MAD 1 Summary

Michael Van Huffel, Dino Colombo

5. August 2021

MAD 1 Summary created by *michavan@student.ethz.ch* && *dicolomb@student.ethz.ch*

This summary has been written based on the Lecture151-0431-00 S Models, Algorithms and Data: Introduction to Computing by Prof. Dr. Jens Honoré Walther and Dr. Georgios Arampatzis (Spring 21s). There is no guarantee for completeness and/or correctness regarding the content of this summary. This summary is basically a modified and a more completed version of the summary of Joshua Naef. Use it at your own discretion

0 Inhaltsverzeichnis

1 Linear Least Squares

We want to minimize error:

$$\|\mathbf{e}\|_2 = \sqrt{\sum_{i=1}^{N} e_i^2} = \sqrt{\sum_{i=1}^{N} (y_i - f(x_i))^2}$$

Choosing different norms yields different results.

N: #Data Points, M: #Free Parameters/Functions

1.1 Linear Least Squares:

Fit data $\{(x_i, y_i)\}_{i=1}^N$ with a function $f(x)$ which can be expressed as M linearly independent functions $\varphi_k(x)$,

$$f(x; \boldsymbol{w}) = \sum_{k=1}^{M} w_k \varphi_k(x)$$

where $\boldsymbol{w} = (w_1, w_2, \ldots, w_M)$ are the unknown weights. Functions can be nonlinear but **weights have to enter lineraly**.

Goal: find $\boldsymbol{w}$ s.t. error $E(\boldsymbol{w}) = \|\mathbf{e}(\boldsymbol{w})\|_2^2$ is minimised.

$$\boldsymbol{w}^\star = \arg\min_{\boldsymbol{w}} E(\boldsymbol{w}) \Rightarrow \frac{dE(\boldsymbol{w})}{d\boldsymbol{w}} = 0$$

Matrix notation leads us to: $A\boldsymbol{w} = \boldsymbol{y}$
$A \in \mathbb{R}^{N\times M}$, $\boldsymbol{w} \in \mathbb{R}^M$, $\boldsymbol{y} \in \mathbb{R}^N$

$$\begin{bmatrix} \varphi_1(x_1) & \varphi_2(x_1) & \ldots & \varphi_M(x_1) \\ \varphi_1(x_2) & \varphi_2(x_2) & \ldots & \varphi_M(x_2) \\ \vdots & \ddots & & \vdots \\ \vdots & & \ddots & \vdots \\ \varphi_1(x_N) & \varphi_2(x_N) & \ldots & \varphi_M(x_N) \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_M \end{bmatrix} = \begin{bmatrix} y1 \\ y_2 \\ \vdots \\ \vdots \\ y_N \end{bmatrix}$$

- M=N there exists unique solution $\boldsymbol{w} = A^{-1}\boldsymbol{y}$.
- M>N System is underdetermined and has infinitely many solutions
- M<N System is overdetermined. We can approximate a solution $A\boldsymbol{w} \approx \boldsymbol{y}$ with least squares requiring $E(\boldsymbol{w}) = \|\boldsymbol{y} - A\boldsymbol{w}\|_2^2$ be minimal.

$$\boldsymbol{w}^\star = (A^\top A)^{-1} A^\top \boldsymbol{y}$$

1.1.1 Solution for Linear Functions:

Consider set of data $\{x_i, y_i\}_{i=1}^N$ which we want to fit to $y = w_1 + w_2 x$

$$\begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ \vdots & \vdots \\ 1 & x_N \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} = \begin{bmatrix} y1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix}$$

$$w_1^\star = \frac{\sum_{i=1}^N x_i^2 \sum_{i=1}^N y_i - \sum_{i=1}^N x_i \sum_{i=1}^N x_i y_i}{N\sum_{i=1}^N x_i^2 - \left(\sum_{i=1}^N x_i\right)^2}$$

$$w_2^\star = \frac{N\sum_{i=1}^N x_i y_i - \sum_{i=1}^N x_i \sum_{i=1}^N y_i}{N\sum_{i=1}^N x_i^2 - \left(\sum_{i=1}^N x_i\right)^2}$$

Note: if you ad any point on the regression line this doesn't change.

1.1.2 Orthonormal Case:

An *orthogonal Matrix* has the following properties:

$$A^\top = A^{-1}; \qquad \|A\mathbf{u}\| = \|\mathbf{u}\|$$

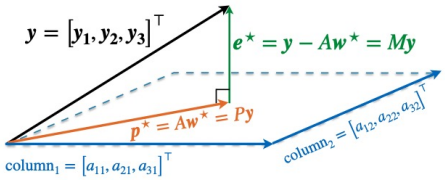
LSQ simplifies to: $\boldsymbol{w}^\star = A^\top \boldsymbol{y}$

Rewrite $y = w_1 + w_2 x$ as $y = w_1 + w_2(x - \bar{x})$. LSQ yields

$$w_1^\star = \frac{\sum y_i}{N}; \quad w_2^\star = \frac{\sum (x_i - \bar{x}) y_i}{\sum (x_i - \bar{x})^2}$$

1.2 Geometric Interpretation:

The columns of A span a M-dimensionl space. The least squares method finds the solution $\boldsymbol{w}^\star$ s.t. $A\boldsymbol{w}^\star$ is the projection of $\boldsymbol{y}$ on the column space of A. We observe that the residual $\mathbf{e} = \mathbf{y} - A\boldsymbol{w}^\star$ is perpendicular to range(A) and has minimal 2-norm ($\|e\| = \|\boldsymbol{y} - A\boldsymbol{w}\|$ is minimal for $\boldsymbol{w} = \boldsymbol{w}^\star$)



Geometric interpretation for $N = 3$ and $M = 2$

$\mathbf{e}^\star \in$ left nullspace A

$A\boldsymbol{w} = \boldsymbol{y}$ has a solution iff $\boldsymbol{y} \in$ range $A \Rightarrow \mathbf{e} = 0$

1.2.1 Projection Matrix P:

Closest point to $\boldsymbol{y}$ is $\boldsymbol{p} = A\boldsymbol{w}^\star = P\boldsymbol{y}$ (P is a projection Matrix with $P = P^\top$ (symmetric), $P^2 = P$ (idemptotent)).

$$P = A(A^\top A)^{-1} A^\top$$

We can write the error as $\boldsymbol{e} = \boldsymbol{y} - A\boldsymbol{w}^\star = \boldsymbol{y} - P\boldsymbol{y} = M\boldsymbol{y}$

$$M = \mathbb{I} - A(A^\top A)^{-1} A^\top$$

where M is also a projection matrix, projecting onto null($A^\top$) (space orthogonal to the column space of A), whereas P projects onto range(A): $P + M = \mathbb{I}$; $PM = 0$; $PA = A$; $MA = 0$

1.3 Numerical Solutions:

1.3.1 Condition Number:

Numerical solution is affected by round-off error. we define $\delta\boldsymbol{w} = \boldsymbol{w} - \tilde{\boldsymbol{w}}$ and $\delta\boldsymbol{y} = \boldsymbol{y} - \tilde{\boldsymbol{y}}$ where $\tilde{\boldsymbol{w}}, \tilde{\boldsymbol{y}}$ are the numerical solutions to LSQ and $A\tilde{\boldsymbol{w}} = \tilde{\boldsymbol{y}}$.

If $\|\delta\boldsymbol{y}\|$ is small is $\|\delta\boldsymbol{w}\|$ small as well? $\Rightarrow$ No.

Condition number:

$$\kappa(A) = \|A\|\|A^{-1}\| \qquad \kappa(A) \in [0, \infty)$$

We call a problem *well defined* if $\kappa(A)$ is not too large. In that case the computed solution will be close to the true solution.

