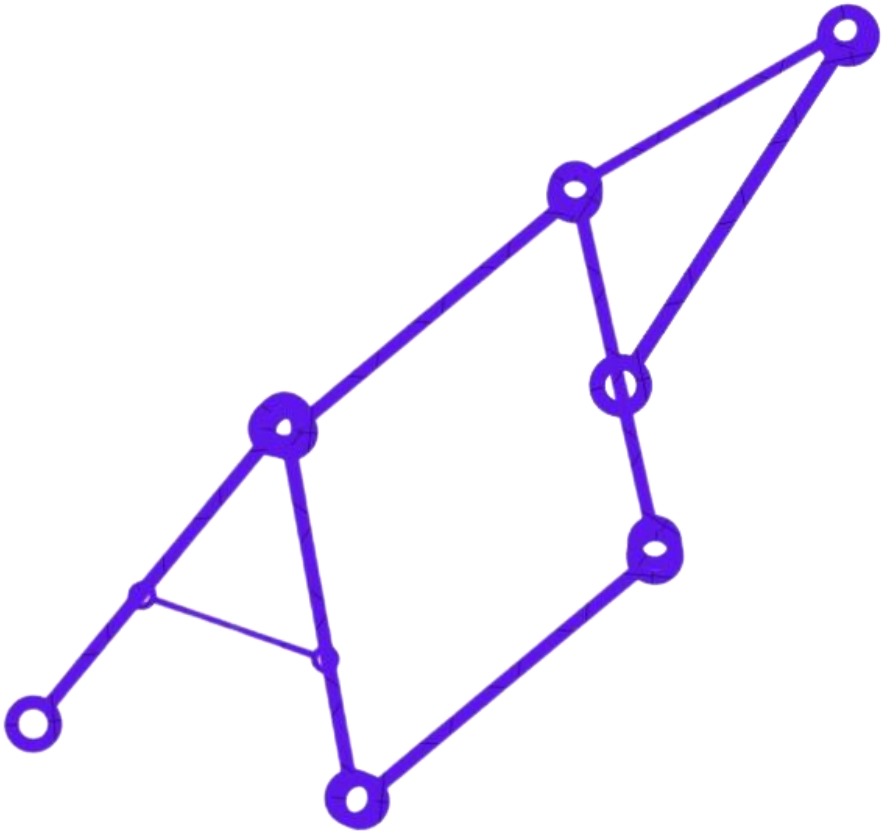

PYERUALJETWORK 4.4 HIGH LEVEL USER MANUAL

Author: Hasan Can Beydili

ABOUT PYERUALJETWORK:

PyerualJetwork 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).

This is HIGH LEVEL user manuel. The functions selected here are those that are as abstracted from the background as possible. Guide for LOW LEVEL users will come soon.

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 user-friendly approach. PLAN's goal is to develop artificial neural network models that are **"as simple & explainable as a perceptron yet as powerful in learning capabilities as multi-layer perceptrons."** For more detailed information, you can check out:

https://github.com/HCB06/PyerualJetwork/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 pyerualjetwork" 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 pyerualjetwork ". (Visual Studio Code reccomended) 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 pyerualjetwork --upgrade". The latest version was “4.4” 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 pyerualjetwork import plan”. Now, you can call the necessary functions from the plan module.

LIBRARY ARCHITECTURE:

The functions of the PyerualJetwork modules, uses snake_case written style.

Main Modules and Functions:

1. **plan & plan_cuda**
    a. fit()
    b. evaluate ()
    c. learner()

2. **planeat & planeat_cuda**
    a. define_genoms()
    b. evaluate()
    c. evolver()

Supportive Modules and Functions:

1. **data_operations & data_operations_cuda**
    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 & model_operations_cuda**
    a. save_model()
    b. load_model()
    c. predict_model_ram()
    d. predict_model_ssd()
    e. reverse_predict_model_ram()
    f. reverse_predict_model_ssd()
    g. get_weights()
    h. get_scaler()
    i. get_preds()

j. get_acc()
k. get_act_pot()

3. **memory_operations**
   a. transfer_to_gpu ()
   b. transfer_to_cpu ()

NOTE:

Non-cuda modules uses 'numpy' arrays(as 'np'), cuda modules uses 'cupy' arrays(as 'cp').
everything else is almost the same. There are some extra parameters only in the functions of cuda modules.
cuda modules runs at GPU, non-cuda modules runs at CPU.

