A New Framework for Evaluating Image Quality Including Deep Learning Task Performances as a Proxy

Pau Gallés, Katalin Takáts, Miguel Hernández-Cabronero, David Berga, Luciano Pega, Laura Riordan-Chen, Clara Garcia, Guillermo Becker, Adan Garriga, Anica Bukva, Joan Serra-Sagristà, *Senior Member, IEEE*, David Vilaseca, and Javier Marín

***Abstract*—IQUAFLOW is a framework that provides a set of tools to assess image quality. The user can add custom metrics that can be easily integrated and a set of unsupervised methods is offered by default. Furthermore, IQUAFLOW measures quality by using the performance of AI models trained on the images as a proxy. This also helps to easily make studies of performance degradation of several modifications of the original dataset, for instance, with images reconstructed after different levels of lossy compression; satellite images would be a use case example, since they are commonly compressed before downloading to the ground. In this situation, the optimization problem involves finding images that, while being compressed to their smallest possible file size, still maintain sufficient quality to meet the required performance of the deep learning algorithms. Thus, a study with IQUAFLOW is suitable for such case. All this development is wrapped in MLFLOW: an interactive tool used to visualize and summarize the results. This document describes different use cases and provides links to their respective repositories. To ease the creation of new studies, we include a cookiecutter repository. The source code, issue tracker and aforementioned repositories are all hosted on GitHub.**

***Index Terms*—Artificial intelligence, data compression, image analysis, image processing, image resolution, image segmentation, object detection, quality control, remote sensing.**

Manuscript received 11 July 2023; revised 7 September 2023 and 18 November 2023; accepted 4 December 2023. Date of publication 13 December 2023; date of current version 22 January 2024. This work was supported in part by the Ministry of Science and Innovation and in part by the European Union within the framework of Retos-Colaboración of the Research, Development and Innovation State Program Oriented to the Challenges of Society, within the Scientific, Technical and Innovation State Research Plan 2017-2020, with the main objective of promoting technological development, innovation, and quality research. Project reference RTC2019-007434-7. *(Corresponding author: Pau Gallés)*

Pau Gallés, Katalin Takáts, Luciano Pega, Laura Riordan-Chen, Clara Garcia, Guillermo Becker, David Vilaseca, and Javier Marín are with the Satellogic Inc, 08010 Barcelona, Spain (e-mail: pau.galles@gmail.com; katalin.takats@satellogic.com; luciano.pega@satellogic.com; laurarc@satellogic.com; clara.garciamoll@gmail.com; guillermo.becker@satellogic.com; vila@satellogic.com; jmarin@satellogic.com).

Miguel Hernández-Cabronero and Joan Serra-Sagristà are with the Universitat Autònoma de Barcelona - UAB-DEIC-GICI, 08193 Bellaterra, Spain (e-mail: miguel.hernandez@uab.cat; joan.serra@uab.cat).

David Berga, Adan Garriga, and Anica Bukva are with the EURECAT - Multimedia Technologies Unit, 08005 Barcelona, Spain (e-mail: david.berga@eurecat.org; adan.garriga@eurecat.org; anicabukva@yahoo.com).

[Online]. Available: https://github.com/satellogic/iquaflow-use-case-cookiecutter

[Online]. Available: https://github.com/satellogic/iquaflow

Digital Object Identifier 10.1109/JSTARS.2023.3342475

I. Introduction

THE increasing interest and investment in low-cost Earth Observation (EO) satellites (such as nanosats and microsats) in recent years has made possible the creation and improvement of multiple applications that feed on the images obtained [1]. Thanks to the emergence of new sensors and advancements in image fusion techniques [2], [3], and [4], the quality of these images has increased significantly, thus contributing to the accuracy and efficiency of the applications that use them, giving rise to the NewSpace era, which has led to the emergence of many new companies.

It is the case of Satellogic, a company that was founded in 2010 and specializes in EO data and analytical imagery solutions. Satellogic designs, builds, and operates its own fleet of EO satellites to frequently collect affordable high-resolution imagery for decision-making in a broad range of industrial, environmental, and government applications. The Satellogic satellite constellation consists of individual small satellites, named NewSats. Each of the NewSat satellites has a multispectral and a hyperspectral sensor. The multispectral data are used in some of the studies mentioned in the present article.

Imagery is a means to an end. Large-scale data analytics and artificial intelligence equipment turns imagery into answers to help industries, governments and individuals solve problems, facilitate decision making, and generate competitive advantage. In this context, the growth in the number of EO users has also increased the land area of interest and thus the amount of imagery required to meet their needs. As a result, an increasing volume of EO image data needs to be transmitted. However, the transmission capacity between satellites and ground stations has not grown at the same rate as the required volume of imagery.

Due to their orbit, satellites can only contact these stations for limited periods of time and with limited bandwidths. Moreover, the total amount of energy available for all satellite tasks—including image capture, processing, and transmission—is also limited. These limitations pose a bottleneck in the operation of EO satellites, since the quantity and quality of images reaching the ground is determined by the efficiency of transmission. In turn, this limitation has a negative impact both on the costs of EO services and on users and applications that increasingly demand

higher quality and frequency of observation of the regions of interest.

Given the aforementioned needs, this project seeks to optimize the decision making process when selecting an image processing algorithm to optimize the storage, quality, and transmission of images either on EO satellites or on the ground. Optimization refers to finding and detecting the satellite image parameters that allow the smallest compressed data volume that provides sufficient quality to meet the required performance of the deep learning algorithms used. In order to achieve this objective, a new framework named IQUAFLOW (acronym of image quality assessment) is proposed. The framework includes the necessary tools to draw conclusions based on specific metrics.

IQUAFLOW consists of multiple Python modules that can be imported in any image quality-related project for research or production. The framework also includes the necessary tools to automatize and organize experiments. Most of similar tools help to implement better tractability model trainings in order to perform model selection or model performance degradation studies (See [5], [6], and [7]). Instead, IQUAFLOW, analyzes the change in performance of the models based on modifications in quality of the training images. Having this perspective in mind, IQUAFLOW organizes training executions based on qualities of the images. Thus, this approach is considered image quality selection rather than the common practice of image selection. In addition, this framework provides a variety of tools that help to speed up the machine learning development such as dataset sanity check, dataset statistics, dataset visualization, and dataset modification algorithms. It is designed to easily adapt to conventions. The tool can be imported into any kind of deep learning framework, such as tensorflow, keras, or pytorch. This allows to effortlessly generate experiments on any existing project regardless of its dependencies. Internally, this project relies in MLFLOW, which enables to organize the information locally or even in a remote server with an interaction that is abstracted to the user. As a result, the user does not need to learn another cumbersome framework. Ultimately, IQUAFLOW only needs logging parameters that are easy to adapt (such as generating a json or saving files in a specific folder that is indicated as an input argument in the user script). Fig. 1 shows the typical workflow of a study made with IQUAFLOW.

