Cognitive Services Project

Stefano Zanatta[1]

[1] *Unipd*

For the **Cognitive Services - Unipd** Project, I implemented a Deep Convolutional Neural Network (DCNN) with direct connections for image denoising, as described in the Aojia Zhao - Stanford University paper[a], for image denoising.

I. INTRODUCTION

This project involves the implementation of the DCNN for image denoising[1] described in the abstract, together with the analisys of the results and the comparison with the state of the art image denoisers.
While classic image denoisers have fully connected layers, the Aojia Zhao one does not have any, reducing the required weights of the model. To further reduce the required weights, this model uses direct connections between each Conv layer and its relative Deconv layer.
The Google Colab implementation of the project can be found in the footnote [2].

II. TECHNOLOGIES

A. Google Colab

The project was developed entirely on Google's Jupyter notebook environment, Google Colab. This platform offers many advantages:

- Code hosted on the Cloud, allowing portability between home, stage and university's computers;
- Code executed by Google's servers, allowing more computational power than a traditional Computer;
- Free GPU, for 10 time faster execution than a CPU (experimented during Cognitive Services class);
- Python environment, with built in machine learning APIs;

During the development of the project, I found some downsides:

- Dataset size and model complexity (e.g. images size, batch size, number of epochs) are limited by the RAM limit of 12GB (within one execution). *This problem is discussed in the dataset sectionII C.*

[a] `https://web.stanford.edu/class/cs331b/2016/projects/zhao.pdf`

[1] `https://web.stanford.edu/class/cs331b/2016/projects/zhao.pdf`

[2] `https://colab.research.google.com/drive/14dAdoKLWbCKEJStnLlHpho0-L2mhsVTG`

B. Keras

Keras is the main API used for the project, using Tensorflow as backend. Keras offers a simpler interface than Tensorflow, making the process of building the model, training and showing the results easier.

C. Dataset

The dataset was the most complex part of the project. Finding the right images and saving them in an efficient way was the biggest challange.
For training and testing, I used the 'google-landmark' dataset[3]. This dataset is downloadable by calling a bash script, and the size of it can be customizable (i chose 1GB of images).
Preprocessing was required to:

- move the images in a "train" and "test" folders;
- resize the images to 128×128 and 64×64;
- insert noise;
- save the train and test images in numpy files, in this form: $2 \times \mathrm{N} \times dim \times dim$, where 'N' is the number of images in the training or testing set, and 'dim' is the dimensions of the images

To avoid running the preprocessing passages all the times, I also created a script to store and load the numpy files from Google Drive.
In the tableII C are listed the dataset used for each experiment. As mentioned in the Google Colab sectionII A, the limited RAM defined the size of the following parameters (higher definition = fewer examples or less epochs). A first experiment with a high definition image saturated the RAM with few epochs, with a validation accuracy of 0.20. For this reason that experiment was not included in the analisys. In total, i did tree main experiments:

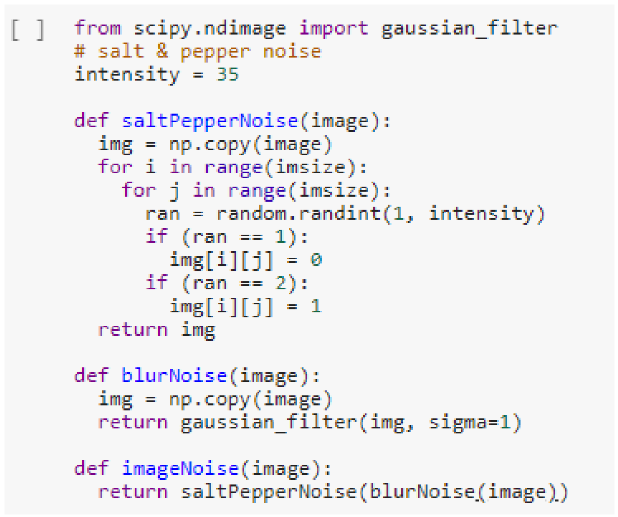
#	Train	val/tot	Test	img size	batch size	epochs
1	4000	0.2	375	128x128	30	35
2	11000	0.1	375	64x64	10	150
3	11000	0.1	375	64x64	10	135

TABLE I. Dataset for the different experiments

[3] `https://github.com/cvdfoundation/google-landmark.git`

* *val/tot indicates the validation images / total training images ratio.*

The following functions were used to insert noise into images:



```
[ ]  from scipy.ndimage import gaussian_filter
     # salt & pepper noise
     intensity = 35

     def saltPepperNoise(image):
       img = np.copy(image)
       for i in range(imsize):
         for j in range(imsize):
           ran = random.randint(1, intensity)
           if (ran == 1):
             img[i][j] = 0
           if (ran == 2):
             img[i][j] = 1
       return img

     def blurNoise(image):
       img = np.copy(image)
       return gaussian_filter(img, sigma=1)

     def imageNoise(image):
       return saltPepperNoise(blurNoise(image))
```

FIG. 1. Model

Example of the salt & pepper noised images in the training set (used for experiment 2).