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, PLANEAT (Genetic Algorithm like-NEAT), 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 explainable 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.

```
    fit Args:

x_train (array-like[num]): List or numarray of input data.

y_train (aray-like[num]): List or numarray of target labels. (one hot encoded)

activation_potentiation (list): For deeper PLAN networks, activation function parameters. For more information please run this code: activation_functions.activations_list() default: [None] (optional)

W (numpy.ndarray): If you want to re-continue or update model

auto_normalization (bool, optional): Normalization may solves overflow problem. Default: False

dtype (np.dtype, cp.dtype):  Data type for the arrays. np.float32 by default. Example: np.float64 or np.float16. [fp32 for balanced devices, fp64 for strong devices, fp16 for weak devices: not reccomended!] (optional) dtype=np.float32, dtype=cp.float32


    Returns:
        numpyarray([num]): (Weight matrix).
```

The output of this function Weight matrix of model.

2. plan.evaluate()

Evaluates the neural network model using the given test data.

```
    Args:
        x_test (np.ndarray): Test data.

        y_test (np.ndarray): Test labels (one-hot encoded).

        W (np.ndarray): Neural net weight matrix.

        activation_potentiation (list): Activation list. Default
= ['linear'].

    Returns:
        tuple: Model (list).
```

3. plan.learner()

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

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

```
    Args:
        x_train (array-like): Training input data.

        y_train (array-like): Labels for training data.

        optimizer (function): PLAN optimization technique with
    hyperparameters. (PLAN using NEAT(PLANEAT) for optimization.) Please
    use this: from pyerualjetwork import planeat (and) optimizer = lambda
    *args, **kwargs: planeat.evolve(*args, 'here give your neat
    hyperparameters for example:  activation_add_prob=0.85', **kwargs)
    Example:

optimizer = lambda *args, **kwargs: planeat.evolver(*args, **kwargs)

model = plan.learner(x_train,
                     y_train,
                     optimizer,
                     fit_start=True,
                     show_history=True,
                     gen=15,
                     batch_size=0.05,
                     interval=16.67)

        fit_start (bool, optional): If the fit_start parameter is
    set to True, the initial generation population undergoes a simple
    short training process using the PLAN algorithm. This allows for a
    very robust starting point, especially for large and complex
    datasets. However, for small or relatively simple datasets, it may
    result in unnecessary computational overhead. When fit_start is True,
    completing the first generation may take slightly longer (this
    increase in computational cost applies only to the first generation
    and does not affect subsequent generations). If fit_start is set to
```

False, the initial population will be entirely random. Options: True or False. Default: True

```

        **weight_evolve (bool, optional):** Activation combinations already optimizes by PLANEAT genetic search algorithm. Should the weight parameters also evolve or should the weights be determined according to the aggregating learning principle of the PLAN algorithm? Default: True (Evolves Weights)

        **gen (int, optional):** The generation count for genetic optimization. Default: max length of activation list in library.

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

        **pop_size (int, optional):** Population size of each generation. Default: count of activation functions

        **auto_normalization (bool, optional):** Normalization may solves overflow problem. Default: False

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

        **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

        **show_history (bool, optional):** If True, displays the training history after optimization. Default is False.

        **acc_impact (float, optional):** Impact of accuracy for optimization [0-1]. Default: 0.9

        **loss_impact (float, optional):** Impact of loss for optimization [0-1]. Default: 0.1
```

```
mm.         loss (str, optional): options: ('categorical_crossentropy'
    or 'binary_crossentropy') Default is 'categorical_crossentropy'.
nn.
oo.         interval (int, optional): The interval at which
    evaluations are conducted during training. (33.33 = 30 FPS, 16.67 =
    60 FPS) Default is 100.
pp.
qq.         target_acc (int, optional): The target accuracy to stop
    training early when achieved. Default is None.
rr.
ss.         target_loss (float, optional): The target loss to stop
    training early when achieved. Default is None.
tt.
uu.         start_this_act (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
vv.
ww.         start_this_W (numpy.array, 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
    weight matrix of this genome. Default is None
xx.
yy.         neurons_history (bool, optional): Shows the history of
    changes that neurons undergo during the TFL (Test or Train Feedback
    Learning) stages. True or False. Default is False.
zz.
aaa.            dtype (np.dtype, cp.dtype):  Data type for the arrays.
    np.float32 by default. Example: np.float64 or np.float16. [fp32 for
    balanced devices, fp64 for strong devices, fp16 for weak devices: not
    reccomended!] (optional) dtype=np.float32, dtype=cp.float32
bbb.
ccc.        memory (str, optional): The memory parameter determines
    whether the dataset to be processed on the GPU will be stored in the
    CPU's RAM or the GPU's RAM. Options:  'gpu', 'cpu'. Default: 'gpu'.
ddd.
eee.      Returns:
fff.            tuple: A list for model parameters: [Weight matrix, Test
    loss, Test Accuracy, [Activations functions]].
ggg.
hhh.
```

