Neuro-Evolution and Gradient-Free Optimization

Intelligent Systems II

Mário Antunes

2026

Universidade de Aveiro

Part I: Foundations of Neural Computation

The Multilayer Perceptron (MLP)

Definition: An MLP is a universal function approximator $f : \mathbb{R}^n \to \mathbb{R}^m$ composed of layers of parameterized non-linear transformations.

The Neuron (Perceptron): The fundamental unit processing an input vector **x**:

$$y = \varphi(\mathbf{w}^T\mathbf{x} + b)$$

where:

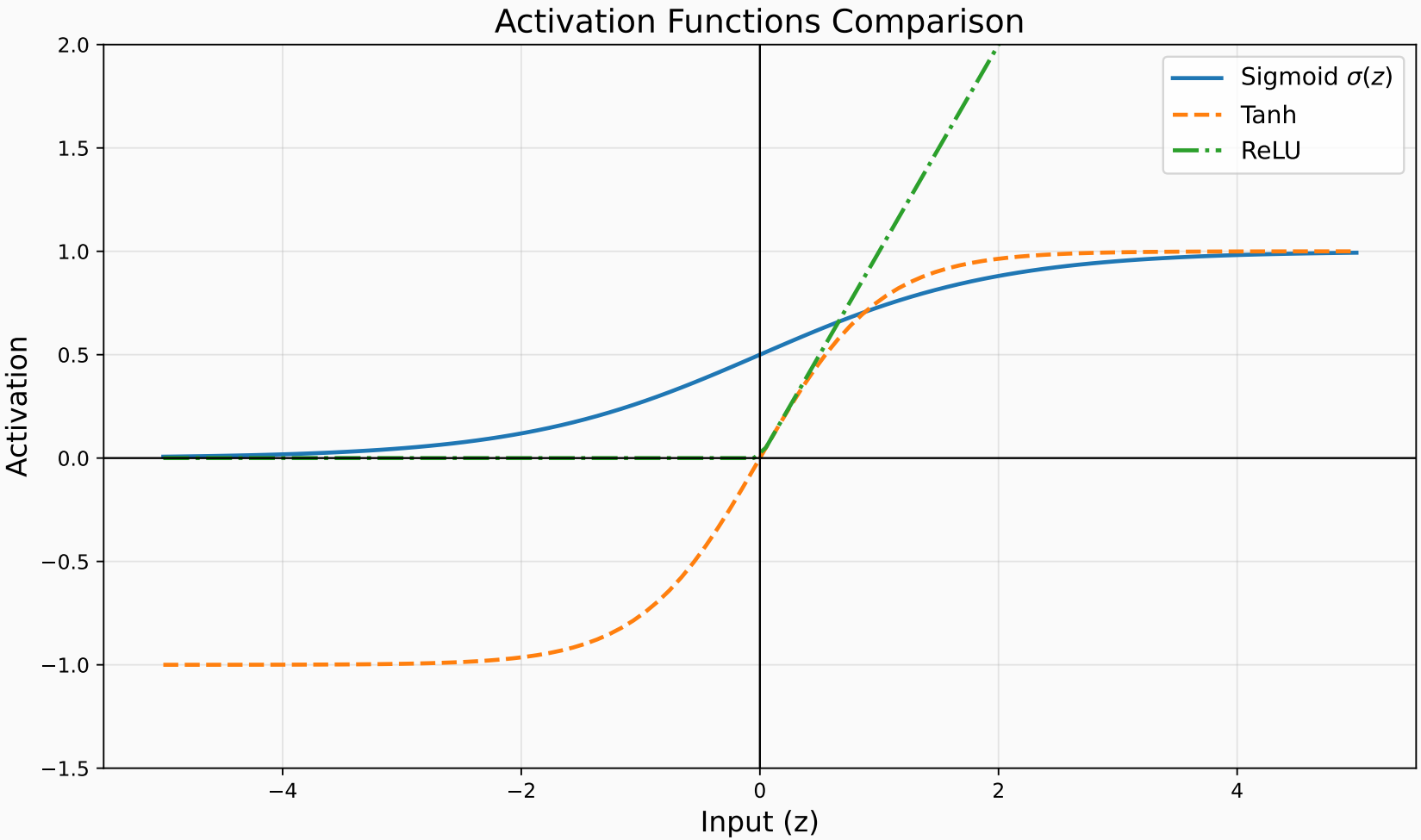
- $\mathbf{w} \in \mathbb{R}^n$: Weight vector (synaptic strength).
- $b \in \mathbb{R}$: Bias (activation threshold).
- $\varphi(\cdot)$: Non-linear activation function.