To further clarify the motivation behind the research, it is essential to address the specific challenges that this framework aims to tackle. The IQUAFLOW framework was developed in response to the increasing demand for high-quality image data in various industries, particularly in the field of EO satellites. As the number of satellites and the volume of image data collected continues to grow, there is a need to optimize the transmission, storage, and quality of these images. EO satellites face challenges such as limited communication bandwidth and energy resources, as well as the need to compress images for efficient transmission. Therefore, the optimization problem lies in finding images with minimal file sizes that still provide sufficient quality to meet the performance requirements of deep learning algorithms. By focusing on image quality assessment and providing tools to evaluate the performance of AI models on these images, IQUAFLOW addresses these challenges and provides a solution for understanding the tradeoffs between image size, quality, and performance.

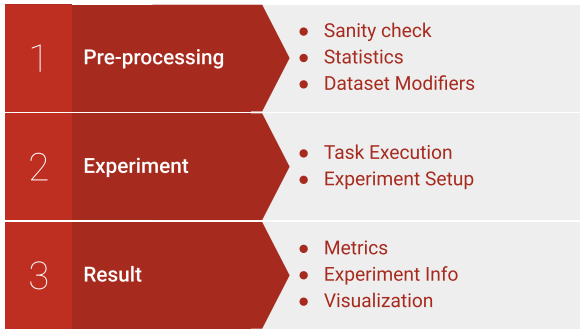

Fig. 1. Typical use case workflow using IQUAFLOW. The first step is to overview and sort the data. Then, some modifiers can be defined. Afterward, the task and the overall experiment is set and executed. Then, additional metrics can be estimated on the results. Finally, one can summarize, sort and visualize the results.

II. USE CASES

IQUAFLOW has already been used in various use cases to solve real problems that are faced in the EO industry. In this section, we summarize some of its use cases highlighting the benefits of using IQUAFLOW in them. Section II-A is actually an example of IQUAFLOW usage rather than a real case. Then, Sections II-B and II-C are describing two groups of use cases related with object detection and super-resolution (SR), respectively.

A. MNIST Showcase

The objective of this study[1] is to examine how the training performance of a deep learning classifier degrades as the amount of noise in the input dataset is increased. The dataset used is MNIST [8], which is widely used for benchmaking deep learning algorithms. It consists of a dataset of handwritten digits (0–9) images that is used to evaluate machine learning pattern recognition algorithms. Each digit image is 28×28 pixels.

The first step in the development process is to prepare the user training script that contains the deep learning classifier. In this case, the model is built using the architecture of a resnet18 [9]. Afterward the user script is adapted to follow the IQUAFLOW conventions. This means to add the appropriate input arguments (see Section III). The output that generates the user script also had to follow the standards of IQUAFLOW. Next step is to create a custom modifier that integrates with IQUAFLOW and does the desired alteration on the original dataset. In this case, the modifier is adding noise following a Gaussian distribution. The standard deviation of that noise distribution is an adjustable parameter. Finally IQUAFLOW executed all requested combinations: hyperparameter variations, number of repetitions, and dataset modifiers.

Despite being a classification task, the loss function was set as mean squared error (MSE) with the aim to penalize further the predicted digits that are more distant from the actual labeled

[1][Online]. Available: https://github.com/satellogic/iquaflow-mnist-use-case

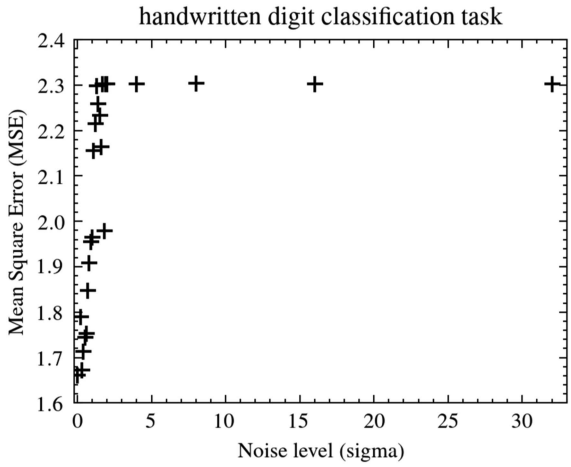

Fig. 2. MSE of the predicted digital label number with respect to the actual target number for different noise level dataset modifications. The noise level is defined as sigma of the Gaussian random noise distribution applied on them. Despite being a classification task the loss function was set as MSE with the aim to penalize further the predicted digits that are more distant from the actual labeled numbers.

Fig. 3. JPEG compression effects (original, JPG10,and JPG5 from left to right).

Fig. 4. Quantization effects (original, 5 and 2-b from left to right).

numbers. Fig. 2 illustrates the impact of varying noise levels on the performance of our model in the validation set. It specifically demonstrates how increasing amounts of noise, defined by sigma values in a Gaussian random noise distribution applied to the dataset, affect the MSE between predicted digital label numbers and actual target numbers. As depicted in Fig. 2, as we increase the noise level up to a sigma value of approximately 2, there is a noticeable degradation in model performance. However, beyond this point, further increases in noise do not appear to significantly worsen performance.

B. Object Detection on Compressed Satellite Images

As mentioned in the introduction, EO satellites capturing images have limited energy resources. Further to that they have limited connection time and data flow capacity between the orbiting equipment and ground station systems on Earth. The images are therefore compressed before downloading. There are compression algorithms that can be reverted recovering the original uncompressed image (lossless algorithms). On the other hand, there are lossy algorithms that will compress further with irreversible operations (such as interpolating the image to a smaller size with less pixels). When the usage of these images is well known the compression can be adjusted. This way, we avoid loss in performance on its intended application. This can be the case of detecting objects with a deep learning algorithm on satellite images.

We have built three use cases to study how the the application of different compression techniques affects object detection performance. In the first use case,[2] we sought to replicate the study from [5], where they study the performance of CNN-based object detectors on constrained devices by applying different image compression techniques to satellite data. In this case, they focus on execution times, memory consumption, and some insights about accuracy. In our case, we have focused on the performance metrics as a function of compression ratios. For this study, the public dataset DOTA [10] was used. The images were collected from different sensors with image sizes ranging from 800×800 to $20\,000 \times 20\,000$ pixels, while the pixel size varies from 0.3 to 2 m resolution. DOTA has different versions, in the present study DOTA-v1.0 has been used, which contains 15 common categories, 2806 images, and more than 188 k object instances. The proportions of the training, validation, and testing sets in DOTA-v1.0 are $1/2$, $1/6$, and $1/3$ [10]. A disadvantage of this dataset is that the test set is not openly available, rather it is in a form of a remote service to query the predictions. This does not allow to compress the test images the same way the other partitions are modified in the present study. To overcome this problem, we divided up the validation set: half of it is used as actual validation and the other half for testing. Then, the images are cropped to 1024×1024 with padding when necessary. After this operation, the amount of crops for the partitions train, validation, and testing are, respectively, 9734, 2670, and 2627.