Returns: model(list)

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 **PLAN-like models** 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.

```
    Initializes a population of genomes, where each genome is represented by a set of weights

    and an associated activation function. Each genome is created with random weights and activation
    functions are applied and normalized. (Max abs normalization.)

    Args:
        input_shape (int): The number of input features for the neural network.
        output_shape (int): The number of output features for the neural network.
        population_size (int): The number of genomes (individuals) in the population.

        dtype (np.dtype, cp.dtype):  Data type for the arrays. np.float32 by default. Example: np.float64 or np.float16. [fp32 for balanced devices, fp64 for strong devices, fp16 for weak devices: not reccomended!] (optional) dtype=np.float32, dtype=cp.float32


    Returns:
        tuple: A tuple containing:
            - population_weights (numpy.ndarray): A 2D numpy array of shape (population_size, output_shape, input_shape) representing the
                weight matrices for each genome.
            - population_activations (list): A list of activation functions applied to each genome.

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

    Notes:
        The weights are initialized randomly within the range [-1, 1].
```

```
z.       Activation functions are selected randomly from a predefined list
   `activations_list()`.
aa.      The weights for each genome are then modified by applying the
   corresponding activation function
bb.      and normalized using the `normalization()` function. (Max abs
   normalization.)
```

2. planeat.evaluate ()

Making predictions for each genome

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

nn.    Returns:
oo.        list: A list of outputs corresponding to each genome in the
pp.
qq.    Example:
rr.        ```python
ss.        outputs = evaluate(x_population, weights,
   activation_potentiations)
tt.        ```
uu.
vv.    - The function returns a list of outputs after processing the
   population, where each element corresponds to
ww.      the output for each genome in `x_population`.
xx.
```

3. planeat.evolver()

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

```
yy.    Applies the learning process of a population of genomes using
   selection, crossover, mutation, and activation function potentiation.
zz.    The function modifies the population's weights and activation
   functions based on a specified policy, mutation probabilities, and
   strategy.
aaa.
bbb.     Args:

weights (numpy.ndarray): Array of weights for each genoms. (first
returned value of define_genomes function)

activation_potentiations (list): A list of activation functions for
each genomes. (second returned value of define_genomes function)

what_gen (int): The current generation number, used for informational
purposes or logging.

fitness (numpy.ndarray): A 1D array containing the fitness values of
each genome. The array is used to rank the genomes based on their
performance. PLANEAT maximizes or minimizes this fitness based on the
`target_fitness` parameter.

fitness_bias (float, optional): Fitness bias must be a probability
value between 0 and 1 that determines the effect of fitness on the
crossover process. Default: 1.

