Νευρο-Ασαφής Υπολογιστική
Neuro-Fuzzy Computing

Διδάσκων –
Δημήτριος Κατσαρός

@ Τμ. ΗΜΜΥ
Πανεπιστήμιο Θεσσαλίας

Διάλεξη 10η

Numerical optimization techniques applied to Backpropagation: Newton, Conjugate Gradient (CGBP), Levenberg-Marquardt (LMBP), Quasi-Newton: BFGS, Adagrad, Adadelta, Adam AdaHessian

Second-order optimization methods

- Recall from your Numerical Analysis course:
  - Steepest Descent
    - is the simplest algorithm, but is often slow in converging
  - Newton's method
    - is much faster, but requires calculation of the Hessian and of its inverse
  - Conjugate Gradient
    - it does not require the calculation of second derivatives, and yet it still has the quadratic convergence property. (It converges to the minimum of a quadratic function in a finite number of iterations
- We will describe the Newton's method
- We will describe how the Conjugate Gradient algorithm can be used to train multilayer networks
  - We call this algorithm *Conjugate Gradient BackPropagation* (CGBP)
- We will briefly describe the rest of the methods

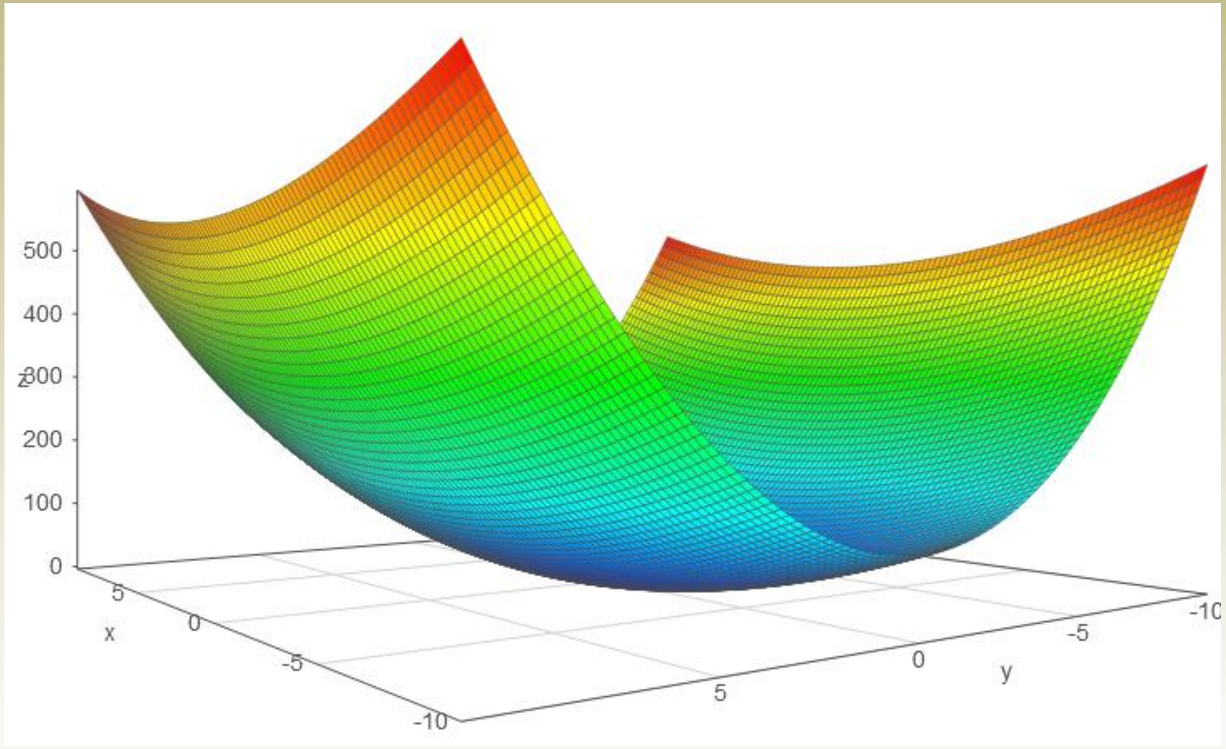
Multivariate Newton’s method

Multivariate Newton's method

- STEP-1
  - Calculate the gradient $\nabla f(\mathbf{x}_k)$ and the Hessian $\nabla^2 f(\mathbf{x}_k)$ (or $\mathbf{H}(\mathbf{x}_k)$)
- STEP-2
  - Compute the minimizing direction: $\mathbf{s}_k = [-\nabla^2 f(\mathbf{x}_k)]^{-1} \nabla f(\mathbf{x}_k)$
- STEP-3
  - Minimize$[f(\mathbf{x}_k+\lambda_k \mathbf{s}_k)]$ to find the optimal λ_κ ($\lambda_\kappa \geq 0$)
    - Either from: $df(\mathbf{x}_k - \lambda_\kappa [\nabla^2 f(\mathbf{x}_k)]^{-1} \nabla f(\mathbf{x}_k))/d\lambda_\kappa = 0$
    - Or using a one-dimensional optimization method
    - If **f** is quadratic, then <u>always</u> $\lambda_\kappa = 1$, for every k
- STEP-4
  - Find the next point $\mathbf{x}_{k+1} = \mathbf{x}_k + \lambda_k \mathbf{s}_k$
- STEP-5
  - Check convergence criterion

Newton's method for quadratic function

- Minimum: $f(\mathbf{x}) = x_1^2 + 5x_2^2 - 4$, starting from $\mathbf{x}_0 = (2, 2)^T$



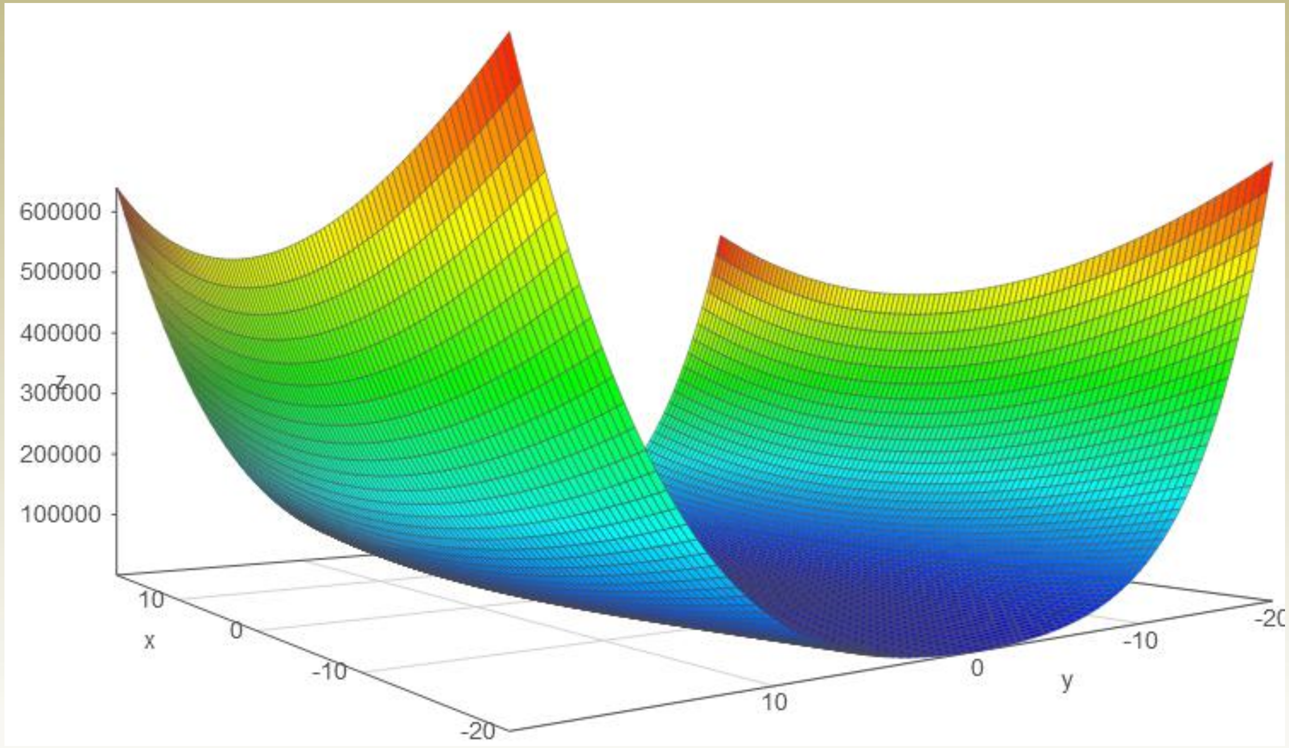
- Thus: $\nabla f(\mathbf{x}) = (2x_1, 10x_2)^T$ and $\mathbf{H} = \begin{bmatrix} 2 & 0 \\ 0 & 10 \end{bmatrix}, \forall k$

Newton's method for quadratic function

- We need to find the point $\mathbf{x}_1 = \mathbf{x}_0 + \lambda_k \mathbf{s}_k = \binom{2}{2} - \lambda_k \mathbf{H}^{-1} \nabla f(\mathbf{x}_0) =$ $\binom{2}{2} - \lambda_k \begin{bmatrix} 1/2 & 0 \\ 0 & 1/10 \end{bmatrix} \binom{4}{20} = \binom{2-2\lambda_k}{2-2\lambda_k}$
- From $\frac{df(\boldsymbol{x}_0 - \lambda_0 [\nabla^2 f(\boldsymbol{x}_0)]^{-1} \nabla f(\boldsymbol{x}_0))}{d\lambda_0} = 0 \rightarrow \frac{d(5(2-2\lambda_0)^2 - 4)}{d\lambda_0} = 0 \rightarrow$ $10(2-2\lambda_k)(-2) = 0$, which gives $\lambda_0 = 1$
- This result was expected (the whole process was not necessary) because the function is quadratic and thus our theory tells us that $\lambda_k = 1$, for every k
- Should we replace $\lambda_0 = 1$ in the previous slide's equation, we get $\mathbf{x}_1 = (0, 0)^T = \mathbf{x}^*$, the optimal point (with zero gradient there)
- The problem was solved in a single step in this case

Newton for non-quadratic function

- Minimum: $f(\mathbf{x}) = x_1^2 + x_1^2 x_2^2 + 3x_2^4$, starting from $\mathbf{x}_0 = (1, 1)^T$



