SoftwareX 21 (2023) 101261

Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx

Original software publication

DeepSR: A deep learning tool for image super resolution

Hakan Temiz

Computer Engineering Department, Artvin Çoruh University, Seyitler Yerleskesi, Artvin, 08000, Turkiye

ARTICLE INFO

Article history:
Received 23 May 2022
Received in revised form 18 October 2022
Accepted 5 November 2022

Keywords:
Software
Tool
Deep learning
Super-resolution
Image

ABSTRACT

An open source tool is introduced that provides a versatile environment to meet the needs of researchers in developing deep learning (DL) algorithms for single image super-resolution reconstruction (SISR). The processes of SISR were carefully studied, unified and integrated to create software that can be used by the community for any type of imaging method such as aerial, medical, optical, etc. DeepSR allows easy implementation of SISR application with rapidly prototyped DL models, and detailed reporting and recording of the results. The entire experiment can be done with simple command line scripts. It can be easily extended by user-defined metrics, augmentations, callbacks, etc.

Code metadata

Current code version	V0.0.80
Permanent link to code/repository used for this code version	https://github.com/ElsevierSoftwareX/SOFTX-D-22-00129
Permanent link to reproducible capsule	https://github.com/htemiz/DeepSR
Legal code license	MIT
Code versioning system used	git
Software code languages, tools and services used	Python
Compilation requirements, operating environments and dependencies	Python 3.6 and higher, NumPy, scipy, pandas, h5py, matplotlib, scikit-image, scikit-video, sporco, Pillow, sewar, sporco, openpyxl, TensorFlow, theano, keras, setuptools, GraphViz.
If available, link to developer documentation/manual	https://github.com/htemiz/DeepSR/tree/master/DeepSR/docs
Support email for questions	htemiz@artvin.edu.tr

1. Motivation and significance

Super-resolution (SR) aims at producing high-resolution (HR) images from low-resolution (LR) images. It is a very promising area of research in image processing applied to all types of imaging modalities such as daily life, aerial, medical, etc. It tries to increase the actual resolution of the given LR image while producing finer details with greater information and accuracy in representing the original HR image. In essence, the whole effort is to find a function $\xi(\cdot)$ that best matches the given LR image to the reference HR image. SR is denominated as an inverse problem as the source information (HR image) is estimated from the observed data (LR image). In estimation, there can be many different solutions to reconstruct the HR counterpart(s) of a given LR image. Therefore, it is an ill-posed problem to solve.

There are two main approaches to solving the problem: single-image super-resolution (SISR) and multi-image super-resolution (MISR). MISR aims to solve the problem by exploiting multiple images of the same scene or imaging medium, while SISR exploits a single image to reconstruct HR counterpart(s).

There are two cases in the implementation of SR, depending on the presence of the reference HR images. In the real-life case, both HR and LR images are available, whereas no HR images are present in the simulation case. In the latter case, LR images are synthetically generated from the existing ones considered HR. LR versions of existing HR images are simulated with the following formula:

$$y(i,j) = D(B(M(x))) + \eta(i,j) \tag{1}$$

E-mail address: htemiz@artvin.edu.tr.

https://doi.org/10.1016/j.softx.2022.101261

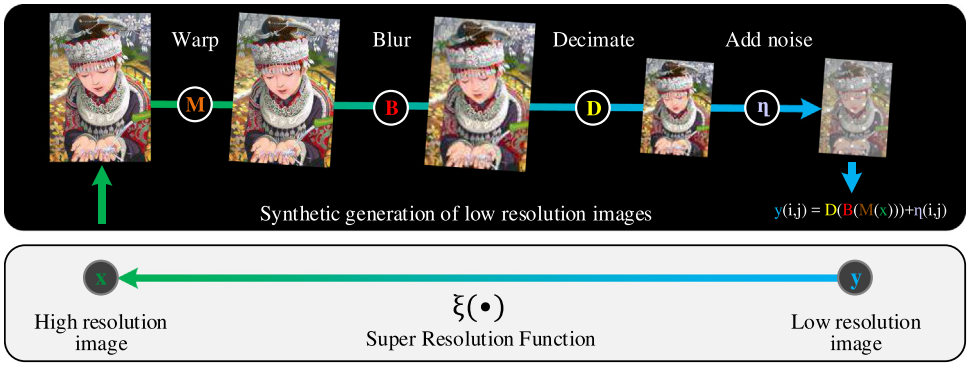

Fig. 1. Conceptual scheme of super-resolution. The dark area illustrates the generation of synthetic LR images. This procedure is applied only in the simulation scenario where no actual HR images are available.

where x and y are HR and LR images, respectively. M and B stand for warping and blurring functions, respectively, and D stands for downsampling. η is additive noise to represent various perturbations (e.g., due to transmission errors, atmospheric turbulence, sensor heat) in the observation. Several downsampling operators (bicubic, lanczos, etc.) are commonly involved to simulate LR counterparts. Some other warping functions and noises can also be applied accordingly. The conceptual scheme of SR is given in Fig. 1. The synthetic LR image generation process which is applied only in the simulation scenario is shown with a dark area in the figure.

Recent research has focused on SISR rather than MISR. These efforts have led to the development of many different approaches. Some of them are interpolation methods [1], frequency domain approaches [2], reconstruction-based methods [3], and learning-based methods [4–7]. Recently, deep learning (DL) algorithms significantly outperform other canonical methods. Since the first designed model SRCNN [8], many variations of successful architectures such as convolutional neural networks (CNNs), autoencoders (AEs), and generative adversarial networks (GANs) have been developed and continue to be developed intensively.

Various machine learning (ML) or DL frameworks such as Keras [9], TensorFlow [10], Theano [11], Caffee [12], CNTK [13], Torch [14], PyTorch [15], and MatconvNet [16] are already available to researchers to develop such algorithms. However, as they are just generic frameworks, they do not offer any built-in functionality to develop environments for SISR with DL algorithms. There is no program tailored to meet the needs of researchers in solving the SR problems. The research community would greatly benefit from having such a common tool that simplifies, integrates, and automates the implementation of SISR with DL algorithms in any imaging technique (medical, aerial, etc.).

The proposed tool, DeepSR [17], successfully meets the above-mentioned needs of researchers, and provides a versatile environment with rich feature sets. DeepSR is an open-source tool that is available under the terms of the MIT license and served in GitHub with a comprehensive program manual, source codes, binaries, samples, etc.

The entire methodology of the SISR was carefully devised, unified, and integrated into the program during development. It enables rapid prototyping of DL models and easy end-to-end execution of numerous SISR applications for different parameter sets and alternative models via simple command line scripts or batch files. Furthermore, it can be extended by user-defined functions in addition to built-in operations. It can be a common environment for the application of SISR with DL algorithms in any imaging technique.

2. Software description

DeepSR comes with many useful features in addition to the common procedures implemented in a typical SISR application. These features ensure the successful completion of the work from start to finish, with a comprehensive assessment, reporting, visualization, and logging capabilities. For example, it can do performance evaluation with 18 image quality measures (IQMs) and store the computed scores in detail. Or, it can visualize the training performances in a given metric (both in loss function and PSNR), predicted images, or the weights or outputs of the layers. In addition, DeepSR can be extended by easily incorporating user-defined functions into relevant processes, transactions, and computations.

DeepSR allows the simple and fast design of DL models. Implementation details for training, evaluation, and other operations are easily determined with simple parameters without writing any additional code. It offers a parametrized and easy scripting interface for user interaction. Besides, multiple tasks can also be performed through batch scripts. The conceptual and functional scheme of DeepSR is given in Fig. 2.

To construct DL models, DeepSR utilizes Keras' API which is capable to run on top of the following prominent DL frameworks: TensorFlow, Theano, and CNTK. However, it has been recently incorporated into the Tensorflow framework.

