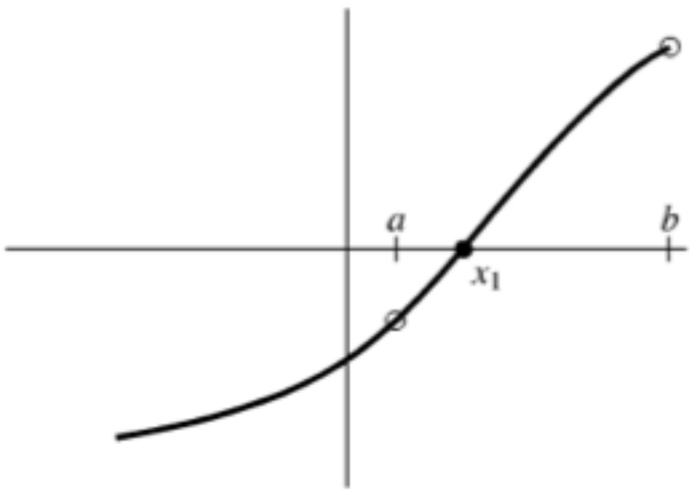
LECTURE 6: Nonlinear Equations and Optimization

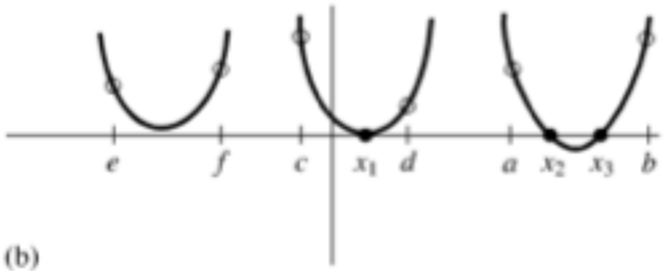
- we cover first solving nonlinear equations and 1-d optimization
- $f(x) = 0$ (either in 1-d or many dimensions)
- In 1-d we can bracket the root and then find it, in high dimensions we cannot

- Bracketing in 1-d: if $f(x) < 0$ at a and $f(x) > 0$ at $b > a$ (or the other way around) and f(x) is continuous then there is a root at $a < x < b$



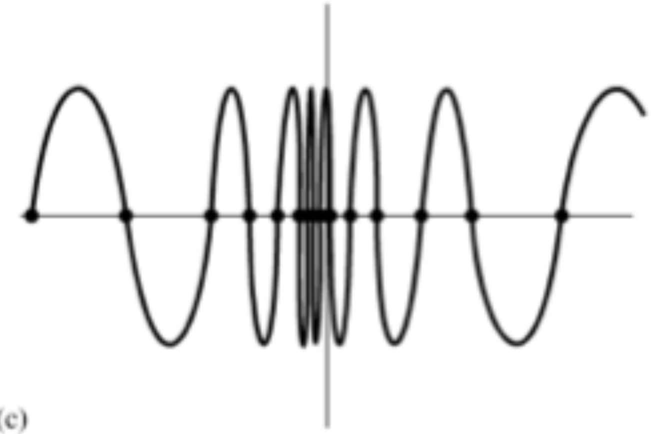
Other Situations:

- No roots or one or two roots but no sign change:



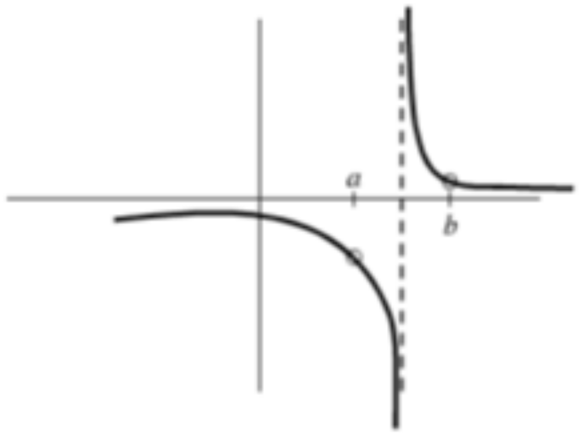
(b)

- Many roots:



(c)

- Singularity:



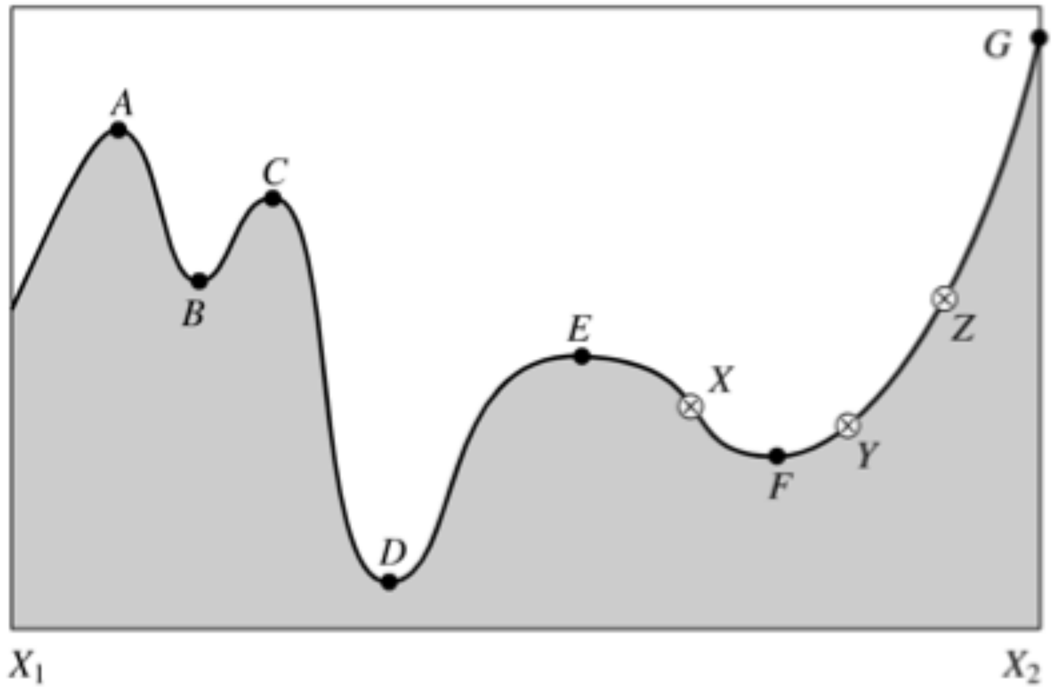
(d)

Bisection for Bracketing

- We can use bisection: divide interval by 2, evaluate at the new position, and choose left or right half-interval depending on where the function has opposite sign. Number of steps is $log_2[(b-a)/\varepsilon]$, where e is the error tolerance. The method must succeed.
- Error at next step is $\varepsilon_{n+1} = \varepsilon_n/2$, so converges linearly
- Higher order methods scale as $\varepsilon_{n+1} = c\varepsilon_n^m$, with $m > 1$

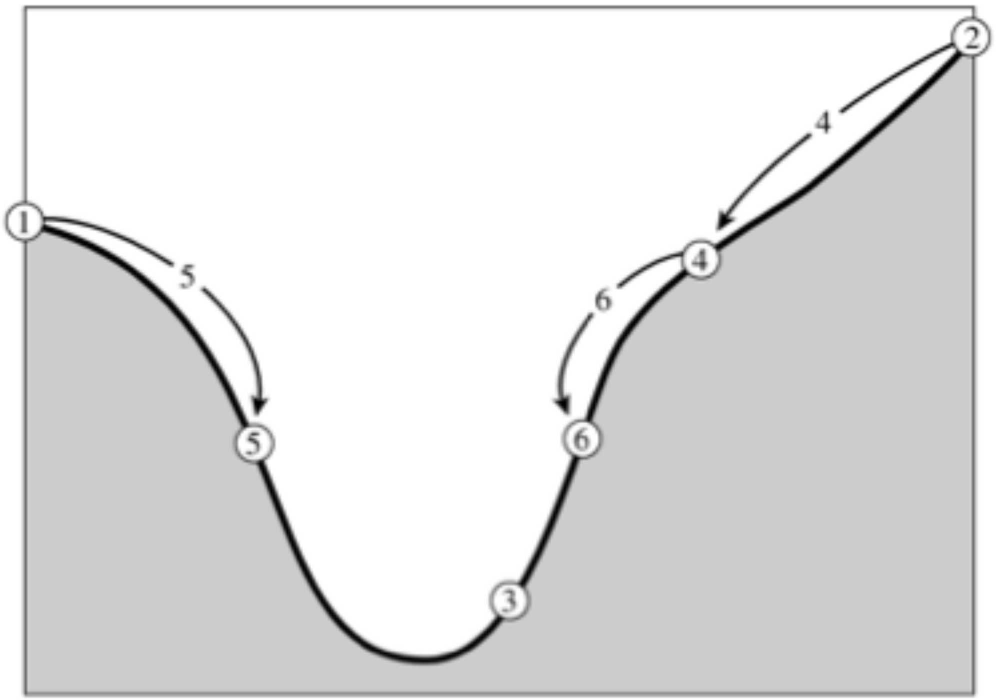
1-d Optimization: Local and Global Extrema, Bracketing

- Optimization: minimization or maximization
- In most cases only local minimum (B,D,F) or local maximum (A,C,E,G) can be found, difficult to prove they are global minimum (D) or global maximum (G)
- We bracket a local minimum if we find $f(X) > f(Y)$ and $f(Z) > f(Y)$ for $X < Y < Z$.



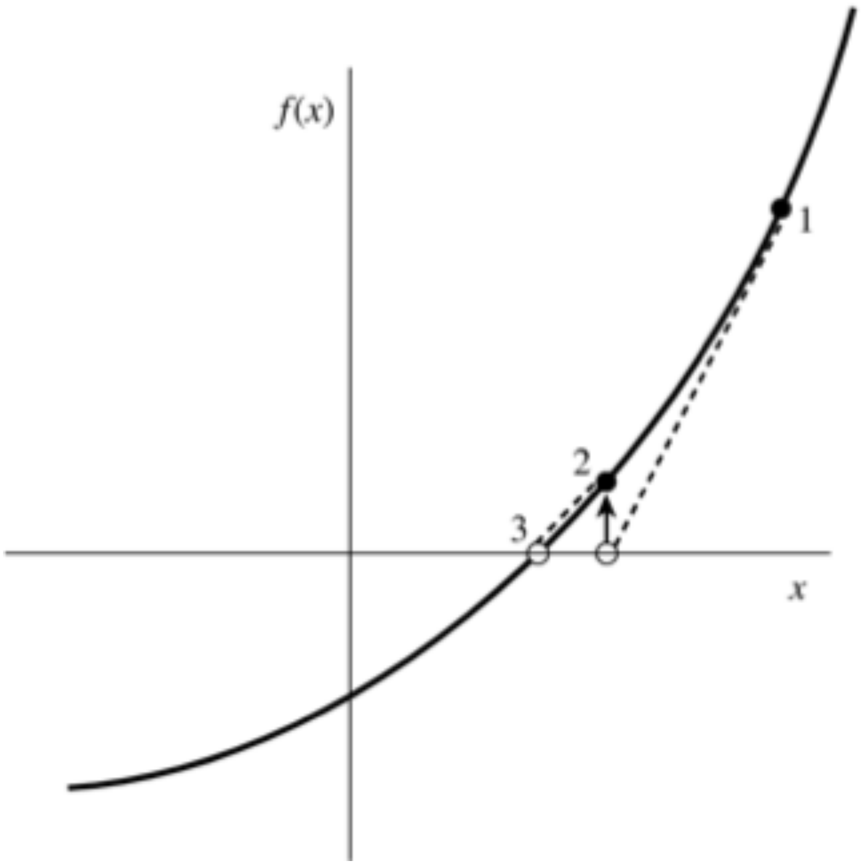
Golden Ratio Search

- Remember that we need a triplet of points to bracket $a < b < c$ such that $f(b)$ is less than $f(a)$ and $f(c)$
- Suppose w = $(b-a)/(c-a)$. We evaluate at x, define $(x-b)/(c-a) = z$. The next bracketing segment will be either w+z or 1-w.
  To minimize the error choose these two to be equal: $z = 1 - 2w$.
  But w was also chosen this way, so $z/(1-w) = w$,
  $1-2w=w(1-w): w^2-3w+1=0$
  and $w = (3-5^{1/2})/2 = 0.382$,
  $1 - w = 0.618$,
  Golden Ratio
  $(1/0.618=1.618=(1+5^{1/2})/2)$.