- Thus: $\nabla f(\mathbf{x}) = (2x_1 + 2x_1x_2^2, 2x_1^2x_2 + 12x_2^3)^T$ and

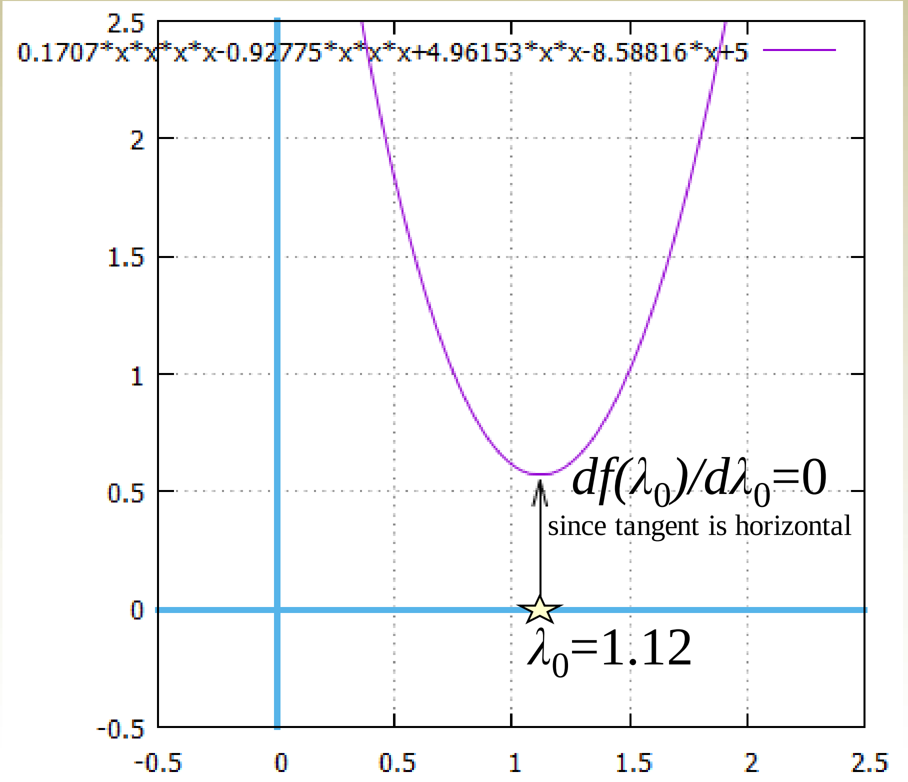
$$\mathbf{H} = \begin{bmatrix} 2x_2^2 + 2 & 4x_1x_2 \\ 4x_1x_2 & 2x_1^2 + 36x_2^2 \end{bmatrix}$$

Newton for non-quadratic function

- From the previous equations (at $\mathbf{x}_0$), we get: $\nabla f(\mathbf{x}_0) = (4, 14)^T$ and $\mathbf{H} = \begin{bmatrix} 4 & 4 \\ 4 & 38 \end{bmatrix}$
- The inverse of **H** is: $\mathbf{H}^{-1} = \frac{1}{136}\begin{bmatrix} 38 & -4 \\ -4 & 4 \end{bmatrix}$
- The first minimizing direction is: $\mathbf{s}_0 = \frac{-1}{136}\begin{bmatrix} 38 & -4 \\ -4 & 4 \end{bmatrix}\begin{pmatrix} 4 \\ 14 \end{pmatrix} = \frac{-1}{136}\begin{pmatrix} 96 \\ 56 \end{pmatrix} = -\begin{pmatrix} 0.70588 \\ 0.41176 \end{pmatrix}$
- So, $\mathbf{x}_1 = \begin{pmatrix} 1 \\ 1 \end{pmatrix} - \lambda_0 \begin{pmatrix} 0.70588 \\ 0.41176 \end{pmatrix} = \begin{pmatrix} 1 - 0.70588\lambda_0 \\ 1 - 0.41176\lambda_0 \end{pmatrix}$
- We now replace $\mathbf{x}_1$ into the formula of $f(\mathbf{x})$ and we get:
- $f\big(\mathbf{x}_0 - \lambda_0[\nabla^2 f(\mathbf{x}_0)]^{-1}\, \nabla f(\mathbf{x}_0)\big) = (1-0.70588\,\lambda_0)^2 + (1-0.70588\,\lambda_0)^2(1-0.41176\,\lambda_0)^2 + 3(1-0.41176\,\lambda_0)^4$

Newton for non-quadratic function

- After calculations we get (purple curve): $f(\mathbf{x}_0-\lambda_0[\nabla^2 f(\mathbf{x}_0)]^{-1}\nabla f(\mathbf{x}_0)) = f(\lambda_0) = 0.1707\,\lambda_0^4 - 0.92775\,\lambda_0^3 + 4.96153\,\lambda_0^2 - 8.58816\,\lambda_0 + 5$
- So we seek that λ_0 for which: $df(\lambda_0)/d\lambda_0 = 0$
- To find λ_0 which satisfies the above, we may use the *golden section method*, which gives: $\lambda_0 = 1.12$
- Substituting this value of λ_0 into the equation for $\mathbf{x}_1$, we get $\mathbf{x}_1 = \begin{pmatrix} 0.20941 \\ 0.53882 \end{pmatrix}$
- We continue by checking the convergence criterion …



- We may use $\lambda_k = 1\ \forall k$, even though the function is non quadratic, but this would result in more iterations before convergence

Background on Conjugate Gradient

Προσέγγιση κατά Taylor μιας συνάρτησης

Ανάπτυγμα της σειράς Taylor μέχρι πρώτης και δεύτερης τάξης για την πολυδιάσταση συνάρτηση f(**x**) γύρω από το σημείο **x*** (ελάχιστο) και αποτελούν γραμμική και τετραγωνική προσέγγιση της συνάρτησης, αντίστοιχα

$$\begin{aligned} f(\mathbf{x}_k) &= f(\mathbf{x}^\star) + \nabla f(\mathbf{x}^\star)(\mathbf{x}_k - \mathbf{x}^\star) \\ &+ O(||\mathbf{x}_k - \mathbf{x}^\star||^2) \end{aligned}$$

$$\begin{aligned} f(\mathbf{x}_k) &= f(\mathbf{x}^\star) + \nabla f(\mathbf{x}^\star)(\mathbf{x}_k - \mathbf{x}^\star) \\ &+ \frac{1}{2}(\mathbf{x}_k - \mathbf{x}^\star)^T \nabla^2 f(\mathbf{x}^\star)(\mathbf{x}_k - \mathbf{x}^\star) \\ &+ O(||\mathbf{x}_k - \mathbf{x}^\star||^3) \end{aligned}$$

Προσέγγιση κατά Taylor μιας συνάρτησης

Ισοδύναμα έχουμε:

$$f(\mathbf{x}_k) = f(\mathbf{x}^\star) + \nabla f(\xi_k)(\mathbf{x}_k - \mathbf{x}^\star)$$

$$\begin{aligned} f(\mathbf{x}_k) &= f(\mathbf{x}^\star) + \nabla f(\mathbf{x}^\star)(\mathbf{x}_k - \mathbf{x}^\star) \\ &\quad + \frac{1}{2}(\mathbf{x}_k - \mathbf{x}^\star)^T \nabla^2 f(\xi_k)(\mathbf{x}_k - \mathbf{x}^\star) \end{aligned}$$

όπου το ξ_k βρίσκεται πάνω στο ευθύγραμμο τμήμα που ορίζεται από τα σημεία x_k και x^*

Αυτή η συνάρτηση ονομάζεται τετραγωνική μορφή (quadratic form την ξανασυναντήσαμε στην εκπαίδευση της ADALINE) και έχει την γενική μορφή:

$$f(\mathbf{x}) = a + \mathbf{x}^T\mathbf{b} + \frac{1}{2}\mathbf{x}^T\mathbf{H}\mathbf{x}$$

Υπενθύμιση της Steepest Descent

- Σε κάθε βήμα, η Steepest Descent προσεγγίζει την αντικειμενική συνάρτηση κάνοντας χρήση της πρώτης παραγώγου (της πρώτης εξίσωσης της προηγούμενης διαφάνειας)

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \lambda_k \frac{\nabla f(\mathbf{x}_k)}{||\nabla f(\mathbf{x}_k)||}$$

όπου το λ_k βρίσκεται με βελτιστοποίηση

Εάν όμως η συνάρτηση είναι τετραγωνική μορφή, τότε:

$$\begin{aligned}\lambda_k &= -\frac{\nabla^T f(\mathbf{x}_k)\mathbf{s}_k}{\mathbf{s}_k^T \mathbf{H} \mathbf{s}_k} \\ &= \frac{||\nabla f(\mathbf{x}_k)||^3}{\nabla^T f(\mathbf{x}_k)\mathbf{H}\nabla f(\mathbf{x}_k)}\end{aligned}$$

Συζυγείς διευθύνσεις

- ΟΡΙΣΜΟΣ. Μια μέθοδος έχει την ιδιότητα του **τετραγωνικού τερματισμού** όταν συγκλίνει στο βέλτιστο σημείο **x*** μιας τετραγωνικής αντικειμενικής συνάρτησης σε γνωστό πεπερασμένο αριθμό επαναλήψεων

- ΟΡΙΣΜΟΣ. Ένα σύνολο n γραμμικώς ανεξαρτήτων μη μηδενικών διανυσμάτων $\mathbf{s}_1, \mathbf{s}_2, \ldots, \mathbf{s}_n$ είναι συζυγή ως προς έναν θετικά ορισμένο πίνακα **H** εάν $\mathbf{s}_i \mathbf{H} \mathbf{s}_j = 0 \ \forall \ 1 \leq i \neq j \leq n$

- ΟΡΙΣΜΟΣ. Μια μέθοδος βελτιστοποίησης ονομάζεται **μέθοδος συζυγών διευθύνσεων** εάν κατά την εφαρμογή της παράγει συζυγείς διευθύνσεις και εφαρμοζόμενη σε μια τετραγωνική μορφή με Hessian πίνακα **H** έχει την ιδιότητα του τετραγωνικού τερματισμού

Συζυγείς διευθύνσεις

- ΘΕΩΡΗΜΑ. Σε κάθε αντικειμενική συνάρτηση που είναι τετραγωνική μορφή και έχει ένα ελάχιστο, εάν ακολουθήσουμε μια μέθοδο που παράγει συζυγείς ως προς τον Hessian πίνακα διευθύνσεις, το ελάχιστο θα εντοπιστεί σε n το πολύ βήματα, ένα για κάθε συζυγή διεύθυνση

- ΘΕΩΡΗΜΑ. Εάν f($\mathbf{x}$) είναι τετραγωνική μορφή, και $\mathbf{s_0}$, $\mathbf{s_1}$, …, $\mathbf{s_k}$ σύνολο συζυγών ως προς τον Hessian πίνακα διευθύνσεων, οι οποίες παράγονται με κάποια μέθοδο, μέχρι την προσέγγιση x_{k+1}, τότε:

$$\nabla^T f(\mathbf{x_{k+1}})\mathbf{s_j} = 0 \text{ για } j=1,2,\ldots k$$

Παράδειγμα

ΠΡΟΒΛΗΜΑ. Να εντοπιστεί το ελάχιστο της $f(x)=x^2_1+5x^2_2-10$

ΛΥΣΗ.