Activation Functions: The Source of Non-Linearity i

Without φ, a neural network is merely a linear regression model ($W_2(W_1 x) = W_{new} x$).

- **Sigmoid**: $\sigma(z) = \frac{1}{1+e^{-z}}$
  - *Range*: $(0, 1)$. Used for probabilities.
  - *Issue*: Vanishing gradient for large $|z|$.
- **Tanh**: $\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$
  - *Range*: $(-1, 1)$. Zero-centered.
- **ReLU (Rectified Linear Unit)**: $R(z) = \max(0, z)$
  - *Standard*: Solves vanishing gradient for positive inputs.

Activation Functions: The Source of Non-Linearity ii



Activation Functions

Forward Propagation: The Inference Step

Let L be the number of layers. For layer $l = 1 \dots L$:

1. **Pre-activation (z)**: Linear combination of inputs from the previous layer.

$$\mathbf{z}^{[l]} = \mathbf{W}^{[l]}\mathbf{a}^{[l-1]} + \mathbf{b}^{[l]}$$

   - $\mathbf{W}^{[l]}$: Weight matrix connecting layer $l-1$ to l.
   - $\mathbf{a}^{[l-1]}$: Output (activation) from the previous layer.

2. **Activation (a)**: Applying the non-linearity.

$$\mathbf{a}^{[l]} = \varphi^{[l]}(\mathbf{z}^{[l]})$$

- **Analogy**: A signal passing through a series of logical gates or filters, where each filter detects increasingly complex features (edges $\rightarrow$ shapes $\rightarrow$ objects).

Backpropagation: Deep Derivation i

Goal: Minimize Loss $\mathcal{L}(\hat{y}, y)$, e.g., Mean Squared Error: $\mathcal{L} = \frac{1}{2}\|\mathbf{a}^{[L]} - \mathbf{y}\|^2$.

We need the gradient $\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{[l]}}$ to update weights. We use the **Chain Rule**.

The Error Term (δ): Let $\delta^{[l]} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{[l]}}$ be the "error" at layer l.

For the Output Layer (L):

$$\delta^{[L]} = \frac{\partial \mathcal{L}}{\partial \mathbf{a}^{[L]}} \cdot \frac{\partial \mathbf{a}^{[L]}}{\partial \mathbf{z}^{[L]}} = (\mathbf{a}^{[L]} - \mathbf{y}) \odot \varphi'(\mathbf{z}^{[L]})$$

- $\odot$: Hadamard (element-wise) product.
- φ': Derivative of activation function.

Backpropagation: Deep Derivation ii

Backpropagating the Error (Recursion): How much did layer l contribute to the error in layer $l+1$?

$$\delta^{[l]} = (\mathbf{W}^{[l+1]T}\delta^{[l+1]}) \odot \varphi'(\mathbf{z}^{[l]})$$

***Interpretation**: We project the error from the future layer back through the weights, scaled by the gradient of the activation (if the neuron didn't fire, it isn't responsible for the error).*

Backpropagation: Deep Derivation iii

Calculating Gradients:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{[l]}} = \delta^{[l]} (\mathbf{a}^{[l-1]})^T$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}^{[l]}} = \delta^{[l]}$$

***Summary**: The gradient is the outer product of the error at the current layer and the input from the previous layer.*

Part II: Representation & Limits

Knowledge Representation: The Manifold Hypothesis i

The Goal of Deep Learning: To transform highly entangled raw data (e.g., pixels) into a **Latent Space** where classes are linearly separable.

