OPTIMIZATION TECHNIQUES

Derivative-based optimization

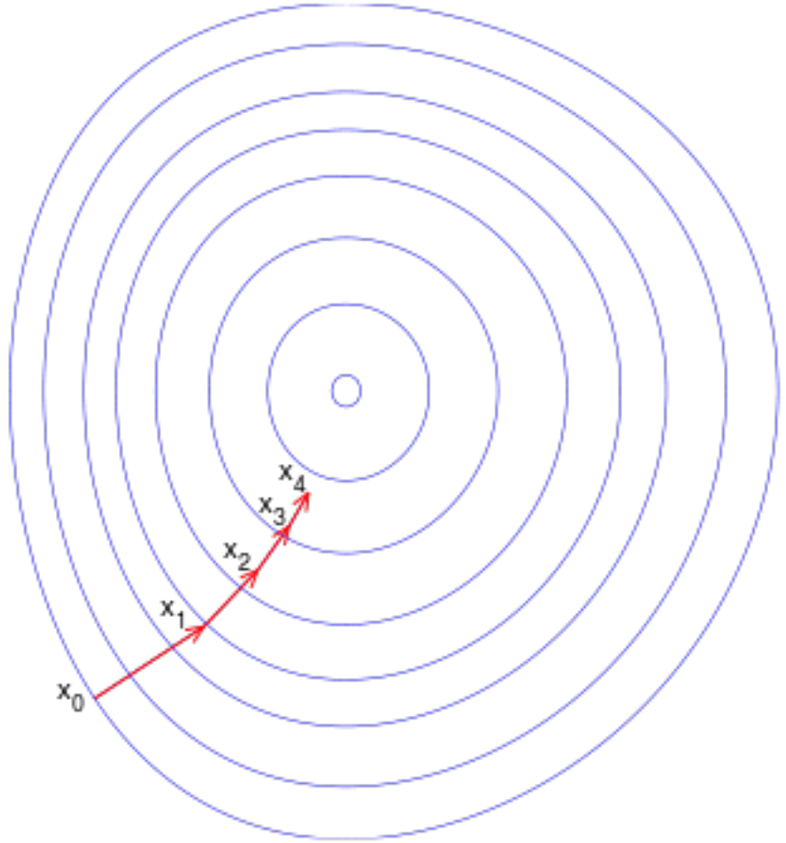
Prof. Giovanni Iacca

giovanni.iacca@unitn.it

UNIVERSITY OF TRENTO - Italy

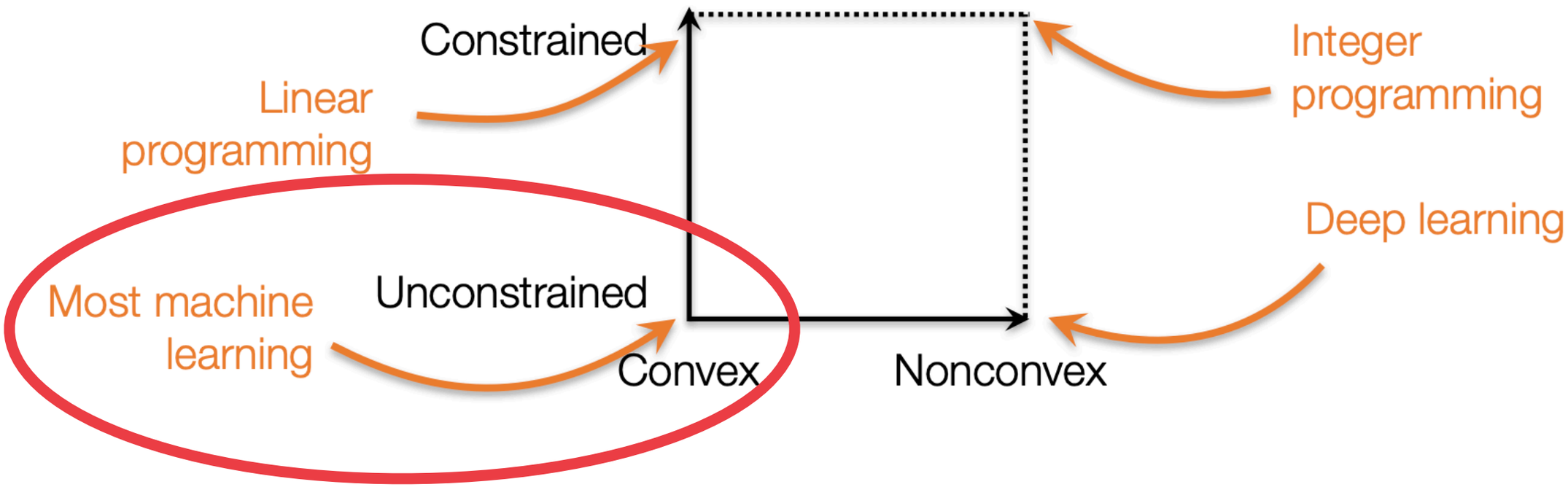
Information Engineering
and Computer Science Department

Derivative-based optimization



OPTIMIZATION

ONE WORD FOR SEVERAL MEANINGS



GRADIENT-BASED OPTIMIZATION

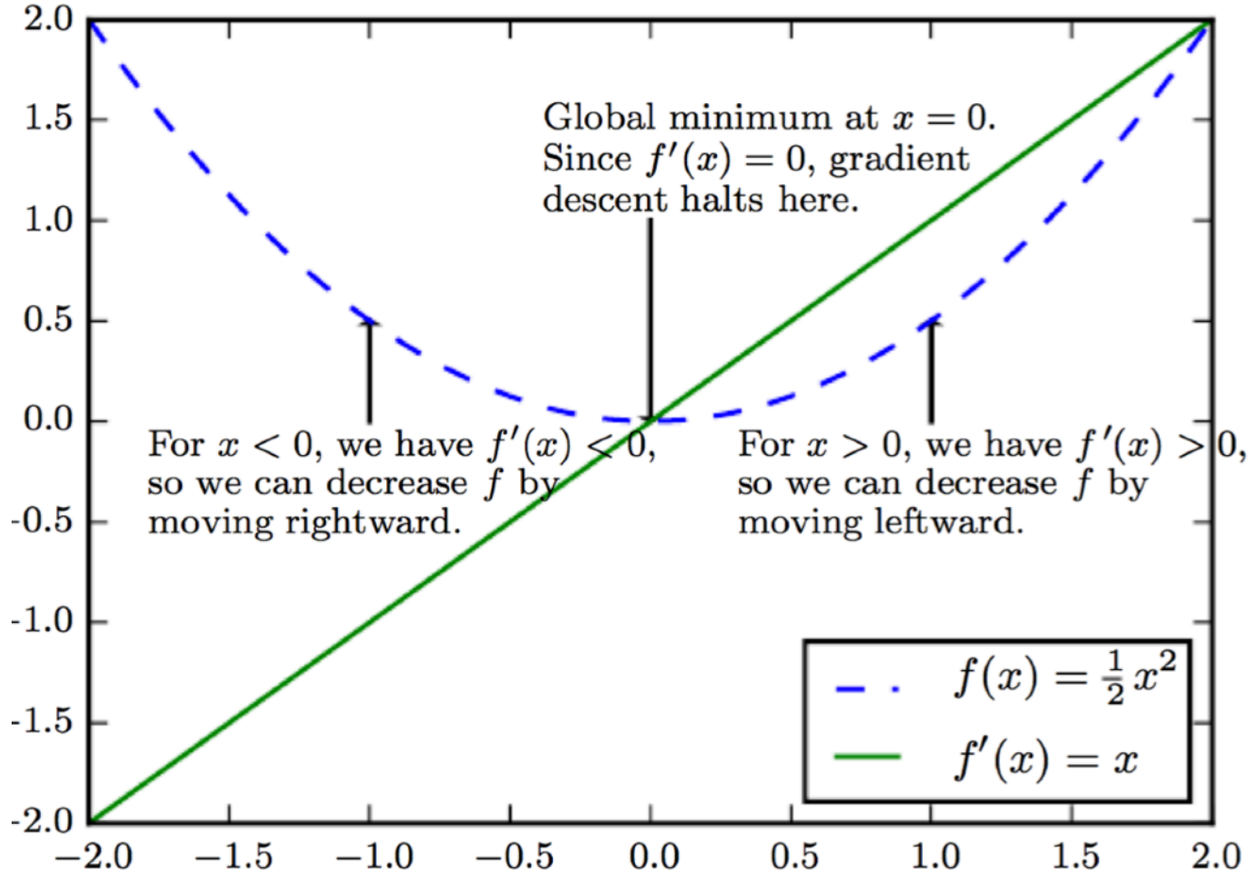
THE GRADIENT - 1-D CASE

Suppose function f(x), with x and y real numbers

- Derivative of function denoted: f'(x) or as df/dx
- Derivative f'(x) gives the slope of f(x) at point x
- It tells us how a small change in input x determines a change in the output f:
  $f(x + \varepsilon) \approx f(x) + \varepsilon f'(x)$

For small $\varepsilon \rightarrow f(x - \varepsilon \text{ sign}(f'(x))) < f(x)$
Thus we can reduce f(x) by *moving x in small steps with opposite sign of derivative*. This technique is called **gradient descent** (Cauchy 1847):

- f(x) increases when f'(x)>0
- f(x) decreases when f'(x)<0
- Idea: use f'(x) to follow function downhill
  → Reduce f(x) by going in the direction opposite to sign of f'(x)

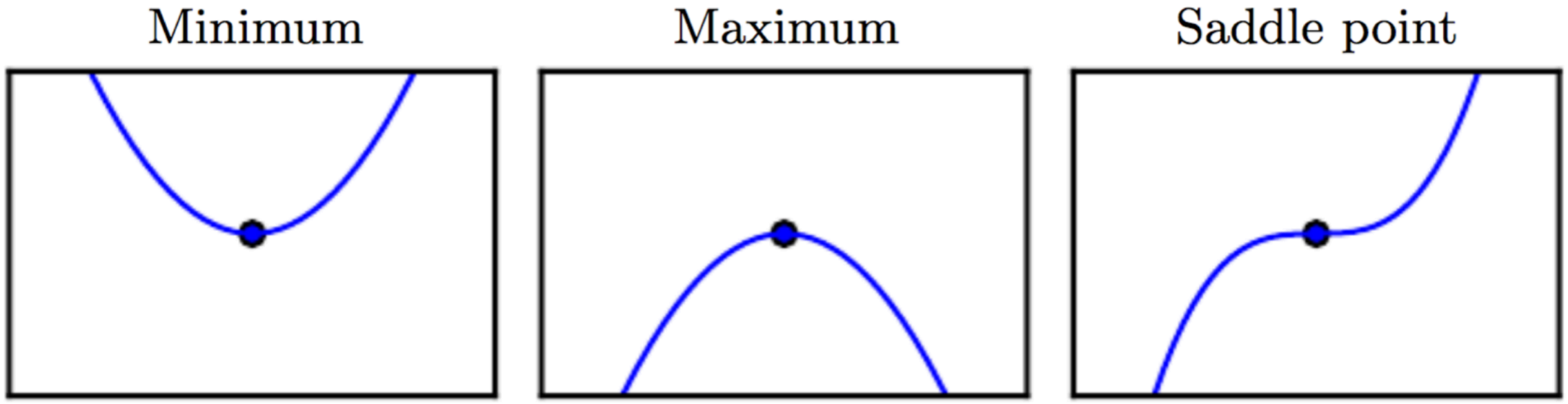


GRADIENT-BASED OPTIMIZATION

THE GRADIENT - 1-D CASE

- When f'(x)=0, derivative provides no information about direction of move

- Points where f'(x)=0 are known as *stationary or critical points*
  – Local minimum/maximum: a point where f(x) is lower/higher than all its neighbors
  – Saddle points: neither maxima nor minima

