IDALib: a Python library for efficient image data augmentation

1st Nicolás Vila-Blanco
Centro de Investigación en Tecnoloxías Intelixentes (CiTIUS)
Universidade de Santiago de Compostela
Santiago de Compostela, Spain
nicolas.vila@usc.es

2nd Raquel R. Vilas
Centro de Investigación en Tecnoloxías Intelixentes (CiTIUS)
Universidade de Santiago de Compostela
Santiago de Compostela, Spain
raquelvilas18@hotmail.com

3rd María J. Carreira
Centro de Investigación en Tecnoloxías Intelixentes (CiTIUS)
Universidade de Santiago de Compostela
Santiago de Compostela, Spain
mariajose.carreira@usc.es

***Abstract*—The accuracy obtained with deep learning-based systems usually depends on the availability of large image datasets, which is not always possible. Consequently, it is necessary to apply techniques to increase the size of these datasets and their variability in a reliable and efficient way. In this respect, a novel tool for image augmentation and the associated data, IDALib, is presented. It provides an automatic method to perform transformation operations jointly on the images and related data, such as landmarks or masks, with the main aim of minimising the computational overhead. Thus, it applies automatically a set of optimisations, such as the vectorisation and composition of operations. Furthermore, the transformations are performed in GPU, which leads to a notable speedup, specially in dual GPU setups. IDALib is publicly available in the PyPI repository, https://pypi.org/project/ida-lib/**

***Index Terms*—Data augmentation, computer vision, deep learning**

I. Introduction

Artificial intelligence —and particularly deep learning— is leading a new sector of technological development that opens up a wide range of opportunities. Specifically, image processing with convolutional neural networks (CNN) is a very broad area with applications such as object detection in smart cars, facial recognition in social networks, or medical image analysis in the world of e-health [1]. However, these models are often very complex and require large-scale databases to be trained properly. When the available image sets are too small or not variable enough, CNNs are prone to overfitting the datasets and making generalisation to unseen data very hard to achieve [2].

This problem has led to the use of image augmentation techniques, which increase the size of available datasets by applying small transformations to the original images, such as translation, rotation, and changes in brightness and contrast, among many others. The new synthetic images are supposed to retain the global characteristics of the original images —such as the presence of objects or their relative position in the image—, but add variability such that the network can interpret them as different images and take advantage of it for improving the results [3].

The benefits of image augmentation are mainly a better generalisation capability, which leads to a better accuracy. For example, in the case of object detection applications, the network becomes more robust to the position and orientation of a certain object if that object appears in different positions in the training database, which can be achieved with translation and rotation operations [4].

It should be noted that input datasets for training neural networks do not always consist of individual images, but also of data associated with those images. For example, if the network is aimed at detecting certain characteristic points in an image, it should be fed with combinations of images and characteristic points to learn how to obtain the mapping between input images and output points. Therefore, the image data augmentation process should transform both the images and their metadata. As shown in Figure 1, the associated data can include segmentation maps —where each pixel has a discrete value—, heatmaps —where each pixel has a continuous value—, or keypoints —vector of coordinates—, among others.

It is widely agreed that training a neural network is an extremely expensive task at the processing level, as it can typically take days [5]. Thus, any additional tasks that are added to this process should be as efficient as possible to avoid a major overhead that is not completely essential. In this work, IDALib is presented as a useful tool for performing highly efficient image augmentation entirely on the GPU. It offers a wide variety of operations and the ability to perform joint transformations not only on images but also on masks, segmentation maps, heatmaps, and landmarks, which are commonly used in computer vision tasks. Moreover, it integrates seamlessly with the PyTorch deep learning framework.

978-1-6654-6623-3/22/$31.00

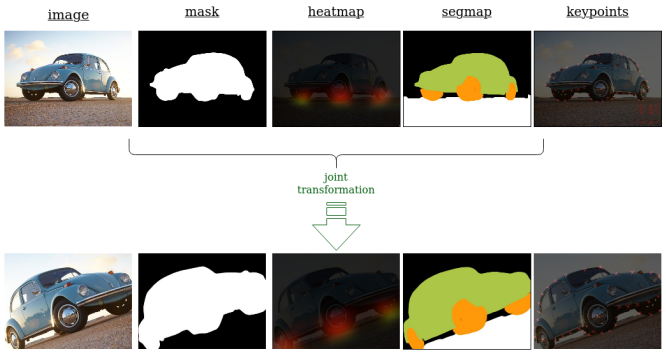


Fig. 1: Example of different types of metadata and their transformation.

