Users' Guide for DeepSR

An Open Source Deep Learning Framework for Super-Resolution

Release 1.0

Hakan Temiz

Hasan Şakir Bilge

May 2019

Contents

Introduction

As far as the deep learning techniques are concerned, there are two main approaches to take. Either we upsample the input image before feeding it to the network [8, 14, 3, 9, 15] (through a fast but rough method, e.g., interpolation based methods), or we simply pass the low-resolution input to the network and let it learn the optimal upsampling during training

We introduce an open source framework that eases the entire processes of the super-resolution (SR) problem in terms of deep learning algorithms. The proposed framework makes it very simple to design and build DL models, and enables researchers to easily and rapidly perform entire steps of the task of SR such as training and testing with parametrized interface. The main goal of the proposed framework is to empower researchers to focus on their studies while pursuing successful DL algorithms for the task of SR, saving them from the workloads of programming, implementing and testing. It offers several ways to use the framework, such as using the command line, using the DeepSR object in another program, or using batch files to automate successive multiple jobs. The proposed framework governs each steps in the workflow of SR task, such as pre-processing, augmentation, normalization, training, post-processing, and testing, in such a simple, easy and fast way that there would remain only a very small amount of work to researchers to accomplish their experiments. Moreover, multiple experiments may also be implemented successively by using batch scripts. It comes with a wide range of tools as well that facilitate particular operations specific to SR. We think that the proposed framework will be extremely helpful tool for researchers in studies of SR. We also encourage researchers to contribute the proposed framework to make it better

DeepSR is provided as open source framework, 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. It is available at http://github .

Main goals of the proposed framework are ensuring a way of fast prototyping and providing a basis for researcher and/or developers eliminating the need of commencing whole job from the scratch, or the need of adapting existing program to new designs and implementations. It saves researchers from dealing with the details, and allows them to focus on the research topic. It also offers additional useful features that may facilitate the entire process of SR workflow.

The SR task requires some special pre and/or post processing operations, along with some particular regularizations before and after data is processed by DL models. After the data was prepared with SR-specific operations for training or testing, a DL model designed using Keras API processes it, and then, the output of the model post processed to be suitable for further SR-specific operations. The large part of the job is handled by DeepSR. The proposed framework utilizes Keras only for training of DL models, or getting the result images of the models either for testing or prediction. Keras is capable of running on top of following prominent DL frameworks: TensorFlow, Theano and CTK. It transforms DL models designed by coding with its API into the models of these frameworks. And thus, our framework is capable of running models on both CPUs or GPUs seamlessly by leveraging these frameworks.

All that needs to be done is to build up a DL model using the API of Keras within a model file (a python file), and then, to provide the program the relevant information and/or instructions. For enabling the framework to accomplish the rest job, all parameters can be given to the program in any of following ways: (i) in a Python dictionary object in the same model file, where the DL model is coded, (ii) directly calling the related member method of the DeepSR class or setting the related parameter of the class, and (iii) by supplementing parameters in a 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 epoch, number of batches, etc., for any task (i.e., training, testing) 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 training methods. Representing the layer weights graphically, or yielding the outputs of each layers of models. It is also capable of evaluating the performance of models by a number of image quality assessment metrics out of the box.

1. Description of Methods

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 (Chollet and others, 2015), Tensorflow (Abadi et al., 2016), Theano (Theano Development Team, 2016), Caffee (Jia et al., 2014), and MatconvNet (Vedaldi and Lenc, 2015).

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 struggling 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 framework facilitating researchers deal with the challenges mentioned above.

Our framework, DeepSR, maintains the reusability of program code and rapid prototyping, in such a manner that is saving time and requiring very little human effort. It empowers researchers to focus on their research providing a parametrized, neat and simple interface. The proposed framework handles all stages of the work. It consists of many useful components and features that researchers may need during the process. Besides, it maintains also all pipelines of the work automatically and very efficiently. 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.