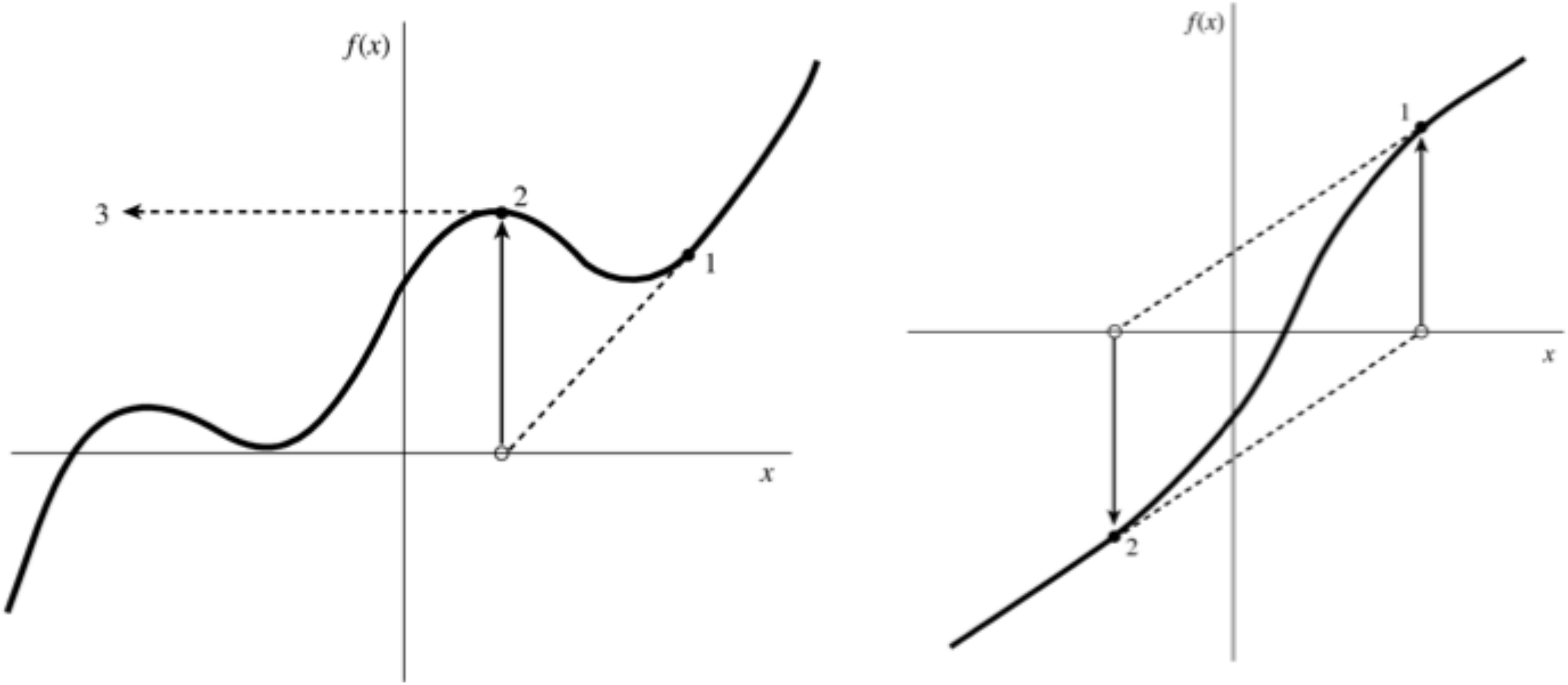


Newton(-Raphson) Method

- Most celebrated of all methods, we will use it extensively in higher dimensions
- Requires a gradient:
  $f(x+\delta) = f(x) + \delta f'(x) + \ldots$
- We want $f(x+\delta) = 0$, hence
  $\delta = -f(x)/f'(x)$
- Rate of convergence is quadratic (NR 9.4) m=2
  $\varepsilon_{i+1} = \varepsilon_i^2 f''(x)/(2f'(x))$



Newton-Raphson is not Failure-free

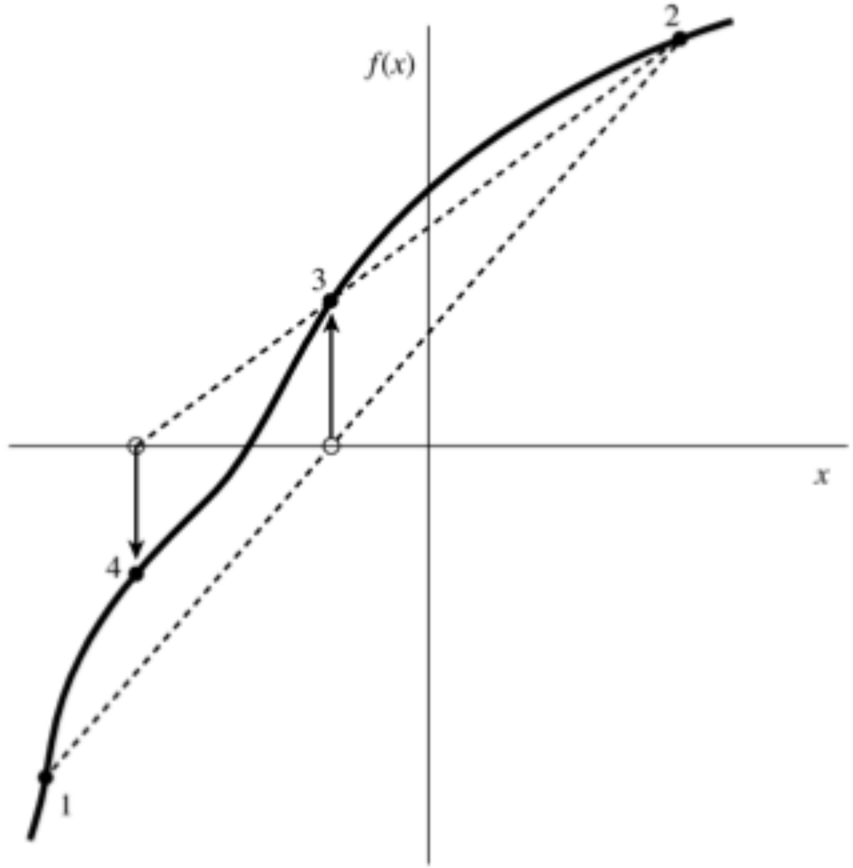


Newton-Raphson for 1-d Optimization

- Expand function to 2nd order (note: we did this already when expanding log likelihood)
- $f(x+\delta) = f(x) + \delta f'(x) + \delta^2 f''(x)/2 + \ldots$
- Expand its derivative $f'(x+\delta) = f'(x) + \delta f''(x) + \ldots$
- Extremum requires $f'(x+\delta) = 0$ hence $\delta = -f'(x)/f''(x)$
- This requires f'': Newton's optimization method

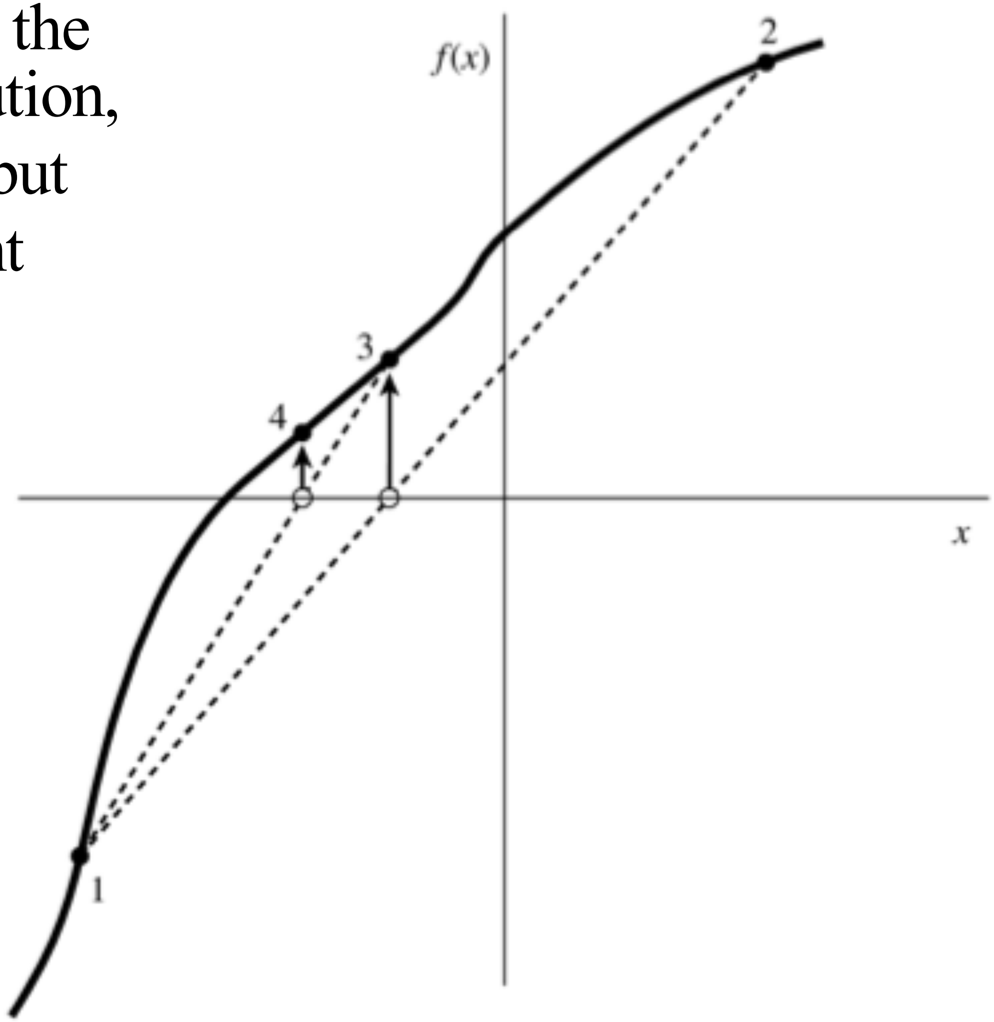
Secant Method for Nonlinear Equations

- Newton's method using numerical evaluation of a gradient defined across the entire interval:
  $f'(x_2) = [f(x_2)-f(x_1)]/(x_2-x_1)$
- $x_3 = x_2 - f(x_2)/f'(x_2)$
- Can fail, since does not always bracket
- $m = 1.618$ (golden ratio), a lot faster than bisection

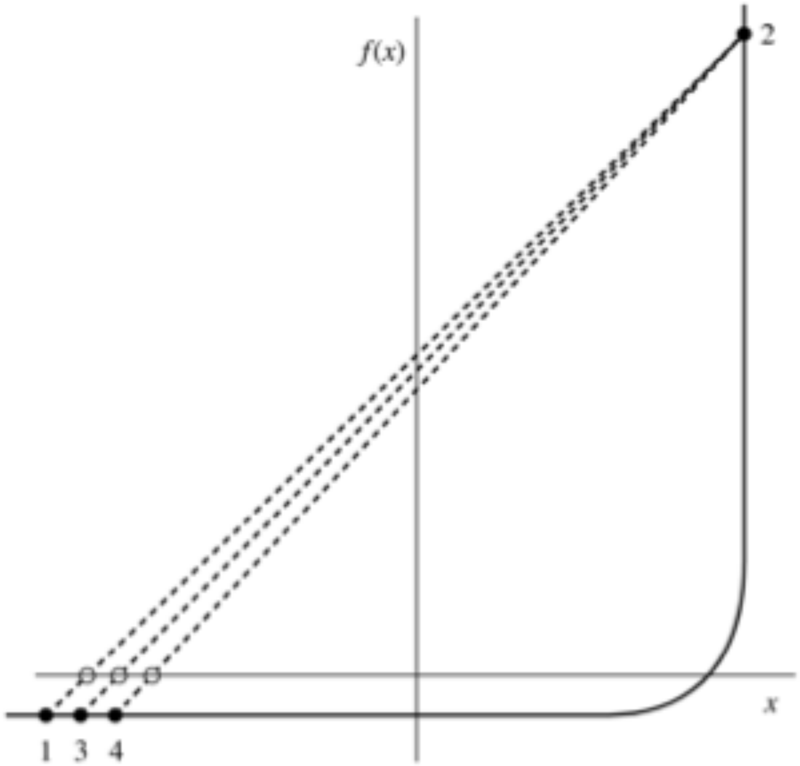


False Position Method for Nonlinear Equations

- Similar to secant, but keep the points that bracket the solution, so guaranteed to succeed, but with more steps than secant



Sometimes convergence can be slow

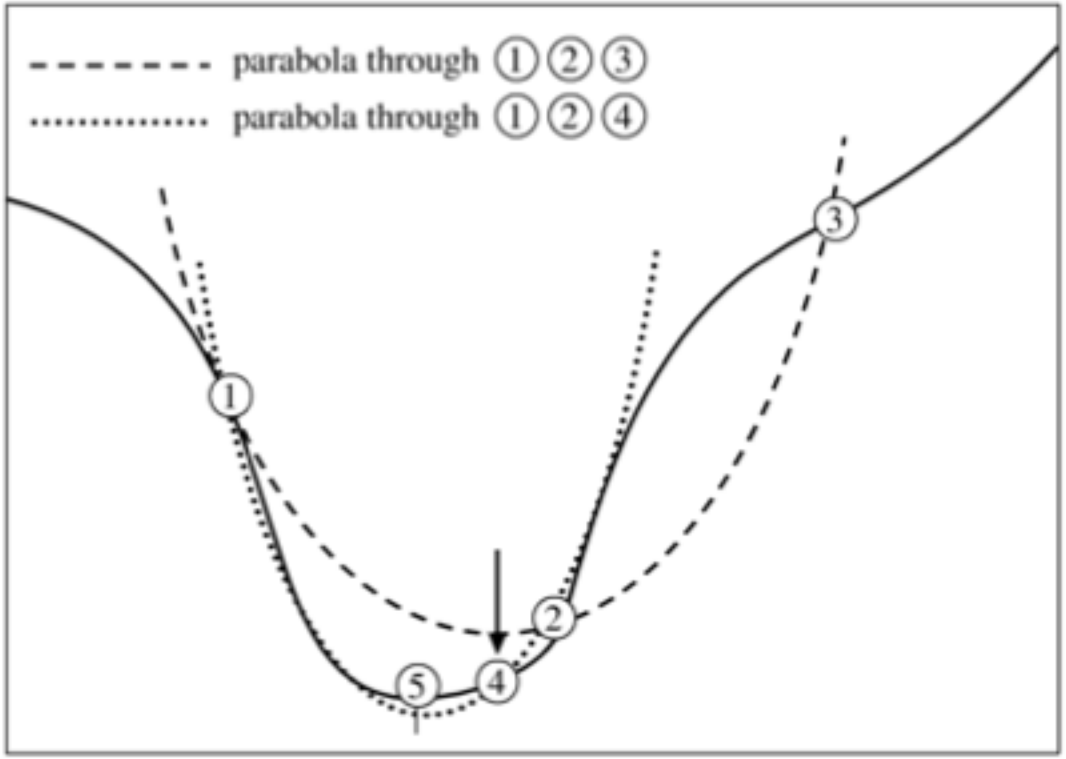


Better methods without derivatives such as Ridders or Brent's method combine these basic techniques: use these as default option and (optionally) switch to Newton once the solution is guaranteed for a higher convergence rate

