




Deep Learning-Based Computer Vision Application with Multiple Built-In Data Science-Oriented Capabilities

Sorin Liviu Jurj^(✉) , Flavius Opritoiu, and Mircea Vladutiu

Advanced Computing Systems and Architectures (ACSA) Laboratory,
Computers and Information Technology Department, “Politehnica”,
University of Timisoara, Timisoara, Romania
jurjsorinliviu@yahoo.de,
{flavius.opritoiu,mircea.vladutiu}@cs.upt.ro

Abstract. This paper presents a Data Science-oriented application for image classification tasks that is able to automatically: a) gather images needed for training Deep Learning (DL) models with a built-in search engine crawler; b) remove duplicate images; c) sort images using built-in pre-trained DL models or user’s own trained DL model; d) apply data augmentation; e) train a DL classification model; f) evaluate the performance of a DL model and system by using an accuracy calculator as well as the Accuracy Per Consumption (APC), Accuracy Per Energy Cost (APEC), Time to closest APC (TTCAPC) and Time to closest APEC (TTCAPEC) metrics calculators. Experimental results show that the proposed Computer Vision application has several unique features and advantages, proving to be efficient regarding execution time and much easier to use when compared to similar applications.

Keywords: Deep learning · Computer vision · Data collection

1 Introduction

Data is at the core of every DL application. Because the Machine Learning lifecycle consists of four stages such as data management, model learning, model verification and model deployment [1], in order to collect, analyze, interpret and make use of this data, e.g. training accurate models for real-life scenarios, in recent years, new specializations were introduced in Universities around the world such as Machine Learning and Data Science, to name only a few. Additionally, also new career positions were created recently such as Machine Learning Engineer and Data Scientist, being some of the top paid positions in the industry [2].

Regarding Computer Vision applications for image classification tasks, a major bottleneck before training the necessary DL models is considered to be the data collection which consists mainly of data acquisition, data labeling and improvement of the existing data in order to train very accurate DL models [3]. Another bottleneck is that, because the amount of data needed to train a DL model is usually required to be very large in size and because most of this important data is not released to the general

public but is instead proprietary, the need of an original dataset for a particular DL project can be very crucial. In general, data can be acquired either by a) buying it from marketplaces or companies such as Quandl [4] and URSA [5]; b) searching it for free on platforms like Kaggle [6]; c) crawling it from internet resources with the help of search engine crawlers [7]; d) paying to a 24×7 workforce on Amazon Mechanical Turk [8] like the creators of the ImageNet dataset did to have all of their images labeled [9]; e) creating it manually for free (e.g. when the user takes all the photos and labels them himself), which can be impossible most of the time because of a low-budget, a low-quality camera or time constraints. The importance of image deduplication can be seen in the fields of Computer Vision and DL where a high number of duplicates can create biases in the evaluation of a DL model, such as in the case of CIFAR-10 and CIFAR-100 datasets [10]. It is recommended that before training a DL classification model, one should always check and make sure that there are no duplicate images found in the dataset. Finding duplicate images manually can be very hard for a human user and a time-consuming process, this being the reason why a software solution to execute such a task is crucial. Some of the drawbacks of existent solutions are that they usually require the user to buy the image deduplication software or pay monthly for a cloud solution, they are big in size or are hard to install and use.

Despite all of these options, especially in the case of scraping the images from the internet, once stored they can still be unorganized or of a lower quality than expected, with images needed to be sorted out each in their respective class folder in order for the user (e.g. data scientist) to be able later to analyze and use this data for training a performant DL model. This kind of sorting task can take a tremendous amount of time even for a team, from several days or weeks to even months [11]. Another difficult task is that once the data is cleaned, organized and ready to be trained from scratch or using transfer learning, because of the variety of DL architectures, each with different sizes and training time needed until reaching convergence [12], it can be very difficult to know from the beginning which DL architecture fits the best a given dataset and will, at the end of the training, result in a DL model that has high accuracy. Because energy consumption in DL started to become a very debated aspect in recent months, especially regarding climate change [13–17], the necessity of evaluating the performance of DL models also by their energy consumption and cost is very crucial.

Considering these aspects, our work introduces a DL-based Computer Vision application that has multiple unique built-in Data Science-oriented capabilities which give the user the ability to train a DL image classification model without any programming skills. It also automatically searches for images on the internet, sort these images each in their individual class folder and is able to remove duplicate images as well as to apply data augmentation in a very intuitive and user-friendly way. Additionally, it gives the user an option to evaluate the performance of a DL model and hardware platform not only by considering its accuracy but also its power consumption and cost by using the environmentally-friendly metrics APC, APEC, TTCAPC and TTCAPEC [16].

The paper is organized as follows. In Sect. 2 we present the related work. Section 3 describes the proposed DL-based Computer Vision application. Section 4 presents the experimental setup and results. Finally, Sect. 5 concludes this paper.

2 Related Work

Considering the advancements of DL in recent years, there is a growing interest in computer vision applications in the literature, such as regarding the automatic sorting of images, as shown by the authors in [18]. The authors propose a solution called ImageX for sorting large amounts of unorganized images found in one or multiple folders with the help of a dynamic image graph and which successfully groups together these images based on their visual similarity. They also created many similar applications, e.g. ImageSorter [19], which besides sorting images based on their color similarity, is also able to search, download and sort images from the internet with a built-in Google Image Search option. A drawback of their applications is that the user is able to only visualize similar images, without also having these images automatically cleaned and sorted in their respective class folder with high accuracy. Also, the authors in [20] created an application called Sharkzor that combines user interaction with DL in order to sort large amounts of images that are similar. By comparison, regarding sorting, their solutions only sort images by grouping them based on how similar they are to each other after a human interacted and sorted these images initially, whereas our application sorts them automatically by using in-built pre-trained DL models or gives the user an option to use his own trained DL models. An on-device option that uses DL capabilities and helps users find similar photos (e.g. finding photos that contain certain objects such as flowers, trees, food, to name only a few) is presented also by Apple in their newest version of Photos app [21].

Regarding the detection of duplicate images, this technique has practical applications in many domains such as social media analysis, web-scale retrieval as well as digital image forensics [22, 23], with several works in the literature applying it for the detection of copyright infringements [24] and fraud detection [25]. Recently, a python package that makes use of hashing algorithms and Convolution Neural Networks (CNNs) that finds exact or near-duplicates in an image collection called Image Deduplicator (Imagededup) was released in [26]. In our Computer Vision application, we make use of this package in order to offer a user the option to remove duplicate images from the images dataset (e.g. right before training a DL model).

When training DL models from scratch or by using transfer learning, usually frameworks such as Tensorflow and PyTorch are used [27], either locally (e.g. on a personal laptop or Desktop PC that contains a powerful GPU) or in cloud services such as Cloud AutoML [28, 29], Amazon AWS [30] or Microsoft Azure [31], with the work in [32] even assessing the feasibility and usefulness of automated DL in medical imaging classification, where physicians with no programming experience can still complete such tasks successfully. The problem when training locally is that the user still has to research on his own which size the images should have for a given DL architecture, which DL architecture to choose for his dataset and if it is necessary to apply fine-tuning and image augmentation. Regarding using the cloud services for training a DL model, even though these may solve most of the problems mentioned above, they still have some drawbacks such as that they can be affected by latency, can be difficult to manage (not user-friendly) and most importantly, they can be very

expensive when training for several hours (e.g. Cloud AutoML from Google costs around \$20 per hour when used for Computer Vision tasks [27]).

