Code Documentation

Daniel Sobral Blanco*
Department of Theoretical Physics,
University of Geneva

June 7, 2023

Introduction

This document provides the documentation for each `.py` script used in the code developed for the project. It includes descriptions of classes, methods, and their usage. The instructions to run the code from terminal are contained in the `README.md` file.

The code has been generated with the help of ChatGPT, using both GPT-3.5 and GPT-4 versions. It has proven very useful at the time of writting the scripts and debugging the methods. The docstrings, code documentation and `README.md` file are entirely generated by the model.

run.py

This is the script that runs the code. It can be used from terminal or in another `Python` interface such as `Jupyter` notebooks.

```
def run(config_file, save_model=True, save_history=True, seed=48):
    """
    Run the Galaxy Zoo training and evaluation process.

    Parameters
    ----------
    config_file : str
        Path to the configuration file. Must be a .yaml file.
    save_model : bool, optional
        Whether to save the trained model. Defaults to True.
    save_history : bool, optional
        Whether to save the training history. Defaults to True.
    seed : int, optional
        Seed for reproducibility. Defaults to 48.
    """
```

If `save_model` and `save_history` flags are set to `True` (default value), the code will save the model `.h5` file and its training history in the correspondents paths. One must set this paths in the `config_file` (see the `README.md`). The code also produces plots of the loss and the first metric passed in the `metrics` list. It also generates the confusion matrices.

models.py

This is the main script of the code. It contains a `class` to build, compile and train CNN models with different architectures. It allows to build standard, single output models and multi-output models

*Email: daniel.sobralblanco@unige.ch

with **base** architecture of the form:

$$\text{IN} + n \cdot (\text{Conv2D} + \text{BN} + \text{MaxPool2D}) + \text{Flattening} + n \cdot (\text{Dense} + \text{Maxout} + \text{Dropout}) + \text{OUT}$$

Here `BN` stands for `Batch Normalization`. The `Flattening` part can be chosen to be the standard `Flatten` layer, or a `GlobalMaxPooling2D` layer. `Maxout` operation is fixed to $n = 2$, which averages out every two hidden neurons. The `OUT` layer will be different depending on the type of model one wants to build (see the report for more details about the model types).

GalaxyZooClassifier

Custom class to build classifiers for the Galaxy Zoo challenge. This class builds a custom model architecture, allowing for base, hierarchical, and branch models. Different parameters allow for flexible customization of the architecture, such as number and type of convolutional layers, dense layers, batch normalization, pooling method, and dropout.

```
def __init__(self, model_type, input_shape, n_classes, conv_layers, dense_units,
                batch_normalization = False ,activation='relu',
                  pool_size=(2,2), flattening = 'Flatten', class_weights = False,
                    out_activation = 'softmax', dropout_rate=None, max_out=False,
                    early_stop_patience=None, monitor='val_loss',
                    data_augmentation=None):
    """
    Initializes a GalaxyZooClassifier object.

    Parameters
    ----------
    model_type: str
        The type of model to be built. Should be either 'base', 'hierarchy', or '
            ↪ branch'.
    input_shape: tuple
        The shape of the input data.
    n_classes: int
        The number of classes for classification.
    conv_layers: list of tuples
        The configuration of the convolutional layers, with each tuple containing
            ↪  (number of filters, kernel size).
    dense_units: list of int
        The number of units for each dense layer.
    batch_normalization: bool, optional (default=False)
        Whether to use batch normalization.
    activation: str, optional (default='relu')
        The activation function to use in the layers.
    pool_size: tuple, optional (default=(2, 2))
        The pool size for the max pooling layers.
    flattening: str, optional (default='Flatten')
        The type of flattening to be used before the dense layers.
    class_weights: bool, optional (default=False)
        Whether to use class weights during training.
    out_activation: str, optional (default='softmax')
        The activation function to use in the output layer.
    dropout_rate: float, optional
        The dropout rate to use after each layer.
    max_out: bool, optional (default=False)
        Whether to use Maxout in the dense layers.
```

```
        early_stop_patience: int, optional
            The number of epochs with no improvement after which training will be
                ↪ stopped. If None, early stopping is not used.
        monitor: str, optional (default='val_loss')
            Quantity to be monitored for early stopping.
        data_augmentation: dict, optional
            The parameters for data augmentation. If None, data augmentation is not
                ↪ used.

        Raises
        ------
        ValueError:
            If `model_type` is not 'base', 'hierarchy', or 'branch'.
        """
```

```
    def build_convolutional_blocks(self, input_layer):
        """
        Builds a sequence of convolutional layers based on the `conv_layers`
            ↪ attribute.

        Parameters
        ----------
        input_layer: tensorflow.python.keras.engine.input_layer.InputLayer
            The input layer of the model.



        Returns
        -------
        x: tensorflow.python.keras.engine.base_layer
            The output of the last layer in the block.

        Notes
        -----
        This method builds a sequence of convolutional layers, each followed by an
            ↪ activation function, a max pooling layer,
        and possibly a batch normalization layer, based on the `conv_layers`
            ↪ attribute. The number of convolutional layers,
        their filter sizes, and their kernel sizes are determined by `conv_layers`.
            ↪ After the last convolutional layer,
        the output is flattened if `flattening` is 'Flatten', or global max pooling
            ↪ is applied if `flattening` is not 'Flatten'.
        """
```

```
    def build_dense_blocks(self, input_layer):
        """
        Builds a sequence of fully connected (dense) layers based on the `
            ↪ dense_units` attribute.

        Parameters
        ----------
        input_layer: tensorflow.python.keras.engine.base_layer
            The input layer of the model.