Parabolic Method for 1-d Optimization

- Approximate the function of *a, b, c* as a parabola

$$x = b - \frac{1}{2}\frac{(b-a)^2[f(b)-f(c)]-(b-c)^2[f(b)-f(a)]}{(b-a)[f(b)-f(c)]-(b-c)[f(b)-f(a)]}$$

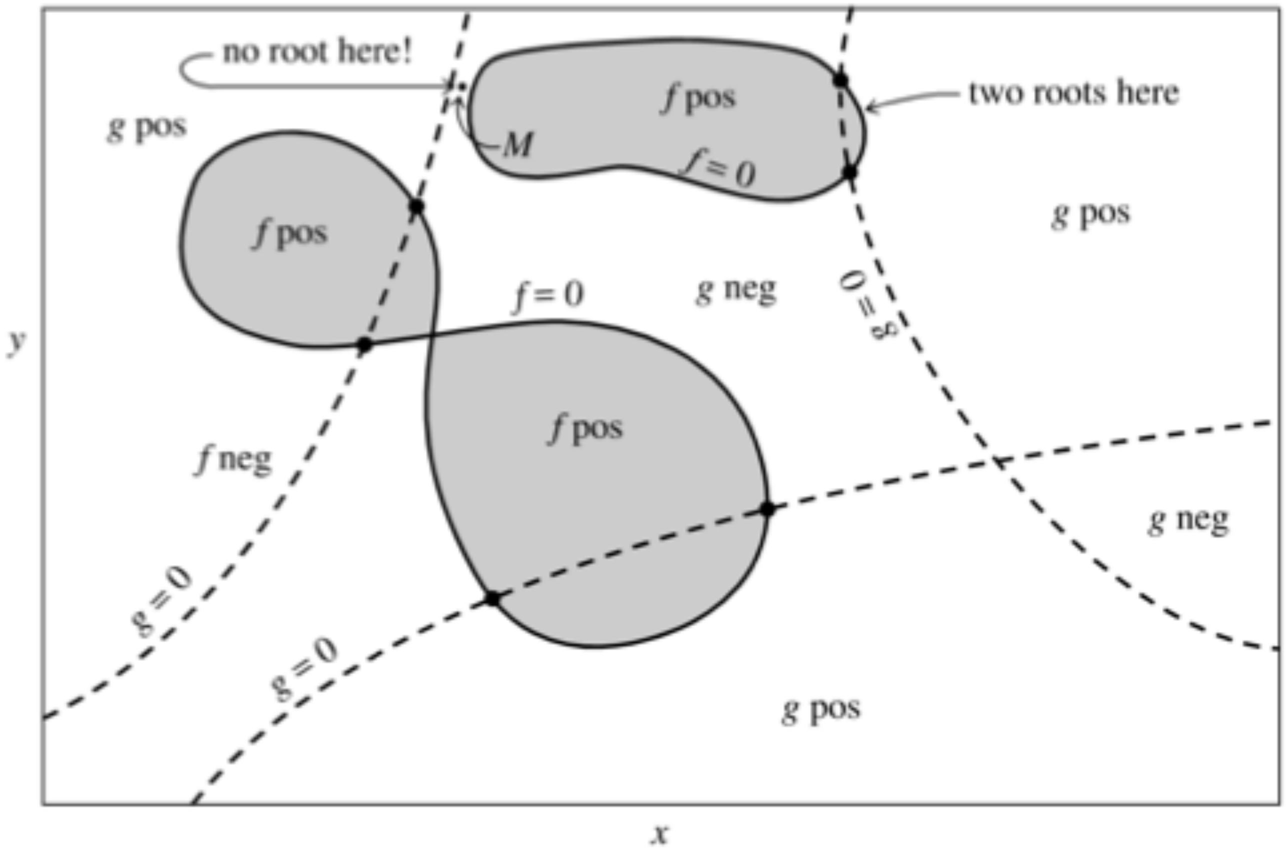


Gradient Descent in 1-d

- Suppose we do not have f'', but we have f': so we know the direction of function descent. We can take a small step in that direction: $\delta = -\eta f'(x)$. We must choose the sign of η to descend (if minimum is what we want) and it must be small enough not to overshoot.
- We can make a secant version of this method by evaluating gradient with finite difference: $f'(x_2) = [f(x_2)-f(x_1)]/(x_2-x_1)$

Nonlinear Equations in Many Dimensions

- $f(x,y) = 0$ and $g(x,y) = 0$: but the two functions f and g are unrelated, so it is difficult to look for general methods that will find all solutions



Newton-Raphson in Higher Dimensions

- Assume *N* functions

$$F_i(x_0, x_1, \dots, x_{N-1}) = 0 \qquad i = 0, 1, \dots, N-1.$$

- Taylor expand $F_i(\mathbf{x} + \delta\mathbf{x}) = F_i(\mathbf{x}) + \sum_{j=0}^{N-1} \frac{\partial F_i}{\partial x_j}\delta x_j + O(\delta\mathbf{x}^2).$
- Define Jacobian $J_{ij} \equiv \frac{\partial F_i}{\partial x_j}$
- In matrix notation $\mathbf{F}(\mathbf{x} + \delta\mathbf{x}) = \mathbf{F}(\mathbf{x}) + \mathbf{J} \cdot \delta\mathbf{x} + O(\delta\mathbf{x}^2).$
- Setting $\mathbf{F}(\mathbf{x} + \delta\mathbf{x}) = 0$, we find $\mathbf{J} \cdot \delta\mathbf{x} = -\mathbf{F}.$
- This is a matrix equations: solve with LU
- Update $\mathbf{x}_{\text{new}} = \mathbf{x}_{\text{old}} + \delta\mathbf{x}$ and iterate again

Globally Convergent Methods and Secant Methods

- If quadratic approximation in N-R method is not accurate taking a full step may make the solution worse. Instead one can do a line search backtracking and combine it with a descent direction (or use a trust region).
- When derivatives are not available we can approximate them: multi-dimensional secant method (Broyden's method).
- Both of these methods have clear analogies in optimization and since the latter is more important for data science we will explain the concepts in optimization.

Relaxation Methods

- Another class of methods solving $x = f(x)$
- Take $x = 2\text{-}e^{-x}$, start at $x_0 = 1$ and evaluate $f(x_0) = 2\text{-}e^{-1} = 1.63 = x_1$
- Now use this solution again: $f(x_1) = 2\text{-}e^{-1.63} = 1.80 = x_2$
- Correct solution is $x = 1.84140\ldots$
- If there are multiple solutions which one one converges to depends on the starting point
- Convergence is not guaranteed: suppose x^0 is exact solution: $x_{n+1} = f(x_n) = f(x^0) + (x_n\text{-}x^0)f'(x^0) + \ldots$ since $x^0 = f(x^0)$ we get $x_{n+1}\text{-}x^0 = f'(x^0)(x_n\text{-}x^0)$ so this converges if $|f'(x^0)| < 1$
- When this is not satisfied we can try to invert the equation to get $u = f^{-1}(u)$ so that $|f'^{-1}(u)| < 1$

Relaxation Methods in Many Dimensions

- Same idea: write equations as $x = f(x,y)$ and $y = g(x,y)$, use some good starting point and see if you converge
- Easily generalized to N variables and equations
- Simple, and (sometimes) works!
- Again impossible to find all the solutions unless we know something about their structure

Over-relaxation

- We can accelerate the convergence:
- $\Delta x_n = x_{n+1} - x_n = f(x_n) - x_n$
- $x_{n+1} = x_n + (1+\omega)\Delta x_n$
- if $\omega = 0$ this is relaxation method
- If $\omega > 0$ this is over-relaxation method
- No general theory for how to select ω : trial and error

Optimization in many dimensions

- Optimization (maximization/minimization) is of huge importance in data analysis and is the basis for recent breakthroughs in machine learning and big data
- A lot of it is application dependent and there is a vast number of methods developed: we cannot cover them all in this lecture
- Broadly can be divided into 1st order (derivatives are available, but not Hessian) and 2nd order (approximate Hessian or full Hessian evaluation)
- 0th order: no gradients available: use finite difference to get the gradient. Works fine in low dimensions
- or use downhill simplex (Nelder & Mead method). Very slow (curse of dimensionality) and we will not discuss it here.
- Warning: we can only get to the local minimum/maximum. Finding the global minimum/maximum is generally impossible (curse of dimensionality).
- Convex optimization: only one minimum (global). Non-convex: everything else

Preparation of Parameters

- Often the parameters are not unconstrained: they may be positive (or negative), or bounded to an interval
- Optimization with constraints is harder: without constraints we can just look for where the gradient is zero
- First step is to make optimization unconstrained: map the parameter to a new parameter that is unbounded. For example, if a variable is positive, $x > 0$, use $z = log(x)$ instead of x.
- One can also change the prior so that it reflects the original prior: $p_{pr}(z)dz = p_{pr}(x)dx$
- If $x > 0$ has uniform prior in x then $p_{pr}(z) = dx/dz = x = e^z$

Automatic Differentiation

- All good optimization methods use gradients
- How do we take a gradient of a complicated function? We divide into a sequence of elementary individual steps where the gradient is simple, then multiply these steps together using the chain rule

$$\nabla_x h(y(x)) = \sum_{i=1}^{m} \frac{\partial h}{\partial y_i} \nabla y_i(x)$$

- Neural networks are a prime example of power of auto-diffs.
- Many packages developed for doing this: tensorflow, theano (no longer developed), keras, (py)torch…
- Alternative is finite differencing. This becomes extremely expensive in high dimensions. Modern NN easily have 10^6 and more dimensions
- Note: NN rarely uses 2nd order optimization methods due to high dimensionality of the problem and due to high data volume, which requires use of stochastic gradient descent

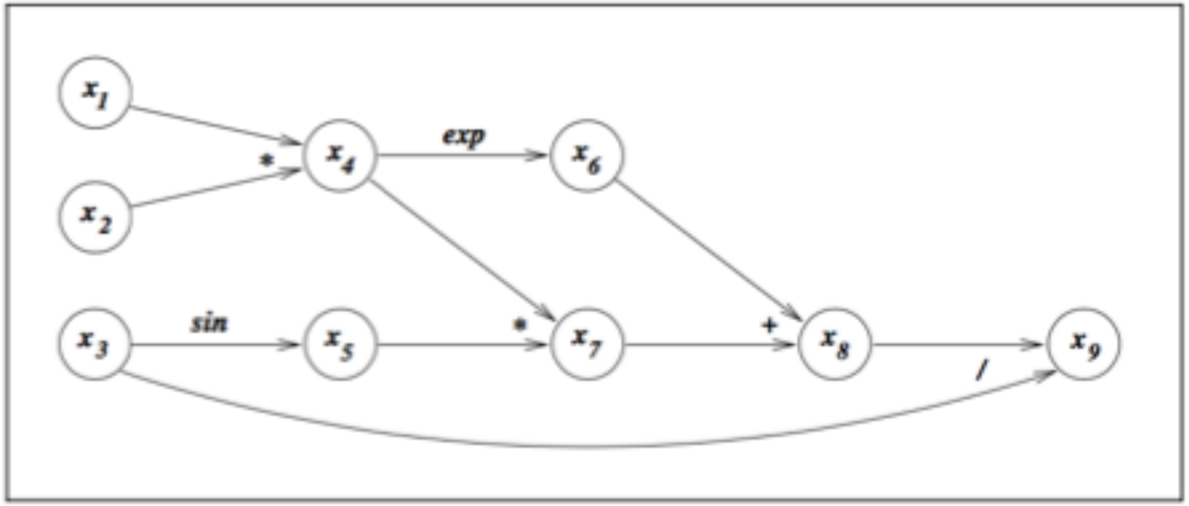
Example

- We have a function of 3 variables

$$f(x) = (x_1 x_2 \sin x_3 + e^{x_1 x_2})/x_3.$$

- We break it down into individual operations

$$
\begin{aligned}
x_4 &= x_1 * x_2, \\
x_5 &= \sin x_3, \\
x_6 &= e^{x_4}, \\
x_7 &= x_4 * x_5, \\
x_8 &= x_6 + x_7, \\
x_9 &= x_8 / x_3.
\end{aligned}
$$



Forward Mode

$$x_4 = x_1 * x_2,$$
$$x_5 = \sin x_3,$$
$$x_6 = e^{x_4},$$
$$x_7 = x_4 * x_5,$$
$$x_8 = x_6 + x_7,$$
$$x_9 = x_8 / x_3.$$

- Here we can only do directional derivatives:

$$D_p x_i \stackrel{\text{def}}{=} (\nabla x_i)^T p = \sum_{j=1}^{3} \frac{\partial x_i}{\partial x_j} p_j, \quad i = 1, 2, \ldots, 9,$$

- To get final answer $D_p x_9$ we will use $p_1 = (1,0,0)$, $p_2 = (0,1,0)$, $p_3 = (0,0,1)$

