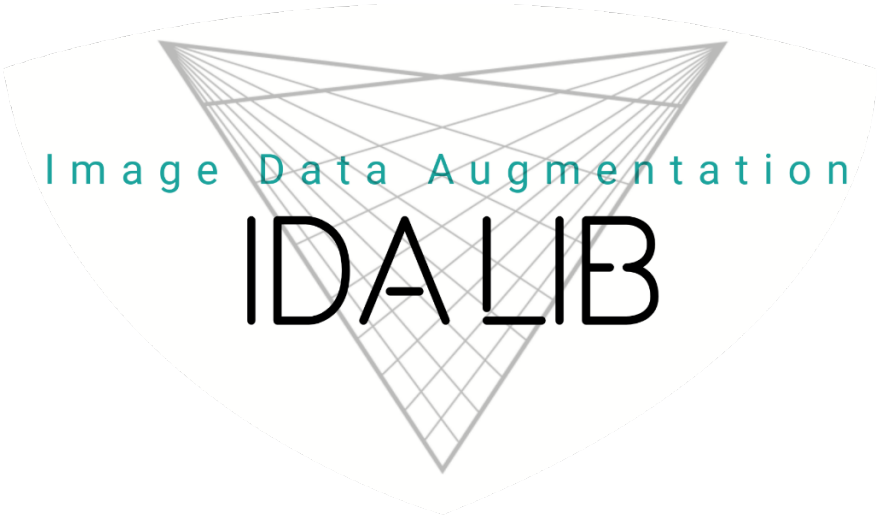
IdaLib

Release 0.0.1

Raquel Vilas

Jun 24, 2020

CONTENTS



IdaLib (Image Data Augmentation Library) is a library dedicated to the task of Image Data Augmentation in a fast, simple, efficient and flexible way This library allows you to convert your input data into a larger and more diverse one to perform a better train of your Neural Network

CHAPTER

ONE

FEATURES

- Multiple fast augmentations based on Kornia an OpenCV libraries
- Flexible
- Complete tool, includes support for tasks associated with Pytorch-based neural networks
- Supports multiple types of combined data (images, heat maps, segmentation maps, masks and keypoints)

CHAPTER

TWO

INSTALATION

You can use pip to install Ida-Lib

```
pip install ida-lib
```

If you want to get the latest version of the code before it is released on PyPI you can install the library from GitHub:

```
pip install -U git+https://github.com/raquelvilas18/ida_lib
```

CHAPTER

THREE

CONTENTS

3.1 Overview

3.1.1 Functionalities

The functionalities on which IdaLib focuses are:

1. Offering a **wide variety of image transformations**.
2. Allow the **joint transformation** of:
    - image
    - heatmap
    - mask
    - segmaps
    - keypoints
    - any combination of them
3. Offer an **interactive visualization tool** that facilitates the debugging of programs.
4. Offer an **efficient operation composition pipeline** (parameterizable at probability and attribute level).
5. Offer a **Dataloader** object (as a generator) that integrates the pipeline and allows the supply of any neural network.
6. Offer a tool to perform **Image Data Augmentation directly to disk** in order to use the increased dataset directly in future situations

3.1.2 First steps

The central object of ida lib is its pipeline. To use it, just decide which PipelineOperations we want it to include. All other parameters are optional.

```
example_pipipeline = Pipeline(pipeline_operations=(
                     ScalePipeline(probability=0.5, scale_factor=0.5),
                     ShearPipeline(probability=0.3, shear=(0.2, 0.2)),
                     TranslatePipeline(probability=0.4, translation=(10,50)),
                     HflipPipeline(probability=0.6, exchange_points=[(0, 5), (1, 6)]),
                     RandomRotatePipeline(probability=0.4, degrees_range=(-20, 20))
                     )
                     )
```

The pipelineOperations can be divided into 2 groups:

- the **classic operations**, where you indicate exactly the parameters of the operation (for example `` `RotatePipeline(degrees=20)` ``). In [transformations](#operations) you can see what each one of them does
- and the **Random pipelineOPerations**, where what you define is a range of possible parameters, and each time the operation is applied it will take a different value within this range (`RandomRotatePipeline(degrees_range=(-20,30))`)

Once you have defined your pipeline, you can pass through it your batch data to be transformed.

Warning: Remember that the entry for the pipeline must be composed of Python dicts. To be treated correctly, each dict's element must be associated with its type (images with 'image1', 'image2'...; masks with 'mask1', 'mask67'...):

```
data1 = {'image': img1, 'keypoints': random_coordinates, 'mask': mask_example1}
data2 = {'image': img2, 'keypoints2': random_coordinates2, 'mask': mask_example2}
data3 = {'image': img3, 'keypoints3': random_coordinates3, 'mask': mask_example3}

batch = [data1, data2, data3]
```

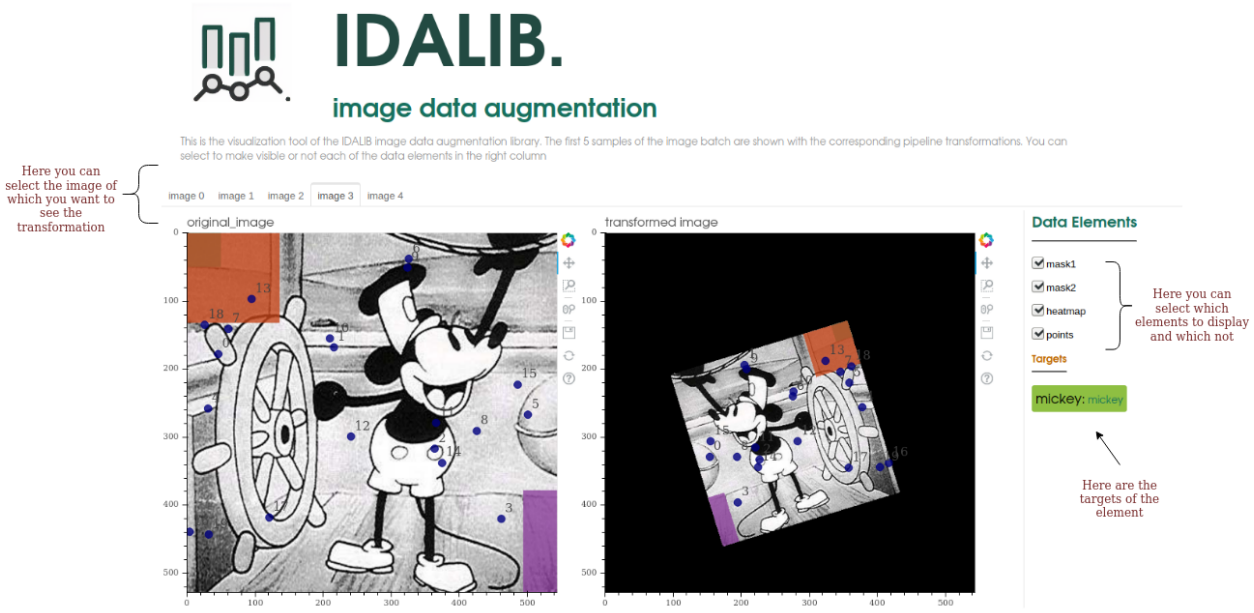
Finally we run the pipeline as many times as necessary:

```
transformed_batch = example_pipipeline(batch)
```

3.1.3 Visualization tool

Ida Lib includes a tool to visualize the transformations to facilitate code debugging. It is an interactive tool developed with the bokeh framework and allows the selection of the data to be displayed in the image.

- The color code is used to differentiate each element and identify it in all the images.
- The dots are numbered in order to see their order
- Allows to compare different transformations obtained by the pipeline
- It also includes the targets in the visualization in order to have a complete view of the elements



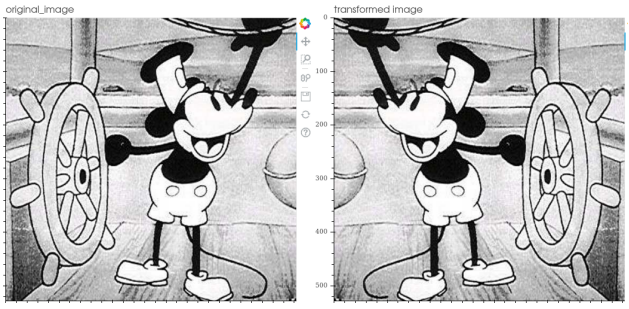
Note: To test the visualization tool you can try this example: https://github.com/raquelvilas18/ida_lib/blob/master/examples/pipeline_usage.py

Warning: For the visualization tool a bokeh server is deployed; therefore it is only possible to have one open execution. It is important to close previous runs in order to open new windows

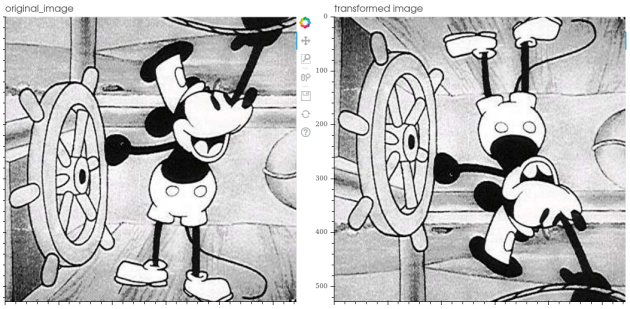
3.2 Transformations

The transformations included in the library are:

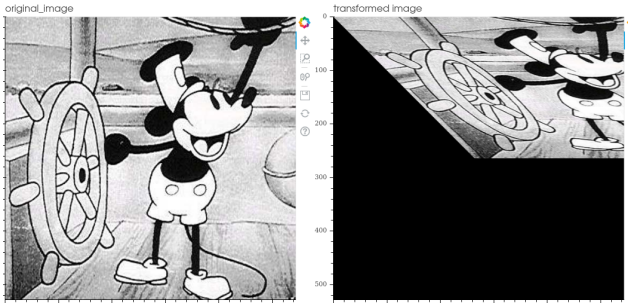
1. **hflip**: horizontal flipping the image



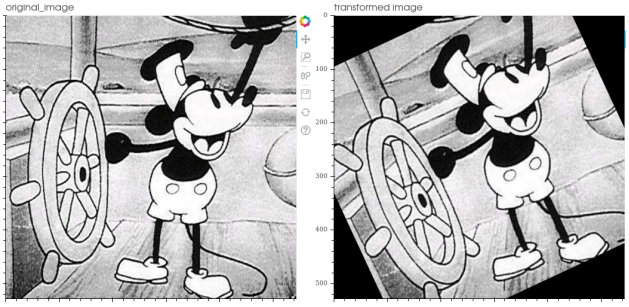
2. **vflip**: vertical flipping the image



