# Potentiation Learning Artificial Neural Network & Eugenic NeuroEvolution

Author: Hasan Can Beydili

tchasancan@gmail.com

## Abstract

In this paper, I aim to present a novel artificial neural network algorithm capable of learning complex relationships while maintaining a simple model architecture, contributing to the field of Explainable Artificial Intelligence (XAI)

This algorithm not uses backpropagation, calculus and hidden layer.

The algorithm was designed to address the gap in the literature regarding the lack of an architecture that is "as transparent as logistic regression and as powerful as Multi Layer Perceptrons (MLP)."

In the realm of artificial intelligence, Long-Term Potentiation serves as a powerful metaphor for understanding how neural systems can store and process information. Long-Term Potentiation is a biological process wherein synaptic connections between neurons are strengthened through repeated stimulation, forming the basis for long-term memory and learning in the brain. Similarly, the Potentiation Learning Artificial Neural Network incorporates a mechanism analogous to Long-Term Potentiation to facilitate learning and memory storage.

Long-Term Potentiation, a phenomenon in neuroscience, refers to the long-lasting strengthening of synaptic connections between neurons. It occurs when synapses are repeatedly activated or strongly stimulated, resulting in enhanced neural transmission. This process plays a crucial role in learning and memory formation and typically involves chemical and structural changes in synaptic connections.

Additionally, the new genetic optimization algorithm I developed to enhance this novel approach has set **new achivements** in several reinforcement learning (RL) environments available in OpenAI Gym. This highly efficient algorithm shows great promise for RL applications. In some cases, I achieved up to a **500% improvement in performance-to-time ratio** compared to existing methods.

# 1. – Introduce

Consider a neural network architecture that eliminates the need for gradient calculation processes inherent in traditional backpropagation and bypasses the repetitive iterative steps. This novel approach offers a more interpretable alternative. Moreover, it redefines the concept of the 'hidden layer' with an innovative design, providing a fresh perspective on this traditionally opaque component.

In the existing literature, research in the field of XAI predominantly focuses on analyzing and improving the interpretability of models trained with conventional MLP architectures [1]. However, this study introduces a fundamentally new learning architecture that has the potential to address these interpretability challenges at their core.

This paper presents the Potentiation Learning Artificial Neural Network (PLAN), a neural network architecture built upon these principles. Through empirical evidence and examples from various projects, I aim to substantiate these claims. The goal is to develop models that are as transparent as logistic regression, while maintaining the computational power and complexity of multi-layer perceptrons.

# 2. – Architectures

## 2.1. – What is Logistic Regression & Multi Layer Perceptron ?

### 2.1.1. – Logistic Regression

**Definition**

Logistic regression is a statistical model that establishes a linear relationship between the input features and the target variable. It is primarily used for classification tasks, especially binary classification, by predicting the probability of a target class using the logistic (sigmoid) function. Unlike more complex models, logistic regression does not incorporate non-linear transformations or hidden layers.[2]

**Mathematical Representation:**

$$P(y = 1 \mid x) = \sigma(w^T x + b), \quad \sigma(z) = \frac{1}{1 + e^{-z}}$$

where ‘w’ represents the feature weights, ‘x’ the input features, and ‘σ’ the sigmoid function.

**Why It Does Not Learn Complex Relationships**

**Logistic regression’s inability to learn complex relationships stems from its architecture:**

a. No Activation Transformations: Logistic regression relies solely on a linear combination of input features, followed by a sigmoid function to output probabilities.

The absence of non-linear activation functions prevents the model from capturing non-linear dependencies in the data.

b. No Hidden Layers: Without hidden layers, the model lacks the hierarchical feature extraction capabilities necessary for learning complex patterns.

## Interpretability

Logistic regression is considered a highly interpretable model due to the following characteristics:

- **Direct Weight Interpretation:** Each weight(w) corresponds to the impact of a specific feature on the target variable. A positive weight indicates a positive correlation, while a negative weight indicates the opposite.
- **Feature Importance:** The magnitude of the weights provides a straightforward measure of feature importance.
- **Decision Boundary:** The linear nature of the decision boundary makes it easy to understand and visualize.
- **Simplicity:** The model's architecture ensures that predictions are easily explainable, even to non-technical audiences.

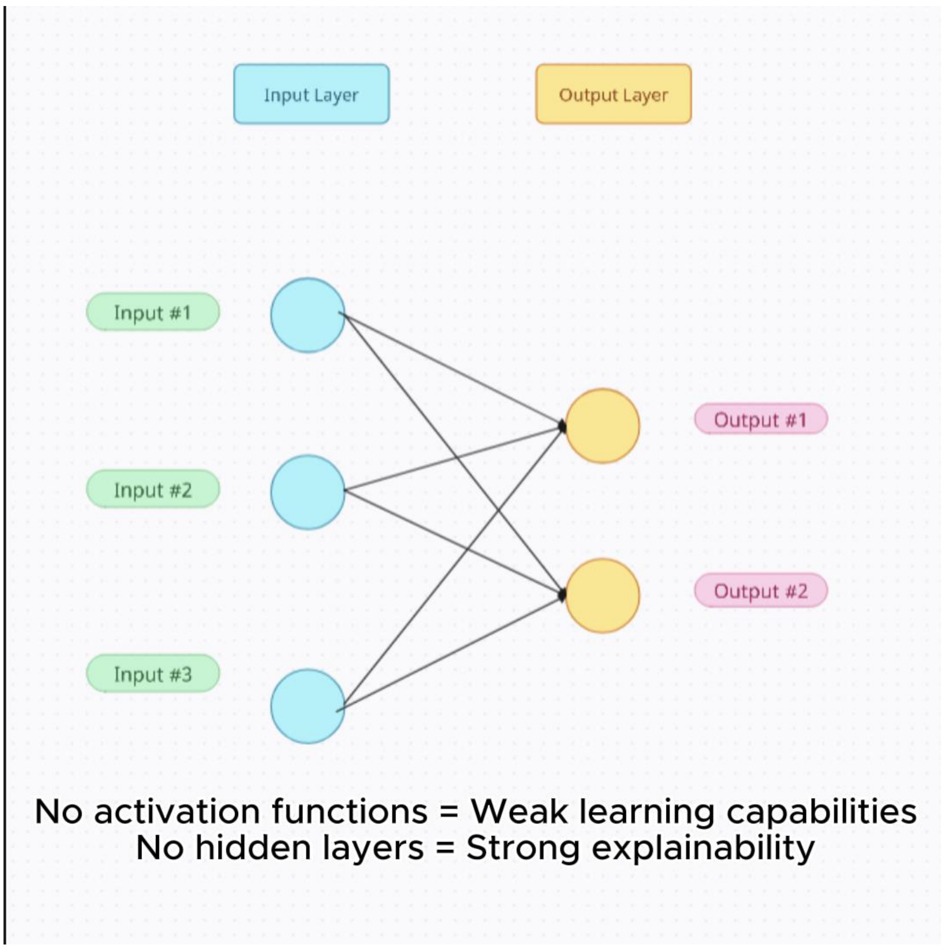

Input Layer
Output Layer
Input #1
Input #2
Input #3
Output #1
Output #2
No activation functions = Weak learning capabilities
No hidden layers = Strong explainability

***Figure 1: Example Logistic Regression Architecture***

### 2.1.2. – Multilayer Perceptron (MLP)

**Definition**

The Multilayer Perceptron is a type of artificial neural network capable of learning non-linear relationships between input features and the target variable. It consists of an input layer, one or more hidden layers, and an output layer, with each layer applying a combination of linear transformations and non-linear activation functions.[3] It is also a machine learning algorithm with a broad range of applications, extensively utilized in GPT (Generative Pretrained Transformers) models and other popular Large Language Models (LLMs).

**Mathematical Representation (for single hidden layer):**

$$h(x) = \varphi(W_1 x + b_1), \quad y = \sigma(W_2 h(x) + b_2)$$

**Note:** where ‘ϕ’ denotes a non-linear activation function (e.g., ReLU, tanh), and W1,W2 are weight matrices.

**How It Learns Complex Relationships**

MLP overcomes the limitations of logistic regression through:

a. **Non-linear Transformations**: Activation functions in hidden layers enable the network to model non-linear relationships in the data.
b. **Hidden Layers**: These layers perform hierarchical feature extraction, allowing the model to learn more abstract and intricate patterns.

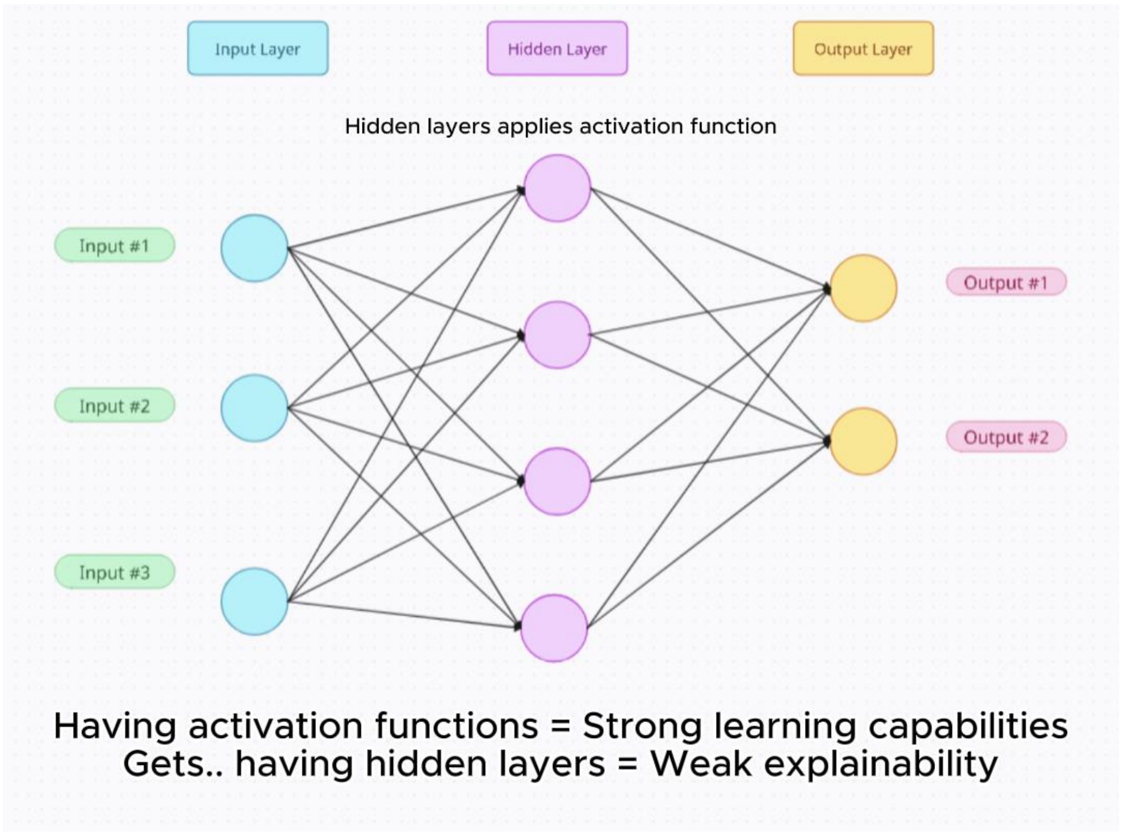

Input Layer
Hidden Layer
Output Layer
Hidden layers applies activation function
Input #1
Input #2
Input #3
Output #1
Output #2
Having activation functions = Strong learning capabilities
Gets.. having hidden layers = Weak explainability

*Figure 2: Example MLP Architecture*

***At this point, I would like to draw your attention to something: In logistic regression, the absence of activation functions prevents strong learning, and the reason it is considered interpretable is the lack of hidden layers. On the other hand, the strong learning capabilities of MLP (Multilayer Perceptron) are due to the presence of activation functions, while the reason it is considered non-interpretable is the presence of hidden layers. In other words, what we truly need are activation functions, not hidden layers. However, it is widely accepted that hidden layers are necessary for the application of activation functions. By stepping outside this conventional understanding, I have built a framework that demonstrates a learning architecture can possess strong learning capabilities while remaining interpretable, even without hidden layers and by applying activation functions directly.***

## 2.2. – What is Potentiation Learning Artificial Neural Network ?

### 2.2.1. – Potentiation Learning Artificial Neural Network (PLAN)

**Definition**

PLAN is a supervised classification algorithm aimed at possessing both the interpretability of logistic regression and the ability to learn complex relationships akin to MLP.

