Opening Black Box: Potentiation Learning Artificial Neural Network

Author: Hasan Can Beydili

tchasancan@gmail.com

Abstract

No gradient descent, no backpropagation, no loss functions, no calculus, no epoch, no hidden layer, no unnecessary neurons, and in some cases no matrix multiplications for training. Yet, faster training and promising results.

Allow me to introduce Potentiation Learning Artificial Neural Network (PLAN), in this research, I have developed an architecture that holds a high potential for solving XAI(Explainable Artifical Intelligence) problem. A new artificial neural network architecture designed to address classification problems typically tackled by conventional Multi Layer Perceptron's. The Potentiation Learning Artificial Neural Network algorithm performs learning in an 'cumulative' rather than 'calculative' manner.

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.

1 - Introduce

1.1 - Introduction

Imagine a neural network architecture that eliminates the need for gradient calculation procedures inherent in the classic backpropagation algorithm and bypasses all the repetitive iterative processes. This new approach offers a faster and more comprehensible alternative. Additionally, it redefines the concept of the 'hidden layer' with a distinctive design that approaches this traditionally opaque area in novel ways.

In this paper, I introduce Potentiation Learning Artificial Neural Network, a neural network architecture that embodies these principles. Throughout this article, I will provide empirical evidence and examples from various projects to substantiate these claims. Our goal is to significantly reduce training times and develop models that are not only efficient but also compact in size.

1.2 - Synaptic Strengthening in Potentiation Learning Artificial Neural Network:

In biological systems, Long-Term Potentiation is characterized by the persistent strengthening of synaptic connections following high-frequency stimulation. This process involves intricate biochemical pathways that lead to long-lasting changes in synaptic efficacy.

The Potentiation Learning Artificial Neural Network's emulates this by enhancing the connections between its computational units when they are frequently activated together. This synaptic-like strengthening allows the network to robustly encode patterns in the data, akin to how Long-Term Potentiation solidifies memory traces in the brain.

1.3 - Activity-Dependent Plasticity:

Long-Term Potentiation's dependency on the activity patterns of neurons ensures that only significant and repeated stimuli lead to synaptic strengthening. This selective process is essential for efficient memory formation and recall.

Potentiation Learning Artificial Neural Network's employs a similar strategy, where the strengthening of connections is driven by the activity levels of its nodes. This means that frequently co-activated nodes within the network reinforce their connections, enabling Potentiation Learning Artificial Neural Network to selectively learn and retain critical information, much like how Long-Term Potentiation prioritizes important neural signals.

1.4 - Dynamic Learning and Adaptation:

In biological systems, Long-Term Potentiation allows for the dynamic balancing of memory retention and plasticity, providing a mechanism for long-term stability while still allowing for adaptive learning.

Potentiation Learning Artificial Neural Network’s mirrors this balance by dynamically adjusting the strength of its internal connections based on the learning experience. This allows the network to retain core knowledge while still being adaptable to new data inputs, effectively integrating the Long-Term Potentiation-like processes into its learning framework.

When updating a Potentiation Learning Artificial Neural Network model, instead of relying on traditional methods like retraining with old and new data as seen in other artificial neural network architectures, Potentiation Learning Artificial Neural Network’s capability for model updating offers exceptional efficiency. This process leverages the ability to enhance the model's performance by incorporating new data while minimizing the need for extensive retraining.

1.5 - Memory Consolidation and Retrieval:

Long-Term Potentiation facilitates the consolidation of long-term memories in the brain, making it easier to retrieve stored information when similar stimuli are encountered.

Similarly, Potentiation Learning Artificial Neural Network’s mechanism of connection strengthening ensures that learned information is consolidated within its architecture. This enhanced connectivity allows Potentiation Learning Artificial Neural Network to efficiently recall and generalize from its learned experiences, enabling it to respond accurately to new inputs that are similar to previously encountered patterns.

By incorporating Long-Term Potentiation-like principles, the Potentiation Learning Artificial Neural Network’s achieves a sophisticated approach to learning and memory storage. This approach not only strengthens its ability to process and store information but also aligns closely with the foundational principles observed in biological neural systems.

Furthermore, in addition to all these, the algorithm is incredibly fast due to not using matrix multiplication for training, no loops and no calculus anymore. It just use basic level linear algebra.

This article begins with an introduction to the architecture and proceeds to discuss the underlying concepts and mathematics of the Potentiation Learning Artificial Neural Network algorithm.

All the codes in this article:

https://github.com/HCB06/Anaplan/tree/main/Welcome_to_PLAN/Codes

https://github.com/HCB06/Anaplan/tree/main/Welcome_to_Anaplan/ExampleCodes

The Potentiaton Learning Artificial Neural Network is bifurcated into two distinct branches: the first is the " **Potentiaton Learning Artificial Neural Network (PLAN)”** which is capable of **linear separability**, and the second is the "**Deep Potentiaton Learning Artificial Neural Network (Deep PLAN)”** designed to handle **non-linear separability**. In the initial section of this work, I will detail the training, mathematical formulation, testing, and evaluation of the basic model, supported by textual and visual representations. Subsequently, I will explore the more sophisticated architecture of the "Deep Potentiaton Learning Artificial Neural Network," which demonstrates enhanced capabilities in handling complex, non-linear data distributions.

2 - Concepts:

2.1 - Aggregation Layer:

Aggregation layers in Deep Potentiation Artificial Neural Networks can follow immediately after the input layer and may occur multiple times throughout the network. It is not fully connected and does not involve any matrix-vector multiplication operations. Its purpose is to pass the input through specific activations and transfer the cumulative results of these transitions to the potentiation layer. (This will be discussed in more detail in the section on Deep Potentiation Learning Artificial Neural Networks.)

2.2 - Potentiation Layer:

In the Potentiation Learning Artificial Neural Network architecture, there is exactly one Potentiation Layer, which is fully connected and positioned immediately prior to the output layer. All features directly given to Potentiation Layer’s connections, this will allow features that occur more frequently in the connections to be summed later and represented by larger numbers in memory, effectively achieving the learning process on its own.

2.3 - Activation Potentiation:

Activation Potentiation is a list containing string expressions that specify the activation functions for Deep Potentiation Learning Artificial Neural Networks. It’s using in Aggregation Layers. (This will be discussed in more detail in the section on Deep Potentiation Learning Artificial Neural Networks.)

2.4 - Long-Term Depression (LTD):

Long-Term Depression (LTD) is a process that plays a crucial role in the memory and learning mechanisms of biological neural networks. Essentially, it induces a reduction in the strength of certain synaptic connections (weights). While excessive LTD can lead to neurological issues, when maintained at a stable level, it serves as a biological mechanism that enhances learning. (Deep Potentiation Learning Artificial Neural Network hyperparameter)

3 - Potentiation Learning Artificial Neural Network Architecture:

Potentiaton Learning Artificial Neural Network enables nearly perfect linear separations without requiring any hyperparameters. All of this is achieved very quickly through a method I call **cumulative learning**.

In Potentiation Learning Artificial Neural Network’s, there are no hidden layers because the architecture is designed to operate without the traditional 'black box' approach of hidden processes. Each layer’s function and transformation are transparent and well-defined. This structure simplifies the understanding and debugging of the network while maintaining high performance.

To illustrate how these components work together, let’s delve into an example application where Potentiation Learning Artificial Neural Network demonstrates its unique capabilities.

3.1 - Training And Math:

Let's consider we have a handwritten digit classification dataset consisting of 28x28 pixel images, with 10 classes (from 0 to 9).

We define 10 neurons for potentiation layer, then our weight matrix should be (10,784) (10 = rows, 784 = columns). In this scenario, the weights affecting the first neuron in the potentiation layer vector resulting from the matrix-vector multiplication will be all the columns in the first row of the 10 rows. To understand, let’s recall the matrix-vector multiplication; in the current scenario, it is worth noting that all elements of the input vector are multiplied by all elements in the first row of the weight matrix and summed up. The first output obtained from this result will represent the first neuron in our hidden layer. The continuation of this process to compute the other neurons means moving down to the next row in the weight matrix and repeating the multiplication and summation with the input. This brings us to the point where we can control the information influencing all neurons in the hidden layer ourselves. ***In fact, each row of the weight matrices holds the information of each neuron separately.*** So, with this technique, we govern the hidden layer and eliminate its secrecy, and we call this layer the **“potentiation layer”**. Thanks to this layer, we encode the brightness values of the pixels and their positions in the photograph we provided as input into a certain neuron of our initial weight matrix.

Configuration of Potentiation Layer in Potentiation Learning Artificial Neural Network’s:

In our Potentiation Learning Artificial Neural Network, the potentiation layer is structured based on the number of classes in the task. The weight matrix weights defining this layer has:

Rows: Equal to the number of classes, representing class-specific neurons.

Columns: Equal to the number of input features, reflecting input dimensionality.

This setup ensures each class has dedicated weights for effective and efficient classification. The programming language to be used in the article is **GNU Octave**, which has MATLAB syntax.

```
weights = ones(10,784); % Matrix formed by ones


% Info: 10 is class count, 784 is feature count.
```

We define the weight for potentiation layer(weights) to be constant and equal to 1 instead of randomly generating them within a certain range, because we will never optimize weight values; we will adjust parameters instead.

It is quite simple to understand compared to other artificial neural network architectures.

Assuming we have performed the same operations for the other classes, we can now write the artificial neural network structure *during the learning process*. Let's consider that we will input the first data point (photograph) belonging to class 1:

```
inputLayer = normalization(inputLayer); % inputs in range 0 - 1

%% POTENTIATION LAYER %%

  weights(class,:) = inputLayer;


[[1,0,0,0]
 [0,0,1,1]]


% Here, the first row, representing the first neuron,
% hold the features of the first class,

% while the last row,representing the last neuron,
% hold the features of the second class.
```

Let's explain the underlying principle of the potentiation layer with a simple example.

In a situation where each feature is completely distinct from the others, the weight matrix of our potentiation layer would take a diagonal form. To illustrate this, let's consider a simple network with three classes.

```
 [[1,0,0]
  [0,1,0]
  [0,0,1]]

% The column corresponding to the 1st row belongs to the 1st class.
% The column corresponding to the 2nd row belongs to the 2nd class.
% The column corresponding to the 3rd row belongs to the 3rd class.
% Emphasis.
```

The matrices constituting the potentiation potentiation layer should typically be in this "diagonal form"**. If you remember, we said that the reason for this is that the single feature coming from the input layer searches for the neuron that belongs from top to bottom. Rows represent neurons.**

```
% x = Single pixel value of photo

[[x*1,0,0]  % x * 1 = x
  [0,1,0]
  [0,0,1]]

 [[1,0,0]   % x * 0 = x
  [x*0,1,0]
  [0,0,1]]

 [[1,0,0]
  [0,1,0]
  [x*0,0,1]] % x * 0 = 0

% Then x pixel represents a feature belonging to the first class.
```

For a data point labeled ‘1’, only the 1st class is ‘1’, while all other classes are ‘0’. **There’s no need to even use an error function because the error is already ‘0’ and we also don’t need even softmax activation function. Softmax is completly optional.** We have now trained a data point with a ‘0’ error as quickly as possible.

So, how can we do this for other data samples? Let’s continue reading to find out.

Training a data sample belonging to a new class and training data samples belonging to the same class.

As I mentioned in the abstract of the paper, Potentiation Learning Artificial Neural Network algorithm performs learning in an 'cumulative' rather than 'calculative' manner.

After performing the same operations for data samples labeled ‘2’ belonging to the second class, we will merge the trained matrices for the first class with the trained matrices for the second class. We can accomplish this merging by adding the matrices together:

Note: In order to make it easier to understand in this paper, weights are described to be saved to and loaded from a file directory. You can implement this within the program, which will make it faster.

If we think 1 labeled data sample is our first training sample:

```
if class == 1 % [[After the initial training,
  % save the matrices to a file following the steps]]

  weights = sprintf('weights/weights.mat');
   save(weights, 'weights');


else % if [[the matrices are already saved, retrieve them and add them
together.
 % When it's a new class, we won't perform addition but merging.
 % For the same class, we'll add them together, thus reinforcing
 % existing features and adding new ones to the matrix.]]

  newWeights = weights;

  weights = sprintf('weights/weights.mat');
    load(weights);

  weights += newWeights;

  weights = sprintf('weights/weights.mat');
   save(weights, 'weights');


end
```

Actually, there is no addition operation here because we separate the rows in such a way that they do not affect each other, so the addition operation here only allows merging. What we will obtain are matrices trained for two separate classes. It would be more logical to do this for other classes as well and then merge this modular structure at the end. If we want to include a new data sample in a model trained for a specific class, what we need to do is to follow the steps above again to determine the range of rows containing the information for the class to which the data sample belongs, and then add the information from the trained matrix (since both will already have the same row range). So, it's about combining the trained model with the model trained for a single data sample.

```
%% FOR OTHER CLASSES %%

  % class1_OldWeights = [[0,0,0,0]
                         [0,0,1,1]]


            +                 they are merging. = [[1,1,0,0]
                                                   [0,0,1,1]]

  % class2_NewWeights = [[1,1,0,0]
                         [0,0,0,0]]





%% FOR SAME CLASSES %%


  % class1_OldWeights = [[0,0,0,0]
                         [0,0,1,1]]


            +                  they are summing. = [[0,0,0,0]
                                                    [0,0,2,2]]

  % class1_NewWeights = [[0,0,0,0]
                         [0,0,1,1]]
```

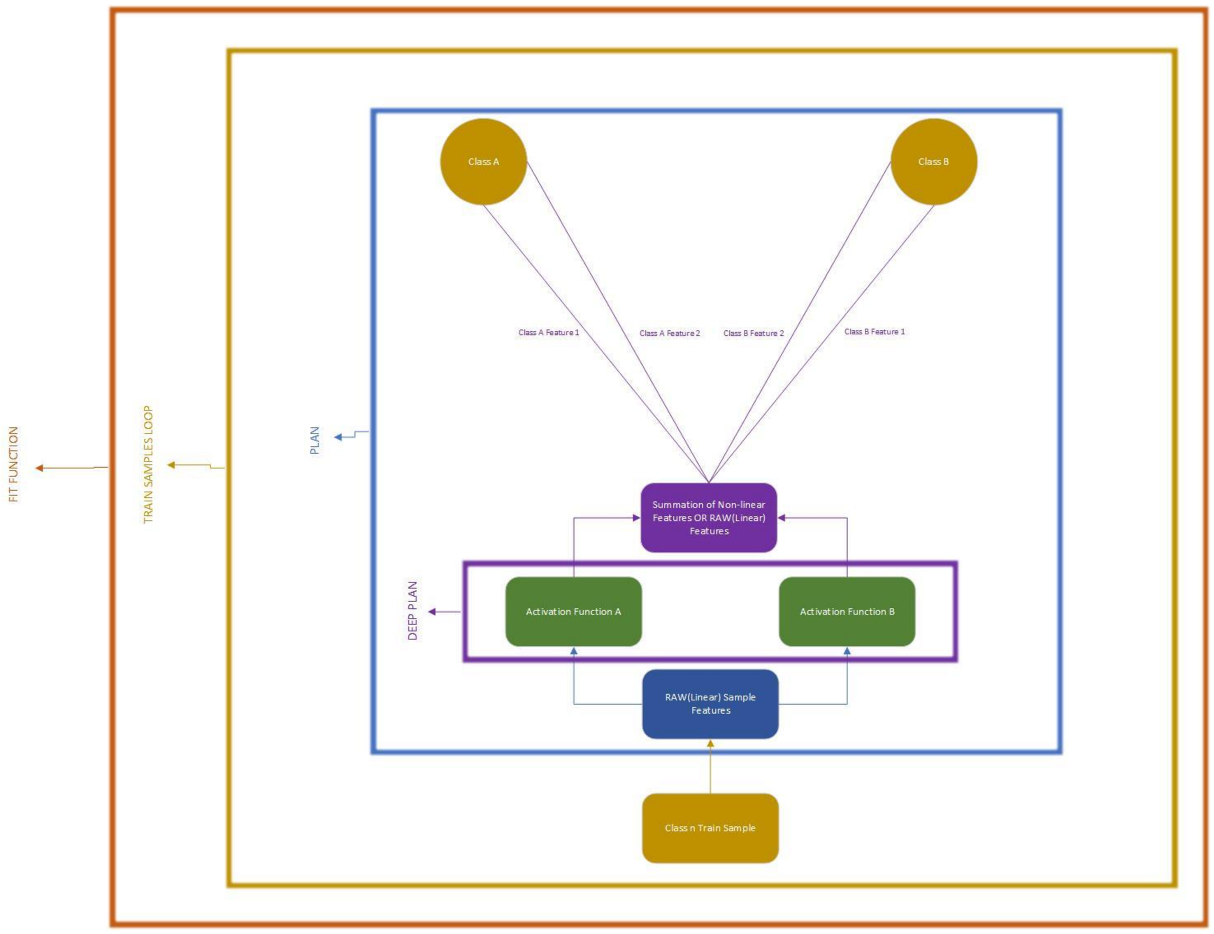


Figure1 PLAN Training Architecture

At the beginning of the text, we were talking about Long-Term Potentiation. When training new data belonging to the same class, we strengthen the connections, meaning we add +feature to the values of the pixels in that class's rows. In the potentiation layer, for indices with the same pattern, we add +feature, while for indices with a new pattern. In other words, we are aggregating the model trained for a single data point with the model trained for multiple data points. This underscores the importance of having equally distributed data when preparing data for similar classes. It is important to gather data with equal distribution specifically for similar classes to directly influence the F1 score value of the model. **Because this architecture not only resembles Long-Term Potentiation in biological neural networks but also resembles how biological neural networks store learned information in memory**, the presence of similar features across multiple classes can dull the model's generalization ability and cause confusion, just like in humans. Humans learn best what they repeat the most. Just like in humans, the more these connections are strengthened (+feature,

+feature, +feature), a bias based on experience is formed, allowing one feature's connection to be triggered without considering the presence of other features. So, it creates a bias based on experience, just like in humans. **In this architecture, the model's generalization ability is parallel to equal data distribution**.