```

```
        Returns
        -------
        x: tensorflow.python.keras.engine.base_layer
            The output of the last layer in the block.

        Notes
        -----
        This method builds a sequence of dense layers, each followed by an activation function, and optionally a dropout layer,
        based on the ‘dense_units‘ attribute. The number of dense layers and their units are determined by ‘dense_units‘.
        If ‘max_out‘ is True, a maxout layer is applied after each dense layer. If ‘dropout‘ is not None, a dropout layer is
        applied after each dense layer with ‘dropout‘ rate.
        """
```

```
    def build_architecture(self, input_layer):
        """
        Builds the architecture of the model by combining convolutional and dense blocks.

        Parameters
        ----------
        input_layer: tensorflow.python.keras.engine.base_layer
            The input layer of the model.

        Returns
        -------
        DenseBlocks: tensorflow.python.keras.engine.base_layer
            The output of the last layer in the dense block.

        Notes
        -----
        This method calls ‘build_convolutional_blocks‘ and ‘build_dense_blocks‘ methods to build the architecture
        of the model. The output of the convolutional blocks is passed as input to the dense blocks.
        """
```

```
    def build_model(self):
        """
        Builds the base model with the defined architecture.

        Returns
        -------
        model: keras.Model
            The base model with the defined architecture.

        Notes
        -----
        This method first creates an input layer with the shape defined in self.in_shape.
        It then calls the ‘build_architecture‘ method to build the architecture of the model.
```

```
        The output layer is a dense layer with a number of units equal to the number
            ↪  of classes,
        and activation function defined in self.out_activation.
        """

    def build_hierarchy_model(self, n_classes, embbedings = False):
        """
        Builds a model with a hierarchical output architecture.

        Parameters
        ----------
        n_classes: tuple
            A tuple specifying the number of classes for each output layer.

        Returns
        -------
        model: keras.Model
            The model with the defined hierarchical architecture.

        Notes
        -----
        This method first creates an input layer with the shape defined in self.
            ↪ in_shape.
        It then calls the ‘build_architecture‘ method to build the architecture of
            ↪ the model.
        The output layers are dense layers with number of units equal to the number
            ↪ of classes
        for each layer, and activation function defined in self.out_activation.
            ↪ These layers
        are then processed to enforce the class hierarchy. The final model has
            ↪ multiple outputs,
        one for each class hierarchy.
        """

    def build_branch_model(self, n_classes):
        """
        Builds a model with a branching output architecture.

        Parameters
        ----------
        n_classes: tuple
            A tuple specifying the number of classes for each output layer.

        Returns
        -------
        model: keras.Model
            The model with the defined branching architecture.

        Notes
        -----
        This method first creates an input layer with the shape defined in self.
            ↪ in_shape.
        It then builds the first level of the model using the ‘build_architecture‘
            ↪ method
```

```
        and adds an output layer. For the second level, it builds two branches each with
        its own architecture and output layer. The outputs of the first level and each
        branch of the second level are concatenated and then processed with dense blocks.
        The outputs of the dense blocks are then processed to enforce the class hierarchy.
        The final model has multiple outputs, one for each branch.
        """

    def compile_model(self, learning_rate, loss_function, metrics=['accuracy']):
        """
        Compiles the model with the specified parameters.

        Parameters
        ----------
        learning_rate: float
            The learning rate to use for the Adam optimizer.

        loss_function: str or dict
            The loss function to use. If it's a string, it is used for all outputs.
            If it's a dictionary, it should map output names to loss functions.

        metrics: list, optional
            The list of metrics to compute during training and testing.
            Default is ['accuracy'].

        Raises
        ------
        Exception
            If the model has not been built before this method is called.

        Notes
        -----
        The method first checks if the model has been built. If not, it raises an exception.
        It then creates an Adam optimizer with the specified learning rate and compiles the model
        with this optimizer, the specified loss function, and the metrics. If there are multiple
        outputs and the loss function is a string, it uses this loss function for all outputs.
        If the loss function is a dictionary, it should map output names to loss functions.
        """

    def fit_model(self, X_train, y_train, X_val, y_val, learning_rate,
                  loss_function = 'categorical_crossentropy', metrics=['accuracy'],
                  batch_size=100, epochs=25, threshold = 0.1, take_weights_log=True):
        """
        Fits the model to the training data.