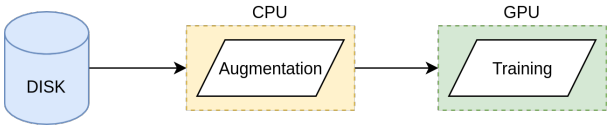
II. Related work

Currently, there are plenty of options for performing data augmentation on images. In Table I, the most widely used libraries are presented. Specifically, a set of features has been analysed: the image type used to read and process the images, the integration with the main deep learning frameworks, the support for Python generators, the ability to perform the transformations on CPU and/or GPU, and the use of third-party libraries to perform low-level operations.

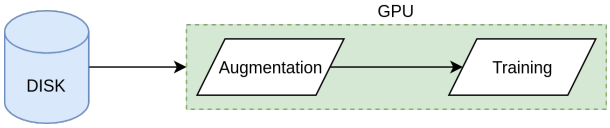
As shown in Table I, there is a great diversity in the properties of the libraries analysed. It is worth noting that both Scikit-image [6] and OpenCV [7] were not specifically designed for image augmentation, so they require that the users compose their own augmentation pipelines. The most commonly used data type is `ndarray` (n-dimensional numpy array), although `PIL image` and in some cases `torch tensor` are also used. Many libraries simply perform the operations of image augmentation without offering integration with any deep learning framework (ImgAug [9], image augmentor [11], scikit-image [6], or OpenCV [7]). Moreover, only four of the analysed libraries perform image augmentation through Python generators, which avoids traversing image batches twice (one for building them and the other for feeding them into the network).

Another relevant feature is the availability of performing image augmentation both in CPU and GPU. On the one hand, CPU operations are easier to debug and in some devices are the only option. In the case a GPU-based image augmentation is possible, geometric transformations benefit a lot from a GPU implementation due to the matrix multiplications involved. Of the libraries analysed in Table I, only three provide an easy way to perform CPU or GPU operations depending on the user requirements, namely torchsample [12], torchvision [13], and Kornia [14].

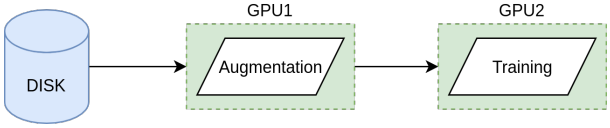
Another notable shortcoming is the performance of image data augmentation directly on disk. In some scenarios, it can be interesting to have this operation done beforehand (freeing the working memory) to be able to train the network efficiently (dedicating all the processing capacity to training). However,



(a) The batch of images is augmented via CPU operations, transferred to GPU and used to train the network.



(b) The batch of images is transferred to GPU, where image augmentation operations are applied. Then, the network is trained in the very same GPU.



(c) The batch of images is transferred to a first GPU, where image augmentation is applied. Then, the augmented batch is transferred into a second GPU, where the network is trained.

Fig. 2: Different configurations for network training.

only Augmentor [10], ImageAugmentor [11], torchvision [13], and NVIDIA DALI [15] offer this functionality.

III. Library description

The architectural design of IDALib is based on a set of key aspects: performance, flexibility, and usability.

A. Data flow and performance considerations

The performance of any image augmentation method is crucial to minimise the overhead in the training process. In addition to the efficiency of the image operations, the data flow between different pieces of hardware is also of particular relevance. Traditionally, once a batch of images and metadata is read from the disk to RAM, the image augmentation is performed directly on the CPU and then the augmented data are transferred to the GPU to run the batch training process (as shown in Figure 2a). Most of the time, this leads to a noticeable CPU overhead, causing the GPU to be idle waiting for the next batch of augmented images.