- **Transformation**: The network performs a series of coordinate transformations (changes of basis).

$$\mathbf{x} \text{ (Input)} \xrightarrow{\mathbf{W}_1} \mathbf{h}_1 \xrightarrow{\mathbf{W}_2} \dots \xrightarrow{\mathbf{W}_L} \mathbf{z} \text{ (Latent Feature)}$$

Knowledge Representation: The Manifold Hypothesis ii

- **The Manifold Hypothesis**: Real-world data lies on a lower-dimensional manifold embedded within the high-dimensional input space.
- **Analogy**: Imagine two colored classes drawn on a crumpled sheet of paper.
  - *Input Space*: The paper is crumpled (non-linear, complex).
  - *Latent Space*: The network "unfolds" the paper until it is flat.
  - *Output*: Now, a single straight line (linear classifier) can separate the colors.

Knowledge Representation: The Manifold Hypothesis iii

Bridge to Autoencoders: If the network can distill the "essence" of the data into this compact latent vector **z**, can we reverse the process to reconstruct the original data from these features alone?

Autoencoders: Manifold Learning

Unsupervised learning to discover internal structure.

$$x \xrightarrow{\text{Encoder}} h \xrightarrow{\text{Decoder}} x'$$

Objective: Minimize $\|x - x'\|^2$.

- **Latent Space (h)**: A compressed representation of the data.
- **Analogy**: A zip file compression. The compressor must find the redundant patterns to shrink the file size.
- **Significance**: Proves that NNs can learn abstract concepts without explicit labels.

From Autoencoders to Generative AI (LLMs) i

The Evolution: Autoencoders taught us that we can learn a **Compressed Representation** (z) of data without labels. This concept scales massively.

- **Denoising Autoencoders**:
  - *Task*: Train the network to recover the original input x from a corrupted version $\tilde{x}$.
  - *Insight*: Forces the model to learn the robust *structure* of the data manifold, not just copy it.

From Autoencoders to Generative AI (LLMs) ii

- **Transformer Architecture (BERT/GPT)**:
  - **Masked Language Modeling (BERT)**: Hide 15% of words in a sentence and predict them. This is effectively a discrete Denoising Autoencoder.
  - **Next Token Prediction (GPT)**: Given the context $x_{1:t}$, predict x_{t+1}. The "Latent Space" here captures syntax, semantics, and world knowledge.

Modern LLMs are essentially massive, autoregressive feature extractors. They learn a manifold where "King - Man + Woman $\approx$ Queen" holds true.

Neural Networks as Continuous World Models i

The Challenge: In Neuro-evolution (and Reinforcement Learning), the agent must navigate a complex, high-dimensional, and continuous world (e.g., a robot in a physics sim).

The Solution: Instead of programming "If $x > 5$ then Jump" (Logic), we evolve a Neural Network. Why?

