Appendices

The following appendix was taken from the Thesis Book of the work ”Implementation of the Enhanced Deep Super-Resolution (EDSR) Algorithm on pathological images for image generation applications” (2024) available in GitHub at: `https://github.com/giancarlocuticchia/Master-sThesis`. For more information please refer there.

Appendix A: Using the EDSR-PyTorch repository

The first thing to do as instructed inside the repository is to clone it: [2]

```
git clone https://github.com/thstkdgus35/EDSR-PyTorch
```

After doing so, we should expect a directory with the following structure:

```
EDSR-PyTorch/
├── experiment/
├── figs/
├── models/
├── src/
│   ├── data/
│   ├── loss/
│   ├── model/
│   ├── __init__.py
│   ├── dataloader.py
│   ├── demo.sh
│   ├── main.py
│   ├── option.py
│   ├── template.py
│   ├── trainer.py
│   ├── utility.py
│   └── videotester.py
├── test/
├── LICENSE
└── README.md
```

Our file of interest is main.py located in the folder EDSR-PyTorch/src. This is the file we are supposed to run on the terminal, by passing several arguments to it. This is the content of the file:

```python
import torch

import utility
import data
import model
import loss
from option import args
from trainer import Trainer

torch.manual_seed(args.seed)
checkpoint = utility.checkpoint(args)

def main():
    global model
    if args.data_test == ['video']:
        from videotester import VideoTester
        model = model.Model(args, checkpoint)
        t = VideoTester(args, model, checkpoint)
        t.test()
    else:
        if checkpoint.ok:
            loader = data.Data(args)
            _model = model.Model(args, checkpoint)
            _loss = loss.Loss(args, checkpoint) if not args.test_only else None
            t = Trainer(args, loader, _model, _loss, checkpoint)
            while not t.terminate():
                t.train()
                t.test()

            checkpoint.done()

if __name__ == '__main__':
    main()
```

Code 1: File main.py inside EDSR-PyTorch/src.

There are several things happening in the script, but of our interest are:

- A manual seed is provided to Pytorch, from the arguments provided to main.py.
- A "checkpoint" is created from the file utility.py, from the arguments.
- Given the argument data_test is not "video", then, if the checkpoint was successfully created:
  - A dataloader is created and assigned to "loader" from "Data" defined inside the __init__.py file in the src/data folder, according to the arguments.
  - The model is loaded and assigned to "_model" from "Model" defined inside the __init__.py file in the src/model folder, according to the arguments and the checkpoint.
  - The loss function is assigned to "_loss" from "Loss" defined inside the __init__.py file in the src/loss folder, according to the arguments and the checkpoint, unless the argument "test_only" is provided.
  - A "Trainer" is created and assigned to the variable "t", from "Trainer" defined in the "trainer.py" file inside the src folder, by using the arguments, the dataloader, model, loss function and the checkpoint.

- While the method "terminate()" of the Trainer is False, the methods "train()" and "test()" will run, corresponding to the training and testing of the model, until it finishes.
- After the training is completed, the method "done()" of the checkpoint is applied (to close the log file that it opened in its initialization), and the script ends.

Of our interest is then understand what does the checkpoint does, as well as how the dataloader is made, the model is loaded, the loss function is selected, and finally how does the Trainer work. We will go over each of them in the following.

A.1 checkpoint (src/utility.py)

There are several definitions inside the utility.py file. Of our particular interest, there is the class "checkpoint". When it's called, all of the content of the __init__ of the class will be executed, depending of the attributes inside of the arguments "args". Among these executions, of our interest are:

- If args.load is provided, will set as it's "self" directory *self.dir* the folder name provided in args.load (that is expected to be inside the "experiment" folder inside the EDSR-PyTorch directory), and will try to read the log of a possible previous training session, and continue the training from the last epoch.
- If it's not provided, but args.save is provided, will set as it's self directory self.dir the path to the folder provided in args.save (expected to be inside the "experiment" folder inside the EDSR-PyTorch directory). If args.save is not provided, will create a folder named as the current date and time inside the "experiment" folder and use it as self.dir.
- If args.reset is provided, will delete all the contents of the self.dir folder, if any.
- If self.dir doesn't exist, it is created.
- Will create the folder "model", and the files "log.txt" and "config.txt" inside self.dir.
- For each element in args.data_test with name "element_name" will create its corresponding folder named "results-element_name" inside self.dir. In this folder is where the super-resoluted images are stored if args.save_results is provided while testing the model.
- Opens the file "log.txt" to use it during the execution of the script and writes in the file "config.txt" all the arguments in args.

A.2 Data (src/data/__init__.py)

The first thing the script does after initializating the checkpoint is to set up images into dataloaders according to the specifications in the arguments passed to the main.py script. To do this, the Data class defined in the __init__.py file inside the "data" folder in the src directory will require some modules when it is initializated. Of our interest is the demo.py module, to test the model, and both srdata.py and div2k.py modules, to train it. All modules are located inside the "data" folder in the directory.

The demo.py module allow us to apply the specified model to images inside a specified folder, useful on a test-only run of the script. It is what we used to upscale images, and they can be saved on disk if the save_results argument is provided.

To train the model we could use the div2k.py module, which has defined a DIV2K class which is a subclass of the SRData class inside the srdata.py module, and it allow us to use the

DIV2K dataset [5] for training. However, we decided to took inspiration of the DIV2K class to define our own Custom class inside a custom.py module, to use our own dataset of images.

The detailed information of our interest for each of the above files, classes and modules, will be explained below.

__init__.py

When the Data class is initializated it will create a list of dataloaders for testing (as self.loader_test) and if the argument test_only is not provided, will also create a dataloader for training (as self.loader_train).

For both kind of dataloaders, the arguments pin_memory and num_workers provided will be assigned from the arguments. If the argument "cpu" is provided (to run the script using CPU only), then pin_memory will be set to False, and True otherwise. The value set to num_workers will be the same as the argument n_threads (which default is 6).

For the training dataloader, the argument batch_size will be the same as the provided in the arguments to the script, and the argument "shuffle" will be set as True. For the testing dataloaders they will be set to 1 and False respectively.

Finally, it will pass as the argument "dataset" the output of one of more of the modules inside of the "data" folder, according to the arguments passed to the script. For the training dataloader, it will pass a concatenation of the modules specified in the argument data_train (which again, of our interest would be the "div2k.py" and "custom.py" modules). For the testing dataloaders, it will pass the modules specified in the argument data_test (which of our interest would be the "demo.py" module).

demo.py

This module will simply search for all files ending in ".png" and ".jp" (PNG, JPG and JPEG images) inside the folder provided in the argument dir_demo, and when called will read the images with the library imageio and use the functions "set_channel" and "np2Tensor" from the common.py file in the "data" folder to properly return the images as a torch tensor to the dataloader.

Note: Only files inside the folder provided in the argument dir_demo will be considered. Other directories inside that folder won't be included.

srdata.py

The SRData class defined in the srdata.py module has several definitions, each of which are necessary at some point or the other during the execution of the script, and not only for the creation of the dataloaders. In summary, it will locate the images in disk upon initialization, place the paths to them into lists, and when asked to retrieve an element will load and return the image appropriately.