```

```
        Parameters
        ----------
        X_train: array
            The training inputs.

        y_train: array
            The training targets.

        X_val: array
            The validation inputs.

        y_val: array
            The validation targets.

        learning_rate: float
            The learning rate to use for the Adam optimizer.

        loss_function: str or dict
            The loss function to use. If it's a string, it is used for all outputs.
            If it's a dictionary, it should map output names to loss functions.

        metrics: list, optional
            The list of metrics to compute during training and testing.
            Default is ['accuracy'].

        batch_size: int, optional
            The number of samples per batch. Default is 100.

        epochs: int, optional
            The number of epochs to train the model. Default is 25.

        threshold: float, optional
            The threshold for the class weights. Default is 0.1.

        take_weights_log: bool, optional
            Whether to take the log of the class weights. Default is True.

        Raises
        ------
        Exception
            If the model has not been built and compiled before this method is called
                ↪ .
        Notes
        -----
        The method first checks if the model has been built and compiled. If not, it
            ↪  raises an exception.
        It then compiles the model with the specified parameters and fits it to the
            ↪ training data. It also computes
        class weights if specified, and uses a separate loss function for each
            ↪ output if there are multiple outputs.
        The training process can also include early stopping and data augmentation
            ↪ if specified.
        """
```

```
def save_model(self, models_path, name, diagram = False):
    """
    Saves the model and optionally its architecture diagram.

    Parameters
    ----------
    models_path: str
        The path to the directory where the model should be saved.

    name: str
        The name of the model. The model will be saved as a .h5 file with this
            ↪ name.

    diagram: bool, optional
        Whether to save a diagram of the model's architecture. Default is False.
        If True, the diagram is saved as a .pdf file with the same name as the
            ↪ model.

    Notes
    -----
    The method first saves the model as a .h5 file in the specified directory.
    If the diagram flag is set to True, it also saves a diagram of the model's
        ↪ architecture as a .pdf file in the same directory.
    """
```

The scripts also contain these additional methods:

```
def F1_score(y_true, y_pred):
  """
  Compute the F1 Score between the true and predicted labels.

  The F1 score is the harmonic mean of precision and recall. Compared to the
      ↪ regular mean,
  the harmonic mean gives much more weight to low values. The best value is 1 and
      ↪ the worst is 0.

  Parameters
  ----------
  y_true : array-like
      Ground truth (correct) target values.
  y_pred : array-like
      Estimated targets as returned by a classifier.

  Returns
  -------
  f1_sc : float
      The F1 Score between 0 and 1.

  Notes
  -----
  The F1 score is especially useful for balanced datasets, as it takes both false
      ↪ positives
  and false negatives into account.
  """
```

```
def calculate_class_weights(y_train, threshold, take_weights_log=True):
    """
    Calculate class weights for imbalanced datasets. The function only considers
        ↪ classes that have a representation
    above a certain threshold.

    Parameters
    ----------
    y_train : numpy.ndarray
        The one-hot encoded target variables for the training set. Shape should be (
            ↪ n_samples, n_classes).
    threshold : float
        The minimum representation threshold for a class to be considered valid.
            ↪ Should be between 0 and 1.

    Returns
    -------
    class_weight_dict : dict
        A dictionary mapping from class index to weight. Only includes classes above
            ↪  the representation threshold.
    valid_classes : numpy.ndarray
        An array of the classes that are above the representation threshold.

    Notes
    -----
    If a class is below the representation threshold, it is not included in the
        ↪ output dictionary.
    """
```