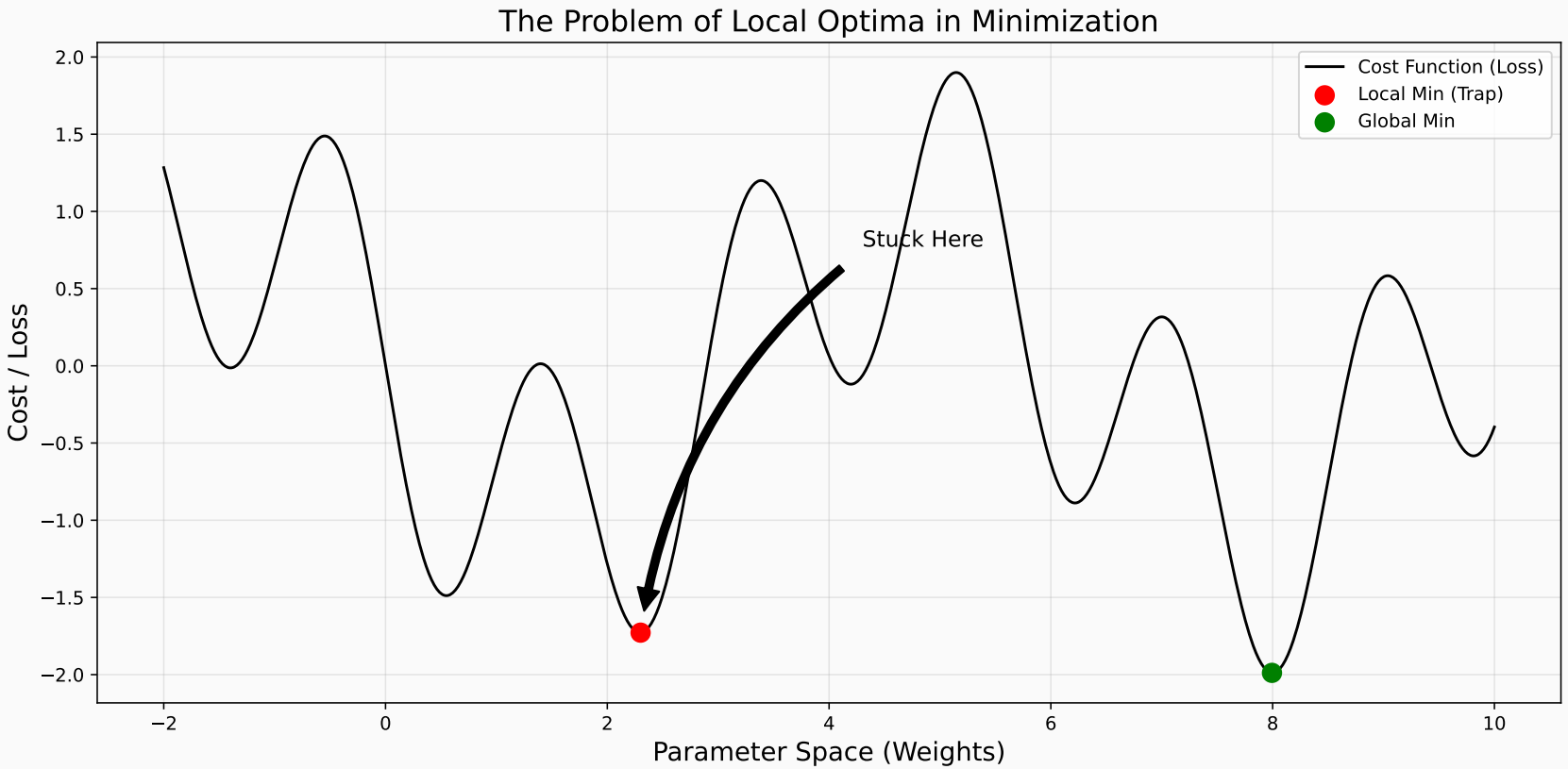
Neural Networks as Continuous World Models ii

- **Manifold Projection**:
  - The network projects the infinite states of the continuous world onto a simpler, lower-dimensional **Decision Manifold**.
  - *Example*: A self-driving car sees millions of pixel combinations. The network compresses this to a simple latent variable: "Distance to Center Lane."
- **Smooth Interpolation**:
  - Because of activation functions (Sigmoid/Tanh), the network **interpolates** between known states.
  - If the agent learns to handle $Speed = 10$ and $Speed = 20$, the topology of the neural manifold likely handles $Speed = 15$ correctly without ever seeing it. This "generalization" is what evolution optimizes.

The Limits of Supervised Learning i

1. **Gradient Dependence**: Requires a smooth, differentiable Loss landscape.
    - *Fail Case*: Discrete choices (e.g., “Go Left” vs “Go Right” in a maze).
2. **Local Optima**: Gradient Descent is a “greedy” search. It follows the steepest slope, potentially getting trapped in a suboptimal valley.
3. **Data Hunger**: Backprop requires massive datasets ($N > 10^4$) to approximate the true gradient.
4. **Vanishing Gradients**: In deep networks, $\prod \varphi'(z) \approx 0$, stopping learning in early layers.

The Limits of Supervised Learning ii



Limits of Supervised Learning: Optimization Landscape

Part III: Gradient-Free Optimization (Blind Search)

Hill Climbing (HC)

Math:

$$\mathbf{x}_{new} = \mathbf{x}_{current} + \mathcal{N}(0, \sigma)$$

$$\text{If } f(\mathbf{x}_{new}) > f(\mathbf{x}_{current}) \implies \mathbf{x}_{current} \leftarrow \mathbf{x}_{new}$$

Analogy: A climber in thick fog trying to reach the peak of Everest. They take a step; if they go up, they stay there. If they go down, they step back.