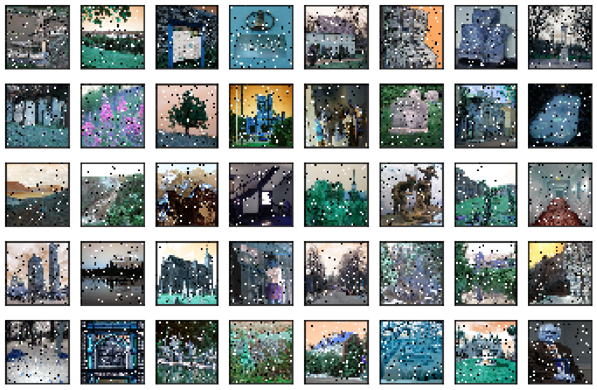


FIG. 2. salt & pepper noised images

Example of the salt & pepper plus "Gaussian filter" (blur) noised images in the training set (used for experiment 3).

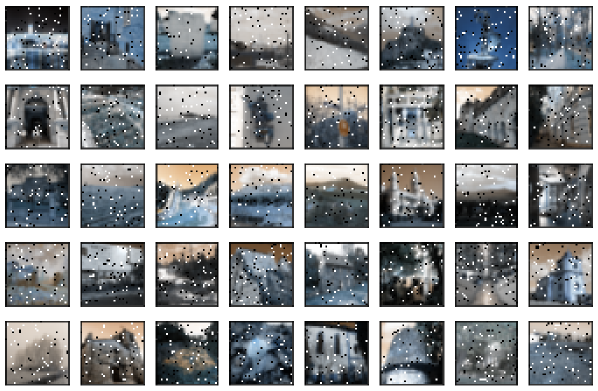


FIG. 3. salt & pepper and blurred noised images

D. Model

The model consists of 5 Convolutional (Conv) layers and 5 Deconvolutional (Deconv) layers. Each layer is connected to the following one (e.g. Conv2 with Conv3). Direct connections are implemented by adding the output of a Conv layer with the output of the "opposite" Deconv layer (e.g. Conv2 with Deconv3), and using that result as the input of the next Deconv layer (e.g. Conv2 + Deconv3 is the input of Deconv4).
The layers connections are better described by the following image:

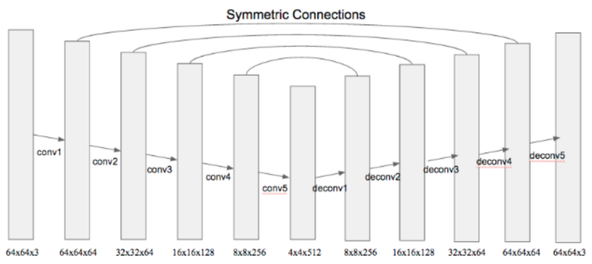
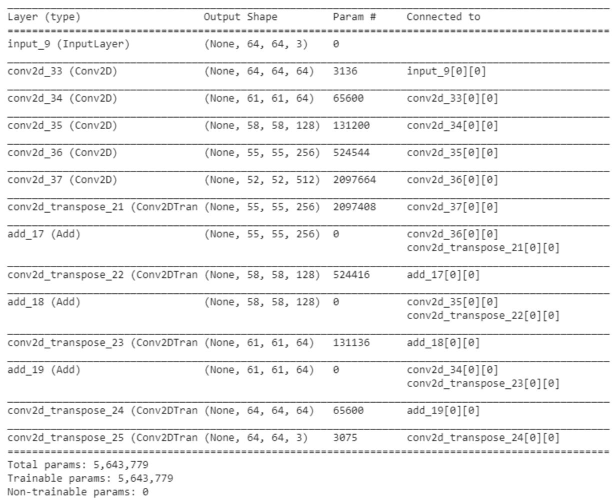


FIG. 4. Model Graph.[4]

