Neuroevolution for Breast Cancer Classification

Implementation of an MLP Optimized by Genetic Algorithms

Pedro Medina

December 2025

Abstract

This project explores the application of Neuroevolution—the use of evolutionary algorithms to generate artificial neural networks—to the field of medical diagnosis. Specifically, a Multi-Layer Perceptron (MLP) was built from scratch using NumPy and trained using a Genetic Algorithm to classify breast tumors as Malignant or Benign using the Wisconsin Diagnostic Breast Cancer (WDBC) dataset. The experiments demonstrate that stochastic optimization can effectively train a neural network without gradient-based methods, achieving a test accuracy of approximately 92%. Furthermore, the implementation of an adaptive mutation mechanism proved essential in overcoming local optima and preventing stagnation during the evolutionary process.

Contents

1 Introduction

Artificial Neural Networks (ANNs) are typically trained using gradient-based methods such as Backpropagation. However, in scenarios where the gradient is not available or the search space is highly complex, Evolutionary Algorithms (EAs) offer a robust alternative. This approach is known as *Neuroevolution*.

The objective of this project is to implement a complete neuro-evolutionary system from scratch. The system is applied to a binary classification problem: diagnosing breast cancer based on 30 numeric features extracted from digitized images of a fine needle aspirate (FNA) of a breast mass.

2 Methodology

The project relies on a custom implementation of both the neural network and the optimization algorithm, avoiding high-level deep learning libraries such as PyTorch or TensorFlow for the core logic.

2.1 Data Preprocessing

The WDBC dataset consists of 569 samples with 30 features each. The preprocessing pipeline includes the following.

- **Cleaning:** Handling missing values (NaNs).
- **Encoding:** Mapping the target labels 'M' (Malignant) to 1.0 and 'B' (Benign) to 0.0.
- **Splitting:** Divide the data into training sets (80%) and testing (20%) using stratified sampling to maintain class balance.
- **Scaling:** Apply `StandardScaler` to normalize features ($\mu = 0, \sigma = 1$), fitting only in the training set to prevent data leakage.

2.2 Neural Network Architecture (MLP)

The Multi-Layer Perceptron was implemented as a configurable class in `mlp.py`.

- **Structure:** The architecture is dynamic. For the baseline experiment, a topology of $[30, 20, 10, 1]$ was selected based on preliminary research carried out in PyTorch.
- **Activation Functions:**
  - **ReLU** ($f(x) = \max(0, x)$) is used for hidden layers.
  - **Sigmoid** ($f(x) = \frac{1}{1+e^{-x}}$) is used for the output layer to provide a probability between 0 and 1.
- **Parameter Mapping:** The network includes a method to "un-flatten" a 1D chromosome vector into the specific weight matrices and bias vectors required by the architecture.

2.3 Evolutionary Algorithm

The optimization engine is a Genetic Algorithm implemented in `evolutionary_algorithm.py`.

- **Representation:** Each individual is a `Chromosome` containing a flat list of real-valued genes (weights and biases). For the $[30, 20, 10, 1]$ architecture, the length of the chromosome is 841 genes.
- **Fitness Function:** Fitness is defined as the classification accuracy in the training set.
- **Selection:** Tournament selection is used to choose parents.
- **Crossover:** Arithmetic crossover combines two parents to produce offspring.
- **Mutation:** Random mutation is applied with a defined probability to maintain diversity.

3 Experimental Results

To evaluate the performance of the neuroevolutionary system, a suite of four distinct experiments was conducted. These experiments were designed to test different neural network topologies (architectures) and population scales, ranging from simple configurations to deeper and wider networks.

3.1 Experimental Setup

The hyperparameters for the Genetic Algorithm were kept constant across most experiments to isolate the impact of the architecture, with the exception of the preliminary test (Experiment 0).

- **Crossover Rate:** 0.8
- **Mutation Rate:** 0.2 (Adaptive)
- **Selection:** Tournament Selection

Table 1 details the specific configurations for each experiment executed by the orchestrator.

Exp ID	Architecture (Layers)	Population	Generations
0	$[30, 15, 1]$ (Shallow - Test)	10	50
1	$[30, 20, 10, 1]$ (Baseline)	100	300
2	$[30, 50, 1]$ (Wide)	100	300
3	$[30, 10, 10, 5, 1]$ (Deep)	100	300

Table 1: Configuration of the experimental suite.

