Helwan University

Faculty of Computers and Artificial Intelligence, Computer Science Department

[Image Restoration App]

A Graduation Project dissertation by:

Ahmed Alaa Mohamed Elmahdy (202000053)

Omnia Arafat Fahmy Soliman (202000155)

Fatma Sherif Ahmed Attia (202000637)

Mayar Ahmed Elsayed Farag Alaa (202000956)

Youssef Mostafa Mohamed (202001109)

Yousef Hesham Mohamed Ali (202001113)

Submitted as partial fulfillment of the requirements for the Bachelor of Science degree in Computer Science and Artificial Intelligence, at the Faculty of Computers and Artificial Intelligence, Computer Science Department, the faculty of Computers and Artificial Intelligence, Helwan university.

Supervised by:

Dr. Wessam El Behaidy

June 2024

Acknowledgement

We would like to extend our heartfelt gratitude and appreciation to all those who have contributed to the successful completion of our graduation project. First and foremost, we express our deepest gratitude to our advisor, Dr. Wessam El-Behaidy, for her unwavering support, exceptional guidance, and constant encouragement throughout our project journey.

Additionally, we would like to extend our sincere thanks to our families, particularly our parents, for their unwavering encouragement, patience, and assistance over the years. Their love, support, and belief in our abilities have been a driving force behind our achievements. We are forever indebted to our parents, who have not only provided us with the necessary resources but have also kept us in their prayers and consistently pushed us towards success.

Lastly, we express our deepest gratitude to everyone who has played a part in our graduation project. Without your support, encouragement, and guidance, this achievement would not have been possible. Thank you all for being a part of our journey and for helping us reach this milestone in our academic endeavors.

Content

Abstract

In today's digital age, the quality and clarity of images are paramount across various applications such as medical imaging, surveillance, and photography. However, images frequently suffer from imperfections including missing regions, noise, and blurriness, which degrade their visual fidelity and usability. These imperfections can obscure important details, mislead interpretations, and diminish the overall impact of the images. Addressing these challenges is crucial to maximizing the effectiveness of visual data. To this end, we propose a comprehensive approach utilizing multi-modal image restoration techniques—specifically inpainting, denoising, and deblurring—through an innovative web application. Our Image Restoration Web App leverages advanced models and algorithms to enhance the quality of degraded images. The app integrates three core functionalities: denoising, deblurring, and inpainting, each designed to tackle specific types of image degradation.

Denoising employs sophisticated algorithms to remove unwanted noise from images, resulting in cleaner and more visually appealing outputs. This process is essential for improving the clarity of images captured in low-light or high-sensitivity conditions. Deblurring, on the other hand, addresses the common issue of image blurriness, which can occur due to camera shake, motion, or out-of-focus lenses. We utilize the NAFNet Model, which we have enhanced with a Multi-head Transposed Attention module, to restore sharpness and fine details to blurred images, making them crisp and clear. Inpainting is particularly useful for restoring old photographs or images with missing or damaged regions. By applying Microsoft's deep latent space translation model, our app can seamlessly fill in these areas, revitalizing old photographs and making them look whole and undamaged.

To run the web app locally, users need to follow a few straightforward steps. First, they must clone the repository to their local machine and navigate to the project directory. Then, they need to install the necessary dependencies, which are specified in the provided requirements file. Additionally, setting up the inpainting model involves downloading files and following instructions from Microsoft's Bringing Old Photos Back to Life project. Once the environment is set up, users can easily run the web app and access it via their browser. The application is designed to be user-friendly, ensuring that even

those with minimal technical expertise can effortlessly restore their images. For users who encounter issues with the provided commands for downloading checkpoints, pre-trained models and checkpoints can be manually downloaded from specified links. This project credits the NAFNet model for its contributions to state-of-the-art image restoration and Microsoft's Bringing Old Photos Back to Life project for its advancements in inpainting old photographs. By combining these powerful techniques, our Image Restoration Web App offers a comprehensive solution for enhancing and restoring the visual quality of various images.

Chapter 1

Introduction

Introduction

The project aims to develop an image restoration web application that can remove noise and blur from images and also fill the missing parts of the images. The application uses computer vision algorithms to restore the images to their original state.

- **Motivation**

  Image restoration is an important problem in the field of image processing using deep learning algorithms, because it addresses the challenge of recovering or enhancing degraded or corrupted images.

  There are several reasons why image restoration is important:

  Preservation of Visual Information: Images can be corrupted due to various factors such as noise, blur, or sensor limitations. Image restoration techniques aim to recover the original visual information that may have been lost or distorted during the image acquisition or storage process. By restoring images, valuable visual details can be preserved.

Enhancing Visual Quality: Image restoration techniques can also be used to enhance the visual quality of images by reducing noise, removing unwanted artifacts, or improving the overall sharpness and clarity. This is particularly valuable in applications such as photography, medical imaging, satellite imagery, and surveillance systems, where high-quality images are crucial for accurate analysis, diagnosis, or decision-making.

Historical Document Preservation: Image restoration plays a significant role in preserving and restoring historical documents, photographs, and artworks. These artifacts may deteriorate over time due to physical degradation, fading, or other environmental factors. By applying restoration techniques, the original appearance and content of these valuable cultural assets can be recovered, ensuring their long-term preservation and accessibility.

Forensic Analysis: Image restoration techniques are widely used in forensic analysis and criminal investigations. In forensic imagery, it is often necessary to enhance low-quality or degraded images to extract important details such as faces, license plates, or other evidence. By restoring and enhancing these images, investigators can improve their ability to identify suspects, reconstruct crime scenes, or analyse critical details.

Computer Vision Applications: Image restoration is an essential component in many computer vision applications such as object recognition, image segmentation, and 3D reconstruction. By restoring images to their optimal quality, these downstream tasks can benefit from cleaner and more accurate input data, leading to improved performance and reliability of computer vision algorithms.

Overall, image restoration techniques are crucial for a wide range of domains. By addressing the challenges of image degradation and corruption, these techniques contribute to the advancement of image processing, computer vision, deep learning, and the overall understanding and utilization of visual data.

- **Problem Definition**

  Digital images are a crucial part of our visual culture, used in various fields such as photography, medical imaging, and computer vision. However, images are often affected by noise and blur, which can significantly impact their quality and usability.

  Understanding Noise in Images: Noise in images refers to unwanted variations in pixel values that result in a loss of image clarity and detail. It can be caused by various factors, including sensor limitations, low lighting conditions, or image

compression. Noise manifests as random speckles or grainy patterns in the image and can be categorized into two main types: luminance noise (grayscale variations) and color noise (color variations)

Effects of Noise:

1-Reduced Image Quality: Noise deteriorates image quality, making it appear less sharp and clear. Fine details can become obscured by noise, especially in low-light or high-ISO images.