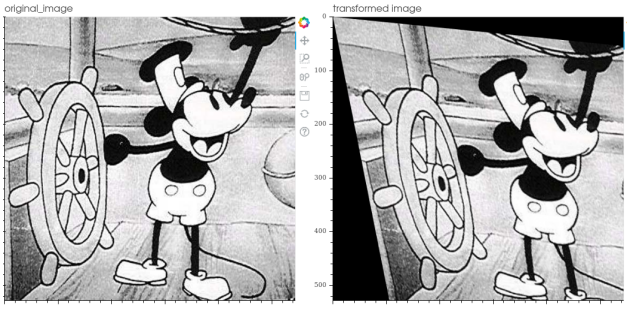
3. **Affine**:carry out the transformation expressed in the operation matrix



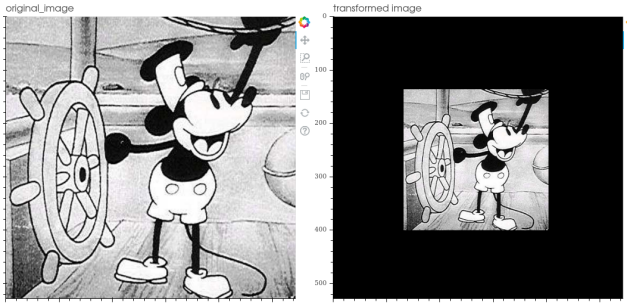
4. **Rotate**:rotate the image by the indicated degrees counterclockwise



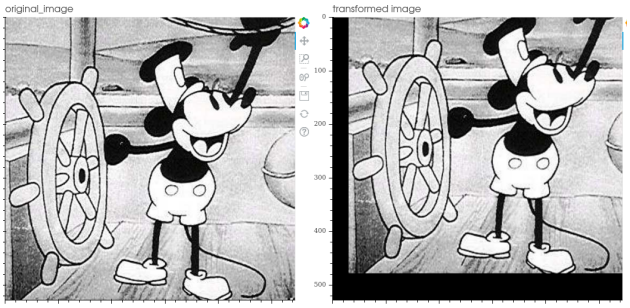
5. **Shear**: linear map that displaces each point in fixed direction, by an amount proportional to its signed distance from the line that is parallel to that direction and goes through the origin



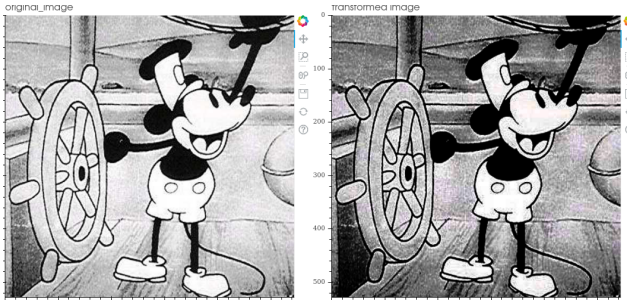
6. **Scale**: scale the image by making it smaller or larger (crop equivalent)



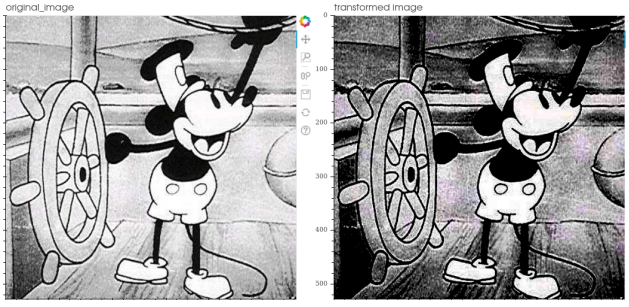
7. **Translate**: moves the image pixels to the positions indicated on each axis



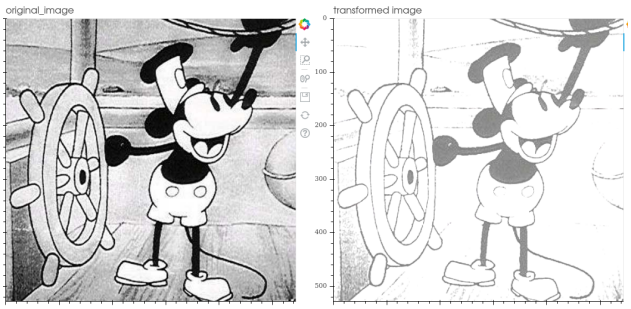
8. **Change gamma**: adjust image's gamma (luminance correction) .



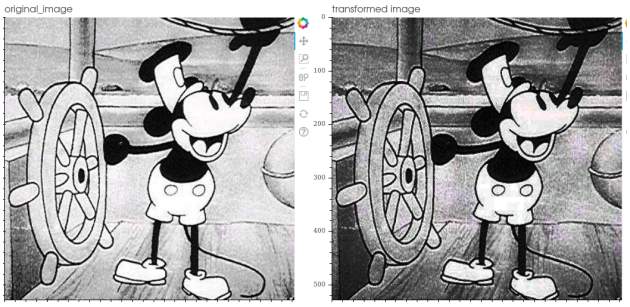
9. **Change contrast:**: change the image contrast.



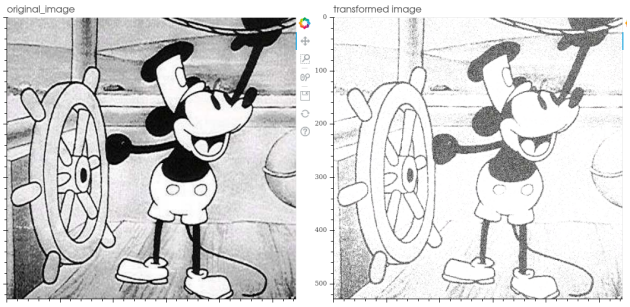
10. **Change brightness**: change the image brightness



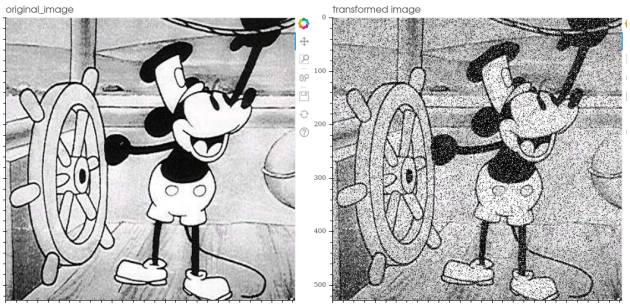
11. **Equalize histogram**: equalize the image histogram



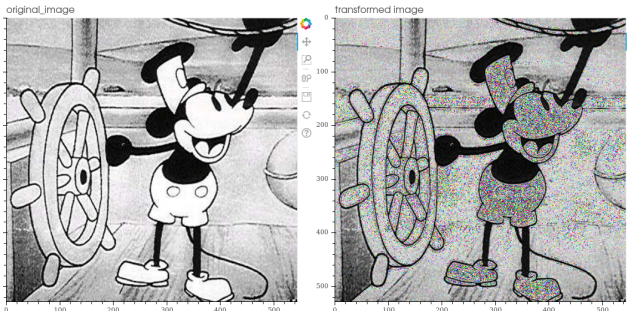
12. **Inject gaussian noise**: gaussian noise is a statistical noise having a probability density function (PDF) equal to that of the normal distribution



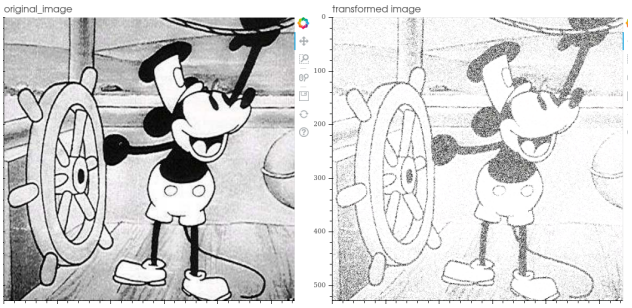
13. **Inject salt and pepper noise**: salt-and-pepper noise is a statistical noise compose of white (salt) and black (pepper) pixels



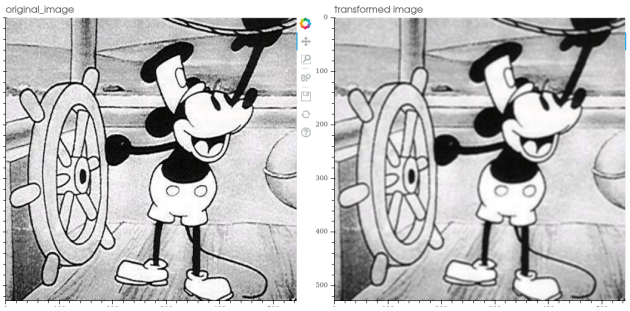
14. **Inject spekle noise**: Speckle is a granular interference that inherently exists in and degrades the quality of the active radar, synthetic aperture radar (SAR), medical ultrasound and optical coherence tomography images. It is applied by adding the image multiplied by the noise matrix -> img + img * uniform_noise



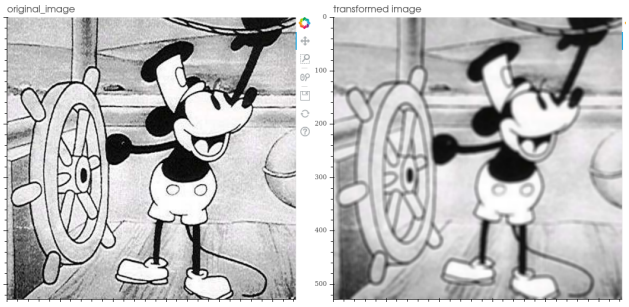
15. **Inject poisson noise**: It is applied by adding Poisson-distributed noise



16. **Blur**: blur image.



17. **Gaussian blur**: blurring an image by a Gaussian function.



3.3 API

3.3.1 API packages

ida_lib.core package

Submodules

ida_lib.core.pipeline module

class ida_lib.core.pipeline.**Pipeline**(*pipeline_operations: list*, *resize: tuple = None*, *interpolation: str = 'bilinear'*, *padding_mode: str = 'zeros'*, *output_format: str = 'dict'*, *output_type: Optional[torch.dtype] = None*)

Bases: object

The pipeline object represents the pipeline with data transformation operations (pictures, points). When executed, on a batch of images,it applies the necessary transformations (being different on each image based on the probabilities of each operation included).