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

- Easy and fast prototyping, development and implementation interface.
- Subject-specific pre and/or post processing procedures and data augmentation tailored particularly for SR problem.
- Capability of performing consecutive training, testing and other transactions through batch scripts.
- Ability of evaluation the success of models by a number of image quality assessment metrics.

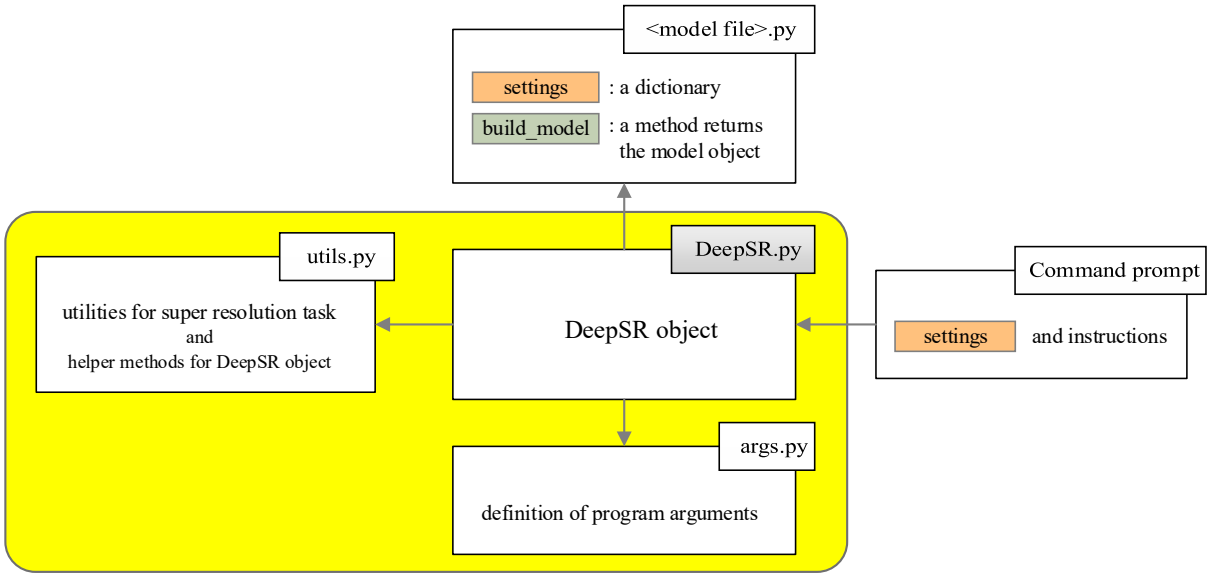
2. DeepSR

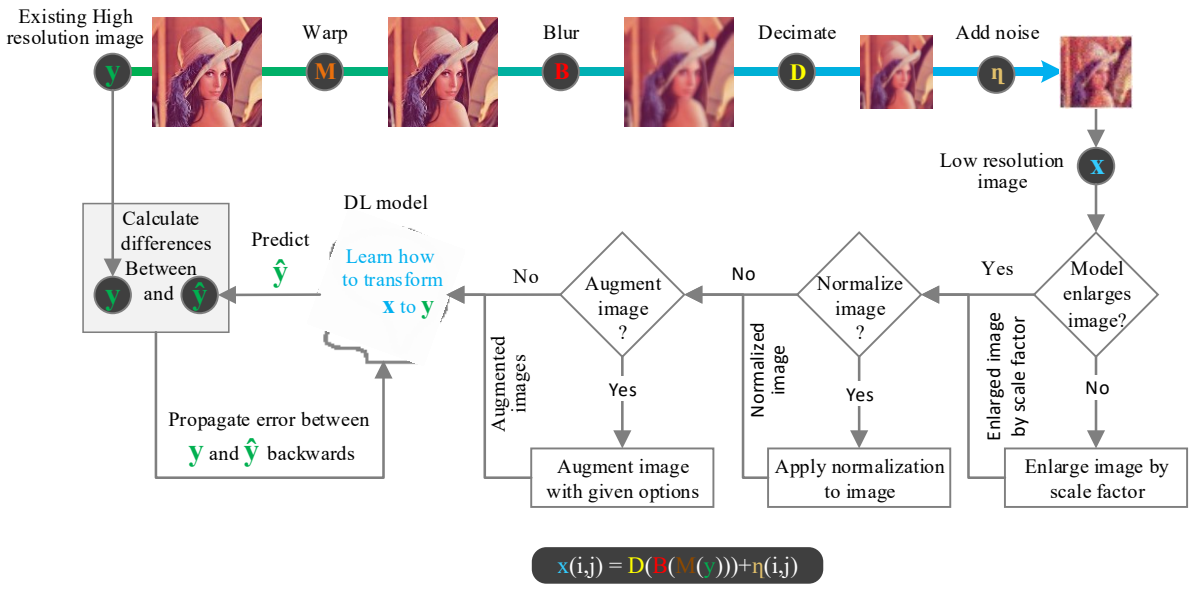
2.1. Program Structure

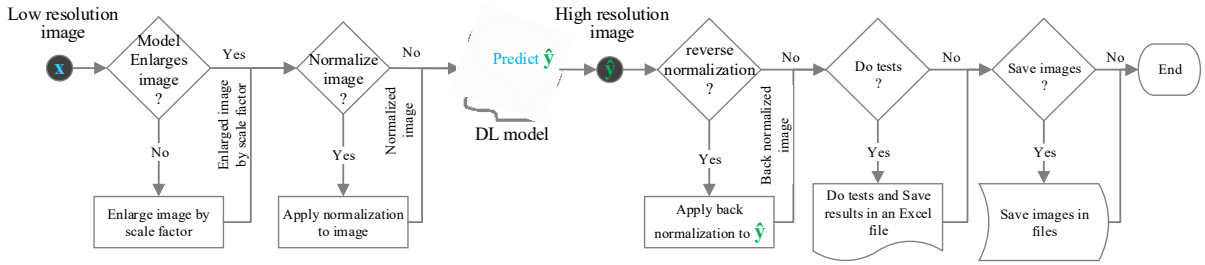
DeepSR fundamentally consists of three Python files: DeepSR.py, args.py, and utils.py. The DeepSR object is developed in the file DeepSR.py. It is the core python module called first for running the program. All parameters set in command prompt are defined in the file, args.py. The module, utils.py accompanies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

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.







2.2. Preprocessing, Data Augmentation and Normalization

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 output single channel HR image should be prepared accordingly for the purpose of training, even though images have more color channels. The proposed framework 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 outputs the 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 of the program just sets a parameter to indicate whether his/her model upscales the input LR image or not, and choses the interpolation method with another parameter. The rest of the work accomplished by the framework, 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 framework, 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 framework. Followings are of some of those operations: min-max normalization, standardization, dividing by a certain value (i.e., maximum intensity value).

2.3. Training

There are several different types of training processes made available by the framework to researchers. Researchers may choose to train models in either online or offline fashion, using a single or multiple batches. In the case of online training, DeepSR uses the images in a folder in the training of models. In the case of offline training, it uses an already prepared dataset (a .h5 file) if exist, or prepares 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 allows the training of the model to continue with the model weights yielded from earlier trainings.

2.4. Testing and Predicting

Another appealing property of the proposed framework is that it automates all testing and predicting works very significantly. The framework automatically prepares ground-truth (reference) images and 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 the model. The evaluation results are composed and saved in an Excel file. All tests are done for a single weight file if the path belongs to a

file or multiple weight files if the given path belongs to a folder in which multiple files of weights are. Every test results of IQM over each test data (image) for each (in cases multiple weight files provided) weights of DL model is given in a tabular format in an Excel work sheet, along with another sheet of mean values of each IQM resulted in over test data (images). As easily seen from above, the testing process is automated so that all the needs of the researchers are met.