Example: Tuning a PID controller. Randomly tweak K_p. If the robot shakes less, keep the new K_p.

Limitation: Cannot cross a valley to reach a higher peak (Local Optima).

Simulated Annealing (SA)

Math: Probability of accepting a *worse* solution $(f(x_{new}) < f(x_{curr}))$:

$$P(\text{accept}) = \exp\left(\frac{f(\mathbf{x}_{new}) - f(\mathbf{x}_{current})}{T_k}\right)$$

Temperature decay: $T_{k+1} = \alpha T_k$ where $\alpha \approx 0.99$.

Analogy: Metallurgy. Heating metal allows atoms to move freely (high energy/randomness). Cooling it slowly (annealing) allows them to settle into a perfect crystal structure (global optimum).

Example: Scheduling classes. Early in the search, we accept a schedule with conflicts (high T) to explore radically different timetables. Later, we only accept improvements (low T).

Comparison: Greedy vs. Stochastic Search i

Feature	Hill Climbing (HC)	Simulated Annealing (SA)
Strategy	**Greedy**: Always takes the best immediate step.	**Probabilistic**: Sometimes accepts worse steps to explore.
Speed	Very Fast (Rapid convergence).	Slower (Depends on cooling schedule).
Local Optima	**High Risk**: Gets stuck in the first peak it finds.	**Low Risk**: Can "jump" out of local peaks early on.
Tuning	None (Deterministic).	Difficult (T_{start}, cooling rate α).
Analogy	Climbing a mountain in fog with amnesia.	Bouncing a ball; it settles in the deepest hole as energy drops.

Comparison: Greedy vs. Stochastic Search ii

Key Insight:

1. Use **HC** when you need a *good* solution instantly (e.g., real-time control).
2. Use **SA** when you need the *best* solution and can afford computation time (e.g., circuit layout).

Evaluation for Neuro-Evolution Tasks i

Feature	Hill Climbing (HC)	Simulated Annealing (SA)
Weight Space	**Exploitation Heavy**: Rapidly fine-tunes weights near initialization.	**Exploration Heavy**: Can jump out of local optima (e.g., a "good enough" strategy) to find distinct behaviors.
Fitness Landscape	Struggles with **Deceptive Gradients** (where a temporary drop in score is needed to learn a complex skill).	Tolerates temporary performance drops, allowing the agent to "unlearn" a bad habit.
Hyperparameters	Minimal (Step size σ).	Complex (Cooling schedule T, decay α).
Verdict	Best for **Polishing** a pre-trained agent.	Best for **Escaping** stagnation in simple control tasks.

Evaluation for Neuro-Evolution Tasks ii

Critical Flaw: Both are **Single-Point** methods. They track only *one* agent. In Neuro-evolution, we often need a **Population** to maintain diversity (e.g., one agent learns to jump high, another learns to fly low) to solve complex tasks.

Part IV: Population-Based Algorithms

Genetic Algorithms (GA)

Math/Operators:

1. **Selection**: $P(x_i) \propto \frac{f(x_i)}{\sum f(x_j)}$ (Roulette Wheel).
2. **Crossover**: $\mathbf{x}_{child} = \alpha \mathbf{x}_{parent1} + (1-\alpha)\mathbf{x}_{parent2}$.
3. **Mutation**: $\mathbf{x}_{i,j} = \mathbf{x}_{i,j} + \text{random}(-0.1, 0.1)$.

Analogy: Biological Evolution. Survival of the fittest + Sexual reproduction mixing DNA.

Example: Designing an antenna shape. Combine the "curvature" of Antenna A and the "length" of Antenna B to see if the child receives signals better.

Differential Evolution (DE)

Math:

$$\mathbf{v}_i = \mathbf{x}_a + F \cdot (\mathbf{x}_b - \mathbf{x}_c)$$
$$\mathbf{u}_i = \text{Crossover}(\mathbf{x}_i, \mathbf{v}_i)$$

where:

- $F \in [0, 2]$: Differential Weight.
- a, b, c: Random distinct indices from population.