Interpretation: *How close a Matrix is to being singular.*

Orthogonal Matrices: $\kappa(A) = 1$

$\|A\|_2 = \sqrt{\rho(A^\top A)}$ where ρ is the largest Eigenvalue

$\kappa_2(A^\top A) = \kappa_2(A)^2 \Rightarrow$ bad for LSQ

1.3.2 QR decomposition:

$Q \in \mathbb{R}^{N\times N}$ is orthogonal and $Q_1 \in \mathbb{R}^{N\times M}$ is a matrix with orthogonal columns and $R_1 \in \mathbb{R}^{M\times M}$ is upper triangle and invertible.

$$A = \underbrace{\begin{bmatrix} Q_1 & Q_2 \end{bmatrix}}_{Q} \underbrace{\begin{bmatrix} R_1 \\ 0 \end{bmatrix}}_{R} = Q_1 R_1 \qquad \boldsymbol{w} = R_1^{-1} Q_1^\top \boldsymbol{y}$$

R_1 is upper triangle $\rightarrow$ **no need to compute inverse**, just solve the Linear SoE

$$R_1\boldsymbol{w} = Q_1^\top \boldsymbol{y}, \quad \text{with: } \|R_1\| = \|R_1 Q_1\| = \|A\|$$

Condition of the matrices involved in the solution is the same as the condition of A

1.3.3 Singular Value Decomp (SVD):

The SVD of a Matrix $A \in \mathbb{R}^{N\times M}$ with $\text{rank}(A) = \rho$ decomposed A as,

$$A = \begin{bmatrix} U_r & U_n \end{bmatrix} \begin{bmatrix} S & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} V_r^\top \\ V_n^\top \end{bmatrix} = U\Sigma V^\top$$

$U \in \mathbb{R}^{N\times N}$ and $V \in \mathbb{R}^{M\times M}$ are orthogonal matrices. $\Sigma \in \mathbb{R}^{N\times M}$ is a diagonal matrix and $S = \text{diag}(\sigma_1, \dots, \sigma_\rho)$ where $\sigma_1 \geq \sigma_2 \geq \dots \sigma_\rho \geq 0$ with σ_i being the *singular values* of A. $\Sigma^+ = \text{diag}(S^{-1}, 0)$. *Moore-Penrose Pesudoinverse:*

$$A^+ = (A^\top A)^{-1} A^\top, \quad A^+ = V\Sigma^+ U^\top$$

$$\boldsymbol{w}^\star = V\Sigma^+ U^\top \boldsymbol{y}$$

$\Rightarrow \Sigma V^\top \boldsymbol{w}^\star = U^\top \boldsymbol{y}$ and $\kappa(\Sigma V^\top) = \kappa(A)$

2 Nonlinear Systems

2.1 Root Finding Problem:

Rewrite nonlin eq. $g(x) = h(x)$ as $g(x) - h(x) := f(x) = 0 \Rightarrow$ Transform eq. into root finding problem for $f(x)$:

$$f(x^\star) = 0$$
$$e_k = x_k - x^\star \ (\approx x_k - x_{k-1})$$

$\nexists$ General formula to find root of nonlinear function $\rightarrow$ Iterative methods which converge $x_0 \xrightarrow{h\to\infty} x^\star$.

2.1.1 Sensitivity and Conditioning:

If $|f(\tilde{x})| \approx 0$ *then* $|\tilde{x} - x^\star| \approx 0$*?* System (f) is **well-conditioned** when a small change in input causes a small change in the output. **Ill-conditioned** if small change in input causes a large change in output.

Condition number of root finding problem:

$$\kappa = \frac{1}{|f'(x^\star)|}$$

If $f'(x) = 0$ the problem is ill-conditioned. This is the case for roots with **multiplicity m** > 1.

2.1.2 Order of Convergence:

Sequence $\{x_k\}_{K=0}^{\infty}$ should converge to $x^\star$ asap. To quantify "*fast*"we define the **order of convergence** r and **rate of Convergence** C as:

$$\lim_{k\to\infty} \frac{|e_{k+1}|}{|e_k|^r} = C$$

- $r = 1$: if $C \in (0, 1)$ linear convergence. If $C = 0$ superlinear, $C = 1$ sublinear
- $r = 2$: quadratic convergence

2.2 Bisection Method:

Order of convergence: 1 and rate of convergence. 1/2

Number of iterations k to achieve required tolerance:

$$|e_k| = \text{tol} \Rightarrow \frac{b-a}{2^k} = \text{tol} \Rightarrow k = \log_2\left(\frac{b-a}{\text{tol}}\right)$$

Advantages:

- Certain to converge
- Only requires signs and not function values
- f does not need to be diff'able only continuous

Disadvantages:

- Convergence is slow
- Initial interval needs to be known beforehand
- Cant be easily generalized to higher dimensions

Example: Bisection pseudocode

Input:
a, b, (initial interval)
tol, (tolerance, minimum length of interval)
$k_{\max}$, (maximum number of iterations)
Output:
x_k, (approximate solution after k iterations)

Steps:
$k \leftarrow 1$
while $(b - a) >$ tol and $k < k_{\max}$ **do**
  $x_k \leftarrow (a + b)/2$
  if $\text{sign}(f(a)) = \text{sign}(f(x_k))$ **then**
    $a \leftarrow x_k$
  else
    $b \leftarrow x_k$
  end if
  $k \leftarrow k + 1$
end while

2.3 Newtons Method:

Function f continuous and differentiable at $x^\star$.

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)} \Rightarrow \underbrace{x_{k+1} = x_k - m\frac{f(x_k)}{f'(x_k)}}_{\text{roots with multiplicity } m>1}$$

Advantages:

- Can be extended to more dimensions
- Quadratic convergence

$$\lim_{k\to\infty} \frac{|e_{k+1}|}{|e_k|^2} = \lim_{k\to\infty} \frac{|f''(\xi_k)|}{2|f'(x_k)|} = \frac{|f''(x^\star)|}{2|f'(x^\star)|} = C < \infty$$

Disadvantages:

- Small changes in IV may change root
- **Root with multiplicity** $m > 1$ $(f'(x^\star) = 0)$**convergence rate only linear**
- **Cubic** convergence if $f'(x^\star) \neq 0, f''(x^\star) = 0$
- Not guaranteed to converge
- if $f'(x_k) = 0$ for some k we cannot proceed
- requires evaluation of both $f(x_k)$ and $f'(x_k)$

Example: Newton's 1D pseudocode

Input:
x_0, {initial condition}
tol, {tolerance: stop if $\|x_k - x_{k-1}\| <$ tol}
$k_{\max}$, {maximal number of iterations: stop if $k > k_{max}$}
Output:
x_k, {solution of $f(x_k) = 0$ within tolerance} (or a message if $k > k_{\max}$ reached)

Steps:
$k \leftarrow 1$
while $k \leq k_{\max}$ **do**
  Calculate $f(x_{k-1})$ and $f'(x_{k-1})$
  Update $x_k \leftarrow x_{k-1} - \frac{f(x_{k-1})}{f(x_{k-1})}$
  if $\|x_k - x_{k-1}\| <$ tol **then**
    break
  end if
  $k \leftarrow k + 1$
end while

2.4 Secant Method:

Modification of Newtons method by approximating $f'(x_k) \approx \frac{f(x_k)-f(x_{k-1})}{x_k-x_{k-1}}$

$$x_{k+1} = x_k - f(x_k)\frac{x_k - x_{k-1}}{f(x_k) - f(x_{k-1})}$$

r: $\varphi = \frac{1+\sqrt{5}}{2} \approx 1.618$ (simple roots).

