Users' Guide for DeepSR

An Open Source Deep Learning Tool for Super-Resolution

Release 0.0.53

Hakan Temiz

December 2020

DOI 10.5281/zenodo.3354245

Contents

Introduction

The super-resolution (SR) task is an effort of producing high resolution (HR) images from a single or multiple low resolution (LR) images. Main goal of SR is to increase actual resolution of a given image with producing finer details either from single image or from multiple images of a scene. Significant amount of research efforts have already been spent for solving the SR problem. Very most of recent research exploited deep learning (DL) algorithms in the task of SR, and such algorithms achieved very significant result then other algorithms reached. Many different types of DL algorithms have been developed and used very efficiently to challenge the problem of SR. Researchers used a variety of DL frameworks and programming languages for implementation of their algorithms. Some of these frameworks are Keras [1], Tensorflow [2], Theano [3], Caffee [4], CNTK [5], and MatconvNet [6].

It is very well known fact that it is a time consuming and an effort taking job to develop a program and to perform all pipelines of development. This applies also to the programs built for working on the SR problem. Even though above mentioned frameworks help researchers challenging with all pipelines of SR task (training, testing, pre and post processing, etc.) providing a basis at their disposal, significant amount of time and human effort are still needed. It is not possible to leverage the functionalities of these frameworks as they are, because the SR problem requires special operations at each step of its pipelines. Researchers still have to develop subject-specific functionality and components for their program and have to deal with adapting them to SR problem. In some cases, various methods, functions or other features of programs may require to be written from the scratch. In order to eliminate such burdens and to ease the entire process, we have commenced an effort to develop a tool facilitating researchers deal with the challenges mentioned above.

DeepSR is an open source progam that eases the entire processes of the super-resolution (SR) problem in terms of Deep Learning (DL) algorithms. DeepSR makes it very simple to design and build DL models. Thus, it empowers researchers to focus on their studies by saving them from struggling with time consuming and challenging workloads while pursuing successful DL algorithms for the task of SR.

Each step in the workflow of SR pipeline, such as pre-processing, augmentation, normalization, training, post-processing, and test are governed by this tool in such a simple, easy and efficient way that there would remain at most very small amount of work for researchers to accomplish their experiments. In this way, DeepSR ensures a way of fast prototyping and providing a basis for researchers and/or developers eliminating the need of starting entire process from scratch, or the need for adapting existing program(s) to new designs and implementations.

DeepSR is designed in such a way that one can interact with it from the command prompt, or use it as class object within another program by importing. It is mainly tailored for using it with scripts from command prompt by providing many ready-to-use functionalities at hand. Furthermore, multiple tasks/experiments can be performed successively by using batch scripts. However, addition to command prompt interface, it is ready also to do the same tasks by calling it as a class object from another Python program. One can also develop his/her own programs or can write Python scripts performing subject specific tasks at his/her disposal. He/she can add new features to the program to contribute this effort for making it better.

1. Installing and Using The DeepSR

Though it is possible to use the DeepSR directly by downloading the source project folder with its contents, more elegant way is installing it, since the tool requires many other Python packages and frameworks. The user will have to install all required packages and/or frameworks manually in the case the user decides to use the program without installing it. The installation procedure shall automatically install those dependencies of the program.

Before using the tool, Python[1] programming language must have already been installed. Depending on the operating system users have, appropriate version of Python must be downloaded, and then, installed on the local computer. Afterwards, the user can proceed to installation of the tool.

The downloadable source code and binaries for Python environment are served at the PyPI[2] repository which helps people find and install software developed and shared by the Python community. The address of the tool is as follows:

[1] https://www.python.org/
[2] https://pypi.org/

https://pypi.org/project/DeepSR/

As it is done usually with typical Python packages, DeepSR can easily be installed from this web site. In order to install the tool, the following command has to be issued in command prompt:

```
python –m pip install DeepSR
```

This command starts the installation procedure. The other frameworks, (Python packages) on which the DeepSR depends are will also be installed automatically without the need of user intervention.

We use the GPU version of the Tensorflow library as default. Therefore, GPU version of Tensorflow shall be installed while installing the tool. Interested users can install CPU version of it after uninstalling the GPU version. Addition to Tensorflow, Theano also comes with this tool. Other deep learning libraries that Keras uses as backend can also be installed by intended users for the use.

The DeepSR exploits the Keras' API for visualization of the models' layout. Hence, Keras uses GraphViz[3] for the visualization. Though the Python pakages of the GraphViz is installed automatically, the user needs to install the executable (binary) code for the appropriate platform (e.g., Windows). The user may also be required to add its installation path to the path environment list of the operating system.

Ones it is installed. It can be imported into a Python file or can be used in Python interpreter. The following code snippet is a simple example for calling (importing) the DeepSR as Python package. After importing an instance of it is being created.

```
from DeepSR import DeepSR

dsr = DeepSR()
```

The tool can be used also from command prompt. In fact, using it from command prompt with scripts make it very versatile and a powerful tool since most of users may prefer to using scripts. Once appropriate settings/instructions are provided in scripts, bunches of works can be pipelined for a series of tasks. The program is directly callable from anywhere in the command prompt.

The following code snippet demonstrates the use of DeepSR in command prompt with scripts.

```
python –m DeepSR.DeepSR --modelfile "sample_model_1.py" --train --test --inputsize 25
--stride 5 --colormode YCbCr  --channels 3 --target_channels 3 --saveimages –plot
--metrics PSNR SSIM MAD
```

The DeepSR can be run as follows also by changing the current working folder in the command prompt to the folder path in which DeepSR.py resides:

```
python DeepSR.py --modelfile "sample_model_1.py" --train --test --inputsize 25 --stride 5
--colormode YCbCr --channels 3 --target_channels 3 --saveimages --plot --metrics PSNR SSIM
MAD
```

Or, regardless of the current working folder, the user can provide the full path of the DeepSR.py to run the program as follows:

```
python "C:\Users\<user name>\AppData\Local\Programs\Python\Python36\lib\site-
packages\DeepSR\DeepSR.py" --modelfile "sample_model_1.py" --train --test --inputsize 25
--stride 5 --colormode YCbCr  --channels 3 --target_channels 3 --saveimages --plot
--metrics PSNR SSIM MAD
```

The <user name> in this command must be replaced with the user name of the local computer. Please note that entire path given above can be totally different depending on the installation of the Python, or its environment.

Please refer to the Section 4. for detailed information about the implementation of the DeepSR tool with command prompt.

[3] https://www.graphviz.org/

2. DeepSR

DeepSR is dedicated to is to enable users to focus on their studies while pursuing successful DL algorithms for the task of Super Resolution (SR), saving them from the workloads of programming, implementing and testing. It maintains the reusability of program code and rapid prototyping, in such a manner that is saving time and requiring very little human effort. It offers a parametrized, neat and simple interface for user interaction. Besides, it governs all pipelines of the work automatically and very efficiently. It consists of many useful components and features that meets the needs of researchers. We took into account more than typical needs of researchers, which make the program versatile in terms of pre and post processing, training and testing.

The SR task requires some special pre and/or post processing operations, along with some particular regularizations before and after images are processed by DL models. After the images were prepared with SR-specific operations for training or testing, a DL model processes them, and then, the outputs of models are post processed to make them be suitable for further SR-specific operations. Each of above mentioned operations and much more are standardized and automated by the DeepSR.

In order to construct DL models, DeepSR utilizes Keras API[4]. Keras is capable of running on top of the following prominent DL frameworks: TensorFlow, Theano and CNTK. It transforms DL models designed by coding with its API into the models of these frameworks. Thus, this program is capable of running models on both CPUs or GPUs seamlessly by leveraging these frameworks.

All needs to be done in the case of interacting the program in command prompt (this is valid also when using the program as a class object from another program) is to build up a DL model using the API of Keras within a model file (a python file), and then, provide the program the relevant information and/or instructions. A model file (.py file), in summary, contains the following two things in essential:

- A dictionary named `settings`. This dictionary is used to supply the DeepSR with all necessary settings and/or instructions to perform tasks.
- A method named `build_model`. This method preserves the definition of the model in Keras API, and returns it the composed version of it. By this way, the DeepSR intakes deep learning models (either from command prompt, or whereas it is used as an class object)

Please refer to the Section 3 for further documentation.

In order to accomplish entire work, all parameters can be given to the DeepSR in any of the following ways:

(i) In a Python dictionary object in the same model file in which the DL model is coded with Keras API.

(ii) Directly calling the relevant member method of the DeepSR class, or setting the relevant parameter of the class.

(iii) Supplying the parameters in the command prompt.

The instructions allow one to perform any task of SR such as pre and/or post processing, training, testing, logging and visualization in a very simple and easy manner. Any parameters required by a DL model, such as learning rate, decay, number of epochs, number of batches, etc., for any task (i.e., training, test) can be provided by using any of above mentioned ways. Thus, there is no need to write a program from scratch or modify the program codes for new tasks or models.

