# MULTI-FRAME SUPER-RESOLUTION WITHOUT PRIORS

**A Thesis Submitted to**
**the Graduate School of Engineering and Sciences of**
**İzmir Institute of Technology**
**in Partial Fulfillment of the Requirements for the Degree of**

**MASTER OF SCIENCE**

**in Computer Engineering**

**by**
**Veli GÜLMEZ**

**July 2023**
**İZMİR**

We approve the thesis of **Veli GÜLMEZ**

**Examining Committee Members:**

---

**Assoc. Prof. Dr. Mustafa ÖZUYSAL**

Department of Computer Engineering, İzmir Institute of Technology

---

**Prof. Dr. Yalın BAŞTANLAR**

Department of Computer Engineering, İzmir Institute of Technology

---

**Prof. Dr. Devrim ÜNAY**

Department of Electrical and Electronics Engineering, İzmir Democracy University

**18 July 2023**

---

**Assoc. Prof. Dr. Mustafa ÖZUYSAL**

Supervisor, Department of Computer Engineering

İzmir Institute of Technology

---

**Prof. Dr. Cüneyt F. BAZLAMAÇCI**

Head of the Department of

Computer Engineering

---

**Prof. Dr. Mehtap EANES**

Dean of the Graduate School of

Engineering and Sciences

# ACKNOWLEDGMENTS

I would like to thank my supervisor, Assoc. Prof. Dr. Mustafa Özuysal, for the valuable feedback and encouragement during the thesis. I also am grateful for helping me improve myself in foreseeing further possible developments.

I would like to acknowledge Ersin Çine for introducing me to how to do research at the beginning of my undergraduate studies.

I would like to thank my mom and my brother for all the endless support and delicious dishes.

Finally, I would like to thank my girlfriend, Ceren, for her support, love, and patience.

# ABSTRACT

MULTI-FRAME SUPER-RESOLUTION WITHOUT PRIORS

There are mainly two types of super-resolution methods: traditional methods and deep learning methods. While traditional methods define closed-form expressions with assumptions, deep learning methods rely on priors learned from data sets. However, both of them have disadvantages such as being too simple and having strong trust in priors. We focus on how to generate a high-resolution image using low-resolution images without priors by utilizing spatial hash encoding. We propose a grid-based super-resolution model using spatial hash encoding to map coordinate information into higher dimensional space. Our aim is to eliminate long training times and not rely on priors from data sets that are not able to cover all real-world scenarios. Therefore, our proposed model is able to do task-specific super-resolution without priors and eliminate potential hallucination effects caused by wrong priors.

# ÖZET

## ÖNSEL BİLGİSİZ ÇOKLU GÖRÜNTÜDEN SÜPER ÇÖZÜNÜRLÜK

Ağırlıklı olarak iki tür süper çözünürlük yöntemi vardır: geleneksel yöntemler ve derin öğrenme yöntemleri. Geleneksel yöntemler varsayımlarla kapalı biçimde ifadeler tanımlarken, derin öğrenme yöntemleri veri kümelerinden öğrenilen önsel bilgilere dayanır. Ancak her ikisinin de çok basit olması ve önsel bilgiye güvenin kuvvetli olması gibi dezavantajları vardır. Uzamsal özet kodlamayı kullanarak önsel bilgiler olmadan düşük çözünürlüklü görüntüler kullanarak yüksek çözünürlüklü bir görüntünün nasıl üretileceğine odaklanıyoruz. Koordinat bilgilerini daha yüksek boyutlu uzaya eşlemek için uzamsal özet kodlamayı kullanan ızgara tabanlı bir süper çözünürlüklü model öneriyoruz. Amacımız, uzun eğitim sürelerini ortadan kaldırmak ve tüm gerçek dünya senaryolarını kapsayamayan veri setlerinden elde edilen verilere güvenmemektir. Bu nedenle, önerdiğimiz model, önsel bilgiler olmadan göreve özel süper çözünürlük yapabilir ve yanlış önceliklerin neden olduğu potansiyel halüsinasyon etkilerini ortadan kaldırabilir.

# TABLE OF CONTENTS

# LIST OF FIGURES

**Figure** **Page**

# LIST OF TABLES

**Table** **Page**

# CHAPTER 1

# INTRODUCTION

Due to the success of artificial intelligence, the demand for generating high-quality images is increasing significantly. Many applications use state-of-the-art methods to obtain better-quality images such as astronomical images,[6] medical images,[12] and mobile phones.[40] These methods employ a combination of denoising and super-resolution approaches which considers only reducing noises and produces an image with high frequency and high spatial resolution using low-resolution image or images, respectively. Therefore, we can obtain images with less noise and high resolution by applying both methods.

(a) Original (b) Alising (c) Blurring (d) Noise

Figure 1.1. Aliasing, blurring, and noise are the main challenges of super-resolution.

However, we have a lack of information to generate a high-resolution image due to hardware and software limitations.[25,34] These limitations are caused by the distance between the object and the camera, and the components of the camera such as aperture, ISO value, and shutter speed value. They affect the amount and the quality of light entering the camera. Hence, we cannot obtain a better high-resolution image.[25] point outs three problems leading to challenges in super-resolution: blurring, aliasing, and noise as shown in Figure 1.1. Blurring is caused by motion blur, optical blur, and lens blur while light noise, hardware specification noise, and quantization noise lead to noise in the image. The final factor is that aliasing loses some signals and occurs when components of a signal are above the Nyquist frequency.

There are two well-known approaches dealing with the denoising and super-

resolution problem: single image super-resolution (SISR) and multi-frame super-resolution (MFSR). SISR only works with low-resolution and high-resolution image pairs while MFSR utilizes multiple low-resolution images to generate a high-resolution image with priors or without priors.

In this thesis, we focus on how to generate a high-resolution image using low-resolution images that come from the same scene with subpixel shifts by utilizing task-agnostic architecture. Therefore, we get rid of long training times and do not rely on learnable priors from data sets that are not able to cover all real-world scenarios.

## 1.1. Related Work

In this section, we briefly discuss proposed methods to obtain a high-resolution image from low-resolution image or images. There are mainly used two approaches: single image super-resolution (SISR) and multi-frame super-resolution (MFSR). We review the proposed methods under traditional image processing and deep learning. Finally, we see two evaluation metrics, peak signal-to-noise ratio (PSNR) and structural similarity index measure (SSIM), to calculate errors between a generated high-resolution image and a ground truth image.

### 1.1.1. Single Image Super-resolution (SISR)

In SISR, we use only low-resolution image or both low-resolution and high-resolution image pairs to obtain a high-resolution image. It is very practical and has less power consumption compared to multiple ones. Most of the research is on this subject.

The first one employs traditional image processing methods.[7,14,22,30,33,37] These methods have very limited information to solve the problem but they are practical and easy to implement. However, their results are not satisfying. Therefore, they can be used in real-time applications with average accurate high-resolution images.

In recent years, the deep learning method becomes very popular due to its success and practicality. It trains a large data set that involves low-resolution and high-resolution

image pairs. Hence, it obtains prior information and with this, it generates a high-resolution image using only a low-resolution image. It achieves better scores than the traditional way.[9,17,20,23,31,32,39,41] In spite of the high scores, it leads to some problems like hallucinations as shown in Figure 1.2. It may create unidentified effects while generating a high-resolution image.[34] Because it strongly relies on priors and data sets used in training that do not cover all real-world scenarios..[1]

(a) Reference (b) SR3[32] (c) StableDiffusion[31]

Figure 1.2. Visualization of hallucination effects, such as leaves and flowers, can be observed in parts (b) and (c), which are not seen in the reference image shown in part (a).

### 1.1.2. Multi-frame Super-resolution (MFSR)

Thanks to advances in camera and hardware technology of mobile phones, we take multiple shots with our cameras. These serial shots are taken in burst mode. We easily benefit from additional data coming from different conditions such as light, motion, and hardware.

In contrast to SISR, MFSR is more general due to exploiting more than one low-resolution images. Thus, the dependence on priors reduces compared to a single one.[29] Although a decrease in undesirable effects with multiple images, we need to figure out emerging problems such as motion and fusion among low-resolution images.

In the traditional way, there are various proposed methods under different algorithms: frequency-domain,[36] kernel-based,[40] reconstruction-based,[15] and maximum a posteriori (MAP)[11] methods. These methods are more robust compared to the single one

despite simple assumptions, i.e., the distribution of noise on the image is the Gaussian noise. However, they also are not able to avoid sub-optimal solutions and hallucination effects.

With the ability to work with huge neural networks, multiple low-resolution images are studied to figure out the super-resolution problem. Since 2012, when deep learning techniques such as AlexNet[19] gained popularity, there has been a surge in research on super-resolution, with many works proposed in different subfields of deep learning.[2,3,5,8,10,13,16] These works give better results, but they try to solve extra problems like translation. For that reason, their training time is too long and is not suitable for real-time applications. Unfortunately, they also cannot prevent undesirable effects, in particular, generative models are not stable while generating high-resolution images.

## 1.2. Motivation

Exploiting priors brings about some undesirable effects like hallucinations while constructing high spatial resolution. Besides that, it suffers from narrow data sets because it cannot cover all cases in real-world applications. We can acquire more meaningful than the single-image approach thanks to the multi-frame approach. It takes advantage of extra useful data from low-resolution frames. Unfortunately, MFSR with priors is not able to diminish prior effects due to learnable parameters using low-resolution images and high-resolution image pairs. If we would like to get high-resolution images without hallucination effects, we take advantage of task-agnostic super-resolution which incorporates only low-resolution images coming from the same scene into the procedure of generating the high-resolution image. Therefore, we can get rid of undesirable effects and training with large data sets.

This work[28] proposes an architecture that learns an image using multiresolution hash encoding and simple multilayer perceptrons (MLPs). It extracts embeddings at different resolutions with 2D coordinate-based information and a learnable hash mechanism. Thus, it provides independence in the task and reduces dependence on the data sets. Also, due to simple architecture, it is easy to implement parallelism. In conclusion, we leverage task agnosticism and parallelism.

We can take advantage of hash encoding architecture to solve super-resolution

problem. In contrast to this architecture, we learn a high spatial resolution image using low-resolution images as ground truths. Our task-agnostic structure utilizes the hash encoding mechanism, which eliminates the need for prior knowledge. Additionally, unlike traditional super-resolution approaches that are stuck with local minima due to their trivial assumptions, our method uses hash encoding and a simple MLP to overcome these limitations.

## 1.3. Thesis Goals and Contributions

The objective of this thesis is to develop a model that generates high-resolution images from low-resolution images, without relying on priors. To achieve this, we utilize a task-agnostic architecture that takes low-resolution images as ground truths. Furthermore, thanks to the task-agnostic architecture, we do not need to make simple assumptions about the distribution of noise, such as assuming it follows a Gaussian distribution.

Our main contributions in this thesis can be summarized as follows:

- design a task-agnostic super-resolution model that can be used in different areas such as astronomical, medical, and mobile phones,
- accelerate the super-resolution model using the Numba framework,
- remove the need for large data sets using only employing low-resolution images from the same scene to generate high-resolution images,
- dispose of any potential hallucination effects due to the prior-free architecture,
- avoid simple assumptions by integrating the task-agnostic super-resolution model implicitly.

## 1.4. Organization of the Thesis

The organization of the thesis is as follows:

- Chapter 2 presents the background of multiresolution hash encoding architecture[28] and the Python implementation of this architecture,

• Chapter 3 proposes a grid-based super-resolution method using a spatial hash encoding that generates a high-resolution image using low-resolution images without prior information,

• Chapter 4 starts with a research summary and contributions and finally ends up with a comparison of obtained results. Then, we mention future works in the last section.

# CHAPTER 2

# PYTHON IMPLEMENTATION OF MULTIRESOLUTION HASH ENCODING

In this chapter, we present the background of multiresolution hash encoding architecture[28] and the Python implementation of this architecture. In Chapter 3, we use this architecture to generate high-resolution images using low-resolution images as ground truths. Section 2.1 presents why we need to parameterize neural graphics primitives with trainable encodings by fully connected neural networks, as well as demonstrates how to provide a task-independent structure exploiting the hash mechanism. Section 2.2 presents the mathematical background and components of the multiresolution hash encoding architecture and the analysis of configurable parameters in detail. On the other hand, Section 2.3 focuses on the Python implementation of the architecture, rather than the original project with CUDA-based implementation due to the slow compilation time, and includes examples of how the model can be used for learning a single image.

## 2.1. Introduction

This section will focus on two key topics related to computer graphics primitives. Firstly, it will explore how to use multi-layer perceptions (MLPs) to parameterize the mathematical functions of these primitives. Secondly, it will investigate the topic of adaptive, effective, and task-agnostic mapping of trainable encodings to higher-dimensional space through the use of multiresolution hash encoding.

### 2.1.1. Neural Graphics Primitives

Computer graphics primitives, represented as mathematical functions, are basic geometric objects to construct more complex graphical images. These primitives are parameterized by neural networks like NeRF[27] that represents radiance fields as neural

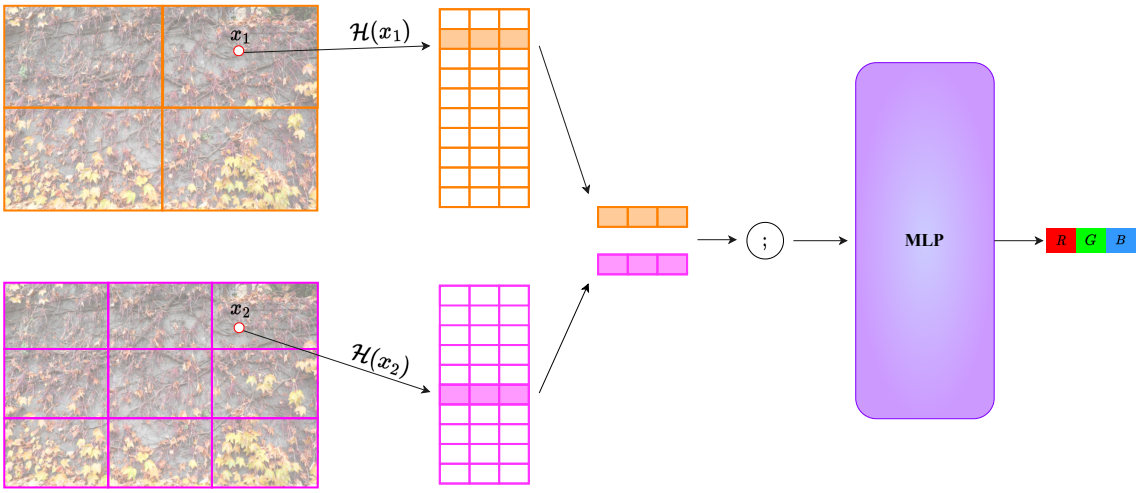



Figure 2.1. An overview of multiresolution hash encoding. It gets 2D coordinate x, concatenates encodings after hash encoding, and then gives the concatenated encodings to MLP as inputs.

graphic primitives for view synthesis. Therefore, we improve quality and acceleration of graphical construction using this implicit representation.

### 2.1.2. Aim of Multiresolution Hash Encoding

Multiresolution hash encoding proposes an architecture to represent graphics primitives utilizing a sparse parametric encoding approach through sparse hashing as shown in Figure 2.1. Thanks to the hash mechanism, the architecture provides three important features. The first one is adaptivity, we modify the architecture by configuring hashing parameters. The other one is efficiency. We get rid of control flows and therefore we implement operations of hash tables in parallel through the use of hashing data structure. The last is the independence of the task. Without prior knowledge, hash tables are able to prioritize the feature vectors according to their importance.

### 2.1.3. Types of Encoding

In the previous subsection, we see the advantages of encoding in terms of convergence and performance. Here, we focus on the other encoding types and the difference between them and the proposed multiresolution hash encoding.

The reason for the use of encoding, we achieve high-frequency details while mapping inputs to higher dimensional space by utilizing high-frequency functions. We review three kinds of encodings: frequency, parametric, and sparse parametric encodings as shown in Figure 2.2.

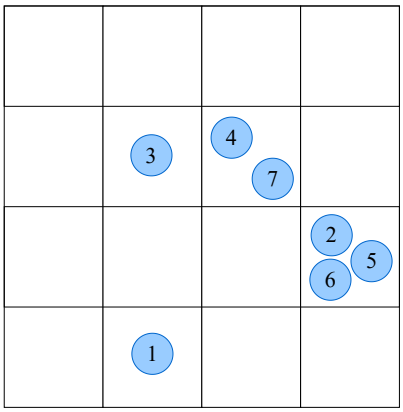
To encode inputs, frequency encoding uses functions that repeat themselves at a certain rate like waveform functions. Transformers[38] show the impact of this encoding to define them as positions of tokens in a sequence through sine and cosine encoding functions:

$$\mathcal{E}(x) = (sin(2^0x), sin(2^1x), ..., sin(2^{L-1}x), cos(2^0x), cos(2^1x), ..., cos(2^{L-1}x)), \quad (2.1)$$

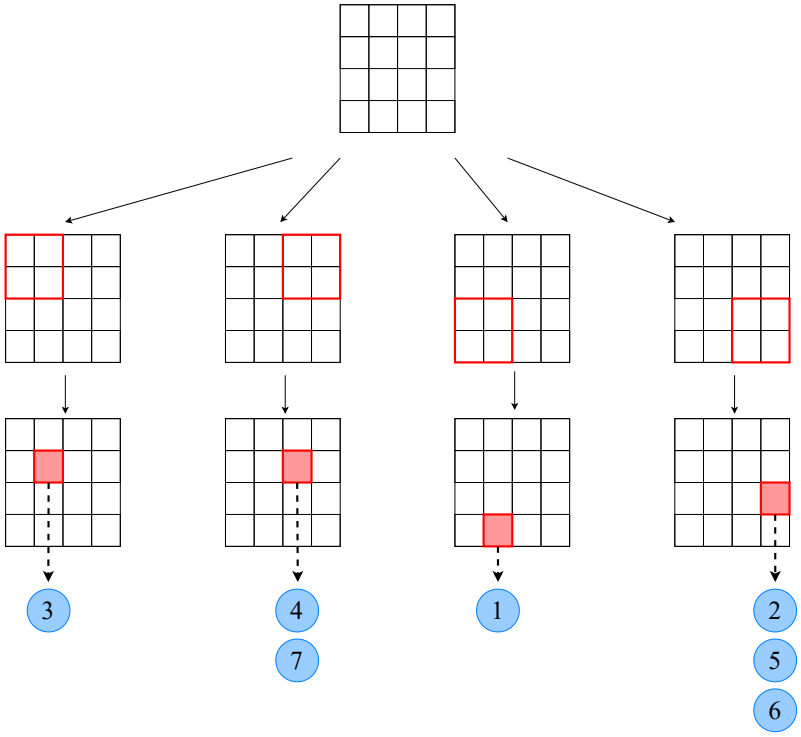
where $x \in \mathbb{R}$ is the scalar positions and $L \in \mathbb{N}$ is the multiresolution sequence. NeRF[27] exploits this function by mapping five-dimensional coordinate into higher dimensional space to generate high-frequency scene details.