Advantages:

- Avoid evaluation of derivative
- Only one function evaluation at each step
- Can be extended to more dimensions

Disadvantages:

- Convergence rate is not quadratic
- Needs two initial approximations

2.5 Order of Convergence:

Compute the error $e_k = x_k - x^\star \; \forall k$ for a function with known root $x^\star$. Then, evaluate the convergence rate r_k as a sequence which will converge to the theoretical r.

$$r \approx \frac{\log\left|\frac{e_{k+2}}{e_{k+1}}\right|}{\log\left|\frac{e_{k+1}}{e_k}\right|} \quad \text{log base 10}$$

3 Sets of equations

General systems of N non-linear functions $f_i(\boldsymbol{x})$, $i = 1, \ldots, N$, where $\boldsymbol{x} = (x_1, \ldots, x_M)^\top$ is a vector of M unknowns. Find $\boldsymbol{x}^\star$, st. $f_i(\boldsymbol{x}^\star) = 0 \quad i = 1, \ldots, M$

We define the *Jacobian Matrix* $J(\boldsymbol{x})$ with elements $J_{ij}(\boldsymbol{x}) = \frac{\partial f_i(\boldsymbol{x})}{\partial x_j}$

$$J(\boldsymbol{x}) = \begin{bmatrix} \frac{\partial f_1(\boldsymbol{x})}{\partial x_1} & \frac{\partial f_1(\boldsymbol{x})}{\partial x_2} & \cdots & \frac{\partial f_1(\boldsymbol{x})}{\partial x_M} \\ \frac{\partial f_2(\boldsymbol{x})}{\partial x_1} & \frac{\partial f_2(\boldsymbol{x})}{\partial x_2} & \cdots & \frac{\partial f_2(\boldsymbol{x})}{\partial x_M} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_N(\boldsymbol{x})}{\partial x_1} & \frac{\partial f_N(\boldsymbol{x})}{\partial x_2} & \cdots & \frac{\partial f_N(\boldsymbol{x})}{\partial x_M} \end{bmatrix}$$

3.1 Condition Number:

For a system of N non-linear equations and M unknowns the condition number of the root finding problem for the root $\boldsymbol{x}^\star$ of F is $\|J^{-1}(\boldsymbol{x}^\star)\|$

3.1.1 Newtons Method:

If we set $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \boldsymbol{z}$ we have to solve $A\boldsymbol{z} = \boldsymbol{b}$ at every step with $A = J(\boldsymbol{x}_k)$ and $\boldsymbol{b} = -F(\boldsymbol{x}_k)$

$$\boldsymbol{0} = F(\boldsymbol{x}^\star) \approx F(\boldsymbol{x}_k) + J(\boldsymbol{x}_k)(\boldsymbol{x}^\star - \boldsymbol{x}_k)$$
$$J(\boldsymbol{x}_k)(\boldsymbol{x}_{k+1} - \boldsymbol{x}_k) = -F(\boldsymbol{x}_k)$$

Netwon-Raphson Method ($N = M$):

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - J^{-1}(\boldsymbol{x}_k)F(\boldsymbol{x}_k) \qquad \begin{aligned} J(\boldsymbol{x}_k)\boldsymbol{z} &= -F(\boldsymbol{x}_k) \\ \boldsymbol{x}_{k+1} &= \boldsymbol{x}_k + \boldsymbol{z} \end{aligned}$$

Provided that the Jacobian is not singular the convergence rate is quadratic.

Pseudo-Newton Method ($N \neq M$):

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - J^{+}(\boldsymbol{x}_k)F(\boldsymbol{x}_k)$$

Assuming J always has full rank

Example: Newton's nD pseudocode

Input:

$\boldsymbol{x}_0$, vector of length N with initial approximation
tol, tolerance: stop if $\|\boldsymbol{x}_{k+1} - \boldsymbol{x}_k\| < \text{tol}$
$k_{\max}$, maximum number of iterations: stop if $k > k_{\max}$

Output:

$\boldsymbol{x}_k$, approximation of solution of $F(\boldsymbol{x}) = \boldsymbol{0}$ within tolerance tol or with $k = k_{\max}$ steps.

Steps:

```
k ← 0
while k ≤ k_max do
    Calculate F(x_k) and N × N matrix J(x_k)
    Solve the N × N linear system J(x_k) z = −F(x_k)
    x_{k+1} ← x_k + z
    if ||z|| < tol then
        break
    end if
    k ← k + 1
end while
```

3.2 Non-Linear Optimization:

Find $\boldsymbol{x}^\star = \arg\min_{\boldsymbol{x}} E(\boldsymbol{x})$ with $E : \mathbb{R}^M \to \mathbb{R}$

Maximizing $E(\boldsymbol{x}) \Leftrightarrow$ minimizing $-E(\boldsymbol{x})$

Sufficient condition for a crit point $\boldsymbol{x}^\star$ is $\nabla E(\boldsymbol{x}^\star) = 0 \Rightarrow$ if Minium Hessian matrix of E is positive semidefinite at x^*.

3.2.1 Newtons Method/Update Rule:

$F(\boldsymbol{x}) = \nabla E(\boldsymbol{x}) \overset{!}{=} 0 \rightarrow$minimization problem to a root finding problem.

$$J(F, \boldsymbol{x}) = \nabla^2 E(\boldsymbol{x}), \qquad \begin{aligned} \nabla^2 E(\boldsymbol{x}_k)\boldsymbol{z} &= -\nabla E(\boldsymbol{x}_k) \\ \boldsymbol{x}_{k+1} &= \boldsymbol{x}_k + \boldsymbol{z} \end{aligned}$$

4 Inter- and Extrapolation

4.1 Lagrange Interpolation:

We want a base polynomial which satisfies $l_i(x_j) = \delta_{ij}$. Create a function of order $N-1$ which perfectly fits N data points. If nodes x_k are equispaced over some interval $x \in [a, b]$, then the Lagrange interpolating function oscillates with greater amplitude near a and b than near the midpoint of the interval.

$$\underbrace{\ell_k(x) = \prod_{\substack{i=1 \\ i \neq k}}^{N} \frac{x - x_i}{x_k - x_i}}_{\text{Lagrange Functions}} \qquad \underbrace{f(x) = \sum_{k=1}^{N} y_k \ell_k(x)}_{\text{Lagrange Interpolator}}$$

Note: if we interpolate with Lagrange a polinomial (degree k) with $\#points \geq k+1$ we get a perfect fit of order k (as long as no noise)!

4.1.1 LSQ vs Interpolation:

Noise: LSQ robust to noise. Interpolation passes exactly through $(x_i, y_i) \forall i$. When fitting N noisy data points, the resulting function from the Lagrange interpolation is the same as the Least-Squares solution using a polynomial basis of degree $N-1$. **Note:** Lagrange function is dependant on x_i only.

Data points: Terms of higher order than order of data cancel out in the Lagrange interpolator. I.e. if data linear but 5 data points, lagrange will still be linear. **LSQ also robust to few data points.**

Stability: Hard to approx. loc. behaviour if interpolator used as global function. Oscillatory behaviour at the endpoints (higher order). Fluctuations in one region may affect the function over the whole domain.

4.2 Cubic Splines:

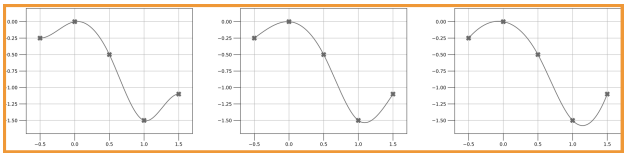
Idea: split interval into subdomains and the construct a piecewise approximation in each subdomain. Cubic functions will lead to piecewise linear f''. We have $4(N-1) = 4(N-1) - 2 + 2$ costraints.

