Computational Intelligence: set of nature-inspired methodologies to address complex real-world problems to which traditional methodologies are useless

- combines 3 frameworks:
  1) ANN
  2) EA
  3) Fuzzy Logic → understand natural language → suspended

Deep Learning: part of ML methods, based on ANN with representation learning
- both supervised or non-supervised

Differences with brain: 1) Dynamic - Plastic
2) Analog → Not just different # of neurons (not so important)

Turing Test → used to understand if a machine shows human-level intelligence
- Judge interacts with two boxes and needs to decide which one is a machine
- Doesn't mean that it has consciousness but that it can have a human conversation by mimicking
- ChatGPT-4o has this abilities

ACI (Artificial Capable Intelligence) → whether the AI is able to accomplish something concrete (like doubling an initial sum of money)
AGI is the next step but it doesn't have an associated step

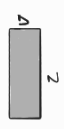
LEAST SQUARES

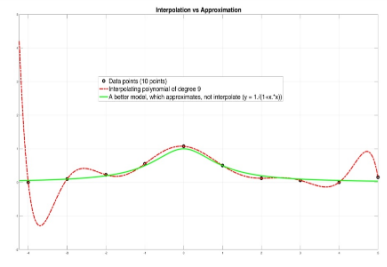
Why not using 1-NN (with a look-up table) instead of a ANN?
- It cannot interpolate and generalize } K-NN would mitigate the problem interpolating
- It cannot perform denoising

But 1) suffers a lot complexity at inference phase
2) since it is model free we cannot compress or exploit regularities in D

Training set $\{P_i, y_i\}_{i=1}^{N_{tr}}$ where $P_i \in \mathbb{R}^d$ → y_i: label in classification or output in regression

TRS ∪ TSS = learning set while TRS ∩ TSS = $\{\emptyset\}$

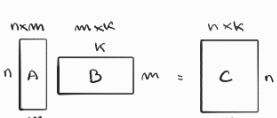
$P = \begin{bmatrix} P_1^1 & \cdots & P_1^D \\ \vdots & & \\ P_N^1 & \cdots & P_N^D \end{bmatrix} \in \mathbb{R}^{N \times D}$ $\quad \vec{o} = \begin{bmatrix} o_1 \\ \vdots \\ o_N \end{bmatrix} \in \mathbb{R}^N$ $\quad$ typically N >> D $\quad P =$ 



Interpolation: estimate f via $D: \langle P, \vec{o} \rangle$ exactly
Approximation: estimate f via $\langle P, \vec{o} \rangle$ approximately, while sacrificing some level of accuracy

LLS (Linear Least Squares)

$f(P) = b + vP$ $\quad b \in \mathbb{R}$ $\quad v \in \mathbb{R}^{1 \times D}$ → row vector



We want to minimize $E_i = \frac{1}{2}[f(P) - o_i]^2$ → minimizing this over the dataset is called LLS

Define $X = \begin{bmatrix} \bar{1} \\ P^T \end{bmatrix} = \begin{bmatrix} 1 & \cdots & 1 \\ P_1^1 & \cdots & P_1^N \\ \vdots & & \\ P_0^1 & \cdots & P_0^N \end{bmatrix}$ (N × D+1) $\quad y = O^T \in \mathbb{R}^{1 \times N}$

$\Rightarrow w = [b, v] \in \mathbb{R}^{1 \times (D+1)}$ can be obtained by $\vec{w} = \underbrace{y}_{\mathbb{R}^{1 \times (D+1)}} \underbrace{X^T}_{N \times (D+1)} \underbrace{(XX^T)^{-1}}_{(D+1) \times (D+1)}$ → Penrose Pseudo-Inverse

- time complexity $O(N(D+1)^2)$
- XX^T: $[(D+1) \times N][N \times (D+1)] \rightarrow \Theta(N(D+1)^2)$
- inverting → $\Theta((D+1)^3)$
- $X^T A$: $[N \times (D+1)] \cdot [(D+1) \times (D+1)] \rightarrow \Theta(N(D+1)^2)$

$$L_w = \frac{1}{n} \sum (\underbrace{x_i}_{\text{i-th column}} w - y_i)^2 = \underbrace{\|wX - y\|_2^2}_{\text{I could call it } v_2} = (wX - y)(wX - y)^T = (wX - y)(\underbrace{X^T w^T}_{N \times 1} - y^T) = \underbrace{wXX^T w^T}_{\text{SCALARS}} - \underbrace{wXy^T}_{\text{SCALARS}} - \underbrace{yX^T w^T}_{\text{SCALARS}} + yy^T = wXX^T w^T - 2wXy^T + yy^T$$

$$\nabla_w L_w = \frac{\partial}{\partial w}[wXX^T w^T] - \frac{\partial}{\partial w}[2wXy^T] = \underbrace{2wXX^T}_{1 \times (D+1)} - \underbrace{2(Xy^T)^T}_{1 \times (D+1)} = 0 \quad \text{(continuous + unconstrained)} \Rightarrow \quad w = yX^T(XX^T)^{-1}$$

$$\frac{\partial}{\partial x_k} x^T A x = \frac{\partial}{\partial x_k} \sum_i \sum_j m_{ij} x_i x_j = \sum_j m_{kj} x_j + \sum_i m_{ik} x_i = \sum_i (m_{ki} + m_{ik}) x_i$$ ↓ k-th component of $(A + A^T)x$

$$\rightarrow \frac{\partial}{\partial x} x^T A x = (A + A^T)x \overset{\text{symm}}{=} 2Ax$$

WLLS If we want to weight each (x_i, y_i) pair using S_i: $w = y\Delta X^T (X \Delta X^T)^{-1}$ $\quad \Delta = \text{diag}(\vec{s})$
- weight less outliers

GRADIENT DESCENT LLS

→ The formula must use the whole dataset at once, making it inefficient to compute → OFFLINE COMPUTATION

$E_n = \frac{1}{2}(\hat{y}_n - y_n)^2 = \frac{1}{2}[\underbrace{b + vP_n}_{(1 \times D)(D \times 1)} - y_n]^2$ (instantaneous error)

$\frac{\partial E_n}{\partial b} = b + vP_n - y_n \in \mathbb{R}$

$\frac{\partial E_n}{\partial v^d} = [b + vP_n - y_n] P_n^d$ $\quad \nabla_v E_n = \underbrace{\frac{\partial E_n}{\partial v}}_{1 \times D} = [b + vP_n - y_n] P_n^T = (\hat{y}_n - y) P_n^T$ → complexity of one iteration of one sample: $O(D+1)$

- Jacobian used when loss function is a vector

Step 1) Initialize randomly b and $\vec{v}$
Step 2) Set a learning rate η → slow convergence or overshooting

for 1 to #iter:
  for 1 to N:
    Compute $\hat{y}_n = b + \vec{v}\vec{p}_n$
    $b' \leftarrow b - \eta \frac{\partial E_n}{\partial b}$
    $v' \leftarrow v - \eta \frac{\partial E_n}{\partial v}$

- Single batch (whole X may not fit in GPU (offline)) or more mini-batches $\vec{g}_v = \frac{1}{|B_i|} \sum_{n=1}^{B} \frac{\partial E_n}{\partial v_d}$
  ↳ peak performance when the batch size saturate GPU memory

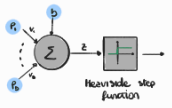
For K iterations and whole dataset $O(KN(D+1))$
- typically smaller than closed formula

BENEFITS:
- efficient for large systems of equations → no pseudo inverse computation
- lower memory requirements → not whole X in memory
- parallelizable

CLASSIFICATION
- 2 class classifier (2C-CP) (class. prob.) is a function that maps $x \in \mathbb{R}^D$ to $y \in \{w_T, w_O\}$ (target, outlier)
- In mutually exclusive multi class (MEMC-CP) $y \in \{w_1, \dots, w_K\} = \Omega$
- In multilabel classification problem (ML-CP) → used in computer vision, like in YOLO $y \in \mathcal{P}^{\Omega}$ (any subset, even empty)

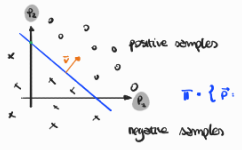
MLP → should the artificial neuron be a perfect replica of the biological neuron?
- No, since we are not using it to understand the brain but to learn new solutions to independent problems → airplanes do not flap their wings

The perceptron is the first ANN ever discovered (1958)



