Optimisation

Dmitry Adamskiy

February 12, 2020

Outline

1. Intro: linear regression
2. First Order Methods. Gradient Descent
3. Conjugate Gradients

Overview

- Why optimisation?
- Gradient Descent
  - Convergence analysis.
  - What can go wrong?
  - Batch and stochastic gradient descent.
- Momentum and other add-ons
  - Nesterov, Adagrad, RMSprop, Adam...
- Conjugate Gradients
- (Optional) Second-order methods

Intro: linear regression

The need for optimisation

Machine learning often requires fitting a model to data. Often this means finding the parameters $\boldsymbol{\theta}$ of the model that 'best' fit the data.

Regression

For example, for regression based on training data $(\mathbf{x}^n, y^n)$ we might have a model $y(x|\boldsymbol{\theta})$ and wish to set $\boldsymbol{\theta}$ by minimising

$$E(\boldsymbol{\theta}) = \sum_n \left(y^n - y(\mathbf{x}^n|\boldsymbol{\theta})\right)^2$$

Complexity

In all but very simple cases, it is extremely difficult to find an algorithm that will guarantee to find the optimal $\boldsymbol{\theta}$.

A simple case: Linear regression

For example for a linear predictor

$$y(\mathbf{x}|\theta) \equiv \mathbf{x}^\mathsf{T}\boldsymbol{\theta}$$

$$E(\boldsymbol{\theta}) = \sum_n \left(y^n - \boldsymbol{\theta}^\mathsf{T}\mathbf{x}^n\right)^2$$

The optimum is given when the gradient wrt $\boldsymbol{\theta}$ is zero:

$$\frac{\partial E}{\partial \theta_i} = 2\sum_n \left(y^n - \boldsymbol{\theta}^\mathsf{T}\mathbf{x}^n\right) x_i^n = 0$$

$$\underbrace{\sum_n y^n x_i^n}_{b_i} = \sum_j \underbrace{\sum_n x_i^n x_j^n}_{X_{ij}} \theta_j$$

Hence, in matrix form, this is

$$\mathbf{b} = \mathbf{X}\boldsymbol{\theta}, \rightarrow \boldsymbol{\theta} = \mathbf{X}^{-1}\mathbf{b}$$

which is a simple linear system that can be solved in $O\left((\dim \boldsymbol{\theta})^3\right)$ time.

Quadratic functions

A class of simple functions to optimise is, for symmetric $\mathbf{A}$:

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\mathsf{T}\mathbf{A}\mathbf{x} - \mathbf{x}^\mathsf{T}\mathbf{b}$$

This has a unique minimum if and only if $\mathbf{A}$ is positive definite. In this case, at the optimum

$$\nabla f = \mathbf{A}\mathbf{x} - \mathbf{b} = \mathbf{0} \rightarrow \mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$$

For $\mathbf{x} + \boldsymbol{\delta}$, the new value of the function is

$$f(\mathbf{x} + \boldsymbol{\delta}) = f(\mathbf{x}) + \underbrace{\boldsymbol{\delta}^\mathsf{T}\nabla f}_{=0} + \frac{1}{2}\underbrace{\boldsymbol{\delta}^\mathsf{T}\mathbf{A}\boldsymbol{\delta}}_{\geq 0}$$

Hence $\mathbf{A}^{-1}\mathbf{b}$ is a minimum.

Take home messages

- Finding optimal parameters for linear regression involves minimising a quadratic function.
- Minimising a quadratic function with symmetric positive-definite matrix A is equivalent to solving a linear system.
- In the rest of the lecture we are going to pretend that the function is a quadratic either for the analysis of the algorithms (GD) or for their design (CG, second-order methods).

First Order Methods. Gradient Descent

Gradient Descent

We wish to find $\mathbf{x}$ that minimises $f(\mathbf{x})$. For general f there is no closed-form solution to this problem and we typically resort to iterative methods.