- **Input:** $(64 \times 64 \times 3)$ single input of the DCNN, image of 64x64 pixels x3 dimensions (RGB);
- **Conv1:** $(64 \times 64 \times 64)$ Conv filters of the same dimensions of the input; connected with Conv2 and Deconv4;
- **Conv2:** $(32 \times 32 \times 128)$ same filters, sizes are halved; connected with Conv3 and Deconv3;
- **Conv3:** $(16 \times 16 \times 128)$ connected with Conv4 and Deconv2;
- **Conv4:** $(8 \times 8 \times 256)$ connected with Conv5 and Deconv1;

- **Conv5:** (4×4×512) doubled filters, image sizes are halved. At this point, the model has many small filters and the image is "deeply encoded". The Deconv layers have to decode the image;
- **Deconv1:** (8 × 8 × 256) from now on, the filter sized are the same of the "opposite" Conv layer, to match the SUM between the two layers;
- **Deconv2:** (16×16×128) connected with Deconv3;
- **Deconv3:** (32 × 32 × 64) connected with Deconv4;
- **Deconv4:** (64 × 64 × 32) connected with Deconv5;
- **Deconv5:** (64 × 64 × 3) last layer. The image is decoded to the original size and color space.



```
____________________________________________________________________________________________________
Layer (type)                     Output Shape         Param #     Connected to
====================================================================================================
input_9 (InputLayer)             (None, 64, 64, 3)    0
____________________________________________________________________________________________________
conv2d_33 (Conv2D)               (None, 64, 64, 64)   3136        input_9[0][0]
____________________________________________________________________________________________________
conv2d_34 (Conv2D)               (None, 61, 61, 64)   65600       conv2d_33[0][0]
____________________________________________________________________________________________________
conv2d_35 (Conv2D)               (None, 58, 58, 128)  131200      conv2d_34[0][0]
____________________________________________________________________________________________________
conv2d_36 (Conv2D)               (None, 55, 55, 256)  524544      conv2d_35[0][0]
____________________________________________________________________________________________________
conv2d_37 (Conv2D)               (None, 52, 52, 512)  2097664     conv2d_36[0][0]
____________________________________________________________________________________________________
conv2d_transpose_21 (Conv2DTran (None, 55, 55, 256)  2097408     conv2d_37[0][0]
____________________________________________________________________________________________________
add_17 (Add)                     (None, 55, 55, 256)  0           conv2d_36[0][0]
                                                                  conv2d_transpose_21[0][0]
____________________________________________________________________________________________________
conv2d_transpose_22 (Conv2DTran (None, 58, 58, 128)  524416      add_17[0][0]
____________________________________________________________________________________________________
add_18 (Add)                     (None, 58, 58, 128)  0           conv2d_35[0][0]
                                                                  conv2d_transpose_22[0][0]
____________________________________________________________________________________________________
conv2d_transpose_23 (Conv2DTran (None, 61, 61, 64)   131136      add_18[0][0]
____________________________________________________________________________________________________
add_19 (Add)                     (None, 61, 61, 64)   0           conv2d_34[0][0]
                                                                  conv2d_transpose_23[0][0]
____________________________________________________________________________________________________
conv2d_transpose_24 (Conv2DTran (None, 64, 64, 64)   65600       add_19[0][0]
____________________________________________________________________________________________________
conv2d_transpose_25 (Conv2DTran (None, 64, 64, 3)    3075        conv2d_transpose_24[0][0]
====================================================================================================
Total params: 5,643,779
Trainable params: 5,643,779
Non-trainable params: 0
```

FIG. 5. Model

E. Training

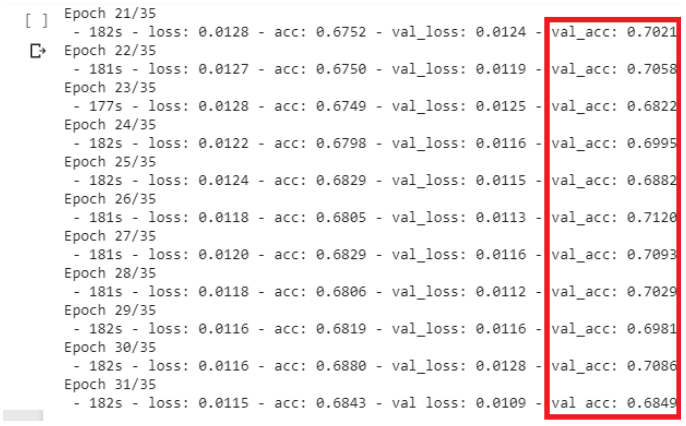
Training was done on two datasetsII C, 128 × 128 and 64 × 128 images. The 128 images required much more resources then the 64 ones, this required reducing the epochs and the number of training dataset.
MSE was used as the minimization function in both cases, and the categorical_accuracy for evaluating the model (the default accuracy for keras).
Stochastic Gradient Descent was used as the optimizer. Early stopping was not used, to reduce the usage of the RAM.
For every experiment, test accuracy was lower then Validation accuracy, meaning that the model suffers of underfitting (should use more images for training).

