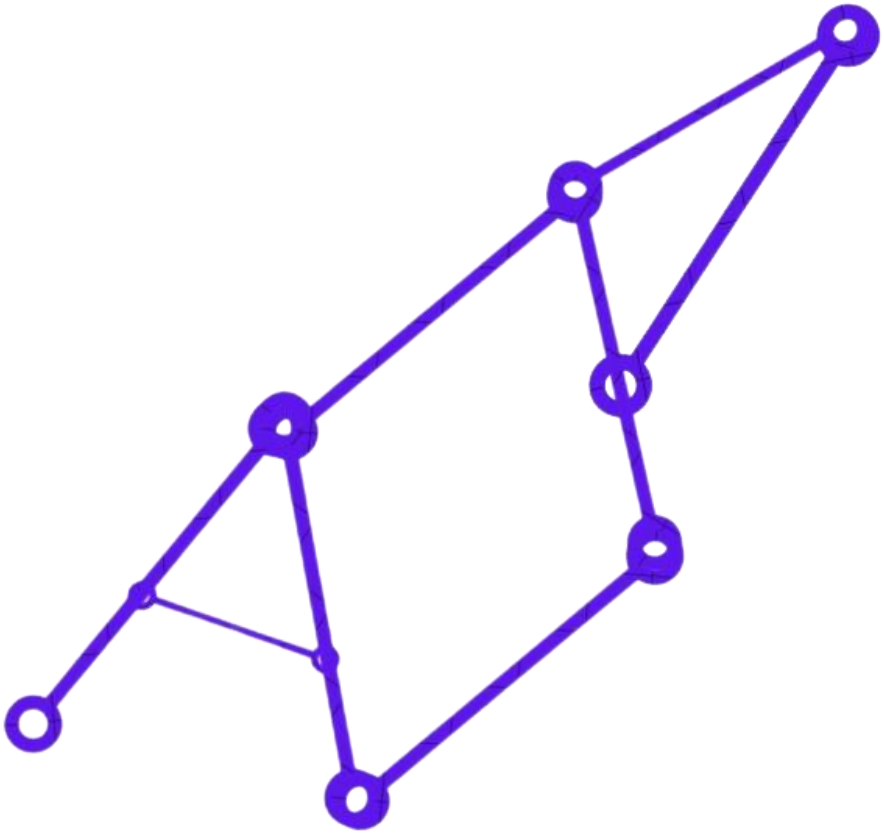

PYERUALJETWORK 5
HIGH LEVEL USER MANUAL

Author: Hasan Can Beydili

ABOUT PYERUALJETWORK:

PyereualJetwork is a GPU-accelerated machine learning library in Python for professionals and researchers. It features PLAN(Potentiation Learning artificial Neural Network), MLP, Deep Learning training, and ENE (Eugenic NeuroEvolution) for genetic optimization, applicable to genetic algorithms or Reinforcement Learning (RL). The library includes data pre-processing, visualizations, model saving/loading, prediction, evaluation, training, and detailed or simplified memory management.

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.

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/MODEL_OPERATIONS_CPU.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".

After installing the module using "pip" you can now call the library module in your project environment. For example: “from pyerualjetwork import neu_cpu”. Now, you can call the necessary functions from the neu_cpu module.

LIBRARY ARCHITECTURE:

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

Main Modules and Functions:

1. **neu_cpu & neu_cuda**
    a. plan_fit()
    b. evaluate()
    c. learn()

2. **ene_cpu & ene_cuda**
    a. define_genomes()
    b. evaluate()
    c. evolver()

Supportive Modules and Functions:

1. **data_operations_cpu & 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_cpu & model_operations_cuda**
    a. save_model()
    b. load_model()
    c. predict_from_memory()
    d. predict_from_storage()
    e. reverse_predict_from_memory()
    f. reverse_predict_from_storage()
    g. get_weights()
    h. get_scaler()
    i. get_preds()

j. get_acc()
k. get_act()
l. get_model_type()
m. get_weights_type()
n. get_weights_format()
o. get_model_version()
p. get_model_df()

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.

