PLAN
(Pruning Learning Artificial Neural Network)

Author: Hasan Can Beydili

22250101008@subu.edu.tr

Abstract

No parameter optimizations, no gradient descent, no backpropagation, no fixed activation functions, no error functions, no loops, no calculus, no hyperparameters, no bias and no matrix multiplications for training. Yet, faster training and promising results.

Allow me to introduce PLAN, in this research, I have developed an architecture that holds a high potential for solving the energy consumption problem in artificial intelligence. a new artificial neural network architecture designed to address classification problems typically tackled by conventional ANNs. Inspired by the brain's synaptic pruning process, which occurs during sleep and removes unused synaptic connections to make room for stronger ones, PLAN adapts this biological mechanism to the digital realm. In this approach, the weights in our network are analogous to synapses, and pruning involves setting specific weight values in the matrices to zero. What do the connections that are not pruned represent? The unpruned connections represent the inputs we will provide.

The challenge lies in determining which weights should be pruned. To address this, I developed a flexible activation process known as "**activation potential**". Instead of using fixed activation functions, activation potential provides a dynamic and adaptable activation process. The results of applying these steps are impressive. 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. We just need linear algebra.

This article begins with an introduction to the architecture and proceeds to discuss the underlying concepts and mathematics of the PLAN algorithm.

All written, visual, and conceptual work in this article belongs to the author.

All the codes in this article:
https://github.com/HCB06/PyerualJetwork/tree/main/Welcome%20to%20PLAN/Codes.

All gif's in this article:
https://github.com/HCB06/PyerualJetwork/tree/main/Welcome%20to%20PLAN/Gifs

1 – Introduce

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 PLAN, 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.

This algorithm is divided into two main approaches: "**binary injection**" and "**direct injection**".

Binary Injection: This method utilizes the activation potential to extract binary features. It encodes features in a binary format, which allows for a more customizable feature extraction process. By leveraging the activation potential, binary injection can fine-tune how features are represented and utilized within the network, providing flexibility in adapting to different tasks or datasets.

Direct Injection: In contrast, the direct injection approach encodes inputs directly into the connections leading to specific neurons without modifying them. This method eliminates the need for hyperparameters, making it more user-friendly. Additionally, because it encodes features in their raw form, direct injection can achieve higher accuracy on certain datasets by preserving the integrity of the input features.

Each approach has its unique advantages. Binary injection offers greater customization and adaptability through its binary feature representation, whereas direct injection is simpler to implement due to the absence of hyperparameters and can be more effective in scenarios where maintaining the original form of features is crucial for achieving high performance.

Before diving into the details, let's explore the foundational technique behind PLAN and how it diverges from conventional neural network approaches.

2 - Concepts:

a. Fex Layer (Feature Extraction Layer):

The Fex Layer acts as an intermediary between the input and output layers in the PLAN architecture. Its primary function is to extract relevant features from the input data based on a specified activation potential threshold. This threshold, which ranges from 0 to 1, determines which input features are significant enough to be propagated forward. Only features with values above this threshold are retained for further processing.

b. Activation Potential:

Activation potential is a dynamic value used to control the flow of information through the Fex Layer. When the activation potential is set to 0.5, for example, all input weights with values less than 0.5 are set to zero, effectively pruning less significant connections. This mechanism allows for flexible and adaptive feature selection, enhancing the efficiency of the network by focusing only on the most pertinent inputs.

c. Cat Layer (Catalyst Layer):

The Cat Layer serves as the output layer in the PLAN architecture, following the Fex Layer. It plays a crucial role in finalizing the prediction by applying an additional thresholding process. Specifically, any connection from the Fex Layer with a weight set to zero is ignored in the Cat Layer, refining the output by considering only the most significant features extracted by the Fex Layer.

In PLAN, 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 PLAN demonstrates its unique capabilities.

---

3 – 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).

Let’s first consider that we have chosen the number of neurons in the fex layer as 10 for our input vector (784), then our weight matrix should be (10,784) (10 = rows, 784 = columns). In this scenario, the weights affecting the first neuron in the fex 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 **“feature extraction 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.

The programming language to be used in the article is **GNU Octave**, which has MATLAB syntax.

```
weights1 = ones(10,784); % Matrix formed by ones

weights2 = eye(10); % Diagonal matrix

% Brief: 10 is class_count.
```

We define the weight for fex layer(weights1) 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. The catalyst matrix, therefore, should be structured as a square diagonal matrix with dimensions corresponding to the number of classes. Our weights2 is now trained. We dont need to change anything for weights2.

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 with binary injection:

```
inputLayer = normalization(inputLayer); % inputs in range 0 - 1
activationPotential = 0.5

%% FEATURE EXTRACTION LAYER (BINARY INJECTION) %%
  inputLayer(inputLayer < activationPotential) = 0
  inputLayer(inputLayer > activationPotential) = 1
  weights1(class,:) = inputLayer;
```

$$\text{If } |(\mathbf{W}'_c)_{i,j}| < \theta, \text{ then } (\mathbf{W}'_c)_{i,j} = 0.$$

with direct injection:

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

%% FEATURE EXTRACTION LAYER (DIRECT INJECTION)%%

  weights1(class,:) = inputLayer;
```

Main difference of binary injection and direct injection is in the feature extraction layer. In the pattern learning process of the feature extraction layer for binary injection, we first find the index values of all pixel values in the input layer that are smaller than the activation potential. These index values are assigned to the variable 'inputConnections using the 'find' method available in Octave. Then, we set all columns of the rows in the weight matrix of our feature extraction layer, denoted as 'weights1', at those indices to '0' this mean this connections just pruned. I called this ***"Use it or lose it rule." This is because in weight matrices, all columns in a single row correspond to a neuron after multiplication, and all rows in a single column are connected to a single pixel in the input layer***. This way, we store the desired features in directly (because its multplying 1's) the row intervals of our weight matrices that we defined in the previous step with certain limits. In direct injection, all the connections of a neuron, selected based on the class variable, are directly designated as the input layer. Here is an example of a feature extraction matrix that has learned features for 2 classes:

```
[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.
```

Next comes our **'catalyst layer'**. I named this layer 'catalyst' because its role is to further enhance the dominant side of the information passing through the feature extraction layer in the network, thus speeding up the process. The layer that directs the information acquired up to that layer like placing the pieces of a puzzle into their respective classes and says "you belong there." The majority of what we call actual learning lies in this structured distribution, which I discovered by experimenting with this distribution. **We actually dont need use catalyser layer for training. Its just using for make predictions.**

```
fexLayer = normalization(fexLayer); % fex neurons in range 0 - 1

%% CATALYST LAYER (BINARY AND DIRECT INJECTION)%%
  catLayer = (weights2 * fexLayer);

%% NOTE: CATALYSER LAYER NOT NECESSARY FOR TRAINING. IT USING FOR MAKE
PREDICTIONS.
```

During the making predictions of the catalyst layer, we make all neurons that meet a certain condition or have a '0' value, which is referred to as zero similarity, in the neurons resulting from the multiplication of our inputs with the feature extraction layer, equal to '0'. We do this for all rows corresponding to the column indexes of the second weight matrix, which is identified as the catalyst layer weight matrix, **as remember in matrix-vector multiplication, columns are important for previous connections, while rows are important for connections that will occur after multiplication**. Weight matrix belonging to a catalyst layer containing typical 3 class properties:

```
 [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 catalyst layer should typically be in this "diagonal form"**. If you remember, we said that the reason for this is that the information coming from the previous layer searches for the neuron that belongs from top to bottom. Rows represent neurons.**

As previously discussed, the catalyst matrix should be represented as a square diagonal matrix with dimensions equal to the number of classes.

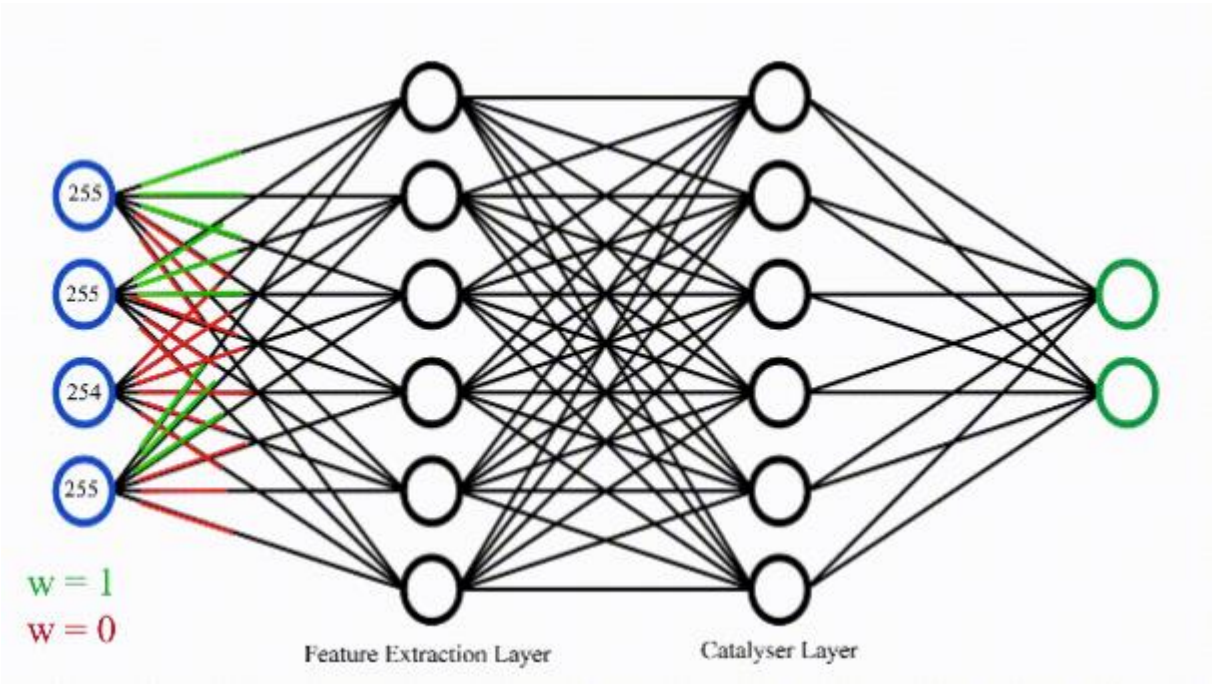
Let's imagine a scenario where the catalyzer layer is directly multiplied with the input without the feature extraction layer to better understand the catalyst layer:

```
% The code we've written above is valid only within this code block.
% x = Pixel

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

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

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

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

There is no restriction on columns in the feature layer. In other words, the feature extraction layer is more flexible and only classifies by rows, while the catalyst layer is more rigid and requires both row and column arrangements. It is a matrix where each row group emphasizes the features belonging to a class. Later, when training data points belonging to a new class, it will be noticed that some neurons are more triggered by certain features with newly added data. We will discuss matrix merging and summation operations when training data points belonging to the same class. Also our 'catLayer' represents our network's output layer.

The underlying logic here is that by assigning zeros to the other rows, their outputs will be '0' regardless of the input. Since the weight rows going to the other classes are assigned as '0', all other classes will already be '0'. For the 1st class, the output will always be '1', regardless of the input. Thus, the desired outcome is automatically achieved. 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.** Congratulations, we have now trained a data point with a '0' error as quickly as possible. But its not over.

Here is my example of all this steps 2 classes and if have more then 2 neurons of fex layer PLAN and there are no input normalization and activationPotential = 255 and added one more layer for output layer (But not necessarily, cat layer already doing this.)



Gif link: *https://github.com/HCB06/PyerualJetwork/tree/main/Welcome%20to%20PLAN/Gifs*

Note: if input normalization is applied, then 255+ is no more, the nodes will be for fex = in range 0 – 1 and will be for cat = 0 or 1.

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:

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

  weights1 = sprintf('weights/weights1.mat');
   save(weights1, 'weights1');

  weights2 = sprintf('weights/weights2.mat');
   save(weights2 , 'weights2');

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.]]

  newWeights1 = weights1;

  weights1 = sprintf('weights/weights1.mat');
    load(weights1);

  weights1 += newWeights1;

  weights1 = sprintf('weights/weights1.mat');
   save(weights1, 'weights1');


end
```

Note: if we think 1 labeled data sample is our first training sample.