eval.py

Script containing utilities for the evaluation of the model performance: calculating the scores, confusion matrices, plotting history, error analysis.

```
def evaluate_my_model(model, test_data, set_name = 'testing'):
    """
    Evaluates a trained model using the provided testing data.

    Parameters
    ----------
    model : keras.Model
        The trained model to be evaluated.
    test_data : list
        A list containing the testing data. The first element of the list
        is expected to be the input data (features), and the second element
        is expected to be the labels (target).
    set_name : str, optional
        A string specifying the name of the testing dataset. It is only used for
        printing purposes to inform the user about the dataset being used for
            ↪ evaluation.
        The default value is 'testing'.
```

```
    Notes
    -----
    This function assumes that the `test_data` list contains exactly two elements.
    The first element is expected to be the input data (features),
    and the second element is expected to be the labels (target).

    This function uses the `evaluate` method of the keras.Model class to evaluate
        ↪ the model.
    The `evaluate` method returns a list of metrics values. The metrics are
        ↪ specified
    during the compilation of the model.

    After the model is evaluated, this function prints the name of each metric
    and its corresponding value.
    """
```

```
def plot_history(history, metric, labels, size=(18, 6), folder_path=None):
    """
    Plots the total training and validation losses and specified metrics from a
        ↪ model's history.

    Parameters
    ----------
    history : keras.callbacks.History object
        The history object generated by the training method `model.fit()`.
    metric : str
        The base name of the metric(s) to be plotted. The function will plot all
            ↪ metrics
        that contain this string in their name.
    labels : tuple
        A tuple containing labels for the generated plot. The tuple is expected
        to have two elements - `model_name` and `learning_rate`.
    size : tuple, optional
        A tuple specifying the size of the generated plot. The default size is (12,
            ↪ 5).
    folder_path : str, optional
        A string specifying the path to the folder where the plot should be saved.
        If this is set to None, the plot is not saved. The default value is None.

    Returns
    -------
    None

    Notes
    -----
    This function generates two plots - one for the total loss and one for the
        ↪ specified metric(s).
    Each plot shows the values of the respective quantity for both the training and
    validation data across the epochs of training. If the model has multiple outputs
        ↪ ,
    the function plots the specified metric(s) for each output on the right subplot.

    If `folder_path` is not None, this function saves the generated plot in that
        ↪ folder
```

```
    as a pdf file. The file is named as `<model_name>_<learning_rate>.pdf`. It is assumed
    that `labels` is a tuple containing exactly two elements - `model_name` and `learning_rate`.

    This function uses matplotlib.pyplot for generating the plots. This must be installed
    in your environment in order to use this function.
    """

def ConfusionMatrix(model, data_test, name, folder_path=None):
    """
    Generates and plots a confusion matrix for the given model and test data.

    Parameters
    ----------
    model : keras Model object
        The model for which the confusion matrix is to be generated.
    data_test : tuple
        A tuple containing the test data. The tuple is expected to contain
        two numpy arrays - `X_test` and `y_test`, where `X_test` is the
        input data and `y_test` are the true labels.
        In the case of multi-output models, `y_test` should be a list
        containing the `y_test[i]` arrays for each output.
    name : str
        A string to be used in the title of the generated plot.
    folder_path : str, optional
        A string specifying the path to the folder where the plot should
        be saved. If this is set to None, the plot is not saved. The default
        value is None.

    Returns
    -------
    None

    Notes
    -----
    This function generates a confusion matrix for the given model and test
    data and plots it using seaborn. It first uses the model to predict labels
    for the test data. It then compares these predicted labels with the true
    labels to generate the confusion matrix.

    The function generates a plot showing the confusion matrix and saves it
    in the specified folder if `folder_path` is not None. The plot is saved
    as a pdf file named ``ConfusionMatrix_<name>.pdf``.

    If `y_test` is a list for multi-output models, the function creates a subplot
    grid based on the number of outputs. Each subplot represents the confusion
    matrix for a specific output. The overall title for the plot indicates the
    name of the model and the type of output. The size of each subplot can be
    adjusted by modifying the `subplot_size` parameter in the function.

    If `y_test` is a single numpy array, the function generates a single plot
    for the confusion matrix.
```

```
    The seaborn and matplotlib libraries must be installed in your environment
    to use this function.
    """

def error_analysis(model, data_test):
    """
    Performs error analysis on the model's predictions and returns indices of the
        ↪ misclassified data.

    Parameters
    ----------
    model : keras Model object
        The trained model which predictions are to be analyzed.
    data_test : tuple
        A tuple containing the test data. The tuple is expected to contain
        two numpy arrays - `X_test` and `y_test`, where `X_test` is the
        input data and `y_test` are the true labels.

    Returns
    -------
    wrong_indices : numpy.ndarray
        An array containing the indices of the test data that were misclassified.
    y_test_labels : numpy.ndarray
        An array of the true labels.
    y_pred_labels : numpy.ndarray
        An array of the predicted labels.

    Notes
    -----
    This function makes predictions on the test data using the provided model,
    then compares the predicted labels with the true labels to identify the indices
    where the model's predictions were incorrect. These indices are then returned
    along with the true and predicted labels for further analysis.
    """

def compare_plot(model, data_test, label_dict, folder_path, num_img = 6):
    """
    Plots and compares a number of randomly selected misclassified test images along
        ↪  with their predicted and true labels.

    Parameters
    ----------
    model : keras Model object
        The trained model which predictions are to be analyzed.
    data_test : tuple
        A tuple containing the test data. The tuple is expected to contain
        two numpy arrays - `X_test` and `y_test`, where `X_test` is the
        input data and `y_test` are the true labels.
    label_dict : dict
        Dictionary mapping the labels represented as integers to their actual values
            ↪ .
    folder_path : str
        Path to the folder where the generated plot will be saved.
```

```
    num_img : int, optional
        Number of misclassified images to be displayed. Default is 6.

    Notes
    -----
    This function selects a number of images that were misclassified by the model,
    then generates a plot displaying these images along with their true and
        ↪ predicted labels.
    The plot is saved to a specified location if ‘folder_path‘ is not None.
    """
```

custom_layers.py

Script containing classes to define the `Maxout` layers and the `MultiOutputDataGenerator`. The latter is a data augmentation generator adapted to work in multi-output set ups.

Maxout

Implements the maxout operation, where the output is the maximum of a group of inputs ($n = 2$ by default). This layer is useful in neural networks to add non-linearity.

```
def __init__(self, group_size, **kwargs):
        """
        Initialize the Maxout layer.

        Parameters
        ----------
        group_size : int
            The number of inputs in each group. The output will be the maximum
            of each group of inputs.

        Notes
        -----
        The __init__ function is special in Python classes. It gets called
        when you create a new instance of the class. In this case, it is
            ↪ initializing
        the group_size attribute and then calling the parent's __init__ function.
        """

def build(self, input_shape):

        """
        Build the Maxout layer.

        Parameters
        ----------
        input_shape : tuple
            The shape of the inputs to this layer.

        Notes
        -----
        The purpose of this method is to validate if the last dimension of the input
            ↪  shape
```

```
        is divisible by the group_size. If not, it raises a ValueError. This
            ↪ function is specific
        to keras.Layer subclasses, which helps in creating the weight variables.
        """

    def call(self, inputs):
        """
        Call the Maxout layer.

        Parameters
        ----------
        inputs : Tensor
            The inputs to the layer.

        Returns
        -------
        outputs : Tensor
            The output of the maxout operation.

        Notes
        -----
        The call method defines the layer's computation. For Maxout, this involves
            ↪ reshaping
        the input tensor and applying the max operation.
        """

    def compute_output_shape(self, input_shape):
        """
        Compute the output shape of the Maxout layer.

        Parameters
        ----------
        input_shape : tuple
            The shape of the inputs to this layer.

        Returns
        -------
        shape : tuple
            The shape of the output of this layer.

        Notes
        -----
        This function calculates the shape of the output of the layer.
        This is required in Keras, to inform the model about the output size of the
            ↪ layer.
        """

    def get_config(self):
        """
        Get the configuration of the Maxout layer.
```

```
        Returns
        -------
        config : dict
            A dictionary containing the configuration of the Maxout layer.

        Notes
        -----
        The get_config method is used when saving the model. It returns a
        dictionary containing the configuration of the layer.
        """

    @classmethod
    def from_config(cls, config):
        """
        Create a Maxout layer from its configuration.

        Parameters
        ----------
        config : dict
            A dictionary containing the configuration of the Maxout layer.

        Returns
        -------
        maxout : Maxout
            A new Maxout layer instance.

        Notes
        -----
        This is a class method that is used to create a new instance of the
        class using the configuration dictionary returned by get_config.
        This is essential for Keras' model saving and loading functionality.
        """
```