Similar work to ours is presented by the authors in [33] where they created the Image ATM (Automated Tagging Machine) tool that automatizes the pipeline of training an image classification model (preprocessing, training with model tweaking, evaluation, and deployment). Regarding preprocessing, the Image ATM tool just resizes the images to fit the model input shape. For training, it uses transfer learning with pre-trained CNNs from Keras by firstly training the last Dense layer followed by the whole network. For evaluation, it calculates the confusion matrix and other metrics. A few disadvantages of Image ATM: the tool is aimed at people with programming knowledge (developers) and is focused mainly on the training function. Also, in order to use the Image ATM tool, the user must take the work of preparing the data in a specific folder structure, e.g. the user must create a .yml file with some of the parameters desired, path to images and destination path. The user must also create a .json file containing the classification of each image. Some advantages of the Image ATM are that the tool offers the possibility for cloud training, has access to more models (although all are trained with the same dataset) and that the evaluation errors can be visualized. When compared to Image ATM, our Computer Vision application has several advantages such as that it is accessible to more kinds of people and offers more functionalities such as image web scraping and sorting, deduplication, calculators for accuracy as well as for the APC, APEC, TTCAPC and TTCAPEC metrics, all in a user-friendly Graphical User Interface (GUI).

3 The Proposed Deep Learning-Based Computer Vision Application

The proposed DL-based Computer Vision application is summarized in Fig. 1 and is built using the Python programming language. It is composed of the most common features needed in the Computer Vision field and facilitate them in the form of a GUI, without requiring the user to have any knowledge about coding or DL in order to be able to fully use it.

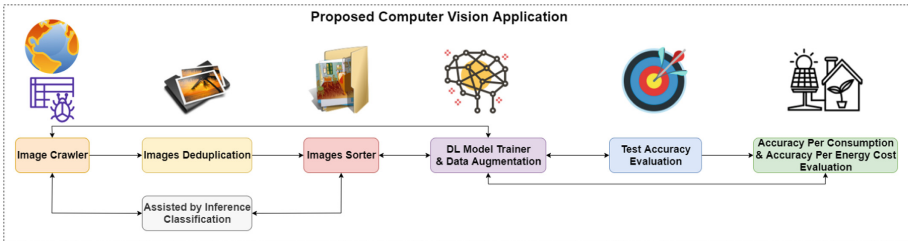


Fig. 1. Summarized view of the proposed Computer Vision application that incorporates features such as an automatic Image Crawler and Image Sorter assisted by inference classification, an Image Deduplicator, a DL Model Trainer with Data Augmentation capabilities as well as calculators regarding Accuracy, APC, APEC, TTCAPC, and TTCAPEC.

Regarding the system, the compilation dependencies and installation requirements of the proposed application are Python 3, Windows 10 (or later version) or Linux (Ubuntu 12 or later version). Regarding the Python libraries, we use PyQt5 for creating the GUI, HDF5 for loading DL model files, Tensorflow for training and inference, OpenCV for image processing, Numpy for data processing, Shutil for copying images in the system, TQDM for showing the terminal progress bar, Imagededup [26] for deduplication of images, Icrawler for crawling the images and fman build system (fbs) for creating installers.

There are certain conventions that are common in all the features of the proposed application:

1. Model files: These are .h5 files that contain the architecture of a Keras model and the weights of its parameters. These are used to load (and save) a previously trained model in order to be able to use it.
2. Model class files: These are extensionless files that contain the labels of each of the classes of a DL model. It contains n lines, where n is the number of classes in the model, and the line i contains the label corresponding to the i th element of the output of the DL model.
3. Preprocessing function: In this convention, a preprocessing function is a function that takes as input the path to an image and a shape, loads the image from the input path, converts the image to an array and fits it to the input of the model.

Images folders structures: We use two different folders structures: unclassified structures and classified structures. The unclassified images folders structure is the simplest one, consisting of just one folder containing images, presumably to be classified or deduplicated. The classified images folders structure consists of a folder which in turn contains subfolders. Each subfolder represents a class of images, is named the same as the label for that class, and contains images tagged or classified belonging to that class.

Following, we will present all the built-in features: Automatic web crawler assisted by inference classification, Images deduplication, Images Sorter assisted by inference classification, DL Model Trainer with Data Augmentation capabilities, Accuracy calculator as well as the APC and APEC [16] calculators.

3.1 Image Crawler Assisted by Inference Classification

The purpose of this feature is to collect images related to a keyword (representing a class) from the web and by using a classification algorithm, to make sure that the images are indeed belonging to this class. During the inference process needed for cleaning the images, a preprocessing is happening in the background, which, depending on the pretrained or custom DL model that is chosen, will resize the images, making them have the correct input shape (e.g. $28 \times 28 \times 1$ for MNIST and $224 \times 224 \times 3$ for ImageNet) for the DL model.

A summarized view of the implemented Image Crawler feature can be seen in Fig. 2 and is composed of the following elements: ‘Model’ - a combo box containing all the existent pretrained in-built DL models such as “mnist” or “resnet50” as well as the ‘Custom’ option which gives the user the possibility to load his own previously trained

DL model; Confidence Slider ('Confidence required') - a slider to select the minimum accuracy value to be used when classifying the images and which ranges from 0 to 99; Image Class Selector ('Select a class of images') - a combo box containing the labels of all the classes from the pretrained built-in selected DL model (e.g. 10 classes for when the "mnist" model is selected and 1000 classes when the "resnet50" model is selected). Additionally, the box contains an autocomplete search function as well; Images Amount ('Max amount to get') - a slider to select the number of images that should be crawled from the internet and which ranges from 1 to 999 and 'Destination Folder' - a browser to select the path for the final location of the obtained images.

The options under 'Custom Model Configuration' only apply when the DL model selected is "Custom" and is not built-in in the proposed Computer Vision application, e.g. when it was trained by the user itself. These options are: 'Model File' - a browser to select the .h5 file the user wishes to use for inference and Model Classes - a browser to select the extensionless file containing the name of each output class on which the selected DL model (.h5 file) was trained on. Finally, this feature's GUI interface has a button ('Add Images!') that begins the web crawling process.

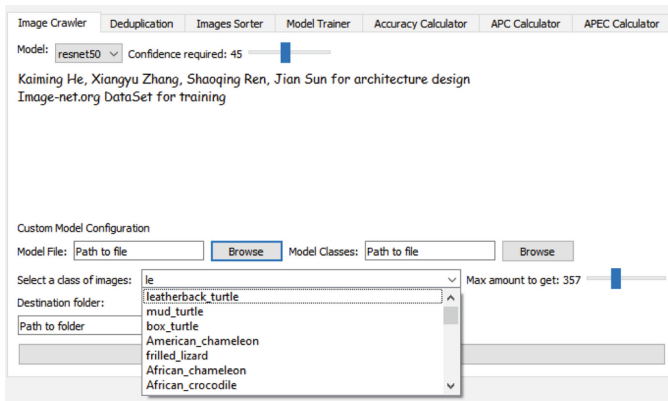


Fig. 2. Summarized view of the proposed Image Crawler feature assisted by inference classification.

