



**ANAPLAN**  
**Trust the PLAN**

## **ANAPLAN 2.4.6 USER MANUAL**

*Author: Hasan Can Beydili*

## **ABOUT ANAPLAN:**

Anaplan is a machine learning library written in Python for professionals, incorporating advanced, unique, new, and modern techniques. Its most important component is the PLAN (Potentiation Learning Artificial Neural Network).

Both the PLAN algorithm and the Anaplan library were created by Author, and all rights are reserved by Author.

Anaplan is free to use for commercial business and individual users.

It is prohibited to copy or share the code and these documents by duplicating or using different names.

As of 12/21/2024, the library includes the PLAN and PLANEAT modules, but other machine learning modules are expected to be added in the future.

The PLAN algorithm will not be explained in this document. This document focuses on how professionals can integrate and use Pyerual Jetwork in their systems. However, briefly, the PLAN algorithm can be described as a classification algorithm. PLAN algorithm achieves this task with an incredibly energy-efficient, fast, and hyperparameter-free user-friendly approach. For more detailed information, you can check out

*[https://github.com/HCB06/Anaplan/blob/main/Welcome\\_to\\_PLAN/PLAN.pdf](https://github.com/HCB06/Anaplan/blob/main/Welcome_to_PLAN/PLAN.pdf)*

## HOW DO I IMPORT IT TO MY PROJECT?

Anaconda users can access the 'Anaconda Prompt' terminal from the Start menu and add the necessary library modules to the Python module search queue by typing "`pip install anaplan`" and pressing enter. If you are not using Anaconda, you can simply open the 'cmd' Windows command terminal from the Start menu and type "`pip install anaplan`". (Visual Studio Code recommended) After installation, it's important to periodically open the terminal of the environment you are using and stay up to date by using the command "`pip install anaplan --upgrade`". The latest version was "2.4" at the time this document was written

After installing the module using "`pip`" you can now call the library module in your project environment. For example: "`from anaplan import plan`". Now, you can call the necessary functions from the plan module.

## **LIBRARY ARCHITECTURE:**

*The functions of the Anaplan modules, uses snake\_case written style.*

### **Main Modules and Functions:**

#### **1. plan**

- a. fit()
- b. evaluate ()
- c. learner()
- d. get\_weights ()
- e. get\_scaler ()
- f. get\_preds ()
- g. get\_acc ()
- h. get\_act\_pot ()

#### **2. planeat**

- a. define\_genoms()
- b. evaluate()
- c. learner()

### **Supportive Modules and Functions:**

#### **1. data\_operations**

- a. split()
- b. one\_hot\_encode()
- c. one\_hot\_decode()
- d. auto\_balancer()
- e. manuel\_balancer()
- f. synthetic\_augmentation()
- g. standard\_scaler()

#### **2. model\_operations**

- a. save\_model()
- b. load\_model()
- c. predict\_model\_ram()
- d. predict\_model\_ssd()

## PLAN MODULE

It applies the PLAN (Potentiation Learning Artificial Neural Network) algorithm, which incorporates a unique learning architecture specifically designed by me to enhance **"explainability and efficiency of AI models."** This algorithm is based on my custom optimization approach, TFL (Test or Train Feedback Learning), which does not rely on the traditional Backpropagation method.

The advantage of this algorithm lies in its ability to create "super perceptrons." For example, imagine a perceptron achieving over 0.9 accuracy in classifying the circles dataset—this is something the PLAN module can achieve. The goal of this algorithm is to develop artificial neural network models that are **"as simple as a perceptron yet as powerful in learning capabilities as multi-layer perceptrons."**