PLAN (Potentiation Learning Artificial Neural Network) is also a continuation of my previously published work, PLAN (Pruning Learning Artificial Neural Network)[6], and includes significant modifications in terms of content. It has undergone so many changes that I had to rename it.

Unlike conventional methods, this algorithm performs the 'learning' process in a fundamentally different manner. In a classical MLP architecture, a technique called 'backpropagation' initializes weights randomly, and then the error function is minimized through 'iterative gradients'. However, in PLAN, weights are not initialized randomly. Instead, the weights corresponding to each class are formed through the accumulation of input samples representing that class. This approach is more aligned with the concept of 'Long Term Potentiation' from neuroscience, which is based on the principle that connections between recurring elements are strengthened. As a result, a single input sample, processed through various activations, is then aggregated to form a highly complex, nonlinear new input. Since this new input retains the same dimension as the original, we can still identify which input element represents which feature of the data. Furthermore, since this newly transformed input layer is directly added to the weights of the corresponding class (output), nonlinear learning is achieved. And because the input dimension remains the same, we can easily interpret and analyze relationships between input and output by observing the connections, just as in logistic regression.

In this architecture, the choice of activation functions and the number of transformations is crucial. Therefore, the activation functions and their combinations are determined through a learning process that experiments with different options based on a specific metric (not necessarily the loss function, but it could even be accuracy). The goal is to iteratively explore and identify the most suitable combination for the given problem. I called this combination **'activation potentiation'**.

**Mathematical Representation:**

$$\boldsymbol{Activation(x_{train}) \leftarrow x_{train}}$$

$$\boldsymbol{W += y_{train}^{T} \cdot X_{train}}$$

$$\boldsymbol{W' = \frac{W}{\max(|W|)}}$$

**Note:** $\boldsymbol{y_{train}}$ is one hot encoded.

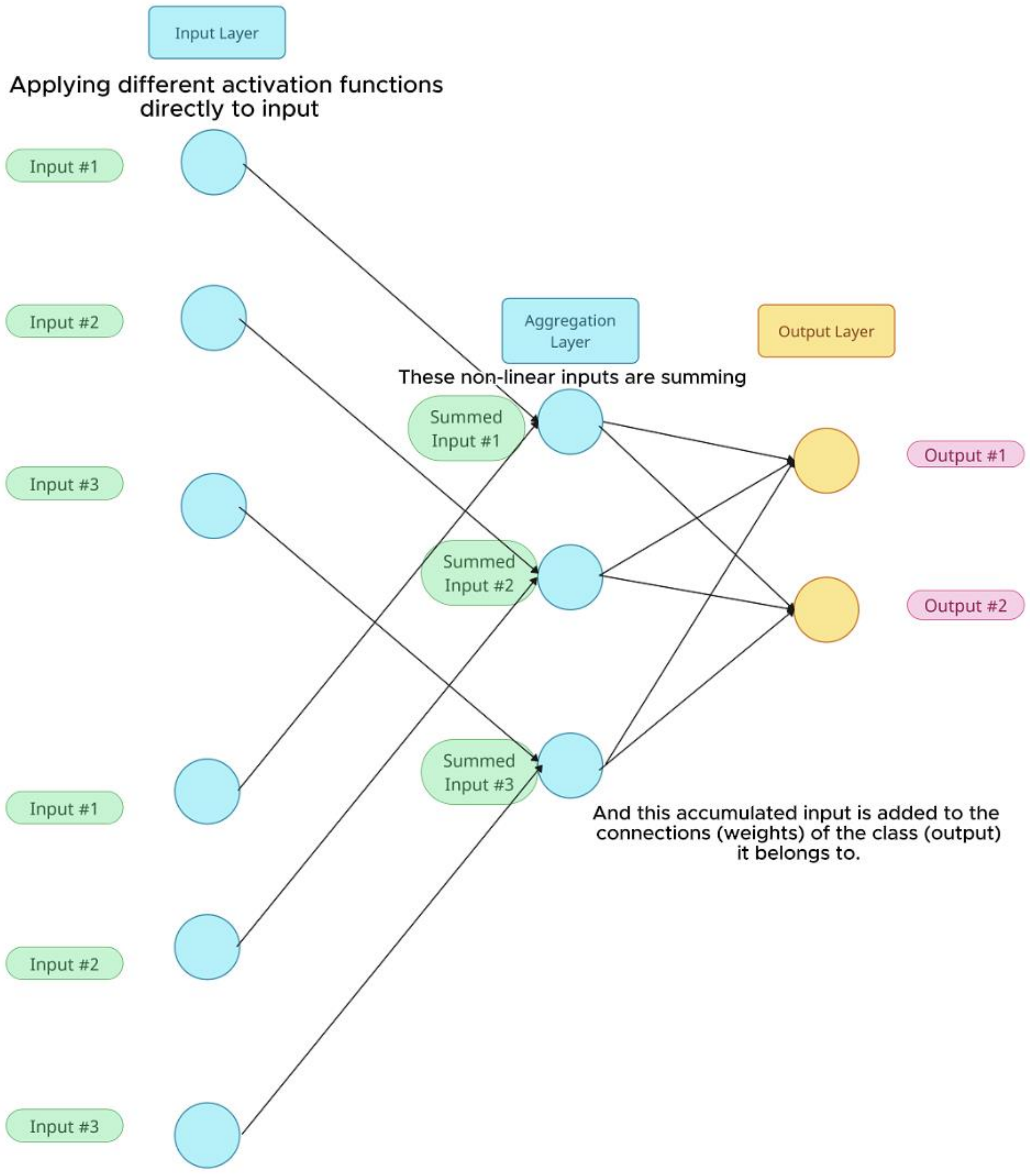

Input Layer
Applying different activation functions directly to input
Input #1
Input #2
Input #3
Aggregation Layer
Output Layer
These non-linear inputs are summing
Summed Input #1
Summed Input #2
Summed Input #3
Output #1
Output #2
And this accumulated input is added to the connections (weights) of the class (output) it belongs to.
Input #1
Input #2
Input #3

*Figure 3: Example PLAN Architecture*

### 2.2.2. – Optimization of PLAN

We are not trying to explore all these activation combinations one by one. The backpropagation optimization algorithm cannot be used in PLAN because, in PLAN, the weights are determined by a fixed rule in which the activations of inputs are accumulated according to a specific method, where each input’s corresponding class connections are updated. In PLAN, the learnable parameters are not the weights but rather the combinations of activation functions. This is a discrete optimization problem. Therefore, I have modified the optimization approach by using a metaheuristic genetic algorithm, NEAT[4], to evolve and optimize the activation combinations of PLAN. I have named this approach ENE(Eugenic NeuroEvolution).

The ENE discussed here is a completely new and original approach, entirely different from previously mentioned ENE in the literature. It aims to redefine ENE from the ground up.[5]

First, we train each activation individually using PLAN's cumulative learning method, and in the end, we have as many models as the number of activations in our pool. Currently, for each activation, we have corresponding A and W. We then make predictions on the training data with these models, obtaining the accuracy values for each model. At this point, we have A, W, and accuracy values for each model. Each A and W is treated as a genome, and the entire population is passed to the ENE algorithm for evolution. The genomes are divided into two groups based on their fitness performance: good and bad genomes. Based on certain mutation and crossover parameters, new activation combinations are selected through a selection process, where the best genome combines with other good genomes, replacing the bad genomes in the next generation. During this process, small mutations also occur, and weights can be involved in the crossover, though this is optional. This is because, in PLAN, the weights are already learnable based on activation combinations, but weight evolution can sometimes speed up the process. After all these steps, the new population makes predictions again, recording new accuracy values, and this process is repeated until the maximum number of generations is reached. The key point here is that, unlike the standard NEAT algorithm, the population size is determined based on the total number of activations in the pool rather than being manually set. This is because, unlike traditional NEAT, ENE does not initialize weight parameters randomly. The initial population formed using PLAN’s cumulative weight accumulation technique provides a strong starting point for ENE, helping us overcome the performance and time limitations of classical NEAT and enabling a strong population to kick-start the process.

## 2.2.3. – Fitness of ENE

ENE optimization evolves these activation combinations by maximizing a fitness function that I refer to as **WALS (Weighted Accuracy-Loss Score).** This new function calculates loss using categorical cross-entropy. To ensure numerical stability, a small positive value is added to both the loss and its impact. The final fitness score is then computed by multiplying accuracy with its impact and adding a term that accounts for the relationship between loss impact and loss. This formulation allows for a dynamic optimization process, balancing both accuracy and loss in an adaptive manner. Beyond the impact values, this fitness function has a unique characteristic. The **accuracy impact** and **loss impact** values are provided externally, allowing the model trainer to adjust their relative influence. When training a classification model, ensuring correct classification is the highest priority. Therefore, in the early generations, if the accuracy impact is nonzero, the function primarily focuses on maximizing accuracy. This prioritization of accuracy ensures that the model learns to classify correctly from the start. However, as the loss value becomes excessively small, the function triggers adaptive responses, aggressively minimizing loss and reinforcing further minimization. In such cases, the function adjusts its focus to further optimize the loss by giving it more weight. The accuracy and loss impact values serve to adjust the relative emphasis on each component. When **accuracy impact** is nonzero, maximizing accuracy takes priority, while the **loss impact** is given more focus once the loss is sufficiently minimized. This flexible adjustment helps guide the evolutionary process, ensuring both the correctness and efficiency of the model. In an evolutionary algorithm, this fitness function is maximized to drive optimal model performance.

**Fitness Calculation:**

$$\boldsymbol{Loss = -\sum_{i=1}^{N} y_{train}\,\mathbf{lo\,g}\left(y_{pred}\right)}$$

$$\boldsymbol{Accuracy = \frac{\sum_{i=1}^{N} \mathbf{1}\left(y_{pred} = y_{train}\right)}{N}}$$

$$\boldsymbol{Fitness = \left(Accuracy \cdot accuracy_{impact}\right) + \left(\frac{loss_{impact}}{Loss} \cdot loss_{impact}\right)}$$

## 2.2.4. – Evolving in ENE

### 2.2.4.1. – Selection

First, we order individuals based on their performance, then we categorize them into two groups: "good" and "bad."

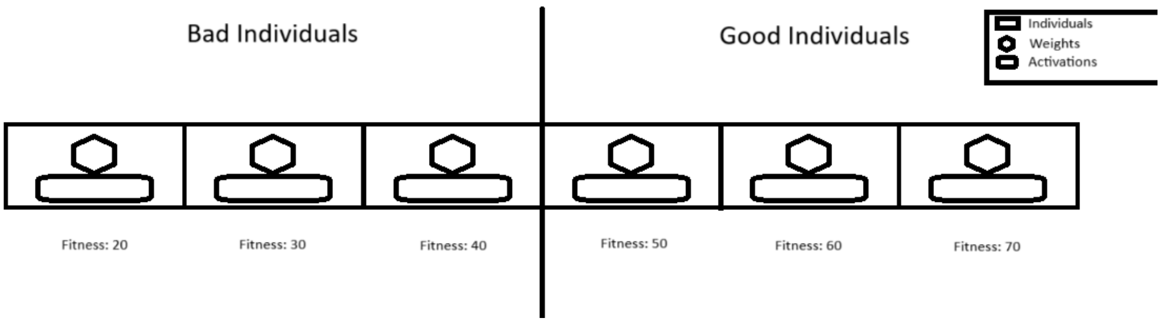


***Figure 4: Population before evolution***

### 2.2.4.2. – Policy & Strategy in selection

The **policy** parameter determines whether the evolutionary process prioritizes maintaining diversity or focuses on continuously optimizing genes based on fitness values. It can take two different values: "explorer" and "aggressive".

When set to "explorer", one of the two individuals selected for crossover is always chosen randomly from the high-performing group based on a predefined probability.

If the "aggressive" parameter is selected, one of the two individuals chosen for crossover is always the one with the highest fitness value.

The **strategy** parameter is a higher-level hyperparameter that predefines the probability settings for selecting the second individual in the crossover process—whether it is chosen from the high-performing or low-performing group. Additionally, it determines the predefined selection probabilities for mutations involving good and bad individuals.