2. Loss of Color Fidelity: Color noise can distort the original colors in an image, resulting in color artifacts that affect the overall visual appeal.

Understanding Image Blurriness: Image blurriness occurs when the pixels in an image deviate from their true positions, creating a lack of sharpness. Blur can result from various sources, including camera shake, motion, or optical imperfections. Dealing with blur involves employing techniques that aim to restore image sharpness and recover lost details.

Effects of Blurriness:

- Reduced Image Clarity: Blurriness directly affects the clarity and sharpness of an image, making fine details less

discernible. This can result in a loss of visual impact and reduced overall image quality.

2. Loss of Information: In applications where precise details are critical, such as medical imaging or forensic analysis, blurriness can lead to a loss of valuable information, potentially affecting diagnostic accuracy or investigative efforts

Noise and blurriness are ubiquitous challenges in digital imagery, exerting substantial influence on image quality and interpretability. Understanding the ramifications of noise and the benefits of deblurring techniques is vital for photographers, medical practitioners, and anyone working with digital images

1.3 Similar System Information

The researchers have made significant contributions in addressing denoising and deblurring issues in image processing by developing various models such as NAFNet, Restormer, UNet, and plain blocks. They measured the effectiveness of these models using PSNR and SSIM metrics. In a recent paper, they explored the use of non-linear activation functions to enhance the performance of the models. Overall, their efforts have led to significant advancements in image processing, which holds great potential for various applications such as medical imaging and satellite imaging.

The researchers have made several important contributions to the field of image denoising and deblurring. These contributions include:

The development of new models for image denoising and deblurring, such as NAFNet, Restormer, and UNet.

The use of non-linear activation functions to enhance the performance of image denoising models.

The development of new evaluation metrics for image denoising and deblurring, such as the Perceptual Evaluation of Signal Quality (PESQ) metric.

The open-source release of their code and datasets, which has enabled other researchers to build on their work.

Their work has led to the development of new models and techniques that can be used to improve the quality of images. Their work also has the potential to be used in a variety of applications, such as medical imaging and satellite imaging.

1.4 Document Overview

Chapter 1: Introduction

This chapter provides an overview of the development of image restoration methods within the context of deep learning. It highlights the significant performance improvements achieved by deep learning-based approaches, citing specific examples. However, it points out that these methods often

suffer from high system complexity, which is further broken down into two parts: inter-block complexity and intra-block complexity.

Chapter 2: Related Work

In this chapter, we explore the landscape of related research, primarily focusing on image restoration and the role of Gated Linear Units (GLU) in achieving top-tier performance. We'll break down the complexities involved in image restoration methods and discuss the significance of GLU in computer vision.

2.1 Image Restoration

Image restoration is about improving degraded images, and fixing issues like noise or blur to make them look better. Recent advances in deep learning have made image restoration techniques very effective. Many of these methods are based on the classic UNet architecture, but they come with added complexities.

2.2 Gated Linear Units

Gated Linear Units (GLU) play a vital role in neural networks. They work by multiplying two linear transformation layers, one of which is activated by a non-linearity. GLU and similar techniques have been successful in natural language processing (NLP) and are gaining popularity in computer vision.

This paper emphasizes the importance of GLU. It explores removing the external nonlinear activation function within GLU

without sacrificing performance. Additionally, it reveals that GLU inherently contains nonlinearity, simplifying the baseline model by replacing external nonlinear activation functions with a simple multiplication of feature maps.

Chapter 4: Nonlinear Activation Free Network

In this chapter, we explore the potential for further improving performance while maintaining simplicity in our image restoration approach. We investigate commonalities in state-of-the-art (SOTA) methods and focus on the use of Gated Linear Units (GLU).

Gated Linear Units (GLU): GLU is a crucial component. It can be expressed as a combination of linear transformations and a non-linear activation function. We observe that adding GLU to our baseline may enhance performance but also increase intra-block complexity, which is not our goal. To address this, we revisit the activation function in our baseline, known as GELU. We discover that GELU can be seen as a special case of GLU, suggesting that GLU could be a generalization of activation functions. This leads to the proposal of a simplified GLU variant called Simple Gate, which divides the feature map into two parts and multiplies them together, significantly reducing complexity.

By replacing GELU with Simple Gate in the baseline, we achieve performance boosts in image denoising and deblurring tasks without compromising quality.

Simplified Channel Attention: We then explore the simplification of channel attention, a mechanism for aggregating global information and facilitating channel interaction. We propose Simplified Channel Attention (SCA) by retaining the essential roles of channel attention while reducing complexity. SCA proves to be simpler and just as effective as the original method.

Nonlinear Activation Free Network (NAFNet): These simplifications result in a baseline we call Nonlinear Activation Free Network (NAFNet). Remarkably, NAFNet achieves similar or superior performance to the baseline while eliminating external nonlinear activation functions. This is a significant achievement in simplifying complex computer vision models.

Chapter 2

Restoration Techniques

- Techniques

Related Work

1. Single-Stage U-Shaped Architecture

Technique: The paper adopts a classic single-stage U-shaped architecture with skip connections.

Scientists: This architectural choice is common in image restoration tasks and is based on prior works by Zamir et al. (Syed Waqas Zamir, Aditya Arora, Salman Khan, Munawar Hayat, Fahad Shahbaz Khan, and Ming-Hsuan Yang).

2. Plain Block

Technique: The authors design a plain block with common components, including convolution, ReLU, and shortcut connections.

Scientists: While not directly attributed to specific scientists, the design of plain blocks is a common practice in neural network architectures.

3 -Layer Normalization

Technique: Layer Normalization is added to stabilize the training process.

Scientists: Layer Normalization is a widely used technique in deep learning and has been discussed in various papers, including the work by Ba et al. (Jimmy Ba, Geoffrey Hinton, et al., 2016) titled "Layer Normalization."

4 -Activation Functions (ReLU and GELU)

Technique: ReLU is replaced with GELU in the plain block for activation.

Scientists: ReLU and GELU are common activation functions in deep learning. GELU, in particular, is discussed in the paper by Hendrycks et al. (Dan Hendrycks and Kevin Gimpel, 2016) titled "Gaussian Error Linear Units (GELUs)."

5 -Channel Attention

Technique: Channel Attention is added to the plain block to improve feature selection.

