Using Deep Convolutional Networks for Image Super-Resolution

Isaac Corley, Jonathan Lwowski, and Sos Agaian

Abstract

In this paper, we present a fast end to end deep convolution network to map a low resolution color image to a higher resolution color image. Furthermore, a comparison of how using different loss measurements such as mean squared error effect the outcome of the network. Along with that a discussion of how slicing the image in various ways can effect the overall quality of the output image.

Index Terms

Super Resolution, Convolutional Neural Network, Deep Learning, Structural Similarity Index

I. Introduction

IMAGE super-resolution, a technique used to recover a high resolution image from a low resolution image, is a classical computer vision problem [1]. This is a very important problem to solve, because when resizing an image from a smaller size to a larger size, information in typically lost. The goal of super resolution is to reduce the amount of data that is lost when scaling images. Image super resolution can also be used overcome some of the limitations of using low cost image sensors, such as cell phone or surveillance cameras [2]. Traditional super resolution approaches require multiple images of the same scene in order to produce the higher quality image [3]. These images also typically need to be aligned with high accuracy. Several methods for super resolution has been developed such as maximum a-posteriori solution using Huber Markov Random Fields or Bilateral Total Variation [3]. The problem with these methods is that their do not have good performance with a large scaling factor. Another method that is currently being used to perform super resolution is the learning of mapping functions from low-resolution and high resolution pairs on images [1].

More recently, deep learning has been used to perform image super resolution. More specifically, convolutional neural networks (CNNs), have been used to learn the end-to-end mapping between low and high resolution images [1]. Super resolution CNNs (SRCNNs) provide better accuracy and faster speeds in comparison to many of the traditional methods, because it does not need to solve any optimization problems and is only feed forward. Currently there are only a few SRCNNs. Dong et. al. developed a three layer SRCNN [1]. This SRCNN requires the image to first be preprocessed using bi-cubic interpolation. Then their SRCNN could be used. The SRCNN's first layer extracts all of the feature maps, while the second layer maps the feature maps nonlinearly to high-resolution patches. The last layer combines these patches with a spatial neighborhood to produce the high resolution output image. Another SRCNN developed by Kim et. al. uses a very deep SRCNN. Their SRCNN uses twenty 3x3 conventional layers, each with 64 channels. The output of the SRCNN is then added to the original image to produce the high resolution output image. Fairly recently, Dong et. al. developed another SRCNN. This SRCNN does not require bi-cubic interpolation, but does require a deconvolution layer. Their newer SRCNN has better performance along with a lower computational cost.

The rest of the paper is organized as follows. Section II will contain a discussion of the algorithm design that was used. The results be will be presented and analyzed Section III. The concluding remarks will be discussed in the final section.

II. Algorithm Design

A. System Overview

Our system, outlined in Figure 1, works by taking in a color image that needs to be scaled into a larger image. That image is then decomposed into its red, blue, and green channels. Each channel is then split into various equal size slices. Each of these slices are then passed into the SRCNN. The SRCNN, seen in Figure 2, will output a enhanced version of image slice. These enhanced slices are then combined back into the individual red, blue or green channels. These enhanced channels are then combined back into a single enhanced color image.

I. Corley, J. Lwowski, and S. Agaian are with the Department of Electrical and Computer Engineering, The University of Texas at San Antonio, San Antonio, TX, 78250 USA

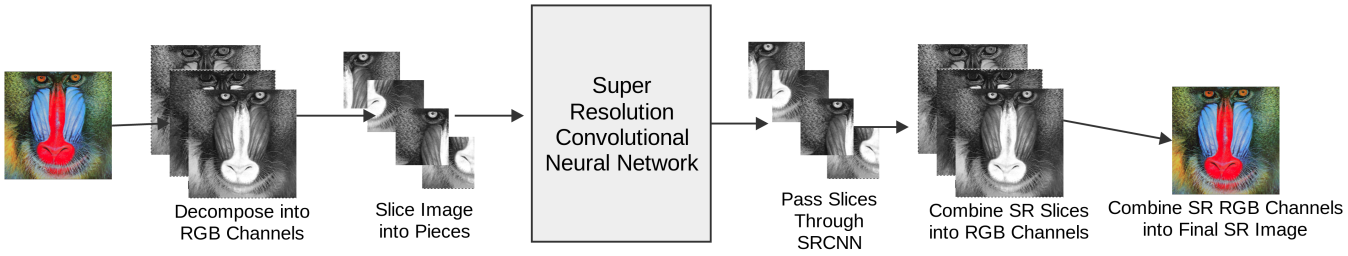


Fig. 1: High Level Flowchart of the System

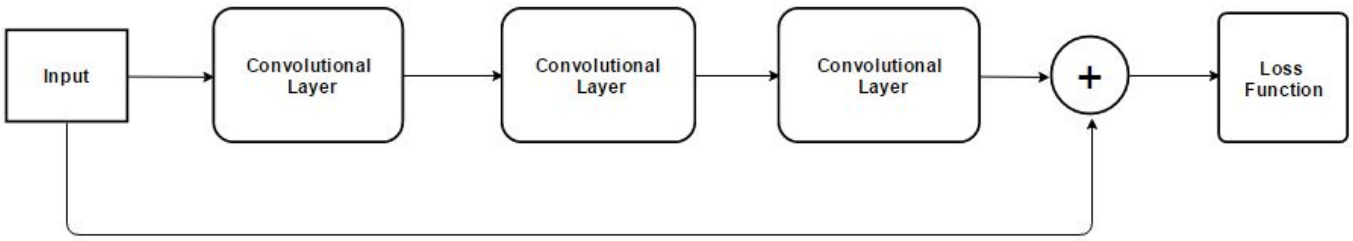


Fig. 2: SRCNN Model Architecture

B. Image Preprocessing

To create a set of high-resolution (HR) or ground truth images, all images in datasets used were reshaped to 256x256. Then to create low-resolution (LR) images, each HR image was downsampled to 128x128 and then upsampled back to 256x256 using bicubic interpolation. This created a set of LR images which had significant information loss and appeared pixelated.The LR images were then used as the model inputs during training to be compared to the HR images through a loss function.