Analogy: A team of snipers. To find a better position, Sniper A looks at the difference in position between Sniper B and Sniper C, and uses that vector to jump to a new spot.

Example: Fitting a polynomial curve to noisy data. The difference vector helps adapt the step size automatically (large difference = large step).

Particle Swarm Optimization (PSO)

Math:

$$\mathbf{v}_i^{t+1} = w\mathbf{v}_i^t + c_1 r_1(\mathbf{pbest}_i - \mathbf{x}_i^t) + c_2 r_2(\mathbf{gbest} - \mathbf{x}_i^t)$$

$$\mathbf{x}_i^{t+1} = \mathbf{x}_i^t + \mathbf{v}_i^{t+1}$$

where:

- w: Inertia (momentum).
- c_1, c_2: Cognitive vs Social coefficients.

Analogy: A flock of birds looking for food. Even if one bird doesn't see food, it follows the bird that is chirping the loudest (closest to food), while maintaining its own momentum.

Example: Controlling a swarm of drones to find a gas leak source.

Grey Wolf Optimizer (GWO)

Math: Distance to leaders (α, β, δ):

$$\mathbf{D}_\alpha = |\mathbf{C}_1 \cdot \mathbf{X}_\alpha - \mathbf{X}|$$

Position Update:

$$\mathbf{X}_1 = \mathbf{X}_\alpha - \mathbf{A}_1 \cdot \mathbf{D}_\alpha, \quad \mathbf{X}_2 = \mathbf{X}_\beta - \mathbf{A}_2 \cdot \mathbf{D}_\beta, \ldots$$

$$\mathbf{X}(t+1) = \frac{\mathbf{X}_1 + \mathbf{X}_2 + \mathbf{X}_3}{3}$$

Analogy: Pack hunting. The Alpha wolf leads the attack, Beta and Delta flank the prey, and the rest (Omega) encircle it based on the leaders' positions.

Example: Optimizing the energy consumption of a smart grid. The "prey" is the optimal configuration, and the "wolves" are the load balancers converging on it.

Part V: Neuro-Evolution

The Concept

Mapping:

- **Genotype**: A vector $\theta = [w_{1,1}, w_{1,2}, \dots, b_1, \dots]$ containing all network weights.
- **Phenotype**: The agent's behavior resulting from the policy $\pi_\theta(state)$.
- **Fitness Function**: $J(\theta) = \sum_{t=0}^{T} Reward_t$.

Process: We treat the Neural Network training as a **Global Optimization Problem**:

$$\theta^* = \arg\max_{\theta} J(\theta)$$

We solve this using DE, PSO, or GA, bypassing the need for $\nabla J(\theta)$.

The Structural Limitation: Permutation Problem i

Question: Does flattening the network damage learning?
Answer: Yes, it introduces the **Competing Conventions** (Permutation) problem.

- **Explanation**: In an MLP, hidden neurons are interchangeable. Swapping Neuron A and Neuron B (and their weights) results in the exact same function output.

The Structural Limitation: Permutation Problem ii

- **The Conflict**:
  - Parent 1 has “Feature Detector X” in Neuron 1.
  - Parent 2 has “Feature Detector X” in Neuron 5.
  - **Crossover**: Mixing the first half of Parent 1 with the second half of Parent 2 might result in a child with *two* Detector X’s or *zero*, destroying the functional structure.
- **Impact**: This makes crossover in GA less effective for large networks. PSO and DE are more robust as they operate on vector differences rather than splicing.

Summary: When to use what?

Method	Best For	Computation Cost	Math Complexity
Backprop	Supervised, differentiable, massive data	Medium (GPU optimized)	High (Calculus)
GA/DE	Discrete, non-differentiable, control	High (Parallelizable)	Low (Algebra)
PSO	Continuous, convex-like spaces	Low	Low
GWO	Complex landscapes, avoiding local optima	Medium	Medium

Initialization & Convergence

Initialization & Convergence

The “Butterfly Effect” in Neuro-Evolution.