Our second use case[3] is similar to the first one with newer implementations of object detection algorithms that are based on oriented annotations. The dataset used was the same as in the first study because the original annotations are oriented.

The final objective was to find an optimal compression ratio, which is defined as the minimum average file size that can be set without lowering the performance. We have found this to be around JPEG quality score of 70 (parameter CV_IMWRITE_JPEG_QUALITY defined in [11]) for both horizontal and oriented models. However, the oriented models had better performances. Figs. 3 and 4 are showing the visual effect of compression on the same image by using JPEG and quantization compression, respectively. It is important to note

[2][Online]. Available: https://github.com/satellogic/iquaflow-dota-use-case

[3][Online]. Available: https://github.com/satellogic/iquaflow-dota-obb-use-case

that human perception of image quality may not necessarily align with the experimental definition of image quality, which considers that these images are ultimately used for airplane detection tasks.

Finally, a use case is done with a prototype of airplane detection.[4] In this use case, we apply different compression algorithms on a new dataset. The objective of this experiment is to determine how much each of the compression algorithms affects the performance of an airplane detector. The airplanes dataset consists of Satellogic images capturing airport areas each of 1024 × 1024 pixels. These captures were made using NewSat Satellogic constellation (1 m GSD). The annotations were made using Happyrobot[5] platform.

C. Super-Resolution

SR refers to any methodology that allows the generation of an image with greater quality which is often represented with a greater amount of pixels. Two use cases were done in this topic: The first one was for single frame SR; the other for multi-frame SR. They are explained in this section.

The single image super-resolution (SISR) use case[6] is built to compare the image quality between different SISR solutions. A SISR algorithm inputs one frame and outputs an image with greater resolution. In this use case, we compare the following methods:

1) Fast super-resolution convolutional neural network (FSRCNN) [12].
2) Local implicit image function (LIIF) [13].
3) Multi-scale residual network (MSRN) [14].
4) Enhanced super-resolution generative adversarial network (ESRGAN) [15].

In this study, the public dataset UC Merced Land Use Dataset [16] was used for training, validating, and testing the models. Fig. 5 shows examples of super resolved images from the test partition using each of the methods. The first example on the left belongs to the high resolution ground truth image for comparison. These examples demonstrate that without objective quality metrics it is impossible to evaluate and compare the performance of the different methods. Because of that, it is wise to rely on objective image quality tools such as IQUAFLOW.

Figs. 6 and 7 shows quality metrics measured on the datasets solved by the different solutions of SISR. The first chart indicates that the best similarity scores with respect to the target image were achieved with the model LIFF. The second chart measures signal-to-noise ratio (SNR) and relative edge response (RER) which, again, the best values correspond to the LIIF model.

We faced the same challenge of measuring image quality with multi-frame super-resolution (MFSR) algorithms that are working similarly to the previous algorithms but with a short video (or multiple frames) as inputs rather than a single frame. The MFSR use case[7] is build to compare the image quality between different MFSR solutions. A MFSR algorithm inputs several frames of the same scene and outputs a single frame at a greater resolution. In this use case, we compare the following methods:

1) Adaptative Gaussian Kernels (AGK) [17].
2) WarpWeights (WARPW). This is our own method based in geometry and precise location of the multiple frames.
3) HighResNet (HRN) [18]. HRN is a deeplearning MFSR method. The public dataset xView [19] was used for the partitions of training, validation, and testing of this model.
4) Multi-scale residual network (MSRN) [14]. A single frame algorithm, added as a benchmark for comparison.

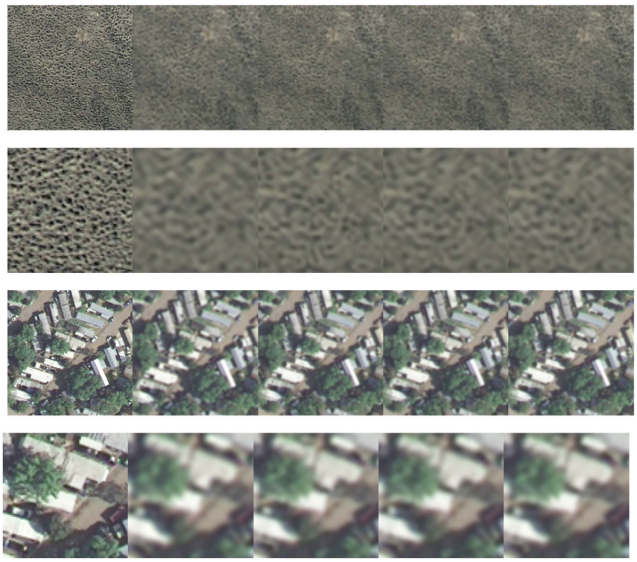

Fig. 5. Example of SR solutions. From left to right the images correspond to: The high resolution target image, FSRCNN, LIIF, MSRN, and ESRGAN. One can observe how difficult it is to compare quality of solutions that look very similar by human eye. That is one of the reasons of using quantitative methods to assess quality and compare solutions.

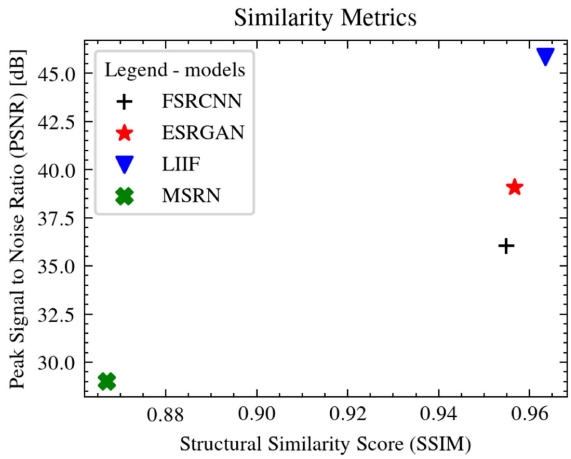

Fig. 6. Average structural similarity score (SSIM) and peak signal to noise ratio (PSNR). These are similarity scores with respect to the target high resolution image. Measured on different single image super resolution (SiSR) algorithm solutions.

[4][Online]. Available: https://github.com/satellogic/iquaflow-airplane-yolov5-use-case
[5][Online]. Available: https://happyrobot.ai
[6][Online]. Available: https://github.com/satellogic/iquaflow-sisr-use-case
[7][Online]. Available: https://github.com/satellogic/iquaflow-mfsr-use-case

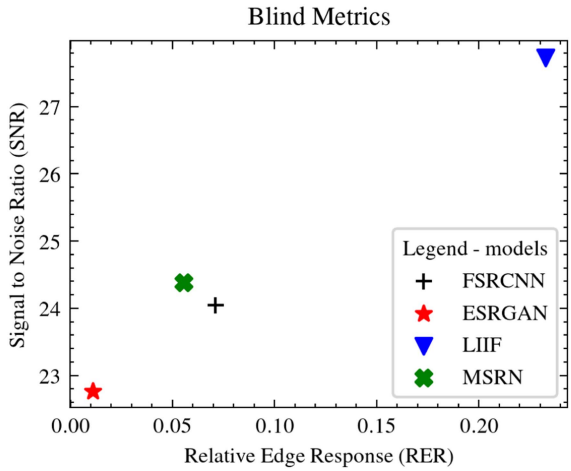

Fig. 7. Average SNR and RER. These are blind metrics that are intrinsic to the image. Metrics measured on different SISR algorithm solutions.