Another type of encoding is parametric encoding. It utilizes grid-based and tree-based approaches to represent inputs as parametric encodings, which are fixed trainable encodings. These structures decrease convergence time significantly as well as have less computational time than large neural networks. However, they suffer from vast memory allocation for storing features of data structures. A work[26] of the parametric encoding maps inputs with a coordinate-based encoder into higher dimensional space. In spite of achieving better performance than previous parametric encodings, it needs to calculate more operations.

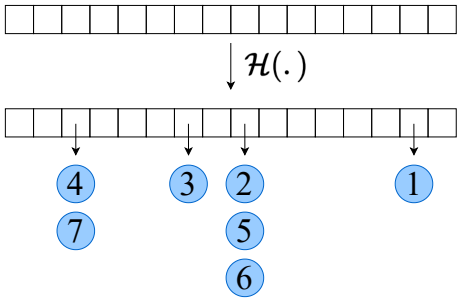
The final encoding type is sparse parametric encoding, also multiresolution hash encoding belongs to this type of parametric encoding. We see that parametric encoding approaches obtain better scores than frequency approaches. Unfortunately, they need to cope with huge memory consumption and flexibility. Multiresolution hash encoding figures out these problems using a spatial hash table, which is controlled by adding configurable hyperparameters, T and F, hash table size, and length of feature vectors respectively. In addition, we do not extract features from images instead we generate our defined features and update them thanks to the task-agnostic structure of the spatial hashing.




(a) Samples




(b) Quadtree




(c) Spatial hashing

Figure 2.2. Examples of quadtree and spatial hashing.

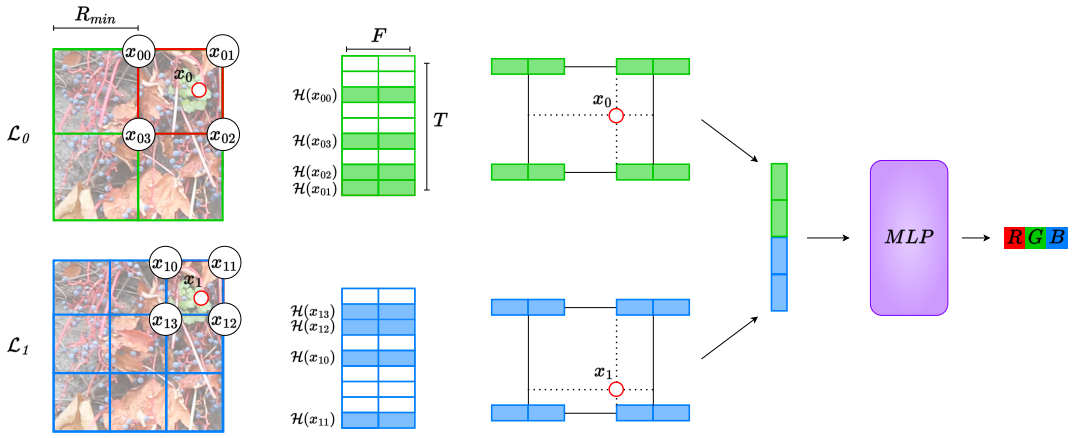



Figure 2.3. The architecture design of multiresolution hash encoding.[28]

## 2.2. Methodology

In this section, we cover the design and components of the architecture. As discussed before in Section 2.1.2, the aim of multiresolution hash encoding is to map 2D coordinate-based information into 3D RGB color space to learn a single image. There are mainly two steps in the learning phase: encoding of 2D coordinate inputs and neural networks. In the encoding phase, spatial hashing is used to encode 2D coordinate inputs at different resolutions using a grid-based approach. Latter, neural networks use extracted encodings as inputs to generate an RGB value of a given coordinate.

We define the hyperparameters of the architecture: T, the size of the hash table; F, the length of the feature vectors; and L, the number of levels. The hash table size T controls memory consumption and quality of the image as well as affects the performance of the architecture as shown in Figure 2.5. The number of resolutions for the grid structure is configured by L. L is also related to minimum resolution (coarsest details), maximum resolution (finest details) parameters, and a proportion between consecutive levels:

$$p^{L-1} = \frac{R_{max}}{R_{min}}, \tag{2.2}$$

where $p$ is the proportion, $R_{max}$ is the finest resolution and $R_{min}$ is the coarsest resolution. The last hyperparameter F is the length of the feature vectors. L and F are also related to quality and performance. The effects of hyperparameters are discussed in Section 2.3.[28] founds that the optimum values of T, L, and F, which are $2^{19}$, 16, and 2 respectively.

In this part, we focus on the details of the architecture in Figure 2.3. In the

beginning, we calculate the corresponding indices of the given 2D coordinate **x** for each level. Thus, we map 2D integer coordinates for each level into higher dimensional space through a spatial hash function[35] with the given formula:

$$\mathcal{H}(\mathbf{x}) = x_i \bigoplus \pi y_i \mod \mathrm{T}, \tag{2.3}$$

where $\pi = 2654435761$ and $\oplus$ is the bit-wise XOR operation. If the size of the hash table is higher than the grid parameters of the current level, one-to-one mappings are applied without collision. However, at the finer levels, the hash function is used with no need to handle collision. Because it explicitly deals with collision handling by neural network optimization.

We get feature vectors using indices of the hash tables with the given 2D coordinate. The reason for the bilinear interpolation of feature vectors is to make them continuous to prevent undesirable visual effects as shown in Figure 2.4.

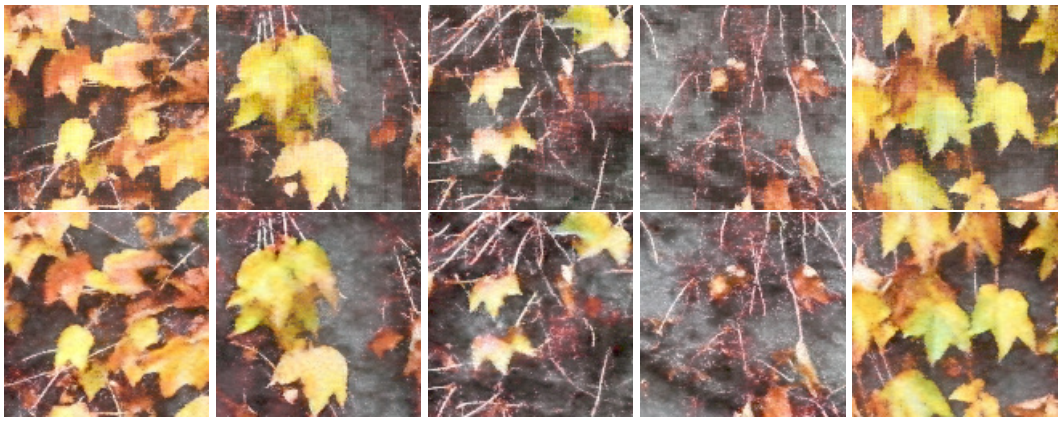


Figure 2.4. Visualization of bilinear interpolation effects after the extraction of encodings. In the first row, the poor results, which are PSNR = 20.57, are demonstrated without bilinear interpolation while the enhanced results, which are PSNR = 21.42, are shown with bilinear interpolation in the last row.

After interpolation, we concatenate the interpolated feature vectors at different levels to give them as inputs to the neural networks. Until here, we encode 2D coordinate information into higher-dimensional space. Now, we utilize neural networks to calculate the corresponding RGB value of the given coordinate x. We take advantage of less memory consumption and nonlinearity thanks to the neural networks.

## 2.3. Implementation

In this section, Python implementation is presented instead of the original architecture written using CUDA. The reason why we implement Python is the long compilation time, which makes it difficult for us to perform our experiments. Therefore, we use the Numba[21] which is a just-in-time (JIT) compiler for Python. It directly translates Python code to machine codes as well as supports CUDA GPU programming using Python code.

In Table 2.1, we compare the execution time of our Python implementation with the original CUDA version after 5000 epochs. Our implementation runs slower than the CUDA version, but we do not need to wait the compilation time. Besides that, we achieve better scores despite using the same configuration. It probably is related to quantization issues. Furthermore, Python benefits from the just-in-time (JIT) compiler, which incurs no compilation time and only has compilation overhead. However, compiling the CUDA version takes significant time, approximately 10 minutes.

Table 2.1. Comparison of our implementation with CUDA implementation of multiresolution hash encoding.

| Method | PSNR ↑ | Time (s) ↓ |
|---|---|---|
| Original | 20.19 | **~ 16** |
| Ours | **25.78** | ~ 88 |

### 2.3.1. Setup

We conduct our experiments on Nvidia GeForce RTX 2060 Super. We use two different images that are 1344 x 896 and 812 x 1084 resolutions. The aim is to learn these images by mapping 2D image coordinates into 3D RGB values. We use PSNR scores to evaluate our experiments.

We follow the configurations of the paper[28] setting the hash table size T to $2^{19}$, the number of levels L to 16, and the length of the feature vectors F to 2. For the neural networks part, we use 2 hidden layers including 64 neurons for each layer and the ReLU activation function is used after the hidden layers. We use uniform distribution $\mathbf{U}(-10^{-4}, 10^{-4})$ for the initialization of feature vectors in the hash tables. In the training

configuration, we choose Adam optimizer[18] with $\beta_1 = 0.9$, $\beta_2 = 0.99$, and $\epsilon = 10^{-15}$. Finally, we use the $L_2$ loss function as defined below:

$$L = \frac{1}{N} \sum_{i=0}^{N} \sum_{c=0}^{C} (y_{ic} - \hat{y}_{ic})^2, \tag{2.4}$$

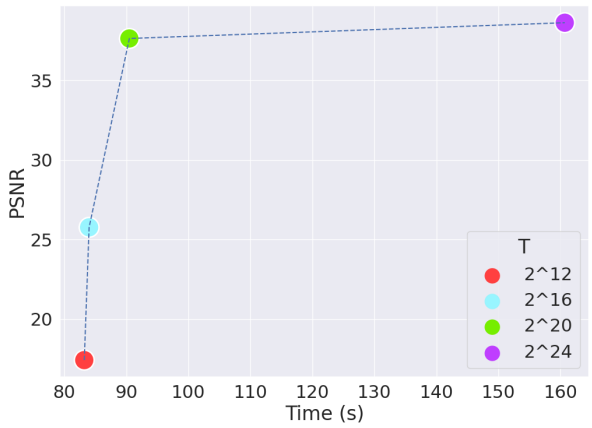
where N is the number of given coordinates, and C is the channel size, which is generally 3. Additional implementation codes are shown in Appendix A.

### 2.3.2. Results

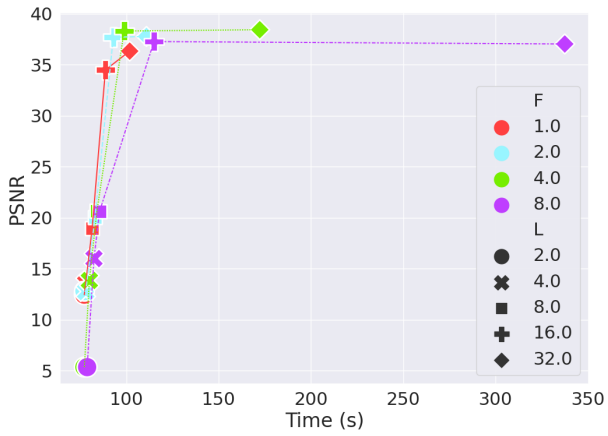
We demonstrate the effects of hyperparameters, T, F, and L in terms of PSNR scores and execution time in Figure 2.5. Besides that, we visualize the estimated image for each determined training interval in Figure 2.7 and Figure 2.6.

We investigate the effects of the hash table size T, the length of feature vectors F, and the number of levels L using an image as shown in Figure 2.5. Firstly, as we increase T, we observe an improvement in the PSNR score without a significant increase in execution time, and also the memory usage increases linearly. The major improvement in the PSNR score is due to fewer hash collisions. Secondly, F does not perform well if we set it to 1. However, there are no significant improvements in the 2, 4, and 8 configurations as well as the memory usage and the execution time. The last one is that while we increase the L value, we obtain higher PSNR scores, but the execution time and memory usage increase significantly.

As we randomly start our estimated image as shown in Figure 2.7 and Figure 2.6, we obtain a black image in the initial state. However, the model learns very fast the structure first without colorization. After that, it colorizes the image and we get the final image.




(a)




(b)

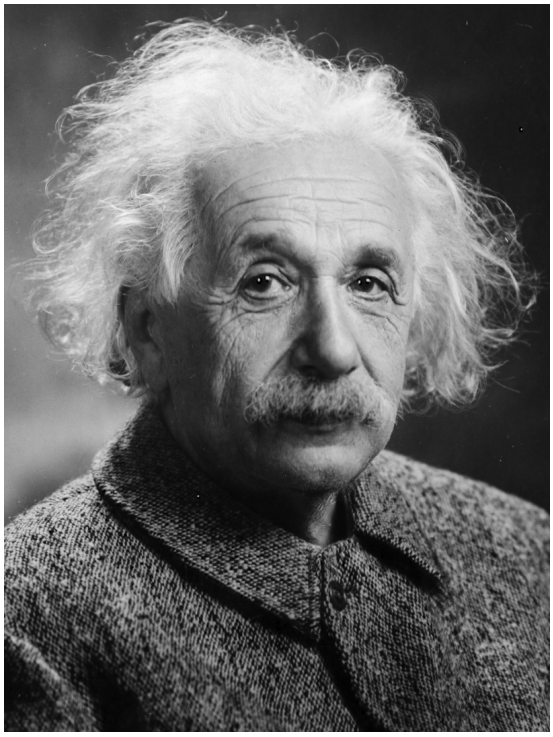
Figure 2.5. We analyze T, F, and L in terms of time and PSNR score using a 1344 x 896 resolution wall image. We choose F=2 and L=16 in the (a) part while we determine T=$2^{20}$ in the (b) part.

(a) Final State

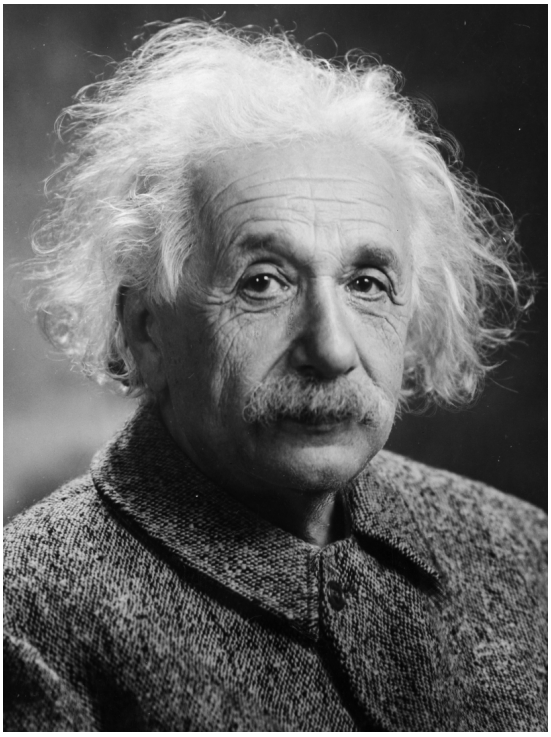
(b) Reference

(c) Initial, 100, 1000, 2500, and 5000 states

Figure 2.6. Visualization of training steps for a 1344 x 896 resolution wall image.



(a) Final State



(b) Reference

(c) Initial, 100, 1000, 2500, and 5000 states

Figure 2.7. Visualization of training steps for an 812 x 1084 resolution Albert Einstein image.

# CHAPTER 3

# MULTI-FRAME SUPER-RESOLUTION WITHOUT PRIORS

In this chapter, we propose a method that generates a high-resolution image using low-resolution images without prior knowledge. Section 3.1 presents the aim of our proposed method and the differences between super-resolution and multiresolution hash encoding[28] as well as gives a general introduction to the methodology. Section 3.2 focuses on the proposed methodology with the calculation of affine transformation among low-resolution images and different setups of our architecture in detail. Section 3.3 demonstrates the results of different experimental setups and comparisons with bicubic interpolation, single-image super-resolution methods,[31,32] and multi-frame super-resolution method.[3] Section 3.4 discusses the results in terms of PSNR scores and obtaining a high-resolution image with or without priors.

## 3.1. Introduction

In this chapter, we focus on how to generate a high-resolution image using low-resolution images of the same scene that have translations and rotations by utilizing spatial hash encoding at different resolutions. We aim to obtain a high-resolution image without learnable priors, only using low-resolution images as ground truths. We achieve this aim through spatial hash encoding that takes coordinate information from the initialized high-resolution image to encode the coordinate-based input to high-dimensional encodings. Then, we give these encodings to a simple multilayer perceptron and finally, we get RGB values of the given coordinates. Therefore, we enhance the randomly initialized high-resolution image iteratively using low-resolution images as ground truths.