1. **Range Matters**:
   - If $W \sim U[-0.1, 0.1]$: Network is linear (Sigmoid is linear near 0). Good for fine-tuning.
   - If $W \sim U[-10, 10]$: Network saturates (output is always 0 or 1). Agent is “stiff” and unreactive.
2. **Symmetry Breaking**:
   - Unlike Backprop, we *can* initialize weights to zero if using Mutation-only evolution (Random Search), but generally, random diversity is required for Population methods to span the space.

Beyond Randomness: Space-Filling Designs i

The Problem: Standard Uniform Random sampling (`np.random.uniform`) is not "uniform" in small samples. It creates **clusters** and **gaps**.

> *Consequence: Parts of the search space are completely unexplored during initialization.*

Beyond Randomness: Space-Filling Designs ii

1. **Hyper-Latin Cube Sampling (LHS)**

- **Method**: Stratified sampling. Divides each dimension into N intervals and places exactly one sample per interval.
- **Analogy**: The "Rook Problem". Placing N rooks on an $N \times N$ chessboard such that no two rooks share a row or column.
- **Benefit**: Guarantees coverage across every dimension independently.

Beyond Randomness: Space-Filling Designs iii

2. **Sobol Sequences (Low-Discrepancy)**

   - **Method**: Uses Gray Codes and direction numbers to generate points that maximally avoid each other.
   - **Benefit**: Asymptotically optimal space-filling. Adding a new point fills the largest existing gap.

Chaotic Initialization

Concept: Replacing the pseudo-random number generator (PRNG) with a deterministic **Chaotic Map**. Chaos is ergodic, meaning it eventually visits every state in the phase space, often faster than random noise.

The Tent Map: Used in `ChaoticSampler`.

$$x_{t+1} = \begin{cases} 2x_t & \text{if } x_t < 0.5 \\ 2(1 - x_t) & \text{if } x_t \geq 0.5 \end{cases}$$

- **Dynamics**: Acts like a "baker's transformation" (stretching and folding the space).
- **Analogy**: Kneading dough. A speck of flour (initial position) is stretched and folded until it is uniformly distributed throughout the bread.

Opposition-Based Learning (OBL) i

Theory: Proposed by Tizhoosh (2005). In the absence of a priori knowledge, the search for the optimal solution is equally likely to be in the "opposite" direction of our current guess.

The Math: For a candidate $x \in [a, b]$, the opposite number $\breve{x}$ is:

$$\breve{x} = a + b - x$$

Opposition-Based Learning (OBL) ii

- **Procedure**:
  1. Generate population P.
  2. Compute opposite population OP.
  3. Select top N individuals from $P \cup OP$.

Analogy: **The Lost Keys**. If you are looking for your keys in the living room (Current Guess) and can't find them, the highest probability is that they are in the exact opposite location (e.g., the car), rather than slightly to the left in the living room.

Quasi-Opposition Based Learning (QOBL)

Refinement: OBL checks the *exact* opposite. QOBL argues that the optimal solution likely lies *between* the center of the search space and the opposite point.

The Math: Let $C = \frac{a+b}{2}$ be the center of the search space.

$$\breve{x} = a + b - x$$
$$x_{quasi} \sim U(\min(C, \breve{x}), \max(C, \breve{x}))$$

- **Probabilistic Proof**: It has been mathematically proven that the quasi-opposite point is closer to the unknown solution than a random point more than 50% of the time.
- **Analogy**: If “North” is wrong, don’t just go “South”. Search the region *between* the Center of the map and South.