3.2 Performance Analysis

The following figures illustrate the fitness evolution (training accuracy) over the generations for each configuration.

Experiment 0: Preliminary Test

This initial experiment used a small population and few generations to validate the pipeline.

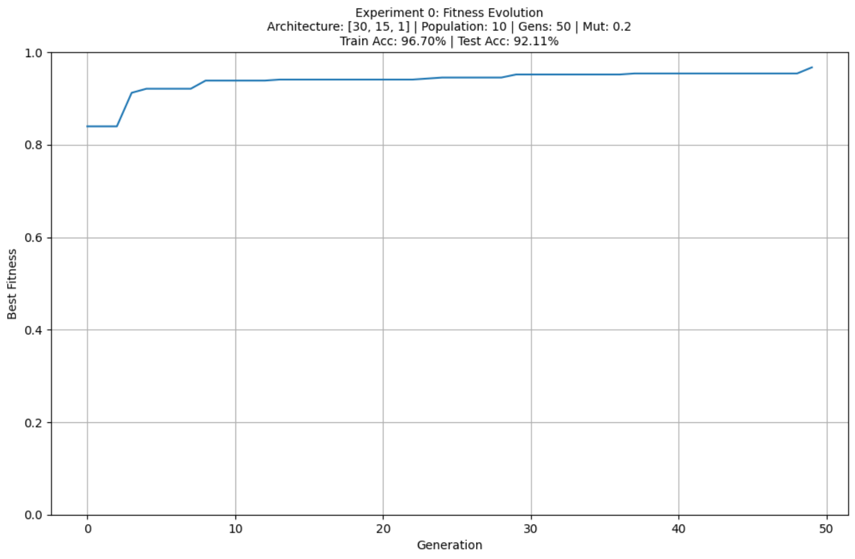

Figure 1: Fitness evolution for Architecture [30, 15, 1].

Experiment 1: Baseline Architecture

Using the architecture derived from the initial research phase $[30, 20, 10, 1]$.

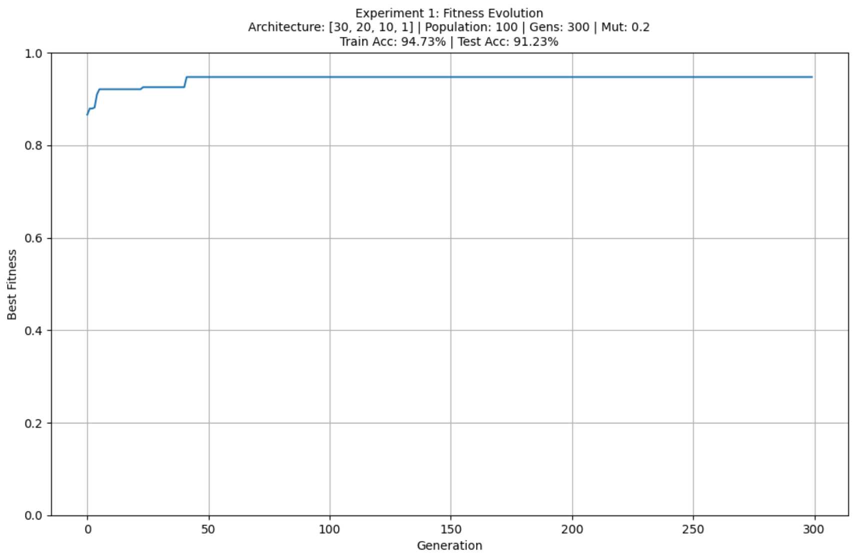

Figure 2: Fitness evolution for the Baseline Architecture.

Experiment 2: Wide Architecture

Testing the Universal Approximation Theorem hypothesis with a single, wide hidden layer $[30, 50, 1]$.