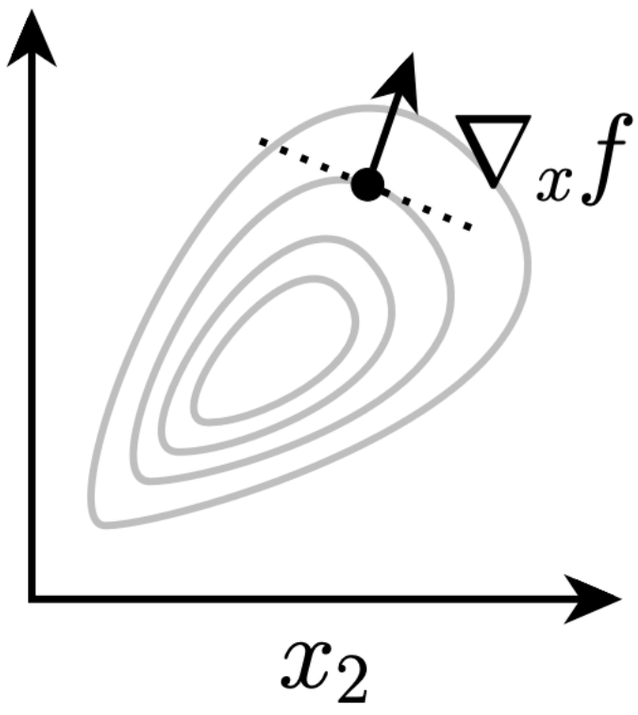


GRADIENT-BASED OPTIMIZATION

THE GRADIENT - N-D CASE

For $f: \mathbb{R}^n \to \mathbb{R}$, gradient is defined as vector of partial derivatives

$$\nabla_x f(x) \in \mathbb{R}^n = \begin{bmatrix} \frac{\partial f(x)}{\partial x_1} \\ \frac{\partial f(x)}{\partial x_2} \\ \vdots \\ \frac{\partial f(x)}{\partial x_n} \end{bmatrix}$$



Points in “steepest direction” of increase in function f

GRADIENT-BASED OPTIMIZATION

GRADIENT DESCENT - GENERAL FRAMEWORK

Gradient motivates a simple algorithm for minimizing $f(x)$: take small steps in the direction of the negative gradient

Algorithm: Gradient Descent
Given:
Function f, initial point x_0, step size $\alpha > 0$
Initialize:
$x \leftarrow x_0$
Repeat until convergence:
$x \leftarrow x \ - \alpha \nabla_x f(x)$

"Convergence" can be defined in a number of ways

Example - Least squares regression

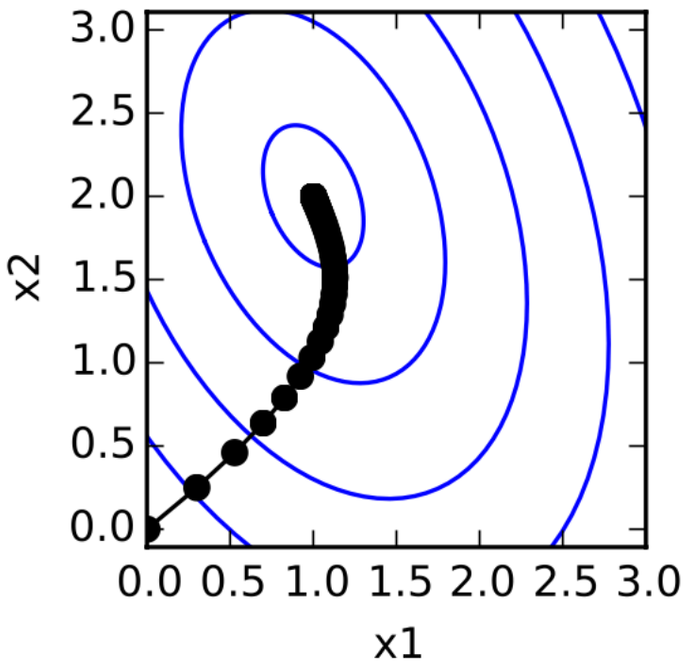
$$f(\mathrm{x}) = \frac{1}{2} || A\mathrm{x} - \mathrm{b} ||^2 \qquad \nabla_x f\left(\boldsymbol{x}\right) = A^T\left(A\boldsymbol{x} - b\right) = A^T A\boldsymbol{x} - A^T\boldsymbol{b}$$

GRADIENT-BASED OPTIMIZATION

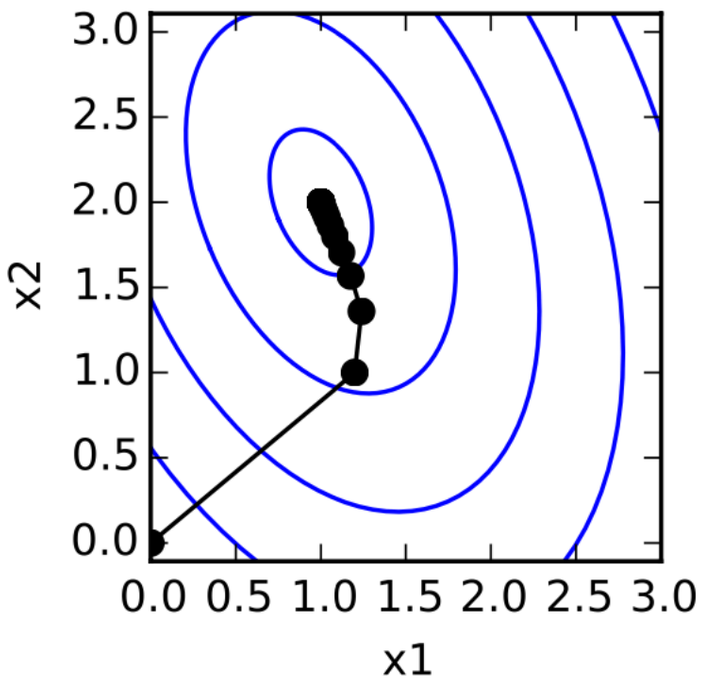
GRADIENT DESCENT - IN PRACTICE

The choice of α matters a lot in practice:

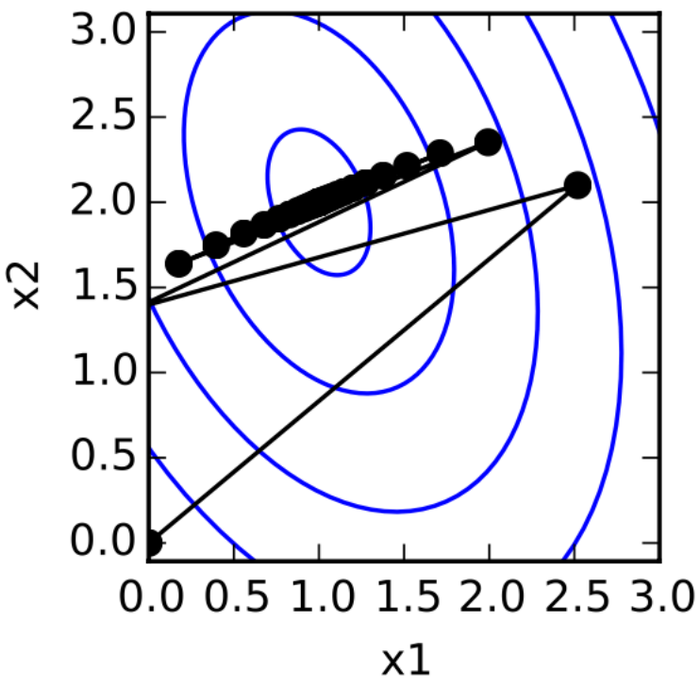
$$\underset{x}{\text{minimize}} \;\; 2x_1^2 + x_2^2 + x_1 x_2 - 6x_1 - 5x_2$$



$\alpha = 0.05$



$\alpha = 0.2$



$\alpha = 0.42$

GRADIENT-BASED OPTIMIZATION

GRADIENT DESCENT - WHY IT WORKS

Theorem: For differentiable f and small enough α, at any point x that is not a (local) minimum

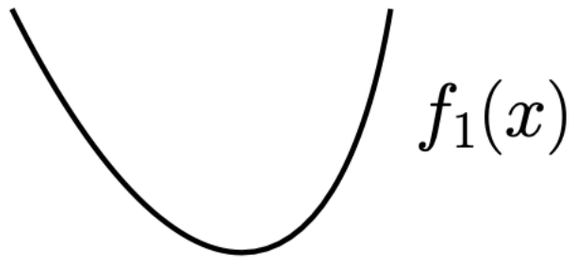
$$f\left(x - \alpha \nabla_x f(x)\right) < f(x)$$

i.e., gradient descent algorithm will decrease the objective

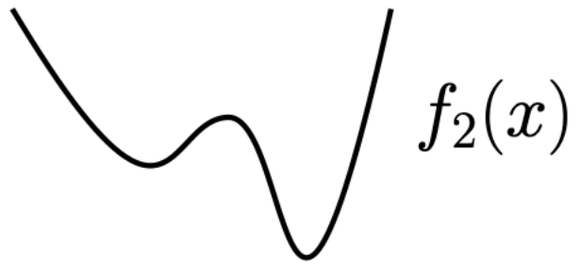
Works for both convex and non-convex functions, but with convex functions guaranteed to find global optimum

GRADIENT-BASED OPTIMIZATION

CONVEX VS. NONCONVEX OPTIMIZATION



Convex function



Nonconvex function

Originally, researchers distinguished between linear (easy) and nonlinear (hard) problems.

But in 80s and 90s, it became clear that this wasn't the right distinction, key difference is between convex and nonconvex problems.

Convex problem:

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & f(x) \\ \text{subject to} \quad & x \in \mathcal{C} \end{aligned}$$

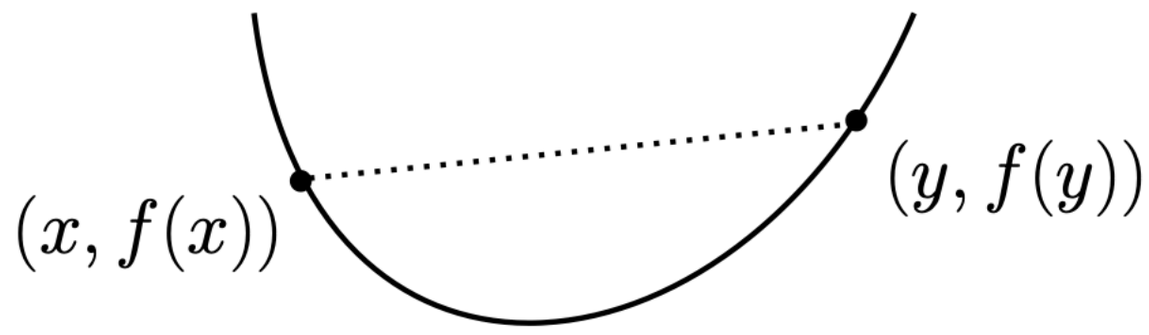
where f is convex and $\mathcal{C}$ is a convex set.

