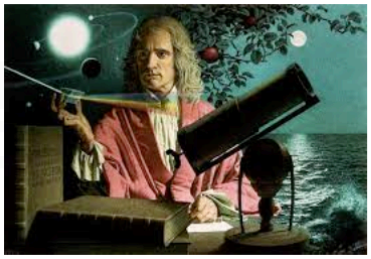
Numerical Solvers

Mathematical paradigms

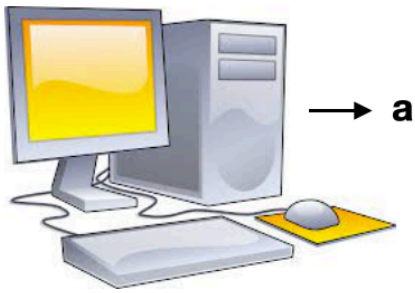
- Typically there are two options when it comes to doing mathematics
- **Analytical methods**: In this case means “by hand”, typically algebra or calculus

→ **elegant, intuitive, compact, efficent**

- **Numerical methods**: In this case means “by computer”, typically iterative “recipes”

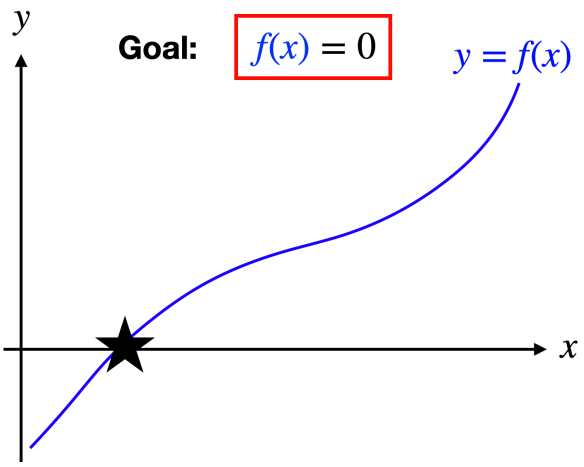
Algorithms and computers

→ **allows us to attack more complex problems**

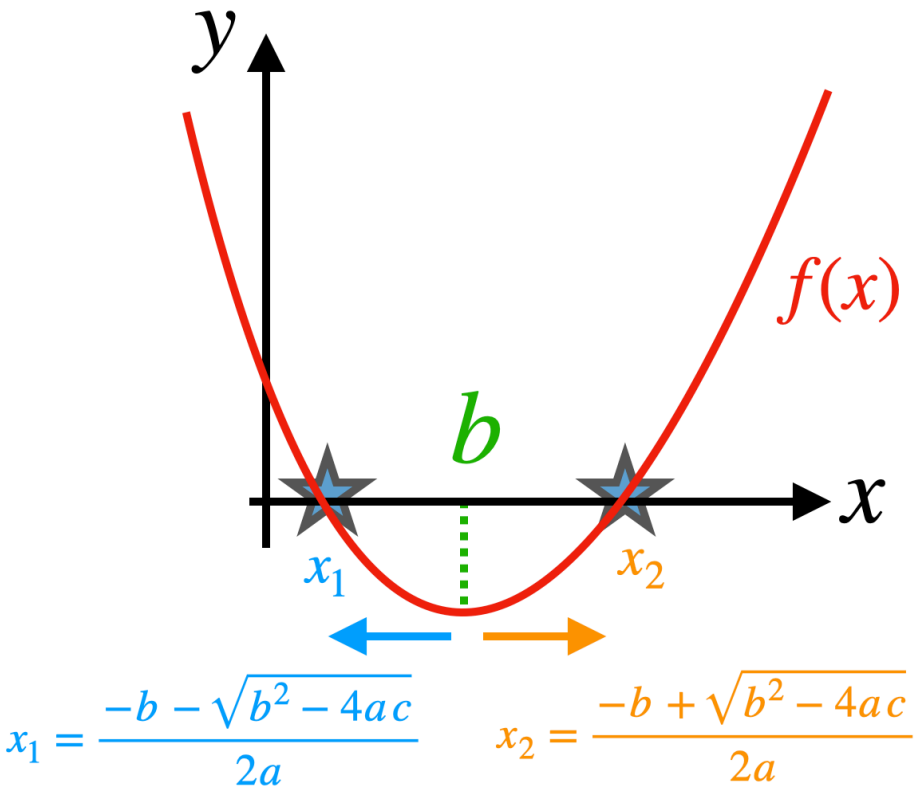
Roots of an equation

- For a single variable function $f(x)$, the goal of a solver is to find root(s) of a function f(x)
- Roots are special x values, denoted $\tilde{x}$, such that $f(\tilde{x}) = 0$
- In short, the goal is to solve $f(x) = 0$



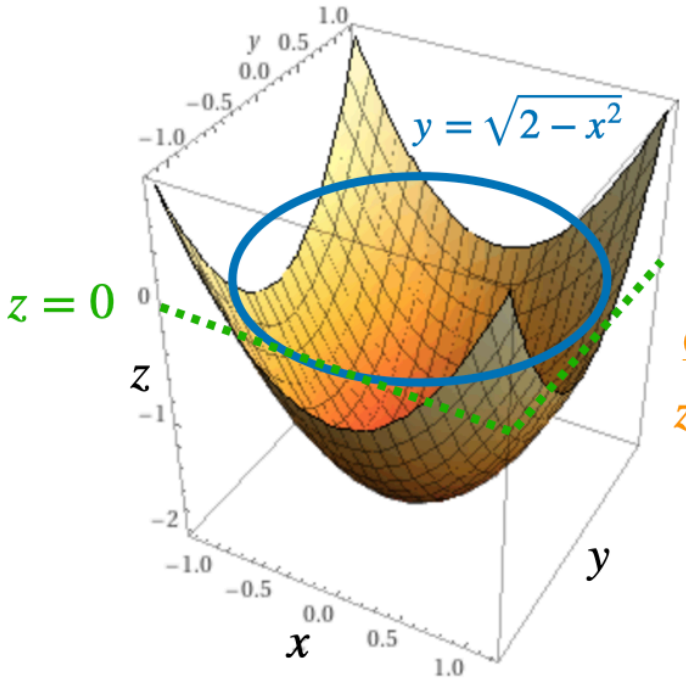
Analytical example

- Another famous analytical result is the quadratic formula
- The formula gives the roots of a parabolic function; given $f(x) = ax^2 + bx + c$
- The two points for which $f(x) = ax^2 + bx + c = 0$ are given by $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$



Aside: under-determined systems

- A system of equations is considered **under-determined** (under-constrained) if there are fewer equations than unknowns. The alternative is an **over-determined** system, where there are more equations than unknowns
- In this case there are infinitely many solutions (the solution is a function not a point) [1]

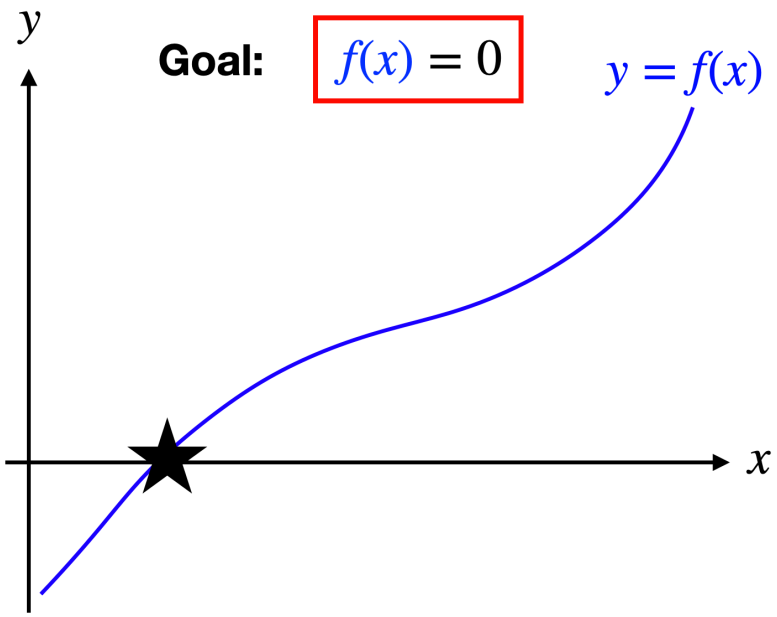


Newton's method

- Newton's method is an extremely famous numerical "solver" for $f(x) = 0$

$$\text{Newton's method: } x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

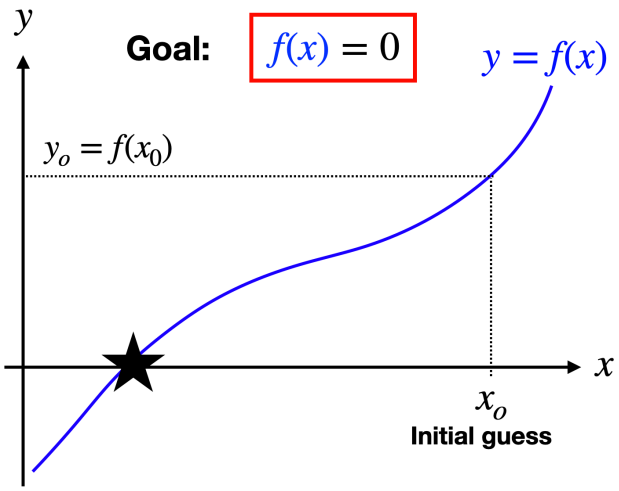
- A strong argument could be made that it is the most important formula in applied mathematics, since it is the corner-stone of ALL modern "gradient" based optimization.