In some cases, where the training GPU does not work at full capacity, the augmentation process can be moved to this GPU to accelerate image transformations, as shown in Figure 2b. If the training process uses a high percentage of GPU and as long as a minimum of two GPUs are available, the augmentation can be performed on a different GPU than that used for training (Figure 2c).

In this regard, the design of IDALib, like the other libraries relying on torch tensors (torchsample, torchvision and Kornia), allows the user to run an augmentation-training process through any of these three augmentation-training pipelines, depending on the hardware capabilities and the training requirements.

TABLE I: Main tools for image augmentation.

Library	Image Type	DL-oriented	Python Generator	CPU	GPU	Augment to disk	3rd party libraries
Scikit-image [6]	ndarray	✗	✗	✓	✗	✗	✗
OpenCV [7]	ndarray	✗	✗	✓	✓	✗	✗
Albumetations [8]	ndarray	Keras, Pytorch	✗	✓	✗	✗	OpenCV
ImgAug [9]	ndarray	✗	✗	✓	✗	✗	Scikit-image, OpenCV
Augmentor [10]	PIL object	✗	✓	✓	✗	✓	PIL
ImageAugmentor [11]	ndarray	✗	✗	✓	✗	✓	Scikit-image
torchsample [12]	torch tensor	Pytorch	✓	✓	✓	✗	Pytorch
torchvision [13]	PIL Image, torch tensor	Pytorch	✓	✓	✓	✓	Pytorch
Kornia [14]	torch tensor	Pytorch	✗	✓	✓	✗	Pytorch
NVIDIA DALI [15]	ndarray	Pytorch, Tensorflow, MXNet	✓	✗	✓	✓	✗
IDALib	torch tensor	Pytorch	✓	✓	✓	✓	Pytorch, Kornia, OpenCV

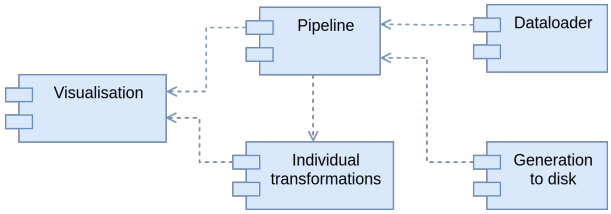


Fig. 3: IDALib structure in modules.

B. Architecture and features

The software structure of IDALib is highly modular, with the aim of easily allowing future extensions and modifications. Specifically, five modules have been developed (see Figure 3): *Dataloader*, *Generation to disk*, *Individual transformations*, *Pipeline*, and *Visualisation*.

The *DataLoader* module provides the functionality to easily interact with the PyTorch framework. In this regard, it allows for encapsulating the dataset containing the images and associated data, as well as the pipeline defined by the user, into a PyTorch data loader which can be used directly in the framework to train a neural network. Also, the *Generation to disk* module allows the user to apply a set of operations to the input dataset and save the results directly to disk to be used later. The user can set not only the operations to be applied, but also the number of output samples generated from each input sample or the total number of samples to be generated.

The *Individual transformations* module implements all the augmentation operations, which work on five different data types: images, both greyscale and colour; masks, which contain binary values; segmentation maps, which contain discrete values; heatmaps, which usually denote pixel probabilities; and point coordinates, which give the position of relevant landmarks in the input image. The wide variety of operations are organised into two main groups, as shown in Figure 4:

The *Pipeline* module has been developed to easily manage all the augmentation operations required by the user and optimise the augmentation process through the composition of transformations. Its behaviour is presented in Figure 5. The pipeline receives as an input a batch of items, each denoted by a Python dictionary whose elements are the input image