With the help of this feature, images are automatically crawled by the crawler and downloaded to a temporal folder location. After that, each image is classified with the selected DL model, and if the classification coincides with the selected class and the confidence is higher than the selected threshold, the image is moved to the 'Destination folder', where each image will be saved in its own class folder.

This feature automatizes the population of image classification datasets by providing a reliable way of confirming that the downloaded images are clean and correctly organized.

3.2 Images Deduplication

The purpose of this feature is to remove duplicate images found in a certain folder. For this, we incorporated the Imagededup techniques found in [26]. A summarized view of the implemented Images Deduplication feature can be seen in Fig. 3 and is composed of the following elements: ‘Images folder’ - a browser to select the location of the folder containing the images that need to be analyzed for duplicate images; ‘Destination folder’ - a browser to select the location of the folder where the deduplicated images will be stored; ‘Duplicates Folder’ - a browser to select the location of the folder where the found duplicate images will be stored.

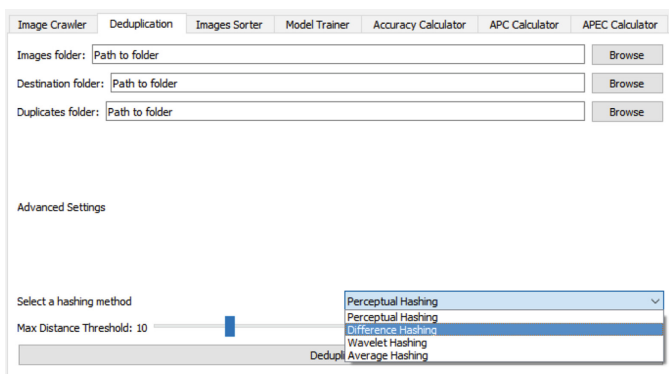


Fig. 3. Summarized view of the proposed Image Deduplication feature.

Each duplicate image found will be stored in a subfolder. Regarding advanced settings, it is composed of: Hashing method selector (‘Select a hashing method’) - a combo box containing 4 hashing methods that can be used for deduplication (Perceptual Hashing (default), Difference Hashing, Wavelet Hashing, and Average Hashing) as well as a ‘Max Distance Threshold’ - the maximum distance by which two images will be considered to be the same (default value is 10). Finally, this interface has a button (‘Deduplicate!’) that begins the deduplication process according to the selected parameters.

Following, we will shortly describe the types of hashes we are using in the images deduplication feature: a) **Average Hash:** the Average Hash algorithm first converts the input image to grayscale and then scales it down. In our case, as we want to generate a 64-bit hash, the image is scaled down. Next, the average of all gray values of the image is calculated and then the pixels are examined one by one from left to right. If the gray value is larger than the average, a 1 value is added to the hash, otherwise a 0 value; b) **Difference Hash:** Similar to the Average Hash algorithm, the Difference Hash algorithm initially generates a grayscale image from the input image. Here, from each row, the pixels are examined serially from left to right and compared to their neighbor to the right, resulting in a hash; c) **Perceptual Hash:** After gray scaling, it applies the discrete cosine transform to rows and as well as to columns. Next, we calculate the median of

the gray values in this image and generate, analogous to the Median Hash algorithm, a hash value from the image; d) **Wavelet Hash**: Analogous to the Average Hash algorithm, the Wavelet Hash algorithm also generates a gray value image. Next, a two-dimensional wavelet transform is applied to the image. In our case, we use the default wavelet function called the Haar Wavelet. Next, each pixel is compared to the median and the hash is calculated.

Regarding this deduplication feature, first, the hasher generates hashes for each of the images found in the images folder. With these hashes, the distances between hashes (images) are then calculated and if they are lower than the maximum distance threshold (e.g. 10), then they are considered duplicates. Secondly, for each group of duplicates, the first image is selected as “original” and a folder is created in the duplicates folder with the name of the “original” folder. Then all duplicates of this image are stored on that folder.

This feature successfully integrates the image deduplication technique [26] and provides a simple and quick way to utilize it.

3.3 Images Sorter Assisted by Inference Classification

This feature helps a user to sort an unsorted array of images by making use of DL models. A summarized view of the implemented Images Sorter feature assisted by inference classification can be seen in Fig. 4 and is composed of elements similar to the ones presented earlier for the Image Crawler feature, but in this case with the function of selecting the path to the folders from which and where images should be sorted.

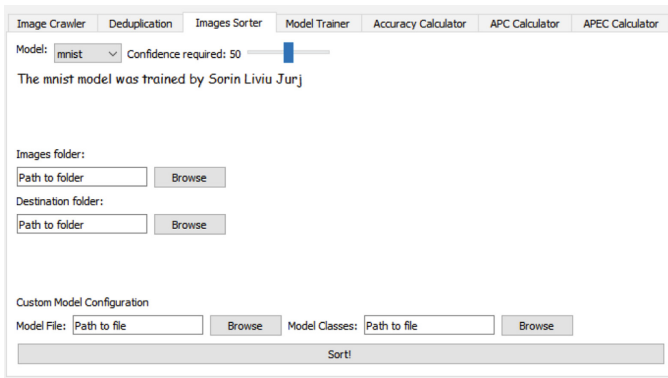


Fig. 4. Summarized view of the proposed Image Sorter feature assisted by inference classification.

In the destination folder, a new folder is created for each possible class, with the name extracted from the extensionless file that contains all the names of the classes, plus a folder named ‘Undetermined’. Then, each image from the ‘Images Folder’ is automatically preprocessed, feed as input to the selected DL model and saved in the corresponding class folder. The highest value from the output determines the predicted

class of the image: if this value is less than the minimum ‘Confidence required’, value, then the image will be copied and placed in the ‘Undetermined’ folder, otherwise, the image will be copied to the folder corresponding to the class of the highest value from the output. We took the decision of copying the files instead of moving them, for data security and backup reasons.

This feature heavily reduces the amount of time required to sort through an unclassified dataset of images by not only doing it automatically but also removing the need to set up coding environments or even write a single line of code.

3.4 Model Trainer with Data Augmentation Capabilities

This feature gives the user a simple GUI to select different parameters in order to train and save a DL image classifier model. A summarized view of the implemented DL Model Trainer feature assisted by inference classification can be seen in Fig. 5 and is composed of the following elements: ‘Model’ – as described earlier for the Image Crawler feature; ‘Sorted images folder’ - a browser to select the folder that contains the classified folder structure with the images to be trained on; ‘Number of training batches’ - an integer input, to specify the number of batches to train and ‘Size of batches’ - an integer input, to specify the number of images per batch. Regarding the custom options, they are the same as mentioned earlier regarding the Image Crawler feature.