Here its a mathematical reason:

```
% x = Same feature element (pixel) for the 1st and 2nd class.
% y = Pixel
% z = Pixel
% x = 255
% y = 255
% z = 255

% TRAINING:

 [[1,1,0]  % A feature matrix trained with an equal number of samples for
each class..
  [1,1,1]
  [0,0,1]]

% TEST (The output expected is for the 2nd class.):

[[x*1,1,0]  % x * 1 = x
  [x*1,1,1] % x * 1 = x
  [x*0,0,1]] % x * 0 = 0

for 1st class = x = 255
for 2nd class = x = 255
for 3rd class = x = 0

% The other features are examined:

[[1,y*1,0]  % y * 1 = y
  [1,y*1,1] % y * 1 = y
  [0,y*0,1]] % y * 0 = 0

for 1st class = x + y = 510
for 2nd class = x + y = 510
for 3rd class = x + y = 0

% The other features are examined:

[[1,1,z*0]  % z * 0 = 0
  [1,1,z*1] % z * 1 = z
  [0,0,z*1]] % z * 1 = z

for 1st class = x + y + z = 510
for 2nd class = x + y + z = 765
for 3rd class = x + y + z = 255

[[x*1,y*1,z*0]  % (x * 1) + (y * 1) + (z * 0) = 510 (1st neuron).
  [x*1,y*1,z*1] % (x * 1) + (y * 1) + (z * 1) = 765 (2nd neuron).
  [x*0,y*0,z*1]] % (x * 1) + (y * 1) + (z * 1) = 255 (3rd neuron).

% IT IS UNDERSTOOD THAT THERE IS AN INPUT BELONGING TO THE 2ND CLASS.
% LET'S ADD A NEW EXAMPLE BELONGING TO THE 1ST CLASS TO THE MODEL.
% TRAINING:

 [[2,1,0]  % One more example has been added for the 1st class.
  [1,1,1]
  [0,0,1]]

% TEST (The output expected is for the 2nd class.):

 [[x*2,y*1,z*0]  % (x * 2) + (y * 1) + (z * 0) = 765 (1st neuron).
  [x*1,y*1,z*1] % (x * 1) + (y * 1) + (z * 1) = 765 (2nd neuron).
  [x*0,y*0,z*1]] % (x * 1) + (y * 1) + (z * 1) = 255 (3rd neuron).

% BIAS HAS LED TO CONFUSION.
% TO SOLVE:

% TRAINING:

 [[2,1,0]  % One more example has been added for the 2nd class. (Samples
equalized again)
  [2,2,1]
  [0,0,1]]

% TEST (The output expected is for the 2nd class.):

[[x*2,y*1,z*0]  % (x * 2) + (y * 1) + (z * 0) = 765 (1st neuron).
  [x*2,y*2,z*1] % (x * 2) + (y * 2) + (z * 1) = 1275 (2nd neuron).
  [x*0,y*0,z*1]] % (x * 1) + (y * 1) + (z * 1) = 255 (3rd neuron).

% IT IS UNDERSTOOD ONCE AGAIN THAT THERE IS AN INPUT BELONGING TO THE 2ND
CLASS
```

Let’s display all the training steps together on a script before the test:

```
%% WEIGHT INITIALIZATION: %%

  weights = ones(10,784);



  %% POTENTIATION LAYER %%

  inputLayer = normalization(inputLayer) %% inputs in range 0 - 1
  weights(class,:) = inputLayer;


%% MERGING/SUMMING - SAVING WEIGHTS: %%

  if class == 1 % [[After the initial training,
   save the matrices to a file following the steps]]

    weights = sprintf('weights/weights.mat');

     save(weights, 'weights');
  else % if [[the matrices are already saved, retrieve them and add them
together.

     % When it's a new class, we won't perform addition but merging.
     % For the same class, we'll add them together, thus reinforcing
     % existing features and adding new ones to the matrix.]]

    newWeights = weights;

    weights = sprintf('weights/weights.mat');

      load(weights);


    weights += newWeights;

    weights = sprintf('weights/weights.mat');

     save(weights, 'weights');

  end
```

3.2 - Test:

3.2.1 - Potentiation Learning Model Testing And Predictions

In the testing and prediction phase, it is sufficient to multiply the weight matrix learned during training with the test samples.

```
%% Only matrix multiplication(For PLAN):

outputLayer = weights * inputLayer;
```

4 - Results For Potentiation Learning Artificial Neural Networks:

All the codes for this application: *https://github.com/HCB06/Anaplan/tree/main/Welcome_to_PLAN/Codes*.

You can train and test models wtih Potentiation Learning Artificial Neural Network architecture. All datasets in this article: *https://github.com/HCB06/Anaplan/tree/main/Welcome_to_Anaplan/ExampleCodes*

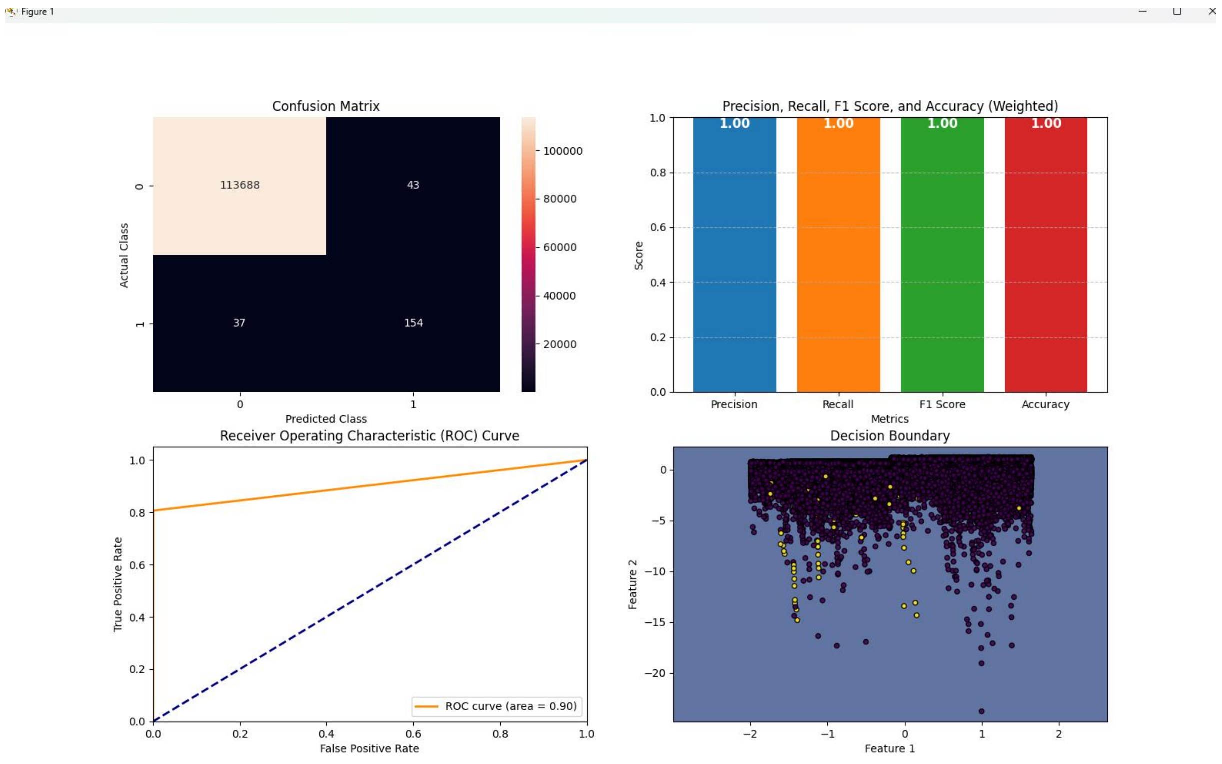
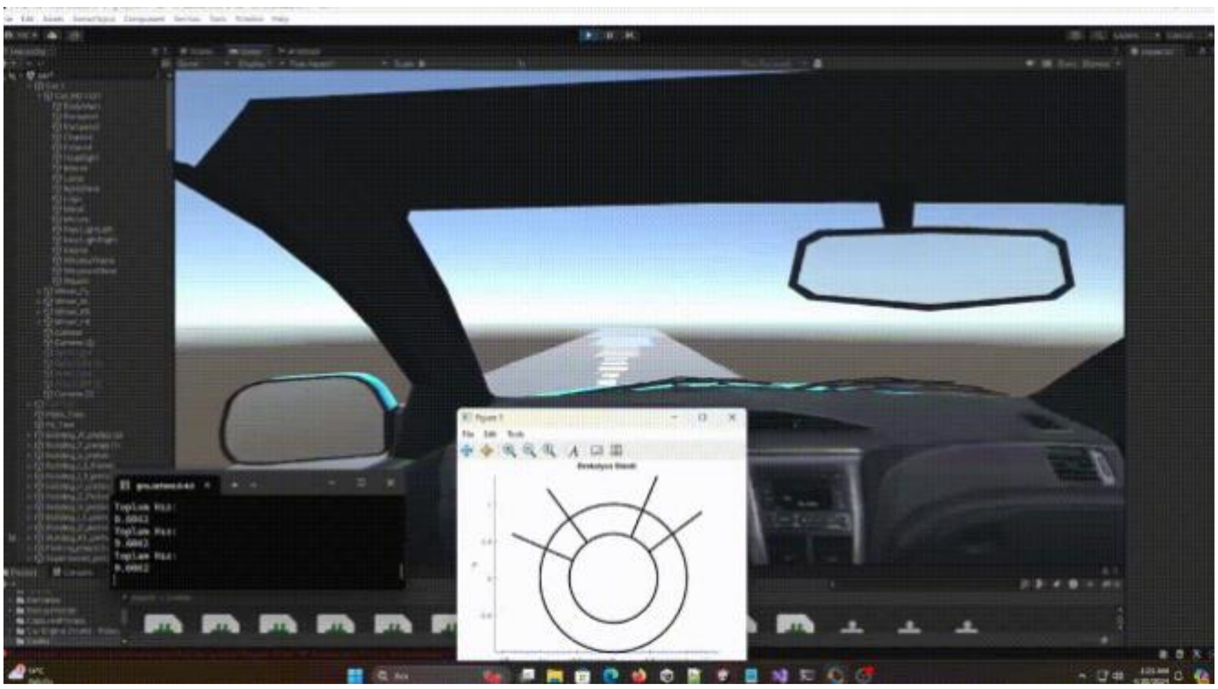


Figure2 Performance of PLAN on credit card fraud detection dataset
https://github.com/HCB06/Anaplan/blob/main/Welcome_to_Anaplan/ExampleCodes/creditcard_fraud(plan).py



```
Toplam Hız:
0.0043
Toplam Hız:
0.0043
Toplam Hız:
0.0042
```

Figure3 Performance of PLAN on visual dataset

https://www.linkedin.com/posts/hasan-can-beydili-77a1b9270_ke%C5%9Ffetti%C4%9Fim-e%C4%9Fitim-mimarisini-kullanan-modelimi-activity-7190854824471543808-CWhb?utm_source=share&utm_medium=member_desktop

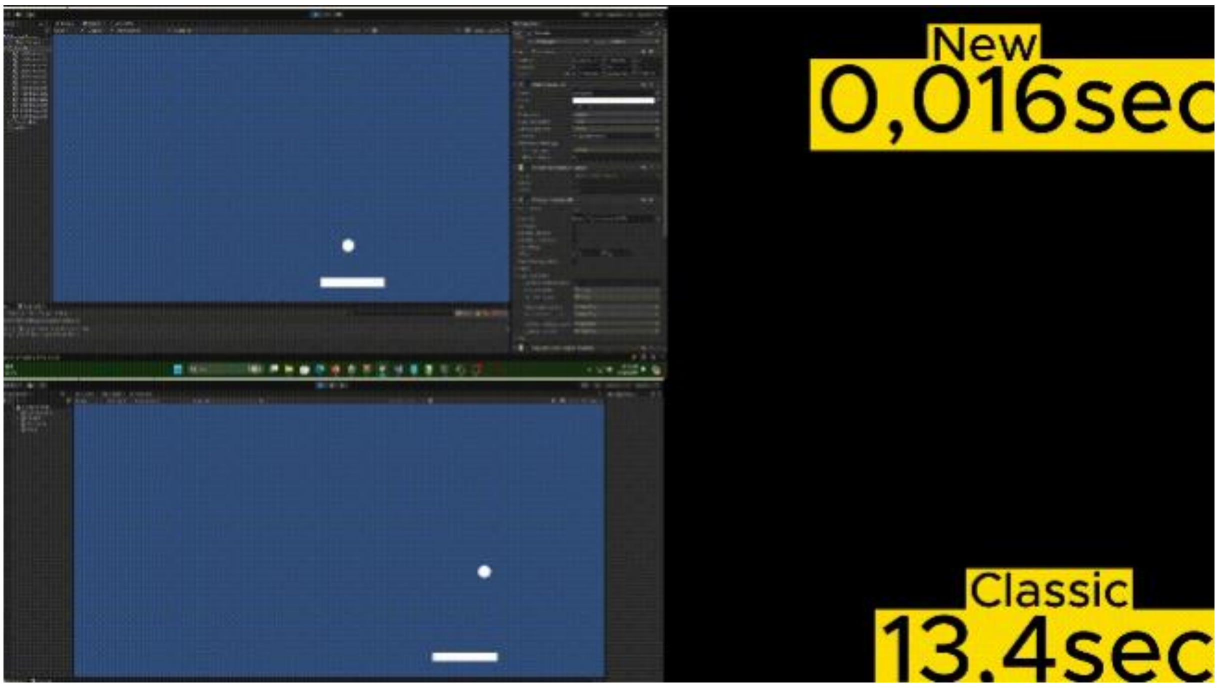


Figure4 Performance of PLAN on brick braker game

https://www.linkedin.com/posts/hasan-can-beydili-77a1b9270_brick-breaker-oyunu-yap%C4%B1yorum-burada-anlatt%C4%B1%C4%9F%C4%B1m-activity-7196972136014434307-6rRQ?utm_source=share&utm_medium=member_desktop

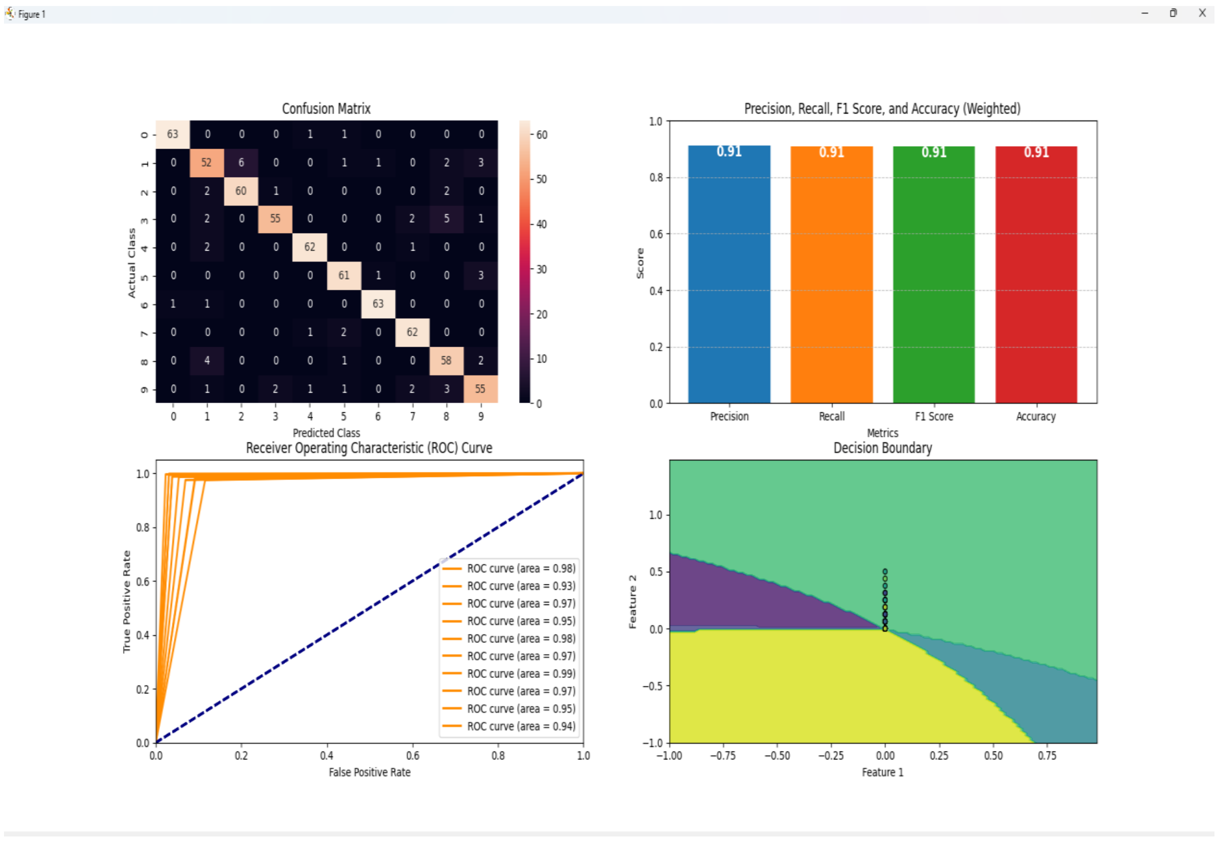


Figure5 Performance of PLAN on digits dataset
[*https://github.com/HCB06/Anaplan/blob/main/Welcome_to_Anaplan/ExampleCodes/digits(plan).py*](https://github.com/HCB06/Anaplan/blob/main/Welcome_to_Anaplan/ExampleCodes/digits(plan).py)

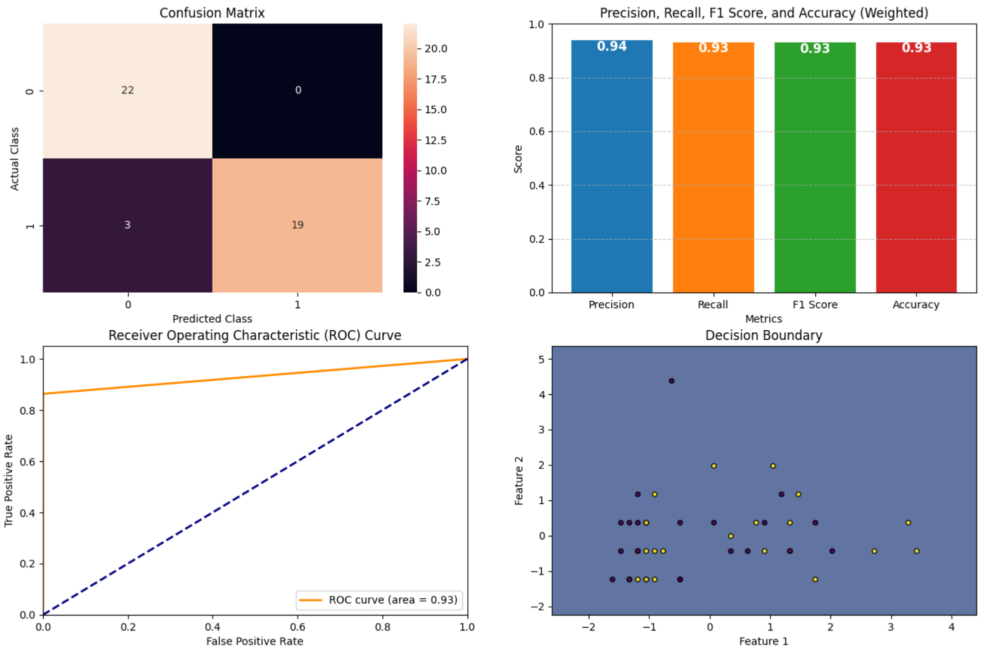


Figure6 Performance of PLAN on cervical cancer dataset
https://github.com/HCB06/Anaplan/blob/main/Welcome_to_Anaplan/ExampleCodes/cervical_cancer(all_algorithms).py