For $\mathbf{x}_{k+1} \approx \mathbf{x}_k$,

$$f(\mathbf{x}_{k+1}) \approx f(\mathbf{x}_k) + (\mathbf{x}_{k+1} - \mathbf{x}_k)^{\mathsf{T}} \nabla f(\mathbf{x}_k)$$

setting

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

gives

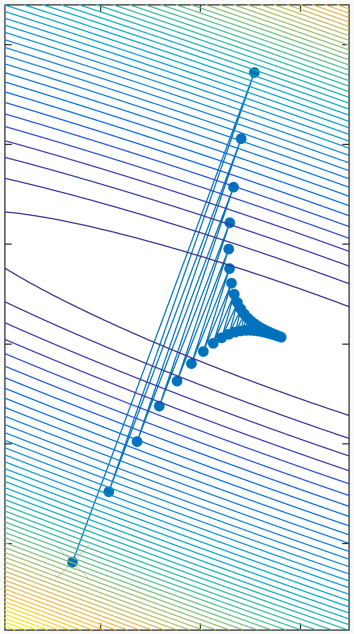
$$f(\mathbf{x}_{k+1}) \approx f(\mathbf{x}_k) - \epsilon \left|\nabla f(\mathbf{x}_k)\right|^2$$

Hence, for a small ϵ (called learning rate), the algorithm

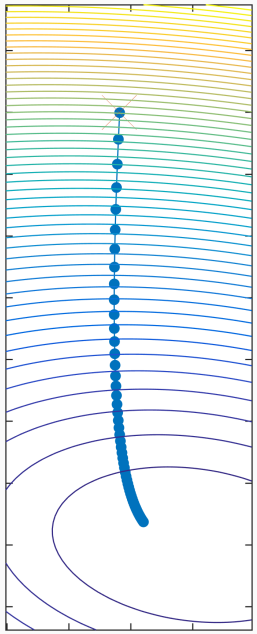
$$\mathbf{x}_{k+1} = \mathbf{x}_k - \epsilon \nabla f(\mathbf{x}_k)$$

decreases f.

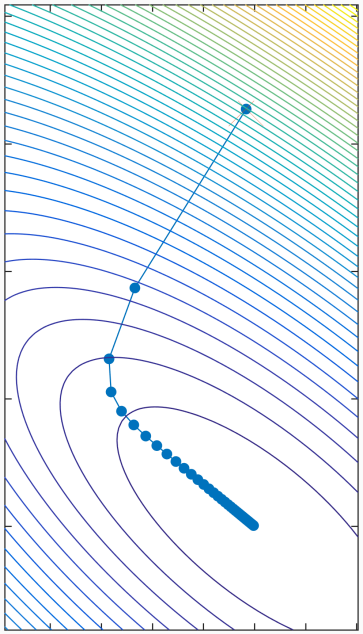
Gradient Descent – learning rate



(a) Learning rate too large



(b) Learning rate too small



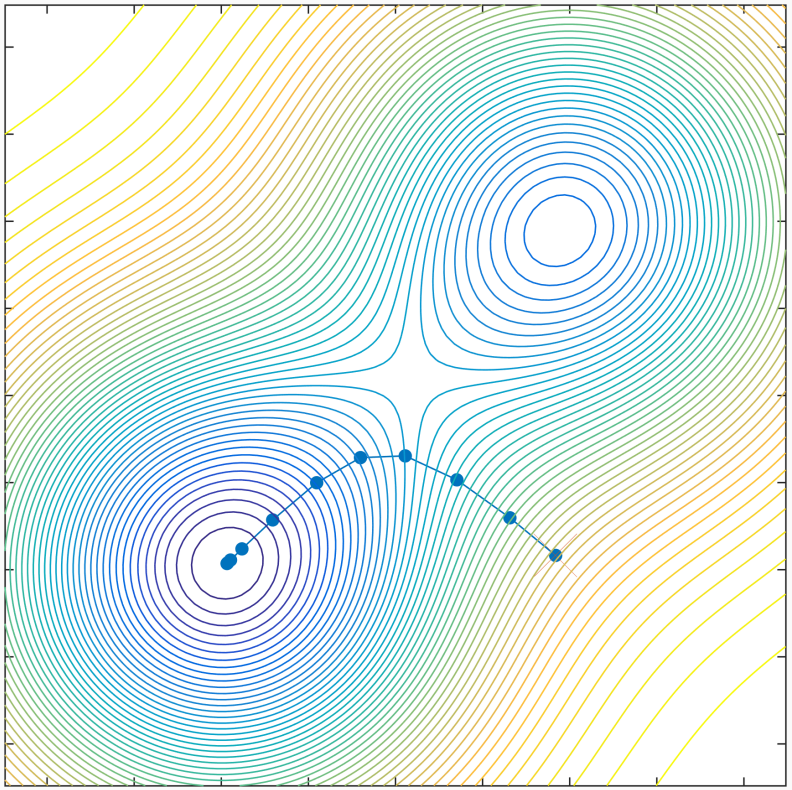
(c) Learning rate OK

Note the difference between converging to the minimal function value, and the optimum parameters; we might have almost converged to the minimal value but still be a long way from the optimum parameters. Common to consider adapting the learning rate. This is usually determined by experimenting with different values or learning schedules.