- Suppose we want to evaluate $D_p x_7$ and we have the values on previous steps (x_4 and x_5 and their D_p's):

$$\nabla x_7 = \frac{\partial x_7}{\partial x_4} \nabla x_4 + \frac{\partial x_7}{\partial x_5} \nabla x_5 = x_5 \nabla x_4 + x_4 \nabla x_5.$$

$$\nabla_x h(y(x)) = \sum_{i=1}^{m} \frac{\partial h}{\partial y_i} \nabla y_i(x)$$

$$D_p x_7 = \frac{\partial x_7}{\partial x_4} D_p x_4 + \frac{\partial x_7}{\partial x_5} D_p x_5 = x_5 D_p x_4 + x_4 D_p x_5.$$

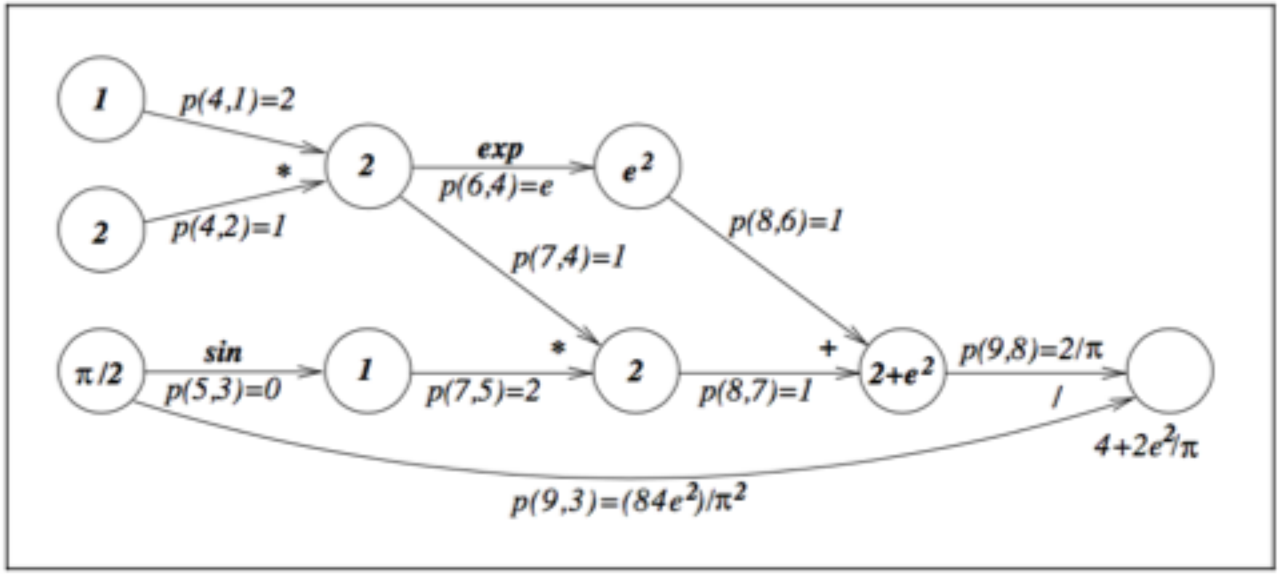
- +: Simple to evaluate, no need to store anything
- - : expensive, typically by a factor of n (# dimensions)

Backward Mode: Backpropagation

- Here we store values at each step and perform reverse sweep over the computational graph
- We associate adjoint variable (scalar) $\bar{x}_i$ to keep track of $\partial f / \partial x_i$ at each node, initializing them to 0, except last one $x_N = 1$ (since $f = x_N$)
- We use chain rule as $\frac{\partial f}{\partial x_i} = \sum_{j \text{ a child of } i} \frac{\partial f}{\partial x_j} \frac{\partial x_j}{\partial x_i}$. performing
- $\bar{x}_i \mathrel{+}= \frac{\partial f}{\partial x_j} \frac{\partial x_j}{\partial x_i}$. $x \mathrel{+}= a$ means $x \leftarrow x + a$. over all children
- Now we can use this as one input into parent of x_i
- We work with numerical values. Forward sweep stores x_i and $\partial x_j / \partial x_i$. as numerical values, which are then used in reverse sweep

Forward Sweep

- For previous example: we have to do it for specific numerical values (no symbolic algebra)
- Assume $x = (1, 2, \pi/2)^T$. Denote $p(x_j,x_i) = \partial x_j/\partial x_i$.



$$
\begin{aligned}
x_4 &= x_1 * x_2, \\
x_5 &= \sin x_3, \\
x_6 &= e^{x_4}, \\
x_7 &= x_4 * x_5, \\
x_8 &= x_6 + x_7, \\
x_9 &= x_8/x_3.
\end{aligned}
$$

Reverse Sweep

- For reverse sweep we start with
- Node 9 is child of 3 and 8:
- Node 8 is finalized, node 3 still needs input from child node 5
- Next we update 6 and 7 with 8
- 6 and 7 are finalized, use them for 4 and 5…
- Final result is:

$$\begin{bmatrix} \bar{x}_1 \\ \bar{x}_2 \\ \bar{x}_3 \end{bmatrix} = \nabla f(x) = \begin{bmatrix} (4+4e^2)/\pi \\ (2+2e^2)/\pi \\ (-8-4e^2)/\pi^2 \end{bmatrix}$$

$$\bar{x}_9 = 1 \qquad \bar{x}_9 = \partial f/\partial x_9$$

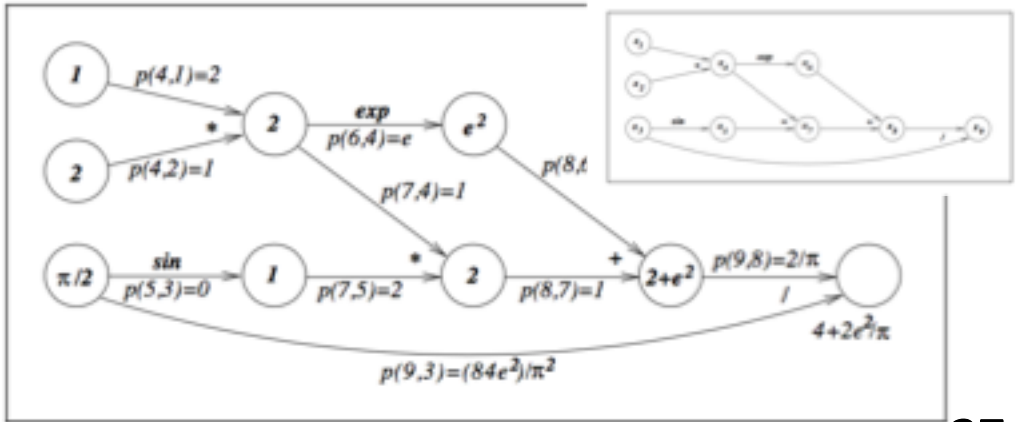
$$\bar{x}_3 += \frac{\partial f}{\partial x_9}\frac{\partial x_9}{\partial x_3} = -\frac{2+e^2}{(\pi/2)^2} = \frac{-8-4e^2}{\pi^2},$$

$$\bar{x}_8 += \frac{\partial f}{\partial x_9}\frac{\partial x_9}{\partial x_8} = \frac{1}{\pi/2} = \frac{2}{\pi}.$$

$$\bar{x}_6 += \frac{\partial f}{\partial x_8}\frac{\partial x_8}{\partial x_6} = \frac{2}{\pi};$$

$$\bar{x}_7 += \frac{\partial f}{\partial x_8}\frac{\partial x_8}{\partial x_7} = \frac{2}{\pi}.$$

$$\begin{aligned} x_4 &= x_1 * x_2, \\ x_5 &= \sin x_3, \\ x_6 &= e^{x_4}, \\ x_7 &= x_4 * x_5, \\ x_8 &= x_6 + x_7, \\ x_9 &= x_8 / x_3. \end{aligned}$$

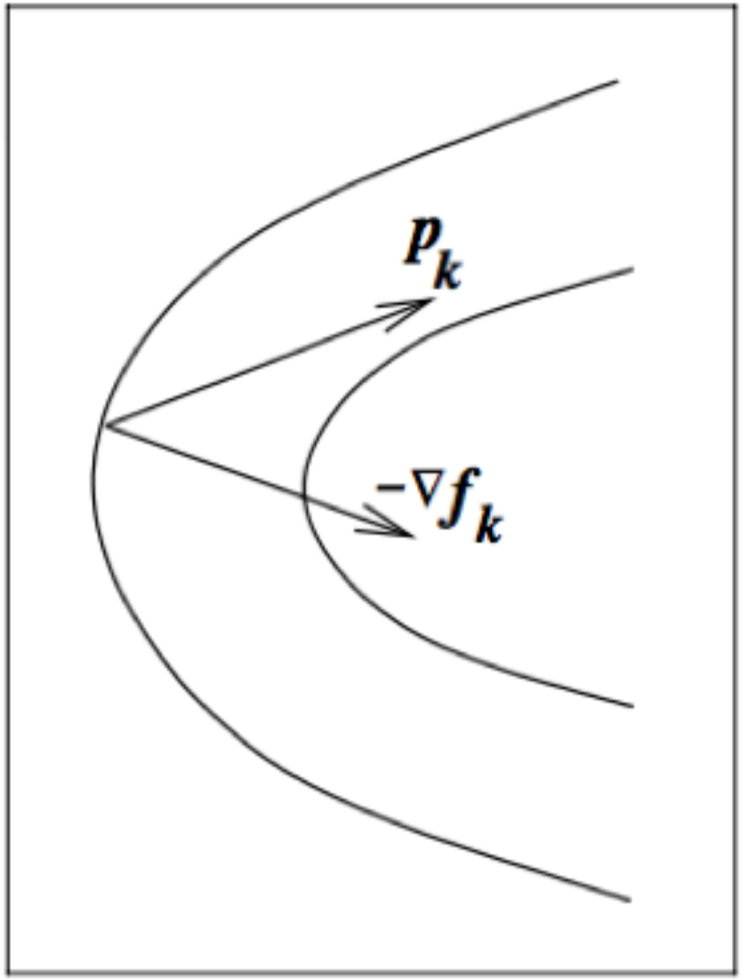


Backpropagation (Dis)Advantages

- +: computationally cheaper if f is a scalar: we get the full gradient with a cost comparable to the function evaluation: typically a few times more to evaluate $p(x_j,x_i)$. It is the only option if number of dimensions 10^6++
- - : we need to store all intermediate steps during forward sweep. This can get expensive if large number of dimensions and many operations
- In NN applications, due to high dimensionality of the networks, backpropagation is used exclusively: size of network will be limited (need to store all hidden layers)
- Note that memory requirements can limit the number of hidden layers
- In ODE/PDE applications important to have low number of time steps

General Strategy for optimization

- We want to descend down a function $J(a)$ (if minimizing) using iterative sequence of steps at a_t. For this we need to choose a direction p_t and move in that direction: $J(a_t+\eta p_t)$
- A few options: fix η
- line search: vary η until $J(a_t+\eta p_t)$ is minimized
- Trust region: construct an approximate quadratic model for J and minimize it but only within trust region where quadratic model is approximately valid



Line Search Directions and Backtracking

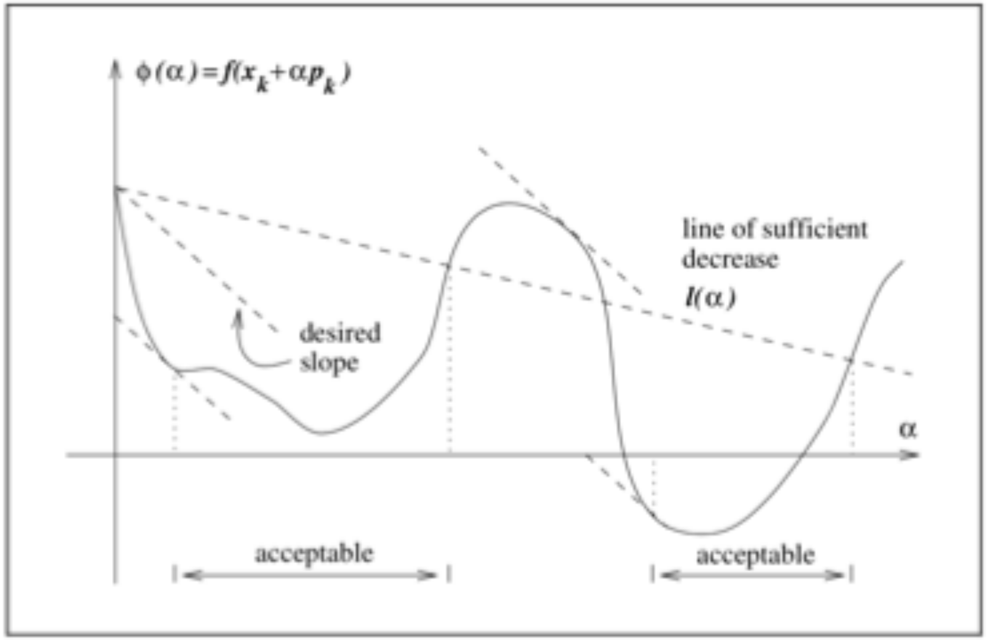
- Gradient descent: Gradient $-\eta \nabla_a J(a,x_t)$
- Newton: Inverse Hessian H^{-1} times gradient
  $-H^{-1} \nabla_a J(a)$
- Quasi-Newton: approximate H^{-1} with B^{-1} (SR1 and BFGS)
- Nonlinear conjugate gradient:
  $p_t = -\nabla_a J(a,x_t) + \beta_t p_{t-1}$, where p_{t-1} and p_t are conjugate
- Step length with backtracking: choose first proposed length
- If it does not reduce the function value reduce it by some factor, check again
- Repeat until step length is ε, at that point switch to gradient descent