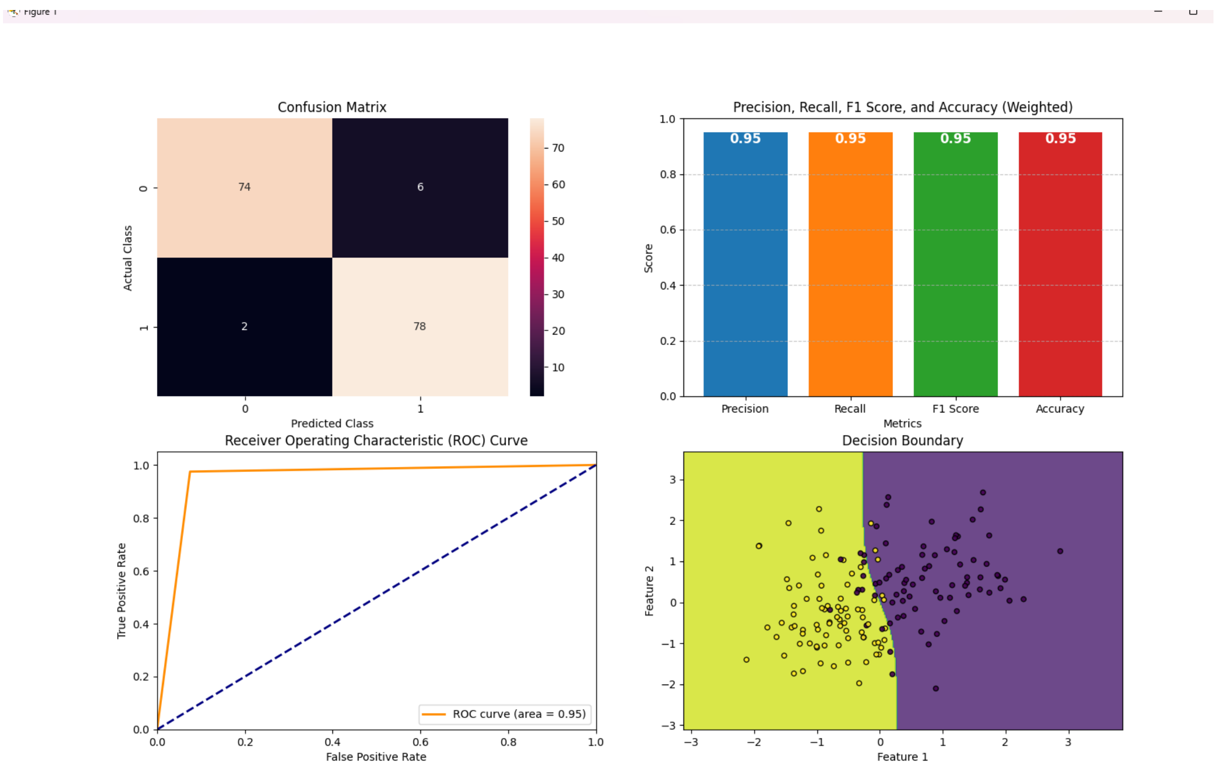


Figure7 Performance of PLAN on breast cancer wisconsin dataset
https://github.com/HCB06/Anaplan/blob/main/Welcome_to_Anaplan/ExampleCodes/breast_cancer_wisconsin(plan).py

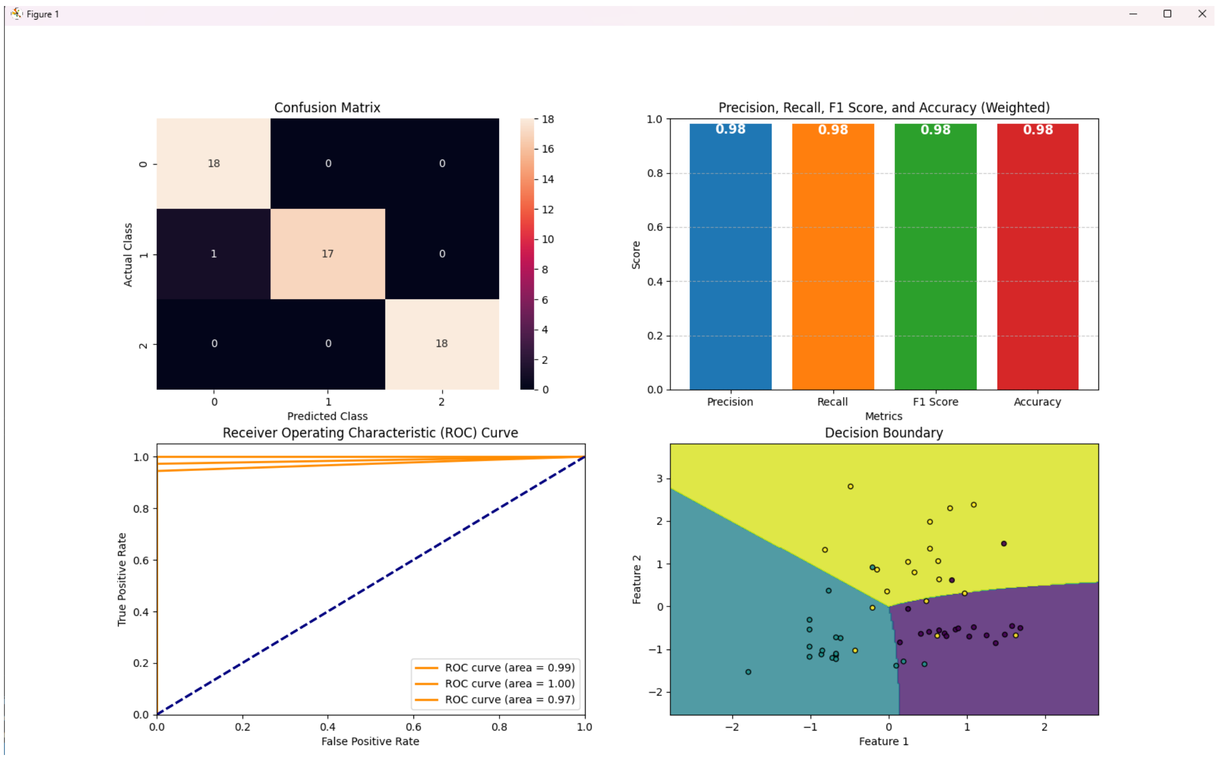


Figure8 Performance of PLAN on wine dataset
https://github.com/HCB06/Anaplan/blob/main/Welcome_to_Anaplan/ExampleCodes/wine(plan).py

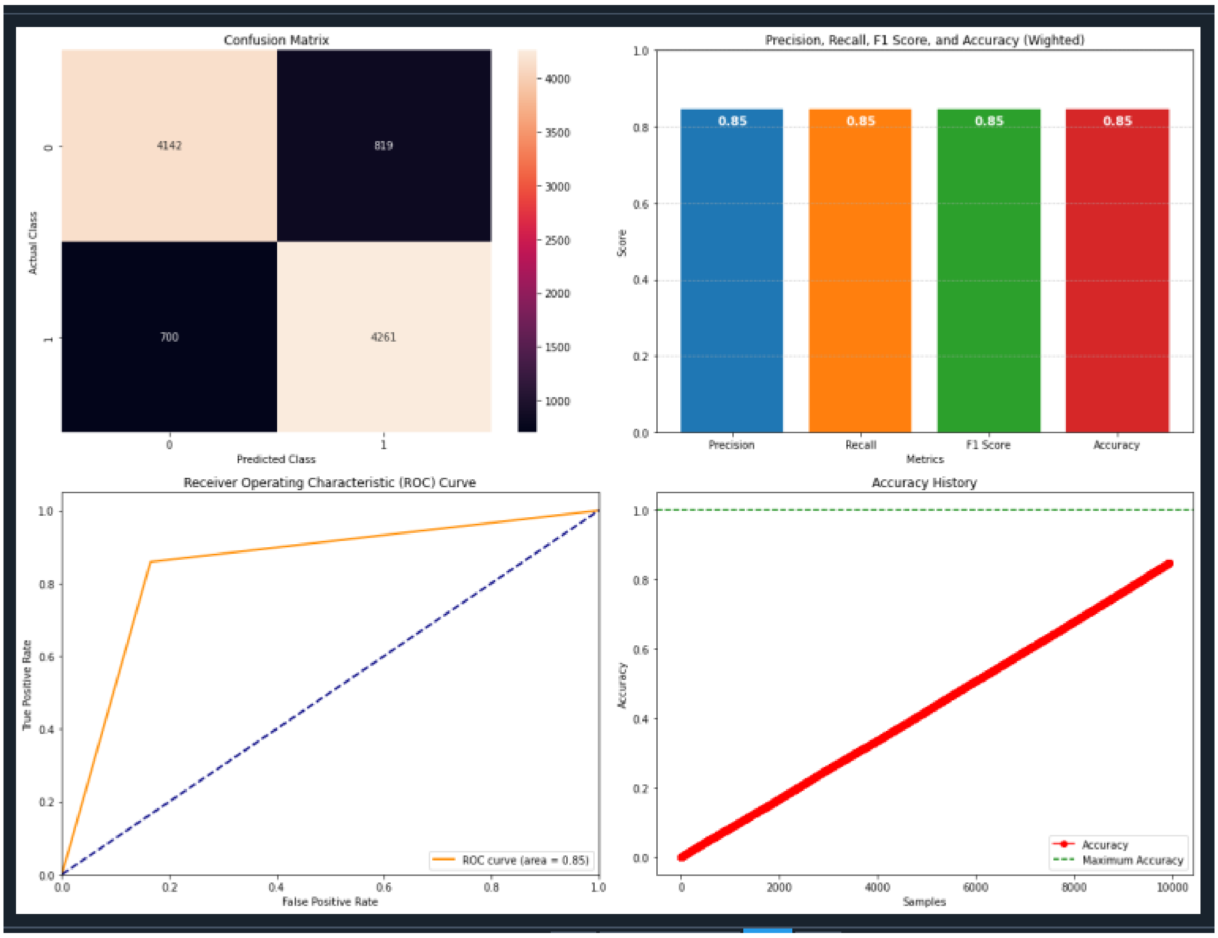


Figure9 Performance of PLAN on IMDB 50K movie reviews dataset
https://github.com/HCB06/Anaplan/blob/main/Welcome_to_Anaplan/ExampleCodes/NLPlan/imdb.py

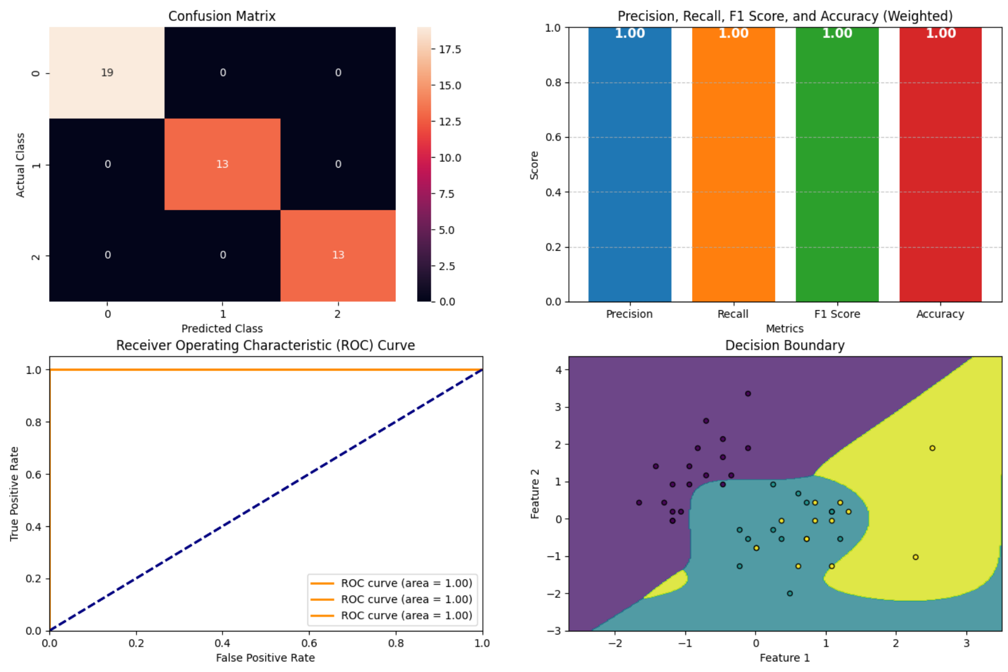


Figure10 Performance of PLAN on iris dataset

[*https://github.com/HCB06/Anaplan/blob/main/Welcome_to_Anaplan/ExampleCodes/iris(plan).py*](https://github.com/HCB06/Anaplan/blob/main/Welcome_to_Anaplan/ExampleCodes/iris(plan).py)

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

Especially the absence of any matrix multiplication during training makes the algorithm particularly special and noteworthy.

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

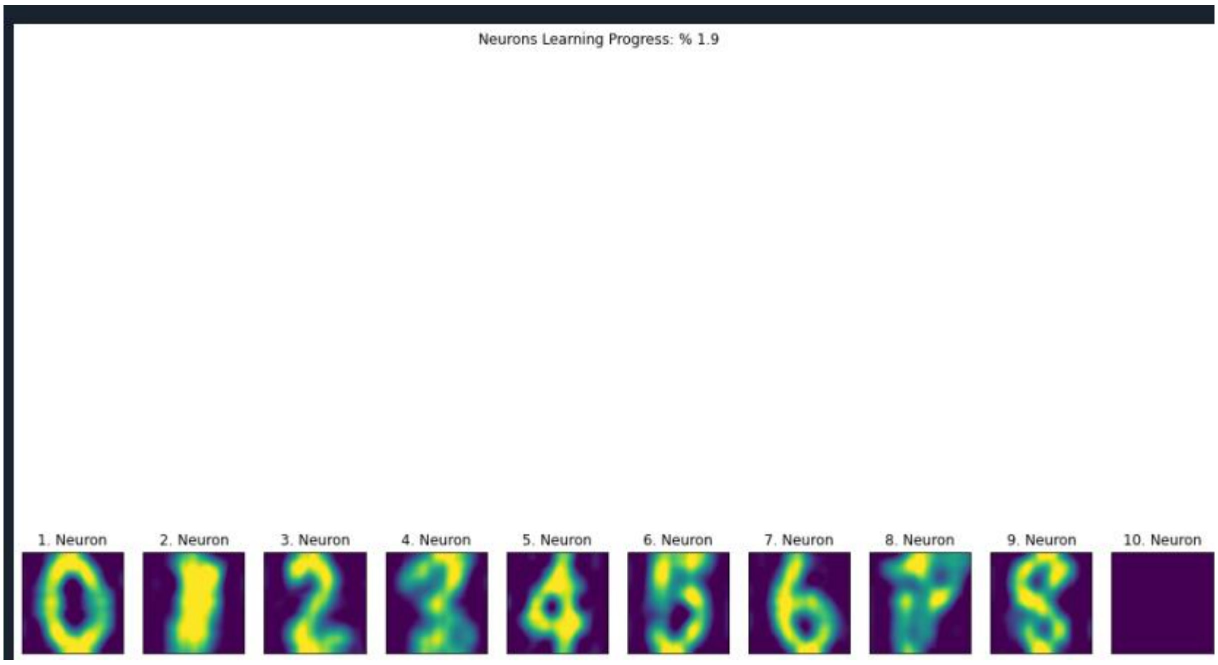


Figure11 Performance of PLAN

Potentiation layer (Learning at %1.9 with digits dataset)

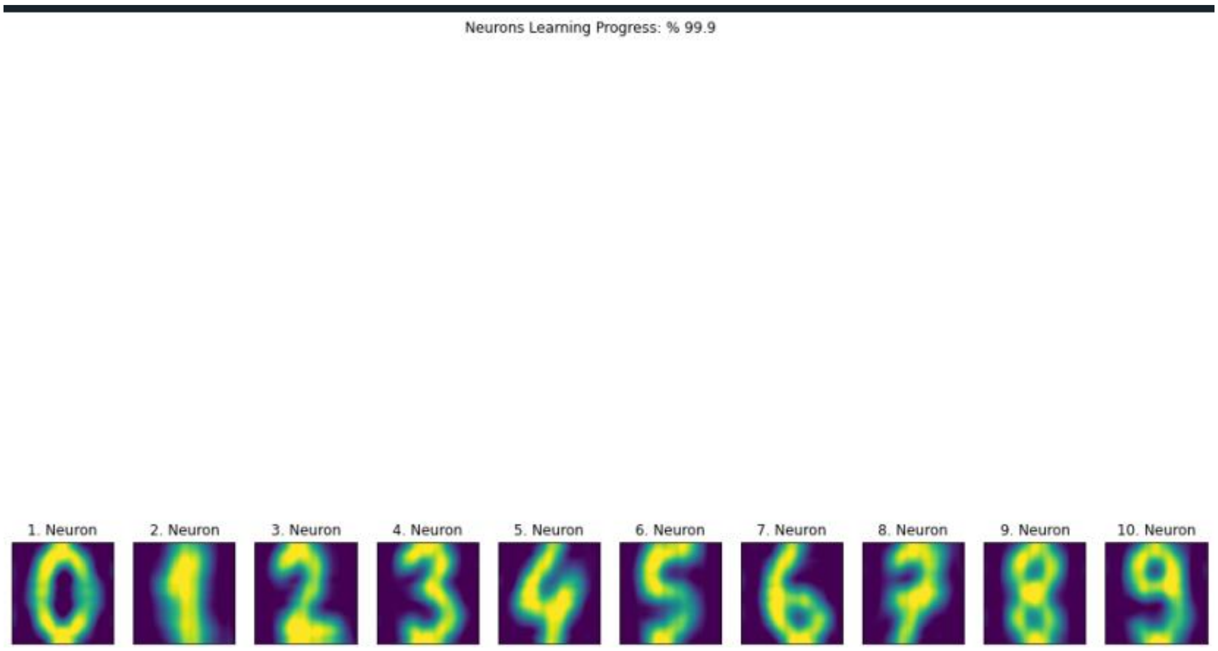


Figure12 Performance of PLAN

Potentiation layer (Learning at %99.9 with digits dataset)

