MAD Formelsammlung

Jorit Geurts jgeurts@student.ethz.ch

Version: 23. August 2021

1 Data Fitting

1.1 Parametric Function Fitting

Given: Data $\{(x_i, y_i)\}_{i=1}^N$, **Fit:** Function $f(x)$, $f(x_i) \approx y_i$.

- **Fitting:** Construct low order Model. Doesn't go through all data points. We assume data has errors.
- **Interpolation:** Fitting a curve to descrete datapoints, to get estimates of datapoint in between
- **Extrapolation:** Fitting a functioin to descrete data, to estimate a trend (data outisde of the scope)
- Interpolation and Extrapolation: We assume data has no error

2 Linear Least Squares

Given: Data $\{(x_i, y_i)\}_{i=1}^N$. → Fit a **Function** $f(x)$.
$\varphi_k(x)$ **are M linearly independent functions**.

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

$\boldsymbol{w} = (w_1, ..., w_M)^T$ are unknown weights we have to find.
$\varphi_k(x)$ are the basis functions, where typically $M \ll N$.
Some typical basis functions:

$$\varphi_k(x) = x^{k-1}, \quad \varphi_k(x) = \cos((k-1)x)$$

$$\varphi_k(x) = e^{\beta_k x}, \quad \varphi_k(x) = 1 - \frac{x - x_k}{\delta}$$

The name Linear Least Squares comes from the fact that the unknown parameter w_k comes in linearly and not that we fit a linear function.

Goal: Find $\boldsymbol{w}$ that **minimizes** the Error Function.

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

$$\boldsymbol{w}^* = \arg\min_{w} E(\boldsymbol{w})$$

2.1 Matrix Formulation

$$\overbrace{\begin{bmatrix} \varphi_1(x_1) & \varphi_2(x_1) & \dots & \varphi_M(x_1) \\ \varphi_1(x_2) & \varphi_2(x_2) & \dots & \varphi_M(x_2) \\ \vdots & \vdots & \vdots & \vdots \\ \varphi_1(x_N) & \varphi_2(x_N) & \dots & \varphi_M(x_N) \end{bmatrix}}^{A \in \mathbb{R}^{N \times M}\text{, Regression or Least Square Matrix}} \overbrace{\begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_M \end{bmatrix}}^{\boldsymbol{w} \in \mathbb{R}^M} = \overbrace{\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix}}^{\boldsymbol{y} \in \mathbb{R}^N}$$

We need to solve the above equation for $\boldsymbol{w}$.

- **M = N:** (Easy case)
  The Matrix A is square. We get the solution $\boldsymbol{w} = A^{-1}\boldsymbol{y}$
- **M > N:** (Not interesting)
  System is underdetemined and has none or infinite solutions.
- **M < N:** (Most of the time its this case)
  The system is overdetermined. We can seek an approximate solution $A\boldsymbol{w} \approx \boldsymbol{y}$ with the least squares method by requiring that $E(\boldsymbol{w}) = \|\boldsymbol{y} - A\boldsymbol{w}\|_2^2$

2.1.1 Solution for M < N

Derivation of Error function $\left(\frac{\partial E}{\partial \boldsymbol{w}}\right)$ → normal equation:

Normal equation:	**Solution for $\boldsymbol{w}$:**
$A^T A\boldsymbol{w} = A^T\boldsymbol{y}$	$\boldsymbol{w}^* = \left(A^T A\right)^{-1} A^T y$

φ_k linearly independent → $A^T A$ symmetric and positive definite → solution for $\boldsymbol{w}$.

2.1.2 Solution for Linear Function

Fit data to $f(x) = w_1 + w_2 x$.

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

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

2.1.3 Special case: orthogonal case

When the columns $\boldsymbol{a}_i$ of $A \in \mathbb{R}^{N \times M}$ are orthogonal.

Condition:	**Therefore:**	**Solution for $\boldsymbol{w}$:**
$\boldsymbol{a}_i \cdot \boldsymbol{a}_j = \delta_{ij}$	$A^T A = I$	$\boldsymbol{w}^* = A^T\boldsymbol{y}$

2.2 Geometric interpretation

The columns $\boldsymbol{a}_i$ of $A \in \mathbb{R}^{N \times M}$ create a M-dimensional space. The solution $A\boldsymbol{w}^*$ is a projection of $\boldsymbol{y}$ onto that space spanned by A. The residual error is perpendicular to that space.
The projected Vector:

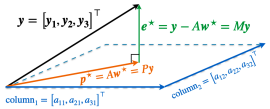
$$\boldsymbol{p}^* = A\boldsymbol{w}^* = A(A^T A)^{-1} A^T\boldsymbol{y} = P\boldsymbol{y}$$

Projection Matrix:

$$P = A(A^T A)^{-1} A^T$$

Properties of the projection matrix:
It is symmetric: $P = P^T$ It is idempotent: $P = P^2$
Same goes for the error:

$$\boldsymbol{e}^* = (I - A(A^T A)^{-1} A^T)\boldsymbol{y} = M\boldsymbol{y}$$



We see that:

$$P + M = I, \quad PM = 0$$

$$PA = A, \quad MA = 0$$

From linear algebra that:

$$\boldsymbol{y} = \boldsymbol{y}_r + \boldsymbol{y}_n = (P + M)\boldsymbol{y}$$

2.3 Algorithms

Algorithms without the use if $A^T A$:

2.3.1 QR-Decomposition

$A = QR = [Q_1 \quad Q_2]\,[R_1 \quad 0]^T$

$\boxed{\mathbf{w}^\star = R_1^{-1} Q_1^T \mathbf{y}}$ $\kappa(A) = \kappa(R_1)$

$A \in \mathbb{R}^{N \times M}$
$Q \in \mathbb{R}^{N \times N}$
$R \in \mathbb{R}^{N \times M}$

R_1 is upper triangle → dont inverse, solve LGS backwards

2.3.2 Singular Value Decomposition

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

Moore-Penrose Pseudo-Inverse:

$$A^+ = V \cdot \Sigma^+ \cdot U^T \qquad \Sigma = \begin{bmatrix} S & 0 \\ 0 & 0 \end{bmatrix} \to \Sigma^+ = \begin{bmatrix} S^{-1} & 0 \\ 0 & 0 \end{bmatrix}$$

Pseudo-Inverse of Σ:

$$\Sigma = \operatorname{diag}(\sigma_1, \sigma_2 \dots, 0), \quad \Sigma^+ = \operatorname{diag}\left(\sigma_1^{-1}, \sigma_2^{-1}, \dots, 0\right)$$

$\boxed{\mathbf{w}^\star = V\Sigma^+ U^T \mathbf{y}}$ $\kappa(A) = \kappa\left(\Sigma V^\top\right)$

2.4 Facts About Linear Least Squares

- (+) quit robust against outliers
- (-) numerical solutions allways have rounding errors, we cant guarantee, that the rounding error has siginificant impact on the result → Condition Number
- Normal Equation: $\kappa_2(A^T A) = \kappa_2(A)^2$
- *the normal equations transform the initial problem into a linear system with a square matrix*
- *The second derivative is:* $\nabla^2 E(\mathbf{w}) = 2A^T A$ which is symmetric & positive definite ⇒ $\mathbf{w}^*$ is a minimum

3 Non Linear Systems

Solve Non Linear Equation → Find roots of Non Linear Equation:

$$g(x) = h(x) \quad \to \quad g(x) - h(x) = f(x) = 0$$

Sensitivity: If $|f(\tilde{x})| \approx 0$, does this mean that $|\tilde{x} - x^*| \approx 0$?