DeepSR comes with several appealing features also. Some of these tools are as follows, data (image) augmenting, different types of pre and post processing, and a variety of training methods. Visualizing models, models' layer weights or layer outputs graphically. Users may also save these in files. It is also capable of evaluating the performance of models by a number of image quality assessment metrics out of the box. Furthermore, it is also possible to compare the performance of the model with the following interpolation methods: Bicubic, Bilinear, Lanczos, and Nearest.

In brief, DeepSR offers at least the following features allowing researchers to turn their ideas into results with eliminating the need for dealing with details:

[4] https://keras.io/

- Easy and fast DL model prototyping, development and implementation interface.
- The ability to use with scripts in command prompt, or to use as a class object within another program.
- The ability to perform multiple consecutive or distinct tasks in one go with batch scripts without the need for user intervention.
- Subject-specific pre and/or post processing procedures and data augmentation tailored particularly for SR problem.
- The ability to evaluate the success of DL models by a number of image quality assessment measures per image in test set(s) and report in an Excel file.
- The ability to visualize and/or save model layouts, layer weights or layer outputs.
- Offering additional neat features such processing images independently for training and test procedures, in different color spaces, as a single color channel or as 3 color channels.

DeepSR is provided as open source tool, available under the terms of the MIT license, which guarantees that the source code of the program can be dissected, modified and such changes be kept as open. Thus, it may be freely used, studied or modified provided that it remains open source. Addition to the given installation address on PyPI in Section 1, entire project with the binaries, source code, documentation and more is served on Github platform in the following address:

https://github.com/htemiz/DeepSR

2.1. Program Structure

DeepSR fundamentally consists of three Python files: DeepSR.py, args.py, and utils.py. The DeepSR object is declared in the file DeepSR.py. It is the core python module called first for running the program. All parameters that can be used with DeepSR in command prompt are defined in the file args.py. The module, utils.py accompanies many tools making the life easier and maintaining the reusability.

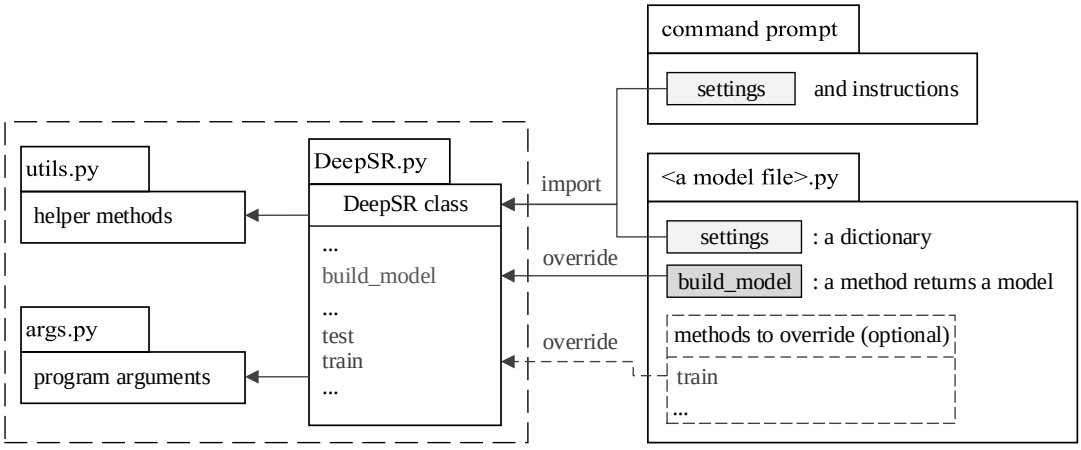

Figure 1. The conceptual scheme of the DeepSR.

File Name	Description
DeepSR.py	main program file
utils.py	Contains helper methods for implementation and additional utilities.
args.py	All arguments of the program are defined within this file.

The DeepSR handles each tasks of super resolution problem for yielding larger images with better quality by usind Deep Learning models. Figure 2 represents the workflow training process of DL models. DeepSR takes each image (y) from a training set and a produces low resolution image(s) (x) according to the problem definition of Super Resolution. After performing a series of pre-processing works (augmenting, normalization, etc.), low resolution image is passes over to the DL model for learning the relations between low resolution image (x) and known high resolution image (y). The model checks

its success on predicting the high resolution image by comparing its predicted with high resolution image (y), and back propagates the error back to its layer weights to minimize (converge) the error. This process is being done for entire images in the training set. When the model has been trained with all images, it is said that '1 epoch is completed'. Training procedure shall be carried on until the model converges a minimum error, or at a point the performance of the model gets worse.

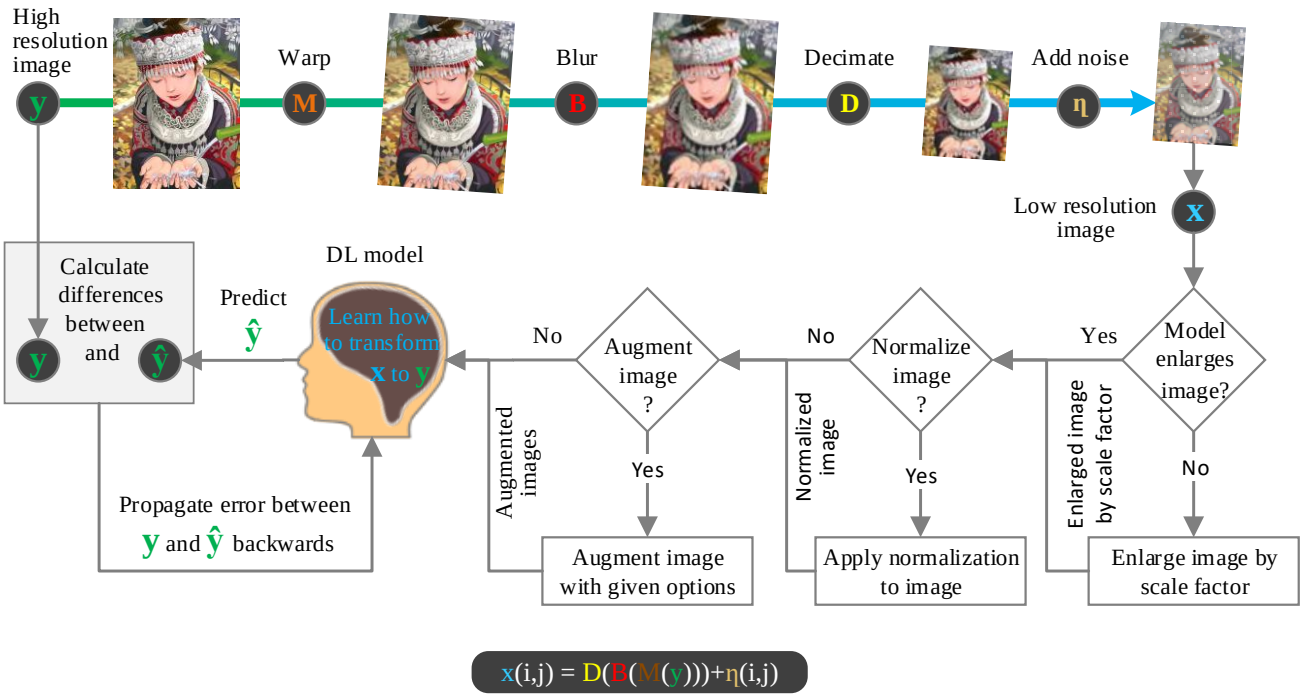

Figure 2. Workflow for the training procedure with DeepSR

In Figure 3 a diagram workflow test procedure with DeePSR is represented. For a given test test of images, each image is taken and pre-processed before passing it to the model, and then, predicted output high resolution image is compared to existing high resolution image by a number of image quality measures. All results gathered from test for each image are saved in an Excel file, along with the average scores of each quality measures over test set.

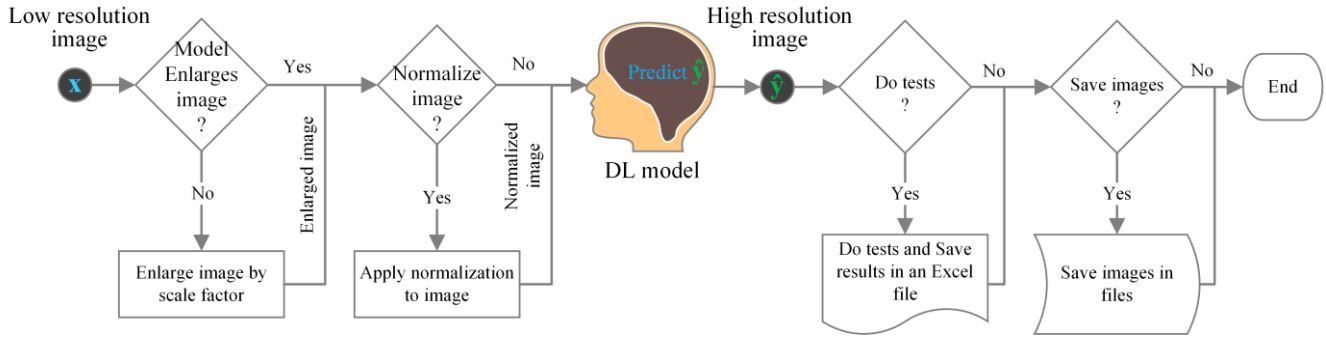

Figure 3. Workflow of the implementation of trained models with DeepSR.