As stated earlier, there are numerous types of DL architectures (e.g. AEs, CNNs, etc.). Of these, the way GANs work is quite different from the others. GANs consist of two sub-networks: a generator to produce plausible data and a discriminator to distinguish fake and real data. This two-component structure requires a special processing mechanism that is very different from that of other architectures. At the time of writing this article, the design process of such a mechanism is ongoing. The future versions of the program will soon allow working with GANs as well. However, users can easily override the built-in procedures (training, testing, etc.) and integrate them accordingly into relevant processes, and hence, the implementation of SISR can be done with their own designed GANs.

It has been assured with meticulous examinations and tests that the software meets the specifications and fulfills the required purpose, in terms of verification and validation. Several unit tests were performed to validate that the tool is up to the mark. There are many warnings to alert users, and exceptions to ensure uninterrupted operation of the program in case the inputs do not meet the requirements or are delivered in such an unexpected way or format that may cause a malfunction. As static testing, the reviews method was adopted for the verification process.

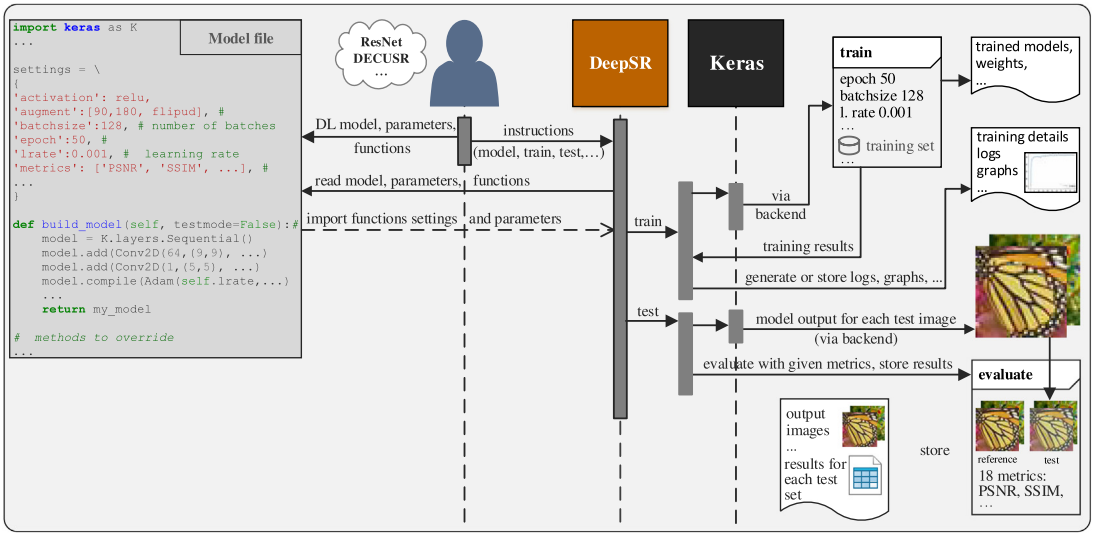

Fig. 2. Conceptual and functional scheme of SISR with DeepSR.

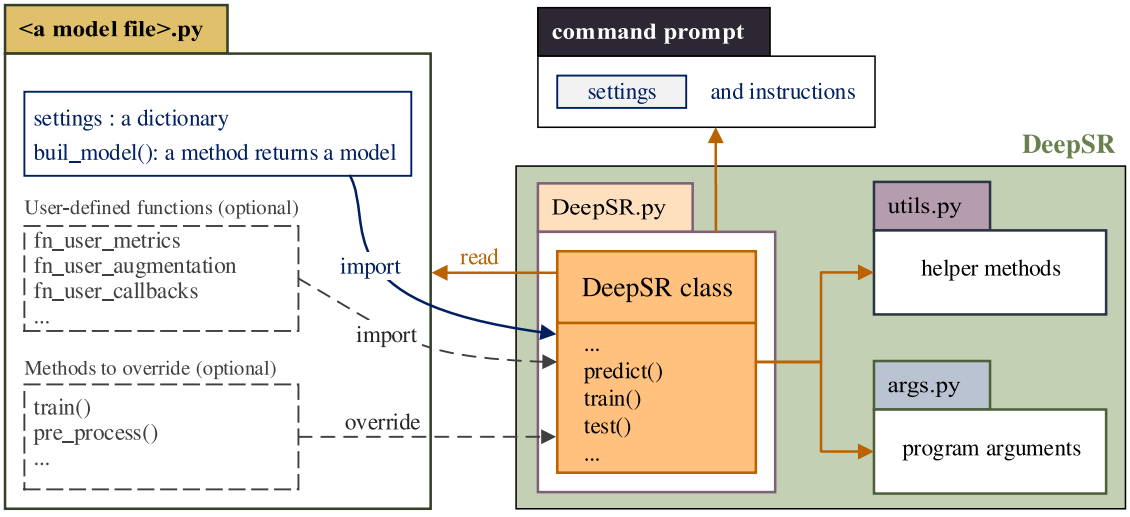

Fig. 3. Structure of DeepSR and interactions with it. Optional functions are shown with dashed lines.

2.1. Software architecture

DeepSR consists of several Python modules as illustrated in Fig. 3. The DeepSR class is served in the file DeepSR.py. It is the core python module executed first when the program is run. The user writes a Python method named 'build_model' which constructs a DL model defined by Keras's API as well as a dictionary named 'settings' that consists of required or optional parameters in key/value pairs. The 'build_model' and 'settings' can be provided either within a typical Python file when the program is executed from the command line, or in the same code file when it is used as a class from another Python program.

A model file, in summary, should contain at least the following two things:

- A dictionary named 'settings'. This dictionary is used to supply DeepSR with all necessary parameters and/or instructions to implement the relevant tasks.
- A function, named 'build_model'. This function returns the compiled version of the model coded with Keras' API.

DeepSR imports user-defined functions and overwrites member methods when given in the model file. In this way, users can easily customize DeepSR according to their needs. Any required parameters such as learning rate, stride, number of epochs, batches, etc., can be provided in the following ways:

i. within the 'settings' dictionary in the model file,
ii. as command arguments in the command line,
iii. as an attribute of the DeepSR class.

Platform and dependencies

DeepSR is implemented in Python and requires at least version 3.6 or higher. It depends on the following packages: Graphviz, h5py, Keras, Matplotlib, NumPy, openpyxl, pandas, Pillow, scikit-image, scikit-video, SciPy, sewar, sporco, TensorFlow. To run DL models with other backends such as Theano or CNTK they should also be installed. However, the main back-end is TensorFlow.

2.2. Software functionalities

2.2.1. Pre- and post-processing, data augmentation and normalization

In the implementation process of SISR, many different domain-specific methods, techniques, and approaches are applied, depending on the idea behind the approach taken to solve the problem. For example, some models take LR images as they are and upscale them in their layers, while others take their upscaled versions by a certain interpolation operator. In addition, models can operate in different color spaces or only on one or more channels. DeepSR allows such application details to be easily determined with a single parameter. Furthermore, it also offers some additional functionalities in addition to common applications. For instance, it allows the use of different operators for both downsampling and upsampling, as opposed to the common use of the same operator. Many more common or specific approaches for the sub-processes of SISR application (normalization, augmentation or pre-processing, etc.) were integrated into the program. Some are discussed in detail below.