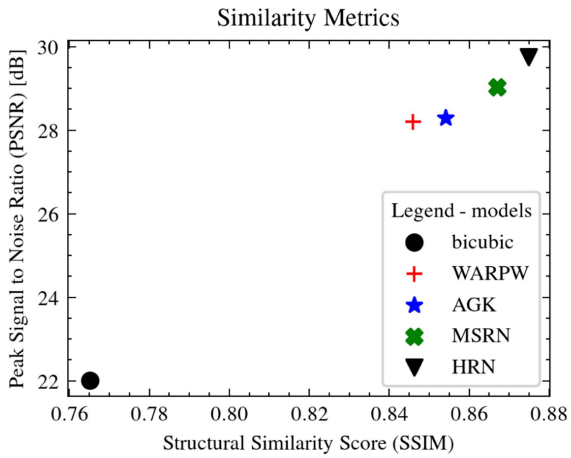

Fig. 8. Average Structural Similarity Score (SSIM) and PSNR. These are similarity scores with respect to the target high resolution image. Measured on different MFSR algorithm solutions.

5) Bi-cubic interpolation. This one is also added as a benchmark for comparison.

In this case, MSRN corresponds to a single frame algorithm and it is included as a benchmark reference of a single frame algorithm to compare with. Figs. 8 and 9 shows the quality metrics measured in the resulting images of the different methods. The similarity scores are indicating a clear improvement with respect to a simple bi-cubic interpolation. The best methods seem to be the ones based on deep learning (HRN and MSRN). Furthermore, Multiframe deep learning solution has greater score than MSRN. Despite the lower scores of methods AGK and WARPW they also are executed much faster and with less computational resources.

III. Conventions

Many tools and frameworks are designed by the philosophy of "convention over configuration" allowing the following small set of conventions to work in an easy manner with the tool. With

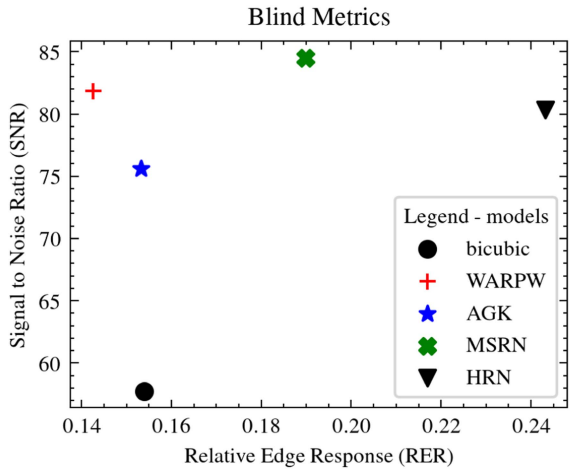

Fig. 9. Average SNR and RER. These are blind metrics that are intrinsic to the image. Measured on different MFSR algorithm solutions.

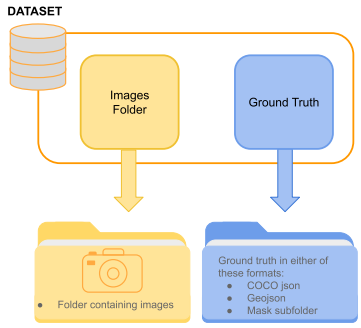

Fig. 10. Dataset structure in IQUAFLOW.

this approach IQUAFLOW can be adapted to any kind of deep learning model and custom training loop. Thus, we defined a set of conventions that the user must adopt in order to create a correct IQUAFLOW analysis.

A. Dataset Formats

IQUAFLOW understands a dataset as a folder containing a subfolder with images and ground truth (see Fig. 10). The name of the subfolder can be anything that does not contain the word "mask" which is reserved for mask annotations. Datasets that do not follow this format should be adapted in order to perform experiments. In case of detection or segmentation tasks any of these options are supported formats

1) Json in COCO [20] format.
2) GeoJson with the minimum required fields: image_filename, class_id, and geometry.
3) A folder named masks with images corresponding to the segmentation annotations.

IQUAFLOW primarily works with COCO [20] json ground truth adopted by most of the datasets and models of the field. In case that the dataset is in other format, the user can transform it to COCO [20]. Otherwise, IQUAFLOW neither can perform sanity nor statistics checks. For other kind of tasks, such as image generation, it is only necessary to have the ground truth in a json

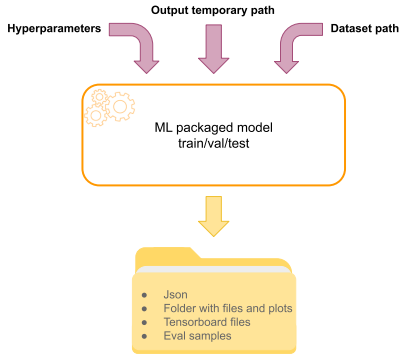

Fig. 11. Training script example structure.

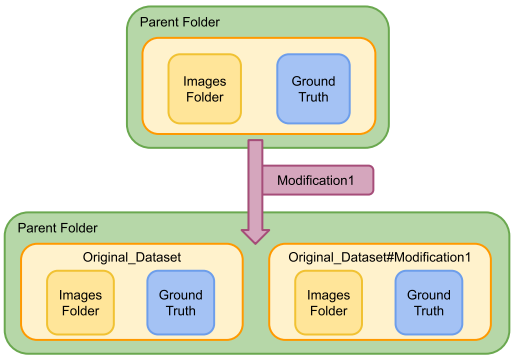

Fig. 12. Original and modified datasets.

format. Alternatively, IQUAFLOW can recognize a dataset without any ground truth file. When the dataset is modified, IQUAFLOW creates a modified copy of the dataset in its parent folder. As a convention, IQUAFLOW adds to the name of the original dataset a "#" followed by the name of the modification as seen in Fig. 12.

B. Training Script

The user is responsible to provide the model training scripts containing the custom training loop of the model. In this way, IQUAFLOW is agnostic to any kind of model or deep learning framework, it interacts with deep learning as a black box, as you can see in Fig. 11.

There are only a few input arguments that are necessary and allow the interaction of IQUAFLOW with the model. Thus, the training script requires the following arguments by convention:

1) *outputpath:* IQUAFLOW logs all the content of this path as artifacts in MLFLOW. Furthermore, it will log as metrics and parameters the files that are following the convention of the outputs as stated in Section III-C.
2) *trainds:* The training dataset path.
3) *valds (optional):* The validation dataset path.
4) *testds (optional):* The testing dataset path.
5) *mlfuri (optional):* The url pointing to the MLFLOW server.
6) *mlfexpid (optional):* The experiment id.
7) *mlfrunid (optional):* The run id.
8) *other hyperparameters (optional):* Any other custom parameter that belongs to the user training script. These parameters can be specified afterward in the ExperimentSetup of IQUAFLOW. They can also be varied by specifying a list with the different values.