## PLAN MODULE FUNCTIONS

### 1. `plan.fit()`

The purpose of this function, as the name suggests, is to train the model.

```
a.      fit Args:
b.          x_train (list[num]): List or ndarray of input data.
c.
d. y_train (list[num]): List or ndarray of target labels. (one hot
   encoded)
e.
f. val (None or True): validation in training process ? None or True
   default: None (optional)
g.
h. val_count (None or int): After how many examples learned will an
   accuracy test be performed? default: 10=(%10) it means every
   approximately 10 step (optional)
```

```

i.
j. activation_potentialiation (list): For deeper PLAN networks, activation
   function parameters. For more information please run this code:
   activation_functions.activations_list() default: [None] (optional)
k.
l. x_val (list[num]): List of validation data. default: x_train (optional)
m.
n. y_val (list[num]): (list[num]): List of target labels. (one hot encoded)
   default: y_train (optional)
o.
p. show_training (bool, str): True or None default: None (optional)
q.
r. LTD (int): Long Term Depression Hyperparameter for train PLAN neural
   network default: 0 (optional)
s.
t. interval (float, int): frame delay (milisecond) parameter for Training
   Report (show_training=True) This parameter effects to your Training
   Report performance. Lower value is more diffucult for Low end PC's
   (33.33 = 30 FPS, 16.67 = 60 FPS) default: 100 (optional)
u.
v. decision_boundary_status (bool): If the visualization of validation and
   training history is enabled during training, should the decision
   boundaries also be visualized? True or False. Default is True.
   (optional)
w.
x. train_bar (bool): Training loading bar? True or False. Default is True.
   (optional)
y.
z. auto_normalization(bool): Normalization process during training. May
   effect training time and model quality. True or False. Default is True.
   (optional)
aa.
bb. neurons_history (bool, optional): Shows the history of changes that
   neurons undergo during the CL (Cumulative Learning) stages. True or
   False. Default is False. (optional)
cc.
dd.     Returns:
ee.     numpyarray([num]): (Weight matrix).

```

The output of this function Weight matrix of model.

## 2. plan.evaluate()

This function calculates the test accuracy of the model using the inputs and labels set aside for testing, along with the weight matrices and other model parameters obtained as output from the training function.

```
a. x_test (list[num]): Test input data.
b.
c. y_test (list[num]): Test labels.
d.
e. W (list[num]): Weight matrix list of the neural network.
f.
g. activation_potentialiation (list): For deeper PLAN networks, activation
   function parameters. For more information please run this function:
   'plan.activations_list()' default: ['linear']
h.
i. loading_bar_status: Evaluate progress have a loading bar ? (True or
   False) Default: True.
j.
k. show_metrics (bool): (True or None) (optional) Default: None
```

The outputs of this function are, in order: weights of test process, a list of test predictions, and test accuracy rate.

## 3. plan.learner()

Optimizes the activation functions for a neural network by leveraging train data to find the most accurate combination of activation potentialiation for the given dataset.

This next-generation generalization function includes an advanced learning feature that is specifically tailored to the PLAN algorithm.

It uniquely adjusts hyperparameters based on test accuracy while training with model-specific training data, offering an unparalleled optimization technique.

Designed to be used before model evaluation. This called TFL(Test or Train Feedback Learning).

```
a.      Args:
b.          x_train (array-like): Training input data.
c.
d.          y_train (array-like): Labels for training data.
e.
f.          x_test (array-like, optional): Test input data (for improve
next gen generilization). If test data is not given then train
feedback learning active
g.
h.          y_test (array-like, optional): Test Labels (for improve next
gen generilization). If test data is not given then train feedback
learning active
i.
j.          strategy (str, optional): Learning strategy. (options:
'accuracy', 'loss', 'f1', 'precision', 'recall', 'adaptive_accuracy',
'adaptive_loss', 'all'): 'accuracy', Maximizes test accuracy during
learning. 'f1', Maximizes test f1 score during learning. 'precision',
Maximizes test precision score during learning. 'recall', Maximizes
test recall during learning. 'loss', Minimizes test loss during
learning. 'adaptive_accuracy', The model compares the current
accuracy with the accuracy from the past based on the number
specified by the patience value. If no improvement is observed it
adapts to the condition by switching to the 'loss' strategy quickly
starts minimizing loss and continues learning. 'adaptive_loss',The
model adopts the 'loss' strategy until the loss reaches or falls
below the value specified by the patience parameter. However, when
the patience threshold is reached, it automatically switches to the
'accuracy' strategy and begins to maximize accuracy. 'all', Maximizes
all test scores and minimizes test loss, 'all' strategy most strong
and most robust strategy. Default is 'accuracy'.
k.
l.          patience ((int, float), optional): patience value for
adaptive strategies. For 'adaptive_accuracy' Default value: 5. For
'adaptive_loss' Default value: 0.150.
m.
n.          depth (int, optional): The depth of the PLAN neural networks
Aggreagation layers.
```



o.