$\hat{y} = \begin{cases} 1 & \text{if } b + \sum v_i p_i > 0 \\ 0 & \text{otherwise} \end{cases}$

- $|v_i|$ signifies the importance of feature p_i in the decision
- $\vec{v}$ is $\perp$ to the decision boundary, that is shifted by b



$\overline{B} = \{\vec{p} : \vec{p}^T \vec{v} + b = 0\}$

The rule for learning b and $\vec{v}$ is called **Delta Rule**

1) Initialize v_i's randomly
for 1 to #iter for 1 to N
2) Compute $\hat{y}_n = \text{Heaviside}(b + \sum v_i p_i^n)$
3) $b \leftarrow b - \eta(\hat{y}_n - y_n)$
   $\vec{v} \leftarrow \vec{v} - \eta(\hat{y}_n - y_n)\vec{p}_n^T$ (row-vector)

$0 < \eta < 1$

$\hat{y}_n - y_n = \begin{cases} 1 & \text{wrong} \\ 0 & \text{correct} \rightarrow \text{no changes} \\ -1 & \text{wrong} \end{cases}$

- problem $\frac{d}{dx}\text{Heaviside}(x) = \delta(x)$ → cannot use backpropagation (it has been found by trial and error) ↳ we need a C^1 counterpoint

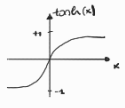
- Problem: it cannot shatter a two classes problem in 2D (when the points are not linearly seperable) → eg XOR problem

↳ first AI winter

ACTIVATION FUNCTION → their role is to introduce non-linearities
- parameter-less
- parameterized → learnable (slows down the convergence) a priori

Sigmoid (logistic) $\sigma(x) = \frac{1}{1+e^{-x}}$

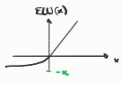
Hyperbolic tangent (tanh) $\tanh(x) = \frac{\sinh(x)}{\cosh(x)} = \frac{e^x - e^{-x}}{e^x + e^{-x}}$



rectified linear unit (ReLU) $\text{ReLU}(x) = \max\{0, x\}$

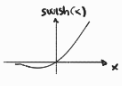
Leaky ReLU $\text{LReLU}_{\alpha}(x) = \begin{cases} x & \text{if } x \geq 0 \\ \alpha x & \text{otherwise} \end{cases}$ α = small constant or learnable param (Parametric ReLU)

Exponential Linear Unit (ELU) $\text{ELU}(x) = \begin{cases} x & \text{if } x \geq 0 \\ \alpha(e^x - 1) & \text{otherwise} \end{cases}$



Linear Unit $\text{LU}(x) = x$

Swish $\text{swish}(x) = x \cdot \sigma(x) = \frac{x}{1+e^{-x}}$



SIGN $\text{sign}(x) = \begin{cases} +1 & \text{if } x \geq 0 \\ -1 & \text{otherwise} \end{cases}$
- used in spiking NN → almost everywhere differentiable

MAX OUT $\text{maxout}(x_1, x_2) = \max\{x_1, x_2\}$
- multi-input $\mathbb{R}^2 \rightarrow \mathbb{R}$

Softmax $\text{softmax}(x_c) = \frac{e^{x_c}}{\sum_{n=1}^{C} e^{x_n}}$
↳ per-layer activation instead of per-node activation → can node a single output

Sigmoid Derivative $\sigma(z) = \frac{1}{1+e^{-z}}$

$\frac{d\sigma(z)}{dz} = -\frac{-e^{-z} \pm 1}{(1+e^{-z})^2} = \frac{1}{1+e^{-z}} - \frac{1}{(1+e^{-z})^2} = \frac{1}{1+e^{-z}}\left[1 - \frac{1}{1+e^{-z}}\right] = \sigma(z)[1-\sigma(z)]$

TanH Derivative $\tau(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} = \frac{\sinh(z)}{\cosh(z)} \Rightarrow \frac{d\tau(z)}{dz} = \frac{\cosh^2(z) - \sinh^2(z)}{\cosh^2(z)} = 1 - \tau(z)^2$ (no minus for cosh')

Logistic Regression → 2 mutually exclusives class · same as perceptron but with a sigmoid · Accept as class 1 if $\sigma(z) \geq \frac{1}{2}$

Define a loss called **BINARY CROSS ENTROPY (BCE)** $\text{BCE} = \underbrace{-y \log(\hat{y})}_{y=1} \underbrace{- (1-y)\log(1-\hat{y})}_{y=0}$ ($\hat{y} \in (0,1)$) → when the two classes are mutually exclusive

$\hat{y}(p) = \sigma(p) = \frac{1}{e^{-(b+\vec{v}\vec{p})}}$

Define $\vec{w} = [b, \vec{v}] \in \mathbb{R}^{1\times(D+1)}$, $\vec{x} = [1, p_1, \dots, p_D]^T$ $\Rightarrow \hat{y}(x) = \sigma(\overbrace{wx}^{z}) = \frac{1}{1+e^{-wx}}$

$\mathcal{L} = -y \log \hat{y} - (1-y)\log(1-\hat{y})$

$\frac{\partial \mathcal{L}(w)}{\partial w_j} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z} \frac{\partial z}{\partial w_j}$

$\frac{\partial \mathcal{L}}{\partial \hat{y}} = -\frac{y}{\hat{y}} + \frac{1-y}{1-\hat{y}}$ $\quad \frac{\partial \hat{y}}{\partial z} = \sigma(z)[1-\sigma(z)] = \hat{y}(1-\hat{y})$ $\quad \frac{\partial z}{\partial w_j} = \frac{\partial wx}{\partial w_j} = x_j$

$\rightarrow \frac{\partial L}{\partial w_j} = \left[-\frac{y}{\hat{y}} + \frac{1-y}{1-\hat{y}} \right] \hat{y}(1-\hat{y}) x_j = -(1-\hat{y}) y x_j + \hat{y}(1-y) x_j = -y x_j + \hat{y} x_j = (\hat{y}-y) x_j$

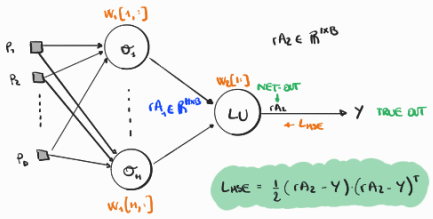
$\rightarrow \boxed{\frac{\partial L}{\partial w} = (\hat{y}-y)\vec{x}}$

In case of a minibatch $X \in \mathbb{R}^{B \times (D+1)}$, $X = [\vec{1}_B, P^T]$ (first column of ones; P pattern matrix $P \in \mathbb{R}^{B \times D}$) $\rightarrow \frac{\partial L(w)}{\partial w} = (\hat{y}-y) X$ (row vectors)

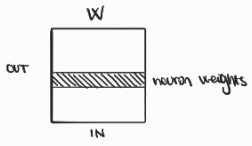
- The problem is convex in $\vec{w}$, since BCE is convex in z: $-y \log \sigma(z) - (1-y) \log(1-\sigma(z))$ and z is linear in $\vec{w}$
  $\Rightarrow$ converges always to the global solution independently from initialization of b and w

- No closed solutions exists $\nabla_w L = (\hat{y}-y)\vec{x} = [\sigma(Wx) - y]\vec{x}$ $\rightsquigarrow$ this can be solved, but not in the case of mini-batch

MLP - MSE LIN $\rightarrow$ use a layer of neurons with non-linear activations and a final neuron with LU. Implement the loss as MSE
- if I need multi-output regression I could add more final neurons



$L_{MSE} = \frac{1}{2}(rA_2 - Y)(rA_2 - Y)^T$



FWD

$rA_0 = P^T \in \mathbb{R}^{D \times B}$

$A_0 = [\vec{1}_B, rA_0] \in \mathbb{R}^{(D+1) \times B}$

$rz_1 = W_1 \cdot A_0 \in \mathbb{R}^{H \times B}$ ($W_1 \in \mathbb{R}^{H \times (D+1)}$)

$rA_1 = \sigma(rz_1)$

$A_1 = [1_B, rA_1] \in \mathbb{R}^{(H+1) \times B}$

$rz_2 = W_2 A_1 \in \mathbb{R}^{1 \times B}$ ($W_2 \in \mathbb{R}^{1 \times (H+1)}$)