MultiOutputDataGenerator

MultiOutputDataGenerator class extends the Sequence class in Keras to generate batches of tensor image data with real-time data augmentation. The data will be looped over (in batches) indefinitely. This class allows the handling of multiple output labels and uses the Keras ImageDataGenerator for the actual data augmentation.

```

    def __init__(self, x_set, y_set, batch_size, parameters, shuffle = True):
        """
        Parameters
        ----------
        x_set : np.ndarray
            Input data.
        y_set : list of np.ndarray or np.ndarray
            List of output data in case of multiple outputs or single output data in
                ↪ case of single output.
        batch_size : int
            Number of samples per gradient update.
        parameters : dict
```

```
14          Parameters to be used for data augmentation by the internal
                ↪ ImageDataGenerator instance.
15          For more details on the parameters refer to the documentation of
                ↪ ImageDataGenerator.
16
17      Notes
18      -----
19      In this class, we assume that ‘y_set‘ is a list of numpy arrays for multiple
            ↪  outputs or a single numpy
20      array for a single output. If you are providing a different format for ‘
            ↪ y_set‘, you might need to adjust
21      the ‘__getitem__‘ method accordingly.
22      """
23
24  def __len__(self):
25      """
26      Get the number of batches per epoch.
27
28      Returns
29      -------
30      int
31          The number of batches per epoch.
32      """
33
34  def __getitem__(self, idx):
35      """
36      Generates a batch of augmented data.
37
38      This method is called for every mini-batch and it should return a complete
            ↪ batch each time.
39      It allows the generator to give the model a type of 'view' on the dataset,
            ↪ returning a batch of
40      inputs (and targets) at each call. In this implementation, the method also
            ↪ performs data augmentation
41      using the ImageDataGenerator instance.
42
43      Parameters
44      ----------
45      idx : int
46          Index of the mini-batch. It ranges between 0 and the total number of
                ↪ batches.
47
48      Returns
49      -------
50      tuple
51          A tuple of two elements. The first element is a numpy array of input data
                ↪  after augmentation.
52          If there are multiple outputs, the second element is a list of numpy
                ↪ arrays corresponding to each
53          output. If there's a single output, the second element is a numpy array
                ↪ for that output.
54
55
```

```
        Notes
        -----
        For multi-output models, the second element of the tuple is a list of numpy arrays. The order of the
        numpy arrays in the list is the same as the order of the outputs in the model.

        In the case of single-output models, the second element of the tuple is a single numpy array
        corresponding to the output of the model.
        """

    def on_epoch_end(self):
        """
        Shuffle the data at the end of every epoch.

        Notes
        -----
        The method is called at the end of every epoch and is used to shuffle the input and output data to
        ensure the model doesn't learn any unintended sequential patterns from the data.
        """

    def save(self, filename):
        """
        Save the current state of the data generator to a file.

        Parameters
        ----------
        filename : str
            The name of the file where the state of the data generator will be saved.
        """

    @classmethod
    def load(cls, filename):
        """
        Load the state of a data generator from a file.

        Parameters
        ----------
        filename : str
            The name of the file from which the state of the data generator will be loaded.

        Returns
        -------
        MultiOutputDataGenerator
            A new instance of MultiOutputDataGenerator with its state loaded from the file.
        """
```

custom_losses.py

This script contains three classes that define custom loss functions: the focal loss and weighted versions of the mean squared error and categorical cross_entropy. The standard Keras losses have been adapted to work in the case of multi-output set ups. Weights are applied 'manually' instead than with the standard Keras interface (via the `model.fit()` method).