p.       **batch\_size (float, optional):** Batch size is used in the prediction process to receive test feedback by dividing the test data into chunks and selecting activations based on randomly chosen partitions. This process reduces computational cost and time while still covering the entire test set due to random selection, so it doesn't significantly impact accuracy. For example, a batch size of 0.08 means each test batch represents 8% of the test set. Default is 1. (%100 of test)

q.

r.       **auto\_normalization (bool, optional):** If auto normalization=False this makes more faster training times and much better accuracy performance for some datasets. Default is True.

s.

t.       **early\_shifting (int, optional):** Early shifting checks if the test accuracy improves after a given number of activation attempts while inside a depth. If there's no improvement, it automatically shifts to the next depth. Basically, if no progress, it's like, "Alright, let's move on!" Default is False

u.

v.       **early\_stop (bool, optional):** If True, implements early stopping during training.(If test accuracy not improves in two depth stops learning.) Default is False.

w.

x.       **show\_current\_activations (bool, optional):** Should it display the activations selected according to the current strategies during learning, or not? (True or False) This can be very useful if you want to cancel the learning process and resume from where you left off later. After canceling, you will need to view the live training activations in order to choose the activations to be given to the 'start\_this' parameter. Default is False

y.

z.       **show\_history (bool, optional):** If True, displays the training history after optimization. Default is False.

aa.

bb.       **loss (str, optional):** For visualizing and monitoring. PLAN neural networks doesn't need any loss function in training(if strategy not 'loss'). options: ('categorical\_crossentropy' or 'binary\_crossentropy') Default is 'categorical\_crossentropy'.

cc.

dd.       **interval (int, optional):** The interval at which evaluations are conducted during training. (33.33 = 30 FPS, 16.67 = 60 FPS) Default is 100.

ee.

ff.       **target\_acc (int, optional):** The target accuracy to stop training early when achieved. Default is None.

gg.

hh.       **target\_loss (float, optional):** The target loss to stop training early when achieved. Default is None.

```

ii.
jj.      except_this (list, optional): A list of activations to
        exclude from optimization. Default is None. (For available activation
        functions, run this function: plan.activations_list())
kk.
ll.      only_this (list, optional): A list of activations to focus on
        during optimization. Default is None. (For available activation
        functions, run this code: plan.activations_list())
mm.
nn.      start_this (list, optional): To resume a previously canceled
        or interrupted training from where it left off, or to continue from
        that point with a different strategy, provide the list of activation
        functions selected up to the learned portion to this parameter.
        Default is None
oo.
pp.      neurons_history (bool, optional): Shows the history of
        changes that neurons undergo during the TFL (Test Feedback Learning)
        stages. True or False. Default is False.
qq.
rr.      Returns:
ss.      tuple: A list for model parameters: [Weight matrix, Test
        loss, Test Accuracy, [Activations functions]].
tt.
uu.

```

Returns: model(list)

#### 4. plan.get\_weights()

This function returns weight matrices list of the selected model. For exp:

```
test_model = plan.evaluate(x_test, y_test)
```

```
W = test_model[plan.get_weights()]
```

#### 5. plan.get\_scaler()

Returns scaler\_params of the selected model For exp:

```
model = plan.learner(x_train, y_train, depth=10)
scaler_params = model[plan.get_scaler()]
```

## 6. `plan.get_preds()`

Returns predictions list of the selected model

## 7. `plan.get_acc()`

Returns accuracy of the selected model

## 8. `plan.get_act_pot()`

Returns activation potential of the selected model.

# PLANEAT MODULE

The PLANEAT module represents a hybrid approach that combines the PLAN algorithm with the NEAT (Neuroevolution of Augmented Topologies) algorithm. PLANEAT is a hybrid algorithm designed for use in Reinforcement Learning problems as well as Genetic Optimization tasks. Its goal is to enhance the efficiency of traditional NEAT by focusing on creating "**super genetically optimized perceptrons**" without

unnecessary layers and connections, similar to the activation-focused approach of the PLAN algorithm.