- Η κλίση είναι $\nabla f(\mathbf{x})= (2x_1, 10x_2)^T$
- Ο Hessian είναι: $\begin{bmatrix} 2 & 0 \\ 0 & 10 \end{bmatrix}$
- Ας εκκινήσουμε από το $\mathbf{x_0}=(2\ 2)^T$
- Τότε, $\nabla f(\mathbf{x_0})= (4, 20)^T$
- Επιλέγοντας **<u>τυχαία</u>** μια διεύθυνση εκκίνησης, έστω την $\mathbf{s_0}=(1/2, \sqrt{3}/2)^T$ που έχει $||\mathbf{s_0}||=1$, παίρνω ότι $\boldsymbol{\lambda_0}= -2.4150635$
  - Δεν είναι θετικό!!! γιατί η $\mathbf{s_0}$ δεν είναι διεύθυνση μείωσης
- $\mathbf{x_1}=\mathbf{x_0}+\boldsymbol{\lambda_0}\mathbf{s_0}=\ldots=\begin{bmatrix} 0.792468250 \\ -0.09150634 \end{bmatrix}$

Παράδειγμα

- Η κλίση εδώ είναι $\nabla f(\mathbf{x}_1) = (1.5849365, -0.9150634)^T$
- Θέλουμε τώρα να υπολογίσουμε την επόμενη διεύθυνση ελαχιστοποίησης με τέτοιον τρόπο ώστε να είναι συζυγής ως προς την $\mathbf{s}_0$ σε σχέση με τον Hessian **H** και να είναι μοναδιαία, δηλαδή να ισχύουν: $\mathbf{s}^T{}_1\mathbf{H}\mathbf{s}_0=0$ και $\mathbf{s}^T{}_1\mathbf{s}_1=1$
- Εάν συμβολίσουμε $\mathbf{s}_1 = (s^1{}_1, s^1{}_2)$ προκύπτει από τις προηγούμενες εξισώσεις ότι:

$$s_1^1 + 5\frac{\sqrt{3}}{2}s_2^1 = 0$$

$$(s_1^1)^2 + (s_2^1)^2 = 1$$

- Από τις δυο λύσεις του συστήματος αυτού, επιλέγουμε εκείνη που δίνει διεύθυνση μείωσης, δηλαδή να ισχύει:

$$\mathbf{s}^T{}_1\nabla f(\mathbf{x}_1) < 0$$

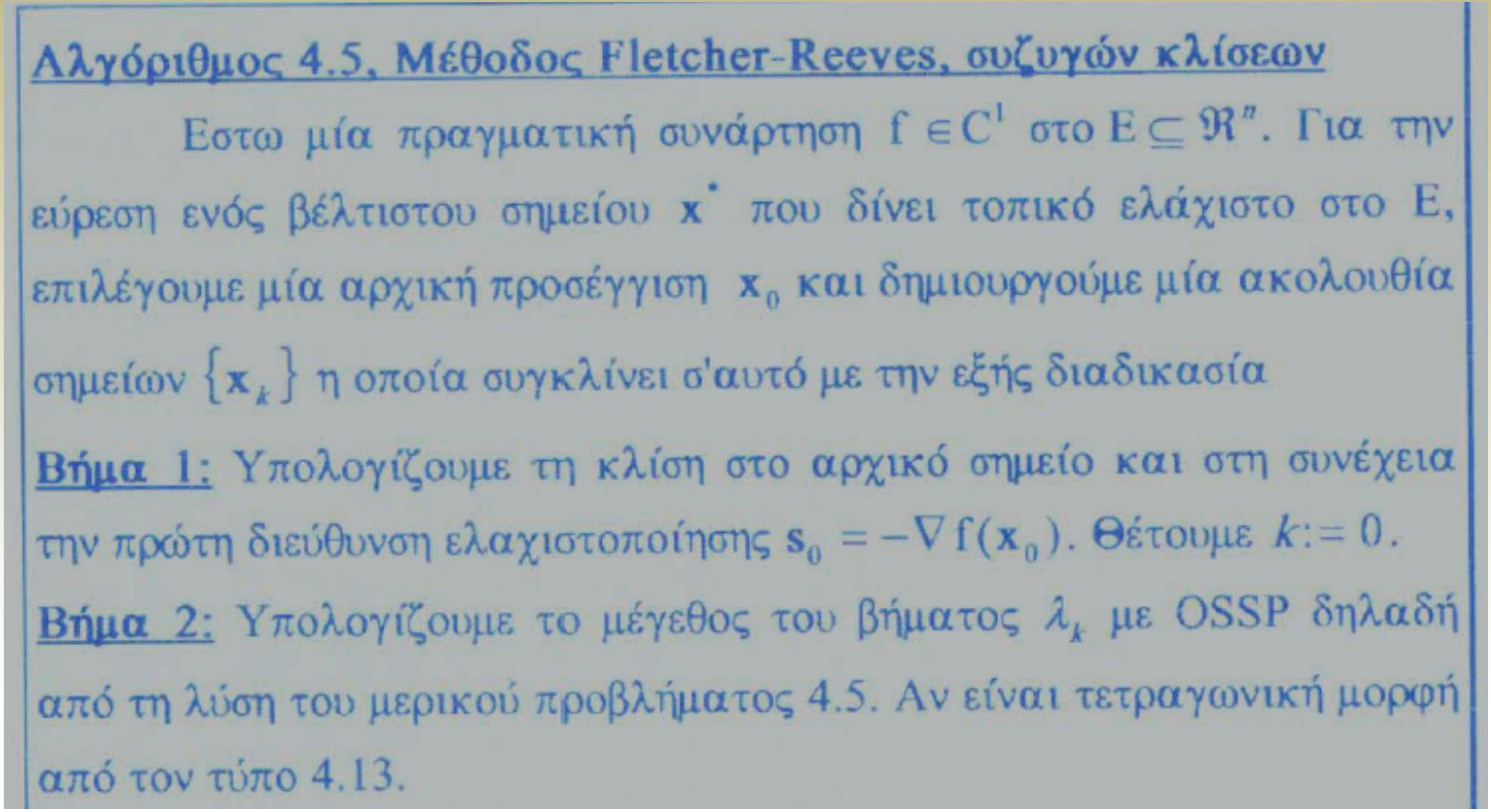
Παράδειγμα

- Αυτή είναι η $\mathbf{s_1}$= (-0.99339927, 0.11470787)^T
- Αυτή είναι η επόμενη διεύθυνση, και είναι μοναδιαία και συζυγή της προηγούμενης, κατά μήκος της οποίας θα κινηθούμε με βήμα λ_1
- Από τον τύπο της Διαφάνειας-14, έχω ότι λ_1=0.79773385
- Παρτηρήστε ότι το λ_1 είναι θετικό όπως θα έπρεπε αφού κινούμαστε σε διεύθυνση μείωσης
- Η επόμενη προσέγγιση είναι: $\mathbf{x_2} = \mathbf{x_1} + \boldsymbol{\lambda_1}\mathbf{s_1} = \dots \approx (0\ \ 0)^T$
- Αυτό είναι το βέλτιστο και βρέθηκε σε 2 βήματα (είναι τεταγωνική η αντικειμενική συνάρτηση με διάσταση 2)

Μέθοδοι συζυγών κλίσεων

- Προτάθηκε αρχικά από τους Hestenes & Stiefel (1952) και Beckman (1960)
- Γνωστότερη παραλλαγή αυτή των Fletcher-Reeves (1964)
  - Παράγει στο βήμα k την διεύθυνση $\mathbf{s}_\mathbf{k}$ η οποία είναι γραμμικός συνδυασμός της $-\nabla f(\mathbf{x}_k)$ δηλαδή της διεύθυνσης της μέγιστης αλλαγής στην τρέχουσα προσέγγιση, και των προηγούμενων διευθύνσεων $\mathbf{s}_0, \mathbf{s}_1, \ldots, \mathbf{s}_{k-1}$
  - Οι συντελεστές του γραμμικού συνδυασμού επιλέγονται ώστε οι παραγόμενες διευθύνσεις από προσέγγιση σε προσέγγιση να είναι συζυγείς ως προς τον Hessian πίνακα της αντικειμενικής συνάρτησης
  - ΤΕΛΙΚΑ προκύπτει ότι για να υπολογίσουμε αυτούς τους συντελεστές χρειαζόμαστε μόνο την τρέχουσα κλίση $\nabla f(\mathbf{x}_k)$ και την ακριβώς προηγούμενη $\nabla f(\mathbf{x}_{k-1})$

Fletcher-Reeves conjugate gradient



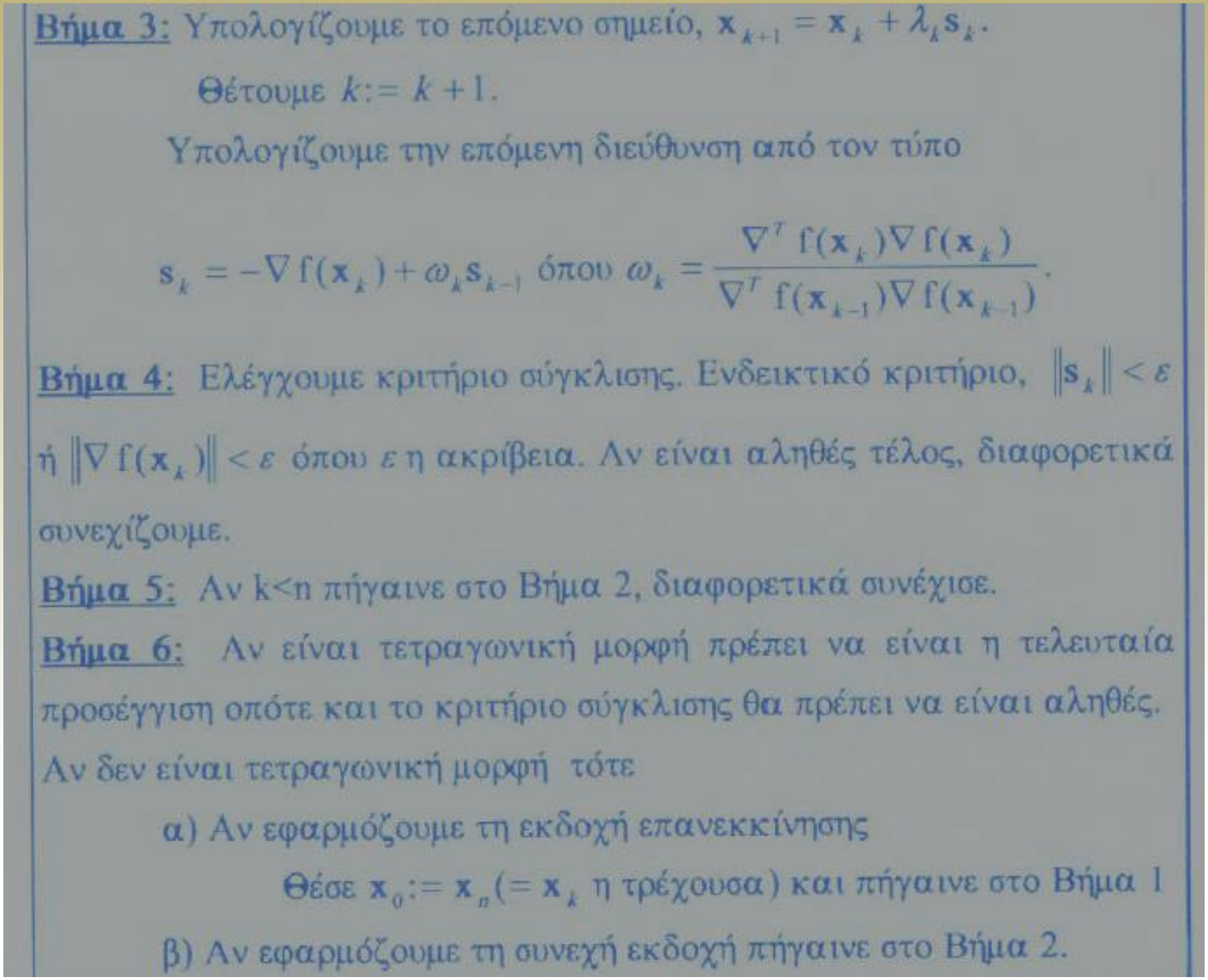
<u>Αλγόριθμος 4.5, Μέθοδος Fletcher-Reeves, συζυγών κλίσεων</u>

Εστω μία πραγματική συνάρτηση $f \in C^1$ στο $E \subseteq \Re^n$. Για την εύρεση ενός βέλτιστου σημείου $\mathbf{x}^*$ που δίνει τοπικό ελάχιστο στο Ε, επιλέγουμε μία αρχική προσέγγιση $\mathbf{x}_0$ και δημιουργούμε μία ακολουθία σημείων $\{\mathbf{x}_k\}$ η οποία συγκλίνει σ'αυτό με την εξής διαδικασία

Βήμα 1: Υπολογίζουμε τη κλίση στο αρχικό σημείο και στη συνέχεια την πρώτη διεύθυνση ελαχιστοποίησης $\mathbf{s}_0 = -\nabla f(\mathbf{x}_0)$. Θέτουμε $k := 0$.