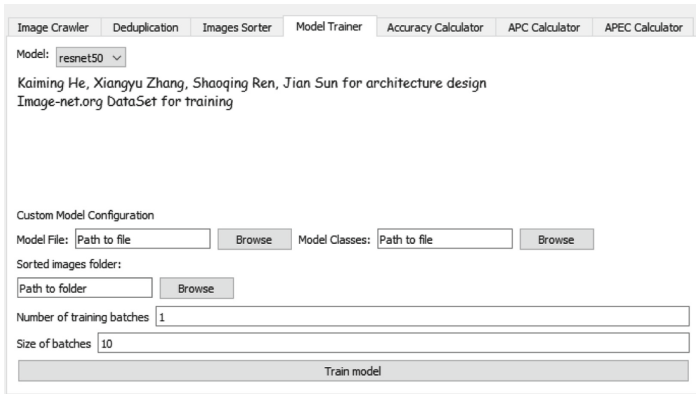


Fig. 5. Summarized view of the proposed DL Model Trainer feature.

Next, this interface has a button (‘Train model’) that, when clicked on, prompts a new window for the user to be able to visualize in a very user-friendly way all the image transformations that can be applied to the training dataset in a random way during training. More exactly, as can be seen in Fig. 6, the user can input the following parameters for data augmentation: Horizontal Flip - if checked the augmentation will randomly flip or not images horizontally; Vertical Flip - if checked the augmentation will randomly flip or not images horizontally; Max Width Shift - Slider (%), maximum percentage (value between 0 and 100) of the image width that it can be shifted left or

right; Max Height Shift - Slider (%), maximum percentage (value between 0 and 100) of the image height that it can be shifted up or down; Max Angle Shift - Slider (degrees $^{\circ}$), the maximum amount of degrees (value between 0 and 90) that an image might be rotated and Max Shear Shift - Slider (%), maximum shear value (value between 0 and 100) for image shearing. The data augmentation feature allows the user to visualize the maximum possible changes that can be made to an image in real-time, without the need of guessing the right parameters.

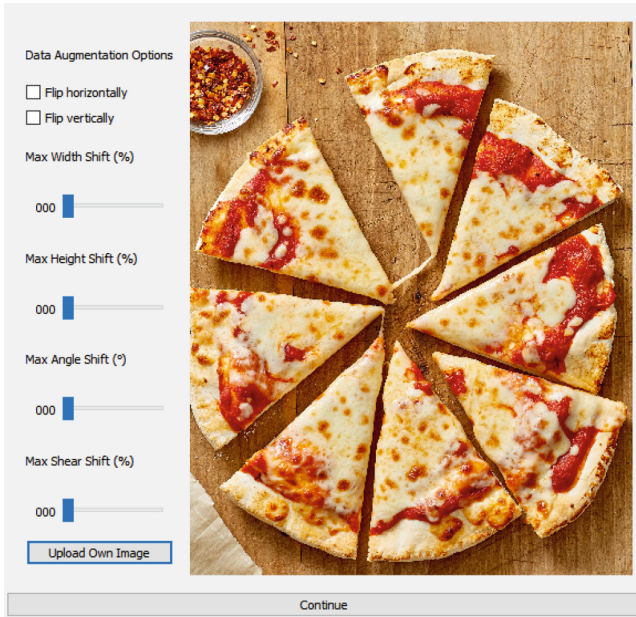


Fig. 6. Summarized view of the proposed Data Augmentation feature.

Following, a training generator is defined with the selected parameters; The generator randomly takes images from the folder structure and fills batches of the selected size, for the number of batches that are selected. These batches are yielded as they are being generated.

Regarding the training, first, the selected DL model is loaded, its output layer is removed, the previous layers are frozen and a new output layer with the size of the number of classes in the folder structure is added. The model is then compiled with the Adam optimizer and the categorical cross-entropy as the loss function. Finally, the generator is fed to the model to be fitted. Once the training is done, the total training time is shown to the user and a model file (.h5) is created on a prompted input location.

This feature achieves the possibility of training a custom DL model on custom classes just by separating images in different folders. There is no knowledge needed about DL and this feature can later also be easily used by the Image Sorting feature described earlier in order to sort future new unsorted images.

3.5 Accuracy Calculator

This section of the application GUI gives a user the option to compute the accuracy of a DL model on the given dataset in the classified images folder structure. A summarized view of the implemented Accuracy Calculator feature can be seen in Fig. 7 and is composed of the following elements: ‘Model’ - as described earlier for the Image Crawler feature; ‘Test images folder’ - a browser to select the folder that contains the classified folder structure to measure the accuracy of a DL classification model; ‘Size of batches’ - an integer input, to specify the number of images per batch. The custom options are the same as mentioned earlier regarding the Image Crawler feature. Finally, this interface has a button (‘Calculate Accuracy’) that starts the accuracy evaluation process.

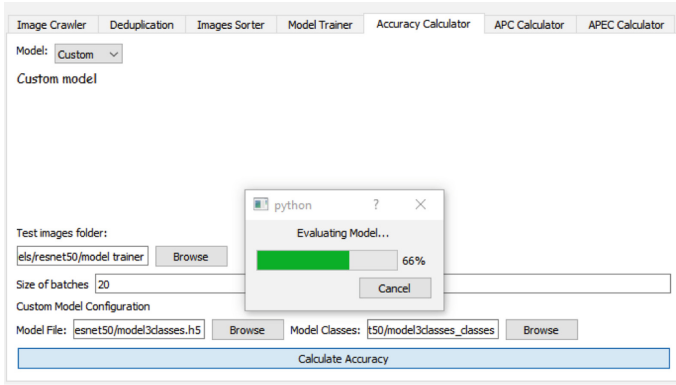


Fig. 7. Summarized view of the proposed Accuracy Calculator feature.

After loading the DL model and the list of classes, it searches for the classes as subfolders names in the classified images folder structure. Then, for each class (or subfolder) it creates batches of the selected batch size, feeds them to the DL model and counts the number of accurate results as well as the number of images. With these results, it calculates the total accuracy of the DL model and shows it to the user directly in the application GUI. This feature provides a simple and intuitive GUI to measure the accuracy of any DL image classification model.

3.6 Accuracy Per Consumption (APC) Calculator

This GUI feature makes use of our APC metric [16] and which is a function that takes into account not only the accuracy of a system (acc) but also the energy consumption of the system (c). The APC metric can be seen in Eq. (1) below:

$$APC_{\alpha,\beta}(c, acc) = \frac{acc}{\beta \cdot WC_{\alpha}(c, acc) + 1} \quad (1)$$

where c stands from energy consumption of the system and it's measured in Watt/hour (Wh) and acc stands for accuracy; α is the parameter for the WC_{α} function, the default

value is 0.1; β is a parameter (ranges from 0 to infinity) that controls the influence of the consumption in the final result: higher values will lower more heavily the value of the metric regarding the consumption. The default value is 1.

The application GUI gives a user the option to define the values for α and β as well as to specify and calculate the accuracy and energy consumption of a DL model using the above APC metric equation.

A summarized view of the implemented APC Calculator feature can be seen in Fig. 8 and is composed of the following elements: ‘Model test accuracy (%)’ - this widget gives a user the option to input the accuracy or use the previously described Accuracy Calculator feature to measure the accuracy of a DL model and ‘Energy Consumption (Wh)’ - float input to specify the power consumption of a user’s DL model.