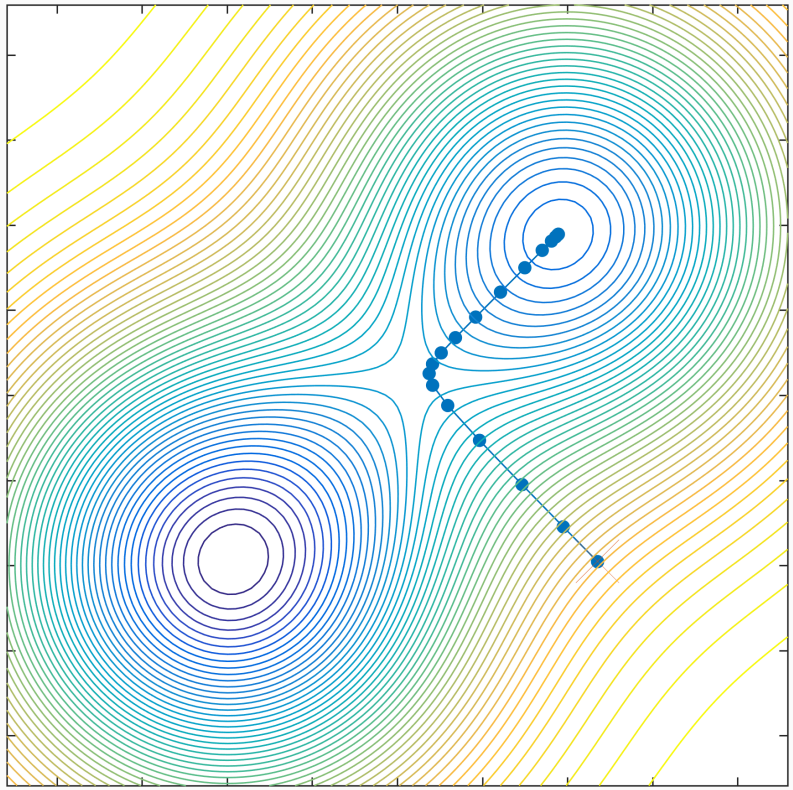
Convergence properties: optimal learning rate

- Let's examine what happens when we optimise a convex quadratic with Graident Descent... (derivation on the board, in the notebook and here).

Gradient Descent – local minima



(d) Found global minimum



(e) Found local minimum

For non-convex functions, depending on the initial point, even for the same function and small learning rate, we can converge to different solutions.

Stochastic Gradient Descent

- In machine learning, the loss function is often the sum (average) of losses for individual examples:

$$L(\boldsymbol{\theta}) = \frac{1}{N}\sum_{i=1}^{N} L_i(\boldsymbol{\theta}) = \frac{1}{N}\sum_{i=1}^{N} l(y_i, f(x_i|\boldsymbol{\theta}))$$

- If the dataset is big, one step of GD could be very expensive.
- Instead, we can use Stochastic Gradient Descent, Iterating over the training set examples:

$$\boldsymbol{\theta}_{t+1} = \theta_t - \alpha \nabla L_i(\boldsymbol{\theta})$$

- We can make progress even before we see all the data.

Mini-batch gradient descent

- The compromise: use mini-batches of training examples and compute the gradient esimtate based on those. This allows to use the benefits of vectorisation while reducing the noise.
- How to choose the batch size?
  - Large batch size: less noisy estimates, convergence might be faster (in number of iterations)
  - Smaller batch size: faster weight updates.
  - GPUs usually allow for larger batch sizes.

Fluctuations

- Stochastic gradient descent introduces fluctuations in the training process. The gradient estimate is noisy, so each update does not necessarily point toward the direction of true gradient.
- As such, it may even increase the loss.
- Reducing learning rate reduces fluctuations (but may slow down convergence).

Momentum

One simple idea to limit the zig-zag behaviour is to make an update in the average direction of the previous updates.