GRADIENT-BASED OPTIMIZATION

CONVEX FUNCTIONS

A function $f: \mathbb{R}^n \to \mathbb{R}$ is convex if, for any $x, y \in \mathbb{R}^n$ and $0 \leq \theta \leq 1$

$$f(\theta x + (1-\theta)y) \leq \theta f(x) + (1-\theta) f(y)$$



Convex functions “curve upwards” (or at least not downwards)

GRADIENT-BASED OPTIMIZATION

OTHER DEFINITIONS

- A function f(x) is concave if $-f(x)$ is convex
- An affine function is a function of the form $f(x)=a^{T}x+b$, where $a \in R^{n}$, $b \in R$
- If a function f(x) is both convex and concave in R^{n}, then f(x) is an affine function

GRADIENT-BASED OPTIMIZATION

CONVEX FUNCTIONS (EXAMPLES)

Exponential: $f(x) = \exp(ax), a \in \mathbb{R}$

Negative logarithm: $f(x) = -\log x$, with domain $x > 0$

Squared Euclidean norm: $f(x) = \|x\|_2^2 \equiv x^T x \equiv \sum_{i=1}^{n} x_i^2$

Euclidean norm: $f(x) = \|x\|_2$

Non-negative weighted sum of convex functions

$$f(x) = \sum_{i=1}^{m} w_i f_i(x)\,, \qquad w_i \geq 0, f_i \text{ convex}$$

GRADIENT-BASED OPTIMIZATION

HOW TO DETERMINE IF A FUNCTIONS IS CONVEX?

- Prove by definition
- Use properties
  - Sum of convex functions is convex
  - Convexity is preserved under a linear transformation:
    if f(x)=g(Ax+b), and g(x) is convex, then f(x) is convex
  - If f(x) is a twice differentiable function of one variable:
    f(x) is convex on an interval [a, b] ⊂ R iff its second derivative f''(x) ≥ 0 in [a, b]
  - If f(x) is a twice continuously differentiable function of n variables:
    f(x) is convex on F ⊂ R^n iff its Hessian H (i.e., the matrix of second partial derivatives) is positive semidefinite on the interior of F

$$\mathbf{H} = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1\,\partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1\,\partial x_n} \\ \frac{\partial^2 f}{\partial x_2\,\partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2\,\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n\,\partial x_1} & \frac{\partial^2 f}{\partial x_n\,\partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}$$

H is positive semidefinite in S if $\forall x \in S, \forall z \in \mathbb{R}^n, z^T H(x) z \geq 0$

H is positive semidefinite in $\mathbb{R}^n$ iff all eigenvalues of H are non-negative

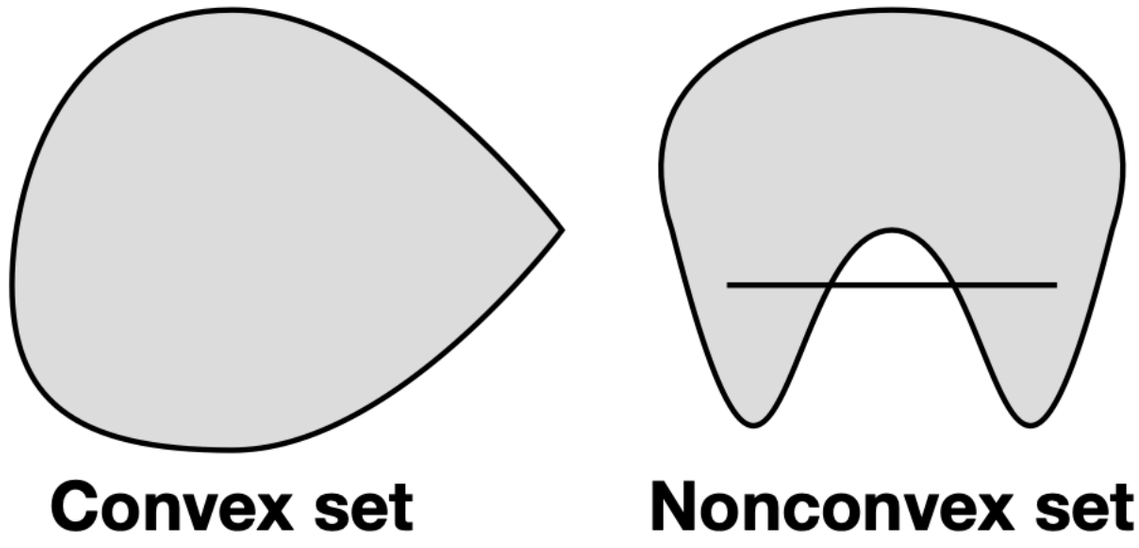
Alternatively, prove $z^T H(x) z = \sum_i \big(g_i(x,z)\big)^2$

GRADIENT-BASED OPTIMIZATION

CONVEX SETS

A set $\mathcal{C}$ is convex if, for any $x, y \in \mathcal{C}$ and $0 \leq \theta \leq 1$

$$\theta x + (1 - \theta) y \in \mathcal{C}$$



Convex set **Nonconvex set**

Examples:

All points $\mathcal{C} = \mathbb{R}^n$

Intervals $\mathcal{C} = \{x \in \mathbb{R}^n | \ l \leq x \leq u\}$ (elementwise inequality)

Linear equalities $\mathcal{C} = \{x \in \mathbb{R}^n | Ax = b\}$ (for $A \in \mathbb{R}^{m \times n}, b \in \mathbb{R}^m$)

Intersection of convex sets $\mathcal{C} = \bigcap_{i=1}^{m} \mathcal{C}_i$

GRADIENT-BASED OPTIMIZATION

CONVEX OPTIMIZATION

The key aspect of convex optimization problems that make them tractable is that *all local optima are global optima*

Definition: a point x is globally optimal if x is feasible and there is no feasible y such that $f(y) < f(x)$

Definition: a point x is locally optimal if x is feasible and there is some $R > 0$ such that for all feasible y with $\|x - y\|_2 \leq R$, $f(x) \leq f(y)$

Theorem: for a convex optimization problem all locally optimal points are globally optimal

GRADIENT-BASED OPTIMIZATION

PROOF OF GLOBAL OPTIMALITY

Proof: Given a locally optimal x (with optimality radius R), and suppose there exists some feasible y such that $f(y) < f(x)$

Now consider the point

$$z = \theta x + (1-\theta)y, \qquad \theta = 1 - \frac{R}{2\|x-y\|_2}$$

1) Since $x, y \in \mathcal{C}$ (feasible set), we also have $z \in \mathcal{C}$ (by convexity of $\mathcal{C}$)

2) Furthermore, since f is convex:
$f(z) = f(\theta x + (1-\theta)y) \leq \theta f(x) + (1-\theta)f(y) < f(x)$ and
$\|x - z\|_2 = \left\| x - \left(1 - \frac{R}{2\|x-y\|_2}\right)x + \frac{R}{2\|x-y\|_2} y \right\|_2 = \left\| \frac{R(x-y)}{2\|x-y\|_2} \right\|_2 = \frac{R}{2}$

Thus, z is feasible, within radius R of x, and has lower objective value, a contradiction of supposed local optimality of x

■

GRADIENT-BASED OPTIMIZATION

EXAMPLE: MACHINE LEARNING

Virtually all (supervised) machine learning algorithms boil down to solving an optimization problem:

$$\underset{\theta}{\text{minimize}} \sum_{i=1}^{m} \ell\left(h_\theta\left(x^{(i)}\right), y^{(i)}\right)$$

Where $x^{(i)} \in \mathcal{X}$ are inputs, $y^{(i)} \in \mathcal{Y}$ are outputs, ℓ is the so called **loss function** (e.g, MSE), and h_θ is a function parameterized by θ, which are the parameters of the model we are optimizing over.

If the loss function is convex → local minimum is global optimum!

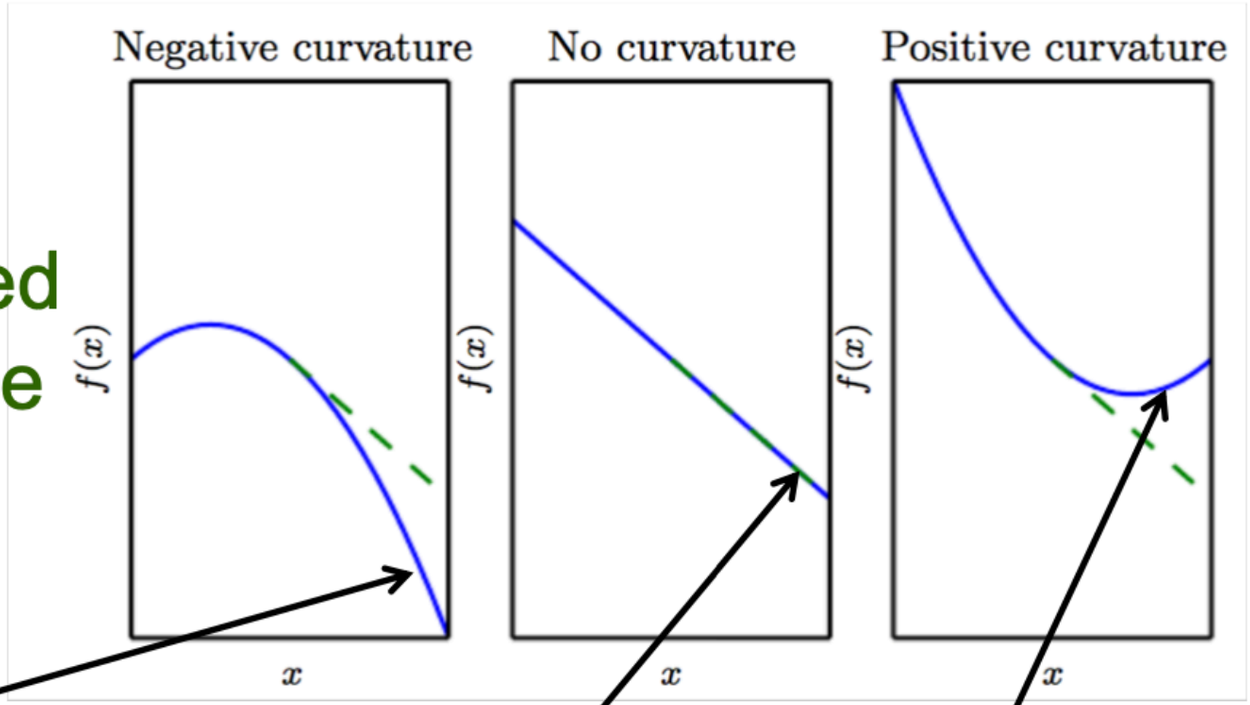
GRADIENT-BASED OPTIMIZATION

2ND ORDER METHODS - CURVATURE INFORMATION

Some methods use the 2nd order derivative (in n dimensions, the Hessian) for further efficiency. 2nd order information, indeed, tells us how the first derivative will change as we vary the input. This is important as it tells us whether a gradient step in a certain direction will cause an improvement.

