A CNN Based Single Image Super-Resolution Using Residual Networks With Non-Local Multiple Image Phases

Al- Khattab Ali Y. Al-Qaseem

Submitted to the
Institute of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Master of Science
in
Electrical and Electronic Engineering

Eastern Mediterranean University
February 2022
Gazimağusa, North Cyprus

Approval of the Institute of Graduate Studies and Research

Prof. Dr. Ali Hakan Ulusoy
Director

I certify that this thesis satisfies all the requirements as a thesis for the degree of Master of Science in Electrical and Electronic Engineering.

Assoc. Prof. Dr. Rasime Uyguroğlu
Chair, Department of Electrical and Electronic Engineering

We certify that we have read this thesis and that in our opinion it is fully adequate in scope and quality as a thesis for the degree of Master of Science in Electrical and Electronic Engineering.

Prof. Dr. Hüseyin Özkaramanlı
Supervisor

Examining Committee

1. Prof. Dr. Hasan Amca
2. Prof. Dr. Bülent Bilgehan
3. Prof. Dr. Hüseyin Özkaramanlı

ABSTRACT

Image-super resolution is a fast-growing research field that has been enhanced by the introduction of deep learning methods that achieved greater results and performance.

Convolutional deep networks that utilize residual architecture have shown the best performance. One of the best models that use this architecture is the EDSR model [16], which showed a great result in extracting and reconstructing images but the features that it extracts are not always optimal due to the locality of the convolutional window which might result in missing some general key features about the image from the areas outside the convolution kernel of the image.

In this project, we propose an improvement on the convolutional network architecture that uses residual blocks to detect the local features of the image by adding a phased version of the input that will add the missing nonlocal features and improve the quality of the feature space that will result in a better reconstruction.

We achieved our objective by introducing a new phasor block to the model which will create different perspectives of the image which we trained using a smaller version of EDSR for each phase then concatenated the results of the original and the phases into one big feature space containing deep and shallow features which enhanced the reconstruction of the image.

Keywords: Convolutional Networks, Residual Networks, EDSR, Phasor

ÖZ

Görüntü ivuper çözünürlüğü, daha büyük sonuçlar ve performans elde eden derin öğrenme yöntemlerinin tanıtılmasıyla geliştirilmiş, hızla büyüyen bir araştırma alanıdır.

Artık mimariyi kullanan evrişimli derin ağlar en iyi performansı göstermiştir. Bu mimariyi kullanan en iyi modellerden biri, görüntülerin çıkarılması ve yeniden yapılandırılmasında harika bir sonuç gösteren EDSR modelidir[16], ancak çıkardığı özellikler, kayıpla sonuçlanabilecek evrişim penceresinin yerelliği nedeniyle her zaman optimal değildir. Görüntünün evrişim çekirdeğinin dışındaki alanlardan görüntü hakkında bazı genel temel özellikler.

Bu projede, eksik yerel olmayan özellikleri ekleyecek ve özellik uzayının kalitesini artıracak girdinin aşamalı bir sürümünü ekleyerek görüntünün yerel özelliklerini algılamak için artık blokları kullanan evrişimli ağ mimarisinde bir iyileştirme öneriyoruz. Daha iyi bir yeniden yapılanma ile sonuçlanacaktır.

Her aşama için daha küçük bir EDSR sürümü kullanarak eğittiğimiz görüntünün farklı perspektiflerini yaratacak modele yeni bir fazör bloğu ekleyerek hedefimize ulaştık, ardından orijinalin ve fazların sonuçlarını derin içeren tek bir büyük özellik alanında birleştirdik. Ye görüntünün yeniden yapılandırılmasını geliştiren sığ özellikler.

Anahtar Kelimeler: Evrişimli Ağlar, Artık Ağlar, EDSR, Fazör

ACKNOWLEDGEMENTS

I would like to thank Prof. Dr Hüseyin özkaramanli for his supervision, assistance, and guidance from the very beginning of this thesis, as well as for providing me with incredible experiences throughout the process. Above all, and most importantly, he constantly encouraged and supported me in many ways. His thoughts, experiences, advice, and ambitions have enriched and motivated my development as a student and as a person. He is a true mentor and it is my greatest honour to be his student. I owe him far more than he realizes.

I would also like to express my gratitude to my dear parents who are the pillars that supported me in all aspects to whom I owe too much. I would also like to thank all my friends who were always there for me and helped me through the toughest time, they are like my second family and I wish the best for them.

Lastly, I would like to thank all the electrical department staff for their work and effort in creating such a great learning, developing and working environment filled with helpful, friendly and wonderful people and I am proud of being a student of this department.

To all who are not mentioned but in one way or another helped in the completion of this study, thank you very much.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

CNN	Convolutional Neural Networks
SR	Super Resolution
PSNR	Peak Signal to Noise Ratio
SSIM	Structural Similarity Index Measurement
PENsr	Name of Our model
EDSR	Enhanced Deep Residual Networks model
SRCNN	Super Resolution Convolutional Neural Network model
ResNet	Residual Network model
SRResNet	Super Resolution Residual Network model
ReLU	Rectified Linear Unit

Chapter 1

INTRODUCTION

In this chapter, we will give a brief introduction to the Super-resolution problem as well as some of the methods used to solve this problem.

1.1 Definition

Image super-resolution is the process of obtaining a high-quality image from a low quality and resolution counterpart of the same image, this process applies to all image restoration fields like image compression and denoising. The field of image reconstruction and especially single image reconstruction has gained a great deal of popularity recently due to the advancement of digital data technologies and the rising need for high-quality images in many fields.

Therefore, many methods have been developed to understand the relation between high-resolution images and their lower-resolution counterparts, but finding a direct relation is not an easy task, mainly because each image is unique and other factors like blur, noise and decimation might lead to inaccuracy interpretation of relation.

1.2 Development of methods

The early work done to solve the Super-resolution and image reconstruction problem with classical methods that use interpolations [10] yielded some good results in reconstructing the general features of the image but failed at reconstructing the details of the image, especially the realistic textures.

Other learning-based methods focused on finding the relation between the lower resolution image and the higher resolution image by utilising methods like sparse coding [11] or embedding [12].

There have been many methods that managed to reconstruct images with a certain degree of success [1,2,3] but the introduction of deep learning methods showed significant improvements in terms of Peak Signal to Noise Ratio (PSNR) and Structural Similarity Index Measurement (SSIM) for those problems, and Convolutional Neural Networks (CNNs) [4,5,6] is one of the most popular frameworks especially the architectures that use the residual networks [7,8,16] are much better than the traditional methods. Below is a comparison between a normal CNN and a CNN used for image super-resolution.

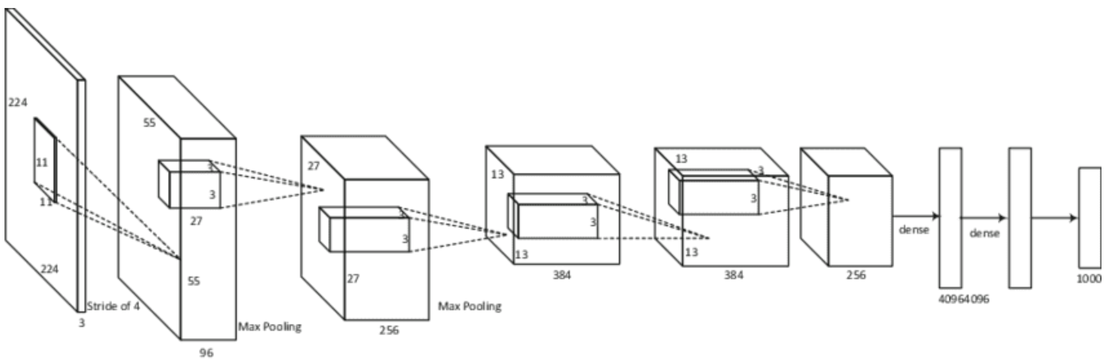


Figure 1: An example of a CNN used for image classification [23]

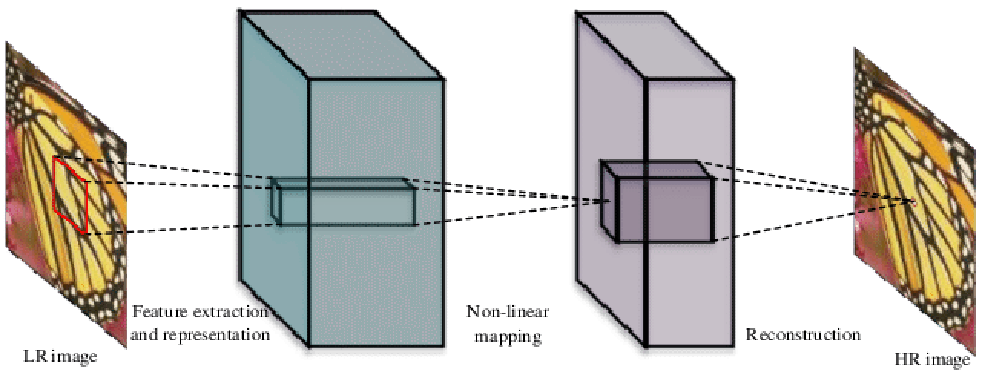


Figure 2: An example of a CNN used for image super-resolution [4]

By observing the architectures of those models, we can see that they all work towards a common goal which is extracting the most meaningful features possible that help in creating a rich feature space for a reconstruction layer (usually a CNN layer) that can be used to reconstruct a better image and usually that is done by adapting a smart architecture capable of reaching this goal but even with smart algorithms and methods they still have their drawbacks that comes from the convolutional layers. These are mainly the local processing of the convolutional kernels that are good at extracting local deep features from a small region but not good at extracting far general features of the image which might lead to missing some key aspects of the image that might help with better reconstruction.

Recently there have been many methods that are focused on going around this issue by adopting an architecture that is capable of extracting those general features like swinIR [9] but those methods use different methods like transformers.

An alternative way we can go around the CNN issues while still using the same structure is to use different phases of the image where the convolutional layer can learn features from alternative interpretations of the image that have different locality in the original image which cannot be accessed by the original kernels thus allowing the network to benefit from both deep and shallow features in the image.

1.3 Objective

In this paper, we propose an image restoration model named PENsr which uses the input and a phased version of the input to enhance the feature space quality of a model structure which we will be implementing using the EDSR architecture. Our model consists of three modules: a phases extractor, deep features extractor and a reconstructor.