Scientists: The concept of channel attention has been explored in various papers, including the work by Hu et al. (Jie Hu, Li Shen, and Gang Sun, 2018) titled "Squeeze-and-Excitation Networks."

6 -Gated Linear Units (GLU)

Technique: GLU is introduced as a nonlinear activation function in the model.

Scientists: GLU is a concept introduced in various papers, and it has been particularly associated with NLP, such as the work by Dauphin et al. (Yann N. Dauphin et al., 2017) titled "Language Modeling with Gated Convolutional Networks."

7 -Simplified Channel Attention (SCA)

Technique: SCA is introduced to replace the channel attention in the Nonlinear Activation Free Network (NAFNet).

Scientists: The specific concept of SCA is rooted in the broader idea of channel attention by Woo et al.

8. Element-Wise Multiplication/Addition:

Technique: Element-wise multiplication and addition are used in various parts of the network for operations.

Scientists: Element-wise operations are fundamental in neural network architectures and have been used extensively.

9 -Multi-Dconv Head Transposed Attention (MDTA):

Technique: MDTA is introduced to address the computational overhead in Transformers caused by self-attention layers, especially in high-resolution image tasks.

Scientists: The specific concept of MDTA may be introduced in this paper, but it's rooted in the broader idea of optimizing attention mechanisms in Transformers for image processing.

10 -Channel-wise Self-Attention:

Technique: Instead of applying self-attention across the spatial dimensions of the input, MDTA applies self-attention across channels to compute cross-covariance across channels and generate an attention map encoding global context implicitly.

Scientists: The concept of channel-wise self-attention is introduced in this paper and may not be directly associated with a specific prior work. However, it builds on the general principles of self-attention.

11-Depth-wise Convolutions:

Technique: Depth-wise convolutions are introduced to emphasize local context before computing feature covariance to produce the global attention map.

Scientists: Depth-wise convolutions are a fundamental technique in convolutional neural networks (CNNs) and have been widely used in various computer vision tasks. While not attributed to specific scientists in this context, depth-wise convolutions have been discussed in many prior works in the field of CNNs.

12- Point-wise Convolutions:

Technique: Point-wise (1x1) convolutions are used to aggregate pixel-wise cross-channel context.

Scientists: Point-wise convolutions are commonly used in neural network architectures and may not be directly associated with specific scientists or works in this context.

13 -Attention Scaling Parameter (α):

Technique: A learnable scaling parameter (α) is introduced to control the magnitude of the dot product of query and key projections before applying the softmax function.

Scientists: The introduction of scaling parameters in attention mechanisms is a common practice in Transformers and deep learning but may not be attributed to specific scientists in this paper.

Chapter 3

System Design

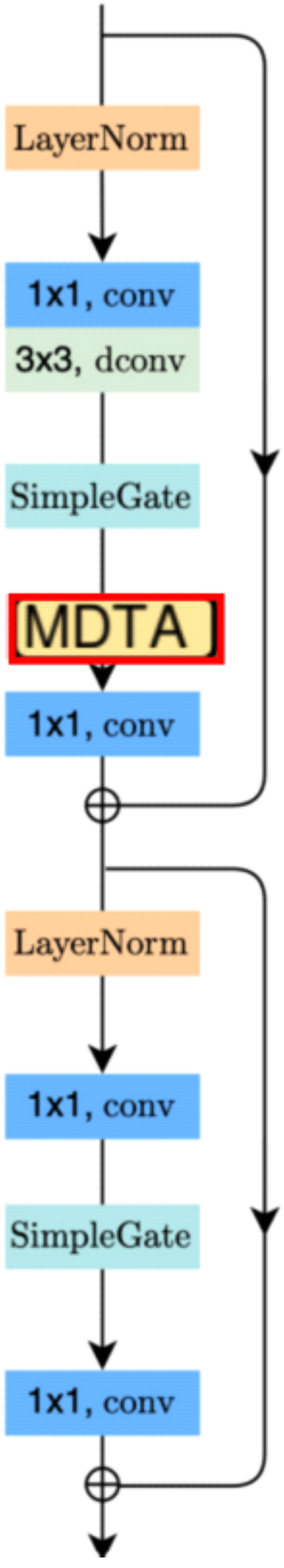
3.1 Design Overview

The original NAFNet model uses a simplified channel attention (SCA) module to learn the importance of each channel in the feature maps. The SCA module is computationally expensive and can be inefficient for feature mapping. And the Multi-dconv

Head Transposed Attention (MDTA) module is an efficient way to learn the importance of each channel and it can also improve the feature mapping.

3.2 Chosen System Architecture

Hesham et al. decided to use the NAFNet model to solve the problem of image denoising and deblurring. We decided to change the simplified channel attention (SCA) module with the Multi-dconv Head Transposed Attention (MDTA) to make it more efficient in feature mapping and less computationally expensive.


LayerNorm
1x1, conv
3x3, dconv
SimpleGate
MDTA
1x1, conv
LayerNorm
1x1, conv
SimpleGate
1x1, conv

3.3 Detailed Description of Components

3.3.1 Architecture

To reduce the inter-block complexity, we adopt the classic single-stage U-shaped architecture with skip-connections, as shown following [39,36].