2.2. Preprocessing: Augmentation, Normalization, and Noising

The task of SR requires special image preprocessing operations in order to have images ready for training models or testing. An example case is that a model may be designed to process a single color channel of images, even though images have more than a single color channel. In such a case, an input image with a single color channel and corresponding reference HR image with single channel should be prepared accordingly for the purpose of training, even though images have more color channels.

The proposed tool handles all these workloads without the user intervention. Another case is that some of DL models take LR image as an input, upscale it by the given scale factor at a stage of processing and finally output the upscaled HR image, while others take a middle LR (MLR) image which is produced by upscaling LR image by the scale factor using an interpolation method such as Lanczos, Bicubic, Bilinear, or Nearest, and then, processes MLR image to output HR image. DeepSR is furnished well for handling such pre-processing burdens. The user just sets a parameter to indicate whether his/her model upscales the input LR image or not, and choses the interpolation method for upscaling LR image. The rest of the work accomplished by the tool, not requiring any other user interaction.

Data augmentation is also a very important process under circumstances that there is not enough training data. It helps models in reinforcing representation ability, improving the capability of learning, and thus, their performance. The process of data augmentation is typically done by applying some sort of translations, transformation and domain-specific operations to existing training data. Hence, much more training data can be obtained than the amount of the original training data. In order to provide the user the ability of data augmentation, we developed a parametrized way of data augmentation. That is, consisting of a variety of image transformation operations (i.e., rotating image by 90, 180, 270 degrees, or flipping it vertically, horizontally, or the both) which are done on an online fashion. One can just augment the original training data (images) at his or her hand just through a single parameter. We note that more image transformation and translation operations to be included in the future releases of the tool, to provide a wide range of means of data augmentation.

Normalization is also very crucial task enables DL models to learn very rapidly and stable. For the sake of reusability and simplicity, most common normalization operations that a researcher would most likely need are also included in the tool. Followings are of some of those operations:

- **Dividing by a certain value.** The intensity values of images are divided by the given value. Default is 255.0 unless any value is specified.
- **Mean.** In the mean normalization the mean value of images are subtracted from images before images are being processed at training stage. The given value shall be used for mean normalization. The mean value is calculated from entire training images if ‘whole’ key is prefixed. For example, ‘—normalize mean whole’, indicates that the mean normalization applied to images by calculating the mean value from the entire training images. Similarly, if the key ‘single’ is prefixed, then, the mean value is calculated from each images individualy while they are being processed. The mean value can be given separately for each color channels.
- **Min-max normalization.** The intensity range is adjusted between the given minimum and maximum values. The minimum and maximum values are 0 and 1, respectively unless any values are specified.
- **Standardization.** Images are standardized in this case, with mean and standard deviation. The mean and standard deviation values can be given by the user, or can be calculated from single image or from entire training images. In order to standardize each image with its own mean and standard deviation, type ‘—normalize standard single’. Or, to use the same mean and standard devation values calculated from entire training set, type ‘—normalize standard whole’. The values are calculated in this case from the entire training images.

Adding noise to images is also one of sub steps of Super Resolution pipelines. The noising is the last operation in the procedure of producing LR images from know HR images. This tool is also capable of noising images with Gaussian noising with the given mean and variances.

2.3. Training

There are several different types of training processes made available by the tool to researchers. Researchers may choose to train models in either online or offline fashion, using a single or multiple epochs. In the case of online training, DeepSR uses images located within a folder. In the case of offline training, it uses an already prepared dataset (a .h5 file) if exist, or creates it first before using it for training. All training methodologies maintained by just typing an appropriate parameter in command prompt or using the relevant method of the DeepSR object. Moreover, it also offers the commencing the training with the model weights yielded from earlier trainings. Models are first loaded with given layer weights from a file, before starting the training procedure.

As default, the model weights after each training epoch are saved as .h5 files in output folder. Users can carry on training models (having the same architecture, for sure) with these weights later on.

2.4. Testing and Predicting

Another appealing property of the proposed tool is that it automates all testing and predicting works in an efficient way. The tool automatically prepares ground-truth (reference) images and gets resulting HR images of models accordingly, and then, evaluates predicted HR images with a variety of image quality metrics (IQM), such that PSNR, SSIM, FSSIM, MSSSIM, and so on. The user may choose any combinations of these metrics for the evaluation of the performance of models. The evaluation results are saved in an Excel file. All tests can be done for a single weight file if the path belongs to a file, or done for multiple weight files if the given path points out to a folder in which multiple files of weights are. Every test results of IQM over each test data (image) for each (in case it is done for multiple weight files) weights of DL model is given in a tabular format in an Excel work sheet, along with another sheet of average values of each IQM resulted from entire test data (images). User may also compare the performance of his/her models with the following interpolation methods: bicubic, bilinear, lanczos, and nearest. The parameter 'interpmethod' provided for this purpose.

2.5. Post Processing

The output HR images may require some additional implementations so as to be ready for testing or for another tasks. For example, some DL models crop images from borders at a certain amount of pixels while processing images through their layers so as to alleviate the border effects. Thus, when it comes to evaluation of the performance of models, ground-truth (reference) images may require to be cropped to have the same dimensions as the output HR images have. Another situation may be the need to use a different color space for testing purposes than the color space used for training of models. Such requirements were taken into account and introduced to the proposed tool during the development of tool. Consequently, a several typical post processing operations that may needed are at the disposal of researchers for their use.

2.6. Visualizing Models and Layers

DeepSR can plot graphically and/or save in files the layout of models, layer weights, or layer outputs. The tool uses Keras' utility functions for plotting the graph of model and saving it to a file. User may choose to plot/save the weights or the outputs of a certain layer, or entire layers of models. This funcionality might be very important for users need to see/check the output images of each layers, especially for evaluation purposes of the success of the layers.

2.7. Program Parameters

Program parameters are the arguments in command prompt that determining a setting or instructing the program to do a specific job(s). They can also set to the DeepSR object with their exact names as class members in the case of interacting with DeepSR as an object.

All parameters can also be given in the dictionary, 'settings', in a model file. They must have the exact same name as given below in this case. Please note that **argument in command prompt has precedence** and overwrites the settings given in 'settings' dictionary. One can easily run models with alternative settings by using scripts in command prompt without changing the original model file.

All program parameters with explanations and their use are given below.

Argument Name	Explanation
activation	Activation function for layers of models. Enables user to change the activation functions in layers with scripts. Default value is 'relu'. In order to programmatically assign the activation function to layers from the command prompt, activation functioncan be typed as **'activation = self.activation'** in layer definition. For example, `my_model = Sequential()` `my_model.add(Conv2D(64,(9,9), activation=self.activation)` The activation function of the last layer in a model is defined with the parameter **'lactivation'**. If it is not given, the activation function of the last layer of a model is set the value of **'activation'** parameter. It can be **relu, elu, lrelu, prelu, selu, tanh, sigmoid,** and **exponential**.

	Usage: `--activation relu  # relu to be used as activation function`
augment	Augment options. The followings may be used: **90**, **180**, **270**, **flipud**, **fliplr**, **flipudlr**. **Usage**: `--augment 90 flipud fliplr  # augment by rotating 90 degrees,` `# flipping the image up-down and` `# left-right`
backend	Used for the selection of the Keras backend: tensorflow, theano. **Usage**: `--backend tensorflow  # backend is tensorflow.`
batchsize	The batch size defines the number of training examples given to the network before calculation of the error of the network and to propagate the error back through the network. **Usage**: `--batchsize 256`
channels	The number of color channels to be used for the trainig of of the network. It might be **1** or **3**. **Usage**: `--channels 3`
colormode	The color space to be used for the training of the model. Can be **RGB** or **YCbCr** **Usage:** `--colormode RGB`
crop	The number of pixels removed from borders of Ground truth and/or Interpolated images to make them to have the same size as the output image. Since some of the models output a cropped image due to the "padding" parameter of layers. For example, the input image of the prominent model SRCNN is of 33x33 pixels, whereas its output image is of 21x21 pixels. This parameter is similar to **crop_test**, but this effects only training stage, while **crop_test** does the same thing for test procedure. **Usage:** `--crop 6`
crop_test	The number of pixels removed from borders of Ground truth and/or Interpolated images to make them to have the same size as the output image. This parameter is similar to **crop**. This parameter is valid for test procedure. Refer to the parameter, **crop**, for further explanation. **Usage:** `--crop_test 6`
decay	The amount of decrease in learning rate parameter of model over training carry on. **Usage:** `--decay 0.5`
decimation	The interpolation method used in obtaining low resolution image from decimating down high resolution image by the scale factor. The following interpolation methods can be used for decimation: **bilinear**, **bicubic**, **nearest** and **lanczos**. The default is **bicubic**. **Usage:** `--decimation bicubic # bicubic interpolation is used in decimation.`
drate	A list of dilation rate values for layers. Default is (1,1). **Usage:** `--drate (2,2)  # dilation rate is 2 by 2`
epoch	The total number of complete pass of training through entire training data. **Usage:** `--epoch 10  # train the model for 10 epochs.`
espatience	Early stopping patience. The number of training epochs to wait before early stop if the model does not progress on validation data. this parameter helps avoiding overfitting when training the model. Deafult is 5 **Usage:** `--espatience 15  # stop training after 15 epochs if no improvement`