The phasor extractor module consists of one layer CNN that has fixed parameters to extract 4 different phases of the image and each one has a different far feature from a different locality. The deep feature extractor is a residual network that uses the EDSR baseline architecture to extract features from the original image and 4 smaller modified versions of it to process each phase and extract features from them then the outputs of all those layers are concatenated together to form the feature space. The reconstructor module is one layer of CNN that uses all the features from the previous module to construct a high-quality image.

We experimented with various parameter numbers, datasets, input methods and training functions to get the best results training results then tested the model on benchmark datasets and compared between our model and the baseline EDSR model and also other methods.

Chapter 2

RELATED WORKS

In this chapter, we will be explaining some of the methods that had an impact on the field of Super-resolution and our research by observing their methodology and results.

2.1 SRCNN

The introduction of Deep learning methods with the SRCNN model [4] made a big jump in the performance and shifted the focus of research towards those methods where a simple 2 layer of convolution can outperform the state-of-the-art classical methods at that time.

The model takes a lower resolution image and upscale it with bicubic interpolation before sending it the 2 layers of convolution that will extract features and enhance the details of the image. One of the most advantages of this method is that it does not require any prior information about the image, the only thing needed is a big dataset that will help the model learn the mapping between the high- and low-quality images. This model was the start and there was lots of room for improvements although 2 layers can construct a good quality image it is possible to add more layers and better structure to get even better images.

2.2 Pixels shuffling

One of the issues with SRCNN was the scaling method they used, where the image is scaled using a bicubic interpolation before entering the model which introduced extra complexity to the model and added more distorted features to the image which decreased the performance. A study [13] showed that using the original low-resolution input and then upscaling it after extracting the features helped in decreasing the computational power and preserved pure features that can construct better results, where they used made the first convolutional model that extracts features in the low-resolution space then use a sub-pixel convolutional layer that learns and upscale the images using feature maps, that resulted in a much better reconstruction and it is one of the most commonly used block to upscaling method used for image reconstruction.

2.3 ResNet

The problem with deep networks is that they are very time consuming to train, although it is generally good to have deep models because it allows for the extraction of better outputs and learning many patterns that shallower networks usually cannot while also keeping in mind the complexity of the task and datasets used for the task because in some cases, training deep networks might result in worse results than the shallow ones.

The introduction of the ResNet [8] was a significant change to the deep learning field in general because this method allows for the usage of deeper networks with less training time and better performance by having short and long skip connections through the network that allowed the layers to train much faster where those skip connections add the input back to a deeper layer of the model which makes them learn faster by warming up their parameters instead of starting from scratch, this method helped with stabilising the training and solved the issue of vanishing gradient that usually comes with deeper networks.

The first model that used the residual structure for a super-resolution task was the SRResNet [7] where they used the same residual block structure to solve the image reconstruction problem which yielded better results than the SRCNN model then later this architecture was improved on by the EDSR model [16] where they got rid of some unnecessary parts of the residual block and made the architecture more suited towards the image reconstruction problem.

Chapter 3

PROPOSED METHOD

In this section, we will describe in detail our method and the model used for this task and the model structure that we applied those methods on, as well as the optimization we did on the model and as well as show the performance compared with the model based on.

As previously stated, our model has 3 modules and we will be explaining each one in detail below.

3.1 Phasor block

A classical CNN is very good at extracting local features compared with its low tuneable parameters and the variables that control it which are just the kernel size, padding and stride.

There have been many debates and research to see the effect of each one of those parameters on the quality of the output mainly the kernel size [14] studies show that increasing the kernel size might help with generalisation at the cost of accuracy and there has been a range where increasing the kernel size might actually be harmful to the performance.

With the introduction of the classification challenge ImgeNet [22] and the winning models for 2012 and 2014 AlexNet [23] and GoogleNet [24] respectively, where they reduced the kernel size to 5x5 and 3x3 respectively when it was the norm to be 12x12 before that. The impressive results that those models achieved with these kernel sizes made 3x3 and 5x5 the most popular and standard ones. Those kernel sizes work fine with classification tasks and it also works fine for Image restoration tasks where many models use it [6,5,6,7,8,16].

This brings us back to the same problem of CNN which is if we increase the kernel size to capture more general features for our task, we will make our ability to detect the key small features which is why most deep learning methods for super-resolution tasks set it to 3x3 and some even showed worse performance by using 5x5. The other observation we need to make between the low-resolution image and the high-resolution one is that the lower resolution image is a down-sampled version of the higher one.

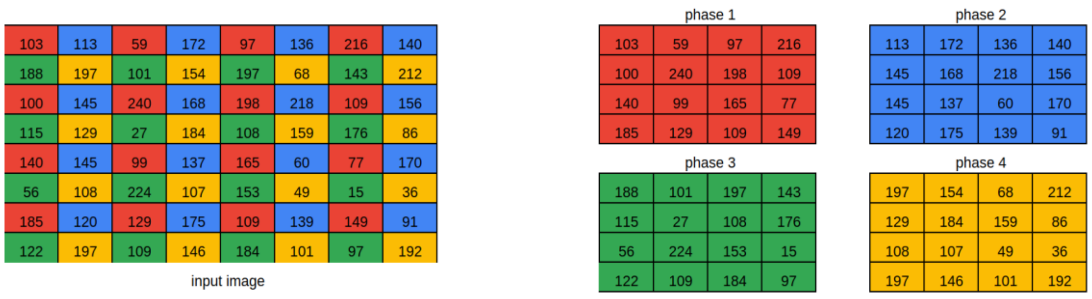
There are many methods to down-sample an image while preserving its details [15] but in general, what they do is essentially take pixels out of the image but we do not know which pixels got removed.

If we understand the process of down-sampling an image and account for all the possibilities of down-sampling, we can make a network that will learn the pattern of the down-sampling and reverse it because the pixels in the up-sampled image share similar relations.

Now that we know that we cannot use normal CNN to get far nonlocal features by just increasing the kernel size, we know that there is a pattern to learn from the down-sampling process.

We designed the phasor block that will account for them, this block is just one convolutional layer with fixed weights, kernel size, stride and padding that will take the input tensor (C, W, H) image and outputs a phased version of that tensor (P, C, W/n, H/n) each phase is calculated based on how far we want the features to be extracted (the nonlocality region) and also the down-sampling type.

The one we are using in our proposed model is a phasor block of depth 2 meaning that the kernel size of the layer is 2x2 with a stride of 2, and the output will be 4 phasors each one has a kernel weight of 1 based on the down-sampling possibility you can see the output of each of those phases as shown below.



103	113	59	172	97	136	216	140
188	197	101	154	197	68	143	212
100	145	240	168	198	218	109	156
115	129	27	184	108	159	176	86
140	145	99	137	165	60	77	170
56	108	224	107	153	49	15	36
185	120	129	175	109	139	149	91
122	197	109	146	184	101	97	192

input image

phase 1

103	59	97	216
100	240	198	109
140	99	165	77
185	129	109	149

phase 2

113	172	136	140
145	168	218	156
145	137	60	170
120	175	139	91

phase 3

188	101	197	143
115	27	108	176
56	224	153	15
122	109	184	97

phase 4

197	154	68	212
129	184	159	86
108	107	49	36
197	146	101	192

Figure 3: Example of the Phasing Process

The example shows one channel input of 8x8 size and how the phases are extracted, those phasors will then be combined in a phasor dimension where the model can access each one to send it towards their corresponding pipeline.

3.2 Deep feature extractor block

To extract deep features from images, the model has to have a deep structure capable of learning and recognizing most of the details in an image. Having a big model also comes at the cost of higher training time, overfitting and learning limits but with the introduction of ResNet [8] which countered these issues by having skip connections in the network that helps the network learn faster while having the benefits of deeper networks.

For image super-resolution tasks there have been many methods that used residual blocks in their structures but we will be using a structure similar to the one that had the best optimization of the residual structure which is the EDSR model [16]. Our deep feature extractor block has two pipelines, The main pipeline which passes the original input through the EDSR baseline structure without the last convolution

The second pipeline consists of 4 smaller versions of the main pipe modified to account for each phase and scale it to match the size of the main. Each of those pipelines consists of the following blocks

1) Shallow feature extractor implemented using one layer of convolution
2) Residual block which is a series of convolutional layers with skip connections
3) Up-sampler block which is a series of convolutional layers and pixel shuffling layer [13] that has a different structure based on the upscaling factor of the network.

The output of the main and the secondary pipelines will be the upscaled features from the lower resolution image, those upscaled features that share the same dimensions are concatenated together to form the feature space needed for the reconstruction layer.

3.3 Reconstructor

This module consists of 1 convolutional layer that will use the upscaled features from the previous module and construct an RGB high resolution and detailed image using the information from various perspectives of the image that allows for more patterns to learn.

The figures below show the model structure with the dimensionality of each part.