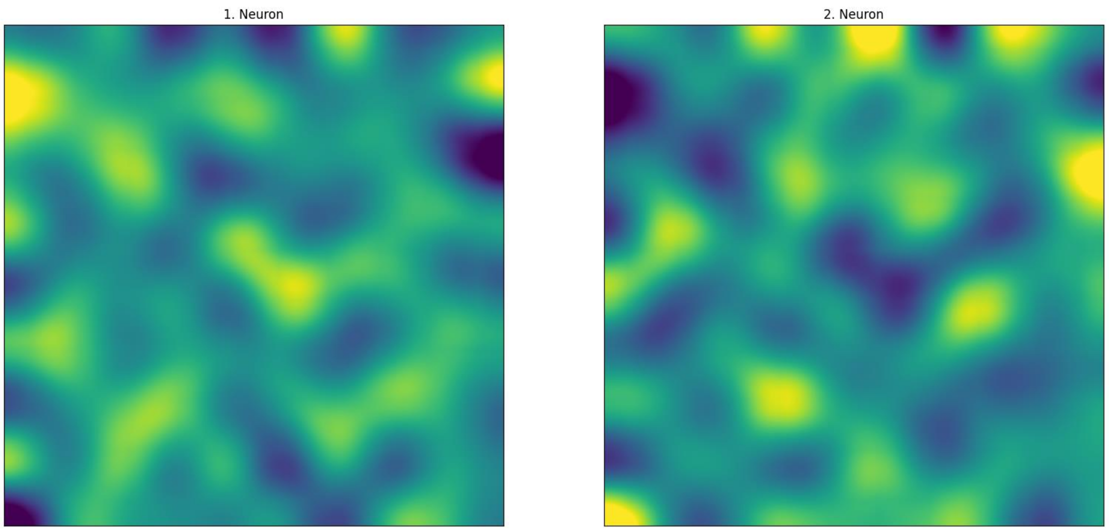


Figure13 Performance of PLAN

Potentiation layer (Learning at %100 with imdb 50k movie reviews dataset - selected feature count: 100)

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:

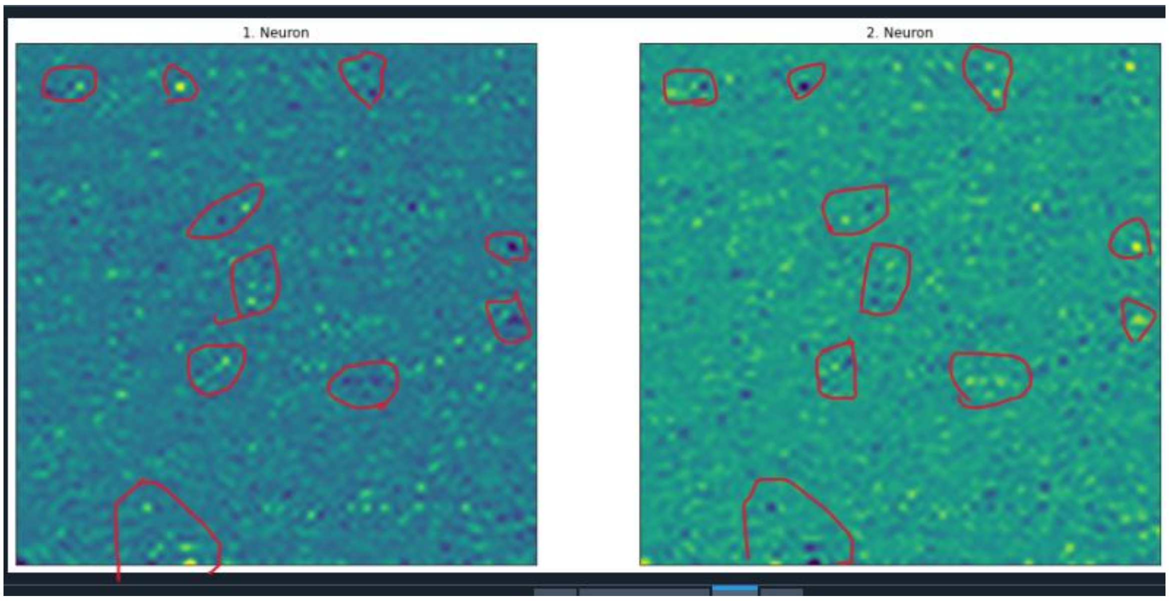


Figure14 Performance of PLAN

Potentiation layer (Learning at %100 with imdb 50k movie reviews dataset - selected feature count: 6084)

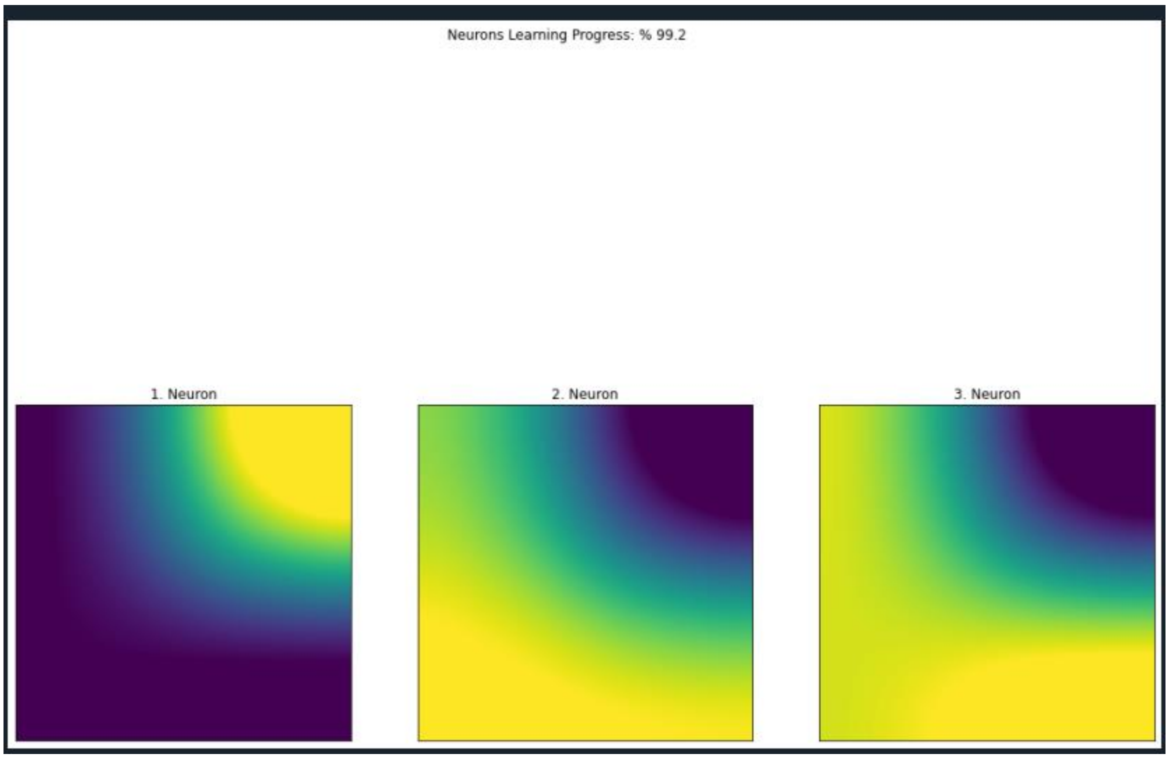


Figure15 Performance of PLAN

Potentiation layer (Learning at %99.2 with iris dataset)

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

5 - Deep Potentiation Learning Artificial Neural Networks:

In Deep Potentiation Learning Artificial Neural Networks, depth is constructed using aggregation layers. These layers possess significantly greater potential for efficient depth creation compared to hidden layers in traditional deep learning architectures

5.1 - Training And Math For Deep Potentiation Learning Artificial Neural Network:

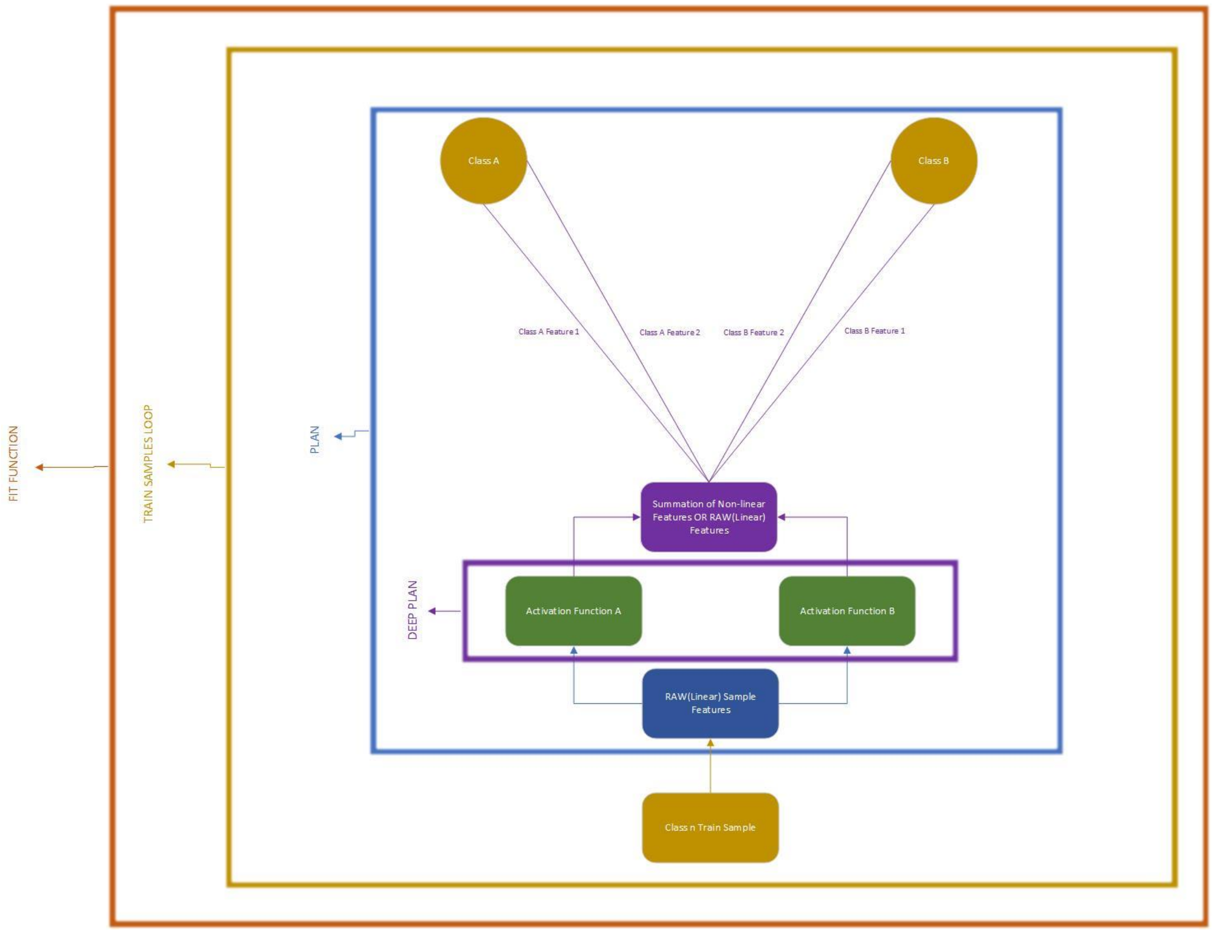


Figure16 PLAN Training Architecture

The training process for the Deep Potentiation Learning Artificial Neural Network mirrors that of the Potentiation Learning Artificial Neural Network. However, before adding the input values to the corresponding rows in the weight matrix, the inputs are passed through an activation function or functions. If multiple activation functions are selected, the outputs of all activations are summed before the input is recorded in the weight matrix. I have designated this list of activation functions as “activation_potentiation.”

Let us recall the training process of the Potentiation Learning Artificial Neural Network:

```
İnputLayer = normalization(inputLayer); % inputs in range 0 – 1

%% AGGREGATION LAYERS %%

newInputLayer = zeros(length(inputLayer))

for i = 1: length(activaiton_potentiation)

      if activation_potentiation(i) == ‘sigmoid’ %% example

       newInputLayer += sigmoid(inputLayer); %% if activation is sigmoid

            %% other activation functions handling…

      end

end


%% POTENTIATION LAYER %%

weights(class,:) = newInputLayer;
```

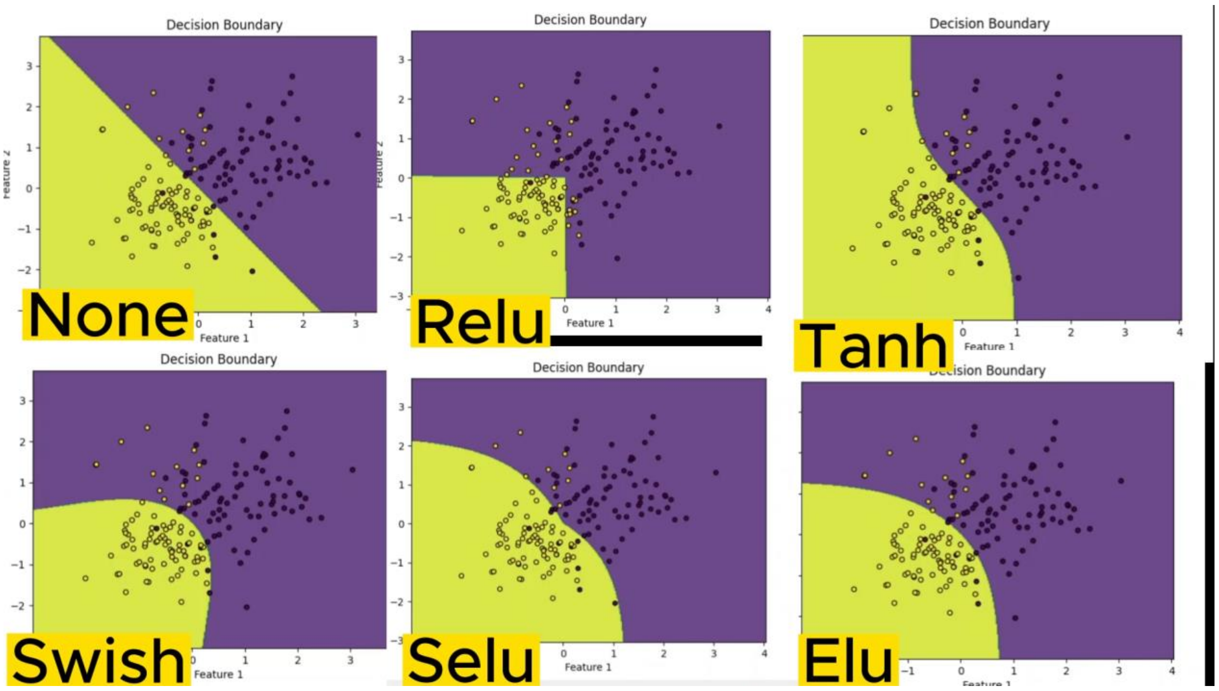


Figure17 Decision boundaries of the model for different activation functions in aggregation layer: (None = no activation function (linear))

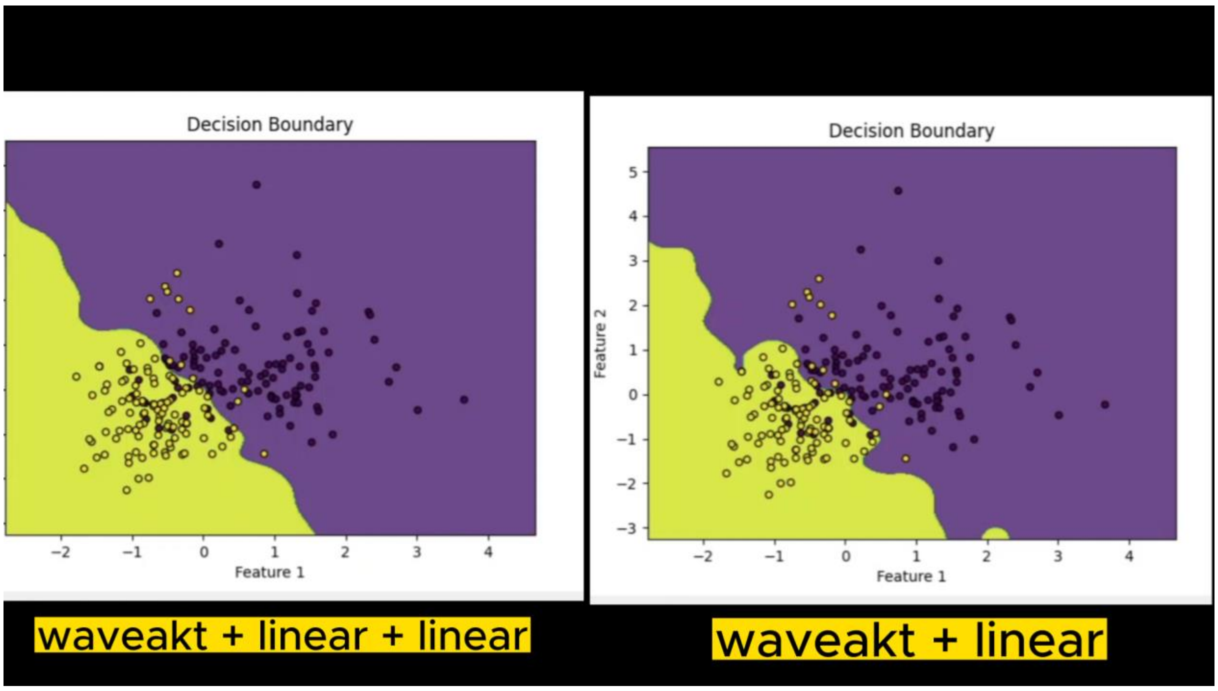


Figure18 Activation functions can be combined to mitigate each other's weaknesses, thereby adapting to the shape of the data and uncovering deeper features.