Data augmentation is a very important process especially when there is not enough training data. It contributes to the data representation and learning capabilities of the models, and prevents overfitting. DeepSR offers a parametrized way of data augmentation. It allows data augmentation by rotating images by 90, 180 and 270 degrees, or flipping images vertically, horizontally, or both. Any combination of these operations can be chosen in the program. However, DeepSR allows to easily extend the image augmentation process with user-defined image transformations. Users can define in the model file a function called 'fn_user_augmentation' returning a list of images generated with user-defined image processing operations. This function becomes a member method of the DeepSR class when a model file is read. It can also be made a member method when DeepSR class is used from another Python program. This function should be defined as demonstrated below:

```
# function for user-defined image augmentation, in addition to built-in ones.
def fn_user_augmentation(self, image): # first parameter must be 'self' since
                                       # this function will become a member method of DeepSR
      image_list = list() # list of augmented images by user-defined transformations.
      # append each image yielded from user-defined image transformations.

      return image_list
```

Normalization is also a very crucial task that enables DL models to learn very quickly and steadily. DeepSR implements the following most common normalization techniques out of the box:

- **Divide**. The intensity values of images are divided by a certain value (e.g., 255.0)
- **Mean normalization**. Subtracts its mean value from the image being processed. The mean value can be calculated by the program in either way: per every single image being processed or from the entire training set of images. Users can also provide an arbitrary mean value, instead of computing.
- **Min–Max normalization**. The intensity values of images are re-arranged between given intervals in this normalization. For example, between -1 and 1.
- **Standardization**. Images are normalized by subtracting a mean value and dividing the result by a standard deviation. The mean and standard deviation are computed by the program either from every single image being processed or from the entire collection of the training set. Arbitrary values for both can also be given by the user, instead of computing.

The output HR images may require some additional adjustments in some cases. For example, some models trim some pixels from the edges when processing images through their layers. In such cases, the output images will be smaller than the reference images. Therefore, during the evaluation process, reference images may need to be cropped from the edges so that both the output and corresponding reference images are identical in size. For other features or detailed descriptions, please refer to the documentation.

2.2.2. Training deep learning models

In practice, models are trained either in an online or offline fashion mostly depending on the training data size. Online training is the most widely used approach by the community due to the high amount of data available today and hence adopted by DeepSR as the main approach. In online training, DeepSR exhausts the entire set of training images located in a directory. The workflow of the training procedure is shown in Fig. 4. Please note that the variables x and y in (1) are interchanged in the figure to provide a more intuitive understanding.

In most cases, real HR images are not available. So, the real-life scenario rarely happens. Therefore, DeepSR was developed to implement the simulation scenario in which LR images are synthetically generated in accordance with (1). DeepSR is currently being developed to support the real-life scenario.

DeepSR takes each image (patches in general) y from a training set and synthetically produces LR image x in accordance with (1). After applying a series of pre-processing operations on x (augmentation, normalization, adding noise, etc.), it is given to the DL algorithm so that the model learns the transformation function between x and y. The error (loss) rate between image y and its predicted (produced) version $\hat{y}$ is measured with a given metric and propagated back through the layers of the model to update layer weights. The next training process starts with updated layer weights. This process continues up until entire images in the training set have been exhausted for a given number of epochs, or until a termination condition is met (e.g., early stopping as no improvement in performance).

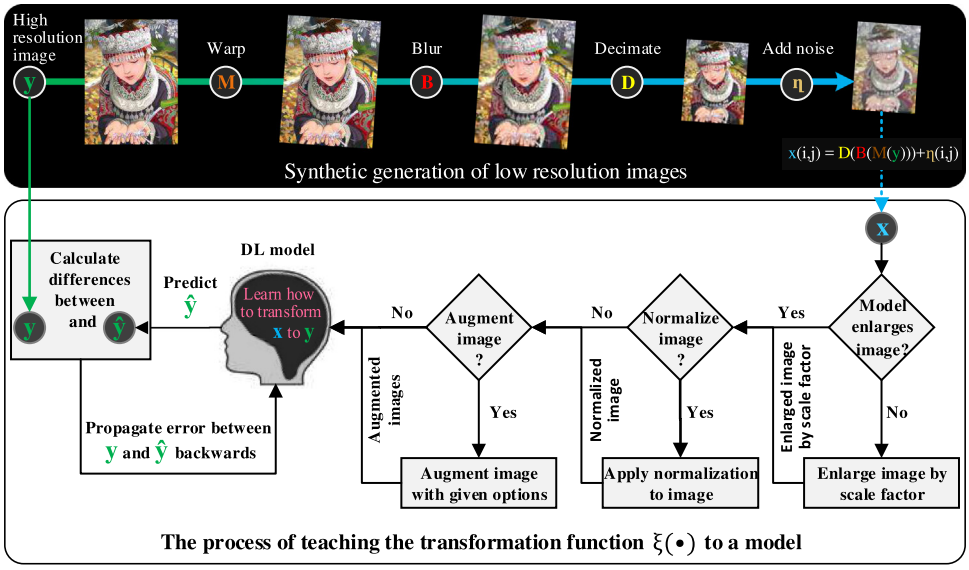

Fig. 4. Workflow of the training procedure in SISR with DeepSR. The terms x and y in (1) are interchanged for simplicity. The dark area shows the process of synthetic production of LR images.

2.2.3. Evaluation of deep learning models

A typical DL algorithm predicts an image $\hat{y}$ for each LR (synthetically generated or real) image y. Then, its performance in reconstructing the image y is evaluated by comparing $\hat{y}$ and y with one or more IQMs. DeepSR can exploit any combination of 18 IQMs for performance evaluation. These IQMs are as follows: BRISQUE [18], BSNR [19], ERGAS [20], GMSD [21], MAD, MSE [22], MSSSIM [23], NIQE [24], NRMSE [22], PAMSE [25], PSNR, RASE [26], SAM [27], SCC [28], SNR, SSIM [29], UQI [30], VIF [30].

The PSNR and SSIM metrics are computed by default. As the formal definitions of the IQMs are beyond the scope of this paper, and for brevity, the definitions of the most commonly used metrics PSNR and SSIM are given below and others in the program manual. However, all metrics are briefly explained in Appendix A.

Peak Signal to Noise Ratio (PSNR)

Let y and $\hat{y}$ be the reference (ground truth) and predicted image of a model, respectively. The first thing is to compute the mean-squared-error (MSE) which is a measure of the absolute discrepancy or distance between y and $\hat{y}$. The MSE is defined as follows:

$$\mathrm{MSE} = \frac{1}{PR}\sum_{i=1}^{P}\sum_{j=1}^{R}\left\|y(i,j) - \hat{y}(i,j)\right\|^2 \tag{2}$$

where P and R are the number of pixels in both directions: height and width. The PSNR is then computed as follows:

$$\mathrm{PSNR} = 10\log_{10}\left(\frac{L^2}{\mathrm{MSE}}\right) \tag{3}$$

where L is the maximum intensity value (255.0 for 8-bit images).

Structural Similarity Index (SSIM)

SSIM is a measure of similarity between two images. It takes into account the interdependencies of pixels in luminance, contrast, and structural terms. SSIM is defined as follows for images y and $\hat{y}$:

$$\mathrm{SSIM}(y,\hat{y}) = \frac{(2\mu_y\mu_{\hat{y}} + c_1)(2\sigma_{y\hat{y}} + c)}{(\mu_y^2 + \mu_{\hat{y}}^2 + c_1)(\sigma_y^2 + \sigma_{\hat{y}}^2 + c_2)} \tag{4}$$

where μ and σ denote the mean and variance of the corresponding image, and $\sigma_{y\hat{y}}$ is the covariance of y and $\hat{y}$. The variables c_1 and c_2 are calculated as follows:

$$c_1 = (k_1 L)^2 \tag{5}$$

$$c_2 = (k_2 L)^2 \tag{6}$$

L is the dynamic range of pixels (255.0 for 8-bit images), and $k_1 = 0.01$ and $k_2 = 0.03$ by default.

The computed IQM scores for each image are stored in an Excel file in a tabular format along with their averages in another sheet in the same workbook (refer to Appendix E). The test procedure can be done either for single or multiple weight files. DeepSR is capable to save every predicted output image, the weights or outputs of the layers during the test. The performance evaluation can be done for multiple test sets located in separate folders for the same experiment. The evaluation can also be done for just a single image and/or weight file by providing the path to the corresponding file rather than specifying the folders. Fig. 5 presents the workflow diagram of the test procedure.