Βήμα 2: Υπολογίζουμε το μέγεθος του βήματος λ_k με OSSP δηλαδή από τη λύση του μερικού προβλήματος 4.5. Αν είναι τετραγωνική μορφή από τον τύπο 4.13.

Fletcher-Reeves conjugate gradient



Βήμα 3: Υπολογίζουμε το επόμενο σημείο, $\mathbf{x}_{k+1} = \mathbf{x}_k + \lambda_k \mathbf{s}_k$.

Θέτουμε $k := k+1$.

Υπολογίζουμε την επόμενη διεύθυνση από τον τύπο

$$\mathbf{s}_k = -\nabla f(\mathbf{x}_k) + \omega_k \mathbf{s}_{k-1} \text{ όπου } \omega_k = \frac{\nabla^T f(\mathbf{x}_k)\nabla f(\mathbf{x}_k)}{\nabla^T f(\mathbf{x}_{k-1})\nabla f(\mathbf{x}_{k-1})}.$$

Βήμα 4: Ελέγχουμε κριτήριο σύγκλισης. Ενδεικτικό κριτήριο, $\|\mathbf{s}_k\| < \varepsilon$ ή $\|\nabla f(\mathbf{x}_k)\| < \varepsilon$ όπου ε η ακρίβεια. Αν είναι αληθές τέλος, διαφορετικά συνεχίζουμε.

Βήμα 5: Αν k<n πήγαινε στο Βήμα 2, διαφορετικά συνέχισε.

Βήμα 6: Αν είναι τετραγωνική μορφή πρέπει να είναι η τελευταία προσέγγιση οπότε και το κριτήριο σύγκλισης θα πρέπει να είναι αληθές. Αν δεν είναι τετραγωνική μορφή τότε

α) Αν εφαρμόζουμε τη εκδοχή επανεκκίνησης

Θέσε $\mathbf{x}_0 := \mathbf{x}_n (= \mathbf{x}_k$ η τρέχουσα) και πήγαινε στο Βήμα 1

β) Αν εφαρμόζουμε τη συνεχή εκδοχή πήγαινε στο Βήμα 2.

Παραλλαγές μεθόδων συζυγών κλίσεων

Ακολουθούν τον προηγούμενο αλγόριθμο, αλλά διαφέρουν στον τύπο υπολογισμού του ω_k

- Μέθοδος του Daniel (1967)

$$\omega_k = \frac{\nabla^T f(\mathbf{x}_k)\nabla^2 f(\mathbf{x}_k)\mathbf{s}_{k-1}}{\mathbf{s}_{k-1}^T \nabla^2 f(\mathbf{x}_k)\mathbf{s}_{k-1}}$$

- Μέθοδος των Crowder & Wolfe (1971)

$$\omega_k = \frac{\nabla^T f(\mathbf{x}_k)(\nabla f(\mathbf{x}_k) - \nabla f(\mathbf{x}_{k-1}))}{\mathbf{s}_{k-1}^T(\nabla f(\mathbf{x}_k) - \nabla f(\mathbf{x}_{k-1}))}$$

- Μέθοδος των Polak & Ribiere (1969), Polyak (1969)

$$\omega_k = \frac{\nabla^T f(\mathbf{x}_k)(\nabla f(\mathbf{x}_k) - \nabla f(\mathbf{x}_{k-1}))}{||\nabla f(\mathbf{x}_{k-1})||^2}$$

Drawbacks of basic Conjugate Gradient

- This conjugate gradient algorithm cannot be applied directly to the neural network training task, because the performance index is not quadratic. This affects the algorithm in two ways:
  - First, we can not minimize the function along a line, as required in Step 2
  - Second, the exact minimum will not normally be reached in a finite number of steps, and therefore the algorithm will need to be reset after some set number of iterations

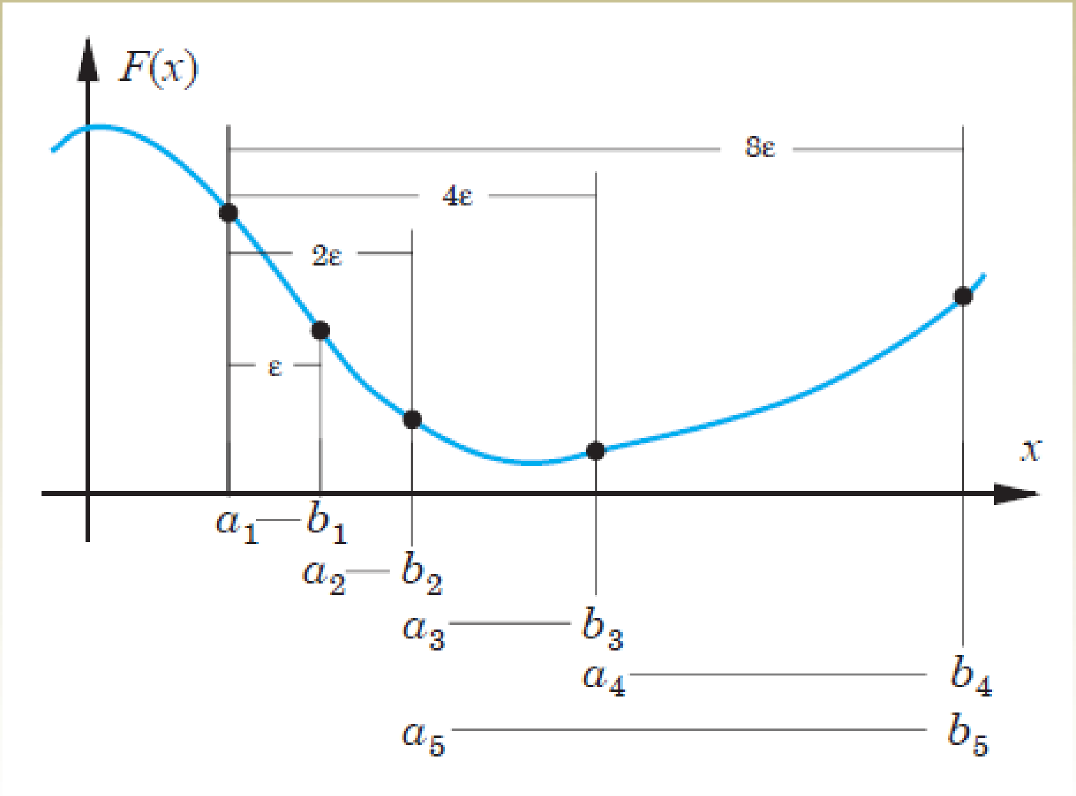
Interval location

- Let’s address the linear search first
- We need to have a general procedure for locating the minimum of a function in a specified direction
- This will involve two steps:
  - interval location
  - interval reduction
- The purpose of the interval location step is to find some initial interval that contains a local minimum
- The interval reduction step then reduces the size of the initial interval until the minimum is located to the desired accuracy

Interval location

- We will use a function comparison method to perform the interval location step, illustrated in the following figure

- We begin by evaluating the performance index at an initial point, represented by α_1. This point corresponds to the current values of the network weights and biases. In other words, we are evaluating $F(\mathbf{x}_0)$
- The next step is to evaluate the function at a second point, represented by b_1, which is a distance ε from the initial point, along the first search direction $\mathbf{p}_0$. In other words, we are evaluating $F(\mathbf{x}_0+\varepsilon\mathbf{p}_0)$



- This process stops when the function increases between two consecutive evaluations. The minimum is bracketed by the two points α_5 and b_5. The minimum may occur either in the interval $[\alpha_3,b_3]$ or in the interval $[\alpha_4,b_4]$.

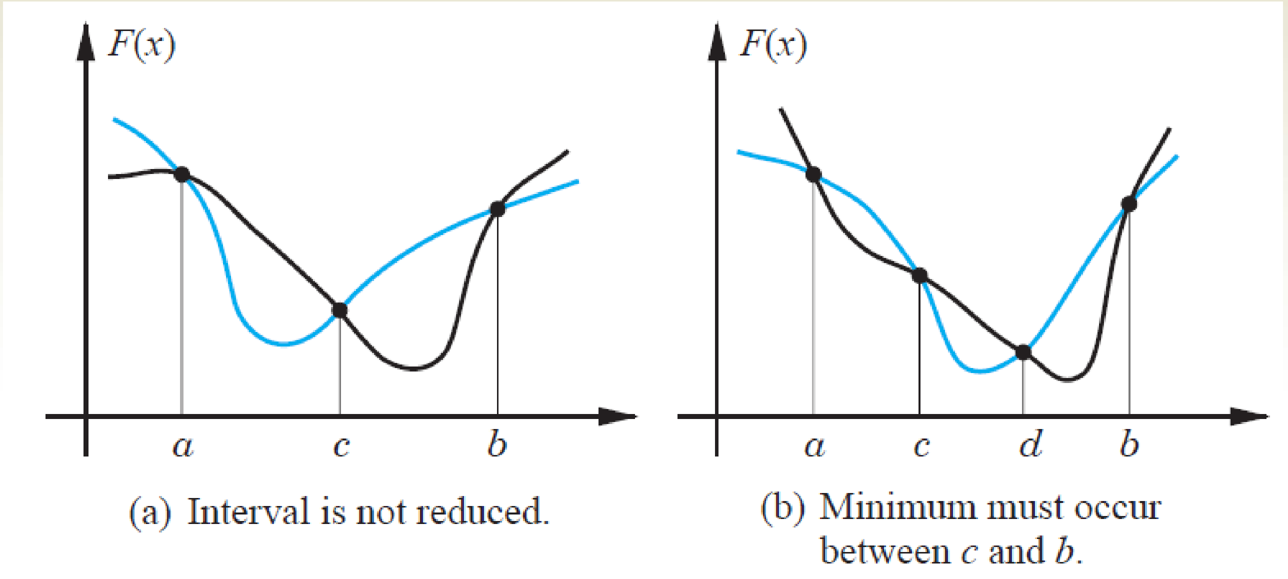
Interval reduction

- The next step is interval reduction, which involves evaluating the function at points inside the interval $[a_5,b_5]$
  - From the figure below we can see that we will need to evaluate the function at two internal points (at least) in order to reduce the size of the interval of uncertainty
- The let figure shows that one internal function evaluation does not provide us with any information on the location of the minimum
- However, if we evaluate the function at two points c and d, as in the right figure, we can reduce the interval of uncertainty

➢ If F(c) > F(d), then the minimum must occur in the interval [c, b]

➢ If F(c) < F(d), then the minimum must occur in the interval [a, d]

We are assuming that there is a single minimum located in the initial interval.



(a) Interval is not reduced.

(b) Minimum must occur between c and b.