Curvature information can be used to test critical points and determine a better learning rate.

Dashed line is value of cost function predicted by gradient alone



Decrease is faster than predicted by Gradient Descent

Gradient Predicts decrease correctly

Decrease is slower than expected Actually increases

GRADIENT-BASED OPTIMIZATION

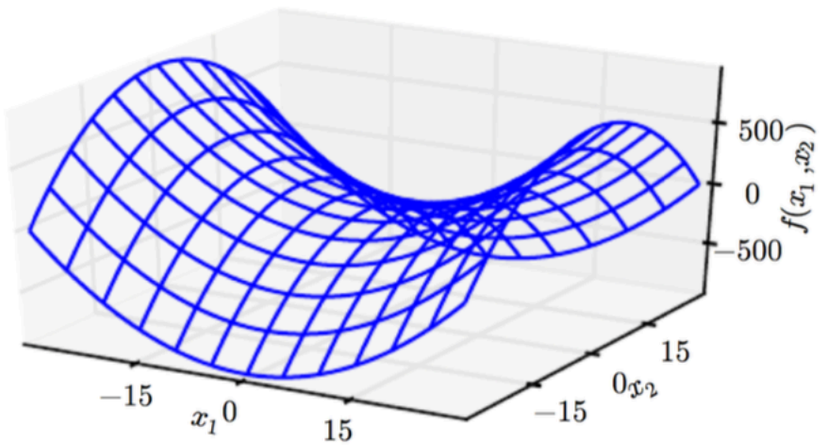
2ND ORDER METHODS - HOW TO TEST CRITICAL POINTS

1-D case:

- On a critical point $f'(x)=0$
- Local minimum: $f'(x)=0$ and $f''(x)>0$, i.e., the first derivative $f'(x)$ increases as we move to the right and decreases as we move to the left
- Local maximum: $f'(x)=0$ and $f''(x)<0$, i.e., the first derivative $f'(x)$ decreases as we move to the right and increases as we move to the left
- When $f''(x)=0$ test is inconclusive: x may be a *saddle point* or part of a flat region!

N-D case:

- We need to examine second derivatives of all dimensions → eigendecomposition of H (i.e., we test the eigenvalues of Hessian to determine whether a critical point is a local maximum, a local minimum, or a saddle point)
- When H is positive definite (all eigenvalues are positive) the point is a local minimum
- When H is negative definite (all eigenvalues are negative) the point is a local maximum
- Saddle point: some eigenvalues are positive, some are negative



GRADIENT-BASED OPTIMIZATION

2ND ORDER METHODS - GRADIENT DESCENT MODIFIED

Consider the N-D case, Taylor's series of f(x) around current point x(0):

$$f(\boldsymbol{x}) \approx f(\boldsymbol{x}^{(0)}) + (\boldsymbol{x} - \boldsymbol{x}^{(0)})^T \boldsymbol{g} + \frac{1}{2}(\boldsymbol{x} - \boldsymbol{x}^{(0)})^T H(\boldsymbol{x} - \boldsymbol{x}^{(0)})$$

where g is the gradient and H is the Hessian at x(0).

If we use learning rate ε the new point x is given by x(0)-εg. Thus we get:

$$f(\boldsymbol{x}^{(0)} - \varepsilon \boldsymbol{g}) \approx f(\boldsymbol{x}^{(0)}) - \varepsilon \boldsymbol{g}^T \boldsymbol{g} + \frac{1}{2}\varepsilon^2 g^T H \boldsymbol{g}$$

original value of f in x(0)

expected improvement due to slope (gradient)

correction to be applied due to curvature

Solving for step size when correction is least gives: $\varepsilon^* \approx \frac{\boldsymbol{g}^T \boldsymbol{g}}{\boldsymbol{g}^T H \boldsymbol{g}}$

GRADIENT-BASED OPTIMIZATION

2ND ORDER METHODS - CONDITION NUMBER

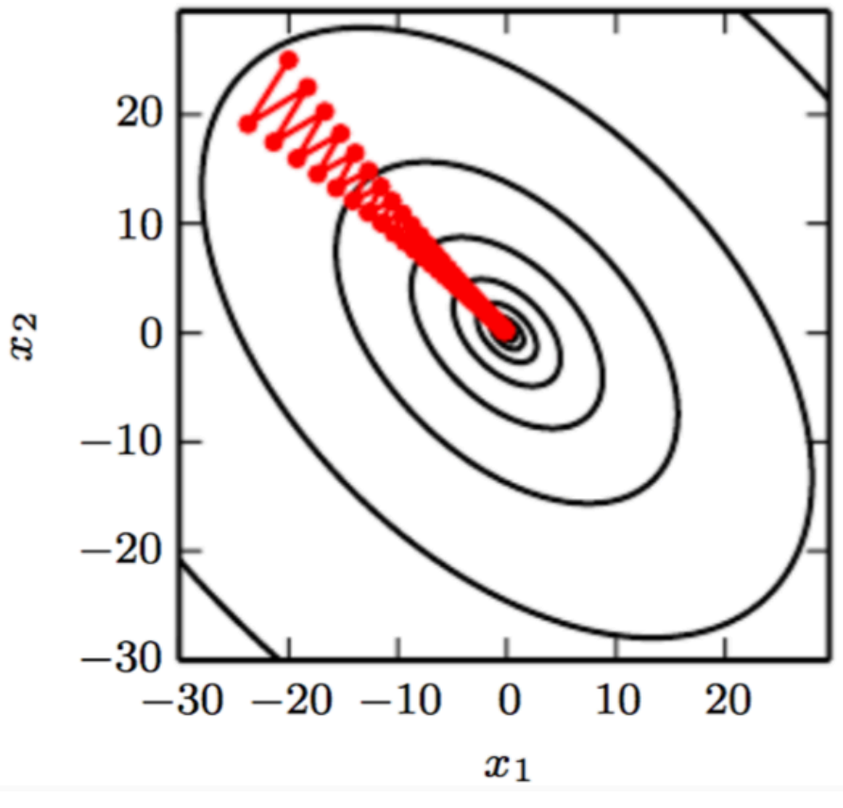
- There are different second derivatives in each direction at a single point

- Condition number of H i.e., $\lambda_{max}/\lambda_{min}$ measures how much they differ
  - Gradient descent performs poorly when H has a high condition number
  - Because in one direction derivative increases rapidly while in another direction it increases slowly
  - Step size should adapt to the curvature along the different directions

- E.g., with condition number 5, the direction of "most curvature" has 5 times more curvature than the direction of "least curvature"

- In such cases, due to small step size, gradient descent wastes time!

- An algorithm based on Hessian can instead decide "where to go" more efficiently than steepest descent



GRADIENT-BASED OPTIMIZATION

2ND ORDER METHODS - NEWTON METHOD

- Second-order information allows us to make a quadratic approximation of the objective function and approximate the right step size to reach a local minimum

- In other words, the benefit of using the Hessian, when available, is that it can be used to determine both the direction and the step size to move in order to change the input parameters to minimize (or maximize) the objective function.

Newton method

$$\boldsymbol{x}^* = \boldsymbol{x}^{(0)} - H(f)(\boldsymbol{x}^{(0)})^{-1}\nabla_{\boldsymbol{x}} f(\boldsymbol{x}^{(0)})$$

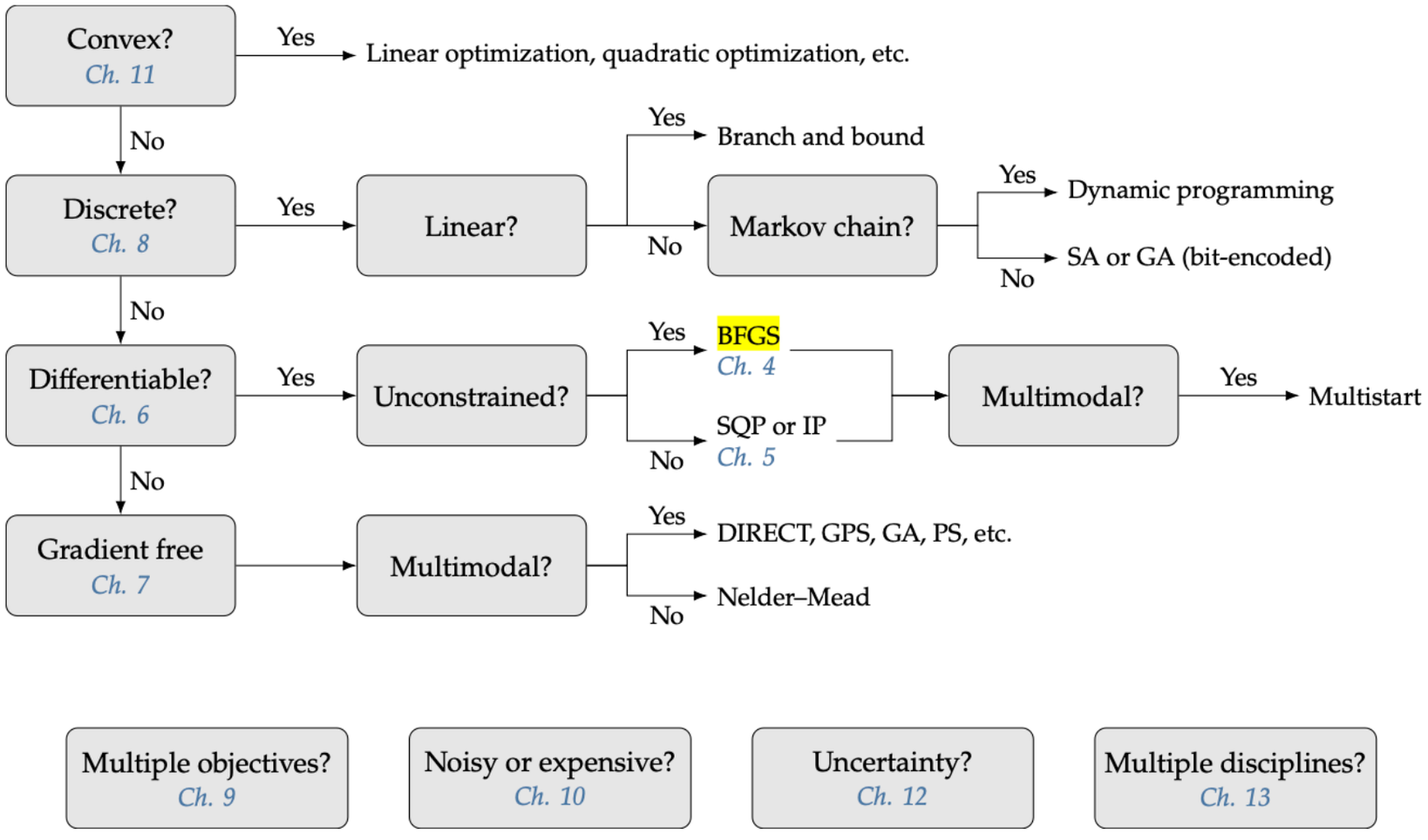
- When f is a quadratic (positive definite) function it jumps to the minimum function directly
- When f is not quadratic, it can be applied iteratively
- Can reach critical point much faster than gradient descent
- But, useful only when nearby point is a minimum!