Moving average

Consider a set of numbers $x_1, \ldots, x_t$. Then the average a_t is given by

$$a_t = \frac{1}{t}\sum_{\tau=1}^{t} x_\tau = \frac{1}{t}\left(x_t + (t-1)a_{t-1}\right) = \epsilon_t x_t + \mu_t a_{t-1}$$

for suitably chosen ϵ_t and $0 \leq \mu_t \leq 1$. If μ_t is small then the more recent x contribute more strongly to the moving average.

Momentum

Idea is to use a form of moving average to the updates:

$$\tilde{\mathbf{g}}_{k+1} = \mu_k \tilde{\mathbf{g}}_k - \epsilon \mathbf{g}_k(\mathbf{x}_k)$$

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \tilde{\mathbf{g}}_{k+1}$$

Hence, instead of using the update $-\epsilon \mathbf{g}_k(\mathbf{x}_k)$, we use the moving average of the update $\tilde{\mathbf{g}}_{k+1}$ to form the new update.

Momentum

- Momentum can increase the speed of convergence since, for smooth objectives, as we get close to the minimum the gradient decreases and standard gradient descent starts to slow down.
- If the learning rate is too large, standard gradient descent may oscillate, but momentum may reduce oscillations by going in the average direction.
- However, the momentum parameter μ may need to be reduced with the iteration count to ensure convergence.
- Particularly useful when the gradient is noisy. By averaging over previous gradients, the noise 'averages' out and the moving average direction can be much less noisy.
- Momentum is also useful to avoid saddles (a point where the gradient is zero, but the objective function is not a minimum, such as the function x^3 at the origin) since typically the momentum will carry you over the saddle.

Nesterov's Accelerated Gradient

Nesterov

- This looks similar to momentum but has a slightly different update

$$\tilde{\mathbf{g}}_{k+1} = \mu_k \tilde{\mathbf{g}}_k - \epsilon \mathbf{g}(\mathbf{x}_k + \mu_k \tilde{\mathbf{g}}_k)$$

  That is, we use the gradient of the point we will move to, rather than the current point.

- Need to choose a schedule for μ_k. Nesterov suggests

$$\mu_k = 1 - 3/(k+5)$$

- For convex functions NAG has rate of convergence to the optimum

$$f(\mathbf{x}_k) - f^* \leq \frac{c}{k^2}$$

  for some constant c, compared to $1/k$ convergence for gradient descent.

'Understanding' Nesterov's Accelerated Gradient

Let's imagine that we've arrived at our current parameter $\mathbf{x}_k$ based on an update

$$\mathbf{x}_k = \mathbf{x}_{k-1} + \mathbf{v}_{k-1} \tag{1}$$

where $\mathbf{v}_{k-1}$ is the update. We now (retrospectively) ask: 'What would have been a better update than the one we actually made?'. Well, using the update we arrived at a function value

$$f(\mathbf{x}_k) = f(\mathbf{x}_{k-1} + \mathbf{v}_{k-1})$$

We could have got to a better value by changing $\mathbf{v}_{k-1}$ to

$$\mathbf{v}_k = \mathbf{v}_{k-1} - \epsilon \frac{\partial}{\partial \mathbf{v}} f(\mathbf{x}_k) = \mathbf{v}_{k-1} - \epsilon \frac{\partial}{\partial \mathbf{x}} f(\mathbf{x}_{k-1} + \mathbf{v}_{k-1})$$

Hence, we define

$$\mathbf{v}_k = \mathbf{v}_{k-1} - \epsilon \mathbf{g}(\mathbf{x}_{k-1} + \mathbf{v}_{k-1})$$

and make the update (note the difference with (1))

$$\mathbf{x}_k = \mathbf{x}_{k-1} + \mathbf{v}_k$$

'Understanding' Nesterov's Accelerated Gradient

- The basic update

$$\mathbf{v}_k = \mathbf{v}_{k-1} - \epsilon \frac{\partial}{\partial \mathbf{x}} f(\mathbf{x}_{k-1} + \mathbf{v}_{k-1})$$

  essentially is a form of momentum that pushes $\mathbf{x}$ along the direction it was previously going.
- Even when we reach a point where the gradient is zero, then $\mathbf{v}_k$ will be the same as $\mathbf{v}_{k-1}$.
- We therefore introduce a term to dampen oscillations and ensure convergence. One can show that this gives the standard Nesterov update:

$$\mathbf{v}_k = \mu_{k-1}\mathbf{v}_{k-1} - \epsilon \frac{\partial}{\partial \mathbf{x}} f(\mathbf{x}_{k-1} + \mu_{k-1}\mathbf{v}_{k-1})$$

$$\mathbf{x}_k = \mathbf{x}_{k-1} + \mathbf{v}_k$$

  Since $\mu < 1$, $\mathbf{v}$ will quickly converge to zero around the minimum.