- $f_i(x_i) = y_i, \quad i = 1, \dots, N-1$
- $f_i(x_{i+1}) = y_{i+1}, \quad i = 1, \dots, N-1$
- $f_i'(x_{i+1}) = f_{i+1}'(x_{i+1}), \quad i = 1, \dots, N-2$
- $f_i''(x_{i+1}) = f_{i+1}''(x_{i+1}), \quad i = 1, \dots, N-2$

And one out of these five more constraints:

- Clamping: $f'(x_1) = f'(x_N) = 0$
- Natural Spline: $f_1'' = f_N'' = 0$
- Parabolic Runout: $f_1'' = f_2''$ and $f_{N-1}'' = f_N''$
- Periodic funcions: $f_1'(x_1) = f_{N-1}'(x_N)$ and $f_1''(x_1) = f_{N-1}''(x_N)$ if function periodic with $T = x_N - x_1$.
- Not-a-knot: $f_2'''(x_1) = f_3'''(x_1)$ and $f_N'''(x_{N-1}) = f_{N-1}'''(x_{N-1})$

  Hint: use $f_i'''(x) = \frac{f_i'' - f_{i-1}''}{\Delta_{i-1}} = const.$



Data: $\{x_i, y_i\}_{i=1,\dots,N}$ (N Data). Let $f_i = f(x_i)$ be a cubic function in between $x_i < x < x_{i+1}$ with $\Delta_i = x_{i+1} - x_i$

$$\frac{\Delta_{i-1}}{6} f_{i-1}'' + \left(\frac{\Delta_{i-1} + \Delta_i}{3}\right) f_i'' + \frac{\Delta_i}{6} f_{i+1}'' \quad i = 2 \dots N-1$$
$$= \frac{y_{i+1} - y_i}{\Delta_i} - \frac{y_i - y_{i-1}}{\Delta_{i-1}}$$

$$f_i(x) = \left[\frac{y_{i+1} - y_i}{\Delta_i} - \left(f_{i+1}'' - f_i''\right)\frac{\Delta_i}{6}\right](x - x_i) \quad i = 1 \dots N-1$$
$$+ f_i'' \frac{(x_{i+1} - x)^3}{6\Delta_i} + f_{i+1}'' \frac{(x - x_i)^3}{6\Delta_i} + \left(y_i - f_i'' \frac{\Delta_i^2}{6}\right)$$

Tridiagonal matrix equation $Af = b$ where $f = [f_1'', f_2'', \dots, f_{N-1}'', f_N'']^\top$, b = column vector of known coefficient!

$$\begin{bmatrix} b_1 & c_1 & 0 & 0 \\ a_2 & b_2 & c_2 & 0 \\ 0 & a_3 & b_3 & c_3 \\ 0 & 0 & a_4 & b_4 \end{bmatrix} \cdot \begin{bmatrix} x1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ d_4 \end{bmatrix} \Rightarrow \begin{bmatrix} 1 & w_1 & 0 & 0 \\ 0 & 1 & w_2 & 0 \\ 0 & 0 & 1 & w_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} g_1 \\ g_2 \\ g_3 \\ g_4 \end{bmatrix}$$

with $w_1 = \frac{c_1}{b_1}, g_1 = \frac{d_1}{b_1}, g_N = \frac{d_N - a_N g_{N-1}}{b_N - a_N w_{N-1}}$ and for $i = 2, \dots, N-1$: $w_i = \frac{c_i}{b_i - a_i w_{i-1}}, g_i = \frac{d_i - a_i g_{i-1}}{b_i - a_i w_{i-1}}$

5 Numerical Integration

5.1 Integration Rules:

Rectangle/Midpoint Rule: (one point Gauss Quadrature)

$$I_{R_i} = f(x_i)\Delta_i, \quad I_{M_i} = f\left(\frac{x_i + x_{i+1}}{2}\right)\Delta_i$$

Trapezoidal Rule: $I_{T_i} = \frac{f(x_i) + f(x_{i+1})}{2}\Delta_i$

Simpson's Rule: $I_{S_i} = \frac{f(x_i) + 4f\left(\frac{x_i + x_{i+1}}{2}\right) + f(x_{i+1})}{6}\Delta_i$

5.1.1 Total Integrals:

Note: Δ_x represents small intervals and $I = \sum_{i=0}^{N-1} I_{Si/Ri/Mi/Ti}$

Rectangle/Midpoint Rule:

$$I \approx \Delta_x \sum_{i=0}^{N-1} f(x_i) \qquad I \approx \Delta_x \sum_{i=0}^{N-1} f\left(\frac{x_i + x_{i+1}}{2}\right)$$

Trapezoidal Rule:

$$I \approx \frac{\Delta_x}{2}\left(f(x_0) + 2\sum_{i=1}^{N-1} f(x_i) + f(x_N)\right)$$

Simpson's Rule:

$$\frac{\Delta_x}{3}\left(f(x_0) + 4\sum_{\substack{i=1 \\ i=\text{odd}}}^{N-1} f(x_i) + 2\sum_{\substack{i=2 \\ i=\text{even}}}^{N-2} f(x_i) + f(x_N)\right)$$

5.2 Newton-Cotes Formulas:

Note: $\Delta_i = b - a$ represents the big interval, $\Delta_x = \Delta_i / M$.

Use $M+1$ equidistant points in $[x_i, x_{i+1}] = [a, b]$ ($x_k = x_i + k \cdot h, k = 0, \dots, M$) and Lagrange interpolation. The Lagrange interpolant through $(x_k, f(x_k))$ is given by

$$I_i \approx \Delta_x \sum_{k=0}^{M} C_k^M f(x_k) \qquad C_k^M = \frac{1}{\Delta_i}\int_{x_i}^{x_{i+1}} l_k^M(x)dx$$

$$l_k^M(x) = \prod_{\substack{i=0 \\ i \neq k}}^{M} \frac{(x - x_i)}{(x_k - x_i)}$$

Properties of C_k^M: $\sum_{k=0}^{M} C_k^M = 1$; $C_k^M = C_{M-k}^M$

5.3 Error Analysis:

Find an upper bound for our error $E_{\text{rule},i} = I_i - I_{\text{rule},i}$

Rectangle Rule: Converges of factor 2 in a single interval.

Midpoint Rule: (third order convergence) Taylor series around $x_{i+1/2}$ yields:

$$E_{M_i} = \frac{1}{24} f''(x_{i+1/2})\Delta_i^3 + \mathcal{O}(\Delta_i^5) + \dots$$

Trapezoidal Rule: third order convergence

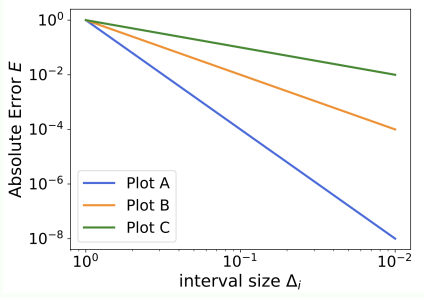
$$E_{T_i} = -\frac{1}{12} f''(x_{i+1/2})\Delta_i^3 + \mathcal{O}(\Delta_i^5) + \dots$$

Simpson's Rule: $I_{S_i} = \frac{2}{3} I_{M_i} + \frac{1}{3} I_{T_i}$

$$E_{S_i} = -\frac{1}{90} f^{(4)} \Delta_i^5 \approx \mathcal{O}(\Delta_i^5)$$

Order reduces by one for whole domain (e.g. 2nd for midpoint/trap; 4th for Simpson).