This parameter can take three different values: "normal selective", "less selective", and "more selective".In the "normal selective" strategy, low-performing individuals have a 25% chance of being included in the crossover process. In "less selective", this probability increases to 50%, while in "more selective", it decreases to 10%.

The probability of undergoing mutation is 70% in "normal selective", 60% in "less selective", and 85% in "more selective".

### 2.2.4.3. – Preparing fitness values for Eugenic Operators

Before applying the eugenic operators, we normalize the fitness values of the individuals using the following mathematical operation, ensuring that the values remain between 0 and 1 and do not drop below zero:

$$\boldsymbol{Max_{abs}} = \mathbf{max}(|\boldsymbol{FitnessList}|)$$

$$\boldsymbol{NormalizedFitnessList} = \frac{\boldsymbol{FitnessList} + \boldsymbol{Max_{abs}}}{\mathbf{2} \cdot \boldsymbol{Max_{abs}}}$$

After this operation, a loop is initiated for **half of the total population size**. For example, if the total population size is **6**, then the eugenic operations loop runs for **3 iterations (6/2 = 3)**.

### 2.2.4.4 – Eugenic Crossover

ENE uses a new type of crossover called **"Eugenic Crossover"**. The term "eugenic" is used because, unlike the random selection method in classical crossover, the genes (weight values and activation functions) are combined through a mathematical process based on the fitness values of the individuals. This approach determines which specific genes are passed to the offspring. First, the dominant individual is selected (the first individual is usually good in the selection process, and the second may be bad), based on a predefined probability value. Then, the genes to be taken from the non-dominant individual are determined according to the following mathematical process:

$$\boldsymbol{r_{start}} = \lfloor \boldsymbol{r_{start}} \cdot \boldsymbol{f_{undom}} \rfloor$$

$$\boldsymbol{r_{end}} = \lceil \boldsymbol{r_{end}} \cdot \boldsymbol{f_{undom}} \rceil$$

$$\boldsymbol{c_{start}} = \lfloor \boldsymbol{c_{start}} \cdot \boldsymbol{f_{undom}} \rfloor$$

$$\boldsymbol{c_{end}} = \lceil \boldsymbol{c_{end}} \cdot \boldsymbol{f_{undom}} \rceil$$

$$\boldsymbol{W_{child}} = \boldsymbol{W_{dom}}$$

$$\boldsymbol{W_{child}}[\boldsymbol{r_{start}}:\boldsymbol{r_{end}}, \boldsymbol{c_{start}}:\boldsymbol{c_{end}}] = \boldsymbol{W_{undom}}[\boldsymbol{r_{start}}:\boldsymbol{r_{end}}, \boldsymbol{c_{start}}:\boldsymbol{c_{end}}]$$

**Note:** $\boldsymbol{f_{undom}}$ is normalized fitness value of undominant individual.

To determine the start and end values for the row and column indices in the Two-Point Matrix Crossover process, we multiply them by the score of the undominant individual in the previously normalized fitness list (ranging from 0 to 1). This score represents how the undominant individual ranks within the overall population.

- If the value is 1 (i.e., the best individual), no eugenic modification is applied, as multiplying by 1 keeps the selected section unchanged.
- If the value is less than 1, the selected portion is scaled down accordingly, reducing the genetic contribution of the undominant individual to the offspring.

**Note:** *Before this operation, the segment between the start and end points must satisfy the condition that its size remains smaller than half of the total elements in the weight matrix (rows × columns / 2). This ensures that even if the undominant individual has the highest fitness value, it can transfer at most half of its genes to the offspring.*

**Note:** *Whether individuals undergo eugenic crossover or classical crossover can be determined using a probability parameter called "**fitness bias**".*

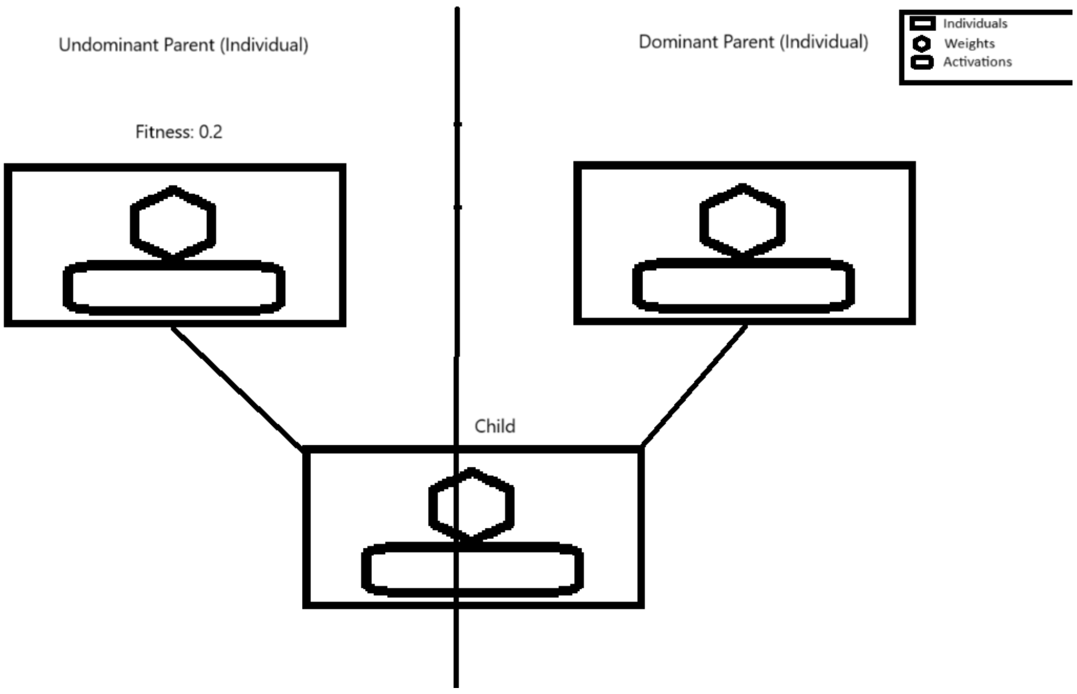


***Figure 5: The formation of a new individual when the undominant fitness value is 0.2***

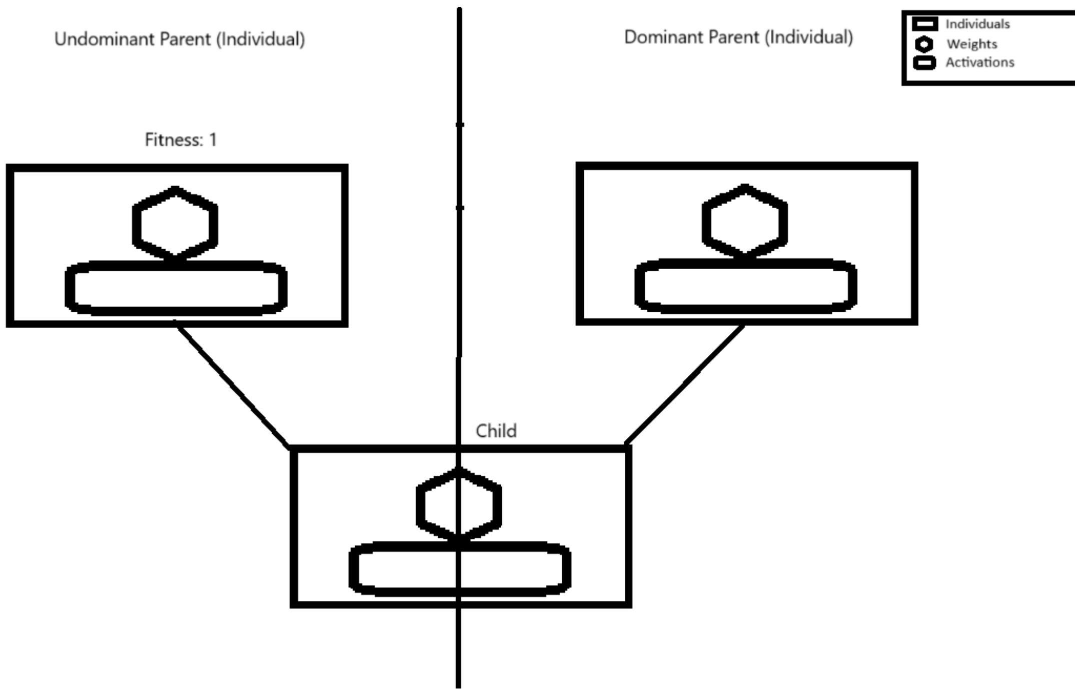


*Figure 6: The formation of a new individual when the undominant fitness value is 1*

### 2.2.4.5 – Eugenic Mutation

The eugenic mutation process is similar to eugenic crossover, but with some key differences. It includes an additional mathematical operation that determines how many elements in the weight matrix should be randomly modified, based on the matrix size.

This process incorporates an external parameter called "weight mutation threshold", which enhances the algorithm’s adaptive nature by ensuring consistent mutation behavior across different weight matrix sizes. This parameter governs the maximum number of mutations allowed per a predefined number of elements in the weight matrix.

Mechanism:

- For example, if the total number of elements in the weight matrix is 32 and the weight mutation threshold is 16, then at most 2 elements (32 / 16 = 2) can undergo mutation.
- However, this threshold can increase but never decrease after an eugenic operation.
- Although the maximum value is set externally, it can decrease dynamically based on a mathematical function. If an individual has a low fitness score relative to the population, the threshold is proportionally reduced.

Evolutionary Impact:

- This mechanism accelerates the elimination of weak genes, as individuals with poor fitness undergo higher mutation rates.
- At the same time, individuals with high fitness scores experience fewer mutations, reducing unnecessary genetic modifications.

- As a result, the population reaches higher fitness levels much faster, while also minimizing computational cost by applying fewer mutations to already well-adapted individuals. I achieved the best results with a **weight mutation threshold** value of **16**.

$$r_{end}, c_{end} = \text{shape}(W)$$

$$Threshold_{max} = r_{end} \cdot c_{end}$$

$$NewThreshold = \frac{Threshold_{mut}}{f_{individual} + \varepsilon}$$

$$n_{mut} = \min(\lfloor NewThreshold \rfloor, Threshold_{max})$$

$$r_{idx} \sim \text{Uniform}(0, r_{end}, n_{mut})$$

$$c_{idx} \sim \text{Uniform}(0, c_{end}, n_{mut})$$

$$V_{new} \sim \text{Uniform}(-1, 1, n_{mut})$$

$$W[r_{idx}, c_{idx}] = V_{new}$$

**Note:** $f_{individual}$ is normalized fitness value of individual.

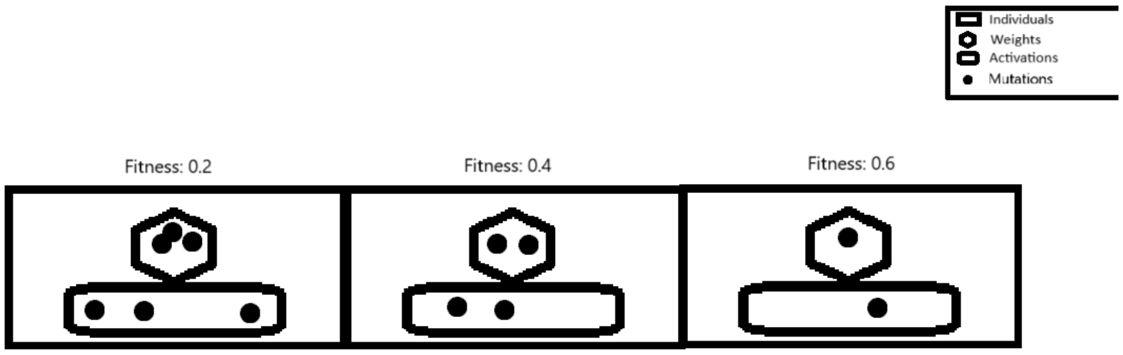


***Figure 7: Different mutation counts for individuals with varying fitness values.***

### 2.2.4.6 – End of the Evolution

As previously mentioned, these crossover and mutation operations are performed every 3 iterations, with individuals selected based on predefined probability parameters. Mutation is generally applied to low-fitness individuals, while crossover is primarily applied to high-fitness individuals.