FocalLoss

Implements the Focal Loss function, which is particularly useful for a classification task when there is an extreme imbalance between the classes. To use this loss, you have to pass the string `'focal_loss'` to the `loss_function` parameter in the `config.yaml` file.

```
    def __init__(self, alpha=0.2, gamma=1.5, alpha_threshold=0.05, class_weight_dict
        ↪ =None, **kwargs):
        """
        Parameters
        ----------
        alpha : float, optional
            The scaling factor for the loss. Default is 0.25.

        gamma : float, optional
            The focusing parameter. Default is 2.0.

        alpha_threshold : float, optional
            The threshold for the alpha value. Default is 0.05.

        class_weight_dict : dict, optional
            A dictionary that maps each class to a custom weight. Default is None.

        """

    def focal_loss(self, y_true, y_pred):
        """
        Implements the logic of the Focal Loss function.

        Parameters
        ----------
        y_true : Tensor
            The true labels.

        y_pred : Tensor
            The predicted labels.

        Returns
        -------
        focal_loss : Tensor
            The computed Focal Loss.

        Note
        -----
        This function is not intended to be used directly. Instead, it's used in the
            ↪ `call` method.
        """

```

```
    def call(self, y_true, y_pred):
        """
        Calls the Focal Loss function.

        Parameters
        ----------
        y_true : Tensor
            The true labels.

        y_pred : Tensor
            The predicted labels.

        Returns
        -------
        focal_loss : Tensor
            The computed Focal Loss.
        """

    def get_config(self):
        """
        Gets the configuration of the FocalLoss class.

        Returns
        -------
        config : dict
            A dictionary containing the configuration of the FocalLoss class.
        """

    @classmethod
    def from_config(cls, config):
        """
        Creates a FocalLoss class from its configuration.

        Parameters
        ----------
        config : dict
            A dictionary containing the configuration of the FocalLoss class.

        Returns
        -------
        FocalLoss
            A new FocalLoss class.

        Note
        -----
        This is a class method that returns a new FocalLoss class given its
            ↪ configuration.
        """
```

WeightedMeanSquaredError

Implements the Weighted Mean Squared Error (WMSE) loss function. This class allows to set custom weights for each sample which is particularly useful when the samples are imbalanced. To use this loss, you need to pass the string '`weighted_mse`' to the `loss_function` in the `config.yaml` file.

```
    def __init__(self, class_weights, **kwargs):
        """
        Initializes the WeightedMeanSquaredErrorLoss class.

        Parameters
        ----------
        class_weights : dict
            A dictionary mapping each output to its weight.
        kwargs : dict
            Additional keyword arguments.
        """

    def call(self, y_true_list, y_pred_list):
        """
        Computes the weighted mean squared error loss for each output.

        Parameters
        ----------
        y_true_list : list of Tensors
            The true labels for each output.
        y_pred_list : list of Tensors
            The predicted labels for each output.

        Returns
        -------
        loss_list : list of Tensors
            The computed weighted mean squared error loss for each output.
        """

    def get_config(self):
        """
        Gets the configuration of the WeightedMeanSquaredErrorLoss class.

        Returns
        -------
        config : dict
            A dictionary containing the configuration of the
                ↪ WeightedMeanSquaredErrorLoss class.
        """

    @classmethod
    def from_config(cls, config):
        """
        Creates a WeightedMeanSquaredErrorLoss class from its configuration.

        Parameters
        ----------
        config : dict
```

```
            A dictionary containing the configuration of the
                ↪ WeightedMeanSquaredErrorLoss class.

        Returns
        -------
        WeightedMeanSquaredErrorLoss
            A new WeightedMeanSquaredErrorLoss class instance.

        Note
        -----
        This is a class method that returns a new WeightedMeanSquaredErrorLoss class
            ↪  instance
        given its configuration.
        """
```

WeightedCategoricalCrossEntropy

Implements the Weighted Categorical Cross Entropy (WCCE) loss function. This class allows to set custom weights for each class, which is particularly useful when dealing with imbalanced classes. To use this loss, you need to pass the string `'weighted_cce'` to the `loss_function` parameter in the `config.yaml` file.

```
    def __init__(self, class_weights, **kwargs):
        """
        Parameters
        ----------
        class_weights : dict
            A dictionary that maps each class to a custom weight.
        """
    def weighted_categorical_crossentropy(self, y_true, y_pred):
        """
        Computes the weighted categorical cross entropy loss.

        Parameters
        ----------
        y_true : Tensor
            The true labels.

        y_pred : Tensor
            The predicted labels.

        Returns
        -------
        weighted_loss : Tensor
            The computed weighted categorical cross entropy loss.

        Note
        -----
        This function is not intended to be used directly. Instead, it's used in the
            ↪  `call` method.
        """
```

```
    def call(self, y_true, y_pred):
        """
        Calls the weighted categorical cross entropy loss function.

        Parameters
        ----------
        y_true : Tensor
            The true labels.

        y_pred : Tensor
            The predicted labels.

        Returns
        -------
        weighted_loss : Tensor
            The computed weighted categorical cross entropy loss.
        """

        return self.weighted_categorical_crossentropy(y_true, y_pred)

    def get_config(self):
        """
        Gets the configuration of the WeightedCategoricalCrossentropy class.

        Returns
        -------
        config : dict
            A dictionary containing the configuration of the
                ↪ WeightedCategoricalCrossentropy class.
        """

    @classmethod
    def from_config(cls, config):
        """
        Creates a WeightedCategoricalCrossentropy class from its configuration.

        Parameters
        ----------
        config : dict
            A dictionary containing the configuration of the
                ↪ WeightedCategoricalCrossentropy class.

        Returns
        -------
        WeightedCategoricalCrossentropy
            A new WeightedCategoricalCrossentropy class instance.

        Note
        -----
        This is a class method that returns a new WeightedCategoricalCrossentropy
            ↪ class instance
        given its configuration.
        """
```