Interval reduction: Golden section search

- We need to determine the locations of points c and d
- We will use *Golden Section search*, which is designed to reduce the number of function evaluations required
  - At each iteration one new function evaluation is required

τ is user-set tolerance

```
$\tau = 0.618$
Set  $c_1 = a_1 + (1-\tau)(b_1 - a_1)$, $F_c = F(c_1)$.
     $d_1 = b_1 - (1-\tau)(b_1 - a_1)$, $F_d = F(d_1)$.
For $k = 1, 2, \ldots$ repeat
    If $F_c < F_d$ then
        Set  $a_{k+1} = a_k$; $b_{k+1} = d_k$; $d_{k+1} = c_k$
             $c_{k+1} = a_{k+1} + (1-\tau)(b_{k+1} - a_{k+1})$
             $F_d = F_c$; $F_c = F(c_{k+1})$
    else
        Set  $a_{k+1} = c_k$; $b_{k+1} = b_k$; $c_{k+1} = d_k$
             $d_{k+1} = b_{k+1} - (1-\tau)(b_{k+1} - a_{k+1})$
             $F_c = F_d$; $F_d = F(d_{k+1})$
    end
end until $b_{k+1} - a_{k+1} < tol$
```

Final adjustments to develop the CGBP

- There is one more modification to the conjugate gradient algorithm that needs to be made before we apply it to neural network training
- For quadratic functions the algorithm will converge to the minimum in at most n iterations, where n is the number of parameters being optimized
- The mean squared error performance index for multilayer networks is not quadratic, therefore the algorithm would not normally converge in n iterations
- The development of the conjugate gradient algorithm does not indicate what search direction to use once a cycle of n iterations has been completed
  - There have been many procedures suggested, but the simplest method is to reset the search direction to the steepest descent direction (negative of the gradient) after n iterations

Critique of CGBP

- CGBP algorithm converges in many fewer iterations than other algorithms
- This is a little deceiving, since each iteration of CGBP requires more computations than these other methods
  - there are many function evaluations involved in each iteration of CGBP

- Even so, CGBP has been shown to be one of the fastest batch training algorithms for multilayer networks
  - C. Charalambous, "*Conjugate gradient algorithm for efficient training of artificial neural networks*," **IEE Proceedings**, vol. 139, no. 3, pp. 301–310, 1992

Numerical optimization techniques applied to Backpropagation: Levenberg-Marquardt algorithm

Levenberg-Marquardt basic algorithm

- The Levenberg-Marquardt algorithm is a variation of Newton's method that was designed for minimizing functions that are sums of squares of other nonlinear functions
- This is very well suited to neural network training (recall that their performance index is the mean squared error)
- Newton's method for optimizing a performance index F(**x**) is as follows:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \mathbf{A}_k^{-1}\mathbf{g}_k$$

where

$$\mathbf{A}_k^{-1} \equiv \nabla^2 F(\mathbf{x})|_{\mathbf{x}=\mathbf{x}_k}$$

and

$$\mathbf{g}_k \equiv \nabla F(\mathbf{x})|\mathbf{x} = \mathbf{x}_k$$

Levenberg-Marquardt basic algorithm

- If we assume that F(**x**) is a sum of squares function

$$F(\mathbf{x}) = \sum_{i=1}^{N} v_i^2(\mathbf{x}) = \mathbf{v}^T(\mathbf{x})\mathbf{v}(\mathbf{x})$$

then the j-th element of the gradient would be

$$[\nabla F(\mathbf{x})]_j = \frac{\partial F(\mathbf{x})}{\partial x_j} = 2\sum_{i=1}^{N} v_i(\mathbf{x}) \frac{\partial v_i(\mathbf{x})}{\partial x_j}$$

- The gradient can therefore be written in matrix form

$$\nabla F(\mathbf{x}) = 2\mathbf{J}^T(\mathbf{x})\mathbf{v}(\mathbf{x})$$

where the *Jacobian matrix* is:

$$\mathbf{J}(\mathbf{x}) = \begin{bmatrix} \frac{\partial v_1(\mathbf{x})}{\partial x_1} & \frac{\partial v_1(\mathbf{x})}{\partial x_2} & \cdots & \frac{\partial v_1(\mathbf{x})}{\partial x_n} \\ \frac{\partial v_2(\mathbf{x})}{\partial x_1} & \frac{\partial v_2(\mathbf{x})}{\partial x_2} & \cdots & \frac{\partial v_2(\mathbf{x})}{\partial x_n} \\ \vdots & \vdots & \vdots & \\ \frac{\partial v_N(\mathbf{x})}{\partial x_1} & \frac{\partial v_N(\mathbf{x})}{\partial x_2} & \cdots & \frac{\partial v_N(\mathbf{x})}{\partial x_n} \end{bmatrix}$$

Levenberg-Marquardt basic algorithm

- Next we wish to find Hessian matrix. The *k,j* element would be:

$$[\nabla^2 F(\mathbf{x})]_{k,j} = \frac{\partial^2 F(\mathbf{x})}{\partial x_k \partial x_j} = 2\sum_{i=1}^{N}\left\{\frac{\partial v_i(\mathbf{x})}{\partial x_k}\frac{\partial v_i(\mathbf{x})}{\partial x_j} + v_i(\mathbf{x})\frac{\partial^2 v_i(\mathbf{x})}{\partial x_k \partial x_j}\right\}$$

- The Hessian can be expressed in matrix form:

$$\nabla^2 F(\mathbf{x}) = 2\mathbf{J}^T(\mathbf{x})\mathbf{J}(\mathbf{x}) + 2\mathbf{S}(\mathbf{x})$$

where

$$\mathbf{S}(\mathbf{x}) = \sum_{i=1}^{N} v_i(\mathbf{x})\nabla^2 v_i(\mathbf{x})$$

- If we assume that **S(x)** is small, the we can approximate

$$\nabla^2 F(\mathbf{x}) = 2\mathbf{J}^T(\mathbf{x})\mathbf{J}(\mathbf{x})$$

Levenberg-Marquardt basic algorithm

- So, the Newton method evolves into the *Gauss-Newton* method:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - [2\mathbf{J}^T(\mathbf{x}_k)\mathbf{J}(\mathbf{x}_k)]^{-1} 2\mathbf{J}^T(\mathbf{x}_k)\mathbf{v}(\mathbf{x}_k)$$

$$\mathbf{x}_{k+1} = \mathbf{x}_k - [\mathbf{J}^T(\mathbf{x}_k)\mathbf{J}(\mathbf{x}_k)]^{-1} \mathbf{J}^T(\mathbf{x}_k)\mathbf{v}(\mathbf{x}_k)$$

- One problem with the Gauss-Newton method is that the matrix $\mathbf{H}=\mathbf{J}^T\mathbf{J}$ may not be invertible
- This can be overcome by using the following modification to the approximate Hessian matrix: $\mathbf{G}=\mathbf{H}+\mu\mathbf{I}$
  - (Figure out why this matrix can be made invertible)

Levenberg-Marquardt basic algorithm

- This lead to the Levenberg-Marquardt algorithm:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - [\mathbf{J}^T(\mathbf{x}_k)\mathbf{J}(\mathbf{x}_k) + \mu_k \mathbf{I}]^{-1}\mathbf{J}^T(\mathbf{x}_k)\mathbf{v}(\mathbf{x}_k)$$

- Or $\Delta\mathbf{x}_k = -[\mathbf{J}^T(\mathbf{x}_k)\mathbf{J}(\mathbf{x}_k) + \mu_k \mathbf{I}]^{-1}\mathbf{J}^T(\mathbf{x}_k)\mathbf{v}(\mathbf{x}_k)$

- The algorithm begins with μ_k set to some small value (e.g., 0.01)
- If a step does not yield a smaller value for F(**x**), then the step is repeated with μ_k multiplied by some factor $\theta>1$ (e.g., $\theta=10$)
- Eventually F(**x**) should decrease, since we would be taking a small step in the direction of steepest descent
- If a step does produce a smaller value for F(**x**), then μ_k is divided by θ for the next step, so that the algorithm will approach Gauss-Newton, which should provide faster convergence
- The algorithm provides a nice compromise between the speed of Newton's method and the guaranteed convergence of steepest descent

Training with Levenberg-Marquardt

- The performance index for multilayer network training is the mean squared error
- If each target occurs with equal probability, the mean squared error is proportional to the sum of squared errors over the Q targets in the training set:

$$F(\mathbf{x}) = \sum_{q=1}^{Q} (\mathbf{t}_q - \mathbf{a}_q)^T (\mathbf{t}_q - \mathbf{a}_q) = \sum_{q=1}^{Q} \mathbf{e}_q^T \mathbf{e}_q = \sum_{q=1}^{Q} \sum_{j=1}^{S^M} (e_{j,q})^2 = \sum_{i=1}^{N} (v_i)^2$$

- This is equivalent to the performance index for which Levenberg-Marquardt was designed
- It should be a straightforward matter to adapt the algorithm for network training, but it turns out that it does require some care in working out the details

Training with Levenberg-Marquardt

- The key step in the Levenberg-Marquardt algorithm is the computation of the Jacobian matrix
- To perform this computation we will use a variation of the backpropagation algorithm
- To create the Jacobian matrix we need to compute the derivatives of the errors, instead of the derivatives of the squared errors (as we did for the standard backpropagation)

Jacobian calculation

- Recall Jacobian's form from (slide 33)
- Error vector: $\mathbf{v}^T = [v_1 v_2 \ldots v_N] = [e_{1,1} e_{2,1} \ldots e_{S^M,1} e_{1,2} \ldots e_{S^M,Q}]$
- Parameter vector: $\mathbf{x}^T = [x_1 x_2 \ldots x_n] = [w^1_{1,1} w^1_{1,2} \ldots w^1_{S^1,R} b^1_1 \ldots b^1_{S^1} w^2_{1,1} \ldots b^M_{S^M}]$

  $N = Q \times S^M$ and $n = S^1(R+1) + S^2(S^1 + 1) + \ldots + S^M(S^{M-1} + 1)$

- So: $$\mathbf{J}(\mathbf{x}) = \begin{bmatrix} \frac{\partial e_{1,1}}{\partial w^1_{1,1}} & \frac{\partial e_{1,1}}{\partial w^1_{1,2}} & \cdots & \frac{\partial e_{1,1}}{\partial w^1_{S^1,R}} & \frac{\partial e_{1,1}}{\partial b^1_1} \\ \frac{\partial e_{2,1}}{\partial w^1_{1,1}} & \frac{\partial e_{2,1}}{\partial w^1_{1,2}} & \cdots & \frac{\partial e_{2,1}}{\partial w^1_{S^1,R}} & \frac{\partial e_{2,1}}{\partial b^1_1} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \frac{\partial e_{S^M,1}}{\partial w^1_{1,1}} & \frac{\partial e_{S^M,1}}{\partial w^1_{1,2}} & \cdots & \frac{\partial e_{S^M,1}}{\partial w^1_{S^1,R}} & \frac{\partial e_{S^M,1}}{\partial b^1_1} \\ \frac{\partial e_{1,2}}{\partial w^1_{1,1}} & \frac{\partial e_{1,2}}{\partial w^1_{1,2}} & \cdots & \frac{\partial e_{1,2}}{\partial w^1_{S^1,R}} & \frac{\partial e_{1,2}}{\partial b^1_1} \\ \vdots & \vdots & \vdots & \vdots & \vdots \end{bmatrix}$$

Jacobian calculation

- The terms in this Jacobian matrix can be computed by a simple modification to the backpropagation algorithm
- Standard backpropagation calculates terms like:

$$\frac{\partial \hat{F}(\mathbf{x})}{\partial x_l} = \frac{\partial \mathbf{e}_q^T \mathbf{e}_q}{\partial x_l}$$

- For the elements of the Jacobian matrix that are needed for the Levenberg-Marquardt algorithm we need to calculate terms like:

$$[\mathbf{J}]_{h,l} = \frac{\partial v_h}{\partial x_l} = \frac{\partial e_{k,q}}{\partial x_l}$$

- In our derivation of standard backpropagation:

$$\frac{\partial \hat{F}}{\partial w_{i,j}^m} = \frac{\partial \hat{F}}{\partial n_i^m} \frac{\partial n_i^m}{\partial w_{i,j}^m} \qquad \text{where} \qquad s_i^m \equiv \frac{\partial \hat{F}}{\partial n_i^m}$$