C. Color Transformations, Channel Separation, and Splicing

There are many different approaches to training a deep neural network for super resolution. One notable method is to train on images which are spliced into sections instead of the entire image. This was proven to reduce the error when compared with training on entire images, in our case 256x256. This is likely due to the network being able to handle learning from a smaller amount of pixels. The size of splices were varied in the algorithm; 8x8, 16x16, 32x32, 64x64.

Another method used in the algorithm was color channel separation by splitting each splice of an image into it's corresponding color channels and training networks with the same architecture separately. This proved to benefit the similarity of the output image with ground truth image greatly when compared with training on a 3-Dimensional color image.

Initially the algorithm was tested on the training image sets converted to color video format YCbCr, due to the luminance channel, Y, containing the majority of the images' edge information. However, training with images in RGB format, proved to be more accurate in providing images with greater similarity.

D. Residual Networks

The final key step of the SRCNN architecture is implementing a residual addition layer by summing the input image with the output of the last convolutional layer before being input into the loss function. Prior to including this residual layer, the output test images were consistently incorrect when compared to the ground truth image due to a difference in shading. After implementing the residual layer in the architecture, this shading error was removed.

E. Loss Functions

The loss function is a function used during training of a deep learning model to calculate the error or determine the performance of the model output with respect to the ground truth. This function is then optimized to minimize error using various optimization techniques. Loss functions are typically chosen based upon the application of the deep learning model. In the case of image super-resolution, other research commonly uses Mean-Squared Error (MSE) as the loss function. MSE between images can be simply described as the average of the pixel by pixel squared differences. the MSE for grayscale images or 2-Dimensional data is displayed in the equation below. The equation can also be extended to handle 3-Dimensional data.

$$MSE = (\frac{1}{MN}) \sum_{m=1}^{M} \sum_{n=1}^{N} [y(n,m) - x(n,m)]^2$$

However, MSE has proven to be inconsistent with actual human visual perception. Although, not commonly used as a loss function, the Structural Similarity Index (SSIM) is a metric of image quality or similarity designed to improve upon the faults of MSE. The SSIM is composed of the luminance, contrast, and structural components, which are each a combination of images' statistics. The SSIM is typically calculated for 2-D or 3-D patches of images. The equation below displays the reduced SSIM for a predicted image $\hat{y}$ and ground truth image y, where $\mu_{\hat{y}}$ and μ_y are the respective expected values, $\sigma_{\hat{y}}$ and σ_y are the variances, $\sigma_{\hat{y}y}$ is the covariance, $L = 255$ is the pixel dynamic range for images of type 8-bit unsigned integers, $k_1 = 0.01$ $k_1 = 0.03$ are variables, and $c_1 = (k_1 L)^2$ and $c_1 = (k_2 L)^2$ are variables to prevent division by 0.

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

The SSIM is bounded between 0,1 where 0 represents no similarity, and 1 represents full similarity. This however, cannot be used as a loss. A variant of this function called the Structural Dissimilarity (DSSIM) metric can be optimized and is provided in the equation below.

$$DSSIM(\hat{y}, y) = \frac{1 - SSIM(\hat{y}, y)}{2}$$

F. Network Summary

The final network architecture consisted of one input layer, 3 convolutional layers, and one residual addition layer. A detailed explanation of each layer's parameters are provided in I below.

TABLE I: Network Architecture Summary (Total Parameters = 109,313)

Layers	Output Shape	Filters	Filter Dimensions	Feature Maps	Parameters
Input	(32, 32, 1)	N/A	N/A	N/A	N/A
2-D Convolution	(32, 32, 64)	64	9x9	7	5,248
2-D Convolution	(32, 32, 4,096)	64	5x5	8	102,464
2-D Convolution	(32, 32, 1)	1	5x5	8	1,601
Addition	(32, 32, 1)	N/A	N/A	8	N/A

III. RESULTS

A. Training

To determine which proposed method produces images with the greatest similarity and quality, each model was trained on two datasets. The first dataset was the Berkeley Segmentation Data Set and Benchmarks 500 (BSDS500), which contains 300 different natural training images. The second dataset that was used was the Caltech 256 image dataset. This dataset contains 30,608 images of objects in 256 different categories. Each model was trained on both datasets for 50 epochs while the SSIM, MSE, PSNR recorded.

B. Testing

After training each of the trained models were verified using 20 different test images collected from the Set5 and Set14 dataset. These datasets contain images with high frequency details and are commonly used in image processing for testing. Each image was resized to 256x256, downsampled to 128x128 and then upsampled back to 256x256 using the standard bilinear interpolation method for image resizing. The models were then tested on the scaled images. The MSE, PSNR, and SSIM was then calculated for each output image and their corresponding ground truth image. A table showing the results of all 20 images for each of the methods can be seen in Tables III-XI, in the appendix.