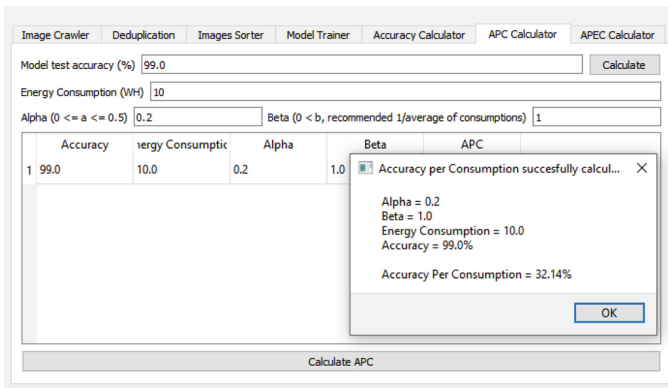


Fig. 8. Summarized view of the proposed APC Calculator feature.

Regarding the advanced options, it has: Alpha (α) - float input to specify the desired value of α (default 0.2) and Beta (β) - float input to specify the desired value of β (default 1). For simplicity, a table is shown with the following columns: Accuracy, Energy Consumption, Alpha, Beta, and APC. Whenever a value is changed, the table is automatically updated as well. Finally, the application GUI has a button (‘Calculate APC’) to begin the calculation of the APC metric. The function itself is a Numpy implementation of our previously defined APC metric [16] seen in Eq. (1) and takes as input parameters the values defined in the application GUI.

The implemented feature brings this new APC metric to any user by allowing them to easily calculate the accuracy per consumption and know the performance of their DL model with regards to not only the accuracy but also to the impact it has on the environment (higher energy consumption = higher negative impact on nature). However, the drawback of the current version of this APC calculator feature in the proposed application GUI is that the user has to measure the energy consumption of the system manually. We plan to implement automatic readings of the power consumption in future updates (e.g. by using the Standard Performance Evaluation Corporation (SPEC)

PTDaemon tool [34, 35], which is also planned to be used for power measurements by the MLPerf Benchmark in their upcoming mid-2020 update).

3.7 Accuracy Per Energy Cost (APEC) Calculator

This metric is a function that takes into account not only the accuracy of a system (acc) but also the energy cost of the system (c). The APEC metric can be seen in Eq. (2) below:

$$APEC_{\alpha,\beta}(c, acc) = \frac{acc}{\beta \cdot WC_{\alpha}(c, acc) + 1} \quad (2)$$

where c stands for the energy cost of the system and it's measured in EUR cents per inference and acc stands for accuracy.

α is the parameter for the WC_{α} function, the default value is 0.1; β is a parameter (ranges from 0 to infinity) that controls the influence of the cost in the final result: higher values will lower more heavily the value of the metric regarding the cost. The default value is 1.

The APEC feature is presented in Fig. 9 and lets a user define the values for α and β , specify or calculate the accuracy of a DL model, specify the energy consumption and the cost of Wh of the DL as well as calculate the APEC using the formula seen earlier in (2).

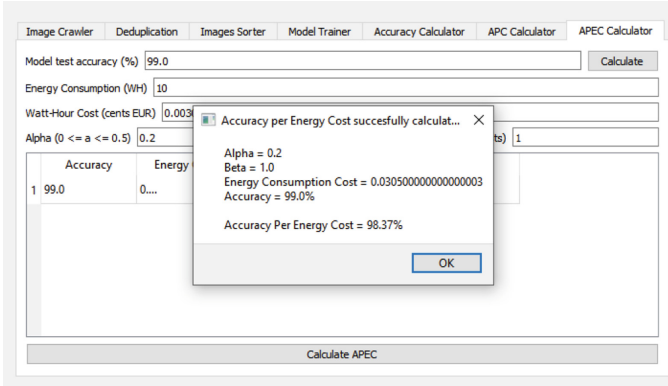


Fig. 9. Summarized view of the proposed APEC Calculator feature.

The APEC feature of the proposed Computer Vision application is composed of the following elements: ‘Model test accuracy (%)’ – works similar to the APC widget described earlier; ‘Energy Consumption (Wh)’ – works also similar to the APC widget described earlier and Watt-Hour Cost – float input to specify the cost in EUR cents of a Wh. Regarding the advanced options, we have: Alpha (α) – float input to specify the desired value of α (default 0.2) and Beta (β) – float input to specify the desired value of β (default 1). A similar table like the one for APC Calculator is shown also here, with the following columns: Accuracy, Energy Cost, Alpha, Beta, and APEC. Whenever a value is changed, the table is automatically updated here as well.

Finally, the application GUI has a button (‘Calculate APEC’) to begin the calculation of the APEC metric.

The function itself is an implementation on Numpy of our previously defined APEC metric [16] seen in Eq. (2) and takes as input parameters the values defined in the application GUI. The implemented feature brings this new APEC metric to any user by allowing them to easily calculate the accuracy per energy cost and evaluate the performance of their DL model with regards to the impact it has on the environment (higher energy consumption = higher cost = negative impact on nature). However, the drawback of the current version of this APEC calculator feature is that the user has to measure the energy consumption of the system and calculate its Wh cost manually.

3.8 Time to Closest APC (TTCAPC) Calculator

The objective of the TTAPC metric [16] is to combine training time and the APC inference metric in an intuitive way. The TTCAPC feature is presented in Fig. 10 and is composed of the following elements: ‘Model test accuracy (%)’ and ‘Energy Consumption (Wh)’, both working similar to the APEC widget described earlier; ‘Accuracy Delta’ – float input to specify the granularity of the accuracy axis; ‘Energy Delta’ – float to specify the granularity of the energy axis. Regarding the advanced options, they are the same as the ones presented earlier regarding the APEC feature.

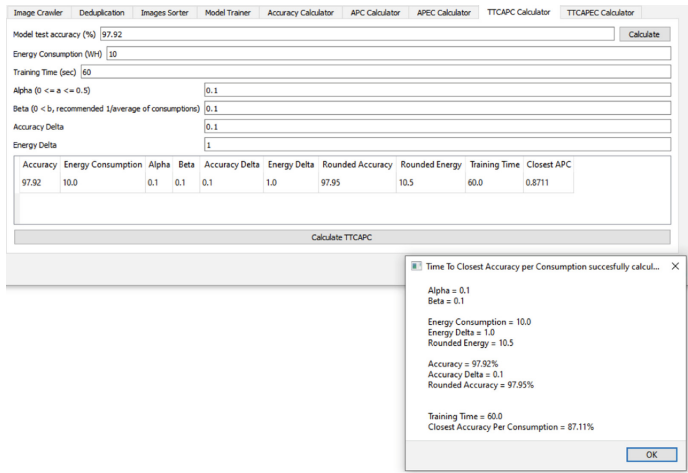


Fig. 10. Summarized view of the proposed TTCAPC Calculator feature.

A similar table like the one for APEC Calculator is shown also here, with the following columns: Accuracy, Energy Consumption, Alpha, Beta, Accuracy Delta, Energy Delta, Rounded Accuracy, Rounded Energy, Training Time and Closest APC. Whenever a value is changed, the table is automatically updated here as well.

Finally, the application GUI has a button (‘Calculate TTCAPC’) to begin the calculation of the TTCAPC metric.

3.9 Time to Closest APEC (TTCAPEC) Calculator

The objective of the TTCAPEC metric [16] is to combine training time and the APEC inference metric. The TTCAPEC feature is presented in Fig. 11 and is composed of the same elements like the TTCAPC feature presented earlier and one additional element called ‘Energy Cost (EUR cents per Wh)’ which is similar to the one presented earlier regarding the APEC metric calculator and where the user can specify the cost in EUR cents of a Wh.