It is important to note that for this class to work, the desired images to be used for training and testing must be placed in disk in a particular directory tree, as seen in Figure 1 and described below. The original images (referred as High Resolution images, or HR) will be located in one folder, while their downscaled versions (referred as Low Resolution images, or LR) will be located in another folder set of folders, depending on the scale used to down-scale them (e.g. 2, 3 or 4).

```
<args.dir_data>/
└── <self.name>/ <---------------- (this is "apath")
    ├── bin/
    │   ├── HR/
    │   │   ├── image001.pt
    │   │   ├── ...
    │   │   ├── image007.pt
    │   │   └── ...
    │   └── LR_bicubic/
    │       ├── X2/
    │       │   └── ...
    │       ├── X3/
    │       │   └── ...
    │       └── X4/
    │           ├── image001x4.pt
    │           ├── ...
    │           ├── image007x4.pt
    │           └── ...
    ├── HR/
    │   ├── image001.png
    │   ├── ...
    │   ├── image007.png
    │   └── ...
    └── LR_bicubic/
        ├── X2/
        │   └── ...
        ├── X3/
        │   └── ...
        └── X4/
            ├── image001x4.png
            ├── ...
            ├── image007x4.png
            └── ...
```

Figure 1: Example of a directory tree expected for the SRData class in src/data/srdata.py.

The path to the main folder containing all the images is provided to the script with the argument dir_data (referred as args.dir_data). Inside this folder, for each different module of data used (for example, DIV2K or Custom) there should be an appropiate folder named as the module. When using the SRData class, this module name should be passed as the argument ”name” of the class, and will be assigned to the self-argument name (referred as self.name). Inside each self.name folder there should be at least two folders: one named ”HR” (to place the HR images) and another one named ”LR_bicubic” (to place the LR images, that are expected to be downscaled versions of the HR ones by using a bicubic algorithm).

Inside the ”LR_bicubic” folder there are expected to be folders for each different scale of which we want to train the model, e.g. 2, 3 or 4. Each of these folders must be named with an (uppercase) ”X” followed by the number of the scale, for example, ”X4” for a scale of ”4”. Inside of these folders there must be a down-scaled version of a HR image, corresponding to the proper scale, named exactly the same as the original one but with a (lowercase) ”x” and the scale number at the end of the name (and before the file extension). For example, if the HR image is named ”image007.png”, its corresponding down-scaled version for the x4 scale should be named "image007x4.png" and should be located inside the folder ./LR_bicubic/X4. All of this can be seen in the diagram of Figure 1.

Additionally, there could be a "bin" folder inside of the self.name folder, that will be produced by the script if the proper arguments are provided (as will be described below), but there is no need for the user to prepare it beforehand.

If the argument "ext" provided to the script includes "sep" (this argument will be referred as args.ext), the script will take the images (e.g. in PNG format) and store them somewhere in disk as binaries (by following the same directory tree structure described above, but inside the "bin" folder), by using the "dump" function from the "pickle" library, and eventually load them from there. This action is recommended by the author of the repository [2]. However, images can be loaded directly with imageio instead if args.ext is "img".

We will now describe some behaviours of the SRData class of what we considered of interest for our current work.

The initialization of the SRData class will then:

- Define some self-arguments depending on the arguments passed to the script. Of particular importance is the argument "name" that is passed down to the class when initilizated (referred as self.name), and in typical use it's given the name of the module of interest, e.g. "DIV2K" or "Custom".
- Set the directories needed to handle the image data. Will define a directory "apath" of a folder named as self.name, inside of the directory provided in args.dir_data. Inside this directory will define the folders "HR" and "LR_bicubic", described above. Additionally, will set a self-argument "ext" (referred as self.ext) to ('.png', '.png'), corresponding to the formats that the HR and LR images must have, respectively.
- Locate in disk the HR and LR images according to the expected directory structure, and file extension provided by self.ext.
- Make a list with the paths to the HR and LR images, named as list_hr and list_lr respectively, where the latter will have a list for each different scale of interest. This is done with the _scan method defined in the class.
- If args.ext includes "img", or if a benchmark is used, will simply assign list_hr and list_lr to self.images_hr and self.images_lr respectively. If not, will create a "bin" folder inside apath, and also if args.ext includes "sep", will:
  - Create (if not already present) folders "HR" and "LR_bicubic", with the corresponding subfolders for the scales, inside the "bin" folder, following the same directory structure as before.
  - For each HR image in list_hr and each LR image (for each scale) in list_lr, will try to find its corresponding binaries inside the "bin" folder, with the same filename but file extension ".pt". If the binaries are not found or if args.ext includes "reset", it will make and save the binaries of the image in its corresponding folder with its expected name and file extension. (This uses the "dump" function from the "pickle" library).
  - The paths of those binaries will be then stored in lists and assigned to self.images_hr and self.images_lr appropriately.
- Finally, if the argument "train" is passed as True, it will define a self-argument "repeat" (self.repeat) as explained below.

The self-argument self.repeat is defined as a way to extend the data during the training. It determines how many times per epoch the train dataloader will pass the image data. For example, if self.repeat = n, it means the whole set of HR images will be loaded n times and that the length of the train dataloader will be n times the number of HR images. This would be like having "n epochs per epoch", as confirmed by the author of the GitHub repository, Sanghyun Son, on a comment on the issues section of the repository on January 2nd, 2018 at `https://github.com/sanghyun-son/EDSR-PyTorch/issues/2` [2], by saying:

> "Also, I want to notice that the term epoch is not exactly same with other codes.
>
> Although I know what the word epoch means, one epoch in my code iterates over 20 * 800 images for some reason.
>
> (One reason is that we need more epochs when we randomly choose patches, as you mentioned. Also, many papers on super-resolution report their configurations using the term 'iterations', or 'updates', not epochs. In my code, one epoch is actually 20 epochs, and contains 1000 iterations(batches).)"

The value self.repeat will depend on the "number of patches" n_patches (number of images seen in training between tests), and the number of images n_images. The value n_patches depends on the batch size (args.batch_size), and in the number of batches we want to process between tests (args.test_every). The value n_images depends on the number of HR images for training (i.e. len(self.images_hr)) and in the number of different datasets for training in the arguments (i.e. len(args.data_train)). These values are then defined by these formulas:

n_patches = args.batch_size * args.test_every

n_images = len(args.data_train) * len(self.images_hr)

self.repeat = max(n_patches // n_images, 1)

where "//" corresponds to the integer division (a division returning only an integer value).