Example: Error of Si (A), Ti (B), Ri (C)



How many function evaluation to get a reduction of 1000 of the error? *Trapezoidal:* $N' = \sqrt{1000} \cdot N$ more evaluation. *Simpson:* $N' = \sqrt[4]{1000} \cdot N$ more evaluation

5.4 Richardson Extrapolation:

Quantity of interest G is discretized by some grid-spacing h (Compuer). $G \approx G(h)$. Since $h \ll 1$ and $G(h) \xrightarrow{h \to 0} G$ we can expand G(h) using a Taylor series.

$$G(h) = G + c_1 h^r + c_2 h^{2r} + \dots$$

c_i constants obtained from the expansion $\leftrightarrow$ Error terms we wish to eliminate. Taylor exapnsion of $G(h/2)$:

$$G(h/2) = G + (\frac{1}{2})^r c_1 h^r + (\frac{1}{4})^r c_2 h^{2r} + \dots$$

subtracting these two equations yields:

$$G_1(h) = 2G(h/2) - G(h) = G + c_2' h^{2r} + c_3' h^{3r} + \dots$$

Leading error term order $h^{2r} \Rightarrow G_1(h)$ much more accurate than $G(h)$ or $G(h/2)$ for $h \to 0$ with little added cost. This can be repeated. We can see that $G_n(h) = G + O(h^{r \cdot n})$.

$$G_n(h) = \frac{1}{2^{rn} - 1}(2^{rn} G_{n-1}(2^{(-1)^k} \cdot h) - G_{n-1}(h))$$

with $k = \begin{cases} 1 & \text{if } r > 0 \\ 0 & \text{if } r < 0 \end{cases}$

Error estimation ($r = 1$)**:**

$$\epsilon(h/2) = G(h/2) - G(h)$$

Example: Error extimation

$G(h)$ può rappresentare ad esempio $I(h), h$ = intevallo, $G(h/2) = I(h/2)$. In adaptive quadrature continuo a suddividere intervallo finchè le specificazioni non sono soddisfatte (applicando il calcolo dell'errore all'intervallo più suddiviso).

Esempio: voglio integrare funzione da 0 a 1($h = 1 - 0$) $\rightarrow \epsilon(h/2) = I[0, 0.5] + I[0.5, 1] - I[0, 1]$

5.5 Romberg Integration:

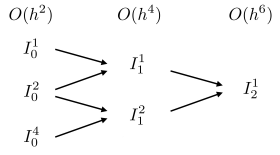
Improve accuracy of integration by using Richardson's extrapolation. Evalutate Trapeziodal in one interval, two, four, etc. $I_0^1, I_0^2, I_0^4, \dots$ In the calculation of I_0^n for n Intervals, **half of the needed functions have already been computed earlier**. **Note:** for trapezoidal: $I_0^n = \frac{b-a}{2n}(f(a) + f(b) + 2\sum_{i=2}^{n} f(x_i))$. (**When?** function evaluations are cheap and you want to achieve a certain error threshold.)

Numerical analysis of the error of trapezoidal rule and eliminating leading error term yields (**Note:** only even exponent of error! I_1^n is 4th order accurate):

$$I = I_0^n + c_1 h^2 + c_2 h^4 + c_3 h^6 \Rightarrow I_0^n = \dots$$

$$I_0^{2n} = I - c_1 \frac{h^2}{4} - c_2 \frac{h^4}{16} - c_3 \frac{h^6}{64}$$

$$\underbrace{I_k^n = \frac{4^k I_{k-1}^{2n} - I_{k-1}^n}{4^k - 1}}_{\text{(simpson: } 4^k \to 4^{k+1})}$$



Example: Rhomberg Pseudocode

Input:
function $f(x)$
interval boundaries a, b
number of iterations K
Output:
$I_K^1 =$ `integral`$[K, 0]$ approximation to the integral $\int_a^b f(x)\,dx$

Steps:

```
maxNumIntervals ← 2^K

// Precompute and store function evaluations
hmin ← (b − a)/maxNumIntervals
for i ← 0, ..., maxNumIntervals do
    fvalues[i] ← f(a + i * hmin)
end for

// Compute level 0 integrals
for r ← 0, ..., K do // refinement
    numIntervals ← 2^r
    step ← 2^(K−r) // step between two function evaluations for this refinement
    result ← 0
    for i ← step, 2 * step, 3 * step, ..., maxNumIntervals − step do
        result ← result + fvalues[i]
    end for
    // composite trapezoidal rule:
    integral[0, r] ← 0.5 (b−a)/numIntervals (fvalues[0] + fvalues[maxNumIntervals]
                                             +2 * result)
end for

//Advance to higher precision according to Romberg
for l ← 1, ..., K do //level
    for r ← 0, ..., K − l do // refinement
        integral[l, r] ← (4^l * integral[l−1, r+1] − integral[l−1, r]) / (4^l − 1)
    end for
end for
```

5.6 Adaptive Quadrature:

Romberg may achieve arbitrary accuracy but is not very efficient (function evaluations). Irregular function may waste evaluations in flat regions.
Note: Refining the grid locally does not change the order of accuracy of the underlying integration scheme. (**When?** can be used with unknown point where derivative non continue is)

Example: Adaptive Quadrature Pseudocode

Steps:

```
Subdivide the interval of the integration into sub-intervals
for all sub-intervals do
    Compute sub-integral, estimate the error with Richardson procedure described earlier.
    if accuracy is worse than desired then
        Subdivide the interval
    else
        Leave the interval untouched
    end if
end for
```

5.7 Gauss Quadrature:

We adapt the weights and abscissas of our quadrature rule to obtain better accuracy. (**When?** The function is very smooth. Function evaluations are costly)

5.7.1 Two-Point Gauss Quadrature:

The two point Gauss quadrature rule approximates I using

$$I = \int_a^b f(x)dx \approx c_1 f(x_1) + c_2 f(x_2)$$

where c_1, c_2, x_1 and x_2 are unknowns. They are evaluated by requiring an exact fit for an arbitrary third degree polynomial.
Note: this approximation is exact for functions f of degree 3.
Solving the nonlinear system of equations yields the two-point Gauss Quadrature rule:

$$I = \int_a^b f(x)dx \approx \frac{b-a}{2} \cdot \left\{ f\left[\left(\frac{b-a}{2}\right)\left(\frac{-1}{\sqrt{3}}\right) + \frac{b+a}{2} \right] + f\left[\left(\frac{b-a}{2}\right)\left(\frac{1}{\sqrt{3}}\right) + \frac{b+a}{2} \right] \right\}$$

5.7.2 n-Point Gauss Quadrature:

Goal: integrate $\int_a^b f(x)dx$.

- Change the area of integration from (a,b) with variable x to $(-1,1)$ with variable z.

$$z = \frac{2x-(a+b)}{b-a}$$

$I = \int_{-1}^{1} \frac{b-a}{2} f\left(\frac{b-a}{2}(z-1)+b\right) dz$

- The integration points z_i and the corresponding weights w_i can be read from a table.
- the Integral is then evaluated using

$$I \approx \frac{b-a}{2} \sum_{i=1}^{n} w_i f\left(\frac{b-a}{2}(z_i-1)+b\right)$$

- Error with n abscissas:

$$\varepsilon = \frac{2^{2n+1}(n!)^4}{(2n+1)(2n!)^3} f^{(2n)}(\xi)$$

Gauss Quadrature gives the best accuracy, in the sense of correctly integrating polynomials of highest possible order for a given number of function evaluations. The Abscissas vary from order to order resulting in new evaluations of the abscissas from scratch if the accuracy of the Quadrature is to be improved. Both ascissa and weights are degree of freedom. (In Newton Cotes only weights are degree of freedom!)