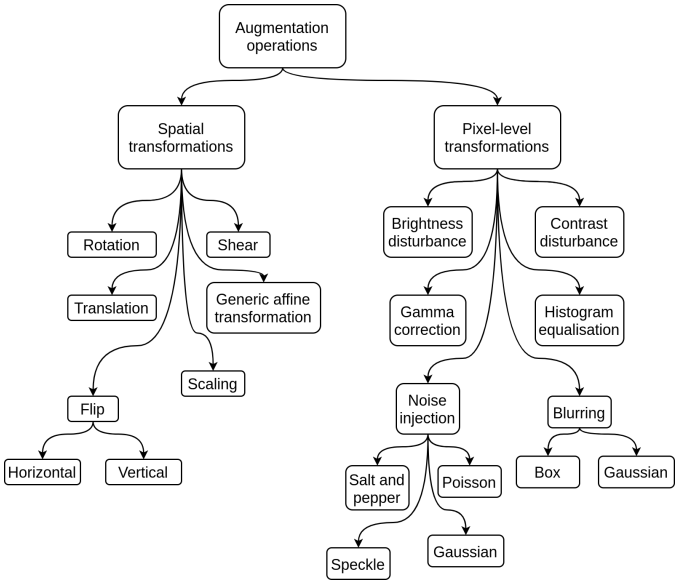


Fig. 4: Operations implemented in IDALib grouped into spatial and pixel-level transformation.

and its associated data (masks, heatmaps, segmentation maps and keypoints). The steps followed by the pipeline for each batch item are the following: 1) all the operations included by the user are split into three groups (geometric, pixel-level, and independent), and those that are going to be applied depending on their probability are collected; 2) a look-up table (LUT) with the composition of the pixel-level transformations is calculated; 3) the LUT is applied to the image in the batch item, if any; 4) the independent operations —those that do not admit composition, such as deblurring or histogram equalisation— are applied, if any; 5) the input item is preprocessed by concatenating the input image and heatmaps into one tensor and the masks and segmentation maps into another tensor; 6) the geometric operations are combined into one single transformation matrix; 7) The transformation matrix is applied to both generated tensors through an affine transformation and to the input keypoints through a matrix multiplication; 8) the transformed data are postprocessed by rencovering the exact

input format.

Furthermore, a *Visualisation module* has been developed to debug the transformations individually. As seen in Fig. 6, the tool allows the observation of how the transformation affects not only to the image, but also to the related data. The different items can be checked or unchecked as desired. Additionally, an individual tab is provided for each image in the batch.

IV. Implementation considerations

As summarised in Table I, there are many options for performing highly optimised low-level operations on images. Some examples shown in this table are OpenCV, which relies on efficient C++ instructions, or Scikit-image, based on a C implementation. In the case of IDALib, two different low-level libraries were chosen depending on the type of transformations. Particularly, Kornia library was chosen for geometric transformations for two reasons: first, it can perform the operations on GPU with a high efficiency and, second, it works directly with Pytorch tensors, which eases the integration with that platform. In the case of pixel-level transformations, it is crucial to have the option to apply LUTs to easily compose multiple transformations. As Kornia does not provide that functionality, OpenCV was chosen for this type of operation.

It is also worth noting that IDALib takes care of the type of input when performing augmentation transformations. For example, geometric transformations can lead to intermediate pixel values in binary masks and segmentation maps, which is not desirable. Thus, IDALib automatically applies a nearest-neighbour interpolation in these cases to preserve the same domain of the input.

An example of usage of the *Pipeline* module is provided in Listing 1. In this way, the user can define a pipeline with the desired operations, each with its specific parameters and the probability of applying it. To transform a set of images and associated data, represented in the variable `batch`, a simple call to the newly created pipeline object is required.

It should also be mentioned that IDALib offers both a functional and an object-oriented interface to apply the available operations individually.

Listing 1: Usage of the *PIPELINE* module in a Python script

```
pipeline = 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)
               )
)

transformed_batch = pipeline(batch)
```

V. Performance evaluation

To assess the performance of IDALib in different scenarios and compare it with other image augmentation libraries, it was benchmarked. The experimental setup was the following: a single computer equipped with an Intel Core i7-8760H CPU and an NVIDIA GeForce GTX1050 GPU was used. All the experiments were performed using the *Pipeline* module. As the implemented operations could have different computational costs, each experiment took a random combination of n operations. The experiments were always repeated 10 times (with different operations each time), and the final result was averaged across all executions.