gpu	The number of GPUs to be used. '0' ... '3' to use 1 to 4. GPU, or 'all' to use all GPUs. Multiple GPUs also can be designated with separating by comma. For example, --gpu '0,1' for using GPUs with number 0 and 1 (0 and 1 indicates the first and second GPUs since it is zero-based indexed). **Usage:** `--gpu 1 # use only second GPU for this task`
kernel_initializer	Enables user to programmatically assign the kernel initializer function from the command prompt. It can be typed in model defined as follows: `model = Sequential()` `model.add(Conv2D(64,(1,1), kernel_intializer=self. kernel_intializer)` **Usage:** `--kernel_initializer glorot_uniform # glorot_uinform is initializer for layers, if layers are defined as above example.`
inputsize	The size of the input image given to model. Only one dimension is given. DeepSR uses the same size in both, width and height. **Usage:** `--Inputsize 35`
interp_compare	Interpolation method(s) to which the performance of the model is compared. It can be any combination of the following interpolation methods: **bicubic**, **bilinear**, **nearest**, **lanczos**. Use the keyword **all** for comparison the performance of the model with all interpolation methods. Use the keyword **same** to use the same interpolation method(s) in decimating down the image (overrides the method given in the argument '**decimation**') and upscaling back. Default is None. **Usage:** `--interp_compare bicubic # evaluate the performance of the bicubic # interpolation method on test data.` `--interp_compare all # compare the performance of the model # with all interpolation method(s).` `--interp_compare lanczos same # lanczos interpolation is being # used now for decimation procedure # also. So, the interpolation method # for decimation given in the # argument 'decimation' is now # overriden.`
interp_up	The interpolation method used in obtaining upscaled image from low resolution image. The following interpolation methods can be used: **bilinear**, **bicubic**, **nearest** and **lanczos**. The default is **bicubic**. **Usage:** `--interp_up bicubic # bicubic interpolation is used in upscaling.`
lactivation	Activation function for the last layer of models. Default value is 'relu'. In order to programmatically assign the activation function to the last layer from the command prompt, activation function can be typed as **'activation = self.lactivation'** in layer definition. For example, `my_model = Sequential()` `my_model.add(Conv2D(64,(9,9), activation=self.lactivation)` If it is not given, the activation function of the last layer of a model is set the value of **'activation'** parameter. It can be **relu, elu, lrelu, prelu, selu, tanh, sigmoid,** and **exponential**. **Usage**: `--lactivation relu # relu to be used as activation function of the last layer of the model`
layeroutput	Yields the result of each layer of the model while performing test. Should be used with the **test** paramater. **Usage:** `--layeroutput # save layers outputs while testing the model.`

layerweights	Returns the layer weights of the model while performing test. Should be used with the **test** paramater. **Usage:** `--layerweights # result in the weights of each layer of the model`
lrate	The amount that the weights are updated during training is referred to as the step size or the "learning rate **Usage:** `--lrate .0001 # learning rate is 0.001`
lrpatience	Lerarning rate changin patience. The number of training epochs to wait before changing the learning rate value. this parameter helps avoiding overfitting when training the model. No default value for this parameter. **Usage:** `--lrpatience 15 # change learning rate after 15 epochs`
lrfactor	The rate for changing the learning rate value. Default is 0.5. **Usage:** `--lrfactor 0.9 # rate for changing learning rate is 0.9`
metrics	Image metrics to be used for assessment of the performance of the model during test. Can be any combination of **MSE**, **NRMSE**, **PSNR**, **SSIM** (from **scikit-video**[5] library), **MAD**, **NIQE**, **BRISQUE** (from **scikit-image**[6] library), **MSSSIM**, **ERGAS**, **RASE**, **SAM**, **SCC**, **UQI**, **VIF** (from **sewar**[7] library), **SNR**, **BSNR**, **PAMSE**, **GMSD** (from **sporco**[8] library) Default is **PSNR**, **SSIM**. **Usage:** `--metrics PSNR SSIM MSSSIM # use this measures for test`
modelfile	Path to the file in which the network model is defined. It should be a Python file containing a method **build_model** which returns a Keras model object. The file can also contain a Python dictionary **settings** for providing any of parameters/settings. This argument is used only in command prompt while interacting with the program in command promtp. Not necessary when using the program as class object. **Usage:** `--modelfile "c:\example_model.py" # the model file`
modelname	The name of the current model. User may assign a name for the current model by this parameter. If it is not given, model name be the name of the model file. The program creates a folder with the same name as the model, if not exist. All outputs are appropriately saved by the program within this folder, provided that any path is not given with the parameter 'workingdir'. **Usage:** `--modelname DeepSR # the model name is DeepSR.`
noise	The Gaussian noise to be added to images. The mean and variance values must be provided with this parameter. Default is None. If it is None, noise will not applied to images. **Usage:** `--noise 0.0 0.01 # Gaussian noise with zero mean and variance 0.01 # to be added to images.`
normalization	Normalization method to be applied to the input images. Can be any of the following: **divide, mean, minmax, standard.** **divide** is used to divide the intensity values of image by a certain value. **minmax** adjusts the intensity values between a certain limits. For example between 0 to 1. **standard** does standardization to the images. It is done like this; first the mean value of image is calculated, and the mean value is subtracted from the image, and then the image is

[5] http://www.scikit-video.org
[6] https://scikit-image.org/
[7] https://pypi.org/project/sewar/
[8] https://pypi.org/project/sporco/

	divided by the standard deviation calculated from the image. The values of mean and standard deviation may also be given, instead of calculating them from the image. If they are not given, they are calculated from the given image. **mean**, subtracts the mean value of the image from the image, If, not any value provided with this parameter. If mean value is provied after this parameter, that value to be subtracted from the image. **Usage:** `--normalize divide 255.0  # normalize image dividing by 255` `--normalize minmax -1 1  # set intensity values between -1 and 1` `--normalize minmax  # set intensity values between 0 and 1` `# even if they are not given.` `--normalize standard single  # standardization with mean and stddev` `# values calculated from the images` `# individually.` `--normalize standard  # the same as above.` `--normalize standard whole  # standardization with mean and stddev` `# values calculated from entire set of` `# training images. One these values are` `# calculated from the training set, the` `# same values are used for each images` `# in this case.` `--normalize standard 1 3  # standardize with mean 1, stddev 3` `--normalize mean  # subtract their mean values from images` `--normalize mean 111  # subtract the value of 111 from image.`
normalizeback	Boolean. The reverse normalization procedure is performed on the result image after model ouputs. The normalization method is defined in the parameter normalize. **Usage:** `--normalizeback  # normalize back the output image of the model.`
normalizeground	Boolean. The ground truth (reference) image will also be normalized, if this parameter is given in command line, or set to True in DeepSR class. **Usage:** `--normalizeground  # ground-truth image is normalized with the same` `# normalization method as applied to input image.`
outputdir	String. The output folder path in which the results are saved. If it is not given, the program creates output folder within the working folder determined by the parameter **workingdir** in the case of running the program from command prompt, or within the current folder the program runs in, in the case of running the program as DeepSR object in another program. Sub folders are also created for the results of (training, test, etc.) each scale factor. **Usage:** `--outputdir "C:\example_model\output"`
plotimage	Boolean. Command to plot and/or save the output image yielded by the model. Does not take any additional parameter. This procedure can also be performed by calling DeepSR object's member method with the same name as this argument. **Usage:** `--plotimage`
plotmodel	Plots the architecture of the model in an image file. Used together with the **test** parameter. **Usage:** `--test --plotmodel # plots the model layout while performing test.`
predict	Command to get the prediction scores of models for a given image with given model weights. Takes no additional arguments. Used to get result image without implementing a test procedure. This procedure can also be performed by calling DeepSR object's member method with the same name as this argument. **Usage:** `--predict`
rebuild	Boolean. If its True, model is being rebuilt for each test file, since some models need to have been feeded with images's width and height. Otherwise, model is built only once at the beginning of Test method. If image's height and width should be given for each images, main input of the model in the definition file should be similar to this: `def build_method(self, testmode=False)`