Marquardt sensitivity

- The backpropagation process computed the sensitivities through a recurrence relationship from the last layer backward to the first layer
- We can use the same concept to compute the terms needed for the Jacobian matrix if we define *Marquardt sensitivity*:

$$\tilde{s}_{i,h}^{m} \equiv \frac{\partial v_h}{\partial n_{i,q}^{m}} = \frac{\partial e_{k,q}}{\partial n_{i,q}^{m}}$$

where h=(q-1)S^M + k

- We can compute the elements of the Jacobian

$$[\mathbf{J}]_{h,l} = \frac{\partial v_h}{\partial x_l} = \frac{\partial e_{k,q}}{\partial w_{i,j}^{m}} = \frac{\partial e_{k,q}}{\partial n_{i,q}^{m}} \times \frac{\partial n_{i,q}^{m}}{\partial w_{i,j}^{m}} = \tilde{s}_{i,h}^{m} \times \frac{\partial n_{i,q}^{m}}{\partial w_{i,j}^{m}} = \tilde{s}_{i,h}^{m} \times \alpha_{j,q}^{m-1}$$

- or if x_l is a bias

$$[\mathbf{J}]_{h,l} = \frac{\partial v_h}{\partial x_l} = \frac{\partial e_{k,q}}{\partial b_{i}^{m}} = \frac{\partial e_{k,q}}{\partial n_{i,q}^{m}} \times \frac{\partial n_{i,q}^{m}}{\partial b_{i}^{m}} = \tilde{s}_{i,h}^{m} \times \frac{\partial n_{i,q}^{m}}{\partial b_{i}^{m}} = \tilde{s}_{i,h}^{m}$$

Marquardt sensitivity

- Marquardt sensitivities can be computed through the same recurrence relations as the standard sensitivities with one modification at the final layer
- For the Marquardt sensitivities at the final layer we have:

$$\tilde{s}_{i,h}^{M} = \frac{\partial v_h}{\partial n_{i,q}^{M}} = \frac{\partial e_{k,q}}{\partial n_{i,q}^{M}} = \frac{\partial (t_{k,q} - \alpha_{k,q}^{M})}{\partial n_{i,q}^{M}} = -\frac{\partial \alpha_{k,q}^{M}}{\partial n_{i,q}^{M}} = \begin{cases} -f'^{M}(n_{i,q}^{M}) & \text{for } i = k \\ 0 & \text{for } i \neq k \end{cases}$$

- Therefore when the input $\mathbf{p_q}$ has been applied to the network and the corresponding network $\mathbf{a^M_q}$ output has been computed, the Levenberg-Marquardt backpropagation is initialized with:

$$\tilde{\mathbf{S}}_q^{M} = -\mathbf{F}'^{M}(\mathbf{n}_q^{M})$$

where $\mathbf{F}'^{M}(\mathbf{n}^{M})$is define as in standard backpropagation

- The columns can also be backpropagated together using

$$\tilde{\mathbf{S}}_q^{m} = \mathbf{F}'^{m}(\mathbf{n}_q^{m})(\mathbf{W}^{m+1})^{T}\mathbf{S}_q^{m+1}$$

Marquardt sensitivity

- The total Marquardt sensitivity matrices for each layer are then created by augmenting the matrices computed for each input:

$$\tilde{\mathbf{S}}^m = \left[\tilde{\mathbf{S}}_1^m | \tilde{\mathbf{S}}_2^m | \ldots | \tilde{\mathbf{S}}_Q^m\right]$$

- Note that for each input that is presented to the network we will backpropagate S^M sensitivity vectors
  - This is because we are computing the derivatives of each individual error, rather than the derivative of the sum of squares of the errors.
- For every input applied to the network there will be S^M errors (one for each element of the network output)
- For each error there will be one row of the Jacobian matrix
- After the sensitivities have been backpropagated, the Jacobian matrix is computed using equation of slide 41

Levenberg-Marquardt backpropagation

1. Present all inputs to the network and compute the corresponding outputs and the errors $\mathbf{e}_q=\mathbf{t}_q-\mathbf{a}^M{}_q$. Compute the sum of squared errors over all inputs, $F(\mathbf{x})$ (slide 37)
2. Compute the Jacobian matrix. Calculate the sensitivities with the recurrence relations (equations in slide 42). Augment the individual matrices into the Marquardt sensitivities. Compute the elements of the Jacobian matrix with equations in slide 41
3. Solve equation in slide 36 to obtain $\Delta\mathbf{x}_k$
4. Recompute the sum of squared errors using $\mathbf{x}_k+\Delta\mathbf{x}_k$. If this new sum of squares is smaller than that computed in Step-1, then divide μ by θ, let $\mathbf{x}_{k+1}=\mathbf{x}_k+\Delta\mathbf{x}_k$ and go back to Step-1. If the sum of squares is not reduced, then multiply μ by θ and go back to Step-3

Critique of LMBP

- Even given the large number of computations, however, the LMBP algorithm appears to be the fastest neural network training algorithm for moderate numbers of network parameters
  - M. T. Hagan and M. Menhaj, "*Training feedforward networks with the Marquardt algorithm*," **IEEE Transactions on Neural Networks**, vol. 5, no. 6, 1994.
- The key drawback of the LMBP algorithm is the storage requirement: The algorithm must store the approximate Hessian matrix $\mathbf{J}^T\mathbf{J}$
  - This is an nxn matrix, where n is the number of parameters (weights and biases)
  - Recall that the other methods discussed need only store the gradient, which is an n-dimensional vector
  - When the number of parameters is very large, it may be impractical to use the Levenberg-Marquardt algorithm

Recent approaches: First-order and second-order optimizers

First-order optimizers

Adagrad (COLT 2010 & JMLR 2011)

$$\mathbf{g}_t = \partial_{\mathbf{w}} F(a_t, f(\mathbf{p}_t, \mathbf{w}))$$

$$\mathbf{s}_t = \mathbf{s}_{t-1} + \mathbf{g}_t^2$$

$$\mathbf{w}_t = \mathbf{w}_{t-1} - \frac{\alpha}{\sqrt{\mathbf{s}_t + \epsilon}} \mathbf{g}_t$$

- The operations are applied *coordinate wise*
- ϵ is an additive constant that ensures that we do not divide by 0
- Accumulating squared gradients in $\mathbf{s}_t$ means that $\mathbf{s}_t$ grows essentially at linear rate (somewhat slower than linearly in practice, since the gradients initially diminish). This leads to $O(1/\sqrt{t})$ learning rate, albeit adjusted on a per coordinate basis. For convex problems this is perfectly adequate
- In deep learning, though, we might want to decrease the learning rate rather more slowly. This led to a number of Adagrad variants

Adagrad (COLT 2010 & JMLR 2011)

- Adagrad relies on only first order information, but has some properties of second order methods
  - While there is the hand tuned global learning rate, each dimension has its own dynamic rate. Since this dynamic rate grows with the inverse of the gradient magnitudes, large gradients have smaller learning rates and small gradients have large learning rates
    - This has the nice property, as in second order methods, that the progress along each dimension evens out over time. This is very beneficial for training deep neural networks since the scale of the gradients in each layer is often different by several orders of magnitude, so the optimal learning rate should take that into account. Additionally, this accumulation of gradient in the denominator has the same effects as annealing, reducing the learning rate over time
  - Since the magnitudes of gradients are factored out in ADAGRAD, this method can be sensitive to initial conditions of the parameters and the corresponding gradients
    - If the initial gradients are large, the learning rates will be low for the remainder of training. This can be combatted by increasing the global learning rate, making the ADAGRAD method sensitive to the choice of learning rate. Also, due to the continual accumulation of squared gradients in the denominator, the learning rate will continue to decrease throughout training, eventually decreasing to zero and stopping training completely. ADADELTA was created to overcome the sensitivity to the hyperparameter selection as well as to avoid the continual decay of the learning rates

RMSProp (Lecture 6.5 – Coursera: NN and ML 2012)

- One of the key issues in Adagrad is that the learning rate decreases at a predefined schedule of effectively $O(1/\sqrt{t})$. While this is generally appropriate for convex problems, it might not be ideal for nonconvex ones, such as those encountered in deep learning. Yet, the coordinate-wise adaptivity of Adagrad is highly desirable as a preconditioner
- We need to decouple rate scheduling from coordinate-adaptive learning rates. The issue is that Adagrad accumulates the squares of the gradient $\mathbf{g}_t$ into a state vector $\mathbf{s}_t = \mathbf{s}_{t-1}+\mathbf{g}^2_t$. As a result $\mathbf{s}_t$ keeps on growing without bound due to the lack of normalization, essentially linearly as the algorithm converges
- One way of fixing this problem would be to use $\mathbf{s}_t/t$. For reasonable distributions of $\mathbf{g}_t$ this will converge. Unfortunately it might take a very long time until the limit behavior starts to matter since the procedure remembers the full trajectory of values. An alternative is to use a leaky average in the same way we used in the momentum method, i.e., $\mathbf{s}_t \leftarrow \gamma\, \mathbf{s}_{t-1} + (1 - \gamma)\, \mathbf{g}^2_t$ for some parameter $\gamma > 0$

RMSProp (Lecture 6.5 – Coursera: NN and ML 2012)

$$\mathbf{s}_t = \gamma \mathbf{s}_{t-1} + (1-\gamma)\mathbf{g}_t^2$$

$$\mathbf{w}_t = \mathbf{w}_{t-1} - \frac{\alpha}{\sqrt{\mathbf{s}_t + \epsilon}}\mathbf{g}_t$$

- The constant $\epsilon > 0$ is typically set to 10^{-6} to ensure that we do not suffer from division by zero or overly large step sizes

Adadelta (arxiv.org 2012)

- Adadelta is a variant of AdaGrad. The main difference lies in the fact that it decreases the amount by which the learning rate is adaptive to coordinates. Moreover, traditionally it referred to as not having a learning rate since it uses the amount of change itself as calibration for future change
- Adadelta uses two state variables, $\mathbf{s}_t$ to store a leaky average of the second moment of the gradient and $\Delta\mathbf{w}_t$ to store a leaky average of the second moment of the change of parameters in the model itself

$$\mathbf{s}_t = \gamma \mathbf{s}_{t-1} + (1-\gamma)\mathbf{g}_t^2$$

$$\mathbf{w}_t = \mathbf{w}_{t-1} - \mathbf{g}'_t, \quad \text{rescaled gradient: } \mathbf{g}'_t = \frac{\sqrt{\Delta\mathbf{w}_{t-1} + \epsilon}}{\sqrt{\mathbf{s}_t + \epsilon}} \mathbf{g}_t$$

$$\Delta\mathbf{w}_t = \gamma \Delta\mathbf{w}_{t-1} + (1-\gamma)\mathbf{g}'^2_t$$

Adam (ICLR 2015)

- Adaptive Moment Estimation (Adam) uses exponential weighted moving averages (also known as leaky averaging) to obtain an estimate of both the momentum and also the second moment of the gradient. That is, it uses the state variables

$$\mathbf{v}_t = \beta_1 \mathbf{v}_{t-1} + (1 - \beta_1)\mathbf{g}_t$$

$$\mathbf{s}_t = \beta_2 \mathbf{s}_{t-1} + (1 - \beta_2)\mathbf{g}_t^2$$

- Here β_1 and β_2 are nonnegative weighting parameters. Common choices for them are β_1=0.9 and β_2 =0.999. That is, the variance estimate moves much more slowly than the momentum term. Note that if we initialize $\mathbf{v}_0 = \mathbf{s}_0 = 0$ we have a significant amount of bias initially towards smaller values. This can be addressed by using the fact that $\sum_{i=0}^{t} \beta^i = \frac{1-\beta^t}{1-\beta}$ to re-normalize terms. Correspondingly the normalized state variables are given by:

$$\hat{\mathbf{v}}_t = \frac{\mathbf{v}_t}{1 - \beta_1^t} \quad \text{and} \quad \hat{\mathbf{s}}_t = \frac{\mathbf{s}_t}{1 - \beta_2^t}$$