$rA_2 = Irz_2 = rz_2 \in \mathbb{R}^{1 \times B}$

BWD

$\frac{\partial L}{\partial W_1} = \frac{\partial L}{\partial rz_1}\frac{\partial rz_1}{\partial W_1} = \frac{\partial L}{\partial rz_1} A_0^T \in \mathbb{R}^{H \times (D+1)}$

$\frac{\partial L}{\partial rz_1} = \frac{\partial L}{\partial rA_1}\frac{\partial rA_1}{\partial rz_1} = \frac{\partial L}{\partial rA_1} \odot \sigma'(rz_1) \in \mathbb{R}^{H \times B}$

$\frac{\partial L}{\partial rA_1} = \frac{\partial L}{\partial A_1}\frac{\partial A_1}{\partial rA_1} = \underbrace{(0, I_H)}_{H \times (H+1)} \frac{\partial L}{\partial A_1} \in \mathbb{R}^{H \times B}$ (remove the first row) $= \text{red}\left(\frac{\partial L}{\partial A_1}\right)$

$\frac{\partial L}{\partial W_2} = \frac{\partial L}{\partial rz_2}\frac{\partial rz_2}{\partial W_2} = (rA_2 - Y) A_1^T \in \mathbb{R}^{1 \times (H+1)}$ $\quad \frac{\partial L}{\partial A_1} = \frac{\partial L}{\partial rz_2}\frac{\partial rz_2}{\partial A_1} = W_2^T (rA_2 - y) \in \mathbb{R}^{(H+1) \times B}$

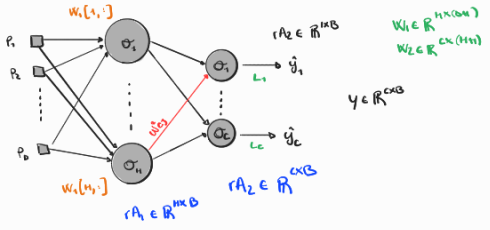
$\frac{\partial L}{\partial rz_2} = \frac{\partial L}{\partial rA_2}\frac{\partial rA_2}{\partial rz_2} = rA_2 - Y \in \mathbb{R}^{1 \times B}$

- considering multiple layers without non-linearities $rA_0 = P^T$ $rz_1 = W_1 A_0$, $rA_1 = rz_1$, $rz_2 = W_2 A_1$, $rA_2 = rz_2$ $\Leftrightarrow$ $rA_2 = W_2 \cdot W_1 \cdot P^T$ $\Leftrightarrow$ $rA_2 = W' \cdot P^T$
  $\rightarrow$ not completely true when using SGD and Dropout — generalizable with induction

- shallow NN $\rightarrow$ one hidden or no one (+ input + final)

- deeper networks $\rightarrow$ hierarchical representations of the input

MLP - BCE SIG With C classes (not exclusive) we need C output neurons with a sigmoid activation



$L = \sum_{c=1}^{C} L_c$ with $L_c = -\sum_{b=1}^{B} \left[y_c^b \cdot \log(rA_{2c}^b) + (1-y_c^b) \cdot \log(1 - rA_{2c}^b) \right]$ ($y \in \mathbb{R}^{C \times B}$, $rA_2 \in \mathbb{R}^{C \times B}$)

$\rightarrow$ BCE for neuron c

b-th pattern: $\frac{\partial L}{\partial rA_{2c}^b} = \frac{\partial L_c}{\partial rA_{2c}^b} = -\frac{y_c^b}{rA_{2c}^b} + \frac{1-y_c^b}{1-rA_{2c}^b}$

$rA_2 = \sigma(rz_2) \rightarrow \frac{\partial rA_{2c}^b}{\partial rz_{2k}^b} = \begin{cases} rA_{2c}^b (1 - rA_{2c}^b) & \text{if } k=c \\ 0 & \text{otherwise} \end{cases}$ $\Rightarrow$ $\boxed{\frac{\partial L}{\partial rz_{2c}^b}} = \sum_{k=1}^{C} \frac{\partial L}{\partial rA_{2k}^b}\frac{\partial rA_{2k}^b}{\partial rz_{2c}^b} = \frac{\partial L}{\partial rA_{2c}^b}\frac{\partial rA_{2c}^b}{\partial rz_{2c}^b} = -\frac{y_c^b}{rA_{2c}^b} \cdot rA_{2c}^b (1 - rA_{2c}^b) + \frac{1-y_c^b}{1-rA_{2c}^b} rA_{2c}^b (1 - rA_{2c}^b)$

$= rA_{2c}^b - y_c^b$ — resemble the MSE-LIN

$rz_2^b = W_2 A_1^b$

$\frac{\partial L}{\partial W_{2cj}} = \sum_{k=1}^{C} \frac{\partial L}{\partial rz_{2k}^b}\frac{\partial rz_{2k}^b}{\partial W_{2cj}} = \sum_{k=1}^{C} (rA_{2k}^b - y_k^b)\frac{\partial rz_{2k}^b}{\partial W_{2cj}^2} = (rA_{2c}^b - y_c^b) rA_{1j}^b$ ($W_2 \in \mathbb{R}^{C \times (1+H)}$)

$\frac{\partial L}{\partial rA_{1j}^b} = \sum_{k=1}^{C} \frac{\partial L}{\partial rz_{2k}^b}\frac{\partial rz_{2k}^b}{\partial A_{1j}^b} = \sum_{k=1}^{C} \frac{\partial L}{\partial rz_{2k}^b} W_{kj}^2 \rightarrow \frac{\partial L}{\partial rA_1} = W_2^T \frac{\partial L}{\partial rz_2}$

$\frac{\partial L}{\partial rz_{1j}^b} = \sum_{k=1}^{H} \frac{\partial L}{\partial rA_{1k}^b}\frac{\partial rA_{1k}^b}{\partial rz_{1j}^b} = \frac{\partial L}{\partial rA_{1j}^b} \sigma'(rz_{1j}^b)$ $\quad \frac{\partial L}{\partial rA_1} = \text{red}\left(\frac{\partial L}{\partial A_1}\right)$

$\frac{\partial L}{\partial W_1} = \frac{\partial L}{\partial rz_1^b}\frac{\partial rz_1^b}{\partial W_1} = \frac{\partial L}{\partial rz_1^b} A_{0b}^T$ $\quad \mathbb{R}^{H \times B} \cdot \mathbb{R}^{B \times (D+1)} = \mathbb{R}^{H \times (D+1)}$

$rA_0 = P^T \in \mathbb{R}^{D \times B}$

$A_0 = \text{ext}(rA_0) \in \mathbb{R}^{(1+D) \times B}$

$rz_1 = W_1 A_0 \in \mathbb{R}^{H \times B}$

$rA_1 = \sigma(rz_1) \in \mathbb{R}^{H \times B}$

$A_1 = \text{ext}(rA_1) \in \mathbb{R}^{(1+H) \times B}$

$rz_2 = W_2 \cdot A_1 \in \mathbb{R}^{C \times B}$

$rA_2 = \sigma(rz_2) \in \mathbb{R}^{C \times B}$

$\frac{\partial L}{\partial W_1} = \frac{\partial L}{\partial rz_1}\frac{\partial rz_1}{\partial W_1} = \frac{\partial L}{\partial rz_1} A_0^T \in \mathbb{R}^{H \times (1+D)}$

$\frac{\partial L}{\partial rz_1} = \frac{\partial L}{\partial A_1}\frac{\partial A_1}{\partial rA_1}\frac{\partial rA_1}{\partial rz_1} = \text{red}\left(\frac{\partial L}{\partial A_1}\right) \odot \sigma'(rz_1) \in \mathbb{R}^{H \times B}$

$\frac{\partial L}{\partial A_1} = \frac{\partial L}{\partial rz_2}\frac{\partial rz_2}{\partial A_1} = W_2^T \frac{\partial L}{\partial rz_2} \in \mathbb{R}^{(1+H) \times B}$

$\frac{\partial L}{\partial W_2} = \frac{\partial L}{\partial rz_2} A_1^T \in \mathbb{R}^{C \times (1+H)}$

$\frac{\partial L}{\partial rz_2} = rA_2 - Y \in \mathbb{R}^{C \times B}$

MLP_CCESM → classes assumed to be **mutually exclusive** ⇒ use categorical CE and softmax activation

softmax is a vectorial function $\mathbb{R}^C \to \mathbb{R}^C$ with $a_k = \frac{e^{z_k}}{\sum_c e^{z_c}}$