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. This calculations mathematical formation is:

$$\mathbf{W}_{\text{final}} = \sum_{c=1}^{C} \mathbf{W}'_c \times N_c$$

```
%% 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]
```

At the beginning of the text, we were talking about synaptic pruning. When training new data belonging to the same class, we strengthen the connections, meaning we add +1 to the values of the pixels in that class's rows. In the feature extraction layer, for indices with the same pattern, we add +1, while for indices with a new pattern, they reflect as '1' instead of '0'. 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

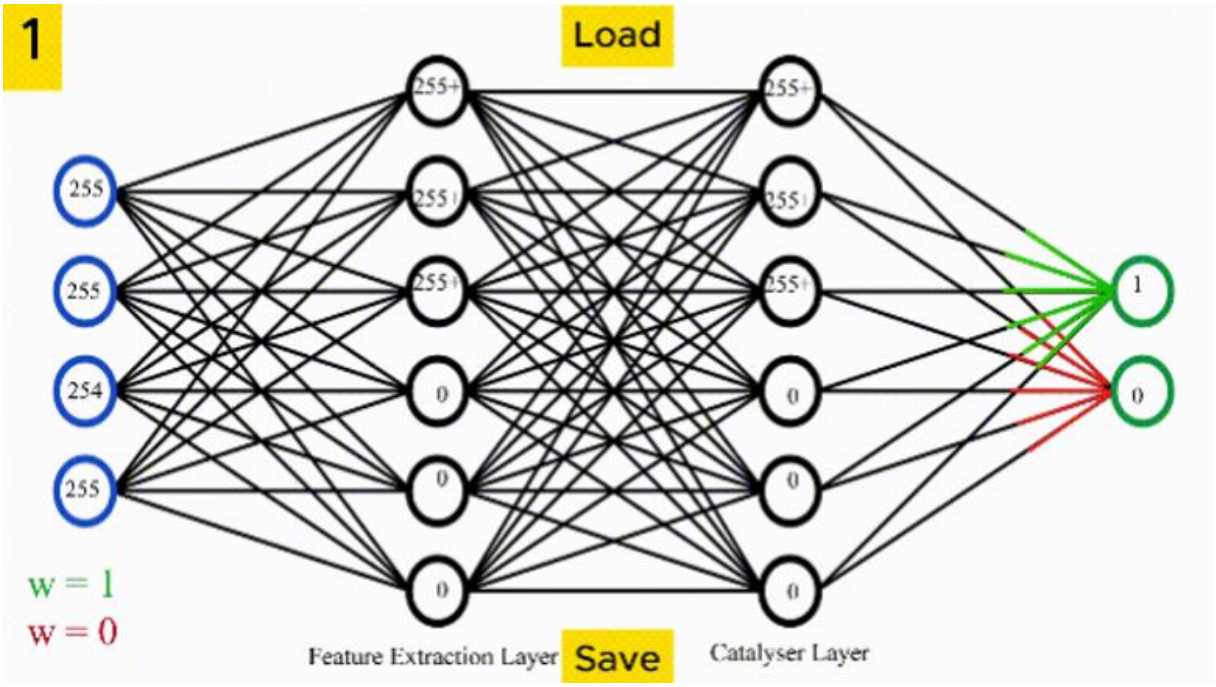
model. **Because this architecture not only resembles synaptic pruning in biological neural networks but also resembles how biological neural networks store learned information in memory**, the presence of similar features across multiple classes can dull the model's generalization ability and cause confusion, just like in humans. Humans learn best what they repeat the most. Just like in humans, the more these connections are strengthened (+1, +1, +1), 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 and determining good feature thresholds**. For example, if we change the threshold value we set as < 1 in the feature extraction layer to < 0.1, we can predict more data examples correctly.

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.
```

All steps for two classes neural network:



Gif link: *https://github.com/HCB06/PyerualJetwork/tree/main/Welcome%20to%20PLAN/Gifs*

4 - Test:

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

```
%% WEIGHT INITIALIZATION: %%

  weights1 = ones(10,784);

  weights2 = eye(10);

%% SET ACTIVATION POTENTIAL (FOR JUST BINARY INJECTION): %%

activationPotential = 0.5

%% WEIGHT DESIGN: %%


  %% FEATURE EXTRACTION LAYER IF USES BINARY INJECTION %%
  inputLayer = normalization(inputLayer); % inputs in range 0 - 1
  inputLayer(inputLayer < activationPotential) = 0
  inputLayer(inputLayer > activationPotential) = 1
  weights1(class,:) = inputLayer;


  %% FEATURE EXTRACTION LAYER IF USES DIRECT INJECTION %%

  inputLayer = normalization(inputLayer) %% inputs in range 0 - 1
  weights1(class,:) = inputLayer;


%% MERGING/SUMMING - SAVING WEIGHTS: %%

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

    weights1 = sprintf('weights/weights1.mat');
     save(weights1, 'weights1');

    weights2 = sprintf('weights/weights2.mat');
     save(weights2 , 'weights2');

  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.]]

    newWeights1 = weights1;

    weights1 = sprintf('weights/weights1.mat');
      load(weights1);


    weights1 += newWeights1;

    weights1 = sprintf('weights/weights1.mat');
     save(weights1, 'weights1');

  end
```

Binary Injected Model Test And Predict

If it's in the testing and prediction phase, it simply involves multiplying the input used during training, preserving only the threshold value we used while training for fex layer, with directly saved weights, without any other operations for cat layer:

```
inputLayer = normalization(inputLayer); % inputs in range 0 - 1
inputLayer(inputLayer < activationPotential) = 0
inputLayer(inputLayer > activationPotential) = 1
featureLayer = weights1 * inputLayer;

fexLayer = normalization(fexLayer); % fex neurons in range 0 - 1
catalystLayer = weights2 * featureLayer;
```

Direct Injected Model Test And Predict

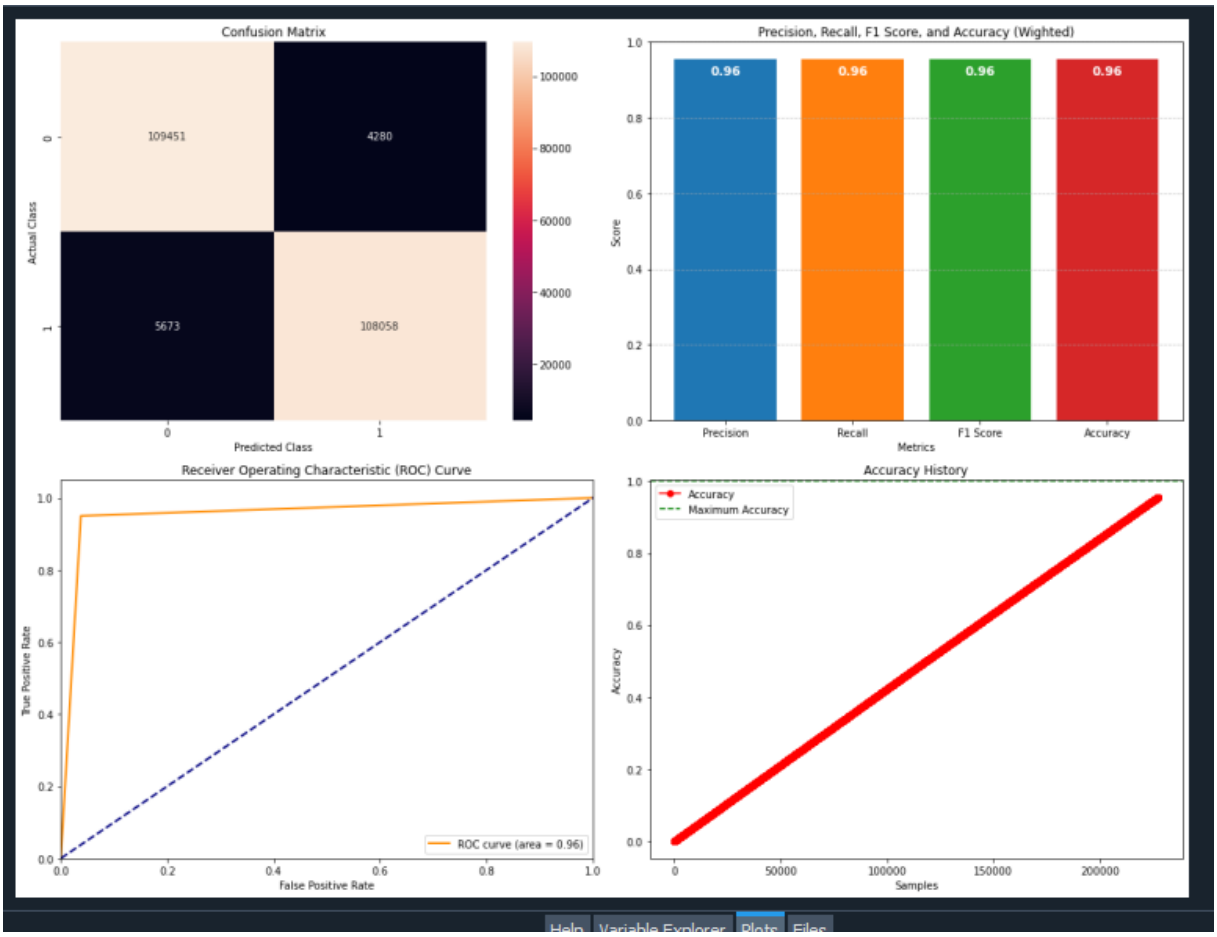
Only matrix multiplications:

```
inputLayer = normalization(inputLayer); % inputs in range 0 - 1
featureLayer = weights1 * inputLayer;