	`` if testmode: input_img= Input(shape=(self.img_height, self.img_width, self.channels)) else: input_size = None input_img = Input(shape=(input_size, input_size, self.channels)) x = Conv2D(16, (3,3), padding='same')(input_img) ... ``` Here, `Input` is the Keras's input Layer. **Usage:** `--rebuild  # model to be re-built for each test file in test.`
saveimages	If set, the output images of the model and/or other interpolation methods are saved in the output forlder, while performing the tests. Used together with the **test** parameter. **Usage:** `--saveimages  # saves output images while testing.`
scale	The magnification factor. **Usage:** `--scale 2  # to use magnification factor of 2.`
seed	The seed value for random number generators. Random number generators are used in assigning values to the layer weight when the network is first created. It provides a way of doing the job in the same conditions repeatedly for different models. **Usage:** `--seed 19`
shuffle	Shuffles the input data (images), if it is given. The program shuffles also image patches taken from images. **Usage:** `--shuffle  # shuffle the input data before given them to the model.`
shutdown	Computer will be turned off after the test has completed. **Usage:** `--test --shutdown  # do test, and after test is finished, shut down the computer`
stride	The stride value (step) for leaving intervals while taking image patches to be given to the model at the training stage. **Usage:** `--stride 11  # leaves a gap of 11 pixels between image patches`
target_channels	The number of color channels to be used in the test of the network performance. User may choose the number of color channels to be used for comparing the output image of the network with the ground truth image. User may choose 1 or 3. **Usage**: `--target_channels 3`
target_cmode	The traget color mode of the output images. The input color space of the input images and the color space of the output images may be different by using this parameter. It will be the same as **colormode** parameter unless it is provided. **Usage:** `--target_cmode YCbCr # the color space of output images is YCbCr`
test	Command to perform test. Test procedure to be performed if it is given in command prompt. Test procedure can also be performed by calling DeepSR object's member method with the same name as this argument. **Usage:** `--test  # do test`
testpath	The folder path or a single file path of the test image(s). If it is a folder path, the images in that folder automatically used for test. If it is path to a file, the test is done only for that file. **Usage:** `--testpath "C:\test_folder"  # take images from this folder for test.`

	`# do test for two separate folders.Each test will be done` `# individually and results to be written in corresponding file.` `--testpath "C:\test_folder_1" "C:\test_folder_2"`
train	Command to train model. Training procedure can also be performed by calling DeepSR object's member method with the same name as this argument. **Usage:** `--train  # start training procedure` `--train --test  # train the model first and then test it.`
traindir	The folder path in which the training data (images) are. Each image in this path is used for training. **Usage:** `--traindir "C:\training_files"  # train with images in this folder`
upscaleimage	Indicates whether the input image is upscaled by the given interpolation method before giving it to the model. In case a model takes the downscaled low resolution image and upscales it by itself this parameter should be **False**. Must be **True**, otherwise. **Usage:** `--upscaleimage  # this means that model will not upscale the input` `# image, and hence, the input image to be upscaled` `# with a given interpolation method before handling` `# it to the model`
valdir	The folder path or a single file path of the validation image(s). It is similar to the testpath parameter. **Usage:** `--valdir "C:\validation_images"  # folder path of validation images`
weightpath	The folder path or a single file path of the weight file(s). If this parameter points to a single weight file, the training will start with assigning the weights in this file to the model. The same applies to the test procedure. If it points to a folder path, the test procedure will be done for each weight files in the path. **Usage:** `--weightpath "C:\weights_folder"  # do test for each weight in this` `# folder`
workingdir	The folder path to the working folder. All outputs to be produced within this directory. If it is not given, DeepSR resolves the path from the parameter **modelfile** in case of calling the program from command prompt, or uses the folder path of the actual Python file being run in case of using DeepSR as class object. The output directory (**outputdir** parameter, or the member of the class) of the project is determined by taking this folder path as root folder of the project unless the output directory is specified by the **outputdir** parameter. Sub directories shall be created under this folder respect to the program parameters. For example, sub directories for each scale is created under 'output' folder in wihin the root folder path (**outputdir** parameter, or member of DeepSR class object). **Usage:** `--workingdir "C:\working_folder"  # working folder for outputs.`

3. Model Files

We provide here in this section two different sample model files for understanding the program pipeline comprehensively.

A Model file is an ordinary Python file (a .py) must have at least two things within itself:

- A dictionary named **'settings'**.
- A method named **'build_model'**.

This dictionary is used to provide the program all necessary settings and/or instructions to perform intended tasks. Settings are given in the dictionary in a key-value pairs. All keys are summarized in the section 1.7. The keys correspond to command arguments in command prompt. One can easily construct and set his/her models with parameters by providing key-value pairs in this dictionary to perform Super Resolution task with Deep Learning methods. However, the same settings can also be

given to the program as arguments in command prompt. Please note that although settings were designated in the dictionary, the same settings provided as arguments in command prompt override them. Thus, the program ignores the values/instructions of those settings designated in the dictionary and uses the ones given as command arguments. This is handled by the DeepSR at the construction stage of the class object. The alteration of the settings is valid only for the class object, not valid for model files. Namely, settings are not changed in the dictionary within the model files. Model files will remain intact.

The build method is assigned to DeepSR object as a member function with the same name. This function must take a parameter for determining whether the model will be constructed for training or test. Default value is **`False`**. Which means that the model to be constructed for training. Thus, the input shape of the first layer of the model will be resolved from the members of DeepSR (or command arguments) **`inputsize`** and **`channels`**. The input shape of the first layer of a model should have the form of (None, None, number of color channels) in test mode. In training mode, it should have form of (input size, input size, number of color channels). For example, let us assume that input size (image width and height) is 30, and color channels are set to 3 in order to use all color channels of images.

In the training mode input shape should be like this:

input_shape(30, 30, 3)

But, in the test mode input size must be None, and thus, the input shape should be like this in order to use input images with full size:

input_shape(None, None, 3)

The first and second parameters in here are, respectively the height (rows) and the width (columns) of the input image. The first layer of models should be formed with input sizes as explained above. In the following code snippet an example of **`build_model`** method is presented. The method constructs a Keras' Sequential model. Before starting the declaration of the model, the layer shape is resolved by checking the parameter, **`testmode`**. After that, the input shape of the first layer is set to that shape in the declaration.

```
#  a method returning a keras model
def build_model(self, testmode=False):

    if testmode:
        input_size = None
    else:
        input_size = self.inputsize

    input_shape = (input_size, input_size, self.channels) # (height, width, channels)

    a_model = Sequential()
    a_model.add(Conv2D(64,(9,9), activation='relu', padding='valid', input_shape=input_shape))
          …
          …
          …
    return a_model
```

Please refer to the Sections 3.1 and 3.2 for examining sample model files in detail.

3.1. Sample Model File 1

```
#  sample_model_1.py file. Returns a simple convolutional neural network (CNN) model #
from keras import losses
from keras.layers import Input
from keras.layers.convolutional import Conv2D, UpSampling2D
from keras.models import Model
from keras.optimizers import Adam
from os.path import  basename

settings = \
{
'augment':[], # can be any combination of [90,180,270, 'flipud', 'fliplr', 'flipudlr' ], or [] for none.
'backend': 'tensorflow, # keras is going to use tensorflow
'batchsize':128, # number of batches
'channels':3,  # color channels to be used in training . Three one channels in this case
```

```
'colormode':'RGB', # 'YCbCr' or 'RGB'
'crop':6,  # crop images by 6 pixesl from each border in training process.
'crop_test':5, # crop the images by 5 pixesl from each border in the test
'decay':1e-6,  # learning rate decay for some optimizers.
'espatience' : 3, # stop after 3 epochs if the performance of the model has not improved.
'epoch':5, # train the model for total 5 passes on training data.
'inputsize':41, #  size of input image patches is 41x41.
'lrate':0.001,
'lrpatience': 2, # number of epochs to wait before reducing the learning rate.
'lrfactor' : 0.5, # the ratio of decrease in learning rate value.
'minimumlrate' : 1e-7, # learning rate can be reduced down to a maximum of this value.
'modelname': 'my_model_1', # the model name is 'my_model_1'.
'metrics': ['ALL'], # measure the output HR images with all image quality metrics.
'normalization': ['divide', 255.0], # normalize images during training by dividing intensities by 255.0
'normalizeground':False, #'normalize': ['minmax', -1, 1],
'normalizeback': False, #use it to have the result image normalize back with inversing the normalization
'outputdir':'c:\output_folder', # all outputs to be written within the following path: 'c:\output_folder'
'scale':2, #  magnification factor is 2.
'stride':17, # give a step of 17 pixels apart between patches while cropping them from images for training.
'target_channels':3, # color channels to be used in tests . Three channels in this case
'target_cmode':'RGB',# we procesed images in YCbCr color space, but test in RGB. Can be 'YCbCr' or 'RGB'
'testpath' : [r'C:\testfolder'], # path to the folder in which test images exist. Can be more than one.
'traindir': r'C:\trainfolder', # path to the folder in which training images exist.
'upscaleimage':False, # Our model is not going to upscale the given low resolution image. Will use as is.
'valdir': '' , # no any validation folder given. Model will not implement validation during training.
'workingdir': 'C:\Results' , # all outputs to be written within this folder.
'weightpath':'C:\weights.h5' # load the weights from this file for training or testing.
}

#  a method returning a keras model
def build_model(self, testmode=False):

    if testmode:
        input_size = None
    else:
        input_size = self.inputsize

    input_shape = (input_size, input_size, self.channels)

    my_model = Sequential()
    my_model.add(Conv2D(64,(9,9), kernel_initializer='glorot_uniform', activation='relu',
                                    padding='valid', input_shape=input_shape))
    my_model.add(Conv2D(32, (1,1), kernel_initializer='glorot_uniform', activation='relu',
                                    padding='valid'))
    my_model.add(Conv2D(1,(5,5), kernel_initializer='glorot_uniform', activation='relu',
                                    padding='valid'))
    my_model.compile(Adam(self.lrate, self.decay), loss=losses.mean_squared_error)
    return my_model
```