We give a detailed version of multiresolution hash encoding in Chapter 2. Here, we discuss what is the difference between multiresolution hash encoding and super-resolution. Super-resolution with spatial hash encoding is to obtain a high-resolution image rather

than the original image resolution as well as to use low-resolution images as ground truths instead of the original image as shown in Figure 3.1. The main similarity is that we update features in the hash tables iteratively while training utilizing multiple low-resolution images corresponding coordinate information.

We propose a multi-frame super-resolution method without considering learnable prior knowledge. We do experiments with different setups. We create two different setups for our experiments: one in which we know the affine matrix between pairs of low-resolution images and another in which we do not know the affine matrix. Besides that, we design a 2D square translation space that covers normalized translation between low-resolution image pairs to generate more accurate RGB generation for each low-resolution image with its translation matrix.

## 3.2. Methodology

In this section, we focus on our proposed architecture for the super-resolution problem. The aim is to obtain a high-resolution image by mapping coordinate information into higher dimensional space with spatial hash encoding and a simple multilayer perceptron architecture. Our motivation is to develop a general super-resolution architecture that does not rely on learnable priors from large data sets. By eliminating undesirable hallucination and colourization effects through our proposed solution that does not rely on priors, we aim to obtain high-quality images. Therefore, we only use low-resolution images from the same scene to generate a corresponding high-resolution image, not train with enormous data sets.

In Chapter 2, we see how to learn an image using learnable features in the hash tables at different levels using multiresolution hash encoding. In this setup, we only use an image to update features iteratively with given coordinate information. When we give a pixel coordinate $(x, y)$ of the given image at the end of the training, we produce an RGB value corresponding to this coordinate through hash tables at different levels. However, our proposed architecture differs from this setup in terms of the generated image, ground truths, and additional 2D translation space covering translation information among ground truth images. The generated image will be our high-resolution image, which is a higher spatial resolution than ground truths images. We utilize the low-resolution images as

ground truths and design a loss function to account for all of them.

We choose a low-resolution image as the base image and determine the transformation between the base image and other image pairs to gather additional information. Therefore, we design an algorithm to calculate the transformation. We utilize SIFT[24] algorithm to extract keypoints from image pairs and to apply a brute-force matcher algorithm to match descriptors. After keypoint matching, we sort these matches in terms of the Euclidean distance of descriptors to select the best minimum three keypoints. Then, with these three keypoints, we calculate the affine transformation between image pairs. In Section 3.2.1, we give the calculation of the affine transformation matrix in detail.

Firstly, we propose a baseline architecture that does not include 2D block space. This architecture generates an RGB value corresponding to the given pixel coordinate and it compares this RGB value with all ground truth images. However, in the second approach, we design a 2D block space to cover affine translations among low-resolution images. We choose a base image from low-resolution images and we calculate the affine transformation between the base image and the other image using Section 3.2.1. Therefore, we produce individual RGB values corresponding to the given pixel coordinate for each low-resolution image. As a result, we observe that this approach increases our PSNR score significantly as shown in Table3.1.

### 3.2.1. Calculation of Affine Matrix

We assume that parallelism, collinearity, and ratio of areas and lengths are preserved among low-resolution images, yet there is no projective transformation. For that reason, we determine the affine transformation matrix between the base image and the other image from our low-resolution images so that we get the corresponding pixels for each image.

The affine transformation is a linear transformation function that maps two vector spaces while preserving our assumptions. It is shown

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{bmatrix} a_{11} & a_{12} & tx \\ a_{21} & a_{22} & ty \\ 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \tag{3.1}$$

or also a simpler form.

$$\mathbf{x}' = \begin{bmatrix} \mathbf{A} & \mathbf{t} \\ \mathbf{0}^\top & \mathbf{1} \end{bmatrix} \mathbf{x} \tag{3.2}$$

Since the affine transformation has six degrees of freedom, we need six equations to determine the affine transformation. Therefore, to obtain six equations, we choose three points from two images, resulting in two equations for each point and a total of six equations.

We utilize SIFT[24] to extract keypoints and descriptors from images and we apply the Brute-Force Matcher algorithm to match keypoints using descriptors based on their distances. Also, at the end of the matching, we check each pair of descriptors with the ratio test to eliminate outliers. After that, we solve the linear equation with selected three points that has the minimum three distances from the matches. Finally, to minimize the error in the affine transformation, we utilize the Levenberg-Marquardt algorithm, which is a non-linear least squares optimization method. We use the linear solution of the affine matrix as the initial state and calculate the least squares using all matched points after the ratio test. Additional implementation codes are shown in Listing B.1 of Appendix B.

As a result, we warp the images depending on the base image using the determined affine transformation as well as we use the translation information in Section 3.2.3 to produce RGB values for each low-resolution image according to its translation.

### 3.2.2. Grid-based Super-Resolution using Spatial Hash Encoding

In this section, we propose a baseline model to generate a higher spatial resolution image using low-resolution images by mapping pixel coordinates into higher dimensional space with spatial hash encoding to produce RGB values corresponding to pixel coordinates.

Firstly, we have to determine translations among low-resolution images. For that reason, we choose a base image from them to calculate the affine transformation between the base image and the others according to Section 3.2.1. After the calculation of the affine translation, we warp the images based on this translation. Now, we get correctly match the pixel coordinates for all low-resolution images to use them as ground truths.

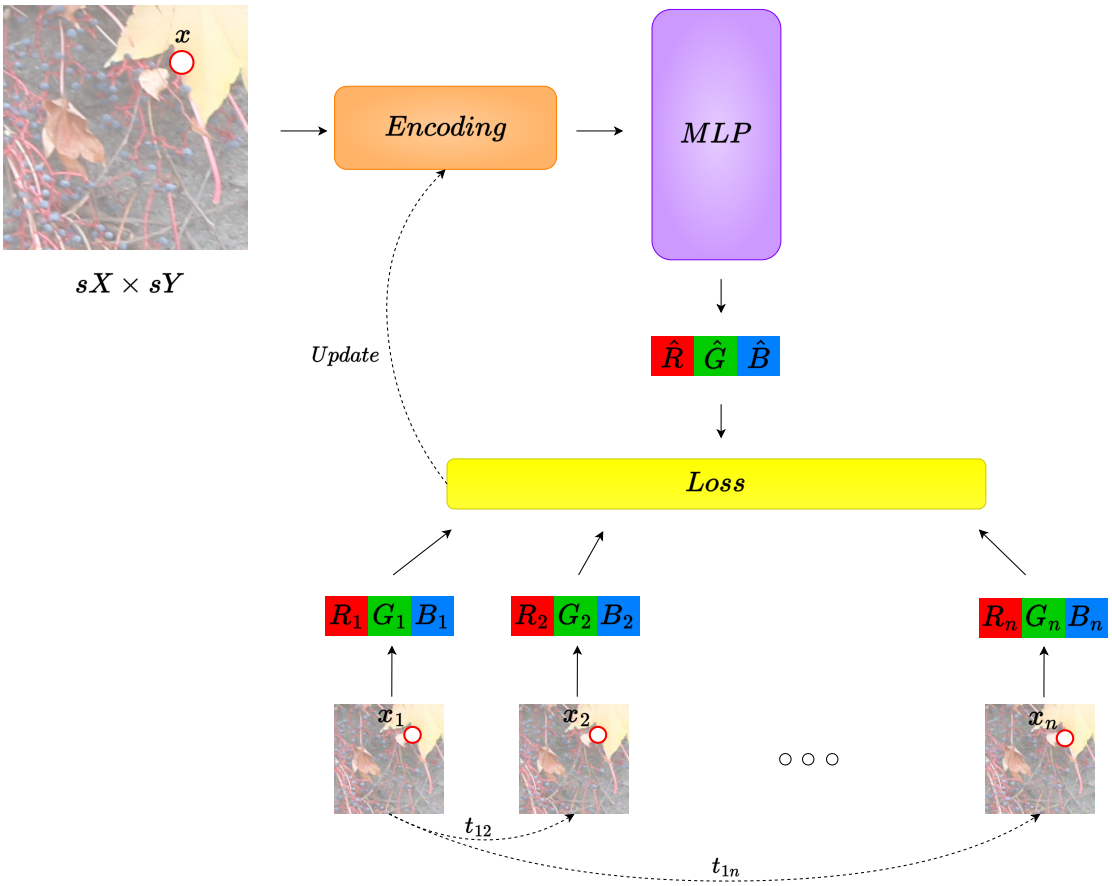



Figure 3.1. Our baseline architecture. It does not include 2D block space after the MLP part.

As shown in Figure 3.1, we first initialize the features into the hash tables to generate a high-resolution image according to configurations in Chapter 2. After the initialization of features, we get pixel coordinates to get indices of the hash table using the spatial hash function $\mathcal{H}(.)$ for each surrounding corner at different levels. We get the features with given indices of the hash tables for each surrounding corner and we apply a bilinear interpolation to these surrounding corners to keep continuity. Then, we concatenate the interpolated features extracted from different levels. We give these extracted features to a simple multilayer perception, which has 2 hidden layers with 64 neurons and an output layer with 3 neurons. The output of the neural network is an RGB value corresponding to a given pixel coordinate for all low-resolution images. We use an $L_2$ loss function which is defined in Equation 2.4 based on each low-resolution image.

### 3.2.3. Grid-based Super-Resolution using Spatial Hash Encoding with 2D Block Space

Our proposed model without 2D block space is stuck at the average RGB values calculated from low-resolution images. Also, in some conditions like high-contrast images, the model is not able to generate accurate RGB values and this leads to underfitting. To solve these problems, we integrate a 2D block space at the end of the MLP part. The MLP part produces 4 RGB values for the 2D block space. We apply bilinear interpolation using 2D block space to generate RGB value for each low-resolution image. The integrated 2D block space covers all low-resolution images independently of each other so that it cannot be underfitting. We see the improvements in Table 3.1.

From the low-resolution images, we select a base image and assume it to be at the center of a 2D block space. According to the affine translations between the base image and the others, we normalize the translation values between [-1, 1] to represent all low-resolution images. As shown in Figure 3.2, the neural network gives four corners with 3D RGB values rather than only a single RGB value as the baseline model. We generate an individual RGB value for each low-resolution image depending on their translation information through bilinear interpolation. This system enables us to utilize a loss function for each low-resolution image independently. As a result, integrating a 2D block space

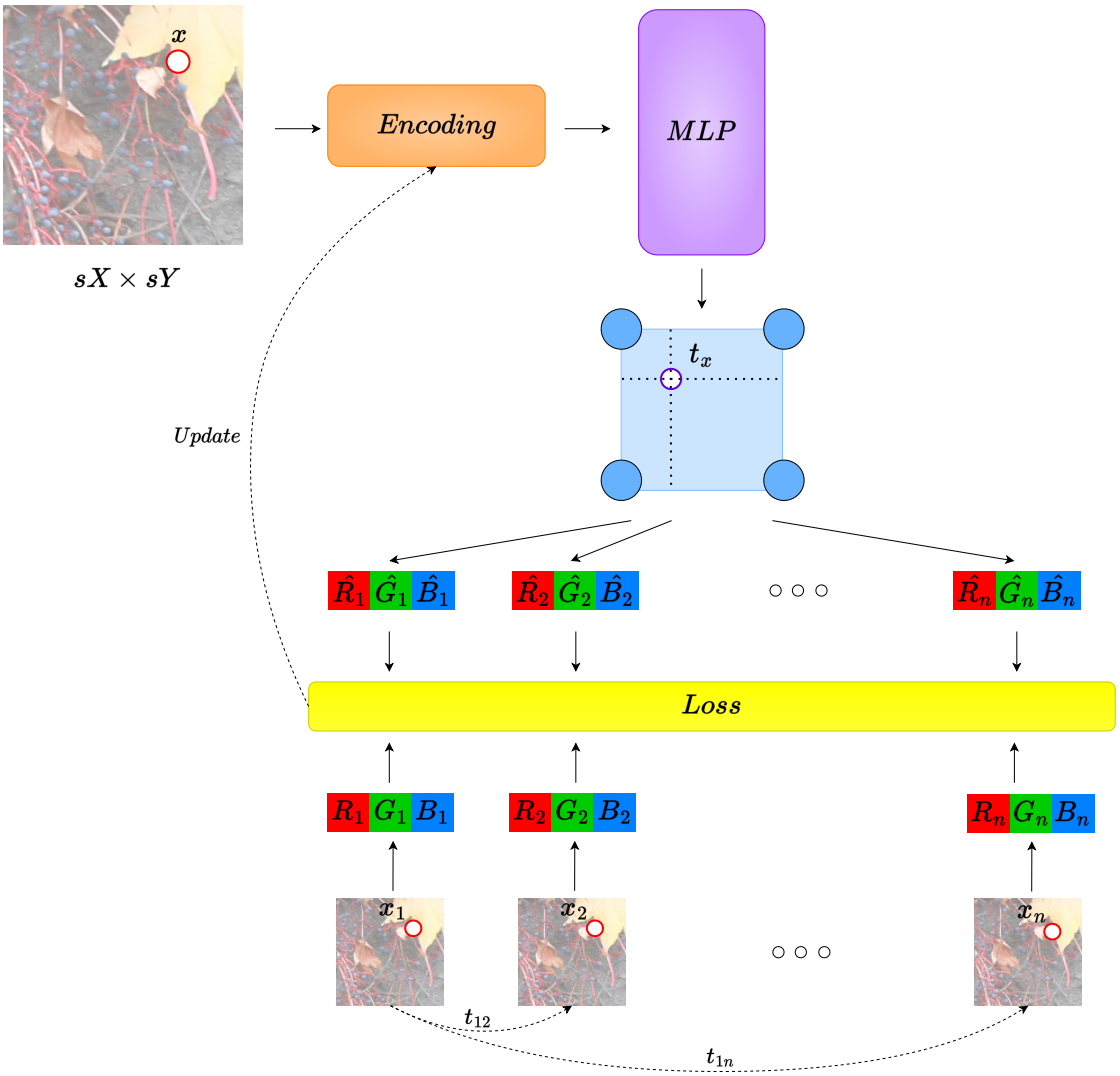



Figure 3.2. Our proposed architecture with 2D block space.

after the neural network enhances the PSNR score. Additional implementation codes are shown in Appendix B between Listing B.2 - B.4.

Table 3.1. Comparison among our proposed architecture with and without 2D block space, and bicubic interpolation on the synthetic dataset of 4x upsampling.

| Method | PSNR↑ | SSIM↑ |
|---|---|---|
| Min. of Bicubic | 23.07 | 0.66 |
| Avg. of Bicubic | 24.76 | 0.75 |
| Max. of Bicubic | 25.27 | 0.78 |
| Ours w/o 2D Block | 27.82 | 0.78 |
| Ours w/ 2D Block | **28.51** | **0.84** |

## 3.3. Experiments

In this section, we evaluate our proposed architecture with and without 2D block space. Besides that, we find the best hyperparameters of our architecture in terms of the number of low-resolution images N, the hash table size T, the length of the feature vector F, and the number of levels L.

After that, we evaluate our proposed architecture against state-of-the-art methods, which include single-image super-resolution and multi-frame super-resolution approaches. For the single-image super-resolution approaches, including bicubic interpolation and SR3,[32] we measure individual low-resolution images in terms of PSNR and SSIM scores. We get their minimum, average, and maximum scores to compare with our proposed method. In the multi-frame super-resolution comparison, including DeepRep,[3] we get only a PSNR and SSIM score using all low-resolution images and we compare these scores with our proposed method.

### 3.3.1. Setup

We conduct our experiments on Nvidia GeForce RTX 2060 Super. We use three different images to generate our synthetic dataset to compare our method with the state-of-the-art methods. We generate our synthetic dataset that includes RAW and sRGB color spaces following inverse camera pipeline[4] and DeepBurst.[2] First of all, we give

an image with sRGB color spaces, then we convert it into raw sensor data using inverse camera pipeline.[4] We apply random translation and rotation for all desired numbers of low-resolution images, [-2, 2] pixels, and [-5, 5] degrees, respectively. After transformation, these translated and rotated images are downsampled by scale factors, 4x and 8x. Before the mosaicking, we add shot and read noise to the low-resolution images and we obtain synthetic low-resolution images with sRGB color space. Finally, we apply the mosaicking algorithm to the image and we obtain the final RAW low-resolution images. We use both sRGB and RAW low-resolution images with desired crop sizes such as $64 \times 64$ for SR3[32] downsampled by 8x and $96 \times 96$ for DeepRep[3] downsampled by 4x in comparison. There are 25 low-resolution and high-resolution image pairs for each dataset.

We use the initialization configurations for the neural network and the encoding parts described in Section 2.3.1 to compare our model versions and to find the optimal hyperparameters such as the hash table size T, the length of feature vectors F, and the number of levels L. Firstly, we compare our baseline model with the 2D block space added to our baseline model. Secondly, we search for the optimum number of low-resolution images for the generation of a high-resolution image. Finally, we demonstrate the ideal T, F, and L.