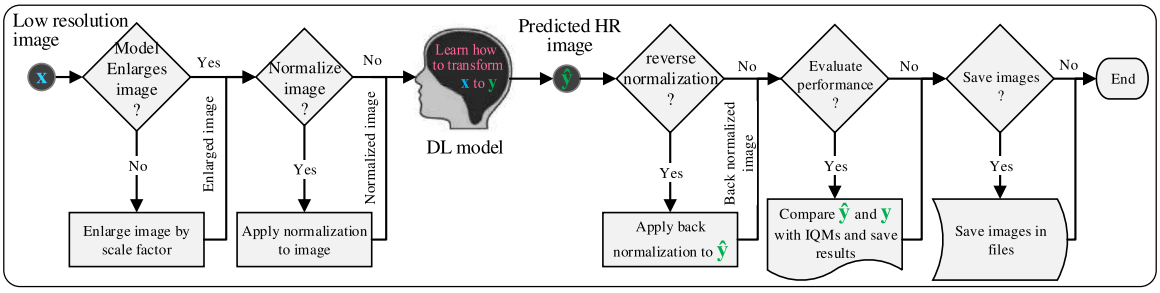

Fig. 5. The evaluation process for a trained model.

User-defined metrics can also be computed during the test procedure. These metrics can be provided through the function named 'fn_user_metrics' in the model file. This function can also be declared as a member method when the class is used from another Python program. It should return a dictionary of name-value pairs for each computed metric, as demonstrated below:

```
# function for user-defined metric(s) to be computed, in addition to built-in ones.
def fn_user_metrics(self, img_ref, img_out): # first parameter must be 'self' since
                                # this function will become a member method of DeepSR

      dct_metrics = dict() # dictionary of metrics
      for metric in user_defined_metrics:
            computed_value = user_function_to_compute_metric(img_ref, img_out)
            dct_metrics[<name of metric>] = computed_value #
            return dct_metrics # return dictionary of computed user-defined metrics
```

2.2.4. Visualization

DeepSR can visualize and store the model structure, layer weights and outputs. This functionality can be very important especially when users need to visually examine model's structure, the output images or outputs of layers. Besides, the performances of the models during the training in both the PSNR and the given loss function are visualized in graphic files, in addition to logs. Some visualizations from the illustrative example in Section 3 are given in the example and Appendix D.

2.2.5. Callbacks and logs

There are several Keras' callbacks at the user's disposal out of the box. These callbacks allow many additional works to be done during the operation, or to take actions depending on the active state of the executed process or the fulfillment of a certain criterion. The built-in callbacks are as follows:

- **LossHistory** to record losses (training and, if exists, validation) within the class in a dictionary.
- **ModelCheckpoint** for recording the model weights after each epoch.
- **CSVLogger** to log in a text file the losses calculated for each epoch.
- **EarlyStopping** to stop training procedure if no improvement is achieved for several successive epochs. The 'espatience' parameter controls the number of epochs to wait before terminating training.
- **ReduceLROnPlateau** to reduce the learning rate value if no improvement is achieved for a number of successive epochs. The 'lrpatience' parameter controls the number of epochs to wait to reduce the value.

In addition to built-in ones, users can write their callbacks within the function named 'fn_user_callbacks' in the model file. This function is expected to return a list of callbacks. It should be defined similarly to the example below:

```
# function for user-defined callbacks. A single callback is defined, as an example.
def fn_user_callbacks(self,): # the first parameter must be 'self' since
                                # this function will become a member method of DeepSR

      callback_list = list() # list of Keras' callbacks.
      from keras.callbacks import ProgbarLogger # Progresbar logger callback function
      cbk_prlogr = ProgbarLogger(count_mode='samples') # callback is defined
      callback_list.append(cbk_prlogr) # add the callback to the list

      # define other other callbacks in a similar fashion

      return callback_list # return list of callbacks
```

DeepSR appends the user-defined call-backs to the built-in list at run time. The same function can be directly fed to the 'prepare_callbacks' member method with the parameter, fn_user_callbacks, when running it from another Python program. An illustration is given below:

```
# function for user-defined callbacks
# we assume that the same function given above is already defined
dsr = DeepSR()
…
# this returns the extended list of callbacks (built-in and user-defined)
cbk_list = dsr.prepare_callbacks(fn_user_callbaks=fn_user_callbacks)
```

In addition, the user can easily override built-in callbacks by redefining the "prepare_callbacks" member method in the model file.

3. Illustrative examples

3.1. Installation

Although it is possible to use DeepSR directly by downloading the source project folder with its contents, it is much more convenient to install as it requires many other Python packages.

The source code and binaries for the Python environment are served at the PyPI[1] repository at the following address: https://pypi.org/project/DeepSR/

The program can be installed with the following command:

```
python --m pip install DeepSR
```

All dependencies will be installed automatically during installation.

3.2. Implementation

After installation, it can be imported into a Python program, or run through a Python interpreter with command line scripts. However, it is mainly designed to interact with it from the command line. In the following sections, it is shown how to use DeepSR from the command line first and then how to use it as a class in the python environment.

A full-fledged code for a modified version of the SRCNN [8] model is given in Appendix B. The reason for using the SRCNN in the illustration is that its simplicity in architecture compared to the others. So, it helps to keep the example simple and the article concise. More examples and detailed explanations are available on the GitHub page, including some known models such as DECUSR [31] and VDSR [32]. The SRCNN was originally designed to process single-channel images in YCbCr color space. Here it is modified to process 3-channel RGB images. It is assumed that the following code is stored in the 'SRCNN.py' file.

The required packages are imported in the first part of the code. The most common hyper-parameters and settings are given in the 'settings' dictionary. A short description for each parameter is given as a comment on the same line. Please refer to the documentation for other parameters and further descriptions.

The following four functions are also defined in the rest of the code: build_model, fn_user_callbacks, fn_user_metrics and fn_user_augmentation. Each of these must be declared with the exact given name for the program to work successfully. They become member methods of the DeepSR class whenever the model file is called from the command line, or they are registered through the 'make_member' method when using DeepSR as a class (demonstrated later on). The last three functions allow the program to be extended by incorporating additional user-defined procedures into relevant processes.

The build_model constructs the SRCNN model accordingly before starting a training or test procedure. The fn_user_callbacks allows to write additional user-defined callback(s). The given code illustrates how to define the Keras' learning rate scheduling callback. This callback will be added to the built-in callbacks list and executed along with the others during the training procedure.

The fn_user_metrics simply returns a dictionary for user-defined metric(s) in <name, value> pairs. Each metric in the dictionary will be added to the built-in metrics list (by default, PSNR and SSIM), and calculated when the model's performance is evaluated. The given metric in the illustration calculates the log10 of the mean absolute deviations between the reference and test image. Some computed values for this IQM and others are given in Appendix E.

Additional image transformations for data augmentation can be provided by users within the fn_user_augmentation. The given code augments training data by rotating images by 10 degrees. Each augmentation operation defined in this function is executed in addition to the built-in augmentation operations in the pre-processing stage of the training process.

3.2.1. Running from command line

As stated earlier, the main purpose of DeepSR is to perform tasks interactively with scripts. It allows to perform multiple tasks sequentially, or in parallel by distributing jobs to separate virtual python environments or separate GPUs (if exist) on the same machine. The below code illustrates how to train and test the SRCNN model defined in the sample file from the command line.

```
python -m DeepSR.DeepSR --train --test --modelfile SRCNN.py --inputsize 33 --epoch 50
--scale 2 --colormode RGB --metrics PSNR SSIM VIF --shuffle --normalization minmax -1 1
--stride 11 --shutdown --gpu 1 --lrpatience 3 --saveimages --noise 0.0 0.1
--batchsize 128
```

[1] https://pypi.org/.

Table 1
Some command arguments in the illustration.