Line search: Wolfe conditions

- We want a sufficient decrease of the loss function along the direction p_k
- We do not want a step that is too short, so we impose curvature condition: the slope at the position where we are stepping to should be shallow

$$f(x_k + \alpha_k p_k) \le f(x_k) + c_1 \alpha_k \nabla f_k^T p_k$$

$$\nabla f(x_k + \alpha_k p_k)^T p_k \ge c_2 \nabla f_k^T p_k,$$



Trust Region Method

- Multi-dim parabola method: define approximate quadratic function, but limit the step

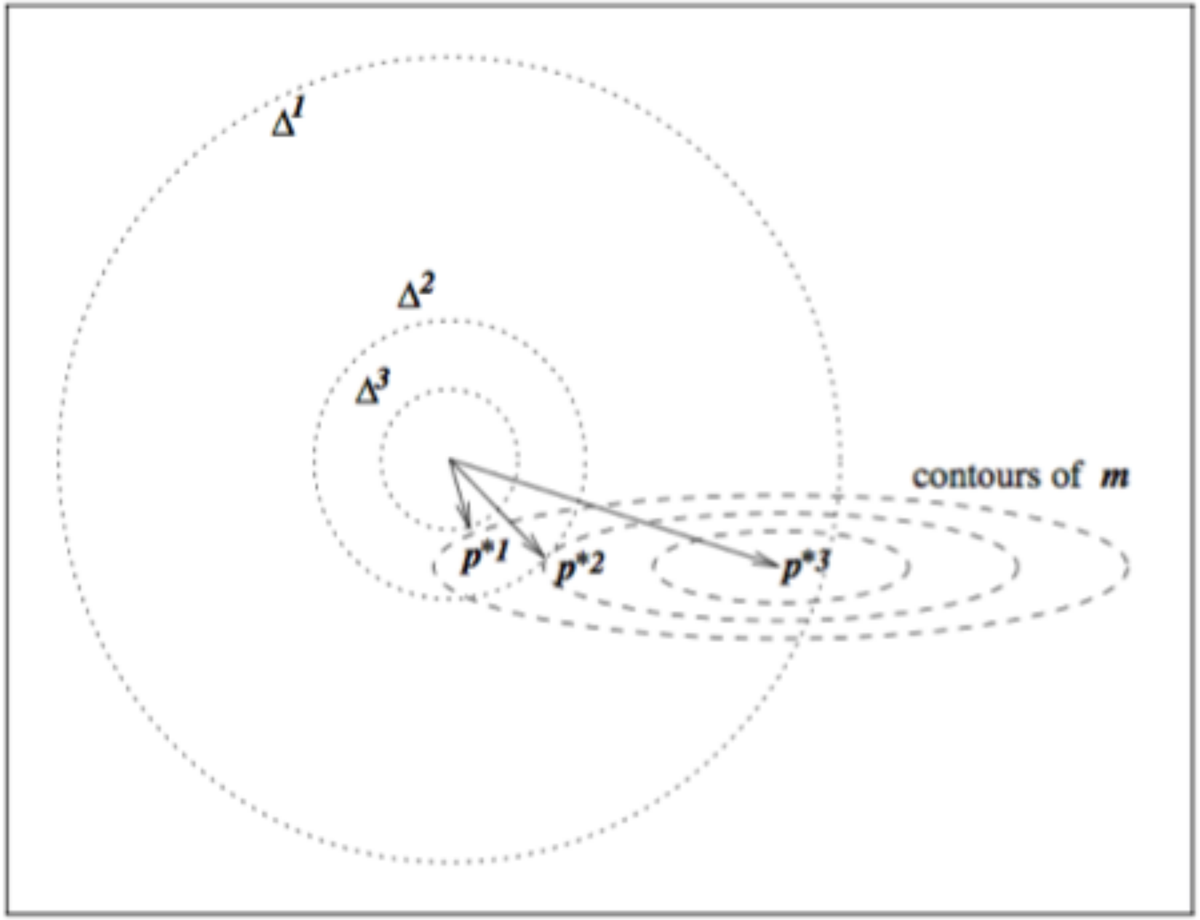
$$\min_{p\in\mathbb{R}^n} m_k(p) = f_k + g_k^T p + \tfrac{1}{2} p^T B_k p \qquad \text{s.t. } \|p\| \le \Delta_k$$

- Here Δ_k is trust region radius
- Evaluate at previous iteration and compare the actual reduction to predicted reduction

$$\rho_k = \frac{f(x_k) - f(x_k + p_k)}{m_k(0) - m_k(p_k)}$$

- If ρ_k around 1 we can increase Δ_k
- If close to 0 or negative we shrink Δ_k

- If trust region covers *m* center step there (same as Newton if B is Hessian)
- Otherwise direction of step changes to the point on the sphere of radius Δ which is the closest to the m



Constrained Optimization: Lagrange Multiplier Method

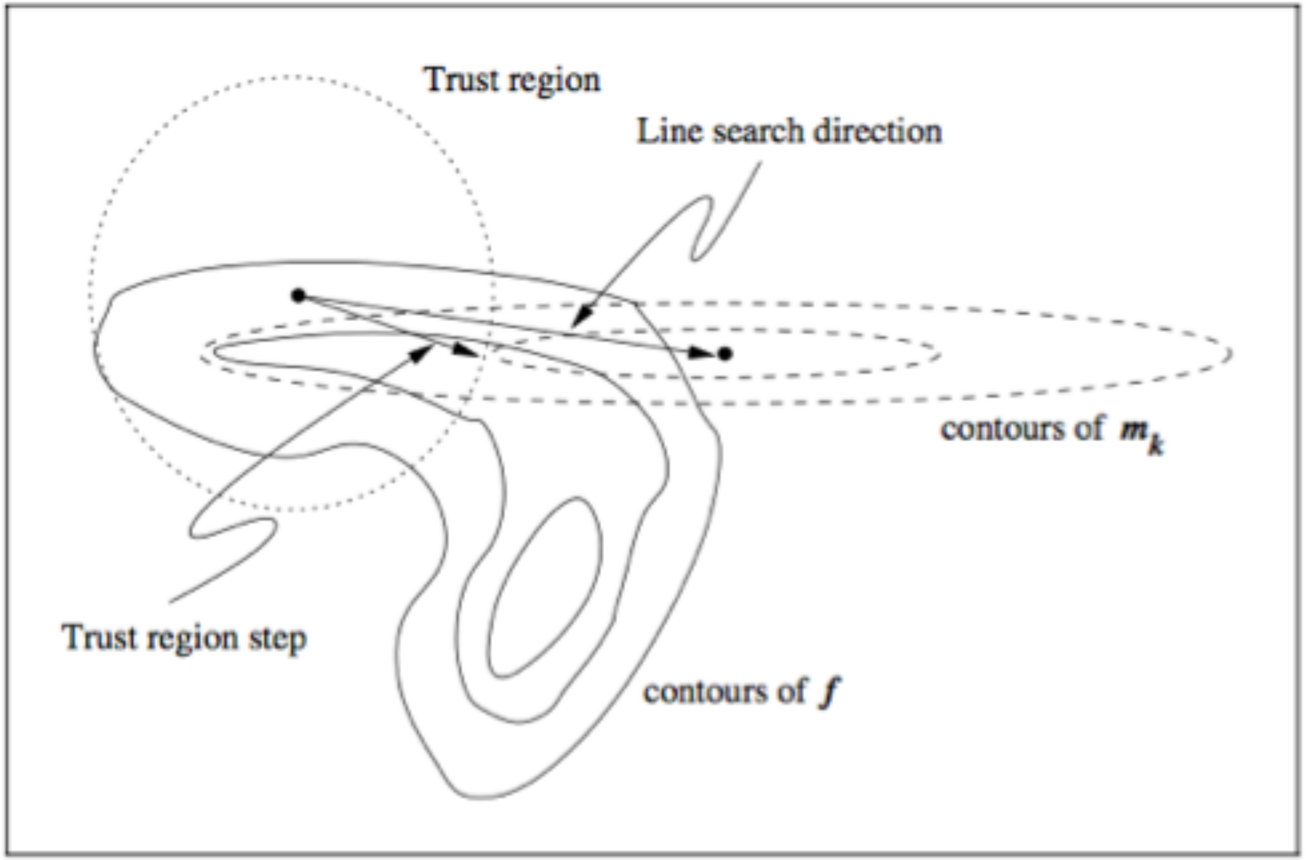
- If the center of m is inside trust region step there
- Otherwise we must solve constrained optimization
- We solve this optimization with Lagrange multiplier method: minimize $f + g^T p + p^T B p + \lambda\,(p^2 - \Delta^2)$ with respect to p and λ. Gradient w.r.t. λ gives the constraint $p^2 = \Delta^2$, thus the constraint is automatically satisfied. This determines the value of λ.
- Minimization with respect to p now includes λp^2
- As a result the step direction is not towards center of m when trust region does not cover it: see picture on previous and next slide

Line Search vs. Trust Region

In 2nd order methods we have a natural choice of step size: $\alpha_k=1$

This does not mean we should actually go there: Wolfe conditions may be violated, or it is outside trust region

The two methods give a different update



1^{st} Order: Gradient Descent

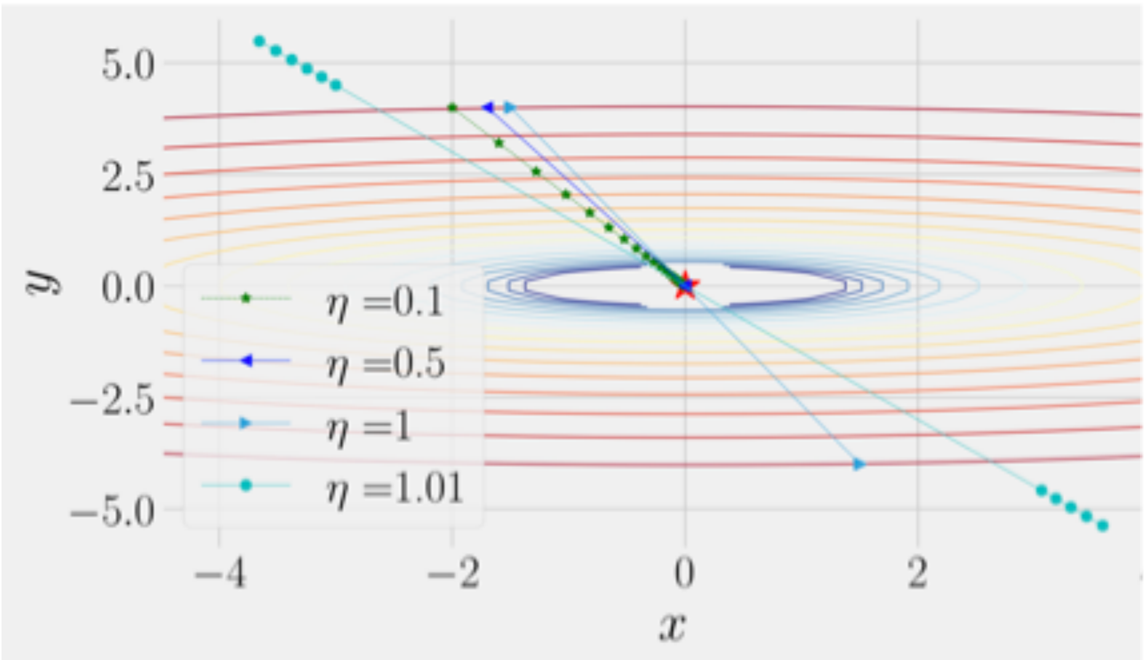
- We have a vector of parameters a and a scalar loss (cost) function $J(a,x,y)$ which is a function of a data vector (x,y) we want to optimize (say minimize). This could be a nonlinear least square loss function: $J = \chi^2$

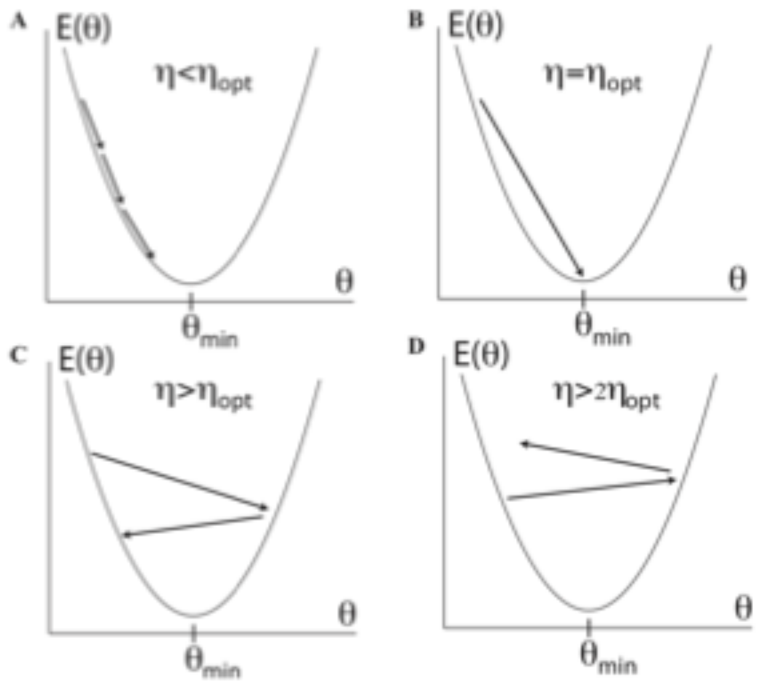
$$\chi^2(\mathbf{a}) = \sum_{i=0}^{N-1} \left[\frac{y_i - y(x_i|\mathbf{a})}{\sigma_i} \right]^2$$