At the end of this process, we obtain half of the population consisting of newly generated offspring (children) from crossover and the other half consisting of mutated individuals. These two groups are then merged to fully replace the previous generation. Optionally, the best individual from the previous generation can be directly transferred to the next generation; however, this is not mandatory. Unlike conventional genetic algorithms, ENE exhibits significantly weaker dependence between successive generations. One of the key characteristics of this algorithm is that while one half of the population (offspring) rapidly converges toward the optimal solution by becoming increasingly refined, the other half undergoes intelligently designed, fitness-based mutations to maintain diversity and prevent stagnation in the search space.

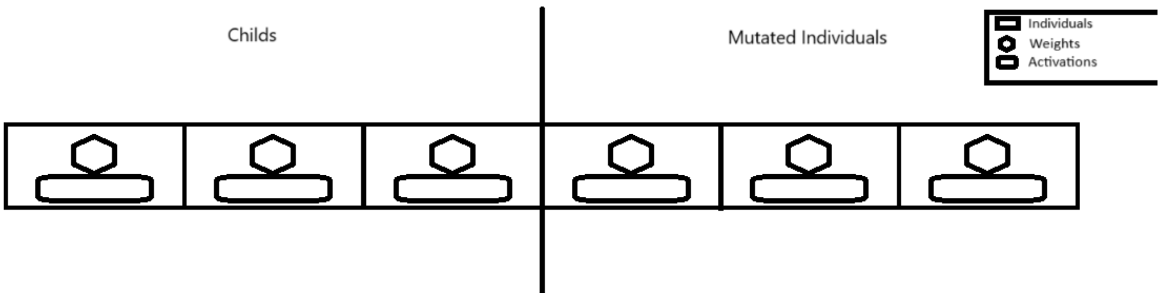


***Figure 8: Population after evolution***

## 2.3. – PTNN (Potentiation Transfer Neural Network)

The PTNN approach is a hybrid artificial neural network learning methodology that combines the advantages of the PLAN architecture with the strengths of the MLP architecture. The learning dynamics of PLAN models are based on a simple yet effective technique of adding inputs to weights, where the input is entirely dependent on the output in a layerless network. This allows for rapid convergence to very high accuracy at the beginning of training in labeled classification models with large inputs or outputs. Through the activation potentiation process, where data passes through certain activation combinations during training and merges, non-linear decision boundaries can be drawn, resulting in performance similar to that of MLPs. However, when the input dimension is very large, activation potentiation begins to pose a problem, as each activation adds a +1 feature vector cost to the model. Here, the advantage of PLAN lies in the fact that, if the extra processing cost is tolerable, it offers a model that is both explainable and transparent, yet still provides performance close to that of MLPs or Deep Learning. However, it may not be as flexible as MLP or Deep Learning, meaning that while its learning capacity far exceeds that of a linear separator, it is not quite at the level of an MLP or Deep Learning architecture. On the other hand, if an MLP is used, the training time may be much longer compared to PLAN, since in MLP or Deep Learning, weights are initialized randomly, and the more parameters that need to be optimized, the longer the process takes. This is where the PTNN algorithm comes in. **The PTNN algorithm takes a pre-trained model using the PLAN architecture and extends it to MLP or Deep Learning architecture, continuing the learning process.** Both architectures are integrated. The benefit here is that after quickly grasping the core of the network problem, regardless of its complexity, using the fast learning strategy of PLAN's weight formation with inputs, the learned information can be transferred to a higher-capacity architecture (MLP, Deep Learning) by adding hidden layers to the network, allowing learning to continue there. **In the end, an MLP architecture will emerge, but this architecture will have PLAN's fast weight creation strategy instead of randomly initialized weights.** (If biases are present, they are also transferred accordingly)

This information transfer process is achieved using identity matrices, which are neutral elements in the multiplication operation in linear algebra.

$$\textbf{for } l = 1 \textbf{ to } L \textbf{ do } \mathbf{W}^{(l)}_{\mathbf{MLP}} \leftarrow I_{m\times n} \ (\textbf{where } m = rows\ of\ \mathbf{W}^{(l-1)}_{\mathbf{MLP}}, n = neurons\,)$$

$$\textbf{end for} \quad \mathbf{W}^{(0)}_{\mathbf{MLP}} \leftarrow \mathbf{W}_{\mathbf{PLAN}}$$

**Note:** $if\ l$ **is 1, m =** $columns\ of\ \mathbf{W}_{\mathbf{PLAN}}$

For example:

$$\begin{matrix} 1.2 \\ -0.5 \\ 0.3 \\ 0.8 \end{matrix}_{\text{InputVector.T}} * \begin{matrix} 0.55 & 0.42 & -0.87 \\ 0.21 & -0.11 & 0.67 \\ 0.98 & 0.96 & 0.87 \\ -0.21 & -0.45 & 0.77 \end{matrix}_{\text{PLAN W}} = \begin{matrix} 0.68 \\ 0.49 \\ -0.50 \end{matrix}_{\text{OutputVector}}$$

$$\begin{matrix} 1.2 \\ -0.5 \\ 0.3 \\ 0.8 \end{matrix}_{\text{InputVector.T}} * \begin{matrix} 0.55 & 0.42 & -0.87 \\ 0.21 & -0.11 & 0.67 \\ 0.98 & 0.96 & 0.87 \\ -0.21 & -0.45 & 0.77 \end{matrix}_{\text{PLAN W}} * \begin{matrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{matrix}_{MLP\ W(1)} * \begin{matrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{matrix}_{\text{MLP W(2)}} = \begin{matrix} 0.68 \\ 0.49 \\ -0.50 \end{matrix}_{\text{OutputVector}}$$

Initially, the trained PLAN model can use the ENE optimization algorithm, or the weights can be created with the aggregation strategy of PLAN directly using the inputs, without the need for activation potentiation learning. However, once the model is converted into an MLP, it becomes a PTNN model, and it is no longer dependent on ENE. It can also be optimized using popular algorithms such as Adam and RMSProp. This can further accelerate the optimization process and may even address the issue of local minima in gradient-based algorithms. Another consideration is that the activation functions in each added hidden layer can directly affect the output, and there may be slight variations between the output of the PLAN model and the newly created PTNN model. However, this does not result in significant information loss.

Still, for complete transfer, it might make sense to use a linear activation function, in which case ENE might be necessary again, as ENE also learns activation functions for MLP or Deep Learning models.

## 2.4. – Implementation methods of ENE

### ENE + PLAN:

ENE can be used in the learning processes of the PLAN models I designed for optimization.

### ENE + MLP:

ENE can also be used to optimize classical MLP architectures. I have achieved quite good results at RL environments.

## 2.5. – Implementation methods of PLAN

### PLAN + ENE (Without weight evolve):

Weights are learned through the cumulative accumulation method of the PLAN algorithm, while activation potentiation is learned with ENE.

### PLAN + ENE (With weight evolve):

Both activation potentiation and weight parameters are learned with ENE. But model structure same as PLAN.

### PLAN + ENE (With weight evolve & fit start):

Both activation potentiation and weight parameters are learned with ENE. But first generation individuals will created by PLAN’s cumulative learning, also model structure same as PLAN.

### PLAN + MLP:

PLAN + MLP = PTNN.

## 3. – Results & Comparisons

### 3.1. – ENE Results

**Note:** *During the tests, I used the name PLANEAT instead of ENE. At that time, PLANEAT was the preliminary name of the algorithm; therefore, the line labeled as PLANEAT in the graph actually represents ENE.*

*The points in the graphs represent a 'generation' for NEAT and ENE, while they represent an 'episode' for other algorithms. The reason for defining the X-axis in terms of seconds is due to the fact that a generation and an episode do not complete in the same amount of time. All the code for the results is available, and all the algorithms were tested consecutively in the same environment and on the same computer.*

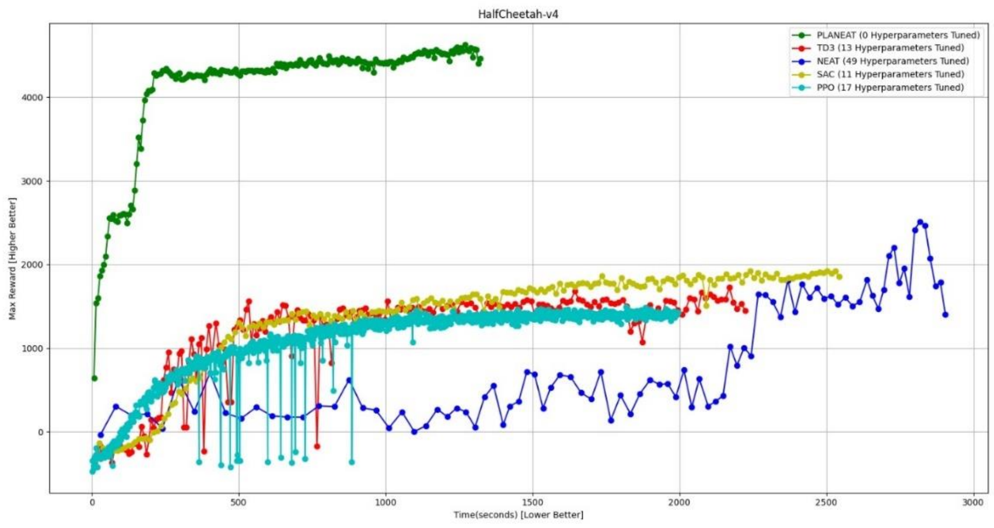


***Figure 9: HalfCheetah-v4***
*https://github.com/HCB06/PyerualJetwork/blob/main/Welcome_to_PyerualJetwork/ExampleCodes/ENE/Comparisons/half_cheetah(ENE%2Ctd3%2Cneat%2Csac%2Cppo).py*

ENE, here uses a very simple structure without hidden layers and can outperform other RL algorithms such as TD3 (Twin Delayed Deep Deterministic Policy Gradient), SAC (Soft Actor Critic), PPO (Proximal Policy Optimization), and NEAT (NeuroEvolution Of Augmenting Topologies) on the OpenAI GYM environment HalfCheetah-v4, without requiring any additional hyperparameter tuning.

*https://www.gymlibrary.dev/environments/mujoco/half_cheetah/*

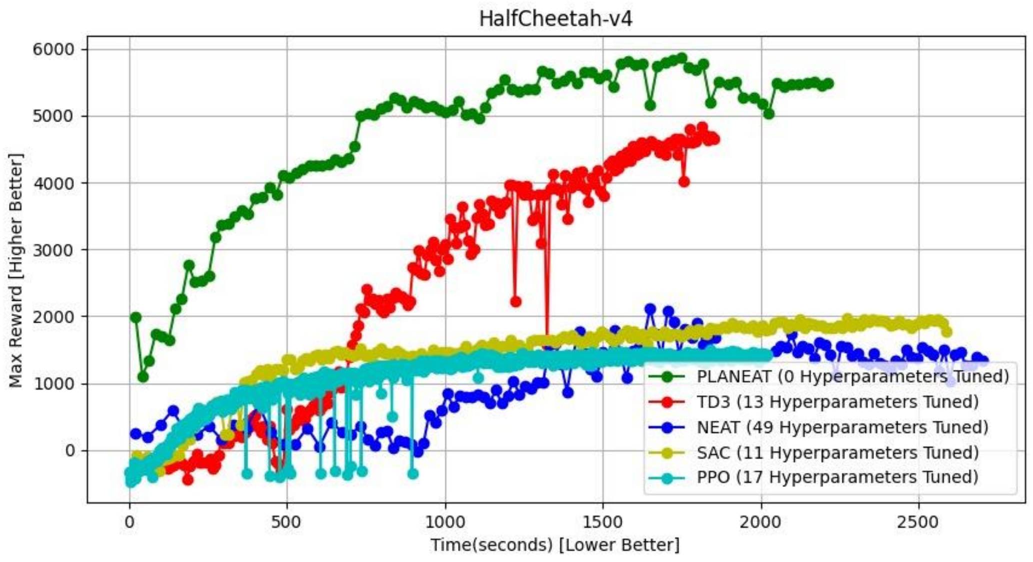


***Figure 10: HalfCheetah-v4(ENE + MLP)***

**Note:** *The NEAT algorithm was discarded from the code because it required an external configuration file, and since I conducted tests in other environments as well, conflicts were occurring.*