NEU MODULE

This module hosts functions for training and evaluating artificial neural networks on CPU or GPU for labeled classification tasks (for now).

Currently, two types of models can be trained:

PLAN (Potentiation Learning Artificial Neural Network)
MLP (Multi-Layer Perceptron → Deep Learning) -- With non-bias

NEU MODULE FUNCTIONS

1. neu_cpu.plan_fit()

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

```
a.      plan_fit Args:
b.
c. x_train (array-like[num]): List or numarray of input data.
d.
e. y_train (aray-like[num]): List or numarray of target labels. (one
   hot encoded)
f.
g. activations (list): For deeper PLAN networks, activation function
   parameters. For more information please run this code:
   activation_functions.activations_list() default: [None] (optional)
h.
i. W (numpy.ndarray): If you want to re-continue or update model
j.
k. auto_normalization (bool, optional): Normalization may solves
   overflow problem. Default: False
l.
m. 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
n.
o.
p.     Returns:
q.         numpyarray([num]): (Weight matrix).
```

The output of this function Weight matrix of model.

2. neu_cpu.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.

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

        is_mlp (bool, optional): Evaluate PLAN model or MLP model
 ?
        Default: False (PLAN)

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

3. neu_cpu.learn()

Optimizes the activation functions for a neural network by leveraging train data to find

the most accurate combination of activation potentiation(or activation function) & weight values for the given dataset.

Why genetic optimization ENE(Eugenic NeuroEvolution) and not backpropagation?

Because PLAN is different from other neural network architectures. In PLAN, the learnable parameters are not the weights; instead, the learnable parameters are the activation functions.

Since activation functions are not differentiable, we cannot use gradient descent or backpropagation. However, I developed a more powerful genetic optimization algorithm: ENE.

This function also able to train MLP(Multi Layer Perceptrons) with ENE(Eugenic NeuroEvolution).

```
    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 = neu_cpu.learn(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 train 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
ff.
gg.         show_history (bool, optional): If True, displays the
    training history after optimization. Default is False.
hh.
ii.         acc_impact (float, optional): Impact of accuracy for
    optimization [0-1]. Default: 0.9
jj.
kk.         loss_impact (float, optional): Impact of loss for
    optimization [0-1]. Default: 0.1
ll.
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.        neurons: (list[int], optional): If you dont want train
    PLAN model this parameter represents neuron count of each hidden
    layer for MLP. Number of elements --> Layer count. Default: [] (No
    hidden layer) --> architecture setted to PLAN, if not -->
    architecture setted to MLP.
bbb.
ccc.        activation_functions: (list[str], optional): If you
    dont want train PLAN model this parameter represents activation
```

```
   function of each hidden layer for MLP. if neurons is not [] --> uses
   default: ['linear'] * len(neurons). if neurons is [] --> uses [].

ddd.
eee.          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
fff.
ggg.          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'.
hhh.
iii.      Returns:
jjj.          tuple: A list for model parameters: [Weight matrix, Test
   loss, Test Accuracy, [Activations functions]].
kkk.
lll.
```

Returns: model(list)

ENE MODULE

This module contains all the functions necessary for implementing and testing the ENE (Eugenic NeuroEvolution) algorithm on CPU or GPU.

ENE MODULE FUNCTIONS

1. ene_cpu.define_genomes ()

Creates ENE 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.

        neurons (list[int], optional): If you dont want train PLAN model this parameter represents neuron count of each hidden layer for MLP. Default: None (PLAN)

        activation_functions (list[str], optional): If you dont want train PLAN model this parameter represents activation function of each hidden layer for MLP. Default: None (PLAN) NOTE: THIS EFFECTS HIDDEN LAYERS OUTPUT. NOT OUTPUT LAYER!