- (Batch) gradient descent updates all the variables at once: $\delta a = - \eta \nabla_a J(a)$: in ML. η is called learning rate. We are not given its value, so we need to guess
- This is a poor strategy if condition number is large
- It can get stuck on saddle points, where gradient is 0 everywhere (see animation later)

Gradient Descent: learning rate

- If too small slow convergence (η=0.1). If optimal immediate convergence (η_{opt}=0.5). If large oscillation (η=1). If even larger(η>2 η_{opt}) divergence (η=1.01)

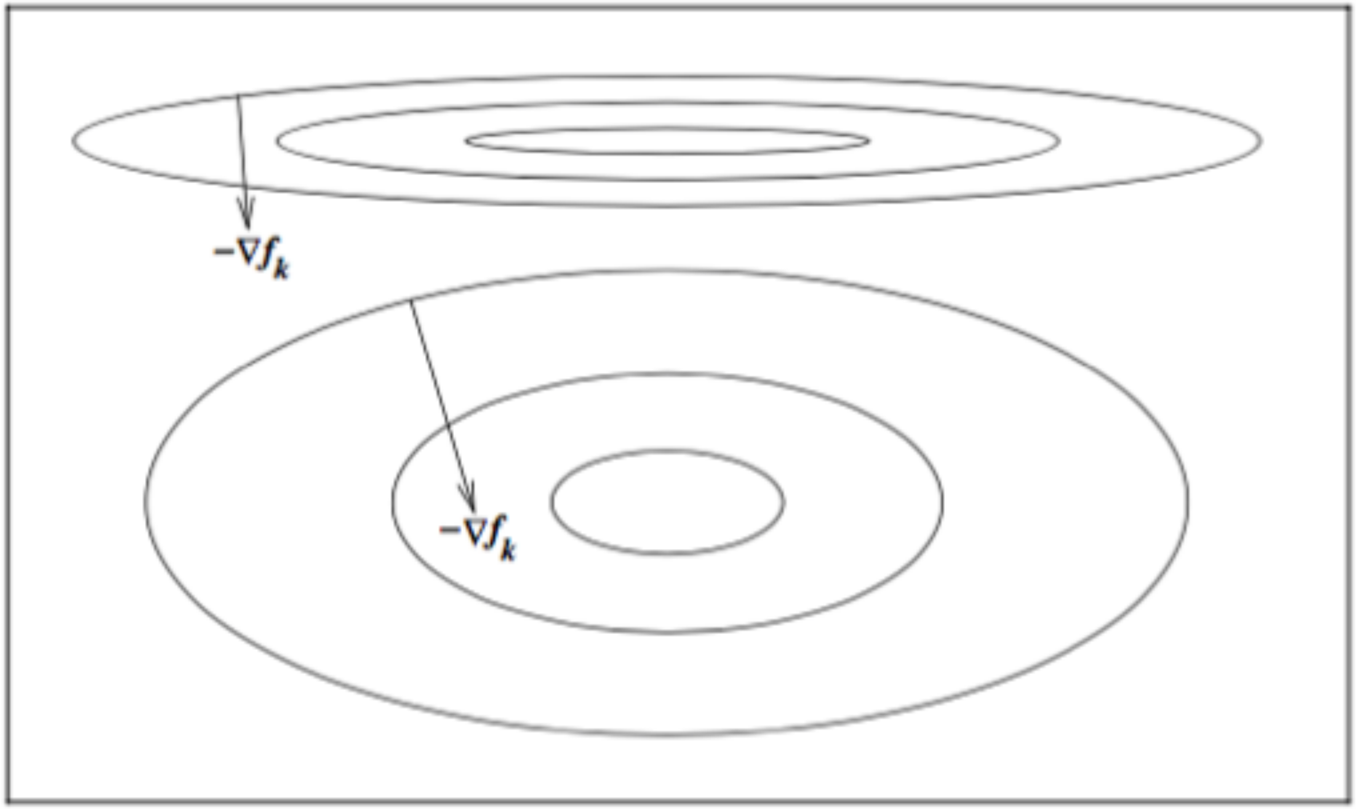




- Newton: $\eta \nabla_a J(a) = H^{-1} \nabla_a J(a)$: optimal η_{opt} is determined by the eigenvalues λ of inverse Hessian: $\eta_{opt} = 1/\lambda$

Scaling

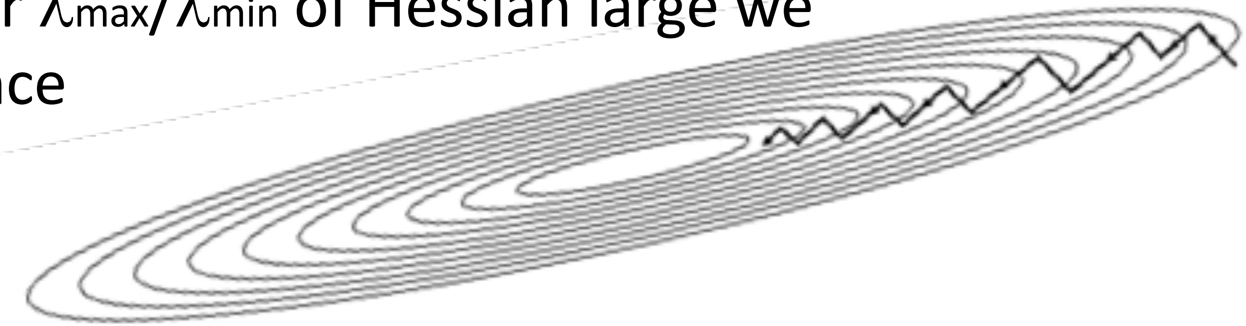
- Change variables to make surface more circular
- Example: change of units (rescale the variables)
- Only works if variables uncorrelated (Hessian is diagonal)



Ravines: large condition number of Hessian

To prevent divergence one must have $\eta<2/\lambda_{max}$

If condition number $\lambda_{max}/\lambda_{min}$ of Hessian large we get slow convergence



(a)

(b)

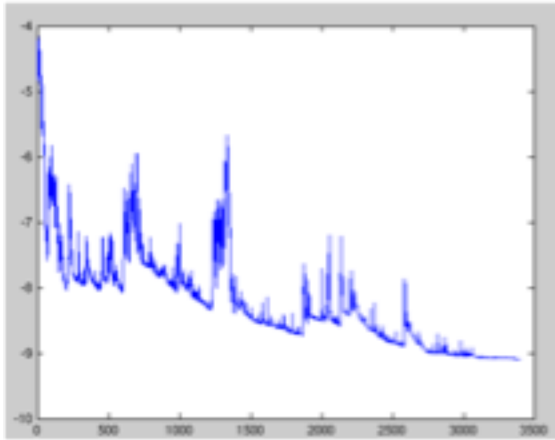
Given Hessian Q the convergence rate of gradient descent is

$$\|x_{k+1}-x^*\|_Q^2 \leq \left(\frac{\lambda_n-\lambda_1}{\lambda_n+\lambda_1}\right)^2 \|x_k-x^*\|_Q^2$$

where $0<\lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_n$ *are the eigenvalues of* Q.

Stochastic Gradient Descent

- Stochastic gradient descent: do this just for one data pair x_i, y_i:
  $\delta a = -\eta \nabla_a J(a, x_i, y_i)$
- This saves on computational cost, but is noisy, so one repeats it by randomly choosing data i
- Has large fluctuations in the cost function



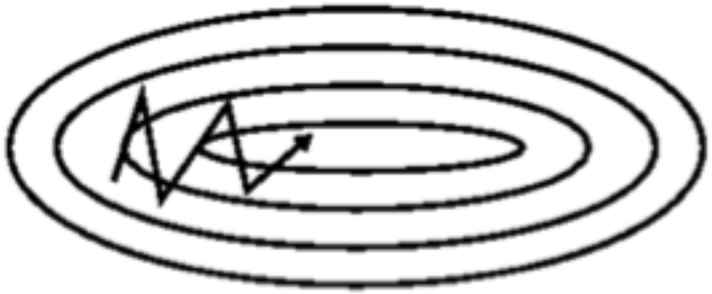
- This is potentially a good thing: it may avoid getting stuck in the local minima (or saddle points)
- Learning rate is slowly reduced
- Has revolutionized machine learning (can handle large data)

Mini-batch Stochastic Gradient

- Mini-batch takes advantage of hardware and software implementations where a gradient w.r.t. to a number of data points can be evaluated as fast as a single data (e.g. mini-batch of $N = 256$ for GPUs)
- We cycle over all minibatches: this is called an epoch
- Randomizing mini batches prevents fitting spurious correlations
- Challenges of (stochastic) gradient descent: how to choose learning rate (in 2nd order methods this is given by Hessian)
- Ravines: still slow

Adding Momentum: Rolling down the hill

- We can add momentum and mimic a ball rolling down the hill
- Use previous update as the direction
- $v_t = \gamma v_{t-1} + \eta \nabla_a J(a)$, $\Delta a_{t+1} = a_{t+1} - a_t = -v_t$ with γ of order 1 (e.g. 0.9)
- Momentum increases descent speed for directions where gradient does not change, while not affecting large gradient directions



- Physics analogy: viscous fluid with drag coefficient μ in external potential E=J

$$m\frac{d^2\mathbf{w}}{dt^2} + \mu\frac{d\mathbf{w}}{dt} = -\nabla_w E(\mathbf{w})$$

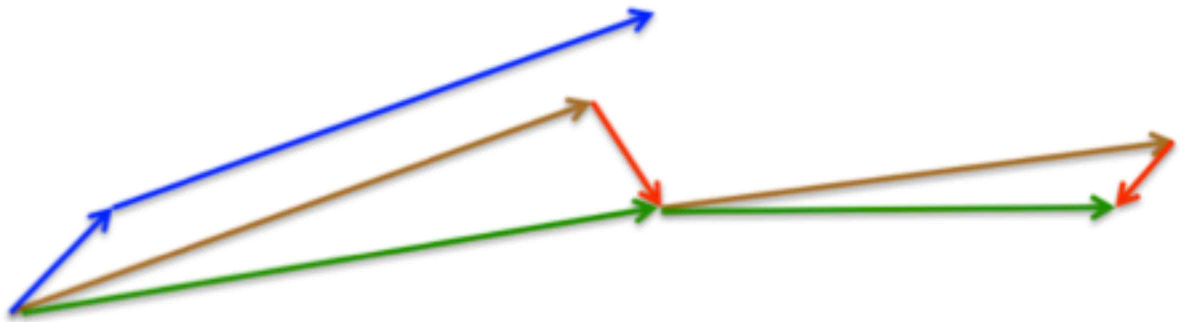
$$m\frac{\mathbf{w}_{t+\Delta t} - 2\mathbf{w}_t + \mathbf{w}_{t-\Delta t}}{(\Delta t)^2} + \mu\frac{\mathbf{w}_{t+\Delta t} - \mathbf{w}_t}{\Delta t} = -\nabla_w E(\mathbf{w})$$

$$\Delta\mathbf{w}_{t+\Delta t} = -\frac{(\Delta t)^2}{m+\mu\Delta t}\nabla_w E(\mathbf{w}) + \frac{m}{m+\mu\Delta t}\Delta\mathbf{w}_t$$

$$\gamma = \frac{m}{m+\mu\Delta t}, \qquad \eta = \frac{(\Delta t)^2}{m+\mu\Delta t}$$

Nesterov Accelerated Gradient

- We can predict where to evaluate the next gradient using previous velocity/momentum update
- $v_t = \gamma v_{t-1} + \eta \nabla_a J(a - \gamma v_{t-1}),\ \Delta a = -v_t$
- Momentum (blue) vs NAG (brown+red=green)



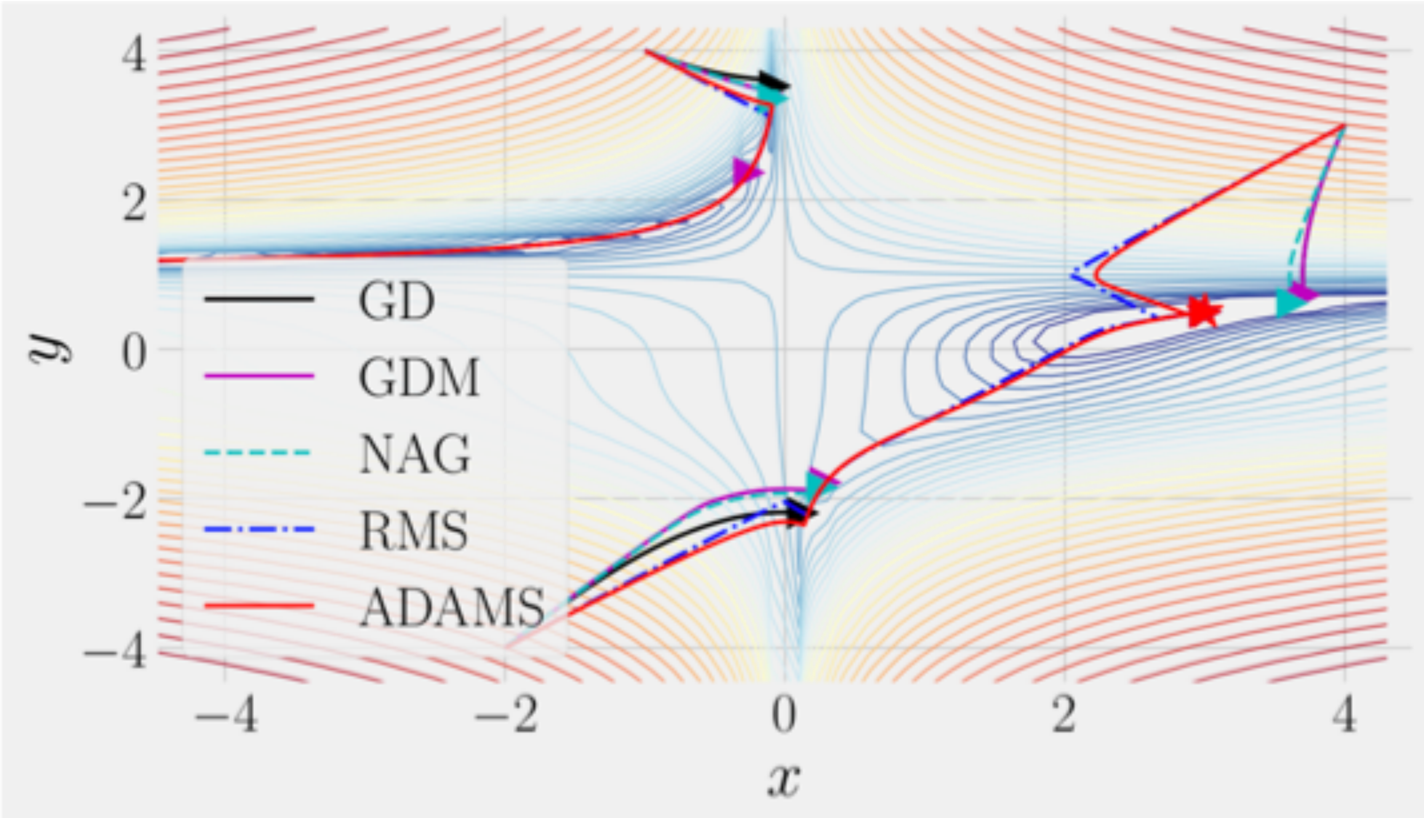
- See https://arxiv.org/abs/1603.04245 for theoretical justification of NAG based on a Bregman divergence Lagrangian