In addition to this, when __getitem__ for SRData is called, it will:

- Use the method _load_file to read the corresponding HR and LR image. If the argument "ext" provided to the script is "img" or if a benchmark is used, it will read the images with imageio. Else, if "ext" includes "sep" it will load the HR and LR images from their binaries with pickle, from their expected locations in disk, as described in the initialization.
- Use the method get_patch on the LR and HR images. If the argument self.train is True, it will pick a random square (patch) of side equal to the provided argument patch_size from the HR image, and its corresponding downscaled one from the LR image, and return them. If the provided argument no_augment is False (or not provided), then data augmentation on the patches will be applied, which consists in a horizontal flip, a vertical flip and a 90° rotation (anti-clockwise), each with an individual probability of 50% of being applied. If self.train is False, it will return as patches the whole LR and HR images respectively.
- Apply set_channel and np2Tensor from the common.py file in the "data" folder to the HR and LR patches, to properly set them as a torch tensor, and return them with their respective filename.

div2k.py

The class DIV2K defined is a subclass of the SRData class and hence inherits all its behaviours, with the following modifications:

- It takes the argument data_range as $start_{train} - end_{train} / start_{test} - end_{test}$ as the number of the images to take as the first and last images for the training and testing sets, respectively. If the argument test_only is provided, and only a pair of boundaries start/end is present, it will be assumed as the boundaries for the testing set.
- It modifies the _scan method to only add to the list of images the ones within the boundaries provided above.
- It modifies the expected paths of the "HR" and "LR_bicubic" folders, so they are named as "DIV2K_train_HR" and "DIV2K_train_LR_bicubic" instead. Note that the user must prepare this folders beforehand.

custom.py

This module defines a class Custom as a subclass of the SRData class, by taking inspiration of the DIV2K module, with the following modifications:

- It takes the argument data_range and processes it in the same way the DIV2K class does.
- It modifies the _scan method to only change the expected name of the LR images. Instead of a lowercase "x" followed by the scale number, it nows expected an underscore too before the "x", like this "_x". For example, if a HR image is named "image007.png", its corresponding down-scaled version for the x4 scale should be named "image007_x4.png".
- It uses the same directory tree as SRData, and doesn't modify the HR and LR_bicubic folder names like DIV2K does, but leaves it as SRData expects them.
- It keeps the self.ext argument to be ('.png', '.png'), which means that the HR and LR images are expected to be in PNG format.

The contents of custom.py can be seen in Code 2.

```python
import glob
import os
from data import srdata

class Custom(srdata.SRData):
    def __init__(self, args, name='Custom', train=True, benchmark=False):
        data_range = [r.split('-') for r in args.data_range.split('/')]
        if train:
            data_range = data_range[0]
        else:
            if args.test_only and len(data_range) == 1:
                data_range = data_range[0]
            else:
                data_range = data_range[1]

        self.begin, self.end = list(map(lambda x: int(x), data_range))
        super(Custom, self).__init__(
            args, name=name, train=train, benchmark=benchmark
        )

    def _scan(self):
        names_hr = sorted(
            glob.glob(os.path.join(self.dir_hr, '*' + self.ext[0]))
        )
        names_lr = [[] for _ in self.scale]
        for f in names_hr:
            filename, _ = os.path.splitext(os.path.basename(f))
            for si, s in enumerate(self.scale):
                names_lr[si].append(os.path.join(
                    self.dir_lr, 'X{}/{}_x{}{}'.format(
                        s, filename, s, self.ext[1]
                    )
                ))

        names_hr = names_hr[self.begin - 1:self.end]
        names_lr = [n[self.begin - 1:self.end] for n in names_lr]

        return names_hr, names_lr

    def _set_filesystem(self, dir_data):
        super(Custom, self)._set_filesystem(dir_data)
        self.ext = ('.png', '.png')
```
Code 2: File custom.py inside EDSR-PyTorch/src/data. This file was made in this work and it's not present in the repository made by Sanghyun Son [2].

A.3 Model (src/model/__init__.py)

The class Model defined is a subclass of "Module" from the torch library "nn". At initialization it will:

- Set several self arguments from the arguments provided to the script.
- Set the torch device in this order: mps, cuda, cpu, depending of availability, unless the argument "cpu" is provided as True, in which case the device will be set to "cpu" directly.
- Import the corresponding module to be used for the model from the "model" folder inside the folder "src", which in our case is edsr.py for the EDSR model. It will then apply the function make_model defined in the module and assign it to the self argument "model" after putting it on the corresponding torch device.
- Load the state dictory of the model, depending on the argument "resume":
  - If resume=0, and the argument "pre_train" is "download", it will download and use the state dictionary according to the model parameters from a list of available files located in the edsr.py file (more details explained below). If desired, the state dictionary of a pre-trained model can be provided by passing the path to the corresponding .pt file as the argument "pre_train". If there is no "pre_train" argument, no state dictionary will be loaded to the model.
  - If resume=n, it will load the state dictionary from a file "model_n.pt" located inside the "model" folder inside the self.dir directory created by the checkpoint during the script execution. If n=-1, it will load "model_latest.pt" instead.

- Write all the model information on the log.txt file created by the checkpoint during the script execution.

edsr.py

The function make_model will call the class EDSR by passing it the provided arguments. The class EDSR is a subclass of "Module" from the torch library "nn". At initialization it will:

- Set several self arguments from the arguments provided to the script.
- Make the key for the dictionary "url" containing the links to download the state dictionary files needed if args.pre_train is "download". This requires the arguments n_resblocks (the number of Residual Blocks), n_feats (the number of Feature Maps) and the scale (if more than one is provided, the first one is used). Note that:
  - If n_resblocks=16 and n_feats=64, the model corresponds to what the author called EDSR_baseline.
  - If n_resblocks=32 and n_feats=256, the model corresponds to the EDSR the author proposed.
- Define the "sub_mean" and "add_mean" layers of the model, by using the MeanShift function on the common.py file inside the src/model folder.
- Define the "head", "body" and "tail" layers of the model, according to the model structure, by using "Sequential" from torch.nn, and the functions default_conv, ResBlock and Upsampler defined in the common.py file inside the src/model folder. This also requires the arguments n_colors (the number of color channels to use, which default is 3) and res_scale (the residual scaling, which default is 1 but authors recommend it to be 0.1).

A.4 Loss (src/loss/__init__.py)

The class Loss is a reclass of nn.modules.loss._Loss from torch. At initialization it will:

- Set several self arguments from the arguments provided to the script.
- Read the provided argument "loss" (args.loss), and extract the information regarding to the "type", "weight" and "function" from each loss configuration provided in args.loss and append them in dictionary-form to the self-argument "loss" (self.loss). Each loss configuration provided is separated from the other by using the plus symbol '+' following the structure: $args.loss = n_1 * type_1 + n_2 * type_2 + ...$, so for each configuration:
  - type = type
  - weight = n
  - function = loss_function (which depends on the input parameters)

  The options for loss_function are:
  - nn.MSELoss() (from torch), if type is "MSE"
  - nn.L1Loss() (from torch), if type is "L1"
  - The module "loss.vgg" (from vgg.py inside the src/loss folder), if type includes "VGG"

  - The module "loss.adversarial" (from adversarial.py inside the src/loss folder), if type includes "GAN"

- For each loss configuration processed this way, will print the weight and type, and append the loss_function to the self-argument loss_module (which was defined by using nn.ModuleList() from torch).
- Defines a self-argument "log" as a torch tensor, to be use as a log for the loss function.
- Set the device as "cpu" if the argument args.cpu is True, or as "cuda" if not. Then will set self.loss_module to this device.
- If the provided argument args.cpu is not provided (or if it's False) and the provided argument args.n_GPUs is bigger than 1, will distribute the application of self.loss_module in different parallel applications for each args.n_GPUs, by using nn.DataParallel from torch.
- If the argument args.load is provided, will load the state dictionary for the lss from the "loss.pt" file, and the log for the loss from the file "loss_log.pt", both located inside the self.dir directory created by the checkpoint during the execution of the script.