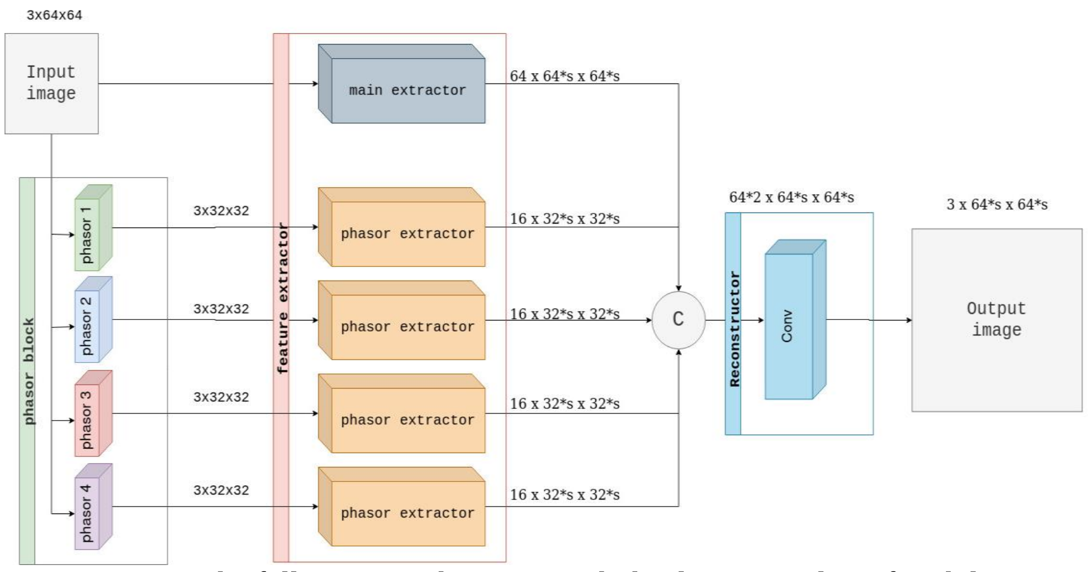


Figure 4: The full PENsr architecture with the dimensionality of each layer

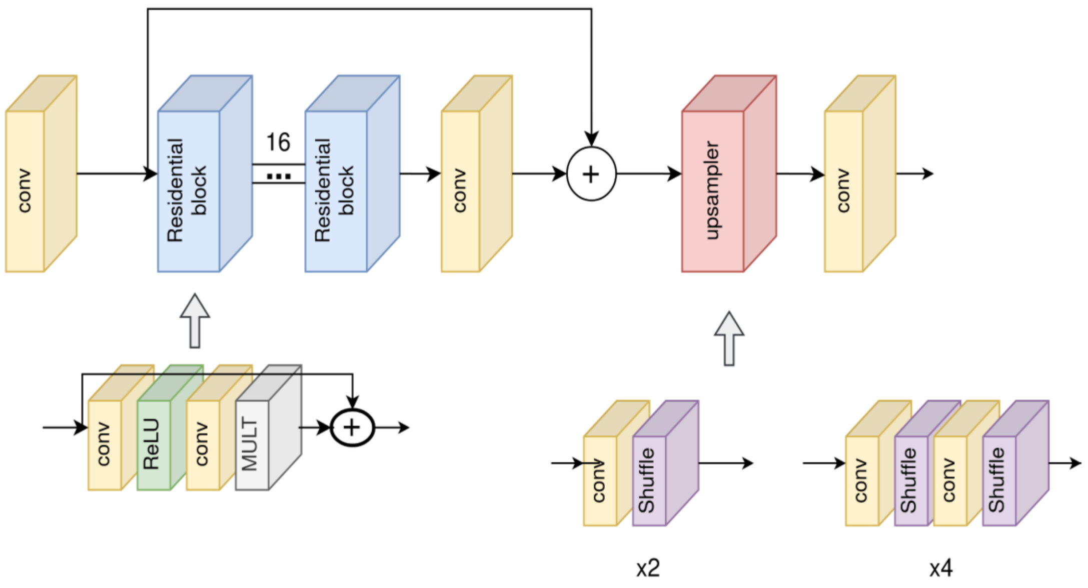


Figure 5: The main pipeline for feature extraction.

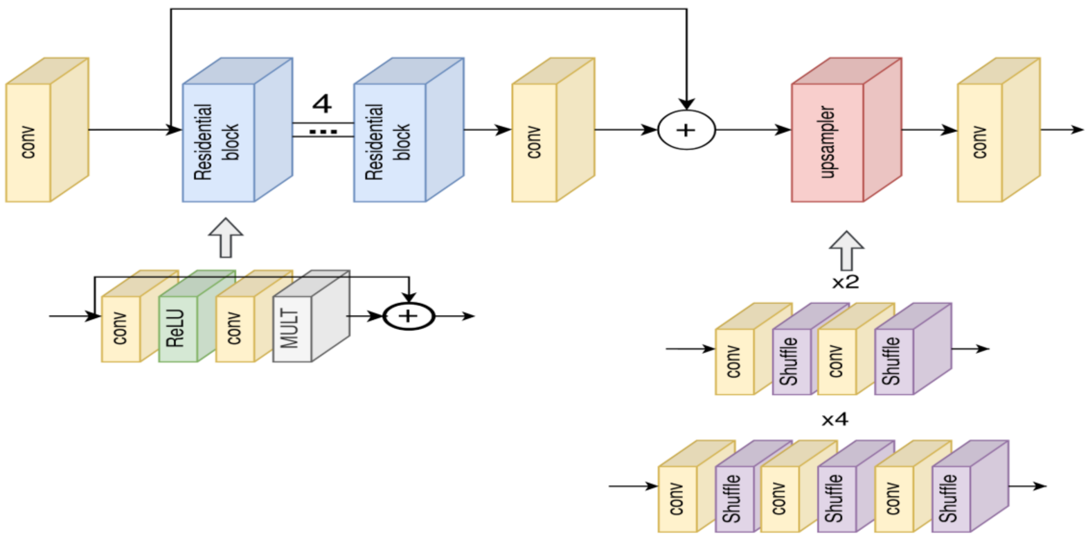


Figure 6: The secondary pipeline for feature extraction that will process the phases

3.4 PENsr model details and variation

We started our modelling process by taking the baseline of the EDSR model and modified it to fit the structure in the figure above. Since the full EDSR model has around 43 million trainable parameters and since we are using 4 more variations of the model to process the phasors that resulted in 57 million parameters to train, and since we are using a new structure we cannot import the trained parameters from the original EDSR model so we had to train the model from scratch and training such a big model will require a tremendous amount of time and computational resources, so we limited our testing to the baseline version that has only 1.5 million parameters which are easier to train and we can demonstrate the effect of adding phasors and with more parameters, we should be getting better results. The baseline model uses 16 residual blocks and 64 feature maps in each layer and no activation functions are used anywhere outside the residual blocks where we used the ReLU function.

Table 1: Model configurations and parameters

Options	SRResNet	Baseline EDSR	EDSR+	PENsr
# Residual blocks	16	16	32	16
# Filters	64	64	256	64
#Parameters	1.5M	1.5M	43M	1.7M
Residual scaling	1	1	0.1	1
Use BN	Yes	No	No	No
Loss function	L2	LI	LI	LI

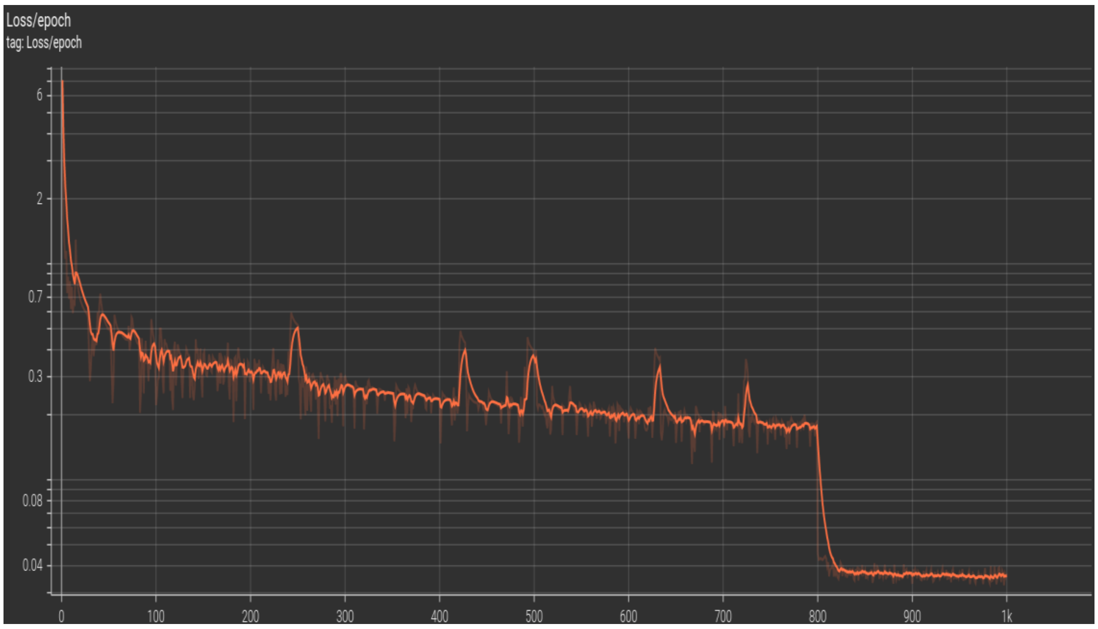


Figure 7: Training loss

The above figure shows the training loss for the x4 model on the DIV2K dataset for 100 epochs where the model converges around the 800-epoch range to an error of 0.04, the little spikes are the results of training checkpoints where if we stopped training and restarted it will result in a small jump in the error.

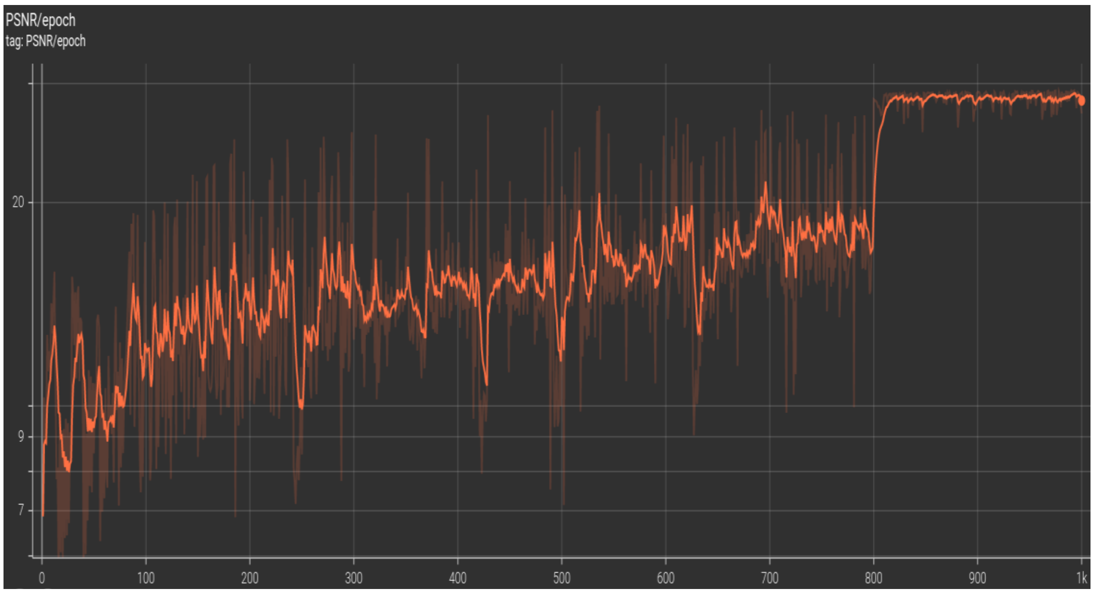


Figure 8: PSNR Evaluation

The above figure shows the PSNR evaluation of the x4 model on the set5 dataset that was obtained during training where the model converges to an average PSNR of 29db.