Using second moment of gradient: Adagrad, Adadelta, Rmsprop, ADAM, …

- How to specify the schedule: updates of η dependent on a_i
- Use past gradient information to update η: " quasi second order"
- Example: RMSprop
- $g_t = \nabla_a J(a);\ v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$: average gradient squared with element wise vector operations
- Update rule: $\Delta a_{t+1} = -\eta\, g_t / (v_t^{1/2} + \varepsilon)$: reduces learning rate where the gradient is large
- Example ADAM: ADAptive Momentum estimation
- $m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$: average gradient
- bias correction: $m_t' = m_t/(1-\beta_1),\ v_t' = v_t/(1-\beta_2)$
- Update rule: $\Delta a_{t+1} = -\eta\, m_t' / (v_t'^{1/2} + \varepsilon)$.
- Recommended values $\beta_1 = 0.9,\ \beta_2 = 0.999,\ \eta = 10^{-3},\ \varepsilon = 10^{-8}$

Practical performance

- ADAM and RMSprop are faster to converge
- But generalization properties may be worse
- Monitor out of sample performance and exit (early termination) when validation error starts increasing (overfitting)



Adding gradient noise

- To avoid getting stuck on local minima it is helpful to add noise (Neelakantan etal 2015)
- Stochastic gradient descent already does that, but sometimes we want more
- If we do full batch we can add noise explicitly to the gradient and then anneal it to zero

$$g_{t,i} = g_{t,i} + N(0, \sigma_t^2). \qquad \sigma_t^2 = \frac{\eta}{(1+t)^\gamma}.$$

- If we do not anneal to zero this becomes one of the MC sampling methods called Langevin sampling (Welling and Teh 2011): next lecture

2^{nd} Order Method: Newton

- We have seen that there is no natural way to choose learning rate in 1^{st} order methods
- But Newton's method provides a clear answer what the learning rate should be:
- $J(a+\delta a) = J(a) + \delta a \nabla_a J(a) + \delta a \delta a' \nabla_a \nabla_{a'} J(a)/2 \ldots$
- Hessian $H_{ij} = \nabla_{a\_i} \nabla_{a\_j} J(a)$
- At the extremum we we want $\nabla_a J(a) = 0$ so a Newton update step is $\delta a = -H^{-1} \nabla_a J(a)$
- We do not need to guess the learning rate
- We do need to evaluate Hessian and invert it (or use LU): expensive in many dimensions!
- In high dimensions we use iterative schemes to solve the inverse problem

Quasi-Newton

- Computing Hessian and inverting it is expensive, but one can approximate it at iteration k with a low rank symmetric tensor B_{k+1}
- Let us construct an approximation $m_{k+1}(p) = f_{k+1} + \nabla f_{k+1}^T p + \frac{1}{2} p^T B_{k+1} p.$
- Wolfe condition for α_k $p_k = -B_k^{-1} \nabla f_k.$ $x_{k+1} = x_k + \alpha_k p_k$
- Remember secant method: we used finite difference to approximate gradient. Here we use finite difference of gradient to approximate Hessian. We'd need (N+1)/2 such terms to get the full Hessian, so if we do r we can only get low rank r approximation
- Secant condition: $s_k = x_{k+1} - x_k, \quad y_k = \nabla f_{k+1} - \nabla f_k. \quad B_{k+1} s_k = y_k$
- We also want B_{k+1} to be positive definite: curvature condition $s_k^T y_k > 0.$
- This is not unique, as we have N(N+1)/2 elements but just N conditions, so we add a condition that B_{k+1} is closest to B_k

$$\min_B \|B - B_k\|$$
$$\text{subject to} \quad B = B^T, \quad B s_k = y_k$$

Quasi-Newton: choice of norm

- Frobenius norm where $\|\cdot\|_F$ is defined by $\|C\|_F^2 = \sum_{i=1}^n \sum_{j=1}^n c_{ij}^2$

$$\|A\|_W \equiv \left\|W^{1/2} A W^{1/2}\right\|_F \quad W y_k = s_k$$

- Unique solution: DFP (Davidon-Fletcher-Powell)

$$\text{(DFP)} \qquad B_{k+1} = \left(I - \rho_k y_k s_k^T\right) B_k \left(I - \rho_k s_k y_k^T\right) + \rho_k y_k y_k^T,$$

$$\rho_k = \frac{1}{y_k^T s_k}.$$

- Inverse H=B^{-1} (Sherman-Morrison-Woodbury formula)

$$H_{k+1} = H_k - \frac{H_k y_k y_k^T H_k}{y_k^T H_k y_k} + \frac{s_k s_k^T}{y_k^T s_k}$$

Quasi-Newton: BFGS and SR1

- BFGS: same as DFP, but we impose secant condition on inverse H_{k+1} (rank 2 update, positive definite): most popular

$$\text{(BFGS)} \qquad H_{k+1} = (I - \rho_k s_k y_k^T) H_k (I - \rho_k y_k s_k^T) + \rho_k s_k s_k^T \quad \rho_k = \frac{1}{y_k^T s_k}$$

- Inverse gives B_{k+1} (Sherman-Morrison-Woodbury formula)

$$B_{k+1} = B_k - \frac{B_k s_k s_k^T B_k}{s_k^T B_k s_k} + \frac{y_k y_k^T}{y_k^T s_k}$$

- Even simpler is rank 1 update: $B_{k+1} = B_k + \sigma v v^T$
- Secant condition: $y_k = B_k s_k + \left[\sigma v^T s_k\right] v$
- Inside bracket is a scalar, so v proportional to $y_k - B_k s_k$
- Solution: Symmetric Rank 1 (SR1)

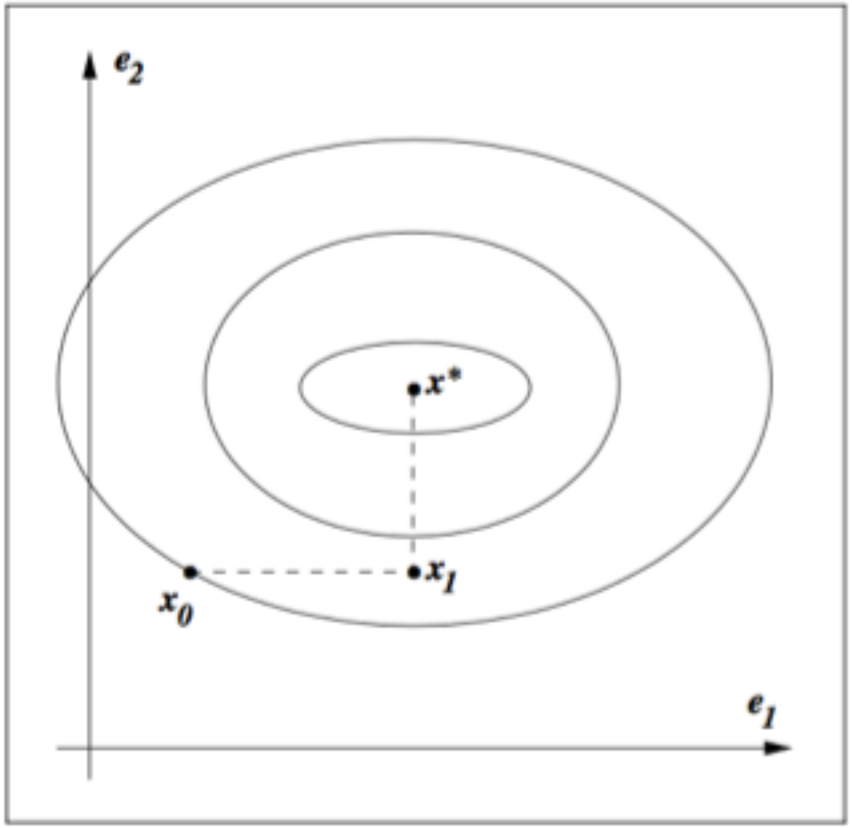
$$B_{k+1} = B_k + \frac{(y_k - B_k s_k)(y_k - B_k s_k)^T}{(y_k - B_k s_k)^T s_k}$$

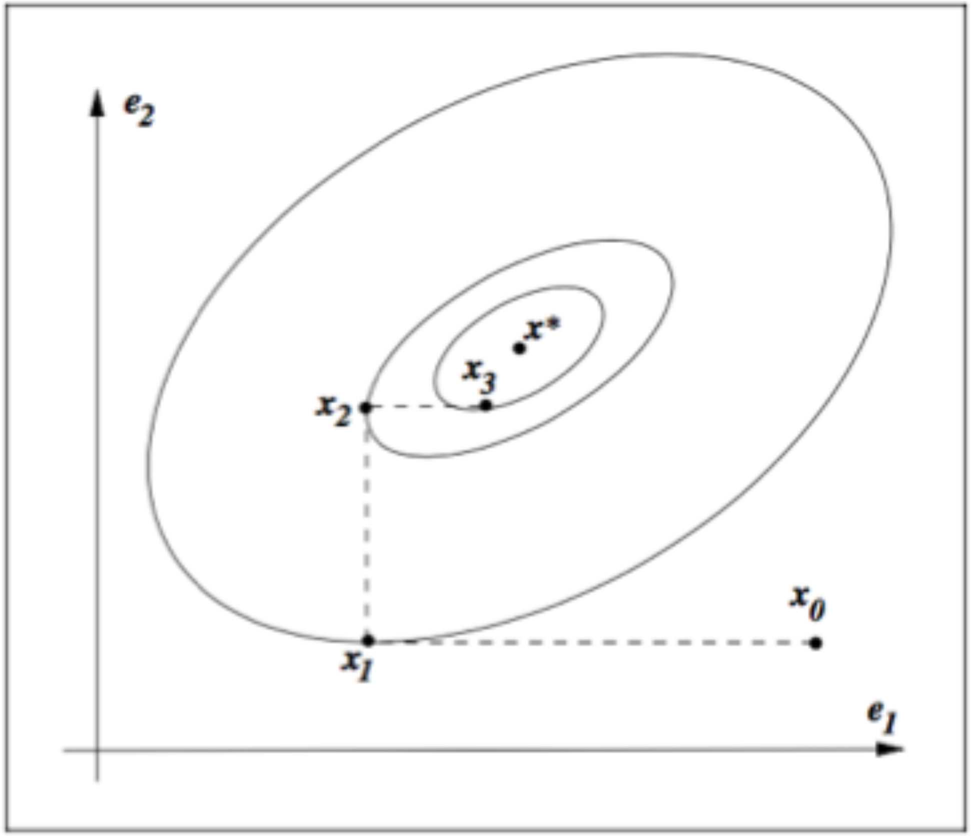
L-BFGS

- For large problems and many iterations this gets too expensive (too high rank). Limited memory BFGS updates only based on last N iterations (N of order 10-100)
- In practice increasing N often does not improve the results
- Historical note: quasi-Newton methods originated from W.C. Davidon’s work in 1950s, a physicist at Argonne national lab.