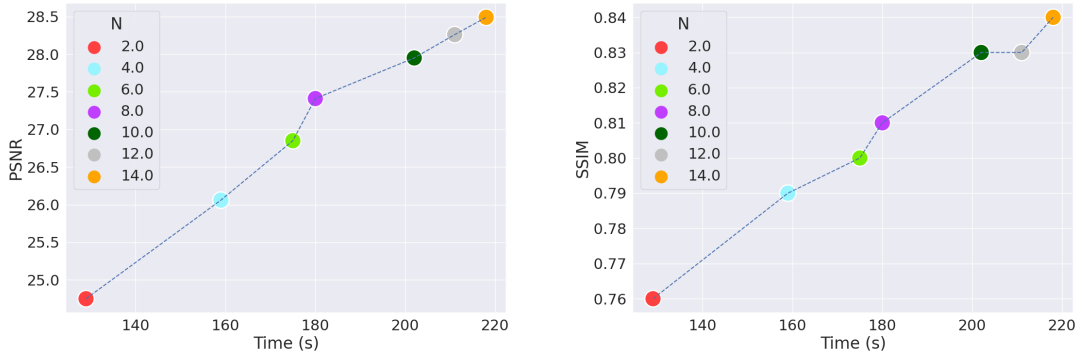
We compare our proposed method with bicubic interpolation and SR3[32] using the generated synthetic dataset which includes sRGB low-resolution images with $64 \times 64$ resolution to generate a high-resolution image with $512 \times 512$ resolution. For the comparison with DeepRep,[3] we utilize RAW images from the synthetic dataset with low-resolution and high-resolution are, $96 \times 96$ and $384 \times 384$, respectively.

## 3.4. Results and Conclusion

In this section, we analyze the effects of the hyperparameters such as the number of low-resolution images N, the hash table size T, the feature vector F, and the number of levels L. Also, we compare our proposed method with bicubic interpolation, SR3[32] and DeepRep[3] in terms of PSNR and SSIM scores.

Firstly, we analyze the number of low-resolution images used to generate a high-resolution image N in Figure 3.3. As we can see if we increase the number of low-resolution images, the model produces more accurate results due to gathering additional information.

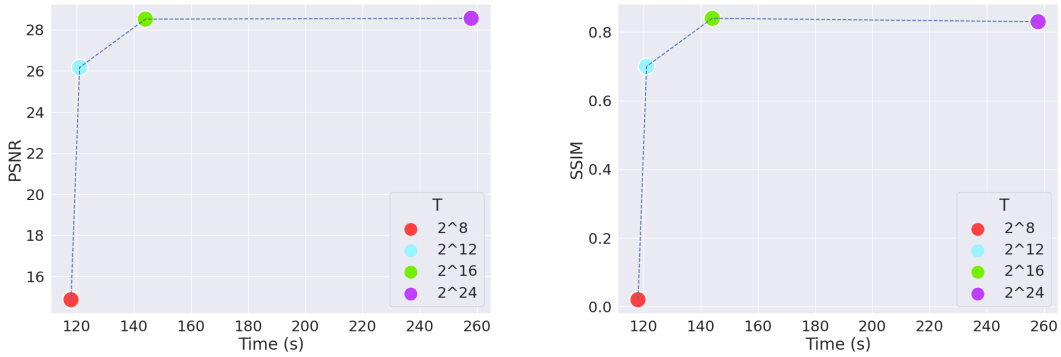
Unfortunately, it sometimes cannot enhance results even decrease the scores because of bad samples. However, when we use more images, the model is able to eliminate bad samples and focus on accurate ones. Therefore, it is close to the saturation point after 12 images. Secondly, we investigate the effects of the hash table size T, the length of feature vectors F, and the number of levels L as shown in Figure 3.4 and Figure 3.5. We mentioned their effects in Chapter 2.3.2. The difference between Chapter 2 and Chapter 3 is that less number of parameters is needed. The reason for this is to leverage additional data from low-resolution images and the model reaches the minima faster than in Chapter 2.




(a) (b)

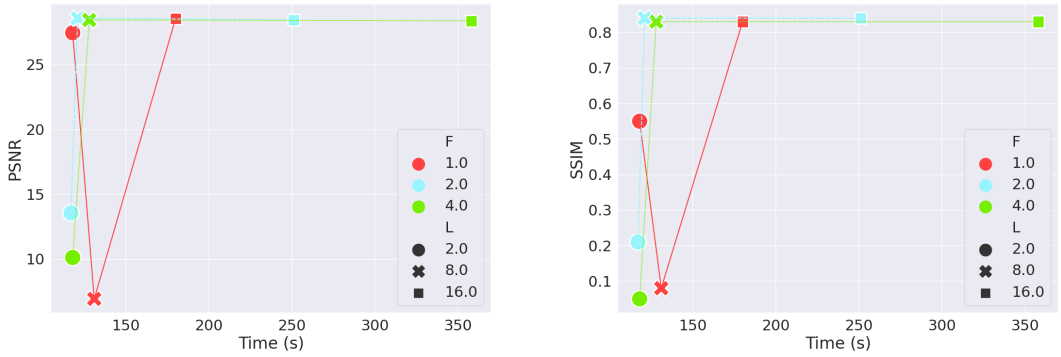
Figure 3.3. We evaluate the PSNR and SSIM scores for different numbers of low-resolution images used to generate a high-resolution image, using the 4x downsampled synthetic dataset. We choose T=$2^{24}$, F=2, and L=16 for the experiment.

We compare our proposed method with bicubic interpolation and SR3[32] on the synthetic dataset downsampled by 8, including 25 low-resolution and high-resolution image pairs, in terms of the single-image super-resolution manner. We evaluate each low-resolution image independently for bicubic interpolation and SR3[32] and we get the minimum, average, and maximum PSNR and SSIM scores for all image scenes in the dataset to compare with our proposed method as shown in Table 3.2. As we can see the generative super-resolution is not able to produce accurate results, or even it is worse than the bicubic interpolation as shown in Figure 3.6. Additional results are shown in Figure C.1 of Appendix C. It generates something other than what it should be and this leads to hallucination effects. Our results are much better than SR3.[32] Also, although we use bicubic interpolation in our system, we get better scores than bicubic interpolation because of exploiting additional data coming from many low-resolution images. We show that we




(a) (b)

Figure 3.4. We evaluate the PSNR and SSIM scores for different hash table sizes to generate a high-resolution image, using the 4x downsampled synthetic dataset. We choose N=14, F=2, and L=16 for the experiment.




(a) (b)

Figure 3.5. We evaluate the PSNR and SSIM scores for different numbers of levels and lengths of feature vectors to generate a high-resolution image, using the 4x downsampled synthetic dataset. We choose N=14 and T=$2^{24}$ for the experiment.

achieve better scores and consistent generation than bicubic interpolation and SR3.[32]

Table 3.2. Comparison among our proposed method with 2D block space, SR3,[32] and bicubic interpolation on the 8x downsampled synthetic dataset.

| Method | PSNR ↑ | SSIM ↑ |
|---|---|---|
| Min. of Bicubic | 20.78 | 0.54 |
| Avg. of Bicubic | 22.87 | 0.63 |
| Max. of Bicubic | 23.82 | 0.68 |
| Min. of SR3[32] | 13.96 | 0.29 |
| Avg. of SR3[32] | 18.01 | 0.44 |
| Max. of SR3[32] | 22.05 | 0.57 |
| Ours | **26.13** | **0.70** |

To compare our proposed method with bicubic interpolation and DeepRep[3] using 4x downsampled synthetic dataset that includes 25 low-resolution and high-resolution image pairs. This dataset has 14 low-resolution images with 96 x 96 resolution and a high-resolution image with 384 x 384 resolution for each pair. We get the minimum, average, and maximum PSNR and SSIM scores for bicubic interpolation while we utilize 14 low-resolution images to generate a high-resolution image for DeepRep[3] and our proposed method. We obtain much better results compared to the 8x synthetic dataset, and the bicubic interpolation results also increased. Because 4x upsampling is an easier problem than 8x upsampling. When we compare our proposed method with DeepRep,[3] DeepRep[3] achieves high scores according to ours significantly. Our approach does not utilize priors and we do not apply offline training before using vast super-resolution datasets as shown in Table 3.3. We see that DeepRep[3] is good at generating high frequency and high spatial resolutions with priors as shown in Figure 3.7. Additional results are shown in Figure C.2 of Appendix C. However, they are not able to solve colorization problems although generating a high-resolution image in detail.

We show that our proposed model can be used to generate high-resolution images using low-resolution images without prior information. It achieves better scores compared to traditional and generative approaches in particular single-image super-resolution. However, the results of our proposed model are poor when we compare it with multi-frame super-resolution approaches like DeepRep.[3] They take advantage of prior information by training immense super-resolution datasets. However, they cannot the colorization problem and sometimes generates unexpected colors. This leads to an unstable high-resolution

Table 3.3. Comparison among our proposed method with 2D block space, DeepRep,[3] and bicubic interpolation on the 4x downsampled synthetic dataset.

| Method | PSNR ↑ | SSIM ↑ |
|---|---|---|
| Min. of Bicubic | 23.07 | 0.66 |
| Avg. of Bicubic | 24.76 | 0.75 |
| Max. of Bicubic | 25.27 | 0.78 |
| DeepRep[3] | **33.12** | **0.92** |
| Ours | 28.32 | 0.83 |

generation. As a result, our proposed model is very stable but it is worse than DeepRep[3] while DeepRep[3] achieves high scores but it sometimes has undesirable effects.

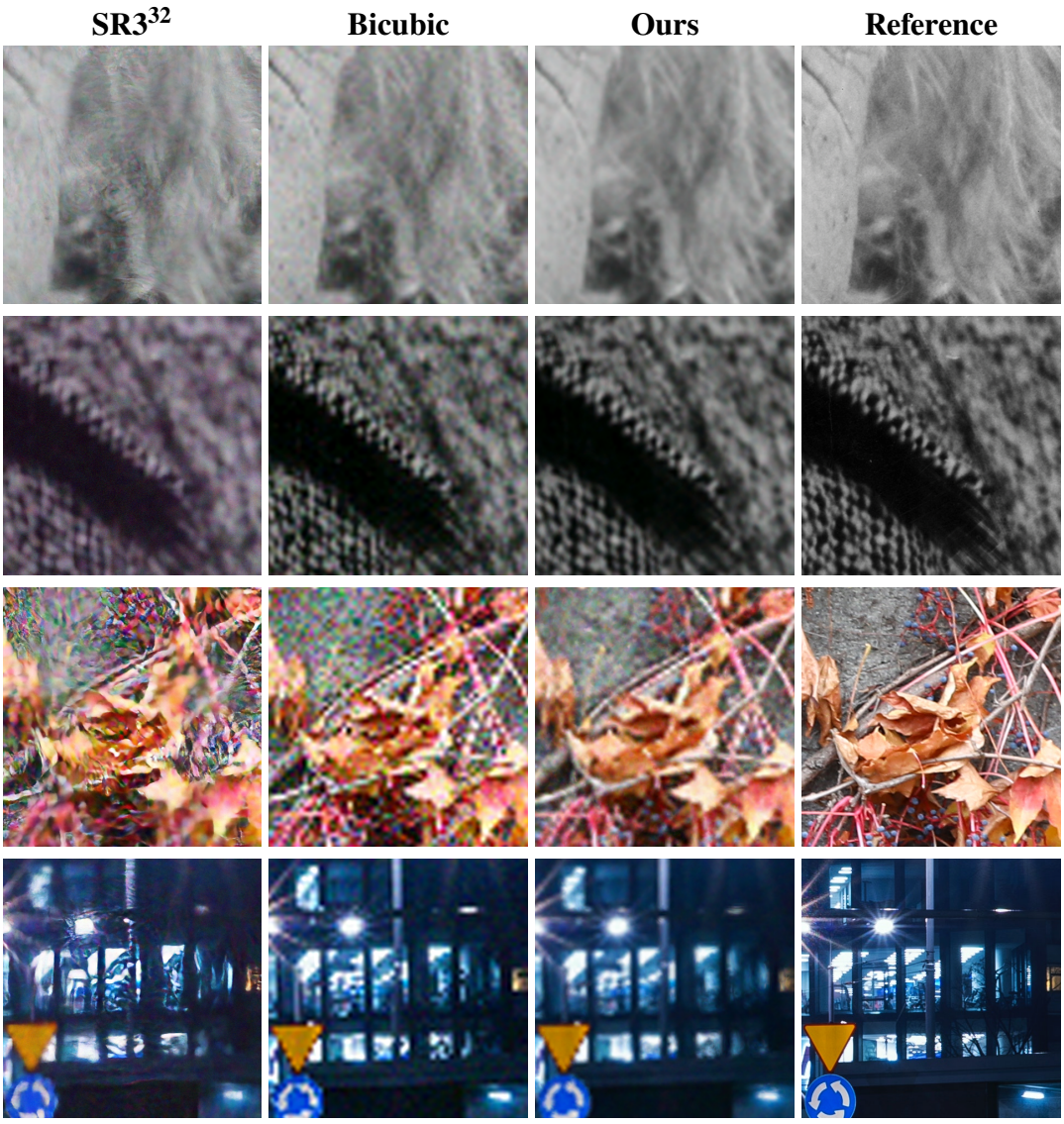



Figure 3.6. The results of bicubic interpolation, SR3[32] and ours.

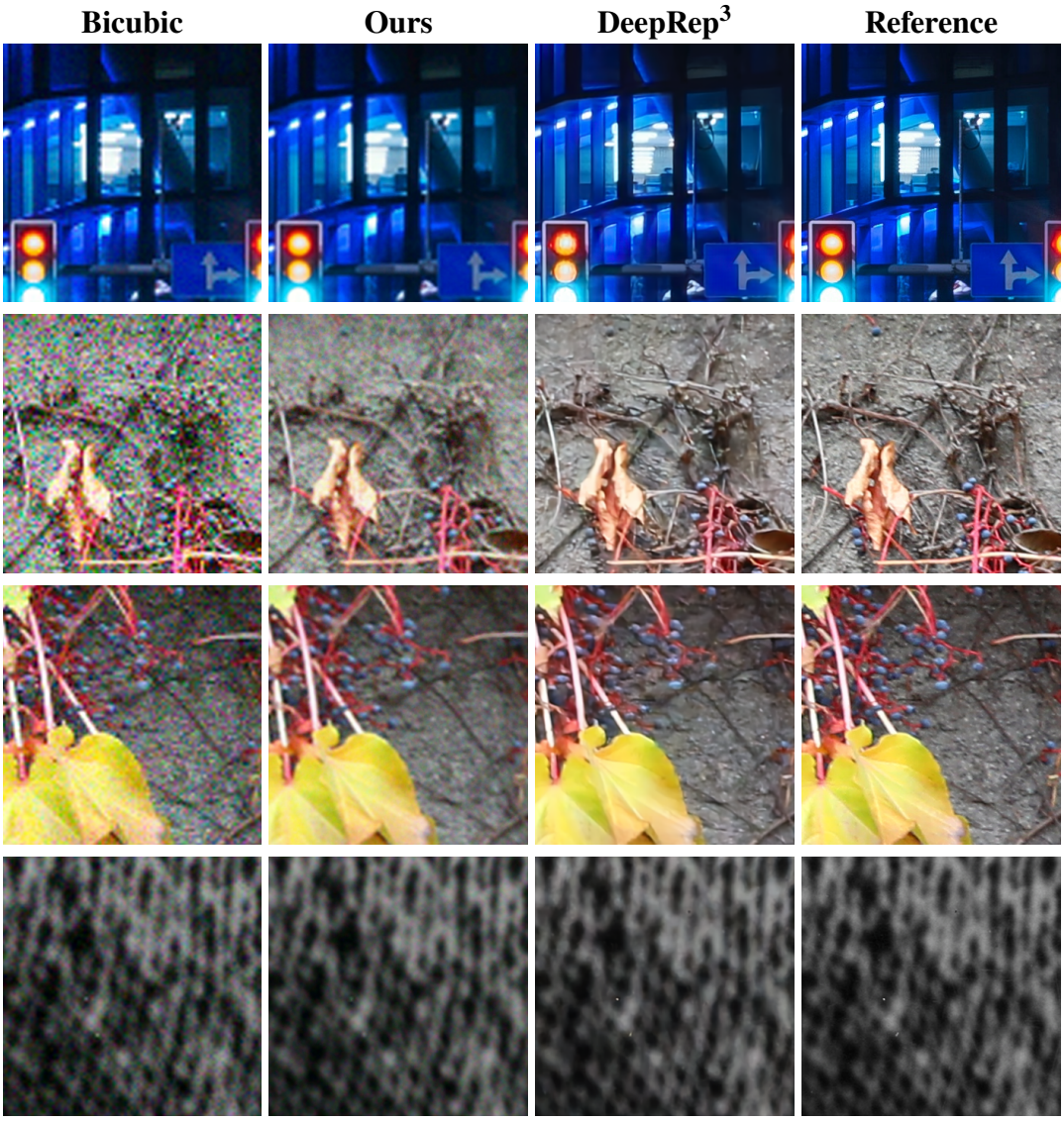



Figure 3.7. The results of bicubic interpolation, DeepRep[3] and ours.

# CHAPTER 4

# CONCLUSION

## 4.1. Conclusion

In this thesis, our aim is to generate a high-resolution image exploiting multiple low-resolution images without prior information. We design a super-resolution model utilizing a task-agnostic architecture. This model randomly initializes learnable features into the hash tables and maps coordinate information into higher dimensional space using the hash tables. After that, it generates RGB values for each coordinate information after the MLP part is given high-dimensional features as input. Thus, we achieve two important advantages such as less training time and incapable data sets. Firstly, we do not need long training times, we only use images coming from scenes. Later, we do not use data sets that do not cover all real-world scenarios, so we eliminate possible undesirable effects only using related low-resolution images.

In Chapter 2, we implement the multiresolution hash encoding architecture,[28] which learns an image by extracting features at different levels into the hash tables using the coordinate information. We demonstrate how to parametrize the graphics primitives through spatial hashing and MLPs. Therefore, we obtain dynamic, fast, and task-agnostic architecture that maps learnable encodings into higher-dimensional space to give them to MLPs as input. Finally, we generate RGB values by parameterizing the given coordinate information.

Multiresolution hash encoding[28] learns an image with the same resolution and only uses a single image as ground truth. In Chapter 3, we propose a model that utilizes the core architecture in Chapter 2. The main differences are that we generate higher resolution rather than given inputs and we exploit multiple low-resolution images as ground truths to generate a high-resolution image. We aim to generate a high-resolution image by encoding the coordinate-based input to high-dimensional encodings through the hash tables. The main contributions are designing a task-agnostic super-resolution model that can be used in different areas, employing low-resolution images from the same scene rather than large

data sets, preventing potential hallucination and colorization effects by not considering prior information, and finally exploiting neural networks implicitly instead of defining simple assumption to generate higher-spatial resolution.