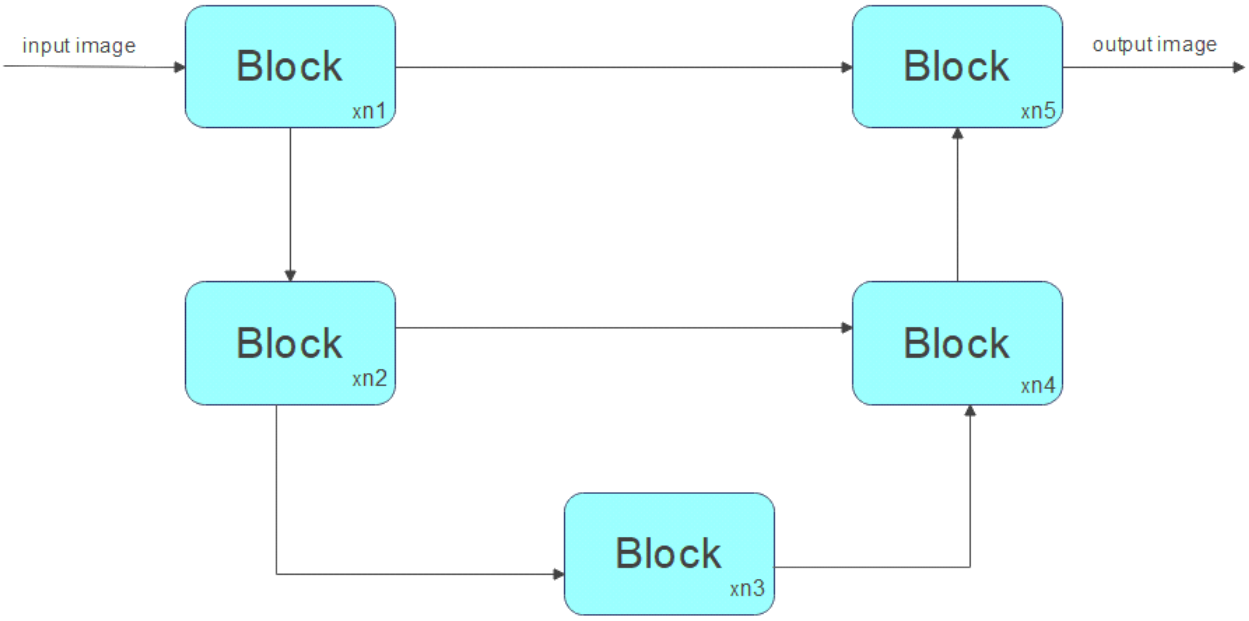


Fig. 2: UNet architecture, which is adopted by some SOTA methods, Some details have been deliberately omitted for simplicity, e.g. downsample/upsample layers, feature, fusion modules, input/output shortcut, and etc.

3.3.2 NAFNet

Neural Networks are stacked by blocks. We have determined how to stack blocks in a UNet architecture, but how to design the internal structure of the block is still a problem.

NAFNet (Nonlinear Activation Free Network) is an image restoration model proposed ,The goal of NAFNet is to achieve state-of-the-art performance while simplifying the architecture by eliminating unnecessary nonlinear activation functions

Hesham et al. decided to use the NAFNet model to solve the problem of image denoising and deblurring. We decided to change the simplified channel attention (SCA) module with the Multi-dconv Head Transposed Attention (MDTA) to make it more efficient in feature mapping and less computationally expensive.
NAFNet consists of the following key components:

- **Layer Normalization (LN)**: Used instead of Batch Normalization (BN) for stable normalization at the layer level.
- **Convolution Layers**: Standard 1x1 convolutions and 3x3 deconvolutions.
- **Activation Functions**:
- **SimpleGate Activation**: Replaces GELU.
- **Transposed Attention Map**
- **Multi Dconv Head Transposed Attention**

Normalization

Normalization is widely adopted in high-level computer vision tasks, and there is also the small batch size issue. we add Layer Normalization to the plain block described above. This change can make training smooth, even with a 10× increase in learning rate. The larger learning rate brings significant performance gain: +0.44 dB (39.29 dB to 39.73 dB) on SIDD[1], +3.39 dB (28.51 dB to 31.90 dB) on GoPro[26] dataset. To

sum up, we add Layer Normalization to the plain block as it can stabilize the training process.

Layer Normalization is a technique used to normalize activations within a neural network layer. Unlike Batch Normalization (BN), which operates at the batch level, LN directly estimates normalization statistics from the summed inputs to the neurons within a hidden layer. Here are the key points:

Purpose:

- LN aims to stabilize training by normalizing activations within each instance (sample) independently.
- It ensures that the distribution of activations remains consistent across features (channels) within a single instance.

Normalization Process:

- Given an input tensor with shape (batch_size, num_features, feature_dim), where:
  - batch_size: Number of instances in the batch.
  - num_features: Number of features (channels) in the layer.
  - feature_dim: Dimension of each feature vector.
- For each instance (sample) in the batch:
  - Compute the mean and variance across all features (channels) within that instance.
  - Normalize each feature by subtracting the mean and dividing by the standard deviation.
  - Learnable scale and shift parameters (gamma and beta) can be applied to the normalized features.

Advantages of Layer Normalization:

- **Stability**: LN remains stable even with small batch sizes, making it suitable for various scenarios.
- **Independence**: It does not introduce dependencies between training cases, as BN does.

Application in NAFNet:

- NAFNet uses LN instead of BN for stability.
- LN is commonly used in transformer architectures and works well with smaller batch sizes.

The formula to calculate layer normalization

$$\mu = \frac{1}{n}\sum_{i=1}^{n} x_i$$

$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2$$

Calculate the mean(μ) and variance ($\sigma 2$) across the features dimension (axis=-1 for tensors in Python libraries like TensorFlow and PyTorch).

Here, n represents the number of features in the input.

$$\hat{x}_i = \frac{x_i - \mu}{\sqrt{\sigma^2 + \epsilon}}$$

Normalize the input using the mean and variance

Where ϵ is a small constant (e.g., $10-5$) added for numerical stability to avoid division by zero.

$$y_i = \gamma \hat{x}_i + \beta$$

3. Scale and shift the normalized values using the parameters $\gamma$and β*:*

Here, yi represents the normalized and scaled input.

Convolutional neural network

Convolutional neural network (**CNN**) is a regularized type of feed- forward neural network that learns feature engineering by itself via filters (or kernel) optimization. Vanishing gradients and exploding gradients, seen during backpropagation in earlier neural networks, are prevented by using regularized weights over fewer connections. For example, for *each* neuron in the fully-connected layer, 10,000 weights would be required for processing an image sized 100 × 100 pixels. However, applying cascaded *convolution* (or cross-correlation) kernels, only 25 neurons are required to process 5x5-sized tiles. Higher-layer features are extracted from wider context windows, compared to lower-layer features.

Feed-forward neural networks are usually fully connected networks, that is, each neuron in one layer is connected to all neurons in the next layer. The "full connectivity" of these networks makes them prone to overfitting data. Typical ways of regularization,

or preventing overfitting, include penalizing parameters during training (such as weight decay) or trimming connectivity (skipped connections, dropout, etc.) Robust datasets also increase the probability that CNNs will learn the generalized principles that characterize a given dataset rather than the biases of a poorly-populated set

Purpose of Convolution Layers:

- Convolutional layers are designed to process and analyze visual data by applying learnable filters (kernels) to local regions.
- Their primary functions include feature extraction and hierarchical representation learning.

How Convolution Works:

- When data passes through a convolutional layer:
  - Each filter (kernel) convolves across the spatial dimensions of the input to produce a 2D activation map.
  - The filter slides over the input, computing a weighted sum of the local input values at each position.
  - The resulting activation map highlights relevant features (edges, textures, patterns) present in the input.