GRADIENT-BASED OPTIMIZATION

2ND ORDER METHODS - QUASI-NEWTON METHODS

- Inverting the Hessian is computationally expensive, especially for large-dimensional problems
  → in general, complexity $O(n^3)$

- Quasi-Newton methods approximate the inverse Hessian, which can then be used to determine the direction to move, but we no longer have the step size → de facto standard method: BFGS



Source: https://openmdao.org/a-new-free-text-book-on-mdo/, p. 27

GRADIENT-BASED OPTIMIZATION

BROYDEN–FLETCHER–GOLDFARB–SHANNO (BFGS) METHOD

The BFGS algorithm addresses this by using a line search in the chosen direction to determine how far to move in that direction. The total algorithm complexity is $O(n^2)$,

Inputs:

x_0: Starting point

τ: Convergence tolerance

Outputs:

x^*: Optimal point

$f(x^*)$: Minimum function value

$k = 0$ — Initialize iteration counter

$\alpha_{\text{init}} = 1$ — Initial step length for line search

while $\|\nabla f_k\|_\infty > \tau$ **do** — Optimality condition

  if $k = 0$ or reset = `true` **then**

    $\tilde{V}_k = \frac{1}{\|\nabla f\|} I$

  else

    $s = x_k - x_{k-1}$ — Last step

    $y = \nabla f_k - \nabla f_{k-1}$ — Curvature along last step

    $\sigma = \frac{1}{s^\intercal y}$

    $\tilde{V}_k = \left(I - \sigma s y^\intercal\right) \tilde{V}_{k-1} \left(I - \sigma y s^\intercal\right) + \sigma s s^\intercal$ — Quasi-Newton update

  end if

  $p = -\tilde{V}_k \nabla f_k$ — Compute quasi-Newton step

  $\alpha = \texttt{linesearch}\left(p, \alpha_{\text{init}}\right)$ — Should satisfy the strong Wolfe conditions

  $x_{k+1} = x_k + \alpha p$ — Update design variables

  $k = k + 1$ — Increment iteration index

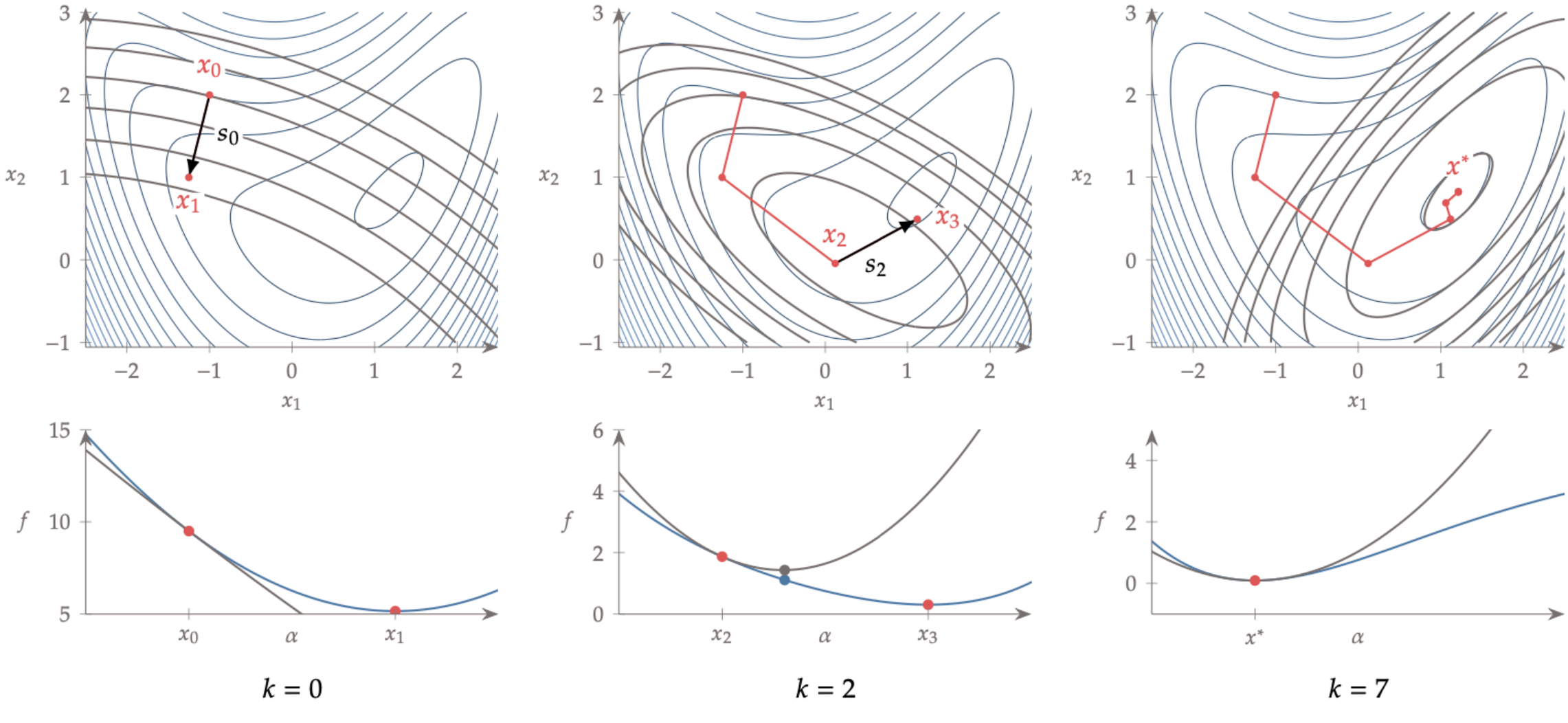
end while

Approximation of H, occasionally reset since curvature info gathered from the current point may become irrelevant or even counterproductive

GRADIENT-BASED OPTIMIZATION

BFGS METHOD IN ACTION

Minimization of the bean function using BFGS. The first quadratic approximation has circular contours (left). After two iterations, the quadratic approximation improves, and the step approaches the minimum (middle). Once converged, the minimum of the quadratic approximation coincides with the bean function minimum (right).



GRADIENT-BASED OPTIMIZATION

LIMITED-MEMORY BFGS (L-BFGS)

When the no. of design variables is large (millions or billions), it might not be possible to store the Hessian inverse approximation matrix in memory. Limited-memory quasi-Newton methods make it possible to handle such problems. In addition, these methods improve the computational efficiency of medium-sized problems (hundreds/thousands of design variables) with minimal sacrifice in accuracy.

Inputs:

∇f_k: Gradient at point x_k

$s_{k-1,\ldots,k-m}$: History of steps $x_k - x_{k-1}$

$y_{k-1,\ldots,k-m}$: History of gradient differences $\nabla f_k - \nabla f_{k-1}$

Outputs:

p: Search direction $-\tilde{V}_k \nabla f_k$

$d = \nabla f_k$

for $i = k-1$ **to** $k-m$ **by** -1 **do**

  $\alpha_i = \sigma_i s_i{}^\intercal d$

  $d = d - \alpha_i y_i$

end for

$\tilde{V}_0 = \left(\frac{s_{k-1}^\intercal y_{k-1}}{y_{k-1}^\intercal y_{k-1}}\right) I$ Initialize Hessian inverse approximation as a scaled identity matrix

$d = \tilde{V}_0 d$

for $i = k-m$ **to** $k-1$ **do**

  $\beta_i = \sigma_i y_i{}^\intercal d$

  $d = d + (\alpha_i - \beta_i) s_i$

end for

$p = -d$

Main intuitions

1. If we save the sequence of s and y vectors we can iteratively update the approximated Hessian.

2. To reduce memory usage, we do not store the entire history of vectors. Instead, we limit the storage to the last m vectors for s and y (usually m between 5 and 20)

3. We make the starting Hessian diagonal such that we only require vector storage (or scalar storage if we make all entries in the diagonal equal). A common choice is to use a scaled identity matrix, which just requires storing one number!

GRADIENT-BASED OPTIMIZATION

LIMITED-MEMORY BFGS (L-BFGS)

The L-BFGS direction update mechanism is plugged here

Inputs:

x_0: Starting point

τ: Convergence tolerance

Outputs:

x^*: Optimal point

$f(x^*)$: Minimum function value

$k = 0$ — Initialize iteration counter

$\alpha_{\text{init}} = 1$ — Initial step length for line search

while $\|\nabla f_k\|_\infty > \tau$ **do** — Optimality condition

  if $k = 0$ or reset = `true` **then**

    $\tilde{V}_k = \frac{1}{\|\nabla f\|} I$

  else

    $s = x_k - x_{k-1}$ — Last step

    $y = \nabla f_k - \nabla f_{k-1}$ — Curvature along last step

    $\sigma = \frac{1}{s^\intercal y}$

    $\tilde{V}_k = \left(I - \sigma s y^\intercal\right) \tilde{V}_{k-1} \left(I - \sigma y s^\intercal\right) + \sigma s s^\intercal$ — Quasi-Newton update

  end if

  $p = -\tilde{V}_k \nabla f_k$ — Compute quasi-Newton step

  $\alpha = \texttt{linesearch}\left(p, \alpha_{\text{init}}\right)$ — Should satisfy the strong Wolfe conditions

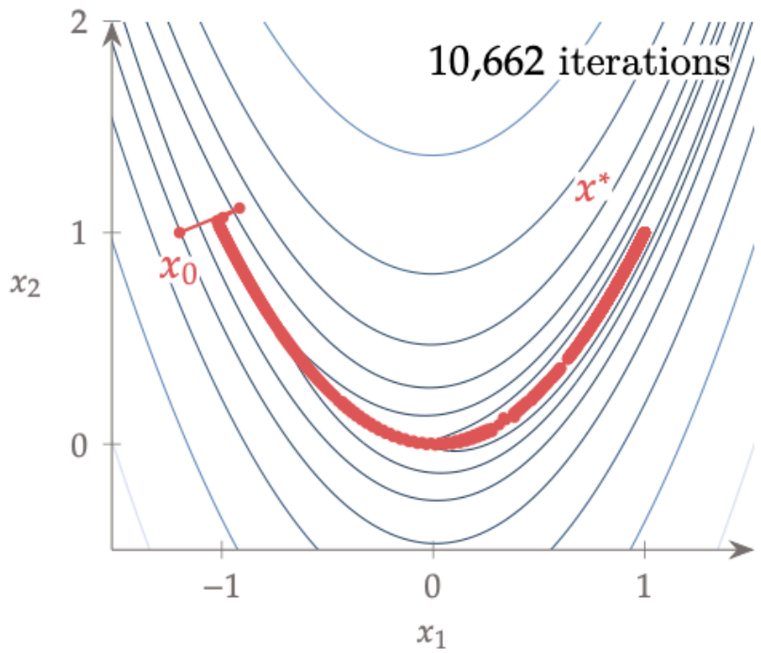
  $x_{k+1} = x_k + \alpha p$ — Update design variables

