Image Super-Resolution using Neural Networks

Author: Kunjan Mhaske

Introduction:

Super-resolution (SR) is an important class of image processing techniques to enhance the resolution of images and videos in computer vision. Recent years have witnessed remarkable progress of image super-resolution using deep learning techniques and it has been widely used in many applications such as remote sensing image, medical image, video surveillance and high definition television. In these methods, the low resolution (LR) input image is upscaled to the high resolution (HR) space using a single filter, commonly bicubic interpolation, before reconstruction. This means that the super-resolution (SR) operation is performed in HR space.

The global SR problem assumes LR data to be a low-pass filtered (blurred), down-sampled and noisy version of HR data. It is a highly ill-posed problem, due to the loss of high-frequency information that occurs during the non-invertible low-pass filtering and subsampling operations. Furthermore, the SR operation is effectively a one-to-many mapping from LR to HR space which can have multiple solutions, of which determining the correct solution is non-trivial. A key assumption that underlies many SR techniques is that much of the high-frequency data is redundant and thus can be accurately reconstructed from low frequency components. SR is therefore an inference problem, and thus relies on our model of the statistics of images in question.

Motivation:

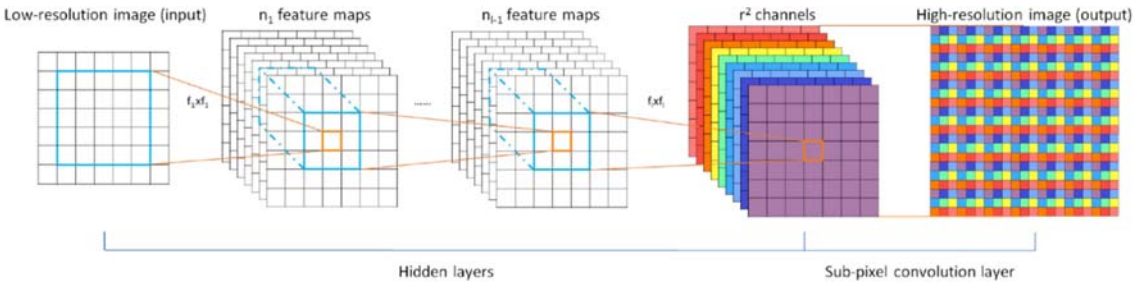
With the development of CNN, the efficiency of the algorithms, especially their computational and memory cost, gains importance. The flexibility of deep network models to learn nonlinear relationships has been shown to attain superior reconstruction accuracy compared to previously hand-crafted models. To super-resolve a LR image into HR space, it is necessary to increase the resolution of the LR image to match that of the HR image at some point.

Goal:

Assumption: The high-frequency data is redundant and thus can be accurately reconstructed from low frequency components.

Aim: To learn the implicit redundancy that is present in natural data to recover missing High-Resolution information from Low-Resolution images.

Model:



In implemented model, I have used the 4 layers of 2D-Convolutions back to back with last layer of sub-pixels convolution that utilizes the upscale_factor to generate the high-resolution data.

```
Neural Network:
self.relu = nn.ReLU()
self.conv1 = nn.Conv2d(1, 64, (5, 5), (1, 1), (2, 2))
self.conv2 = nn.Conv2d(64, 64, (3, 3), (1, 1), (1, 1))
self.conv3 = nn.Conv2d(64, 32, (3, 3), (1, 1), (1, 1))
self.conv4 = nn.Conv2d(32, upscale_factor ** 2, (3, 3), (1, 1), (1, 1))
self.pixel_shuffle = nn.PixelShuffle(upscale_factor)
self._initialize_weights()
```

In this implementation, to initialize the weights in neural network, the weight matrix should be chosen as a random orthogonal matrix, i.e., a square matrix W for which $W^TW=I$. While performing the orthogonal matrix, the gain is calculated using relu.

```
Forward method:
x = self.relu(self.conv1(x))
x = self.relu(self.conv2(x))
x = self.relu(self.conv3(x))
x = self.pixel_shuffle(self.conv4(x))
```

In the last step of forward method, the pixels are shuffled while feeding to 4^{th} layer of convolution.