Considerations: 1) The images must be of the same size, or the RESIZE operation must be included so that the transformations can be applied correctly 2) To run the pipeline, it accepts any type of input metadata named in the input dict. In particular it gives special treatment

to data named as:

- Mask: it is affected by geometric transformations and its output is discrete to values of 0-1
- Segmap: generalization of mask. Every value is discrete
- Image: affected by geometric and color transformations
- Keypoints: geometric transformations are applied to them as coordinates.
- Others: any other metadata will not be transformed (example: 'tag', 'target'...)

Example:

pip = pipeline(resize = (25, 25), pipeline_operations=(translate_pipeline(probability=0.5, translation=(3, 0.05)), vflip_pipeline(probability=0.5), hflip_pipeline(probability=0.5), contrast_pipeline(probability=0.5, contrast_factor=1), random_brightness_pipeline(probability=0.2, brightness_range=(1.5, 1.6)), random_scale_pipeline(probability=1, scale_range=(0.5, 1.5), center_deviation=20), random_rotate_pipeline(probability=0.2, degrees_range=(-50, 50), center_deviation=20))

))

get_data_types() → tuple

Returns the tuple of data types identified on the input data

ida_lib.core.pipeline_functional module

ida_lib.core.pipeline_functional.**get_compose_matrix**(*operations: list*, *data_info: Optional[dict] = None*) → None._VariableFunctionsClass.tensor

Returns the transformation matrix composed by the multiplication in order of the input operations (according to their probability) If data_info is not None, go through the operations by entering the necessary information about the images (image center, shape..)

Parameters

- **operations** – list of pipeline operations
- **data_info** – dict with data info to configure operations parameters

Returns torch tensor of the transform matrix

ida_lib.core.pipeline_functional.**get_compose_function**(*operations: list*) → numpy.ndarray

returns the LUT table with the correspondence of each possible value according to the color operations to be implemented (according to their probability)

Parameters **operations** – list of pipeline operations

Returns compose function

ida_lib.core.pipeline_functional.**preprocess_data**(*data: Union[list, dict]*, *batch_info: Union[list, dict] = None*, *interpolation: str = None*, *resize: Optional[tuple] = None*) → list

Combines the 2d information in a tensor and the points in a homogeneous coordinate matrix that allows applying the geometric operations in a single joint operation on the data and another on the points.

- Loads the data as tensor in GPU to prepare them as input to a neural network
- Analyze the data info required for the transformations (shape, bpp...)
- Resize the 2d data and keypoints to the new shape

Parameters

- **resize** – if it is wanted to resize the data, indicate the new size
- **data** – list of elements to be transformed through the pipe
- **batch_info** – dict with the required data info
- **interpolation** – desired interpolation mode to be applied

Returns preprocessed and resized data, and dict with batch info

ida_lib.core.pipeline_functional.**split_operations_by_type**(*operations: list*) → tuple

Split input operations into sub-lists of each transformation type the normalization operation is placed last to apply correctly the other operations

Parameters **operations** – list of pipeline operations

Returns tuple of lists of the operations separated into color, geometry and independent

ida_lib.core.pipeline_functional.**postprocess_data**(*batch: list*, *batch_info: dict*, *data_original: Optional[list] = None*, *visualize: bool = False*, *original_type: torch.dtype = torch.uint8*, *output_format: str = 'dict'*) → list

Restores the data to the original form; separating the matrix into the different 2d input data and point coordinates.

Parameters

- **batch** – list of elements to be transformed through the pipe
- **batch_info** – dict with necessary information about the batch data
- **data_original** – original batch before transforms
- **visualize** – whether to run the visualization tool or not
- **original_type** – torch original type of the input data to do the conversion of the output data to this type
- **output_format** – whether to format the output element as a dict or as a tuple

Returns processed data

ida_lib.core.pipeline_functional.**switch_point_positions**(*point_matrix*, *input_list*)

ida_lib.core.pipeline_geometric_ops module

class ida_lib.core.pipeline_geometric_ops.**HflipPipeline**(*probability: float = 1*, *exchange_points: tuple = None*)

Bases: *ida_lib.core.pipeline_operations.PipelineOperation*

Horizontally flip the input image-mask-keypoints and 2d data

config_parameters(*data_info: dict*)

get_op_matrix() → None._VariableFunctionsClass.tensor

need_data_info() → bool

switch_points()

class ida_lib.core.pipeline_geometric_ops.**VflipPipeline**(*probability: float*, *exchange_points: tuple = None*)

Bases: *ida_lib.core.pipeline_operations.PipelineOperation*

Vertically flip the input image-mask-keypoints and 2d data

config_parameters(*data_info: dict*)

get_op_matrix() → None._VariableFunctionsClass.tensor

need_data_info() → bool

switch_points()

class ida_lib.core.pipeline_geometric_ops.**RotatePipeline**(*degrees: int*, *center: None._VariableFunctionsClass.tensor = None*, *probability: float = 1*)

Bases: *ida_lib.core.pipeline_operations.PipelineOperation*

Rotate the input image-mask-keypoints and 2d data by the input degrees

config_parameters(*data_info: dict*)

get_op_matrix() → None._VariableFunctionsClass.tensor

need_data_info() → bool

static switch_points()

class ida_lib.core.pipeline_geometric_ops.**ShearPipeline**(*shear: tuple*, *probability: float = 1*)

Bases: *ida_lib.core.pipeline_operations.PipelineOperation*

Shear the input image-mask-keypoints and 2d data by the input shear factor

get_op_matrix() → None._VariableFunctionsClass.tensor

static need_data_info() → bool

static switch_points()

class ida_lib.core.pipeline_geometric_ops.**ScalePipeline**(*scale_factor: Union[float, tuple]*, *probability: float = 1*, *center: None._VariableFunctionsClass.tensor = None*)

Bases: *ida_lib.core.pipeline_operations.PipelineOperation*

Scale the input image-mask-keypoints and 2d data by the input scaling value

config_parameters(*data_info: dict*)

get_op_matrix()

need_data_info()

static switch_points()

class ida_lib.core.pipeline_geometric_ops.**TranslatePipeline**(*translation: tuple*, *probability: float = 1*)

Bases: *ida_lib.core.pipeline_operations.PipelineOperation*

Translate the input image-mask-keypoints and 2d data by the input translation

get_op_matrix() → None._VariableFunctionsClass.tensor

static need_data_info() → bool

static switch_points()

class ida_lib.core.pipeline_geometric_ops.**RandomScalePipeline**(*probability: float*, *scale_range: tuple*, *keep_aspect: bool = True*, *center_deviation: int = None*, *center: None._VariableFunctionsClass.tensor = None*)

Bases: *ida_lib.core.pipeline_operations.PipelineOperation*

Scale the input image-mask-keypoints and 2d data by a random scaling value calculated within the input range

config_parameters(*data_info: dict*)

get_op_matrix() → None._VariableFunctionsClass.tensor

need_data_info() → bool

static switch_points()

class ida_lib.core.pipeline_geometric_ops.**RandomRotatePipeline**(*degrees_range: tuple*, *probability: float = 1*, *center_deviation: int = None*, *center: None._VariableFunctionsClass.tensor = None*)

Bases: *ida_lib.core.pipeline_operations.PipelineOperation*

Rotate the input image-mask-keypoints and 2d data by a random scaling value calculated within the input range

config_parameters(*data_info: dict*)

get_op_matrix() → None._VariableFunctionsClass.tensor

need_data_info() → bool

static switch_points()

class ida_lib.core.pipeline_geometric_ops.**RandomShearPipeline**(*probability: float*, *shear_range: tuple*)

Bases: *ida_lib.core.pipeline_operations.PipelineOperation*

Shear the input image-mask-keypoints and 2d data by a random shear value calculated within the input range

get_op_matrix() → None._VariableFunctionsClass.tensor

static need_data_info() → bool

static switch_points()

class ida_lib.core.pipeline_geometric_ops.**RandomTranslatePipeline**(*probability: float*, *translation_range: tuple*, *same_translation_on_axis: bool = False*)

Bases: *ida_lib.core.pipeline_operations.PipelineOperation*

Translate the input image-mask-keypoints and 2d data by a random translation value calculated within the input range

get_op_matrix()

static need_data_info() → bool

static switch_points()

ida_lib.core.pipeline_local_ops module

class ida_lib.core.pipeline_local_ops.**BlurPipeline**(*probability: float = 1*, *blur_size: tuple = 5, 5*)

Bases: *ida_lib.core.pipeline_operations.PipelineOperation*

Blur input image (non-weighted blur)

apply_to_image_if_probability(*img: numpy.ndarray*) → numpy.ndarray

get_op_matrix()

class ida_lib.core.pipeline_local_ops.**GaussianBlurPipeline**(*probability: float = 1*, *blur_size: tuple = 5, 5*)

Bases: *ida_lib.core.pipeline_operations.PipelineOperation*

Blur input image by a Gaussian function

apply_to_image_if_probability(*img: numpy.ndarray*) → numpy.ndarray

get_op_matrix()

class ida_lib.core.pipeline_local_ops.**GaussianNoisePipeline**(*probability: float = 1*, *var: float = 0.5*)

Bases: *ida_lib.core.pipeline_operations.PipelineOperation*

Add gaussian noise to the input image (gaussian noise is a statistical noise having a probability density function (PDF) equal to that of the normal distribution)

apply_to_image_if_probability(*img: numpy.ndarray*) → numpy.ndarray

get_op_matrix()

class ida_lib.core.pipeline_local_ops.**PoissonNoisePipeline**(*probability: float = 1*)

Bases: *ida_lib.core.pipeline_operations.PipelineOperation*

Add poison noise to the input image (Speckle is a granular interference that inherently exists in and degrades the quality of the active radar, synthetic aperture radar (SAR), medical ultrasound and optical coherence tomography images. It is applied by adding Poisson-distributed noise)

apply_to_image_if_probability(*img: numpy.ndarray*) → numpy.ndarray

get_op_matrix()

class ida_lib.core.pipeline_local_ops.**SaltAndPepperNoisePipeline**(*probability=1*, *amount: Optional[float] = 0.01*, *s_vs_p: Optional[float] = 0.5*)