The screenshot shows the TTCAPEC Calculator interface. It includes input fields for Model test accuracy (%), Energy Consumption (Wh), Energy Cost (EUR cents per Wh), Training Time (sec), Alpha (0 <= a <= 0.5), Beta (0 < b, recommended leverage of costs), Accuracy Delta, and Energy Delta. A 'Calculate' button is present. Below the inputs is a table with the following data:

Accuracy	Energy Cost	Alpha	Beta	Accuracy Delta	Energy Delta	Rounded Accuracy	Rounded Energy	Training Time	Closest APEC	Closest APEC Green Powered
97.92	0.1525	0.1	50.0	0.1	0.001	97.95	0.1525	60.0	0.5146	97.95

Below the table is a 'Calculate TTCAPEC' button. A dialog box titled 'Time To Closest Accuracy per Energy Cost successfully calculated...' is open, displaying the following values:

- Alpha = 0.1
- Beta = 50.0
- Energy Consumption = 50.0
- Energy Cost per Wh = 0.00305
- Energy Consumption Cost = 0.1525
- Energy Delta = 0.001
- Rounded Energy = 0.1525
- Accuracy = 97.92%
- Accuracy Delta = 0.1
- Rounded Accuracy = 97.95%
- Training Time = 60.0
- Closest Accuracy Per Energy Cost = 51.46%

Fig. 11. Summarized view of the proposed TTCAPEC Calculator feature.

A similar table like the one for TTCAPC Calculator is shown also here, with the following columns: Accuracy, Energy Cost, Alpha, Beta, Accuracy Delta, Energy Delta, Rounded Accuracy, Rounded Energy, Training Time and Closest APEC. Finally, the application GUI has a button (‘Calculate TTCAPEC’) to begin the calculation of the TTCAPEC metric.

4 Experimental Setup and Results

Following, we will show the experimental results regarding all the implemented features in comparison with existing alternatives found in the literature and industry.

We run our experiments on a Desktop PC with the following configuration: on the hardware side we use an Intel(R) Core(TM) i7-7800X CPU @ 3.50 GHz, 6 Core(s), 12 Logical Processor(s) with 32 GB RAM and an Nvidia GTX 1080 Ti as the GPU; on the software side we use Microsoft Windows 10 Pro as the operating system with CUDA 9.0, CuDNN 7.6.0 and Tensorflow 1.10.0 using the Keras 2.2.4 framework.

4.1 Image Crawler

As can be seen in Table 1, our proposed Image Crawler feature outperforms existent solutions and improves upon them. Even though the crawling took the same amount of

time, this is not the case regarding the cleaning part, where, because this feature is not available in any of the existent solutions, this needed to be done manually and took 47 s for a folder containing 97 images as compared to only 10 s for our proposed solution which executed the task automatically.

Table 1. Comparison between existent and the proposed Image Crawling solution.

Features	Existent solutions [7]	Proposed solution
Image crawler	Yes	Yes
Built-in DL models	No	Yes
Custom DL models	No	Yes
Cleans dataset automatically?	No	Yes
Speed Test (sec)		
Crawling 97 images	23	23
Cleaning 97 images	47	10

A comparison between “dirty” images and clean images can be seen in Fig. 12 where, for simplicity, we searched for 97 pictures of “cucumber”, which is one class from the total of 1000 classes found in the ImageNet dataset [9].

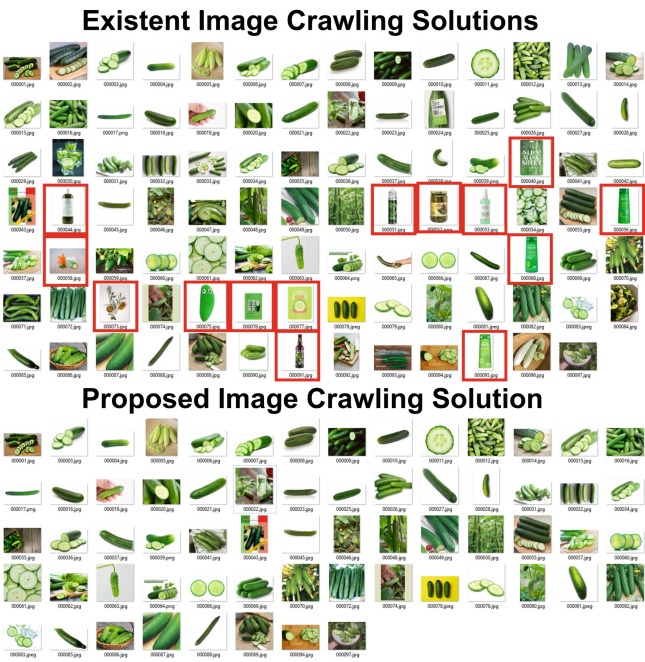


Fig. 12. Summarized view of existent and the proposed image crawling solution. The pictures marked with a red rectangle are examples of “dirty” images found when crawling with existent solutions. By comparison, the proposed image crawling feature assisted by DL inference contains only clean images.

It can be easily observed how the existent solutions provide images that don't represent an actual cucumber, but products (e.g. shampoos) that are made of it. After automatically cleaning these images with a confidence rate of 50% with the proposed feature, only 64 clean images remained in the folder.

4.2 Deduplication

For the experiments seen in Table 2, we tested the speed time of the proposed built-in image deduplication feature that uses the Imagededup python package [26]. We run these experiments on finding only exact duplicates on the same number of images with a maximum distance threshold of 10 for all four hashing methods.

Table 2. Speed Results for the 4 hashing methods of the proposed Image Deduplication feature.

Nr. of images	Hashing method	Speed time (sec)
1.226	Perceptual Hashing	16
	Difference Hashing	15
	Wavelet Hashing	17
	Average Hashing	16

As can be seen, the average speed is about 16 s for finding duplicates in a folder containing 1.226 images, with Difference Hashing being the fastest hashing method from all four.

4.3 Images Sorter

For our experiments regarding the sorting of images with the proposed images sorter feature, we used both the MNIST as well as the ImageNet pre-trained models with a confidence rate of 50% and presented the results in Table 3.

Table 3. Speed Time for the proposed Images Sorting feature.

DL model	Nr. of classes	Nr. of images	Undetermined images	Speed time (sec)
MNIST	10	70.000	69	307
ImageNet [9]	1000	456.567	135.789	40.817
Custom [37]	34	2.380	34	223

Regarding MNIST experiments, we converted the MNIST dataset consisting of 70.000 images of 28×28 pixels to PNG format by using the script in [36] and mixed all these images in a folder. After that, we run our image sorter feature on them and succeeded to have only 0.09% of undetermined images, with a total speed time of around 6 min. Regarding ImageNet, we used the ImageNet Large Scale Visual Recognition Challenge 2013 (ILSVRC2013) dataset containing 456.567 images belonging to 1000 classes with a confidence rate of 50%. Here we successfully sorted

all images in around 11 h and 20 min, more exactly in 40.817 s, with 29.74% (135.789) undetermined images.