Activation Function: ReLU

ReLU stands for rectified linear unit, and is a type of activation function. Mathematically, it is defined as y = max(0, x). Visually, it looks like the following:

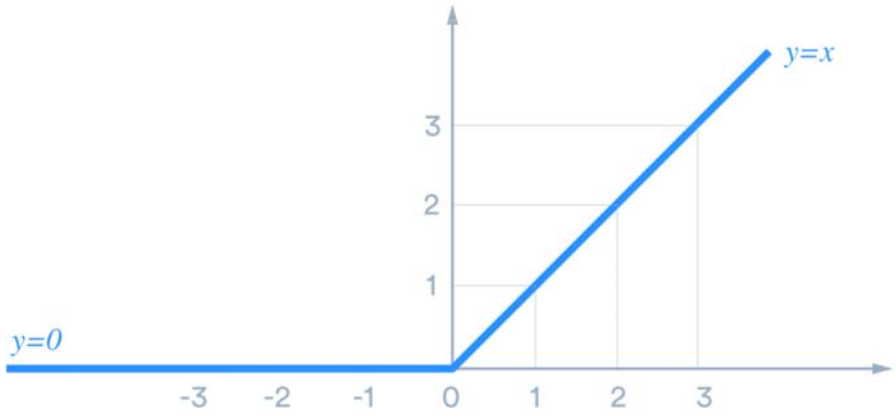


Figure: ReLU graph

ReLU is the most commonly used activation function in neural networks, especially in CNNs. ReLU is linear (identity) for all positive values, and zero for all negative values. This means that:

- It's cheap to compute as there is no complicated math. The model can therefore take less time to train or run.
- It converges faster. Linearity means that the slope doesn't plateau, or "saturate," when *x* gets large. It doesn't have the vanishing gradient problem suffered by other activation functions like sigmoid or tanh.
- It's sparsely activated. Since ReLU is zero for all negative inputs, it's likely for any given unit to not activate at all.

Loss Function: MSELoss

Mean Squared Error (MSE), or quadratic, loss function is widely used in linear regression as the performance measure.

```
criterion = nn.MSELoss()
```

It Creates a criterion that measures the mean squared error (squared L2 norm) between each element in the input x and target y where x and y are tensors of arbitrary shapes with a total of n elements each.

Optimizer: Adam - moment estimation

Adam is an optimization algorithm that can used instead of the classical stochastic gradient descent procedure to update network weights iterative based in training data.

```
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

 Adam is a replacement optimization algorithm for stochastic gradient descent for training deep learning models.
 Adam combines the best properties of the AdaGrad and RMSProp algorithms to provide an optimization algorithm that can handle sparse gradients on noisy problems.
 Adam is relatively easy to configure where the default configuration parameters do well on most problems.

Data Transformation:

Before feeding data to neural networks, the transformation of data should be done to maintain the uniform dimensions of training tensors. While taking input images, they are resized using proper_crop_size which is (crop_size – (crop_size mod upscale_factor)) that depends on upscale factor and while taking target images, they are just center-cropped using the same proper_crop_size. Data transformation is used while getting both training dataset as well as testing dataset.

Dataset: BSDS300 or BSDS500

In this implementation, I have used BSDS300 dataset fetched from its source which is publicly available with citation for research in computer vision. The images are divided into a training set of 200 images, and a test set of 100 images. They are grayscale images and in

implementation, they are further converted to YCbCr format to visualize in human sensitive color shades.

```
# For BSDS500 Dataset
url =
"www.eecs.berkeley.edu/Research/Projects/CS/vision/grouping/BSR/BSR_bsds500.tgz"