5.1.1 - Long-Term Depression (LTD) Role For Deep Potentiation Learning Artificial Neural Network:

LTD is optional. The concept of Long-Term Depression (LTD) was discussed in the conceptual section. In the context of the algorithm, LTD is implemented by introducing random noise equivalent to the LTD parameter before the input vector is added to the weight matrix during training. The primary goal of this approach is to prevent the model from overfitting and to enhance its generalization performance. By employing this technique, improved performance can be achieved with large and medium-sized datasets.

```
%% AGGREGATION LAYERS %%

İnputLayer = normalization(inputLayer); % inputs in range 0 – 1

newInputLayer = zeros(length(inputLayer))

for i = 1: length(activaiton_potentiation)

      if activation_potentiation(i) == 'sigmoid' %% example

       newInputLayer += sigmoid(inputLayer); %% if activation is sigmoid

            %% other activaiton functions handling…
```

```
        end

end

depression_vector = rand(length(newInputLayer))

for i = 1:LTD

        newInputLayer -= depression_vector

end


%% POTENTIATION LAYER %%

weights(class,:) = newInputLayer;
```

In this process, data loss is almost nonexistent, as cumulative learning takes place, making it easier to capture the most general features.

5.2 - Testing And Predictions For Deep Potentiation Learning Artificial Neural Network:

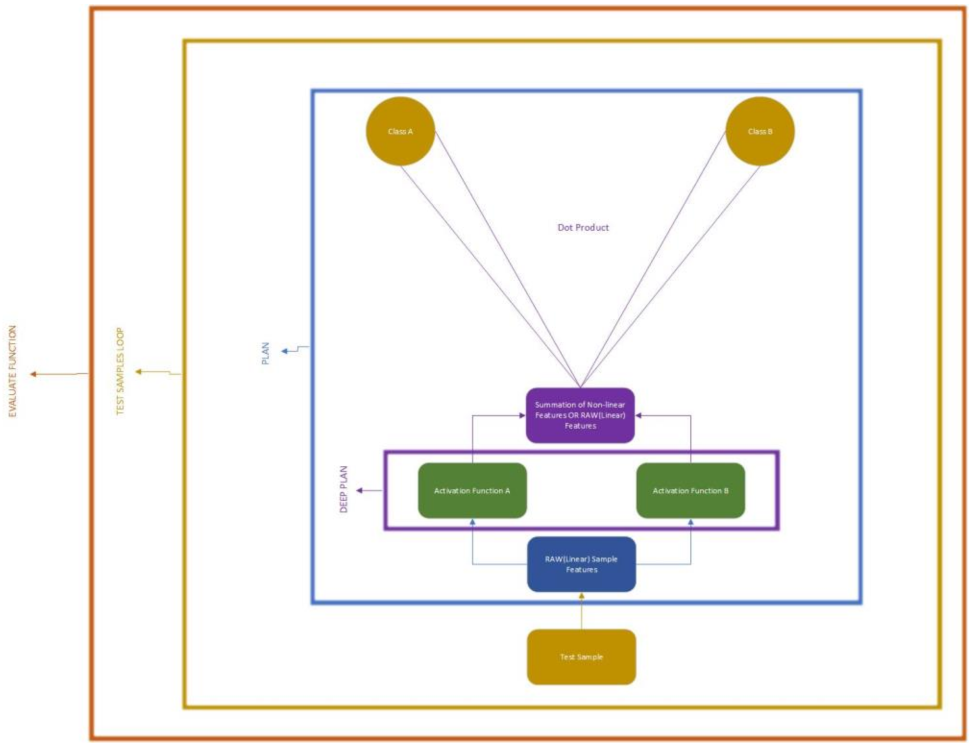


Figure19 PLAN Testing (Predicting) Architecture

```
for i = 1: length(activaiton_potentiation)

      if activation_potentiation(i) == 'sigmoid' %% example

            inputLayer  = sigmoid(inputLayer); %% if activation is sigmoid

      end

end

outputLayer = weights * inputLayer;
```

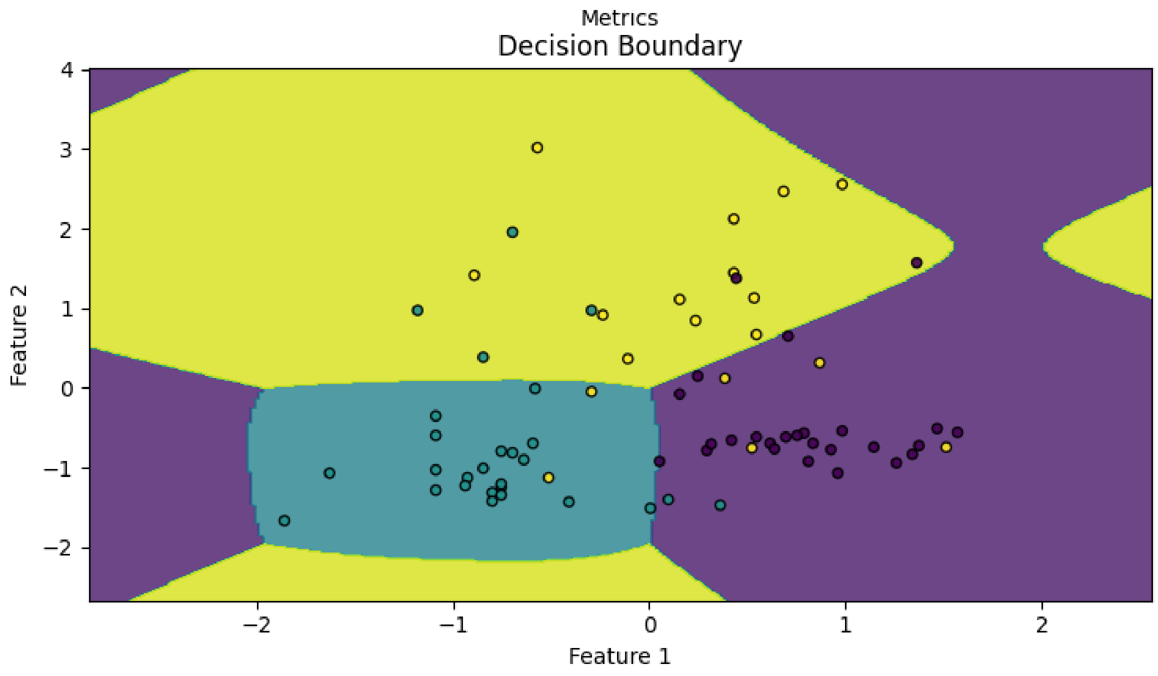


Figure20 (Decision Boundary of Deep PLAN)

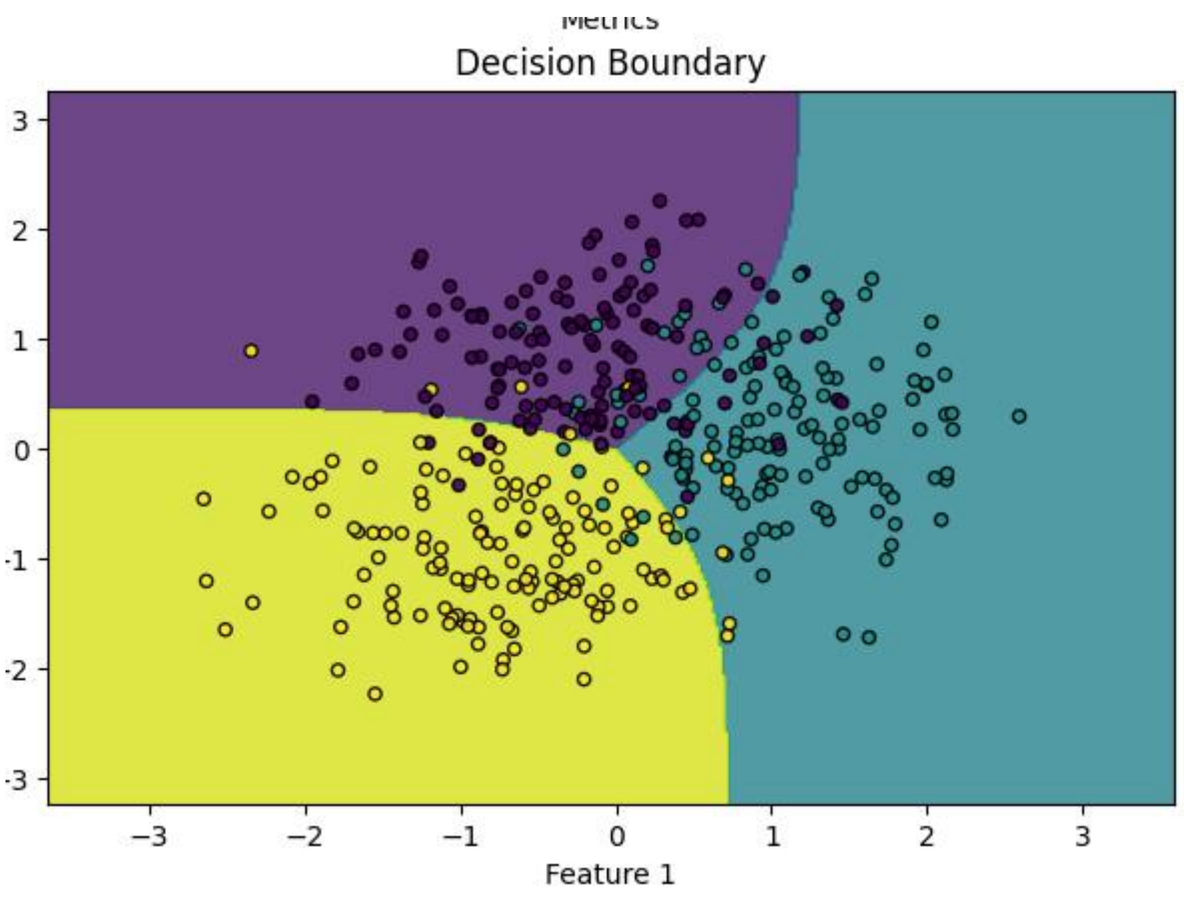


Figure21 (Decision Boundary of Deep PLAN)

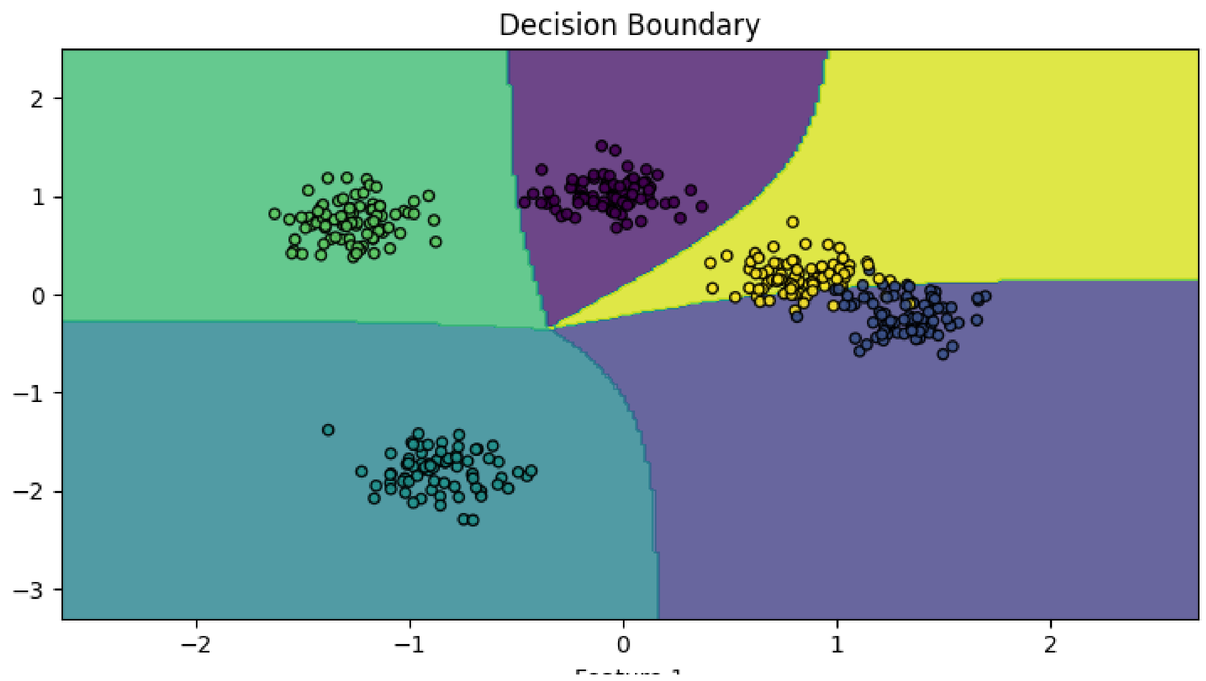


Figure22 (Decision Boundary of Deep PLAN)

Diffirences of other classificiation algorithms:

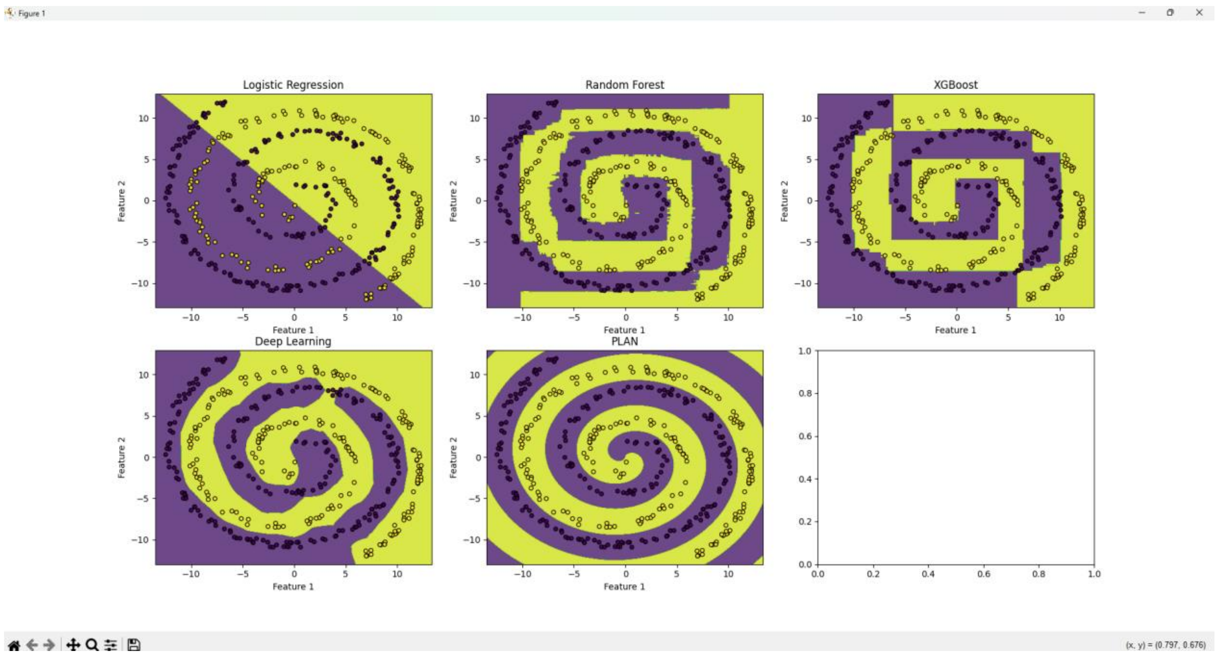


Figure23 (Deep PLAN's Spiral activation on Test dataset)

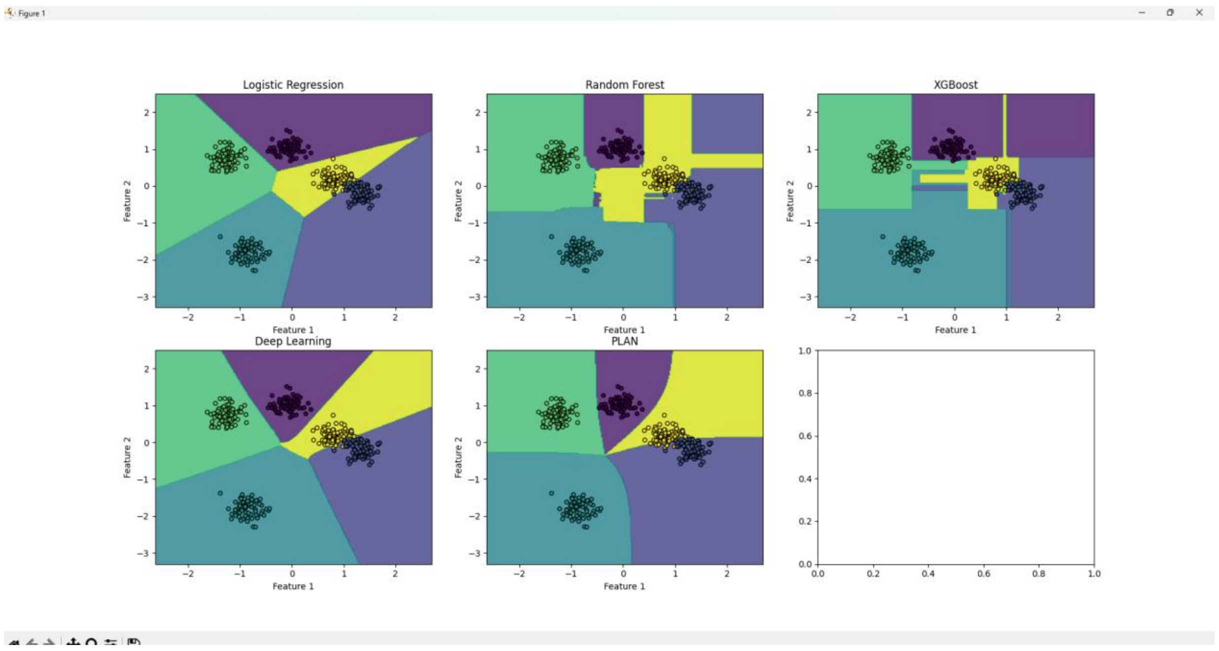


Figure24 (Scikit-learn's blobs dataset)