Root Finding Problem	**f(x) finding Problem**
y (=0) given, searching x; $x = f^{-1}(y)$	x given, searching y; $y = f(x)$
Well Conditioned: small δx → big $\delta f(x)$	Well Conditioned: small δx → small $\delta f(x)$
Ill Conditioned: small δx → small $\delta f(x)$	Ill Conditioned: small δx → big $\delta f(x)$
Condition Number: $\kappa = \frac{\lvert\delta x\rvert}{\lvert\delta y\rvert} = \frac{1}{\lvert f'(x^*)\rvert}$	**Condition Number:** $\kappa = \frac{\lvert\delta y\rvert}{\lvert\delta x\rvert} = \lvert f'(x^*)\rvert$

$f'(x^*) = 0$, ill conditioned = roots of multiplicity $m > 1$.

$$\kappa_{(f(x)-finding)} = \frac{|\delta y|}{|\delta x|} = \frac{|f(x+\delta x) - f(x)|}{|\delta x|} \approx |f'(x)|$$

3.0.1 Order of Convergence

With $e_k = x_k - x^\star$, if $x_k \overset{k\to\infty}{=} x^\star$, there exists:

$$\lim_{k\to\infty} \frac{|e_{k+1}|}{|e_k|^r} = C \quad \text{with} \quad \begin{cases} r, & \text{Order of convergence} \\ C, & \text{Rate of convergence} \end{cases}$$

- $r = 1$: if $0 < C < 1$ linear, $C = 0$ superlinear, $C = 1$ sublinear.
- $r = 2$: quadratic convergence.
- $r \approx \log\left|\frac{e_{k+2}}{e_{k+1}}\right| / \log\left|\frac{e_{k+1}}{e_k}\right|$

3.1 Bisection Method

Error: $|e_{k+1}| = \frac{1}{2^{k+1}}(b-a) \;\to\; \frac{|e_{k+1}|}{|e_k|^2} \approx \frac{(b-a)^{1-r}}{2} 2^{k(r-1)}$

Only converges for $r = 1$ to $C = \frac{1}{2}$
Number of Iterations until tol is reached:
$|e_k| = tol \quad \Rightarrow \quad \frac{b-a}{2^k} = tol \quad \Rightarrow \quad k = log_2\left(\frac{b-a}{tol}\right)$

Advantages:	**Disadvantages:**
Certain to converge	Convergence slow
Only needs the sign	Initial interval needs to be known
f(x) doesn't have to be differentiable, but continuous	Can't be generalised for higher dimensions

3.2 Newtons Method (Tangent Method)

Algorithm:	**Algorithm:** roots with $m > 1$
$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$	$x_{k+1} = x_k - m\frac{f(x_k)}{f'(x_k)}$

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

- Exact for Linear Functions → only one iteration
- f needs to be differentiable and continuous at x^*
- Qubic convergence if $f''(x^*) = 0$ but $f'(x^*) \neq 0$, → $m = 1$

Advantages:	**Disadvantages:**
Quadratic convergence	Convergence not guaranteed
Linear convergence for root with multiplicity $m > 1$, → $f'(x^*) = 0$	if $f'(x_k) = 0$ it breaks needs to be differentiable requires $f(x_k)$ and $f'(x_k)$

3.3 Secant Method

Secant through $\{x_k, x_{k-1}\}$

Approximate the Derivative: $f'(x_k) \approx \frac{f(x_k) - f(x_{k-1})}{x_k - x_{k-1}}$

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

Convergence: $r = \varphi = \frac{1+\sqrt{5}}{2} \approx 1.618$

Advantages	**Disadvantages**
only $f(x)$ is needed	not quadratic convergence
only one evaluation per iteration	two first approximations are needed

3.3.1 Newton and Secant Problem

- Both can get stuck in a local minimum, as they follow the slope.
- Therefore they are sensitive to the initial condition.
- Not certain to converge.

3.4 Set of Equations

Find the root of N non linear functions $f_i(\boldsymbol{x})$.
N = Number of Equations (f_i)
M = Number of Variables (x_i)

$$\boxed{F(\boldsymbol{x}^*) = \left[f_1(x^*), f_2(x^*), \dots, f_N(x^*)\right]^T = \mathbf{0}}$$

Taylor Expansion for Matrices and Vectors:

$$F(\boldsymbol{x} + \boldsymbol{y}) = F(\boldsymbol{x}) + J(\boldsymbol{x})\boldsymbol{y} + O\left(\|\boldsymbol{y}\|^2\right)$$

The Jacobian Matrix:

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

Condition Number:

$$\kappa = \left\|J^{-1}(\boldsymbol{x}^*)\right\|$$

General Algorithm:

$$J(\boldsymbol{x}_k)(\boldsymbol{x}_{k+1} - \boldsymbol{x}_k) = -F(\boldsymbol{x}_k), \quad \boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \boldsymbol{z}$$
$$A\boldsymbol{z} = \boldsymbol{b}, \quad A = J(\boldsymbol{x}_k) \quad \boldsymbol{b} = -F(\boldsymbol{x}_k)$$

3.4.1 Newton-Raphson Method: ($N = M$)

J is a square matrix

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

In Practise don't invert J, instead we solve for

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

Facts:

\+ Convergence is quadratic ($r = 2$) if J is not singular

\- cost is substantial. $O(N^2)$ for building J and $O(N^3)$ for solving the linear system.

3.4.2 Pseudo-Newton method: ($M \neq N$)

J has Full Rank

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

Moore Penrose pseudo inverse matrix: J^+

$$J^+ = \begin{cases} (J^T J)^{-1} J^T & \text{for } M > N \\ J^T (JJ^T)^{-1} & \text{for } M < N \end{cases}$$

3.4.3 Modified Newton Method

Instead of computing J every iteration we only use one.

$$J_1\boldsymbol{z} = -F(\boldsymbol{x}_k), \quad J_0 = J(\boldsymbol{x_0})$$

- Gets rid of $O(N^3)$ and leaves us with $O(N^2)$.
- Only good if J doesn't change to rapidly.

3.5 Non Linear Optimization

We want to solve a minimization problem:

$$x^* = \arg\min_{\boldsymbol{x}} E(\boldsymbol{x})$$

$\boldsymbol{x} = (x_1, ..., x_M)^T$ and $E : \mathbb{R}^M \to \mathbb{R}$
Maximisation equal to minimisation of $-E(\boldsymbol{x})$.
Sufficient Condition:

$$F(\boldsymbol{x}) = \nabla E(\boldsymbol{x}) = \left(\frac{\partial E}{\partial x_1}(\boldsymbol{x}^*), ..., \frac{\partial E}{\partial x_M}(\boldsymbol{x}^*)\right)^T = \mathbf{0}$$

Critical Condition:
Hessian matrix positive definite! $\nabla^2 E(\boldsymbol{x}) = H(\boldsymbol{x})$

$$H(\boldsymbol{x}^*) = \begin{pmatrix} \frac{\partial^2 E(\boldsymbol{x}^*)}{\partial x_1^2} & \frac{\partial^2 E(\boldsymbol{x}^*)}{\partial x_1 \partial x_2} & \dots & \frac{\partial^2 E(\boldsymbol{x}^*)}{\partial x_1 \partial x_M} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 E(\boldsymbol{x}^*)}{\partial x_M \partial x_1} & \frac{\partial^2 E(\boldsymbol{x}^*)}{\partial x_M \partial x_2} & \dots & \frac{\partial^2 E(\boldsymbol{x}^*)}{\partial x_M^2} \end{pmatrix}$$

3.5.1 Newtons Method

Algorithm: new Jacobian: → $J(\boldsymbol{x}) = \nabla^2 E(\boldsymbol{x})$

$$\nabla^2 E(\boldsymbol{x}_k)\boldsymbol{z} = -\nabla E(\boldsymbol{x}), \quad \to \quad \boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \boldsymbol{z}$$

A lot of computations and not guaranteed to converge.

4 Interpolation and Splines $M = N$

4.1 Lagrange Interpolation

Fit a function of degree $N - 1$ to N data points.