Adam (ICLR 2015)

- We can now write out the update equations. First, we rescale the gradient in a manner very much akin to that of RMSProp to obtain:

$$\mathbf{g}'_t = \frac{\alpha \hat{\mathbf{v}}_t}{\sqrt{\hat{\mathbf{s}}_t} + \epsilon}$$

- Unlike RMSProp the update uses the momentum $\hat{v}_t$ rather than the gradient itself. Moreover, there is a slight cosmetic difference as the rescaling happens using $1/(\sqrt{\hat{s}_t} + \in)$ instead of $1/\sqrt{\hat{s}_t + \in}$. The former works arguably slightly better in practice, hence the deviation from RMSProp. Typically we pick $\epsilon = 10^{-6}$ for a good trade-off between numerical stability and fidelity
- Now we have all the pieces in place to compute updates. That is:

$$\mathbf{w}_t = \mathbf{w}_{t-1} - \mathbf{g}'_t$$

- Reviewing the design of Adam its inspiration is clear. Momentum and scale are clearly visible in the state variables. Their rather peculiar definition forces us to debias terms (this could be fixed by a slightly different initialization and update condition). Second, the combination of both terms is pretty straightforward, given RMSProp. Last, the explicit learning rate α allows us to control he step length to address issues of convergence

Second-order optimizers

Introduction

- Choosing the right hyper-parameter for optimizing a NN training has become **dark-art**
- Even the *choice of the optimizer* is a *hyper-parameter*

Task	**CV**	**NLP**	**Recommendation System**
Optimizer choice	SGD	AdamW	Adagrad

- First order methods only use gradient information and do not consider the curvature properties of the loss landscape, thereby leading to their suboptimal behaviour
- Second order methods, on the other hand, are specifically designed to capture and exploit the curvature of the loss landscape and to incorporate both gradient and Hessian information

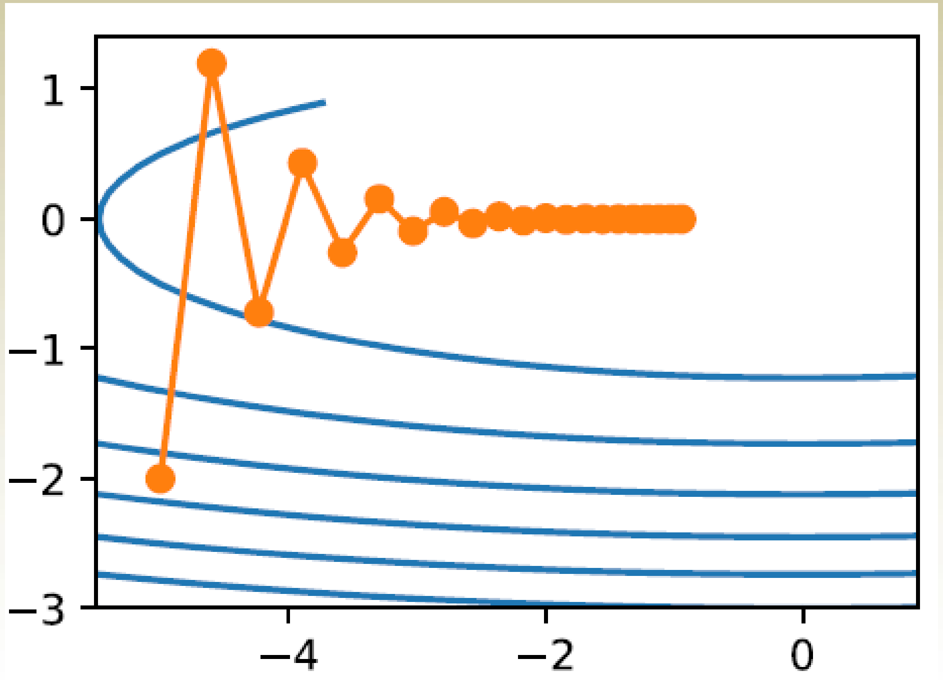
Introduction

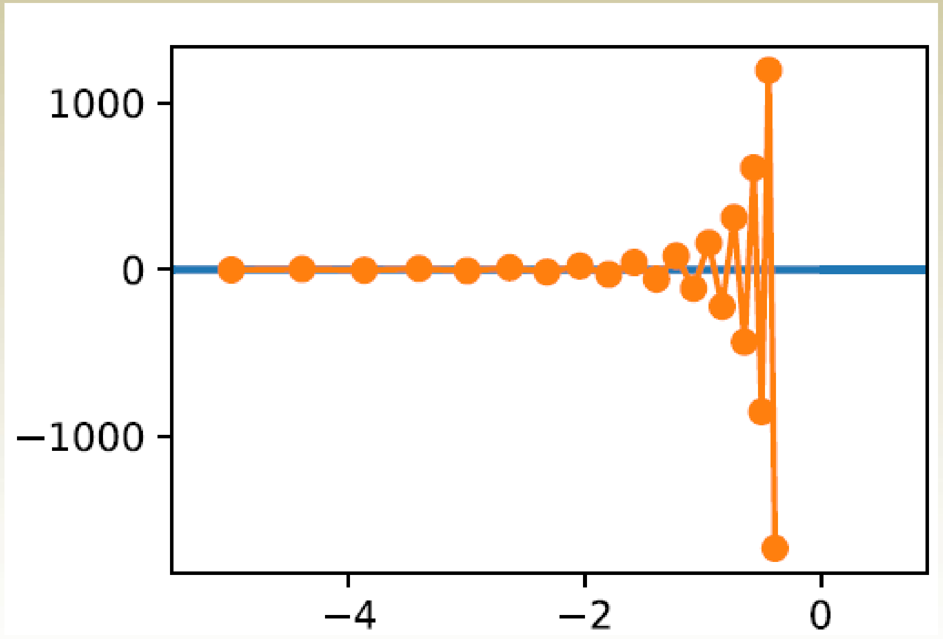
- The main idea underlying second order methods involves *preconditioning* the gradient vector before using it for weight update. The preconditioner automatically rotates and rescales the gradient vector. This has a very intuitive motivation related to the curvature of the loss function landscape
- For a general problem, different parameter dimensions exhibit different curvature properties
  - For example, the loss could be very flat in one dimension and very sharp in another. As a result, the step size taken by the optimizer should be different for these dimensions, and we would prefer to take bigger steps for the flatter directions and relatively smaller steps for the sharper directions
- Second order methods capture this curvature difference, by normalizing different dimensions through rotation and scaling of the gradient vector before the weight update. Nonetheless, this comes at a cost
- Despite the theoretically faster convergence rate of second order methods, they are rarely used for training NN models. This is due in part to their high computational cost

Introduction

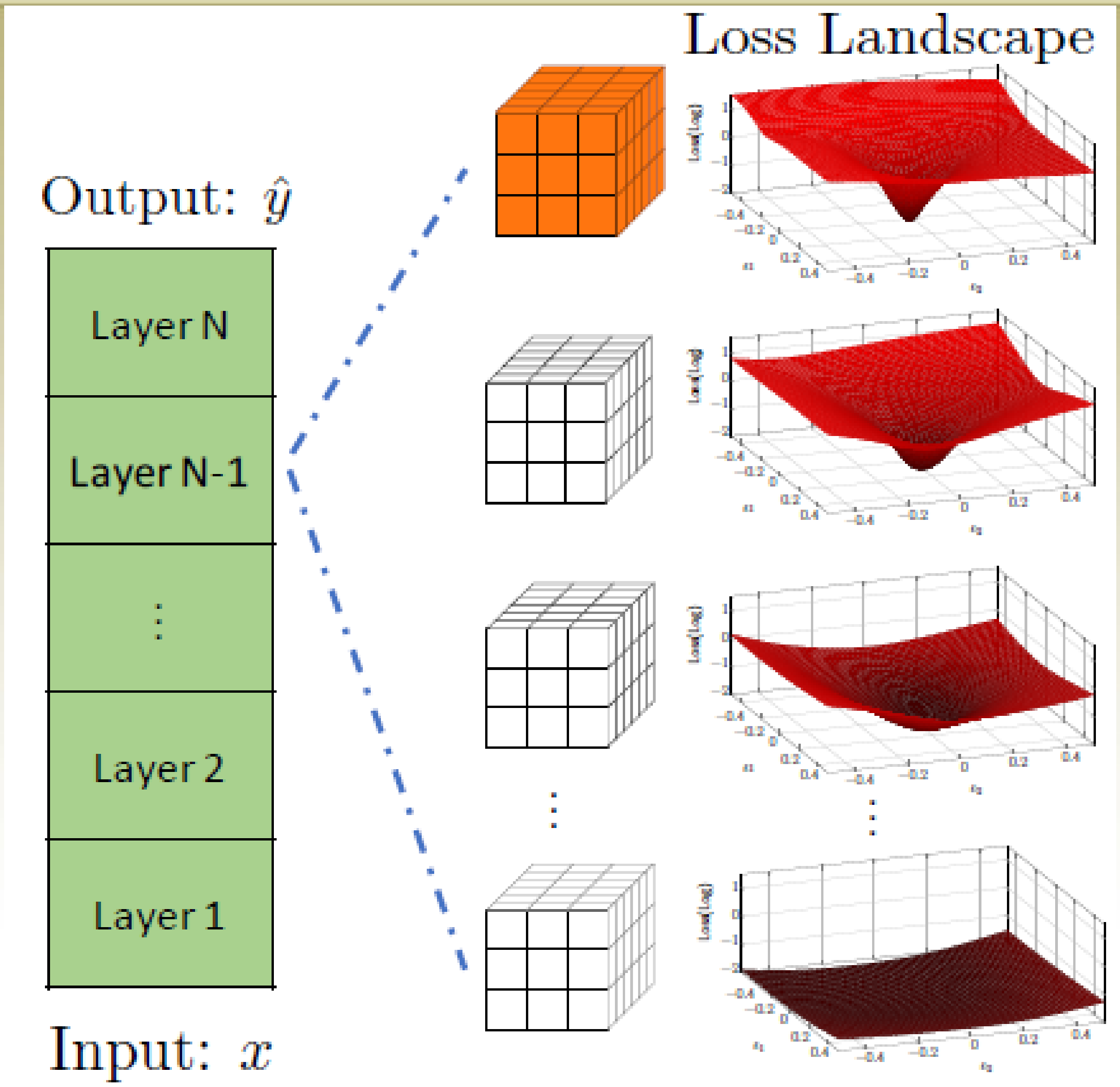
$$f(\mathbf{x}) = 0.1\,x^2 + 2y^2$$

- Gradient descent: learning rate 0.4 (left) and 0.6 (right)





Introduction



Quasi-Newton methods: BFGS

- The **Broyden–Fletcher–Goldfarb–Shanno** (BFGS) algorithm attempts to bring some of the advantages of Newton's method without the computational burden. In that respect, BFGS is similar to the conjugate gradient method
- However, BFGS takes a more direct approach to the approximation of Newton's update. Recall that Newton's update is given by:

$$\mathbf{W}^* = \mathbf{W}_0 - \mathbf{H}^{-1}\nabla_{\mathbf{W}}F(\mathbf{W}_0)$$

- The primary computational difficulty in applying Newton's update is the calculation of the inverse Hessian $\mathbf{H}^{-1}$. The approach adopted by quasi-Newton methods (of which the BFGS algorithm is the most prominent) is to approximate the inverse with a matrix $\mathbf{M}_t$ that is iteratively refined by low rank updates to become a better approximation of $\mathbf{H}^{-1}$