$rA_2 \in \mathbb{R}^{C\times B} = \text{softmax}(rz_2) \quad \to \quad \frac{\partial rA_{2c}}{\partial rz_{2k}} = \begin{cases} rA_{2c}\,[1-rA_{2c}] & \text{if } c=k \\ -rA_{2c}\cdot rA_{2k} & \text{if } c\neq k \end{cases}$

if $c=k \Rightarrow$ $\underbrace{e^{rz_{2c}} \cdot \frac{1}{\sum_j e^{rz_{2j}}}}_{rA_{2c}} + e^{rz_{2c}} \cdot \frac{-1}{(\sum_j e^{rz_{2j}})^2} e^{rz_{2c}} = rA_{2c}\,[1-rA_{2c}]$ $\quad \left[\frac{\partial}{\partial z}\left(\frac{f}{g}\right) = \frac{\partial f}{\partial z}\frac{1}{g} + f\cdot\frac{-1}{g^2}\frac{\partial g}{\partial z}\right]$

if $c\neq k \Rightarrow$ $-\frac{e^{rz_{2c}}}{(\sum_j e^{rz_{2j}})^2} \cdot e^{rz_{2k}} = -rA_{2c}\cdot rA_{2k}$

$L = \sum_c \sum_b -y_c^b \log(rA_{2c}^b)$ (one hot encoding)

$\frac{\partial L}{\partial rA_{2c}^b} = \frac{-y_c^b}{rA_{2c}^b}$

$\frac{\partial rA_{2c}^b}{\partial rz_{2k}^b} = \begin{cases} rA_{2c}\,[1-rA_{2c}] & \text{if } c=k \\ -rA_{2c}\cdot rA_{2k} & \text{if } c\neq k \end{cases} \Rightarrow \frac{\partial L}{\partial rz_{2c}^b} = \sum_{k=1}^{C} \frac{\partial L}{\partial rA_{2k}^b}\frac{\partial rA_{2k}^b}{\partial rz_{2c}^b} = \frac{\partial L}{\partial rA_{2c}^b}\frac{\partial rA_{2c}^b}{\partial rz_{2c}^b} + \sum_{k\neq c}\frac{\partial L}{\partial rA_{2k}^b}\frac{\partial rA_{2k}^b}{\partial rz_{2c}^b}$

$= -\frac{y_c^b}{rA_{2c}^b} rA_{2c}^b\,(1-rA_{2c}^b) + \sum_{k\neq c} \frac{-y_k^b}{rA_{2k}^b} \cdot -rA_{2k}^b\, rA_{2c}^b$

$= -y_c^b + rA_{2c}^b \underbrace{\sum_{k=1}^{C} y_k^b}_{=1} = rA_{2c}^b - y_c^b$

$\frac{\partial L}{\partial W_{2ij}} = \sum_k^C \frac{\partial L}{\partial rz_{2k}^b}\frac{\partial rz_{2k}^b}{\partial W_{2ij}} = \sum_k (rA_{2k}^b - y_k^b)\frac{\partial rz_{2k}^b}{\partial W_{2ij}} = (rA_{2c}^b - y_c^b)\, rA_{1j}^b$

$\frac{\partial L}{\partial rA_{1j}^b} = \sum_{k=1}^{C} \frac{\partial L}{\partial rz_{2k}^b}\frac{\partial rz_{2k}^b}{\partial rA_{1j}^b} = \sum_{k=1}^{C} \frac{\partial L}{\partial rz_{2k}^b} W_{kj}^2$

$\frac{\partial L}{\partial rz_{1j}^b} = \sum_{k=1}^{H} \frac{\partial L}{\partial rA_{1k}^b}\frac{\partial rA_{1k}^b}{\partial rz_{1j}^b} = \frac{\partial L}{\partial rA_{1j}^b}\sigma'(rz_{1j}^b)$

$\frac{\partial L}{\partial W_1} = \frac{\partial L}{\partial rz_1^b}\frac{\partial rz_1^b}{\partial W_1} = \frac{\partial L}{\partial rz_1^b} A_0^T$

$rA_0 = P^T \in \mathbb{R}^{D\times B}$

$A_0 = \text{ext}(rA_0) \in \mathbb{R}^{(1+D)\times B}$

$rz_1 = W_1 A_0 \in \mathbb{R}^{H\times B}$

$rA_1 = \sigma(rz_1) \in \mathbb{R}^{H\times B}$

$A_1 = \text{ext}(rA_1) \in \mathbb{R}^{(1+H)\times B}$

$rz_2 = W_2 \cdot A_1 \in \mathbb{R}^{C\times B}$

$rA_2 = \text{softmax}(rz_2) \in \mathbb{R}^{C\times B}$

$\frac{\partial L}{\partial W_1} = \frac{\partial L}{\partial rz_1}\frac{\partial rz_1}{\partial W_1} = \frac{\partial L}{\partial rz_1} A_0^T \in \mathbb{R}^{H\times(1+D)}$

$\frac{\partial L}{\partial rz_1} = \frac{\partial L}{\partial A_1}\frac{\partial A_1}{\partial rA_1}\frac{\partial rA_1}{\partial rz_1} = \text{red}\left(\frac{\partial L}{\partial A_1}\right) \odot \sigma'(rz_1) \in \mathbb{R}^{H\times B}$

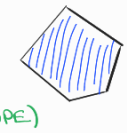
$\frac{\partial L}{\partial A_1} = \frac{\partial L}{\partial rz_2}\frac{\partial rz_2}{\partial A_1} = W_2^T \frac{\partial L}{\partial rz_2} \in \mathbb{R}^{(1+H)\times B}$

$\frac{\partial L}{\partial W_2} = \frac{\partial L}{\partial rz_2} A_1^T \in \mathbb{R}^{C\times(1+H)}$

$\frac{\partial L}{\partial rz_2} = rA_2 - Y \in \mathbb{R}^{C\times B}$

CONSIDERATION WITH C=2

1 hidden layer: n hyperplanes defined by $w_i x + b_i = 0$ (since ReLU activation)



the output is $o = \sum_{i=1}^{n} w_i \text{ReLU}(z_i)$ ($z_i = w_i x + b_i$) → intersection of active hyperplanes that is a **convex region** (POLYTOPE)

2 hidden layers: I can represent union of convex polytopes also disconnected → no more convex
- each of them can be mapped differentely

3 hidden layers: can represent structure with holes

Insufficient training data can lead to underfitting, while excessive increase in training time

1) Hold out partition (Train-Val-Test) → performances depends only on the randomness of a single partition
   - only a fraction of data is used for training
2) K-Fold Cross Validation → Divide Train+Val in K partition, then K-1 for training and the last for validation
   - reduce significantly the variance of Val performance estimate
3) Stratified K-Fold → important for imbalanced dataset
4) Leave one out cross validation → K = # of samples
   - too expensive and not used in DL
5) Time series splitting → here random sampling is invalid
   - cannot use future data to train and then test on the past since it violates causality
   - non-random split (sequential) - (t_0, t_i) training
     - $[t_i, t_j)$ val - $[t_j, t_k]$ test

- rolling cross validation where val/test window moves forward → preserves causality

Weight Initialization → improper init will lead to vanishing or exploding gradients

- **XAVIER initialization**: given a layer (n_{in} = # of inputs, n_{out} = # of outputs) ⇒ initialize W such that it has a constant variance across layers
  - keeps the variance of the input and of backprop constant

FWD: If $\mu_x = 0$ and $\sigma_x = Var[x]$ ⇒ $Var[y] = n_{in} Var[W] Var[x]$ (output / input) since $y_j = W^T x + b_j$ ⇒ $Var(y_j) \overset{\mu=0}{=} \sum_i^{n_{in}} Var[w_{ij} x_i] \overset{indep}{=} \sum_i^{n_{in}} Var[w_{ij}] Var(x_i)$

- to maintain $Var[y] = Var(x)$ we need $n_{in} Var[W] = 1$

BWD: $n_{out} Var[W] = 1$

- To satisfy both conditions (approx) we set $Var[W] = \frac{2}{n_{in} + n_{out}}$ → Applicable if the function is symmetric around zero (tanh or linear)

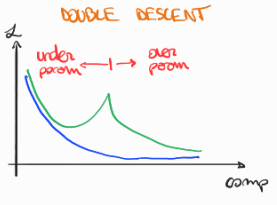
Batch Norm (for a single neuron) $\hat{x} = \gamma \frac{x - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}} + \beta$ (μ_B in the mini-batch) → mitigates covariance shifts during training → stabilize training + speed up
- γ and β allows the network to learn an optimal range for each feature
- during inference use μ_B and σ_B learned during training