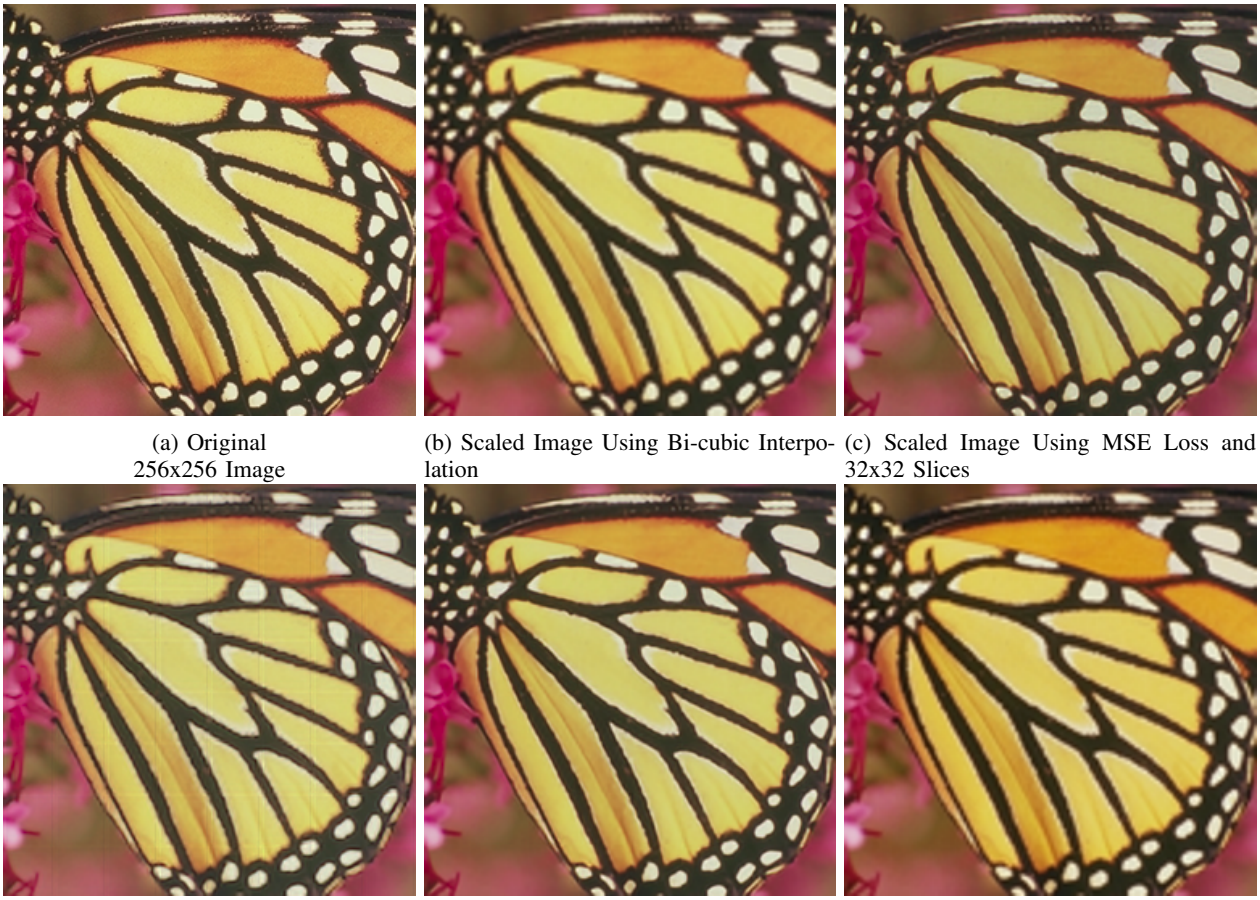
C. Discussion

The statistics recorded from the testing of each model displays some surprising results. For each model trained using the DSSIM as the loss function, the test results displayed MSE, PSNR, and SSIM values incredibly similar to the original scaled input images into the models, regardless of the size of the slices. It can then be inferred that it is likely not beneficial to use DSSIM as a loss function for this particular application.

On the other hand, using MSE as a loss function proved to result in greater visual similarity between the original ground truth image and the output image. While looking at the average test MSE of each MSE trained model in Table II, it may

TABLE II: Average Results for the Different Methods Tested

Dataset	Loss Function	Color Space	Slice Size	Average MSE	Average PSNR	Average SSIM
BSDS500	Scaled	RGB	N/A	116.19	28.68	0.82
BSDS500	SSIM	RGB	8x8	116.53	28.69	0.83
BSDS500	SSIM	RGB	16x16	116.28	28.71	0.83
BSDS500	SSIM	RGB	32x32	116.38	23.42	0.81
BSDS500	SSIM	RGB	64x64	116.08	28.71	0.82
Caltech256	SSIM	RGB	32x32	116.08	28.70	0.83
BSDS500	MSE	RGB	8x8	336.86	27.62	0.84
BSDS500	MSE	RGB	16x16	378.83	27.23	0.84
BSDS500	MSE	RGB	32x32	347.77	27.25	0.85
BSDS500	MSE	RGB	64x64	359.46	27.15	0.85
Caltech256	MSE	RGB	32x32	162.99	29.49	0.86



(a) Original 256x256 Image

(b) Scaled Image Using Bi-cubic Interpolation

(c) Scaled Image Using MSE Loss and 32x32 Slices

(d) Scaled Image Using SSIM Loss and 32x32 Slices

(e) Scaled Image Using MSE Loss and 32x32 Slices (Caltech256)

(f) Scaled Image Using SSIM Loss and 32x32 Slices (Caltech256)

Fig. 3: Comparison of Different Methods (Butterfly)

appear that the MSE trained models have a lower performance, reviewing Tables III-XI, in the appendix show that for many images MSE trained models perform significantly greater than the original scaled bilinear interpolation method. However, for some images MSE trained models, perform significantly worse than all other methods. It can be assumed that MSE trained models perform well for images similar to the training dataset, but poorly for images with features highly different than the model has seen before. An example of this scenario is displayed in Figure 4g-4i. The reason why the SRCNN performs poorly in this scenario could be due to the fact that no images with text were included in the training dataset. This would result in the SRCNN not learning how to handle images of this type.