Lagrange Polynomial:	**Lagrange interpolation function:**
$\ell_k(x) = \prod_{\substack{i=1 \\ i\neq k}}^N \frac{x - x_i}{x_k - x_i}$	$f(x) = \sum_{i=1}^N y_k \ell_k(x)$

$$l_k(x) = \frac{(x-x_1)\dots(x-x_{k-1})(x-x_{k+1})\dots(x-x_N)}{(x_k-x_1)\dots(x_k-x_{k-1})(x_k-x_{k+1})\dots(x_k-x_N)}$$

Facts:

- Polynomials with degree $N - 1$ for N data points
- Interpolate data not extrapolate data
- Analytic expression from data points
- Lowest degree polynomial through data points
- Derivatives $f'(x)$ and $P'(x)$ not guaranteed to match
- (-) Sensitive to noise / Predictability issues
- (-) High degrees give rise to huge oscillations, at the edges
- (-) they are global and can't represent the local behaviour
- (-) small fluctuations in the data, end in remodelling of the whole function
- Error minimal for $x_k = \cos\left(\frac{2k-1}{2n}\pi\right)$: Chebyshev polyn. roots

Approximation Error

$$|y(x) - f(x)| = \left|\frac{y^{(n)}(\xi)}{n!}\prod_{k=1}^{n}(x - x_k)\right|$$

Algorithm Lagrange Interpolation
Input:
$\boldsymbol{x}$, $\boldsymbol{y}$, (arrays)
N, (size)
$\bar{\boldsymbol{x}}$, point
Output:
$\bar{\boldsymbol{y}}$, (interpolation value at $\bar{\boldsymbol{x}}$ ot the data $\boldsymbol{x}, \boldsymbol{y}$)
Steps:
$\bar{\boldsymbol{y}} \leftarrow 0$
for $k = 0, 1, ..., N-1$ **do**
  $l \leftarrow 1$
  for $i = 0, 1, ..., N-1$ **do**
    if $i \neq k$ **then**
      $l \leftarrow l * (\bar{\boldsymbol{x}} - \boldsymbol{x}[i])/(\boldsymbol{x}[k] - \boldsymbol{x}[i])$
    end if
  end for
  $\bar{\boldsymbol{y}} \leftarrow \bar{\boldsymbol{y}} + l * \boldsymbol{y}[k]$
end for
return $\bar{\boldsymbol{y}}$

4.2 Cubic Splines

Locally defined cubic functions to represent data. Given data $\{(x_i, y_i)\}_{i=0,...N}$ with $x_i < x_{i+1}$. In every interval $[x_{i-1}, x_i]$, $i = 1, ..., N$ we define a cubic function:

$$f_i(x) = \alpha_i x^3 + \beta_i x^2 + \gamma_i x + \delta_i, \quad i = 1, ..., N \qquad (1)$$

$4N$ unknowns → **4 Constraints:**

- $f_i(x_{i-1}) = y_{i-1}$, $(i = 1, ..., N)$
- $f_i(x_i) = y_i$, $(i = 1, ..., N)$
- $f_i'(x_i) = f_{i+1}'(x_i)$, $(i = 1, ..., N-1)$
- $f_i''(x_i) = f_{i+1}''(x_i)$, $(i = 1, ..., N-1)$

→ $4N-2$ constraints, we need 2 more.
Possible Conditions:

- Natural spline: Set $f_1''(x_0) = f_N''(x_N) = 0$
- Parabolic runout: Set $f_1''(x_0) = f_1''(x_1)$ and $f''N(x_N) = f_N''(x_{N-1})$
- Clamping: Set $f_1'(x_0) = f_N'(x_N) = 0$
- not-a-knot: $f_1'''(x_1) = f_2'''(x_1)$ and $f_N'''(x_{N-1}) = f_N'''(x_{N-1})$
- Periodic Function: $f''(x_0) = f''(x_N)$, $f'(x_0) = f'(x_N)$ and $f(x_0) = f(x_N)$

We can now solve the problem: $(i = 1, ..., N-1)$
From: $f_i''(x_i) = f_{i+1}''(x_i)$ we get:
$f_i''(x) = \frac{a_{i-1}}{\Delta x_i}(x_i - x) + \frac{a_i}{\Delta x_i}(x - x_{i-1})$
$f_i'(x) = \frac{a_i}{2\Delta x_i}(x - x_{i-1})^2 - \frac{a_{i-1}}{2\Delta x_i}(x_i - x)^2 + b_i$
$f_i(x) = a_{i-1}\frac{(x_i - x)^3}{6\Delta x_i} + a_i\frac{(x - x_{i-1})^3}{6\Delta x_i} + b_i(x - x_{i-1}) + c_i$
$b_i = \frac{\Delta y_i}{\Delta x_i} - \frac{a_i - a_{i-1}}{6}\Delta x_i, \quad c_i = y_{i-1} - \frac{a_{i-1}}{6}\Delta x_i^2$

Equations to be solved: $(i = 1, ..., N-1)$

$$\Delta x_i a_{i-1} + 2(\Delta x_i + \Delta x_{i+1})a_i + \Delta x_{i+1}a_{i+1} = 6\frac{\Delta y_{i+1}}{\Delta x_{i+1}} - 6\frac{\Delta y_i}{\Delta x_i}$$

with: $a_i = f''(x_i)$, $\Delta x_i = x_i - x_{i-1}$ and $\Delta y_= y_i - y_{i-1}$
This ends up as a Matrix Equation; find the vector $\boldsymbol{a}$:

$$\begin{bmatrix} B_0 & C_0 & 0 & 0 & \cdots \\ A_1 & B_1 & C_1 & 0 & \cdots \\ 0 & A_2 & B_2 & C_2 & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots \\ 0 & 0 & A_{N-1} & B_{N-1} & B_{N-1} \\ 0 & 0 & 0 & A_N & B_N \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{N-1} \\ a_N \end{bmatrix} = \begin{bmatrix} D_0 \\ D_1 \\ D_2 \\ \vdots \\ D_{N-1} \\ D_N \end{bmatrix}$$

with (for $i = 1, ..., N-1$)
$A_i = \Delta x_i$ $\quad B_i = \Delta x_i + \Delta x_{i+1}$
$C_i = \Delta x_{i+1}$ $\quad D_i = 6\left(\frac{\Delta y_{i+1}}{\Delta x_{i+1}} - \frac{\Delta y_i}{\Delta x_i}\right)$

For $i = 0$ and $i = N$ we use the special conditions and determine the Coefficients B_0, C_0, A_N, B_N by hand.
Data Points - 1 = Number of Segments

5 Numerical Integration

Main Idea: $I \approx \sum_{i=0}^{N-1} \int_{x_i}^{x_{i+1}} p_i(x)dx$

5.1 Numerical Quadrature, $\Delta_i = x_{i+1} - x_i$

We split Interval $[a, b]$ into N Intervals $[x_i, x_{i+1}]$
Rectangle Rule: $I_{R_i} = f(x_i)\Delta_i$
Midpoint Rule: $I_{M_i} = f\left(\frac{x_i + x_{i+1}}{2}\right)\Delta_i$
Trapezoidal Rule: $I_{T_i} = \frac{f(x_i) + f(x_{i+1})}{2}\Delta_i$
Simpsons Rule:

$$I_{S_i} = \frac{\Delta_i}{6}[f(x_i) + 4f\left(\frac{x_i + x_{i+1}}{2}\right) + f(x_{i+1})]$$

5.2 Total Integrals Δ_i = const.

Rectangle Rule: $I \approx \Delta_i \sum_{i=0}^{N-1} f(x_i)$
Midpoint Rule: $I \approx \Delta_i \sum_{i=0}^{N-1} f\left(\frac{x_i + x_{i+1}}{2}\right)$
Trapezoidal Rule:

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

Simpsons Rule: accurate for 3^{rd} order polynomials
$I \approx \frac{\Delta_i}{3}[f(x_0) + 4\sum_{\substack{i=1 \\ i=\text{odd}}}^{N-1} f(x_i) + 2\sum_{\substack{i=2 \\ i=\text{even}}}^{N-2} f(x_i) + f(x_N)]$

5.3 Error Analysis

Find an upper bound for our Integral. $E_{\text{rule},i} = I_i - I_{\text{rule},i}$
Rectangle Rule: Second Order Accurate
$E_{R_i} = \frac{1}{2}f'(x_i)\Delta_i^2 + \frac{1}{6}f''(x_i)\Delta_i^3 + \frac{1}{24}f'''(x_i)\Delta_i^4 + \mathcal{O}(\Delta_i^5)$
Midpoint Rule: Third Order Accurate
$E_{M_i} = \frac{1}{24}f''(x_{i+1/2})\Delta_i^3 + \mathcal{O}(\Delta_i^5) + \dots$
Trapezoidal Rule: Third Order Accurate
$E_{T_i} = -\frac{1}{12}f''(x_{i+1/2})\Delta_i^3 + \mathcal{O}(\Delta_i^5) + \dots$
Simpsons Rule: Fifth Order Accurate
$I_{S_i} = \frac{2}{3}I_{M_i} + \frac{1}{3}I_{T_i} \rightarrow E_{S_i} = \mathcal{O}(\Delta_i)^5$

For the whole integral the Order is -1

Exact Integration for degree = (Order of Accuracy - 2)
The leading derivative of the error has to be 0. Always check!
Has nothing to do with Δ_i.