DropOut → regularization technique by randomly dropping out a subset of neurons during training
- avoids being over-reliant on a neuron output
- in inference scale the output by $\frac{1}{1-p}$ to maintain consistency

Bias-Variance Trade-Off
- overfitting when test error >> train error
- underfitting when training error stops decreasing

until 2012 { → rule of thumb: keep the # of params ≤ $\frac{|D|}{5}$
- 5 observation per free-parameter }

Why large models don't overfit (hypothesis):
- Lottery Ticket: because they contains subnetworks that are selected during training and can generalize well
- Implicit regularization of SGD that promotes simpler models (min $\sum w_i^2$ eg)
- The effective model capacity is limited by SGD + regularization + augmentation + early stopping
- Batch Norm have some regularization effects by preventing to relie too heavely on a single feature



Sample from $p(x,y)$ K datasets, then $A(D_i) \rightarrow \hat{f}_i(x_0)$ (learning function; evaluation point) $[y = f(x) + \varepsilon]$

⇒ $EPE_\lambda(x_0) = \mathbb{E}_{D_K, p(y|x_0)}\left[(y - \hat{f}_K(x_0))^2\right] \approx \frac{1}{K}\sum_{k=1}^{K}(y_k - \hat{f}_k(x_0))^2$ ($y_k \sim p(y|x_0)$; expected prediction error)

$= \sigma_\varepsilon^2 + \left(\mathbb{E}_{D_K}[\hat{f}_K(x_0)] - f(x_0)\right)^2 + \mathbb{E}_{D_K}\left[(\hat{f}_K(x_0) - \mathbb{E}_{D_K}[\hat{f}_K(x_0)])^2\right]$

irreducible error = randomness on y that doesn't depends on x

→ typically decreasing one will increase the other (like regularization will decrease var, but increase bias)
- more params allows to approximate $f(x_0)$ correctly, but increases variance since overfit to noise (ε)

Pearson statistics measures the linear (only linear relations) dependency of two vectors $R \in [-1, +1]$

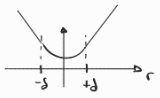
$R = \frac{cov(y, \hat{y})}{s_y \cdot s_{\hat{y}}}$ → $\sqrt{Var[\hat{y}]}$

Error function: is a more general term that can be used to describe different discrepancy metrics (MAE, MSE, ---)
- used as a measure to asses the model performance

Loss function: what it is optimized

- MAE = $\frac{1}{N}\sum |y_i - \hat{y}_i|$ Differentiable almost everywhere ($\hat{y}_i \neq y_i$)
  → used in regression since it is more robust to outliers

- $L_\delta(r) = \begin{cases} \frac{1}{2}r^2 & |r| \le \delta \\ \delta(|r| - \frac{1}{2}\delta) & |r| > \delta \end{cases}$ ($r = y - \hat{y}$) — HUBER LOSS

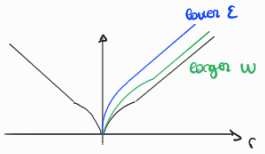


$L_\delta(r) \in C^1$ → robust to outliers
- larger δ more L_2, smaller more L_1

- $HL = \frac{1}{N}\sum_{i=1}^{N} \max(0, 1 - y_i \hat{y}_i)$ ($\hat{y}_i \in [-1,+1]$, $y_i \in \{-1,+1\}$) — HINGE LOSS → used in SVM for binary classification → penalize $\hat{y}_i$ closer to the decision boundary

- $KLD = \frac{1}{N}\sum y_i \log \frac{y_i}{\hat{y}_i}$ $\quad D_{KL}(P\|Q) = \sum_{x \in X} P(x) \log \frac{P(x)}{Q(x)} = \mathbb{E}_{P(x)}\left[\log \frac{P(x)}{Q(x)}\right]$

- $L_{wing}(r) = \begin{cases} w \ln(1 + \frac{|r|}{\varepsilon}) & |r| < w \\ |r| - C & |r| \ge w \end{cases}$ with $C = w - w\ln(1 + \frac{w}{\varepsilon})$
  - params $w, \varepsilon > 0$
  - continous in $|r| = w$



- reduce outliers since grows linearly
- steeper gradient around zero

REGULARIZATION → parameter surface will have fewer local minima and a shifted global minima

L1 (lasso) $+\lambda \sum_i |w_i|$ → encourages sparsity some $w_i = 0$ (eg. when some input features are redundant) → feature selection ∧ interpretable model

$+\log P(w|X,y) = \log P(y|X,w) + \log P(w) = -\frac{1}{2\sigma^2}\|y - Xw\|^2 - \sum_i \frac{|w_i|}{b}$
(regression, prior; MAP)

$\Leftrightarrow P(y|X,w) = \mathcal{N}(Xw, \sigma) \wedge P(w_i) = \frac{1}{2b} e^{-\frac{|w_i|}{b}}$

L2 (ridge) $+\lambda \sum_i w_i^2$ → encourages smaller weights
- penalization on all terms rather than L_1 ⇒ smoother model
- better generalization

In regression if define $E = \sum_{i=1}^{m} (y_i - p(x_i))^2 + \lambda w^T w$ ⇒ $p = (A^T A + \lambda I_m)^{-1} A^T y$

Offline Learning
- entire training processed at once (batches)
- repeat multiple epochs — weights update when all points are seen
- requires going through all dates for each epoch

Online Learning
- individual points or small batches are processed sequentially
- weights are updated after each point
- learns continuously when new data is available
- handles streaming of data, adapt to changing environments
- efficient for large dataset, but less stable than batch learning

GRADIENT DESCENT

- learning rate can be manually tuned (starting small and in case increase until divergence) or by a grid search or we could use a scheduler in order to not have it fixed during training:
  1) step decay: reduce by a factor x every y epoch
  2) exponential decay over epochs
  3) Adaptive methods (like Adam, RMSprop, ...) → automatically adjust lr based on gradient history
     - validation set performance decreasing lr if doesn't improve
- If the training loss oscillates too much ⇒ decreases lr, while if too slow to decrease increase it

- SGD $\theta_{t+1} = \theta_t - \eta \nabla_\theta L(\theta_t, \underbrace{x_i, y_i}_{\text{mini-batch}})$

- **AdaGrad** $\theta_0^i = 0$, $G_0^i = 0$ $\Rightarrow$ $g_t = \nabla_\theta \mathcal{L}(\theta_{t-1})$ (typically in batches)

$G_t^i = G_{t-1}^i + g_{t,i}^2$ (never decreases)

$\Rightarrow \quad \theta_t^i = \theta_{t-1}^i - \frac{\eta}{\sqrt{G_t^i + \varepsilon}} g_{t,i}$ (each parameter has it's own learning rate based on it's past updates; ε avoid zero division)

- doesn't work for long trainings

- **RMSProp** root mean square propagation

$v_{t-1}^i = \beta v_t^i + (1-\beta)\, \nabla_\theta \mathcal{L}(\theta_t)_i^2$ → gradient moving average, solves AdaGrad problem; β → decay rate of exponential decay

$\theta_{t+1}^i = \theta_t^i - \eta \frac{\nabla_\theta \mathcal{L}(\theta_t)_i}{\sqrt{v_{t+1}^i} + \varepsilon}$ (v_{t+1}^i: variance of the gradients)

- **Adam** $m_{t+1}^i = \beta_1 m_t^i + (1-\beta_1)\, \nabla_\theta \mathcal{L}(\theta_t)_i$ (moving avg of squared grad (second moment))

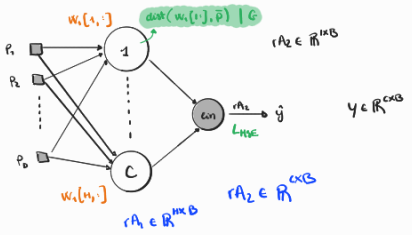
$v_{t+1}^i = \beta_2 v_t^i + (1-\beta_2)\, \nabla_\theta \mathcal{L}(\theta_t)_i^2$

$\hat{m}_{t+1}^i = \frac{m_{t+1}^i}{1-\beta_1^{t+1}}$, $\hat{v}_{t+1}^i = \frac{v_{t+1}^i}{1-\beta_2^{t+1}}$ } bias correction otherwise toward zero for initialization