  $k = k + 1$ — Increment iteration index

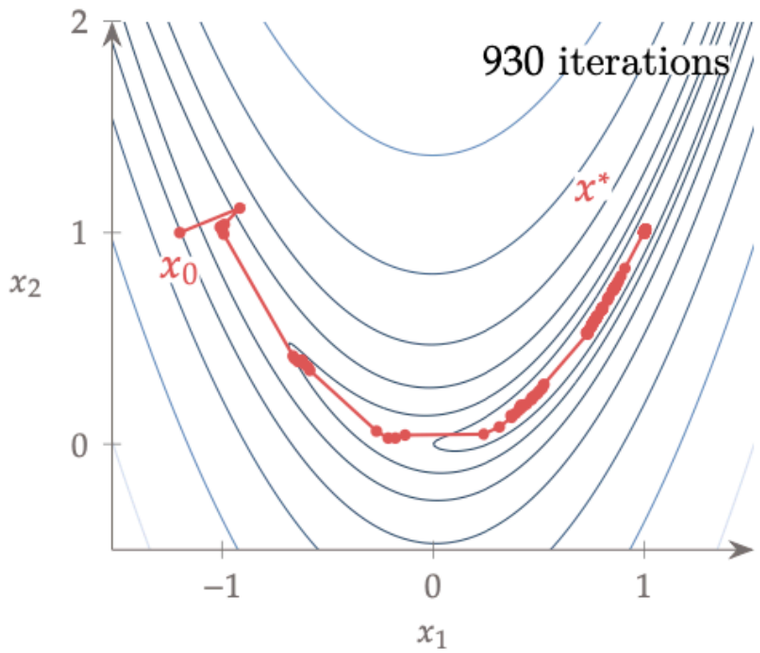
end while

GRADIENT-BASED OPTIMIZATION

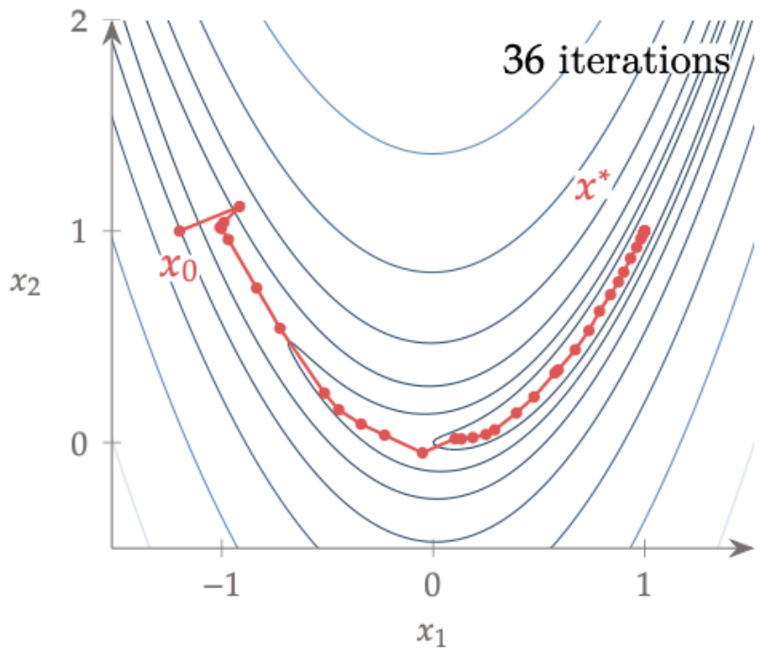
HOW DO GRADIENT-BASED METHODS COMPARE?



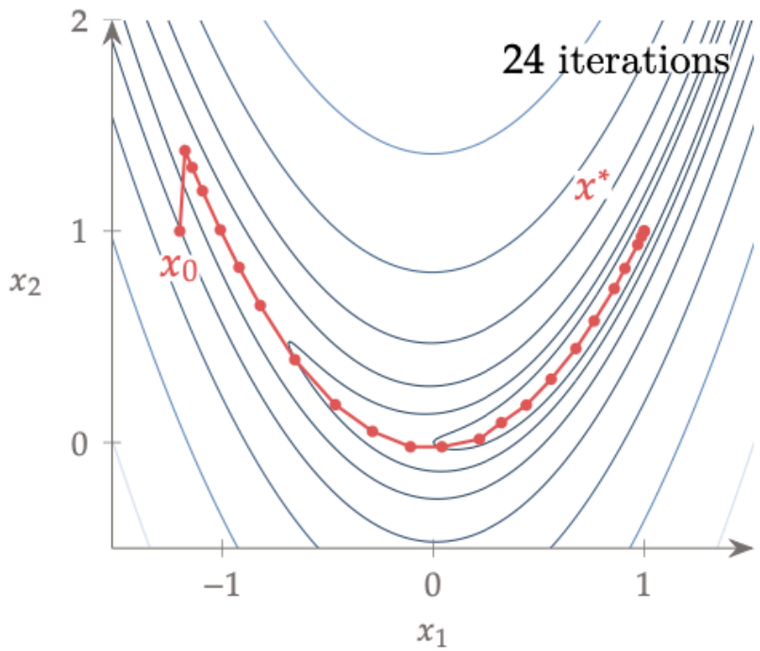
Steepest descent



Conjugate gradient



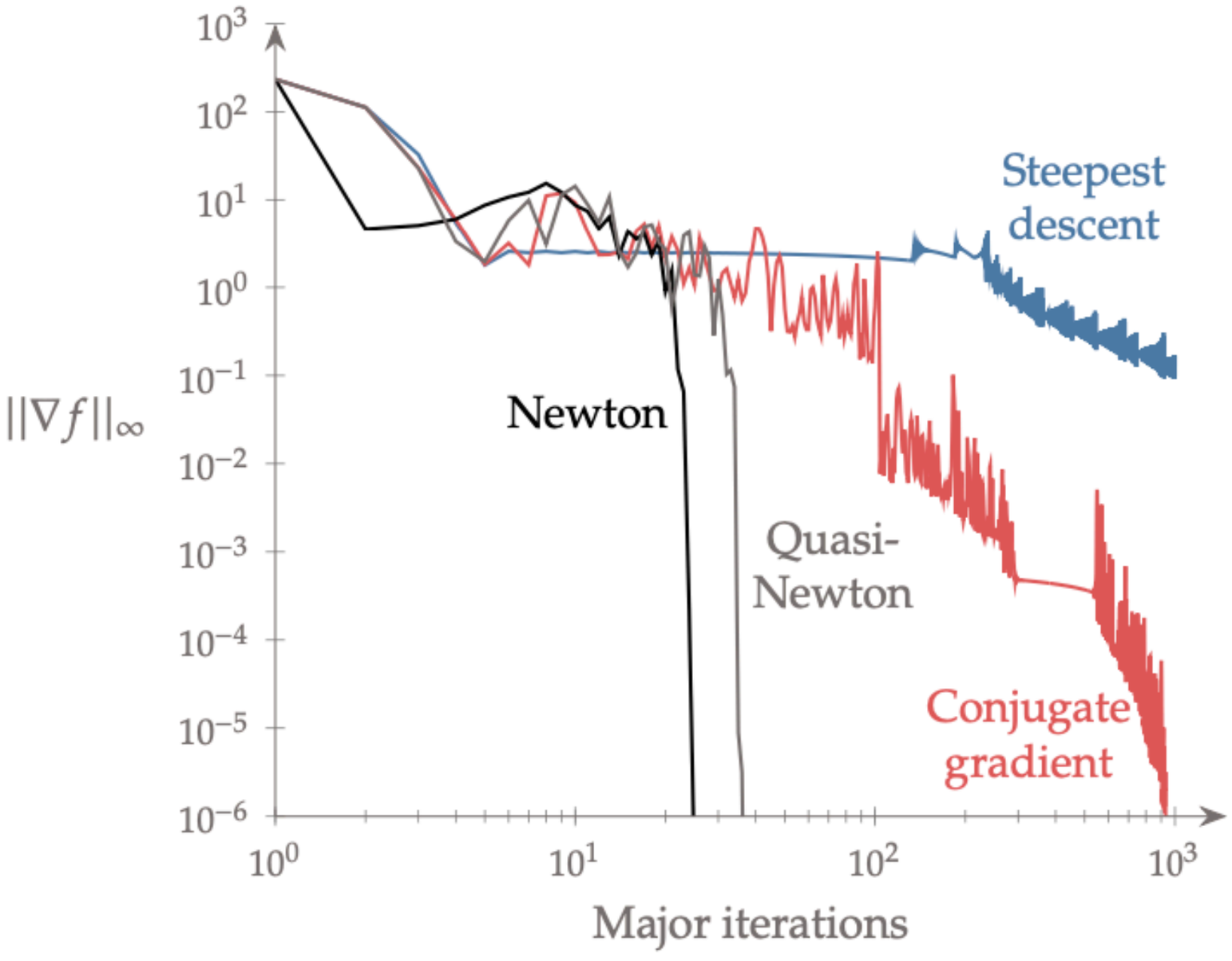
Quasi-Newton



Newton

GRADIENT-BASED OPTIMIZATION

HOW DO GRADIENT-BASED METHODS COMPARE?



Steepest descent
→ linear convergence

Conjugate gradient method
(not seen in this lecture)
→ super-linear convergence

Second-order methods
→ quadratic convergence

OPTIMIZATION

BFGS IN PYTHON

minimize(method='BFGS')

```
scipy.optimize.minimize(fun, x0, args=(), method='BFGS', jac=None, tol=None,
callback=None, options={'gtol': 1e-05, 'norm': inf, 'eps':
1.4901161193847656e-08, 'maxiter': None, 'disp': False, 'return_all': False,
'finite_diff_rel_step': None})
```

Minimization of scalar function of one or more variables using the BFGS algorithm.

See also

For documentation for the rest of the parameters, see `scipy.optimize.minimize`

Options: ---

disp : *bool*

Set to True to print convergence messages.

maxiter : *int*

Maximum number of iterations to perform.

gtol : *float*

Gradient norm must be less than *gtol* before successful termination.

norm : *float*

Order of norm (Inf is max, -Inf is min).

eps : *float or ndarray*

If *jac is None* the absolute step size used for numerical approximation of the jacobian via forward differences.

return_all : *bool, optional*

Set to True to return a list of the best solution at each of the iterations.

finite_diff_rel_step : *None or array_like, optional*

If *jac in ['2-point', '3-point', 'cs']* the relative step size to use for numerical approximation of the jacobian. The absolute step size is computed as `h = rel_step * sign(x) * max(1, abs(x))`, possibly adjusted to fit into the bounds. For `method='3-point'` the sign of *h* is ignored. If None (default) then step is selected automatically.

OPTIMIZATION

L-BFGS IN PYTHON

minimize(method='L-BFGS-B')

```
scipy.optimize.minimize(fun, x0, args=(), method='L-BFGS-B', jac=None,
bounds=None, tol=None, callback=None, options={'disp': None, 'maxcor': 10,
'ftol': 2.220446049250313e-09, 'gtol': 1e-05, 'eps': 1e-08, 'maxfun': 15000,
'maxiter': 15000, 'iprint': - 1, 'maxls': 20, 'finite_diff_rel_step': None})
```

Minimize a scalar function of one or more variables using the L-BFGS-B algorithm.