fexLayer = normalization(fexLayer); % fex neurons in range 0 - 1
catalystLayer = weights2 * featureLayer;
```

5 – Results:

All the codes for this application:
https://github.com/HCB06/PyerualJetwork/tree/main/Welcome%20to%20PLAN/Codes.

You can train and test models wtih PLAN architecture all datasets in this article:
https://github.com/HCB06/PyerualJetwork/tree/main/Welcome%20to%20Pyerual%20Jetwork/ExampleCodes



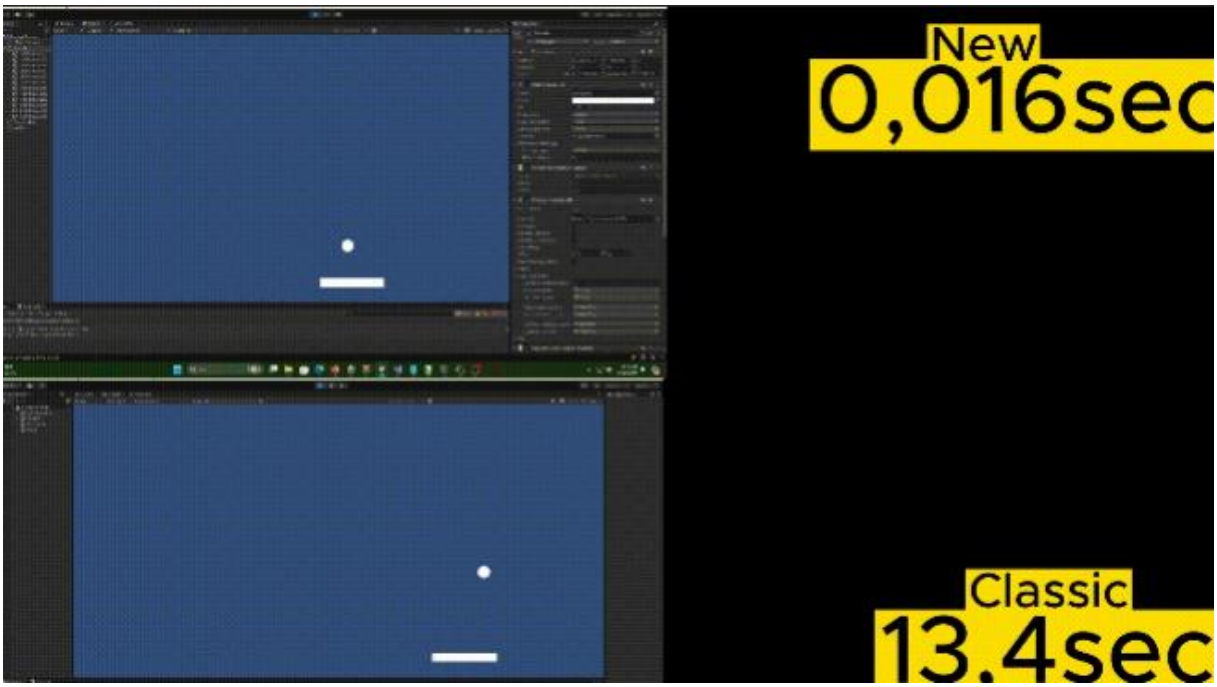
Dataset from: *https://www.kaggle.com/datasets/mlg-ulb/creditcardfraud*

Also I created this Unity demo for autonomous vehicles lane tracking AI with PLAN algorithm:

My YouTube video(Tr)*:* *https://www.youtube.com/watch?v=ApCKXqSMqT4&t=507s*

And with output layer normalization finalized: *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*

Training speed and overall performance comprasion with Tensorflow’s Dense ANN’s:



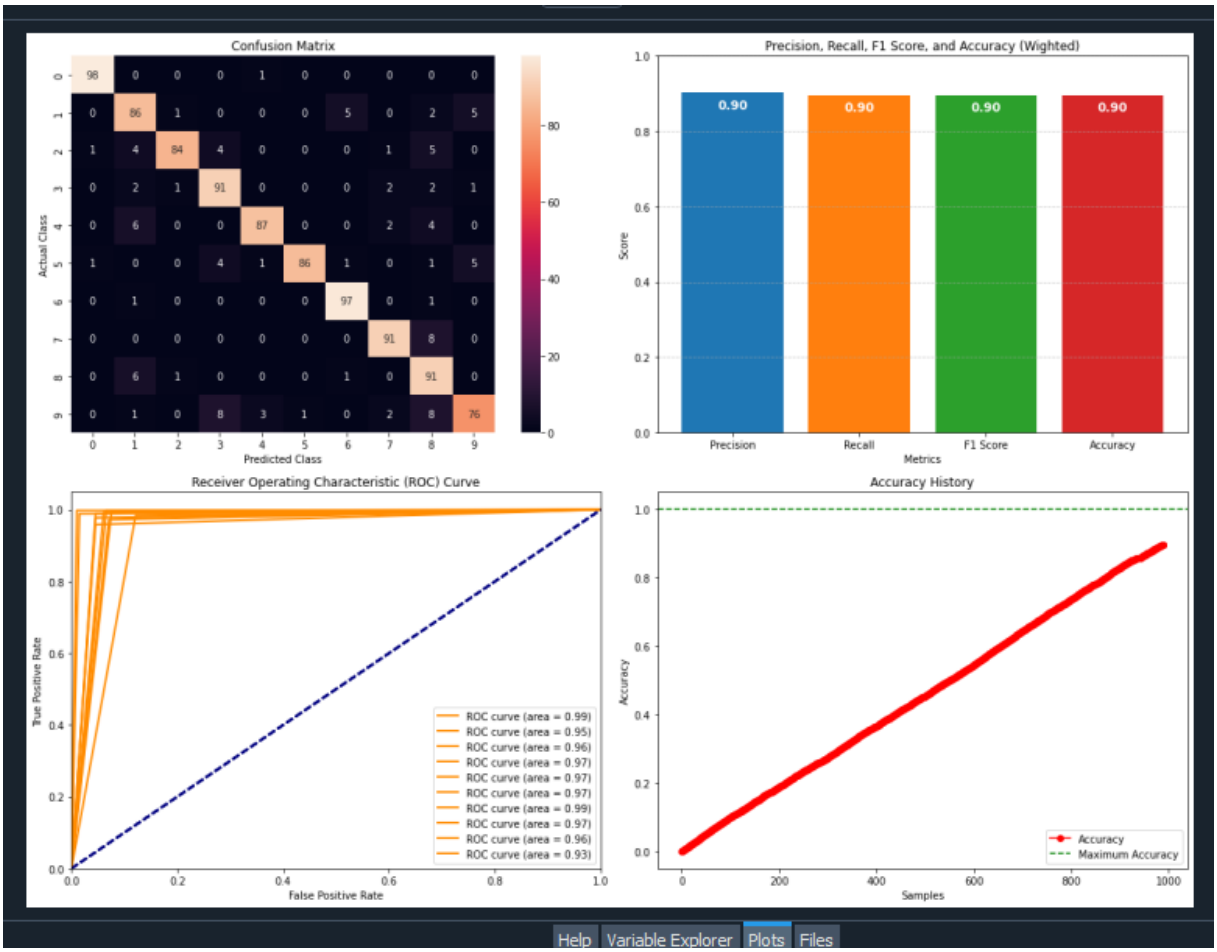
My YouTube video(Tr): *https://www.youtube.com/watch?v=tH7pgPSbyUo&t=1s*

Shortly: *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*

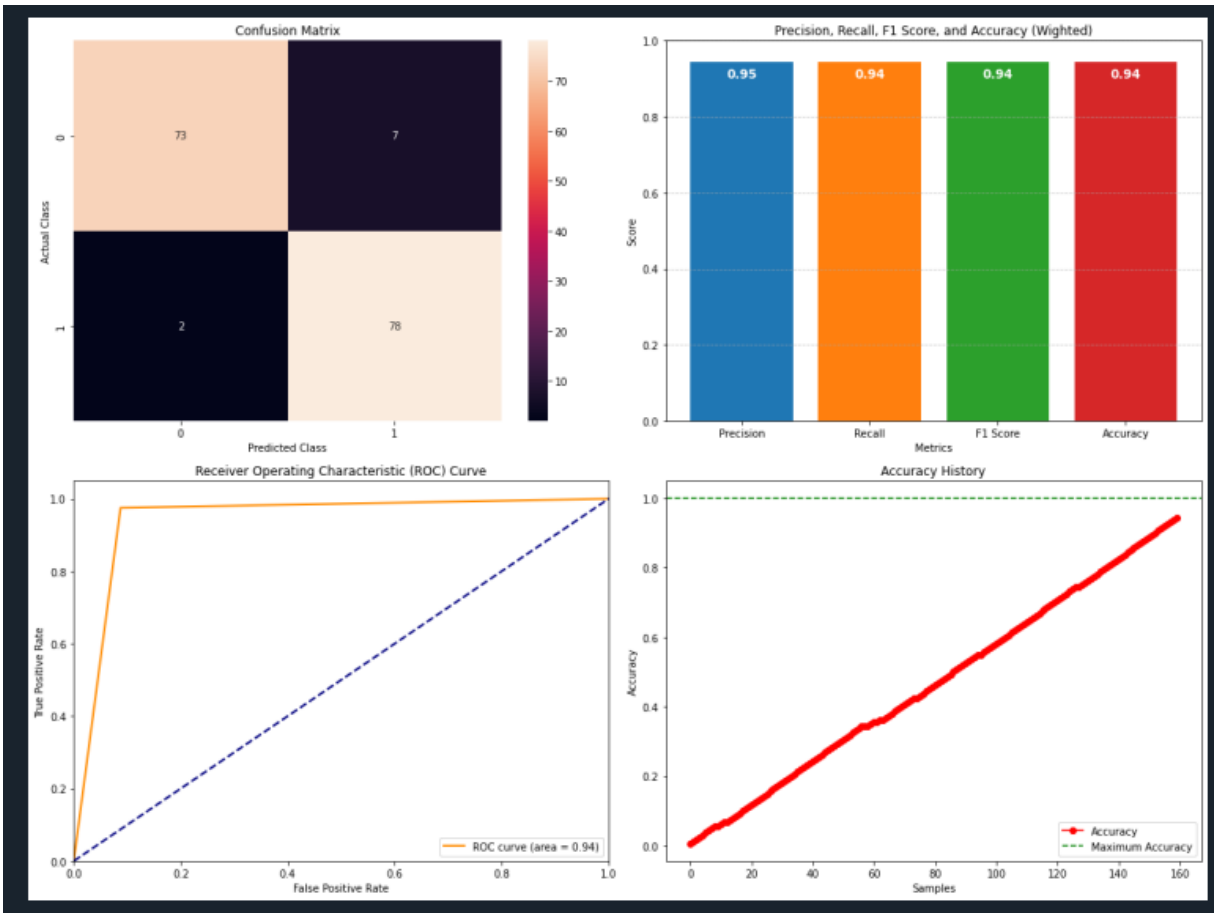
Trained and tested with digits dataset in sckitlearn library and comprasion wtih Tensorflow's fully connected ANN's (for equal states epoch selected 1):

My YouTube video(Tr): *https://www.youtube.com/watch?v=uT5mYvUVdmA&t=2s*

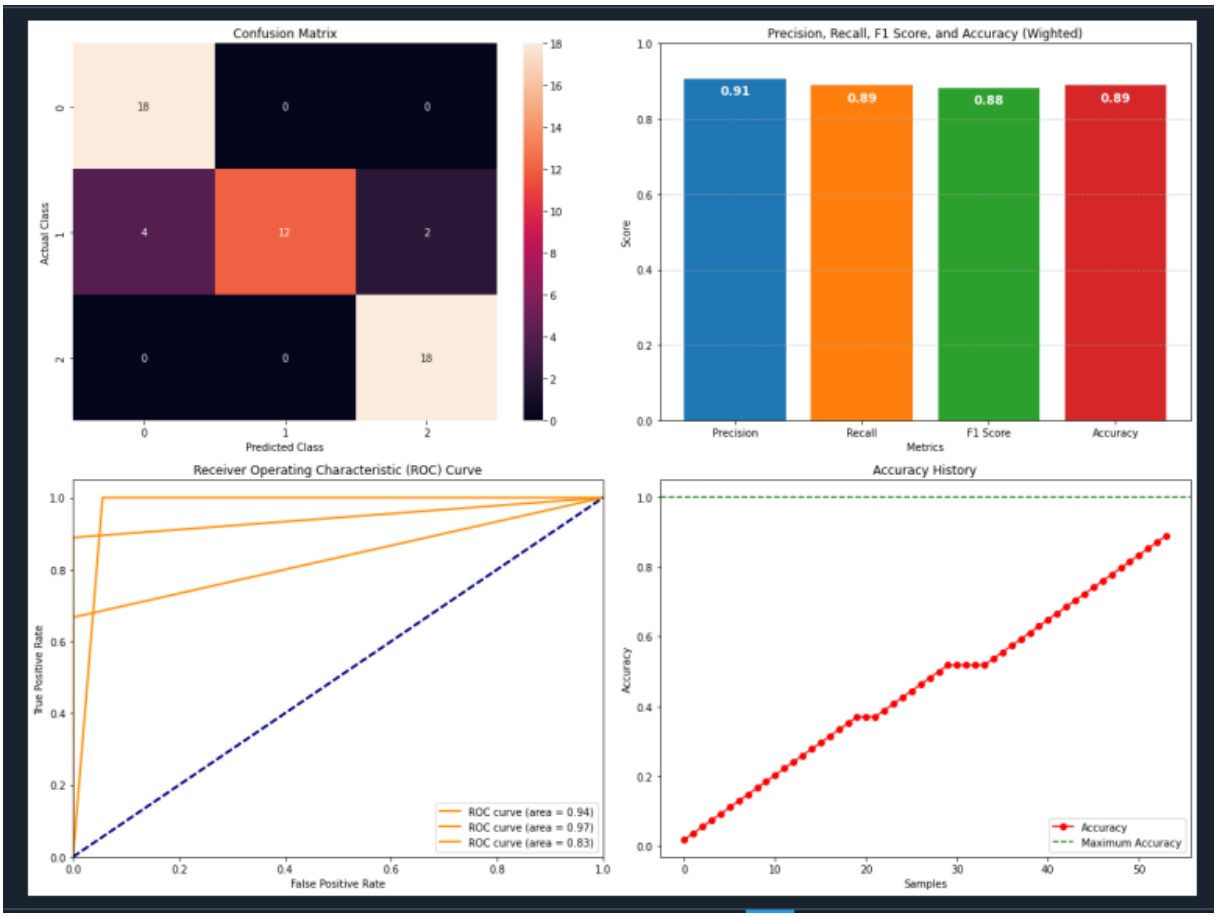
Shortly: *https://www.linkedin.com/posts/hasan-can-beydili-77a1b9270_bu-httpslnkdingvsmjtx-15-dakikal%C4%B1k-activity-7203772851785527299-KcfP?utm_source=share&utm_medium=member_desktop*



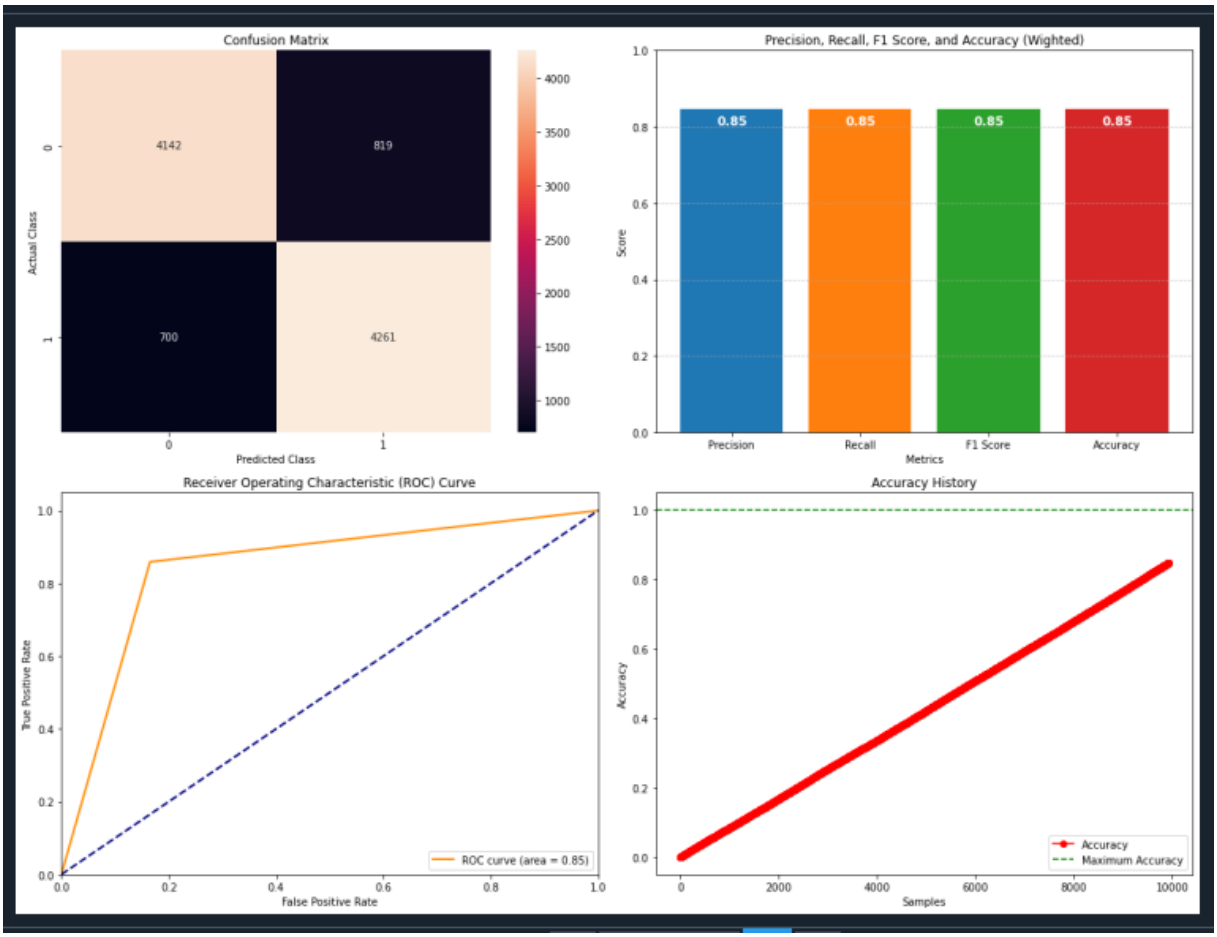
Dataset From: https://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_digits.html



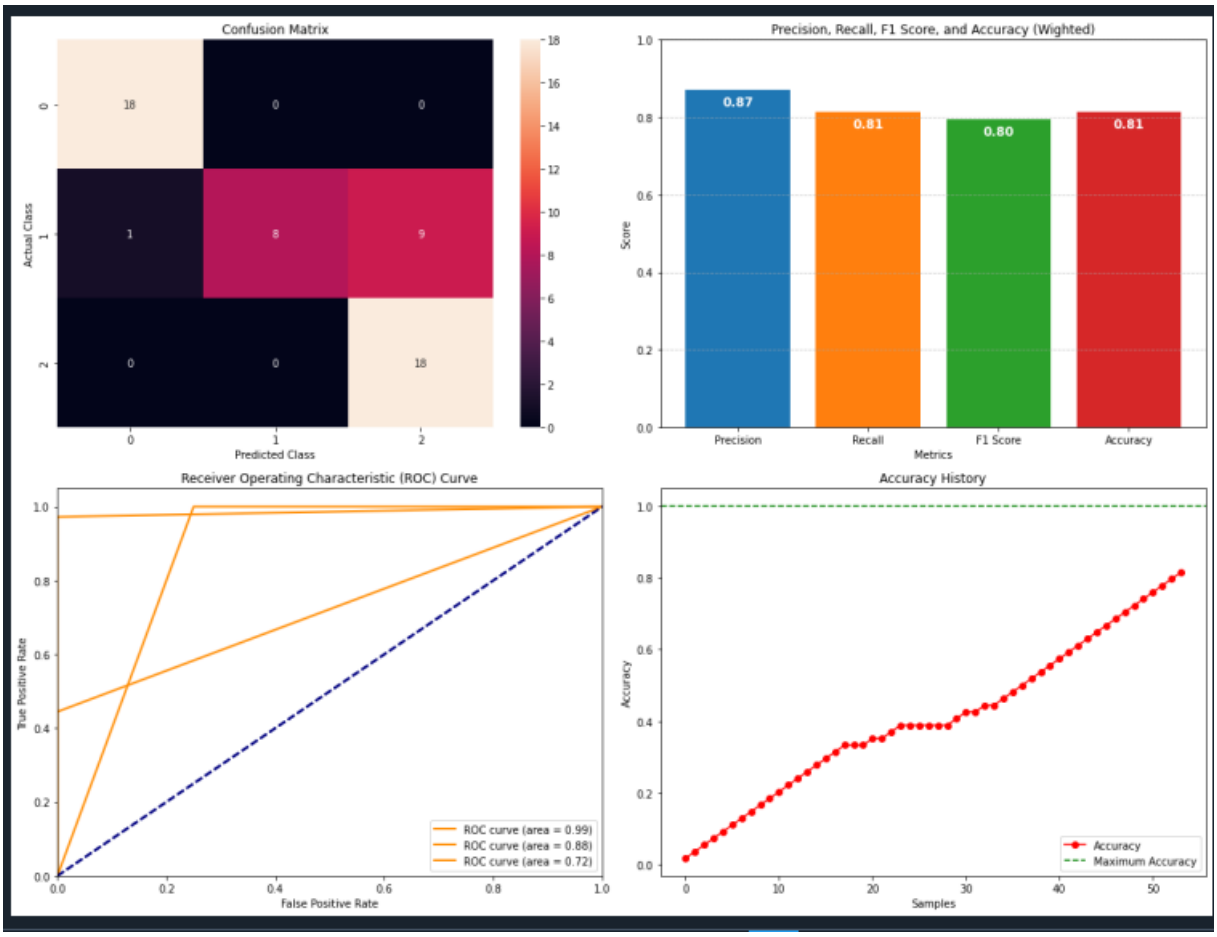