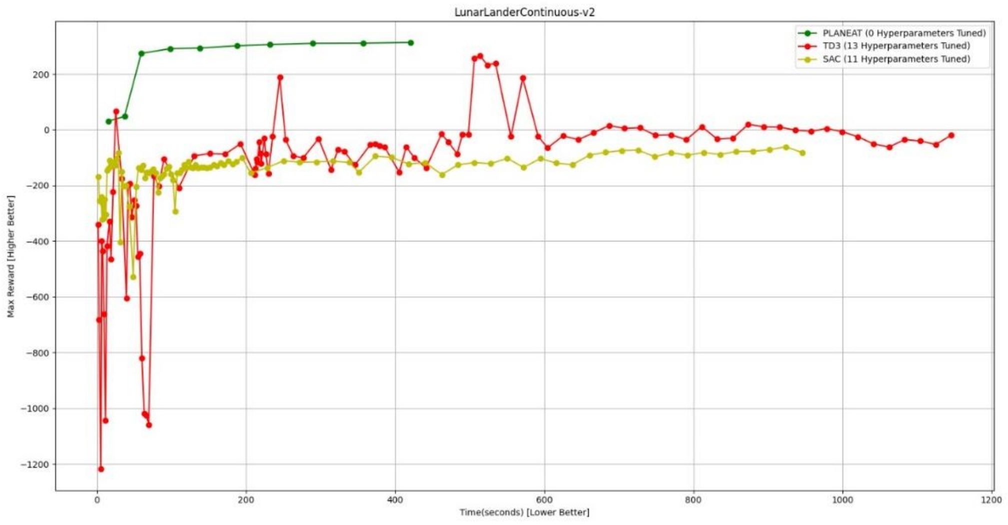
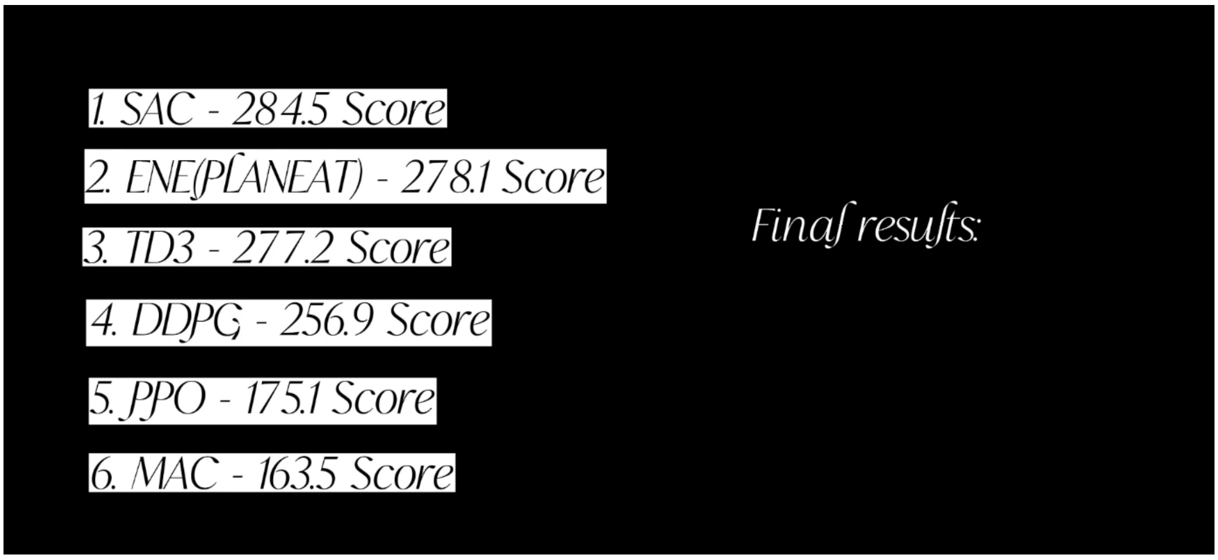


***Figure 11: LunarLanderContinuous-v2***

*https://github.com/HCB06/PyerualJetwork/blob/main/Welcome_to_PyerualJetwork/ExampleCodes/ENE/Comparisons/LunarLanderContinuous_v2(ENE%2Ctd3%2Csac).py*

In the LunarLanderContinuous-v2 environment(*https://paperswithcode.com/sota/continuous-control-on-lunar-lander-openai-gym*), ENE demonstrated exceptional efficiency, achieving a score of over 300 within minutes without any additional hyperparameter tuning. The network architecture consists of two hidden layers, each comprising 256 neurons with tanh activation functions. However, to ensure fairness, this configuration was also applied to ENE. Under standard conditions, the ENE algorithm in this environment was able to attain a score of 324 within seconds, using 300 individuals and no hidden layers. A fixed seed enables learning at a significantly faster rate compared to all other algorithms. When trained specifically to solve the LunarLanderContinuous-v2 environment, the results are as follows:



1. SAC - 284.5 Score
2. ENE(PLANEAT) - 278.1 Score
3. TD3 - 277.2 Score
4. DDPG - 256.9 Score
5. PPO - 175.1 Score
6. MAC - 163.5 Score

Final results:

***Figure 12:*** LunarLanderContinuous-v2 Test results from

[https://github.com/HCB06/PyerualJetwork/blob/main/Welcome_to_PyerualJetwork/ExampleCodes/ENE/LunarLanderContinuous_v2_expert_training.py](https://github.com/HCB06/PyerualJetwork/blob/main/Welcome_to_PyerualJetwork/ExampleCodes/ENE/LunarLanderContinuous_v2_expert_training.py)

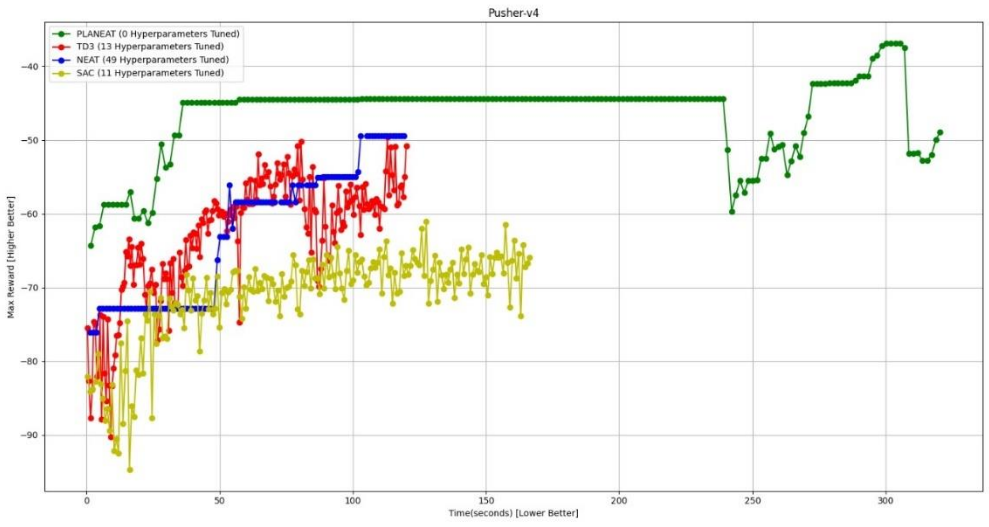


***Figure 13: Pusher-v4***

[https://github.com/HCB06/PyerualJetwork/blob/main/Welcome_to_PyerualJetwork/ExampleCodes/ENE/Comparisons/pusher_v4(ENE%2Ctd3%2Cneat%2Csac).py](https://github.com/HCB06/PyerualJetwork/blob/main/Welcome_to_PyerualJetwork/ExampleCodes/ENE/Comparisons/pusher_v4(ENE%2Ctd3%2Cneat%2Csac).py)

You can observe a similar table in the Pusher-v4 environment, which is also part of OpenAI GYM. *https://gymnasium.farama.org/environments/mujoco/pusher/*

Under equal conditions, the ENE algorithm outperforms almost all SOTA (State of the Art) algorithms in OpenAI GYM across various environments. I have achieved significantly better results than other algorithms in Humanoid-v4 and HumanoidStandup-v4 as well.

## 3.2. – PLAN Results

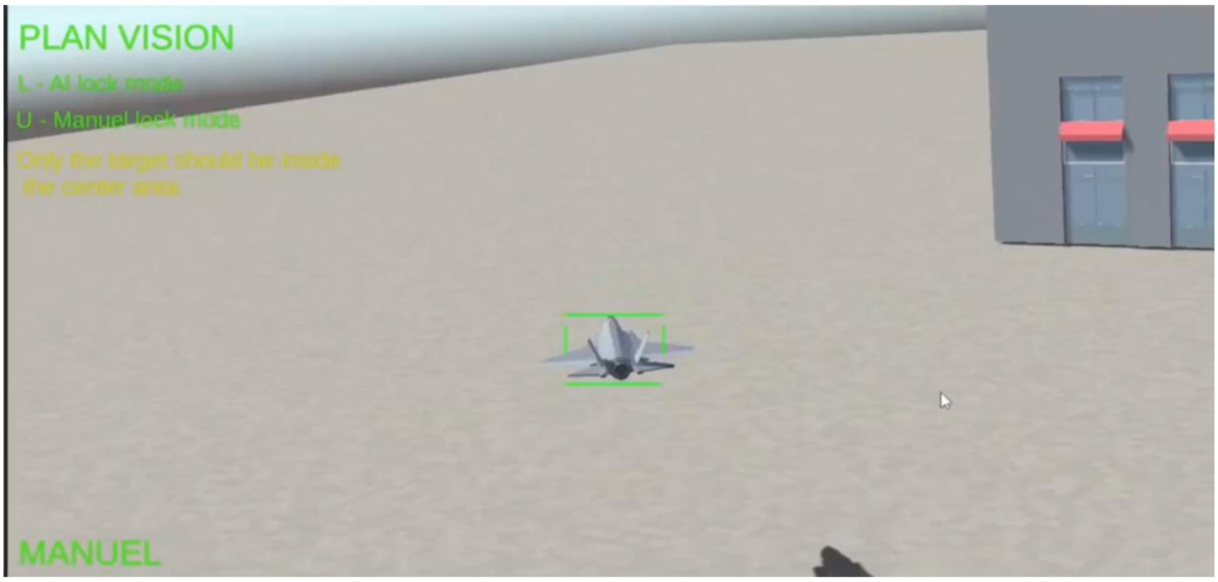


***Figure 14: Performance of PLAN on target tracking***

*https://github.com/HCB06/PyerualJetwork/blob/main/Welcome_to_PyerualJetwork/ExampleCodes/PLAN%20Vision/---%20SMART%20LOCKING%20SYSTEM%20---.txt*

In PLAN, neurons directly memorize what each class looks like. These memories enable the system to identify which class the input data belongs to.