Argument	Description
–batchsize 128	The model's weight shall be updated (the model's error is back propagated) after 128 image patches are given to the model for training. Overrides the value of 64 in the settings.
–colormode RGB	The images to be processed in RGB color space. The same value as given in the settings. It has no different effect.
–epoch 50	The model will be trained for at most 50 epochs. Overrides the value of 10 in the definition file.
–gpu 1	The second GPU (as it is zero-ordered) only will be used for this task, if exists. In this way, multiple tasks can be performed on multiple GPUs on the same machine. Not given in the settings, but here in the command script.
–inputsize 33	The size of input patches (image) for training is 33 × 33 pixels. Overrides the value of 21.
–lrpatience 3	The learning rate value will be reduced if no improvement is achieved for 3 epochs. The same value as in the definition file.
–metrics PSNR SSIM VIF	The PSNR, SSIM, and VIF image quality measures will be used for the evaluation. Overrides the list of measures in the definition file.
–noise 0.0 0.1	Additive noise with 0.0 mean and 0.1 standard deviation to be applied to the images. None is given in the settings but here in the script.
–normalization minmax −1 1	Normalize input images by Min–Max normalization between −1 and 1. Overrides the normalization method given in the settings.
–saveimages	Predicted (output of the model) images will be stored during the test procedure. Not given in the settings but here in the script.
–scale 2	The scale factor is 2 now. Overrides the value of 4 in the settings.
–shutdown	The computer will be shut down after all tasks have been finished.
–stride 11	The patches of training images to be cropped 11 pixels apart. Same as in the settings.

Parameters or instructions can be given as command arguments as shown above. Command arguments override the settings in the model file. Some descriptions for the arguments are given below.

The –train and –test respectively instruct the program to train the model first, then evaluate its performance. The –train parameter should not be given when training is not needed. In this example, the training procedure was stopped by the early stopping callback at the 23rd epoch as no performance improvements were achieved for the last 5 epochs. The BSDS100, Set14, and Set5 datasets [33] were used for training, validation, and test processes, respectively. These sets contain 100, 14, and 5 images, respectively. The definition file, 'SRCNN.py', is declared by the — modelfile argument. Some of the other arguments are described in Table 1.

The command line outputs of the program during this experiment are given in Appendix C. After completion of the training and test procedures, depending on the performed tasks, various directories and files similar to the following illustrations will be created by the program:

<working directory>/output/<model>.png (graphic of model. refer to Figure D.1 in Appendix D)
<working directory>/output/<scale>/
batch_losses.txt (losses calculated for each batch)
<model>_<test>_scale_<scale>.xlsx (refer to Appendix E)
training.txt (training and validation losses at each epoch)
Training graph in PSNR.png (refer to Figure D.2 in Appendix D)
Training graph <loss>.png (refer to Figure D.3 in Appendix D)
training_summary.txt (layer details, parameters, etc.)
training.log (training and validation losses at each epoch)
weigths.01.h5
…
weights.23.h5
<working directory>/output/<scale>/images/
z_baby_GT_scale_2_weights.01.png 
…
z_baby_GT_scale_2_weights.01.png 
…
z_butterfly_GT_scale_2_weights.01.png 
…
z_head_GT_scale_2_weights.01.png 
…
z_woman_GT_scale_2_weights.23.png 
<working directory>/output/<scale>/layerweights/
…
<working directory>/output/<scale>/layeroutputs/
…

Please note that many other command arguments are available. More detailed information and samples can be found in the documentation.

3.2.2. Running as Python class

The DeepSR class can be exploited by another Python program. A very short example is shown in the below code snippet. It is assumed that the sample code is appended to the end of the SRCNN.py file. So, the same settings and functions apply to this illustration as well.

An instance of the class can be created by providing the dictionary of settings. This is not the only way but it should be preferred for simplicity and convenience. All parameters can be re-assigned to the class instance. User-defined functions can be made members of the instance by using the member method make_member, as shown in the example below. Certain tasks (e.g. training) can be performed by calling relevant methods after all necessary arrangements have been made for these tasks. The example illustrates how to start the training procedure after the necessary adjustments have been made.

```
# An example of how to use the DeepSR as a class.
# It is assumed that this code snippet is within the SRCNN.py file

from DeepSR import DeepSR
DSR = DeepSR(settings)  # make an instance of DeepSR with given hyper-parameters.

# register the construction method and user-defined functions as a member of the class.
DSR.make_member( [build_model, fn_user_metrics, # input(s) can be a single function
      fn_user_augmentation, fn_user_callbacks]) # or a list of functions
… # do other arrangements
DSR.train()   # train the model.
…
```

4. Impact

The foundations of this program were laid in studies [31,34], which required software that ensures rapid model prototyping and integrated execution of the SISR pipeline for multiple tasks. Even the basic versions of this program have allowed the successful completion of countless studies with many different DL models and alternative parameter sets. There are several other studies currently under peer review that benefited from DeepSR to perform the experiments.

In brief, DeepSR offers many features including but not limited to the followings:

- Unified and integrated environment for SISR application with DL algorithms.
- Easy, fast, and simple model prototyping.
- Easy and simple command line interface.
- Ability to perform multiple tasks with batch files.
- Comprehensive pre-processing, evaluation, visualization, and reporting capabilities.
- Extensibility with user-defined procedures.
- Ability to distribute tasks on separate GPUs.

5. Conclusions

This article introduces an open-source tool that provides a versatile environment for the design and implementation of SISR with DL algorithms in any imaging technique (medical, aerial, etc.). The entire pipeline of a typical SISR application has been carefully analyzed and integrated into the program. DeepSR successfully meets the needs of researchers with rich feature sets. It is very easy to design DL models, and implement multiple SISR applications for different parameter sets or alternative models through simple command line scripts or batch files. It can be extended by incorporating user-defined functions into related processes. DeepSR can be a common environment for SISR with DL algorithms in any imaging technique.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

No data was used for the research described in the article.

Appendix A. Available image quality measures

See Table A.1.

Table A.1
Available image quality measures.