Newton's method

- It is a very simple idea, let's take a quick look at how it works

- The derivation relies on the fact that the derivative is the slope of the tangent line
- Coupling this with the "point slope" formula allows us to derivate and **iterative procedure** for finding the roots of equations
- As with most numerical methods, it starts with an estimation x_0 for the root of the equation, known as the "initial guess"



Point-slope

$$m = \frac{rise}{run} = \frac{\Delta y}{\Delta x} = \frac{y - y_0}{x - x_0}$$

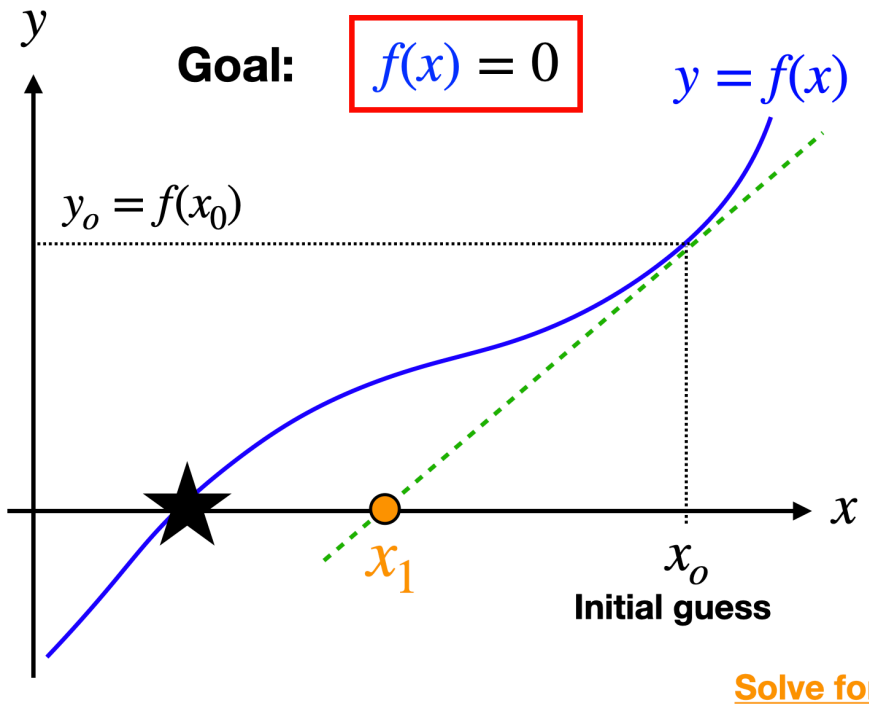
$$(y - y_0) = m(x - x_0)$$

Newton's method

initial guess $\rightarrow x_o$

Newton's method

- Next, we follow the tangent line "down" to obtain an approximation of the root



Point-slope

$$m = \frac{rise}{run} = \frac{\Delta y}{\Delta x} = \frac{y - y_0}{x - x_0}$$

$$(y - y_0) = m(x - x_0)$$

Newton's method

initial guess $\rightarrow x_o$

Tangent line

$$(y - y_0) = m(x - x_0)$$

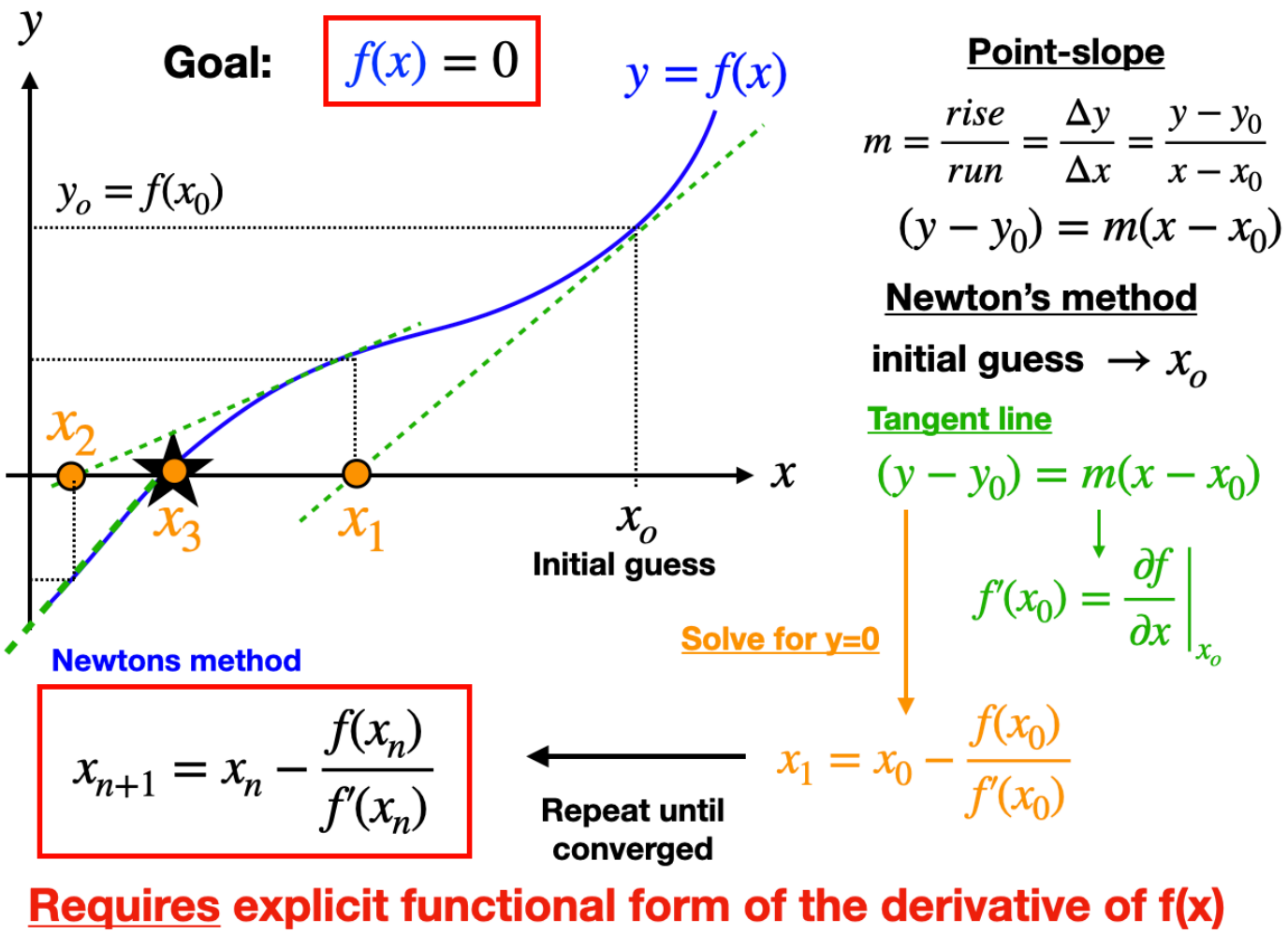
$$f'(x_0) = \left.\frac{\partial f}{\partial x}\right|_{x_o}$$

Solve for y=0

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

Newton's method

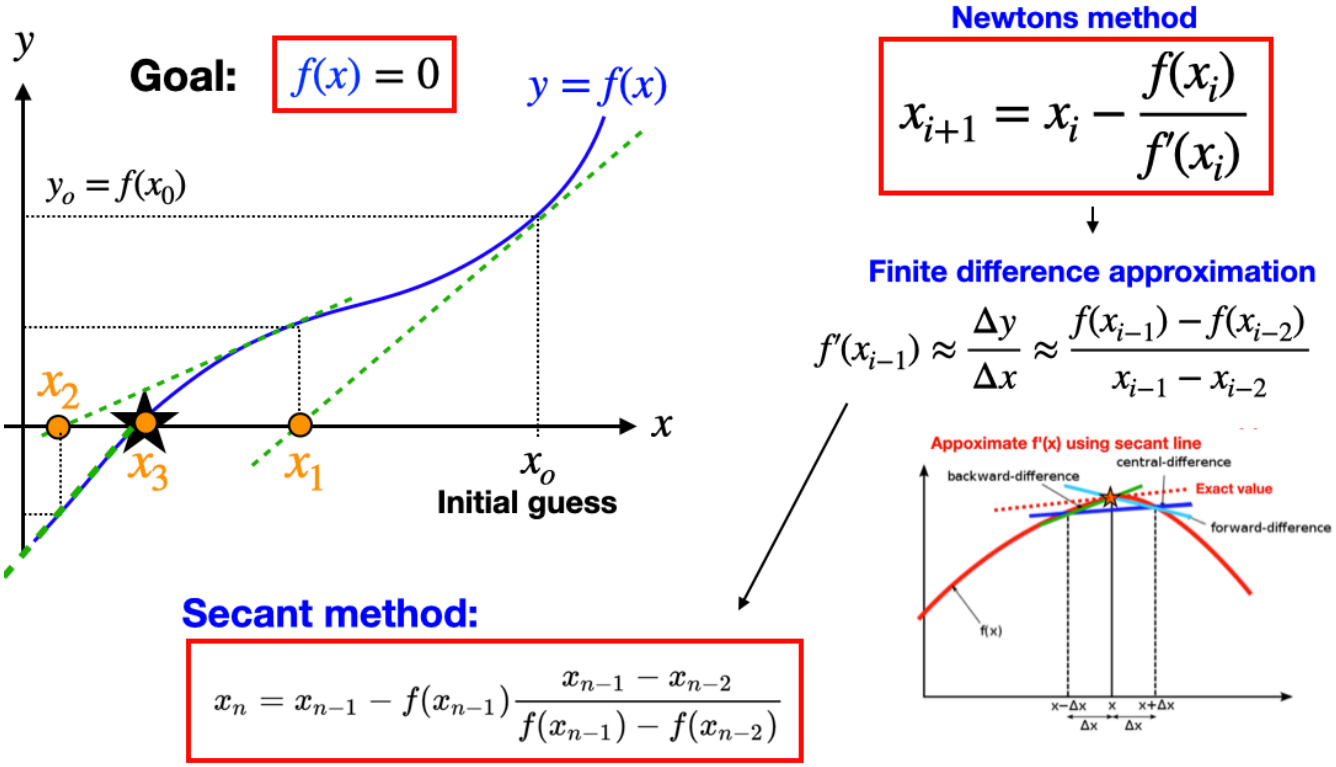
- By repeating the process we eventually converge to the solution