Bases: *ida_lib.core.pipeline_operations.PipelineOperation*

Add salt and pepper noise to the input image (salt-and-pepper noise is a statistical noise compose of white (salt) and black (pepper) pixels)

apply_to_image_if_probability(*img: numpy.ndarray*) → numpy.ndarray

get_op_matrix()

class ida_lib.core.pipeline_local_ops.**SpekleNoisePipeline**(*probability: float = 1*, *mean: Optional[float] = 0*, *var: Optional[float] = 0.01*)

Bases: *ida_lib.core.pipeline_operations.PipelineOperation*

Add spekle noise to the input image (Speckle is a granular interference that inherently exists in and degrades the quality of the active radar, synthetic aperture radar (SAR), medical ultrasound and optical coherence tomography images. It is applied by adding the image multiplied by the noise matrix -> img + img * uniform_noise)

apply_to_image_if_probability(*img: numpy.ndarray*) → numpy.ndarray

get_op_matrix()

ida_lib.core.pipeline_operations module

class ida_lib.core.pipeline_operations.**PipelineOperation**(*op_type: str*, *probability: float = 1*)

Bases: abc.ABC

Abstract class of pipeline operations

apply_according_to_probability() → bool

abstract get_op_matrix()

get_op_type()

ida_lib.core.pipeline_pixel_ops module

class ida_lib.core.pipeline_pixel_ops.**ContrastPipeline**(*contrast_factor: float*, *probability: float = 1*)

Bases: *ida_lib.core.pipeline_operations.PipelineOperation*

Change the contrast of the input image.

get_op_matrix()

transform_function(*x: int*) → float

class ida_lib.core.pipeline_pixel_ops.**RandomContrastPipeline**(*contrast_range: tuple*, *probability: float = 1*)

Bases: *ida_lib.core.pipeline_operations.PipelineOperation*

Change the contrast of the input image with a random contrast factor calculated within the input range

get_op_matrix()

transform_function(*x*)

class ida_lib.core.pipeline_pixel_ops.**BrightnessPipeline**(*brightness_factor: float*, *probability: float = 1*)

Bases: *ida_lib.core.pipeline_operations.PipelineOperation*

Change brightness of the input image

get_op_matrix()

get_op_type()

transform_function(*x: int*) → float

class ida_lib.core.pipeline_pixel_ops.**RandomBrightnessPipeline**(*probability: float*, *brightness_range: tuple*)

Bases: *ida_lib.core.pipeline_operations.PipelineOperation*

Change brightness of the input image to random amount calculated within the input range

get_op_matrix()

get_op_type()

transform_function(*x: int*) → float

class ida_lib.core.pipeline_pixel_ops.**GammaPipeline**(*gamma_factor: float*, *probability: float = 1*)

Bases: *ida_lib.core.pipeline_operations.PipelineOperation*

Change the luminance of the input image

get_op_matrix()

transform_function(*x: int*) → float

class ida_lib.core.pipeline_pixel_ops.**RandomGammaPipeline**(*gamma_range: tuple*, *probability: float = 1*)

Bases: *ida_lib.core.pipeline_operations.PipelineOperation*

Change the luminance of the input image by a random gamma factor calculated within the input range

get_op_matrix()

transform_function(*x: int*) → float

class ida_lib.core.pipeline_pixel_ops.**NormalizePipeline**(*probability: float = 1*, *old_range: tuple = 0, 255*, *new_range: tuple = 0, 1*)

Bases: *ida_lib.core.pipeline_operations.PipelineOperation*

Change the pixels value to a normalize range

get_op_matrix()

static transform_function(*x: int*) → float

class ida_lib.core.pipeline_pixel_ops.**DenormalizePipeline**(*probability: float = 1*, *old_range: tuple = 0, 1*, *new_range: tuple = 0, 255*)

Bases: *ida_lib.core.pipeline_operations.PipelineOperation*

Denormalize pixel value

get_op_matrix()

transform_function(*x: int*) → float

Module contents

ida_lib.image_augmentation package

Submodules

ida_lib.image_augmentation.augment_to_disk module

class ida_lib.image_augmentation.augment_to_disk.**AugmentToDisk**(*dataset: torch.utils.data.dataset.Dataset*, *samples_per_item: Optional[int] = 2*, *total_output_samples: Optional[int] = None*, *operations: Optional[list] = None*, *interpolation: str = 'bilinear'*, *padding_mode: str = 'zeros'*, *resize: Optional[tuple] = None*, *output_extension: str = '.jpg'*, *output_csv_path: str = 'annotations.csv'*, *output_path: str = './augmented'*)

Bases: object

The AugmentToDisk object allows to perform Data Image Augmentation directly to disk. That is, to save the images generated to disk to be used in future processes.

final_save()

This method can be overwritten to make a customized saving of the items according to the interests of the user

Method that runs only once, once all the images have been processed. Useful for writing csv with image annotations. By default the annotations of all images are saved in the same file. The csv file will have one row for each generated element, identified by its id. Each column will correspond with the labels associated to each generated element. In the case of coordinate lists, their coordinates are arranged in columns separating the x and y coordinates in each element (point0_x, point0_y, point1_x, . . . , point_y)

save_item(*item: dict*, *index: int*, *output_path: str*, *types_2d: list*, *other_types: list*, *element: int*)

This method can be overwritten to make a customized saving of the items according to the interests of the user

Method that implements the way to save to disk each of the generated elements. By default it saves all the generated images in the specified path. The samples are organized by name following the form:

- images: <id_image>_<sample number> <extension>
- other two-dimensional types: <id_image>-<data_type>_<sample number> <extension>

Annotations on the data, such as labels, or point coordinates are stored in dictionaries that will be written when all the images have been processed.

Parameters

- **item** – input element to be saved to disk
- **element** – input element number to identify it
- **index** – sample number to which the input item corresponds
- **output_path** – path to the directory in which to save the generated data
- **types_2d** – list of types of two dimensional data of the input item
- **other_types** – list of types that are not two-dimensional elements

ida_lib.image_augmentation.data_loader module

class ida_lib.image_augmentation.data_loader.**AugmentDataLoader**(*batch_size*, *dataset: torch.utils.data.dataset.Dataset*, *shuffle=True*, *pipeline_operations=None*, *resize=None*, *interpolation: str = 'bilinear'*, *padding_mode: str = 'zeros'*, *output_format: str = 'dict'*, *output_type: Optional[torch.dtype] = None*)

Bases: torch.utils.data.dataloader.DataLoader

The DataAugmentDataLoader class implements a Pytorch DataLoader but groups it into one class:

- The Dataset object that takes care of reading the data
- The iterative DataLoader object that will serve as an input system for a neural network.
- A pipeline that applies data image Augmentation operations over the input data.

To make use of this class, it is necessary to provide a dataset to make a personalized reading of your data.

class InnerDataset(*pipeline*, *dataset*)

Bases: torch.utils.data.dataset.Dataset

inner dataset is an internal class that uses the DataAugmentDataLoader to add the pipeline to the input dataset

Module contents

ida_lib.operations package

Submodules

ida_lib.operations.geometry_ops_functional module

ida_lib.operations.geometry_ops_functional.**affine_compose_data**(*data: dict*, *matrix: None._VariableFunctionsClass.tensor*) → dict

Parameters

- **data** – dict of elements to be transformed
- **matrix** – matrix of transformation

Returns transformed data

ida_lib.operations.geometry_ops_functional.**affine_coordinates_matrix**(*matrix_coordinates: None._VariableFunctionsClass.t ma-trix_transformation: None._VariableFunctionsClass.t* → None._VariableFunctionsClass.t

ida_lib.operations.geometry_ops_functional.**affine_image**(*img: None._VariableFunctionsClass.tensor*, *matrix: None._VariableFunctionsClass.tensor*, *interpolation: str = 'bilinear'*, *padding_mode: str = 'zeros'*) → None._VariableFunctionsClass.tensor

ida_lib.operations.geometry_ops_functional.**config_scale_matrix**(*scale_factor*, *center*, *matrix*)

ida_lib.operations.geometry_ops_functional.**get_rotation_matrix**(*center: None._VariableFunctionsClass.tensor*, *degrees: None._VariableFunctionsClass.tensor*)

ida_lib.operations.geometry_ops_functional.**get_scale_matrix**(*center: None._VariableFunctionsClass.tensor*, *scale_factor: Union[float, None._VariableFunctionsClass.tensor]*)

ida_lib.operations.geometry_ops_functional.**get_shear_matrix**(*shear_factor: tuple*) → None._VariableFunctionsClass.tensor

ida_lib.operations.geometry_ops_functional.**get_squared_scale_matrix**(*center: None._VariableFunctionsClass.te scale_factor: Union[float, None._VariableFunctionsClass.te*

ida_lib.operations.geometry_ops_functional.**get_squared_shear_matrix**(*shear_factor: tuple*) → None._VariableFunctionsClass.te

ida_lib.operations.geometry_ops_functional.**get_translation_matrix**(*translation: None._VariableFunctionsClass.tenso* → None._VariableFunctionsClass.tenso

ida_lib.operations.geometry_ops_functional.**hflip_compose_data**(*data: dict*) → dict

Parameters **data** – dict of elements to be transformed

Returns transformed data

ida_lib.operations.geometry_ops_functional.**hflip_coordinates_matrix**(*matrix: None._VariableFunctionsClass.te width: int*) → None._VariableFunctionsClass.te

ida_lib.operations.geometry_ops_functional.**hflip_image**(*img: None._VariableFunctionsClass.tensor*) → None._VariableFunctionsClass.tensor

ida_lib.operations.geometry_ops_functional.**own_affine**(*tensor: torch.Tensor, matrix: torch.Tensor, interpolation: str = 'bilinear', padding_mode: str = 'border'*) → torch.Tensor

Apply an affine transformation to the image.

Parameters

- **tensor** – The image tensor to be warped.
- **matrix** – The 2x3 affine transformation matrix.
- **interpolation** – interpolation mode to calculate output values 'bilinear' | 'nearest'. Default: 'bilinear'.
- **padding_mode** – padding mode for outside grid values 'zeros' | 'border' | 'reflection'. Default: 'zeros'.

Returns The warped image.

ida_lib.operations.geometry_ops_functional.**prepare_data**(*func*)

Decorator that prepares the input data to apply the geometric transformation. For this purpose, it concatenates all the two-dimensional elements of the input data in the same tensor on which a single transformation is applied. If the input data contains point coordinates, they are grouped in a matrix as homogeneous coordinates, over which a single matrix multiplication is performed.