$\rightarrow \quad \theta_{t+1}^i = \theta_t^i - \eta \frac{\hat{m}_{t+1}^i}{\sqrt{\hat{v}_{t+1}^i} + \varepsilon}$

RADIAL BASIS FUNCTION NETWORKS → 3 layers network, used for regression problems

$\varphi(\vec{x}) = \varphi'(\|x - c\|)$ (depends only on the distance to a center); $\varphi: [0, +\infty) \to \mathbb{R}$



- Typically the activation function of the hidden layer is a gaussian with higher activation if the feature is near the center

$G(x \mid \mu, \sigma) = e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$ (ε) → the normalization is not interesting

$G_\varepsilon(x \mid \mu, \sigma) = e^{-\varepsilon^2(\mu - x)^2}$ (z) $\Rightarrow \quad z(w, x) = (w-x)^2$ and $G_\varepsilon(z \mid \varepsilon) = e^{-\varepsilon^2 z^2}$

(z: preactivation function; G_ε: given, not optimized)

For multivariate case: $z(\vec{w}, \vec{x}) = \|\vec{w} - x\|_2 = \sqrt{\sum_{i=1}^{n} (x_i - w_i)^2}$ ($\vec{w}$: centroid)

I could use mini-batches $W_1 \in \mathbb{R}^{C \times D}$, $rA_0 \in \mathbb{R}^{D \times B} \Rightarrow rZ_1 = dist(W_1, rA_0) \in \mathbb{R}^{C \times B}$ with $rZ_1(i,j) = dist(W_1[i,:], rA_0[:,j])$ (i-th row; j-th sample in mini-batch)

→ Weights of the hidden layer lives in the same dimension of our input

→ We know how to initialize the centroids based on a clustering algorithm

$p(x) = \frac{1}{(2\pi)^{D/2} |\Sigma|^{1/2}} \exp\left(-\frac{1}{2}(x-\mu)^T \Sigma^{-1} (x-\mu)\right)$ (omitted; $\mathbb{R}^{D \times D}$)

we constrain the matrix to be diagonal and with constant variance $G(\vec{x} \mid \vec{\mu}, \sigma) = \exp\left(-\frac{1}{2\sigma^2}(x-\mu)^T(x-\mu)\right)$

$G_\varepsilon(z) = \exp(-\varepsilon^2 z^T z)$, $\varepsilon = \frac{1}{\sqrt{2}\sigma}$

- Typically trained offline and with a single batch → If we fixed W_1 using clustering algorithm (suboptimal) the MSE w.r.t W_2 is linear and can be computed with the pseudo-inverse or I could use online learning with backprop on W_1 and W_2

$rZ_1 = dist(W_1, P) = \begin{bmatrix} \|w_1^1 - p^1\| & \cdots & \|w_1^1 - p^B\| \\ \vdots & & \vdots \\ \|w_1^H - p^1\| & \cdots & \|w_1^H - p^B\| \end{bmatrix}$ (first row of W_1)

$rA_1 = G_\varepsilon(dist(W_1, P)) \quad \Rightarrow \quad \hat{y}(\vec{p}) = w_2^0 + \sum_{h=1}^{H} w_2^h\, \varphi_\varepsilon(\|w_1^h - p\|)$

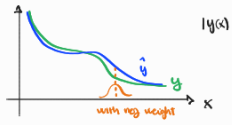
(equivalent of the linear combination of the kernels (typically denoted with λ_h))

$\hat{y}(p) = \lambda_0 + \sum_{h=1}^{H} \lambda_h e^{-\varepsilon^2 \|\vec{y}_h - p\|_2^2}$ (h-th centroid)

→ Write the derivative for back prop

→ can be demonstrated that this is a universe approximator if the radial kernel has infinite support

- Similar to kernel density estimation (KDE) that is used to model pdf
  - gaussian kernel will leads to vanishing gradients (when we are far from the center)
  - I could embed RBF within an AE



$|y(x) - \hat{y}(x)| < \varepsilon \quad \forall x \in \text{domain}$

- Occam's Razor: when two hypotheses have the same explanatory power we should select the one with fewest assumptions
- Minimum Description Length (MDL): select the model with shortest encoding (model + data given) if same metric → more interpretable and generalizable
- Graceful degradation principle: when something goes wrong fall back to minimal functionality instead of crashing
- The progressive enhancements principle: bottom-up improvement of a system → use models that can be expanded

EVOLUTIONARY ALGORITHMS

→ finding the global optimum of non-convex (thus not-linear) and non-differentiable is NP-hard

- If domain is discrete => exhaustive search (not feasible)
- If continous I could discretize between min/max => grid search (curse of dimensionality)

Random Search is highly inefficient since it doesn't exploit informations. Stochastic Search is better

ANNEALING
- In physics heats a metal and cool it slowly
  - soften the material, restore original properties ...
- Tempering heats it at a lower temp to make it stronger

Simulated Annealing → probabilistic optimization algorithm, used to find near optimum solutions
- It moves thrug neighbour solutions, accepting worse solutions proportional to a temperature parameter
  - as temp decreases I will accept worse solutions more difficulty

- I could have linear, log or exp cooling
  → EXPLORATION vs EXPLOITATION trade-off
- could be used for HT in ML
- proven that sim. annealing asymptotically find the global optimum

1) Initialize temp, solution, schedule
2) generate a neighbor solution
3) if $f(x_{t-1}) < f(x_t)$ → x_t new solution (rand() < temp())
   else accept x_t according to a criterion as Metropolis
4) adjust temp and if not finished go to 2)
5) return the best solution found

1) Evolution Strategies → simplest EA that uses mutation only
2) Genetic Algorithms → solutions as strings (typically binary), uses mutation / cross-over
3) Genetic Programming → solutions are in the form of computer programs (typically trees) and are evaluated to solve a target problem
4) NeuroEvolutions → similar to above but solutions represents NN (structure or weights)

} numerical optimization

(1+1) ES
- one offspring generated per epoch
- maintains a single individual

1) Initialization: parent solution x_0 (using some heuristic), $t \leftarrow 0$
2) Mutation → random/gaussian
3) Evaluation
4) Selection (if it is better)
5) Termination or goto 2)

- Typically used as baseline for new models
- memory less strategy

$(\mu + \lambda)$ ES
- maintains a population
- we have a pool of best solutions → memory
- still memoryless since it depends only on P_{t-1}

- allows to use cross-over
- can be //

GENETIC ALGORITHMS → are meta-heuristic algorithms (higher level frameworks to solve difficult optimization problems)
↳ designed to navigate large search space efficiently

- used for problems that are difficult, large or lacks concise math formulation → no assumptions
- mimics Darwin Evolution
- a Population individual is encoded via a chromosome (string)
- offspring selection is normally carried out via fitness
- Offspring population substitutes partially or completely

- Fitness function can be equivalent to the objective functor or be a modification (e.g. to handle constraints)

```
function [best_x, best_y] = ES_MU_plus_LAMBDA(mu, lambda, ngenes)
    % Initialize population P with mu random individuals
    P = initializePopulation(mu,ngenes);

    % Evaluate fitness of each individual in P
    fit_P = evaluateFitness(P);

    % Set the current generation
    g = 0;

    % Repeat until termination condition is met
    while terminationConditionNotMet()
        % Create an empty offspring population Q
        Q = [];

        % Repeat until Q is of size lambda
        while size(Q, 1) < lambda
            % Select a parent from P using any selection method (random, binary tou
            parent = selectionMethod(P);

            % Perform mutation on the parent to create an offspring
            offspring = mutation(parent);

            % Add the offspring to Q
            Q = [Q; offspring];
        end

        % Evaluate fitness of each individual in Q
        fit_Q = evaluateFitness(Q);

        % Combine P and Q to obtain a combined population R
        R = [P; Q];  fit_R = [fit_P; fit_Q];

        % Sort population R in descending order based on fitness
        [-, indices] = sort(fit_R, 'descend');

        % Select the top mu individuals from R to form the next generation P
        P = R(indices(1:mu), :); fit_P = fit_R(indices(1:mu));

        % Increment generation counter g by 1
        g = g + 1;
    end

    % Return the best individual in the final population P as the result
    [-, best_index ] = min(fit_P);
    best_x = P(best_index,:);   best_y = fit_P(best_index);
end
```

```
function [x_opt, f_opt] = GA_BE_1D(f,a,b,generations,chrsize,popsize,pmutallgenes,displayAndPlot)

% Generate the initial population
P = BE_initpop(chrsize,popsize);

% Computing the fitness of each individual in the initial population
fit_P = evaluatefitness(f,P,a,b);

for t = 2:generations
    % 1. Create the Mating Pool (MP) using binary tournament selection
    MP = binarytournement(P,fit_P,popsize);

    % 2. Reshuffle the Mating Pool
    MP = MP(randperm(popsize),:);

    % 3. Apply the crossover operator to each pair in the MP, to obtain offsprings Q
    for i=1:2:popsize
       Q([i,i+1],:) = BE_XOVER_singlepoint(MP(i,:),MP(i+1,:));
    end

    % 4. Apply mutation operator to obtain the mutated version of Q
    for i=1:popsize
       BE_MUT_unif_all(Q(i,:),pmutallgenes);
    end

    % 5. Evaluate the fitness of Q
    fit_Q = evaluatefitness(f,Q,a,b);

    % 6. Compute R as the union of P and Q
    R = [P; Q];     fit_R = [fit_P; fit_Q];

    % 7. Compute the new population P, by taking the best individuals from R
    [-,ind] = sort(fit_R,"ascend");
    new_P = R(ind(1:popsize),:);     fit_new_P = fit_R(ind(1:popsize));

    % 8. Let the new population new_P become the current population and do the same for the fitness
    P = new_P;    fit_P = fit_new_P;
end

[f_opt, ind] = min(fit_P);
x_opt = logical2real(P(ind,:),a,b);
```