Note that these arguments are fed to the user's training script by IQUAFLOW itself as the framework is responsible for combining the parameters into a set of runs, executing modifications of the dataset, and logging the details of the experiments in MLFLOW. Optional parameters are only added when they are also specified in the ExperimentSetup. It is the case of the arguments starting with `mlf`, which are only used when the flag mlflow_monitoring is activated in the ExperimentSetup. Their purpose is to monitor in streaming the training, without them the user will still get everything logged in MLFLOW at the end of each training.

C. Output Formats

Any of the data that is saved by the user training script in the output folder will be picked up by IQUAFLOW and be parsed as experiment parameters, metrics, or artifacts. These are the key files that can be saved by convention in the output folder:

1) *results.json:* JSON with the names of parameters and metrics for the keys. Then, the values are a single scalar for parameters and a list of values for metrics.

```
{
 "learning_rate": 0.83,
 "num_epochs": 100,
 "train_focal_loss": [1.34, 1.29, 1.24, 0.01]
 "val_focal_loss": [1.34, 1.29, 1.24, 0.01]
}
```

2) *output.json:* It is the predicted output in COCO json format [20]. It contains as many elements as detections have been made in the dataset. An object detection metric will rely on this file.

```
{
    "image_id" : 85
    "iscrowd" : 0
    "bbox":[
        522.5372924804688
        474.1499938964844
        28.968505859375
        27.19696044921875
    ]
    "area": 2427.050960971974
    "category_id": 1
    "id": 1
    score : 0.9709288477897644
}
```

3) *Image generation:* The json may contain the relative path to the generated images. Imagine the packaged model is SR model that generates five SR images. The package may store a folder named generated_sr_image in the output temporary file with this five images. Hence, the output.json should be as following:

```
{
 [
   "generated_sr_image/image_1.png",
   "generated_sr_image/image_2.png",
   "generated_sr_image/image_3.png",
   "generated_sr_image/image_4.png",
   "generated_sr_image/image_5.png",
 ]
}
```

IV. Preprocessing Stage

SanityCheck and DSStatistics are the classes that will perform sanity check and statistics of image datasets and ground truth. They are stand alone classes, it is to say they can work by proving the path folder of images and ground truth, or they can work with DSWrapper class.

A. Sanity Check

The SanityCheck module performs sanity to image datasets and ground truth. It can either work as standalone class or with DSWrapper class. It will remove all corrupted samples following the logic in the argument flags. The new sanitized dataset is located in output_path attribute from the SanityCheck instance. A usage example:

```
from iquaflow.sanity import SanityCheck

sc = SanityCheck(data_path, output_folder)
sc.check_annotations()
```

These are the tasks that are done when calling the check_annotations method:

1) Finding duplicates in COCO [20] json images list.
2) Check if the image format is a valid image file format.
3) Check integrity of one COCO [20] annotation.
4) Fix height and width in COCO [20] json images list.
5) In geojson annotations, remove all rows containing a Nan value, empty geometries in any of the required field columns.
6) In geojson annotations, try to fix geometries with buffer = 0 and remove the persistent invalid geometries.

Note the difference between missing, empty and invalid geometries in a geojson:

1) *Missing Geometries:* This is when the attribute geometry is empty or unknown. Most libraries load it as None type in python. These values were typically propagated in operations (for example in calculations of the area or of the intersection), or ignored in reductions, such as unary_union.
2) *Empty Geometries:* This happens when the coordinates are empty despite having a geometry type defined. This can happen as a result of an intersection between two polygons that have no overlap.
3) *Invalid Geometries:* Problematic features such as edges of a polygon intersecting themselves. This could have happened due to a mistake from the annotator. For the case of invalid geometry. The tool will also attempt to fix them with buffer=0 functionality prior to removing. In future releases an additional argument to simplify geometries will be offered.

B. Statistics and Exploration

The module DsStats calculates different statistics on the image datasets and their annotations. It allows the user to explore the data interactively, and to visualize the statistics, as well as to export them for later use. It can either work as standalone class or with DSWrapper class. A usage example:

```
from iquaflow.ds_stats import DsStats

dss = DsStats(data_path, output_folder)
stats = dss.perform_stats(show_plots = True)
```

Calling the perform_stats method calculates and reports the following statistics:

1) Average height and width of the images.
2) Histogram of the occurrences of the classes in the data.
3) Image and bounding box aspect ratio and area histograms.
4) Autogenerates the best fitting bounding box and rotated bounding box of the dataset annotations. It also adds the high, width angle from this new generated attributes.
5) Compactness, centroid, and area of the polygons.
6) Estimates the min, mean, and max from any given dataframe field from annotations that are in geojson format.

There are also two interactive exploratory tools. Some functionalities can be used in line in notebooks or exported as an interactive html. The first tool is for visualizing the annotations an the second tool shows a summary of some images. These are as follows:

1) notebook_annots_summary.
2) notebook_imgs_preview.

Usage example:

```
from iquaflow.ds_stats import DsStats

DsStats.notebook_annots_summary(
    df,
    export_html_filename=html_filename,
    fields_to_include=["image_filename", "class_id",
    ↪  "area"],
    show_inline=True,
)

from iquaflow.ds_stats import DsStats

DsStats.notebook_imgs_preview(
        data_path=data_path,
        sample=100,
        size=100,
)
```

C. Dataset

DSWrapper is the class that IQUAFLOW uses for handling datasets. Section III-A explains the conventions of a dataset in IQUAFLOW. To create an instance of DSWrapper one must specify the location path to the dataset as follows:

```
from iquaflow.datasets import DSWrapper
ds_wrapper =
↪  DSWrapper(data_path="[path_to_the_dataset]")
```

IQUAFLOW parses the dataset structure the following way:

1) *ds_wrapper.parent_folder:* Path of the folder containing the dataset.
2) *ds_wrapper.data_path:* Root path of the dataset.
3) *ds_wrapper.data_input:* Path of the folder that contains the images.
4) *ds_wrapper.json_annotations:* Path to the json annotations. Preferred COCO [20] annotations.
5) *ds_wrapper.geojson_annotations:* Path to the geojson annotations.

6) *ds_wrapper.params:* Contains metainfomation of the dataset. Initially "ds_name":"[name_of_the_dataset]"

Furthermore, DSWrapper contains an editable dictionary that describes the dataset. Initially, this dictionary contains the key ds_name that is the name of the dataset. The user can populate this dictionary with any key/value parameter. Afterward, this dictionary will be populated and changed automatically by DSModifier classes and it will be used for experiments logs.