Another important conclusion to draw from the recorded data, is that the highest performing size of slices appears to be 32x32 with 16x16 a close second. Slice sizes of 64x64 is likely too great of pixel information to model well, while 8x8 is likely too little of pixel information to model with the surrounding slices.

(a) Original 256x256 Image
(b) Scaled Image Using Bi-cubic Interpolation
(c) Scaled Image Using MSE Loss and 32x32 Slices (Caltech256)
(d) Original 256x256 Image
(e) Scaled Image Using Bi-cubic Interpolation
(f) Scaled Image Using MSE Loss and 32x32 Slices (Caltech256)
(g) Original 256x256 Image
(h) Scaled Image Using Bi-cubic Interpolation
(i) Scaled Image Using MSE Loss and 32x32 Slices (Caltech256)

Fig. 4: Comparison of Our Method to Bi-cubic Interpolation. The best testing result can be seen in Figures 4a-4c. The average testing result can be seen in Figures 4d-4f. The worst testing result can be seen in Figures 4g-4i.

IV. Conclusion

An overview of different super resolution through deep learning methods were modeled and tested. Training deep learning models to produce images with greater quality can be difficult due to the complicated and imprecise mathematical modeling of visual perception. Testing trained models and viewing the results is highly important for gaining intuition about why a model may or may not be performing as expected. Although SSIM is held as a high quality metric for image similarity, it does not perform well being used as a loss function for super resolution applications. While MSE is not considered a quality determinant of image similarity, it surprisingly performs well as a SRCNN loss function.

In conclusion, the highest performing SRCNN model consists of our proposed architecture using an MSE loss function, trained on 32x32 image slices with color channels split and trained separately.

REFERENCES

[1] C. Dong, C. C. Loy, K. He, and X. Tang, "Image super-resolution using deep convolutional networks," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 38, no. 2, pp. 295–307, Feb 2016.

[2] J. Yang, J. Wright, T. S. Huang, and Y. Ma, "Image super-resolution via sparse representation," *Trans. Img. Proc.*, vol. 19, no. 11, pp. 2861–2873, Nov. 2010. [Online]. Available: http://dx.doi.org/10.1109/TIP.2010.2050625

[3] J. Yang, J. Wright, T. Huang, and Y. Ma, "Image super-resolution as sparse representation of raw image patches," in *2008 IEEE Conference on Computer Vision and Pattern Recognition*, June 2008, pp. 1–8.

APPENDIX
TEST RESULTS

TABLE III: Results of using the traditional bicubic interpolation method of resizing on RGB images

Dataset	Loss Function	Color Space	Slice Size	Image	MSE	PSNR	SSIM
BSDS500	scaled	RGB	N/A	baboon	198.75	25.15	0.6507
BSDS500	scaled	RGB	N/A	baby	35.2	32.67	0.8852
BSDS500	scaled	RGB	N/A	barbara	118.98	27.38	0.7905
BSDS500	scaled	RGB	N/A	baboon	198.67	25.15	0.6503
BSDS500	scaled	RGB	N/A	baby	34.74	32.72	0.8856
BSDS500	scaled	RGB	N/A	barbara	118.66	27.39	0.7903
BSDS500	scaled	RGB	N/A	bird	21.86	34.73	0.9394
BSDS500	scaled	RGB	N/A	butterfly	159.01	26.12	0.798
BSDS500	scaled	RGB	N/A	coastguard	103.73	27.97	0.7029
BSDS500	scaled	RGB	N/A	comic	208.3	24.94	0.7591
BSDS500	scaled	RGB	N/A	face	28	33.66	0.8243
BSDS500	scaled	RGB	N/A	flowers	127.63	27.07	0.8099
BSDS500	scaled	RGB	N/A	foreman	51.09	31.05	0.9086
BSDS500	scaled	RGB	N/A	lenna	47.35	31.38	0.8806
BSDS500	scaled	RGB	N/A	monarch	121.07	27.3	0.8875
BSDS500	scaled	RGB	N/A	pepper	51.59	31.01	0.8957
BSDS500	scaled	RGB	N/A	ppt3	328.2	22.97	0.8318
BSDS500	scaled	RGB	N/A	woman	48.55	31.27	0.914
BSDS500	scaled	RGB	N/A	zebra	206.13	24.99	0.7603

TABLE IV: Results of using MSE as a loss function with 64x64 image slicing method of resizing on RGB images

Dataset	Loss Function	Color Space	Slice Size	Image	MSE	PSNR	SSIM
BSDS500	MSE	RGB	64x64	baboon	168.11	25.87	0.7007
BSDS500	MSE	RGB	64x64	baby	50.32	31.11	0.9056
BSDS500	MSE	RGB	64x64	barbara	93.19	28.44	0.824
BSDS500	MSE	RGB	64x64	bird	34.74	32.72	0.9492
BSDS500	MSE	RGB	64x64	butterfly	82.89	28.95	0.8758
BSDS500	MSE	RGB	64x64	coastguard	177.46	25.64	0.7399
BSDS500	MSE	RGB	64x64	comic	530.34	20.89	0.7963
BSDS500	MSE	RGB	64x64	face	24.56	34.23	0.838
BSDS500	MSE	RGB	64x64	flowers	262.63	23.94	0.8391
BSDS500	MSE	RGB	64x64	foreman	73.1	29.49	0.9286
BSDS500	MSE	RGB	64x64	lenna	39.72	32.14	0.9017
BSDS500	MSE	RGB	64x64	monarch	105.73	27.89	0.9124
BSDS500	MSE	RGB	64x64	pepper	133.71	26.87	0.906
BSDS500	MSE	RGB	64x64	ppt3	3378.11	12.84	0.7128
BSDS500	MSE	RGB	64x64	woman	35.15	32.67	0.9312
BSDS500	MSE	RGB	64x64	zebra	561.53	20.64	0.7828