Parameters **func** – geometric function to be applied to the data

Returns processed data

ida_lib.operations.geometry_ops_functional.**rotate_compose_data**(*data: dict*, *degrees: None._VariableFunctionsClass.tensor*, *center: None._VariableFunctionsClass.tensor*)

Parameters

- **data** – dict of elements to be transformed
- **degrees** – counterclockwise degrees of rotation
- **center** – center of rotation. Default, center of the image

Returns transformed data

ida_lib.operations.geometry_ops_functional.**rotate_coordinates_matrix**(*matrix_coordinates: None._VariableFunctionsClass.t matrix: None._VariableFunctionsClass.t* → None._VariableFunctionsClass.t

ida_lib.operations.geometry_ops_functional.**rotate_image**(*img: None._VariableFunctionsClass.tensor*, *degrees: None._VariableFunctionsClass.tensor*, *center: None._VariableFunctionsClass.tensor*) → None._VariableFunctionsClass.tensor

mode

ida_lib.operations.geometry_ops_functional.**scale_compose_data**(*data: dict*, *scale_factor: Union[float, None._VariableFunctionsClass.tensor]*, *center: Optional[None._VariableFunctionsClass.tens = None*) → dict

Parameters

- **data** – dict of elements to be transformed
- **scale_factor** – factor of scaling
- **center** – center of scaling. By default its taken the center of the image

Returns transformed data

ida_lib.operations.geometry_ops_functional.**scale_coordinates_matrix**(*matrix_coordinates: None._VariableFunctionsClass.te matrix: None._VariableFunctionsClass.te* → None._VariableFunctionsClass.te

ida_lib.operations.geometry_ops_functional.**scale_image**(*img: None._VariableFunctionsClass.tensor*, *scale_factor: None._VariableFunctionsClass.tensor*, *center: None._VariableFunctionsClass.tensor*) → None._VariableFunctionsClass.tensor

ida_lib.operations.geometry_ops_functional.**shear_compose_data**(*data: dict*, *shear_factor: tuple*) → dict

Parameters

- **data** – dict of elements to be transformed
- **shear_factor** – pixels of shearing

Returns transformed data

ida_lib.operations.geometry_ops_functional.**shear_coordinates_matrix**(*matrix_coordinates: None._VariableFunctionsClass.te*
matrix: None._VariableFunctionsClass.te
→ None._VariableFunctionsClass.te

ida_lib.operations.geometry_ops_functional.**shear_image**(*img: None._VariableFunctionsClass.tensor*, *shear_factor: None._VariableFunctionsClass.tensor*) → None._VariableFunctionsClass.tensor

ida_lib.operations.geometry_ops_functional.**translate_compose_data**(*data: dict*, *translation: Union[int, None._VariableFunctionsClass.tenso*
→ dict

Parameters

- **data** – dict of elements to be transformed
- **translation** – number of pixels to translate

Returns transformed data

ida_lib.operations.geometry_ops_functional.**translate_coordinates_matrix**(*matrix_coordinates: None._VariableFunctionsC*
translation: None._VariableFunctionsC
→ None._VariableFunctionsC

ida_lib.operations.geometry_ops_functional.**translate_image**(*img: None._VariableFunctionsClass.tensor*, *translation: None._VariableFunctionsClass.tensor*) → None._VariableFunctionsClass.tensor

ida_lib.operations.geometry_ops_functional.**vflip_compose_data**(*data: dict*) → dict

Parameters **data** – dict of elements to be transformed

Returns transformed data

ida_lib.operations.geometry_ops_functional.**vflip_coordinates_matrix**(*matrix: None._VariableFunctionsClass.te*
height: int) → None._VariableFunctionsClass.te

ida_lib.operations.geometry_ops_functional.**vflip_image**(*img: None._VariableFunctionsClass.tensor*) → None._VariableFunctionsClass.tensor

ida_lib.operations.pixel_ops_functional module

ida_lib.operations.pixel_ops_functional.**apply_blur**(*img*, *blur_size=5, 5*)

ida_lib.operations.pixel_ops_functional.**apply_gaussian_blur**(*img*, *blur_size=5, 5*)

Parameters

- **img** – input image to be transformed
- **blur_size** – number of surrounding pixels affecting each output pixel. (pixels on axis X, pixels on axis y)

Returns returns the transformed image

ida_lib.operations.pixel_ops_functional.**apply_gaussian_noise**(*image*, *var=20*)

ida_lib.operations.pixel_ops_functional.**apply_lut_by_pixel_function**(*function*, *image: numpy.ndarray*) → numpy.ndarray

Applies the input operation to the image using a LUT

Parameters

- **function** – Mathematical function that represents the operation to carry out in each pixel of the image
- **image** – input image

Returns

ida_lib.operations.pixel_ops_functional.**apply_poisson_noise**(*image*)

ida_lib.operations.pixel_ops_functional.**apply_salt_and_pepper_noise**(*image*, *amount=0.05*, *s_vs_p=0.5*)

ida_lib.operations.pixel_ops_functional.**apply_spekle_noise**(*image*, *mean=0*, *var=0.01*)

ida_lib.operations.pixel_ops_functional.**blur**(*img: Union[dict, None._VariableFunctionsClass.tensor, numpy.ndarray]*, *blur_size*) → Union[dict, None._VariableFunctionsClass.tensor, numpy.ndarray]

Parameters

- **img** – input image to be transformed
- **blur_size** – number of surrounding pixels affecting each output pixel. (pixels on axis X, pixels on axis y)

Returns returns the transformed image

ida_lib.operations.pixel_ops_functional.**change_brightness**(*image: Union[dict, None._VariableFunctionsClass.tensor, numpy.ndarray]*, *brightness: int*) → Union[dict, None._VariableFunctionsClass.tensor, numpy.ndarray]

Change the brightness of the input image.

Parameters

- **image** – input image to be normalized
- **brightness** – desired amount of brightness for the image 0 - no brightness 1 - same 2 - max brightness

Returns transformed image

ida_lib.operations.pixel_ops_functional.**change_contrast**(*image: Union[dict, None._VariableFunctionsClass.tensor, numpy.ndarray]*, *contrast*) → Union[dict, None._VariableFunctionsClass.tensor, numpy.ndarray]

:param image : input image to be transformed :param contrast: modification factor to be applied to the image contrast

- 0 - total contrast removal
- 1 - dont modify
- >1 - augment contrast

Returns returns the transformed image

ida_lib.operations.pixel_ops_functional.**change_gamma**(*image: Union[dict, None._VariableFunctionsClass.tensor, numpy.ndarray]*, *gamma: float*) → Union[dict, None._VariableFunctionsClass.tensor, numpy.ndarray]

Parameters

- **image** – input image to be transformed

- **gamma** – desired factor gamma * gamma = 0 -> removes image luminance (black output image) * gamma = 1 -> remains unchanged * gamma > 1 -> increases luminance

Returns returns the transformed image

ida_lib.operations.pixel_ops_functional.**gaussian_blur**(*img: Union[dict, None._VariableFunctionsClass.tensor, numpy.ndarray]*, *blur_size*) → Union[dict, None._VariableFunctionsClass.tensor, numpy.ndarray]

Parameters

- **img** – input image to be transformed
- **blur_size** – number of surrounding pixels affecting each output pixel. (pixels on axis X, pixels on axis y)

Returns returns the transformed image

ida_lib.operations.pixel_ops_functional.**gaussian_noise**(*image: Union[dict, None._VariableFunctionsClass.tensor, numpy.ndarray]*, *var=20*) → Union[dict, None._VariableFunctionsClass.tensor, numpy.ndarray]

Parameters

- **image** – input image to be transformed
- **var** – var of the gaussian distribution of noise

Returns returns the transformed image

ida_lib.operations.pixel_ops_functional.**get_brightness_function**(*brightness: int*)

Parameters **brightness** – brightness factor

Returns Return the lambda function of the brightness operation

ida_lib.operations.pixel_ops_functional.**get_contrast_function**(*contrast: float*)

Parameters **contrast** – modification factor to be applied to the image contrast

Returns Return the lambda function of the contrast operation

ida_lib.operations.pixel_ops_functional.**get_gamma_function**(*gamma*)

Parameters **gamma** – desired factor gamma

Returns Returns the lambda function of the gamma adjust operation

ida_lib.operations.pixel_ops_functional.**histogram_equalization**(*img: Union[dict, None._VariableFunctionsClass.tensor, numpy.ndarray]*) → Union[dict, None._VariableFunctionsClass.tensor, numpy.ndarray]

Parameters **img** – input image to be transformed

Returns returns the transformed image

ida_lib.operations.pixel_ops_functional.**normalize_image**(*img: numpy.ndarray*, *norm_type: int = 32*) → numpy.ndarray

Normalize the input image

Parameters

- **img** – input image to be normalized
- **norm_type** – opencv normalization type (‘ cv2.NORM_MINMAX’ |cv2.NORM_HAMMING |cv2.NORM_HAMMING2 |cv2.NORM_INF |cv2.NORM_RELATIVE . . .)

Returns normalized image

ida_lib.operations.pixel_ops_functional.**poisson_noise**(*image: Union[dict, None._VariableFunctionsClass.tensor, numpy.ndarray]*) → Union[dict, None._VariableFunctionsClass.tensor, numpy.ndarray]

Parameters **image** – input image to be transformed

Returns returns the transformed image

ida_lib.operations.pixel_ops_functional.**prepare_data_for_opencv**(*func*)

Decorator that prepares the input data to apply the transformation that only affects the image (color). :param func: color function to be applied to the data :return: processed data

ida_lib.operations.pixel_ops_functional.**salt_and_pepper_noise**(*image: Union[dict, None._VariableFunctionsClass.tensor, numpy.ndarray]*, *amount*, *s_vs_p*) → Union[dict, None._VariableFunctionsClass.tensor, numpy.ndarray]

Parameters

- **image** – input image to be transformed
- **amount** – percentage of image’s pixels to be occupied by noise
- **s_vs_p** – percentage of salt respect total noise. Default same salt (white pixel) as pepper (black pixels)

Returns returns the transformed image

ida_lib.operations.pixel_ops_functional.**spekle_noise**(*image: Union[dict, None._VariableFunctionsClass.tensor, numpy.ndarray]*, *mean=0*, *var=0.01*) → Union[dict, None._VariableFunctionsClass.tensor, numpy.ndarray]

Parameters

- **image** – input image to be transformed
- **mean** – mean of noise distribution
- **var** – variance of noise distribution

Returns returns the transformed image

ida_lib.operations.transforms module

ida_lib.operations.transforms.**hflip**(*data: dict*, *visualize: bool = False*) → dict

Horizontally flip the input data.