Solver: Secant method

- We can combine Newton's method with the finite difference formula to remove ANY need for pen-and-paper work

- This is known as the “secant method”

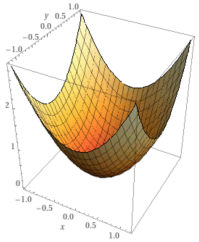


DOESN’T requires explicit functional form of the derivative of f(x)

Numerical Optimization: Uni-variate

Example: Analytic optimization

- Find the value of $\mathbf{x}_0$ that minimizes $f = f(\mathbf{x}) = f(x, y) = 5 + (x - 10)^2 + y^2$.



- This is the functional form of the loss function for single variable linear regression.
- It’s such a simple function, we can infer the solution $\mathbf{x}_0 = (x_0, y_0) = (10, 0) \rightarrow f = 5$
- However, It can easily be proven using the condition $\nabla f(\mathbf{x}) = \mathbf{0} = (0, 0)$
  - $0 = \frac{\partial f}{\partial x} = 2(x - 10) \rightarrow x = 10 \qquad 0 = \frac{\partial f}{\partial y} = 2y \rightarrow y = 0$
- **This is a simple example, but the process is general**
  - (1): Compute the gradient and set it equal to zero
  - (2): Results in a system of N-equation and N-unknowns: $f_x(\mathbf{x}) = 0,\ f_y(\mathbf{x}) = 0 \ldots$
  - (3): `Solve` this system of equations to find the $\mathbf{x}_0$ (roots) which satisfies it

Newton’s method for optimization

- If the function only has a single-variable. Then the derivative is a function (not a vector of partial derivatives)

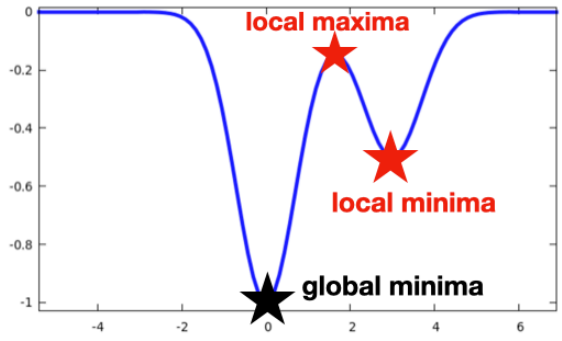
$$\text{Newton's method for solving f(x)=0: } x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

- In this case, optimization means finding the roots of the the derivative. We can just apply Newton's method to the derivative. We just get another differentiation in the formula

$$\text{Newton's method optimization f '(x)=0: } x_{n+1} = x_n - \frac{f'(x_n)}{f''(x_n)}$$

Numerical Optimizers:
Goal: find x that minimizes f(x)
f(x) —> Objective function

$$f'(x) = \frac{df(x)}{dx} = 0$$



Gradient descent for optimization

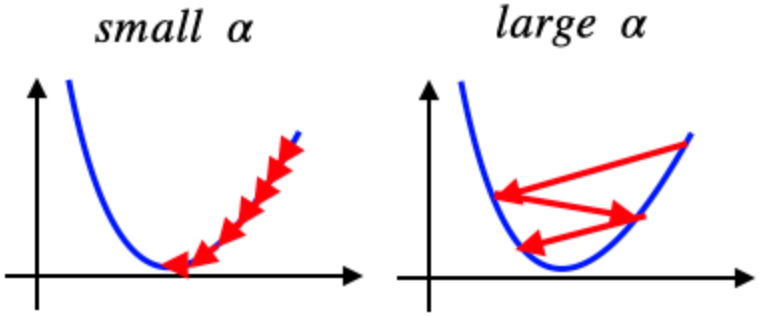
- For high dimensions, computing the second derivative (Hessian matrix) isn't realistic

$$\text{Newton's method optimization: f '(x)=0 } x_{n+1} = x_n - \frac{f'(x_n)}{f''(x_n)}$$

- Therefore, it is common to simplify Newton's method for optimization.
- This is done by replacing the second derivative, with as hyper-parameter α, known as the learning rate (this simplification is known as gradient descent)

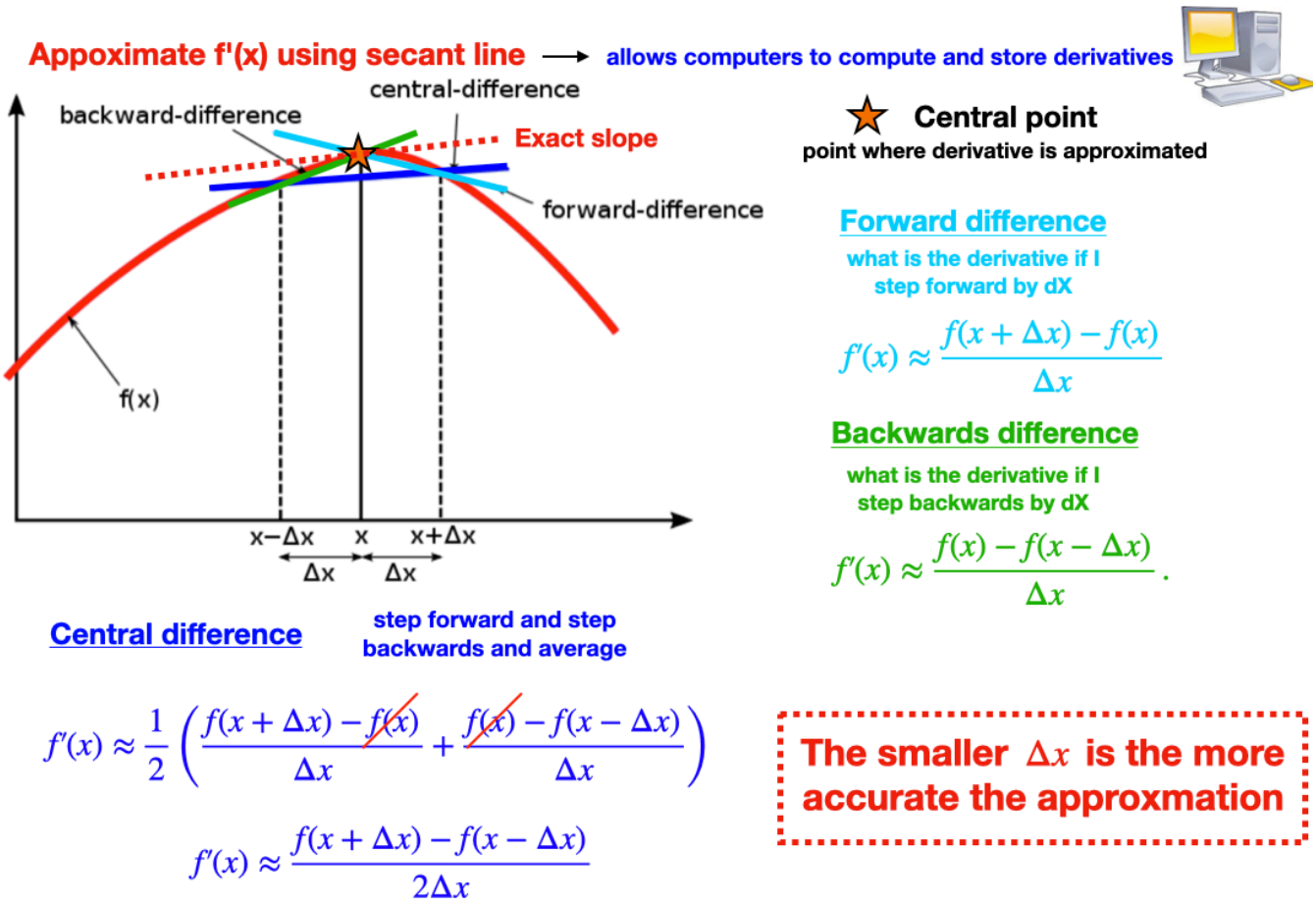
$$\text{Gradient descent optimization: f '(x)=0 } x_{n+1} = x_n - \alpha\, f'(x_n)$$

- Tuning α allows us to control how big our "steps" and therefore convergence rates.