TABLE V: Results of using MSE as a loss function with 32x32 image slicing method of resizing on RGB images

Dataset	Loss Function	Color Space	Slice Size	Image	MSE	PSNR	SSIM
BSDS500	MSE	RGB	32x32	baboon	168.61	25.86	0.7017
BSDS500	MSE	RGB	32x32	baby	38.85	32.24	0.906
BSDS500	MSE	RGB	32x32	barbara	95.14	28.35	0.8254
BSDS500	MSE	RGB	32x32	bird	39.14	32.2	0.9499
BSDS500	MSE	RGB	32x32	butterfly	85.55	28.81	0.8769
BSDS500	MSE	RGB	32x32	coastguard	189.77	25.35	0.7394
BSDS500	MSE	RGB	32x32	comic	543.51	20.78	0.7961
BSDS500	MSE	RGB	32x32	face	24.62	34.22	0.8379
BSDS500	MSE	RGB	32x32	flowers	251.45	24.13	0.8409
BSDS500	MSE	RGB	32x32	foreman	52.73	30.91	0.9272
BSDS500	MSE	RGB	32x32	lenna	43.3	31.77	0.9016
BSDS500	MSE	RGB	32x32	monarch	97.79	28.23	0.9134
BSDS500	MSE	RGB	32x32	pepper	118.24	27.4	0.907
BSDS500	MSE	RGB	32x32	ppt3	3106.28	13.21	0.7147
BSDS500	MSE	RGB	32x32	woman	35.02	32.69	0.9318
BSDS500	MSE	RGB	32x32	zebra	674.26	19.84	0.7792

TABLE VI: Results of using MSE as a loss function with 16x16 image slicing method of resizing on RGB images

Dataset	Loss Function	Color Space	Slice Size	Image	MSE	PSNR	SSIM
BSDS500	MSE	RGB	16x16	baboon	168.16	25.87	0.7016
BSDS500	MSE	RGB	16x16	baby	50.39	31.11	0.9018
BSDS500	MSE	RGB	16x16	barbara	92.93	28.45	0.8268
BSDS500	MSE	RGB	16x16	bird	21.6	34.79	0.9501
BSDS500	MSE	RGB	16x16	butterfly	80.93	29.05	0.8757
BSDS500	MSE	RGB	16x16	coastguard	161.51	26.05	0.7428
BSDS500	MSE	RGB	16x16	comic	561	20.64	0.793
BSDS500	MSE	RGB	16x16	face	24.53	34.23	0.8383
BSDS500	MSE	RGB	16x16	flowers	238.89	24.35	0.8412
BSDS500	MSE	RGB	16x16	foreman	88.92	28.64	0.9221
BSDS500	MSE	RGB	16x16	lenna	37.95	32.34	0.9
BSDS500	MSE	RGB	16x16	monarch	98.6	28.19	0.9137
BSDS500	MSE	RGB	16x16	pepper	139.5	26.68	0.9042
BSDS500	MSE	RGB	16x16	ppt3	3622.88	12.54	0.6632
BSDS500	MSE	RGB	16x16	woman	35.03	32.69	0.9306
BSDS500	MSE	RGB	16x16	zebra	638.45	20.08	0.7796

TABLE VII: Results of using MSE as a loss function with 8x8 image slicing method of resizing on RGB images

Dataset	Loss Function	Color Space	Slice Size	Image	MSE	PSNR	SSIM
BSDS500	MSE	RGB	8x8	baboon	172.46	25.76	0.692
BSDS500	MSE	RGB	8x8	baby	44.07	31.69	0.8981
BSDS500	MSE	RGB	8x8	barbara	91.3	28.53	0.8251
BSDS500	MSE	RGB	8x8	bird	22.88	34.54	0.9468
BSDS500	MSE	RGB	8x8	butterfly	76.53	29.29	0.8704
BSDS500	MSE	RGB	8x8	coastguard	130.97	26.96	0.7385
BSDS500	MSE	RGB	8x8	comic	427.49	21.82	0.793
BSDS500	MSE	RGB	8x8	face	25.43	34.08	0.8338
BSDS500	MSE	RGB	8x8	flowers	200.53	25.11	0.8374
BSDS500	MSE	RGB	8x8	foreman	59.65	30.37	0.922
BSDS500	MSE	RGB	8x8	lenna	39.6	32.15	0.8977
BSDS500	MSE	RGB	8x8	monarch	93.21	28.44	0.9107
BSDS500	MSE	RGB	8x8	pepper	118.21	27.4	0.9029
BSDS500	MSE	RGB	8x8	ppt3	3245.41	13.02	0.6552
BSDS500	MSE	RGB	8x8	woman	36.76	32.48	0.9274
BSDS500	MSE	RGB	8x8	zebra	605.38	20.31	0.7734

TABLE VIII: Results of using SSIM as a loss function with 8x8 image slicing method of resizing on RGB images

Dataset	Loss Function	Color Space	Slice Size	Image	MSE	PSNR	SSIM
BSDS500	SSIM	RGB	8x8	baboon	199.25	25.14	0.6503
BSDS500	SSIM	RGB	8x8	baby	35.41	32.64	0.8854
BSDS500	SSIM	RGB	8x8	barbara	119.29	27.36	0.7904
BSDS500	SSIM	RGB	8x8	bird	22.5	34.61	0.9368
BSDS500	SSIM	RGB	8x8	butterfly	159.67	26.1	0.798
BSDS500	SSIM	RGB	8x8	coastguard	104.35	27.95	0.703
BSDS500	SSIM	RGB	8x8	comic	209.01	24.93	0.7591
BSDS500	SSIM	RGB	8x8	face	28.65	33.56	0.8221
BSDS500	SSIM	RGB	8x8	flowers	128.31	27.05	0.8098
BSDS500	SSIM	RGB	8x8	foreman	51.65	31	0.9086
BSDS500	SSIM	RGB	8x8	lenna	48.02	31.32	0.8805
BSDS500	SSIM	RGB	8x8	monarch	121.72	27.28	0.8875
BSDS500	SSIM	RGB	8x8	pepper	52.11	30.96	0.8956
BSDS500	SSIM	RGB	8x8	ppt3	328.82	22.96	0.8323
BSDS500	SSIM	RGB	8x8	woman	49.2	31.21	0.9134
BSDS500	SSIM	RGB	8x8	zebra	206.58	24.98	0.7602