See also

For documentation for the rest of the parameters, see `scipy.optimize.minimize`

Options:

disp : *None or int*

If *disp is None* (the default), then the supplied version of *iprint* is used. If *disp is not None*, then it overrides the supplied version of *iprint* with the behaviour you outlined.

maxcor : *int*

The maximum number of variable metric corrections used to define the limited memory matrix. (The limited memory BFGS method does not store the full hessian but uses this many terms in an approximation to it.)

ftol : *float*

The iteration stops when `(f^k - f^{k+1})/max{|f^k|,|f^{k+1}|,1} <= ftol`.

gtol : *float*

The iteration will stop when `max{|proj g_i | i = 1, ..., n} <= gtol` where `pg_i` is the i-th component of the projected gradient.

eps : *float or ndarray*

If *jac is None* the absolute step size used for numerical approximation of the jacobian via forward differences.

maxfun : *int*

Maximum number of function evaluations.

maxiter : *int*

Maximum number of iterations.

iprint : *int, optional*

Controls the frequency of output. `iprint < 0` means no output; `iprint = 0` print only one line at the last iteration; `0 < iprint < 99` print also f and `|proj g|` every iprint iterations; `iprint = 99` print details of every iteration except n-vectors; `iprint = 100` print also the changes of active set and final x; `iprint > 100` print details of every iteration including x and g.

callback : *callable, optional*

Called after each iteration, as `callback(xk)`, where `xk` is the current parameter vector.

maxls : *int, optional*

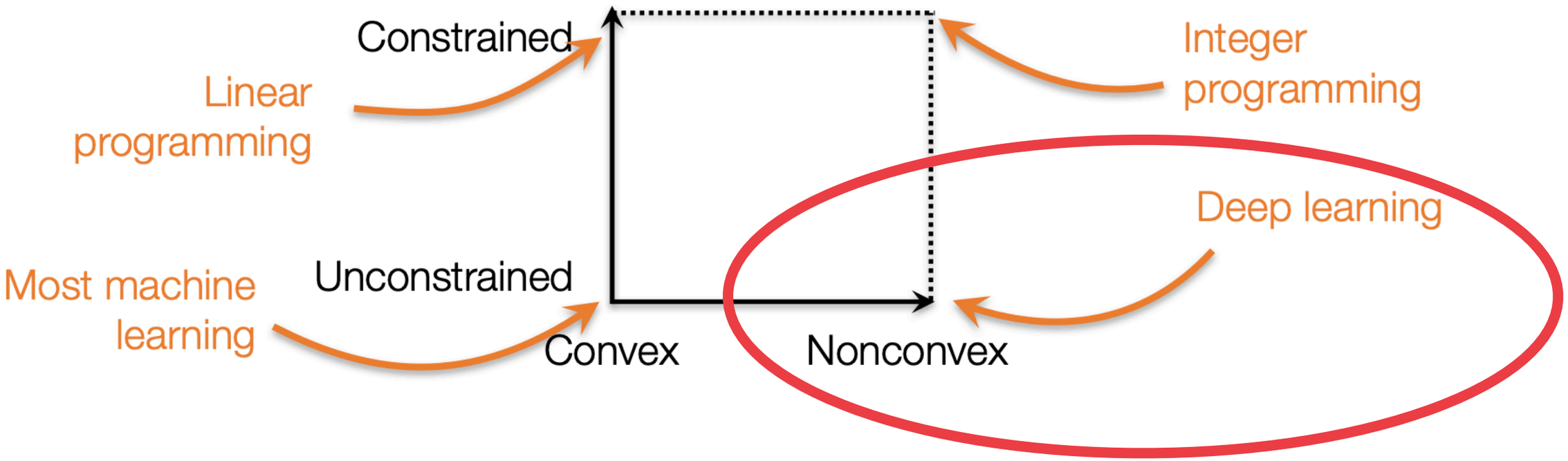
Maximum number of line search steps (per iteration). Default is 20.

finite_diff_rel_step : *None or array_like, optional*

If *jac in ['2-point', '3-point', 'cs']* the relative step size to use for numerical approximation of the jacobian. The absolute step size is computed as `h = rel_step * sign(x) * max(1, abs(x))`, possibly adjusted to fit into the bounds. For `method='3-point'` the sign of *h* is ignored. If None (default) then step is selected automatically.

OPTIMIZATION

ONE WORD FOR SEVERAL MEANINGS

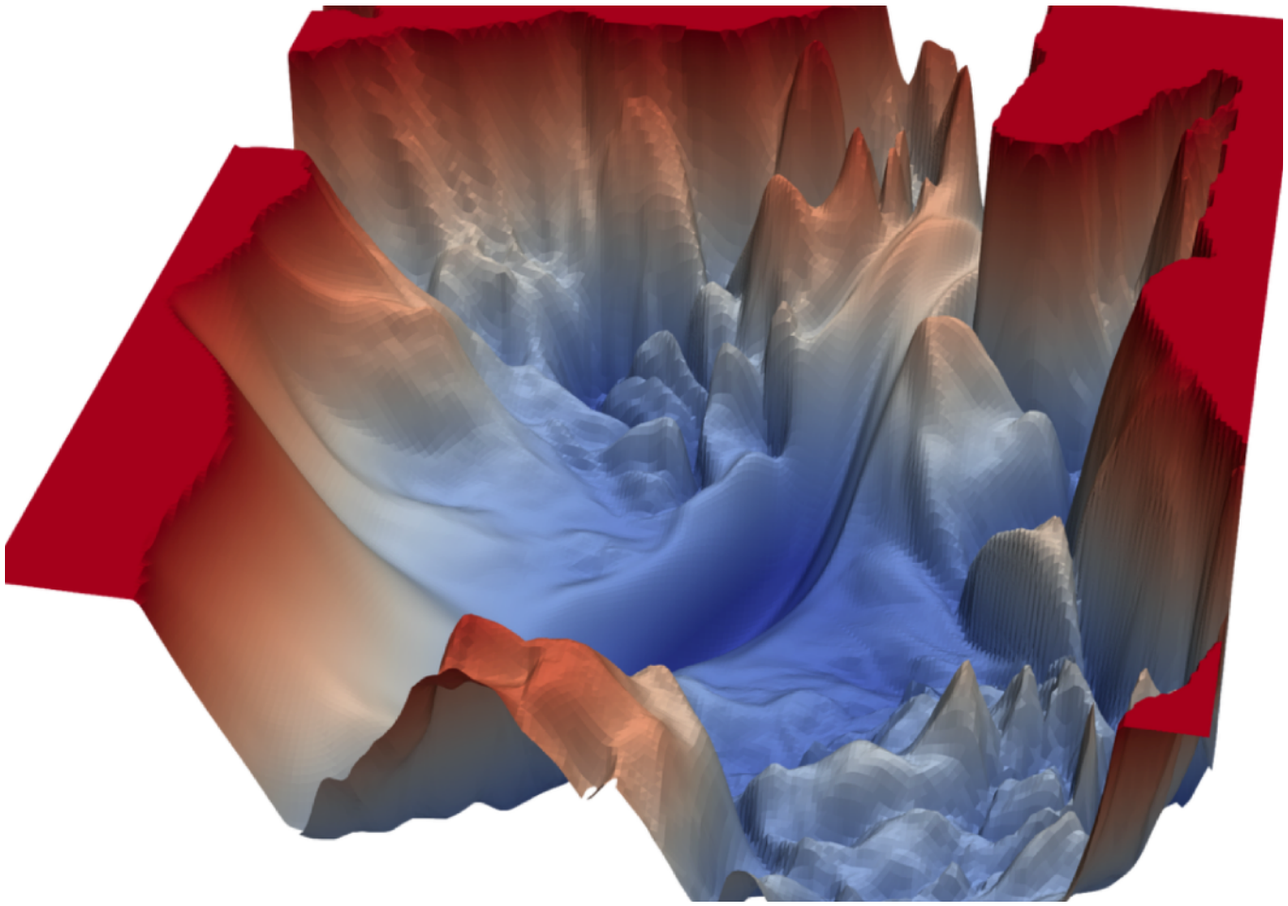


GRADIENT-BASED OPTIMIZATION

THE CASE OF DEEP LEARNING

"*If you have a million dimensions, and you're coming down, and you come to a ridge, even if half the dimensions are going up, the other half are going down! So you always find a way to get out, you never get trapped*" on a ridge, at least, not permanently.

https://www.zdnet.com/article/ai-pioneer-sejnowski-says-its-all-about-the-gradient/



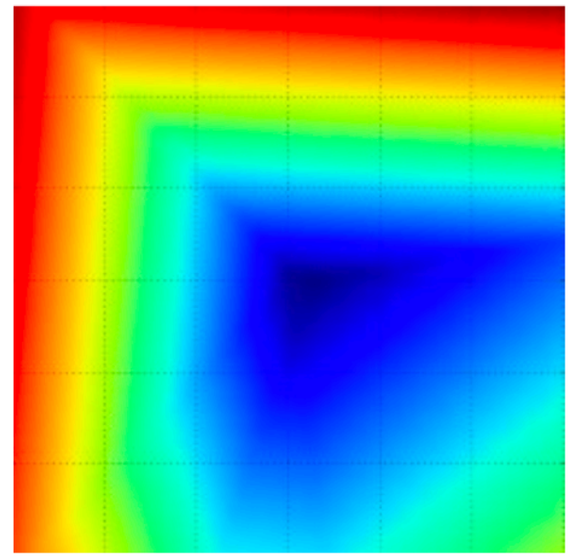
GRADIENT-BASED OPTIMIZATION

STOCHASTIC GRADIENT DESCENT

At each step, pick a sample randomly from the training data and compute:

$$\theta_{t+1} = \theta_t - \alpha \delta L(\theta_t)$$

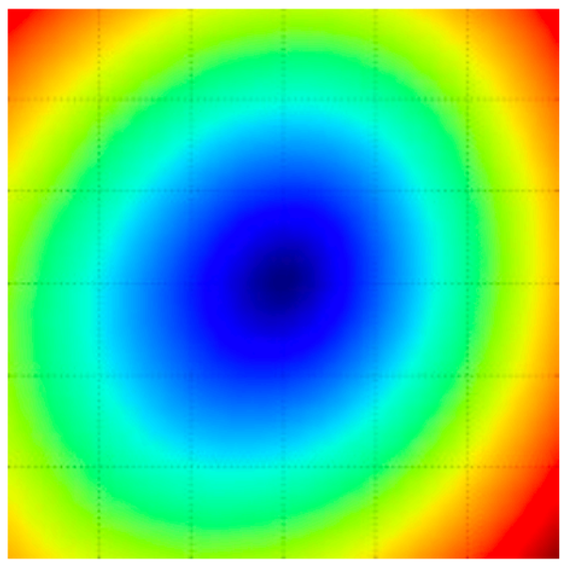
- θ (weights) are updated according to the gradient of the loss with respect to θ.
- α is the learning rate.
- If α is very small, convergence will be very slow. On the other hand, large α will lead to divergence.



Loss for single example

NOTES

- SGD uses a single training sample at each update step.
- However, a single sample may be uninformative or misleading.
- In principle, we should apply gradient descent by averaging gradient over the entire dataset (not scalable for very large data sets).
- Solution: mini-batch of such gradient estimates to construct the update step (**batch stochastic gradient**).
- Because we are averaging more component gradients, we can expect the variance of such an estimate to be lower.