utils.py

This script contains miscelaneous methods to load and preprocess the data.

```
  def image_loader(folder, indices=None, crop=False, crop_size=None, img_size=(64,
      ↪ 64), normalize = True, grayscale = False):
    """
    Loads all the images contained in the specified folder.

    Parameters
    ----------
    folder : string
        Path to the folder where the images are stored.
    indices : array-like, optional
        Indices of specific images to load. Defaults to None, which loads all images
            ↪ .
    crop : bool, optional
        Whether to crop images before resizing. Defaults to False.
    crop_size : tuple, optional
        Area to crop from the images (left, upper, right, lower). Defaults to None.
    img_size : tuple, optional
        The size to which the original image should be resized. Defaults to (64, 64)
            ↪ .
    normalize : bool, optional
        Whether to normalize pixel values to be between 0 and 1. Defaults to True.
    grayscale : bool, optional
        Whether to convert images to grayscale. Defaults to False.

    Returns
    -------
    images : numpy.ndarray
        An array with the images arrays. Shape is (n_images, img_size[0], img_size
            ↪ [1], n_channels),
        where n_channels is 1 for grayscale and 3 for RGB.

    Raises
    ------
    ValueError
        If crop is True but no crop_size is specified.
    """

def labels_loader(folder, task = [1, 2], min=0.8, one_hot_enc = True):
    """
    Loads labels from the specified file and performs optional transformations.

    Parameters
    ----------
    folder : string
        Path to the file where the labels are stored.
    task : list, optional
        Selects the relevant labels according to the Task. Default value [1, 2]
            ↪ gives the labels dataset.
    min : float, optional
```

```
45          The minimum value to keep a label. Rows with all label values below this
                ↪ threshold are dropped. Defaults to 0.8.
46      one_hot_enc : bool, optional
47          Whether to perform one-hot encoding on the labels. Defaults to True.
48
49      Returns
50      -------
51      final_df : pandas.DataFrame
52          A DataFrame with the labels arrays.
53
54      Notes
55      -----
56      The function assumes that the labels are stored in a CSV file with a specific
            ↪ structure. Make sure that the
57      task parameter matches the structure of your file.
58      """
59
60  def data_loader(images_folder, labels_path, task, min=0.8, one_hot_labels = False,
        ↪ crop=False, crop_size=None, img_size=(64, 64), normalize=True, grayscale = 0,
        ↪  training_size = 0.8, test_size = None, random_seed=48):
61      """
62      Load and split the data into training and validation (and optionally test) sets.
63
64      Parameters
65      ----------
66      images_folder : str
67          Path to the folder where the images are stored.
68      labels_path : str
69          Path to the file where the labels are stored.
70      task : int
71          Task to work on.
72      min : float, optional
73          The minimum value to keep a label. Rows with all label values below this
                ↪ threshold are dropped. Defaults to 0.8.
74      one_hot_labels : bool, optional
75          Whether to perform one-hot encoding on the labels. Defaults to False.
76      crop : bool, optional
77          Whether to crop images before resizing. Defaults to False.
78      crop_size : tuple, optional
79          Area to crop from the images (left, upper, right, lower). Defaults to None.
80      img_size : tuple, optional
81          The size to which the original image should be resized. Defaults to (64, 64)
                ↪ .
82      normalize : bool, optional
83          Whether to normalize pixel values to be between 0 and 1. Defaults to True.
84      grayscale : int, optional
85          Whether to convert images to grayscale. 1 means grayscale, 0 means RGB.
                ↪ Defaults to 0 (RGB).
86      training_size : float, optional
87          Proportion of the dataset to include in the training split. Default is 0.8.
88      test_size : float, optional
89          If not None, proportion of the training set to include in the test split.
                ↪ Default is None.
```

```
    random_seed : int, optional
        Seed used by the random number generator for reproducibility. Default is 48.

    Returns
    -------
    tuple
        Contains the training data (X_train, y_train), validation data (X_val, y_val
            ↪ ), and if test_size is not None,
        test data (X_test, y_test) too.

    Notes
    -----
    The function assumes that the labels are stored in a CSV file with a specific
        ↪ structure.
    Make sure that the task parameter matches the structure of your file.
    """

def image_plotter(images, names, ncols, plot_size=(12,4), spacing=(0.025, 0.025),
    ↪ title=None, folder_path=None):
    """
    Plot and optionally save an array of images in a grid layout.

    Parameters
    ----------
    images : array
        The array that contains the images to be plotted.
    names : list
        List of image names/titles to be displayed above each image.
    ncols : int
        The number of columns in the plot grid.
    plot_size : tuple, optional
        The overall size of the entire plot. Defaults to (32,20).
    spacing : tuple, optional
        The spacing between images in the plot, defined as (vertical space,
            ↪ horizontal space). Defaults to (0.025, 0.025).
    folder_path : str, optional
        If provided, the plot will be saved as a PDF file in the specified folder.
            ↪ The folder is created if it does not exist. Default is None.

    Returns
    -------
    None
        This function does not return a value. It shows the plot in the output.

    Notes
    -----
    The images are plotted in a grid layout with a specified number of columns (
        ↪ ncols). The number of rows is determined
    based on the total number of images and ncols. Each image is displayed with its
        ↪ respective title from 'names'.
    """
```

```
def load_file(file):
    """
    Loads a YAML configuration file.

    Parameters
    ----------
    file : str
        The path to the YAML configuration file to load.

    Returns
    -------
    dict