if constrained priority to the feasible solutions

- if both violates → random / CV(x)
- if both feasible the one with better f

BINARY ENCODING: each gene is a bit

MUTATION

1) single point mutation: randomly select a gene and negate it
2) uniform: define a probability to mutate a gene and do it on all

if too high → destroy genetic information of the offspring
if too low → premature convergence

CROSS-OVER

1) one point CO: select a point and swaps the genes that comes after

$$O_{1i} = \begin{cases} P_{1,i} & \text{if } i \leq \text{crossover\_point} \\ P_{2,i} & \text{otherwise} \end{cases} \qquad O_{2i} = \begin{cases} P_{2i} & \text{if } i \leq \text{crossover\_point} \\ P_{1i} & \text{otherwise} \end{cases}$$

2) two point cross-over $\quad O_{1i} = \begin{cases} P_{1i} & \text{if } i \leq \text{co\_point1} \vee i > \text{co\_point2} \\ P_{2i} & \text{otherwise} \end{cases}$

REAL ENCODING → each gene is a real number

MUTATION

1) Gaussian Mutation: mutated gene $M_i = G_i + \sigma N(0,1)$
2) Uniform Mutation: $M_i = G_i + \Delta_i \, U(-1,+1)$ — Δ_i: typically $\frac{1}{10}$ or $\frac{1}{20}$ the len of the domain of i variable

CROSSOVER

1) discrete crossover $\quad O_{1i} = \begin{cases} P_{1i} & \text{if } rand() \leq P_r \\ P_{2i} & \text{otherwise} \end{cases}$ (real number)
2) Arithmetic cross over $\quad O_{1i} = \alpha_i P_{1i} + (1-\alpha_i) P_{2i}$ $\quad \alpha_i \in$ random in $[0,1]$
3) convex → Arithmetic CO with $\alpha_i \in [0,1]$ fixed, typically 0.5
4) simulated Binary CO (SBX): inspired by single point CO

$\beta_q = 1 \Rightarrow$ parents; $|\beta_q| < 1 \Rightarrow$ between parents

$$O_{1i} = \frac{1}{2}\left[(1+\beta_q) P_{1i} + (1-\beta_q) P_{2i}\right]$$
$$O_{2i} = \frac{1}{2}\left[(1-\beta_q) P_{1i} + (1+\beta_q) P_{2i}\right]$$

(β_q: controls distribution spread)

$$\beta_q = \begin{cases} (2u)^{\frac{1}{\eta+1}} & u \leq 0.5 \\ \left[\frac{1}{2(1-u)}\right]^{\frac{1}{\eta+1}} & \text{otherwise} \end{cases}$$

with $u \sim U(0,1)$

η set to values small $(2,\dots,5)$ or large $(20,\dots,100)$ — small: further from parents (exploration); large: exploitation

- optional check x_{lb}, x_{ub}

A more advanced version:

1) $x_{min} = \min(P_{1i}, P_{2i})$, $x_{max} = \max(P_{1i}, P_{2i})$
2) $u_i \sim U(0,1)$ (open interval)
3) $\beta_i = 1 + 2\frac{x_{min} - e_{bi}}{x_{max} - x_{min}}$
4) $\alpha_i = 2 - \beta^{-(\eta+1)}$
5) compute $\beta_{q_i} = \begin{cases} (u_i \alpha_i)^{\frac{1}{\eta+1}} & \text{if } u_i \leq \frac{1}{\alpha_i} \\ \left(\frac{1}{2 - u_i \alpha_i}\right)^{\frac{1}{\eta+1}} & \text{if } u_i > \frac{1}{\alpha_i} \end{cases}$
6) compute c_1, c_2
7) from c_1 and c_2 get o_1, o_2 by selecting some genes from c_1 and the other from c_2 and viceversa

Number of Generation typically in $\frac{10 \cdot 50 \cdot K}{N_{pop}}$ (# of fitness eval)

- sometimes computing so many times f could be prohibit → surrogate model (e.g. for CFD or ML hyperturning)

GAs don't guarantees convergence to global optimum in all cases but:

1) **Schema Theorem**: $\mathbb{E}[\#\text{ of individuals in a schema}] \propto e^n$ if the schema has above avg fitness and C.O. and mutation are not disruptive
   ↳ set of individuals with similar characteristics
2) **Building Block hypothesis**: small low order schema with high fitness are called building blocks
   • GA are good at preserving and recombining building blocks
3) **Markov Chain Analysis**: GA can be modelled as MC where $x_t = P_t$
   • under certain conditions GA are stationary → converge to global optimum
   and if it is concentrated around the optimum

MEMETIC ALGORITHM (global search (GA)) → combine metaheuristic with local search strategy
• GA (crossover and mutation) + gradient descent to optimize the offspring in a deterministic way

Can be used in Travelling Salesman Problem (TSP)
- over all pairs of edges, check if swapping them the result improves → until no improvements || too many iterations } 2-OPT ALGORITHM (local search operator)
  $(i,j),(k,e) \rightarrow (i,k),(j,e)$
→ must start from a feasible solution

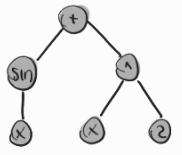
SURROGATES (reduces complexity)
1) DNN → complex non-linearities
2) Gaussian Process Regression → provides uncertainty estimates
3) Radial Basis Function → simple
4) Polynomial Regression → fast but may lack accuracy

Procedure:
1) Generate P_0 and evaluate with f
2) Train surrogate model g on the evaluation
while gen < T:
3) select, C.O., mutate
4) evaluate using g
5) Update g with new evals periodically

NO FREE LUNCH THEOREM (NFL) No optimization algorithm outperforms the others when averaged over all possible problems
- universal optimality is impossible
- algorithm is problem dependent → important to have domain knowledge
- started choice analysis among opt. algorithms to test in relevant domains

Limitations:
1) the theorem assumes f to optimize to be uniformally probable → we do not encounter random arbitrary problems
2) NFL considers all opt. problems as black boxes → in reality we have some exploitable patterns as continuity, symmetry, ...
3) Typically we are not interests of performances among all functions

GENETIC PROGRAMMING evolves computer programs to solve a problem
- problems as tree structures (syntax trees)
  1) functions in internal nodes
  2) terminal in leaves (var/const)



$f(x) = \sin x + x^2$

Crossover will swap subtrees, while mutation randomly altering a node/subtree

Can be used for **Symbolic Regression** (find a function that approximates a Dataset)
- fitness typically MSE
- doesn't assume any model form

Problem: tree grows exponentially without improvement → because genetic operators introduces redundant code

Solution: use HOGP to balance fitness and complexity

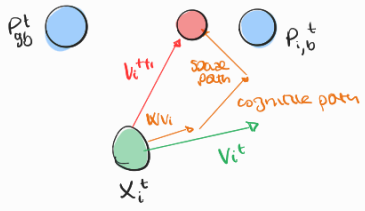
PSO (Particle Swarm Optimization)

- population based stochastic optimization technique (fish or birds)