General minimization along coordinate direction

- If we have the matrix **A** in diagonal form so that basis vectors are orthogonal we can find the minimum trivially along the axes, otherwise not





Linear Conjugate Direction

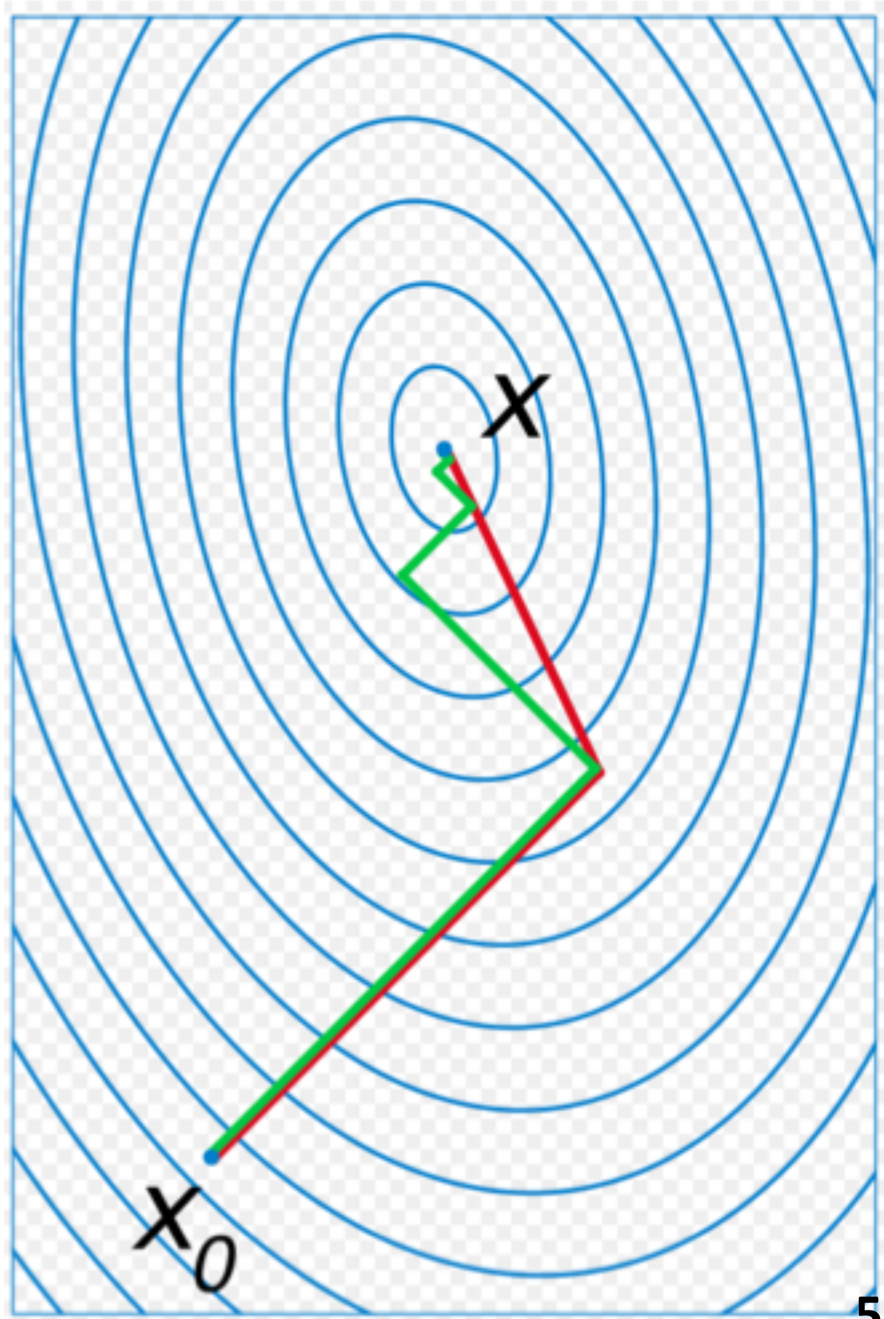
- Is an iterative method to solve $\boldsymbol{Ax} = \boldsymbol{b}$ (so belongs to linear algebra)
- Can be used for optimization: min $J = \boldsymbol{x}^T\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}^T\boldsymbol{x}$
- Assume we have conjugate vectors defined as $\boldsymbol{p}_i\boldsymbol{A}\boldsymbol{p}_j = 0$ for all *i, j* not equal *i*
- Construction of **x** similar to Gram-Schmidt (QR), where A plays the role of scalar product norm. Start at any $\mathbf{x}_0$
- $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \alpha_k\boldsymbol{p}_k$ where we choose α_k so that it minimizes J along $\mathbf{p}_k$. By construction of conjugate vectors this also minimizes along previous directions: $\alpha_k = -\boldsymbol{r}_k^T\boldsymbol{p}_k/(\boldsymbol{p}_k^T\boldsymbol{A}\boldsymbol{p}_k)$ and $\boldsymbol{r}_k = \boldsymbol{A}\boldsymbol{x}_k - \boldsymbol{b}$
- Essentially we are taking a dot product (with **A** norm) of the residual with previous vector to project it perpendicular to previous vectors
- Since the space is *N*-dim after *N* steps we have spanned the full space and converged to true solution, $\boldsymbol{r}_N=0$.
- How do we construct p_k?

Linear Conjugate Gradient: construction of $\mathbf{p}_k$

- Computes $\mathbf{p}_k$ from $\mathbf{p}_{k-1}$. $\mathbf{p}_0 = \mathbf{r}_0$
- We want the step to be linear combination of residual $-\mathbf{r}_k$ and previous direction $\mathbf{p}_{k-1}$ such that it is conjugate to it
- $p_k = -r_k + \beta_k p_{k-1}$ premultiply by $p^T{}_{k-1}A$ and require $p^T{}_{k-1}Ap_k = 0$
- $\beta_k = (r_k A p_{k-1})/(p^T{}_{k-1} A p_{k-1})$

CG vs. Gradient Descent

- In 2-d CG has to converge in 2 steps
- In general CG will converge much faster than gradient descent

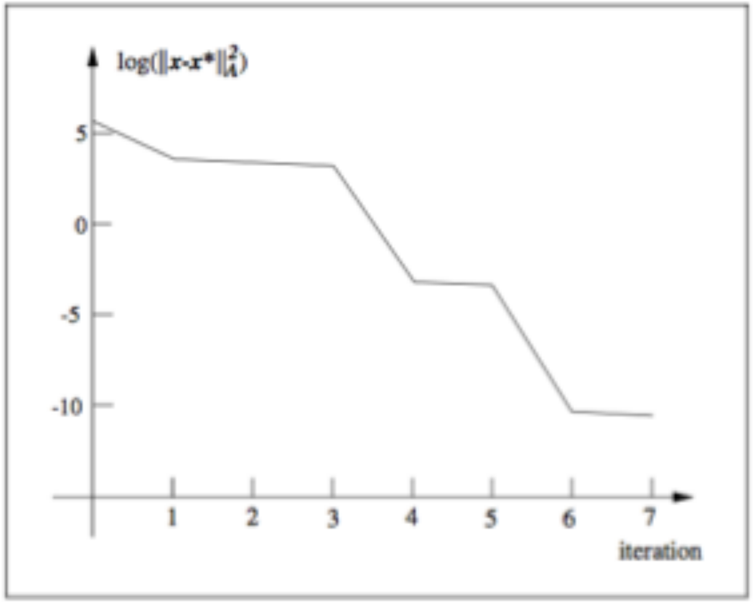


CG convergence rate

- Converges rapidly for similar eigenvalues: if matrix A has r distinct eigenvalues CG converges after k=r steps
- If r clusters of eigenvalues it converges approximately in r steps
- not so fast if condition number is high: slow convergence

$$\kappa(A) = \|A\|_2 \|A^{-1}\|_2 = \lambda_n/\lambda_1.$$

$$\|x_k - x^*\|_A \leq 2\left(\frac{\sqrt{\kappa(A)}-1}{\sqrt{\kappa(A)}+1}\right)^k \|x_0 - x^*\|_A$$



Preconditioning

- Tries to improve condition number of **A** by multiplying by another matrix **C** that is simple

$$\hat{x} = Cx.$$

$$\hat{\phi}(\hat{x}) = \tfrac{1}{2}\hat{x}^T (C^{-T} A C^{-1})\hat{x} - (C^{-T} b)^T \hat{x}.$$

$$(C^{-T} A C^{-1})\hat{x} = C^{-T} b$$

- We wish to reduce condition number of $C^{-T} A C^{-1}$
- Example: incomplete Cholesky $\mathbf{A} = \mathbf{L}\mathbf{L}^{T}$ by computing only a sparse **L**
- Preconditioners are very problem specific (use physical understanding of the problem to devise it)

Nonlinear Conjugate Gradient

- Replace $\alpha_{\mathbf{k}}$ with line search that minimizes J, and use $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \alpha_k \boldsymbol{p}_k$
- Replace $\boldsymbol{r}_k = \boldsymbol{A}\boldsymbol{x}_k - \boldsymbol{b}$ with gradient of J: $\nabla_{\boldsymbol{x}} J$
- $\beta_k = \nabla_{\boldsymbol{x}} J_k \nabla_{\boldsymbol{x}} J_k / \nabla_{\boldsymbol{x}} J_{k-1} \nabla_{\boldsymbol{x}} J_{k-1}$
- $\boldsymbol{p}_k = -\nabla_{\boldsymbol{x}} J_k + \beta_k \boldsymbol{p}_{k-1}$
- This is Fletcher-Reeves version, Polak-Ribiere modifies β
- CG is one of the most competitive methods, but requires the Hessian to have low condition number
- Typically we do a few CG steps at each k, then move on to a new gradient evaluation

Gauss-Newton for Nonlinear Least Squares

$$\chi^2(\mathbf{a}) = \sum_{i=0}^{N-1} \left[\frac{y_i - y(x_i|\mathbf{a})}{\sigma_i} \right]^2$$

$$\frac{\partial \chi^2}{\partial a_k} = -2 \sum_{i=0}^{N-1} \frac{[y_i - y(x_i|\mathbf{a})]}{\sigma_i^2} \frac{\partial y(x_i|\mathbf{a})}{\partial a_k} \qquad k = 0, 1, \ldots, M-1$$

$$\frac{\partial^2 \chi^2}{\partial a_k \partial a_l} = 2 \sum_{i=0}^{N-1} \frac{1}{\sigma_i^2} \left[\frac{\partial y(x_i|\mathbf{a})}{\partial a_k} \frac{\partial y(x_i|\mathbf{a})}{\partial a_l} - [y_i - y(x_i|\mathbf{a})] \frac{\partial^2 y(x_i|\mathbf{a})}{\partial a_l \partial a_k} \right]$$

$$\beta_k \equiv -\frac{1}{2} \frac{\partial \chi^2}{\partial a_k} \qquad \alpha_{kl} \equiv \frac{1}{2} \frac{\partial^2 \chi^2}{\partial a_k \partial a_l} \qquad \sum_{l=0}^{M-1} \alpha_{kl}\, \delta a_l = \beta_k$$

$$\alpha_{kl} = \sum_{i=0}^{N-1} \frac{1}{\sigma_i^2} \left[\frac{\partial y(x_i|\mathbf{a})}{\partial a_k} \frac{\partial y(x_i|\mathbf{a})}{\partial a_l} \right]$$

Gauss-Newton approximation: we drop 2^{nd} term in Hessian because residual r = y_i – y is small, fluctuates around 0 and because y" may be small (or zero for linear problems)

Line search in direction δa

Gauss-Newton + Trust Region = Levenberg-Marquardt Method

- Solving $\mathbf{A}^T\mathbf{A}\delta\mathbf{a} = \mathbf{A}^T\mathbf{b}$ is equivalent to minimize $|\mathbf{A}\delta\mathbf{a}-\mathbf{b}|^2$
- if trust region is within the solution just solve this equation
- If not we need to impose $||\delta\mathbf{a}||=\Delta_k$
- Lagrange multiplier minimization equivalent to $(\mathbf{A}^T\mathbf{A}+ \lambda I)\ \delta\mathbf{a} = \mathbf{A}^T\mathbf{b}$ and $\lambda(\Delta-||\delta\mathbf{a}||) = 0$
- For small λ this is Gauss-Newton (use close to minimum), for large λ this is steepest descent (use far from minimum)
- A good method for nonlinear least squares
- Steepest descent can be poor far away from the minimum: sometimes better to start with BFGS, then switch to L-M

Inexact Newton methods

- In high dimensions we cannot do linear algebra Hessian matrix H inversion to do the exact Newton (or Gauss Newton)
- We use conjugate gradient to solve the sub-problem $\nabla_x \nabla_x J\, \delta x = -\nabla_x J$ by iterating on a few conjugate directions
- Both line search and trust regions methods exist. The latter is called CG-Steihaug method
- Requires Hessian vector product: automatic derivative methods exist that do this (both forward and reverse modes)
- We do one or a few CG updates, then move on to recompute the gradient and Hessian vector product

Summary

- Optimization one of key numerical methods of modern data analysis. Typical examples are nonlinear least square problem and ML parameters (e.g. neural networks etc.)
- If at this point you are confused which methods you should use you are not alone: it depends on application and often the best way to answer is to try
- 2^{nd} order methods usually better in low dimensions
- 1^{st} order methods may be the only choice in high dimensions
- Analytic gradient is worth having: Auto-diff with backpropagation
- Even Hessian vector product is useful (e.g. CG-Steihaug)
- Alternative is finite difference gradient, but this suffers from numerical issues and gets very expensive in high dimensions

Summary

- If the data is independent and there is a lot of data then use stochastic 1st order methods, e.g. ADAM
- If the likelihood evaluation is slow and number of parameters low use Newton or Gauss-Newton (e.g. Levenberg-Marquardt)
- If likelihood slow and number of parameters large use approximate Newton or Gauss-Newton (e.g. Steihaug with nonlinear CG), or use quasi-Newton (e.g. L-BFGS)
- Choosing a method is not enough: you also need to choose line search method (e.g. backtracking, Wolfe conditions) or trust region determination
- Typically these methods only find local minimum. Non-convex problems are hard: we will look at some stochastic methods (e.g. simulated annealing) in next lecture

Literature

- *Numerical Recipes*, Press et al., Chapter 9, 10, 15
- *Computational Physics,* M. Newman, Chapter 6
- Nocedal and Wright, Optimization
- https://arxiv.org/abs/1609.04747