weight_evolve (bool, optional): Are weights to be evolves or just
activation combinations Default: True. Note: Regardless of whether this
parameter is True or False, you must give the evolver function a list of
weights equal to the number of activation potentiations. You can create
completely random weights if you want. If this parameter is False, the
weights entering the evolver function and the resulting weights will be
exactly the same.
```

show_info (bool, optional): If True, prints information about the current generation and the maximum reward obtained. Also shows current configuration. Default is False.

strategy (str, optional): The strategy for combining the best and bad genomes. Options:
        - 'normal_selective': Normal selection based on fitness
, where a portion of the bad genes are discarded.
        - 'more_selective': A more selective strategy, where fewer bad genes survive.
        - 'less_selective': A less selective strategy, where more bad genes survive.
        Default is 'normal_selective'.

bar_status (bool, optional): Loading bar status during evolving process of genomes. True or False. Default: True

policy (str, optional): The selection policy that governs how genomes are selected for reproduction. Options:

           - 'aggressive': Aggressive policy using very aggressive selection policy.
                Advantages: fast training.
                Disadvantages: may lead to fitness stuck in a local maximum or minimum.

           - 'explorer': Explorer policy increases population diversity.
                Advantages: fitness does not get stuck at local maximum or minimum.
                Disadvantages: slow training.

                Suggestions: Use hybrid and dynamic policy. When fitness appears stuck, switch to the 'explorer' policy.

           Default: 'aggressive'.
.
ccc.
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 mutatation 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.

ddd.
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)

bad_genomes_selection_prob (float, optional): The probability of crossover parents are bad genomes ? [0-1] Default: Determined by `policy`.

cross_over_mode (str, optional): Specifies the crossover method to use. Options:
- 'tpm': Two-Point Matrix Crossover
Default is 'tpm'.

activation_mutate_add_prob (float, optional): The probability of adding a new activation function to the genome for mutation.
Must be in the range [0, 1]. Default is 0.5.

activation_mutate_delete_prob (float, optional): The probability of deleting an existing activation function
from the genome for mutation. Must be in the range [0, 1]. Default is 0.5.

activation_mutate_change_prob (float, optional): The probability of changing an activation function in the genome for mutation.
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 (%100).

weight_mutate_threshold (int): Determines max how much weight mutaiton operation applying. (Function automaticly determines to min) Default: 16

activation_selection_add_prob (float, optional): The probability of adding an existing activation function for crossover.
Must be in the range [0, 1]. Default is 0.5. (WARNING! Higher values increase complexity. For faster training, increase this value.)

activation_selection_change_prob (float, optional): The probability of changing an activation function in the genome for crossover.
Must be in the range [0, 1]. Default is 0.5.

activation_mutate_threshold (int, optional): Determines max how much activation mutaiton operation applying. (Function automaticly determines to min) Default: 2

activation_selection_threshold (int, optional): Determines max how much activaton transferable to child from undominant parent. (Function automaticly determines to min) Default: 2

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

```
dtype (np.dtype, cp.dtype):  Data type for the arrays. np.float32 by
default. Example: np.float64 or np.float16. [fp32 for balanced devices,
fp64 for strong devices, fp16 for weak devices: not reccomended!]
(optional) dtype=np.float32, dtype=cp.float32


eee.         Raises:
fff.             ValueError:
ggg.     - If `policy` is not one of the specified values ('aggressive',
   'explorer').
hhh.             - If 'strategy' is not one of the specified values
   ('less_selective', 'normal_selective', 'more_selective')
iii.             - If `cross_over_mode` is not one of the specified values
   ('tpm').
jjj.             - If `bad_genomes_mutation_prob`,
   `activation_mutate_prob`, or other probability parameters are not in the
   range 0 and 1.
kkk.             - If the population size is odd (ensuring an even number
   of genomes is required for proper selection).
lll.             - If 'fitness_bias' value is not in range 0 and 1.
mmm.
nnn.
ooo.         Returns:
ppp.             tuple: A tuple containing:
qqq.                 - weights (numpy.ndarray): The updated weights for the
   population after selection, crossover, and mutation.
rrr.                                          The shape is
   (population_size, output_shape, input_shape).
sss.                 - activation_potentiations (list): The updated list of
   activation functions for the population.
ttt.
uuu.         Notes:
vvv.             - **Selection Process**:
www.                 - The genomes are sorted by their fitness (based on
   `fitness`), and then split into "best" and "bad" half.
xxx.                 - The best genomes are retained, and the bad genomes
   are modified based on the selected strategy.
yyy.
zzz.             - **Crossover Strategies**:
aaaa.                - 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.
bbbb.
cccc.            - **Mutation**:
dddd.                - Mutation is applied to both the best and bad
   genomes, depending on the mutation probability and the `policy`.
eeee.                - `bad_genoms_mutation_prob` determines the
   probability of applying mutations to the bad genomes.
```

```
ffff.                - If `activation_mutate_prob` is provided, activation
   function mutations are applied to the genomes based on this probability.
gggg.
-               - **Population Size**: The population size must be an even
number to properly split the best and bad genomes. If `fitness` 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:
hhhh.             ```python
iiii.     weights, activation_potentiations = planeat.evolver(weights,
   activation_potentiations, 1, fitness, show_info=True,
   strategy='normal_selective', policy='aggressive')
jjjj.             ```
kkkk.
llll.         - The function returns the updated weights and activations
   after processing based on the chosen strategy, policy, and mutation
   parameters.
mmmm.       """
nnnn.
```

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.
c.
d. dtype (np.dtype, cp.dtype):  Data type for the arrays. np.float32 by
   default. Example: np.float64 or np.float16. [fp32 for balanced devices,
   fp64 for strong devices, fp16 for weak devices: not reccomended!]
   (optional) dtype=np.float32, dtype=cp.float32
e.
f. memory (str): The memory parameter determines whether the dataset to
   be processed on the GPU will be stored in the CPU's RAM or the GPU's
   RAM. Options:  'gpu', 'cpu'. Default: 'gpu'.
g.
h. shuffle_in_cpu (bool): If True, output will be same cpu's
   auto_balancer function. (Use this for direct comparison of cpu
   training.) Default: False.
i.
j.
```