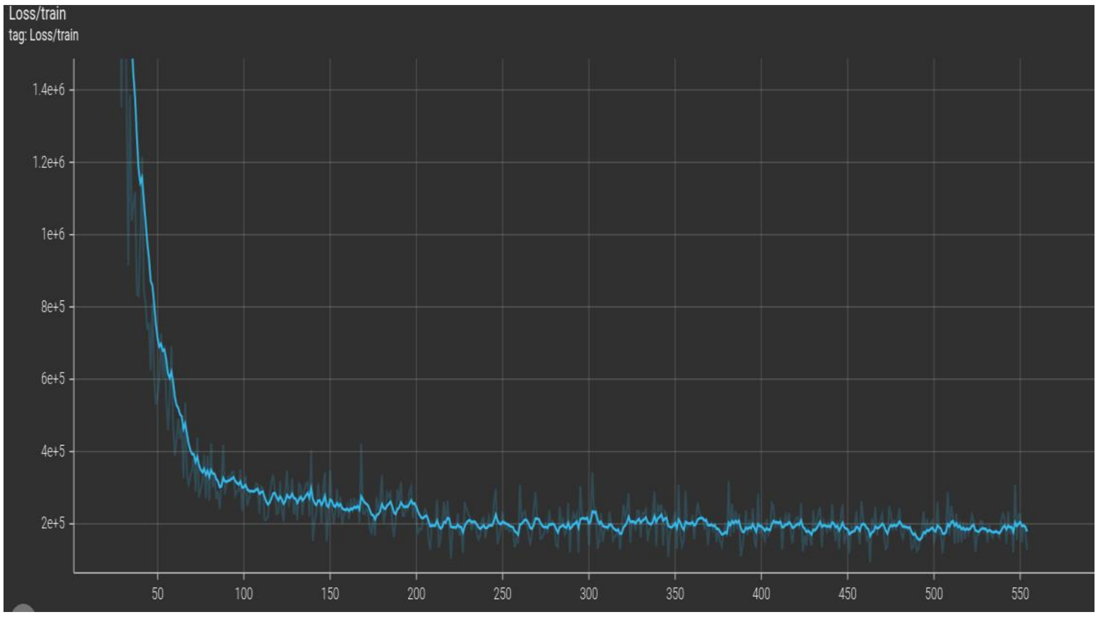


Figure 9: Stable Training

The above figure shows the loss for the x2 model that was trained on the DIV2K training set for 1000 epochs without stopping the process in between.