Adagrad

Suppose that your data is sparse: some features are rare (and thus relevant weights rarely get updates). Using the same learning rate for all of them seems detrimental. Adagrad fixes that:

$$g_{t,i} = (\nabla L(\boldsymbol{\theta}^t))_i$$

$$\theta_i^{t+1} = \theta_i^t - \frac{\eta}{\sqrt{G_{t,ii} + \varepsilon}} g_{t,i}$$

Here, $G_{t,ii} = \sum_{\tau=1}^{t} g_{\tau,i}^2$

$+$

Adaptive learning rates. Most implementations are happy with default $\eta = 0.01$

Adagrad

Suppose that your data is sparse: some features are rare (and thus relevant weights rarely get updates). Using the same learning rate for all of them seems detrimental. Adagrad fixes that:

$$g_{t,i} = (\nabla L(\boldsymbol{\theta}^t))_i$$

$$\theta_i^{t+1} = \theta_i^t - \frac{\eta}{\sqrt{G_{t,ii} + \varepsilon}} g_{t,i}$$

Here, $G_{t,ii} = \sum_{\tau=1}^{t} g_{\tau,i}^2$

$+$

Adaptive learning rates. Most implementations are happy with default $\eta = 0.01$

$-$

The accumulated gradients eventually make learning very slow and the learning rate never recovers.

RMSprop

RMSprop fixes the problem with Adagrad vanishing learning rates by using moving average of past squared gradients instead (like momentum is doing for the gradients themselves):

$$s_{t,i} = (1 - \gamma) s_{t-1,i} + \gamma g_{t,i}^2$$
$$\theta_i^{t+1} = \theta_i^t - \frac{\eta}{\sqrt{s_{t,i} + \varepsilon}} g_{t,i}$$

The algorithm was first presented in Geoff Hinton Coursera course (Lecture 6e).

Adam

Adam is a very popular optimiser, basically combining RMSprop ideas with momentum.

$$m_{t,i} = (1-\beta)m_{t-1,i} + \beta g_{t,i}$$
$$s_{t,i} = (1-\gamma)s_{t-1,i} + \gamma g_{t,i}^2$$
$$\theta_i^{t+1} = \theta_i^t - \frac{\eta}{\sqrt{s_{t,i}+\varepsilon}} m_{t,i}$$

All of those algorithms are implmented in the popular machine learning libraries. A bunch of tweaks appeared since (AdamW, NAdam, Amsgrad, ...).

Conjugate Gradients

Line Search

One way to potentially improve on gradient descent is choose a particular direction $\mathbf{p}_k$ and search along there. We then find the minimum of the one dimensional problem

$$F(\alpha) = f(\mathbf{x}_k + \alpha \mathbf{p}_k)$$

Finding the optimal α^* can be achieved using a standard one-dimensional optimisation method. However, for a quadratic one can do it analytically setting $F'(\alpha) = \nabla F^T \mathbf{p}_k = 0$:

$$A(x_k + \alpha p_k)^T p_k = b^T p_k$$

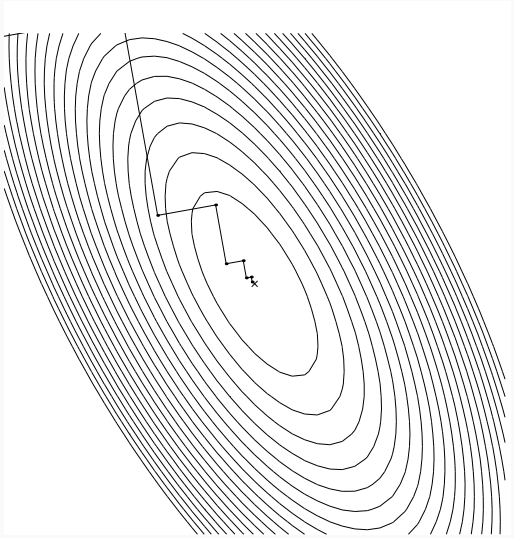
$$\alpha = \frac{(b^T - Ax_k)^T p_k}{p_k^T A p_k} = \frac{-\nabla F(x_k)^T p_k}{p_k^T A p_k}$$