        A dictionary containing the configuration options loaded from the file.

    Raises
    ------
    FileNotFoundError
        If the specified file does not exist.

    Notes
    -----
    The function raises an exception if the file does not exist. The function
        ↪ assumes the file content is in the YAML format.
    """

def set_paths(paths):
    """
    Sets the paths for loading data and saving outputs (models and plots).

    Parameters
    ----------
    paths : dict
        A dictionary containing the paths. It should include the following keys:
        'images_path' - Path to the directory where the image data is stored.
        'labels_path' - Path to the file where the labels are stored.
        'models_path' - Path to the directory where models should be saved.
        'plots_path' - Path to the directory where plots should be saved.

    Returns
    -------
    tuple
        A tuple containing four strings: the path to the images, the path to the
            ↪ labels, the path to the models directory,
        and the path to the plots directory.

    Notes
    -----
    The function creates the directories for saving models and plots if they do not
        ↪ already exist.
    """
```

```
186 def save_config(config_file):
187     """
188     Prints the configuration parameters and saves them into a log file.
189 
190     Parameters
191     ----------
192     config_file : str
193         Path to the YAML configuration file.
194 
195     Notes
196     -----
197     This function captures the current state of the standard output (stdout),
        ↪ redirects it to a string IO object,
198     and restores it after the operation. The captured output (configuration
        ↪ parameters) is written into a text file.
199 
200     The 'sort_keys' parameter in the yaml.dump() method is set to False. This means
        ↪ the order of keys in the original
201     YAML file is preserved in the output, rather than being sorted in default Python
        ↪ 's lexicographical order.
202     The function creates a directory named 'logs' if it does not already exist to
        ↪ store the log files.
203 
204     """
205 
206 def save_hist(history, models_path, name):
207     """
208     Saves the history of a model's training into a pickle file.
209 
210     Parameters
211     ----------
212     history : History object
213         The history object generated by the training process of a model.
214     models_path : str
215         The path to the directory where the history file will be saved.
216     name : str
217         The name to be given to the saved history file.
218 
219     Notes
220     -----
221     The history object typically includes useful data like loss and accuracy metrics
        ↪  recorded at each epoch
222     during the training process. It is saved using the pickle module, which allows
        ↪ for serialization and de-serialization
223     of Python object structures.
224     """
225 
226 def one_hot_encoder(df):
227     """
228     Converts a DataFrame to one-hot encoding.
229 
230     This function takes a DataFrame and converts it to one-hot encoded form. This
        ↪ means that each row is a vector
```

```
    where only the element corresponding to the category of that sample is set to 1 and all other elements are 0.

    Parameters
    ----------
    df : DataFrame
        The DataFrame to be converted to one-hot encoding.
        It is expected that the DataFrame is numeric and that the column with the highest value in each row
        corresponds to the category of that sample.

    Returns
    -------
    one_hot_df : DataFrame
        A DataFrame that is the one-hot encoded version of the input df.
        It has the same shape as the input df, with values replaced by 0s and 1s.
    """

def hierarchy_one_hot(df, min = 0.5):
    """
    Converts a DataFrame to one-hot encoding based on the maximum value in each hierarchical group.

    Parameters
    ----------
    df : pandas.DataFrame
        The DataFrame to be converted to one-hot encoding.
        It should contain columns for each class and sub-class.
    min : float, optional
        The threshold value used for the one-hot encoding of Class1. Defaults to 0.5.

    Returns
    -------
    df_one_hot : pandas.DataFrame
        A DataFrame that is the one-hot encoded version of the input df based on the  max value in each group.
    """

def one_hot_encoder_array(array):
    """
    Converts an array to one-hot encoding.

    This function takes a 2D numpy array and converts it to one-hot encoded form. This means that each row is a vector
    where only the element corresponding to the category of that sample is set to 1 and all other elements are 0.

    Parameters
    ----------
    array : ndarray
        The 2D numpy array to be converted to one-hot encoding.
```

```
        It is expected that the array is numeric and that the column with the highest value in each row
        corresponds to the category of that sample.

    Returns
    -------
    new_array : ndarray
        A 2D numpy array that is the one-hot encoded version of the input array.
        It has the same shape as the input array, with values replaced by 0s and 1s.
    """

def data_inspector(labels):
    """
    Inspects the assigned classes based on the predicted label probabilities.

    Parameters
    ----------
    labels : pandas.DataFrame
        The predicted label probabilities for each sample. The DataFrame is expected to have the following columns:
        'Class1.1', 'Class1.2', 'Class1.3', 'Class2.1', 'Class2.2', 'Class7.1', 'Class7.2', 'Class7.3'.

    Returns
    -------
    None

    Prints
    ------
    Counts of Class1 Subclasses:
    Prints the counts of samples assigned to each Class1 subclass and their corresponding labels.

    Counts of Class2 Subclasses among those assigned to Class1.2:
    Prints the counts of samples assigned to each Class2 subclass among those assigned to Class1.2 and their corresponding labels.

    Counts of Class7 Subclasses among those assigned to Class1.1:
    Prints the counts of samples assigned to each Class7 subclass among those assigned to Class1.1 and their corresponding labels.
    """
```

Conclusion

In summary, this document provides detailed documentation for the code developed in the project. It serves as a reference for understanding the classes, methods, and their usage.