5.4 Newton Cotes Formula

We use $M+1$ equidistant points in $[x_i, x_{i+1}]$ $(x_k = x_i + k \cdot h)$, $(k \in [0, M])$ and Lagrange Interpolation.
The Integral: $h = \frac{x_{i+1} - x_i}{M}$, $\Delta_i = x_{i+1} - x_i$

$$I_i \approx \int_{x_i}^{x_{i+1}} p_i(x)\mathrm{d}x = \sum_{k=0}^{M} f(x_k) \int_{x_i}^{x_{i+1}} \ell_k^M(x)\mathrm{d}x$$
$$I \approx \sum_i I_i, \quad I_i \approx \Delta_i \sum_{k=0}^{M} C_k^M f(x_k)$$
$$C_k^M = \frac{1}{\Delta_i}\int_{x_i}^{x_{i+1}} l_k^M(x)dx$$

Properties: $C_k^M = C_{M-k}^M$ and $\sum_{k=0}^{M} C_k^M = 1$
$l_k^M(x_i + x_{i+1} - x_{M-j}) = l_k^M(x_j) = l_{M-k}^M(x_{M-j})$

5.4.1 Error

N Datapoints → $N-1$ intervals. If the error of the integration-rule for a single interval scales with: $\mathcal{O}(h^n)$, than the error for the whole integration is:

$$E \leq (b-a) \cdot \max(c_i) \cdot \mathcal{O}\left(h^{n-1}\right)$$

A higher order rule can perform worse than a lower order rule, if the constant factor in the error of the ho. rule (C1) is greater than the factor of the lo. rule (C2). $C1 \cdot h^n > C2 \cdot h^m m < n$

5.4.2 Error reduction

Error of our intergration rule scales with: $\mathcal{O}(h^n)$. How many more evaluations to decrease error by a factor z?

$$E^* = \frac{1}{z} \Rightarrow \frac{1}{z}C \cdot \mathcal{O}\left(\left(\frac{b-a}{N}\right)^n\right) = C \cdot \mathcal{O}\left(\left(\frac{b-a}{N^*}\right)^n\right)$$
$$\Rightarrow N^* = \sqrt[n]{z} \cdot N$$

Refining the grid locally does not change the order of accuracy of the underlying integration scheme.

6 Richardson and Romberg

6.1 Richardson Extrapolation

A quantity of interest G is discretized by some grid spacing h: $G \approx G(h)$. For $h \rightarrow 0$ we should obtain the exact value G.
Expanding with a Taylor Series we get: (with $G(0) = G$)

$$G(h) = G(0) + c_1 h + c_2 h^2 + \dots$$
$$G(h/2) = G + \frac{1}{2}c_1 h + \frac{1}{4}c_2 h^2 + \dots$$

Subtracting the to equations gives us:

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

Increased exponent of leading error term, by subtraction.
General Case:

$$G_n(h) = \frac{1}{2^n - 1}(2^n G_{n-1}(h/2) - G_{n-1}(h)) = G + O(h^{n+1})$$

Error Estimation: For small h the estimation is good.

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

Relative Tolerance: $\epsilon(h/2) < 3 \cdot \text{ tol } \cdot \frac{h}{h_0}$
Error order: $E_{n-1}(h) > E_{n-1}(h/2) > E_n(h)$

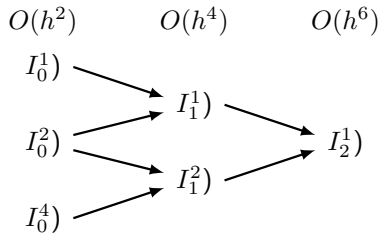
6.2 Romberg Integration

Improve an inaccurate, but "cheap"method, and improve it by using Richardson Extrapolation: $I_0^1, I_0^2, I_0^4, ...$.
We start by using the Trapezoidal Rule and improve it.
Resulting Integral: (Trapezoidal Rule)

$$I_k^n = \frac{4^k I_{k-1}^{2n} - I_{k-1}^n}{4^k - 1}$$

Resulting Integral: (Simpsons Rule)

$$I_k^n = \frac{4^{k+1} I_{k-1}^{2n} - I_{k-1}^n}{4^{k+1} - 1}$$



7 Adaptive Quadrature

- Optimize quadrature by sampling the funtion non-uniformly.
- Evaluate the integral with more precision at points with sudden changes.
- Use Rhomberg integration and error estim. to evaluate locally.

7.1 Gauss-Quadrature

Main Idea: $I = \int_a^b f(x)dx \approx \sum_i c_i \cdot f(x_i)$
Choose c_i and x_i to minimise the error.
Undetermined coefficients: (Trapezoidal Rule)
- Only exact for a straight line

$$I = \int_a^b f(x)dx = \int_a^b (a_0 + a_1 x)dx \approx c_1 f(a) + c_2 f(b)$$

Integration and comparing coefficients: $c_1 = c_2 = \frac{b-a}{2}$
2-point Gauss Quadrature:
Same as above, but variable function evaluation points.

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

with $f(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3$.
Solving it the same way as above we get:

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

Integral is exact for: polynomials of degree N-1, with N = degrees of freedom on the right side.

7.2 Hermite Interpolation

Interpolate the values y_k and derivatives y_k'.

$$f(x) = \sum_{k=1}^{n} U_k(x)y_k + \sum_{k=1}^{n} V_k(x)y_k'$$

U_k and V_k: polynomials of degree $2n-1$, with properties:

$$U_k(x_j) = \delta_{jk}, \quad U_k'(x_j) = 0, \quad V_k(x_j) = 0, \quad V_k'(x_j) = \delta_{jk}$$

$$U_k(x) = \left[1 - 2l_k'(x_k)(x - x_k)\right] l_k^2(x)$$
$$V_k(x) = (x - x_k)l_k^2(x)$$

7.3 n-point Gauss Quadrature

Move Interval $[a, b]$ to $[-1, 1]$. $(x \in [a, b] \rightarrow z \in [-1, 1])$

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

We then get the following integral:

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

Approximating $f(x)$ with Hermite Polynomials:

$$\int_{-1}^{1} f(x)dx = \sum_{k=1}^{n} y_k \int_{-1}^{1} U_k(x)dx + \sum_{k=1}^{n} y_k' \int_{-1}^{1} V_k(x)dx$$
$$\int_{-1}^{1} f(x)dx = \sum_{k=1}^{n} u_k\ y_k + \sum_{k=1}^{n} v_k\ y_k'$$

with $u_k = \int_{-1}^{1} U_k(x)dx$ and $v_k = \int_{-1}^{1} V_k(x)dx = 0 \,\forall\, k$
Resulting Integral: (u_k is tabulated)

$$I = \int_{-1}^{1} f(x)dx = \sum_{k=1}^{n} u_k f(x_k)$$
$$u_k = \frac{2}{(1 - x_k^2)(P_n'(x_k))^2}$$

Error with n abscissas:

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

In Practice we get: w_i is tabulated

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

for $z \in [-1, 1]$:

$$I = \sum_{i=1}^{n} w_i f(z)$$

7.4 Multidimensional Quadrature

Integrate a function in D dimensions:

$$I = \int_{a_1}^{b_1} \cdots \int_{a_D}^{b_D} f(x_1, \dots, x_D)\,\mathrm{d}x_1 \cdots \mathrm{d}x_D$$

Using Quadrature in every dimension with N gridpoints:

$$\int_{a_d}^{b_d} f(x_d)\,\mathrm{d}x_d \approx \sum_{i_d=1}^{N} w_{i_d} f\left(x_{i_d}\right)$$

$$\boxed{I \approx \sum_{i_1=1}^{N} \dots \sum_{i_N=1}^{N} w_{i_1} \dots w_{i_N} f\left(x_{i_1}, ..., x_{i_D}\right)}$$

7.5 Curse of Dimensionality

Quadrature in D dimensions requires $M = N^D$ function evaluations
Additionally, order of accuracy depends on dimension D, one-dimensional order of acc. s and grid spacing $h = \frac{b-a}{N}$:

$$I - I_Q = \mathcal{O}\left(h^s\right) = \mathcal{O}\left(N^{-s}\right) = \mathcal{O}\left(M^{-s/D}\right)$$

With $M = n^d$ and s as the order of the used 1D Method.
d is dimension.