Choosing the search directions

It would seem reasonable to choose a search direction that points 'maximally downhill' from the current point $\mathbf{x}_k$. That is, to set

$$\mathbf{p}_k = -\nabla f(\mathbf{x}_k)$$

This is sometimes known as Steepest Descent. However, at least for quadratic functions, this is not optimal and leads to potentially zig-zag behaviour:



This zig-zag behaviour can occur for non isotropic surfaces.

Orthogonal directions

What if we try to search in the orthogonal directions $d_0, d_1, \ldots, d_{n-1}$, once in each direction? If we define an error $e_t = x_t - x^*$ (where x^* is the minimum for which $Ax^* = b$), this means that we would like $e_{t+1} \perp d_t$. Does that help us find a step size α_t?

$$
\begin{aligned}
e_{t+1}^T d_t &= 0 \\
(e_t + \alpha_t d_t)^T d_t &= 0 \\
\alpha_t &= -\frac{e_t^T d_t}{d_t^T d_t}
\end{aligned}
$$

Orthogonal directions

What if we try to search in the orthogonal directions $d_0, d_1, \ldots, d_{n-1}$, once in each direction? If we define an error $e_t = x_t - x^*$ (where x^* is the minimum for which $Ax^* = b$), this means that we would like $e_{t+1} \perp d_t$. Does that help us find a step size α_t?

$$\begin{aligned} e_{t+1}^T d_t &= 0 \\ (e_t + \alpha_t d_t)^T d_t &= 0 \\ \alpha_t &= -\frac{e_t^T d_t}{d_t^T d_t} \end{aligned}$$

The problem is that we don't know e_t (and if we knew it the problem would be solved). However, note that
$Ae_t = Ax_t - Ax^* = Ax_t - b = \nabla F(x_t)$ and this is something we know.

Conjugate (A-orthogonal) directions

Instead of using a set of orhogonal directions, we are going to use the set of A-orthogonal ones, $d_0, d_1, \ldots, d_{n-1}$.

Definition
d_i is A-orthogonal to d_j if $d_i^T A d_j = 0$

Conjugate (A-orthogonal) directions

Instead of using a set of orhogonal directions, we are going to use the set of A-orthogonal ones, $d_0, d_1, \ldots, d_{n-1}$.

Definition
d_i is A-orthogonal to d_j if $d_i^T A d_j = 0$

Now we want $e_{t+1}^T A d_t = 0$. What is the condition for this?

$$d_t^T A e_{t+1} = 0$$
$$d_t^T A(e_t + \alpha_t d_t) = 0$$
$$\alpha_t = -\frac{d_t^T \nabla F(x_t)}{d_t^T A d_t}$$

Unlike the previous attempt, this is something that we can compute!

Conjugate (A-orthogonal) directions

Instead of using a set of orhogonal directions, we are going to use the set of A-orthogonal ones, $d_0, d_1, \ldots, d_{n-1}$.

Definition
d_i is A-orthogonal to d_j if $d_i^T A d_j = 0$

Now we want $e_{t+1}^T A d_t = 0$. What is the condition for this?

$$d_t^T A e_{t+1} = 0$$
$$d_t^T A(e_t + \alpha_t d_t) = 0$$
$$\alpha_t = -\frac{d_t^T \nabla F(x_t)}{d_t^T A d_t}$$

Unlike the previous attempt, this is something that we can compute!
Also, this is the update that minimises F!

Convergence

Why would this procedure converge to the minimum? Suppose we have a set of n linearly independent A-orthogonal directions. Let's express the initial error e_0 in this basis, $e_0 = \sum_i \delta_i d_i$. What are the coefficients δ_i?

$$d_t^T A e_0 = d_t^T A \sum_i \delta_i d_i$$

$$d_t^T A e_0 = \delta_t d_t^T A d_t$$

$$\delta_t = \frac{d_t^T \nabla F(x_t)}{d_t^T A d_t} = -\alpha_t$$

That is, every step cuts down one of the components of the initial error in the basis of the search directions. After n steps all the components are zero and we are done.

How to get conjugate directions?