## 4.2. Discussion

We show that our proposed model achieves better PSNR and SSIM scores compared to single-image super-resolution approaches, but worse when compared to multi-frame super-resolution approaches with priors. In the multi-frame super-resolution approach, they utilize prior information coming from vast super-resolution data sets and they are able to catch high-frequency details in the images. However, they cannot solve the colorization problem and sometimes generated unstable high-resolution images when we compare them to our proposed model.

We expect that our proposed method increases scores of super-resolution using additional information from multiple low-resolution images of the same scene without super-resolution data sets. The method obtains remarkable scores but cannot achieve higher than state-of-the-art models with priors. Because we do not know which sample is the best or the worse. We think that the model may solve this problem by utilizing neural networks if we get more samples from the scene. However, we cannot always get the best samples from the scene due to environmental and hardware settings as well as we use bicubic interpolation to upsample. This sometimes leads to converging lower scores. If a loss function is designed to cover low-resolution images all in one piece rather than a regression loss for each low-resolution image, we do not suffer from the local minima. But on the other side, if we would like to generate stable results not suffering from hallucination and colorization problems, our proposed method is more suitable than other methods with priors.

## 4.3. Future Work

We use bicubic interpolation to get a higher spatial resolution for each low-resolution image in our proposed model. If the low-resolution images are not well,

this means there are high noise and blur, our model cannot exploit additional data coming from low-resolution images. It even performs worse scores when using bad low-resolution images. We may increase our scores by exploiting additional data if we use multiple super-resolution techniques to generate ground truths for our model. Therefore, we may reduce hallucination and colorization effects by combining multiple methods.

We utilize a regression loss function for RGB color values for each low-resolution image. We consider them individually. Instead of a regression loss, a smooth or curvature loss function is used to cover samples from low-resolution images all in one piece.

The aim of this work is to produce the optimum high-resolution resolution image using low-resolution images with spatial hash encoding, MLPs, and regression loss. Instead of MLPs and regression loss, we take advantage of generative models to generate high frequency and high spatial resolutions. If we gather enough samples from the low-resolution images, we may estimate the distribution of the high-resolution image. Therefore, we do not need to additional loss functions explicitly.

# REFERENCES


1. Anwar, S.; Khan, S.; Barnes, N. A Deep Journey into Super-Resolution: A Survey. *ACM Comput. Surv.* **2020**, *53*, DOI: 10.1145/3390462.

2. Bhat, G.; Danelljan, M.; Van Gool, L.; Timofte, R. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2021, pp 9209–9218.

3. Bhat, G.; Danelljan, M.; Yu, F.; Van Gool, L.; Timofte, R. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, 2021, pp 2460–2470.

4. Brooks, T.; Mildenhall, B.; Xue, T.; Chen, J.; Sharlet, D.; Barron, J. T. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019.

5. Bulat, A.; Yang, J.; Tzimiropoulos, G. In *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018.

6. Dabbech, A.; Terris, M.; Jackson, A.; Ramatsoku, M.; Smirnov, O. M.; Wiaux, Y. First AI for Deep Super-resolution Wide-field Imaging in Radio Astronomy: Unveiling Structure in ESO 137-006. *The Astrophysical Journal Letters* **2022**, *939*, L4, DOI: 10.3847/2041-8213/ac98af.

7. Dai, S.; Han, M.; Xu, W.; Wu, Y.; Gong, Y. In *2007 IEEE Conference on Computer Vision and Pattern Recognition*, 2007, pp 1–8, DOI: 10.1109/CVPR.2007.383028.

8. Deudon, M.; Kalaitzis, A.; Goytom, I.; Arefin, M. R.; Lin, Z.; Sankaran, K.; Michalski, V.; Kahou, S. E.; Cornebise, J.; Bengio, Y. HighRes-net: Recursive Fusion for Multi-Frame Super-Resolution of Satellite Imagery, 2020, arXiv: 2002.06460 [cs.CV].

9. Dong, C.; Loy, C. C.; He, K.; Tang, X. Image Super-Resolution Using Deep Convolutional Networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence* **2016**, *38*, 295–307, DOI: 10.1109/TPAMI.2015.2439281.

10. Dudhane, A.; Zamir, S. W.; Khan, S.; Khan, F. S.; Yang, M.-H. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2022, pp 5759–5768.

11. Fischler, M. A.; Bolles, R. C. Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography. **1981**, *24*, 381–395, DOI: 10.1145/358669.358692.

12. Georgescu, M.-I.; Ionescu, R. T.; Miron, A.-I.; Savencu, O.; Ristea, N.-C.; Verga, N.; Khan, F. S. In *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision (WACV)*, 2023, pp 2195–2205.

13. Haris, M.; Shakhnarovich, G.; Ukita, N. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019.

14. Hou, H.; Andrews, H. Cubic splines for image interpolation and digital filtering. *IEEE Transactions on Acoustics, Speech, and Signal Processing* **1978**, *26*, 508–517, DOI: 10.1109/TASSP.1978.1163154.

15. Irani, M.; Peleg, S. Improving resolution by image registration. *CVGIP: Graphical Models and Image Processing* **1991**, *53*, 231–239, DOI: 10.1016/1049-9652(91)90045-L.

16. Kawulok, M.; Benecki, P.; Hrynczenko, K.; Kostrzewa, D.; Piechaczek, S.; Nalepa, J.; Smolka, B. In *Real-Time Image Processing and Deep Learning 2019*, ed. by Kehtarnavaz, N.; Carlsohn, M. F., SPIE: 2019; Vol. 10996, 109960B, DOI: 10.1117/12.2519579.

17. Kim, J.; Lee, J. K.; Lee, K. M. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

18. Kingma, D. P.; Ba, J. Adam: A Method for Stochastic Optimization, 2017, arXiv: 1412.6980 [cs.LG].

19. Krizhevsky, A.; Sutskever, I.; Hinton, G. E. ImageNet Classification with Deep Convolutional Neural Networks. *Commun. ACM* **2017**, *60*, 84–90, DOI: 10.1145/3065386.

20. Lai, W.-S.; Huang, J.-B.; Ahuja, N.; Yang, M.-H. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

21. Lam, S. K.; Pitrou, A.; Seibert, S. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, 2015, pp 1–6.

22. Li, X.; Orchard, M. New edge-directed interpolation. *IEEE Transactions on Image Processing* **2001**, *10*, 1521–1527, DOI: 10.1109/83.951537.

23. Lim, B.; Son, S.; Kim, H.; Nah, S.; Mu Lee, K. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, 2017.

24. Lowe, D. G. Distinctive Image Features from Scale-Invariant Keypoints. *International Journal of Computer Vision* **2004**, *60*, 91–110.

25. Lukac, R., *Computational photography: methods and applications*; CRC press: 2017.

26. Martel, J. N. P.; Lindell, D. B.; Lin, C. Z.; Chan, E. R.; Monteiro, M.; Wetzstein, G. Acorn: Adaptive Coordinate Networks for Neural Scene Representation. *ACM Trans. Graph.* **2021**, *40*, DOI: 10.1145/3450626.3459785.

27. Mildenhall, B.; Srinivasan, P. P.; Tancik, M.; Barron, J. T.; Ramamoorthi, R.; Ng, R. NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis. **2021**, *65*, 99–106, DOI: 10.1145/3503250.

28. Müller, T.; Evans, A.; Schied, C.; Keller, A. Instant neural graphics primitives with a multiresolution hash encoding. *ACM Transactions on Graphics* **2022**, *41*, 1–15, DOI: 10.1145/3528223.3530127.

29. Nasrollahi, K.; Moeslund, T. Super-resolution: A comprehensive survey. *Machine Vision and Applications* **2014**, *25*, 1423–1468, DOI: 10.1007/s00138-014-0623-4.

30. Nguyen, N.; Milanfar, P. A Wavelet-Based Interpolation-Restoration Method for Superresolution. *Circuits, Systems, and Signal Processing* **2000**, *19*, 321–338, DOI: 10.1007/BF01200891.

31. Rombach, R.; Blattmann, A.; Lorenz, D.; Esser, P.; Ommer, B. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2022, pp 10684–10695.

32. Saharia, C.; Ho, J.; Chan, W.; Salimans, T.; Fleet, D. J.; Norouzi, M. Image Super-Resolution via Iterative Refinement. *IEEE Transactions on Pattern Analysis and Machine Intelligence* **2023**, *45*, 4713–4726, DOI: 10.1109/TPAMI.2022.3204461.

33. Stone, H.; Orchard, M.; Chang, E.-C.; Martucci, S. A fast direct Fourier-based algorithm for subpixel registration of images. *IEEE Transactions on Geoscience and Remote Sensing* **2001**, *39*, 2235–2243, DOI: 10.1109/36.957286.

34. Szeliski, R., *Computer Vision: Algorithms and Applications*, 1st; Springer-Verlag: Berlin, Heidelberg, 2010.

35. Teschner, M.; Heidelberger, B.; Müller, M.; Pomerantes, D.; Gross, M. H. In *International Symposium on Vision, Modeling, and Visualization*, 2003.

36. Tsai, R. Y.; Huang, T. S. Multiframe image restoration and registration. **1984**.

37. Vandewalle, P.; Süsstrunk, S.; Vetterli, M. A Frequency Domain Approach to Registration of Aliased Images with Application to Super-Resolution. *Eurasip Journal on Applied Signal Processing* **2006**, *2006*, DOI: 10.1155/ASP/2006/71459.

38. Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A. N.; Kaiser, L. u.; Polosukhin, I. In *Advances in Neural Information Processing Systems*, ed. by Guyon, I.; Luxburg, U. V.; Bengio, S.; Wallach, H.; Fergus, R.; Vishwanathan, S.; Garnett, R., Curran Associates, Inc.: 2017; Vol. 30.

39. Wang, X.; Yu, K.; Wu, S.; Gu, J.; Liu, Y.; Dong, C.; Qiao, Y.; Change Loy, C. In *Proceedings of the European Conference on Computer Vision (ECCV) Workshops*, 2018.

40. Wronski, B.; Garcia-Dorado, I.; Ernst, M.; Kelly, D.; Krainin, M.; Liang, C.-K.; Levoy, M.; Milanfar, P. Handheld Multi-Frame Super-Resolution. *ACM Trans. Graph.* **2019**, *38*, DOI: 10.1145/3306346.3323024.

41. Zhang, Y.; Tian, Y.; Kong, Y.; Zhong, B.; Fu, Y. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.

# APPENDIX A

# PYTHON CODE FOR MULTIRESOLUTION HASH ENCODING

```
import numpy as np
from numba import cuda
import math
from model_utils import next_multiple, div_round_up, _forward, _sum_update_grid_params, _update_grid_params


class Encoding:
    def __init__(self, scale, n_levels, base_resolution, n_features, hashmap_size, lr):
        self.scale = scale
        self.n_levels = n_levels
        self.base_resolution = base_resolution
        self.n_features = n_features
        self.hashmap_size = hashmap_size
        self.lr = lr
        self.n_backward_contents = 9  # 4 indices, grid index, and 4 derivatives
        self.hashmap_offsets_table, self.n_params = self.determine_grids()
        self.grid_params = np.full((self.n_params,), fill_value=np.random.uniform(-1e-4, 1e-4), dtype=np.float32)

    def determine_grids(self):
        hashmap_offsets_table = np.empty((self.n_levels + 1,), dtype=np.float32)
        offset = 0
        max_params = np.Inf
        for i in range(self.n_levels):
            scale = math.pow(2, i * math.log2(self.scale)) * self.base_resolution - 1.0
            # to eliminate the floating point rounding
            scale = round(scale, 10)
            resolution = int(math.ceil(scale)) + 1
            params_in_level = max_params if np.power(resolution, 2) > max_params else np.power(resolution, 2)
            # it is not allowed to exceed the hashmap size
            params_in_level = min(params_in_level, self.hashmap_size)
            params_in_level = next_multiple(params_in_level, 8)

            hashmap_offsets_table[i] = offset
            offset += params_in_level

        hashmap_offsets_table[self.n_levels] = offset
        n_params = int(hashmap_offsets_table[self.n_levels] * self.n_features)
        return hashmap_offsets_table, n_params

    def forward(self, xs_and_ys, num_elements, is_inference=False):
        self.hashmap_offsets_table = cuda.to_device(self.hashmap_offsets_table)
        self.grid_params = cuda.to_device(self.grid_params)

        forward_output = cuda.device_array((num_elements * self.n_levels * self.n_features,))
        xs_and_ys = cuda.to_device(xs_and_ys)

        # execute this function for all pixels given xs_and_ys for each level.
        threads_per_block = 512
        x = div_round_up(num_elements, threads_per_block)
        blocks_hash_grid = [x, self.n_levels, 1]

        if is_inference:
```

```
            backward_output = cuda.device_array(0)
            _forward[blocks_hash_grid, threads_per_block](xs_and_ys, num_elements, self.hashmap_offsets_table,
                                                          self.n_features, self.scale, self.base_resolution,
                                                          self.grid_params, self.n_levels, forward_output,
                                                          self.n_backward_contents, backward_output, 0)
        else:
            backward_output = cuda.device_array((num_elements * self.n_levels * self.n_backward_contents,))
            _forward[blocks_hash_grid, threads_per_block](xs_and_ys, num_elements, self.hashmap_offsets_table,
                                                          self.n_features, self.scale, self.base_resolution,
                                                          self.grid_params, self.n_levels, forward_output,
                                                          self.n_backward_contents, backward_output, 1)
        return forward_output, backward_output

    def update_grid_params(self, num_elements, inputs_grad, encoding_backward_output):
        """
        Backward processes.
        """
        threads_per_block = 512
        x = div_round_up(num_elements, threads_per_block)
        blocks_hash_grid = [x, self.n_levels, 1]

        inputs_grad = cuda.to_device(inputs_grad)
        encoding_backward_output = cuda.to_device(encoding_backward_output)
        sum_updated_grid_params = cuda.device_array((self.n_params,))

        # calculates the sum of grids.
        _sum_update_grid_params[blocks_hash_grid, threads_per_block](num_elements, inputs_grad,
                                                                     encoding_backward_output, self.n_features,
                                                                     sum_updated_grid_params)
        # updates the grid parameters.
        threads_per_block = 512
        x = div_round_up(self.n_params, threads_per_block)
        blocks_hash_grid = [x, 1]
        _update_grid_params[blocks_hash_grid, threads_per_block](self.n_params, sum_updated_grid_params, self.lr,
                                                                 self.grid_params, self.n_features)
```

Listing A.1. encoding.py

```
import torch
import torch.nn as nn
from model_utils import determine_activation

torch.set_printoptions(precision=8)


class Network(nn.Module):
    def __init__(self, activation_name, n_input, n_neurons, n_hidden_layers, n_output):
        super().__init__()
        self.activation = determine_activation(activation_name)
        self.n_input = n_input
        self.n_neurons = n_neurons
        self.n_hidden_layers = n_hidden_layers
        self.n_output = n_output

        self.inputs = None  # we add this variable for differentiation.
        self.inputs_layer = nn.Linear(n_input, n_neurons, bias=False)
        self.hidden_layers = nn.Sequential()
        for _ in range(n_hidden_layers):
            self.hidden_layers.append(nn.Linear(n_neurons, n_neurons, bias=False))
            self.hidden_layers.append(self.activation)
        self.output_layer = nn.Linear(n_neurons, n_output, bias=False)

    def forward(self, x):
        self.inputs = x
        x = self.activation(self.inputs_layer(self.inputs))
        x = self.hidden_layers(x)
```

```
        x = self.output_layer(x)
        return x[:, :3]
```