Experiment	Validation acc	Test acc
1	0.7067	0.5800
2	0.7663	0.6200
3	0.7209	0.6420

TABLE II. Training results

F. Experiment 1 - 128 × 128

128 × 128 images images were used for the first experiment, using doubled the size required from the Zhao paper. Salt & pepper was used as the noise. This quickly saturated the RAM, so reduced the dataset elements and the epochs by more than 50%.
Using 35 epochs, the model stabilized on 0.70 accuracy after few epochs. Changing epochs did not change this result. This probably means that increasing the complexity of the model is required to fit bigger images. This is not possible with Google Colab resources, for the RAM and GPU limits.

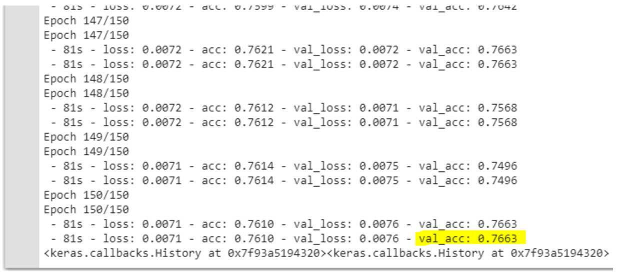


```
Epoch 21/35
 - 182s - loss: 0.0128 - acc: 0.6752 - val_loss: 0.0124 - val_acc: 0.7021
Epoch 22/35
 - 181s - loss: 0.0127 - acc: 0.6750 - val_loss: 0.0119 - val_acc: 0.7058
Epoch 23/35
 - 177s - loss: 0.0128 - acc: 0.6749 - val_loss: 0.0125 - val_acc: 0.6822
Epoch 24/35
 - 182s - loss: 0.0122 - acc: 0.6798 - val_loss: 0.0116 - val_acc: 0.6995
Epoch 25/35
 - 182s - loss: 0.0124 - acc: 0.6829 - val_loss: 0.0115 - val_acc: 0.6882
Epoch 26/35
 - 181s - loss: 0.0118 - acc: 0.6805 - val_loss: 0.0113 - val_acc: 0.7120
Epoch 27/35
 - 181s - loss: 0.0120 - acc: 0.6829 - val_loss: 0.0116 - val_acc: 0.7093
Epoch 28/35
 - 181s - loss: 0.0118 - acc: 0.6806 - val_loss: 0.0112 - val_acc: 0.7029
Epoch 29/35
 - 182s - loss: 0.0116 - acc: 0.6819 - val_loss: 0.0116 - val_acc: 0.6981
Epoch 30/35
 - 182s - loss: 0.0116 - acc: 0.6880 - val_loss: 0.0128 - val_acc: 0.7086
Epoch 31/35
 - 182s - loss: 0.0115 - acc: 0.6843 - val loss: 0.0109 - val acc: 0.6849
```

FIG. 6. Training results for experiment 1

G. Experiment 2 - 64 × 64

64 × 64 sized images were used for the second experiment as suggested in the Zhao paper. Images were altered only with salt and pepper noise. The training was done on 11000 training images, 1100 of them are used for validation.
Experiment 2 showed better results than Experiment 1, because the accuracy did not saturate at 0.70, but it kept increasing with the epochs (as shown in the following image).