Parameters

- **data** – dict of elements to be transformed
- **visualize** – if true it activates the display tool to debug the transformation

Returns transformed data

ida_lib.operations.transforms.**vflip**(*data: dict*, *visualize: bool = False*) → dict

Vertically flip the input data.

Parameters

- **data** – dict of elements to be transformed
- **visualize** – if true it activates the display tool to debug the transformation

Returns transformed data

ida_lib.operations.transforms.**affine**(*data: dict*, *matrix: None._VariableFunctionsClass.tensor*, *visualize: bool = False*) → dict

Applies affine transformation to the data

:param data dict of elements to be transformed :param matrix: matrix of transformation :param visualize: if true it activates the display tool to debug the transformation :return: transformed data

ida_lib.operations.transforms.**rotate**(*data: dict*, *degrees: float*, *visualize: bool = False*, *center: Union[None, torch.Tensor] = None*) → dict

Rotate each element of the input data by the indicated degrees counterclockwise

Parameters

- **data** – dict of elements to be transformed
- **degrees** – degrees of rotation
- **center** – center of rotation. If center is None, it is taken the center of the image to apply the rotation
- **visualize** – if true it activates the display tool to debug the transformation

Returns transformed data

ida_lib.operations.transforms.**shear**(*data: dict*, *shear_factor: tuple*, *visualize: bool = False*) → dict

Shear input data by the input shear factor

Parameters

- **data** – dict of elements to be transformed

- **shear_factor** – pixels to shear
- **visualize** – if true it activates the display tool to debug the transformation

Returns transformed data

ida_lib.operations.transforms.**scale**(*data: dict*, *scale_factor*, *visualize: bool = False*, *center: Union[None, torch.Tensor] = None*) → dict

Scale each element of the input data by the input factor.

Parameters

- **data** – dict of elements to be transformed
- **scale_factor** – factor of scaling to be applied * scale_factor < 1 -> output image is smaller than input one * scale_factor = 1 -> output image is is the same as the input image * scale_factor = 2 -> each original pixel occupies 2 pixels in the output image
- **center** – center of scaling. If center is None, it is taken the center of the image to apply the scale
- **visualize** – if true it activates the display tool to debug the transformation

Returns transformed data

ida_lib.operations.transforms.**translate**(*data: dict*, *translation: tuple*, *visualize: bool = False*) → dict

Translate input by the input translation.

Parameters

- **data** – dict of elements to be transformed
- **translation** – number of pixels to be translated
- **visualize** – if true it activates the display tool to debug the transformation

Returns transformed data

ida_lib.operations.transforms.**change_gamma**(*data: Union[dict, torch.Tensor, numpy.ndarray]*, *gamma*, *visualize: bool = False*) → Union[dict, torch.Tensor, numpy.ndarray]

Adjust image's gamma (luminance correction) . if the input data is a dictionary, only those corresponding

to an image are altered

Parameters

- **data** – dict of elements to be transformed
- **gamma** – desired gamma factor (luminance of image)
  - gamma = 0 -> removes image luminance (black output image)
  - gamma = 1 -> remains unchanged
  - gamma > 1 -> increases luminance
- **visualize** – if true it activates the display tool to debug the transformation

Returns transformed data

ida_lib.operations.transforms.**change_contrast**(*data: Union[dict, torch.Tensor, numpy.ndarray]*, *contrast: float*, *visualize: bool = False*) → Union[dict, torch.Tensor, numpy.ndarray]

Change the image contrast. if the input data is a dictionary, only those corresponding to an image are altered

Parameters

- **data** – dict of elements to be transformed
- **contrast** – desired contrast factor to the data * contrast = 0 -> removes image contrast (white output image) * contrast = 1 -> remains unchanged * contrast > 1 -> increases contrast

:param visualize : if true it activates the display tool to debug the transformation :return: transformed data

ida_lib.operations.transforms.**change_brightness**(*data: Union[dict, torch.Tensor, numpy.ndarray]*, *bright=1*, *visualize: bool = False*) → Union[dict, torch.Tensor, numpy.ndarray]

Change the image brightness. if the input data is a dictionary, only those corresponding to an image are altered

Parameters

- **data** – dict of elements to be transformed
- **bright** – desired brightness amount for the data
  - brightness = 0 -> removes image brightness (black output image)
  - brightness = 1 -> remains unchanged
  - brightness > 1 -> increases brightness
- **visualize** – if true it activates the display tool to debug the transformation

Returns transformed data

ida_lib.operations.transforms.**equalize_histogram**(*data: Union[dict, torch.Tensor, numpy.ndarray]*, *visualize: bool = False*) → Union[dict, torch.Tensor, numpy.ndarray]

Equalize image histogram. if the input data is a dictionary, only those corresponding to an image are altered

Parameters

- **data** – dict of elements to be transformed
- **visualize** – if true it activates the display tool to debug the transformation

Returns transformed data

ida_lib.operations.transforms.**inject_gaussian_noise**(*data: Union[dict, torch.Tensor, numpy.ndarray]*, *var=0.5*, *visualize: bool = False*) → Union[dict, torch.Tensor, numpy.ndarray]

Inject gaussian noise. If the input data is a dictionary, only those corresponding to an image are altered

Parameters

- **data** – dict of elements to be transformed
- **var** – variance of the noise distribution
- **visualize** – if true it activates the display tool to debug the transformation

Returns transformed data

ida_lib.operations.transforms.**inject_poisson_noise**(*data: Union[dict, torch.Tensor, numpy.ndarray]*, *visualize: bool = False*) → Union[dict, torch.Tensor, numpy.ndarray]

Inject poisson noise. if the input data is a dictionary, only those corresponding to an image are altered

Parameters

- **data** – dict of elements to be transformed
- **visualize** – if true it activates the display tool to debug the transformation

Returns transformed data

ida_lib.operations.transforms.**inject_spekle_noise**(*data: Union[dict, torch.Tensor, numpy.ndarray]*, *mean=0*, *var=0.01*, *visualize: bool = False*) → Union[dict, torch.Tensor, numpy.ndarray]

Inject poisson noise. if the input data is a dictionary, only those corresponding to an image are altered

Parameters

- **data** – dict of elements to be transformed
- **mean** – mean of noise distribution
- **var** – variance of noise distribution
- **visualize** – if true it activates the display tool to debug the transformation

Returns transformed data

ida_lib.operations.transforms.**inject_salt_and_pepper_noise**(*data: Union[dict, torch.Tensor, numpy.ndarray]*, *amount=0.05*, *s_vs_p=0.5*, *visualize: bool = False*) → Union[dict, torch.Tensor, numpy.ndarray]

Inject salt and pepper noise if the input data is a dictionary, only those corresponding to an image are altered

Parameters

- **data** – dict of elements to be transformed
- **amount** – percentage of image's pixels to be occupied by noise

:param s_vs_p : noise type distribution. Default same salt (white pixel) as pepper (black pixels) :param visualize: if true it activates the display tool to debug the transformation :return: transformed data

ida_lib.operations.transforms.**blur**(*data: Union[dict, torch.Tensor, numpy.ndarray]*, *blur_size=5, 5*, *visualize: bool = False*) → Union[dict, torch.Tensor, numpy.ndarray]

Blur image. if the input data is a dictionary, only those corresponding to an image are altered

Parameters

- **data** – dict of elements to be transformed
- **blur_size** – number of surrounding pixels affecting each output pixel. (pixels on axis X, pixels on axis y)
- **visualize** – if true it activates the display tool to debug the transformation

Returns transformed data

ida_lib.operations.transforms.**gaussian_blur**(*data: Union[dict, torch.Tensor, numpy.ndarray]*, *blur_size=5, 5*, *visualize: bool = False*) → Union[dict, torch.Tensor, numpy.ndarray]

Blurring an image by a Gaussian function. if the input data is a dictionary, only those corresponding to an image are altered

Parameters

- **data** – dict of elements to be transformed
- **blur_size** – number of surrounding pixels affecting each output pixel. (pixels on axis X, pixels on axis y)
- **visualize** – if true it activates the display tool to debug the transformation

Returns transformed data

ida_lib.operations.utils module

ida_lib.operations.utils.**add_new_axis**(*arr: numpy.ndarray*)

ida_lib.operations.utils.**arrays_equal**(*arr1*, *arr2*)

ida_lib.operations.utils.**data_to_numpy**(*data*)

ida_lib.operations.utils.**dtype_to_torch_type**(*im_type: numpy.dtype*)

Maps the numpy type to the equivalent torch.type

Parameters **im_type** – numpy type

Returns torch.type

ida_lib.operations.utils.**element_to_dict_csv_format**(*item*, *name*)

ida_lib.operations.utils.**get_principal_type**(*data: dict*)

ida_lib.operations.utils.**get_torch_image_center**(*data*)

ida_lib.operations.utils.**homogeneous_points_to_list**(*keypoints*)

ida_lib.operations.utils.**homogeneous_points_to_matrix**(*keypoints*)

ida_lib.operations.utils.**is_a_normalized_image**(*image*)

ida_lib.operations.utils.**is_numpy_data**(*data*)

ida_lib.operations.utils.**keypoints_to_homogeneous_and_concatenate**(*keypoints*, *resize_factor=None*)

ida_lib.operations.utils.**keypoints_to_homogeneous_functional**(*keypoints*)

ida_lib.operations.utils.**map_value**(*x*, *in_min*, *in_max*, *out_min*, *out_max*)

ida_lib.operations.utils.**mask_change_to_01_functional**(*mask*)

ida_lib.operations.utils.**remove_digits**(*label: str*)

ida_lib.operations.utils.**round_torch**(*arr: None._VariableFunctionsClass.tensor*, *n_digits: int = 3*)

ida_lib.operations.utils.**save_im**(*tensor*, *title*)

Module contents

3.3.2 Submodules

3.3.3 ida_lib.visualization module

ida_lib.visualization.**visualize**(*images: dict*, *images_originals: dict*, *max_images: int = 5*)
Generate the bokeh plot of the input batch transformation