- **particle** is a possible solution in the search space
- **swarm** is a collection of particles
- **velocity** is the rate at which a particle moves in the search space

pBestPos$_i$ → best position of i-th particle so far
gBestPos → best pos found by a particle in the swarm

1) init the swarm with random pos - vel
2) evaluate and update pBestPos$_i$ and gBestPos
3) update velocities and positions



$$\begin{cases} V_i(t+1) = w \cdot V_i(t) + c_1 r_1 (pBestPos_i - x_i(t)) + c_2 r_2 (gBestPos - x_i(t)) \\ x_i(t+1) = x_i(t) + V_i(t+1) \end{cases}$$

$r_1 \sim U(0,1)$, $r_2 \sim U(0,1)$

- w: inertia weight
- c_1: cognitive acceleration
- c_2: social acceleration

- PSO doesn't have genetic operators, particles share info directly with gBestPos

- fewer params to adjust, simpler implementation wrt GA and good at handling non-linear, non-diff function

Limitations → converge prematurely in a local optima
- 3 params to set (w, c_1, c_2)
- less effective for discrete optimization

- also used for NN training

ACO (Ant Colony Optimization) → discrete opt. in graphs MULTIAGENT

- we use **pheromone trails** left in the environment to stimulate an action in same or different agent → positive fitness benefit

1) Initialization $\tau_{ij} = 0$ (pheromone matrix) on all edges or a small value to avoid $p_{ij} = 0$ $\forall j$

2) Ant Path Construction: each of the m ants will build the solution graph: use P to know where to move next.

$$p_{ij} = \frac{\tau_{ij}^{\alpha} \eta_{ij}^{\beta}}{\sum_{i \in \text{allowed}} \tau_{iu}^{\alpha} \eta_{iu}^{\beta}}$$

- p_{ij}: on-the-fly by each ant; transition prob. matrix
- τ_{ij}^{α}: pheromone
- η_{ij}^{β}: heuristic desirability, e.g. inverse distance → never changes; TSP closer cities are preferable (prior) → GREEDY
- α, β: controlles the influence of pheromons / heuristic desirability; ↑ then more blindely follow pheromones
- denominator: normalizes to 1

P represents knowledge learnt by the colony

- η_{ij} is the **Heurist Metrix**: greedy problem specific informations

3) After all ants have built solutions, update pheromons:

$$\tau_{ij} = (1-\rho)\tau_{ij} + \sum_{k=1}^{m} \Delta\tau_{ij}^{k}$$

- $(1-\rho)\tau_{ij}$: evaporation
- $\sum_{k=1}^{m} \Delta\tau_{ij}^{k}$: deposition; amount of pheromons deposited by ant k on edge (i,j)
- ρ = evaporation rate $\in (0,1]$
- prevents premature sub-optimal convergence

$$\Delta\tau_{ij}^{k} = \begin{cases} 0 & \text{otherwise} \\ \frac{Q}{L_k} & \text{if } (i,j) \text{ in ant k path} \end{cases}$$

Q → constant
L_k → tour length (cost)

4) Repeat ant construction until convergence criteria met
- return best solution (e.g. shortest path)

Good solution found → τ_{ij} increases → p_{ij} increases

↑ influences

positive feedback loop

- to remove an edge $\tau_{ij} = 0$

τ: exploitation
η: exploration

α and β balances the two behaviours

$\alpha \uparrow$ ⇒ more exploitation
$\beta \uparrow$ → less exploration (greedy)
$\rho \uparrow$ → fast forgetting, more exploration

VARIANTS

EAS (elitist Ant System): gives extra push to the best-so-far solution speeding-up convergence (+ new pheromone "elitist")

MMAS (max-min ant system): impose explicit upper and lower bound on pheromons to encouraged continued exploration

ACS (Ant colony system): only allows the best so far ant to deposit pheromones

ACO encodes the state of the search in a pheromone matrix. This allows a warm start of the problem (not feasible in SA, EA, PSO)
↳ non-random start

SA → single solution based
- probabilistic acceptance of worse solutions to escape local minima

EA is population based
- CO, selection and mutation
- discrete or continous probs.

ACO → population based
- indirect communication via pheromons
- can only solve discrete problems over graph
→ particularly suited for combinatory problems like TSP

→ there is actually a extension on the continuum with probability densities
→ I could add metric by storing the best solution per each ant. The source push comes from P or the best ant only
→ It's not Markov Algorithm since it is based on the history of the past nodes visited, in order to understand where to move next → more powerful than RL

NEURAL ARCHITECTURE SEARCH (NAS)

- using GAs I could select network hyperparams → evaluation is time consuming and encoding difficult and time consuming
  - slow convergence
- using RL: hyperparms space is the environment. An agent (controller) proposes an architecture, that is trained and evaluated. The performance eg. accuracy is used as a reward signal
  → may requires many iterations to converge

DARTS differentiable architecture search
- use a continous search space and ∇ to optimize
  - significantly reduces time wrt GAs and RL
  - the supernetwork is memory intensive and we may stuck in local optima
    - it encounters also approximation challenges

1) Define a supernet that contains all possible operations in the computational graph with different edges (eg linear vs. conv3D)
   - use different weights on edges and softmax to combine the operations on one node → DIFFERENTIABLE
2) Optimize the edges weights on validation loss
3) Select the architecture with operations with highest weights

KNOWLEDGE DISTILLATION → transfer knowledge from a large teacher model to a smaller (more efficient) student model

Problems with large models 1) latency 2) resource limits (e.g. edge devices) 3) memory footprint

- KD has better performance than a direct training + converge more quickly
  → student learns from imitating internal states of teacher and/or labels

- ideas from 90s and formalized in 2015

→ FOR CLASSIFICATION (original paper) [multiclass]

- teacher uses softmax as final layer → the full distribution serves as soft target for the student
  → this will improve the model performance (eg. the relative probabilities of the wrong classes helps the model to learn)
  → a cat is more similar to a truck than a carrot

PROBLEM: The CE loss will not change much since the probabilities are so close to zero

=> define a **temperature** parameter for the smoothness of the softmax $p_i = \frac{e^{z_i/T}}{\sum_j e^{z_j/T}}$

- higher means less sharp (smooth) → more easy to learn
- In earlier implementation I have the model to match directly the **logits** (not so small, but more unstable) → almost the same if I zero mean the logits with high temp softmax

The knowledge of a teacher is passed with **distillation signals** (e.g. softmax)

- loss for classification: $\mathcal{L} = D_{KL}(p^t \| p^s) = \sum p_i^t \log \frac{p_i^t}{p_i^s}$
- loss for regression: $\mathcal{L} = \| y^t - y^s \|_2^2$ (MSE) → High precision logs, refined/denoised estimate of ground truth

Intermediate **feature matching** $\mathcal{L} = \| f^t(x) - f_w(f^s(x)) \|_2^2$ (f_w: to project features with correct shapes)

- in addition or in substitution

↳ for a specific layer

Gradient Matching $\mathcal{L}_{GR} = \| \nabla_x \mathcal{L}^t - \nabla_x \mathcal{L}^s \|_2^2$ (same loss (e.g. CE)) → reduce distance between gradients wrt x

→ the loss is added to the student overall loss
→ learns teacher inputs sensitivity → but requires an additional backward pass
- could suffer from vanishing gradients problem especially in deep architectures

Adversarial Gradient Matching: a new model (discriminator D) learns to distinguish between teacher and student gradients $\nabla_x \mathcal{L}^s$ vs. $\nabla_x \mathcal{L}^t$

The student learns to fool D and create gradients with same distribution as teacher

$\mathcal{L} = -\log D(\nabla_x \mathcal{L}^s(x))$ → to be minimized

↳ =1 when student fools D

→ gradients wrt Intermediate Layers (intermediate feature map) $\nabla_f \mathcal{L}$ eg. here 

- match how the teacher will correct the intermediate features to lower the loss

→ since copying gradient path wrt the weight is infeasible

1) using the teacher checkpoints for the distillations signals → CURRICULUM LEARNING
2) preserving feature space similarity

$\mathcal{L}_{total} = \alpha \mathcal{L}_{task} + \sum_k \lambda_k \mathcal{L}^k_{distill}$

↳ wrt G.T. HARD ↳ distill signals

eg for classification

$\mathcal{L}_T = (1-\lambda)\mathcal{L}_{task} + \lambda \mathcal{L}_{soft}$

DISTILBERT → using temperature probs and internal hidden states during training