It is also designed to cater to all user groups by offering user-friendly preset modes alongside limitless configuration options for advanced users, ensuring satisfaction for a broad audience.

## PLANEAT MODULE FUNCTIONS

### 1. `planeat.define_genomes ()`

Creates PLANEAT environment.

```
a.   Initializes a population of genomes, where each genome is represented
    by a set of weights
b.
c.   and an associated activation function. Each genome is created with
    random weights and activation
d.   functions are applied and normalized. (Max abs normalization.)
e.
f.   Args:
g.       input_shape (int): The number of input features for the neural
    network.
h.       output_shape (int): The number of output features for the neural
    network.
i.       population_size (int): The number of genomes (individuals) in the
    population.
j.
k.   Returns:
l.       tuple: A tuple containing:
m.           - population_weights (numpy.ndarray): A 2D numpy array of shape
    (population_size, output_shape, input_shape) representing the
n.               weight matrices for each genome.
o.           - population_activations (list): A list of activation functions
    applied to each genome.
p.
```

```

q.     Raises:
r.         ValueError:
s.         - If the population size is odd (ensuring an even number of
t.           genomes is required for proper selection).
u.     Notes:
v.         The weights are initialized randomly within the range [-1, 1].
w.         Activation functions are selected randomly from a predefined list
x.         `activations_list()`.
y.         The weights for each genome are then modified by applying the
z.         corresponding activation function
        and normalized using the `normalization()` function. (Max abs
        normalization.)

```

## 2. planeat.evaluate ()

Making predictions for each genome in single generation. (Or if `rl_mode=True` then making prediction for single genome.)

```

aa.     Evaluates the performance of a population of genomes, applying
bb.         different activation functions
cc.         and weights depending on whether reinforcement learning mode is
dd.         enabled or not.
ee.     Args:
ff.         x_population (list or numpy.ndarray): A list or 2D numpy array
gg.             where each element represents
hh.             a genome (A list of input features for each genome, or a single set of
ii.             input features for one genome (only in rl_mode)).
jj.         weights (list or numpy.ndarray): A list or 2D numpy array of
kk.             weights corresponding to each genome in `x_population`. This determines
ll.             the strength of connections.
mm.         activation_potentiations (list or str): A list where each entry
nn.             represents an activation function
oo.             or a potentiation
pp.             strategy applied to each genome. If only one
qq.             activation function is
rr.             used, this can be a single string.
ss.         rl_mode (bool, optional): If True, reinforcement learning mode
tt.             is activated, this accepts x_population is a single genom. (Also weights
uu.             and activation_potentiations a single genomes part.)

```

```

nn.                                     Default is False.
oo.
pp.   Returns:
qq.       list: A list of outputs corresponding to each genome in the
rr.           population after applying the respective
ss.           activation function and weights.
tt.   Notes:
uu.       - If `rl_mode` is True:
vv.           - Accepts x_population is a single genom
ww.           - The inputs are flattened, and the activation function is
              applied across the single genom.
xx.
yy.       - If `rl_mode` is False:
zz.           - Accepts x_population is a list of genomes
aaa.          - Each genome is processed individually, and the
              results are stored in the `outputs` list.
bbb.
ccc.          - `fex()` function is the core function that processes the
              input with the given weights and activation function.
ddd.
eee.       Example:
fff.           ```python
ggg.           outputs = evaluate(x_population, weights,
              activation_potentiations, rl_mode=False)
hhh.           ```
iii.
jjj.       - The function returns a list of outputs after processing the
              population, where each element corresponds to
kkk.          the output for each genome in `x_population`.
lll.