Standard 1x1 Convolution:

- A **1x1 convolution** is a specialized type of convolution that operates on individual pixels (or channels) without considering spatial neighborhoods.
- Key characteristics:
  - It performs a linear combination of input channels at each pixel location.
  - The output channel dimensions remain the same as the input.
  - Often used for channel-wise transformations, such as adjusting feature maps' depth or combining information from different channels.

3x3 Deconvolution (Transposed Convolution):

- A **3x3 deconvolution** (also known as a transposed convolution) is used for upsampling or feature expansion.
- Key points:
  - It increases the spatial resolution of feature maps.
  - The learnable weights (filters) are applied to overlapping regions of the input.

The output dimensions can be adjusted by choosing the stride and padding

Simple gate

Gated Linear Units (GLU) could be formulated as:

(1) $$GLU(x,g,\sigma)=x\odot\sigma(g(x))$$

where X represents the feature map, f and g are linear transformers, σ is a non-linear activation function, e.g. Sigmoid, and ⊙ indicates element-wise multiplication. As discussed above, adding GLU to our baseline may improve the performance yet the intra-block complexity is increasing as well. This is not what we expected.

GELU activation is defined as

$$0.5x\left(1+\tanh\left[\sqrt{\tfrac{2}{\pi}}\left(x+0.044715x^3\right)\right]\right)$$

(2) $$GELU(x)=x\Phi(x)$$

where Φ represents indicates the cumulative distribution function of the standard normal distribution, , GELU could be approximated and implemented by:

(3)

From Eqn. 1 and Eqn. 2, it can be noticed that GELU is a special case of GLU, i.e. f, g are identity functions and take σ as Φ. Through the similarity, we conjecture from another perspective that GLU may be regarded as a generalization of activation functions, and it might be able to replace the nonlinear activation functions. Further, we note that the GLU itself contains nonlinearity and does not depend on σ: even if the σ is removed, Gate(X) = f(X) ⊙ g(X) contains nonlinearity. Based on these, we propose a simple GLU variant: directly divide the feature map into two parts in the channel dimension and multiply them, as we shown in Figure 4c, noted as SimpleGate. Compared to the complicated implementation of GELU in Eqn.3, our SimpleGate could be implemented by an element-wise multiplication, that's all:

(4) $$SimpleGate(\mathbf{X},\mathbf{Y})=\mathbf{X}\odot\mathbf{Y}$$

where X and Y are feature maps of the same size. By replacing GELU in the baseline to the proposed SimpleGate, the performance of image denoising (on SIDD[1]) and

image deblurring (on GoPro[26] dataset) boost 0.08 dB (39.85 dB to 39.93 dB) and 0.41 dB (32.35 dB to 32.76 dB) respectively. The results demonstrate that GELU could be replaced by our proposed SimpleGate. At this point, only a few types of nonlinear activations left in the network: Sigmoid and ReLU in the channel attention module[16], and we will discuss the simplifications of it next

Multi-Dconv Head Transposed Attention

The major computational overhead in Transformers comes from the self-attention layer. the time and memory complexity of the key-query dotproduct interaction grows quadratically with the spatial resolution of input, i.e., O(W2H2) for images of W×H pixels. Therefore, it is infeasible to apply SA on most image restoration tasks that often involve high-resolution im ages. To alleviate this issue, we propose MDTA, that has linear complexity. The key ingredient is to apply SA across channels rather than the spatial dimension, i.e., to compute cross-covariance across channels to generate an attention map encoding the global context implicitly. As another essential component in MDTA, we introduce depth-wise convolutions to emphasize on the local context before computing feature covariance to produce the global attention map. From a layer normalized tensor Y ∈ R Hˆ ×Wˆ ×Cˆ , our MDTA first generates query (Q), key (K) and value (V) projections, enriched with local context. It is achieved by applying 1×1 convolutions to aggregate pixel-wise cross-channel context followed by 3×3 depth-wise convolutions to encode channel-wise spatial context, yielding Q=W Q d WQ p Y, K=W K d W K p Y and V=WV d WV p Y. Where W (·) p is the 1×1 point-wise convolution and W (·) d is the 3×3 depth-wise convolution. We use bias-free convolutional layers in the network. Next, we reshape query and key projections such that their dot-product interaction generates a transposed-attention map A of size R Cˆ×Cˆ , instead of the huge regular attention map of size R HˆWˆ ×HˆWˆ [17, 77]. Overall, the MDTA process is defined as: Xˆ = Wp Attention Qˆ , Kˆ , Vˆ + X, Attention Qˆ , Kˆ , Vˆ = Vˆ · SoftmaxKˆ · Qˆ /α , (1) where X and Xˆ are the input and output feature maps; Qˆ ∈ R HˆWˆ ×Cˆ ; Kˆ ∈ R Cˆ×HˆWˆ ; and Vˆ ∈ R HˆWˆ ×Cˆ matrices are obtained after reshaping tensors from the original size R Hˆ ×Wˆ ×Cˆ . Here, α is a learnable scaling parameter to control the magnitude of the dot product of Kˆ and Qˆ before applying the softmax function. Similar to the conventional multi-head SA [17], we divide the number of channels into ‘heads’ and learn separate attention maps in parallel.

Chapter 4

Implementation

4.1) Architecture

Layer Normalization Architecture

```
class LayerNormFunction(torch.autograd.Function):
    @staticmethod
    def forward(ctx, x, weight, bias, eps):
        ctx.eps = eps
        N, C, H, W = x.size()
        mu = x.mean(1, keepdim=True)
        var = (x - mu).pow(2).mean(1, keepdim=True)
        y = (x - mu) / (var + eps).sqrt()
        ctx.save_for_backward(y, var, weight)
        y = weight.view(1, C, 1, 1) * y + bias.view(1, C, 1, 1)
        return y

    @staticmethod
    def backward(ctx, grad_output):
        eps = ctx.eps

        N, C, H, W = grad_output.size()
        y, var, weight = ctx.saved_tensors
        g = grad_output * weight.view(1, C, 1, 1)
        mean_g = g.mean(dim=1, keepdim=True)

        mean_gy = (g * y).mean(dim=1, keepdim=True)
        gx = 1. / torch.sqrt(var + eps) * (g - y * mean_gy - mean_g)
        return gx, (grad_output * y).sum(dim=3).sum(dim=2).sum(dim=0), grad_output.sum(dim=3).sum(dim=2).sum(
            dim=0), None


class LayerNorm2d(nn.Module):

    def __init__(self, channels, eps=1e-6):
        super(LayerNorm2d, self).__init__()
        self.register_parameter('weight', nn.Parameter(torch.ones(channels)))
        self.register_parameter('bias', nn.Parameter(torch.zeros(channels)))
        self.eps = eps

    def forward(self, x):
        return LayerNormFunction.apply(x, self.weight, self.bias, self.eps)
```