8 Monte Carlo

Monte Carlo only makes sense for more than one Variable.
Sample random points in the domain Ω and count how many are inside the area we want to calculate:

$$I = |\Omega|\langle f \rangle$$

with

$$\langle f \rangle = \frac{1}{|\Omega|}\int_\Omega f(\vec{x})d\vec{x}, \quad |\Omega| = \int_\Omega d\vec{x}$$

Sample the function at M random uniform distributed points $\vec{x}_i$:

$$\langle f \rangle \approx \langle f_M \rangle = \frac{1}{M}\sum_{i=1}^{M} f(\vec{x}_i)$$

$$I \approx |\Omega|\langle f_M \rangle = |\Omega|\frac{1}{M}\sum_{i=1}^{M} f(\vec{x}_i)$$

Assuming the samples $\vec{x}_i$ are independent, the Expectation value of the random function $\langle f_M \rangle$ is equal $\langle f \rangle$, and so is the integral:

$$I = \mathbb{E}[|\Omega|\langle f_M \rangle]$$

Error:

$$\varepsilon_M = \sqrt{\frac{\mathrm{Var}[f]}{M}} \propto \mathcal{O}\left(M^{-1/2}\right)$$

with

$$\mathrm{Var}[f] \approx \frac{M}{M-1}\left(\frac{1}{M}\sum_{i=1}^{M} f(\vec{x}_i)^2 - \langle f \rangle_M^2\right)$$

For large number of samples M, we expect the following errors:

$$|\langle f \rangle - \langle f_M \rangle| < \begin{cases} \varepsilon_M, & \text{with probability of } 68\% \\ 2\varepsilon_M, & \text{with probability of } 95\% \\ 3\varepsilon_M, & \text{with probability of } 99\% \end{cases}$$

8.0.1 Recipe for Monte Carlo Integration

1. Sample points $\vec{x}_i$ from a uniform distribution and evaluate integrand f to get $f(\vec{x}_i)$.
2. Store number of samples, the sum of the values and the sum of squares

$$M, \quad \sum_{i=1}^{M} f(\vec{x}_i), \quad \sum_{i=1}^{M} f(\vec{x}_i)^2$$

3. Compute mean as the estimate of the expectation (**normalized integral**)

$$\frac{I}{|\Omega|} = \langle f \rangle \approx \langle f \rangle_M = \frac{1}{M}\sum_{i=1}^{M} f(\vec{x}_i)$$

4. Estimate the variance using the unbiased sample variance:

$$\mathrm{Var}[f] \approx \frac{M}{M-1}\left(\frac{1}{M}\sum_{i=1}^{M} f(\vec{x}_i)^2 - \langle f \rangle_M^2\right)$$

5. Estimate error

$$\varepsilon_M = \sqrt{\frac{\mathrm{Var}[f]}{M}} \propto \mathcal{O}\left(M^{-1/2}\right)$$

8.0.2 Inverse Sampling

Generate any random with PDF $p_X(x)$ and CDF $F_X(x)$ distribution form an uniform distribution $U \in [0,1]$.

$$F_X(x) = u \;\rightarrow\; x = F_X^{-1}(u), \quad F(x) = \int_0^x p(x)dx$$

$$x^{(i)} = F_X^{-1}(u^{(i)})$$

with $u^{(i)}$ form a uniform distribution interval.

8.0.3 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

8.0.4 Ubiased Estimator

An unbiased estimator of a statistical parameter means that the expected value equals the true value of the parameter, e.g.

9 Neural Networks

9.1 General Structure

$$\mathbf{y} = F(\mathbf{x}, \mathbf{w}), \quad \mathbb{R}^{n_0} \rightarrow \mathbb{R}^{n_L}$$

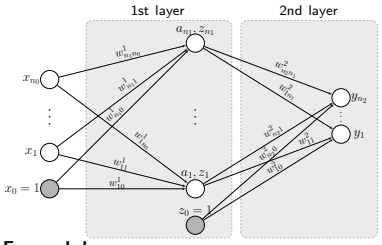
$$\text{output} = F(\text{input}, \text{weight})$$

Different Types of Neural Networks:

- Fully Connected Neural Networks
- Convolutional Neural Networks (CNN)
- Recurrent Neural Networks (RNN)

Function $\boldsymbol{y}(\cdot, \boldsymbol{w}) : \mathbb{R}^{n_0} \rightarrow \mathbb{R}^{n_L}$ parametrized by weights $\boldsymbol{w}$.
w_{ji}: destination node j and source node i

9.2 2-Layer NN



For each layer:

1. Input x_i is weighted by w_i
2. Summed
3. Activation function φ is applied

Map Input to First Layer:

$$a_j^1 = \textstyle\sum_{i=0}^{n_0} w_{ji}^1 x_i \quad \text{and} \quad z_j^1 = \varphi_1\left(a_j^1\right)$$

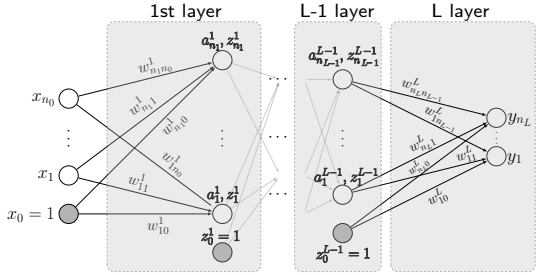
Map First to Second Layer / Output

$$a_j^2 = \textstyle\sum_{i=0}^{n_1} w_{ji}^2 z_i^1 \quad \text{and} \quad y_j = z_j^2 = \varphi_2\left(a_j^2\right)$$

Compact Notation

$$\boxed{\boldsymbol{y}(\boldsymbol{x};\boldsymbol{w}) = \varphi_2\left(W^2\varphi_1\left(W^1\boldsymbol{x}\right)\right)}$$

9.3 L-layer NN



Compact Notation for $\boldsymbol{y}(\boldsymbol{x};\boldsymbol{w}) =$

$$\boxed{\varphi_L\left(W^L\varphi_{L-1}\left(W^{L-1}\varphi_{L-2}\left(\cdots W^2\varphi_1\left(W^1\boldsymbol{x}\right)\right)\right)\right)}$$

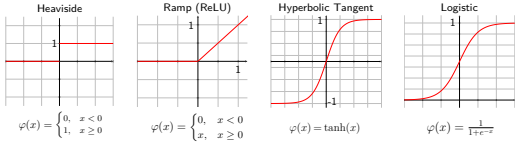
Elements of W^l:

$$W_{pk}^l = w_{pk}^l = w_{ji}^l, \quad \text{with} \quad p = j, k = i$$

d = destination and s = source:

$$\begin{bmatrix} a_{1=d} \\ a_{2=d} \\ a_{3=d} \end{bmatrix} = \begin{bmatrix} w_{1=d,0=s} & w_{1=d,1=s} & w_{1=d,2=s} \\ w_{2=d,0=s} & w_{2=d,1=s} & w_{2=d,2=s} \\ w_{3=d,0=s} & w_{3=d,1=s} & w_{3=d,2=s} \end{bmatrix} \begin{bmatrix} x_{0=s} \\ x_{1=s} \\ x_{2=s} \end{bmatrix}$$

9.4 Activation Function



$$\varphi(x) = \begin{cases} 0, & x < 0 \\ 1, & x \geq 0 \end{cases} \qquad \varphi(x) = \begin{cases} 0, & x < 0 \\ x, & x \geq 0 \end{cases} \qquad \varphi(x) = \tanh(x) \qquad \varphi(x) = \frac{1}{1+e^{-x}}$$