Regarding the custom model, we used one of our previously trained DL models (ResNet-50) that can classify 34 animal classes [37] on a number of 2.380 images of $256 \times$ Ratio pixels (70 images for each of the 34 animal classes) with a confidence rate of 50%. Here we succeeded to have 1.42% undetermined images, with a total speed time of almost 4 min. The percentage of the undetermined images for all cases can be improved by modifying the confidence rate, but it is out of this paper’s scope to experiment with different confidence values.

The time that a DL prediction task takes depends on a few variables, mainly the processing power of the machine used to run the model, the framework used to call the inference of the model and the model itself. Since processing power keeps changing and varies greatly over different machines, and all the frameworks are optimized complexity wise and keep evolving, we find that among these three the most important to measure is, therefore, the model itself used in the prediction. Models vary greatly in their architecture, but all DL models can be mostly decomposed as a series of floating points operations (FLOPs). Because, generally, more FLOPs equal more processing needed and therefore more time spent in the whole operation, we measured the time complexity of the built-in ImageNet and MNIST models in FLOPS and presented the results in Table 4.

Table 4. The time complexity of the built-in DL models measured in the number of FLOPS.

DL model	Dataset	MFLOPS	GFLOPS
ResNet-50	ImageNet	3.800	3.8
MNIST	MNIST	9	0.009

4.4 Model Trainer

For the experiments regarding the DL model training feature, because we want to evaluate the application on a real-world problem, we will attempt to show that this feature could be very useful for doctors or medical professionals in the aid of detecting diseases from imaging data (e.g. respiratory diseases detection with x-ray images). In order to prove this, we will attempt to automatically sort between the images of sick patients versus healthy patients regarding, firstly, pneumonia [38], and secondly, COVID-19 [39], all within our application and doing it only with the training feature that the application provides.

For this, first, in order to classify between x-ray images of patients with pneumonia versus x-ray images of healthy patients, we made use of transfer learning and trained a ‘resnet50’ architecture for around 2 h without data augmentation on pneumonia [38] dataset containing 6.200 train images by selecting 10 as the value for the number of training batches and 10 as the value for the size of batches (amount of images per batch) and achieved 98.54% train accuracy after 10 epochs. Secondly, in order to classify between x-ray images of patients with COVID-19 versus x-ray images of negative patients, we again made use of transfer learning and trained a ‘resnet50’

architecture for around 1 h without data augmentation on the COVID-19 [39] dataset containing 107 train images by selecting the same values for the number and size of training batches as the pneumonia model mentioned above and achieved 100% train accuracy after 100 epochs.

4.5 Accuracy Calculator

For the experiments regarding the accuracy calculator feature, we used the two custom DL models trained earlier to classify x-ray images of patients with pneumonia versus x-ray images of healthy patients and between x-ray images of patients with COVID-19 versus x-ray images of negative patients, with 20 as the size of batches (20 images per batch).

The evaluation took in both cases around 50 s with a test accuracy of 93.75% regarding the pneumonia model on 620 test images and 91% regarding the COVID-19 model on 11 test images, proving that the proposed Computer Vision application can easily be used by any medical personal with very basic computer knowledge in order to train and test a DL classification model for medical work purposes.

4.6 APC and APEC Calculators

Regarding the experiments with the proposed APC [16] calculator feature, we presented the simulated results for different model test accuracy (%) and energy consumption (Wh) values in Table 5. We run all the experiments with 0.2 as the alpha value and with 1.0 as the beta value.

Table 5. Summarized Results of the proposed APC Calculator feature.

Energy consumption [Wh]	DL model test accuracy [%]	APC [%]
10	99.0	32.14
2	99.0	69.91
1	99.7	82.91
10	99.7	32.96
50	99.7	8.96
10	94.5	27.47
50	50.0	1.61
1	50.0	31.25
10	50.0	7.14
10	40.0	5.12
1	40.0	23.8
1	100	83.33

It is important to mention that our recommendation for a correct comparison between two DL models, is that it is always necessary that they are both tested with the same alpha and beta values. As can be seen in Table 5 where we experimented with

random energy consumption and test accuracy values, our APC Calculator feature is evaluating the performance of a DL model by considering not only the accuracy but also the power consumption. Therefore, DL models that consume around 50 Wh (e.g. when running inference on a laptop) instead of 10 Wh (e.g. when running inference on a low-cost embedded platform such as the Nvidia Jetson TX2) [15], are penalized more severely by the APC metric.

Regarding the experiments with the proposed APEC [16] calculator feature, we presented the simulated results for different model test accuracy (%) and energy cost in Table 6. We run all the experiments with 0.2 as the alpha value and with 1.0 as the beta value.

Table 6. Summarized Results of the proposed APEC Calculator feature.

Energy consumption [Wh]	Power cost [cents EUR]	DL model test accuracy [%]	APEC [%]	APEC green energy [%]
10	0.03050	99.0	98.37	99.0
2	0.0061	99.0	98.87	99.0
1	0.00305	99.7	99.63	99.7
10	0.03050	99.7	99.08	99.7
50	0.1525	99.7	96.71	99.7
10	0.03050	94.5	93.8	94.5
50	0.1525	50.0	45.8	50.0
1	0.00305	50.0	49.9	50.0
10	0.03050	50.0	49.1	50.0
10	0.03050	40.0	39.18	40.0
1	0.00305	40.0	39.91	40.0
1	0.00305	100	99.93	100

For simplicity, regarding electricity costs, we took Germany as an example. According to “Strom Report” (based on Eurostat data) [40], German retail consumers paid 0.00305 Euro cents for a Wh of electricity in 2017. We used this value to calculate the cost of energy by plugging it in the equation presented in (2)”, where “ c ” in this case stands for the energy cost. As can be seen, the APEC metric favors lower power consumption and cost, favoring the use of green energy (free and clean energy).

4.7 TTCAPC and TTCAPEC Calculators

Regarding the experiments with the proposed TTCAPC [16] calculator feature, we simulated a custom DL model on two platforms and presented the results in Table 7.

Table 7. TTCAPC with Accuracy delta = 0.1, Energy delta = 1, beta = 0.1, alpha = 0.1.

	Desktop PC	Nvidia Jetson TX2
Accuracy [%]	97.92	
Energy Consumption [Wh]	50	10
Rounded Accuracy [%]	97.95	
Rounded Energy Consumption [Wh]	50.5	10.5
Closest APC [%]	61.28	87.11
Train seconds	60	

As can be seen, even though the accuracy and training time is the same for both platforms, the TTCAPC feature favors the platform which has less power consumption.

Regarding the experiments with the proposed TTCAPEC [16] calculator feature, we simulated with the same DL model values used also in the experiments regarding the TTCAPC calculator earlier and presented the results in Table 8.

Table 8. TTCAPEC with Accuracy delta = 0.1, Energy delta = 1, beta = 0.1, alpha = 0.1.

	Desktop PC	Nvidia Jetson TX2
Accuracy [%]	97.92	
Energy Cost (cents)	0.1525	0.0305
Rounded Energy Cost (cents)	0.1525	0.0305
Rounded Accuracy [%]	97.95	
Closest APEC [%]	51.46	82.96
Closest APEC Green (Solar) Powered [%]	97.95	
Train seconds	60	

As can be also seen in this case, the TTCAPEC feature favors the lower power consumption of a system because it results in a lower cost. Additionally and more importantly, it favors DL-based systems that are powered by green energy, because they have 0 electricity costs and no negative impact on our environment.