D. Modifiers

Modifiers take a dataset D and process it to obtain a D' dataset with the modification defined by the user. There are built-in modifiers and the user can also create their own. When the dataset is modified, IQUAFLOW creates a modified copy of the dataset in its parent folder. As a convention, IQUAFLOW adds to the name of the original dataset a "#" followed by the name of the modification as you can see in the following image. To use a modifier one just needs to import the desired modifier and run it.

```
from iquaflow.datasets import DSModifier_jpg

img_path = "test_datasets/ds_coco_dataset/images")
jpg85 = DSModifier_jpg(params={"quality": 85})
jpg85.modify(data_input=img_path)
```

After running, a *test_datasets/ds_coco_dataset#jpg85_modifier/images/* folder should be created with the modified images.

Alternatively, any new designed custom modifier can be integrated in IQUAFLOW. To do so, one can follow the example in the code module *modifier_jpg.py*. The user needs to inherit from *DSModifier_dir* and write the *internal_mod_img()* member function.

V. Experiment

A. Task Execution

IQUAFLOW can automatize and log all the information that the user saves in the output files folder. This way, the user has the flexibility to log experiment information without knowing any specific logging tool. TaskExecution is the generic class that wraps the user packaged model and provides arguments to the training script. It is also responsible for translating all the experiment information to the MLFLOW tracking server. Ultimately, IQUAFLOW uses MLFLOW to organize the experiments and the user does not need to understand how MLFLOW works.

1) Pythonscripttaskexecution: This particular class extends from TaskExecution and knows how to execute a model that is encapsulated in a python script. The user needs to instantiate the class, adding the path to the python script as argument.

```
task = PythonScriptTaskExecution(
    model_script_path="./path_to_script.py"
)
```

Alternatively the user can execute the task, but is not recommended since IQUAFLOW will perform executions internally when the whole experiment is defined. In order to execute the run, the user must provide the experiment name, the name of the run and the training dataset path or training DSWrapper. Optionally, the user can provide a training dataset path or ds_wrapper and a python dictionary with model hyper_parameters (that will be used when executing the package)

```
task.train_val(
    experiment_name="name of the experiment",
    run_name="test_run",
    train_ds=ds_wrapper_train,
    val_ds=ds_wrapper_validation,
    hyper_parameters={"lr": 1e-6},
)
```

2) SagemakerTaskExecution: Our application can run on Amazon Sagemaker https://aws.amazon.com/sagemaker/ by passing a SageMakerEstimatorFactory as an argument of our TaskExecution. In which case, it becomes a SageMakerTaskExecution. See an example on how to define it.

```
from sagemaker.pytorch import PyTorch
from iquaflow.experiments.task_execution import (
    SageMakerEstimatorFactory,
    SageMakerTaskExecution
)

sage_estimator_factory = SageMakerEstimatorFactory(
   PyTorch,
   {
       "entry_point": "train.py",
       "source_dir": "yolov5",
       "role":role,
       "framework_version": "1.8.1",
       "py_version": "py3",
       "instance_count": 1,
       "instance_type": "ml.g4dn.xlarge"
   }
)

task = SageMakerTaskExecution( sage_estimator_factory )
```

Then, in the user training script, one might want to connect the argument script variables that are defined by convention in IQUAFLOW (see Conventions III) to SageMaker environmental variables to take full advantage of the SageMaker tools. As an example:

```
import argparse

parser = argparse.ArgumentParser()

# Define some defaults
trainds_default = (os.environ["SM_CHANNEL_TRAINDS"] if
↪  "SM_CHANNEL_TRAINDS" in os.environ else "")
valds_default = (os.environ["SM_CHANNEL_VALDS"] if
↪  "SM_CHANNEL_VALDS" in os.environ else "")
outputpath_default = (os.environ["SM_OUTPUT_DATA_DIR"]
↪  if "SM_OUTPUT_DATA_DIR" in os.environ else
↪  "./output")

# IQF arguments
parser.add_argument("--trainds",
↪  default=trainds_default, type=str, help="training
↪  dataset path")
parser.add_argument("--valds", default=valds_default,
↪  type=str, help="validation dataset path")
parser.add_argument("--outputpath",
↪  default=outputpath_default, type=str, help="path
↪  output")
```

Also, for these approaches one might want IQUAFLOW to upload the modifed datasets (by IQUAFLOW-modifiers) on a bucket on the fly. To do so, the user must indicate the bucket_name in the cloud_options whithin ExperimentSetup.

3) Experiment Setup: IQUAFLOW allows to formulate experiments taking as reference the modified training dataset. In

order to perform this task, the package provides tools that allow to automatize these kinds of experiments that are composed by:

1) A training dataset.
2) A list of dataset modifiers.
3) A machine learning Task.

The first two components are covered by DSWrapper and DSModifer, respectively. The last one requires a TaskExecution. Having defined all the components the user is able to perform a IQUAFLOW experiment by using ExperimentSetup. The user must define the name of the experiment, the reference datasets, the list of datasets modifiers, and the packaged model, as following:

```
experiment = ExperimentSetup(
   experiment_name="experimentA",
   task_instance=PythonScriptTaskExecution(
        model_script_path="./path_to_script.py"
    ),

   ↪  ref_dsw_train=DSWrapper(data_path="path_to_dataset"),
   ds_modifiers_list=[
        DSModifier_jpg(params={'quality': i})
        for i in [10,30,50,70,90]
    ]
)
```

And then just execute the experiment by:

```
experiment.execute()
```

Some other options for the ExperimentSetup are as follows:

1) *repetitions:* Each combination of parameters and modifiers results in a run. Scipts might contain randomness (i.e., Random partitions). For those cases, the user might want to average out several executions to have a relevant statistic or study the variability. To do so, one can set the number of repetitions to greater than 1.
2) *mlflow_monitoring:* This allows monitoring the training script in real time. When turned on, IQUAFLOW will pass these additional arguments to the training script (see Section III-B):
   a) mlfuri
   b) mlfexpid
   c) mlfrunid

   Thus, the user will be responsible to add these in the user training script when required. Then, the user can activate the current experiment and run in the the script with a snippet such as

```
mlflow.set_tracking_uri(args.mlfuri)

mlflow.start_run(
    run_id=args.mlfrunid,
    experiment_id=args.mlfexpid
)
```

3) *cloud options:* It is a dictionary of options useful for indicating endpoints such as
   a) *bucket_name:* If set, modified data (by IQUAFLOW-modifiers) will be uploaded to the bucket.
   b) *tracking_uri:* trackingURI for MLFLOW. default is local to the. /mlflow folder
   c) *registry_uri:* registryURI for MLFLOW. default is local to the. /mlflow folder

   Indicating the bucket is useful for SageMakerTaskExecution instances.