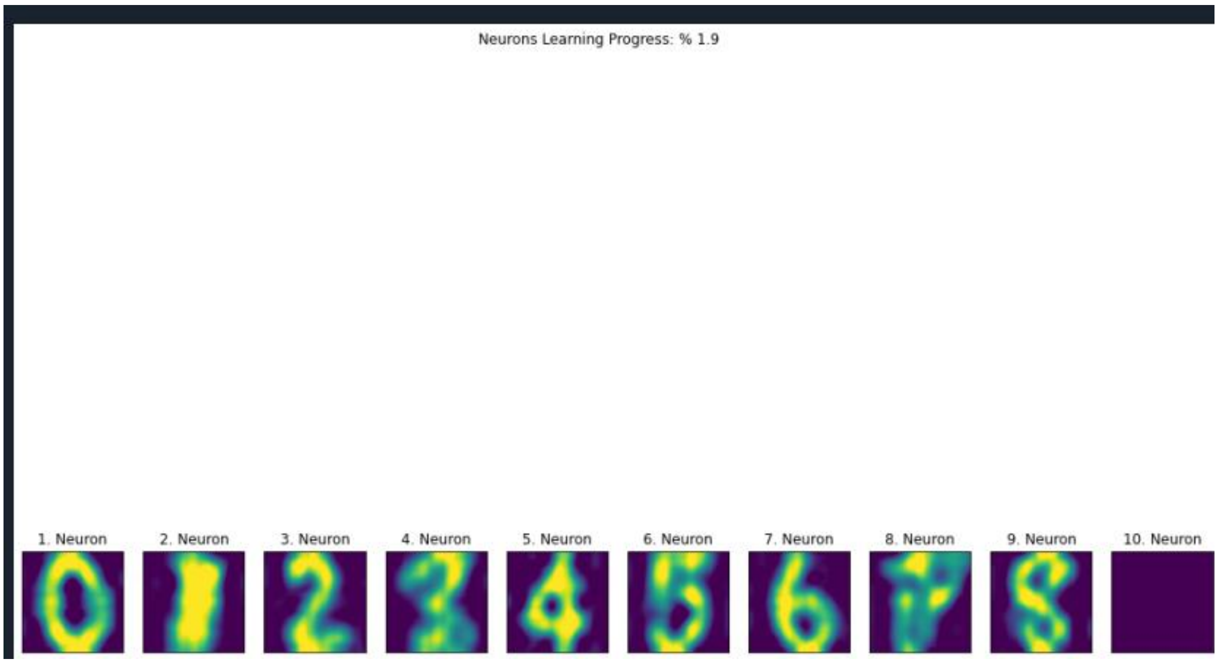


***Figure 15: Performance of PLAN***

**Output neurons (Learning at %1.9 with digits dataset)**

*https://github.com/HCB06/PyerualJetwork/blob/main/Welcome_to_PyerualJetwork/ExampleCodes/PLAN%20Vision/digits(plan).py*

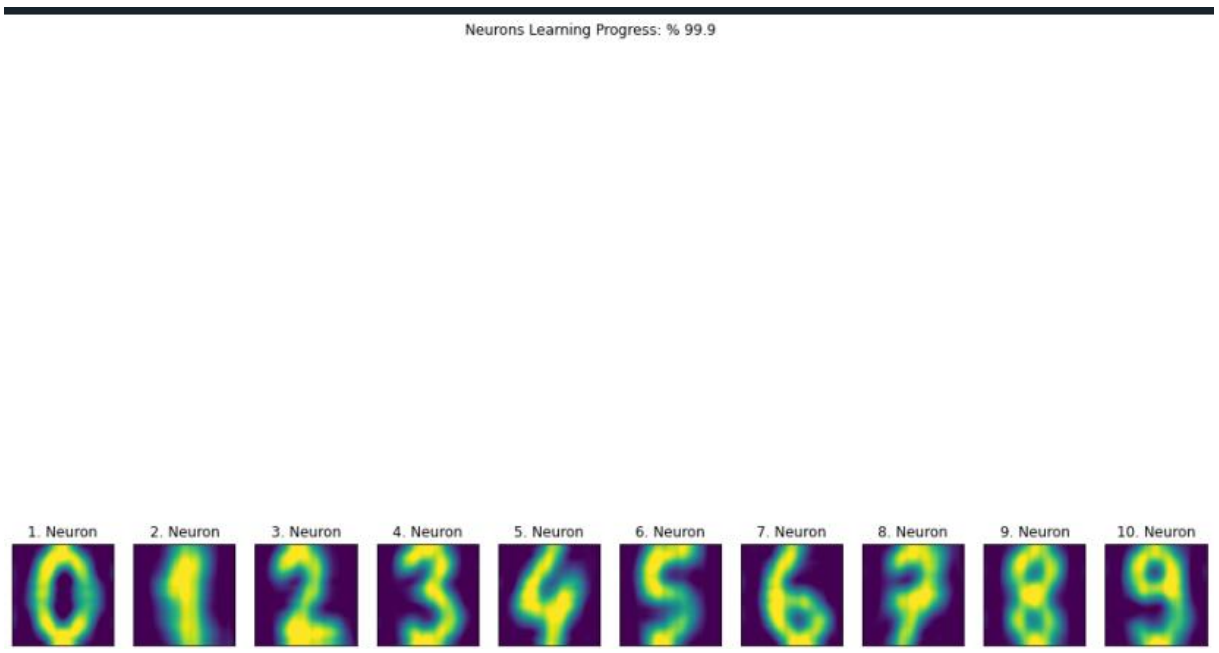


***Figure 16: Performance of PLAN***

**Output neurons (Learning at %99.9 with digits dataset)**

*[https://github.com/HCB06/PyerualJetwork/blob/main/Welcome_to_PyerualJetwork/ExampleCodes/PLAN%20Vision/digits(plan).py](https://github.com/HCB06/PyerualJetwork/blob/main/Welcome_to_PyerualJetwork/ExampleCodes/PLAN%20Vision/digits(plan).py)*

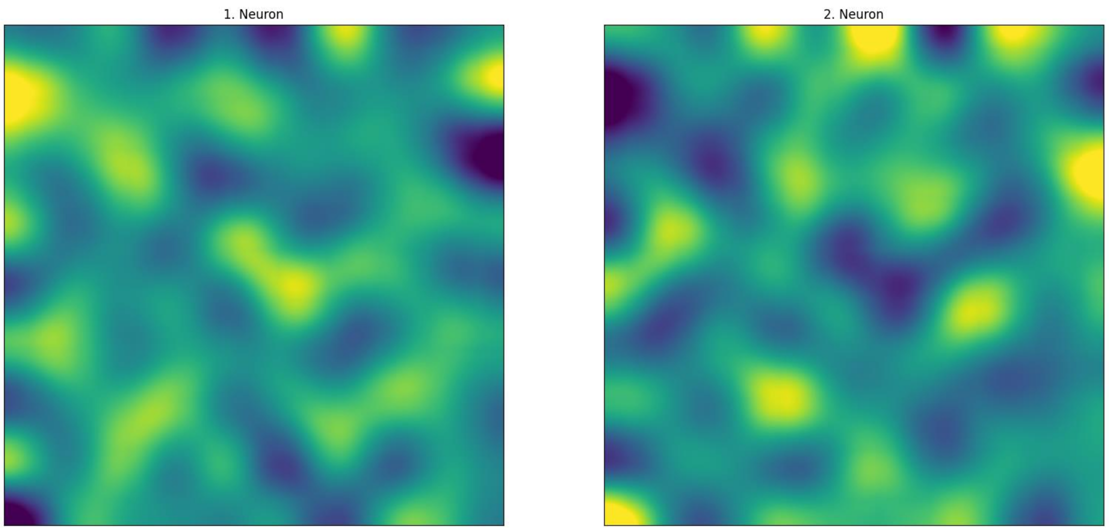


***Figure 17: Performance of PLAN***

**Output neurons (Learning at %100 with imdb 50k movie reviews dataset - selected feature count: 100)**

*[https://github.com/HCB06/PyerualJetwork/blob/main/Welcome_to_PyerualJetwork/ExampleCodes/NLPlan/imdb.py](https://github.com/HCB06/PyerualJetwork/blob/main/Welcome_to_PyerualJetwork/ExampleCodes/NLPlan/imdb.py)*

I conducted tests particularly on text, images, and other diverse data types, and achieving consistently above a certain standard in all of them is quite impressive. This underscores how widely applicable the Potentiation Learning Artificial Neural Network algorithm can be to a general audience.

Neurons can learn diffirences of classes:

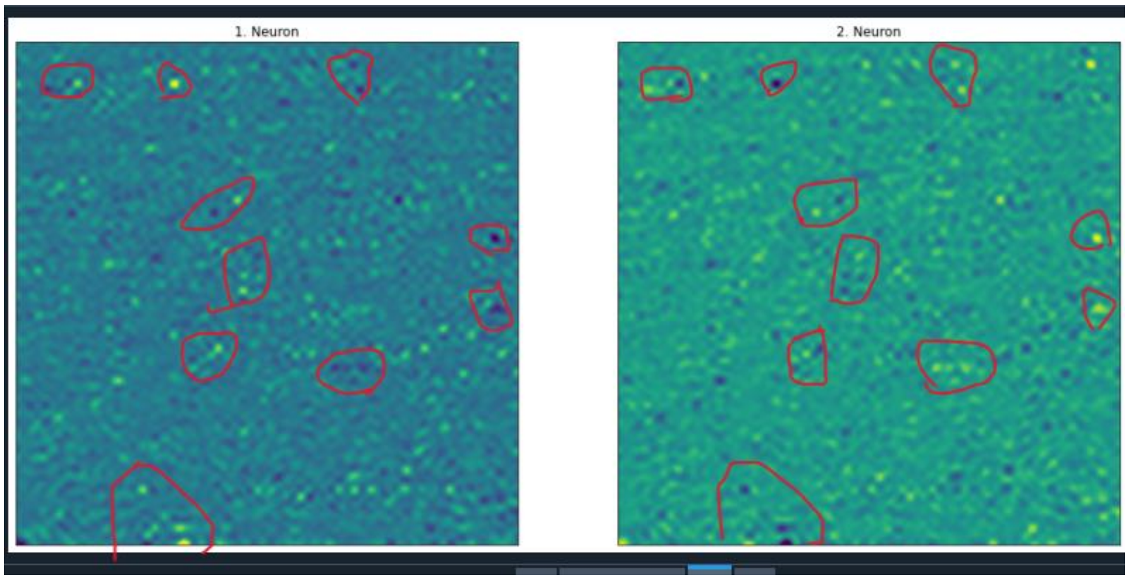


***Figure 18: Performance of PLAN***

**Output neurons (Learning at %100 with imdb 50k movie reviews dataset - selected feature count: 6084)**

[https://github.com/HCB06/PyerualJetwork/blob/main/Welcome_to_PyerualJetwork/ExampleCodes/NLPlan/imdb.py](https://github.com/HCB06/PyerualJetwork/blob/main/Welcome_to_PyerualJetwork/ExampleCodes/NLPlan/imdb.py)

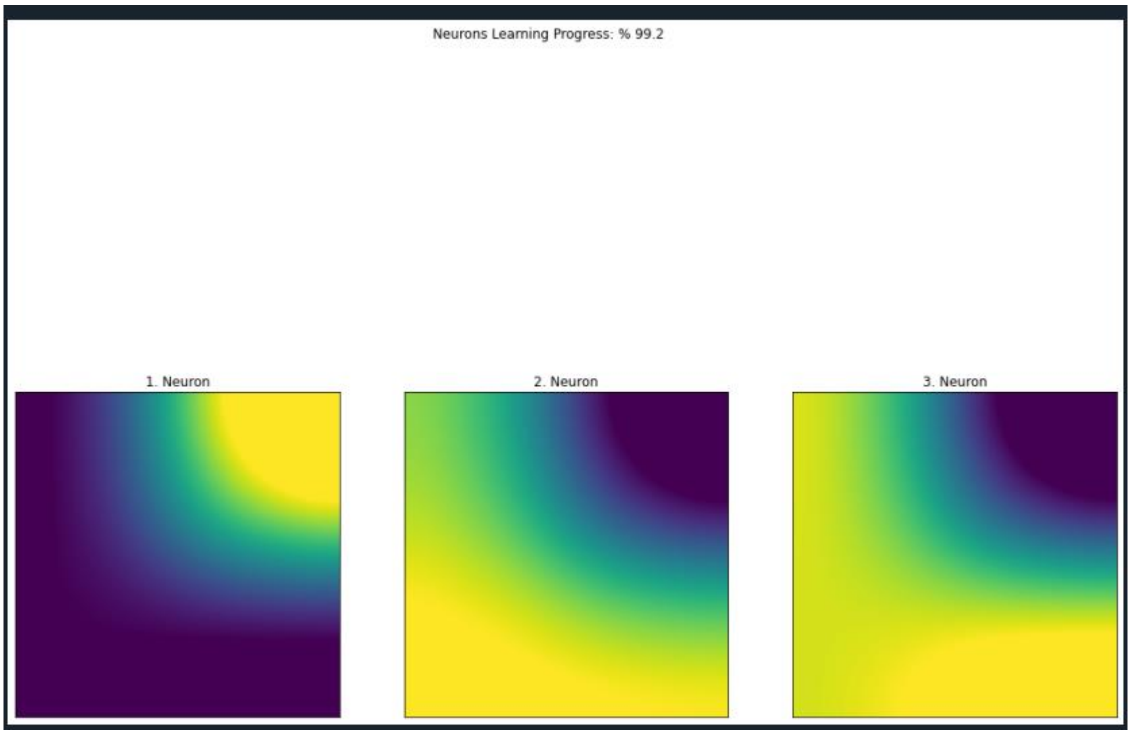


***Figure 19: Performance of PLAN***

***Output neurons (Learning at %99.2 with iris dataset)***
*https://github.com/HCB06/PyerualJetwork/blob/main/Welcome_to_PyerualJetwork/ExampleCodes/iris(plan).py*

Neurons can autonomously learn distinctive features by mimicking Long-Term Potentiation. This ability can be utilized to analyze differences between classes.