The default loss configuration is "1*L1".

A.5 Trainer (src/trainer.py)

The trainer is the responsible to perform the training and testing of the model. Inside of the file trainer.py is defined the class Trainer. Of particular importance are the methods train(), test() and terminate(), which will dictate the behaviour of the script. At initialization, this class will it will:

- Set several self arguments from the arguments provided to the script, including the train and test dataloaders, the model, the loss function and the checkpoint.
- If the argument args.load is provided, will load the state dictionary of the optimizer from the file "optimizer.pt" in the self.dir directory of the checkpoint, and set the corresponding epoch based on it.

train()

This method will:

- Step up the loss (by calling its corresponding method: self.loss.step())
- Get the last epoch and learning rate from the optimizer, move to the next epoch and write it on log file of the checkpoint.
- Start the log from the loss, put the model in training mode and start timers to measure the time taking by processing involving the use of the data and the model.
- For each batch generated by the train dataloader, will:
  - Put the LR and HR image (or patch) in the proper device.
  - Set to zero the gradients on the optimizer (by calling its corresponding method).

  - Apply the model on the LR image (forward pass). This will produce a SR (Super-Resolution) image.
  - Calculate the loss between the SR image and the HR image.
  - Performs backpropagation on the loss (by calling its corresponding method).
  - Step up the optimizer (by calling its corresponding method).
  - Depending on the current batch and the provided argument args.print_every, will write on the log of the checkpoint information regarding the batch, dataset, loss and elapsed times.

- End the log, store the last error obtained in the loss and schedule the optimizer.

test()

This method will:

- Disable the gradients (by using torch.set_grad_enabled(False)).
- Get the last epoch from the optimizer.
- Print "Evaluation" and write on the log of the checkpoint.
- Put the model in evaluation mode and start a timer to measure the testing time.
- If the provided argument args.save_results is True, will start processes in the background (using methods from the checkpoint) needed to save the resulted images.
- For each different dataset loaded into the test dataloder and for each scale (provided in args.scale), will set the corresponding scale for the dataset, and for each LR and HR images in that dataset, will:
  - Put the LR and HR image in the proper device.
  - Apply the model on the LR image (forward pass). This will produce a SR (Super-Resolution) image.
  - Apply the method "quantize" from utility.py (inside the src folder) on the SR image (this requires the provided argument args.rgb_range).
  - Calculate the PSNR between SR and the HR image, using the function "calc_psnr" from utility.py (this requires the scale, args.rgb_range and which dataset is being used, just in case a benchmark is employed). This value is then accumulated with further ones in a single sum to use later.
  - If the provided argument args.save_results is True, it will save the SR image by using the save_results function from the checkpoint. If the provided argument args.save_gt is also True, it will save the LR and HR images as well. For more information, read the note below.
- After all LR and HR images were processed for the corresponding dataset and scale, will:
  - Divide the accumulated sum of the PSNR by the number of images, to get the average PSNR for the current epoch.
  - Read from the log the best value of PSNR obtained for all epochs that has passed.

  - Write on the log the obtained average PSNR value for the current epoch, next to the dataset name, scale value, and the best overall PSNR obtained and in which epoch it was obtained.
  - Move to the next scale (if any). After all scales, move to the next dataset (if any).

- Will write on the log the elapsed time.
- If the provided argument args.save_results is True, will close the operations in the background (by calling its corresponding method from the checkpoint).

Note: The images saved if args.save_results is True will be in PNG format and have the following filename structure: imagename_xS_postfix.png. Here "imagename" is the original filename of the image, "xS" is a lowercase "x" followed by the number of the scale used (i.e. 2, 3 or 4) and "postfix" would be "SR" for the super-resolution image, "LR" for the low-resolution image, and "HR" for the high-resolution image. The images will be stored in disk inside the "results-element_name" folder in the self.dir folder created by the checkpoint, where "element_name" corresponds to element from args.data_test that is being tested on, as explained in the appendix A.1.

If the argument args.test_only is False or not provided, it will apply the "save" function from the checkpoint, that will:

- Apply the "save" function from the model. This will:
  - Save the state dictionary of the model as the "model_latest.pt" file inside the self.dir folder made by the checkpoint, for the current epoch.
  - If the current best value of PSNR is obtained, it will save the state dictionary of the model as "model_best.py".
  - If the provided argument args.save_models is True, it will save the state dictionary of the model for each epoch as "model_n.pt", where "n" is the number of the current corresponding epoch.
- Apply the "save" function from the loss. This will save the state dictionary of the loss in "loss.pt" and its log in "loss_log.pt", both located inside the self.dir folder made by the checkpoint.
- Apply the "plot_loss" method from the loss. This will make a plot of the value of the loss respect the number of epochs and save it as "loss_losstype.pdf" inside the self.dir folder made by the checkpoint, for each "losstype" provided in the arguments to the Loss.
- Plot the value of the PSNR respect the number of epochs and save it as "test_element.pdf" inside the self.dir folder made by the checkpoint, for each "element" provided in the arguments to args.data_test.
- Apply the "save" function from the optimizer. This will save the sate dictionary of the optimizer as "optimizer.pt" inside the self.dir folder made by the checkpoint.
- Save the log from the checkpoint as "psnr_log.pt" inside the self.dir folder made by the checkpoint.

After all of that, will write the total elapsed time in the log file of the checkpoint and enable back the gradients with torch (by using torch.set_grad_enabled(True)).

terminate()

If the provided argument args.test_only is True, it will apply test() from the Trainer and then return True (which will then end the loop in the main.py script).

If not, it will get the last epoch from the optimizer, and while that epoch value is strictly lower than the value provided in the argument args.epoch, it will return False, which will continue the loop in the main.py script. Otherwise it will return True, efectively ending the loop in the main.py script.

Note: If for example args.epoch=300, we will get training for values of epoch from 1 to 299 (both included).

A.6 How to use the model

In order to test or train the model, we must:

- Have the GitHub repository where we are going to use it (for example, cloned or downloaded), as well as any other custom files and modules, if we are also going to use those.
- Prepare the image dataset beforehand. In particular for training, the images of our interest must be of a particular file extension and be placed inside folders with a particular directory structure, as explained in appendix A.2 (in particular, we can see Figure 1).
- If we desire to train an already trained model, or if we want to test a model, we need the state dictionary of the pretrained model, and provide the path to its file on the pre_train argument to the script. If we want to train the model from scratch, we can leave this argument empty.
- Carefully select the arguments to provide to the script. We can set-up a template with some parameters by modifying the template.py file in src directory, or provide the arguments directly on the command line on the terminal.
- Of particular importance in what regards the data, we should provide:
    - dir_demo: Path to the folder containing the images we want to super-resolute, while using the "Demo" module for testing.
    - dir_data: Path to the main folder containing all the image data.
    - data_train = Name of the module to use for the training dataloader (there must be a folder with this name inside of dir_data).
    - data_test = Name of the module to use for the testing dataloader (there must be a folder with this name inside of dir_data).
    - data_range: Range of data to use for training and testing in the format: $start_{train} - end_{train}/start_{test} - end_{test}$ as the number of the images to take as the first and last images for the training and testing sets, respectively.
    - ext: To indicate the script if it must use the images directly or convert them into binaries.
- Of particular importance in what regards the model, we should provide:
    - scale: Upscaling scale. It can be either 2, 3 or 4, or even more than one. If more than one is provided, they must be separated with a plus symbol (+), for example: 2+3+4.