Numerical derivative calculation

- One major downside of Newton's method is the use of the analytic derivative
- We can get around this by approximating the derivative via the "finite difference" approximation of the derivative
- With finite difference, we approximate the derivative via the secant line. (Back-propagation is a much more efficient option)



Numerical Optimization: Multi-variate

The previous single variable discussion generalizes to the multi variable case

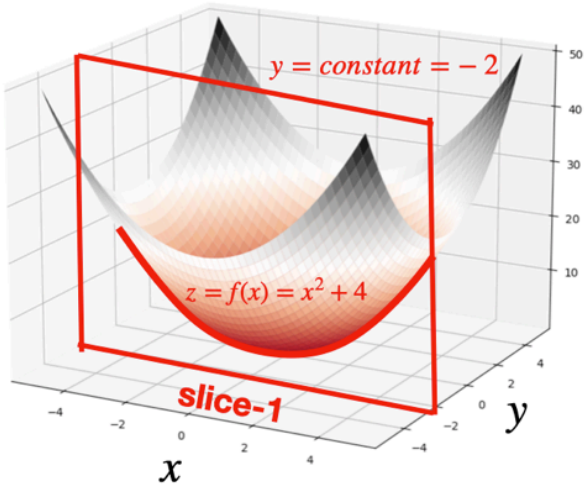
Numeric partial derivatives

- We can approximate partial differences using the find a different method.

- This requires a loop over each variable, perturbing one at a time.

Scalar function of multiple variables

$$\mathbb{R}^2 \to \mathbb{R}^1 \quad z = f(\mathbf{x}) = f(x, y)$$



Approximate components of gradient

First derivatives: $N_{eval} = 2N_{dim}$

$$f_x(x, y) \approx \frac{f(x + h, y) - f(x - h, y)}{2h}$$

$$f_y(x, y) \approx \frac{f(x, y + k) - f(x, y - k)}{2k}$$

Second derivatives:

$$f_{xx}(x, y) \approx \frac{f(x + h, y) - 2f(x, y) + f(x - h, y)}{h^2} \qquad f_{yy}(x, y) = \frac{\partial^2 f}{\partial y^2}$$

$$f_{yy}(x, y) \approx \frac{f(x, y + k) - 2f(x, y) + f(x, y - k)}{k^2} \qquad f_{xx}(x, y) = \frac{\partial^2 f}{\partial x^2}$$

$$f_{xy}(x, y) \approx \frac{f(x + h, y + k) - f(x + h, y - k) - f(x - h, y + k) + f(x - h, y - k)}{4hk}.$$

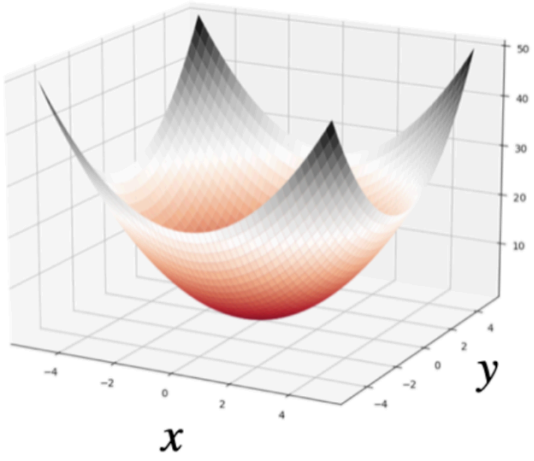
Solver: Multi-variable case

- Consider a system of scalar equations with multiple variables, for example $\mathbf{x} = (x, y)$.
- This system of equations can be thought of as a vector field $\mathbf{f}(\mathbf{x}) = (\, f_1(x, y), f_2(x, y)\,)$
- The goal is to find the special vector $\tilde{\mathbf{x}}$ for which all equations are **simultaneously** zero, this means $f_1(\tilde{x}, \tilde{y}) = 0$ and $f_2(\tilde{x}, \tilde{y}) = 0$
- As seen before, this is the mathematical formalism used for **numerical optimization**
- **IMPORTANT:** "Training" in supervised learning is typically equivalent to this problem
- Why is this true?
  - Because to "train" we need to define a loss function (which is a scalar function)
  - We compute can nu gradient (vector field)
  - Minima correspond to the roots of the gradient ($\nabla f(\mathbf{x}) = \mathbf{0}$)

Example-1

- The example we saw before is probably the simplest possible example

$$z = f(x, y) = x^2 + y^2$$

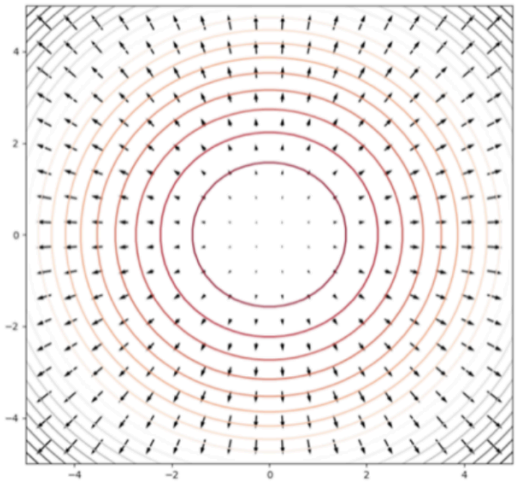


Partial derivatives can be calculated
for any arbitrary x and y points

$$f_x(x, y) = \frac{\partial f}{\partial x} = \frac{\partial}{\partial x}\left(x^2 + y^2\right) = 2x + 0 = 2x$$

$$f_y(x, y) = \frac{\partial f}{\partial y} = \frac{\partial}{\partial y}\left(x^2 + y^2\right) = 0 + 2y = 2y$$

(Gradient Vector field)



Gradient —> vector of partial derivatives:

$$\nabla f(x, y) = \left(f_x(x, y), f_y(x, y)\right) = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}\right) = (2x, 2y)$$

Multi-variable newtons method: (solver)

- We want to find the solutions to systems of equations $\mathbf{F}(\mathbf{x}) = \mathbf{0}$
- Which is EXACTLY what we need for optimization ($\nabla f = \mathbf{0}$)
- We have N-equations and N-unknowns.

$$f_1(x_1, \cdots, x_N) = f_1(\mathbf{x}) = 0$$
$$f_2(x_1, \cdots, x_N) = f_2(\mathbf{x}) = 0$$
$$\cdots\cdots\cdots\cdots$$
$$f_N(x_1, \cdots, x_N) = f_N(\mathbf{x}) = 0$$

- We can derive Newton's method as

Express f(x) as a multivariable Taylor series to first order:

$$f_i(\mathbf{x}+\delta\mathbf{x}) = f_i(\mathbf{x}) + \sum_{j=1}^{N} \frac{\partial f_i(\mathbf{x})}{\partial x_j}\delta x_j + O(\delta\mathbf{x}^2) \approx f_i(\mathbf{x}) + \sum_{j=1}^{N} \frac{\partial f_i(\mathbf{x})}{\partial x_j}\delta x_j, \qquad (i = 1, \cdots, N)$$

Or in matrix form:

$$\underline{\mathbf{f}(\mathbf{x}+\delta\mathbf{x})} = \begin{bmatrix} f_1(\mathbf{x}+\delta\mathbf{x}) \\ \vdots \\ f_N(\mathbf{x}+\delta\mathbf{x}) \end{bmatrix} \approx \begin{bmatrix} f_1(\mathbf{x}) \\ \vdots \\ f_N(\mathbf{x}) \end{bmatrix} + \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_N} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_N}{\partial x_1} & \cdots & \frac{\partial f_N}{\partial x_N} \end{bmatrix} \begin{bmatrix} \delta x_1 \\ \vdots \\ \delta x_N \end{bmatrix}$$

$$\mathbf{f}(\mathbf{x}) + \mathbf{J}_\mathbf{f}(\mathbf{x})\,\delta\mathbf{x} \qquad \text{where } \mathbf{J}_\mathbf{f}(\mathbf{x}) \text{ is an } N \times N \textit{ Jacobian matrix}$$

Solve iteratively:

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \delta\mathbf{x}_n \rightarrow \underline{\mathbf{f}(\mathbf{x}+\delta\mathbf{x}) =} \; \mathbf{f}(\mathbf{x}) + \mathbf{J}_\mathbf{f}(\mathbf{x})\,\delta\mathbf{x} \rightarrow \boxed{\mathbf{x}_{n+1} = \mathbf{x}_n - \mathbf{J}_n^{-}\,\mathbf{f}(\mathbf{x}_n)}$$

set equal to zero

source: https://people.duke.edu/~kh269/teaching/b553/newtons_method.pdf

Multi-variable newtons method: (optimization)

- We can apply this method to the gradient $\nabla f(\mathbf{x}) = \mathbf{0}$ of a function to find the minima and maxima
- The Jacobian (matrix of 1st deriv) becomes the Hessian (matrix of 2nd deriv).