Listing A.2. network.py

```
import math
import torch
from numba import cuda
import numpy as np
import torch.nn as nn


def determine_activation(activation_name):
    if activation_name == "ReLU":
        return nn.ReLU()
    else:
        print(f"Activation is not defined! --> {activation_name}")
        return None


def determine_optimizer(optimizer_type, network_params, learning_rate):
    if optimizer_type == "Adam":
        optimizer = torch.optim.Adam(network_params, lr=learning_rate)
    else:
        optimizer = None
        print("Optimizer is None!!!")
    return optimizer


def determine_criterion(loss_type):
    if loss_type == "RelativeL2":
        criterion = torch.nn.MSELoss()
    else:
        criterion = None
        print("Loss function is None!!!")
    return criterion


def calculate_xs_and_ys(width, height, n_coords_padded):
    """
    For each pixel, we calculate x and y coordinates (+0.5 for truncating) and normalize them.
    """
    xs_and_ys = np.zeros((n_coords_padded * 2,), dtype=np.float32)
    for y in range(height):
        for x in range(width):
            idx = (y * width + x) * 2
            xs_and_ys[idx] = float(x + 0.5) / float(width)
            xs_and_ys[idx + 1] = float(y + 0.5) / float(height)
    return xs_and_ys


def div_round_up(val, div):
    return int(((val + div - 1) / div))


def next_multiple(val, div):
    """
    For using GPU efficiently
    """
    dru = div_round_up(val, div)
    res = int(dru * div)
    return res


@cuda.jit(inline=True)
def determine_xy0(xi, yi, width, height):
```

```
    if xi < 0:
        x0 = 0
    elif xi > width - 1:
        x0 = width - 1
    else:
        x0 = xi

    if yi < 0:
        y0 = 0
    elif yi > height - 1:
        y0 = height - 1
    else:
        y0 = yi

    return int(x0), int(y0)


@cuda.jit(inline=True)
def determine_xy1(x0, y0, width, height):
    if x0 + 1 < 0:
        x1 = 0
    elif x0 + 1 > width - 1:
        x1 = width - 1
    else:
        x1 = x0 + 1

    if y0 + 1 < 0:
        y1 = 0
    elif y0 + 1 > height - 1:
        y1 = height - 1
    else:
        y1 = y0 + 1
    return int(x1), int(y1)


@cuda.jit(inline=True)
def fast_hash(x, y, hashmap_size):
    index = 0
    index ^= x * 1
    index ^= y * 2654435761
    index = (index % hashmap_size)
    return int(index)


@cuda.jit(inline=True)
def calculate_filter_mode_linear(o1, o2, o3, o4, lwx, lwy):
    return (o1 * (1.0 - lwx) * (1.0 - lwy) +
            o2 * lwx * (1.0 - lwy) +
            o3 * (1.0 - lwx) * lwy +
            o4 * lwx * lwy)


@cuda.jit
def relu(v):
    if v > 0.0:
        return v
    else:
        return 0.0


@cuda.jit()
def _forward(xs_and_ys, n_elements, hashmap_offsets_table, n_features, log2_per_level_scale, base_resolution,
             grid_params, n_levels, forward_output, n_backward_contents, backward_output, derivative):
    i = cuda.blockIdx.x * cuda.blockDim.x + cuda.threadIdx.x
    level = cuda.blockIdx.y  # < - the level is the same for all threads
    if i >= n_elements or level >= n_levels:
```

```
        return

    grid_index = int(hashmap_offsets_table[level] * n_features)
    hashmap_size = int(hashmap_offsets_table[level + 1]) - int(hashmap_offsets_table[level])

    scale = math.pow(2, level * math.log2(log2_per_level_scale)) * base_resolution - 1.0
    scale = round(scale, 10)
    resolution = int(math.ceil(scale)) + 1

    pos_x = xs_and_ys[i * 2] * scale + 0.5
    pos_y = xs_and_ys[i * 2 + 1] * scale + 0.5

    pos_grid_x = math.floor(pos_x)
    pos_grid_y = math.floor(pos_y)

    lerp_weight_x = pos_x - pos_grid_x
    lerp_weight_y = pos_y - pos_grid_y

    x0, y0 = determine_xy0(pos_grid_x, pos_grid_y, resolution, resolution)
    x1, y1 = determine_xy1(x0, y0, resolution, resolution)
    i1 = x0 + y0 * resolution
    i2 = x1 + y0 * resolution
    i3 = x0 + y1 * resolution
    i4 = x1 + y1 * resolution

    if math.pow(resolution, 2) > hashmap_size:
        i1 = fast_hash(x0, y0, hashmap_size)
        i2 = fast_hash(x1, y0, hashmap_size)
        i3 = fast_hash(x0, y1, hashmap_size)
        i4 = fast_hash(x1, y1, hashmap_size)

    for f in range(n_features):
        g1 = grid_params[grid_index + (i1 * n_features) + f]
        g2 = grid_params[grid_index + (i2 * n_features) + f]
        g3 = grid_params[grid_index + (i3 * n_features) + f]
        g4 = grid_params[grid_index + (i4 * n_features) + f]

        features = calculate_filter_mode_linear(g1, g2, g3, g4, lerp_weight_x, lerp_weight_y)

        forward_output[(i * n_features * n_levels) + (level * n_features) + f] = features

    if derivative:
        d1 = (1.0 - lerp_weight_x) * (1.0 - lerp_weight_y)
        d2 = lerp_weight_x * (1.0 - lerp_weight_y)
        d3 = (1.0 - lerp_weight_x) * lerp_weight_y
        d4 = lerp_weight_x * lerp_weight_y

        backward_output[(i * n_levels * n_backward_contents) + (level * n_backward_contents) + 0] = i1
        backward_output[(i * n_levels * n_backward_contents) + (level * n_backward_contents) + 1] = i2
        backward_output[(i * n_levels * n_backward_contents) + (level * n_backward_contents) + 2] = i3
        backward_output[(i * n_levels * n_backward_contents) + (level * n_backward_contents) + 3] = i4

        backward_output[(i * n_levels * n_backward_contents) + (level * n_backward_contents) + 4] = grid_index

        backward_output[(i * n_levels * n_backward_contents) + (level * n_backward_contents) + 5] = d1
        backward_output[(i * n_levels * n_backward_contents) + (level * n_backward_contents) + 6] = d2
        backward_output[(i * n_levels * n_backward_contents) + (level * n_backward_contents) + 7] = d3
        backward_output[(i * n_levels * n_backward_contents) + (level * n_backward_contents) + 8] = d4


@cuda.jit()
def _sum_update_grid_params(n_elements, inputs_grad, encoding_backward_output, n_features, sum_updated_grid_params):
    i = cuda.blockIdx.x * cuda.blockDim.x + cuda.threadIdx.x
    if i >= n_elements:
        return
    l = cuda.blockIdx.y
```

```
    i1, i2, i3, i4, grid_index, d1, d2, d3, d4 = encoding_backward_output[i][l]

    for f in range(n_features):
        input_grad = inputs_grad[i][l * n_features + f]

        sum_updated_grid_params[int(grid_index + (i1 * n_features) + f)] += input_grad * d1
        sum_updated_grid_params[int(grid_index + (i2 * n_features) + f)] += input_grad * d2
        sum_updated_grid_params[int(grid_index + (i3 * n_features) + f)] += input_grad * d3
        sum_updated_grid_params[int(grid_index + (i4 * n_features) + f)] += input_grad * d4


@cuda.jit()
def _update_grid_params(n_elements, sum_updated_grid_params, lr, grid_params, n_features):
    i = cuda.blockIdx.x * cuda.blockDim.x + cuda.threadIdx.x
    if i >= n_elements:
        return
    grid_params[i] -= (sum_updated_grid_params[i] / n_features) * lr
```

Listing A.3. model_utils.py

```
import numpy as np
import torch
from image_utils import write_image


def determine_samples(batch_size, n_coords, img):
    xs_and_ys_indices = []
    gt = []
    h, w = img.shape[:2]
    for bi in range(batch_size):
        r = np.random.randint(n_coords)
        xs_and_ys_indices.append(r * 2)
        xs_and_ys_indices.append(r * 2 + 1)
        gt.append(img[int(r // w), int(r % w)])
    return xs_and_ys_indices, np.asarray(gt)


class Trainer:
    def __init__(self, encoding, network, optimizer, criterion, n_epochs,
                       img, xs_and_ys, n_coords, batch_size, output_folder):
        self.encoding = encoding
        self.network = network
        self.optimizer = optimizer
        self.criterion = criterion
        self.n_epochs = n_epochs
        self.img = img
        self.xs_and_ys = xs_and_ys
        self.n_coords = n_coords
        self.batch_size = 1 << batch_size
        self.output_folder = output_folder

    def train(self):
        n_levels = self.encoding.n_levels
        n_features = self.encoding.n_features
        self.network.train().cuda()
        # we only select pixels as many as the number of batch size for each epoch.
        for epoch in range(1, self.n_epochs+1):
            ts = timer()
            xs_and_ys_indices, gt = determine_samples(self.batch_size, self.n_coords, self.img)
            inputs_xs_and_ys = self.xs_and_ys[xs_and_ys_indices]

            encoding_forward_output, encoding_backward_output = self.encoding.forward(inputs_xs_and_ys, self.
     batch_size)
            encoding_forward_output = encoding_forward_output.copy_to_host()
            encoding_backward_output = encoding_backward_output.copy_to_host()
            # reshape the outputs of the encoding to give them to the network.
```

```
        encoding_forward_output = encoding_forward_output.reshape(-1, n_levels * n_features).astype(np.float32)
        encoding_backward_output = encoding_backward_output.reshape(-1, n_levels, self.encoding.
 n_backward_contents).astype(np.float32)

        self.optimizer.zero_grad()

        encoding_outputs = torch.from_numpy(encoding_forward_output).cuda()
        gt = torch.from_numpy(gt).cuda()
        # to update the grid params, we need to derivatives of the input layer.
        encoding_outputs.requires_grad = True

        network_outputs = self.network(encoding_outputs)
        loss = self.criterion(network_outputs, gt)
        loss.backward()

        inputs_grad = self.network.inputs.grad
        # backward for grids
        self.encoding.update_grid_params(self.batch_size, inputs_grad, encoding_backward_output)
        self.optimizer.step()
        te = timer()

        if epoch % 100 == 0:
            print(f"Epoch {epoch} | # of samples: {self.batch_size} | Time: {te - ts: .2f}")
            self.inference(name=str(epoch))

def inference(self, name=""):
    n_levels = self.encoding.n_levels
    n_features = self.encoding.n_features
    h, w = self.img.shape[:2]
    self.network.eval()
    with torch.no_grad():
        encoding_forward_outputs, _ = self.encoding.forward(self.xs_and_ys, self.n_coords, is_inference=True)
        encoding_forward_outputs = encoding_forward_outputs.copy_to_host()

        encoding_forward_outputs = encoding_forward_outputs.reshape(-1, n_levels * n_features).astype(np.float32)
        encoding_outputs = torch.from_numpy(encoding_forward_outputs).cuda()

        network_outputs = []
        for b in range(0, encoding_outputs.size(0), self.batch_size):
            network_output = self.network(encoding_outputs[b:b+self.batch_size])
            network_outputs.append(network_output.cpu().detach().numpy())

        network_outputs = np.concatenate(network_outputs, axis=0)

        output_img = np.reshape(network_outputs, (h, w, 3))
        write_image(f"{self.output_folder}/inference_{name}.jpg", output_img)
    print("Inference is done.")
```

Listing A.4. train.py

# APPENDIX B

# PYTHON CODE FOR MULTI-FRAME SUPER-RESOLUTIN WITHOUT PRIORS

```
import cv2
import numpy as np
from scipy.optimize import least_squares

sift = cv2.SIFT_create()
bf = cv2.BFMatcher()

def func(mat, x0, x1):
    mat = np.reshape(mat, (2, 3))
    x_prime = mat @ x0.T
    return np.sum((x1 - x_prime.T)**2, axis=1)

def reject_outliers(data, m=6.):
    d = np.abs(data - np.median(data))
    mdev = np.median(d)
    s = d / (mdev if mdev else 1.)
    return data[s < m]

def cal_affine_matrix(img1, img2):
    img1 = cv2.cvtColor(img1, cv2.COLOR_BGR2GRAY)
    kps1, descs1 = sift.detectAndCompute(img1, None)
    descs1 /= (descs1.sum(axis=1, keepdims=True) + 1e-7)
    descs1 = np.sqrt(descs1)
    img2 = cv2.cvtColor(img2, cv2.COLOR_BGR2GRAY)
    kps2, descs2 = sift.detectAndCompute(img2, None)
    descs2 /= (descs2.sum(axis=1, keepdims=True) + 1e-7)
    descs2 = np.sqrt(descs2)

    matches = bf.knnMatch(descs1, descs2, k=2)

    slctd = []
    rt = 0.2
    for _ in range(6):
        good = []
        for m,n in matches:
            if m.distance < (rt * n.distance):
                good.append(m)
        if len(good) < 10:
            rt += 0.05
        else:
            slctd = [elem for elem in good]
            break

    if len(slctd) < 6:
        raise Exception(f"Number of matched keypoints is {len(slctd)}, less than 6!")

    matches = sorted(slctd, key=lambda x : x.distance)

    m_kps1 = []
    m_kps2 = []
    for m in matches:
        m1 = m.queryIdx
```

```
        m2 = m.trainIdx
        m_kps1.append(kps1[m1].pt)
        m_kps2.append(kps2[m2].pt)

    m_kps1 = np.asarray(m_kps1, dtype=np.float32)
    m_kps2 = np.asarray(m_kps2, dtype=np.float32)

    # gets 3 keypoints having the minimum distance
    warp_mat = cv2.getAffineTransform(m_kps2[:3], m_kps1[:3])
    warp_mat = warp_mat.flatten()
    n, _ = m_kps1.shape
    ones = np.ones((n,1))
    m_kps2 = np.hstack((m_kps2, ones))

    warp_mat = least_squares(func, warp_mat, args=(m_kps2, m_kps1), method="lm")
    warp_mat = np.reshape(warp_mat.x, (2, 3))
    return warp_mat
```

Listing B.1. calculate_affine_matrix.py

```
import random
from tqdm import tqdm
import cv2
import numpy as np
import torch
from numba import cuda
from utils.image_utils import write_image
from utils.model_utils import div_round_up, _determine_input_samples_cuda, _determine_gt_samples_cuda

random.seed(42)
np.random.seed(42)
torch.manual_seed(42)
torch.cuda.manual_seed(42)


class Trainer:
    def __init__(self, encoding, network, optimizer, criterion, n_epochs, xs_and_ys,
                 n_coords, batch_size, scale_factor, width, height, imgs,
                 interpolation_type, result_dir_path):
        self.interpolation_dict = {"bilinear": 0, "bicubic": 1}
        self.encoding = encoding
        self.network = network.cuda()
        self.optimizer = optimizer
        self.scheduler = torch.optim.lr_scheduler.StepLR(self.optimizer, 100000, gamma=0.8)
        self.criterion = criterion
        self.n_epochs = n_epochs
        self.imgs = imgs
        self.interpolation_type = self.interpolation_dict[interpolation_type]
        self.scale_factor = scale_factor


        self.xs_and_ys = xs_and_ys
        self.n_coords = n_coords
        self.batch_size = 1 << batch_size
        self.width = width
        self.height = height
        self.result_dir_path = result_dir_path


    def train(self):
        ...
        translated_random_coords_x = []
        translated_random_coords_y = []

        for img, t in self.imgs:
            random_coords_y_ = np.full((encoding_outputs.size(0),), t[1])
```

```
            random_coords_x_ = np.full((encoding_outputs.size(0),), t[0])

            translated_random_coords_x.append(random_coords_x_)
            translated_random_coords_y.append(random_coords_y_)

        translated_random_coords_x = np.asarray(translated_random_coords_x, dtype=np.float32)
        translated_random_coords_y = np.asarray(translated_random_coords_y, dtype=np.float32)

        translated_random_coords_x = torch.from_numpy(translated_random_coords_x).cuda()
        translated_random_coords_y = torch.from_numpy(translated_random_coords_y).cuda()

        # the outputs convolutions with size of batch
        network_outputs = self.network(encoding_outputs, translated_random_coords_x, translated_random_coords_y, w, h)

        gts_torch = []
        for i, gt in enumerate(gts):
            gt = gt.copy_to_host()
            gt = gt.reshape(-1, 3).astype(np.float32)
            gts_torch.append(gt)

        gts_torch = np.asarray(gts_torch, dtype=np.float32)
        gts_torch = torch.from_numpy(gts_torch).cuda()

        relative_l2_error = torch.abs(gts_torch.to(network_outputs.dtype) - network_outputs) / (network_outputs.detach
    () + 0.01)
        loss = relative_l2_error.mean()
        ...

    def inference(self, name=""):
        ...
        center_x = torch.full((1, encoding_outputs.size(0)), 0.0).cuda()
        center_y = torch.full((1, encoding_outputs.size(0)), 0.0).cuda()
        network_output = self.network(encoding_outputs, center_x, center_y, w, h)
        network_output = network_output.detach().cpu().numpy()[0]
        ...

    def _determine_samples(self, random_coords):
        for i in range(len(self.imgs) - 1):
            assert self.imgs[i][0].shape == self.imgs[i+1][0].shape, "Source images must be the same dimensionality."

        random_coords = cuda.to_device(random_coords)

        h, w = self.imgs[0][0].shape[:2]

        xs_and_ys = cuda.to_device(self.xs_and_ys)

        inputs_xs_and_ys = cuda.device_array((self.batch_size * 2,))

        threads_per_block = 32
        x = div_round_up(self.batch_size, threads_per_block)
        blocks_hash_grid = [x, 1]
        _determine_input_samples_cuda[blocks_hash_grid, threads_per_block](self.batch_size, random_coords,
                                                                           xs_and_ys, inputs_xs_and_ys)
        inputs_xs_and_ys = inputs_xs_and_ys.copy_to_host()

        gts = []

        for img, t in self.imgs:
            gt = cuda.device_array((self.batch_size * 3))
            img = cuda.to_device(img)

            threads_per_block = 16
            x = div_round_up(self.batch_size, threads_per_block)
            blocks_hash_grid = [x, 1]
            _determine_gt_samples_cuda[blocks_hash_grid, threads_per_block](self.batch_size, random_coords,
                                                                            self.scale_factor, w, h, gt, img, t[0], t
```

```
    [1],
                                                                    self.interpolation_type)

            gt.copy_to_host()
            gts.append(gt)
        return inputs_xs_and_ys, gts
```