  - model: Which model module to use. In our case, it is EDSR.
  - n_resblocks: Number of residual blocks. For EDSR, it is 32, according to its authors [1].
  - n_feats: Number of feature maps. For EDSR, it is 256, according to its authors [1].
  - res_scale: Value for the residual scaling. According to its authors, the value 0.1 provide best results [1].

- Of particular importance in what regards the training, we should provide:
  - test_every: Number of batches processed between tests. Default is 1000.
  - epochs: Number of epochs for the training.
  - batch_size: Number of images per batch. Default is 16.

- Other optional parameters, such as:
  - test_only: As "True", if we only want to test the model.
  - save_results: As "True", if we want to store in disk the super-resoluted images.
  - save: Name of the folder to make and store all the files the training will produce while running the main.py script. This folder will be inside of the "experiment" folder of the repository.
  - template: The name of the template we want to use (from the template.pt file in the src folder).
  - save_models: As "True", if we want to save the state dictionaries of the model for each epoch, instead of only the last one and the best one.
  - chop: As "True" to enable a "memory efficient forward" of the model during training. We were required to use as we were running out of memory during the Evaluation part of the training.*
  - Parameters we constructed for Transfer Learning and saving the state dictionaries of the model during training, that we will discuss in sections below.

- On a terminal, and by having the "src" folder of our own EDSR-PyTorch repository as our current working directory, run an appropriate command line for either testing or training.

Testing example

We can simply test the model by using the Demo module on the image 0853x4.png image provided in the "test" folder of the repository. We will also need, for EDSR x4, the state dictionary file edsr_x4-4f62e9ef.pt provided by the authors, which we can obtained from `https://cv.snu.ac.kr/research/EDSR/models/edsr_x4-4f62e9ef.pt`. Then, we can just run the following command line on a terminal (from the src folder as working directory):

```
python main.py --data_test Demo --scale 4 --save edsr_x4_test
    --n_resblocks 32 --n_feats 256 --res_scale 0.1 --pre_train
    "../pre-train/edsr_x4-4f62e9ef.pt" --test_only --save_results
```

By doing so, we can also expect something like the following to be printed on the terminal:

*Using the "chop" argument was a recommendation the author of the GitHub repository, Sanghyun Son, on a comment on the issues section of the repository on April 23rd, 2019 at `https://github.com/sanghyun-son/EDSR-PyTorch/issues/154`.

```
Making model...
Load the model from ../pre-train/edsr_x4-4f62e9ef.pt

Evaluation:
100%|██████████████████████████████████████| 1/1 [00:05<00:00,  5.63s/it]
[Demo x4]       PSNR: 0.000 (Best: 0.000 @epoch 1)
Forward: 5.77s

Saving...
Total: 6.97s
```

From where we can see:

- The progress bar of the Evaluation process (from 0 to 100%, in real time).
- The current elapsed time in the Evaluation process and the estimated remaining time (shown as 00:05 and 00:00 in the example, respectively, in a minutes:seconds format).
- The expected time required per iteration during the Evaluation process, in seconds per iteration (5.63s/it in the example).
- The data module used and the scale (Demo x4 in the example).
- The average value of PSNR obtained in the Evaluation (0 in the example), next to the best one obtained in all the current training and in which epoch (0 and epoch 1, in the example).
- The elapsed time for the Evaluation process in seconds (5.77s in the example) and the total time elapsed of the Evaluation including saving, also in seconds (6.97s in the example).

We can also expect a folder named "edsr_x4_test" inside of the "experiment" folder of the repository (as we provided in the "save" argument), with the following structure:

```
EDSR-PyTorch/
 ├─ experiment/
 │   └─ edsr_x4_test/
 │       ├─ model/
 │       ├─ results-Demo/
 │       │   └─ 0853x4_x4_SR.png
 │       ├─ config.txt
 │       └─ log.txt
 ├─ figs/
 ├─ pre-train/
 │   └─ edsr_x4-4f62e9ef.pt
 ├─ src/
 ├─ test/
 │   └─ 0853x4.png
 ├─ LICENSE
 └─ README.md
```

Where the file 0853x4_x4_SR.png inside of the "results-Demo" folder in the "edsr_x4_test" folder is the super-resoluted image after a x4 upscaling of the 0853x4.png image, by using the EDSR x4 model.

Training example

In order to train the model, we need to first prepare the dataset, get a pretrained model and prepare the arguments to provide on a template inside of the template.py file. Additionally, we are going to use the Custom module we explained in the section custom.py on appendix A.2.

For the dataset, we prepared 2500 images by following the same directory structure we described in the section custom.py on appendix A.2, similar to the one shown in Figure 1. We place it inside a folder ”image-data” inside the EDSR-PyTorch directory, as follow:

```
EDSR-PyTorch/
├── experiment/
├── figs/
├── image-data/
│   └── Custom/
│       ├── HR/
│       │   ├── image0001.png
│       │   ├── ...
│       │   └── image2500.png
│       └── LR_bicubic/
│           └── X4/
│               ├── image0001_x4.png
│               ├── ...
│               └── image2500_x4.png
├── pre-train/
├── src/
├── test/
├── LICENSE
└── README.md
```

We placed the state dictionary file edsr_x4-4f62e9ef.pt provided by the authors (pretrained EDSRx4, obtainable from `https://cv.snu.ac.kr/research/EDSR/models/edsr_x4-4f62e9ef.pt`) inside a ”pre-train” folder in the EDSR-PyTorch directory, and we also placed the custom.py file from A.2 inside of the ”data” folder inside the ”src” folder of the EDSR-PyTorch directory. Finally, we added the following lines to the template.py file inside the ”src” folder:

```python
    if args.template.find('EDSR_custom') >= 0:
        args.dir_data =  "../image-data"
        args.data_train = "Custom"
        args.data_test = "Custom"
        args.data_range = "1-2400/2401-2500"
        args.ext = "sep"
        args.scale = "4"
        args.model = "EDSR"
        args.pre_train = "../pre-train/edsr_x4-4f62e9ef.pt"
        args.n_resblocks = 32
        args.n_feats = 256
        args.res_scale = 0.1
        args.test_every = 100
        args.epochs = 11
        args.batch_size = 16
        args.save = "edsr_x4_train"
```

Our directory should then look like this:

```
EDSR-PyTorch/
├── experiment/
├── figs/
├── image-data/
│   └── Custom/
│       ├── HR/
│       │   ├── image0001.png
│       │   ├── ...
│       │   └── image2500.png
│       └── LR_bicubic/
│           └── X4/
│               ├── image0001_x4.png
│               ├── ...
│               └── image2500_x4.png
├── pre-train/
│   └── edsr_x4-4f62e9ef.pt
├── src/
│   ├── data/
│   │   ├── ...
│   │   └── custom.py
│   ├── loss/
│   ├── model/
│   ├── ...
│   ├── main.py
│   └── template.py
├── test/
├── LICENSE
└── README.md
```

Figure 2: Folder structure of the EDSR-PyTorch directory prepared for training using the Custom module explained in appendix A.2.