Parameters

- **images** – list of transformed items (dict of image and other objects)
- **images_originals** – list of original items (dict of image and other objects)
- **max_images** – max number of tabs to be shown

ida_lib.visualization.**plot_image_transformation**(*data*, *data_original*)
Generate the bokeh plot of the input batch transformation

Parameters

- **data** – input dict element
- **data_original** – original input element (before transforms)

3.4 Examples

You can find all the examples here: https://github.com/raquelvilas18/ida_lib/tree/master/examples

3.4.1 Pipeline usage example

Pipeline Usage example

```
"""In this file an example of how to use the idaLib pipeline is shown, in which you␣
↪can see:
* how to declare the pipeline
* which format to use for the input elements
* how to display or not the results
* and how to execute it in general.
```

(continues on next page)

(continued from previous page)

```
For more information see the documentation
"""

from time import time

import numpy as np

from ida_lib.core.pipeline import *
from ida_lib.core.pipeline_geometric_ops import *
from ida_lib.core.pipeline_local_ops import *

data_type = np.uint8

# Read the example image
img: np.ndarray = cv2.imread('../micky.jpg', )
# opencv read in format BGR but IDALib works on RGB
img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
#img = img.astype('float32')  # Example of bits per pixel used

short_size = min(img.shape[0], img.shape[1])

# Generate an example of segmentation map over the image
segmap = np.zeros((img.shape[0], img.shape[1], 1), dtype=data_type)
segmap[28:171, 35:485, 0] = 1
segmap[10:25, 30:245, 0] = 2
segmap[10:25, 70:385, 0] = 3
segmap[10:110, 5:210, 0] = 4
segmap[18:223, 10:110, 0] = 5

# Generate 2 examples of masks
mask_example1 = np.zeros((img.shape[0], img.shape[1], 1), dtype=data_type)
mask_example1[0:50, 0:50] = 1
mask_example2 = np.zeros((img.shape[0], img.shape[1], 1), dtype=data_type)
mask_example2[-150:, -50:] = 1

# Generate an example of heatmap over the image
x = np.random.randn(img.shape[0] // 4 * img.shape[1] // 4)
y = np.random.randn(img.shape[0] // 4 * img.shape[1] // 4)
# Create heatmap
heatmap, xedges, yedges = np.histogram2d(x, y, bins=(img.shape[0] // 4, img.shape[1] /
↪/ 4))
heatmap_complete = np.zeros((img.shape[0], img.shape[1], 1))
heatmap_complete[0:img.shape[0] // 4, 0:img.shape[1] // 4, 0] = 1

# Generate an example of coordinates of keypoints over the image
# the list of keypoints can be expressed as an array of nx2 dimensions or as a list␣
↪of arrays (1x2) (of 2 coordinates)
number_of_points = 20
# generate 20 random coordinates (to make sure they don't go outside the image␣
↪boundaries set short_Size as the limit)
random_coordinates = np.random.randint(1, short_size, number_of_points * 2).
↪reshape(number_of_points, 2)

# Generate the input item of the pipeline. Its very importantto name each element␣
↪with its data type so that the
# pipeline understands them. *If the item contains more than one element of each type,
↪ just number them like mask1
```

(continues on next page)

(continued from previous page)

```
# and mask2
data = {'image': img, 'keypoints': random_coordinates, 'mask1': mask_example1, 'mask2
↪': mask_example2,
        'heatmap': heatmap_complete}

# For this example we are going to use the same identical input element but repeated␣
↪n times to create a batch so we
# can see the different transformations
samples = 10
batch = [data.copy() for _ in range(samples)]

start_time = time()  # time measurement

# Define the pipeline and operations.
pip = Pipeline(interpolation='nearest',
               pipeline_operations=(
                   ScalePipeline(probability=1, scale_factor=0.5),
                   ShearPipeline(probability=0, shear=(0.2, 0.2)),
                   TranslatePipeline(probability=0, translation=(10,50)),
                   HflipPipeline(probability=0, exchange_points=[(0, 5), (1, 6)]),
                   RandomRotatePipeline(probability=0, degrees_range=(-20, 20)),
                   GaussianNoisePipeline(probability=0)))

# pass the batch through the pipeline and visualize the transformations
batch = pip(batch, visualize=True)

consumed_time = time() - start_time
# keep in mind that visualization is a significant overhead, so to take a good␣
↪measure of
# performance set visualize=False
print("Total time consumed to process " + str(samples) + " samples: " + str(consumed_
↪time))
print("Time per sample: :" + str(consumed_time / samples))
```

Dataloader Usage example

```
"""
This file has an example of how to use IDALib's own DataLoader which includes a␣
↪pipeline to perform image data
augmentation on your data.
This code follows the pytorch example of  of using a dataloader
https://pytorch.org/tutorials/beginner/data_loading_tutorial.html  but adapted to the␣
↪ida-lib dataloader
"""

import os

import kornia
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from skimage import io

from ida_lib.core.pipeline_geometric_ops import TranslatePipeline, VflipPipeline,␣
↪HflipPipeline, RandomShearPipeline
```

(continues on next page)

(continued from previous page)

```
from ida_lib.core.pipeline_pixel_ops import ContrastPipeline
from ida_lib.image_augmentation.data_loader import *


# Firstly create custom dataset to read the input data
class FaceLandmarksDataset(Dataset):
    """Face Landmarks dataset."""

    def __init__(self, csv_file, root_dir):
        """
        Args:
            csv_file (string): Path to the csv file with annotations.
            root_dir (string): Directory with all the images.
        """
        self.landmarks_frame = pd.read_csv(csv_file)
        self.root_dir = root_dir

    def __len__(self):
        return len(self.landmarks_frame)

    def __getitem__(self, idx):
        if torch.is_tensor(idx):
            idx = idx.tolist()

        img_name = os.path.join(self.root_dir,
                                self.landmarks_frame.iloc[idx, 0])
        image = io.imread(img_name)
        landmarks = self.landmarks_frame.iloc[idx, 1:]
        landmarks = np.array([landmarks])
        landmarks = landmarks.astype('float').reshape(-1, 2)
        sample = {'id': self.landmarks_frame.iloc[idx, 0], 'image': image, 'keypoints
↪': landmarks}
        return sample


# Auxiliar function to display elements
def show_landmarks(image, landmarks):
    """Show image with landmarks"""
    img = kornia.tensor_to_image(image.byte())
    plt.imshow(img)
    landmarks = landmarks.cpu().numpy()
    plt.scatter(landmarks[:, 0], landmarks[:, 1], s=10, marker='o', c='r')
    plt.show()
    plt.pause(0.001)  # pause a bit so that plots are updated


# initialize custom dataset
face_dataset = FaceLandmarksDataset(csv_file='faces/face_landmarks.csv',
                                    root_dir='faces/')

# initialite the custom dataloader
dataloader = AugmentDataLoader(dataset=face_dataset,
                               batch_size=4,
                               shuffle=True,
                               pipeline_operations=(
                                   TranslatePipeline(probability=1, translation=(30,␣
↪10)),
```

(continues on next page)

(continued from previous page)

```
                                    VflipPipeline(probability=0.5),
                                    HflipPipeline(probability=0.5),
                                    ContrastPipeline(probability=0.5, contrast_
↪factor=1),
                                    RandomShearPipeline(probability=0.5, shear_
↪range=(0, 0.5))),
                                resize=(500, 300),
                                # we must indicate the size of the resize because the␣
↪images are not all the same size
                                interpolation='bilinear',
                                padding_mode='zeros'
                                )

number_of_iterations = 3  # number of times the entire dataset is processed
for epoch in range(number_of_iterations - 1):
    for i_batch, sample_batched in enumerate(dataloader):  # our dataloader works␣
↪like a normal dataloader
        print(i_batch, )
        keypoints = sample_batched['keypoints'][0, :, :]
        show_landmarks(sample_batched['image'][0], keypoints)
    print('all elements of the original dataset have been displayed and processed')
```

Image Augmentation to Disk example

```
import os

import numpy as np
import pandas as pd
import torch
from skimage import io
from torch.utils.data import Dataset

from ida_lib.core.pipeline_geometric_ops import RandomScalePipeline, HflipPipeline
from ida_lib.core.pipeline_pixel_ops import RandomContrastPipeline
from ida_lib.image_augmentation.augment_to_disk import AugmentToDisk


# Create custom dataset to read the input data to be augmented
class FaceLandmarksDataset(Dataset):
    """Face Landmarks dataset."""

    def __init__(self, csv_file, root_dir, transform=None):
        """
        Args:
            csv_file (string): Path to the csv file with annotations.
            root_dir (string): Directory with all the images.
            transform (callable, optional): Optional transform to be applied
                on a sample.
        """
        self.landmarks_frame = pd.read_csv(csv_file)
        self.root_dir = root_dir
        self.transform = transform

    def __len__(self):
        return len(self.landmarks_frame)
```

(continues on next page)

(continued from previous page)

```
    def __getitem__(self, idx):
        if torch.is_tensor(idx):
            idx = idx.tolist()
        img_name = os.path.join(self.root_dir,
                                self.landmarks_frame.iloc[idx, 0])
        item_id = (self.landmarks_frame.iloc[idx, 0]).split('.')[0]
        image = io.imread(img_name)
        landmarks = self.landmarks_frame.iloc[idx, 1:]
        landmarks = np.array([landmarks])
        landmarks = landmarks.astype('float').reshape(-1, 2)
        sample = {'id': item_id, 'image': image, 'landmarks': landmarks}
        return sample


# Inicialize the custom datset

face_dataset = FaceLandmarksDataset(csv_file='faces/face_landmarks.csv',
                                    root_dir='faces/')

# parameter setting and initialization

augmentor = AugmentToDisk(dataset=face_dataset,  # custom dataset that provides the␣
↪input data
                          samples_per_item=5,  # number of samples per imput item
                          operations=(RandomScalePipeline(probability=0.6, scale_
↪range=(0.8, 1.2), center_deviation=20),
                                      HflipPipeline(probability=0.5),
                                      RandomContrastPipeline(probability=0.5, contrast_
↪range=(1, 1.5))),
                          interpolation='nearest',
                          padding_mode='zeros',
                          resize=(250, 250),  # Here resizing is necessary because␣
↪the input images have different sizes
                          output_extension='.jpg',
                          output_csv_path='anotations.csv',
                          output_path='./augmented_custom')

augmentor()  # Run the augmentation
```