Quasi-Newton methods: BFGS

- Once the inverse Hessian approximation $\mathbf{M}_t$ is updated, the direction of descent $\boldsymbol{\rho}_t$ is determined by $\boldsymbol{\rho}_t = \mathbf{M}_t \mathbf{g}_t$. A line search is performed in this direction to determine the size of the step ε^* taken in this direction. The final update to the parameters is given by:
  - $\mathbf{W}(k+1) = \mathbf{W}(k) + \boldsymbol{\rho}_t \varepsilon^*$
- Like the method of conjugate gradients, the BFGS algorithm iterates a series of line searches with the direction incorporating second-order information. However unlike conjugate gradients, the success of the approach is not heavily dependent on the line search finding a point very close to the true minimum along the line
- Thus, relative to conjugate gradients, BFGS has the advantage that it can spend less time refining each line search. On the other hand, the BFGS algorithm must store the inverse Hessian matrix, **M**, that requires $O(n^2)$ memory, making BFGS impractical for most modern deep learning models that typically have millions of parameters
  - See however the Limited memory BFGS (L-BFGS)

Quasi-Newton methods: L-BFGS

- L-BFGS method has a desirable linear computational and memory complexity
- It approximates the Hessian as a series sum of first order information from prior iterations
  - These approaches do not directly use the Hessian operator
- While this approach works well for many optimization problems, it does not work well for many machine learning problems
  - One reason for this is that L-BFGS method requires full batch gradients, as stochastic gradients can lead to drastic errors in the approximation
  - This is one of the main challenges with Quasi-Newton methods applied to machine learning problems
- One of the reasons that second order methods have not been successful yet for ML, as opposed to other domains such as scientific computing, is due to the stochastic nature of the problem
- Such stochastic noise leads to an erroneous approximation of the Hessian, leading to suboptimal descent directions

Quasi-Newton methods critique

- SGD is more robust to such noise since we can efficiently incorporate moving averages and momentum
- Ideally, if there was a way to apply the same moving average method to the Hessian, then that would help smooth out local curvature noise to get a better approximation to the non-noisy curvature of the loss landscape
- However, such an approximation is challenging since the Hessian is a matrix that cannot be explicitly formed to be averaged, whereas it is easy to form the gradient vector
- But …

AdaHessian (AAAI 2021) summary

Generic update formula for the first-order methods, where η_t is the learning rate, m_t and v_t denote the first and second moment terms, and $\mathbf{g}_t$ is the gradient of a mini-batch at t-th iteration:

$$\theta_{t+1} = \theta_t - \eta_t \frac{m_t}{v_t}$$

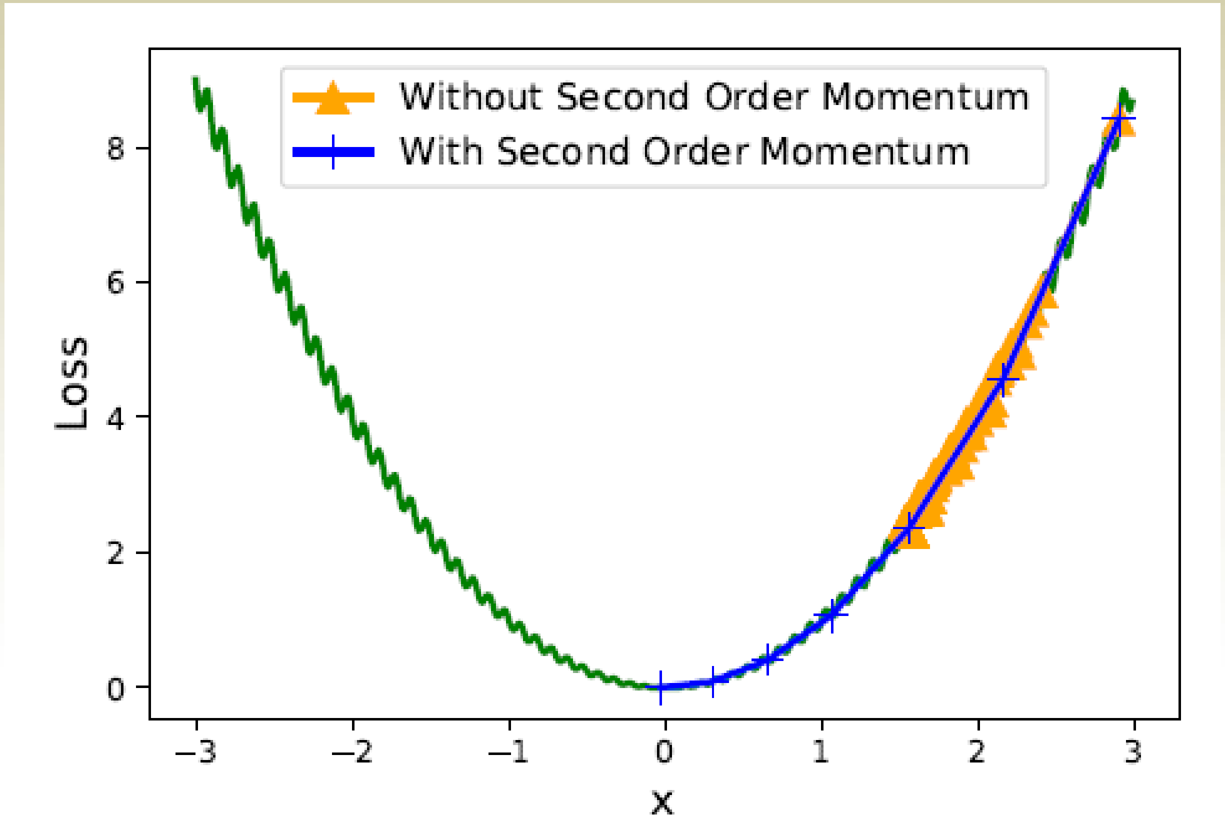
Optimizer	m_t	v_t
SGD with momentum	$\beta_1 m_{t-1} + (1-\beta_1)\mathbf{g}_t$	1
Adagrad	$\mathbf{g}_t$	$\sqrt{\sum_{i+1}^{t} \mathbf{g}_i \mathbf{g}_i}$
Adam	$\frac{(1-\beta_1)\sum_{i=1}^{t} \beta_1^{t-i} \mathbf{g}_t}{1-\beta_1^t}$	$\sqrt{\frac{(1-\beta_2)\sum_{i=1}^{t} \beta_2^{t-i} \mathbf{g}_i \mathbf{g}_i}{1-\beta_2^t}}$
RMSProp	$\mathbf{g}_t$	$\sqrt{\beta_2 v_{t-1}^2 + (1-\beta_2)\mathbf{g}_t \mathbf{g}_t}$
AdaHessian	$\frac{(1-\beta_1)\sum_{i=1}^{t} \beta_1^{t-i} \mathbf{g}_t}{1-\beta_1^t}$	$\sqrt{\frac{(1-\beta_2)\sum_{i=1}^{t} \beta_2^{t-i} \mathbf{D}_i^{(s)} \mathbf{D}_i^{(s)}}{1-\beta_2^t}}$

AdaHessian (AAAI 2021)

- Problems that exhibit this behaviour (i.e., curvature is generally different across different directions/layers) are *ill-conditioned*
- The role of the Hessian is to automatically normalize this ill-conditionedness by stretching and contracting different directions to accommodate for the curvature differences (full Newton method also rotates the gradient vector along with adjusting the step size)
- There are two major problems with this approach:
  - The first problem is that a naïve use of the Hessian preconditioner comes at the prohibitively high cost of applying Hessian inverse to the gradient vector at every iteration ($\mathbf{H}^{-k}\,\mathbf{g}$ term)
  - The second and more challenging problem is that local Hessian (curvature) information can be very misleading for a noisy loss landscape

AdaHessian (AAAI 2021)

- A simple example is illustrated below, where we plot a simple parabola with a small sinusoidal noise as the loss landscape (shown in green). As one can see, the local Hessian (curvature) information is completely misleading, as it computes the curvature of the sinusoidal noise instead of global Hessian information for the parabola



AdaHessian: Diagonal approximation

- The most simple and computationally efficient approach is to approximate the Hessian as a diagonal operator:

$$\Delta \mathrm{w} = \mathit{diag}(\mathbf{H})^{-1}\,\mathbf{g} = \mathbf{D}\,\mathbf{g}$$

- The Hessian diagonal **D** can be efficiently computed using the Hutchinson’s method
- Another important advantage, besides computational efficiency, of using the Hessian diagonal is that we can compute its moving average to resolve the local noisy Hessian as mentioned earlier
- This allows us to smooth out noisy local curvature information, and to obtain estimates that use global Hessian information instead
- We incorporate both spatial averaging and momentum (temporal averaging) to smooth out this noisy Hessian estimate as described next

AdaHessian: Spatial averaging

- The Hessian diagonal can vary significantly for each single parameter dimension of the problem
- It was found helpful to perform spatial averaging of Hessian diagonal and use the average to smooth out spatial variations
- We can perform a simple spatial averaging on the Hessian diagonal as follows:

$$\mathbf{D}^{(s)}[ib+j] = \frac{\sum_{k=1}^{b} \mathbf{D}[ib+k]}{b}, \quad \text{for } 1 \leq j \leq b \leq \frac{d}{b} - 1$$

where $\mathbf{D}^{(s)}$ is the spatially averaged Hessian diagonal, $\mathbf{D}$[i] ($\mathbf{D}^{(s)}$[i]) refers to the i-th element of $\mathbf{D}$ ($\mathbf{D}^{(s)}$), b is the spatially average block size, and d is the number of model parameters divisible by b

AdaHessian: Momentum

- We can easily apply momentum to Hessian diagonal since it is a vector instead of a quadratically large matrix
- If \bar{**D**}$_t$ is the Hessian diagonal with momentum, then:

$$\bar{\mathbf{D}}_t = \sqrt{\frac{(1-\beta_2)\sum_{i=1}^{t}\beta_2^{t-i}\mathbf{D}_i^{(s)}\mathbf{D}_i^{(s)}}{1-\beta_2^t}}$$

where 0<b2<1 is the second momentum hyperparameter

This is exactly the same as the momentum in Adam or RMSProp except that we are using the spatial averaging Hessian diagonal instead of the gradient

AdaHessian: Momentum example

- To illustrate the importance of Hessian momentum, we provide a simple example in 1D (as shown in an earlier slide) by considering:

$$f(x) = x^2 + 0.1 \sin(20 \pi x)$$

- The method without the second order momentum gets trapped at a local minima even with more than 1000 iterations (orange trajectory)
- On the contrary, the optimization converges within 7 iterations with Hessian momentum (blue trajectory)

AdaHessian algorithm

```
• Require: Initial parameter: $\theta_0$
• Require: Learning rate: $\eta$
• Require: Exponential decay rates: $\beta_1, \beta_2$
• Require: Block size: $b$
• Require: Hessian Power: $k$ (=1)
• Set: $m_0$=0, $v_0$=0
• for $t$=1,2,,... do        //Training iterations
    $\boldsymbol{g}_t$ = current step gradient
    $\boldsymbol{D}_t$ = current step estimated diagonal Hessian
    Compute $\boldsymbol{D}^{(s)}{}_t$
    Update \bar{$\boldsymbol{D}$}$_t$
    Update $m_t$, $v_t$                //See equations in the AdaHessian summary slide
    $\theta_t = \theta_{t-1} - \mathrm{n}\ m_t/v_t$
```