# For BSDS300 Dataset
url = "http://www2.eecs.berkeley.edu/Research/Projects/CS/vision/bsds/BSDS300-
images.tgz"
```

```
Citation -
https://www2.eecs.berkeley.edu/Research/Projects/CS/vision/grouping/segbench/
@InProceedings{MartinFTM01,
  author = {D. Martin and C. Fowlkes and D. Tal and J. Malik},
  title = {A Database of Human Segmented Natural Images and its
           Application to Evaluating Segmentation Algorithms and
           Measuring Ecological Statistics},
  booktitle = {Proc. 8th Int'l Conf. Computer Vision},
  year = {2001},
  month = {July},
  volume = {2},
  pages = {416--423}
}
```

The dataset is extracted and removed after generating new data after performing the transformation process on them. The transformed images are further used for training and testing dataset using dataloader.

Training and Testing Details:

Epochs: 25 (10-15 epochs give admissible results)

Learning Rate: 0.001

Training Batch Size: 4

Testing Batch Size: 64-100 preferable

At first, get training and testing data loader from transformed training and testing datasets. Then perform training on dataset with given epochs and learning rate and backpropagate the error in the neural network to update the weights accordingly using Adam optimizer and MSError. Calculate the error for each epoch to keep track of model's learning process.

While testing, calculate the average PSNR which is based on the MSE from testing dataset.

PSNR: Peak Signal-to-Noise Ratio

The PSNR block computes the peak signal-to-noise ratio, in decibels, between two images. It is the ratio between the maximum possible power of a signal and the power of corrupting noise that affects the fidelity of its representation. This ratio is often used as a quality measurement between the original and a compressed image. The higher the PSNR, the better the quality of the compressed, or reconstructed image.

The *Mean Square Error (MSE)* and the *Peak Signal to Noise Ratio (PSNR)* are the two error metrics used to compare image compression quality. The MSE represents the cumulative

squared error between the compressed and the original image, whereas PSNR represents a measure of the peak error. The lower the value of MSE, the lower the error.

To compute the PSNR, the block first calculates the mean-squared error using the following equation:

$$MSE = \frac{\sum_{M,N} [I_1(m,n) - I_2(m,n)]^2}{M * N}$$

In the previous equation, *M* and *N* are the number of rows and columns in the input images, respectively. Then the block computes the PSNR using the following equation:

$$PSNR = 10 \log_{10}\left(\frac{R^2}{MSE}\right)$$

In the previous equation, *R* is the maximum fluctuation in the input image data type. For example, if the input image has a double-precision floating-point data type, then *R* is 1.

Finally, the trained model is saved in the pytorch supported file (.pth) after each epoch, by which we can visualize the results after each epoch and compare between them. The trained model then used for generating the super resolution image from the given input image using generate_super_resolution.py program.

Instructions to run the programs:

- Keep all model.py data.py dataset.py generate_model.py generate_super_resolution.py files in same directory
- Open terminal and run command: *python3 generate_model.py -h*
- It gives the argument information on which the model can be trained. For e.g. –

```
python3 generate_model.py --upscale_factor 3 --trainBatchSize 4 --testBatchSize 100
--nEpochs 10 --lr 0.001 –-cuda --threads 4 --seed 123
```

- The program will download the dataset from the internet and starts training on it
- The program saves the model after each epoch in same directory with different name
- The saved model then can be used to generate the super resolution image from any given lower resolution image.
- Keep input low resolution image in same directory as above programs.
- Run command: *python3 generate_super_resolution.py -h*
- It gives the argument information. For e.g. –

```
python generate_super_resolution.py --input_image army.jpg --model
model_epoch_10.pth --output_filename super_army.jpg --cuda
```

- The program will generate the super resolution image and saves in same directory with output name as provided in the argument. The Output image will be upscaled by trained model upscale_factor.

Note: For upscaling factor 1, GPU vram runs out hence it is trained on CPU for 10 epochs only.

Results:

Run command to train the model:

```
python generate_model.py --upscale_factor 3 --trainBatchSize 4 --testBatchSize 100 --nEpochs 15 --lr 0.001 –cuda
```

It gives output as:

Epoch 15 Complete: Avg. Loss: 0.0033
===> Avg. PSNR: 24.4175 dB
Checkpoint saved to model_epoch_15_upscale_3.pth

Run command to generate super resolution from input image:

```
python generate_super_resolution.py --input_image bird.jpg --model model_epoch_15_upscale_3.pth --output_filename bird_e15.jpg –cuda
```

It gives output as:

Loading the given model...
The Output Image is Saved as bird_e15.jpg

Figure: Input Image 481x321 size: 43.9kb

Figure: EPOCH 1 Output 1443x963 size: 206kb

Figure: EPOCH 5 Output 1443x963 size: 134kb

Figure: EPOCH 10 Output 1443x963 size: 103kb

Figure: EPOCH 15 Output 1443x963 size: 100kb

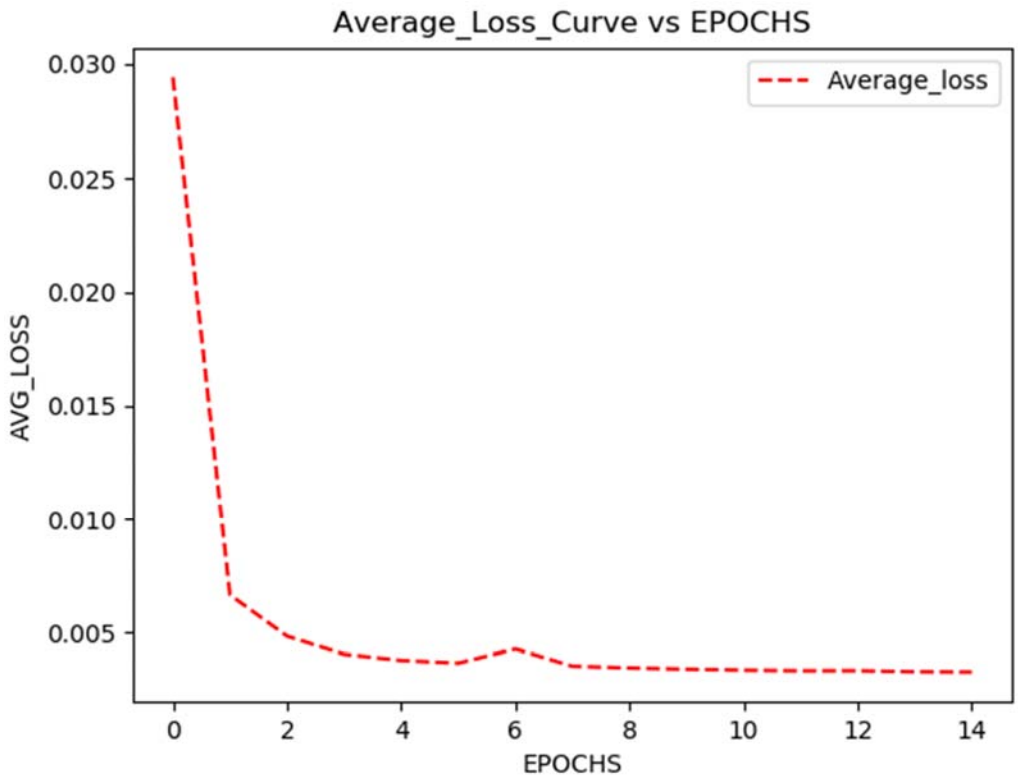


Figure: Output: Average Loss curve per epochs
X: Epochs and Y: Average Loss

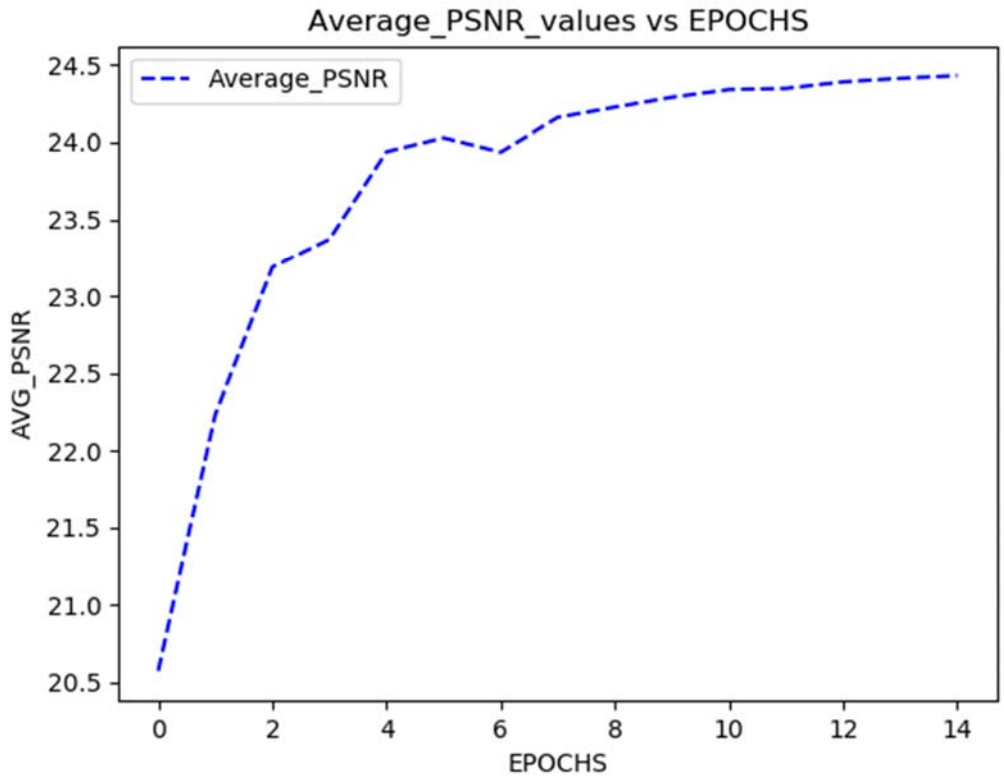


Figure: Output: Average PSNR values per epochs
X: Epochs and Y: Average PSNR values

Results Comparisons:

Upscale Factor	EPOCHS	AVG_LOSS	AVG_PSNR (in dB)	Upscale Factor	EPOCHS	AVG_LOSS	AVG_PSNR (in dB)
1	1	0.0099	31.3915	6	1	0.0402	18.4375
	2	0.006	36.6675		5	0.0075	21.0204
	3	0.0003	38.6528		10	0.0064	21.6369
	4	0.0001	40.2229		15	0.0069	21.6355
	5	0.0001	41.2345		20	0.0062	21.7715
	6	0.0001	40.9389		25	0.0062	21.7938
	7	0	43.0093		30	0.0062	21.7258
	8	0	43.5895		35	0.0061	21.8483
	9	0	43.2342		40	0.006	21.8598
	10	0	44.215				
				7	1	0.0484	17.9543
2	1	0.0139	23.5367		5	0.0091	20.3257
	5	0.002	26.5851		10	0.0075	20.8836
	10	0.0018	27.0264		15	0.0071	21.0381
	15	0.0018	27.0146		20	0.0069	21.2825
	20	0.0018	26.7195		25	0.0068	21.3162