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
c.
d. dtype (np.dtype, cp.dtype):  Data type for the arrays. np.float32 by
   default. Example: np.float64 or np.float16. [fp32 for balanced devices,
```

```
   fp64 for strong devices, fp16 for weak devices: not reccomended!]
   (optional) dtype=np.float32, dtype=cp.float32
e.
f. shuffle_in_cpu (bool): If True, output will be same cpu's
   auto_balancer function. (Use this for direct comparison of cpu
   training.) Default: False.
g.
```

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.
g. y_test (numpy.ndarray): Labeled test data.
h.
i. summary (bool): If True, prints the class-to-index mapping.
   Default: False
j.
k.
```

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. dtype (np.dtype, cp.dtype):  Data type for the arrays. np.float32 by
   default. Example: np.float64 or np.float16. [fp32 for balanced devices,
   fp64 for strong devices, fp16 for weak devices: not reccomended!]
   (optional) dtype=np.float32, dtype=cp.float32
e.
f. shuffle_in_cpu (bool): If True, output will be same cpu's
   auto_balancer function. (Use this for direct comparison of cpu
   training.) Default: False.
g.
h.
i. test_size (float or int): Proportion or number of samples for the
   test subset.
j.
k. 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.manuel_balancer ()

Same operation of auto_balacner, 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
f. dtype (np.dtype, cp.dtype):  Data type for the arrays. np.float32 by
   default. Example: np.float64 or np.float16. [fp32 for balanced devices,
   fp64 for strong devices, fp16 for weak devices: not reccomended!]
   (optional) dtype=np.float32, dtype=cp.float32
g.
h. shuffle_in_cpu (bool): If True, output will be same cpu's
   auto_balancer function. (Use this for direct comparison of cpu
   training.) Default: False.
i.
```

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)
f.
g. dtype (np.dtype, cp.dtype):  Data type for the arrays. np.float32
   by default. Example: np.float64 or np.float16. [fp32 for balanced
   devices, fp64 for strong devices, fp16 for weak devices: not
   reccomended!] (optional) dtype=np.float32, dtype=cp.float32
h.
```

Returns: If x_test and x_train given but scaler_params not given then returns: standart scaled parameters, standard scaled x_train, standard scaled y_test. If x_test is not given and x_train given but scaler_params not given then returns: standard scaled parameters, standard scaled x_train. If just one x_test(your real-world sample) and scaler_params given then returns scaled x_test(your real-world sample).

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.

```
oooo.    Function to save a potentiation learning model.
pppp.
qqqq.        Arguments:
rrrr.
ssss.    model_name (str): Name of the model.
tttt.
uuuu.    model_type (str): Type of the model. default: 'PLAN'
vvvv.
wwww.    test_acc (float): Test accuracy of the model. default: None
xxxx.
yyyy.    weights_type (str): Type of weights to save (options: 'txt',
   'pkl', 'npy', 'mat'). default: 'npy'
zzzz.
aaaaa.   weights_format (str): Format of the weights (options: 'f',
   'raw'). default: 'raw'
bbbbb.
ccccc.   model_path (str): Path where the model will be saved. For
   example: C:/Users/beydili/Desktop/denemePLAN/ default: ''
ddddd.
eeeee.   scaler_params (num[num, num]): standard scaler params list:
   mean,std. If not used standard scaler then be: None.
fffff.
ggggg.   W: Weights of the model.
hhhhh.
```

```
iiiii.   activation_potentiation (list): For deeper PLAN networks,
   activation function parameters. For more information please run this
   code: activation_functions.activations_list() default: ['linear']
jjjjj.
kkkkk.   show_architecture (bool): It draws model architecture. True or
   False. Default: False
lllll.
mmmmm.   show_info (bool): Prints model details into console. default:
   True
nnnnn.
ooooo.
ppppp.       Returns:
qqqqq.       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.
c. model_name (str): Name of the model.
d.
e. model_path (str):  Path of the model. Default: ‘’
f.
g. dtype (np.dtype, cp.dtype):  Data type for the arrays.
   np.float32 by default. Example: np.float64 or np.float16.
   [fp32 for balanced devices, fp64 for strong devices, fp16
   for weak devices: not reccomended!] (optional)
   dtype=np.float32, dtype=cp.float32
h.
i.
```