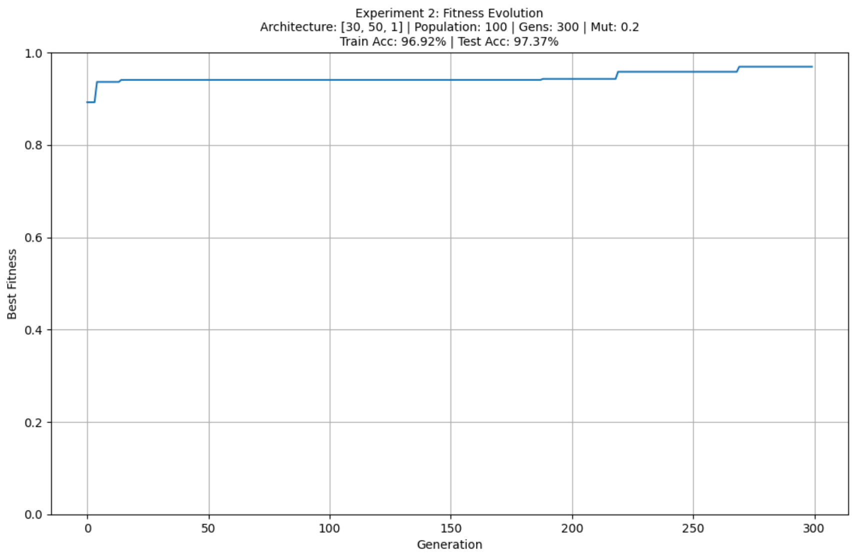

Figure 3: Fitness evolution for the Wide Architecture.

Experiment 3: Deep Architecture

Testing a deeper network $[30, 10, 10, 5, 1]$ to capture potentially more complex hierarchical features.

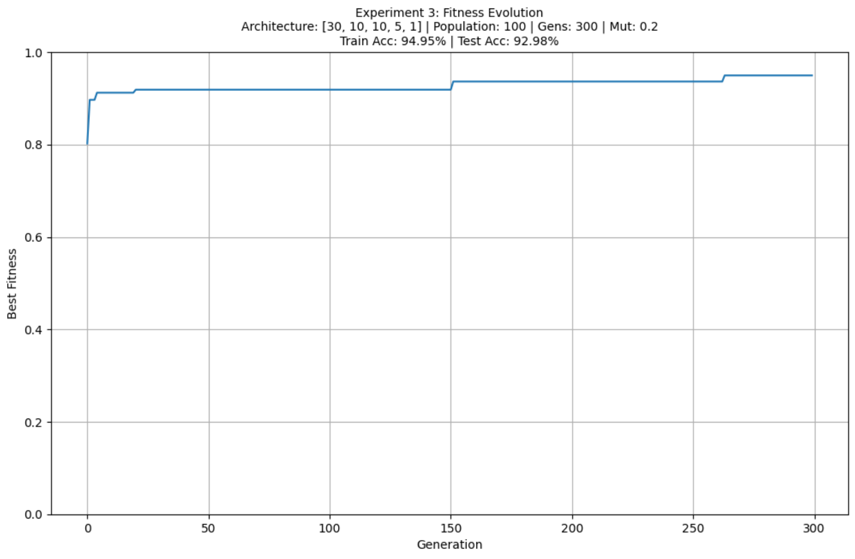

Figure 4: Fitness evolution for the Deep Architecture.

3.3 Summary of Results

Table 2 summarizes the final metrics obtained for each experiment.

Exp	Architecture	Train Accuracy	Test Accuracy
0	$[30, 15, 1]$	**96.70%**	**92.11%**
1	$[30, 20, 10, 1]$	**94.73%**	**91.23%**
2	$[30, 50, 1]$	**96.92%**	**97.37%**
3	$[30, 10, 10, 5, 1]$	**94.95%**	**92.98%**

Table 2: Final accuracy comparison across all experiments.

The results indicate that the baseline architecture we obtained from the PyTorch implementation turned out to be the worst performing in the test set. As we can see in the plots, the model tends to stagnate. Fortunately, we could say that the model works quite well under unknown data. The accuracy between train and test is very small, which says that the model is not overfitting and the neural network has generalized correctly.

In addition, this accuracy is surprisingly good, knowing that we do not use gradients for the implementation. Those points below perfection were assumed before starting, since the evolutionary algorithm is not that accurate.

4 Project Evolution and Methodology

The development of this project can be divided into three distinct phases. This approach allowed for the validation of the core concepts before tackling the complexity of a custom implementation.

4.1 Phase 1: Research & Prototyping (PyTorch)

Goal: To establish a baseline and determine the optimal Neural Network architecture without struggling with low-level implementation.

In this preliminary phase, the `PyTorch` library was utilized to rapidly experiment with different network topologies.