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \mathbf{H}^{-1}(\mathbf{x}_n)\nabla F(\mathbf{x}_n)$$

Hessian: (Matrix of second partial derivatives) $(\mathbf{H}_f)_{i,j} = \dfrac{\partial^2 f}{\partial x_i \, \partial x_j}$

$$H(x) = \nabla^2 f(x) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2}(x) & \frac{\partial^2 f}{\partial x_1 \partial x_2}(x) & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n}(x) \\ \frac{\partial^2 f}{\partial x_2 \partial x_1}(x) & \frac{\partial^2 f}{\partial x_2^2}(x) & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n}(x) \\ \vdots & \vdots & & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1}(x) & \frac{\partial^2 f}{\partial x_n \partial x_2}(x) & \cdots & \frac{\partial^2 f}{\partial x_n^2}(x) \end{bmatrix}$$

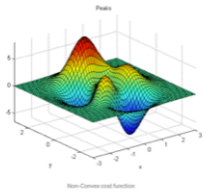
- This formula can be simplified by assuming the Hessian is constant (Gradient descent)

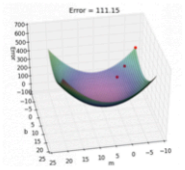
$$\text{Gradient descent optimization: } \nabla f(\mathbf{x}) = \mathbf{0} \quad \mathbf{x}_{n+1} = \mathbf{x}_n - \lambda \nabla F(\mathbf{x}_n)$$

Types of optimizers

OPTIMIZERS

(Iterative methods)





Non-gradient based

- Don't require derivatives
- Rely on function evaluation to search

- Mayfly Optimization Algorithm[4]
- Memetic algorithm
- Differential evolution
- Evolutionary algorithms
- Dynamic relaxation
- Genetic algorithms
- Hill climbing with random restart
- Nelder–Mead simplicial heuristic: A popular heuristic for approximate minimization (without calling gradients)
- Particle swarm optimization
- Gravitational search algorithm
- Simulated annealing
- Stochastic tunneling
- Tabu search
- Reactive Search Optimization (RSO)[5] implemented in LIONsolver
- Forest Optimization Algorithm

Mastering optimization is an entire career, for 512 we just need to understand the fundamentals

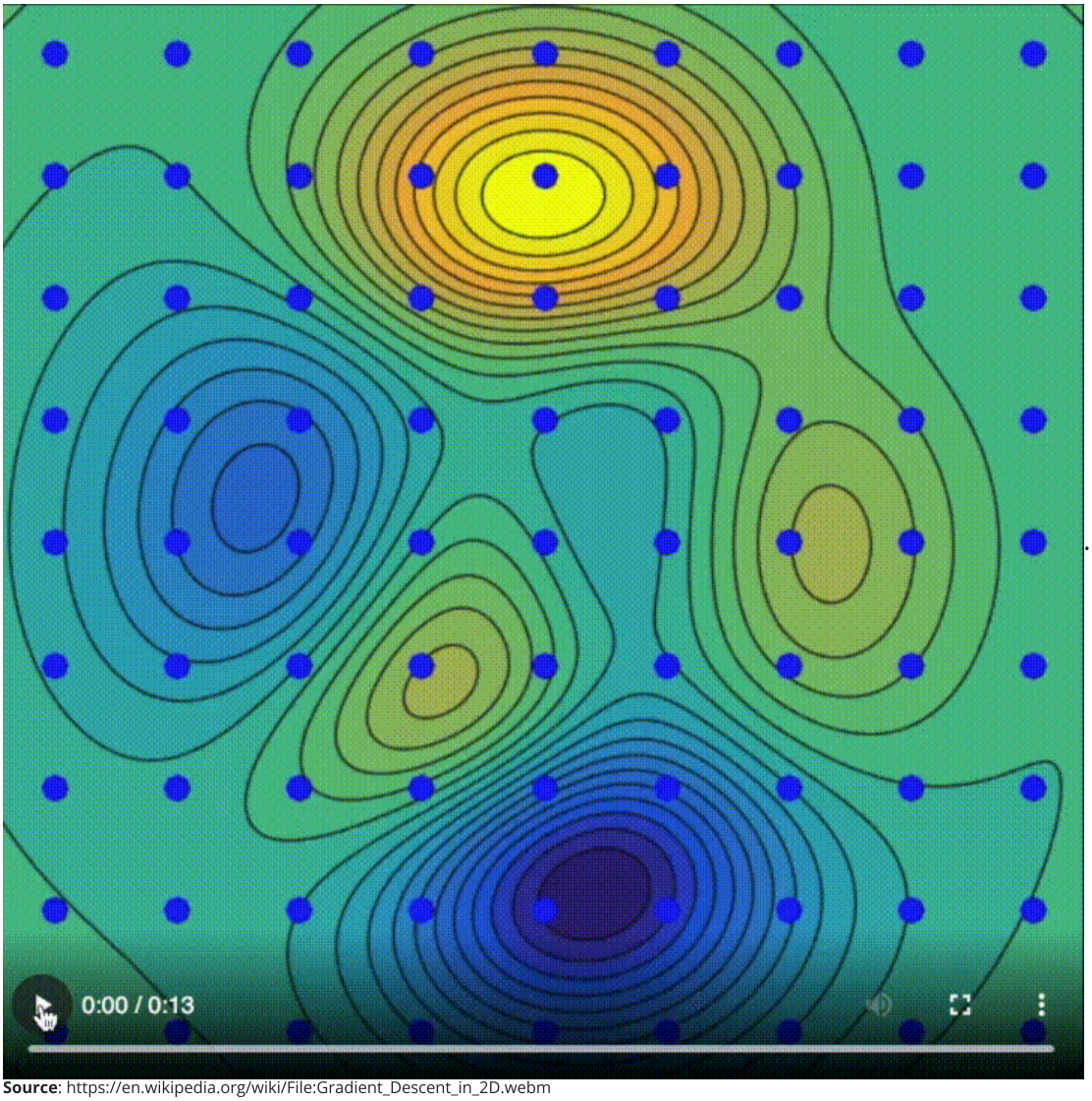
Gradient based methods

- **Often "Quasi" Newton methods**
  - i.e modified forms of Newtons method
- **Require knowledge of first (or second) derivatives**
  - Either provided explicitly
  - or approximated (e.g. via finite difference)

- Methods that evaluate Hessians (or approximate Hessians, using finite differences):
  - Newton's method
  - Sequential quadratic programming: A Newton-based method for small-medium scale *constrained* problems. Some versions can handle large-dimensional problems.
  - Interior point methods: This is a large class of methods for constrained optimization. Some interior-point methods use only (sub)gradient information and others of which require the evaluation of Hessians.
- Methods that evaluate gradients, or approximate gradients in some way (or even subgradients):
  - Coordinate descent methods: Algorithms which update a single coordinate in each iteration
  - Conjugate gradient methods: Iterative methods for large problems. (In theory, these methods terminate in a finite number of steps with quadratic objective functions, but this finite termination is not observed in practice on finite–precision computers.)
  - Gradient descent (alternatively, "steepest descent" or "steepest ascent"): A (slow) method of historical and theoretical interest, which has had renewed interest for finding approximate solutions of enormous problems.
  - Subgradient methods: An iterative method for large locally Lipschitz functions using generalized gradients. Following Boris T. Polyak, subgradient–projection methods are similar to conjugate–gradient methods.
  - Bundle method of descent: An iterative method for small–medium-sized problems with locally Lipschitz functions, particularly for convex minimization problems (similar to conjugate gradient methods).
  - Ellipsoid method: An iterative method for small problems with quasiconvex objective functions and of great theoretical interest, particularly in establishing the polynomial time complexity of some combinatorial optimization problems. It has similarities with Quasi-Newton methods.
  - Conditional gradient method (Frank–Wolfe) for approximate minimization of specially structured problems with linear constraints, especially with traffic networks. For general unconstrained problems, this method reduces to the gradient method, which is regarded as obsolete (for almost all problems).
  - Quasi-Newton methods: Iterative methods for medium-large problems (e.g. N<1000).
  - Simultaneous perturbation stochastic approximation (SPSA) method for stochastic optimization; uses random (efficient) gradient approximation.