MDTA Attention Module

```python
class Attention(nn.Module):
    def __init__(self, dim, num_heads, bias):
        super(Attention, self).__init__()
        self.num_heads = num_heads
        self.temperature = nn.Parameter(torch.ones(num_heads, 1, 1))

        self.qkv = nn.Conv2d(dim, dim * 3, kernel_size=1, bias=bias)
        self.qkv_dwconv = nn.Conv2d(dim * 3, dim * 3, kernel_size=3, stride=1, padding=1, groups=dim * 3, bias=bias)
        self.project_out = nn.Conv2d(dim, dim, kernel_size=1, bias=bias)

    def forward(self, x):
        b, c, h, w = x.shape

        qkv = self.qkv_dwconv(self.qkv(x))
        q, k, v = qkv.chunk(3, dim=1)

        q = rearrange(q, 'b (head c) h w -> b head c (h w)', head=self.num_heads)
        k = rearrange(k, 'b (head c) h w -> b head c (h w)', head=self.num_heads)
        v = rearrange(v, 'b (head c) h w -> b head c (h w)', head=self.num_heads)

        q = torch.nn.functional.normalize(q, dim=-1)
        k = torch.nn.functional.normalize(k, dim=-1)

        attn = (q @ k.transpose(-2, -1)) * self.temperature
        attn = attn.softmax(dim=-1)

        out = (attn @ v)

        out = rearrange(out, 'b head c (h w) -> b (head c) h w', head=self.num_heads, h=h, w=w)

        out = self.project_out(out)
        return out
```

Simple Gate Implementation

```python
class SimpleGate(nn.Module):
    def forward(self, x):
        x1, x2 = x.chunk(2, dim=1)
        return x1 * x2
```

NAFNet Block Architecture

```python
class NAFBlock(nn.Module):
    def __init__(self, c, DW_Expand=2, FFN_Expand=2, dropout_rate=0.):
        super().__init__()
        dw_channel = c * DW_Expand
        self.conv1 = nn.Conv2d(in_channels=c, out_channels=dw_channel, kernel_size=1, padding=0, stride=1, groups=1,
                               bias=True)
        self.conv2 = nn.Conv2d(in_channels=dw_channel, out_channels=dw_channel, kernel_size=3, padding=1, stride=1,
                               groups=dw_channel, bias=True)
        self.conv3 = nn.Conv2d(in_channels=dw_channel // 2, out_channels=c, kernel_size=1, padding=0, stride=1,
                               groups=1, bias=True)

        self.mdta = Attention(c, 8, True)

        self.sg = SimpleGate()

        ffn_channel = FFN_Expand * c

        self.conv4 = nn.Conv2d(in_channels=c, out_channels=ffn_channel, kernel_size=1, padding=0, stride=1, groups=1,
                               bias=True)
        self.conv5 = nn.Conv2d(in_channels=ffn_channel // 2, out_channels=c, kernel_size=1, padding=0, stride=1,
                               groups=1, bias=True)

        self.norm1 = LayerNorm2d(c)
        self.norm2 = LayerNorm2d(c)

        self.dropout1 = nn.Dropout(dropout_rate) if dropout_rate > 0. else nn.Identity()
        self.dropout2 = nn.Dropout(dropout_rate) if dropout_rate > 0. else nn.Identity()

        self.beta = nn.Parameter(torch.zeros((1, c, 1, 1)), requires_grad=True)
        self.gamma = nn.Parameter(torch.zeros((1, c, 1, 1)), requires_grad=True)

    def forward(self, input):
        x = input

        x = self.norm1(x)

        x = self.conv1(x)
        x = self.conv2(x)
        x = self.sg(x)
        x = x * self.mdta(x)
        x = self.conv3(x)

        x = self.dropout1(x)

        y = input + x * self.beta

        x = self.conv4(self.norm2(y))
        x = self.sg(x)
        x = self.conv5(x)

        x = self.dropout2(x)

        return y + x * self.gamma
```

NAFNet Unet

```python
class NAFNet(nn.Module):
    def __init__(self, img_channel=3, width=16, middle_blk_num=1, enc_blk_nums=[], dec_blk_nums=[]):
        super().__init__()

        self.intro = nn.Conv2d(in_channels=img_channel, out_channels=width, kernel_size=3, padding=1, stride=1,
                               groups=1,
                               bias=True)
        self.ending = nn.Conv2d(in_channels=width, out_channels=img_channel, kernel_size=3, padding=1, stride=1,
                                groups=1,
                                bias=True)

        self.encoders = nn.ModuleList()
        self.decoders = nn.ModuleList()
        self.middle_blks = nn.ModuleList()
        self.ups = nn.ModuleList()
        self.downs = nn.ModuleList()

        chan = width
        for num in enc_blk_nums:
            self.encoders.append(
                nn.Sequential(
                    *[NAFBlock(chan) for _ in range(num)]
                )
            )
            self.downs.append(
                nn.Conv2d(chan, 2 * chan, 2, 2)
            )
            chan = chan * 2
```

```
self.middle_blks = \
    nn.Sequential(
        *[NAFBlock(chan) for _ in range(middle_blk_num)]
    )

for num in dec_blk_nums:
    self.ups.append(
        nn.Sequential(
            nn.Conv2d(chan, chan * 2, 1, bias=False),
            nn.PixelShuffle(2)
        )
    )
    chan = chan // 2
    self.decoders.append(
        nn.Sequential(
            *[NAFBlock(chan) for _ in range(num)]
        )
    )

self.padder_size = 2 ** len(self.encoders)
```

```python
def forward(self, inp):
    B, C, H, W = inp.shape
    inp = self.check_image_size(inp)

    x = self.intro(inp)

    encs = []

    for encoder, down in zip(self.encoders, self.downs):
        x = encoder(x)
        encs.append(x)
        x = down(x)

    x = self.middle_blks(x)

    for decoder, up, enc_skip in zip(self.decoders, self.ups, encs[::-1]):
        x = up(x)
        x = x + enc_skip
        x = decoder(x)

    x = self.ending(x)
    x = x + inp

    return x[:, :, :H, :W]

def check_image_size(self, x):
    _, _, h, w = x.size()
    mod_pad_h = (self.padder_size - h % self.padder_size) % self.padder_size
    mod_pad_w = (self.padder_size - w % self.padder_size) % self.padder_size
    x = F.pad(x, (0, mod_pad_w, 0, mod_pad_h))
    return x
```