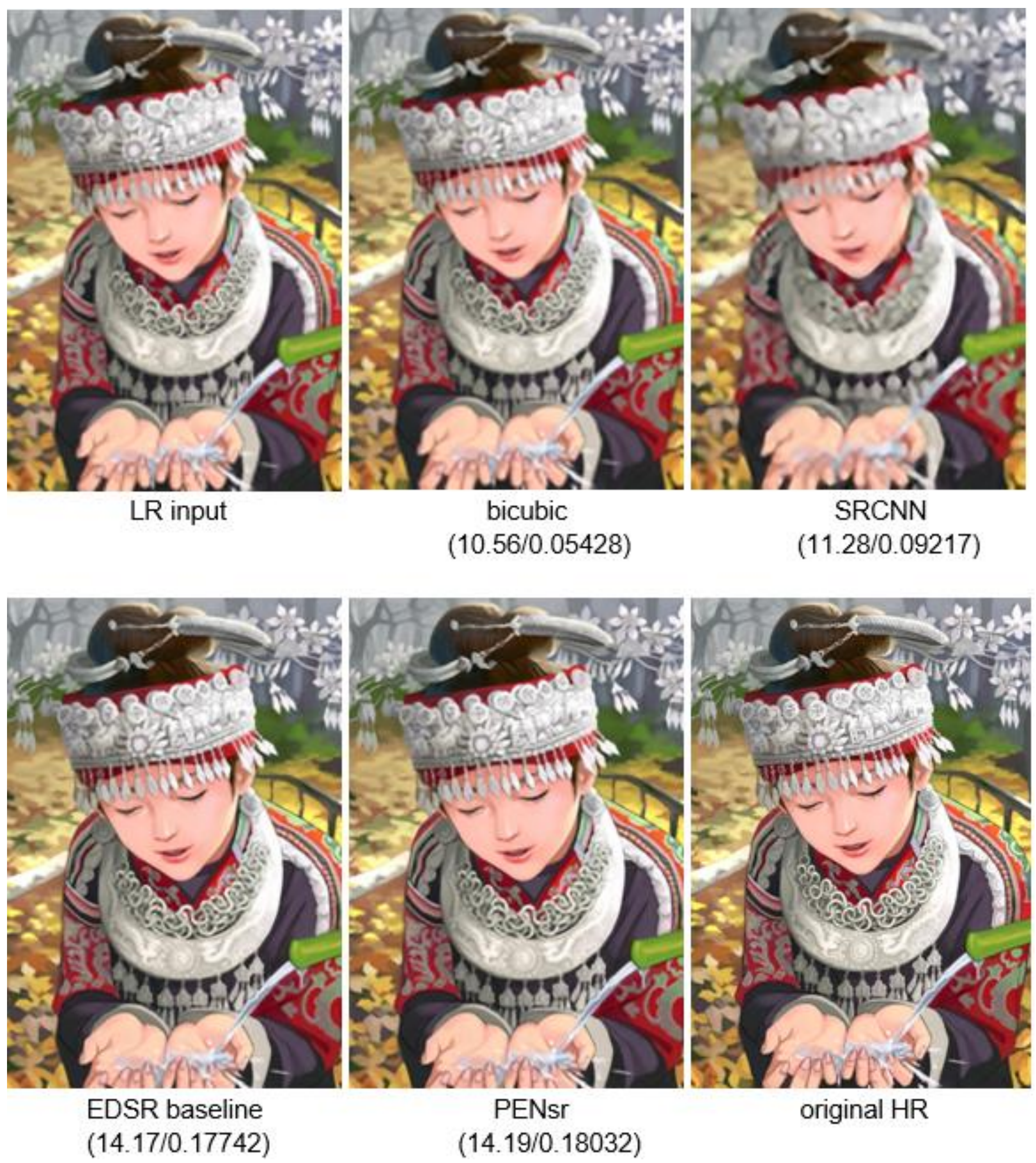


Figure 10: Test Image from set 14

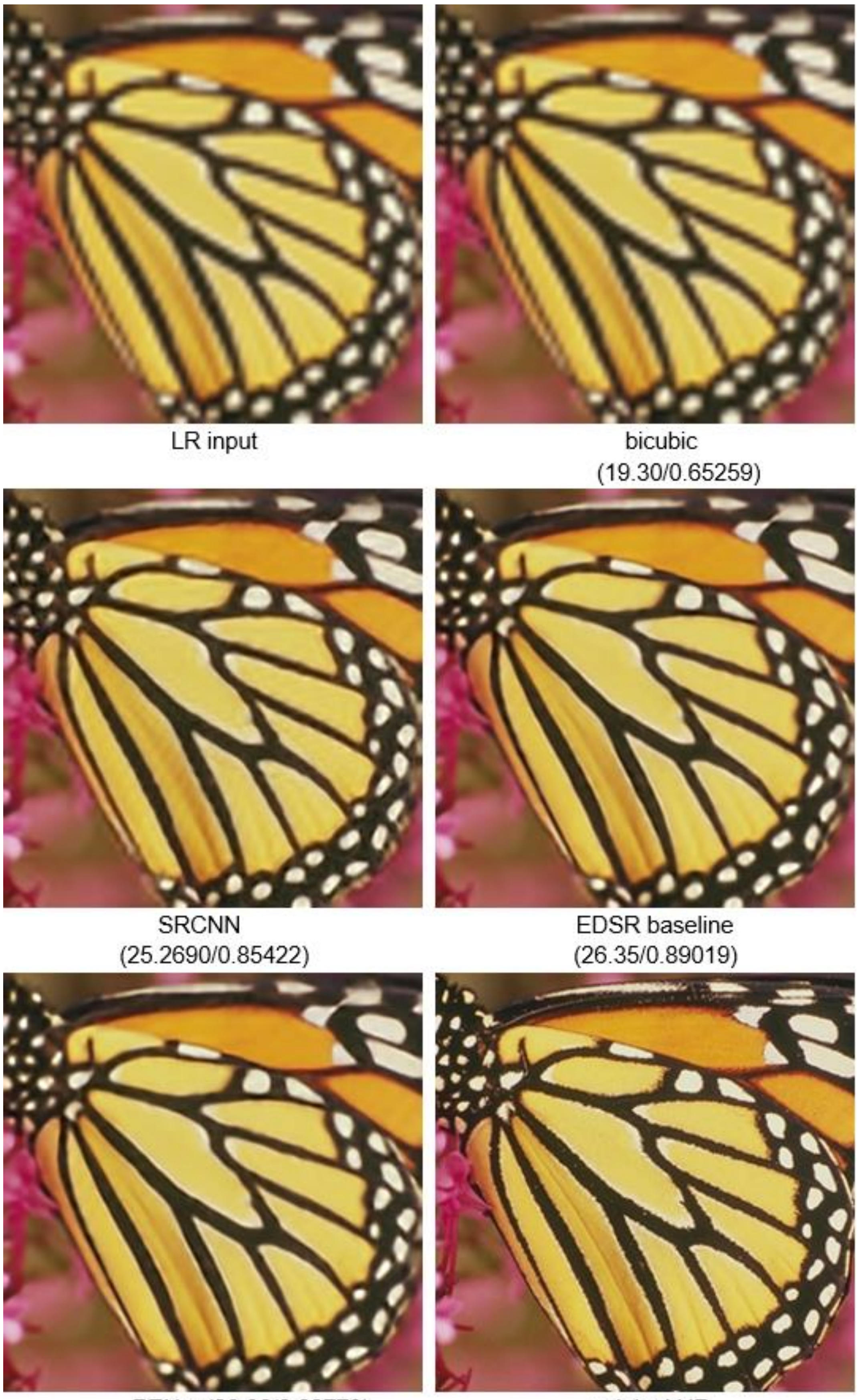


Figure 11: Test Image from set 5

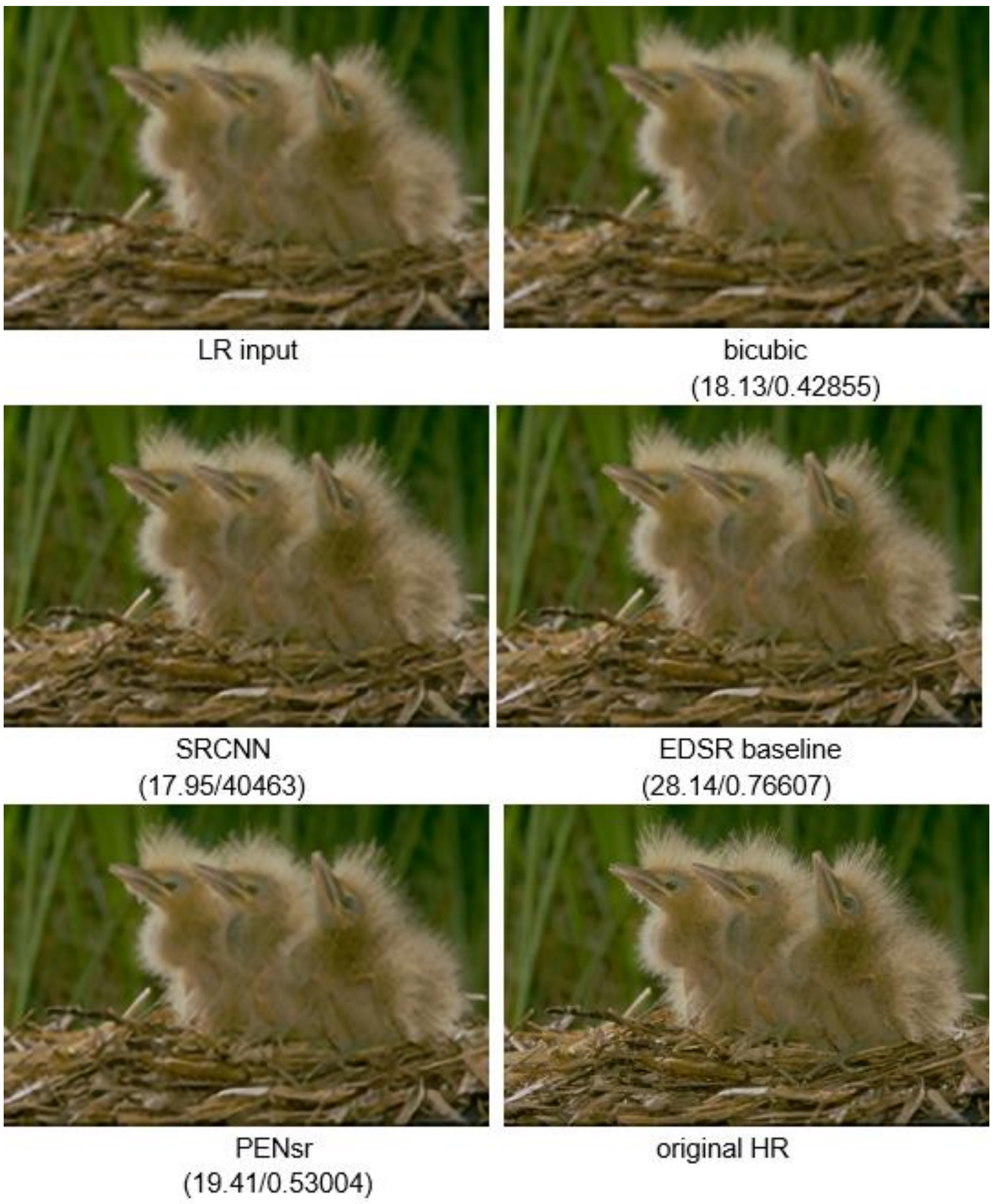


Figure 12: Test Image from BSD100

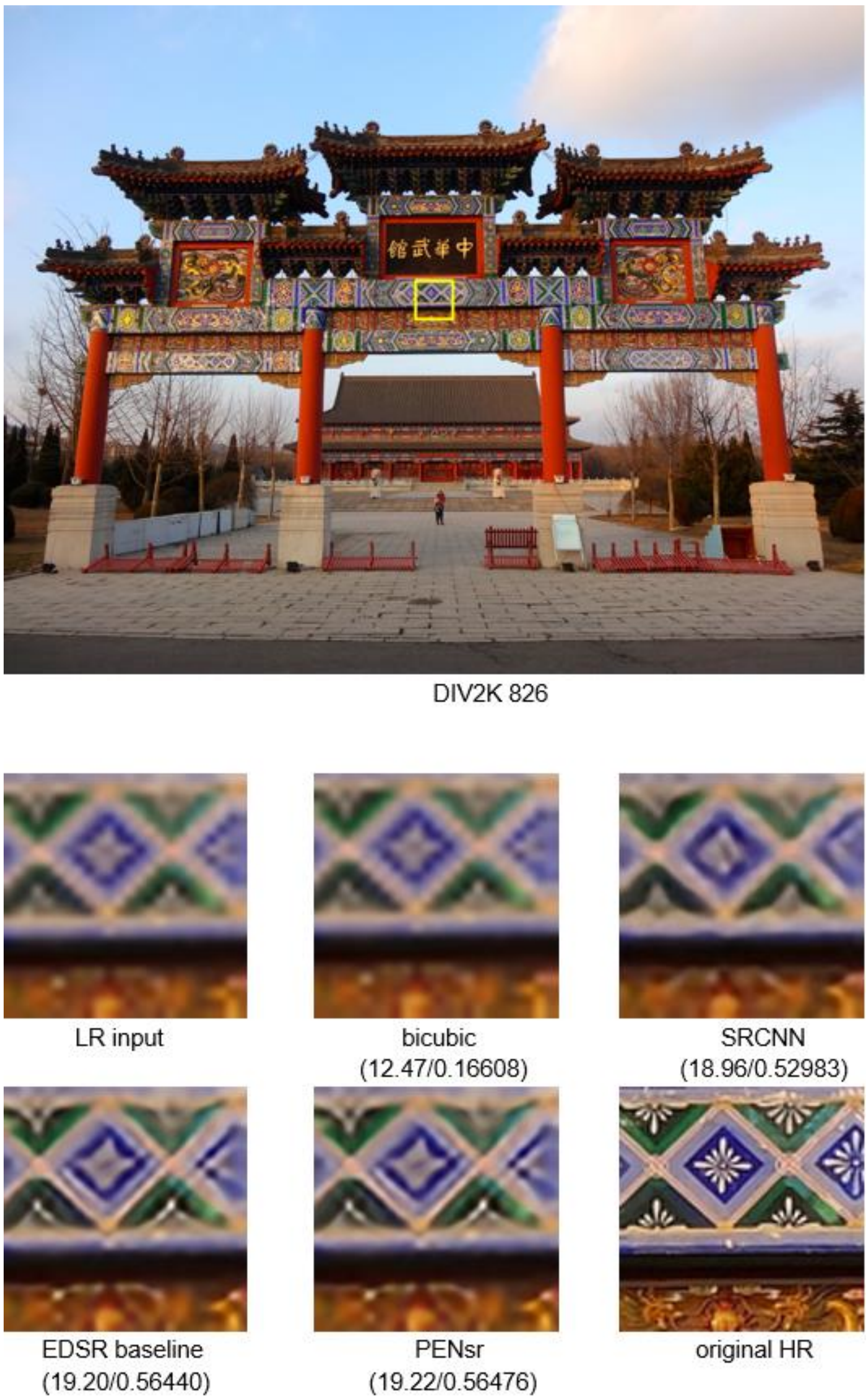


Figure 13: Test Image 1 from DIV2K validation set

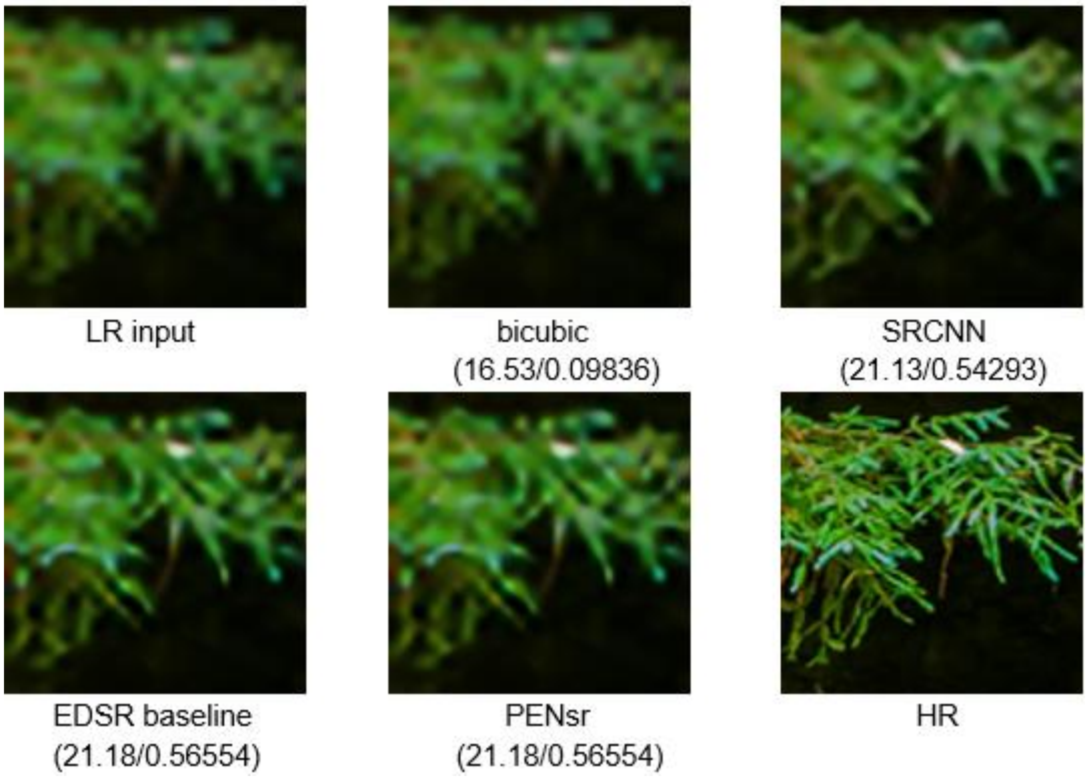


Figure 14: Test Image 2 from DIV2k validation set

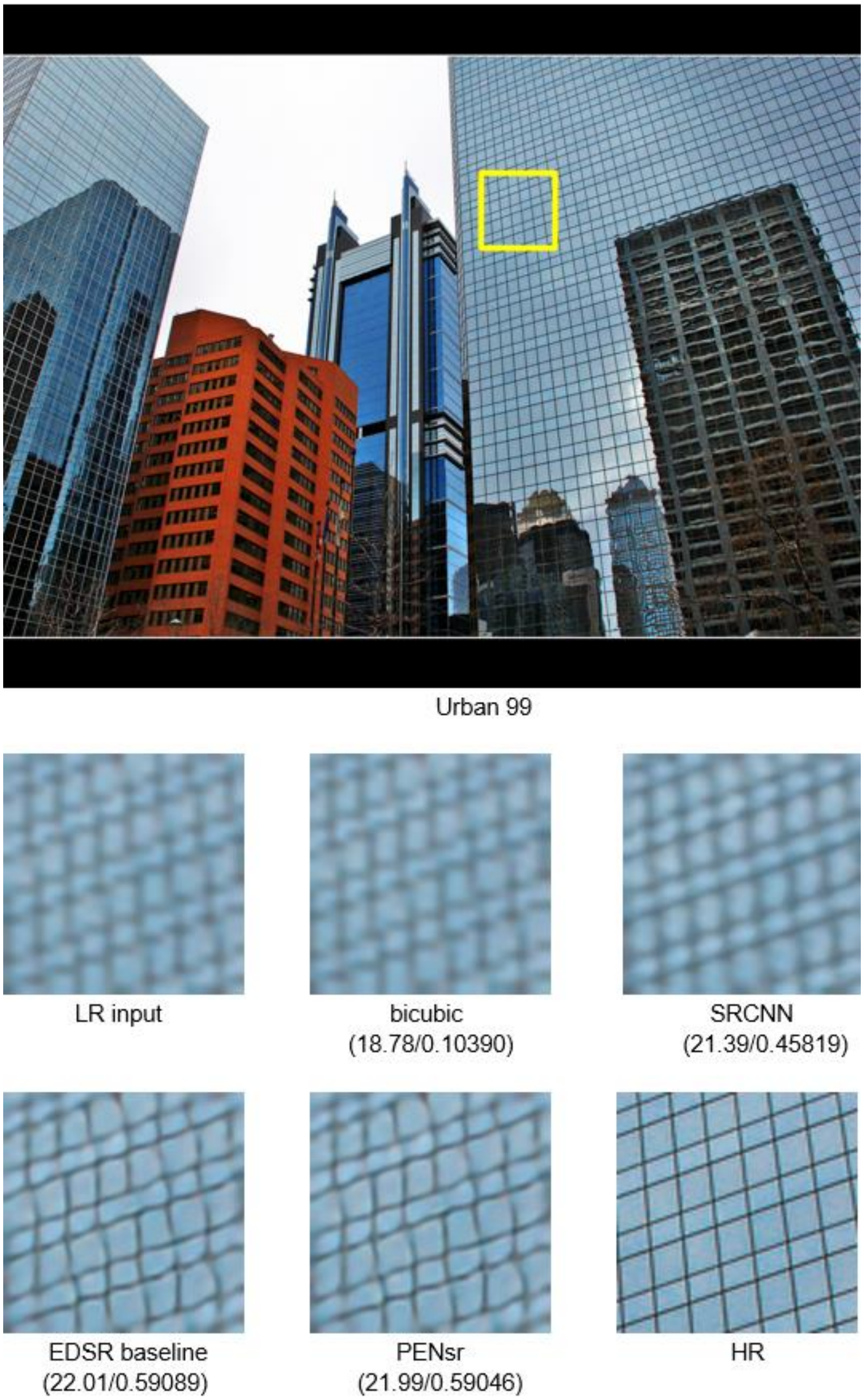


Figure 15: Test Image 3 from DIV2k validation set

Chapter 4

EXPERIMENTS

In this chapter, we will explain our modelling, training and testing methods as well as the results of our experiments.

4.1 Datasets

For our initial experiments, we use a small subset of the popular DIV2K dataset [17] which has 800 training and 200 validations of high-quality 2k resolution images that are used in challenges to train and benchmark super-resolution models.

We used patches of size 48x48 of random 192 images from the training set to test the convergence of the model, then we expanded it to the entire training set after the success of the initial results.

Then we found out that increasing the patch size to 64x64 yielded much better results so now we augmented the DIV2K 800 training images using the same pre-processing method as Wang et al [24] and we ended up with 4000 patches of size 64x64 for our final model. It is worth mentioning that during the pre-processing step we use a transformation where we select the patches as 4 corner patches and a central patch.

For the test datasets, we used the 100 images from the DIV2K validation set to see and compare the performance of our model. We also used some of the standard benchmarking datasets like set5 [18], set14 [19], Urban100 [20] and BSD100 [20].

4.2 Training details

For training, we are using the Low-resolution RGB patch of size 64x64 and their corresponding 128x128 or 256x256 high-resolution version based on the upscaling factor. We also introduce a transformation on the input patch which is a random horizontal flip as a pre-processing step to help with the training, we also subtract the RGB mean of the DIV2K from the images as the last pre-processing step.

For optimization, we use ADAM [21] for the network with Batas range 0.9-0.999 and eps of 1e-08. The learning rate is 1e-4 which is halved after every 2x10^5 mini-batch update with a batch size of 16. The x2 model is trained from scratch and after converging it is used as a pre-trained model to the x4 version.

The calculation of the loss is done with the use of the L1 Loss method which yields better results for this structure as Bee Lim [16] explains it helps with faster conversions if we use L1 instead of L2.

The loss function is what forms the objective function for the process and no other constrains has been added and below are the equations that describes the two types of loss functions and difference between them.

$$L1LossFunction = \sum_{i=1}^{n} |y_{true} - y_{predicted}|$$

$$L2LossFunction = \sum_{i=1}^{n} (y_{true} - y_{predicted})^2$$

Figure 16: Loss Functions

4.3 Development environment

We developed our proposed model using the PyTorch library and using the hugging face super image library [25] to load the datasets and evaluation metrics.

The models have been trained using NVIDIA RTX3060 GPU. It took 2 days to train the x4 model and 18 hours for the x2 model.

4.4 Model evaluation

The model has been tested and evaluated on the DIV2K test set (801 to 900) images using a modified version of the super image evaluation function, a new adjustment has been added to accommodate the compatibility issue of odd dimensions due to the addition of the phasor block.