First, the impact of the number of operations applied was assessed. As shown in Table II, the overhead caused by including additional operations in the composition is negligible. The difference of applying one operation or ten is less than 0.15 milliseconds, which represents about a 3% extra time. This is a notable improvement with respect to the time required if the same operations are applied sequentially, as it would present a linear growth behaviour.

TABLE II: Impact of the number of operations on the execution time (in milliseconds).

#Operations	Composition time	Sequential time
1	4.603	4.603
2	4.576	9.207
3	4.600	13.810
4	4.602	18.413
5	4.600	23.016
6	4.608	27.620
7	4.618	32.223
8	4.652	36.826
9	4.651	41.429
10	4.674	46.033

Another aspect to evaluate is how the library scales according to the number of metadata provided with the input image. Table III shows the time variation between transforming data composed of a single mask and transforming data with up to 10 masks. It can be seen that scaling capabilities with number of metadata presents a notable optimisation. Although it is not as impressive as in the previous case, it is still a positive measure considering that the transformation of a mask is practically the same as the transformation of an image, and these operations are performed by low-level libraries that are so optimised that further performance improvements are difficult to achieve. Measures on masks can be representative of other metadata, such as heatmaps or segmentation maps.

With respect to point-type metadata, its treatment is different from masks, heatmaps or segmentation masks. For this reason, it is necessary to evaluate the behaviour of the system according to the number of points to be transformed. Table IV shows the time taken by the pipeline to transform an element with one point, and the time to transform elements with almost 1,000 points. In this case, no reference to sequential time is included, since the transformation of point coordinates accompanied by an image is measured (without direct reference to the time