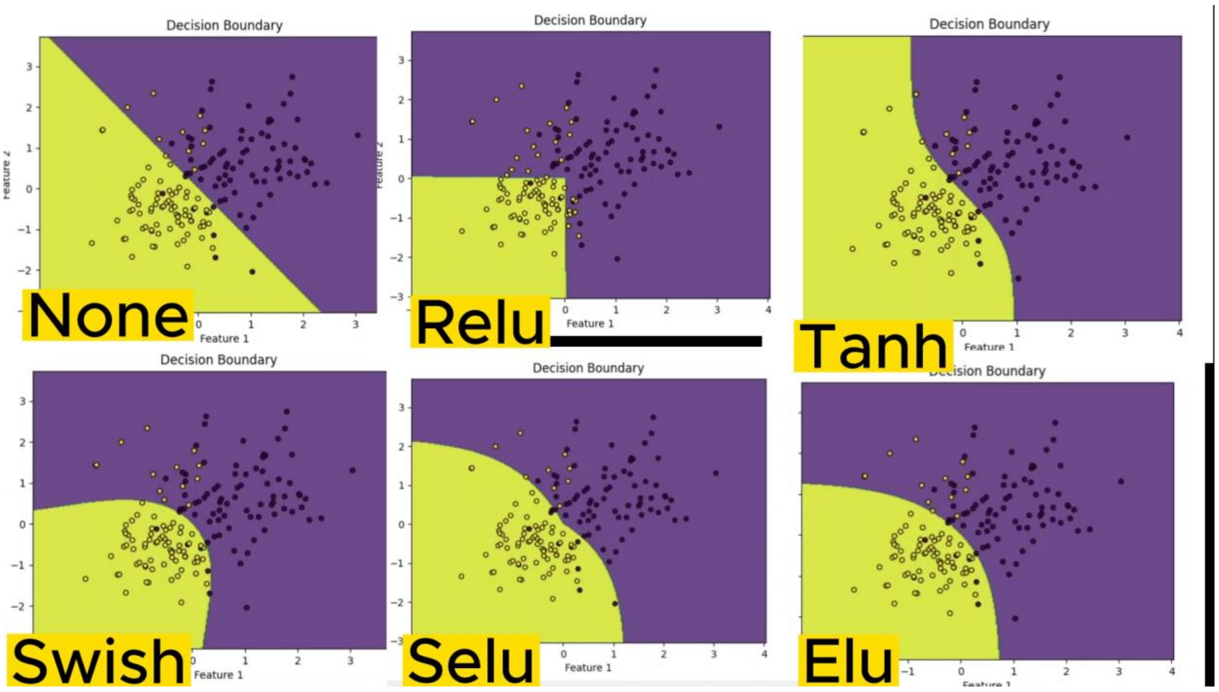


*Figure 20: Decision boundaries of the model for different activation functions in aggregation layer: (None = no activation function(linear))*

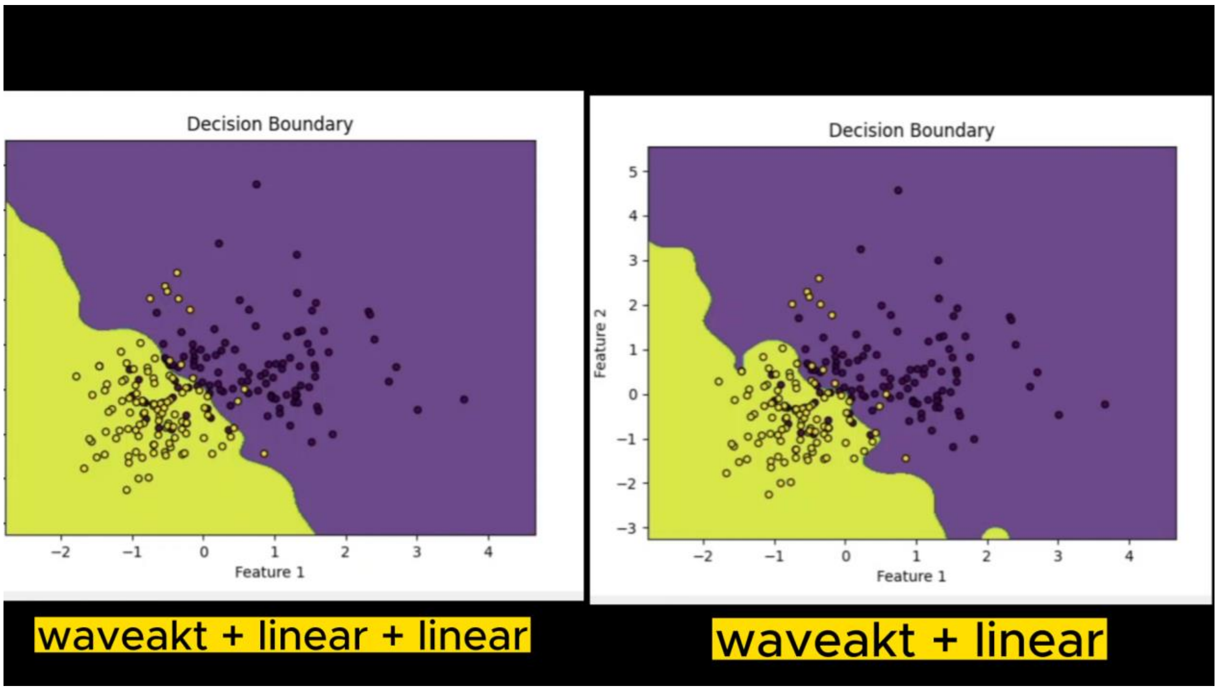


*Figure 21: Activation functions can be combined to mitigate each other's weaknesses, thereby adapting to the shape of the data and uncovering er features.*

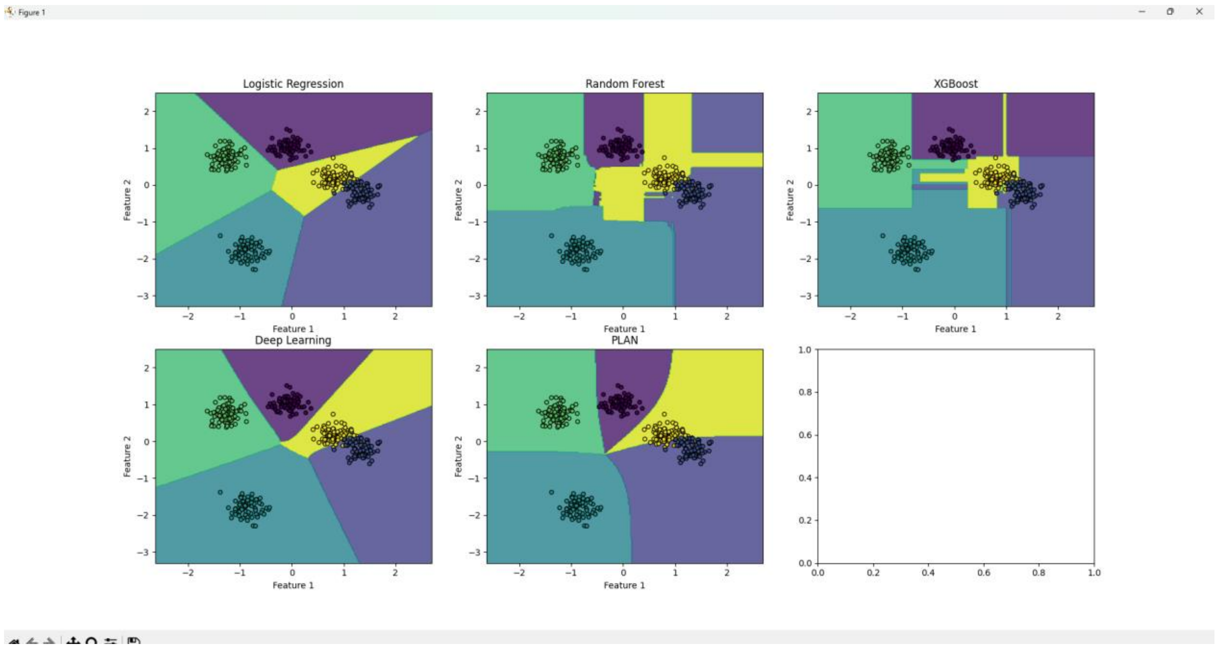


***Figure 22: (Scikit-learn's blobs dataset)***

[*https://github.com/HCB06/PyerualJetwork/blob/main/Welcome_to_PyerualJetwork/ExampleCodes/blobs(all_algorithms).py*](https://github.com/HCB06/PyerualJetwork/blob/main/Welcome_to_PyerualJetwork/ExampleCodes/blobs(all_algorithms).py)

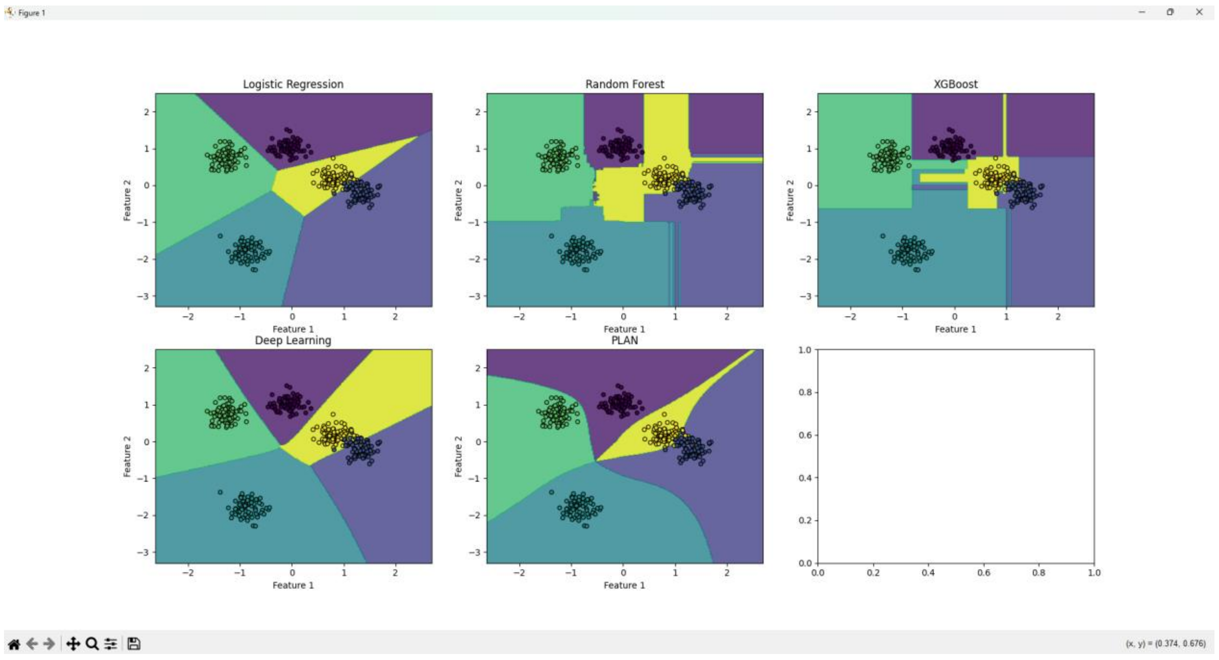


***Figure 23: (Scikit-learn's blobs dataset with different activation combinations)***

More models will be added to the PyerualJetwork Library GitHub page:
*https://github.com/HCB06/PyerualJetwork/tree/main*

The results of implementing the Potentiation Learning Artificial Neural Network architecture have been promising, though there are areas for improvement. While the current performance is not flawless, the Potentiation for future advancements is significant.

**Heart Diease Dataset**

In this comparison, we will examine the robustness of different algorithms using the Heart Disease dataset, which is derived from real-world data.

The first criterion is the ability to produce consistent results across at least five consecutive code compilations. The hyperparameters of the models have been adjusted accordingly to ensure stable outcomes.

The second criterion is consistency. For example, a model's success with a large dataset but failure with a smaller dataset is considered inconsistent.