OBLESA

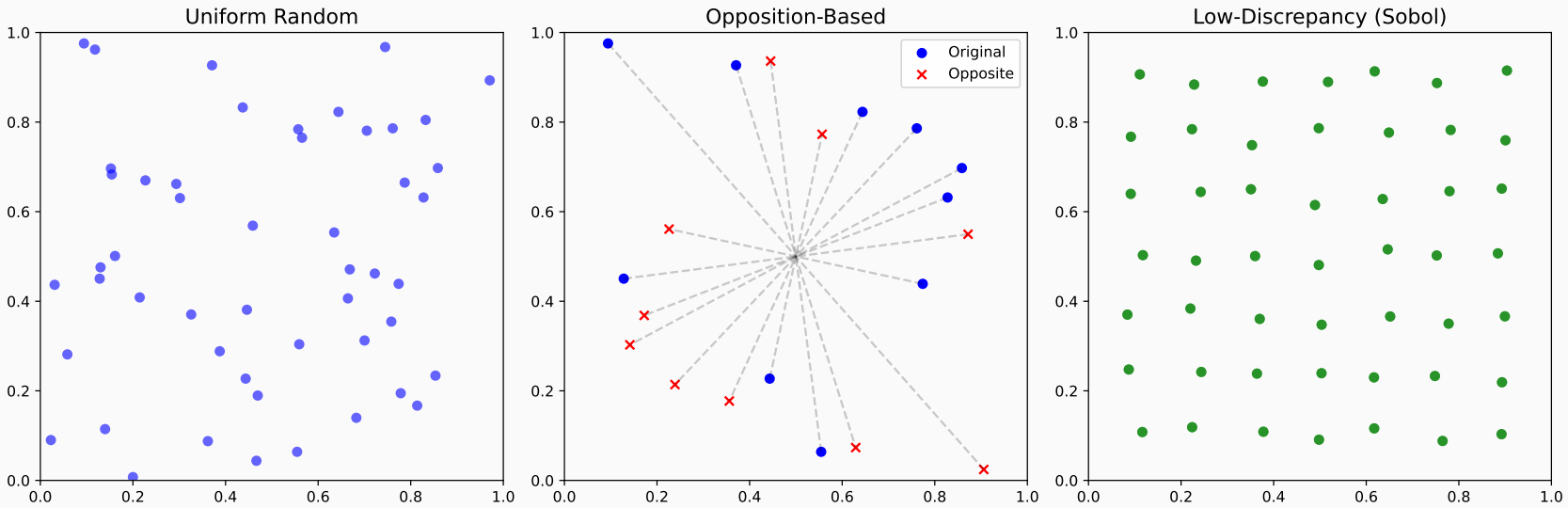
The Algorithm: `oblesa` (Opposition-Based Learning with Empty Space Search).

Mechanism:

1. **OBL**: Generates high-quality candidates by checking opposites.
2. **ESA (Empty Space Search)**: Simulates "electrons" repelling each other to find the largest empty voids in the search space.
3. **Selection**: Merges all candidates ($Random \cup Opposite \cup Repulsed$).

The Selection Metric: We select the final population based on a weighted score of **Fitness** (Performance) and **Crowding Distance** (Diversity).

Initialization Strategy i



Initialization Strategy

Initialization Strategy ii

Method	Mechanism	Computational Cost	Diversity	Best For
Random	Pure Stochastic $U(a, b)$	Low ($1 \times N$)	Low (Clumping)	Baseline / Simple problems
Sobol / LHS	Stratified / Low-Discrepancy	Low ($1 \times N$)	High (Space Filling)	High-dimensional, expensive functions
OBL / QOBL	Symmetry (x vs $\breve{x}$)	Medium ($2 \times N$ evals)	Medium (Focuses on symmetry)	Accelerating convergence in centered problems
OBLESA	Repulsion + Opposition	High (Physics Sim + $2 \times N$)	**Maximum**	Complex landscapes where getting stuck is fatal

Initialization Strategy iii

Key Takeaway: "There is no free lunch."

- **OBL/QOBL** pays double the evaluation cost to potentially skip early generations.
- **OBLESA** pays a high pre-computation cost to ensure the population never starts in a "corner."
- **Sobol** is the "safe bet" for almost any black-box task.

Summary

1. **The Shift**: We moved from **Logic** (Exact, Brittle) to **Neural Networks** (Approximate, Robust).
2. **The Learning**: We replaced **Backpropagation** (Gradient-based, Local) with **Evolution** (Population-based, Global).
3. **The Criticality**: We discovered that *how* you start (Initialization) and *how* you search (Mutation/Crossover) determines if you find the solution or get stuck in a local optimum.

Final Thought

“In a continuous, non-differentiable world, the ‘smartest’ agent is not the one with the most complex brain, but the one that explores the search space most effectively.”