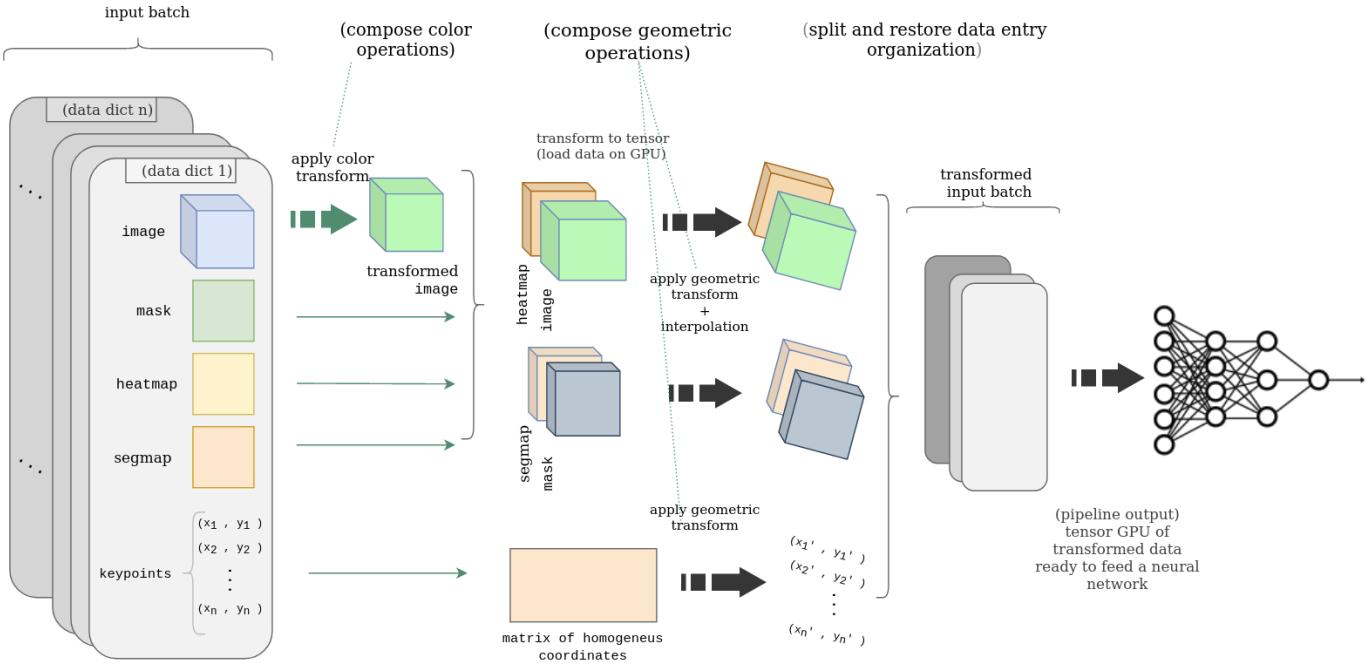


Fig. 5: Pipeline workflow. The process starts with a batch of items, each one containing an image and its associated data. First, the colour transformations are applied to the input images. Second, the continuous matrix inputs (images and heatmaps), the discrete matrix inputs (masks and segmentation maps), and the points (array of 2-D coordinates) are concatenated into three input groups. Third, the geometric operations are applied to each group and the output arrays are split to preserve the structure of the input data. Finally, the resulting output batch can be used to train a neural network.

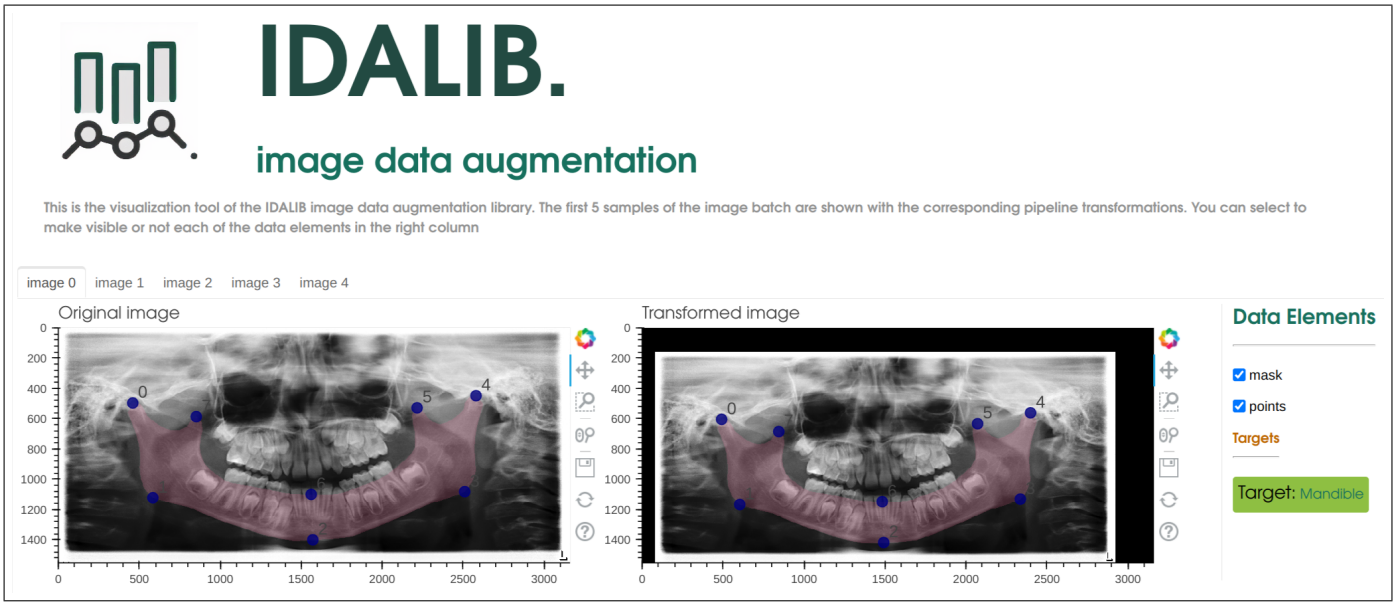


Fig. 6: IDALib visualisation tool. In the example, a dental radiograph with an associated mandible mask and a set of mandible landmarks is transformed through a combination of scaling and translation operations.

TABLE III: Impact of the number of masks when performing five operations on the execution time (in milliseconds).