The abscissas are the zeros of the Legendre polynomial of degree n.

5.8 Quadrature in multiple dimensions:

Goal: Integrate $f : \mathbb{R}^d \to \mathbb{R}$ over a multi-dimensional domain $\Omega = \Omega_1 \times \dots \Omega_d$ i.e. $I = \int_\Omega f d\vec{x}$

$$I \overset{\text{Fubinis theorem}}{=} \int_\Omega \cdots \int_{\Omega_d} f(x^{(1)}, \dots, x^{(d)}) dx^{(d)} \dots dx^{(1)}$$
$$\approx \sum_{i_1=1}^{N_1} \cdots \sum_{i_d=1}^{N_d} \underbrace{w_{i_1} \dots w_{i_N}}_{=\underline{\underline{W}}_{i_1,\dots,i_d}} f(x_{i_1}^{(1)}, \dots, x_{i_d}^{(d)})$$

$\underline{\underline{W}} = \boldsymbol{w}\boldsymbol{w}^\top$ (2D) where $\boldsymbol{w}$ is the $N \times 1$ dimensional weight vector as specififed by section 5.1.1. In general, $\underline{\underline{W}} \in N_1 \times \cdots \times N_d$. In Notebook w_{i1} = vettore riga degli spazi in direzione 1.

5.9 Curse of Dimensionality:

Integrating in multiple dimensions leads to a rapid decrease in accuracy with increasing number of dimensions. The quadrature rules in 5.1 become of order $\mathcal{O}(M^{-s/d})$ where s is the order over a whole domain in 1 dimension and d corresponds to the dimensionality of the Integration with total number of quadrature points M.

5.10 Probability:

Probability that $x \in (a,b)$ in a PDF:

$$P(a \le X \le b) = \int_a^b p(x)dx$$

Uniform distribution $\mathcal{U}([a,b])$ and Normal distribution $\mathcal{N}(\mu, \sigma^2)$:

$$p_{\mathcal{U}}(x) = \begin{cases} \frac{1}{b-a} & x \in (a,b) \\ 0 & \text{otherwise} \end{cases}$$

$$p_{\mathcal{N}}(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

Expected value $\mathbb{E}$ over a domain Ω:

$$\mathbb{E}[x] = \langle x \rangle = \int_\Omega x p(x) dx \quad (= \mu \text{ mean})$$

$$\mathbb{E}[h(x)] = \int_\Omega h(x)p(x)dx$$

Note:

- $\mathbb{E}[aX + bY] = a\mathbb{E}[X] + b\mathbb{E}[Y]$
- **Discrete:** $\mathbb{E}[X] = \bar{x} = \sum_i x_i P(x_i) = \underset{\text{equally likely}}{\frac{1}{M}\sum_i^M p_i}$
- **Variance:** $\text{Var}[X] = \sigma^2[X] = \mathbb{E}[X^2] - \mathbb{E}[X]^2 = \frac{1}{n}\sum_{i=1}^n (x_i - \mu)^2$

5.10.1 Inverse Transform Sampling:

Solve for x. We can thus sample any distribution from a uniform distribution $\Rightarrow$ $F(x) = \int_0^x p(x)dx = u$

5.10.2 Rejection Sampling:

Generate samples from $p(x)$ from a simple distribution function $h(x)$ from which we already know how to generate samples. $h(x)$ has to bound $p(x)$. $p(x) < \lambda p(x)$

1. draw random sample x for $h(x)$
2. draw uniform random number $u \in [0,1]$
3. accept x if $u < \frac{p(x)}{\lambda h(x)}$, else reject x
4. $I_{MC} = \frac{\#acceptedsampling}{M} \cdot |\Omega_{\lambda h(x)}|$

5.10.3 Importance Sampling:

"Bad"fitting $p(x)$ might waste a lot of evaluations. We want to draw samples x from probability $w(x)$ which fits better. We compensate for the bias by normalizing $p(x)$ by $w(x)$ and thus sample $p(x)/w(x)$.

$$\langle f \rangle_p = \int_a^b f(x)\frac{p(x)}{w(x)}w(x)dx \approx \frac{1}{M}\sum_{i=1}^M f(x_i)\frac{p(x_i)}{w(x_i)}$$

With each x_i being sampled from distribution $w(x)$.

5.11 Monte Carlo:

1. Sample points $\boldsymbol{x}_i$ from a uniform distribution and evaluate integrand f to get $f(\boldsymbol{x}_i)$.
2. Store M (= #sampling), the sum of the values and the sum of squares
$$M, \quad \sum_{i=1}^M f(\boldsymbol{x}_i), \quad \sum_{i=1}^M f(\boldsymbol{x}_i)^2$$
3. Compute mean as the estimate of the expectation (**normalized integral**, Ω = sampling space)
$$\frac{I}{|\Omega|} = \langle f \rangle \approx \langle f \rangle_M = \frac{1}{M}\sum_{i=1}^M f(\boldsymbol{x}_i)$$
4. Estimate the variance using the unbiased sample variance:
$$\text{Var}[f] \approx \frac{M}{M-1}\left(\underbrace{\frac{1}{M}\sum_{i=1}^M f(\boldsymbol{x}_i)^2}_{=\langle f^2 \rangle_M} - \langle f \rangle_M^2\right)$$
5. Estimate error
$$\varepsilon_M = \sqrt{\text{Var}[\langle f_M \rangle]} = \sqrt{\frac{\text{Var}[f]}{M}} \propto \mathcal{O}\left(M^{-1/2}\right) \propto \Omega$$

and $\text{Var}[\langle f \rangle_M] = \frac{1}{M^2}\sum_{i,j=1}^M (\mathbb{E}[f(\boldsymbol{x}_i)f(\boldsymbol{x}_j)] - \langle f \rangle^2) = \begin{cases} 0, i \neq j \\ \frac{\text{Var}[f]}{M} \end{cases}$

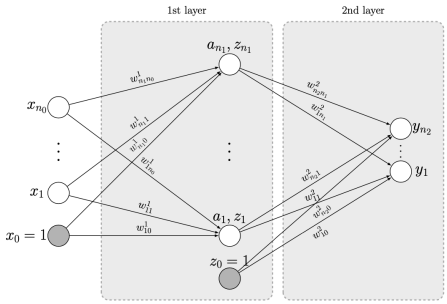
6 Neural Networks

Function $\boldsymbol{y}(\cdot, \boldsymbol{w}) : \mathbb{R}^{n_0} \to \mathbb{R}^{n_L}$ parametrized by weights $\boldsymbol{w}$.