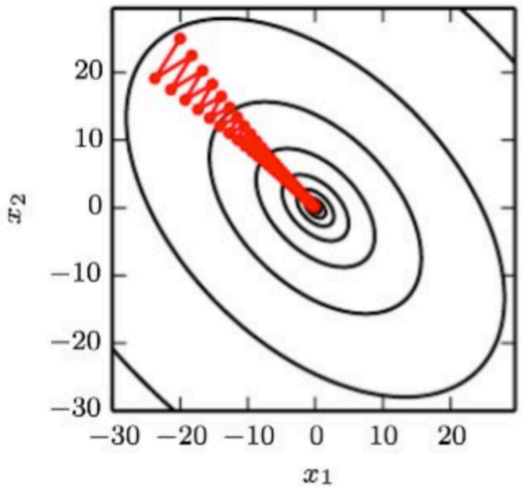
https://en.wikipedia.org/wiki/Mathematical_optimization

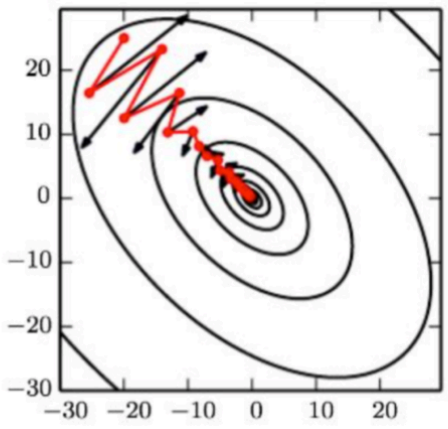
Example



Source: https://en.wikipedia.org/wiki/File:Gradient_Descent_in_2D.webm

Momentum methods





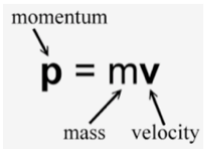
If curvature is stronger in some dimensions than others, the gradient is dominated by one direction, leading to oscillatory behavior. This is wasteful and also causes us to move **very** slowly in the other dimensions.

Methods that incorporate **momentum** seek to use not just the instantaneous gradient, but also the past gradients of previous locations visited.

- Keep track of a velocity vector that indicates the direction you were recently moving
  - Over time, the velocity vector sums up all the previous gradients
- Now the update is not simply a step in the direction down the gradient
- The update is a weighted sum of the gradient and the previous direction of motion

Momentum methods increase efficiency by mitigating this Oscillatory behavior

Physics:



The Momentum algorithm

SGD has trouble navigating ravines

Image 2: SGD without momentum

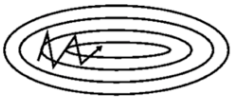


Image 3: SGD with momentum

Gradient Decent (GD or SDG)

initalize: $n = 1 \rightarrow \theta_1 = random$

(change in parameters) $\Delta\theta_n = \eta \nabla L(\theta_n)$

(updated parameters) $\theta_{n+1} = \theta_n - \Delta\theta_n = \theta_n - \eta \nabla L(\theta_n)$

$L(\theta)$ = **loss function**

n = **optimizer iteration**

η = **learning rate** (hyper parameter)

θ = **model parameters**

Momentum: (GD with momentum)

initalize: $\theta_1 = random \quad \Delta\theta_0 = 0$

(change in parameters) $\Delta\theta_n = \eta \nabla L(\theta_n) + \alpha \Delta\theta_{n-1}$

(determines strength of contributionsfrom previous iterations)

α = **exponential decay factor** $\in [0 : 1)$ (hyper parameter)

(updated parameters) $\theta_{n+1} = \theta_n - \Delta\theta_n = \theta_n - \eta \nabla L(\theta_n) - \alpha \Delta\theta_{n-1}$

Example: $\theta_1 = random \quad \Delta\theta_0 = 0$

(updated parameters)

$$n = 1 \quad \Delta\theta_1 = \eta \nabla L(\theta_1) + \alpha \Delta\theta_0 = \eta \nabla L(\theta_1)$$ first step is regular GD $\quad \theta_2 = \theta_1 - \Delta\theta_1$

$$n = 2 \quad \Delta\theta_2 = \eta \nabla L(\theta_2) + \alpha \Delta\theta_1$$
$$= \eta \nabla L(\theta_2) + \alpha \left(\eta \nabla L(\theta_1) \right)$$ $\quad \theta_3 = \theta_2 - \Delta\theta_2$

$$n = 3 \quad \Delta\theta_3 = \eta \nabla L(\theta_3) + \alpha \Delta\theta_2$$
$$= \eta \nabla L(\theta_3) + \alpha \left(\eta \nabla L(\theta_2) + \alpha \left(\eta \nabla L(\theta_1) \right) \right)$$
$$= \eta \nabla L(\theta_3) + \alpha \left(\eta \nabla L(\theta_2) \right) + \alpha^2 \left(\eta \nabla L(\theta_1) \right)$$ $\quad \theta_4 = \theta_3 - \Delta\theta_3$

$$n = 4 \quad \Delta\theta_4 = \eta \nabla L(\theta_4) + \alpha \Delta\theta_3$$
$$= \eta \nabla L(\theta_4) + \alpha \left(\eta \nabla L(\theta_3) \right) + \alpha^2 \left(\eta \nabla L(\theta_2) \right) + \alpha^3 \left(\eta \nabla L(\theta_1) \right)$$ $\quad \theta_5 = \theta_4 - \Delta\theta_4$

$\vdots$

(exponential decay of past terms)

Adaptive methods

For now, we will focus on gradient descent and momentum. But many newer methods are used in practice, and these methods build upon the same principles but with adaptive learning rates and other innovations.

- AdaGrad
- RMSProp
- Adam

Recommended reading
https://ruder.io/optimizing-gradient-descent/index.html#gradientdescentoptimizationalgorithms

You can read about them in section 8.5 of Goodfellow to gain familiarity. We won't dive into their details, but you might leverage them when using Keras or Tensorflow.

Adam

Adaptive Moment Estimation (Adam)

Given parameters $w^{(t)}$ and a loss function $L^{(t)}$
t indexes the current training iteration

$$m_w^{(t+1)} \leftarrow \beta_1 m_w^{(t)} + (1-\beta_1)\nabla_w L^{(t)}$$
$$v_w^{(t+1)} \leftarrow \beta_2 v_w^{(t)} + (1-\beta_2)(\nabla_w L^{(t)})^2$$
$$\hat{m}_w = \frac{m_w^{(t+1)}}{1-\beta_1}$$
$$\hat{v}_w = \frac{v_w^{(t+1)}}{1-\beta_2}$$
$$w^{(t+1)} \leftarrow w^{(t)} - \eta \frac{\hat{m}_w}{\sqrt{\hat{v}_w} + \epsilon}$$

ϵ is a small scalar (e.g. 10^{-8}) used to prevent division by 0,
β_1 (e.g. 0.9) and β_2 (e.g. 0.999) are the forgetting factors

RMSprop

$$E[g^2]_t = 0.9E[g^2]_{t-1} + 0.1g_t^2$$
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

RMSprop as well divides the learning rate by an exponentially decaying average of squared gradients. Hinton suggests γ to be set to 0.9, while a good default value for the learning rate η is 0.001.

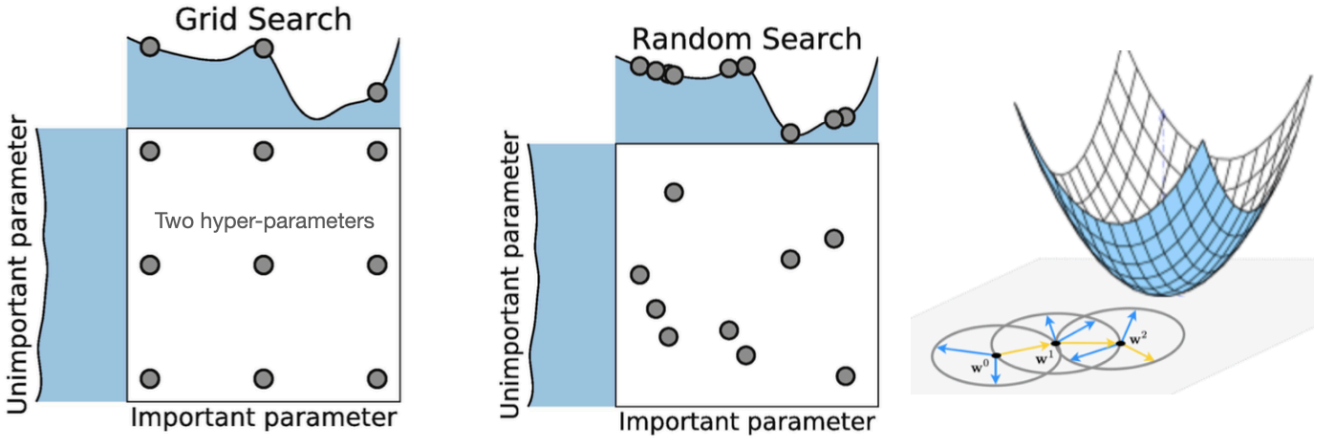
$$v(w,t) := \underbrace{\gamma}_{\text{forgetting factor}} v(w,t-1) + (1-\gamma)(\nabla Q_i(w))^2$$
$$w := w - \frac{\eta}{\sqrt{v(w,t)}} \nabla Q_i(w)$$

Non-Gradient Numerical optimization

Grid and random search

- Random search (RS) is the simplest non-gradient based optimization method

- It is commonly used for automated hyper-parameter tuning of for modeling.