3.2. Sample Model File 2

```
#  sample_model_2.py file. Returns an autoencoder model
from keras import losses
from keras.layers import Input
from keras.layers.convolutional import Conv2D, UpSampling2D
from keras.models import Model
from keras.optimizers import Adam
from os.path import basename

settings = \
{
'activation':relu, # activation function in layers is 'relu'
'augment':[90, 180], # can be any combination of [90,180,270, 'flipud', 'fliplr', 'flipudlr' ], or []
'backend': 'tensorflow', # keras is going to use tensorflow framework in processing.
'batchsize':256, # number of batches
'channels':1, # color channels to be used in training . Only one channel in this case
'colormode':'YCbCr', # the color space is YCbCr. 'YCbCr' or 'RGB'
```

```
'crop':0,  # do not crop from borders of images.
'crop_test':0,  # do not crop from borders of images in tests.
'decay':1e-6,   # learning rate decay.
'espatience' : 5,  # stop after 5 epochs if the performance of the model has not improved.
'epoch':10,  # train the model for total 10 passes on training data.
'inputsize':33, #  size of input image patches is 33x33.
'interp_compare': 'lanczos', #  compare the performance of the model with lanczos interpolation
'lrate':0.001,
'lrpatience': 3, # number of epochs to wait before reducing the learning rate.
'lrfactor' : 0.5, # the ratio of decrease in learning rate value.
'minimumlrate' : 1e-7,  # learning rate can be reduced down to a maximum of this value.
'modelname':basename(__file__).split('.')[0], # modelname is the same as the name of this file.
'metrics': ['PSNR', 'SSIM'], # the model name is the same as the name of this file.
'normalize': ['standard', 53.28, 40.732], # apply standardization to input images (mean, std)
'outputdir':'', # sub directories automatically created.
'scale':4,  #  magnification factor is 4.
'stride':11, # give a step of 11 pixels apart between patches while cropping them from images for training.
'target_channels':1, # color channels to be used in tests . Only one channel in this case
'target_cmode':'RGB', # 'YCbCr' or 'RGB'
'testpath' : [r'c:\test_images'], # path to the folder in which test images are. Can be more than one.
'traindir': r'c:\training_images', # path to the folder in which training images are.
'upscaleimage':False, # The model is going to upscale the given low resolution image. Thus, do not upscale.
'valdir': r'c:\validation_images' , # path to the folder in which validation images are.
'workingdir': '', # path to the working directory. All outputs to be produced within this directory
'weightpath':'', # path to model weights either for training to start with, or for test.
}

#  a method returning an autoencoder model
def build_model(self, testmode=False):

        if testmode:
         input_size = None
        else:
         input_size = self.inputsize

    # encoder
    input_img = Input(shape=(input_size, input_size, self.channels))
    x = Conv2D(32, (3, 3), activation=self.activation, padding='same')(input_img)
    x = Conv2D(16, (1, 1), activation= self.activation, padding='same')(x)
    x = Conv2D(8, (3, 3), activation= self.activation, padding='same')(x)
    x = Conv2D(3, (1, 1), activation= self.activation, padding='same')(x)

    # decoder

    x = UpSampling2D((self.scale, self.scale))(x)  # upscale by the scale factor

    x = Conv2D(8, (3, 3), activation= self.activation, padding='same')(x)
    x = Conv2D(16, (1, 1), activation= self.activation, padding='same')(x)
    x = Conv2D(32, (3, 3), activation= self.activation, padding='same')(x)
    decoded = Conv2D(self.channels, (3, 3), activation= self.activation, padding='same')(x)
    autoencoder = Model(input_img, decoded)
    autoencoder.compile(Adam(self.lrate, self.decay), loss=losses.mean_squared_error)
    return autoencoder
```

4. Interacting with Command Prompt

We provide a various of examples in interacting the program in this section. We consider in using sample model files, **sample_model_1** and **sample_model_2**, given in previous sections. We start with very simple example.

Before diving into details, we note again that even all parameters of model is determined within the dictionary “settings” in a model file, the command arguments supplied in the command prompt overrides them. For example, let assume that, as in the mode **sample_model_1py**, model name is determined as follows:

```
'modelname': 'my_model_1', # the model name is 'my_model_1'.
```

But we give model name as **“new_model_name”** in the command prompt as follows:

```
python -m DeepSR.DeepSR --modelfile "sample_model_1.py" --modelname new_model_name
```

Thus, the actual name of model model is now **"new_model_name"**, since command prompt arguments overrides.

The path of the DeepSR tool should be added to systems "PATH' environment for direct use the tool from command prompt. Otherwise, the full path of the DeepSR.py has to be typed in the command prompt.

4.1. Training

In order to train models, the only argument in command prompt is "—train" as long as all required parameters given in the model file within the dictionary "settings". For example, we just train the model **sample_model_1** with the settings given in the dictionary "settings" within the model file.

```
python -m DeepSR.DeepSR --modelfile "sample_model_1.py" --train
```

To do test also after the model is trained;

```
python -m DeepSR.DeepSR --modelfile "sample_model_1.py" --train --test
```

In case one needs to start the training procedure loading the network with the weights obtained from prior trainings, he/she can use the command prompt like this:

```
python -m DeepSR.DeepSR --modelfile "sample_model_1.py" --train --weightpath
'c:\previous_training\precedent_weights.h5'
```

or provide the path to existing weight in the dictionary like this:

```
'weightpath':' c:\previous_training\precedent_weights.h5' # we provided the path to
existing weights.
```

Let us say that we want to train our model with the previous weights for 10 epochs instead of 5 as set in the model file:

```
python -m DeepSR.DeepSR --modelfile "sample_model_1.py" --train --weightpath
'c:\previous_training\precedent_weights.h5' --epoch 5
```

Following command starts training for scale factor 8, rather than 2 as written in the settings:

```
python -m DeepSR.DeepSR --modelfile "sample_model_1.py" --train --scale 8
```

The following standardize images before feeding them to the model in training. The mean and standard deviation are calculated from the entire training images.

```
python -m DeepSR.DeepSR --modelfile "sample_model_1.py" --train --normalize standard
```

The same can be done by changing the "normalize" parameter in the model file:

```
python -m DeepSR.DeepSR --modelfile "sample_model_1.py" --train --normalize standard
```

Standardize images with mean 67.7 and standard deviation 13.5 (do not calculate the mean and the deviation from the training set), shuffle images before feeding to model and do test after training has done:

```
python -m DeepSR.DeepSR --modelfile "sample_model_1.py" --train --test --normalize
standard 67.7  13.5 --shuffle
```

There is no validation path given in the model file. Let us provide the path in which validation images exist for activating validation procedure in the training. The following validates the model's performance with images given path after each epochs:

```
python -m DeepSR.DeepSR --modelfile "sample_model_1.py" --train  --valdir
"c:\validation_images"
```

Crop image patches from training images with 25x25 pixels with stride 5 in YCbCr color space, with a single channel only:

```
python -m DeepSR.DeepSR --modelfile "sample_model_1.py" --train  --inputsize 25 --stride 5
--colormode YCbCr  --channels 1
```

4.2. Test

The basic command argument to start the test procedure is "—test", if all settings are given in the model file appropriately. For example, the following command implements test procedure with the settings given in the model file "sample_model_2.py"

```
python -m DeepSR.DeepSR --modelfile "sample_model_2.py" --test
```

According to the model file, the test will be done with the following settings

- test files are in the path : "c:\test_images"
- metrics to be used : PSNR, SSIM
- test will be done in RGB color space (target_cmode:'RGB'),
- since the value of the key "weightpath" is empty, weight file to be searched in the following path and the test to be implemented for each weight file in this path:

Actually, two image quality metrics (PSNR, SSIM) are determined in the model file to use. The following implements the above test by measuring the image qualities with the following metrics; VIF, GMSD, PSNR, SSIM, MSE MAD:

```
python -m DeepSR.DeepSR --modelfile "sample_model_2.py" --test --metrics VIF, GMSD, PSNR,
SSIM, MSE, MAD
```

The following code is the same as above but, only one channel is compared and images now are in YCbCr color space.

```
python -m DeepSR.DeepSR --modelfile "sample_model_2.py" --test --metrics VIF, GMSD, PSNR,
SSIM, MSE, MAD -target_channels 1 -target_cmode "YCbCr"
```

The key "ALL" can also be used for indicating all image quality metrics during test.

```
python -m DeepSR.DeepSR --modelfile "sample_model_2.py" --test --metrics ALL
```

The following commences the test procedure only for a specific weight file ("model_weights.h5" in this case) in the given path:

```
python -m DeepSR.DeepSR --modelfile "sample_model_2.py" --test --weightpath
"c:\weights_folder\model_weights.h5"
```

The above test uses only PSNR and SSIM measures, since only these two metrics determined in the model file. In fact, even nothing was determined in the model file or in the command prompt, though, they would be used since they are the default metrics.