6.1 2-Layer NN:

Notation for weights is w_{ji}^{ℓ} with destination node j and source node i in layer level ℓ. We denote the input of the layer ℓ and node j as a_j^{ℓ}. Define Matrix W^{ℓ} with $W_{ij}^{\ell} = w_{ij}^{\ell}$. Activation function acts elementwise on the vectors. So fir each layer:

1. Input x_i is weighted by w_i
2. Summed
3. Activation function φ is applied (2 layer = output)

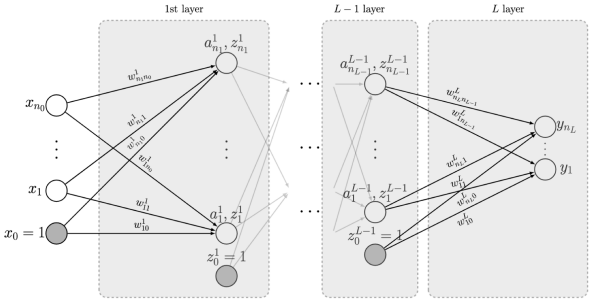


Map Input to First Layer: $a_j^1 = \sum_{i=0}^{n_0} w_{ji}^1 \cdot x_i, z_j^1 = \varphi_1(a_j^1)$

Map Input to Output: $a_j^2 = \sum_{i=0}^{n_1} w_{ji}^2 \cdot z_i^1, y_j = z_j^2 = \varphi_2(a_j^2)$

Compact Notation: $\boldsymbol{y}(\boldsymbol{x}; \boldsymbol{w}) = \varphi_2\Big(W^2 \varphi_1\Big(W^1 \boldsymbol{x}\Big)\Big)$

6.2 L-layer NN:



Input layer	$a_j^1 = \sum_{i=0}^{n_0} w_{ij}^1 x_i,\ j \in \{1, \dots, n_1\},\ z_j^1 = \varphi_1(a_j^1)$
Middle layers	$a_j^k = \sum_{i=0}^{n_{k-1}} w_{ij}^k z_i^{k-1},\ j \in \{1, \dots, n_k\},\ z_j^k = \varphi_k(a_j^k)$
Output layer	$a_j^L = \sum_{i=0}^{n_{L-1}} w_{ij}^L z_i^{L-1},\ j \in \{1, \dots, n_L\},\ y_j = \varphi_L(a_j^L)$

$$\boldsymbol{y}(\boldsymbol{x}; \boldsymbol{w}) = \varphi_L\Big(W^L \varphi_{L-1}\Big(\dots W^2 \varphi_1\big(W^1 \boldsymbol{x}\big)\Big)\Big)$$

Attention: Increasing # hidden nodes/layer in a feed-forward NN improves representation ability of the NN. Generalization performance is not improved since the NN is more prone to overfitting (number of weights ↑ and NN may memorize training data-set).

6.3 Activation Function:

Functions: $\varphi(x) = \begin{cases} 1, x \geq 0 \\ 0, \text{ else} \end{cases}$ (Heavyside), $\varphi(x) = \tanh x$,

$\varphi(x) = \frac{1}{1+e^{-x}}$ (Logistic), $\varphi(x) = \begin{cases} x, x \geq 0 \\ 0, \text{ else} \end{cases}$ (ReLu)

Introduce nonlinearity into the NN. Otherwise, the output of the NN would solely be a lin. comb. of the inputs. Smooth functions are preferred since they are differentiable (needed for backpropagation)

6.4 Backpropagation/Gradient Descendent:

6.4.1 Learning (Training):

Goal: update the weights w such that the output y_n given an input x_n matches a target $\hat{y}_n$. We want to find $\boldsymbol{w}^{\star} = \arg\min E(\boldsymbol{w})$ using gradient descent $\Big($stochastic gradient descent: Change $E(\boldsymbol{w}^{(k)})$ to $E_n(\boldsymbol{w}^{(k)})$/batchSGD: Change $E(\boldsymbol{w}^{(k)})$ to $\sum_{n \in \mathcal{I}} E_n(\boldsymbol{w}^{(k)})\Big)$.
Given: dataset $d = \{\boldsymbol{x}_n, \hat{\boldsymbol{y}}_n\}$, $n = 1, \dots, N$ **Steps:**

1. Build a model $y(x_n, w)$, initialize the weights $\boldsymbol{w} = \{w^1, \dots, w^L\}$
2. Perform the forward pass, i.e. produce the output y_n for all x_n in the dataset.
3. Compute the loss with respect to the target:

$$E_n = \frac{1}{2}|\hat{\boldsymbol{y}}_n - \boldsymbol{y}(\boldsymbol{x}_n, \boldsymbol{w})|^2 \Rightarrow E(\boldsymbol{w}) = \sum_{n=1}^{N} E_n(\boldsymbol{w})$$

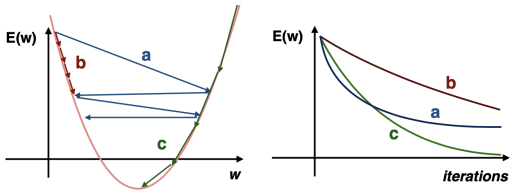
4. Perform the backward pass, i.e. **update the weights**
5. Repeat until minimum reached.

6.4.2 Weights updating:

We update the weights using gradient descend (GD):

$$\boldsymbol{w}^{(k+1)} = \boldsymbol{w}^{(k)} - \eta \nabla_{\boldsymbol{w}} E(\boldsymbol{w}^{(k)})$$

If the learning rate is too low, the algorithm will need maybe too many iterations to reach the minimum. On the other hand, if the learning rate is too high, we might fail to reach the minimum and oscillate between suboptimal values. The right learning rate is one of the crucial hyper-parameters to be dealt with in deep learning. In the following picture you can see the relationship between the error and the number of iterations for different choices of the learning rate η. Case (a) corresponds to a very high learning rate, case (b) to a very slow η and case (c) to a desired value of η.



To minimize E we have so to compute the derivative w.r.t w_{ji}^{ℓ} using chain rule (**Note:** $\frac{\partial x_i}{\partial x_j} = \delta_{ij}$ && $\frac{\partial w_{ij}}{\partial w_{kl}} = \delta_{ik}\delta_{jl}$)

$$\frac{\partial E_n}{\partial w_{ij}^k} = \frac{\partial E_n}{\partial a_j^k}\frac{\partial a_j^k}{\partial w_{ij}^k} = \delta_j^k \frac{\partial a_j^k}{\partial w_{ij}^k} = \delta_j^k z_i^{k-1} = z_i^{k-1}\varphi'(a_j^k)\sum_{i=1}^{n_{k+1}} w_{ji}^{k+1}\delta_i^{k+1}$$

$$\begin{cases} \frac{\partial a_j^k}{\partial w_{ij}^k} = \frac{\partial}{\partial w_{ij}^k}\left(\sum_{i'=1}^{n_{k-1}} w_{i'j}^k z_{i'}^{k-1}\right) = z_i^{k-1}, \\ \delta_j^k = \frac{\partial E_n}{\partial a_j^k} = \sum_{i=1}^{n_{k+1}} \frac{\partial E_n}{\partial a_i^{k+1}}\frac{\partial a_i^{k+1}}{\partial a_j^k} = \sum_{i=1}^{n_{k+1}} \delta_i^{k+1}\frac{\partial a_i^{k+1}}{\partial a_j^k} \\ \frac{\partial a_i^{k+1}}{\partial a_j^k} = \frac{\partial}{\partial a_j^k}\left(\sum_{j'=1}^{n_k} w_{j'i}^{k+1} z_{j'}^k\right) = \frac{\partial}{\partial a_j^k}\left(\sum_{j'=1}^{n_k} w_{j'i}^{k+1}\varphi(a_{j'}^k)\right) = w_{ji}^{k+1}\varphi'(a_j^k) \\ \delta_j^k = \sum_{i=1}^{h_{k+s1}} \delta_i^{k+1}\frac{\partial a_i^{k+1}}{\partial a_j^k} = \sum_{i=1}^{n_{k+1}} w_{ji}^{k+1}\delta_i^{k+1}\sigma'(z_j^k) = \\ = \sigma'(z_j^k)\sum_{i=1}^{n_{k+1}} w_{ji}^{k+1}\delta_i^{k+1} \end{cases}$$

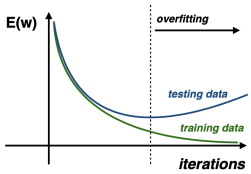
The last step of the recursion is the derivative of the loss w.r.t the network output i.e the derivative of the loss function: $\delta^L = \frac{\partial E_n}{\partial y_i} = \frac{\partial}{\partial y_i}|\hat{y}_n - y(x_n)|^2$. We observe so that at the last layer, the computation of the gradients $\frac{\partial E_n}{\partial w_{ji}}$ and $\frac{\partial E_n}{\partial a_j}$ do not depend on the neural network.