Hyperparameters vector: $\mathbf{H} = (H_1, H_2 \ldots, N_N)$ min(validation loss) $L_v = L_v(\mathbf{H})$

Algorithm:

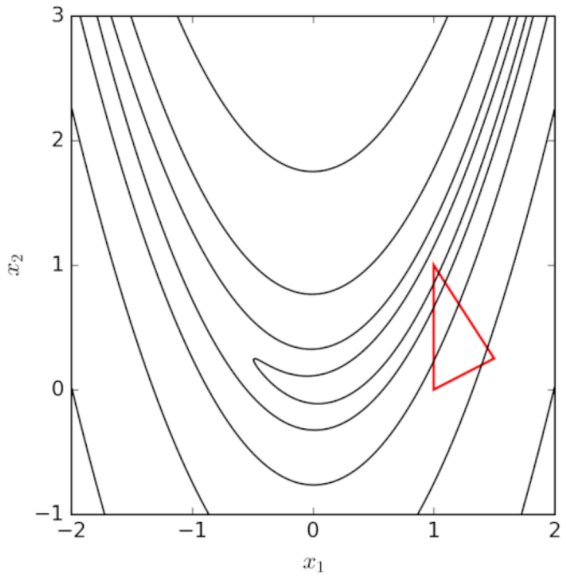
Let f: $\mathbb{R}^n \to \mathbb{R}$ be the fitness or cost function which must be minimized. Let $\mathbf{x} \in \mathbb{R}^n$ designate a position or candidate solution in the search-space. The basic RS algorithm can then be described as:

1. Initialize **x** with a random position in the search-space.
2. Until a termination criterion is met (e.g. number of iterations performed, or adequate fitness reached), repeat the following:
    1. Sample a new position **y** from the hypersphere of a given radius surrounding the current position **x** (see e.g. Marsaglia's technique for sampling a hypersphere.)
    2. If $f(\mathbf{y}) < f(\mathbf{x})$ then move to the new position by setting $\mathbf{x} = \mathbf{y}$

Source: https://en.wikipedia.org/wiki/Random_search

Nelder-Mead

- Nelder-Mead is probably the most famous **non-gradient** based optimization method.
- It is a direct search method (based on function comparison not differentiation)
- It uses a geometric construction in the higher dimensional space known as a simplex:
- e.g. line segments in 1D, a triangle in 2D, a tetrahedron in 3D space, ... etc



- Also known as the downhill simplex method or amoeba method

- **One major downside is that it can converge to non-stationary points**

Algorithm

- This geometric construction does a “walk” around the objective function surface.
- As it “walks”, it also shrinks, so the prediction of the minima get more localized
- An iteration of the Nelder-Mead method over two-dimensional space.

We are trying to minimize the function $f(\mathbf{x})$, where $\mathbf{x} \in \mathbb{R}^n$. Our current test points are $\mathbf{x}_1, \ldots, \mathbf{x}_{n+1}$.

1. Order according to the values at the vertices:

$f(\mathbf{x}_1) \leq f(\mathbf{x}_2) \leq \cdots \leq f(\mathbf{x}_{n+1})$.

Check whether method should stop. See **Termination** below. Sometimes inappropriately called "convergence".

2. Calculate $\mathbf{x}_o$, the centroid of all points except $\mathbf{x}_{n+1}$.

3. Reflection

Compute reflected point $\mathbf{x}_r = \mathbf{x}_o + \alpha(\mathbf{x}_o - \mathbf{x}_{n+1})$ with $\alpha > 0$.

If the reflected point is better than the second worst, but not better than the best, i.e. $f(\mathbf{x}_1) \leq f(\mathbf{x}_r) < f(\mathbf{x}_n)$,

then obtain a new simplex by replacing the worst point $\mathbf{x}_{n+1}$ with the reflected point $\mathbf{x}_r$, and go to step 1.

4. Expansion

If the reflected point is the best point so far, $f(\mathbf{x}_r) < f(\mathbf{x}_1)$,

then compute the expanded point $\mathbf{x}_e = \mathbf{x}_o + \gamma(\mathbf{x}_r - \mathbf{x}_o)$ with $\gamma > 1$.

If the expanded point is better than the reflected point, $f(\mathbf{x}_e) < f(\mathbf{x}_r)$,

then obtain a new simplex by replacing the worst point $\mathbf{x}_{n+1}$ with the expanded point $\mathbf{x}_e$ and go to step 1;

else obtain a new simplex by replacing the worst point $\mathbf{x}_{n+1}$ with the reflected point $\mathbf{x}_r$ and go to step 1.

5. Contraction

Here it is certain that $f(\mathbf{x}_r) \geq f(\mathbf{x}_n)$. (Note that $\mathbf{x}_n$ is second or "next" to highest.)

Compute contracted point $\mathbf{x}_c = \mathbf{x}_o + \rho(\mathbf{x}_{n+1} - \mathbf{x}_o)$ with $0 < \rho \leq 0.5$.

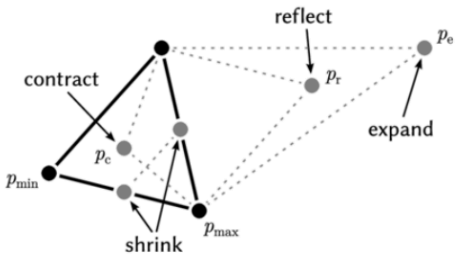
If the contracted point is better than the worst point, i.e. $f(\mathbf{x}_c) < f(\mathbf{x}_{n+1})$,

then obtain a new simplex by replacing the worst point $\mathbf{x}_{n+1}$ with the contracted point $\mathbf{x}_c$ and go to step 1;

6. Shrink

Replace all points except the best ($\mathbf{x}_1$) with

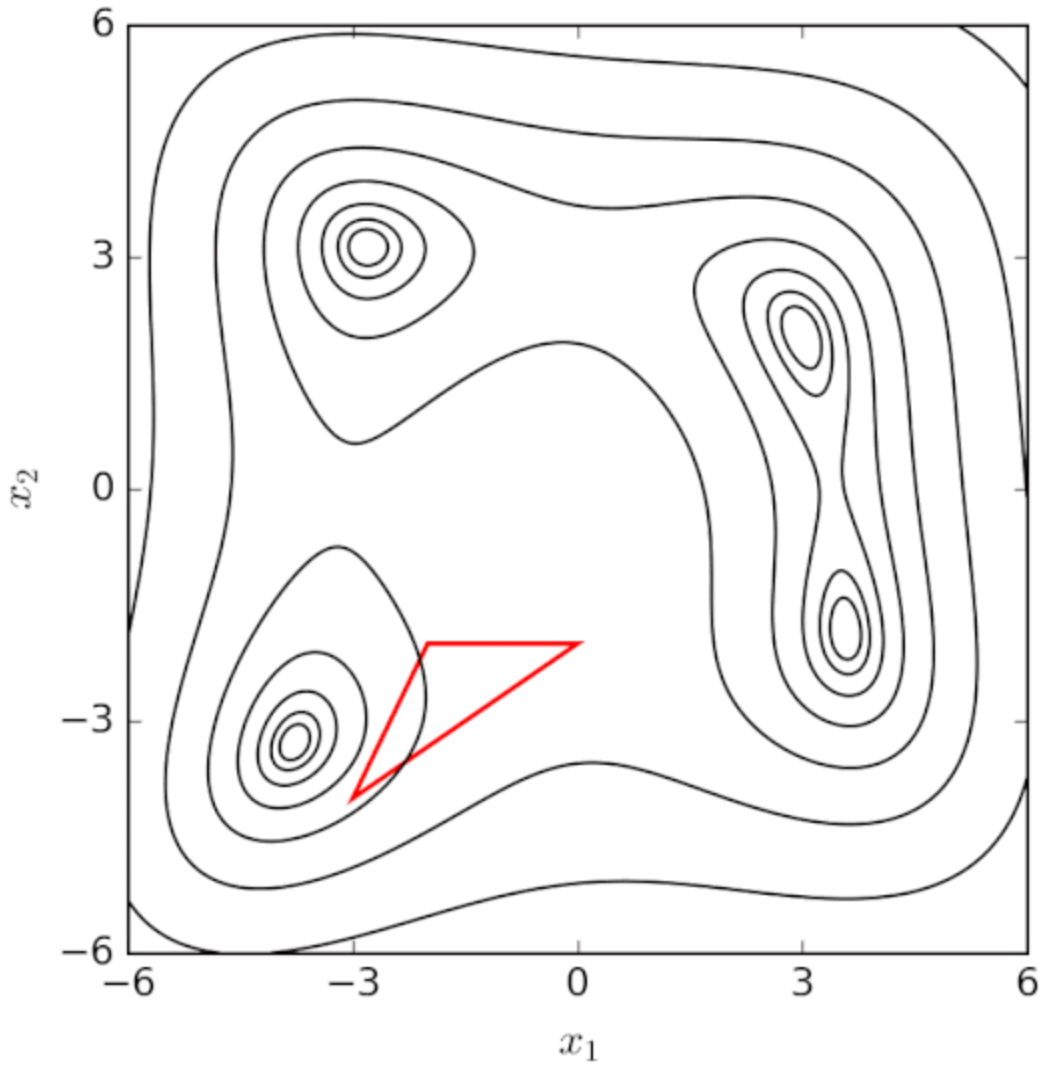
$\mathbf{x}_i = \mathbf{x}_1 + \sigma(\mathbf{x}_i - \mathbf{x}_1)$ and go to step 1.



https://en.wikipedia.org/wiki/Nelder%E2%80%93Mead_method

Example

- The following movie shows the algorithm in action for a low-dimensional case



Particle swarm

- The concept is loosely inspired by the coordinated movement of flocks of birds or fish

- There is a set of candidate solutions (particles), which move around the search-space.