Using the Heart Disease dataset, I trained nearly all classification algorithms (Logistic Regression, Random Forest, XGBoost, MLP, and PLAN) with the same training and test data in a single script. After a lengthy optimization process, the results are as follows:

- **Logistic Regression Test Accuracy: 0.8833**
- **Random Forest Test Accuracy: 0.9000**
- **XGBoost Test Accuracy: 0.8833**
- **MLP Test Accuracy: 0.8333-0.9300**
- **PLAN Test Accuracy: 0.8833-0.9500**

Code:

*https://github.com/HCB06/PyerualJetwork/blob/main/Welcome_to_PyerualJetwork/ExampleCodes/heart_disease(all_algorithms).py*

When reducing the training data and increasing the test data without altering any hyperparameters, the results are:

- **Logistic Regression Test Accuracy: 0.7353**
- **Random Forest Test Accuracy: 0.7757**
- **XGBoost Test Accuracy: 0.7647**
- **MLP Test Accuracy: 0.4596**
- **PLAN Test Accuracy: 0.8235**

The dramatic drop in MLP accuracy occurs with a moderately complex model, which meets the first criterion by consistently yielding a value of 0.4596. However, this result indicates inconsistency, which is undesirable.

To address this, I simplified the model to have only one hidden layer with 100 neurons. The results are now between 0.79 and 0.81, indicating a model training process based on exact matching. However, stability is lost in this case. With the full dataset, the accuracy is 0.88.

PLAN models are significantly more robust compared to MLP models.

Additionally, even when adding more layers to  Learning, the model does not robustly exceed the 0.90 accuracy threshold with the full dataset. No matter how many layers or neurons are added, the accuracy remains capped at 0.90.

When selecting a model architecture, considering that the model will continuously interact with users and that computational costs can increase exponentially, even someone with limited experience in machine learning can infer which architecture is more robust by examining the results. PLAN allows you to achieve the highest results with whatever data you have, without manual adjustments.

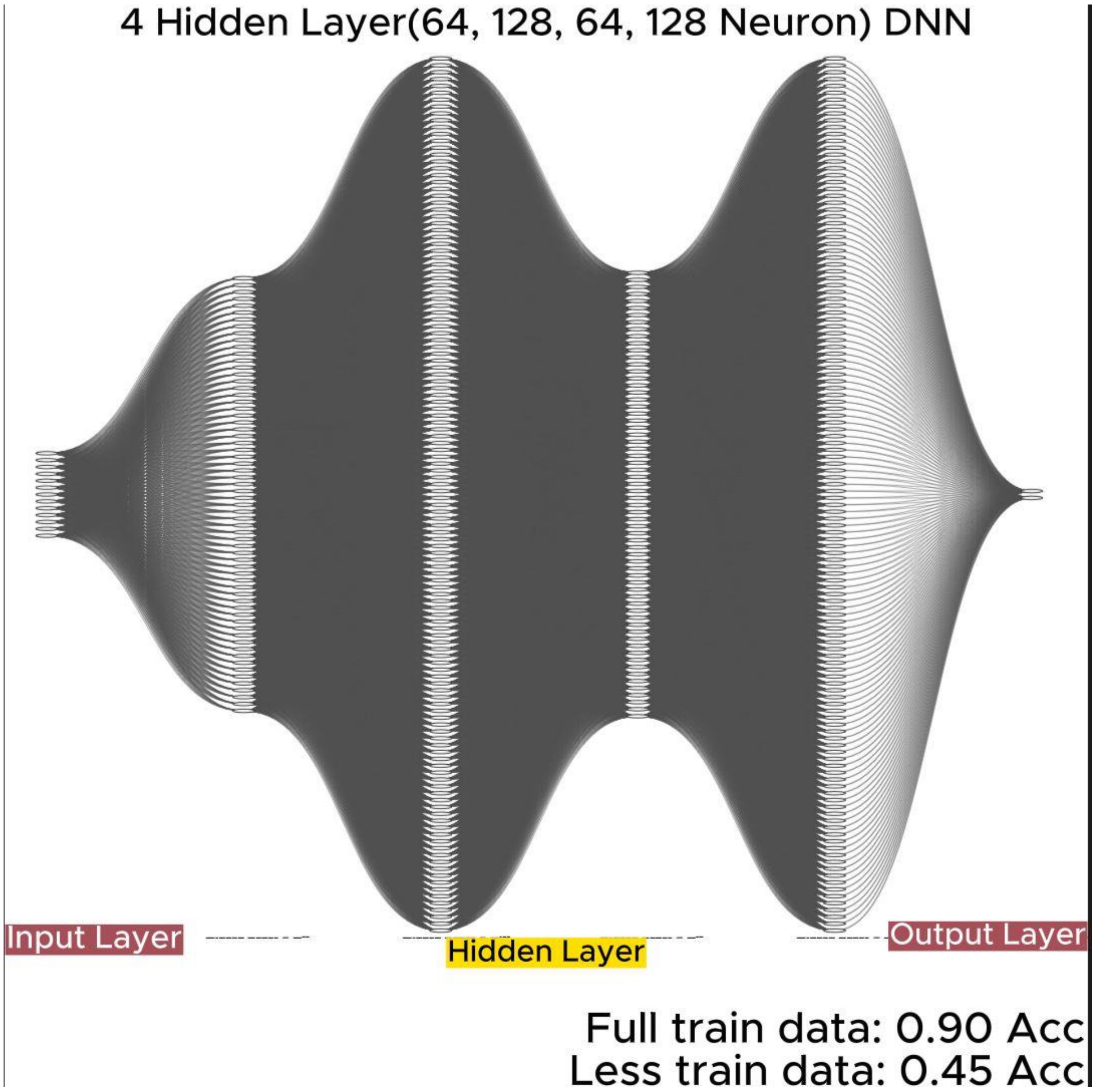

4 Hidden Layer(64, 128, 64, 128 Neuron) DNN
Input Layer
Hidden Layer
Output Layer
Full train data: 0.90 Acc
Less train data: 0.45 Acc

***Figure 24: (Deep Neural Network(DNN) in Heart Disease dataset)***

Furthermore, the PLAN architecture offers a significant advantage in model updating (adding new data). Instead of creating and training a new architecture with entirely new layers, its inherent robustness means this is not necessary.

For other comparisons:
*https://github.com/HCB06/PyerualJetwork/tree/main/Welcome_to_PyerualJetwork/ExampleCodes*

## 4. – Extra Comments

### *Multi-functional memories*

I previously mentioned that in PLAN, neurons directly memorize what each class looks like. This gives us not only an abstract representation, such as 'Class 0' and 'Class 1,' but also a model that provides what Class 0 and Class 1 actually look like. This is a feature not found in other classification algorithms. While we typically train these models solely for classification, the fact that they also reveal what each class looks like demonstrates their utility for **data analysis.** Furthermore, the information stored in the weight matrix can be retrieved later for other purposes, allowing the model to serve multiple functions. For example, for the class representing photos labeled '1,' if we input an image of a '1,' not only will it predict '1,' but it can also be used to complete missing parts, adding additional functionality to the model. This enables a 'kill two birds with one stone' approach by embedding multiple capabilities into a single model. **For instance, PLAN models trained for classification can easily be operated in reverse. Instead of predicting the output given the input, they can estimate the input given the output. This is achievable with a very simple modification to the standard operation.**

For class predicting: weight * input layer

For input prediction: output layer * weight

```
weight matrix: [[0.5,0.7,0.2]   *   output: [0.2,0.9]   =   input: [0.37,

                [0.3,0.6,0.8]]                                      0.68,

                                                                    0.76]

0.37 ≈ 0.3

0.68 ≈ 0.6

0.76 ≈ 0.8


This is achievable in PLAN because, in PLAN, each row in the weight matrix
represents a class.
```

This is because, in PLAN, learning is achieved through the cumulative combination of features, forming the connections. In other words, the numbers in a row of the weight matrix represent the features influencing the class associated with that row. This structure allows us to feed probabilities into the output layer and predict the input with a straightforward operation. This reverse operation can be performed effectively in tasks like image and text classification and can be adapted for more specific tabular data by adding or subtracting certain bias values to the predicted input as needed.

## *Greater parallelism compared to MLPs*

Compared to MLPs, greater parallelism can be achieved despite the fact that in state-of-the-art MLP designs, matrix-vector multiplications can be parallelized. This is due to the horizontal depth concept (depth with hidden layers), where the output of the previous layer must be awaited before the computations for the next layer can begin, making it impossible to initiate the computation of all layers simultaneously. However, in the PLAN algorithm, depth follows a vertical concept, meaning that the activation transformations applied to the input occur independently of each other. This allows all transformations to be initiated simultaneously, regardless of their number.

## *Evading local minimums (or maximums)*

Another advantage is its near impossibility of getting stuck in local minima (or perhaps maxima would be a more appropriate term, as PLAN does not rely on an error function—errors are not minimized; instead, fitness is maximized). Unlike backpropagation, PLAN is a more stochastic algorithm. Its stochastic nature stems from the fact that, unlike backpropagation, it does not follow a fixed gradient. This characteristic plays a pivotal role in discovering more creative solutions and potentially identifying different solution paths with each run.

## *Potentiation Learning Artificial Neural Network Library: PyerualJetwork*

To facilitate broader experimentation and practical application, I have encapsulated the Potentiation Learning Artificial Neural Network architecture in a Python library called "PyerualJetwork." This library is now available for both experimental and commercial use, allowing researchers and developers to easily integrate Potentiation Learning Artificial Neural Network into their projects. You can access the "plan" module within this library for various applications. **The PyerualJetwork library made an impressive debut, garnering over 100.000 downloads in just 7 months. This clearly highlights people's interest in the PLAN algorithm and ENE algorithm.**

For those interested in exploring or utilizing this framework, the library and algorithm are hosted on GitHub. You can find them at: *GitHub - PyerualJetwork*. Additionally, for practical demonstrations and tutorials on how to use the Potentiation Learning Artificial Neural Network architecture, please visit my YouTube channel: *YouTube - Hasan Can Beydili*.

## 5. – Advantages

### 5.1 – PLAN

1. Easy to learn and implement. (non-calculus)
2. Easy to maintain and update. (non-blackbox)
3. It occupies less memory space. (only can have one weight matrix)
4. Multi-functional single model architecture. (cumulative learning)
5. Strong generalization even small training splits.
6. It meets the potential to be as explainable as logistic regression and as powerful as multilayer artificial neural networks.
7. Greater parallelism compared to MLPs
8. Evades local minimum and maximum compared to MLPs Backpropagation

### 5.2 – ENE

1. Superior performance compared to SOTA algorithms in many OpenAI GYM environments.
2. Compared to the slow training time of the NEAT algorithm, ENE offers a training speed that is over and above 500% faster while simultaneously producing higher-quality models.

## 6. – Conclusion

ENE, as a genetic optimization algorithm, not only demonstrates superior performance compared to SOTA algorithms in almost all OpenAI GYM environments, but it also achieves this with simpler models and lower energy consumption, highlighting the potential of this algorithm. On the other hand, PLAN (The Potentiation Learning Artificial Neural Network) architecture, through its innovative approach to neural network design, offers a compelling alternative to traditional methods. It is more interpretable, and strong learning capabilities like MLP's makes it a significant development in the AI and XAI landscape. The availability of the Potentiation Learning Artificial Neural Network library on GitHub further encourages exploration and adoption of this architecture, paving the way for new applications and advancements in artificial intelligence.

Codes in this article:

*https://github.com/HCB06/PyerualJetwork/tree/main/Welcome_to_PyerualJetwork/ExampleCodes*

PLAN & ENE:

*https://github.com/HCB06/PyerualJetwork*

## References

**[1] Ribeiro, M. T., Singh, S., & Guestrin, C. (2016).** "Why Should I Trust You?": Explaining the Predictions of Any Classifier. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (pp. 1135–1144).

**[2] Cox, D. R. (1958).** The regression analysis of binary sequences. *Journal of the Royal Statistical Society: Series B (Methodological), 20(2), 215-242.*

**[3] Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986).** Learning representations by back-propagating errors. *Nature, 323(6088), 533–536.*

**[4] Stanley, K. O., & Miikkulainen, R. (2002).** Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2), 99-127.

**[5] Polani, D., & Miikkulainen, R. (2000).** Eugenic neuro-evolution for reinforcement learning. In Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2000) (pp. 1041–1046). San Francisco: Morgan Kaufmann.

**[6] Beydili, H. C. (2024).** Plan (Pruning Learning Artificial Neural Network). *SSRN Electronic Journal*. https://doi.org/10.2139/ssrn.4862342