6.5 Overfitting:

A model with N free parameters should be able to fit exactly N data points. However, if the model passes exactly through the points it is unlikely to generalize efficiently since it fits the behaviour of noise. If too few parameters are used, the model might be forced to ignore meaningful data. **Bias-Variance Trade-off**: a reliable estimate of a biased model or a poor estimate of a model that is capable of a better fit.
This behaviour of overfitting can be avoided, if we require the error of data and fit always to be higher than zero. We can achieve this, by splitting our data set in two subsets: (a) a training set, which consists of, say, 90% of the data and (b) a test set, which consists in that case of 10% of the data. Both sets are used in the SGD algorithm, i.e the error in every iteration is computed using both the training and test sets.
Generalization is the property of a learning system to approximate the target output values for inputs that are not included in the training data-set.

If we plot the evolution of the error over the course of iterations of the SGD algorithm, we observe that:

- The error for the training set is being continuously reduced
- The error for the test set will reduce and after one point increase again ⇒ Termination point.
- To make our result statistically independent → follow the same procedure multiple times, selecting a different training and test set.
- Termination point of the algorithm is determined from the curve of the mean error over all the realizations.



6.6 Artificial Neural Network Training Loop:

Example: ANN Pseudocode ⇒ COVID-19 spread case

Input:
X, {Input dataset}
Y, {Target dataset}
n_batch, { batch size}
n_epochs, {number of training epochs}
η, {learning rate}
Output:
W, {weights} or $y = f_{ann}(x)$, {the mapping}

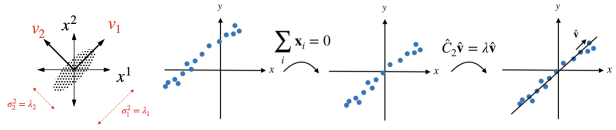
Steps:
```
X_train, X_test = split_dataset(X) Split input
Y_train, Y_test = split_dataset(Y) Split output
testing_loss = [ ]
for i in range(n_epochs) do Epoch loop
    X_train,Y_train = shuffle(X_train,Y_train) Shuffle dataset
    for j in range(N // n_batch) do Batch loop
        X_batch,Y_batch = get_batch(X_batch,Y_batch,j, n_batch) Get batch of

        out = forward_pass(X_batch)
        L = compute_loss(out,Y_batch)
        grad = compute_gradient(L,W)
        update_weigths(η*grad) update weight + learning rate
    end for
    out_test = forward_pass(X_test) Test output
    L_test = compute_loss(out_test,Y_test) Test loss
    testing_loss.append( L_test)
    if testing_loss[i] > testing_loss[i-1] then Check the loss on the test set for overfitting.

        Stop training
    end if
end for
```

Assume you have implemented a functioning artificial neural network (ANN) model with the following functions.
You are also given the input dataset $X = [x_1, x_2, \dots, x_N]$ where $x_i \in \mathbb{R}^d$ and the target dataset $Y = [y_1, y_2, \dots, y_N]$ where $y_i \in \mathbb{R}^d$. The goal of the ANN is to learn the mapping $y = f_{ANN}(x)$ between the input and the target.

- `forward.pass (x)`: Given an input x, returns the output y through the neural network.
- `compute.loss(x,y)`: Computes and returns the loss (L2 norm in this case) between 2 entries x and y.
- `compute.gradients(x,y)`: Computes the gradient of x with respect to y, i.e. $\frac{\partial x}{\partial y}$.
- `update.weights(x)`: Updates the weights of the model by x, i.e. $w^{k+1} = w^k + x$.
- `split.dataset(X,r)`: Given a dataset X of shape $[N, n.dim]$, where N is the number of samples, and $n.dim$ the dimension of a sample, splits the dataset into 2 subsets of relative size r and $1 - r$.
- `shuffle(X,Y)`: Shuffles the content of X and Y.
- `get.batch(X,i,n.batch)`: Returns the i-th batch of size $n.batch$ from the dataset X.

7 Dimensionality Reduction

7.1 Principal Component Analysis:



Linear transformation onto a subspace explaining the most variance (minimize reconstruction loss). Given dataset $X^\top = (\boldsymbol{x}_1, \dots, \boldsymbol{x}_N) \in \mathbb{R}^{D\times N}$ with N Vectors of D elements $\boldsymbol{x}_n \in \mathbb{R}^D$, $n = 1, 2, \dots, N$. Solving for the first principal component (direction in $\boldsymbol{v}_1^\star \in \mathbb{R}^D$ s.t. variance of projected data is maximized) yields $\sigma_1^{\star 2} = \lambda_1^\star \rightarrow$ Maximum of objective function correspond to the EigVec of the max EigVal.

Recipe:

1. Create centered data matrix (i.e. subtract mean singolarmente per ogni dimensione D)
$$\tilde{X}^\top \in \mathbb{R}^{D\times N}$$
2. Construct Covariance Matrix $C = \frac{1}{N-1}\tilde{X}^\top\tilde{X}$, $C \in \mathbb{R}^{D\times D}$
3. Perform eigenvector decomposition of C. (**Note:** $||v_i||_2 = 1$)
4. Sort EigVec in decr. EigVal order $\lambda_1 \geq \lambda_2 \geq \dots \geq \lambda_D$ to construct $V_r = (\boldsymbol{v}_1, \dots, \boldsymbol{v}_r) \in \mathbb{R}^{D\times r}$ for the transformation
$$Y_r = V_r^\top \tilde{X}^\top, \quad Y_r \in \mathbb{R}^{r\times N}$$
Using the principal component $V_r = v_1^T$, the data can be projected on a one-dimensional manifold: $Z = \tilde{X} \cdot v_1^T$ with mean $\bar{Z} = 0$. The totale explained variance in the reduced order subspace is $Var[Z] = \sigma_Z^2 = \mathbb{E}[(Z - \bar{Z})^2] = v_1^T C v_1 = \frac{1}{N}\sum(componentiZ)^2$
5. $\left(\text{Reconstruction: } \hat{X} = \tilde{X}V_rV_r^\top\right)$

Percentage of retained variance: $\text{ratio} = \frac{\sum_{i=1}^r \lambda_i}{\sum_{i=1}^D \lambda_i} = \frac{\sigma_Z^2}{\sigma_{\tilde{D}}^2}$

Note:

1. The variance in the original D-dimensional space (for a set of points) is $Var[X] = \frac{1}{N}\begin{bmatrix}\sum_{i=1}^N x_{i,1}\\ \sum_{i=1}^N x_{i,2}\\ \dots \\ \sum_{i=1}^N x_{i,D}\end{bmatrix} = \begin{bmatrix}\sigma_1^2\\ \sigma_2^2\\ \dots\\ \sigma_D^2\end{bmatrix} \Rightarrow$ $\sigma_{\tilde{D}}^2 = \sigma_1^2 + \sigma_2^2 + \dots + \sigma_D^2$ or:
2. The variance in the original D-dimensional space (for a set of points) can be also calculated as: $\sigma_{\tilde{D}}^2 = Spur[C]$, where $Spur[C]$ is the traccia of the covariance matrix.