This adjustment trims the edges of the image to a dimension that is acceptable by the network then the evaluation happens on the trimmed version of the input image. The same thing is done to the other benchmarking sets to produce the results in the following table.

Table 2: Performance comparisons between various models

Dataset	Scale	Bicubic	SRCNN	SRResNet	EDSR baseline	MDSR	EDSR+	PENsr (ours)
Set5	x2	33.66/ 0.9299	36.66/ 0.9542	-/ -	38.11/ 0.9601	38.11/ 0.9602	35.20/ 0.9806	38.17/ 0.9973
	x4	28.42/ 0.8104	30.48/ 0.8628	32.05/ 0.8910	32.46/ 0.8968	32.50/ 0.8973	32.62/ 0.5954	31.26 /0.8841
Setl4	x2	30.24/ 0.8688	32.42/ 0.9063	-/ -	33.92/ 0.9195	33.85/ 0.9198	34.02/ 0.9204	32.83/ 0.9112
	x4	26.00/ 0.7027	27.49/ 0.7503	28.53/ 0.7804	28.80/ 0.7876	28.72/ 0.7857	25.94/ 0.7901	28.19 /0.7514
BSD100	x2	29.56/ 0.8431	31.36/ 0.8879	-/ -	32.32/ 0.9013	32.29/ 0.9007	32.37/ 0.9015	33.28/ 0.9193
	x4	25.96/ 0.6675	26.90/ 0.7101	27.57/ 0.7354	27.71/ 0.7420	27.72/ 0.7418	27.79/ 0.7437	28.16/ 0.7530
Urbanl00	x22	26.88/ 0.8403	29.50/ 0.8946	-/ -	32.93/ 0.9351	32.84/ 0.9347	33.10/ 0.9363	30.00/ 0.9040
	x4	23.14/ 0.6577	24.52/ 0.7221	26.07/ 0.7839	26.64/ 0.8033	26.67/ 0.8041	26.86/ 0.8080	25.13/ 0.7514
DIV2K validation	x 2	31.01 / 0.9393	33.05 / 0.9581	- / -	35.03/ 0.9695	34.96/ 0.9692	35.12/ 0.9699	35.06/ 0.9371
	x 4	26.66 / 0.8521	27.78 / 0.8753	- / -	29.25/ 0.9017	29.26/ 0.9016	29.35 / 0.9032	29.87/ 0.8239

The above table shows the performance comparisons between various models using publicly available results with our obtained values, measured in PSNR (in dB)/SSIM.

Chapter 5

CONCLUSION AND FUTURE WORK

In this paper, we introduced a super-resolution model that uses phasors and residual networks to reconstruct images based on a convolutional architecture which bypasses the locality issue of the convolutional kernel and extracts more meaningful features from the image as well as creating a structure capable of learning the down-sampling process and using it to reconstruct better images.

Our model has achieved improvements on the EDSR baseline model and the same thing can be done on the final EDSR to show better results, given the availability of time and hardware, some possible ways to improve the model is by adding more residual blocks and feature maps.

Our model has been tested on standard benchmarking datasets and the PSNR/SSIM measurement showed an improvement in the performance compared with the original baseline model that has been used.

To further expand this work, there are a few things that will help achieve better results, like expanding the model with more feature maps and/or more residual blocks which will help extract more features and patterns, one more thing that will enhance the richness of the feature space is to add more ways to incorporate non-local features to the pipeline. Lastly, the usage of better hardware like a better GPU or a TPU will help increase the speed of the training process.

REFERENCES

[1] S. Yu, R. Li, R. Zhang, M. An, S. Wu, and Y. Xie, “Performance evaluation of edge-directed interpolation methods for noise-free images,” *Proceedings of the Fifth International Conference on Internet Multimedia Computing and Service - ICIMCS '13*, 2013.

[2] Lei Zhang and Xiaolin Wu, "An edge-guided image interpolation algorithm via directional filtering and data fusion," in *IEEE Transactions on Image Processing*, vol. 15, no. 8, pp. 2226-2238, Aug. 2006, DOI: 10.1109/TIP.2006.877407.