9.5 Training

Goal:
Update the weights w so that the output y_n given an input x_n matches a target $\hat{y}_n$.
Steps:

1. Build a model $\mathbf{y}(\mathbf{x}_n, \mathbf{w})$ with the initial weights $\mathbf{w} = \{W^1, W^2, \ldots, W^L\}$
2. Perform the forward pass, i.e. produce the output $\mathbf{y}_n$ for all $\mathbf{x}_n$ in the dataset
3. Compute the loss with respect to the target:

$$E(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N}\left(\hat{\mathbf{y}}_n - \mathbf{y}\left(\mathbf{x}_n, \mathbf{w}\right)\right)^2 = \sum_{n=1}^{N} E_n$$

4. Perform the backward pass, i.e. update weights (see 6.6)
5. Repeat until you reach a minimum: $\mathbf{w}^\star = \arg\min E(\mathbf{w})$

9.6 Gradient Descent (GD) and Variations

Key Idea:
Use derivatives (gradient) of the cost function E with respect to the weights $\mathbf{w}$ to update the parameters:

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

with *iteration index* k and *learning parameter* η

9.6.1 Stochastic Gradient Descent (SGD):

Alternative to GD with derivative of local error E_n related to the pair $\{x_n, \hat{y}_n\}$:

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

with sequential or random choice of E_n.

9.6.2 Batch Stochastic Gradient Descent (batchSGD):

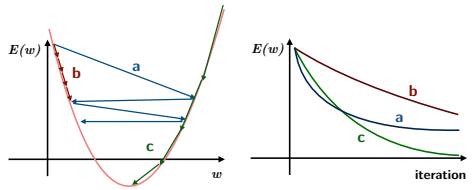
Method between GD and SGD with gradient on subset $\mathcal{I}$, with $\mathcal{I} \subset \{1, 2, \ldots, N\}$, chosen randomly.

$$\boldsymbol{w}^{(k+1)} = \boldsymbol{w}^{(k)} - \eta\nabla_{\boldsymbol{w}}\sum_{n\in\mathcal{I}} E_n\left(\boldsymbol{w}^{(k)}\right)$$

Learning Parameter η

- Crucial hyper-parameter in deep learning
- Not a priori clear how to be chosen

(a) η too high/fast: oscillates between suboptimal values
(b) η too low/slow: takes too many iterations to reach $\mathbf{w}^\star$
(c) desired value of η



9.7 Backpropagation

Update weights using Gradient Descent:

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

and rewrite gradient in terms of $a_j = \sum_k w_{jk}\tilde{z}_k$ (chain rule):

$$\frac{\partial E_n}{\partial w_{ji}} = \frac{\partial E_n}{\partial a_j}\frac{\partial a_j}{\partial w_{ji}} = \frac{\partial E_n}{\partial a_j}\tilde{z}_i = \delta_j\tilde{z}_i$$

Derivative of the Error δ_j

$$\delta_j = \frac{\partial E_n}{\partial a_j} = \sum_k \frac{\partial E_n}{\partial \tilde{a}_k}\frac{\partial \tilde{a}_k}{\partial a_j} = \sum_k \tilde{\delta}_k\frac{\partial \tilde{a}_k}{\partial a_j}$$

with $\tilde{a}_k = \sum_j \tilde{w}_{kj}z_j = \sum_j \tilde{w}_{kj}\varphi(a_j)$:

$$\frac{\partial \tilde{a}_k}{\partial a_j} = \varphi'(a_j)\,\tilde{w}_{kj} \;\Rightarrow\; \delta_j = \varphi'(a_j)\sum_k \tilde{w}_{kj}\tilde{\delta}_k$$

Last Layer of Neural Network, i.e. $a_j = y_j$

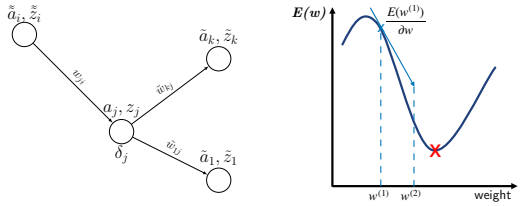
$$\frac{\partial E_n}{\partial w_{ji}} = \frac{\partial E_n}{\partial a_j}\frac{\partial a_j}{\partial w_{ji}} = \delta_j z_i$$

$$\delta_j = \frac{\partial E_n}{\partial a_j} = \frac{\partial}{\partial y_j}\frac{1}{2}\left\|\boldsymbol{y}(\boldsymbol{x}_n;\boldsymbol{w}) - \hat{\boldsymbol{y}}_n\right\|^2$$

$$= \frac{\partial}{\partial y_j}\frac{1}{2}\sum_k\left(y_k(\boldsymbol{x}_n;\boldsymbol{w}) - \hat{y}_{ni}\right)^2 = y_j(\boldsymbol{x}_n;\boldsymbol{w}) - \hat{y}_{nj}$$

9.7.1 Key Idea of Backpropagation

- At last layer, gradients $\frac{\partial E_n}{\partial w_{ji}}$ and $\frac{\partial E_n}{\partial a_j}$ don't depend on Neural Network
- Calculate δ_j at last layer first, then back-propagate to acquire the δ_j's at every previous layer

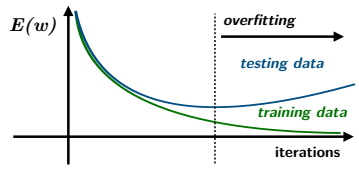


9.8 Overfitting

Bias-Variance-Tradeoff

- Overfitted: Model fits behaviour of noise and does not generalize efficiently (model estimation errors)
- Underfitted: Too few parameters, model ignores meaningful data (model mismatch errors)

Key Idea:
Introduce subset of data (~10%) as a test set and run SGD on both training and test data.
Plot error for both subsets over iterations:



9.9 Facts about Neural Networks

- Increasing hidden hodes improves representation ability for test data. Is more prone to Overfitting
- Increase the hidden Layers increases representation ability for test data. Is more prone to Overfitting
- Two linear (no activation function) NN, can approximate the same class of functions. Regardless of the number of Layers.
- When Batches are used, the gradient is avaraged. Shuffling the data of the batch has no effect.
- For too big learning rates the error can increase

10 Dimensionality Reduction

10.1 Principal Component Analysis

Goal:

- Decrease dimension of the data while either explaining most of the variance or minimizing the reconstruction loss.
- Changes the coordinate system of the data while aligning the axes to the directions with the most variance
- Data must have zero mean, i.e. $\tilde{x}_n = x_n - \bar{x} \in \mathbb{R}^D$
  $\Rightarrow$ centered data matrix $X_C = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N)^T \in \mathbb{R}^{N\times D}$
- Variance of a Matrix X (zero mean):

$$\mathrm{Var}[X] = \sigma^2 = \sigma_1 + \ldots + \sigma_N^2$$

$$\begin{bmatrix} \sigma_1 \\ \vdots \\ \sigma_N \end{bmatrix} = \frac{1}{N-1}\begin{bmatrix} \sum_i x_{1,i} \\ \vdots \\ \sum_i x_{N,i} \end{bmatrix}$$

Maximum Variance Formulation
Find direction $\mathbf{v}^\star = \mathrm{argmax}\,\mathbf{v}^T C\mathbf{v}$ s.t. variance is max.:

$$\boxed{\mathrm{Var}[\{\hat{\boldsymbol{x}}\}_{i=1}^N] = \sigma_1^2 = \tfrac{1}{N-1}\textstyle\sum_{n=1}^N\left(\mathrm{x_n^T v}\right)^2 = \mathrm{v}^T C\mathrm{v}_1}$$

$\boldsymbol{v} \in \mathbb{R}^D$ needs to fullfill $\quad C\mathbf{v}_1^\star = \lambda_1^\star\mathbf{v}_1^\star \quad , \quad \mathbf{v}_1^T\mathbf{v}_1 = 1$