This function returns the 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 (numpy.array): standard scaler params list: mean,std.
   (optional) Default: None.
f.
g. activation_potentiation (list): ac list for deep PLAN. default:
   ['linear'] (optional)
h.
i. dtype (np.dtype, cp.dtype):  Data type for the arrays. np.float32 by
   default. Example: np.float64 or np.float16. [fp32 for balanced devices,
   fp64 for strong devices, fp16 for weak devices: not reccomended!]
   (optional) dtype=np.float32, dtype=cp.float32
j.
```

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

5. model_operations.reverse_predict_model_ssd()

This function loads the model directly from its saved location, predicts a requested output, and returns the input. It using reverse run.

```
a. output (list or ndarray): output layer for the model
   (single probability vector, output layer of trained model).
b.
c. model_name (str): Name of the model.
d.
e. model_path (str):  Path of the model. Default: ‘’
f.
g. dtype (np.dtype, cp.dtype):  Data type for the arrays.
   np.float32 by default. Example: np.float64 or np.float16.
   [fp32 for balanced devices, fp64 for strong devices, fp16
   for weak devices: not reccomended!] (optional)
   dtype=np.float32, dtype=cp.float32
h.
```

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

6. model_operations.reverse_predict_model_ram()

This function predicts and returns the input for a requested output using a model that has already been loaded into the program (located in the computer's RAM). It using reverse run.

```
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. dtype (np.dtype, cp.dtype):  Data type for the arrays. np.float32 by
   default. Example: np.float64 or np.float16. [fp32 for balanced devices,
   fp64 for strong devices, fp16 for weak devices: not reccomended!]
   (optional) dtype=np.float32, dtype=cp.float32
f.
g.
```

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

7. plan.get_weights()

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

test_model = plan.evaluate(x_test, y_test)

W = test_model[plan.get_weights()]

8. 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()]

9. plan.get_preds()

Returns predictions list of the selected model

10. plan.get_acc()

Returns accuracy of the selected model

11. plan.get_act_pot()

Returns activation potential of the selected model.

MEMORY OPERATIONS MODULE FUNCTIONS

1. memory_operations.transfer_to_gpu()

```
h. The `transfer_to_gpu` function in Python converts input data to GPU
   arrays, optimizing memory usage by
i.     batching and handling out-of-memory errors.
j.
k.    X: The `x` parameter in the `transfer_to_gpu` function is the input
   data that you want to transfer to the GPU for processing. It can be
```

```
   either a NumPy array or a CuPy array. If it's a NumPy array, the
   function will convert it to a CuPy array and
l.
m.    dtype: The `dtype` parameter in the `transfer_to_gpu` function
   specifies the data type to which the input array `x` should be converted
   when moving it to the GPU. By default, it is set to `cp.float32`, which
   is a 32-bit floating-point data type provided by the CuPy
n.
o.    Return: The `transfer_to_gpu` function returns the input data `x`
   converted to a GPU array of type `dtype` (default is `cp.float32`). If
   the input `x` is already a GPU array with the same dtype, it returns `x`
   as is. If the data size of `x` exceeds 25% of the free GPU memory, it
   processes the data in batches to
p.
q.
```

2. memory_operations.transfer_to_cpu()

```
r.     The `transfer_to_cpu` function converts data to a specified data
   type on the CPU, handling memory constraints
s.     by batching the conversion process and ensuring complete GPU memory
   cleanup.
t.
u.    x: Input data to transfer to CPU (CuPy array)
v.
w.    dtype: Target NumPy dtype for the output array (default: np.float32)
x.
y.    Return: NumPy array with the specified dtype
```

LAST PART:

Despite being in its early stages of development, PyerualJetwork 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 PyerualJetwork, 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 integratd into the PyerualJetwork library. For your suggestions, lists and feedback, my e-mail address is: tchasancan@gmail.com

Trust the PLAN…