Metric	Description	Full ref.	Library
BRISQUE [18]	Blind/Referenceless Image Spatial Quality Evaluator. Computes the no-reference image quality score (IQS) for an image. Takes only one image to compute the IQS. It is typically used for natural scene images.	–	scikit-video
BSNR [19]	Blurred Signal to Noise Ratio. Computes a score for the discrepancy between a blurred noise-free image and a blurred image with additive noise.	+	sporco
ERGAS [20]	Erreur relative globale adimensionnelle de synthèse. A channel-wise NRMSE is multiplied by the ground sampling distances (GSD) ratio which is the square root of the pixel count ratio between LR and HR images. A lower score indicates a better quality.	+	sewar
GMSD [21]	Gradient Magnitude Similarity Deviation. An efficient IQM that is responsive to compression artifacts, blur, or additive noise on images. Computed on the gradient magnitude maps of reference and distorted image, on their correlation. Ensures a fast and easier optimization, high accuracy, and less time and memory complexity than SSIM. A lower score indicates better quality.	+	sporco
MAD	Mean Absolute Deviation. Computes the mean of the absolute differences between the reference and distorted image. A lower score indicates better quality.	+	scikit-video
MSE [22]	Mean Squared Error. The overall sum of the squared differences between corresponding pixels of two images. Its definition is given in (2).	+	scikit-image
MSSSIM [23]	Multi-Scale Structural Similarity Index. Computes the overall SSIM over each version of the reference and the distorted image obtained by iterative downsampling by a factor of 2. The luminance component is computed only at the initial scale, while the contrast and structure comparisons are computed at each scale. The authors state that it outperforms the SSIM.	+	sewar
NIQE [24]	Naturalness Image Quality Evaluator. A completely blind IQM based on building a collection of "quality aware" features obtained from local patches of undistorted natural images in a collection and fitted to a multivariate Gaussian model (MVG). The score is computed as the distance between the MVG's of the test image and the image corpus.	–	scikit-video
NRMSE [22]	Normalized Root Mean Square Error. Designed to ensure scale-independency, unlike root mean square error (RMSE). Simply calculated by multiplying RMSE with the reciprocal of a normalization factor, which is usually equal to either the maximum observed value or the difference between the maximum and minimum values.	+	scikit-image
PAMSE [25]	Perceptual-fidelity Aware Mean Squared Error. Designed to enhance the perceptual fidelity awareness of MSE by introducing an l_2 norm term. Originally designed for natural images. It is said by designers that it works better than SSIM and can be used as the objective function in perceptual quality-based image processing tasks. A lower score indicates a better quality.	+	sporco
PSNR	Peak Signal to Noise Ratio. The ratio of the maximum possible power of a signal to the power of noise affecting its representation fidelity. In short, it is the ratio of the maximum intensity value to the MSE between two images in logarithmic scale. The formal definition is given in (3).	+	scikit-image
RASE [26]	Relative Average Spectral Error. Originally designed to evaluate the quality of multispectral images. It characterizes the average performance of a technique in percent. Computed by dividing the square root of the mean RMSE over all spectral bands by the mean radiance over all spectral bands.	+	sewar
SAM [27]	Spectral Angle Mapper. A measure of the spectral similarity of an image to reference spectra obtained by calculating the angle between the two. A lower score indicates a better quality.	+	sewar
SCC [28]	Spatial Correlation Coefficient. Originally designed to score the correlation between panchromatic and fused images in multispectral imaging. Computed by dividing the covariance of the two images by the product of their standard deviations.	+	sewar
SNR	Signal to Noise Ratio. Computes the ratio of the signal power to the background noise power, expressed in a logarithmic decibel scale. The ratio above 1.0 is a good indication of more signal-to-noise.	+	sporco
SSIM [29]	Structural Similarity Index. It is one of the gradient-based criterion incorporating gradient similarity functions. The perceptual quality of an image is measured from three aspects: luminance, contrast, and structure. The similarity is computed for each of these. The overall score is obtained by the product of these three.	+	scikit-image
UQI [30]	Universal Quality Image Index. A score to determine the degree of the deformation on an image compared to the reference image. The degree is calculated by combining three factors: correlation loss, luminance, and contrast deformation. These factors are calculated through the statistical properties of the images.	+	sewar
VIF [30]	Visual Information Fidelity. It uses a natural scene statistics model in the wavelet domain using Gaussian scale mixtures (GSMs). It is also derived from a model for image deformations and the information-theoretical setting of the human visual system. Specifically, it captures the following distortion types: additional noise, blur, and local or general contrast changes.	+	sewar

The plus sign in the Full Ref. column indicates whether the metric is full-reference or not.

Appendix B. Content of SRCNN.py file, used in the illustrative example in Section 3

```
# SRCNN.py. A modified SRCNN model for a 3-channel RGB image super-resolution.
import numpy as np
from keras import losses
from keras.layers import Input
from keras.layers.convolutional import Conv2D
from keras.models import Model
from keras.optimizers import Adam
from os.path import basename
from keras.callbacks import LearningRateScheduler

# hyper-parameters in order. Can be overriden when given as command arguments.
settings = \ # please, refer to program manual for other hyper-parameters.
    {
      'activation': 'relu',     # relu is the activation function for the layers.
      'augment':[90, 180, 270],# augment data by rotating images by these degrees.
      'backend': 'tensorflow', # use TensorFlow as Keras' backend.
      'batchsize': 64, # batch size. Do backpropagation after processing 64 images.
      'channels': 3,       # input image is of 3 channels.
      'colormode': 'RGB', # use 'RGB' color space.
      'crop': 6,           # crop 6 pixels from borders since SRCNN yields so.
      'crop_test': 0,      # number of pixels cropped from borders of test images.
      'decay': 0.9,        # weight decay value for the optimizer.
      'decimation': 'bicubic', # downsample images with bicubic interpolation.
      'dilation_rate':(1,1), # dilation factor for Keras' dilated layers.
      'espatience': 5,     # wait for 5 epochs to stop when no improvement is achieved.
      'epoch': 10,         # train the model for 10 epochs at maxium.
      'inputsize': 21,     # train with image patches of size 21x21 pixels.
      'interp_compare': '',    # no interpolation method given to compare the model.
      'interp_up': 'bicubic', # upscale images with bicubic interpolation method.
      'kernel_initializer': 'glorot_uniform', # kernel initialization method.
      'lrate': 0.001, # initial learning rate value for training the model.
      'lrpatience': 3, # wait for 3 epochs before reducing 'lrate' by 'lrfactor'.
      'lrfactor': 0.5, # reduce learning rate by a half when no improvement.
      'metrics': ['PSNR', 'SSIM'], # evaluate the model with these IQMs.
      'minimumlrate': 1e-7,      # the minimum learning rate value.
      'modelname': basename(__file__).split('.')[0], # same name as this file.
      'noise': '',               # noise to be added to input image before feeding.
      'normalization': ['divide', 255.0], # divide values by 255.0 to normalize.
      'outputdir': '',           # provide a path to change the output directory.
      'scale': 4,                # LR image will be upscaled by a factor of 4.
      'seed': 19,                # fix the randomness to make results reproducible.
      'shuffle': True,           # shuffle images before any pre-process for training.
      'stride': 11,              # the interval in pixels between image patches.
      'target_channels': 3,      # model should yield 3 channel image.
      'target_cmode': 'RGB',     # process image in RGB color space.
      'testpath': [r'C:\datasets\set5'], # path(s) to test folders or files.
      'traindir': r'C:\datasets\BSDS100',
      'upscaleimage': True,  # upscale downsampled image before feeding to model.
      'valdir': r'C:\datasets\set14', # path to the folder of validation files.
      'weightpath': '', # no path to a weight file is given to load the model with.
      'workingdir': '', # provide a path to change the current working directory.
    }


def build_model(self, testmode=False):
"""Construction method for modified SRCNN model for 3 channel processing in RGB"""
    # full-size image for the test, otherwise given size image patch for training
    input_size = None if testmode else self.inputsize
    input_shape = (input_size, input_size, self.channels)

    INPUT = Input(shape=input_shape)
```

```
    SRCNN = Conv2D(64,(9,9), kernel_initializer=self.kernel_initializer,
        padding='valid', input_shape=input_shape, activation=self.activation)(INPUT)
    SRCNN = Conv2D(32, (1,1), kernel_initializer=self.kernel_initializer,
                           padding='valid', activation=self.activation)(SRCNN)
    SRCNN = Conv2D(self.channels,(5,5), kernel_initializer=self.kernel_initializer,
                           padding='valid', activation=self.activation)(SRCNN)
    SRCNN = Model(INPUT, outputs=SRCNN)
    SRCNN.compile(Adam(self.lrate, self.decay), loss=losses.mean_squared_error)
    return SRCNN


def fn_user_callbacks(self): # to be a member method of DeepSR via 'make_member'
    """User-defined callback(s) for learning rate scheduling"""
    callbacks_list=[]

    def step_decay(epoch):
        from tensorflow.math import pow, floor
        initial_lrate = self.lrate # take learning rate from the class
        drop = 0.5 # reduce learning rate by a half
        epochs_drop = 20.0 # after every 20 epochs
        lrate = initial_lrate * pow(drop, floor((1 + epoch) / epochs_drop))
        return np.float(lrate)

    lrate = LearningRateScheduler(step_decay) # Keras' learning rate scheduler.
    callbacks_list.append(lrate)

    return callbacks_list


def fn_user_metrics(self, img_ground, img):
    """User-defined evaluation metrics.
Returns a dictionary of metric(s) with <name, value> pairs."""
    # return log10 of mean absolute deviation between reference and test image.
    return {'LOGMAD': np.log10(np.abs(img_ground - img).mean())}


def fn_user_augmentation(self, img):
    """User-defined augmentation. Rotate the image by 10 degrees.
    :parameter img: source image to be augmented by rotating by 10 degrees.
    """
    from PIL import Image
    im_list =[]
    im_list.append(np.asarray(Image.fromarray(img).rotate(10)))# rotate by 10 degrees
    return im_list
```