VI. Managing Results

A. Experiment Info

This class allows to manage the experiment information. It simplifies the access to MLFLOW and allows to apply new metrics to previous executed experiments. Basic usage example:

```
from iquaflow.experiments import ExperimentInfo

experiment_info = ExperimentInfo(experiment_name)
runs = experiment_info.get_mlflow_run_info() # runs is
↪  a python dict
```

These are the main methods:

1) *get_mlflow_run_info:* It gathers the experiment information in a python dictionary.
2) *apply_metric_per_run:* Applies a new metric to previously executed experiments.
3) *get_df:* Retrieves a selection of data in a suitable format so that it can be used as an input in the Visualization moduleVII.

In the following sections, we provide some examples of how to use the last two methods.

B. Metrics

The module metrics contains functionalities to estimate model performance metrics. IQUAFLOW includes some metrics and it also provides a generic Metric class that allows the user to easily implement their own custom metrics. To make a custom metrics one must inherit from the class Metrics as follows:

```
from iquaflow.metrics import Metric

class CustomMetric(Metric):
    def __init__(self) -> None:
        self.metric_names = coco_eval_metrics_names
    def apply(self, predictions: str, gt_path: str) ->
    ↪  Any:
        # Your custom code here
        # Then return a dictionary of names and values
        ↪  for each metric
        return {k: v for k, v in zip(metric_names,
        ↪  stats)}
```

Then to calculate a metric to an executed experiment do:

```
from iquaflow.experiments import ExperimentInfo

experiment_info = ExperimentInfo(experiment_name)
my_custom_metric = CustomMetric()
experiment_info.apply_metric_per_run( my_custom_metric,
↪  json_annotations_name )
```

Some relevant available metrics offered by IQUAFLOW are described in the following sections:

C. Bbdetectionmetrics

BBDetectionMetrics is a metric for object detection. It can be applied between bounding boxes of ground truth and predicted elements. The the ground-truth annotations must be in COCO format and the predictions in COCO-inference format [20] (see COCO detection and COCO data). When this metric is applied the metrics from COCOeval (see COCO detection) are estimated. This includes metrics such as (Recall, mAP, etc.)

```
from iquaflow.metrics import BBDetectionMetrics
```

D. SNRMetric

SNR is a metric that measures the strength/level of the considered signal relative to the background noise. Currently, there are two algorithms implemented in IQUAFLOW that measure SNR of an image: homogeneous blocks (HB; the default method) and homogeneous area (HA). The main difference between the two methods is that while HB uses the whole image, HA intends to find homogeneous areas in the image to calculate the SNR. This leads to a tradeoff: HB is faster than HA; HB returns a value every time while HA return a none value if it fails to find homogeneous areas; HB has higher uncertainty than HA. Both methods result in approximated values only, since a precise measurement requires a more careful consideration of the image content.

```
from iquaflow.metrics import (
    SNRMetric,
    snr_function_from_array,
    snr_function_from_fn
)
```

E. Sharpnessmetric

Image sharpness can be defined in several ways. The most common metrics include the RER, the FWHM value of the point spread function (PSF) of the image, or the modulation transfer function (MTF), which is often evaluated the Nyquist frequency [21]. The SharpnessMetric class implements all three metrics, and reports the values in three edge directions. The implemented algorithms are based on [22].

```
from iquaflow.metrics import SharpnessMetric
```

VII. Visualization

Apart from the visualization tools explained in the sanity check and statistics section, there are also tools for plotting and summarizing the results. One key service is MLFLOW that is accessed from the browser and allows to visualize, query, and compare runs. These are the following main features of MLFLOW:

1) Experiment-based run listing and comparison.
2) Searching for runs by parameter or metric value.
3) Visualizing run metrics.
4) Downloading run results.

Furthermore the ExperimentVisual class offers plotting utilities that can be either inline or saved into files. It uses a pandas data-frame [23] extracted from an ExperimentInfo as input.

VIII. Contributing

These are the tools, environment, and procedures required for developing and collaborating with this project.

A. Package Overview

The python package structure of this tool box is based on cookiecutter. This library provides a standard workflow for developing production level packages. The tools that will be used are:

1) *setuptools* for packaging.
2) *versioneer* for versioning.
3) *GitLab CI* for continuous integration.
4) *tox* for managing test environments.
5) *pytest* for tests.
6) *sphinx* for documentation.
7) *black*, *flake8*, and *isort* for style checks.
8) *mypy* for type checks.

More information can be found in:

1) python_packaging.
2) readthedocs.
3) Modules_and_Packages.

B. Environment Installation

This repository does not require any specific python environment. The file setup.py allows to install IQUAFLOW as a python package via pip. Once the new environment has been created, one must clone the repository. Then, the user can do the wallowing command to install the IQUAFLOW as a softlink in the environment. Dependencies are defined in setup.cfg under install_requires tag. So, first the package is installed in the local environment and then the dependency is added in the setup.cfg with its corresponding version.

C. Documentation

We use Sphinx to automatically update our documentation. This allows to update package documentation as new code is added. The documentation and Sphinx configuration can be found inside /doc.

D. Continuous Integration

In our project we use TOX. This tool allows to manage multiple environments in order to automatically validate code. More information about TOX can be found in here.

```
# For quality check:
tox -e check

# For automatic code reformat:
tox -e reformat

# For executing all test for first time use
tox -r -e py36

# Alternatively, if it is not the first time it is not
↪  necessary to recreate the tox environment.
tox -e py36
```

E. Testing

Unit tests are performed using PyTest. All tests are included in the test folder located in the repository main folder. Once a new test module that includes python assertions is made (e.g., test_new_module). Then, one must simply type in the console pytest or:

```
pytest <module name>
```

to run the tests.

We strongly recommend to use “test_” as the prefix of every test you create.

One can also run test manually using tox(recommended) (use -r parameter for creating tox environment for the first time):

```
tox -e py36
```

More information can be found in here.

F. Initial Development Process

Below, we describe usual steps when developing from scratch:

```
# Setup python environment:
conda create -n iqt-env python=3.6
# Clone repository:
git clone https://github.com/satellogic/iquaflow
#Create branch:
git checkout -b <new_branch_name>
# Install soft link via:
python -m pip install -e .
# Create test that defines modules functionality.
# Solve the test by adding package functionality.
# If new branch pulled use tox -r to recreate tox
↪  environments.
# Reformat code:
python -m pip install tox
tox -e reformat
# Check code and solve:
tox -e check
# Run tests:
tox -e py36
# Push to remote branch.
# Create MR and assign reviewer.
# Refreshing local repository for running tests (after
↪  pip install -e .):
tox -r -e py36Sphinx
```

IX. Conclusion

One of the major advantages of the new framework presented is that it allows custom metrics to be easily integrated, providing flexibility in measuring image quality. In addition, it enables the measurement of quality by using the performance of AI models trained on the images as a proxy. This feature is particularly useful for studying the performance degradation of images after different levels of lossy compression, such as satellite images. The framework also includes MLFLOW, an interactive tool used to visualize and summarize the results, making it easier for researchers to analyze and interpret the data. Overall, the framework offers a comprehensive and customizable approach to measuring image quality, making it a valuable tool for researchers in various fields.