#Masks	Composition time	Sequential time
1	4.831	4.831
2	7.452	9.662
3	6.763	14.493
4	8.694	19.324
5	11.625	24.155
6	14.986	28.986
7	19.027	33.817
8	23.228	38.648
9	27.489	43.479
10	32.140	48.314

required to transform a point). As seen in Table IV, the scaling is again optimal, demonstrating that IDALib takes almost the same time to transform 1 point or 1,000 points.

TABLE IV: Impact on the number of points transformed in the execution time (in milliseconds).

#Points	Time
1	4.752
120	4.677
240	4.718
480	4.669
960	4.678

Finally, in order to have a reference of the improvement produced by the composition of operations in IDALib, we compared its time performance in the transformation with composition of operations with respect to the low-level libraries.

Table V and Figure 7 show the transformation times of IDALib compared to a subset of the libraries showed in Section II, specifically those working at low-level: Kornia, OpenCV (both CPU and GPU implementations), and NVIDIA DALI. To perform these measurements, a transformation consisting of five different operations is defined. In the case of IDALib, this is done by means of the *Pipeline* module (which performs the composition of operations). For Kornia and OpenCV, these five operations are performed sequentially (they do not provide operation composition). For NVIDIA DALI, operation composition is implemented by using its own pipeline.

The composition of operations makes IDALib one of the best options. Although NVIDIA DALI performed better for larger image sizes, IDALIb still offers other functionalities that this library do not have, as the ease of use in both CPU or GPU environments. The time difference compared to the other libraries is a guarantee that IDALib is an efficient and valuable tool.

VI. Discussion and Conclusions

In this paper, IDALib, a library for image augmentation focused on the efficiency and flexibility, is presented. It allows the user to leverage the GPU capabilities to accelerate image augmentation, and offers an automatic operation composition

TABLE V: Execution time (in milliseconds) corresponding to an image transformation with five operations, using different low level libraries and different image resolutions.

	IDALib	Kornia GPU	OpenCV CPU	OpenCV GPU	NVIDIA DALI
50x50	**1.598**	5.293	15.228	16.750	1.776
100x100	**1.618**	5.391	15.940	18.331	1.855
250x250	**1.758**	6.803	27.063	21.650	1.869
500x500	2.716	12.460	48.960	36.720	**1.981**
750x750	4.679	22.186	57.521	50.909	**2.800**
1000x1000	7.594	35.731	64.415	58.658	**4.700**

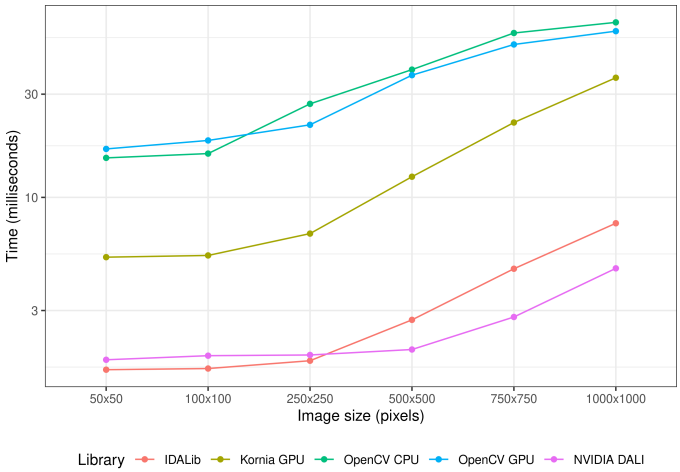


Fig. 7: Graphic comparison of the time consumption in the evaluated libraries (Table V).

to apply multiple operations in one step. At the same time, IDALib provides an easy-to-use interface and a seamless integration with the deep learning framework Pytorch.

As shown in the previous section, IDALib outperforms other commonly used libraries in terms of image size scalability. The only library that gave better performance in this regard was NVIDIA DALI for images larger than 250x250 pixels, because of the high efficiency of low-level GPU-based transformations. However, it should be noted that most widely used CNN architectures take images that are not larger than 224x224 pixels to achieve good efficiency without losing too much information [16], [17], [18]. Furthermore, IDALib has advantages over NVIDIA DALI, such as the possibility to run on both the CPU and GPU, the ease of use, or the better integration with the Pytorch framework.