[3] D. Khaledyan, A. Amirany, K. Jafari, M. H. Moaiyeri, A. Z. Khuzani, and N. Mashhadi, “Low-cost implementation of bilinear and bicubic image interpolation for real-time image Super-Resolution,” *2020 IEEE Global Humanitarian Technology Conference (GHTC)*, 2020.

[4] C. Dong, C. C. Loy, K. He, and X. Tang, “Image Super-Resolution Using Deep Convolutional Networks,” 2015.

[5] J. Johnson, A. Alahi, and L. Fei-Fei, “Perceptual losses for real-time style transfer and Super-Resolution,” *Computer Vision – ECCV 2016*, pp. 694–711, 2016.

[6] Jiwon Kim, J. K. Lee, and K. M. Lee, “Accurate Image Super-Resolution Using Very Deep Convolutional Networks,” Nov. 2016.

[7] C. Ledig, L. Theis, F. Huszar, J. Caballero, A. Cunningham, A. Acosta, A. Aitken, A. Tejani, J. Totz, Z. Wang, and W. Shi, “Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network,” May 2017.

[8] K. He, X. Zhang, S. Ren, and J. Sun, “Deep Residual Learning for Image Recognition,” Dec. 2015.

[9] J. Liang, J. Cao, G. Sun, K. Zhang, L. Van Gool, en R. Timofte, “SwinIR: Image Restoration Using Swin Transformer ", *arXiv [eess.IV]*. 2021.

[10] S. Gu, W. Zuo, Q. Xie, D. Meng, X. Feng and L. Zhang, "Convolutional Sparse Coding for Image Super-Resolution," 2015 *IEEE International Conference on Computer Vision (ICCV)*, 2015, pp. 1823-1831, DOI: 10.1109/ICCV.2015.212.

[11] S. Roweis and L. Saul, “Nonlinear Dimensionality Reduction by Locally Linear Embedding,” Nov. 2000.

[12] K. He, X. Zhang, S. Ren, en J. Sun, “Deep Residual Learning for Image Recognition”, *arXiv [cs.CV]*. 2015.

[13] W. Shi *et al.*, “Real-Time Single Image and Video Super-Resolution Using an Efficient Sub-Pixel Convolutional Neural Network”, *arXiv [cs.CV]*. 2016.

[14] D. Chansong and S. Supratid, "Impacts of Kernel Size on Different Resized Images in Object Recognition Based on Convolutional Neural Network," 2021 *9th*

International Electrical Engineering Congress (iEECON), 2021, pp. 448-451, DOI: 10.1109/iEECON51072.2021.9440284.

[15] A. Youssef, "Analysis and comparison of various image down-sampling and up-sampling methods," Proceedings DCC '98 Data Compression Conference (Cat. No.98TB100225), 1998, pp. 583-, DOI: 10.1109/DCC.1998.672325.

[16] B. Lim, S. Son, H. Kim, S. Nah, en K. M. Lee, “Enhanced Deep Residual Networks for Single Image Super-Resolution”, *arXiv [cs.CV]*. 2017.

[17] R. Timofte *et al.*, “NTIRE 2018 Challenge on Single Image Super-Resolution: Methods and Results”, in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, 2018.

[18] M. Bevilacqua, A. Roumy, C. Guillemot, and M.-L. Alberi Morel, “Low-Complexity Single-Image Super-Resolution based on Nonnegative Neighbour Embedding,” 2012.

[19] J.-B. Huang, A. Singh, en N. Ahuja, “Single Image Super-Resolution From Transformed Self-Exemplars”, in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015, bll 5197–5206.

[20] D. Gurari *et al.*, “How to collect segmentations for biomedical images? A benchmark evaluating the performance of experts, crowdsourced non-experts, and algorithms”, in *2015 IEEE Winter Conference on Applications of Computer Vision*, Waikoloa, HI, USA, 2015.

[21] D. P. Kingma en J. Ba, “Adam: A Method for Stochastic Optimization”, *arXiv [cs.LG]*. 2017.

[22] K. Yang, K. Qinami, L. Fei-Fei, J. Deng, en O. Russakovsky, “Towards Fairer Datasets: Filtering and Balancing the Distribution of the People Subtree in the ImageNet Hierarchy”, in *Conference on Fairness, Accountability, and Transparency,* 2020.

[23] A. Krizhevsky, I. Sutskever, en G. E. Hinton, “ImageNet Classification with Deep Convolutional Neural Networks”, in *Advances in Neural Information Processing Systems 25,* F. Pereira, C. J. C. Burges, L. Bottou, en K. Q. Weinberger, Reds Curran Associates, Inc., 2012, bll 1097–1105.

[24] C. Szegedy *et al.*, “Going Deeper with Convolutions”, *arXiv [cs.CV]*. 2014.

[25] F. Wang, H. Hu, en C. Shen, “BAM: A Balanced Attention Mechanism for Single Image Super Resolution”, *arXiv [eess.IV]*. 2021.

[26] Eugene Siow, “super-image,” *hugging face.* 2021.