4.2) Training

Denoise Training

```
from image_restoration_web_app.Utils.addToExcelsheet import add_to_excelsheet
import random
import torch
import torch.nn as nn
from image_restoration_web_app.Archs.NafnetArch import NAFNet


def train(sidd_dataloader_train, div2k_dataloader_train, folder, checkpoint_path,
          excel_file, progress_file, device="cuda", lr=0.0001):

    device = torch.device(device)

    img_channel = 3
    width = 32

    enc_blks = [1, 1, 1, 28]
    middle_blk_num = 1
    dec_blks = [1, 1, 1, 1]

    model = NAFNet(img_channel=img_channel, width=width, middle_blk_num=middle_blk_num,
                   enc_blk_nums=enc_blks, dec_blk_nums=dec_blks).to(device)

    criterion = nn.MSELoss()
    optimizer = torch.optim.Adam(params=model.parameters(), lr=lr)
    checkpoint = torch.load(checkpoint_path, map_location=torch.device(device))
    model.load_state_dict(checkpoint['model_state_dict'])
    device = torch.device(device)
    epoch = int(checkpoint_path.split("/")[-1].split("_")[-1][:-4])
    print(epoch)
    num_epochs = epoch + 2
    with open(progress_file, "w") as file:
        for epoch in range(epoch + 1, num_epochs):
            i = 0
            batch_loss = []
            for batch_no, ((sidd_noisy_images, sidd_gt_images), (div2k_noisy_images,
div2k_gt_images)) in enumerate(
                    zip(sidd_dataloader_train, div2k_dataloader_train)):
                sidd_patches_loss = []
                div2k_patches_loss = []
                for patch_index in range(div2k_noisy_images[0].size(0)):
                    # SIDD
                    random_index = random.randint(0, (sidd_noisy_images[0].size(0)) - 1)
                    sidd_noisy_image = sidd_noisy_images[:, random_index].to(device)
                    sidd_gt_image = sidd_gt_images[:, random_index].to(device)
                    optimizer.zero_grad()
                    sidd_outputs = model(sidd_noisy_image)
                    sidd_loss = criterion(sidd_outputs, sidd_gt_image)
                    sidd_loss.backward()
                    optimizer.step()
                    sidd_patches_loss.append(sidd_loss.item())
                    # DIV2K
                    div2k_noisy_image = div2k_noisy_images[:, patch_index].to(device)
                    div2k_gt_image = div2k_gt_images[:, patch_index].to(device)
                    optimizer.zero_grad()
                    div2k_outputs = model(div2k_noisy_image)
                    div2k_loss = criterion(div2k_outputs, div2k_gt_image)
                    div2k_loss.backward()
                    optimizer.step()
                    div2k_patches_loss.append(div2k_loss.item())
```

```
                    progress_str = f"Epoch [{epoch}], Step [{batch_no +
1}/{len(sidd_dataloader_train)}], Patch [{patch_index + 1}/{min(sidd_noisy_images[0].size(0),
div2k_noisy_images[0].size(0))}], SIDD Loss: {sidd_loss.item()}, DIV2K Loss: {div2k_loss.item()}"
                    print(progress_str)
                    file.write(progress_str + "\n")

                sidd_batch_loss = sum(sidd_patches_loss) / len(sidd_patches_loss)
                div2k_batch_loss = sum(div2k_patches_loss) / len(div2k_patches_loss)
                batch_loss.append((sidd_batch_loss, div2k_batch_loss))
                i += 1
                if i % 20 == 0 and i != 0:
                    torch.save({
                        'model_state_dict': model.state_dict(),
                        'optimizer_state_dict': optimizer.state_dict(),
                    }, f"{folder}/Denoise_epoch_{epoch}.pth")
                    sidd_epoch_loss = sum([s[0] for s in batch_loss]) / len(batch_loss)
                    div2k_epoch_loss = sum([s[1] for s in batch_loss]) / len(batch_loss)
                    progress_str = f"Epoch [{epoch}], SIDD Epoch Loss: {sidd_epoch_loss}, DIV2K
Epoch Loss: {div2k_epoch_loss}"
                    print(progress_str)
                    file.write(progress_str + "\n")
            add_to_excelsheet(excel_file, epoch, sidd_epoch_loss, div2k_epoch_loss, lr)
    file.close()
```