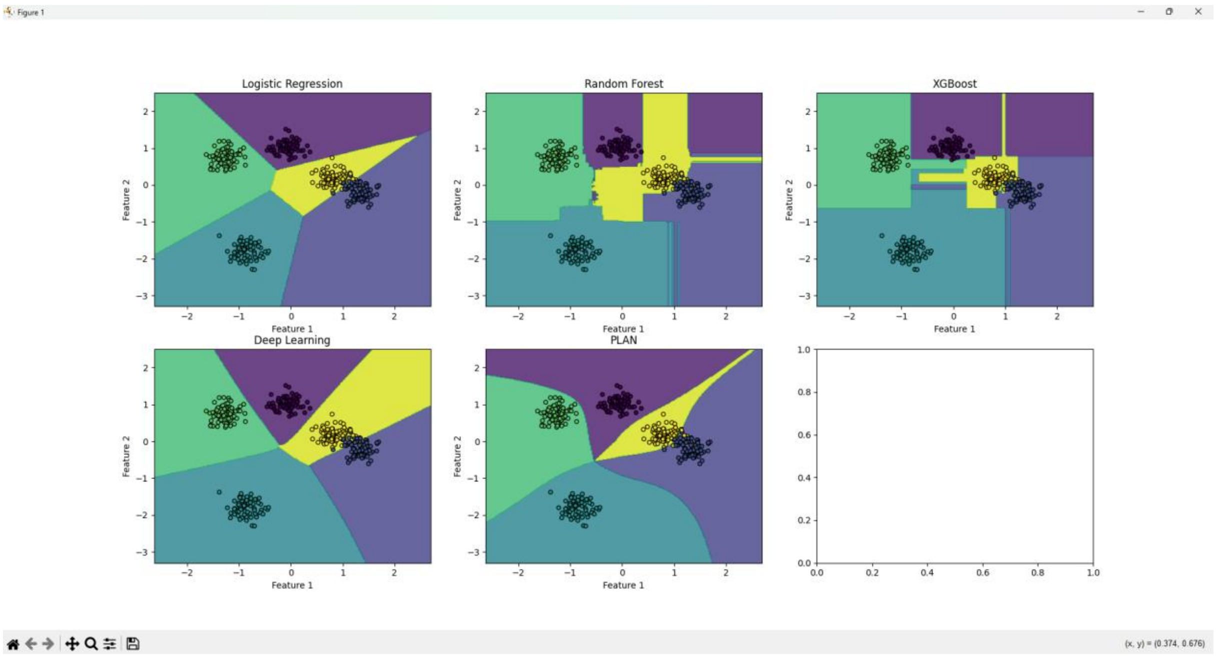
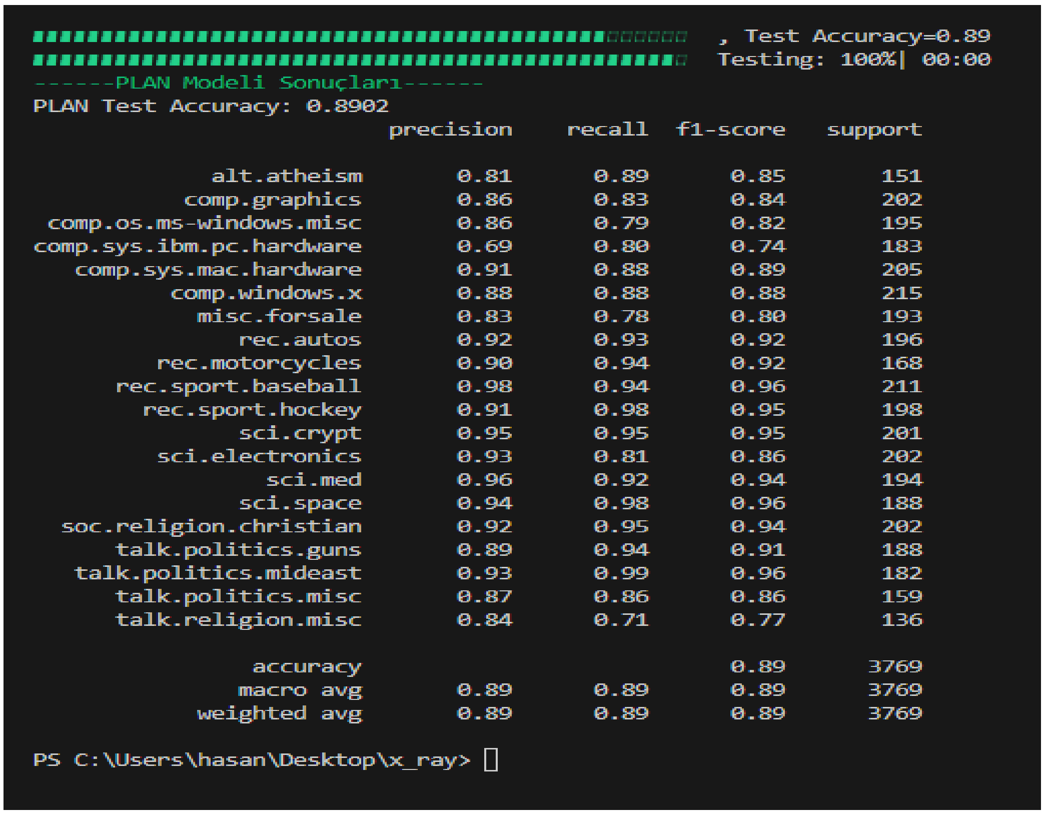


Figure25 (Scikit-learn's blobs dataset with different aggregation layers)
https://github.com/HCB06/Anaplan/blob/main/Welcome_to_Anaplan/ExampleCodes/NLPlan/20newsgroup(20%20categories%2C%200.89%20acc).py



```
███████████████████████████████████████████████▒▒▒▒▒▒▒ , Test Accuracy=0.89
██████████████████████████████████████████████████████▒ Testing: 100%| 00:00
------PLAN Modeli Sonuçları------
PLAN Test Accuracy: 0.8902
                          precision    recall  f1-score   support

             alt.atheism       0.81      0.89      0.85       151
           comp.graphics       0.86      0.83      0.84       202
 comp.os.ms-windows.misc       0.86      0.79      0.82       195
comp.sys.ibm.pc.hardware       0.69      0.80      0.74       183
   comp.sys.mac.hardware       0.91      0.88      0.89       205
          comp.windows.x       0.88      0.88      0.88       215
            misc.forsale       0.83      0.78      0.80       193
               rec.autos       0.92      0.93      0.92       196
         rec.motorcycles       0.90      0.94      0.92       168
      rec.sport.baseball       0.98      0.94      0.96       211
        rec.sport.hockey       0.91      0.98      0.95       198
               sci.crypt       0.95      0.95      0.95       201
         sci.electronics       0.93      0.81      0.86       202
                 sci.med       0.96      0.92      0.94       194
               sci.space       0.94      0.98      0.96       188
  soc.religion.christian       0.92      0.95      0.94       202
      talk.politics.guns       0.89      0.94      0.91       188
   talk.politics.mideast       0.93      0.99      0.96       182
      talk.politics.misc       0.87      0.86      0.86       159
      talk.religion.misc       0.84      0.71      0.77       136

                accuracy                           0.89      3769
               macro avg       0.89      0.89      0.89      3769
            weighted avg       0.89      0.89      0.89      3769

PS C:\Users\hasan\Desktop\x_ray> ▯
```

Figure26 (PLAN 20 Newsgroups Dataset performance)

5.2.1 - Deep PLAN's advantages of other classification algorithms:

5.2.1.1 - Circles Dataset

In a world where obtaining clean data is challenging, it has been demonstrated that Deep PLAN achieves better generalization with less training data compared to other classification algorithms:

For the robustness test against noise, we use the circles dataset available in scikit-learn. Based on the information in the bottom right corner of the first image, the model training results are as follows:

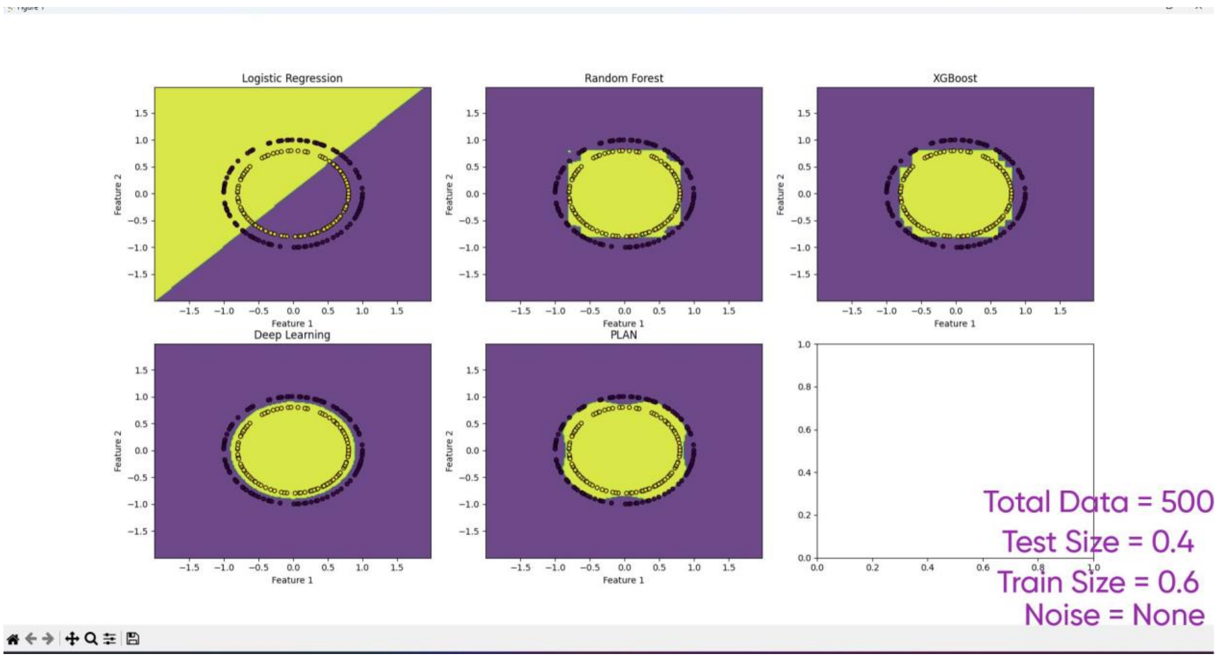


Figure27 (Scikit-learn's circles dataset)

- **Logistic Regression Test Accuracy: 0.4700**
- **Random Forest Test Accuracy: 0.9750**
- **XGBoost Test Accuracy: 0.9900**
- **Deep Learning Test Accuracy: 1.0000**
- **Deep PLAN Test Accuracy: 1.0000**

Both Deep Learning and the Deep PLAN algorithm performed well in this scenario. However, real-world data often contains complex relationships that cannot be separated as cleanly. Therefore, I introduced some noise to the second image and retrained all models without adjusting any hyperparameters. The results for the second image are as follows:

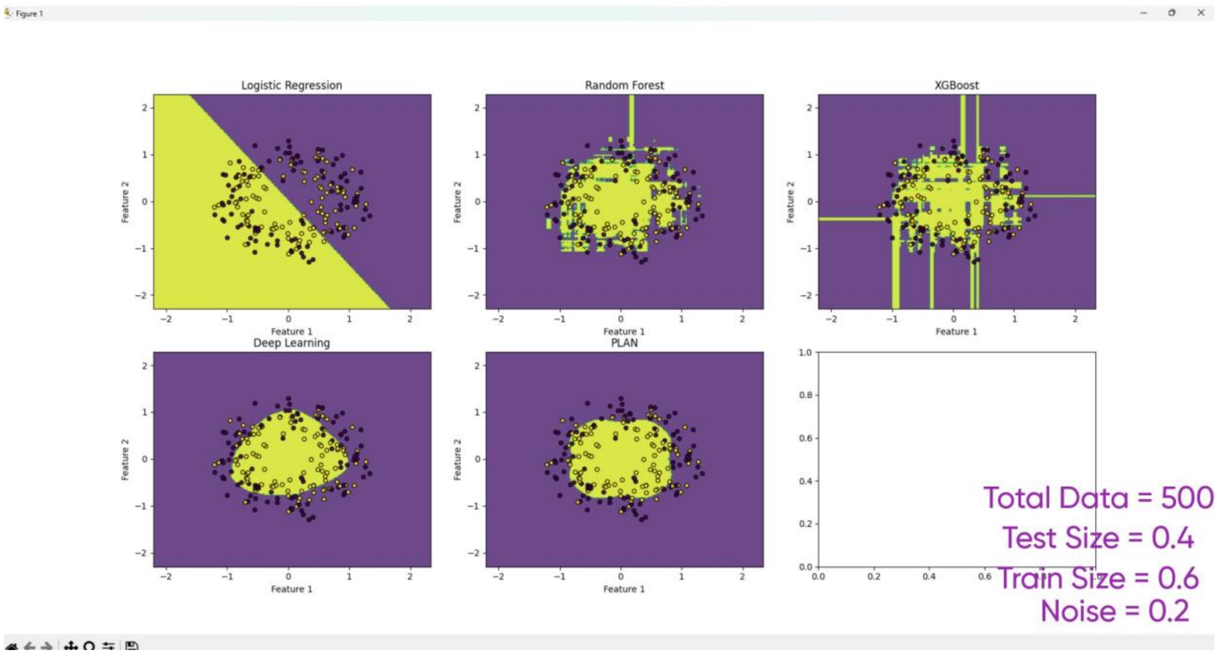


Figure28 (Scikit-learn's circles dataset)

- **Logistic Regression Test Accuracy: 0.5150**
- **Random Forest Test Accuracy: 0.5800**
- **XGBoost Test Accuracy: 0.5650**
- **Deep Learning Test Accuracy: 0.6500**
- **Deep PLAN Test Accuracy: 0.7050**

In this case, Deep PLAN achieved the highest accuracy with the same hyperparameters. For the third image, I increased the noise level to 0.3. The results for the third image are:

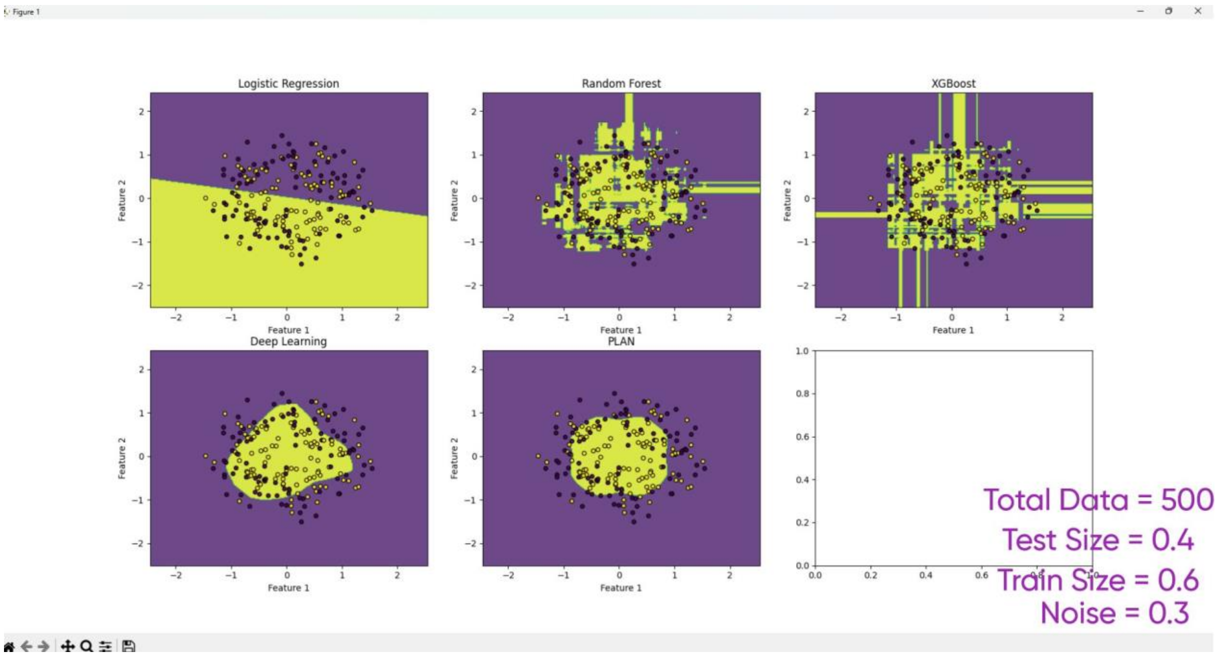


Figure29 (Scikit-learn's circles dataset)

- **Logistic Regression Test Accuracy: 0.5200**
- **Random Forest Test Accuracy: 0.5350**
- **XGBoost Test Accuracy: 0.5200**
- **Deep Learning Test Accuracy: 0.6000 - 0.6100**
- **Deep PLAN Test Accuracy: 0.6500**

The ranking remained unchanged. The ability to adapt to noisy data is commendable. This aspect could potentially simplify the hyperparameter tuning process for model developers.

Speaking of hyperparameters, the deepest model I developed using the PLAN algorithm, which includes a total of 10 aggreagation layers. However, do not confuse these aggreagation layers with hidden layers in deep learning. They require significantly less computational cost. They do not involve matrix-vector multiplications. Therefore, despite performance considerations, this remains the most efficient algorithm on this list. Since PLAN is a type of artificial neural network, tuning its parameters is akin to cooking; a small mistake in seasoning can significantly impact the result. A balanced list of activations, akin to proper seasoning, is essential. Removing or adding one activation can drastically alter accuracy.