- One way to get the set of conjugate directions is called Gram-Schmidt conjugation and goes like this.
- Start with linear independent $u_0, u_1, \ldots, u_{n-1}$. Set $d_0 = u_0$ and for each d_i, subtract from it all the components that a not A-orthogonal to the previous d_j, $j < i$:

$$d_i = u_i + \sum_{j<i} \beta_{ij} d_j$$

- The coefficients β_{ij} are found from the A-orthogonality condition:

$$d_j^T A d_i = 0, j < i$$
$$0 = d_j^T A u_i + \beta_{ij} d_j^T A d_j$$
$$\beta_{ij} = -\frac{d_j^T A u_i}{d_j^T A d_j}$$

How to get conjugate directions?

- One way to get the set of conjugate directions is called Gram-Schmidt conjugation and goes like this.
- Start with linear independent $u_0, u_1, \ldots, u_{n-1}$. Set $d_0 = u_0$ and for each d_i, subtract from it all the components that a not A-orthogonal to the previous d_j, $j < i$:

$$d_i = u_i + \sum_{j<i} \beta_{ij} d_j$$

- The coefficients β_{ij} are found from the A-orthogonality condition:

$$d_j^T A d_i = 0, j < i$$
$$0 = d_j^T A u_i + \beta_{ij} d_j^T A d_j$$
$$\beta_{ij} = -\frac{d_j^T A u_i}{d_j^T A d_j}$$

- Problem: this is $O(n^3)$ just like the original problem. In fact if u_i are aligned with the coordinate axis, this becomes Gaussian elimination.

Properties of conjugate directions algorithm

- The gradient at each point is orthogonal to all the past search directions (and to the original vectors u_i). Also

$$d_t^T \nabla F(x_t) = u_t^T \nabla F(x_t)$$

- The point that is found after t iterations is minimising the target function among all the possible linear combinations of search vectors:

$$f(x_t) = \min_{\alpha} F(x_0 + \sum_{i=0}^{t-1} \alpha_i d_i)$$

Congugate Gradients

Conjugate gradients method has a particular choice of u_t, namely $u_t = r_t = -\nabla F(x_t)$ (r stands for residual, $r_t = b - Ax_t$). Why is that useful? Let's have a look at β_{ij}. It turns out (derivation on the board) that the only non-zero one is $\beta i, i-1$:

$$\beta_{(i)} := \beta_{i,i-1} = \frac{1}{\alpha_{i-1}} \frac{\nabla F(x_i)^T \nabla F(x_i)}{\nabla F(x_{i-1})^T A \nabla F(x_{i-1})}$$

which further simplifies to

$$\beta_{(i)} = \frac{\nabla F(x_i)^T \nabla F(x_i)}{\nabla F(x_{i-1})^T \nabla F(x_{i-1})}$$

This means that at each iteration the most expensive operation is one matrix-vector product, with the complexity $O(m)$ where m is the number of non-zero elements in the matrix A. Hence the benefit for the sparse matrices.

Conjugate Gradients: Algorithm

1: $k = 0$
2: Choose x_0.
3: $d_0 = -g_0 = -\nabla F(x_0)$
4: **while** $g_k \neq \mathbf{0}$ **do**
5: $\quad \alpha_k = \underset{\alpha_k}{\operatorname{argmin}}\ f(x_k + \alpha_k d_k) = \frac{g_k^T g_k}{d_k^T A d_k}$
6: $\quad x_{k+1} = x_k + \alpha_k d_k$
7: $\quad g_{k+1} = g_k - \alpha_k A d_k$
8: $\quad \beta_{k+1} = g_{k+1}^T g_{k+1} / g_k^T g_k$
9: $\quad d_{k+1} = -g_{k+1} + \beta_{k+1} d_k$
10: $\quad k = k + 1$
11: **end while**

Non-linear CG

- The algorithm could be applied to non-quadratic functions with a few tweaks.
- Generic line search procedure is needed
- 'Conjugate' now means conjugate with respect to the Hessian. The notion of conjugacy changes from one point to another (with the Hessian). The closer the function is to the quadratic the better.
- Restarts might be needed.
- Several formulas for the correction coefficients β are known. All are equivalent for quardatics, but become distinct in general case.
- More details in [1].

References

[1] J. R. Shewchuk. An introduction to the conjugate gradient method without the agonizing pain. Technical report, USA, 1994.