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

2. ene_cpu.evaluate ()

Making predictions for each genome

```
    Evaluates the performance of a population of genomes, applying
    different activation functions
    and weights depending on whether reinforcement learning mode is
    enabled or not.

Args:

Input (list or numpy.ndarray): A list or 2D numpy array where each
    element represents a genome (A list of input features for each genome,
    or a single set of input features for one genome).

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.


activations (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.

    Is_mlp (bool, optional): Evaluate PLAN model or MLP model ? Default:
    False (PLAN)
```

```
    Returns:
        list: A list of outputs corresponding to each genome in the

    Example:
        ```python
        outputs = evaluate(Inputs, weights, activations)
        ```

    - The function returns a list of outputs after processing the
   population, where each element corresponds to
            the output for each genome in population.
```

3. ene_cpu.evolver()

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

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

    Args:

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

activations (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'.
.
ggg.

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.

hhh.
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.

```
save_best_genome (bool, optional): Save the best genome of the
previous generation to the next generation. Default: False

is_mlp (bool, optional): Evolve PLAN model or MLP model ? Default: False
(PLAN)

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
is_mlp (bool, optional): Evolve PLAN model or MLP model ? Default: False
(PLAN)

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


iii.         Raises:
jjj.             ValueError:
kkk.     - If `policy` is not one of the specified values ('aggressive',
   'explorer').
lll.             - If 'strategy' is not one of the specified values
   ('less_selective', 'normal_selective', 'more_selective')
mmm.             - If `cross_over_mode` is not one of the specified values
   ('tpm').
nnn.             - If `bad_genomes_mutation_prob`,
   `activation_mutate_prob`, or other probability parameters are not in the
   range 0 and 1.
ooo.             - If the population size is odd (ensuring an even number
   of genomes is required for proper selection).
ppp.             - If 'fitness_bias' value is not in range 0 and 1.
qqq.
rrr.
sss.         Returns:
ttt.             tuple: A tuple containing:
uuu.                 - weights (numpy.ndarray): The updated weights for the
   population after selection, crossover, and mutation.
vvv.                                          The shape is
   (population_size, output_shape, input_shape).
www.                 - activation_potentiations (list): The updated list of
   activation functions for the population.
xxx.
yyy.         Notes:
zzz.             - **Selection Process**:
```

```
aaaa.                - The genomes are sorted by their fitness (based on
   `fitness`), and then split into "best" and "bad" half.
bbbb.                - The best genomes are retained, and the bad genomes
   are modified based on the selected strategy.
cccc.
dddd.            - **Crossover Strategies**:
eeee.                - 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.
ffff.
gggg.            - **Mutation**:
hhhh.                - Mutation is applied to both the best and bad
   genomes, depending on the mutation probability and the `policy`.
iiii.                - `bad_genoms_mutation_prob` determines the
   probability of applying mutations to the bad genomes.
jjjj.                - If `activation_mutate_prob` is provided, activation
   function mutations are applied to the genomes based on this probability.
kkkk.
-               - **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:
llll.            ```python
mmmm.     weights, activation_potentiations = ene.evolver(weights,
   activation_potentiations, 1, fitness, show_info=True,
   strategy='normal_selective', policy='aggressive')
nnnn.            ```
oooo.
pppp.        - The function returns the updated weights and activations
   after processing based on the chosen strategy, policy, and mutation
   parameters.
qqqq.       """
rrrr.
```

DATA OPERATIONS MODULE FUNCTIONS

1. data_operations_cpu.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. Args:
b.
c. x_train (list): Input data for training.
d.
e. y_train (list): Labels corresponding to the input data.
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. 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'.
j.
k. 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.
l.
m.
```

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_cpu.synthetic_augmentation()

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

```
a. Args:
b.
c. x_train: numpy array format
d.
e. y_train: numpy array format (one-hot encoded)
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. 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.
j.
```

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_cpu.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_cpu.split()

This function splits all data for train and test

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

Returns: x_train, x_test, y_train, y_test

5. data_operations_cpu. 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_cpu.manuel_balancer ()

Same operation of auto_balacner, but this function gives the limit of sample addition to user.

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

Returns: x_train, y_train

7. data_operations_cpu.standard_scaler()

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

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_cpu.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.