Listing B.2. sr_train.py

```
import math
import torch
import torch.nn as nn
from utils.model_utils import determine_activation, determine_xy0, determine_xy1, calculate_filter_mode_linear
import torch.nn.functional as F

torch.set_printoptions(precision=8)
torch.manual_seed(42)
torch.cuda.manual_seed(42)


class Bilinear(torch.autograd.Function):
    @staticmethod
    def forward(ctx, convs, ixs, iys, width, height):
        n_imgs = ixs.size(0)

        # translation must be at most 0.5 or at least -0.5
        ixs = (ixs + 0.5).clamp(0.0, 1.0)
        iys = (iys + 0.5).clamp(0.0, 1.0)

        pos_grid_x = torch.div(ixs, 1, rounding_mode="floor")
        pos_grid_y = torch.div(iys, 1, rounding_mode="floor")

        lwx = ixs - pos_grid_x
        lwy = iys - pos_grid_y

        b =  convs[:, 0, 0, 0] * (1.0 - lwx) * (1.0 - lwy) + \
             convs[:, 0, 0, 1] * lwx * (1.0 - lwy) + \
             convs[:, 0, 1, 0] * (1.0 - lwx) * lwy + \
             convs[:, 0, 1, 1] * lwx * lwy

        g =  convs[:, 1, 0, 0]  * (1.0 - lwx) * (1.0 - lwy) + \
             convs[:, 1, 0, 1]  * lwx * (1.0 - lwy) + \
             convs[:, 1, 1, 0]  * (1.0 - lwx) * lwy + \
             convs[:, 1, 1, 1]  * lwx * lwy

        r =  convs[:, 2, 0, 0]  * (1.0 - lwx) * (1.0 - lwy) + \
             convs[:, 2, 0, 1]  * lwx * (1.0 - lwy) + \
             convs[:, 2, 1, 0]  * (1.0 - lwx) * lwy + \
             convs[:, 2, 1, 1]  * lwx * lwy
        results = torch.stack((b, g, r), -1)

        w1 = (1.0 - lwx) * (1.0 - lwy)
        w2 = lwx * (1.0 - lwy)
        w3 = (1.0 - lwx) * lwy
        w4 = lwx * lwy

        w1 = torch.stack((w1, w1, w1), -1)
        w2 = torch.stack((w2, w2, w2), -1)
        w3 = torch.stack((w3, w3, w3), -1)
        w4 = torch.stack((w4, w4, w4), -1)

        w = torch.stack((w1, w2, w3, w4), -1).view(n_imgs, -1, 3, 2, 2)
        ctx.save_for_backward(w)
        return results

    @staticmethod
```

```
    def backward(ctx, grad_output):
        """
        In the backward pass we receive a Tensor containing the gradient of the loss
        with respect to the output, and we need to compute the gradient of the loss
        with respect to the input.
        """
        w, = ctx.saved_tensors
        w[:, :, 0, 0, 0] = grad_output[:, :,  0] * w[:, :, 0, 0, 0]
        w[:, :, 0, 0, 1] = grad_output[:, :,  0] * w[:, :, 0, 0, 1]
        w[:, :, 0, 1, 0] = grad_output[:, :,  0] * w[:, :, 0, 1, 0]
        w[:, :, 0, 1, 1] = grad_output[:, :,  0] * w[:, :, 0, 1, 1]

        w[:, :, 1, 0, 0] = grad_output[:, :,  1] * w[:, :, 1, 0, 0]
        w[:, :, 1, 0, 1] = grad_output[:, :,  1] * w[:, :, 1, 0, 1]
        w[:, :, 1, 1, 0] = grad_output[:, :,  1] * w[:, :, 1, 1, 0]
        w[:, :, 1, 1, 1] = grad_output[:, :,  1] * w[:, :, 1, 1, 1]

        w[:, :, 2, 0, 0] = grad_output[:, :,  2] * w[:, :, 2, 0, 0]
        w[:, :, 2, 0, 1] = grad_output[:, :,  2] * w[:, :, 2, 0, 1]
        w[:, :, 2, 1, 0] = grad_output[:, :,  2] * w[:, :, 2, 1, 0]
        w[:, :, 2, 1, 1] = grad_output[:, :,  2] * w[:, :, 2, 1, 1]
        w = torch.sum(w, dim=0)
        return w, None, None, None, None


class Network(nn.Module):
    def __init__(self, activation_name, n_input, n_neurons, n_hidden_layers):
        super().__init__()
        self.activation = determine_activation(activation_name)
        self.n_input = n_input
        self.n_neurons = n_neurons
        self.n_hidden_layers = n_hidden_layers
        self.n_output =  2 * 2 * 3

        self.inputs = None  # we add this variable for differentiation.
        self.inputs_layer = nn.Linear(n_input, n_neurons, bias=False)# bias=True)
        self.hidden_layers = nn.Sequential()
        #self.dropout = nn.Dropout(p=0.2)

        for _ in range(n_hidden_layers):
            self.hidden_layers.append(nn.Linear(n_neurons, n_neurons, bias=False))# bias=True)
            self.hidden_layers.append(self.activation)

        self.output_layer = nn.Linear(self.n_neurons, self.n_output, bias=False) # bias=True)
        self.bilinear = Bilinear.apply

    def forward(self, x, translated_batched_coords_x, translated_batched_coords_y, width, height):
        self.inputs = x
        x = self.inputs_layer(self.inputs)
        x = self.activation(x)
        x = self.hidden_layers(x)
        x = self.output_layer(x)
        x = self.activation(x)
        convs = x.view(-1, 3, 2, 2)
        results = self.bilinear(convs, translated_batched_coords_x, translated_batched_coords_y, width, height)
        return results
```

Listing B.3. sr_network.py

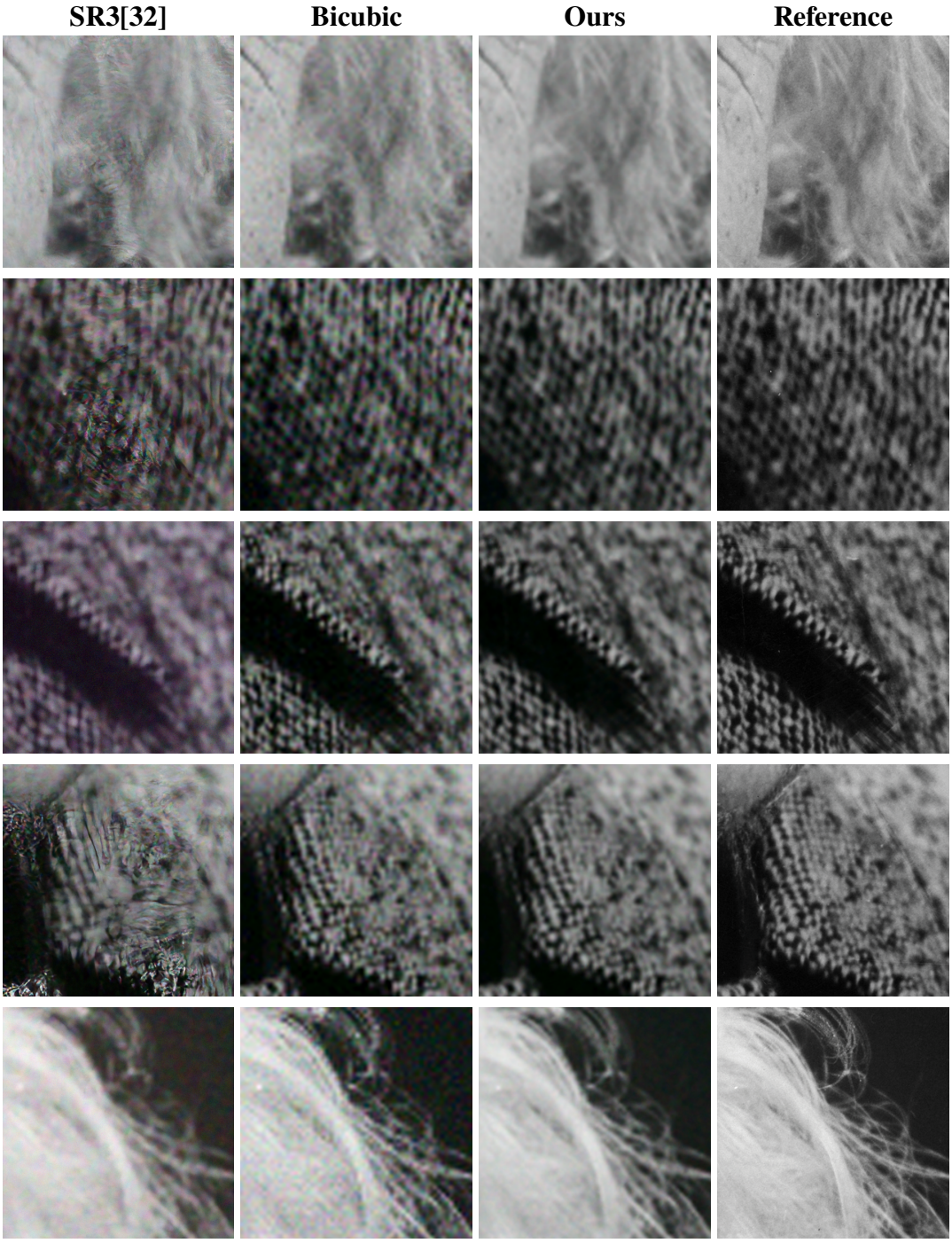
```
...
@cuda.jit(inline=True)
def _determine_input_samples_cuda(batch_size, random_coords, xs_and_ys, inputs_xs_and_ys):
    i = cuda.blockIdx.x * cuda.blockDim.x + cuda.threadIdx.x
    if i >= batch_size:
        return
```

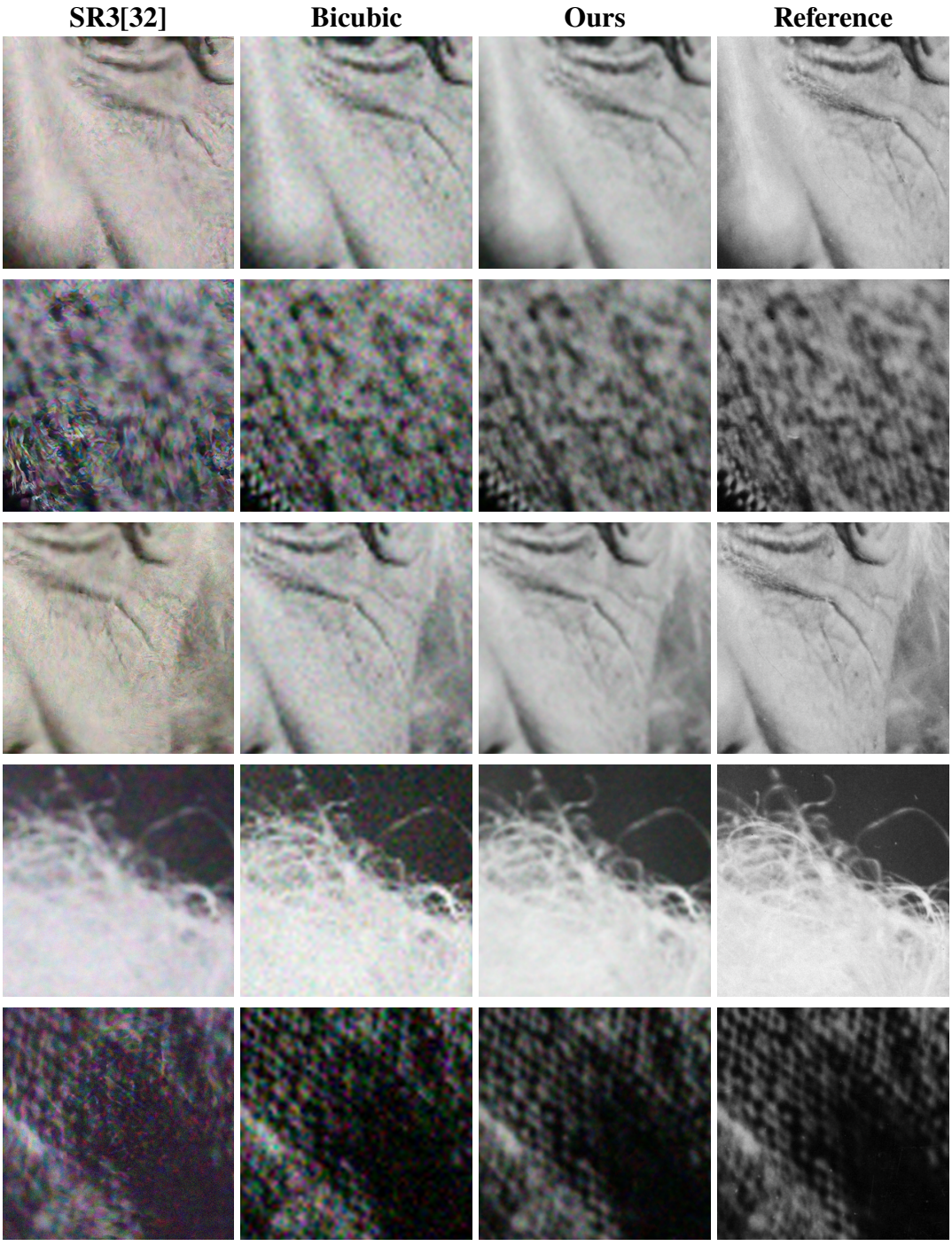
```
    r = random_coords[i]

    inputs_xs_and_ys[i * 2] = xs_and_ys[r * 2]
    inputs_xs_and_ys[i * 2 + 1] = xs_and_ys[r * 2 + 1]


@cuda.jit(inline=True)
def _determine_gt_samples_cuda(batch_size, random_coords, scale_factor, w, h, gt, img, tx, ty, interpolation_type):
    i = cuda.blockIdx.x * cuda.blockDim.x + cuda.threadIdx.x
    if i >= batch_size:
        return
    r = random_coords[i]

    iy = (r // w)
    ix = (r %  w)

    gt[i * 3 + 0] = img[iy][ix][0]
    gt[i * 3 + 1] = img[iy][ix][1]
    gt[i * 3 + 2] = img[iy][ix][2]
...
```