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 framework during the development of framework. Consequently, a several typical post processing operations that may needed are at the disposal of researchers for their use.

2.6. Program Parameters

Argument Name	Explanation
augment	Augment options. The followings may be used: 90, 180, 270, flipud, fliplr, flipudlr. **Usage**: `--augment 90 flipud fliplr # augment by rotating 90 degrees and # flipping the image up-down and # left-right`
backend	Used for the selection of the Keras backend: theano, tensorflow. **Usage**: `--backend tensorflow # tensorflow is backend`
batchsize	The batch size defines the number of training examples given to the network before calculatin of the error of the network and to propagate the error back through the network. **Usage**: `--batchsize 256`
cchannels	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**: `--cchannels 3`
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 program is capable of using any of the following interpolation methods: **bilinear**, **bicubic**, **nearest** and **lanczos**. The default is bicubic. Use the keyword **same** to use the same interpolation method(s) as given with the command parameter 'interpmethod'.
	Usage:
	`--decimation bicubic # bicubic interpolation is used in decimation.`
	`--decimation same # the same interpolation method(s) given in # 'interpmethod' command argument to be used # for decimation.`

epoch	The total number of complete pass of training through entire training data. **Usage:** `--epoch 10 # train the model for 10 epochs.`
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`

interpmethod	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:
	`--interpmethod bicubic # evaluate the performance of the bicubic # interpolation method on test data.`
	`--interpmethod all # compare the performance of the model with # all interpolation method(s).`
	`--interpmethod same # the interpolation method is being used now # for decimation procedure also. So, the # interpolation method for decimation given # in the argument 'decimation' is now # overriden.`

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`
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**[1] library), **MAD**, **NIQE**, **BRISQUE** (from **scikit-image**[2] library), **MSSSIM**, **ERGAS**, **RASE**, **SAM**, **SCC**, **UQI** (from **sewar**[3] library), **SNR**, **BSNR**, **PAMSE**, **GMSD** (from **sporco**[4] library) Default is **PSNR**, **SSIM**. **Usage:** `--metrics PSNR SSIM MSSSIM  # use this measures for test`
model	Used to locate 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 may also contain a Python dictionary **settings** for providing any of parameters/settings. **Usage:** `--model "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.`

normalize	Normalization method to be applied to the input images. Can be any of the following:
	divide, minmax, std, mean.
	divide is used to divide the intensity values of image by a certain value.
	minmax adjusts the intensity value between a certain limits. For example between 0 to 1.
	std 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 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 standard  # do standardization`
	`--normalize standard 1 3  # standardize with mean 1, stddev 3`
	`--normalize mean  # subtract its mean value from the image`
	`--normalize mean 111  # subtract the value of 111 from image.`

normalizeback	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	The ground truth image will also normalized, if this parameter is set.
	Usage:
	`--normalizeground  # ground-truth image is normalized with the same`
	`# normalization method as applied to input image.`

outputdir	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', or within the current folder the program runs in. Sub folders for the outputs are created based on the scale factor. **Usage:**

[1] http://www.scikit-video.org
[2] https://scikit-image.org/
[3] https://pypi.org/project/sewar/
[4] https://pypi.org/project/sporco/

	`--outputdir "C:\example_model\output"`
plotimage	Plots the output image yielded by the model. Does not take any additional parameter **Usage:** `--getimage`
plotmodel	Plots the architecture of the model in an image file. Used with the **test** parameter. **Usage:** `--test --plotmodel # plots the model layout while performing test.`
predict	Returns the result image of the deep network with some evaluation scores. Used to get result image without implementing a test procedure. **Usage:** `--predict`
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 # after test has 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_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	The test procedure will be performed if this parameter is set. **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	The command parameter to start the training procedure. If it is given with the test parameter, the training procedure is first performed, and then, the test is done.
	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 by the program with a given interpolation method before hanlding 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. **Usage:** `--workingdir "C:\working_folder"  # working folder for outputs.`

2.7. Documentation and Housing

We also provide the documentation of the proposed framework to the service of researchers along with very rich examples of use of the program. Referring to the documentation and examples of the proposed framework, researchers may easily accomplish their experiments with a simple and efficient way with little effort. We have put great effort into keeping the framework itself, its documentation and examples simple enough to make it easier to understand.