	25	0.0017	27.2766		30	0.0068	21.3387
	30	0.0016	26.4719		35	0.0068	21.3556
	35	0.0016	27.3102		40	0.0068	21.3621
	40	0.0017	27.3048				
				8	1	0.0426	17.7054
3	1	0.0294	20.5728		5	0.0082	20.5975
	5	0.004	23.7103		10	0.0077	20.826
	10	0.0034	24.3121		15	0.0077	20.3686
	15	0.0033	24.3953		20	0.0075	20.9327
	20	0.0033	24.3237		25	0.0075	20.9581
	25	0.0032	24.5198		30	0.0076	20.5417
	30	0.0032	24.5422		35	0.0074	20.9892
	35	0.0031	24.5696		40	0.0073	20.9124
	40	0.0031	24.5976				
				9	1	0.0466	17.315
4	1	0.0271	19.8737		5	0.0092	19.9908
	5	0.0053	22.515		10	0.0084	20.3887
	10	0.0047	22.9736		15	0.0082	20.5487
	15	0.0045	23.1139		20	0.0082	20.5902
	20	0.0044	23.1705		25	0.008	20.6158
	25	0.0045	22.9914		30	0.008	20.6278
	30	0.0044	23.2189		35	0.0079	20.6372
	35	0.0044	23.242		40	0.008	20.6166
	40	0.0044	23.2204				
				10	1	0.0464	17.2421
5	1	0.0317	19.0039		5	0.0098	19.82
	5	0.0064	21.8126		10	0.0091	20.1056
	10	0.0056	22.2204		15	0.0088	20.2183
	15	0.0057	22.2809		20	0.0087	20.2779
	20	0.0054	22.3643		25	0.0087	20.3009
	25	0.0054	22.397		30	0.0086	20.3354
	30	0.0053	22.4187		35	0.0085	20.2906
	35	0.0053	22.4284		40	0.0086	20.3537
	40	0.0054	21.6624				

Observations and Inferences:

When the model is run for 1 to 10 scaling factors, each 40 epochs (except scaling factor 1 in which epochs are 10 only because of it runs on CPU). The PSNR do not get increases after the 15-20 epochs, because the data generated from low resolution is enough for upscaling up to 3 to 5 times only, and after that, the model generates data but it does not significantly increases the details of output image, instead it just increases the resolution of image. As the image get more cleaner, the redundant pixels values are vanishing and hence the size of image is decreasing accordingly. Hence, after certain limit of SR, image dimension increases but it will not get good quality output in visual better than upscaling factor 4-5.

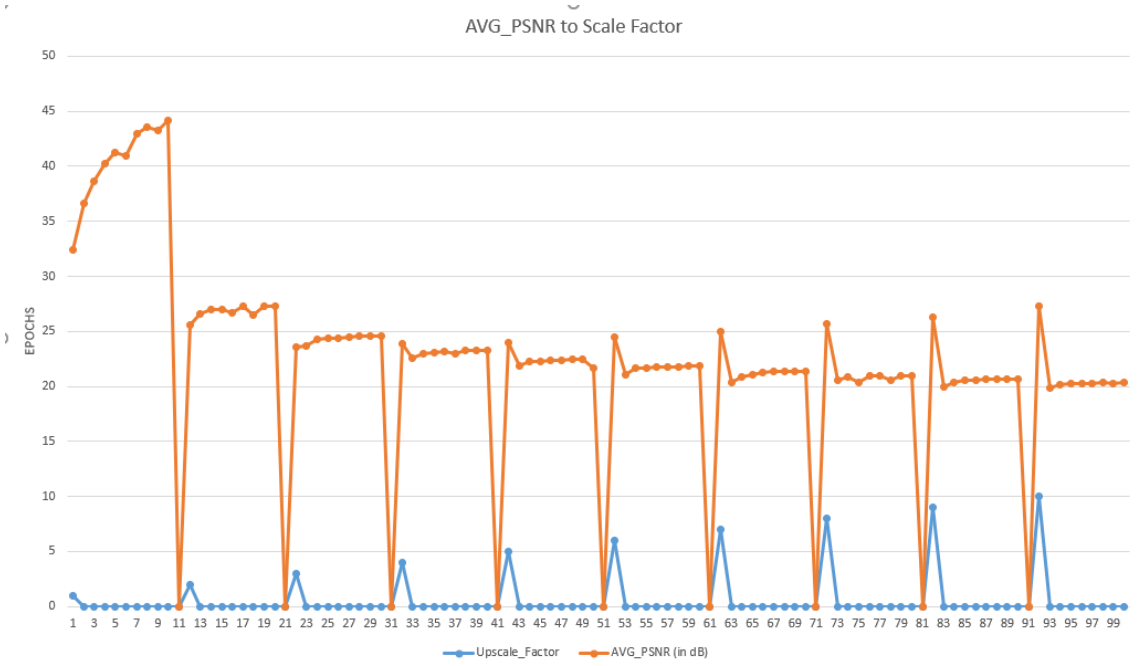


Figure: Average PSNR value for 1 to 10 upscaling factor

PSNR Value gradually stabilize after particular value of scaling factor

---

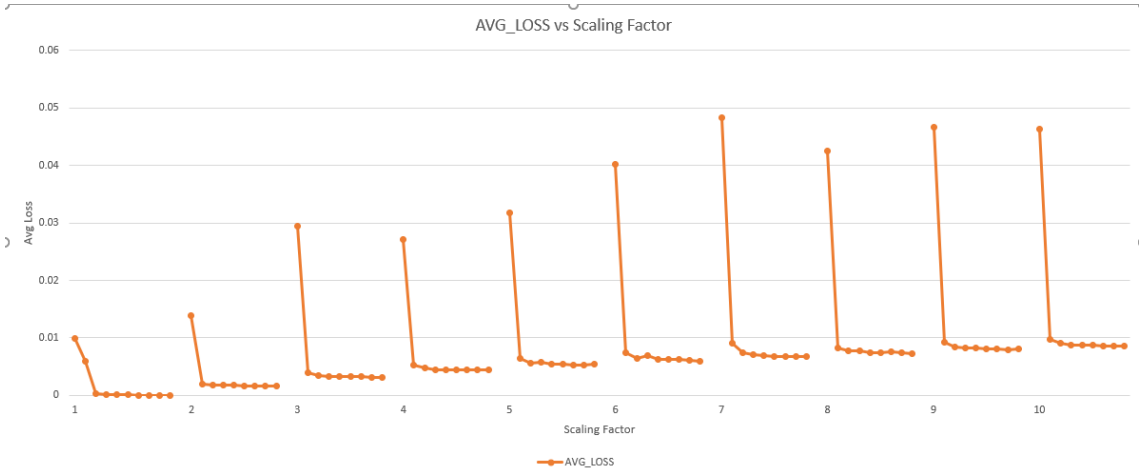


Figure: Average Loss values for 1 to 10 scaling factor

After particular value of scaling factor and epoch, loss is not decreasing

References:

1] Real-Time Single Image and Video Super-Resolution Using an Efficient Sub-Pixel Convolutional Neural Network – Shi et.al - https://arxiv.org/abs/1609.05158

2] Wikipedia - https://www.wikipedia.org/

3] Pytorch documentation - https://pytorch.org/docs/stable/nn.html

4] Github for debugging and code snippets - https://github.com/

5] Debugging and code snippets - https://stackoverflow.com/