TABLE IX: Results of using SSIM as a loss function with 16x16 image slicing method of resizing on RGB images

Dataset	Loss Function	Color Space	Slice Size	Image	MSE	PSNR	SSIM
BSDS500	SSIM	RGB	16x16	baboon	199.02	25.14	0.6497
BSDS500	SSIM	RGB	16x16	baby	35.16	32.67	0.8843
BSDS500	SSIM	RGB	16x16	barbara	119.02	27.37	0.7896
BSDS500	SSIM	RGB	16x16	bird	22.25	34.66	0.9355
BSDS500	SSIM	RGB	16x16	butterfly	159.41	26.11	0.7971
BSDS500	SSIM	RGB	16x16	coastguard	104.17	27.95	0.7022
BSDS500	SSIM	RGB	16x16	comic	208.71	24.94	0.7586
BSDS500	SSIM	RGB	16x16	face	28.39	33.6	0.8224
BSDS500	SSIM	RGB	16x16	flowers	128.06	27.06	0.8092
BSDS500	SSIM	RGB	16x16	foreman	51.21	31.04	0.9076
BSDS500	SSIM	RGB	16x16	lenna	47.75	31.34	0.8793
BSDS500	SSIM	RGB	16x16	monarch	121.45	27.29	0.8865
BSDS500	SSIM	RGB	16x16	pepper	52.04	30.97	0.8946
BSDS500	SSIM	RGB	16x16	ppt3	328.58	22.96	0.8309
BSDS500	SSIM	RGB	16x16	woman	48.9	31.24	0.9123
BSDS500	SSIM	RGB	16x16	zebra	206.37	24.98	0.7599

TABLE X: Results of using SSIM as a loss function with 32x32 image slicing method of resizing on RGB images

Dataset	Loss Function	Color Space	Slice Size	Image	MSE	PSNR	SSIM
BSDS500	SSIM	RGB	32x32	baboon	241.08	24.31	0.6596
BSDS500	SSIM	RGB	32x32	baby	273.46	23.76	0.8637
BSDS500	SSIM	RGB	32x32	barbara	134.86	26.83	0.7929
BSDS500	SSIM	RGB	32x32	bird	223.83	24.63	0.918
BSDS500	SSIM	RGB	32x32	butterfly	148.34	26.42	0.8092
BSDS500	SSIM	RGB	32x32	coastguard	326.9	22.99	0.7056
BSDS500	SSIM	RGB	32x32	comic	536.1	20.84	0.757
BSDS500	SSIM	RGB	32x32	face	89.35	28.62	0.8181
BSDS500	SSIM	RGB	32x32	flowers	213.54	24.84	0.8087
BSDS500	SSIM	RGB	32x32	foreman	440.83	21.69	0.8798
BSDS500	SSIM	RGB	32x32	lenna	109.97	27.72	0.8693
BSDS500	SSIM	RGB	32x32	monarch	115.75	27.5	0.8899
BSDS500	SSIM	RGB	32x32	pepper	462.84	21.48	0.867
BSDS500	SSIM	RGB	32x32	ppt3	5562.5	10.68	0.6395
BSDS500	SSIM	RGB	32x32	woman	164.84	25.96	0.9014
BSDS500	SSIM	RGB	32x32	zebra	1474.03	16.45	0.712

TABLE XI: Results of using SSIM as a loss function with 64x64 image slicing method of resizing on RGB images

Dataset	Loss Function	Color Space	Slice Size	Image	MSE	PSNR	SSIM
BSDS500	SSIM	RGB	64x64	baboon	198.75	25.15	0.6507
BSDS500	SSIM	RGB	64x64	baby	35.2	32.67	0.8852
BSDS500	SSIM	RGB	64x64	barbara	118.98	27.38	0.7905
BSDS500	SSIM	RGB	64x64	bird	22.36	34.64	0.9367
BSDS500	SSIM	RGB	64x64	butterfly	158.75	26.12	0.798
BSDS500	SSIM	RGB	64x64	coastguard	104.12	27.96	0.703
BSDS500	SSIM	RGB	64x64	comic	208.27	24.94	0.7592
BSDS500	SSIM	RGB	64x64	face	28.61	33.57	0.822
BSDS500	SSIM	RGB	64x64	flowers	127.96	27.06	0.8097
BSDS500	SSIM	RGB	64x64	foreman	51.14	31.04	0.9085
BSDS500	SSIM	RGB	64x64	lenna	47.81	31.34	0.8804
BSDS500	SSIM	RGB	64x64	monarch	121.17	27.3	0.8874
BSDS500	SSIM	RGB	64x64	pepper	51.65	31	0.8956
BSDS500	SSIM	RGB	64x64	ppt3	327.85	22.97	0.8322
BSDS500	SSIM	RGB	64x64	woman	48.86	31.24	0.9133
BSDS500	SSIM	RGB	64x64	zebra	205.89	24.99	0.7602

TABLE XII: Results of using MSE as a loss function with 32x32 image slicing method of resizing on RGB images