Average loss for 100 examples (convex function)

GRADIENT-BASED OPTIMIZATION

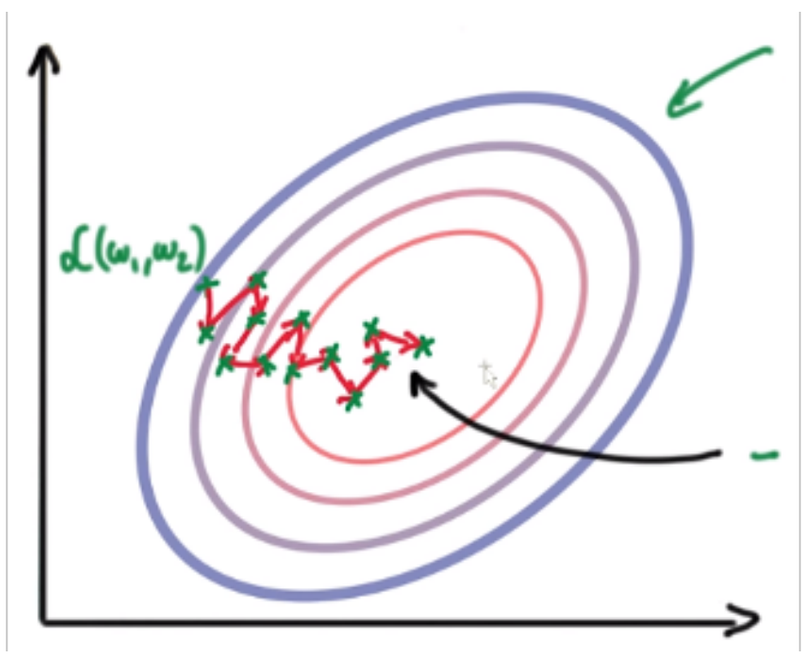
ANOTHER SOLUTION: MOMENTUM

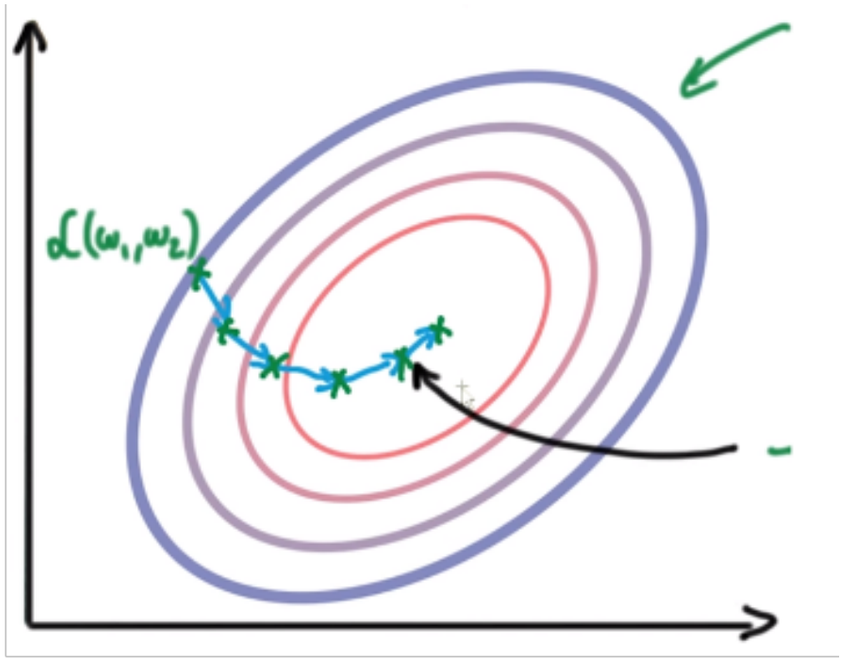
- Due to the diversity of each training sample, the gradient of the loss changes quickly after each iteration. We are taking small steps, but they are follow a zig-zag pattern (even though we slowly reach, eventually, a loss minimum).

- To overcome this, we introduce **momentum**. Basically, taking knowledge from previous steps about where we should be heading. We introduce a new hyper-parameter μ:

$$v_{t+1} = \mu v_t - \alpha \delta L(\theta_t)$$

$$\theta_{t+1} = \theta_t + v_{t+1}$$





GRADIENT-BASED OPTIMIZATION

MANY DIFFERENT OPTIMIZATION ALGORITHMS IN DL!

- RMSprop (Root Mean Square Propagation)
- Adaptive gradient descent variants
  - Adagrad (Adaptive Gradient Algorithm)
  - Adadelta (an extension of Adagrad that seeks to reduce its aggressive, monotonically decreasing learning rate)
  - Adam (Adaptive Moment Estimation – keeps separate learning rates for each weight as well as an exponentially decaying average of previous gradients)
  - Adamax (A variant of Adam that scales the gradient inversely proportionally to the $\ell 2$ norm of the past gradients)
  - Nadam (Nesterov-accelerated Adaptive Moment Estimation)

- The main difference is actually how they treat the learning rate.

GRADIENT-BASED OPTIMIZATION

ADAPTIVE MOMENT ESTIMATION (ADAM)

Like Adadelta and RMSprop, but in addition to storing an exponentially decaying average of past squared gradients g_t, Adam also keeps an exponentially decaying average of past gradients m_t, similar to momentum:

$$m_{t+1} = \gamma_1 m_t + (1 - \gamma_1) \nabla \mathcal{L}(\theta_t)$$

$$g_{t+1} = \gamma_2 g_t + (1 - \gamma_2) \nabla \mathcal{L}(\theta_t)^2$$

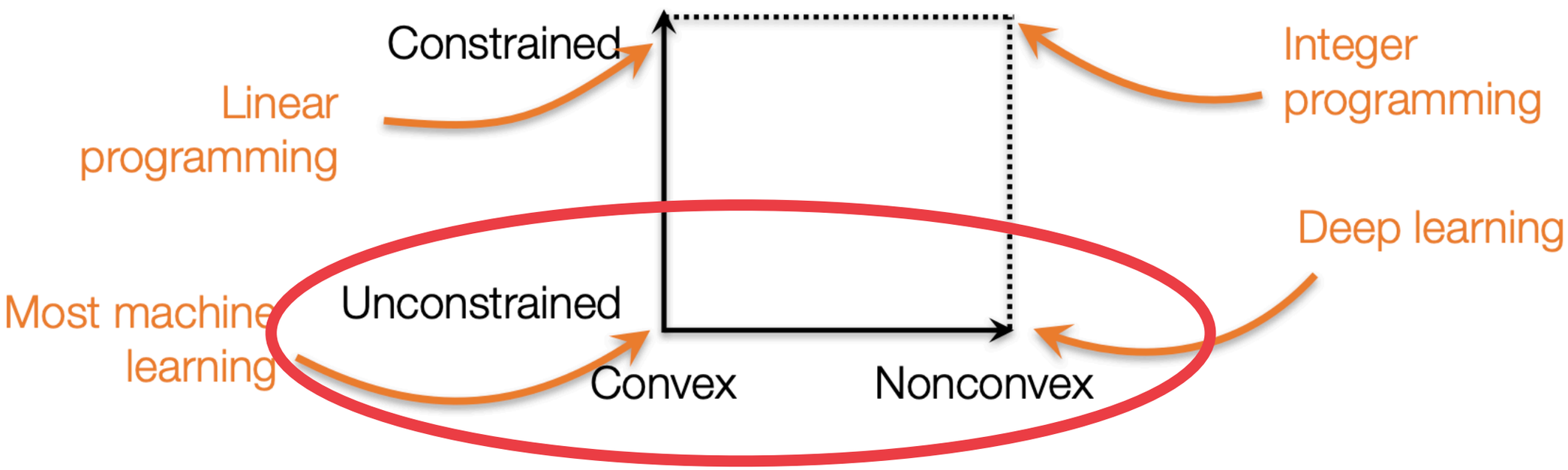
$$\hat{m}_{t+1} = \frac{m_{t+1}}{1 - \gamma_1^{t+1}}$$

$$\hat{g}_{t+1} = \frac{g_{t+1}}{1 - \gamma_2^{t+1}}$$

$$\theta_{t+1} = \theta_t - \frac{\alpha \hat{m}_{t+1}}{\sqrt{\hat{g}_{t+1}} + \epsilon}$$

GRADIENT-BASED OPTIMIZATION

ONE WORD FOR SEVERAL MEANINGS

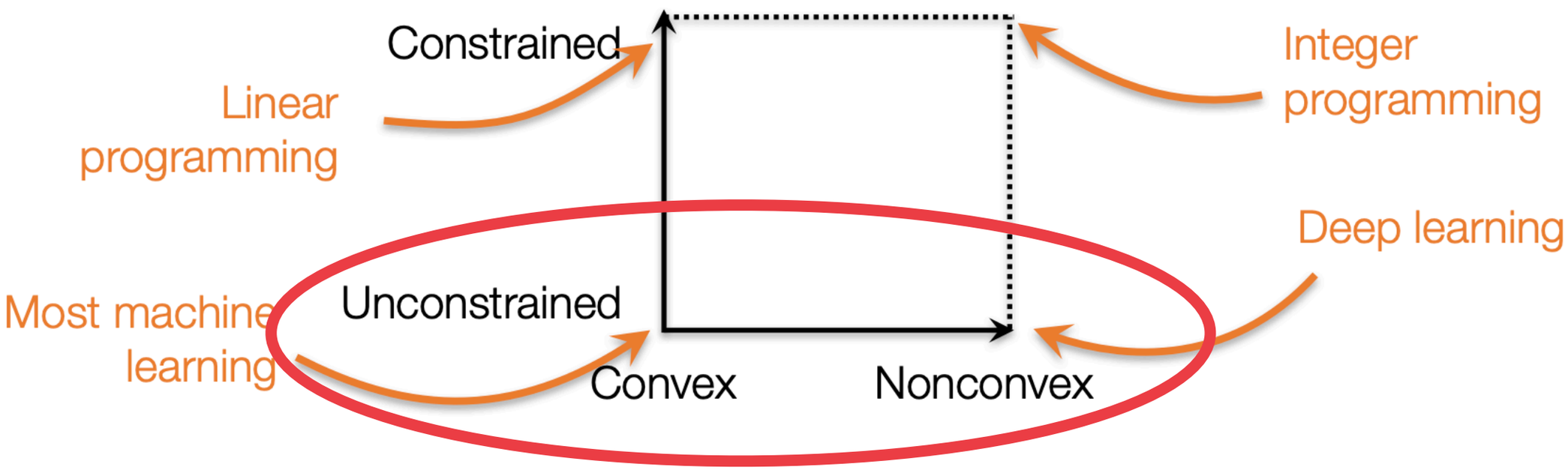


To recap *(topics BRIEFLY covered in this lecture)*

- Unconstrained and differentiable
  - Set derivative to be 0 → Closed form solution
  - 1st order method: gradient descent (for deep learning: SGD and alike)
  - 2nd order methods (if twice differentiable): Newton method, quasi-Newton methods (they approximate the Hessian, e.g. BFGS/L-BFGS)

GRADIENT-BASED OPTIMIZATION

ONE WORD FOR SEVERAL MEANINGS



To recap *(topics NOT covered in this lecture, partially covered in the next lectures)*

- Constrained and differentiable
  - Projected gradient descent (iteratively update the value of x while ensuring $x \in F$)
  - Lagrangian or KKT multipliers
  - Interior point method
- (Constrained) non-differentiable
  - ε-subgradient method, cutting plane method, metaheuristics, etc.

Questions?