Then, to perform the training we can just run the following command line on a terminal (from the src folder as working directory):

```
python main.py --template EDSR_custom --save_models --chop
```

After doing so, the first thing we will see the main.py script to do is to create the binaries for the images, if they are not already present. We can expect something like the following to be printed on the terminal:

```
Making a binary: ../image-data/Custom/bin/HR/image_0001.pt
...
Making a binary: ../image-data/Custom/bin/HR/image_2500.pt
Making a binary: ../image-data/Custom/bin/LR_bicubic/X4/image_0001_x4.pt
...
Making a binary: ../image-data/Custom/bin/LR_bicubic/X4/image_2500_x4.pt
```

After that, the script will make the model, load the state dictionary file of the pretrained model, prepare the loss, and start the training for each epoch. We can expect something like the following to be printed on the terminal:

```
Making model...
```

```
Load the model from ../pre-train/edsr_x4-4f62e9ef.pt
Preparing loss function:
1.000 * L1

[Epoch 1]       Learning rate: 1.00e-4

[1600/2400]     [L1: 8.8694]    50.1+80.7s

Evaluation:
100%|██████████████████████████████████████| 100/100 [09:39<00:00,
    5.79s/it]
[Custom x4]     PSNR: 26.497 (Best: 26.497 @epoch 1)
Forward: 579.41s

Saving...
Total: 586.38s


[Epoch 2] ...
```

Where it is mentioned the path from where it is taking the state dictionary file to use as pretrained model, and what type of Loss function configuration it will use (in this case: type L1 and weight 1.0), and for each Epoch it will show:

- The learning rate (0.0001 in this case).
- For whenever the current batch times the batch size gives a number multiple of the argument print_every[†], for that batch, will show:
  - The number of images (or patches) processed until that point (1600 in the example).
  - The length of the train dataloader[‡] (2400 in the example, which in this case it's the same as the number of images given in the training dataset).
  - The value obtained for the loss for the current printed batch (L1: 8.8694 in the example).
  - The time registered by the model timer, in seconds (50.1 in the example).
  - The time registered by the data timer, in seconds (80.7 in the example).
- The progress bar of the Evaluation process (from 0 to 100%, in real time).
- The current elapsed time in the Evaluation process and the estimated remaining time (shown as 09:39 and 00:00 in the example, respectively).
- The expected time required per iteration during the Evaluation process, in seconds per iteration (5.79s/it in the example).
- The data module used and the scale (Custom x4 in the example).

[†]The actual formula is $(batch + 1)\%print\_every == 0$.

[‡]Which for training it is equal to the number of HR images provided for the train dataset times the "repeat" value explained in the srdata.py file section on appendix A.2.

- The average value of PSNR obtained in the Evaluation (26.497 in the example), next to the best one obtained in all the current training and in which epoch (26.497 and epoch 1, in the example).
- The elapsed time for the Evaluation process in seconds (579.41s in the example) and the total time elapsed of the Evaluation including saving, also in seconds (586.38s in the example).

Note: The number of printed batches that are shown will depend then on the arguments batch_size and print_every. The length of the training dataloader depends on the "repeat" value (depending also in the batch_size, the argument test_every and the number of HR images provided, as shown in the srdata.py file section on appendix A.2). If the length of the training dataloader is not a multiple of the criteria for printing, the last set of batches (from the previous printing to the end) won't be shown and hence the time required won't be registered. Like in the example, the elapsed time during the training for images from 1601 to 2400 is not shown (which for this case were on average around 70 seconds). The whole training took around 2 hours and 13 minutes.

After the last epoch, and the training is completed, we should expect a folder named "edsr_x4_train" (as we named it in the template) inside of the "experiment" folder of the EDSR-PyTorch directory, with the following structure:

```
EDSR-PyTorch/
└─ experiment/
   ├─ model/
   │  ├─ model_1.pt
   │  ├─ model_2.pt
   │  ├─ ...
   │  ├─ model_10.pt
   │  ├─ model_best.pt
   │  └─ model_latest.pt
   ├─ results-Custom/
   ├─ config.txt
   ├─ log.txt
   ├─ loss.pt
   ├─ loss_L1.pdf
   ├─ loss_log.pt
   ├─ optimizer.pt
   ├─ psnr_log.pt
   └─ test_Custom.pdf
```

Where:

- Inside the "model" folder we have the state dictionary files for the latest trained epoch and for the epoch which performed better during the Evaluation (measuring PSNR), named model_latest.pt and model_best.pt respectively. Additionally, since we provided the argument "save_models" as True, we are also getting the state dictionary files for each epoch as well, labeled accordingly.
- The "results-Custom" folder is empty, as we didn't provide the "save_results" argument (to save the SR images).
- The config.txt file contains all the parameters fetched to the script.

- The log.txt file contains all the information handled by the log created by the checkpoint (more information on appendix A.1). (Example of this can be seen in our GitHub repository).
- The loss.pt file is the state dictionary file of the Loss, loss_L1.pdf is a plot of the value of the Loss for each epoch, and loss_log.pt is the log file of the loss.
- The optimizer.pt file is the state dictionary file of the optimizer.
- The psnr_log.pt is the log file of the PSNR measurements and the test_Custom.pdf file is a plot of the PSNR measured for each epoch.

A.7 Freezing layers during training

In order to perform Transfer Learning, i.e. to select which layers of the model we want to train, first we need to study the model structure in the code. We can start by loading the model like the following:

```python
import torch

import utility
import model

torch.manual_seed(args.seed)
checkpoint = utility.checkpoint(args)

# Loading the model
model = model.Model(args, checkpoint)
```

We note that we also had to provide the arguments "args", which must be properly set beforehand (an example of how to do it can be seen in our GitHub repository).

After loading the model, we can print it's named parameters with the following code:

```python
for name,value in model.named_parameters():
  print(name)
```

An example of this print can be seen in our GitHub repository. There we can see that there are 5 categories of named parameters, each corresponding of each big layer group of the model:

- sub_mean: The weight and the bias of the MeanShift layer of the sub_mean. Total of parameters: 2.
- add_mean: The weight and the bias of the MeanShift layer of the add_mean. Total of parameters: 2.
- head: The weight and the bias of the Sequential (Conv2d) layer of the head. Total of parameters: 2.
- body: The weight and the bias of both Conv2d layers (one labeled 0 and the other labeled 2) present on each of the 32 ResBlocks of the body (each labeled from 0 to 31) and the weight and the bias of the final Conv2d layer (labeled as 32), of the body. There are then 4 parameters per ResBlock, and a total of 130 parameters for the body (including the last Conv2d layer).
- tail: The weight and the bias of both of the Conv2d layers (labeled 0 and 2) of the Upsampler (labeled 0) and the final Conv2d layer (labeled 1) of the tail. Total of parameters: 6.