- **Input:** 30 features from the WDBC dataset.
- **Outcome:** Through iterative testing, a topology of $[30, 20, 10, 1]$ (30 input neurons, two hidden layers of 20 and 10 neurons, and 1 output neuron) was identified as the most effective, achieving an accuracy of approximately 98%.
- **Activation Functions:** ReLU for hidden layers and Sigmoid for the output were selected as the standard configuration.

This phase provided the "blueprint" architecture for the subsequent custom implementation.

4.2 Phase 2: Implementation "From Scratch" (NumPy)

Goal: To demonstrate a deep understanding of the underlying mathematics by rebuilding the neural network logic without automatic differentiation libraries.

All core components were implemented using only `NumPy` for matrix operations:

1. **The MLP Engine (`mlp.py`):** A flexible class capable of constructing a neural network from a dynamic list of layer sizes. It includes manual-implemented *feed-forward* logic.

2. **The Evolutionary Algorithm (`evolutionary_algorithm.py`):** A generic optimization engine implementing Tournament Selection, Arithmetic Crossover, and Random Mutation.

3. **The "Glue System" (`cancer_problem.py`):** A custom interface designed to map the generic "genes" of a chromosome to the specific weights and bias matrices of the MLP.

4.3 Phase 3: Experiment Automation

Goal: To perform rigorous hyperparameter tuning and stability testing through automation. Capable of running experiments for different network architectures and plotting the results of each one of them

The initial execution script evolved into a complete automated script (`main.py`). This system allows for:

- **Configurable Suites:** Defining a list of experimental configurations (dictionaries) to vary population size, mutation rates, and architectures sequentially.
- **Automated Execution:** Running multiple simulations without human intervention.
- **Visualization:** Automatically generating and saving convergence plots (fitness over generations) to the `results/` directory for analysis.

4.4 Phase 4: Adaptive Parameter Control

During the experimentation phase, a tendency towards premature convergence (stagnation) was observed, where the population fitness would stop improving for many generations. To mitigate this, an **Adaptive Mutation Strategy** was implemented within the optimization engine.

The algorithm monitors the best fitness score across generations. If no improvement is detected for a predefined threshold (e.g., 50 generations), the system triggers a *Hyper-Mutation Mode*:

- **Stagnation Detection:** A counter tracks consecutive generations without a new global best fitness.
- **Dynamic Adjustment:** Upon reaching the threshold, the mutation rate is temporarily increased significantly (e.g., from 0.2 to 0.5).
- **Recovery:** Once a better individual is found, the mutation rate reverts to the baseline value, allowing for fine-tuning.

This mechanism successfully forces the population out of local optima by injecting a burst of genetic diversity when needed. All experiments start with a `mutation_rate=0.2`.

5 Future Work

Although the current implementation achieves satisfactory results, several avenues for further improvement have been identified for future iterations of this project:

- **Early Stopping:** Currently, the algorithm runs for a fixed number of generations. Implementing an early stopping mechanism would halt the training once the fitness stabilizes or the generalization gap widens, saving computational resources and preventing potential overfitting.

6 Conclusions

This project demonstrated that it is possible to train a neural network using evolutionary principles. While computationally more expensive than gradient descent for this specific problem size, the approach offers great flexibility and requires no differentiable activation functions. This approach also allows me to fully comprehend the math and core concepts behind neural networks, as I had never "hard-coded" one before.

7 Code Availability

The complete source code for this project, including the implementation of the Multi-Layer Perceptron, the Evolutionary Algorithm, and the experiment orchestrator, is hosted on GitHub.

For the most up-to-date version of the implementation and access to the raw experimental data, please refer to the repository at:

`https://github.com/pedromedinatech/Neural-Network-Project.git`

References

[1] Wisconsin Diagnostic Breast Cancer (WDBC) Dataset. UCI Machine Learning Repository. Available at: `https://archive.ics.uci.edu/dataset/17/breast+cancer+wisconsin+diagnostic`

[2] Nielsen, M. A. (2015). *Neural Networks and Deep Learning.* Determination Press. Available at: `http://neuralnetworksanddeeplearning.com/`

[3] Pinkus, A. (1999). Approximation theory of the MLP model in neural networks. *Acta Numerica*, 8, 143-195. Available at: `https://pinkus.net.technion.ac.il/files/2021/02/acta.pdf`

[4] Henrythe9th. (n.d.). *AI-Crash-Course.* GitHub Repository. Available at: `https://github.com/henrythe9th/AI-Crash-Course`