In the fourth image, I reduced the total data from 500 to 50. In this scenario, all algorithms except Random Forest and Deep PLAN were eliminated (hyperparameters remained the same as in the first image). The results are:

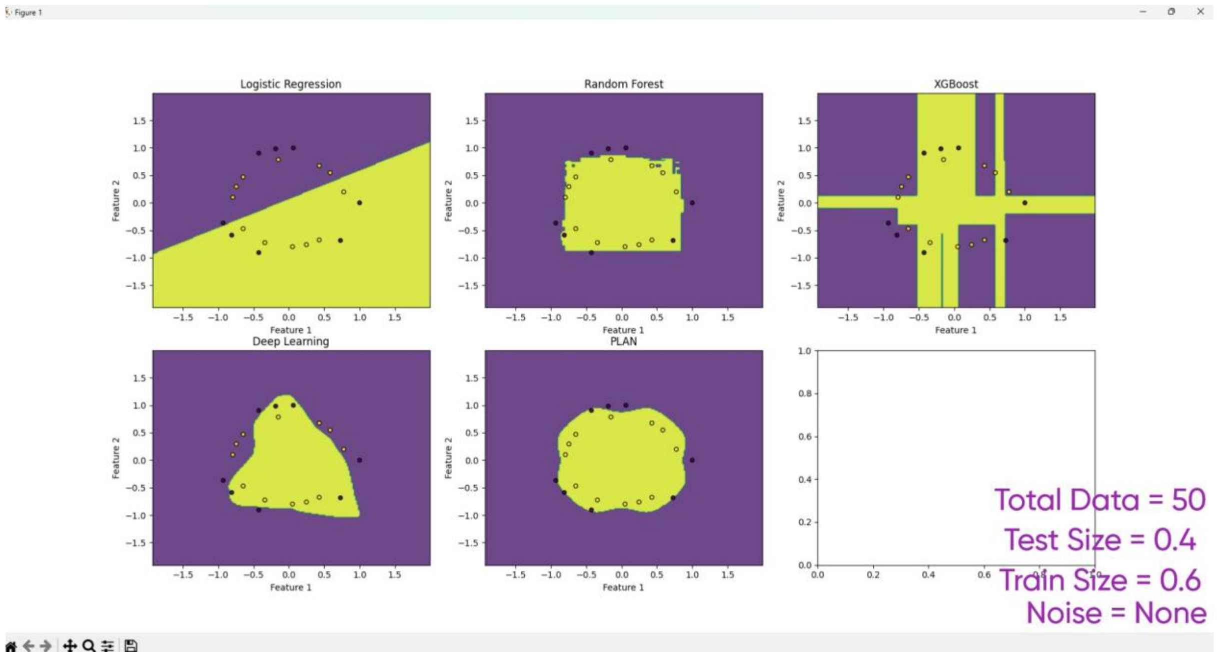


Figure30 (Scikit-learn's circles dataset)

- **Logistic Regression Test Accuracy: 0.5000**
- **Random Forest Test Accuracy: 0.9000**
- **XGBoost Test Accuracy: 0.4500**
- **Deep Learning Test Accuracy: 0.6000**
- **Deep PLAN Test Accuracy: 0.9000**

Random Forest performed well, but it also struggled when the total data was reduced to 25. The results for the fifth image are:

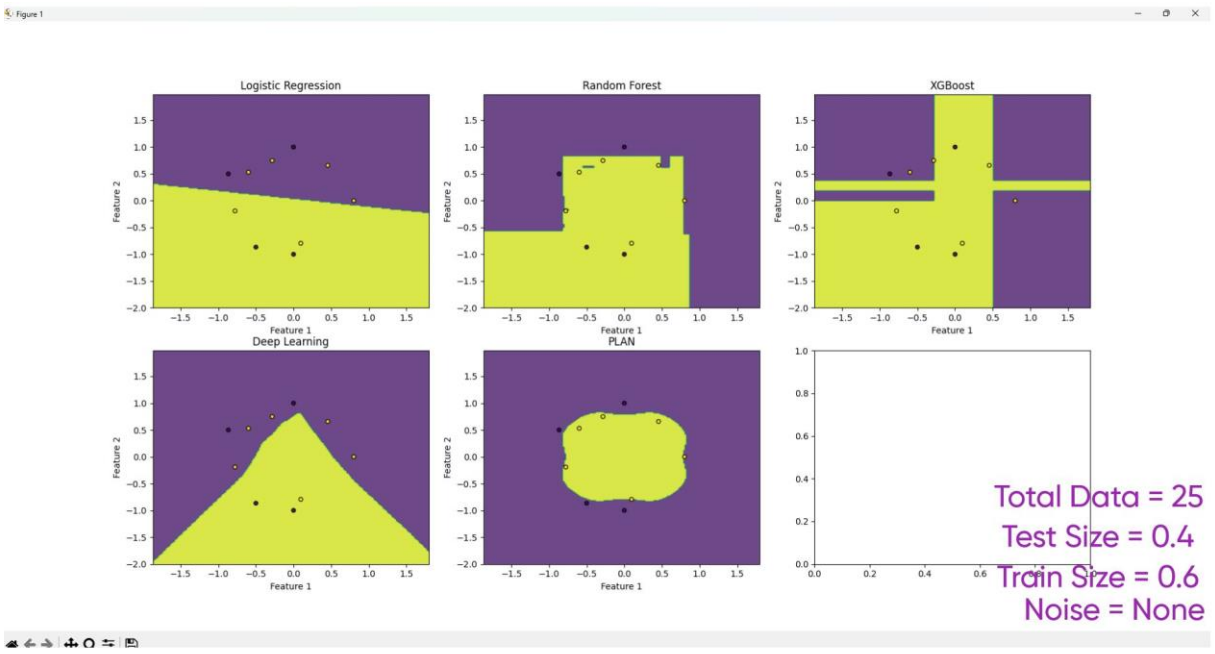


Figure31 (Scikit-learn's circles dataset)

- **Logistic Regression Test Accuracy: 0.4000**
- **Random Forest Test Accuracy: 0.7000**
- **XGBoost Test Accuracy: 0.5000**
- **Deep Learning Test Accuracy: 0.3000**
- **Deep PLAN Test Accuracy: 0.8000**

In the sixth image, I changed the training data to 10% and the test data to 90%, with a total of 500 data points:

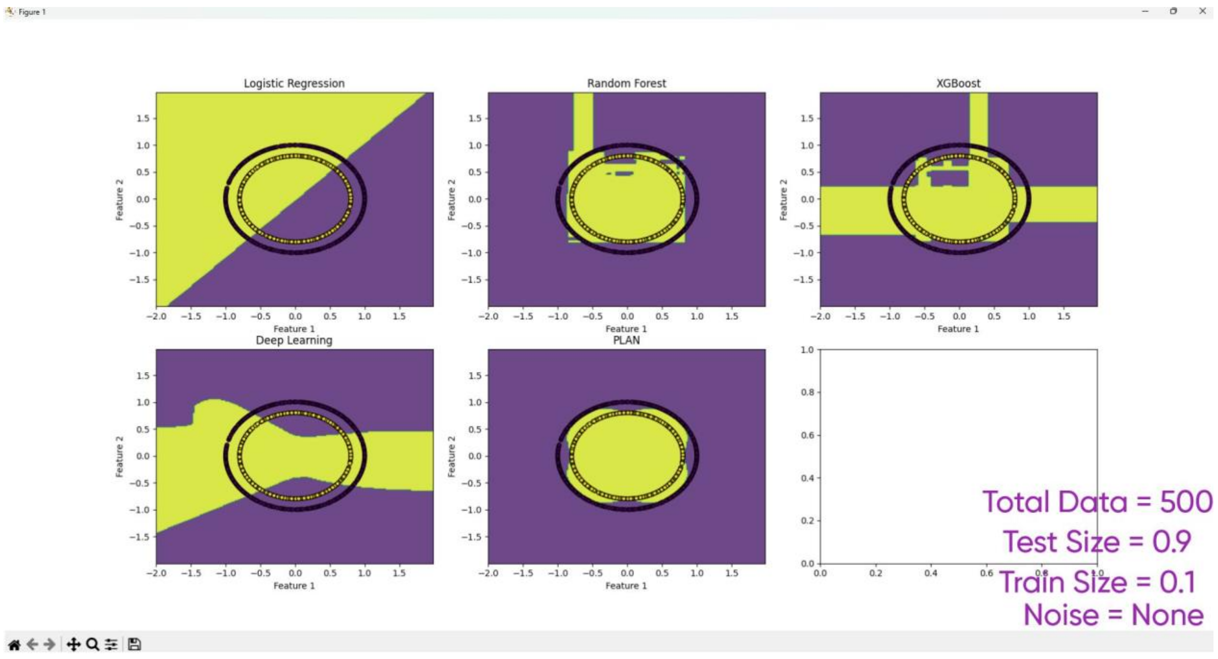


Figure32 (Scikit-learn's circles dataset)

- **Deep PLAN Test Accuracy: 1.0000**
- **Deep Learning Test Accuracy: 0.5089**
- **Random Forest Test Accuracy: 0.8311**
- **XGBoost Test Accuracy: 0.6689**
- **Logistic Regression Test Accuracy: 0.4911**

Code:

https://github.com/HCB06/Anaplan/blob/main/Welcome_to_Anaplan/ExampleCodes/circles(all_algorithms).py

5.2.1.2 - 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, Deep Learning, and Deep 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**
- **Deep Learning Test Accuracy: 0.9000**
- **Deep PLAN Test Accuracy: 0.9000**

Code:

https://github.com/HCB06/Anaplan/blob/main/Welcome_to_Anaplan/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**
- **Deep Learning Test Accuracy: 0.4596**
- **Deep PLAN Test Accuracy: 0.8235**

The dramatic drop in Deep Learning 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.

Deep PLAN models are significantly more robust compared to Deep Learning models.

Additionally, even when adding more layers to Deep 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.

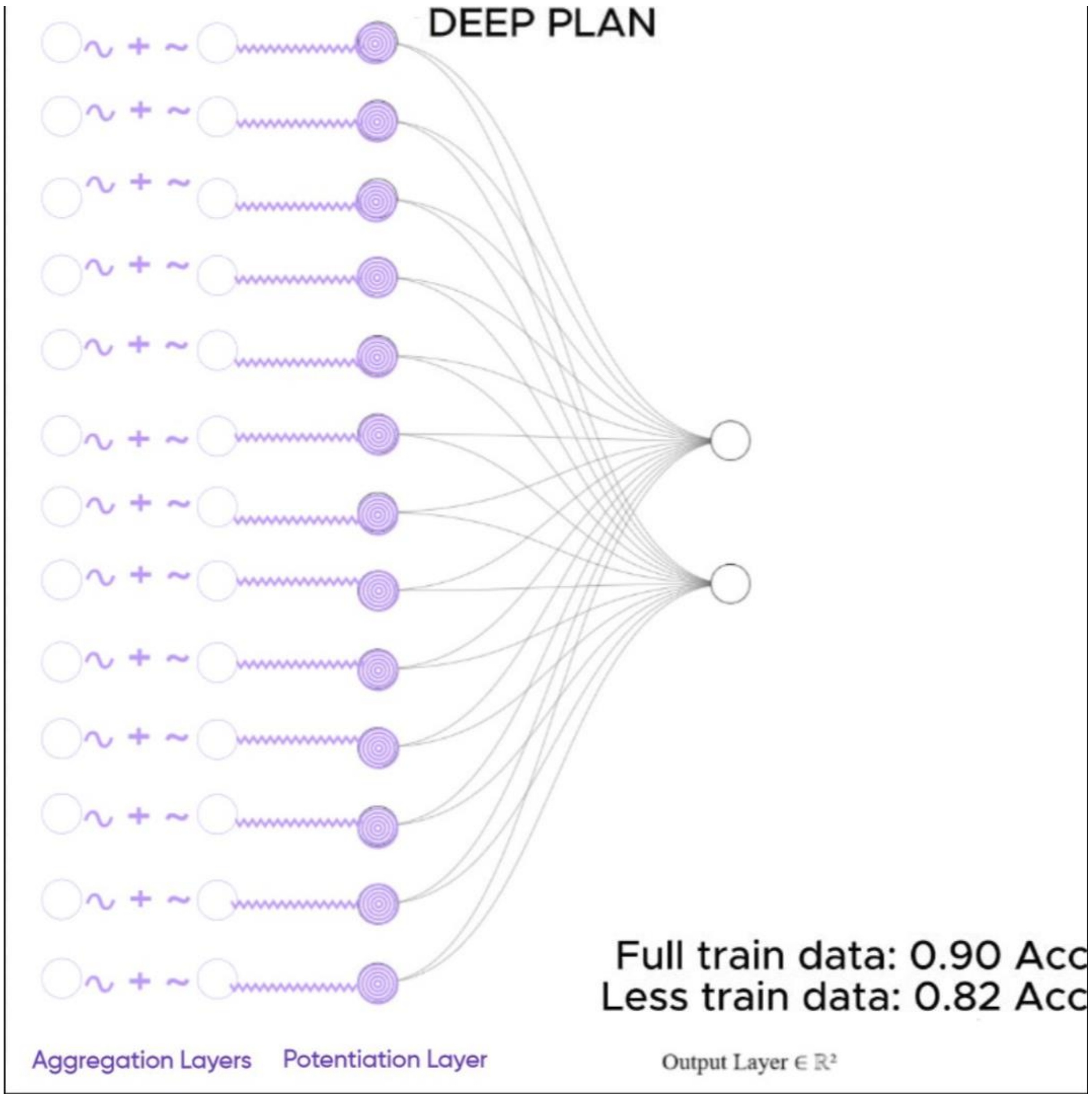


Figure33 (Deep PLAN in Heart Disease dataset)

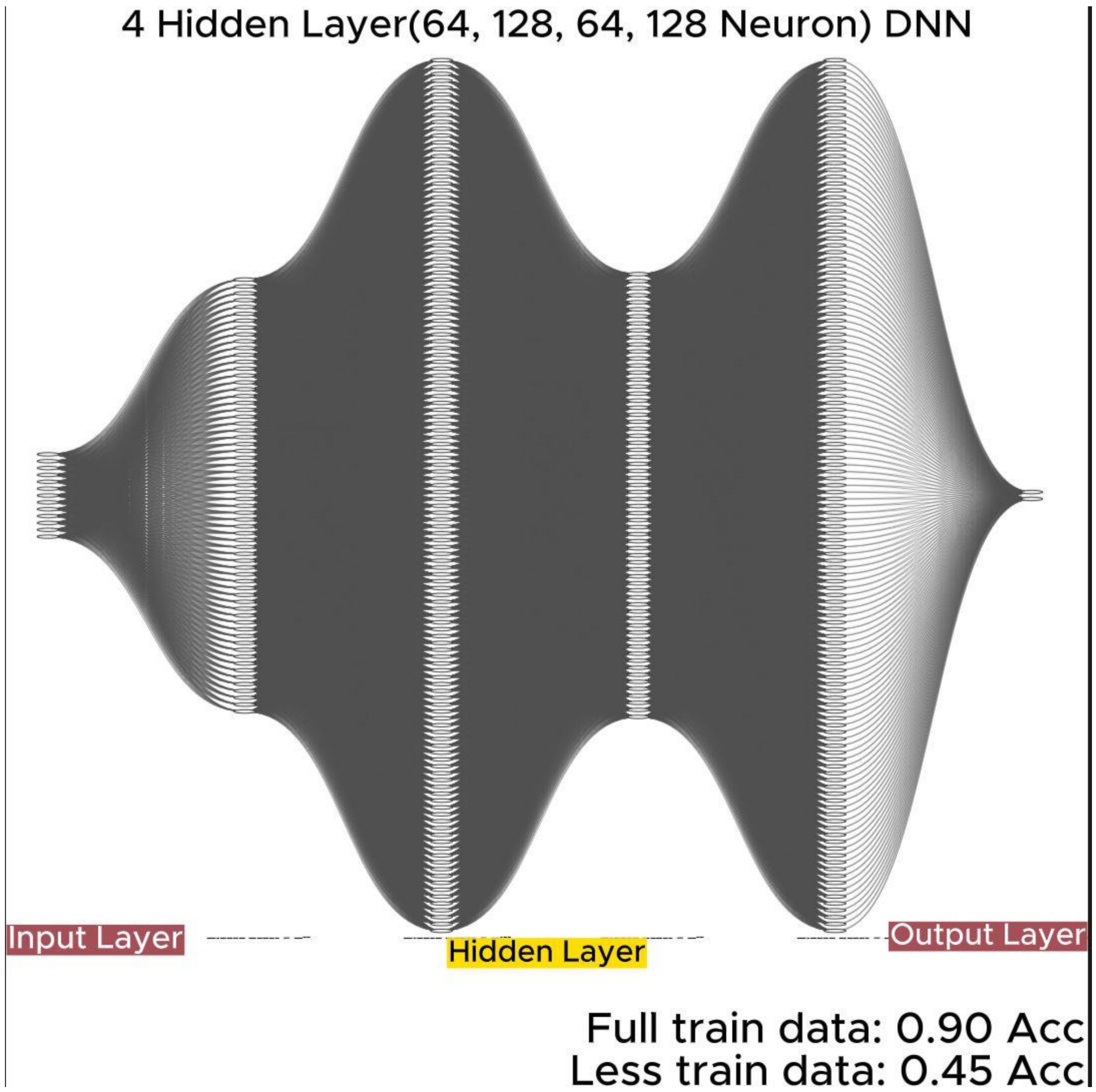

4 Hidden Layer(64, 128, 64, 128 Neuron) DNN
Input Layer
Hidden Layer
Output Layer
Full train data: 0.90 Acc
Less train data: 0.45 Acc

Figure34 (Deep Neural Network(DNN) in Heart Disease dataset)

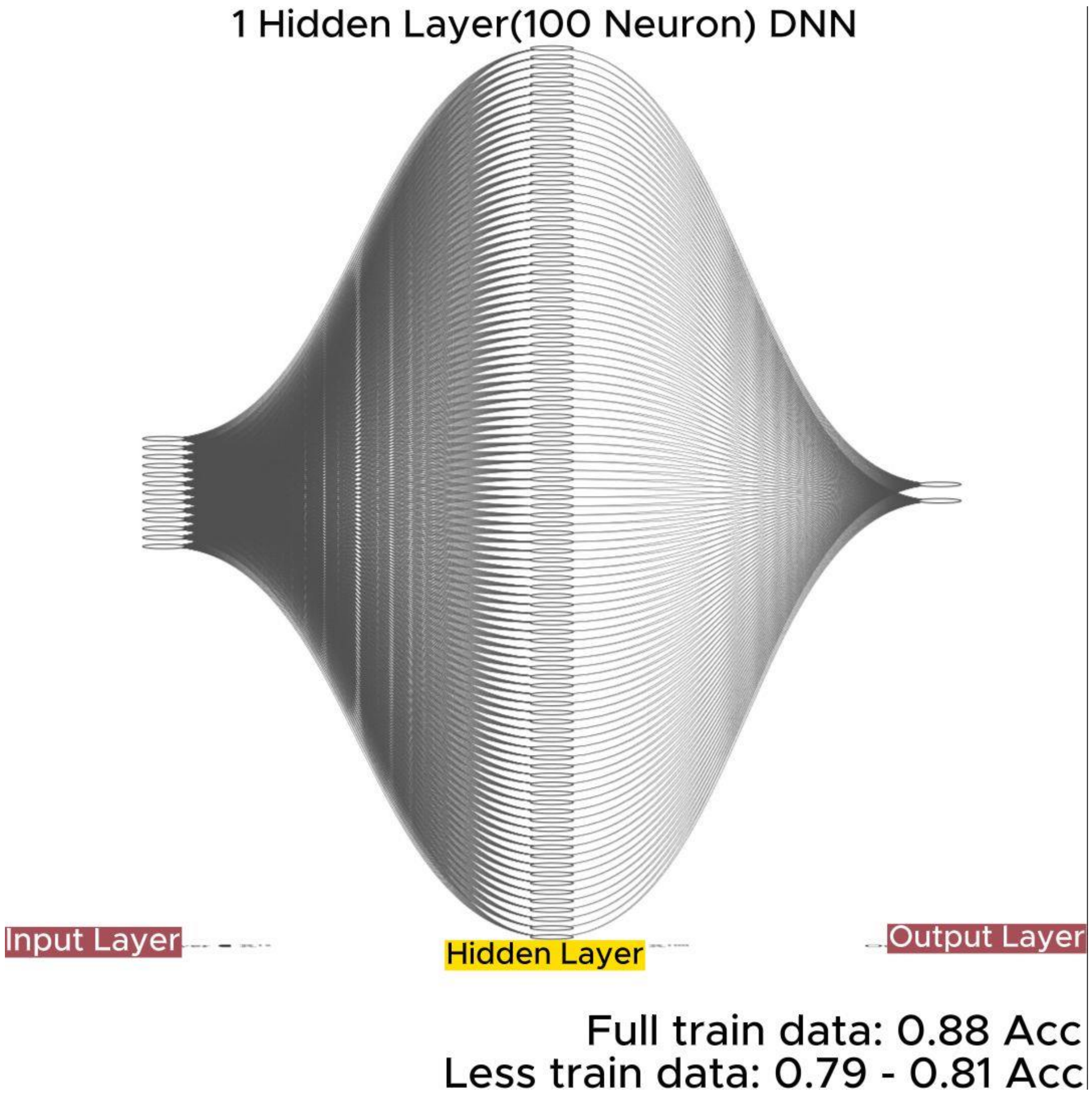

1 Hidden Layer(100 Neuron) DNN
Input Layer
Hidden Layer
Output Layer
Full train data: 0.88 Acc
Less train data: 0.79 - 0.81 Acc

Figure35 (Deep 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.