Appendix C. Command line logs generated by DeepSR in the illustration in Section 3

```
[START TRAINING]
Calculating the number of training samples...Done!
Totally 553500 of training samples to be extracted from training set.
Calculating the number of validation samples...Done!
Totally 118845 of validation samples to be extracted from training set.
Totally 4324 batch updates to be calculated for training.
Model: "model_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_2 (InputLayer)        [(None, 33, 33, 3)]       0
 conv2d_3 (Conv2D)           (None, 25, 25, 64)        15616
 conv2d_4 (Conv2D)           (None, 25, 25, 32)        2080
 conv2d_5 (Conv2D)           (None, 21, 21, 3)         2403
=================================================================
Total params: 20,099
Trainable params: 20,099
Non-trainable params: 0
_________________________________________________________________
None
Batch generator starting...
  0% 0/100 [00:00<?, ?it/s]Epoch 1/50
100% 100/100 [00:49<00:00,  2.02it/s]
  0% 0/100 [00:00<?, ?it/s]
  0% 0/14 [00:00<?, ?it/s]
  7% 1/14 [00:00<00:07,  1.69it/s]
 14% 2/14 [00:00<00:04,  2.77it/s]
 21% 3/14 [00:01<00:06,  1.78it/s]
 29% 4/14 [00:02<00:05,  1.83it/s]
 36% 5/14 [00:02<00:04,  1.83it/s]
 43% 6/14 [00:03<00:04,  1.85it/s]
 50% 7/14 [00:03<00:03,  1.87it/s]
 57% 8/14 [00:03<00:02,  2.43it/s]
 64% 9/14 [00:04<00:02,  1.81it/s]
 71% 10/14 [00:05<00:02,  1.89it/s]
 79% 11/14 [00:05<00:01,  2.34it/s]
 86% 12/14 [00:05<00:00,  2.87it/s]
 93% 13/14 [00:06<00:00,  2.19it/s]
100% 14/14 [00:06<00:00,  2.11it/s]

[00:00<?, ?it/s]4324/4324 - 56s - loss: 619.1965 - val_loss: 563.3568 - lr: 0.0010 - 56s/epoch - 13ms/step
Epoch 2/50
…
…
[00:00<?, ?it/s]4324/4324 - 52s - loss: 203.2506 - val_loss: 399.8089 - lr: 5.0000e-04 - 52s/epoch - 12ms/step
Epoch 23/50
100% 100/100 [00:52<00:00,  1.91it/s]
 99% 99/100 [00:45<00:00,  2.16it/s]
  7% 1/14 [00:46<10:06, 46.65s/it]
 14% 2/14 [00:47<03:53, 19.48s/it]
 21% 3/14 [00:47<01:59, 10.83s/it]
 29% 4/14 [00:48<01:08,  6.83s/it]
 36% 5/14 [00:48<00:39,  4.42s/it]
 43% 6/14 [00:49<00:24,  3.09s/it]
 50% 7/14 [00:49<00:15,  2.14s/it]
 57% 8/14 [00:49<00:09,  1.57s/it]
 64% 9/14 [00:50<00:06,  1.32s/it]
 71% 10/14 [00:50<00:04,  1.07s/it]
 79% 11/14 [00:50<00:02,  1.26it/s]
 86% 12/14 [00:51<00:01,  1.24it/s]
 93% 13/14 [00:52<00:00,  1.61it/s]
100% 14/14 [00:52<00:00,  3.75s/it]

Epoch 23: ReduceLROnPlateau reducing learning rate to 0.0002500000118743628.
4324/4324 - 52s - loss: 198.4574 - val_loss: 293.5032 - lr: 2.5000e-04 - 52s/epoch - 12ms/step
Epoch 23: early stopping
Training has taken 1219.377 seconds.
[TRAIN FINISHED]

Writing Batch losses...Done!

[START TEST]
../datasets/set5
  0% 0/5 [00:00<?, ?it/s]
 20% 1/5 [00:26<01:46, 26.69s/it]
 40% 2/5 [00:51<01:17, 25.79s/it]
 60% 3/5 [01:18<00:52, 26.02s/it]
 80% 4/5 [01:43<00:25, 25.59s/it]
100% 5/5 [03:02<00:00, 36.54s/it]
[TEST FINISHED]
```

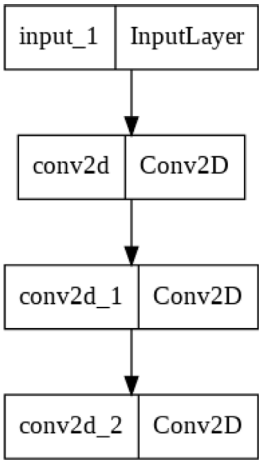

Fig. D.1. The graph of model's architecture stored in the 'SRCNN.png' file.

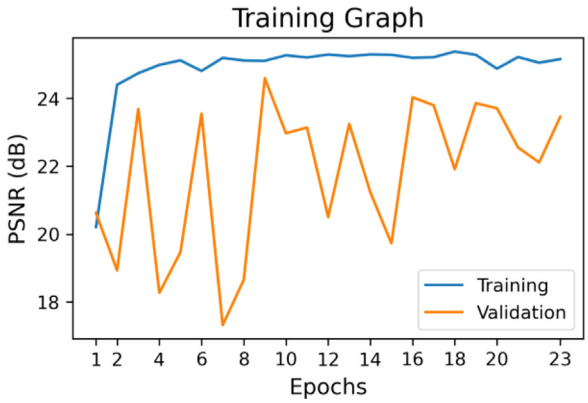

Fig. D.2. Content of 'Training graph of SRCNN in PSNR.png' file. It visualizes the training and validation scores at each epoch in the PSNR measure.

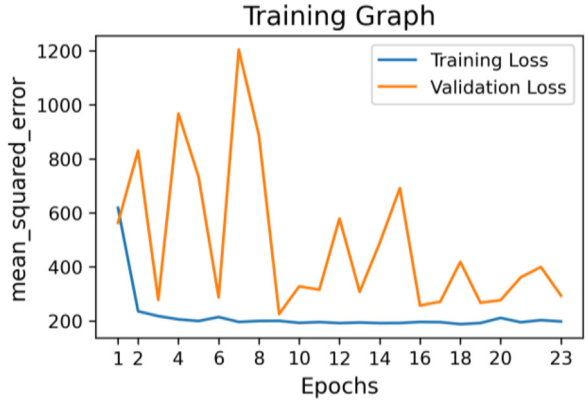

Fig. D.3. Content of 'Training graph of SRCNN.png' file. It visualizes the training and validation scores at each epoch in the given loss function (MSE in the illustration).

Appendix D. Visualizations of model architecture and training graphs generated by DeepSR for the illustration in Section 3

See Figs. D.1–D.3.

Appendix E. IQM scores stored in the SRCNN_set5_scale_2.xlsx file as a result of the evaluation in the example in Section 3

See Tables E.1 and E.2.