APPENDIX

Project Code

```
import torch.utils.data as data

import torch

#library to manipulating huge amount of data in numpy format

import h5py

%matplotlib inline

import matplotlib.pyplot as plt

import numpy as np

class DatasetFromHdf5(data.Dataset):

        def __init__(self, file_path):

        super(DatasetFromHdf5, self).__init__()

        hf = h5py.File(file_path)

        self.data = hf.get('data')

        self.target = hf.get('label')

def __getitem__(self, index):

        return torch.from_numpy(self.data[index,:,:,:]).float(),

torch.from_numpy(self.target[index,:,:,:]).float()

def __len__(self):

        return self.data.shape[0]

from torchsummary import summary

import torch

import torch.nn as nn

import math

class MeanShift(Conv2d):
        def __init__(self, rgb_mean, sign):
        super(MeanShift, self).__init__(3, 3, kernel_size=1)
        self.weight.data = torch.eye(3).view(3, 3, 1, 1) # W H C N
        self.bias.data = float(sign) * torch.Tensor(rgb_mean)
        # Freeze the MeanShift layer
        for params in self.parameters():
        params.requires_grad = False
class phase(nn.Module):
        def phasing_filters (self,kernal_size):
        phases = kernal_size * kernal_size
        tensor = torch.zeros((phases,kernal_size,kernal_size), dtype=torch.float,
device = 'cuda')
        x = 0
        while (x<phases):
        for y in range(kernal_size):
                for z in range(kernal_size):
                tensor[x][y][z] = 1.;
                x = x+1
        tensor = tensor[None,:,:,:]
        tensor = tensor.permute(1,0,2,3)
        return tensor
def __init__(self, size):
        self.phases = size*size
        weight = self.phasing_filters(size)
        super(phase, self).__init__()
```

```
self.conv = Conv2d(1, 1,self.phases, (2, 2),False , (2,2))
self.conv.weight = torch.nn.Parameter(weight)
self.conv.weight.to(device='cuda', dtype=torch.float)
# Freeze the MeanShift layer
for params in self.parameters():
params.requires_grad = False
def forward(self, x):
final = []
for i in range(len(x[:])):
x1 = x[None,i,0,:,:]
x2 = x[None,i,1,:,:]
x3 = x[None,i,2,:,:]
x1 = torch.unsqueeze(x1, dim=0).to('cuda')
x2 = torch.unsqueeze(x2, dim=0).to('cuda')
x3 = torch.unsqueeze(x3, dim=0).to('cuda')
conv_x1 = self.conv(x1)
conv_x2 = self.conv(x2)
conv_x3 = self.conv(x3)
outputs = []
for c in range(self.phases):
        F =
torch.cat((conv_x1[None,:,c,:,:],conv_x2[None,:,c,:,:],conv_x3[None,:,c,:,:]),0)
        outputs.append(F)
        results = torch.cat(outputs, dim=1)
        results = results.permute(1,0,2,3)
```

```
                results = results[None,:,:,:,:]
        final.append(results)
        final = torch.cat(final, dim=0)
        final = final.permute(1,0,2,3,4)
        return final

class _Residual_Block(nn.Module):
        def __init__(self):
        super(_Residual_Block, self).__init__()

        self.conv1 = Conv2d(256, 256, 3, stride=1, 1,False)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = Conv2d(256, 256, 3, 1, 1, False)

        def forward(self, x):
        identity_data = x
        output = self.relu(self.conv1(x))
        output = self.conv2(output)
        output *= 0.1
        output = torch.add(output,identity_data)
        return output

class _Residual_Block_phase(nn.Module):
        def __init__(self):
        super(_Residual_Block_phase, self).__init__()
        self.conv1 = Conv2d(64, 64, 3,1, 1, False)

        self.relu = nn.ReLU(inplace=True)

        self.conv2 = Conv2d(64, 64, 3, 1, 1, False)


        def forward(self, x):

        identity_data = x

        output = self.relu(self.conv1(x))

        output = self.conv2(output)

        output *= 0.1

        output = torch.add(output,identity_data)

        return output


class Net(nn.Module):

        def __init__(self):

        super(Net, self).__init__()


        rgb_mean = (0.4488, 0.4371, 0.4040)

        self.sub_mean = MeanShift(rgb_mean, -1)

        self.phases = phase(2)

        #input conv

        self.conv_input = Conv2d(3,256, 3, 1, 1,False)

        self.input_phases = Conv2d(3,64, 3, 1, 1,False)

        #residual block

        self.residual = self.make_layer(_Residual_Block, 32)
        self.phase_residual = self.make_layer(_Residual_Block_phase, 8)


        #mid convolution

        self.conv_mid = Conv2d( 256,  256, 3, 1, 1, False)

        self.conv_mid_phases = Conv2d( 64, 64,  3, 1, 1, False)


        self.upscale4x = nn.Sequential(

        Conv2d(256, 256*4, 3, 1, 1, False),

        nn.PixelShuffle(2),

        nn.Conv2d(256, 256*4, 3, 1, 1, False),

        nn.PixelShuffle(2),)

        self.upscale4x_phases = nn.Sequential(

        Conv2d(64, 64*4,3, 1, 1, bias=False),

        nn.PixelShuffle(2),

        Conv2d(64, 64*4, 3, 1, 1, False),

        nn.PixelShuffle(2),

        Conv2d( 64,  64*4, 3,1, 1, False),

        nn.PixelShuffle(2),)


        self.conv_output = Conv2d(256*2, 3, 3, 1, 1,False)


        self.add_mean = MeanShift(rgb_mean, 1)


        for m in self.modules():

        if isinstance(m, nn.Conv2d):
```

```
          n = m.kernel_size[0] * m.kernel_size[1] * m.out_channels
          m.weight.data.normal_(0, math.sqrt(2. / n))
          if m.bias is not None:
          m.bias.data.zero_()
elif isinstance(m, nn.BatchNorm2d):
          m.weight.data.fill_(1)
          if m.bias is not None:
          m.bias.data.zero_()


def make_layer(self, block, num_of_layer):
layers = []
for _ in range(num_of_layer):
layers.append(block())
return nn.Sequential(*layers)


def forward(self, x):
out = self.sub_mean(x)
phases_list = []
for p in range (self.phases(out).shape[0]):
phases_list.append(self.phases(out)[p])
#p1  = self.phases(out)[0]
#p2  = self.phases(out)[1]
#p3  = self.phases(out)[2]
#p4  = self.phases(out)[3]
```

```
out = self.conv_input(out)
for p in enumerate(phases_list):
phases_list[p] = self.input_phases(p)
#p1  = self.input_phases(p1)
#p2  = self.input_phases(p2)
#p3  = self.input_phases(p3)
#p4  = self.input_phases(p4)
residual = out
rp_list = []
for rp in phases_list:
rp_list.append(rp)
#rp1 = p1
#rp2 = p2
#rp3 = p3
#rp4 = p4

out = self.conv_mid(self.residual(out))
for p in phases_list:
phases_list[p] = self.conv_mid_phases(self.phase_residual(p))


#p1  = self.conv_mid_phases(self.phase_residual(p1))
#p2  = self.conv_mid_phases(self.phase_residual(p2))
#p3  = self.conv_mid_phases(self.phase_residual(p3))
#p4  = self.conv_mid_phases(self.phase_residual(p4))
```

```
        out = torch.add(out,residual)


        for p in phases_list:

        phases_list[p] = torch.add(p,rp_list[p])


        #p1  = torch.add(p1,rp1)

        #p2  = torch.add(p2,rp2)

        #p3  = torch.add(p3,rp3)

        #p4  = torch.add(p4,rp4)


        out = self.upscale4x(out)

        for p in phases_list:

        phases_list[p] = self.upscale4x_phases(p)

        #p1  = self.upscale4x_phases(p1)

        #p2  = self.upscale4x_phases(p2)

        #p3  = self.upscale4x_phases(p3)

        #p4  = self.upscale4x_phases(p4)

        for p in phases_list:

        final = torch.cat((out,p),dim = 1)

        final = self.conv_output(final)

        out = self.add_mean(final)




        return out

model = Net()
```

```
model.cuda()

summary(model, input_size=(3, 48, 48))

import argparse, os

import torch

import math, random

import torch.backends.cudnn as cudnn

import torch.nn as nn

import torch.optim as optim

from torch.autograd import Variable

from torch.utils.data import DataLoader

torch.cuda.empty_cache()



# Training settings

batchSize       = 16            #training batch size

nEpochs         = 960           #number of epochs to train for

tlr     = 1e-4          #Learning Rate. Default=1e-4

step    = 200           #Sets the learning rate to the initial LR decayed by momentum
every n epochs, Default: n=10

cuda            = True          #use cuda

start_epoch  = 1        #manual epoch number (useful on restarts)

threads         = 4             #number of threads for data loader to use

momentum        = 0.9           #momentum

tweight_decay = float(1e-4)  #weight decay, Default: 0
```

```
global model


#check if you can use the gpu
if cuda and not torch.cuda.is_available():
        raise Exception("No GPU found")


#preaparing a seed to randomly inisilize wieghts
seed = random.randint(1, 10000)
print("Random Seed: ", seed)
torch.manual_seed(seed)
if cuda:
        cudnn.benchmark = True


print("===> Loading datasets")
train_set = DatasetFromHdf5("data/edsr_x4.h5")
training_data_loader = DataLoader(dataset=train_set, num_workers=threads,
batch_size= batchSize, shuffle=True)
#print(training_data_loader.shape
print("===> Building model")
model = Net()
criterion = nn.L1Loss(size_average=False)
print("===> Setting GPU")
if cuda:
        torch.cuda.empty_cache()
        model = model.cuda()
```

```
	criterion = criterion.cuda()



checkpoint =
torch.load('/content/drive/MyDrive/EDSR/checkpoint/model_epoch_135.pth')
start_epoch = checkpoint["epoch"] + 1
model.load_state_dict(checkpoint["model"].state_dict())

# optionally resume from a checkpoint
#if opt.resume:
#	if os.path.isfile(opt.resume):
#	print("=> loading checkpoint '{}'".format(opt.resume))
#	checkpoint = torch.load(opt.resume)
#	opt.start_epoch = checkpoint["epoch"] + 1
#	model.load_state_dict(checkpoint["model"].state_dict())
#	else:
#	print("=> no checkpoint found at '{}'".format(opt.resume))

def adjust_learning_rate(optimizer, epoch):
	"""Sets the learning rate to the initial LR decayed by 10"""
	lr = tlr * (0.1 ** (epoch // step))
	return lr

def train(training_data_loader, optimizer, model, criterion, epoch):
	lr = adjust_learning_rate(optimizer, epoch-1)
	for param_group in optimizer.param_groups:
	param_group["lr"] = lr


	print("Epoch={}, lr={}".format(epoch, optimizer.param_groups[0]["lr"]))
	model.train()
	running_loss = 0
	for iteration, batch in enumerate(training_data_loader, 1):
	input, target = Variable(batch[0]), Variable(batch[1], requires_grad=False)
	if cuda:
	input = input.cuda()
	target = target.cuda()
	torch.cuda.empty_cache()
	loss = criterion(model(input), target)
	running_loss =+ loss.item() *input.size(0)
	optimizer.zero_grad()
	loss.backward()
	optimizer.step()
	torch.cuda.empty_cache()
	if iteration%116 == 0:
	print("===> Epoch[{}]({}/{}): Loss: {:.5f}".format(epoch, iteration,
len(training_data_loader), loss.item()))
	loss_value = (running_loss /len(training_data_loader))
	return (loss_value)


def save_checkpoint(model, epoch):
        model_folder = "checkpoint/"
        model_out_path = model_folder + "model_epoch_{}.pth".format(epoch)
        state = {"epoch": epoch ,"model": model}
        if not os.path.exists(model_folder):
        os.makedirs(model_folder)
        torch.save(state, model_out_path)
        print("Checkpoint saved to {}".format(model_out_path))
loss_log =  open("log.txt","w+")
print("===> Setting Optimizer")
optimizer = optim.Adam(filter(lambda p: p.requires_grad, model.parameters()), lr=
tlr, weight_decay = tweight_decay , betas = (0.9, 0.999), eps=1e-08)
print("===> Training")
losses = []
for epoch in range(start_epoch,nEpochs + 1):
        torch.cuda.empty_cache()
        eloss = train(training_data_loader, optimizer, model, criterion, epoch)
        losses.append(eloss)
        print(eloss)
        loss_log.write("{}\r\n".format(eloss))
        save_checkpoint(model, epoch)
loss_log.close()
import matlab.engine
import argparse
import torch
from torch.autograd import Variable
```

```
import numpy as np

import time, math, glob

import scipy.io as sio

import cv2


# evalution parameters

cuda    = True

tmodel  = "checkpoint/model_edsr.pth"

dataset = "Set5"

scalet  = 4


def PSNR(pred, gt, shave_border=0):

        height, width = pred.shape[:2]

        pred = pred[shave_border:height - shave_border, shave_border:width -
shave_border]

        gt = gt[shave_border:height - shave_border, shave_border:width -
shave_border]

        imdff = pred - gt

        rmse = math.sqrt(np.mean(imdff ** 2))

        if rmse == 0:

        return 100

        return 20 * math.log10(255.0 / rmse)

eng = matlab.engine.start_matlab()


if cuda and not torch.cuda.is_available():
        raise Exception("No GPU found, please run without --cuda")

model = torch.load(tmodel)["model"]

image_list = glob.glob(dataset+"/*.*")

avg_psnr_predicted = 0.0
avg_psnr_bicubic = 0.0
avg_elapsed_time = 0.0

for image_name in image_list:
        print("Processing ", image_name)
        im_gt_y = sio.loadmat(image_name)['im_gt_y']
        im_b_y = sio.loadmat(image_name)['im_b_y']
        im_l = sio.loadmat(image_name)['im_l']

        im_gt_y = im_gt_y.astype(float)
        im_b_y = im_b_y.astype(float)
        im_l = im_l.astype(float)

        psnr_bicubic = PSNR(im_gt_y, im_b_y,shave_border=opt.scale)
        avg_psnr_bicubic += psnr_bicubic

        im_input = im_l.astype(np.float32).transpose(2,0,1)
        im_input =
im_input.reshape(1,im_input.shape[0],im_input.shape[1],im_input.shape[2])
        im_input = Variable(torch.from_numpy(im_input/255.).float())


        if cuda:
        model = model.cuda()
        im_input = im_input.cuda()
        else:
        model = model.cpu()


        start_time = time.time()
        HR_4x = model(im_input)
        elapsed_time = time.time() - start_time
        avg_elapsed_time += elapsed_time


        HR_4x = HR_4x.cpu()


        im_h = HR_4x.data[0].numpy().astype(np.float32)


        im_h = im_h*255.
        im_h = np.clip(im_h, 0., 255.)
        im_h = im_h.transpose(1,2,0).astype(np.float32)


        im_h_matlab = matlab.double((im_h / 255.).tolist())
        im_h_ycbcr = eng.rgb2ycbcr(im_h_matlab)
        im_h_ycbcr = np.array(im_h_ycbcr._data).reshape(im_h_ycbcr.size,
order='F').astype(np.float32) * 255.
        im_h_y = im_h_ycbcr[:,:,0]


        psnr_predicted = PSNR(im_gt_y, im_h_y,shave_border=opt.scale)
        avg_psnr_predicted += psnr_predicted


print("Scale=", scale)
print("Dataset=", dataset)
print("PSNR_predicted=", avg_psnr_predicted/len(image_list))
print("PSNR_bicubic=", avg_psnr_bicubic/len(image_list))
print("It takes average {}s for
processing".format(avg_elapsed_time/len(image_list)))
```