Moving forward, there are numerous potential pathways for future research in this field. One promising direction could involve a more in-depth investigation of diverse metrics and optimization techniques within the IQUAFLOW framework. This exploration could potentially amplify image quality assessment across an array of applications.

Moreover, while the current study uses deep learning task performance as a proxy for image quality within the IQUAFLOW framework, it does not incorporate this performance into the loss function of a parent deep learning model. This presents an exciting opportunity for future research to innovate by integrating deep learning task performance into such a loss function. By doing so, it may be possible to further enhance image quality in deep learning algorithms.

Acknowledgement

Development: The project has been coordinated by SATELLOGIC and with the participation of the multimedia technologies unit of Eurecat along with the group on interactive coding of images (GICI) at Universitat Autònoma de Barcelona (UAB). It is a multidisciplinary team, which was absolutely necessary to successfully achieve the objectives set out in the project. SATELLOGIC, as project leader, assumed a decisive role in the management and coordination, as well as in the analysis of specifications and requirements. Its experience in the development of solutions applicable to the field of OE systems allowed it to lead the design and development of the IQUAFLOW Framework, acting as responsible for the technical coordination of the project. It also participated in the integration of the final prototype, validation tests, functional tests, and the integration and validation of the complete system both at laboratory level and at real environment validation level. UAB-DEIC-GICI took advantage of its experience in the field of data compression, in particular, in remote sensing image coding on board satellite to work in the design, development, and characterization of a compression system. They also played an active role in the definition and scoping of the project as well as the final prototype and validation tests for the code and configurations of the compression system software implementation. EURECAT performed the design and implementation of the image quality measurement algorithms based on deep learning. They also participated in the final prototype and validation tests for the code and configurations of the deep learning image quality module in IQUAFLOW.

References

[1] E. Buchen, "Spaceworks' 2014 nano/microsatellite market assessment," in *Proc. Conf. Small Satellites, Ser. 28th Annu. AIAA/USU*, 2014, p. 7.

[2] X. Liu, C. Deng, J. Chanussot, D. Hong, and B. Zhao, "STFNet: A two-stream convolutional neural network for spatiotemporal image fusion," *IEEE Trans. Geosci. Remote Sens.*, vol. 57, no. 9, pp. 6552–6564, Sep. 2019.

[3] M. Zhou, J. Huang, X. Fu, F. Zhao, and D. Hong, "Effective pan-sharpening by multiscale invertible neural network and heterogeneous task distilling," *IEEE Trans. Geosci. Remote Sens.*, vol. 60, 2022, Art. no. 5411614.

[4] M. Zhou, J. Huang, D. Hong, F. Zhao, C. Li, and J. Chanussot, "Rethinking pan-sharpening in closed-loop regularization," *IEEE Trans. Neural Netw. Learn. Syst.*, 2023, to be published, doi: 10.1109/TNNLS.2023.3279931.

[5] M. Lofqvist and J. Cano, "Optimizing data processing in space for object detection in satellite imagery," 2021. [Online]. Available: https://arxiv.org/abs/2107.03774

[6] Y.-Y. Jo et al., "Impact of image compression on deep learning-based mammogram classification," *Sci. Rep.*, vol. 11, 2021, Art. no. 7924.

[7] K. Delac, M. Grgic, and S. Grgic, "Effects of JPEG and JPEG2000 compression on face recognition," in *Pattern Recognition and Image Analysis*, S. M. Singh, C. Apte, and P. Perner, Eds. Berlin, Germany: Springer, 2005, pp. 136–145.

[8] L. Deng, "The MNIST database of handwritten digit images for machine learning research," *IEEE Signal Process. Mag.*, vol. 29, no. 6, pp. 141–142, Nov. 2012.

[9] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Las Vegas, NV, USA, 2016, pp. 770–778, doi: 10.1109/CVPR.2016.90.

[10] G.-S. Xia et al., "DOTA: A large-scale dataset for object detection in aerial images," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2018, pp. 3974–3983.

[11] G. Bradski, "The OpenCV library," *Proc. Dr Dobb's J. Softw. Tools Professional Programmer*, vol. 25, no. 11, pp. 120–123, 2000.

[12] C. Dong, C. C. Loy, and X. Tang, "Accelerating the super-resolution convolutional neural network," 2016. [Online]. Available: https://arxiv.org/abs/1608.00367

[13] Y. Chen, S. Liu, and X. Wang, "Learning continuous image representation with local implicit image function," in *IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Nashville, TN, USA, 2021, pp. 8624–8634, doi: 10.1109/CVPR46437.2021.00852

[14] X. Zhong, O. Gong, W. Huang, J. Yuan, B. Ma, and R. W. Liu, "Multi-scale residual network for image classification," in *Proc. IEEE Int. Conf. Acoust., Speech, Signal Process.*, 2020, pp. 2023–2027.

[15] X. Wang et al., "ESRGAN: Enhanced super-resolution generative adversarial networks," in *Proc. Comput. Vis. – ECCV2018 Workshops* (Lecture Notes in Computer Science), L. Leal-Taixé and S. Roth, Eds., 2019, vol. 11133, doi: 10.1007/978-3-030-11021-5_5.

[16] Y. Yang and S. Newsam, "Bag-of-visual-words and spatial extensions for land-use classification," in *Proc. ACM SIGSPATIAL Int. Conf. Adv. Geographic Inf. Syst.*, 2010, pp. 270–279.

[17] B. Wronski et al., "Handheld multi-frame super-resolution," *ACM Trans. Graph.*, vol. 38, no. 4, pp. 1–18, Aug. 2019, doi: 10.1145/2F3306346.3323024.

[18] M. Deudon et al., "Highres-Net: Recursive fusion for multi-frame super-resolution of satellite imagery," 2020. [Online]. Available: https://arxiv.org/abs/2002.06460

[19] D. Lam et al., "xview: Objects in context in overhead imagery," 2018. [Online]. Available: https://arxiv.org/abs/1802.07856

[20] T.-Y. Lin et al., "Microsoft coco: Common objects in context," in *Proc. 13th Eur. Conf. Comput. Vis.*, 2014, pp. 740–755.

[21] USGS, "Spatial resolution digital imagery guideline," 2020. [Online]. Available: https://www.usgs.gov/media/images/spatial-resolution-digital-imagery-guideline

[22] L. Cenci, V. Pampanoni, G. Laneve, C. Santella, and V. Boccia, "Presenting a semi-automatic, statistically-based approach to assess the sharpness level of optical images from natural targets via the edge method. case study: The Landsat 8oli–L1t data," *Remote Sens.*, vol. 13, no. 8, 2021, Art. no. 1593.

[23] W. McKinney, "Data structures for statistical computing in Python," in *Proc.9th Python Sci. Conf.*, S. van der Walt and J. Millman, Eds., 2010, vol. 445, pp. 56–61, doi: 10.25080/Majora-92bf1922-00a.