Regarding the IDALib development roadmap, we considered three main points of action. First, we will add support for bounding boxes, as they are required in many applications such as object detection. Second, it is important to improve the scalability of the operations when the images are larger than 250x250 pixels. Although not a common requirement, some specific applications, such as small object detection, need the to keep the objects in the image as big as possible and thus the input image size can not be reduced to a large extent. Second, the number of operations will be further increased by adding

random erasing, hue correction, atmospheric events, or elastic transformations, among others.

In conclusion, IDALib demonstrated good performance and user-friendliness as a library for developing augmentation pipelines jointly on images and associated data. The library is hosted in the PyPI repository at https://pypi.org/project/ida-lib/. Furthermore, the documentation for installing and using IDALib is provided in https://ida-lib.readthedocs.io/

REFERENCES

[1] J. Gu, Z. Wang, J. Kuen, L. Ma, A. Shahroudy, B. Shuai, T. Liu, X. Wang, G. Wang, J. Cai *et al.*, "Recent advances in convolutional neural networks," *Pattern Recognition*, vol. 77, pp. 354–377, 2018.

[2] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *The journal of machine learning research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[3] L. Taylor and G. Nitschke, "Improving deep learning with generic data augmentation," in *2018 IEEE Symposium Series on Computational Intelligence (SSCI)*. IEEE, 2018, pp. 1542–1547.

[4] B. Zoph, E. D. Cubuk, G. Ghiasi, T.-Y. Lin, J. Shlens, and Q. V. Le, "Learning data augmentation strategies for object detection," in *European conference on computer vision*. Springer, 2020, pp. 566–583.

[5] G. Huang, Y. Sun, Z. Liu, D. Sedra, and K. Q. Weinberger, "Deep networks with stochastic depth," in *European conference on computer vision*. Springer, 2016, pp. 646–661.

[6] S. van der Walt, J. L. Schönberger, J. Nunez-Iglesias, F. Boulogne, J. D. Warner, N. Yager, E. Gouillart, T. Yu, and the scikit-image contributors, "scikit-image: image processing in Python," *PeerJ*, vol. 2, p. e453, 6 2014. [Online]. Available: https://doi.org/10.7717/peerj.453

[7] G. Bradski, "The OpenCV Library," *Dr. Dobb's Journal of Software Tools*, 2000.

[8] A. Buslaev, V. I. Iglovikov, E. Khvedchenya, A. Parinov, M. Druzhinin, and A. A. Kalinin, "Albumentations: fast and flexible image augmentations," *Information*, vol. 11, no. 2, p. 125, 2020.

[9] A. B. Jung, K. Wada, J. Crall, S. Tanaka, J. Graving, C. Reinders, S. Yadav, J. Banerjee, G. Vecsei, A. Kraft, Z. Rui, J. Borovec, C. Vallentin, S. Zhydenko, K. Pfeiffer, B. Cook, I. Fernández, F.-M. De Rainville, C.-H. Weng, A. Ayala-Acevedo, R. Meudec, M. Laporte *et al.*, "imgaug," https://github.com/aleju/imgaug, 2020, online; accessed 12-Jul-2022.

[10] M. D. Bloice, "Augmentor," https://github.com/mdbloice/Augmentor, 2016, online; accessed 12-Jul-2022.

[11] M. Mendoza, "Augmentor," https://github.com/michaelmendoza/image_augmentor, 2018, online; accessed 12-Jul-2022.

[12] N. Cullen, "TorchSample," https://github.com/ncullen93/torchsample, 2017, online; accessed 12-Jul-2022.

[13] Facebook, "torchvision," https://pytorch.org/vision/stable/index.html, 2017, online; accessed 12-Jul-2022.

[14] E. Riba, D. Mishkin, D. Ponsa, E. Rublee, and G. Bradski, "Kornia: an open source differentiable computer vision library for pytorch," in *Winter Conference on Applications of Computer Vision*, 2020.

[15] NVIDIA, "DALI," https://developer.nvidia.com/dali, 2020, online; accessed 12-Jul-2022.

[16] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, 2012.

[17] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[18] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.