Dataset From: *https://www.kaggle.com/datasets/uciml/breast-cancer-wisconsin-data*



Dataset From: *https://www.geeksforgeeks.org/wine-dataset/*



Dataset From (6000 Features Selected):
https://www.kaggle.com/datasets/lakshmi25npathi/imdb-dataset-of-50k-movie-reviews



Dataset From: *https://www.kaggle.com/datasets/uciml/iris*

The results of implementing the PLAN architecture have been promising, though there are areas for improvement. While the current performance is not flawless, the potential for future advancements is significant.

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

Performance on Different Dataset Sizes

Through my research, I have observed that PLAN performs particularly well on large datasets. The architecture's ability to handle extensive input features and capture complex patterns makes it suitable for scenarios with abundant data. For instance, when applied to large datasets, PLAN can efficiently extract and process the underlying features, leading to high classification accuracy.

Conversely, when working with smaller datasets, such as the Iris dataset, the architecture shows a heightened sensitivity to the specific samples used for training. This sensitivity impacts the generalization performance significantly. The reliance on specific data points means that the model's performance can vary greatly with different training subsets,

highlighting the need for careful sample selection and potentially more sophisticated techniques to ensure robust generalization.

Also, while it exhibits near-perfect performance on datasets with a high number of features, it tends to struggle with datasets containing fewer features. However, in terms of overall performance, the model architecture trained without any hyperparameter tuning (direct injection) consistently achieved impressive results, never dropping below an 0.8 accuracy rate across all datasets.

