in the weight vector $\mathbf{w}$, i.e. randomly choose a j and replace w_j by $w_j + \epsilon$ for some random ϵ (usually a small random number).

These two processes introduce many new offsprings, but only the relatively fit offsprings have a high probability of surviving to reproduce. With the "survival of the fittest" principle pruning the offsprings, successive generations eventually converge towards the global optimum.

One must specify a *fitness function* f to evaluate the fitness of the individuals in a population. If for the ith individual in the population, its fitness is $f(i)$, then a fitness probability $P(i)$ can be defined as

$$P(i) = \frac{f(i)}{\sum_{i=1}^{N} f(i)} \quad , \tag{18}$$

where N is the total number of individuals in a population.

Individuals with high P will be given greater chances to reproduce, while those with low P will be given greater chances to die off.

The basic GA structure:
(i) Choose the population size (N) and the number of generations (N_g). Initialize the weight vectors of the population.
Repeat the following steps (ii)-(v) N_g times:
(ii) Calculate the fitness function f and the fitness probability P for each individual in the population.
(iii) Select a given number of individuals from the population, where the chance of an individual getting selected is given by its fitness probability P.

(iv) Duplicate the weight vectors of these individuals, then apply either the cross-over or the mutation operation on the various duplicated weight vectors to produce new offsprings.
(v) To keep the population size constant, individuals with poor fitness are selected (based on the probability $1 - P$) to die off, and are replaced by the new offsprings. (The fittest individual is never chosen to die).

Finally, after N_g generations, the individual with the greatest fitness is chosen as the solution.

To monitor the evolutionary progress over successive generations, one can check the average fitness of the population, by averaging the fitness f over all individuals in the population.

Q1: For the MLP NN model, you want to replace the nonlinear optimization using gradient descent methods by GA instead. What should the fitness function be?

In general, deterministic optimization using gradient descent methods would converge much quicker than stochastic optimization methods such as GA.

Three advantages with GA:

(1) In problems where the fitness function cannot be expressed in closed analytic form, gradient descent methods cannot be used effectively, whereas GA works well.

(2) When there are many local optima in the fitness function, gradient descent methods can be trapped too easily.

(3) GA can utilize parallel processors much more readily than gradient descent algorithms, since in GA different individuals in a population can be computed simultaneously on different processors.

In some NN applications, the individuals in a population do not all have the same network topology, e.g. they can have different number of hidden neurons. In such cases, GA can be used to find not only the optimal weights but also the optimal network topology.

E.g. evolutionary algorithm for robotic soccer team on Youtube:
`http://www.youtube.com/watch?v=cP035M_w82s&list=FLZKRbMQD_bb7Xc1y8NLIHzA&index=2`
`http://www.hindawi.com/journals/jr/2010/841286/`

Nonlinear optimization functions in Matlab

Deterministic optimization:

Matlab Nonlinear Optimization Toolbox:
fminunc (Quasi-Newton method)
`www.mathworks.com/help/toolbox/optim/ug/fminunc.html`
x = fminunc(f, x0)
starts at the point x0 and attempts to find a local minimum x of the function f. x0 can be a scalar, vector, or matrix.

For NN models, Matlab Neural Network Toolbox has many options for "training functions"
`www.mathworks.com/help/toolbox/nnet/ref/f7-23438.html#f7-9361`
For instance, trainlm uses Levenberg-Marquardt backpropagation to solve for the NN weights during training.

Stochastic optimization:
Matlab Global Optimization Toolbox:
Genetic algorithm:
`www.mathworks.com/products/global-optimization/description4.html`

x = ga(fitnessfcn, nvars)
finds a local unconstrained minimum, x, to the fitness function, fitnessfcn.
nvars is the number of variables in x.
The fitness function, fitnessfcn, accepts a vector x of size 1-by-nvars, and returns a scalar evaluated at x.

References:

Fogel, D. (2005). *Evolutionary Computation: Toward a New Philosophy of Machine Intelligence*. Wiley-IEEE, 3rd edition.

Haupt, R. L. and Haupt, S. E. (2004). *Practical Genetic Algortihms*. Wiley.

Price, K. V., Storn, R. M., and Lampinen, J. A. (2005). *Differential Evolution: A Practical Approach to Global Optimization*. Springer, Berlin.

Storn, R. and Price, K. (1997). Differential evolution – A simple and efficient heuristic for global optimization over continuous spaces. *Journal of Global Optimization*, 11:341–359. DOI 10.1023/A:1008202821328.