10.1.1 Spectral Value Decomposition

1. Construct centered data matrix $X_C \in \mathbb{R}^{N\times D}$
2. Construct covariance matrix $C = \frac{1}{N-1} X_C^T X_C \in \mathbb{R}^{D\times D}$
3. Preform Eigenvalue Decomposition: $V^{-1} = V^T \in \mathbb{R}^{D\times D}$
   $C = V\Lambda V^{-1}, \quad \Lambda = \text{diag}(\lambda_1, ..., \lambda_d)$
4. Sort Eigenvalues, decreasing order: $(\lambda_1 > \lambda_2 > ... > \lambda_d)$
5. Set $V = (\boldsymbol{v}_1, ..., \boldsymbol{v}_d)$
6. **Now Switch of Data Matrix:** $X = (\boldsymbol{x}_1, ..., \boldsymbol{x}_N) \in \mathbb{R}^{D\times N}$
7. Set $\hat{X} = V^T X$, with $\hat{X}$ as the transformed Data Matrix

10.1.2 Data Compression:

Instead of using the whole matrix V we only use the first R rows:
$$V_r = (\boldsymbol{v}_1, ..., \boldsymbol{v}_r) \in \mathbb{R}^{D\times R}$$
The Reduced Data set is then given by with $X \in \mathbb{R}^{D\times N}$:
$$\hat{X}_r = V_r^T X \in \mathbb{R}^{R\times N}$$
or in Terms of Original Centered Matrix $X_C \in \mathbb{R}^{N\times D}$:
$$X_{r,C} = X_C V_r$$

10.1.3 Reconstruct Data

: The Reconstructed Data $\tilde{X}$ is given by:
$$\tilde{X} = V_r \hat{X}_r = V_r V_r^T X$$
the retained variance (in percent) is computed by
$$\sigma_{\text{retained}} = \frac{\sum_{i=1}^{R} \lambda_i}{\sum_{i=1}^{D} \lambda_i}$$

Kernel PCA:

Nonlinear cluster of data made linearly separable by transforming the data by using some kernel functions ϕ:

$$C \text{ changes to } C = \frac{1}{N}\sum_{i=1}^{N} \phi(x_i)\,\phi(x_i)^T$$

10.2 Autoencoder

Key Idea

- Use Neural Network to learn dimension reduction
- Map input $\mathbf{x}_n \in \mathbb{R}^D$ onto an output $\tilde{\mathbf{x}}^n \in \mathbb{R}^D$ through an intermediate layer $\mathbf{y}_n \in \mathbb{R}^r$ using a matrix $W \in \mathbb{R}^{r\times D}$

$$\boxed{\mathbf{y}_n = W\mathbf{x}_n, \quad \tilde{\mathbf{x}}_n = W^T\mathbf{y}_n}$$

Error Function:

Find optimal weights by minimizing the error,
i.e. the difference between input and output:

$$\mathbf{w}^\star = \underset{\mathbf{w}}{\text{argmin}} \frac{1}{N}\sum_{n=1}^{N} \|\mathbf{x}_n - W^T \underbrace{W\mathbf{x}_n}_{\mathbf{y}_n}\|_2^2$$

Nonlinear Problems
Capture non-linear problems by adding non-linear activation function φ and more intermittent layers:

$$\mathbf{y}_n = \varphi_L\left(W_L\varphi_{L-1}\left(\cdots W_2\varphi_1\left(W_1\mathbf{x}_n\right)\right)\right)$$

$$\tilde{\mathbf{x}}_n = W_1^\top \varphi_2\left(\cdots W_{L-1}^\top \varphi_L\left(W_L^\top \mathbf{y}_n\right)\right)$$

Note: Deeper networks increase expressiveness
but are easier to overfit and memorize the training dataset.

11 Varia

11.1 Modelfitting

Model architecture:
The functional form of $f(x)$. We can choose every function.
It could be a straight line, a polynomial, a exponential etc.
Measure of the "best": (Cost function: $\boldsymbol{e} = \boldsymbol{y} - \boldsymbol{f}(\boldsymbol{x}_i)$)

2-Norm:	**1-Norm:**
$\|\boldsymbol{e}\|_2 = \sqrt{\sum_{i=1}^{N}(y_i - f(x_i))^2}$	$\|\boldsymbol{e}\|_1 = \sum_{i=1}^{N}\|y_i - f(x_i)\|$
∞-Norm:	**General Form:**
$\|\boldsymbol{e}\|_\infty = \max_i \|y_i - f(x_i)\|$	$\|e\|_p = \left(\sum_{i=1}^{N} e_i^p\right)^{\frac{1}{p}}$

Notice: $\|e\|_2^2 = \boldsymbol{e}\cdot\boldsymbol{e}$

11.2 Linear Algebra

Inner Product (Skalarprodukt):	**Image/Range:**
$\boldsymbol{x}\cdot\boldsymbol{y} = \boldsymbol{x}^T\boldsymbol{y} = \sum_{i=1}^{N} x_i y_i$	Space spanned by the rows of A, range(A).
Null Space/Kernel $A \in \mathbb{R}^{N\times M}$:	**Cokernel:**
$A\boldsymbol{x} = 0$, null(A) = all $(\boldsymbol{x}), \boldsymbol{x} \in \mathbb{R}^M$	Cokernel = null(A^T)
Norm:	**Rank:**
$\|\delta\boldsymbol{w}\| = \|A^{-1}\delta\boldsymbol{y}\| \le \|A^{-1}\|\|\delta\boldsymbol{y}\|$	rank A = rank A^T
Norm calculation:	
$\|A\|_2 = \sqrt{\max(\lambda(A^T A))}$	

Inverse Matrix Formulas

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}^{-1} = \frac{1}{ad-bc}\cdot\begin{pmatrix} d & -b \\ -c & a \end{pmatrix}$$

$$\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix}^{-1} = \frac{1}{\det A}\cdot\begin{pmatrix} ei-fh & ch-bi & bf-ce \\ fg-di & ai-cg & cd-af \\ dh-eg & bg-ah & ae-bd \end{pmatrix}$$

11.3 Condition Number

Computers can't calculate exact numbers → rounding error:
$\delta\boldsymbol{w} = \boldsymbol{w} - \tilde{\boldsymbol{w}}$ and $\delta\boldsymbol{y} = \boldsymbol{y} - \tilde{\boldsymbol{y}} = A\boldsymbol{w} - A\tilde{\boldsymbol{w}} = A\delta\boldsymbol{w}$

Condition Number: $\frac{\|\delta\boldsymbol{w}\|}{\|\boldsymbol{w}\|} = \kappa(A)\frac{\|\delta\boldsymbol{y}\|}{\|\boldsymbol{y}\|}$,	well conditioned $= \kappa(A)$ not too large $\kappa(A) = \|A\|\|A^{-1}\|$, $\kappa(A) \in [1,\infty)$
A orthogonal: $\kappa_2(A) = 1$	$\kappa(A) = \|A\|\|A^{-1}\| \overset{(1)}{=} \frac{\sigma_{max}(A)}{\sigma_{min}(A)}$ (1) for 2-norm & positive definite
L_2-norm $\Rightarrow \kappa_2$	Tells us how stable a fit is

11.4 Taylor Expansion

Around a Arbitrary point:
$$f(x) = f(x_0) + \frac{f'(x_0)}{1!}(x-x_0) + \frac{f''(x_0)}{2!}(x-x_0)^2 + ...$$
Around a variable point x: ($x_0 = x$, $x = x + h$)
$$f(x \pm h) = f(x) \pm hf'(x) + \frac{h^2}{2}f''(x) \pm ...$$
Vectors:
$$f_i(\boldsymbol{x}+\boldsymbol{y}) = f_i(\boldsymbol{x}) + \sum_{j=1}^{M}\frac{\partial f_i(\boldsymbol{x})}{\partial x_j}y_j + \mathcal{O}\left(\|y\|^2\right)$$
For the whole System of Equations:
$$F(\boldsymbol{x}+\boldsymbol{y}) = F(\boldsymbol{x}) + J(\boldsymbol{x})\boldsymbol{y} + \mathcal{O}\left(\|y\|^2\right)$$
With $J(x)$ as the Jacobian, see First Page.