```

### 3. planeat.learner()

Applies (Adjust weight and actiavation parameters) PLANEAT algorithm for each genome in population.

```

mmm.       Applies the learning process of a population of genomes using
              selection, crossover, mutation, and activation function potentiation.
nnn.       The function modifies the population's weights and activation
              functions based on a specified policy, mutation probabilities, and
              strategy.
ooo.
ppp.       Args:

```

```

qqq.         weights (numpy.ndarray): Array of weights for each genomes.
              (first returned value of define_genomes function)
rrr.
sss.         activation_potentiations (list): A list of activation
              functions for each genomes. (second returned value of define_genomes
              function)
ttt.
uuu.         what_gen (int): The current generation number, used for
              informational purposes or logging.
vvv.
www.         y_reward (numpy.ndarray): A 1D array containing the
              fitness or reward values of each genome. The array is used to rank the
              genomes based on their performance. And PLANEAT maximize the reward.
xxx.
yyy.         show_info (bool, optional): If True, prints information
              about the current generation and the maximum reward obtained. Also shows
              current configuration. Default is False.
zzz.
aaaa.        strategy (str, optional): The strategy for combining the
              best and bad genomes. Options:
bbbb.        - 'cross_over': Perform Two-
              Point Matrix Crossover between the best genomes and replace bad genomes.
              (Classic NEAT cross over)
cccc.        - 'potentiate': Cumulate the
              weight of the best genomes and replace bad genomes. (PLAN feature. 'Like
              Arithmetic Crossover but different.') Default is 'cross_over'.
dddd.
eeee.        policy (str, optional): The selection policy that governs
              how genomes are selected for reproduction. Options:
ffff.        - 'normal_selective':
gggg.        Normal selection based on reward, where a portion of the bad genes
              are discarded.
hhhh.        - 'more_selective': A more
              selective policy, where fewer bad genes survive.
iiii.        - 'less_selective': A less
              selective policy, where more bad genes survive.
jjjj.        Default is 'normal_selective'.
kkkk.
llll.        mutations (bool, optional): If True, mutations are applied
              to the bad genomes and potentially to the best genomes as well. Default
              is True.
mmmm.        bad_genoms_mutation_prob (float, optional): The
              probability of applying mutation to the bad genomes. Must be in the
              range [0, 1]. Also effects best genoms mutation prob. For example 0.7
              value for bad genoms then 0.3 value for best genoms. Default is None,
              which means it is determined by the `policy` argument.
nnnn.

```

**activation\_mutate\_prob (float, optional):** The probability of applying mutation to the activation functions. Must be in the range [0, 1]. Default is 0.5 (% 50)

**cross\_over\_mode (str, optional):** Specifies the crossover method to use.  
Options:

- 'tpm': Two-Point Matrix Crossover
  - 'plantic': plantic Crossover
- Default is 'tpm'.

**activation\_add\_prob (float, optional):** The probability of adding a new activation function to the genome.  
Must be in the range [0, 1]. Default is 0.9.

**activation\_delete\_prob (float, optional):** The probability of deleting an existing activation function  
from the genome. Must be in the range [0, 1]. Default is 0.1.

**activation\_change\_prob (float, optional):** The probability of changing an activation function in the genome.  
Must be in the range [0, 1]. Default is 0.5.

**weight\_mutate\_prob (float, optional):** The probability of mutating a weight in the genome.  
Must be in the range [0, 1]. Default is 1.

**weight\_mutate\_rate (int, optional):** If the value you enter here is equal to the result of input layer \* output layer,  
only a single weight will be mutated during each mutation process.  
If the value you enter here is half  
of the result of input layer \* output layer, two weights in the weight matrix will be mutated.  
WARNING: if you don't understand do NOT change this value. Default is 32.

**save\_best\_genom (bool, optional):** If True, ensures that the best genome are saved and not mutated or altered during reproduction. Default is True.

oooo.

pppp. **Raises:**

qqqq. **ValueError:**

rrrr. - If `policy` is not one of the specified values  
(`normal\_selective`, `more\_selective`, `less\_selective`).

ssss.

tttt. - If `bad\_genoms\_mutation\_prob`,  
`activation\_mutate\_prob`, or other probability parameters are not in the range [0, 1].

uuuu. - If the population size is odd (ensuring an even  
number of genomes is required for proper selection).