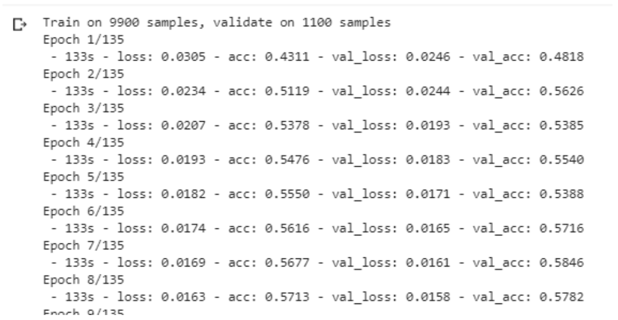


```
 - 81s - loss: 0.0072 - acc: 0.7599 - val_loss: 0.0074 - val_acc: 0.7642
Epoch 147/150
Epoch 147/150
 - 81s - loss: 0.0072 - acc: 0.7621 - val_loss: 0.0072 - val_acc: 0.7663
 - 81s - loss: 0.0072 - acc: 0.7621 - val_loss: 0.0072 - val_acc: 0.7663
Epoch 148/150
Epoch 148/150
 - 81s - loss: 0.0072 - acc: 0.7612 - val_loss: 0.0071 - val_acc: 0.7568
 - 81s - loss: 0.0072 - acc: 0.7612 - val_loss: 0.0071 - val_acc: 0.7568
Epoch 149/150
Epoch 149/150
 - 81s - loss: 0.0071 - acc: 0.7614 - val_loss: 0.0075 - val_acc: 0.7496
 - 81s - loss: 0.0071 - acc: 0.7614 - val_loss: 0.0075 - val_acc: 0.7496
Epoch 150/150
Epoch 150/150
 - 81s - loss: 0.0071 - acc: 0.7610 - val_loss: 0.0076 - val_acc: 0.7663
 - 81s - loss: 0.0071 - acc: 0.7610 - val_loss: 0.0076 - val_acc: 0.7663
<keras.callbacks.History at 0x7f93a5194320><keras.callbacks.History at 0x7f93a5194320>
```

FIG. 7. Last Epochs - Experiment 2

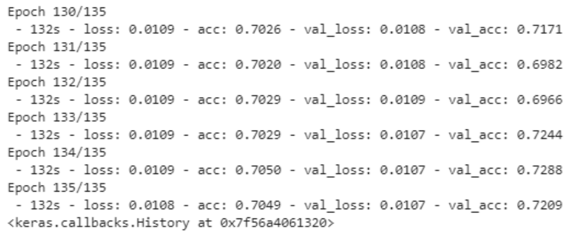
H. Experiment 3 - 64×64

Experiment 3 was almost the same as Experiment 2II G, with the difference that blur noise were added to salt & pepper noise, and the epochs were reduced to 135. As especited, the train time for each epoch increased by 50 seconds (from 83 to 132).
Test accuracy was better then the Experiment 2. This may be because, with more features to learn, the algorithm managed to learn more significant weights (each weight has more 'value').



```
Train on 9900 samples, validate on 1100 samples
Epoch 1/135
 - 133s - loss: 0.0305 - acc: 0.4311 - val_loss: 0.0246 - val_acc: 0.4818
Epoch 2/135
 - 133s - loss: 0.0234 - acc: 0.5119 - val_loss: 0.0244 - val_acc: 0.5626
Epoch 3/135
 - 133s - loss: 0.0207 - acc: 0.5378 - val_loss: 0.0193 - val_acc: 0.5385
Epoch 4/135
 - 133s - loss: 0.0193 - acc: 0.5476 - val_loss: 0.0183 - val_acc: 0.5540
Epoch 5/135
 - 133s - loss: 0.0182 - acc: 0.5550 - val_loss: 0.0171 - val_acc: 0.5388
Epoch 6/135
 - 133s - loss: 0.0174 - acc: 0.5616 - val_loss: 0.0165 - val_acc: 0.5716
Epoch 7/135
 - 133s - loss: 0.0169 - acc: 0.5677 - val_loss: 0.0161 - val_acc: 0.5846
Epoch 8/135
 - 133s - loss: 0.0163 - acc: 0.5713 - val_loss: 0.0158 - val_acc: 0.5782
```

FIG. 8. First Epochs - Experiment 3



```
Epoch 130/135
 - 132s - loss: 0.0109 - acc: 0.7026 - val_loss: 0.0108 - val_acc: 0.7171
Epoch 131/135
 - 132s - loss: 0.0109 - acc: 0.7020 - val_loss: 0.0108 - val_acc: 0.6982
Epoch 132/135
 - 132s - loss: 0.0109 - acc: 0.7029 - val_loss: 0.0109 - val_acc: 0.6966
Epoch 133/135
 - 132s - loss: 0.0109 - acc: 0.7029 - val_loss: 0.0107 - val_acc: 0.7244
Epoch 134/135
 - 132s - loss: 0.0109 - acc: 0.7050 - val_loss: 0.0107 - val_acc: 0.7288
Epoch 135/135
 - 132s - loss: 0.0108 - acc: 0.7049 - val_loss: 0.0107 - val_acc: 0.7209
<keras.callbacks.History at 0x7f56a4061320>
```