Do the same test in YCbCr color space:

```
python -m DeepSR.DeepSR --modelfile "sample_model_2.py" --test --weightpath
"c:\weights_folder\model_weights.h5" --target_cmode YCbCr
```

The following implements the above test in YCbCr color space by cropping the images by 10 pixels from each border:

```
python -m DeepSR.DeepSR --modelfile "sample_model_2.py" --test --weightpath
"c:\weights_folder\model_weights.h5" --target_cmode YCbCr --crop_test 10
```

Please note that the test still to be done with PSNR and SSIM measures.

4.3. Preprocessing

The following operation may be implemented by DeepSR: augmenting, normalization, noising.

Augmentation is used only in training. In order to augment images, use the command parameter (or as a member of DeepSR object) `augment` is used. If there is no any augmentation type is specified, Augmentation will not be applied.

In the following example, the images are augmented by rotation them 90, 180, 270 degrees.

```
python -m DeepSR.DeepSR --modelfile "sample_model_1.py" --train --augment 90 180 270
```

In the following example, the images are augmented by flipping left-right, up-down, and left-right and up-down together.

```
python -m DeepSR.DeepSR --modelfile "sample_model_1.py" --train --augment fliplr flipul
flipudlr
```

The following types of normalization operations offered by DeepSR: `divide`, `mean`, `minmax`, `standard`.

Let us train "sample_model_1" with different types of normalizations. First, we normalize images by dividing a particular value. Below images are being normalized by dividing their intensity values by 127.0. Please note that we still augment images. Images are first being augmented, and then, normalization procedure is applied to entire original and augmented images.

```
python -m DeepSR.DeepSR --modelfile "sample_model_1.py" --train --normalization divide
127.0 --augment fliplr flipul flipudlr
```

In the following standardization applied to images. The mean and the standard deviation are calculated from training set of images.

```
python -m DeepSR.DeepSR --modelfile "sample_model_1.py" --train --normalization standard
whole --augment fliplr flipul flipudlr
```

Each images (even if they are augmented images) are standardized individually with the values calculated from themselves.

```
python -m DeepSR.DeepSR --modelfile "sample_model_1.py" --train --normalization standard
single --augment fliplr flipul flipudlr
```

Standardization is applied with the values given by user manually in below code. The mean is 2.23 and the standard deviation is 7.85 in this case.

```
python -m DeepSR.DeepSR --modelfile "sample_model_1.py" --train --normalization standard
2.23 7.85 --augment fliplr flipul flipudlr
```

The following is the min-max normalization with minimum and maximum values 0 and 1 respectively.

```
python -m DeepSR.DeepSR --modelfile "sample_model_1.py" --train --normalization minmax
--augment fliplr flipul flipudlr
```

The minimum and maximum values can also be given by users. For example, the minimum is -1 and the maximum is 2.

```
python -m DeepSR.DeepSR --modelfile "sample_model_1.py" --train --normalization minmax -1
2       --augment fliplr flipul flipudlr
```

Another normalization type: mean normalization. The following code normalizes images by subtracting their own mean values from themselves.

```
python -m DeepSR.DeepSR --modelfile "sample_model_1.py" --train --normalization mean
single  --augment fliplr flipul flipudlr
```

Or, we can subtract the mean value calculated from the entire training set of images.

```
python -m DeepSR.DeepSR --modelfile "sample_model_1.py" --train --normalization mean whole
--augment fliplr flipul flipudlr
```

Or, user provides the mean value. It is 125.0 in the following case.

```
python -m DeepSR.DeepSR --modelfile "sample_model_1.py" --train --normalization mean 125.0
--augment fliplr flipul flipudlr
```

We may also add Gaussian noise to images after they are downscaled as in the pipeline of Super Resolution. Like this.

```
python -m DeepSR.DeepSR --modelfile "sample_model_1.py" --train --normalization mean 125.0
--augment fliplr flipul flipudlr -noise 0 0.01
```

The images in the training set are first augmented, than be normalized by subtracting the mean value of 125.0 and, the Gaussian noise with zero mean and variance of 0.01 is added to them in the above case.

4.4. Misceallenous

To shut down the computer after all jobs has been accomplished, i.e., we instruct the program to train the model, test it afterwards, and shut down the computer:

```
python -m DeepSR.DeepSR --modelfile "sample_model_2.py" --test --train --shutdown
```

The following command visualize the outputs of each layer while testing:

```
python -m DeepSR.DeepSR --modelfile "sample_model_2.py" --test --layeroutputs
```

The following command visualize the outputs of each layer and saves them as images in files with layer names while testing:

```
python -m DeepSR.DeepSR --modelfile "sample_model_2.py" --test --layeroutputs --saveimages
```

One can also visualize layer's weights in the test procedure:

```
python -m DeepSR.DeepSR --modelfile "sample_model_2.py" --test --layerweights
```

Add `--saveimages` to save layer weights as images in files during test procedure:

```
python -m DeepSR.DeepSR --modelfile "sample_model_2.py" --test --layerweights --saveimages
```

In order to save the output HR images of model during testing:

```
python -m DeepSR.DeepSR --modelfile "sample_model_2.py" --test --saveimages
```

The following command results in the test by saving the model's architecture in an image file:

```
python -m DeepSR.DeepSR --modelfile "sample_model_2.py" --test --plotmodel
```

The same task can be done by calling the `plot_model()` method of DeepSR class. The architecture is saved in the just under the path stored by the `outputdir` member of DeepSR object.

Let us name our model as "Our_model":

```
python -m DeepSR.DeepSR --modelfile "sample_model_2.py" --modelname Our_model
```

There is also an important command argument to maintain (almost) the same initial weights all the time for models before starting training. This ensures users to compare models' performances by training the models with the same initial conditions. This functionality is provided with the `seed` argument. Now, we train the **sample_model_2** by initializing weights with random numbers generated by seed of 19.

```
python -m DeepSR.DeepSR --modelfile "sample_model_2.py" --train --seed 19
```

By this way, one can train different models with feeding them the same initial random numbers, and check the results to see which model offers better results.

5. Using DeepSR as Class Object

DeepSR can be used as a Python object from another program as well. The key point here is, that all parameters along with settings must be assigned to the object before doing any operation with it. User need to designate each parameter/instruction as class member or methods by manually. On the other hand, there is another way of doing this procedure.

DeepSR object takes only one parameter in class construction phase for setting it up with parameters: `args` (arguments or setting, in another word). The argument, `args` must be a dictionary with key-value pairs similar to the dictionary, `settings` in model files. The parameters/instructions in `args` can be taken from a file programmatically or can be written in actual python code where the actual program is being coded. It is up to the user.

DeepSR can be constructed without providing any settings in `args`. The class will have no any members or methods in such case. This user is informed about this situation in command prompt. However, each parameter of the program (and also build method) must still be designated in the class as members before they are being used for any operations.