Dataset	Loss Function	Color Space	Slice Size	Image	MSE	PSNR	SSIM
Caltech	MSE	RGB	32x32	baboon	161.93	26.04	0.7069
Caltech	MSE	RGB	32x32	baby	22.47	34.61	0.9097
Caltech	MSE	RGB	32x32	barbara	85.47	28.81	0.8365
Caltech	MSE	RGB	32x32	bird	14.2	36.61	0.9564
Caltech	MSE	RGB	32x32	butterfly	55.89	30.66	0.8923
Caltech	MSE	RGB	32x32	coastguard	110.98	27.68	0.747
Caltech	MSE	RGB	32x32	comic	239.48	24.34	0.8273
Caltech	MSE	RGB	32x32	face	24.05	34.32	0.8362
Caltech	MSE	RGB	32x32	flowers	122.06	27.27	0.8564
Caltech	MSE	RGB	32x32	foreman	35.54	32.62	0.9351
Caltech	MSE	RGB	32x32	lenna	30.55	33.28	0.9073
Caltech	MSE	RGB	32x32	monarch	63.88	30.08	0.9252
Caltech	MSE	RGB	32x32	pepper	47.34	31.38	0.919
Caltech	MSE	RGB	32x32	ppt3	1222.28	17.26	0.8245
Caltech	MSE	RGB	32x32	woman	24.11	34.31	0.9406
Caltech	MSE	RGB	32x32	zebra	347.73	22.72	0.8071

TABLE XIII: Results of using SSIM as a loss function with 32x32 image slicing method of resizing on RGB images

Dataset	Loss Function	Color Space	Slice Size	Image	MSE	PSNR	SSIM
Caltech	SSIM	RGB	32x32	baboon	198.97	25.14	0.6504
Caltech	SSIM	RGB	32x32	baby	35.21	32.66	0.8854
Caltech	SSIM	RGB	32x32	barbara	119.02	27.37	0.7905
Caltech	SSIM	RGB	32x32	bird	22.44	34.62	0.9349
Caltech	SSIM	RGB	32x32	butterfly	159.16	26.11	0.798
Caltech	SSIM	RGB	32x32	coastguard	104.08	27.96	0.7032
Caltech	SSIM	RGB	32x32	comic	208.34	24.94	0.7593
Caltech	SSIM	RGB	32x32	face	28.55	33.57	0.8228
Caltech	SSIM	RGB	32x32	flowers	128	27.06	0.81
Caltech	SSIM	RGB	32x32	foreman	51.28	31.03	0.9087
Caltech	SSIM	RGB	32x32	lenna	47.85	31.33	0.8807
Caltech	SSIM	RGB	32x32	monarch	121.31	27.29	0.8876
Caltech	SSIM	RGB	32x32	pepper	52.07	30.97	0.8957
Caltech	SSIM	RGB	32x32	ppt3	327.47	22.98	0.8324
Caltech	SSIM	RGB	32x32	woman	48.84	31.24	0.9131
Caltech	SSIM	RGB	32x32	zebra	206.12	24.99	0.7601

PYTHON SCRIPTS

A. *TrainRGB.py*

```
import numpy as np
import keras.backend as K
from os import listdir
from os.path import join
from scipy import misc
from keras.models import Model
from keras.layers import Input, Conv2D
from keras.layers.merge import add
from keras.optimizers import Adam
from keras.callbacks import CSVLogger, EarlyStopping, ModelCheckpoint
from keras.initializers import RandomNormal
from keras_contrib.losses import DSSIMObjective

#%% Define paths and variables
input_dir = 'Caltech15k/input'
label_dir = 'Caltech15k/label'
model_dir = 'Caltech15k/model'
model_name = ('Rmodel.json', 'Gmodel.json', 'Bmodel.json')
model_weights_name = ('Rmodel_weights.h5', 'Gmodel_weights.h5', 'Bmodel_weights.h5')
log_name = ('Rmodel', 'Gmodel', 'Bmodel')
epochs= 50
batch_size=500


# Define the window size
windowsize_r = 32
windowsize_c = 32

#%% Import and splice images
X = []
for f in listdir(input_dir):
        test_image = np.asarray(misc.imread(join(input_dir, f)))
        # Crop out the window and calculate the histogram
        for r in range(0,test_image.shape[0] - windowsize_r, windowsize_r):
                    for c in range(0,test_image.shape[1] - windowsize_c, windowsize_c):
                        X.append(test_image[r:r+windowsize_r,c:c+windowsize_c])
y = []
for f in listdir(label_dir):
        test_image = np.asarray(misc.imread(join(label_dir, f)))
        # Crop out the window and calculate the histogram
        for r in range(0,test_image.shape[0] - windowsize_r, windowsize_r):
                    for c in range(0,test_image.shape[1] - windowsize_c, windowsize_c):
                        y.append(test_image[r:r+windowsize_r,c:c+windowsize_c])

X = np.asarray(X)
y = np.asarray(y)

#%% Define model

# Define weights and bias intializers
kernel_init = RandomNormal(mean=0.0, stddev=0.001)

# Model
inputs = Input(shape=(windowsize_r, windowsize_c, 1))
x = Conv2D(64, (9, 9), input_shape=(windowsize_r, windowsize_c, 1), activation='relu',
```

```
            kernel_initializer=kernel_init, padding='same')(inputs)
x = Conv2D(64, (5, 5), activation='relu', kernel_initializer=kernel_init,
            padding='same')(x)
x = Conv2D(1, (5, 5), kernel_initializer=kernel_init, padding='same')(x)
x = add([inputs, x])
model = Model(inputs=inputs, outputs=x)

# Define SSIM Loss
SSIM = DSSIMObjective(k1=0.01, k2=0.03, kernel_size=2, max_value=1.0)


# Stop early if loss converges
early_stop = EarlyStopping(monitor='loss', patience=2)

# Define performance metrics / loss functions
def mse(y_true, y_pred):
    return K.mean(K.square(y_pred - y_true))

def psnr(y_true, y_pred):
    max_pixel_val = float(255)
    return 10 * K.log(K.square(max_pixel_val) / mse(y_true, y_pred))

# Compile model
model.compile(optimizer=Adam(lr=0.001), loss=SSIM, metrics=[psnr, mse])


#%% Loop for R G and B channels
for j in range(3):
    # Checkpoint for saving the best performing weights
    chckpnt = ModelCheckpoint(join(model_dir, model_weights_name[j]), monitor='loss', save_bes

    # Log epoch results to CSV file
    csv_logger = CSVLogger(join(model_dir, log_name[j] + 'training.log'))

    # Train model
    model.fit(np.expand_dims(X[:,:,:,j], axis=3), np.expand_dims(y[:,:,:,j], axis=3), epochs=e
              batch_size=batch_size, callbacks=[early_stop, chckpnt, csv_logger])

    # Save parameters
    config = model.to_json()
    open(join(model_dir, model_name[j]), "w").write(config)
```