```
    Function to save a potentiation learning model.

        Arguments:

    model_name (str): Name of the model.

    model_type (str): Type of the model. Options: 'PLAN', 'MLP'.

    test_acc (float): Test accuracy of the model. default: None

    weights_type (str): Type of weights to save (options: 'txt',
    'pkl', 'npy', 'mat'). default: 'npy'

    weights_format (str): Format of the weights (options: 'f',
    'raw'). default: 'raw'

    model_path (str): Path where the model will be saved. For
    example: C:/Users/beydili/Desktop/denemePLAN/ default: ''

    scaler_params (num[num, num]): standard scaler params list:
    mean,std. If not used standard scaler then be: None.

    W: Weights of the model.

    activation_potentiation (list): For deeper PLAN networks,
    activation function parameters. For more information please run this
    code: activation_functions.activations_list() default: ['linear']

    show_architecture (bool): It draws model architecture.
     NOTE! draw architecture only works for PLAN models. Not MLP
    models for now, but it will be. True or False. Default: False

    show_info (bool): Prints model details into console. default:
    True


        Returns:
        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_cpu.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, None, test_acc, activations, scaler_params, None, model_type, WeightType, WeightFormat, device_version, df(Pandas dataframe for model)

3. model_operations_cpu.predict_from_storage()

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

```
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.
```

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

4. model_operations_cpu.predict_from_memory()

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.)

```
b. Input (list or ndarray): Input data for the model (single vector or
   single matrix).
c.
d. W (list of ndarrays): Weights of the model.
e.
f. scaler_params (numpy.array): standard scaler params list: mean,std.
   (optional) Default: None.
g.
h. activation_potentiation (list): ac list for deep
   MODEL_OPERATIONS_CPU. default: ['linear'] (optional)
i.
j. İs_mlp (bool, optional): Predict from PLAN model or MLP model ?
   Default: False (PLAN)
```

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

5. model_operations_cpu.reverse_predict_from_storage()

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: ‘’
```

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

6. model_operations_cpu.reverse_predict_from_memory()

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.
```

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

7. model_operations_cpu.get_weights()

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

test_model = neu_cpu.evaluate(x_test, y_test)

W = test_model[model_operations_cpu.get_weights()]

8. model_operations_cpu.get_scaler()

Returns scaler_params of the selected model For exp:

model = neu_cpu.learn(x_train, y_train, optimizer, gen=10)

scaler_params = model[model_operations_cpu.get_scaler()]

9. model_operations_cpu.get_preds()

Returns predictions list of the selected model

10. model_operations_cpu.get_acc()

Returns accuracy of the selected model

11. model_operations_cpu.get_act()

Returns activation function list of the selected model.

12. model_operations_cpu.get_model_type()

Returns weight type of the selected model.

13. model_operations_cpu.get_model_format()

Returns weight format of the selected model.

14. model_operations_cpu.get_model_version()

Returns version of the selected model.

15. model_operations_cpu.get_model_df()

Returns all about the model with Pandas dataframe form.

MEMORY OPERATIONS MODULE FUNCTIONS

1. memory_operations.transfer_to_gpu()

```
d. The `transfer_to_gpu` function in Python converts input data to GPU
   arrays, optimizing memory usage by
e.     batching and handling out-of-memory errors.
f.
g.    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
h.
i.    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
j.
k.    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
l.
m.
```

2. memory_operations.transfer_to_cpu()

```
n.     The `transfer_to_cpu` function converts data to a specified data
   type on the CPU, handling memory constraints
o.     by batching the conversion process and ensuring complete GPU memory
   cleanup.
p.
```

```
    x: Input data to transfer to CPU (CuPy array)

    dtype: Target NumPy dtype for the output array (default: np.float32)

    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) & ENE (Eugenic NeuroEvolution), 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…