The following code snippet is an example for creating DeepSR object from Python scripts by assigning settings to class in construction stage of the class.

```
from DeepSR import DeepSR
from os.path import basename

settings = \
    {
        'augment': [90, 180],  # can be any combination of [90,180,270, 'flipud', 'fliplr', 'flipudlr' ], or []

        'backend': 'tensorflow', # keras is going to use tensorflow framework in processing.
        'batchsize':9,  # number of batches
        'channels': 1,  # color channels to be used in training . Only one channel in this case
        'colormode': 'YCbCr',  # the color space is YCbCr. 'YCbCr' or 'RGB'
        'crop': 0,  # do not crop from borders of images.
        'crop_test': 0,  # do not crop from borders of images in tests.
        'decay': 1e-6,  # learning rate decay.
        'espatience': 5,  # stop after 5 epochs if the performance of the model has not improved.
        'epoch': 2,  # train the model for total 10 passes on training data.
        'inputsize': 33,  # size of input image patches is 33x33.
        'lrate': 0.001,
        'lrpatience': 3,  # number of epochs to wait before reducing the learning rate.
        'lrfactor': 0.5,  # the ratio of decrease in learning rate value.
        'minimumlrate': 1e-7,  # learning rate can be reduced down to a maximum of this value.
        'modelname': basename(__file__).split('.')[0],  # modelname is the same as the name of this file.
        'metrics': ['PSNR', 'SSIM'],  # the model name is the same as the name of this file.
        'normalization': ['standard', 53.28, 40.732],  # apply standardization to input images (mean, std)
```

```
        'outputdir': '',  # sub directories automatically created.
        'scale': 4,  # magnification factor is 4.
        'stride': 11,  # give a step of 11 pixels apart between patches while cropping them from images for trai
ning.
        'target_channels': 1,  # color channels to be used in tests . Only one channel in this case
        'target_cmode': 'RGB',  # 'YCbCr' or 'RGB'
        'testpath': [r'C:\test_images'],  # path to the folder in which test images are. Can be more than one.
        'traindir': r'C:\training_images',  # path to the folder in which training images are.
        'upscaleimage': False,  # The model is going to upscale the given low resolution image.
        'valdir': r'',  # path to the folder in which validation images are.
        'workingdir': r'',  # path to the working directory. All outputs to be produced within this directory
        'weightpath': '',  # path to model weights either for training to start with, or for test.
    }

DSR = DeepSR.DeepSR(settings)  # instance of DeepSR object without the build_model method.
```

At this point, DeepSR object was created with parameters but without the method “build_model”. Therefore this method must be declared in the class object in order to compose a deep learning model. User can write this method in the same script and assign it to the class by calling the member method of the DeepSR object: set_model. In the following code snippet, a sample method for constructing a deep learning model defined and assigned to DeepSR object by the member method.

```
from keras import  losses
from keras.layers import Input
from keras.layers.convolutional import Conv2D, UpSampling2D
from keras.models import  Model
from keras.optimizers import Adam

#  a method returning an autoencoder model
def build_model(self, testmode=False):
    if testmode:
        input_size = None

    else:
        input_size = self.inputsize

    # encoder
    input_img = Input(shape=(input_size, input_size, self.channels))
    x = Conv2D(32, (3, 3), activation='relu', padding='same')(input_img)
    x = Conv2D(16, (1, 1), activation='relu', padding='same')(x)
    x = Conv2D(8, (3, 3), activation='relu', padding='same')(x)
    x = Conv2D(3, (1, 1), activation='relu', padding='same')(x)

    # decoder
    x = UpSampling2D((self.scale, self.scale))(x)  # upscale by the scale factor
    x = Conv2D(8, (3, 3), activation='relu', padding='same')(x)
    x = Conv2D(16, (1, 1), activation='relu', padding='same')(x)
    x = Conv2D(32, (3, 3), activation='relu', padding='same')(x)
    decoded = Conv2D(self.channels, (3, 3), activation='relu', padding='same')(x)
    autoencoder = Model(input_img, decoded)
    autoencoder.compile(Adam(self.lrate, self.decay), loss=losses.mean_squared_error)
    return autoencoder
```

Now, the class object is ready for further processes. A training and test procedure is being implemented below.

```
DSR.set_model(build_model)  # set build_model function to compose a DL model in the class.

DSR.epoch =1 # training will be implemented for 1 time instead of 10 as defined in settings.

DSR.train()  # training procedure.

# the performance of the model is evaluated below.
DSR.test(testpath=DSR.testpath, weightpath=DSR.weightpath, saveimages=False, plot=False)
```

To wrap all together, whole code is below.

```
import DeepSR
from os.path import basename

settings = \
    {
        'augment': [90, 180],  # can be any combination of [90,180,270, 'flipud', 'fliplr', 'flipudlr' ], or []
```

```
        'backend': 'tensorflow', # keras is going to use tensorflow framework in processing.
        'batchsize':9,  # number of batches
        'channels': 1,  # color channels to be used in training . Only one channel in this case
        'colormode': 'YCbCr',  # the color space is YCbCr. 'YCbCr' or 'RGB'
        'crop': 0,  # do not crop from borders of images.
        'crop_test': 0,  # do not crop from borders of images in tests.
        'decay': 1e-6,  # learning rate decay.
        'espatience': 5,  # stop after 5 epochs if the performance of the model has not improved.
        'epoch': 2,  # train the model for total 10 passes on training data.
        'inputsize': 33,  # size of input image patches is 33x33.
        'lrate': 0.001,
        'lrpatience': 3,  # number of epochs to wait before reducing the learning rate.
        'lrfactor': 0.5,  # the ratio of decrease in learning rate value.
        'minimumlrate': 1e-7,  # learning rate can be reduced down to a maximum of this value.
        'modelname': basename(__file__).split('.')[0],  # modelname is the same as the name of this file.
        'metrics': ['PSNR', 'SSIM'],  # the model name is the same as the name of this file.
        'normalization': ['standard', 53.28, 40.732],  # apply standardization to input images (mean, std)
        'outputdir': '',  # sub directories automatically created.
        'scale': 4,  # magnification factor is 4.
        'stride': 11,  # give a step of 11 pixels apart between patches while cropping them from images for trai
ning.
        'target_channels': 1,  # color channels to be used in tests . Only one channel in this case
        'target_cmode': 'RGB',  # 'YCbCr' or 'RGB'
        'testpath': [r'C:\test_images'],  # path to the folder in which test images are. Can be more than one.
        'traindir': r'C:\training_images',  # path to the folder in which training images are.
        'upscaleimage': False,  # The model is going to upscale the given low resolution image.
        'valdir': r'',  # path to the folder in which validation images are.
        'workingdir': r'',  # path to the working directory. All outputs to be produced within this directory
        'weightpath': '',  # path to model weights either for training to start with, or for test.
    }

DSR = DeepSR.DeepSR(settings)  # instance of DeepSR object without the build_model method.

from keras import  losses
from keras.layers import Input
from keras.layers.convolutional import Conv2D, UpSampling2D
from keras.models import  Model
from keras.optimizers import Adam

#  a method returning an autoencoder model
def build_model(self, testmode=False):
    if testmode:
        input_size = None

    else:
        input_size = self.inputsize

    # encoder
    input_img = Input(shape=(input_size, input_size, self.channels))
    x = Conv2D(32, (3, 3), activation='relu', padding='same')(input_img)
    x = Conv2D(16, (1, 1), activation='relu', padding='same')(x)
    x = Conv2D(8, (3, 3), activation='relu', padding='same')(x)
    x = Conv2D(3, (1, 1), activation='relu', padding='same')(x)

    # decoder
    x = UpSampling2D((self.scale, self.scale))(x)  # upscale by the scale factor
    x = Conv2D(8, (3, 3), activation='relu', padding='same')(x)
    x = Conv2D(16, (1, 1), activation='relu', padding='same')(x)
    x = Conv2D(32, (3, 3), activation='relu', padding='same')(x)
    decoded = Conv2D(self.channels, (3, 3), activation='relu', padding='same')(x)
    autoencoder = Model(input_img, decoded)
    autoencoder.compile(Adam(self.lrate, self.decay), loss=losses.mean_squared_error)
    return autoencoder

DSR.set_model(build_model)  # set build_model function to compose a DL model in the class.

DSR.epoch =1 # model shall be trained only for 1 epoch, instead of 10 as defined in settings.

DSR.train()  # training procedure.

# evaluate the performance of the model.
DSR.test(testpath=DSR.testpath, weightpath=DSR.weightpath, saveimages=False, plot=False)
```

After the test has been done, the results of the evaluation are saved by DeepSR in an Excel file. This file provides scores of each image quality metrics designated to the class (in the parameter, 'metrics') calculated from each individual test images, and the averages of those metrics over entire test images.

6. References

[1] F. Chollet and others, "Keras." GitHub, 2015.

[2] M. Abadi *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th ${$USENIX$}$ Symposium on Operating Systems Design and Implementation (${$OSDI$}$ 16)*, 2016, pp. 265–283.

[3] Theano Development Team, "Theano: A {Python} framework for fast computation of mathematical expressions," *arXiv e-prints*, vol. abs/1605.0, May 2016.

[4] Y. Jia *et al.*, "Caffe: Convolutional Architecture for Fast Feature Embedding," *arXiv preprint arXiv:1408.5093*, 2014.

[5] F. Seide and A. Agarwal, "CNTK: Microsoft's open-source deep-learning toolkit," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2016, p. 2135.

[6] A. Vedaldi and K. Lenc, "MatConvNet -- Convolutional Neural Networks for MATLAB," in *Proceeding of the {ACM} Int. Conf. on Multimedia*, 2015.