Table E.1
An excerpt from the sheet 'Sheet1' containing IQM scores for the test set SET5. The scores are shown here only for the first three weight files though were calculated for all 23 weights.

		z_baby_GT	z_bird_GT	z_butterfly_GT	z_head_GT	z_woman_GT	Mean
weights.01	LOGMAD	2.306116999	2.329695148	2.230845727	2.288190131	2.321773703	2.295324342
	PSNR	12.83152473	16.11840697	20.10118145	24.4279485	19.46410879	18.58863409
	SSIM	0.80506518	0.643771988	0.808581708	0.662451043	0.853833709	0.754740726
	VIF	0.473069228	0.346252884	0.383726238	0.351653166	0.529528967	0.416846097
weights.02	LOGMAD	2.143824653	2.329594248	2.166860029	2.277821981	2.168088205	2.217237823
	PSNR	18.91673633	18.95096219	21.82249616	18.14825744	19.43879108	19.45544864
	SSIM	0.866140035	0.687892704	0.861675382	0.587578911	0.835882265	0.767833859
	VIF	0.538831384	0.490040977	0.438283222	0.362120201	0.553530764	0.47656131
weights.03	LOGMAD	2.071449529	2.344916356	2.035864712	2.24133311	2.25774318	2.190261377
	PSNR	23.40321595	22.84971818	23.27355096	26.5855944	20.08202425	23.23882075
	SSIM	0.915120912	0.777738945	0.886369003	0.744606858	0.898945072	0.844556158
	VIF	0.611636574	0.49583484	0.472299892	0.409342034	0.621139825	0.522050633

Table E.2
An excerpt from the sheet 'Mean' containing the averages of IQMs over the test set SET5. Results are shown for the first eight weights only though were computed for all 23 weights.

	weights.01	weights.02	weights.03	weights.04	weights.05	weights.06	weights.07	weights.08
LOGMAD	2.295324342	2.217237823	2.190261377	1.99821398	2.310226459	2.108024681	2.239865887	2.294093174
PSNR	18.58863409	19.45544864	23.23882075	20.18519851	19.51954813	22.76999693	15.88578492	15.05101649
SSIM	0.754740726	0.767833859	0.844556158	0.838367357	0.780252939	0.869301395	0.69541375	0.703740951
VIF	0.416846097	0.47656131	0.522050633	0.462978142	0.508991838	0.525262134	0.400267013	0.425709425

References

[1] Keys RG. Cubic convolution interpolation for digital image processing. IEEE Trans Acoust Speech Signal Process 1981;29(6):1153–60.
[2] Tsai R. Multiframe image restoration and registration. Adv Comput Visual Image Process 1984;1:317–39.
[3] Aly HA, Dubois E. Image up-sampling using total-variation regularization with a new observation model. IEEE Trans Image Process 2005;14(10):1647–59.
[4] Yang J, Wright J, Huang T, Ma Y. Image super-resolution as sparse representation of raw image patches. In: 26th IEEE conference on computer vision and pattern recognition. 2008, p. 1–8.
[5] Chang H, Yeung DY, Xiong Y. Super-resolution through neighbor embedding. In: Proceedings of the IEEE computer society conference on computer vision and pattern recognition. 2004, p. 1.
[6] Romano Y, Isidoro J, Milanfar P. RAISR: Rapid and accurate image super resolution. IEEE Trans Comput Imaging 2016;3(1):110–25.
[7] Tang Y, Yan P, Yuan Y, Li X. Single-image super-resolution via local learning. Int J Mach Learn Cybern 2011;2(1):15–23.
[8] Dong C, Loy CC, He K, Tang X. Learning a deep convolutional network for image super-resolution. In: 13th European Conference on Computer Vision. 2014, p. 184–99.
[9] Chollet F. Keras: The Python deep learning library, v2.8.0. https://Keras.Io.
[10] Abadi M, et al. Tensorflow: A system for large-scale machine learning. In: 12th Symposium on operating systems design and implementation. 2016, p. 265–83.
[11] Bergstra J, et al. Theano: A CPU and GPU math compiler in Python. In: Proceedings of the Python for scientific computing conference. 2010, p. 1–7.
[12] Jia Y, et al. Caffe: Convolutional architecture for fast feature embedding. In: Proceedings of the 2014 ACM conference on multimedia. 2014, p. 675–8.
[13] Seide F, Agarwal A. 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.
[14] Collobert R, Kavukcuoglu K, Farabet C. Torch7: A matlab-like environment for machine learning. In: BigLearn, NIPS workshop. 2011.
[15] Paszke A, et al. Pytorch: An imperative style, high-performance deep learning library. In: Advances in neural information processing systems. 2019, p. 32.
[16] Vedaldi A, Lenc K. MatConvNet – Convolutional neural networks for MATLAB. In: Proceedings of the 23rd ACM international conference on multimedia. 2015, p. 689–92.
[17] Temiz H. Htemiz/DeepSR: A Python tool for super resolution with deep learning algorithms. Zenodo; p. v080. http://dx.doi.org/10.5281/zenodo.4310169.
[18] Mittal A, Moorthy AK, Bovik AC. Blind/referenceless image spatial quality evaluator. In: 2011 Conference record of the forty fifth asilomar conference on signals, systems and computers. 2011, p. 723–7.
[19] Yan L, Jin M, Fang H, Liu H, Zhang T. Atmospheric-turbulence-degraded astronomical image restoration by minimizing second-order central moment. IEEE Geosci Remote Sens Lett 2012;9(4):672–6.
[20] Raimundo J, Lopez-Cuervo Medina S, Prieto JF, de Mata J. Super resolution infrared thermal imaging using pansharpening algorithms: Quantitative assessment and application to UAV thermal imaging. Sensors 2021;21(4):1265.
[21] Xue W, Zhang L, Mou X, Bovik AC. Gradient magnitude similarity deviation: A highly efficient perceptual image quality index. IEEE Trans Image Process 2013;23(2):684–95.
[22] Shcherbakov MV, Brebels A, Shcherbakova NL, Tyukov AP, Janovsky TA, Kamaev VA. A survey of forecast error measures. World Appl Sci J 2013;24(24):171–6.
[23] Wang Z, Simoncelli EP, Bovik AC. Multiscale structural similarity for image quality assessment. In: The thrity-seventh asilomar conference on signals, systems & computers. 2003, p. 1398–402.
[24] Mittal A, Soundararajan R, Bovik AC. Making a 'completely blind' image quality analyzer. IEEE Signal Process Lett 2012;20(3):209–12.
[25] Xue W, Mou X, Zhang L, Feng X. Perceptual fidelity aware mean squared error. In: Proceedings of the IEEE international conference on computer vision. 2013, p. 705–12.
[26] Ranchin T, Wald L. Fusion of high spatial and spectral resolution images: The ARSIS concept and its implementation. Photogramm Eng Remote Sens 2000;66(1):49–61.
[27] Yuhas RH, Goetz AFH, Boardman JW. Discrimination among semi-arid landscape endmembers using the spectral angle mapper (SAM) algorithm. In: JPL, summaries of the third annual JPL airborne geoscience workshop. 1992.
[28] Zhou J, Civco DL, Silander JA. A wavelet transform method to merge landsat TM and SPOT panchromatic data. Int J Remote Sens 1998;19(4):743–57.
[29] Wang Z, Bovik AC, Sheikh HR, Simoncelli EP. Image quality assessment: From error visibility to structural similarity. IEEE Trans Image Process 2004;13(4):600–12.
[30] Alparone L, Aiazzi B, Baronti S, Garzelli A, Nencini F, Selva M. Multispectral and panchromatic data fusion assessment without reference. Photogramm Eng Remote Sens 2008;74(2):193–200.
[31] Temiz H, Bilge HS. Super resolution of B-mode ultrasound images with deep learning. IEEE Access 2020;8:78808–20.

[32] Yang F, Xu W, Tian Y. Image super resolution using deep convolutional network based on topology aggregation structure. In: AIP conference proceedings. AIP Publishing LLC.; 2017, p. 020185.
[33] Temiz H, Tufekci A, Bilge HS. A comparative study on super resolution with deep learning. In: 26th IEEE signal processing and communications applications conference. 2018, p. 1–4.
[34] Temiz H. An experimental study on hyper parameters for training deep convolutional networks. In: 4th International symposium on multidisciplinary studies and innovative technologies. 2020, p. 1–8.