Neural Net example

```
from __future__ import print_function

import os
import os.path
import sys
import torch
import torch.optim as optim
import matplotlib.pyplot as plt
import numpy as np
import torch.nn as nn
import torch.nn.functional as F

from ida_lib.core.pipeline_geometric_ops import HflipPipeline, RandomShearPipeline, \
```

(continues on next page)

(continued from previous page)

```
    RandomRotatePipeline
from ida_lib.core.pipeline_pixel_ops import NormalizePipeline, RandomContrastPipeline
from ida_lib.image_augmentation.data_loader import AugmentDataLoader

import kornia
if sys.version_info[0] == 2:
    import cPickle as pickle
else:
    import pickle

import torch.utils.data as data
from torchvision.datasets.utils import download_url, check_integrity

'''https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html#define-a-
↪convolutional-neural-network'''

# create a custom cifar Dataset to read the data
class custom_CIFAR10(data.Dataset):
    """`CIFAR10 <https://www.cs.toronto.edu/~kriz/cifar.html>`_ Dataset.

    Args:
        root (string): Root directory of dataset where directory
            ``cifar-10-batches-py`` exists.
        train (bool, optional): If True, creates dataset from training set, otherwise
            creates from test set.
        transform (callable, optional): A function/transform that  takes in an PIL␣
↪image
            and returns a transformed version. E.g, ``transforms.RandomCrop``
        target_transform (callable, optional): A function/transform that takes in the
            target and transforms it.
        download (bool, optional): If true, downloads the dataset from the internet␣
↪and
            puts it in root directory. If dataset is already downloaded, it is not
            downloaded again.

    """
    base_folder = 'cifar-10-batches-py'
    url = "http://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz"
    filename = "cifar-10-python.tar.gz"
    tgz_md5 = 'c58f30108f718f92721af3b95e74349a'
    train_list = [
        ['data_batch_1', 'c99cafc152244af753f735de768cd75f'],
        ['data_batch_2', 'd4bba439e000b95fd0a9bffe97cbabec'],
        ['data_batch_3', '54ebc095f3ab1f0389bbae665268c751'],
        ['data_batch_4', '634d18415352ddfa80567beed471001a'],
        ['data_batch_5', '482c414d41f54cd18b22e5b47cb7c3cb'],
    ]

    test_list = [
        ['test_batch', '40351d587109b95175f43aff81a1287e'],
    ]

    def __init__(self, root, train=True,
                 transform=None, target_transform=None,
                 download=False):
        self.root = os.path.expanduser(root)
        self.transform = transform
```

(continues on next page)

(continued from previous page)

```
        self.target_transform = target_transform
        self.train = train  # training set or test set

        if download:
            self.download()

        if not self._check_integrity():
            raise RuntimeError('Dataset not found or corrupted.' +
                               ' You can use download=True to download it')

        # now load the picked numpy arrays
        if self.train:
            self.train_data = []
            self.train_labels = []
            for fentry in self.train_list:
                f = fentry[0]
                file = os.path.join(root, self.base_folder, f)
                fo = open(file, 'rb')
                if sys.version_info[0] == 2:
                    entry = pickle.load(fo)
                else:
                    entry = pickle.load(fo, encoding='latin1')
                self.train_data.append(entry['data'])
                if 'labels' in entry:
                    self.train_labels += entry['labels']
                else:
                    self.train_labels += entry['fine_labels']
                fo.close()

            self.train_data = np.concatenate(self.train_data)
            self.train_data = self.train_data.reshape((50000, 3, 32, 32))
            self.train_data = self.train_data.transpose((0, 2, 3, 1))  # convert to␣
↪HWC
        else:
            f = self.test_list[0][0]
            file = os.path.join(root, self.base_folder, f)
            fo = open(file, 'rb')
            if sys.version_info[0] == 2:
                entry = pickle.load(fo)
            else:
                entry = pickle.load(fo, encoding='latin1')
            self.test_data = entry['data']
            if 'labels' in entry:
                self.test_labels = entry['labels']
            else:
                self.test_labels = entry['fine_labels']
            fo.close()
            self.test_data = self.test_data.reshape((10000, 3, 32, 32))
            self.test_data = self.test_data.transpose((0, 2, 3, 1))  # convert to HWC

    def __getitem__(self, index):
        """
        Args:
            index (int): Index

        Returns:
            tuple: (image, target) where target is index of the target class.
```

(continues on next page)

(continued from previous page)

```
        """
        if self.train:
            img, target = self.train_data[index], self.train_labels[index]
        else:
            img, target = self.test_data[index], self.test_labels[index]

        item = {'image': img, 'target': target}
        return item  # modified to return a dict instead of a tuple

    def __len__(self):
        if self.train:
            return len(self.train_data)
        else:
            return len(self.test_data)

    def _check_integrity(self):
        root = self.root
        for fentry in (self.train_list + self.test_list):
            filename, md5 = fentry[0], fentry[1]
            fpath = os.path.join(root, self.base_folder, filename)
            if not check_integrity(fpath, md5):
                return False
        return True

    def download(self):
        import tarfile

        if self._check_integrity():
            print('Files already downloaded and verified')
            return

        root = self.root
        download_url(self.url, root, self.filename, self.tgz_md5)

        # extract file
        cwd = os.getcwd()
        tar = tarfile.open(os.path.join(root, self.filename), "r:gz")
        os.chdir(root)
        tar.extractall()
        tar.close()
        os.chdir(cwd)

#auxiliar function to plot batches images
def plot_tuple_batch(images, labels):
    batch_size = images.shape[0]
    images = images.cpu()
    labels = labels.cpu()

    fig, axs = plt.subplots(1, batch_size, figsize=(16, 10))
    for i in range(batch_size):
        axs[i].axis('off')
        axs[i].set_title(classes[labels[i].item()])
        img: np.ndarray = kornia.tensor_to_image((images[i] * 255).byte())
        axs[i].imshow(img)
    plt.show()

# initialize train dataset
```

(continues on next page)

(continued from previous page)

```
trainset = custom_CIFAR10(root='./data', train=True,
                          download=True)
# define the cnn model
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

# def train loop
def train():
    net = Net()
    net = net.cuda()

    # Configure parameters
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)

    # TRAIN
    from time import time
    start_time = time()
    for epoch in range(1):  # loop over the dataset multiple times
        running_loss = 0.0
        for i, data in enumerate(trainloader, 0):
            # get the inputs; data is a list of [inputs, labels]
            inputs, labels = data
            inputs = (inputs.float())
            labels = labels.to('cuda')
            optimizer.zero_grad()

            # forward + backward + optimize
            outputs = net(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            # print statistics
            running_loss += loss.item()
            if i % 2000 == 1999:  # print every 2000 mini-batches
                print('[%d, %5d] loss: %.3f' %
                      (epoch + 1, i + 1, running_loss / 2000))
                running_loss = 0.0
    consumed_time = time() - start_time
    print(consumed_time)
```

(continues on next page)

(continued from previous page)

```
    print('Finished Training')
    torch.save(net.state_dict(), PATH)

#def test loop
def test():
    images, labels = dataiter.next()

    # print images
    plot_tuple_batch(images, labels)
    print('GroundTruth: ', ' '.join('%5s' % classes[labels[j]] for j in range(4)))
    net = Net()
    net = net.cuda()
    net.load_state_dict(torch.load(PATH))
    outputs = net(images)
    _, predicted = torch.max(outputs, 1)

    print('Predicted: ', ' '.join('%5s' % classes[predicted[j]]
                                  for j in range(4)))
    class_correct = list(0. for i in range(10))
    class_total = list(0. for i in range(10))
    with torch.no_grad():
        for data in testloader:
            images, labels = data
            labels = labels.to('cuda')
            outputs = net(images)
            _, predicted = torch.max(outputs, 1)
            c = (predicted == labels).squeeze()
            for i in range(4):
                label = labels[i]
                class_correct[label] += c[i].item()
                class_total[label] += 1

    for i in range(10):
        print('Accuracy of %5s : %2d %%' % (
            classes[i], 100 * class_correct[i] / class_total[i]))

# Create the dataloader with ida_lib augmentations
trainloader = AugmentDataLoader(dataset=trainset,
                                batch_size=4,
                                shuffle=True,
                                resize=(500, 500),
                                pipeline_operations=(NormalizePipeline(probability=1),
                                                     HflipPipeline(probability=1),
 ↪RandomRotatePipeline(probability=0, degrees_range=(-15, 15)),
 ↪RandomContrastPipeline(probability=0, contrast_range=(0.8, 1.2)),
 ↪RandomShearPipeline(probability=0, shear_range=(0, 0.5))),
                                interpolation='bilinear',
                                padding_mode='zeros',
                                output_format='tuple',
                                output_type=torch.float32
                                )
# initialize test dataset
testset = custom_CIFAR10(root='./data', train=False,
                         download=True)
```

(continues on next page)

(continued from previous page)

```
# Create the dataloader with ida_lib augmentations
testloader = AugmentDataLoader(dataset=testset,
                               batch_size=4,
                               shuffle=False,
                               pipeline_operations=(NormalizePipeline(probability=1),
                                                    HflipPipeline(probability=0.5),
↪RandomRotatePipeline(probability=0.8, degrees_range=(-15, 15)),
↪RandomContrastPipeline(probability=0, contrast_range=(0.8, 1.2)),
                                                    RandomShearPipeline(probability=0,
↪ shear_range=(0, 0.5))),
                               interpolation='bilinear',
                               padding_mode='zeros',
                               output_format='tuple',
                               output_type=torch.float32
                               )
# clases
classes = ('plane', 'car', 'bird', 'cat',
           'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
# path to save weights
PATH = './cifar_net2.pth'


# get some random training images
dataiter = iter(trainloader)
images, labels = dataiter.next()

# plot some items of train
plot_tuple_batch(images, labels)

# train the net
train()

# test the results
test()
```

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

4.1 Built with

- Pytorch. - The machine learning framework used
- Bokeh. - The visualization library used
- Kornia. - computer vision library that is used as a base for transformations
- OpenCV. - computer vision library that is used as a base for transformations
- Pycharm. - Development IDE

4.2 Acknowledgements

- **Nicolás Vila Blanco** : project co-author
- **María José Carreira Nouche**: project co-author
- **CITiUS**: support company

PYTHON MODULE INDEX

i

INDEX

H

I

K

M

N

O

P

R

S

T

V