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 PLAN algorithm can be to a general audience.

Future Directions and Developments

Developing Deeper Architectures: The "MOFEX"

I am currently conducting research on an advanced architecture that enables the extraction of multiple features, allowing for the development of sophisticated models capable of learning deep features, performing detailed analyses, and making informed decisions. This architecture, which I plan to announce soon, is named MOFEX (Multi Model Feature Extraction Architecture).

PLAN Library: Pyerual Jetwork

To facilitate broader experimentation and practical application, I have encapsulated the PLAN architecture in a Python library called "Pyerual Jetwork." This library is now available for both experimental and commercial use, allowing researchers and developers to easily integrate PLAN 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 - Pyerual Jetwork. Additionally, for practical demonstrations and tutorials on how to use the PLAN architecture, please visit my YouTube channel: YouTube - Hasan Can Beydili.

Integration with Existing Architectures

One of PLAN'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 (CNN), the PLAN framework's FEX (Feature Extraction Layer) and CAT (Catalyst Layer) can be employed. This approach not only leverages the

interpretability and efficiency of PLAN but also enhances the overall model by replacing complex layers with more manageable components.

Environmental Impact

Due to its non-iterative learning process, PLAN offers a significant advantage in terms of computational efficiency. In scenarios where PLAN is widely adopted, its streamlined approach to training can lead to a substantial reduction in the computational resources required. This efficiency has the potential 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 AI models, PLAN could contribute to reducing the environmental footprint of AI technologies.

6 – Advantages and Disadvantages:

Advantages:

1. Faster training times. (no loop usage, no epochs, no parameter optimizations)
2. Extremly fast training times. (dont need matrix multiplicaitons for training.)
3. There is no underfit problem (0 errors) → 0
4. Easy to learn and implement. (no calculus and no hyperparameter)
5. Easy to maintain and update. (knowing which neuron contains which information)
6. It occupies less memory space. (we can set all parameters in weights are integer)
7. It is a very flexible architecture (it can be easily hybridized with other artificial neural network architectures, theoretically).

Disadvantages:

1. 0 error may cause overfit problems.
2. Needs more improvements.

The PLAN architecture, through its innovative approach to neural network design, offers a compelling alternative to traditional methods. Its potential for faster, more interpretable, and environmentally friendly learning processes makes it a significant development in the AI landscape. The availability of the PLAN library on GitHub further encourages exploration and adoption of this architecture, paving the way for new applications and advancements in artificial intelligence.

All the codes in this article:
https://github.com/HCB06/PyerualJetwork/tree/main/Welcome%20to%20PLAN/Codes.

All gif's in this article:
https://github.com/HCB06/PyerualJetwork/tree/main/Welcome%20to%20PLAN/Gifs

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

Logo of Pyerual Jetwork library