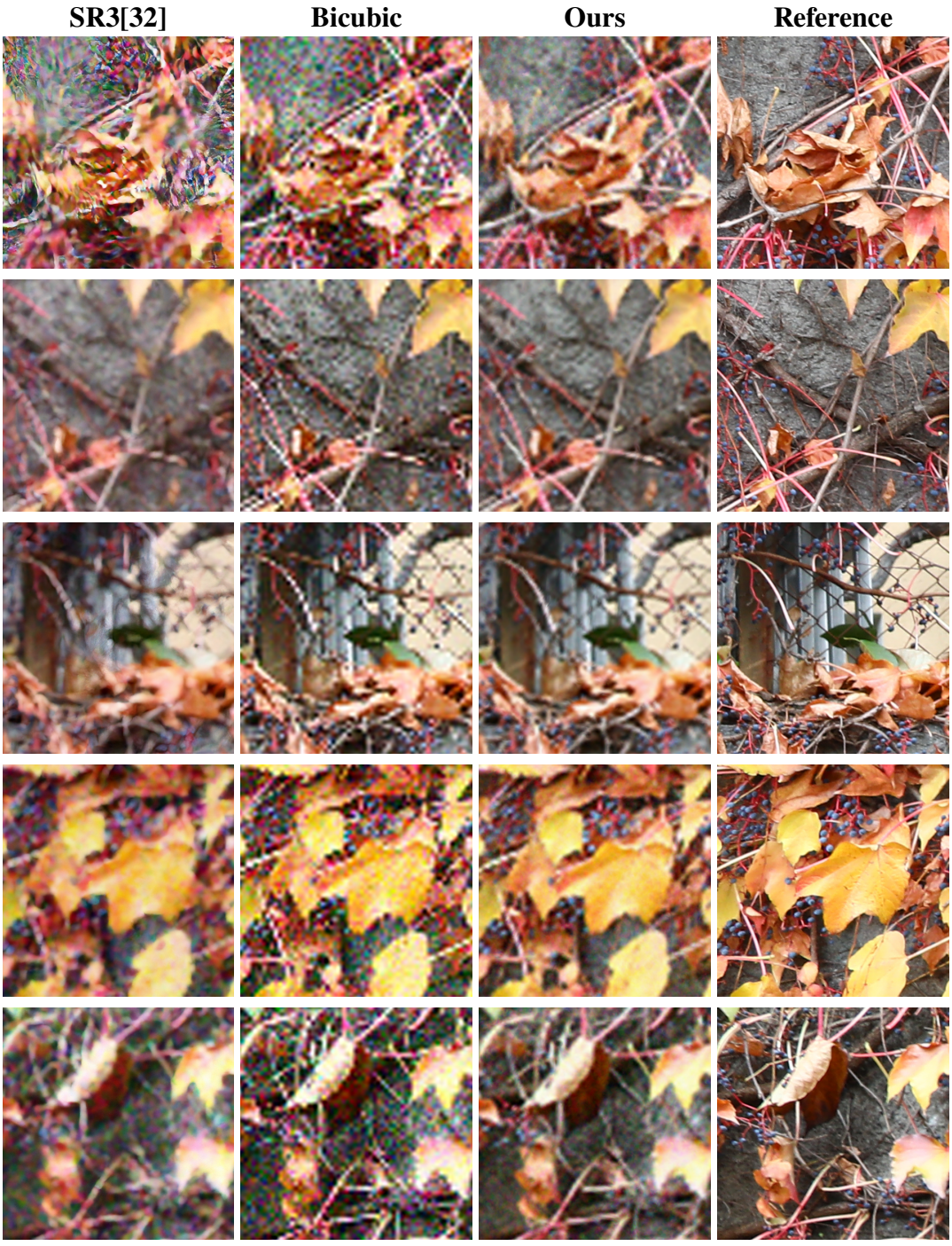
Listing B.4. sr_model_utils.py

# APPENDIX C

# ADDITIONAL RESULTS OF MULTI-FRAME SUPER-RESOLUTION WITHOUT PRIORS


SR3[32]
Bicubic
Ours
Reference


SR3[32]
Bicubic
Ours
Reference


SR3[32]
Bicubic
Ours
Reference

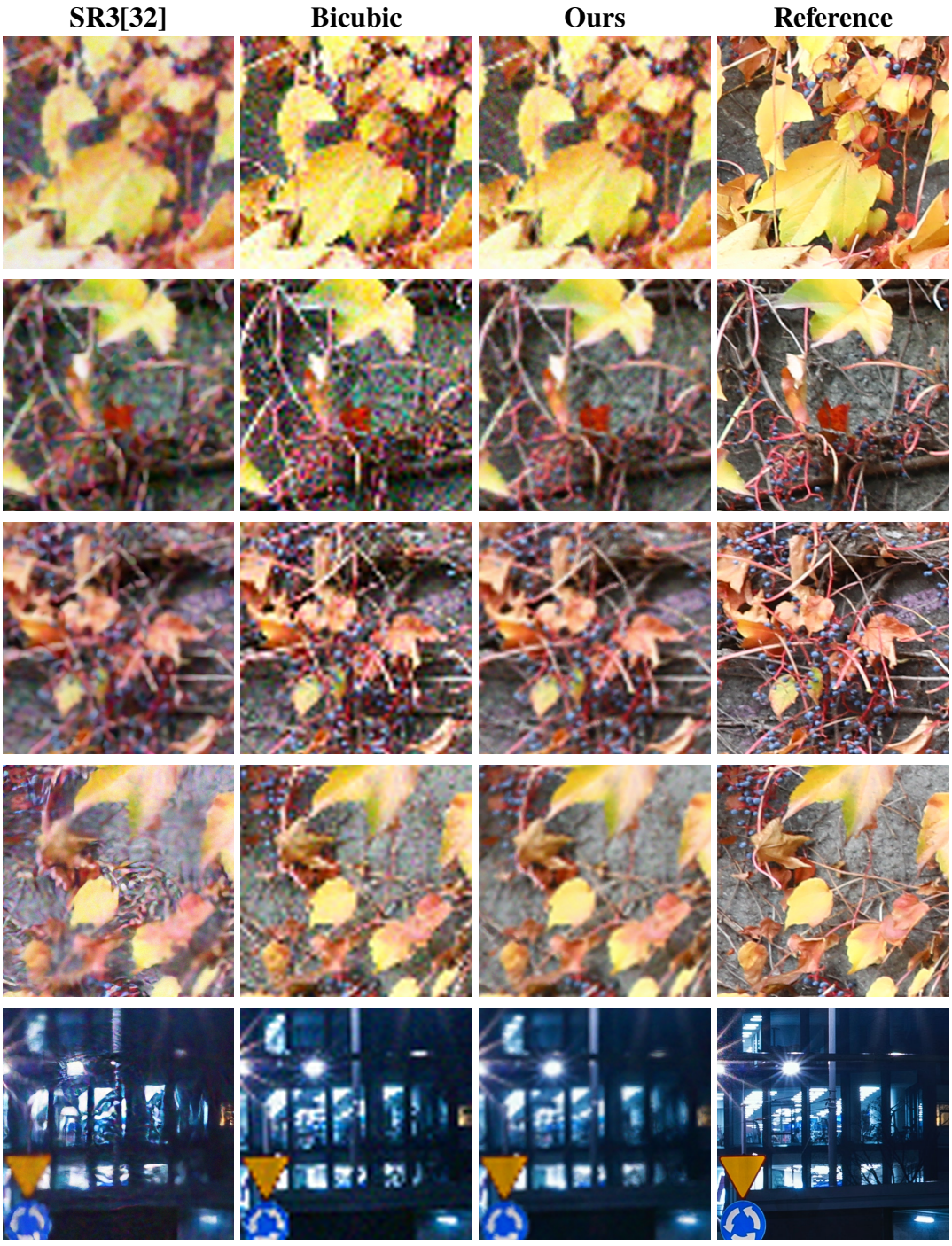
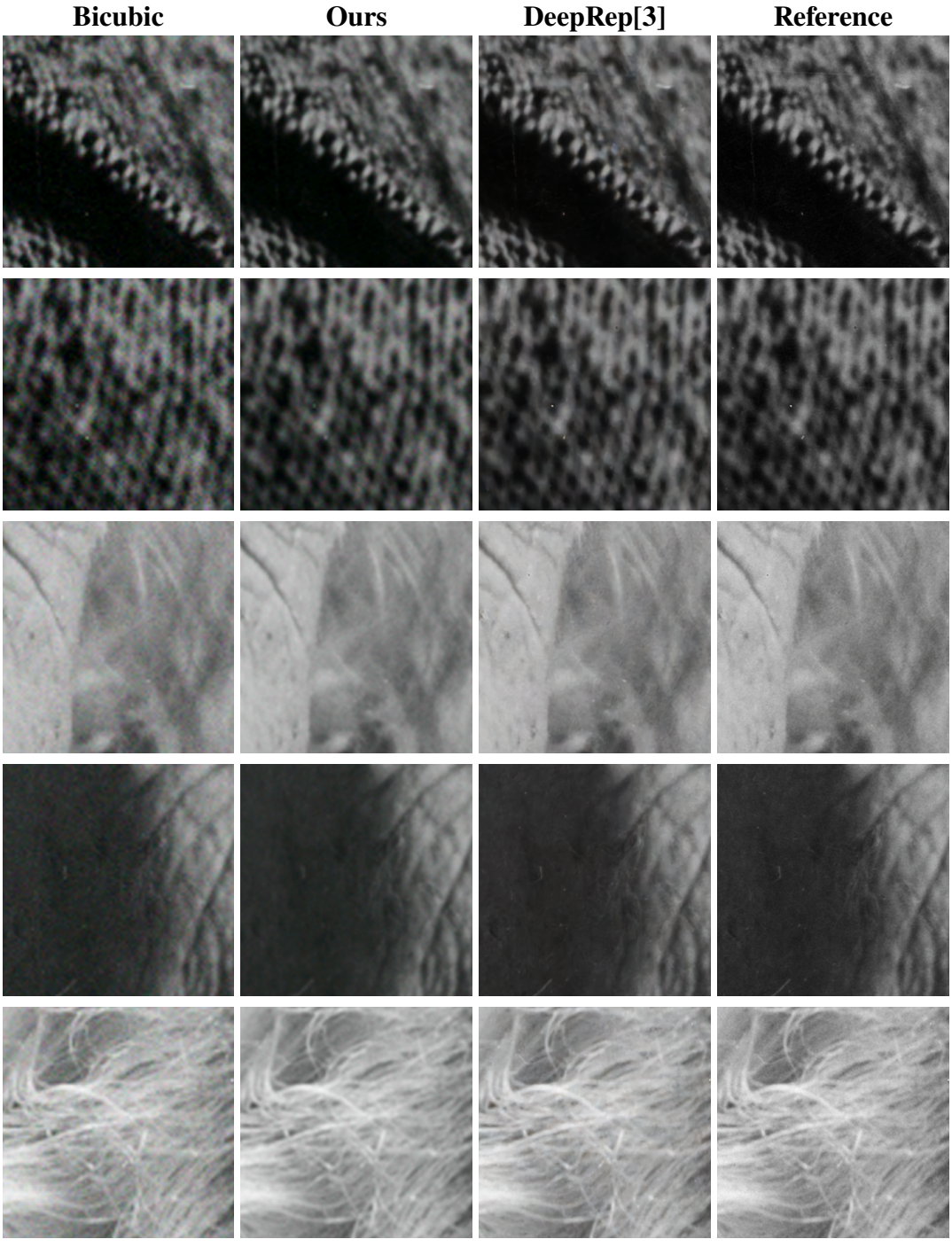
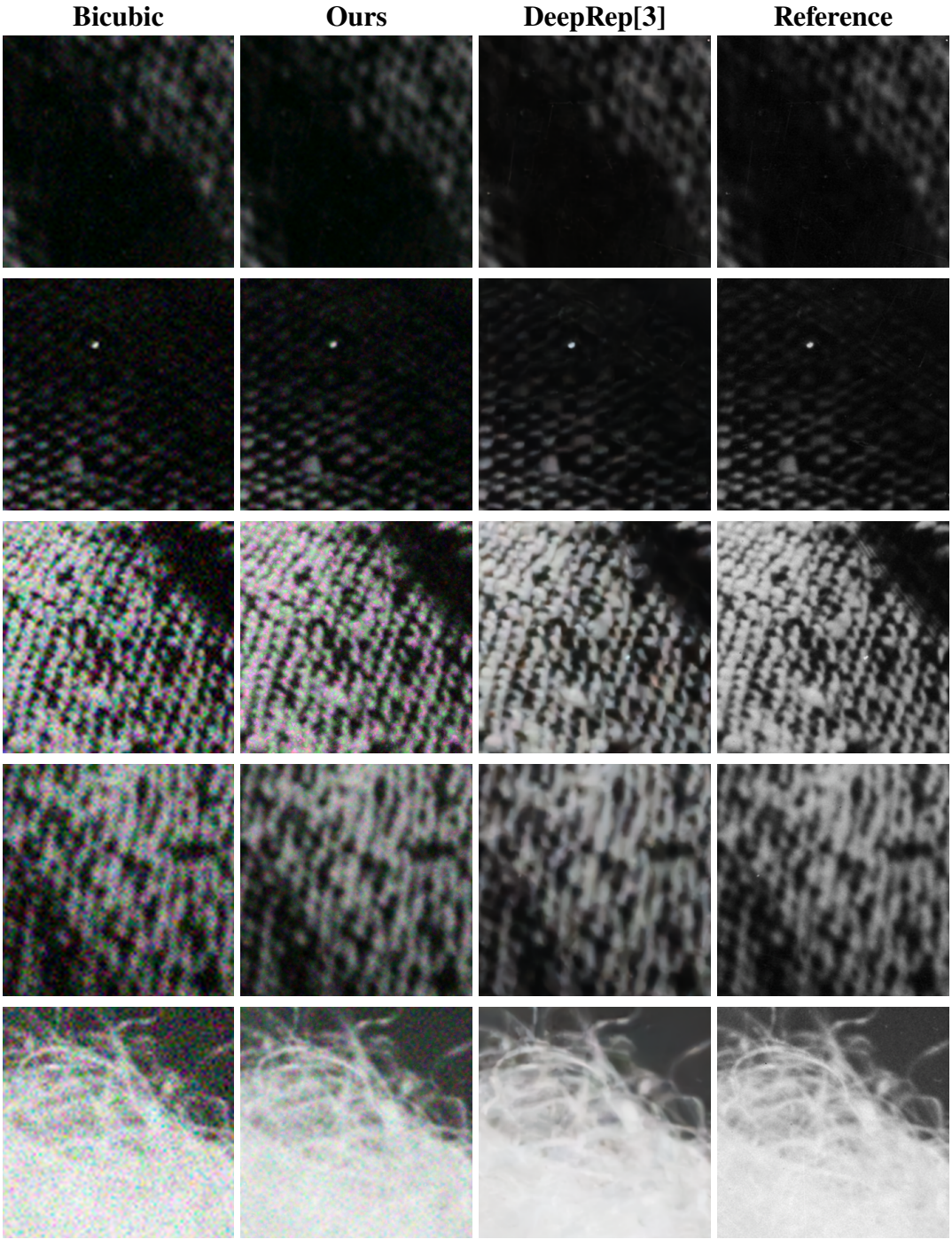
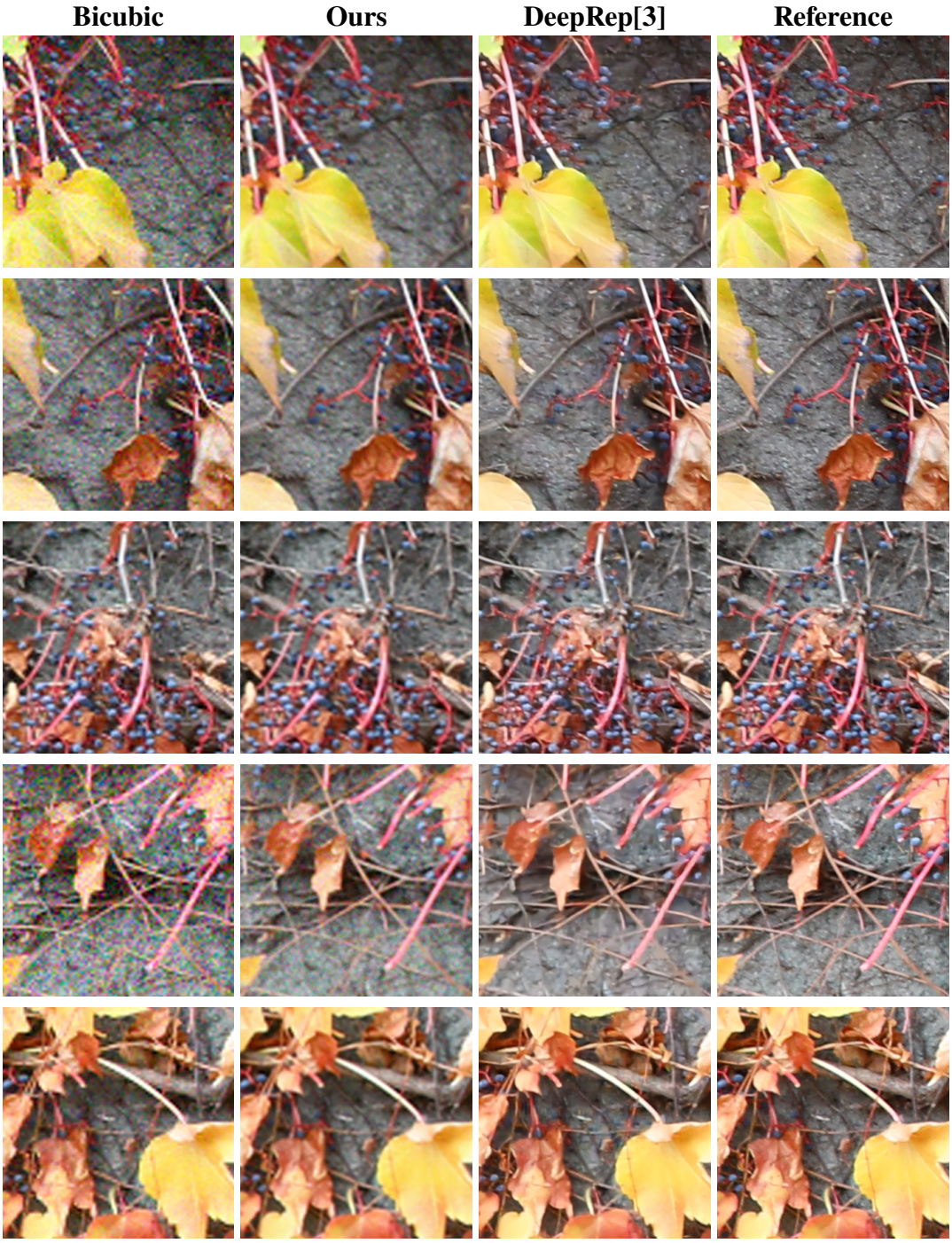



Figure C.1. The additional results of bicubic interpolation, SR3[32] and ours.


Bicubic
Ours
DeepRep[3]
Reference


Bicubic
Ours
DeepRep[3]
Reference


Bicubic
Ours
DeepRep[3]
Reference

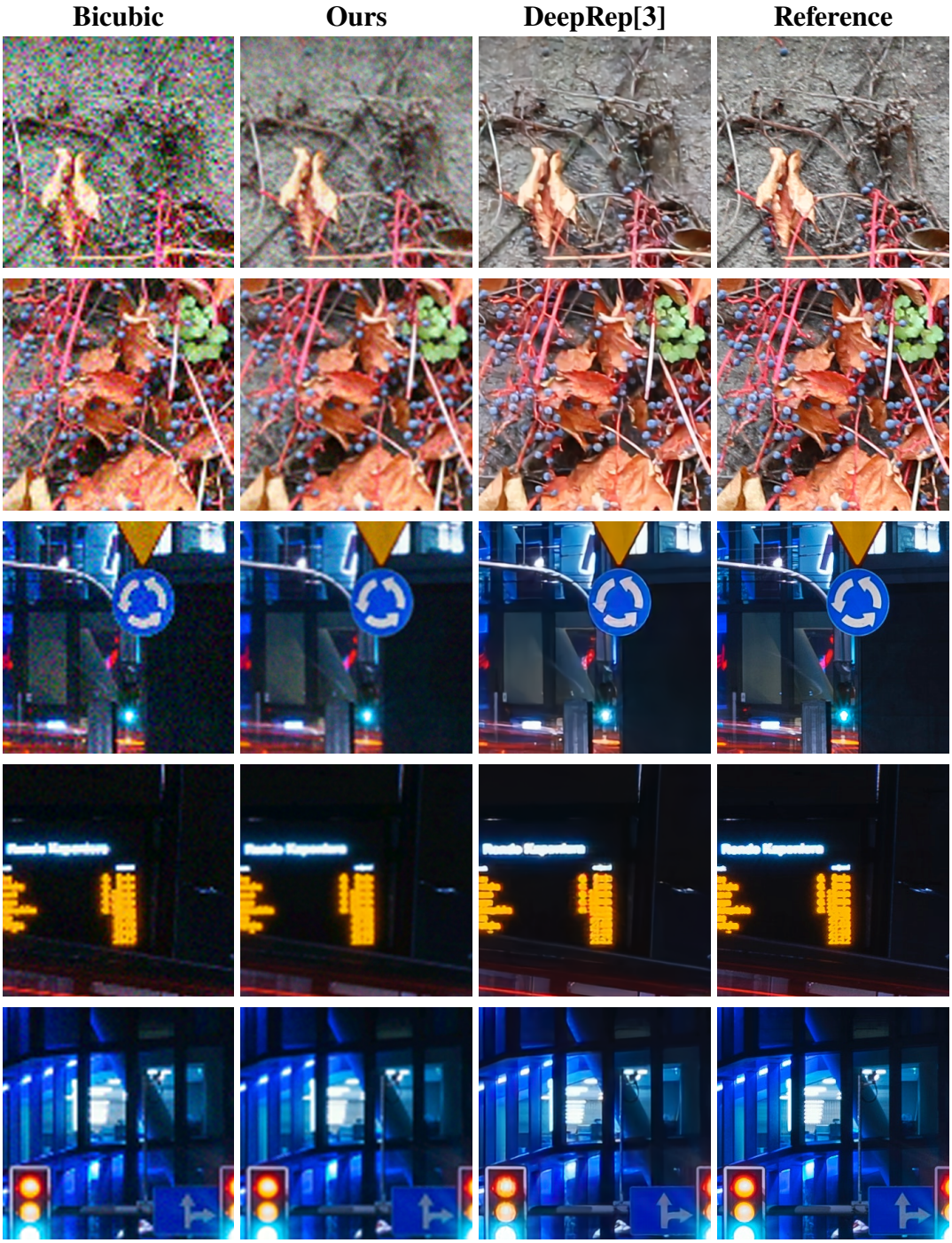



Figure C.2. The additional results of bicubic interpolation, DeepRep[3] and ours.