5 Conclusions

In this paper, we present a Computer Vision application that succeeds in bringing common DL features needed by a user (e.g. data scientist) when performing image classification related tasks into one easy to use and user-friendly GUI.

From automatically gathering images and classifying them each in their respective class folder in a matter of minutes, to removing duplicates, sorting images, training and evaluating a DL model in a matter of minutes, all these features are integrated in a sensible and intuitive manner that requires no knowledge of programming and DL. Experimental results show that the proposed application has many unique advantages and also outperforms similar existent solutions. Additionally, this is the first Computer

Vision application that incorporates the APC, APEC, TTCAPC and TTCAPEC metrics [16], which can be easily used to calculate and evaluate the performance of DL models and systems based not only on their accuracy but also on their energy consumption and cost, encouraging new generations of researchers to make use only of green energy when powering their DL-based systems [15].

References

1. Ashmore, R., Calinescu, R., Paterson, C.: Assuring the machine learning lifecycle: desiderata, methods, and challenges. [arXiv:1905.04223](#), May 2019
2. Soni, N., et al.: Impact of artificial intelligence on businesses: from research, innovation, market deployment to future shifts in business models. [arXiv:1905.02092v1](#), May 2019
3. Roh, Y., Heo, G., Whang, S.E.: A survey on data collection for machine learning: a big data – AI integration perspective. [arXiv:1811.03402v2](#), August 2019
4. Quandl. <https://www.quandl.com/>. Accessed 29 Feb 2020
5. URSA. <https://www.ursaspace.com/>. Accessed 29 Feb 2020
6. Kaggle. <https://www.kaggle.com/datasets>. Accessed 29 Feb 2020
7. Icrawler. <https://pypi.org/project/icrawler/>. Accessed 29 Feb 2020
8. Amazon Mechanical Turk. <https://www.mturk.com/>. Accessed 29 Feb 2020
9. Deng, J., et al.: ImageNet: a large-scale hierarchical image database. In: 2009 IEEE Conference on Computer Vision and Pattern Recognition, Miami, FL, USA (2009)
10. Barz, B., Denzler, J.: Do we train on test data? Purging CIFAR of near-duplicates. [arXiv:1902.00423](#), February 2019
11. Swanson, A., et al.: Snapshot Serengeti, high-frequency annotated camera trap images of 40 mammalian species in an African savanna. *Sci Data* **2**, 150026 (2015). <https://doi.org/10.1038/sdata.2015.26>
12. Nakkiran, P., et al.: Deep double descent: where bigger models and more data hurt. [arXiv:1912.02292](#), December 2019
13. Schwartz, R., Dodge, J., Smith, N.A., Etzioni, O.: Green AI. [arXiv:1907.10597v3](#). August 2019
14. Strubell, E., Ganesh, A., McCallum, A.: Energy and policy considerations for deep learning in NLP. [arXiv:1906.02243](#), June 2019
15. Jurj, S.L., Rotar, R., Opritoiu, F., Vladutiu, M.: Efficient Implementation of a Self-Sufficient Solar-Powered Real-Time Deep Learning-Based System, to appear
16. Jurj, S.L., Opritoiu, F., Vladutiu, M.: Environmentally-Friendly Metrics for Evaluating the Performance of Deep Learning Models and Systems, to appear
17. Rolnick, D., et al.: Tackling climate change with machine learning. [arXiv:1906.05433v2](#). November 2019
18. Nico, H., Kai-Uwe, B.: Dynamic construction and manipulation of hierarchical quartic image graphs. In: ICMR 2018 Proceedings of the 2018 ACM on International Conference on Multimedia Retrieval, New York, pp. 513–516 (2018)
19. Image Sorter. <https://visual-computing.com/project/imagesorter/>. Accessed 29 Feb 2020
20. Pirrung, M., et al.: Sharkzor: interactive deep learning for image triage, sort, and summary. [arXiv:1802.05316](#) (2018)
21. Apple Photos. https://www.apple.com/ios/photos/pdf/Photos_Tech_Brief_Sept_2019.pdf. Accessed 29 Feb 2020
22. Morra, L., Lamberti, F.: Benchmarking unsupervised near-duplicate image detection. [arXiv:1907.02821](#), July 2019

23. Zhou, W., et al.: Recent advance in content-based image retrieval: a literature survey. [arXiv:1706.06064v2](#), September 2017
24. Zhou, Z., et al.: Effective and efficient global context verification for image copy detection. *IEEE Trans. Inf. Forens. Secur.* **12**(1), 48–63 (2017)
25. Cicconet, M., et al.: Image Forensics: detecting duplication of scientific images with manipulation-invariant image similarity. [arXiv:1802.06515v2](#), August 2018
26. Jain, T., et al.: Image Deduplicator (Imagededup). <https://idealo.github.io/imagededup/>. Accessed 29 Feb 2020
27. Florencio, F., et al.: Performance analysis of deep learning libraries: TensorFlow and PyTorch. *J. Comput. Sci.* **15**(6), 785–799 (2019)
28. Cloud AutoML. <https://cloud.google.com/automl/>. Accessed 29 Feb 2020
29. He, X., et al.: AutoML: A Survey of the State-of-the-Art. [arXiv:1908.00709v2](#), August 2019
30. PyTorch on AWS. <https://aws.amazon.com/pytorch/>. Accessed 29 Feb 2020
31. Microsoft Azure. <https://azure.microsoft.com/>. Accessed 29 Feb 2020
32. Faes, L., et al.: Automated deep learning design for medical image classification by health-care professionals with no coding experience: a feasibility study. *Lancet Dig. Health* **1**(5), e232–e242 (2019)
33. Lennan, C., et al.: Image ATM. <https://github.com/idealo/imageatm/>. Accessed 29 Feb 2020
34. von Kistowski, J., et al.: Measuring and benchmarking power consumption and energy efficiency. In: *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering (ICPE 2018)*, pp. 57–65. ACM, New York (2018). <https://doi.org/10.1145/3185768.3185775>
35. Standard Performance Evaluation Corporation (SPEC) Power. https://www.spec.org/power_ssj2008/. Accessed 29 Feb 2020
36. MNIST converted to PNG format. https://github.com/myleott/mnist_png. Accessed 29 Feb 2020
37. Jurj, S.L., Opritoiu, F., Vladutiu, M.: Real-time identification of animals found in domestic areas of Europe. In: *Proceedings SPIE 11433, Twelfth International Conference on Machine Vision (ICMV 2019)*, 1143313, 31 January (2020). <https://doi.org/10.1117/12.2556376>
38. Chest X-Ray Images (Pneumonia). <https://www.kaggle.com/paultimothymooney/chest-xray-pneumonia/>. Accessed 01 Apr 2020
39. Cohen, J.P., Morrison, P., Dao, L.: COVID-19 image data collection. [arXiv:2003.11597](#) (2020). <https://github.com/ieee8023/covid-chestxray-dataset>. Accessed 01 Apr 2020
40. Strom-Report. <https://1-stromvergleich.com/electricity-prices-europe/>. Accessed 29 Feb 2020