Being a total of 142 named parameters in the model. In order to freeze them, i.e. to set their corresponding requires_grad to False, we created several arguments to be added to the option.py file of the repository, to provide instructions to the script on how to use the freeze_model function from the freeze.py module to perform the actions mentioned above. These arguments are:

- param_to_freeze: List with the name of the named parameters to freeze. Options are: "sub_mean", "add_mean", "head", "body" and "tail". Default is empty. It is expected to be a string, where more than one parameter can be provided while separated by a plus (+) symbol. E.g.: if the argument is "sub_mean+add_mean+head+body+tail", all 5 group of layers will be frozen.
- body_to_freeze: Range of sub-layers of the layer "body" to be frozen. It is expected to be a string, where a range from "a" to "b" is denoted "a-b", and multiple ranges and individual positions can be provided and separated with a plus (+) symbol. Default is empty. E.g.: if the argument is "1-10+21-30+32", then sub-layers labeled from 1 to 10, from 21 to 30 and as 32, will be frozen.
- tail_to_freeze: Range of sub-layers of the layer "tail" to be frozen. Options are "0", "1" or "0-1". Default is empty. E.g.: if the argument is "0-1", then both sub-layers labeled as 0 and 1 will be frozen.
- print_frozen_param: If provided (as True) it will print the total number of named parameters and the total number of frozen parameters.
- torchinfo_summary: If provided (as True) it will show an output of summary from torchinfo with the modified model after the freezing.
- torchinfo_inputsize: Parameter required for the function summary from torchinfo. It is expected to be a string formated as "width,height" to be the input size to be used for summary. Default is: "510,339".[§]

We had to include lines of code in the option.py file in order to be able to pass these arguments to the main.py script. (We mentioned how we add those lines of code in our GitHub repository). We also modified the main.py in order to use the freeze.py module, by adding the following lines:

```python
import freeze
            _model = freeze.freeze_model(_model, args)
```

Where the second line was inserted in the line number 23, and then the first line was inserted in the line number 8, of the main.py file.

After that, we can perform a training session by freezing some layers of the model by using a command line like the following:

```
python main.py --template EDSR_custom --param_to_freeze
    sub_mean+add_mean+head+body --body_to_freeze "0-32"
    --print_frozen_param --save_models --chop
```

[§]These are the dimensions of the image 0853x4.png inside of the "test" folder of the EDSR-PyTorch repository. They are provided as default just as an example.

A.8 New argument save_models_each

Given the fact that properly training the model could take up to a few hundred epochs, and that each state dictionary file of the EDSR x4 model can be of size around 160 MB, it became undesirable, in terms of disk space, to store the state dictionary file of the model for each epoch of training if the argument save_models was provided. However, sticking only to the last and best cases of the training seemed insufficient, in case we wanted to have access to the state of the model in a particular stage of the training.

To overcome this issue, we defined an argument "save_models_each" (default: 1) to be able to have more control on how many state dictionary files we would store on disk during training. For example, if we train for 30 epochs and we provide save_models_each = 5 (given the fact we also provided the argument save_models), we would store in disk the state dictionary of the model during training for epochs 5, 10, 15, 20, 25 and 30, instead of for every single epoch.

To do this, we made some modifications to the option.py file (to add the corresponding new argument), as well to some modifications to the __init__.py file of the "model" folder in the src folder of the repository. The modification to the option.py file and to the __init__.py file are shown in our GitHub repository.

With this, we can perform a training session and save models as desired by providing a command line like the following:

```
python main.py --template EDSR_custom --save_models --save_models_each 5
    --chop
```

A.9 Minimal use of main.py

In order to have a simple way to apply a trained EDSR model to a set of images to get the Super-Resolution upscalings, we modified the original main.py file for what we consider is a minimal version, i.e., using the minimal required libraries and modules from the repository, as well as reducing the amount of actions made by main.py (changing printings on screen, the creation of folders and files, or running processes in the background). We did this as a way to simplify our work, and not with the intention to correct or replace in any away the original labor of the authors of the repository, which work consists in many more applications that the ones employed in the present work.

We then produced a new script main_use.py and placed it inside the src folder of the repository. The contents of main_use.py can be seen in our GitHub repository. To use the script, we just need to run on a terminal (with the src folder as working dictionary) the following command:

```
python main_use.py --data_test Demo --dir_demo {dir_demo} --scale 4
    --n_resblocks 32 --n_feats 256 --res_scale 0.1 --pre_train
    {pretrain_path} --test_only --save_results
```

Where we must provided the following arguments:

- dir_demo : Path to the folder containing the images we want to upscale (in either PNG or JPG format).
- pretrain_path : Path to the state dictionary file of the EDSR x4 model we want to use to upscale the images.

The script will load the images into the dataloader, make the EDSR x4 model with weights and biases from the state dictionary provided, and upscale the images. The resulting Super-Resoluted images will be named with the same name as their low-resolution counterparts with the added "_x4" at the end, and stored in PNG format in a folder named "results" inside the directory provided as dir_demo.

Note: A log file log.txt will be also created inside dir_demo. This is because the __init__.py file of the model folder inside the src folder of the repository, in its line 45, makes a print of the model in the log file. This could have been avoided by simply commenting (or removing) such print line from the __init__.py file, but we decided to keep it in case any user would like to be able to use the main_use.py file without further tempering with the original code of the repository. If such log.txt file is undesirable, the user could just comment line 45 from the __init__.py file and lines 30 and 31 from main_use.py.

Bibliography

[1] Bee Lim, Sanghyun Son, Heewon Kim, Seungjun Nah, and Kyoung Mu Lee, ”Enhanced Deep Residual Networks for Single Image Super-Resolution,” 2nd NTIRE: New Trends in Image Restoration and Enhancement workshop and challenge on image super-resolution in conjunction with CVPR 2017. PyTorch implementation at: https://github.com/sanghyun-son/EDSR-PyTorch

[2] Author: Sanghyun Son. Repository name: EDSR-PyTorch. ”PyTorch version of the paper ’Enhanced Deep Residual Networks for Single Image Super-Resolution’ (CVPRW 2017)”. Url: https://github.com/sanghyun-son/EDSR-PyTorch. Accessed from April 2022 until January 2024.

[3] Paszke A., Gross S., Massa F., Lerer A., Bradbury J., Chanan G., Killeen T., Lin Z., Gimelshein N., Antiga L., Desmaison A., Kopf A., Yang E., DeVito Z., Raison M., Tejani A., Chilamkurthy S., Steiner B., Fang L., Bai J., & Chintala S. (2019). PyTorch: An Imperative Style, High-Performance Deep Learning Library [Conference paper]. Advances in Neural Information Processing Systems 32, 8024–8035. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[4] Yep, T. (2020). torchinfo [Computer software]. https://github.com/TylerYep/torchinfo

[5] E. Agustsson and R. Timofte, ”NTIRE 2017 Challenge on Single Image Super-Resolution: Dataset and Study,” 2017 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW), Honolulu, HI, USA, 2017, pp. 1122-1131, doi: 10.1109/CVPRW.2017.150.

[6] K. He, X. Zhang, S. Ren, and J. Sun. ”Deep residual learning for image recognition”. In CVPR 2016.