B. PredictRGBLoop.py

```
import numpy as np
from scipy.misc import imread, imsave, imresize
from keras.models import model_from_json
from os.path import join
from os import listdir
from skimage.measure import compare_ssim, compare_mse, compare_psnr

#%% Define variables
input_size = (128, 128)
label_size = (256, 256)
saved_model_dir = 'Caltech15k/model'
model_name = ('Rmodel.json', 'Gmodel.json', 'Bmodel.json')
saved_weights_name = ('Rmodel_weights.h5', 'Gmodel_weights.h5', 'Bmodel_weights.h5')
output_dir = 'output/caltech'
input_dir = 'test2'

# Define the window size
windowsize_r = 32
windowsize_c = 32

for n in listdir(input_dir):
    image_dir = n
    #%% Rescale images
    image = imread(join(input_dir,image_dir))
    scaled = imresize(image, input_size, 'bicubic')
    scaled = imresize(scaled, label_size, 'bicubic')
    ref_img = (imresize(image, label_size, 'bicubic'))

    input_img = scaled.copy()
    output_img = np.zeros(input_img.shape)

    input_img = np.asarray(input_img, dtype='float32')
    output_img = np.asarray(output_img, dtype='float32')
    ref_img = np.asarray(ref_img, dtype='float32')
    scaled = np.asarray(scaled, dtype='float32')

    #%% Loop over all channels
    for i in range(3):

        # Load model and weights separately due to error in keras for custom loss functions
        model = model_from_json(open(join(saved_model_dir, model_name[i])).read())
        model.load_weights(join(saved_model_dir, saved_weights_name[i]))

        # Separate by channel
        input_chan = input_img[:,:,i]
        input_chan = np.expand_dims(input_chan, axis=3)
        model_input = []
        model_output = np.zeros(input_chan.shape)

        # Splice image channel
        for r in range(0,input_chan.shape[0] - windowsize_r + 1, windowsize_r):
                for c in range(0,input_chan.shape[1] - windowsize_c + 1, windowsize_c):
                        model_input.append(input_chan[r:r+windowsize_r,c:c+windowsize_c])
        model_input = np.asarray(model_input)

        # Predict output
```

```
    output = model.predict(model_input)


    # Reconstruct channel from slices
    j = 0
    for r in range(0, model_output.shape[0] - windowsize_r + 1, windowsize_r):
            for c in range(0,model_output.shape[1] - windowsize_c + 1, windowsize_c):
                    model_output[r:r+windowsize_r,c:c+windowsize_c] = output[j,:,:]
                    j += 1

    # Add reconstructed output to channel index
    model_output = np.squeeze(model_output)
    output_img[:,:,i] = model_output


#%% Save images
#output_img = output_img.astype('uint8')
#imsave(join(output_dir, image_dir[:-4] + "_scaled_image.bmp"), scaled)
imsave(join(output_dir, image_dir[:-4] + "_output_image.bmp"), output_img)
#imsave(join(output_dir, image_dir[:-4] + "_reference_image.bmp"), ref_img)

#%% Print statistics
output_img = np.asarray(output_img, dtype='uint8')
ref_img = np.asarray(ref_img, dtype='uint8')
scaled = np.asarray(scaled, dtype='uint8')
print('Scaled MSE:', compare_mse(ref_img, scaled), 'PSNR:', compare_psnr(ref_img, scaled),
print('Output MSE:', compare_mse(ref_img, output_img), 'PSNR:', compare_psnr(ref_img, outp

#%% Output statistics to csv
with open("SuperRes.csv", "a") as fp:
        fp.write(saved_model_dir + ',')
        fp.write(image_dir[:-4] + ',')
        fp.write('%.2f,' % compare_mse(ref_img, output_img))
        fp.write('%.2f,' % compare_psnr(ref_img, output_img))
        fp.write('%.4f\n' % compare_ssim(ref_img, output_img,win_size=3,multichannel=True)
```

C. Resize.py

```
from os import listdir
from os.path import join
from scipy import misc

input_size = (128, 128)
label_size = (256, 256)

input_dir = 'BSDS500'
output_dir = 'BSDS500Resized'

for f in listdir(input_dir):
    input_name = f
    f = join(input_dir, f)

    image = misc.imread(f, mode='RGB')

    scaled = misc.imresize(image, input_size, 'bicubic')
    scaled = misc.imresize(scaled, label_size, 'bicubic')

    image = misc.imresize(image, label_size, 'bicubic')

    misc.imsave(join(output_dir, "input",  input_name), scaled)
    misc.imsave(join(output_dir, "label", input_name), image)
```