vvvv.



```

wwww.
xxxx.      Returns:
yyyy.      tuple: A tuple containing:
zzzz.      - weights (numpy.ndarray): The updated weights for the
           population after selection, crossover, and mutation.
aaaaa.      The shape is
           (population_size, output_shape, input_shape).
bbbbbb.     - activation_potentiations (list): The updated list of
           activation functions for the population.
cccccc.
ddddd.      Notes:
eeeee.      - **Selection Process**:
ffffff.     - The genomes are sorted by their fitness (based on
           `y_reward`), and then split into "best" and "bad" half.
ggggg.     - The best genomes are retained, and the bad genomes
           are modified based on the selected strategy.
hhhhh.
iiiiii.     - **Crossover and Potentiation Strategies**:
jjjjj.     - The **'cross_over'** strategy performs crossover,
           where parts of the best genomes' weights are combined with the other
           good genomes to create new weight matrices.
kkkkk.     - The **'potentiate'** strategy strengthens the best
           genomes by potentiating their weights towards the other good genomes.
lllll.
mmmmm.     - **Mutation**:
nnnnn.     - Mutation is applied to both the best and bad
           genomes, depending on the mutation probability and the `policy`.
ooooo.     - `bad_genoms_mutation_prob` determines the
           probability of applying mutations to the bad genomes.
ppppp.     - If `activation_mutate_prob` is provided, activation
           function mutations are applied to the genomes based on this probability.
qqqqq.
rrrrr.
sssss.
- **Policy Types**:

- **'normal_selective'**: A typical selection policy where a portion of the
bad genomes is discarded. The mutation probability for the bad genomes is
set to 0.7 by default.

- **'more_selective'**: A stricter selection policy where more of the bad
genomes are discarded. The mutation probability for the bad genomes is set
to 0.85 by default.

- **'less_selective'*
z*: A more lenient selection policy where fewer bad genomes are discarded.
The mutation probability for the bad genomes is set to 0.6 by default.

```

```

- **Population Size**: The population size must be an even number to
properly split the best and bad genomes. If `y_reward` has an odd length,
an error is raised.

- **Logging**: If `show_info=True`, the current generation and the maximum
reward from the population are printed for tracking the learning progress.

    Example:
ttttt.      ```python
uuuuu.      weights, activation_potentiations = learner(weights,
    activation_potentiations, 1, y_reward, info=True, strategy='cross_over',
    policy='normal_selective')
vvvvv.      ```
wwwww.
xxxxx.      - The function returns the updated weights and activations
    after processing based on the chosen strategy, policy, and mutation
    parameters.
yyyyy.      """
zzzzz.

```

## DATA OPERATIONS MODULE **FUNCTIONS**

### 1. **data\_operations.auto\_balancer()**

This function aims to balance all training data according to class distribution before training the model. All data is reduced to the number of data points of the class with the least number of examples.

```

a.    x_train (list): Input data for training.
b.    y_train (list): Labels corresponding to the input data.

```

This function returns the following outputs in order: a list containing the balanced training data and a list containing the balanced training labels.

## 2. `data_operations.synthetic_augmentation()`

This function creates synthetic data samples with given data samples for balance data distribution.

```
a.      x -- Input dataset (examples) - array format
b.      y -- Class labels (one-hot encoded) - array format
```

This function returns the following outputs in order: a list containing the balanced training data and a list containing the balanced training labels. or testing labels.

## 3. `data_operations.encode_one_hot()`

```
a. Performs one-hot encoding on y_train and y_test data.
b.
c. Args:
d.
e. y_train (numpy.ndarray): Labeled train data.
f. y_test (numpy.ndarray): Labeled test data.
g. summary (bool): If True, prints the class-to-index mapping. Default:
   False
h.
```

Returns one hot encoded labels.

#### 4. `data_operations.split()`

This function splits all data for train and test

```
a.      X (numpy.ndarray): Features data.  
b.  
c.      y (numpy.ndarray): Labels data.  
d.  
e.      test_size (float or int): Proportion or number of samples for  
      the test subset.  
f.  
g.      random_state (int or None): Seed for random state.
```

Returns: x\_train, x\_test, y\_train, y\_test

#### 5. `data_operations.decode_one_hot()`

```
a. encoded_data (numpy.ndarray): One-hot encoded data with shape  
   (n_samples, n_classes).
```

Returns: decoded y\_test given input

#### 6. `data_operations.manual_balancer ()`

Same operation of auto\_balancer, but this function gives the limit of sample addition to user.

```
a.    x_train -- Input dataset (examples) - NumPy array format
b.
c.    y_train -- Class labels (one-hot encoded) - NumPy array format
d.
e.    target_samples_per_class -- Desired number of samples per class
```

Returns: x\_train, y\_train

## 7. data\_operations.standard\_scaler()

```
a. train_data: numpy.ndarray
b.
c. test_data: numpy.ndarray (optional)
d.
e. scaler_params (optional for using model)
```

Returns: If x\_test given then returns: standard scaled parameters, standard scaled x\_train, standard scaled y\_test. If x\_test is not given then returns: standard scaled parameters, standard scaled x\_train.