3. Model Files

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

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
'cchannels':3, # color channels to be used in tests . Three channels in this case
'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.
'earlystoppingpatience' : 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,
'lrateplateaupatience': 2, # number of epochs to wait before reducing the learning rate.
'lrateplateaufactor' : 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.
'normalize': ['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, #  maginification factor is 2.
'stride':17, # give a step of 17 pixels apart between patches while cropping them from images for training.
'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 = \
{
'augment':[90, 180], # can be any combination of [90,180,270, 'flipud', 'fliplr', 'flipudlr' ], or []
'backend': 'theano', # keras is going to use theano framework in processing.
'batchsize':256, # number of batches
'cchannels':1, # color channels to be used in tests . Only one channel in this case
'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.
```

```
'earlystoppingpatience' : 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.
'lrate':0.001,
'lrateplateaupatience': 3, # number of epochs to wait before reducing the learning rate.
'lrateplateaufactor' : 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,  #  maginification factor is 4.
'stride':11, # give a step of 11 pixels apart between patches while cropping them from images for training.
'target_cmode':'RGB', # 'YCbCr' or 'RGB'
'testpath' : [r'D:\test_images'], # path to the folder in which test images are. Can be more than one.
'traindir': r'D:\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='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
```

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 DeepSR.py --model "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.

4.1.1. 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 DeepSR.py --model "sample_model_1.py" --train
```

To do test also after the model is trained;

```
python DeepSR.py --model "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 DeepSR.py --model "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\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 DeepSR.py --model "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 DeepSR.py --model "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 DeepSR.py --model "sample_model_1.py" --train --normalize std
```

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

```
python DeepSR.py --model "sample_model_1.py" --train --normalize std
```

```
'normalize': ['standard'], # apply standardization to input images
```

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 DeepSR.py --model "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 DeepSR.py --model "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 DeepSR.py --model "sample_model_1.py" --train  -inputsize 25 --stride 5
--color YCbCr  --channels 1
```

4.1.2. 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 imlements test procedure with the settings given in the model file "sample_model_2.py"

```
python DeepSR.py --model "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 : "d:\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 DeepSR.py --model "sample_model_2.py" --test --metrics VIF, GMSD, PSNR,
SSIM, MSE, MAD
```

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

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

The following implements the above test by cropping the images by 10 pixels from each border:

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

please not that the test still to be done with PSNR and SSIM measures.

4.1.3. 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 DeepSR.py --model "sample_model_2.py" --test --train --shutdown
```

The following command saves all layer's outputs in separate files with layer names as images while testing:

```
python DeepSR.py --model "sample_model_2.py" --test --layeroutputs
```

One can also print layer's weights in in separate files with layer names as images during test procedure:

```
python DeepSR.py --model "sample_model_2.py" --test --layerweights
```

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

```
python DeepSR.py --model "sample_model_2.py" --test --saveimages
```

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

```
python DeepSR.py --model "sample_model_2.py" --test --plotmodel
```

Let us name our model as "Our_model":

```
python DeepSR.py --model "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. Let us train the **sample_model_2** by initializing weights with random numbers generated by seed of 19.

```
python DeepSR.py --model "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 an Object

DeepSR can be used an 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 using it. For example, in order to use the DeepSR object for training a model, model the followings have all ready been done:

- Build method must have already been defined.
- Training path
- Magnification (scale) factor