The Deep Potentiation Learning Artificial Neural Network is as simple as a perceptron yet as powerful as a deep learning algorithm. It is an extraordinary algorithm with the potential to learn deep features through a cumulative learning process, rather than a calculative one, while performing fewer mathematical operations.

For other comparisons: *https://www.linkedin.com/in/hasan-can-beydili-77a1b9270/*

5.3 - New perspective for model training optimization & alternative for Backpropogation: TFL (Test or Train Feedback Learning) Achitecture

I previously mentioned that the PLAN algorithm does not perform any matrix-vector multiplication during training; it learns solely through cumulative connections. The activations forming the aggregation layer facilitate this process. Activations hold a critical role in PLAN models, as different combinations may be required for each dataset, unlike traditional neural networks. In PLAN neural networks, activation functions are the most important hyperparameter. It is essential to find the most suitable and optimal ones for the given training data. This is where the cumulative training process comes into play, allowing for a significant computational efficiency, which in turn enabled me to introduce what might be the **TFL (Test or Train Feedback Learning) most critical and innovative contribution of this paper.** I utilized this efficiency to develop a groundbreaking optimization technique that identifies the activations that maximize the overall accuracy of predictions made on test data while the model is cumulatively trained on training data. This technique not only improves the model's predictions on unseen data but also enhances generalization during training, offering flexibility in the process.

For instance, in a model where high sensitivity (i.e., how confident the model is in its correct predictions) is crucial, the next activation combination can be selected to minimize the loss of predictions on test data after cumulative training with training data. Alternatively, if the primary goal is simply to classify the classes accurately, one can select activations that maximize the direct test accuracy. This approach is exclusive to the PLAN algorithm, as PLAN's performance can be optimized on test data after cumulative training. While only the training data undergoes cumulative learning, the activations that the data will transform through are determined based on either accuracy or loss values on the test data, making this a revolutionary training strategy with the potential to break new ground in the field.

The advantages of this approach are clear: the flexibility it offers allows for the rapid adaptation of powerful, targeted models to project requirements in the business domain. Additionally, through hyperparameter optimization, it enables a non-hyperparameter model training process. Moreover, it has the potential to automatically enhance the model's generalization capacity based on feedback from the test data, making it a highly innovative and promising technique.

TFL can be considered a type of **genetic optimization algorithm**. This is because it involves an optimization process where successful activation functions pass on their "genes" to the next generation, facilitating the transfer of advantageous traits.

In TFL, we are not limited to using only test feedback; we can also use training data, which would then be referred to as Train Feedback Learning. The key distinction here is that TFL for DPLAN(Deep Potentiation Learning Artificial Neural Network) offers numerous options, whereas this flexibility is not possible in networks that rely on backpropagation.

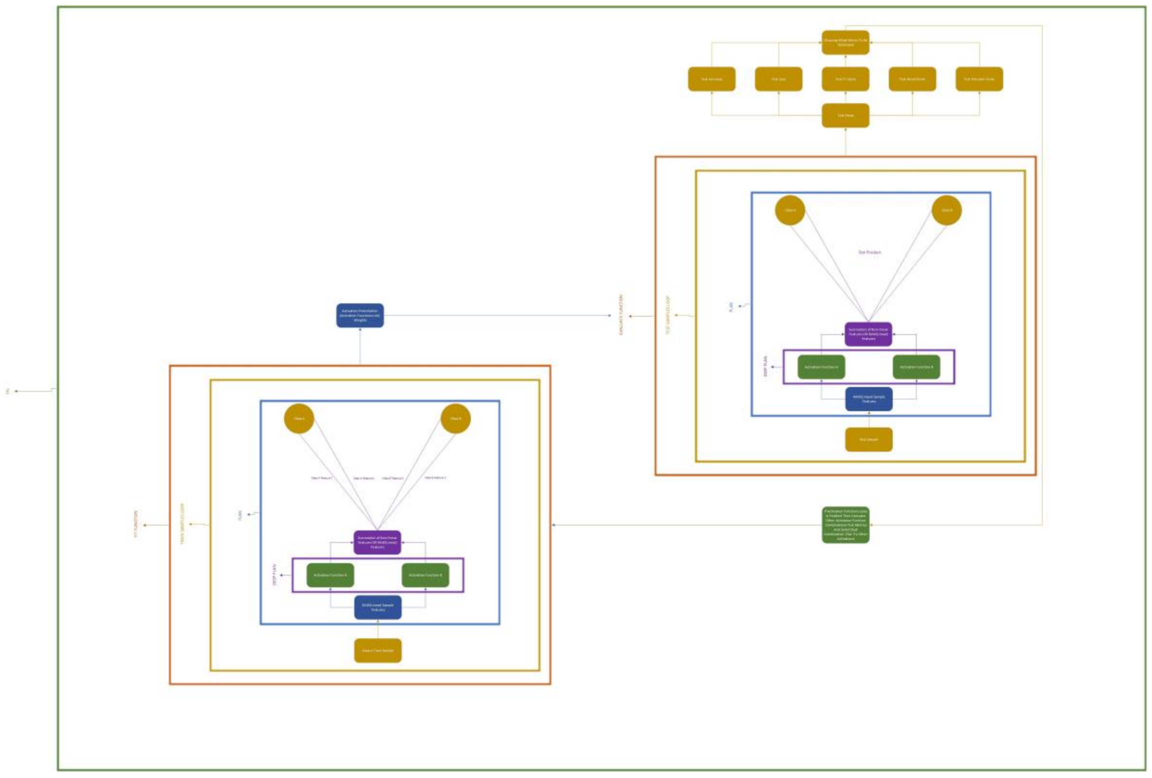


Figure36 TFL Architecture

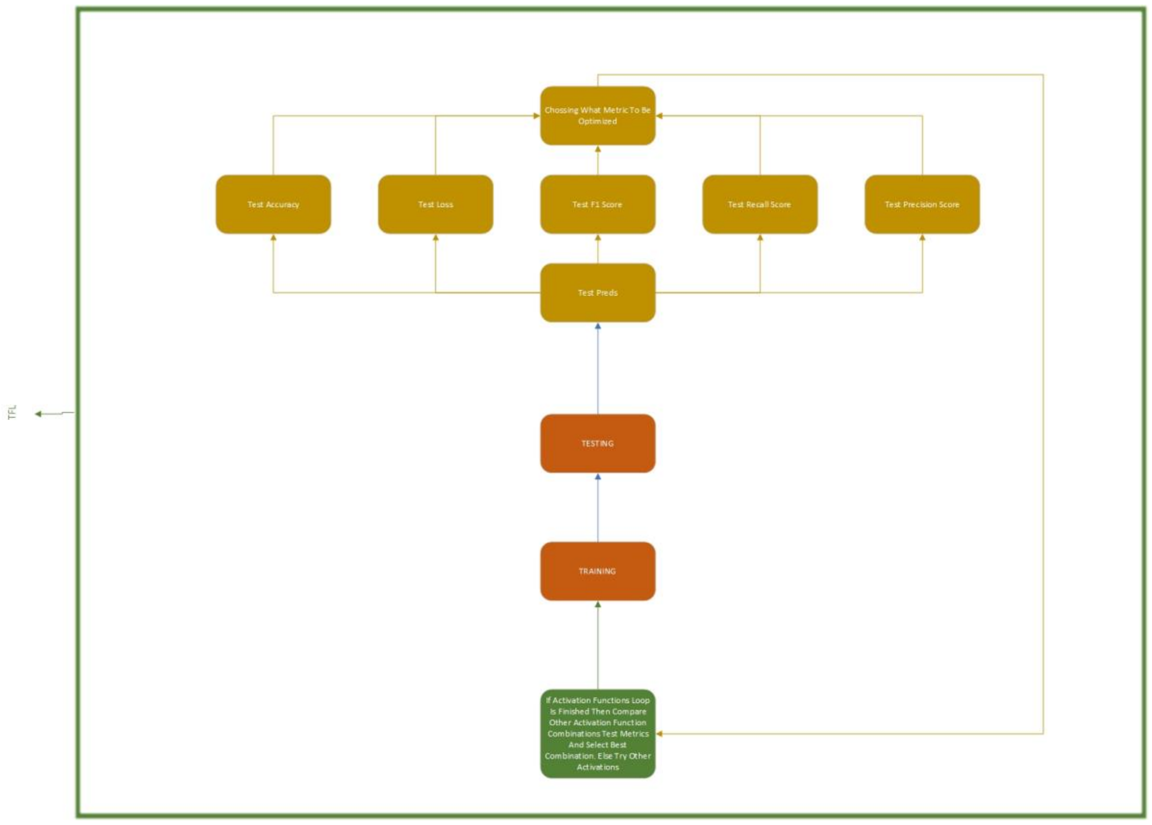

Choosing What Metric To Be Optimized
Test Accuracy
Test Loss
Test F1 Score
Test Recall Score
Test Precision Score
Test Preds
TESTING
TRAINING
If Activation Functions Loop Is Finished Then Compare Other Activation Function Combinations Test Metrics And Select Best Combination. Else Try Other Activations
TFL

Figure37 TFL Architecture (Simplified)

The fact that the weight parameters in PLAN models are not directly learnable is an advantage.

In classical approaches, weight parameters require computational optimization architectures, with one of the most widely accepted being the backpropagation algorithm. However, in PLAN models, the learnable parameters are not the weights but the activation functions. As a result, instead of a computational optimization architecture, **a combinatorial optimization architecture** was needed, since we are dealing with a finite solution set. At this point, I developed TFL. TFL is such a perfect algorithm that it not only optimizes training but also enhances the model's generalization capacity, offering virtually unlimited flexibility.

TFL selects activation functions to be applied to the training data's features, based on the metric that best optimizes predictions on the test data.

During training, the PLAN algorithm adds the sum of these activation transformations to the relevant class connections of the weight matrices, meaning that we indirectly optimize the weights. However, the weights are optimized according to the feedback from the test data during the testing phase.

It is not possible to use TFL in other neural network architectures because in traditional approaches, the model is only optimized by reducing the loss value on the training data. Even if it could somehow be implemented, it would not be as effective as PLAN because PLAN does not perform any matrix-vector multiplication operations during training. The training data is directly passed through the activation functions, and the sums of these activations are cumulatively added to the corresponding class label's row. This is the learning method of the classic PLAN algorithm. Therefore, even if applicable, it would be much slower compared to PLAN.

At the core of PLAN lies combinatorics, not calculus. Understanding this is crucial above all else. And this is an advantage that cannot be overstated. We can write and integrate our own activation functions, meaning the possibilities are limitless. I'm not even mentioning the other advantages that PLAN offers on its own. The single weight matrix itself is already a major advantage.

From a business perspective, in scenarios where the cost of false positives is higher (such as marking legitimate emails as spam), we may choose to maximize the Precision metric. Conversely, in situations where the cost of false negatives is higher (such as ensuring accurate diagnoses in cancer detection), maximizing the Recall metric would be more appropriate. For these specific cases, PLAN's TFL offers a unique algorithm that provides flexibility in adapting to the nature of the problem.

While Backpropogation employs simple *artificial cognition*, TFL, which I developed for DPLAN, represents a *artificial metacognition*.

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: weights * input layer

For input prediction: output layer * weights

```
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]
```

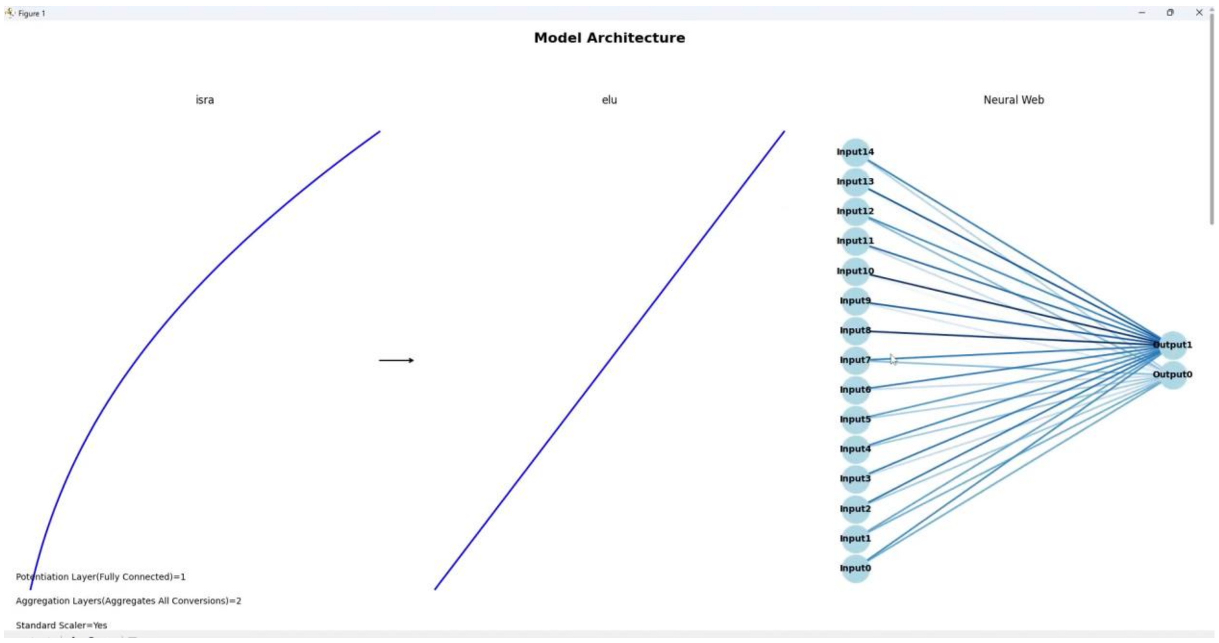
```
This is achievable in PLAN because, in PLAN, each row in the weight matrix
represents a class.
```

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.

XAI (Explainable Artficial Intelligence) Achievements

XAI (Explainable Artificial Intelligence) is an active and rapidly evolving research area focused on the transparency and interpretability of AI models.

In PLAN models, we can easily visualize which feature has a greater impact on which class. While this can also be understood in a model without hidden layers, such as logistic regression, the distinction of PLAN lies in its ability to maintain a single fully connected layer, regardless of how deep the network becomes. This allows us to comprehend complex problems that logistic regression cannot solve, in a simpler manner. Thus, PLAN is as simple as logistic regression but possesses the capability to learn complex relationships akin to deep learning. Leveraging this advantage could lead to a significant breakthrough in the field of Explainable Artificial Intelligence (XAI).



Potentiation Layer(Fully Connected)=1
Aggregation Layers(Aggregates All Conversions)=2
Standard Scaler=Yes

Figure38* Example *Model Architecture (Each connections represents features)

Potentiation Learning Artificial Neural Network Library: Anaplan

To facilitate broader experimentation and practical application, I have encapsulated the Potentiation Learning Artificial Neural Network architecture in a Python library called "Anaplan." 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.

For those interested in exploring or utilizing this framework, the library and the GNU Octave codes referenced in this paper are hosted on GitHub. You can find them at: *GitHub - Anaplan*. 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*.

Integration with Existing Architectures

One of Potentiation Learning Artificial Neural Network's strengths is its flexibility to integrate with existing artificial neural network architectures. For instance, instead of using a traditional fully connected layer at the end of a Convolutional Neural Network, the Potentiation Learning Artificial Neural Network framework's Potentiation Layer. This approach not only leverages the interpretability and efficiency of Potentiation Learning Artificial Neural Network but also enhances the overall model by replacing complex layers with more manageable components.

Environmental Impact

Due to its non-iterative learning process, Potentiation Learning Artificial Neural Network's offers a significant advantage in terms of computational efficiency. In scenarios where Potentiation Learning Artificial Neural Network's is widely adopted, its streamlined approach to training can lead to a substantial reduction in the computational resources required. This efficiency has the Potentiation to positively impact global warming by lowering the energy consumption associated with large-scale neural network training. By promoting a more sustainable method for training Artifical Intelligence models, Potentiation Learning Artificial Neural Network's could contribute to reducing the environmental footprint of Artificial Intelligence technologies.

6 - Advantages and Disadvantages:

Advantages:

1. Faster training times. (cumulative learning)
2. Easy to learn and implement. (no calculus)
3. Easy to maintain and update. (non-blackbox)
4. It occupies less memory space. (only can have one weight matrix)
5. Multi-functional single model architecture. (cumulative learning)
6. Energy efficency. (less mathematical operations)
7. Strong generalization even small training splits. (TFL)

Disadvantages:

1. In early research stage

7 - Conclusion:

The Potentiation Learning Artificial Neural Network architecture, through its innovative approach to neural network design, offers a compelling alternative to traditional methods. Its Potentiation for faster, more interpretable, and environmentally friendly learning processes 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/Anaplan/tree/main/Welcome_to_PLAN/Codes

https://github.com/HCB06/Anaplan/tree/main/Welcome_to_Anaplan/ExampleCodes

Potentiation Learning Artificial Neural Network library:

https://github.com/HCB06/Anaplan