[7] C. Ledig, L. Theis, F. Huszár, J. Caballero, A. Cunningham, A. Acosta, A. Aitken, A. Tejani, J. Totz, Z. Wang, et al. ”Photo-realistic single image super-resolution using a generative adversarial network”. arXiv:1609.04802, 2016.

[8] Bradski, G. (2000). The OpenCV Library. Dr. Dobb’s Journal of Software Tools. https://opencv.org/

[9] OpenSlide. C library that provides a simple interface to read whole-slide images (also known as virtual slides). Python API. https://openslide.org/

[10] Roy, Prasun and Ghosh, Subhankar and Bhattacharya, Saumik and Pal, Umapada. (2018). Effects of Degradations on Deep Neural Network Architectures. Journal: *arXiv preprint arXiv:1807.10108*. Available at: https://www.kaggle.com/datasets/prasunroy/natural-images?datasetId=42780

[11] Google Colaboratory, Google Colab. Website: https://colab.research.google.com/. Documentation available at: https://cloud.google.com/compute/docs.

[12] Uwe Schmidt et al. "Cell Detection with Star-Convex Polygons". In: Medical Image Computing and Computer Assisted Intervention - MICCAI 2018 - 21st International Conference, Granada, Spain, September 16-20, 2018, Proceedings, Part II. 2018, pp. 265–273. doi: 10.1007/978-3-030-00934-2_3010.1007/978-3-030-00934-2_30 (cit. on p. 11).

[13] Martin Weigert et al. "Star-convex Polyhedra for 3D Object Detection and Segmentation in Microscopy". In: The IEEE Winter Conference on Applications of Computer Vision (WACV). Mar. 2020. doi: 10.1109/WACV45572.2020.909343510.1109/WACV45572.2020.9093435 (cit. on p. 11).

[14] Martin Weigert and Uwe Schmidt. "Nuclei Instance Segmentation and Classification in Histopathology Images with Stardist". In: The IEEE International Symposium on Biomedical Imaging Challenges (ISBIC). 2022. doi: 10.1109/ISBIC56247.2022.985453410.1109/ISBIC56247.2022.9854534 (cit. on p. 11).

[15] Author: StarDist. Repository name: stardist. "StarDist - Object Detection with Star-convex Shapes". Url: `https://github.com/stardist/stardist`. Accessed from April 2024 until May 2024.

[16] Jonathan Ho, Ajay Jain, and Pieter Abbeel. "Denoising Diffusion Probabilistic Models". In: CoRR abs/2006.11239 (2020). arXiv: 2006.112392006.11239. url: `https://arxiv.org/abs/2006.11239https://arxiv.org/abs/2006.11239` (cit. on p. 11).

[17] Clark K, Vendt B, Smith K, Freymann J, Kirby J, Koppel P, Moore S, Phillips S, Maffitt D, Pringle M, Tarbox L, Prior F. **The Cancer Imaging Archive (TCIA): maintaining and operating a public information repository.** *Journal of Digital Imaging.* 2013 Dec;26(6):1045-57. DOI: 10.1007/s10278-013-9622-7

[18] Cancer Moonshot Biobank. (2022). Cancer Moonshot Biobank - Multiple Myeloma Collection (CMB-MML) (Version 1) [Data set]. The Cancer Imaging Archive. https://doi.org/10.7937/SZKB-SW39.

[19] National Cancer Institute Clinical Proteomic Tumor Analysis Consortium (CPTAC). (2018). The Clinical Proteomic Tumor Analysis Consortium Glioblastoma Multiforme Collection (CPTAC-GBM) (Version 15) [Data set]. The Cancer Imaging Archive. https://doi.org/10.7937/K9/TCIA.2018.3RJE41Q1

[20] National Cancer Institute Clinical Proteomic Tumor Analysis Consortium (CPTAC). (2020). The Clinical Proteomic Tumor Analysis Consortium Breast Invasive Carcinoma Collection (CPTAC-BRCA) (Version 1) [Data set]. The Cancer Imaging Archive. https://doi.org/10.7937/TCIA.CAEM-YS80

[21] National Cancer Institute Clinical Proteomic Tumor Analysis Consortium (CPTAC). (2020). The Clinical Proteomic Tumor Analysis Consortium Colon Adenocarcinoma Collection (CPTAC-COAD) (Version 1) [Data set]. The Cancer Imaging Archive. https://doi.org/10.7937/TCIA.YZWQ-ZZ63

[22] Cancer Moonshot Biobank. (2022). Cancer Moonshot Biobank - Gastroesophageal Cancer Collection (CMB-GEC) (Version 1) [Data set]. The Cancer Imaging Archive. https://doi.org/10.7937/E7KH-R486

[23] National Cancer Institute Clinical Proteomic Tumor Analysis Consortium (CPTAC). (2018). The Clinical Proteomic Tumor Analysis Consortium Clear Cell Renal Cell Carcinoma Collection (CPTAC-CCRCC) (Version 10) [Data set]. The Cancer Imaging Archive. https://doi.org/10.7937/K9/TCIA.2018.OBLAMN27

[24] National Cancer Institute Clinical Proteomic Tumor Analysis Consortium (CPTAC). (2018). The Clinical Proteomic Tumor Analysis Consortium Lung Adenocarcinoma Collection (CPTAC-LUAD) (Version 12) [Data set]. The Cancer Imaging Archive. https://doi.org/10.7937/K9/TCIA.2018.PAT12TBS

[25] National Cancer Institute Clinical Proteomic Tumor Analysis Consortium (CPTAC). (2020). The Clinical Proteomic Tumor Analysis Consortium Ovarian Serous Cystadenocarcinoma Collection (CPTAC-OV) (Version 3) [Data set]. The Cancer Imaging Archive. https://doi.org/10.7937/TCIA.ZS4A-JD58

[26] National Cancer Institute Clinical Proteomic Tumor Analysis Consortium (CPTAC). (2018). The Clinical Proteomic Tumor Analysis Consortium Pancreatic Ductal Adenocarcinoma Collection (CPTAC-PDA) (Version 13) [Data set]. The Cancer Imaging Archive. https://doi.org/10.7937/K9/TCIA.2018.SC20FO18

[27] Cancer Moonshot Biobank. (2022). Cancer Moonshot Biobank - Prostate Cancer Collection (CMB-PCA) (Version 2) [Data set]. The Cancer Imaging Archive. https://doi.org/10.7937/25T7-6Y12

[28] National Cancer Institute Clinical Proteomic Tumor Analysis Consortium (CPTAC). (2018). The Clinical Proteomic Tumor Analysis Consortium Cutaneous Melanoma Collection (CPTAC-CM) (Version 10) [Data set]. The Cancer Imaging Archive. https://doi.org/10.7937/K9/TCIA.2018.ODU24GZE

[29] National Cancer Institute Clinical Proteomic Tumor Analysis Consortium (CPTAC). (2019). The Clinical Proteomic Tumor Analysis Consortium Uterine Corpus Endometrial Carcinoma Collection (CPTAC-UCEC) (Version 10) [Data set]. The Cancer Imaging Archive. https://doi.org/10.7937/K9/TCIA.2018.3R3JUISW