FIG. 9. Last Epochs - Experiment 3

I. Results

Even if the model reached an accuracy of about 0.75, the model could probably reach better results with doubled epochs (around 9 hours of training).
The batch size of 10 avoided overfitting. In fact, in the first experiments, increasing the batch size implied a faster growth of the train accuracy, but reduced the validation accuracy. As I mentioned in the Experiment 1 sectionII F, the model is made for small images, so just for research purposes. Increasing the model complexity would allow to use bigger images, but at the cost of GPU power.

The cleaned images are kinda blurred, but the model did a great job at removing the salt and pepper noise. With more training, the results would have been less blurred (and closer to the original image).

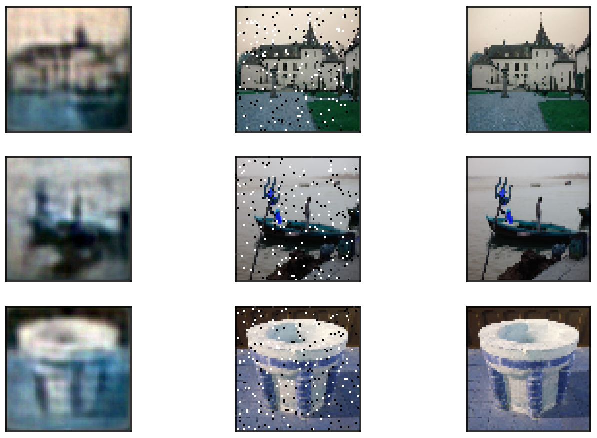


FIG. 10. Clean - Noised - Original {experiment 2}

With a white uniform background, it was difficult for the model to remove the black pixels.

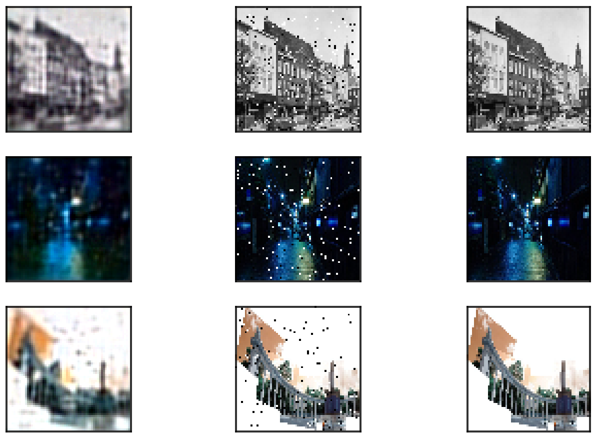


FIG. 11. Clean - Noised - Original {Experiment 3}

J. Comparison with state of the art models and possible improvements to this model

To understand how a state of the art models work, I took **Xiao-Jiao Mao, Chunhua Shen, Yu-Bin Yang's project**[5] as benchmark. They worked with bigger images, days of training, bigger models (30 layers instead of 10) and different evaluating methods, but still using a DCNN with direct connections.
This paper suggests to use small filter size, in fact the filters Zhao (*the author of the model I implemented*) suggested are 4×4.
The evaluating method they used is the *Peak Signal to Noise Ratio* (PSNR).

---

[5] `https://arxiv.org/pdf/1606.08921v3.pdf`

$$PSNR = 10 \times \log_{10}(R^2/MSE)$$

Where R is the maximum fluctuation in the input image data type (e.g. 8bit image has R = 255) and MSE is the Mean Squared Error. This metric is better rappresented in a graph than the MSE, because of the logarithm.

ACKNOWLEDGMENTS

- Image Denoising with Deep Convolutional Neural Networks[6] - Author: Aojia Zhao - Stanford University, on which I based my project;
- Image Restoration Using Convolutional Autoencoders with Symmetric Skip Connections[7] - Authors: Xiao-Jiao Mao, Chunhua Shen, Yu-Bin Yang;
- Papers with code website[8], where I found the state of the art papers;
- Google Colab[9]

[6] `https://web.stanford.edu/class/cs331b/2016/projects/zhao.pdf`
[7] `https://arxiv.org/pdf/1606.08921v3.pdf`
[8] `https://paperswithcode.com/task/image-denoising`
[9] `https://colab.research.google.com/notebooks/`