- The movements are governed by mathematical formula controlling each particle's position and velocity.
- The motion is influenced by (1) the particles local best known position (2) the current best known positions in the search-space (as observed by all particles in the swarm)
- The dynamics of the system tend to move the majority of swarm toward a locally optimal solution.

Source: https://en.wikipedia.org/wiki/Particle_swarm_optimization

Algorithm

- Particles are "attracted" to minima, and governed by laws akin to Newtonian physics
- Here is some pseudo-code for the algorithm implementation.
- This is outside the scope of this course, but is provided for completeness
- Here is some pseudo-code for the algorithm implementation.
- S=number of particles, each with position and velocity $\mathbf{x}_i$,$\mathbf{v}_i$. The variable $\mathbf{p}_i$ is the best known position of particle i. Finally, $\mathbf{g}$ be the best *global* position of the swarm.

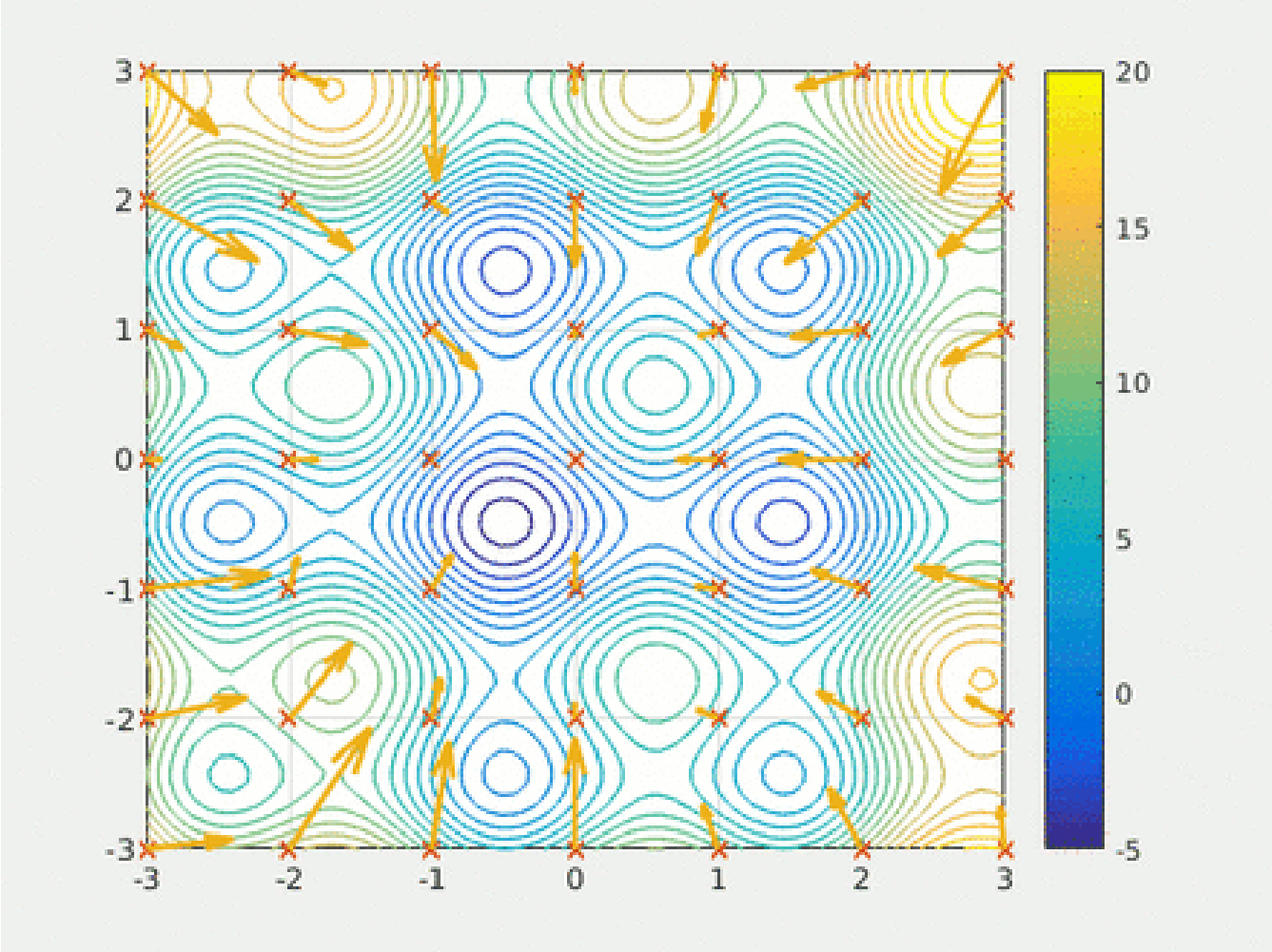
```
for each particle i = 1, ..., S do
    Initialize the particle's position with a uniformly distributed random vector: x_i ~ U(b_lo, b_up)
    Initialize the particle's best known position to its initial position: p_i ← x_i
    if f(p_i) < f(g) then
        update the swarm's best known position: g ← p_i
    Initialize the particle's velocity: v_i ~ U(-|b_up-b_lo|, |b_up-b_lo|)
while a termination criterion is not met do:
    for each particle i = 1, ..., S do
        for each dimension d = 1, ..., n do
            Pick random numbers: r_p, r_g ~ U(0,1)
            Update the particle's velocity: v_i,d ← w v_i,d + φ_p r_p (p_i,d-x_i,d) + φ_g r_g (g_d-x_i,d)
        Update the particle's position: x_i ← x_i + v_i
        if f(x_i) < f(p_i) then
            Update the particle's best known position: p_i ← x_i
            if f(p_i) < f(g) then
                Update the swarm's best known position: g ← p_i
```

- Parameters: w "inertia", while ϕ_p and ϕ_g are the cognitive and social coefficients.

Source: https://en.wikipedia.org/wiki/Particle_swarm_optimization

Example

- A particle swarm searching for the global minimum of an objective function



Source: https://en.wikipedia.org/wiki/Particle_swarm_optimization

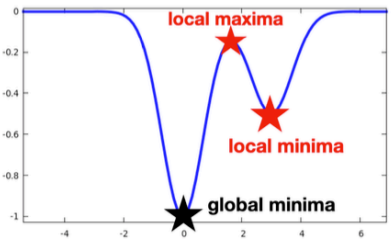
Numerical Optimization: Summary

Single variable case

$$f'(x) = \frac{df(x)}{dx} = 0$$

Newton's method for optimization $x_{n+1} = x_n - \frac{f'(x_n)}{f''(x_n)}$

Gradient decent → $x_{n+1} = x_n - \lambda f'(x_n)$



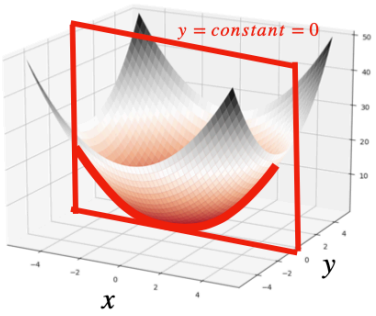
Mult-variable case

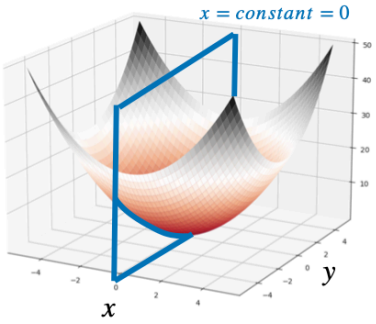
$\mathbb{R}^N \rightarrow \mathbb{R}^1 \quad y = f(\mathbf{x}) = f(x_1, x_2, \ldots, x_N)$

$$\nabla f(\mathbf{x}) = \left[\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2} \ldots \frac{\partial f}{\partial x_N}\right] = \mathbf{0}$$

⟶ find roots for system of equations $\mathbf{f}'(\mathbf{x}) = 0$

$f_x(x,y) = \frac{\partial f}{\partial x}$ $f_y(x,y) = \frac{\partial f}{\partial y}$





Minima at (x,y)=(0,0) $z = f(x,y) = x^2 + y^2$

Netwons Method

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \mathbf{H}^{-1}(\mathbf{x}_n)\nabla F(\mathbf{x}_n)$$

Hessian: (Matrix of second partial derivatives) $(\mathbf{H}_f)_{i,j} = \frac{\partial^2 f}{\partial x_i \partial x_j}$

$$H(x) = \nabla^2 f(x) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2}(x) & \frac{\partial^2 f}{\partial x_1 \partial x_2}(x) & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n}(x) \\ \frac{\partial^2 f}{\partial x_2 \partial x_1}(x) & \frac{\partial^2 f}{\partial x_2^2}(x) & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n}(x) \\ \vdots & \vdots & & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1}(x) & \frac{\partial^2 f}{\partial x_n \partial x_2}(x) & \cdots & \frac{\partial^2 f}{\partial x_n^2}(x) \end{bmatrix}$$

Gradient Decent

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \lambda \nabla F(\mathbf{x}_n)$$

Footnotes

1. In this case a numerical solver will not be able to converge. Fortunately, for most well posed Machine learning problems there is a well defined point representing an acceptable solution. ↩