11.5 Monte Carlo Integration Scheme

Sample $\vec{x}_i$ from $U(\Omega)$ (D-Dimensions) for $i = 1, ..., M$
Evaluate function $f(\vec{x}_i)$ for $i = 1 \dots M$
Accept sample if $f(\vec{x}_i) \le y_i$, *reject* otherwise:
$$I \approx \frac{\#\text{ accepted samples}}{M}\cdot|\Omega|$$

11.6 Probability Theory

11.6.1 Porbability Distributions:

Binomial Distribution:
$$P(k) = \binom{n}{k}p^k(1-p)^{n-k}$$
Uniform Distribution:
$$p_{\mathcal{U}}(x) = \begin{cases}\frac{1}{b-a} & x \in (a,b)\\ 0 & \text{otherwise}\end{cases}$$
Uniform Distribution for Higher Dimensions:
$$p_{\mathcal{U}}(x) = \begin{cases}\frac{1}{|\Omega|} & x \in \Omega\\ 0 & \text{otherwise}\end{cases}$$
Normal Distribution: μ = mean, σ = standard deviation
$$p_{\mathcal{N}} = \frac{1}{\sqrt{2\pi\sigma^2}}\exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$
Exponential Distribution:
$$p(x) = \lambda e^{-\lambda x}$$

11.6.2 Discrete Porbability:

Discrete Probability Function:
$$P(x) \in [0,1], \quad x \in \Omega \subseteq \mathbb{N}, \quad \sum_i P(x_i) = 1$$
Expected Value: = mean
$$\mathbb{E}[X] = \bar{x} = \sum_i x_i P(x_i) = \underbrace{\frac{1}{M}\sum_i^M p_i}_{p_i \text{ equally likely}} \quad (= \mu \text{ mean})$$

11.6.3 Continuous Probability:

Cumulative Distribution Function: CDF
$$F_X(x) = P(X \le a) = \int_{-\infty}^{a} p(x)dx$$
Probability density function: PDF
$$p(x) = \frac{d}{dx}F_X(x) \ge 0, \quad x \in \Omega \subseteq \mathbb{R}, \quad \int p(x)dx = 1$$
The probability that a value is inside a interval $[a,b]$ **is:**
$$P(a \le X \le b) = \int_a^b p(x)dx$$
Expected Value:
$$\mathbb{E}[X] = \langle X\rangle = \int_\Omega xp(x)dx$$
Expected Value for a Function:
$$\mathbb{E}[h(x)] = \int_\Omega h(x)p(x)dx$$

11.6.4 Identitys for Both

Expectation Value:
$$\mathbb{E}[aX+bY] = a\mathbb{E}[X] + b\mathbb{E}[Y]$$
For two uncorrelated random variables:
$$\mathbb{E}[XY] = \mathbb{E}[X]\mathbb{E}[Y]$$
Variance: σ = **standard deviation**
$$\text{Var}[X] = \sigma^2[X] = \mathbb{E}[X^2] - \mathbb{E}[X]^2 = \frac{1}{n}\sum_{i=1}^{n}\left(x_i - \mu\right)^2$$

11.7 Error of Monte Carlo

$$\varepsilon_M^2 = \text{Var}\left[\langle f\rangle_M\right] = \left\langle\langle f\rangle_M^2\right\rangle - \langle\langle f\rangle_M\rangle^2$$
$$= \frac{1}{M^2}\sum_{i,j=1}^{M}\left(\mathbb{E}\left[f(\boldsymbol{x}_i)f(\boldsymbol{x_j})\right] - \langle f\rangle^2\right)$$
$$= \frac{1}{M^2}\sum_{i=1}^{M}\left(\mathbb{E}\left[f(\boldsymbol{x}_i)^2\right] - \langle f\rangle^2\right)$$
$$+ \frac{1}{M^2}\sum_{i,j=1,i\neq j}^{M}\left(\underbrace{\mathbb{E}\left[f(\boldsymbol{x}_i)f(\boldsymbol{x_j})\right]}_{=\langle f\rangle^2} - \langle f\rangle^2\right)$$
$$= \frac{1}{M^2}\sum_{i=1}^{M}\left(\langle f^2\rangle - \langle f\rangle^2\right) = \frac{\text{Var}[f]}{M}$$

12 Algorithms

Algorithm Romberg Integration (Trapezoidal Quadrature)
Input:
function $f(x)$ / interval a, b/ numer of iterations K
Output:
I_K^1 = integral$[K, 0]$ approximation to the integral $\int_a^b f(x)dx$
Steps:
Precompute and store function evaluations
$maxNumIntervals \leftarrow 2^K$
$hmin \leftarrow (b-a)/maxNumIntervals$
for $i \leftarrow 0, ..., maxNumIntervals$ **do**
$fvalues[i] \leftarrow f(a + i * hmin)$ **end for**
for $r \leftarrow 0, ..., K$ **do**
$numIntervals \leftarrow 2^r$ / $step \leftarrow 2^{K-r}$ / $result \leftarrow 0$
for $i \leftarrow 1, ..., numIntervals - 1$ **do**
$result \leftarrow result + fvalues[i * step]$ **end for**
$f_0 = f[0]$ / $f_N = f[maxNumIntervals]$
$integral[0, r] \leftarrow 0.5\frac{b-a}{numIntervals}(f_0 + 2 * result + f_N)$
end for
for $l \leftarrow 1, ..., K$ **do**
for $r \leftarrow 0, .., K - l$ **do**
$integral[l, r] \leftarrow \frac{4^l * integral[l-1, r+1] - integral[l-1, r]}{4^l - 1}$
end for
end for

Algorithm Adaptive integration
Steps:
Subdivide the interval of the integration into sub-intervals
for all *sub-intervals* **do**:
Compute sub-integral, estimate the error with Richardson
if *accuracy is worse than desired* **then:**
Subdivide the interval
else
Leave the interval untouched **end if** **end for**

Algorithm Bisect Method
Input:
a,b, (initial interval, $a < b$)
tol, (tolerance, minimum lenght of interval, tol > 0)
k_{max}, (maximum number of iterations, $k_{max} > 1$)
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 sign$(f(a))$ = sign$(f(x_k))$ **then**
$a \leftarrow x_k$
else
$b \leftarrow x_k$ **end if**
$k \leftarrow k + 1$
end while

Algorithm Newton Method
Input:
x_0, (initial condition)
tol, (tolerance, stop if $\|x_k - x_{k-1}\| < tol$)
k_{max}, (maximum number of iterations, $k_{max} > 1$)
Output:
x_k, (approximate solution of $f(x_k) = 0$ after k iterations)
Steps:
$k \leftarrow 1$
while $k \le k_{max}$ **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

Algorithm Newton Method
Input:
$\boldsymbol{x}_0$, (initial condition)
tol, (tolerance, stop if $\|x_k - x_{k-1}\| < tol$)
k_{max}, (maximum number of iterations, $k_{max} > 1$)
Output:
$\boldsymbol{x}_k$, (approximate solution of $f(x_k) = 0$ after k iterations)
Steps:
$k \leftarrow 1$
while $k \le k_{max}$ **do**
Calculate $F(\boldsymbol{x}_k)$ and $N \times N$ matrix $J(\boldsymbol{x}_k)$
Solve the $N \times N$ linear system: $J(\boldsymbol{x}_k)\boldsymbol{z} = -F(\boldsymbol{x}_k)$
$\boldsymbol{x_{k+1}} \leftarrow \boldsymbol{x}_k + \boldsymbol{z}$
if $\|z\| < tol$ **then**
break **end if**
$k \leftarrow k + 1$
end while

Algorithm ANN Training Loop (SGD)
Input:
X, {Input dataset} / Y, {Target dataset } / η, {learning rate}
n_{batch}, {batch size} / n_{epochs}, {number of training epochs}
Output:
W, {weight} or $y = f_{ANN}(x)$, {the mapping}
Steps:
Split data into Testing and Training / Create Loss Vector
Iterate over all Epochs:
Schuffle Data
Iterate over Batch:
Get Batch data
Forward Pass, Loss Computation, Gradient Computation
Update Wheights
End Batch Iteration
Test Data, Store Loss in Loss Vector
if testing_loss[i] > testing_loss[i-1] **then**
Stop training
end if
Epochs Iteration