## MODEL OPERATIONS MODULE FUNCTIONS

### 1. model\_operations.save\_model ()

This function creates log files in the form of a pandas DataFrame containing all the parameters and information of the trained and tested model, and saves them to the specified location along with the weight matrices.

```
aaaaaa.    Function to save a potentiation learning model.
bbbbbb.
cccccc.    Arguments:
ddddd.
```

```

eeeeee. model_name (str): Name of the model.
ffffff.
gggggg. model_type (str): Type of the model. default: 'PLAN'
hhhhhh.
iiiiii. test_acc (float): Test accuracy of the model. default: None
jjjjjj.
kkkkkk. weights_type (str): Type of weights to save (options: 'txt',
      'pkl', 'numpy', 'mat'). default: 'numpy'
llllll.
mmmmmm. weights_format (str): Format of the weights (options: 'f', 'raw').
      default: 'raw'
nnnnnn.
oooooo. model_path (str): Path where the model will be saved. For example:
      C:/Users/beydili/Desktop/denemePLAN/ default: ''
pppppp.
qqqqqq. scaler_params (list[num, num]): standard scaler params list:
      mean,std. If not used standard scaler then be: None.
rrrrrr.
ssssss. W: Weights of the model.
tttttt.
uuuuuu. activation_potentiation (list): For deeper PLAN networks,
      activation function parameters. For more information please run this
      code: activation_functions.activations_list() default: ['linear']
vvvvvv.
wwwwww. Returns:
xxxxxx. str: Message indicating if the model was saved successfully or
      encountered an error.

```

This function returns messages such as 'saved' or 'could not be saved' as output.

## 2. model\_operations.load\_model ()

This function retrieves everything about the model into the Python environment from the saved log file and the model name.

```

a. model_name (str): Name of the model.
b. model_path (str): Path where the model is saved.

```

This function returns the following outputs in order: W, activation\_potentiation, df (Data frame of model).

### 3. `model_operations.predict_model_ssd ()`

This function loads the model directly from its saved location, predicts a requested input, and returns the output. (It can be integrated into application systems and the output can be converted to .json format and used in web applications.)

```
a. Input (list or ndarray): Input data for the model (single vector or single matrix).  
b. model_name (str): Name of the model.
```

This function returns the last output layer of the model as the output of the given input.

### 4. `model_operations.predict_model_ram ()`

This function predicts and returns the output for a requested input using a model that has already been loaded into the program (located in the computer's RAM). (It can be integrated into application systems and the output can be converted to .json format and used in web applications.) (Other parameters are information about the model and are defined as described and listed above.)

```
a. Input (list or ndarray): Input data for the model (single vector or single matrix).  
b.  
c. W (list of ndarrays): Weights of the model.  
d.
```

```
e.     scaler_params (list): standard scaler params list: mean,std.  
      (optional) Default: None.  
f.  
g.     activation_potentialiation (list): ac list for deep PLAN. default:  
      [None] ('linear') (optional)
```

This function returns the last output layer of the model as the output of the given input.

### LAST PART:

**Despite being in its early stages of development, Anaplan has already demonstrated its potential to deliver valuable services and solutions in the field of machine learning. Notably, it stands as the first library dedicated to PLAN (Potentiation Learning Artificial Neural Network), embracing innovation and welcoming new ideas from its users with open arms. Recognizing the value of diverse perspectives and fresh ideas, Hasan Can Beydili the creator of Anaplan, am committed to fostering an open and collaborative environment where users can freely share their thoughts and suggestions. The most promising contributions will be carefully considered and potentially integrated into the Anaplan library. For your suggestions, lists and feedback, my e-mail address is: [tchasancan@gmail.com](mailto:tchasancan@gmail.com)**

*Trust the PLAN...*