Deblur Train

```
from image_restoration_web_app.Utils.addToExcelsheet import add_to_excelsheet
import torch
import torch.nn as nn
from image_restoration_web_app.Archs.NafnetArch import NAFNet

def train(gopro_dataloader_train, div2kblur_dataloader_train, folder, checkpoint_path, excel_file,
progress_file, device="cuda", lr=0.0001):
    device = torch.device(device)

    img_channel = 3
    width = 32

    enc_blks = [1, 1, 1, 28]
    middle_blk_num = 1
    dec_blks = [1, 1, 1, 1]

    model = NAFNet(img_channel=img_channel, width=width, middle_blk_num=middle_blk_num,
                   enc_blk_nums=enc_blks, dec_blk_nums=dec_blks).to(device)

    criterion = nn.MSELoss()
    lr = lr
    optimizer = torch.optim.Adam(params=model.parameters(), lr=lr)
    checkpoint = torch.load(checkpoint_path, map_location=torch.device(device))
    model.load_state_dict(checkpoint['model_state_dict'])
    device = torch.device(device)
    epoch = int(checkpoint_path.split("/")[-1].split("_")[-1][:-4])
    print(epoch)
    i = 0
    num_epochs = epoch + 2
    with open(progress_file, "w") as file:
        for epoch in range(epoch + 1, num_epochs):
            i = 0
            batch_loss = []
            for batch_no, ((gopro_blurry_images, gopro_gt_images), (div2k_blurry_images,
div2k_gt_images)) in enumerate(
```

```
                    zip(gopro_dataloader_train, div2kblur_dataloader_train)):
                gopro_patches_loss = []
                div2kblur_patches_loss = []
                for patch_index in range(gopro_blurry_images[0].size(0)):
                    ## gorpro
                    gopro_blurry_image = gopro_blurry_images[:, patch_index].to(device)
                    gopro_gt_image = gopro_gt_images[:, patch_index].to(device)
                    optimizer.zero_grad()
                    outputs = model(gopro_blurry_image)
                    gopro_loss = criterion(outputs, gopro_gt_image)
                    gopro_loss.backward()
                    optimizer.step()
                    gopro_patches_loss.append(gopro_loss.item())
                    ## Div2kblur
                    div2k_blurry_image = div2k_blurry_images[:, patch_index].to(device)
                    div2k_gt_image = div2k_gt_images[:, patch_index].to(device)
                    optimizer.zero_grad()
                    outputs = model(div2k_blurry_image)
                    div2kblur_loss = criterion(outputs, div2k_gt_image)
                    div2kblur_loss.backward()
                    optimizer.step()
                    div2kblur_patches_loss.append(div2kblur_loss.item())

                    progress_str = f"Epoch [{epoch}], Step [{batch_no +
1}/{len(gopro_dataloader_train)}], Patch [{patch_index + 1}/{min(gopro_blurry_images[0].size(0),
div2k_blurry_images[0].size(0))}], GoPro Loss: {gopro_loss.item()}, DIV2KBlur Loss:
{div2kblur_loss.item()}"
                    print(progress_str)
                    file.write(progress_str + "\n")

                gopro_batch_loss = sum(gopro_patches_loss) / len(gopro_patches_loss)
                div2k_batch_loss = sum(div2kblur_patches_loss) / len(div2kblur_patches_loss)
                batch_loss.append((gopro_batch_loss, div2k_batch_loss))
            i += 1
            if i % 50 == 0 and i != 0:
                torch.save({
                    'model_state_dict': model.state_dict(),
                    'optimizer_state_dict': optimizer.state_dict(),
                }, f"{folder}/GoPro_epoch_{epoch}.pth")
                gopro_epoch_loss = sum([s[0] for s in batch_loss]) / len(batch_loss)
                div2k_epoch_loss = sum([s[1] for s in batch_loss]) / len(batch_loss)
                progress_str = f"Epoch [{epoch}], GoPro Epoch Loss: {gopro_epoch_loss}, Div2k Epoch
Loss: {div2k_epoch_loss}"
                print(progress_str)
                add_to_excelsheet(excel_file, epoch, gopro_epoch_loss, div2k_epoch_loss, lr)
                file.write(progress_str + "\n")
            torch.save({
                    'model_state_dict': model.state_dict(),
                    'optimizer_state_dict': optimizer.state_dict(),
                }, f"{folder}/GoPro_epoch_{epoch}.pth")
            gopro_epoch_loss = sum([s[0] for s in batch_loss]) / len(batch_loss)
            div2k_epoch_loss = sum([s[1] for s in batch_loss]) / len(batch_loss)
            progress_str = f"Epoch [{epoch}], GoPro Epoch Loss: {gopro_epoch_loss}, Div2k Epoch
Loss: {div2k_epoch_loss}"
            print(progress_str)
            add_to_excelsheet(excel_file,epoch, gopro_epoch_loss, div2k_epoch_loss, lr)
            file.write(progress_str + "\n")
    file.close()
```

4.3) Utilities

Denoise Inference

```
import torch
from torchvision import transforms
from PIL import Image
import matplotlib.pyplot as plt
from Archs.NafnetArch import NAFNet


def predict(input_image_path):
    checkpoint_path = "../checkpoints/Denoise_epoch_68.pth"

    img_channel = 3
    width = 32
    enc_blks = [1, 1, 1, 28]
    middle_blk_num = 1
    dec_blks = [1, 1, 1, 1]
    device = torch.device("cpu")

    predict_model = NAFNet(img_channel=img_channel, width=width, middle_blk_num=middle_blk_num,
                           enc_blk_nums=enc_blks, dec_blk_nums=dec_blks).to(device)
    optimizer = torch.optim.Adam(params=predict_model.parameters(), lr=0.0001)


    checkpoint = torch.load(checkpoint_path, map_location=device)
    predict_model.load_state_dict(checkpoint['model_state_dict'])
    optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
    predict_model.eval()

    noisy_image = Image.open(input_image_path)
    transform = transforms.Compose([
        transforms.ToTensor(),
    ])

    noisy_image = transform(noisy_image).to(device)


    with torch.no_grad():
        output_chunk = predict_model(torch.unsqueeze(noisy_image, 0))

    image_np1 = noisy_image.permute(1, 2, 0).cpu().numpy()
    image_np2 = output_chunk[0].permute(1, 2, 0).cpu().numpy()
    image_np2 = (image_np2 - image_np2.min()) / (image_np2.max() - image_np2.min())
    return image_np1, image_np2
```

Deblur Inference

```
import torch
from torchvision import transforms
from PIL import Image
import matplotlib.pyplot as plt
from Archs.NafnetArch import NAFNet


def predict(input_image_path):
    checkpoint_path = "../checkpoints/GoPro_epoch_58.pth"

    img_channel = 3
    width = 32
    enc_blks = [1, 1, 1, 28]
    middle_blk_num = 1
    dec_blks = [1, 1, 1, 1]
    device = torch.device("cpu")

    predict_model = NAFNet(img_channel=img_channel, width=width, middle_blk_num=middle_blk_num,
                           enc_blk_nums=enc_blks, dec_blk_nums=dec_blks).to(device)
    optimizer = torch.optim.Adam(params=predict_model.parameters(), lr=0.0001)


    checkpoint = torch.load(checkpoint_path, map_location=device)
    predict_model.load_state_dict(checkpoint['model_state_dict'])
    optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
    predict_model.eval()

    noisy_image = Image.open(input_image_path)
    transform = transforms.Compose([
        transforms.ToTensor(),
    ])

    noisy_image = transform(noisy_image).to(device)


    with torch.no_grad():
        output_chunk = predict_model(torch.unsqueeze(noisy_image, 0))

    image_np1 = noisy_image.permute(1, 2, 0).cpu().numpy()
    image_np2 = output_chunk[0].permute(1, 2, 0).cpu().numpy()
    image_np2 = (image_np2 - image_np2.min()) / (image_np2.max() - image_np2.min())

    return image_np1, image_np2
```

Inpaint Inference

```
import os.path
from subprocess import Popen, PIPE
from PIL import Image
import io
import base64
import os


def run_inference(image_path, output_folder="F:/machine-
learning/Image_restoration_UI/database/output", gpu=-1, with_scratch=True):
    command = [
        "python", "../Bringing-Old-Photos-Back-to-Life/run.py",
        "--input_folder", str(image_path),
        "--output_folder", str(output_folder),
        "--GPU", str(gpu),
    ]
    if with_scratch:
        command.append("--with_scratch")

    process = Popen(command, stdout=PIPE, stderr=PIPE)
    stdout, stderr = process.communicate()

    if process.returncode != 0:
        print("Error running inference:")
        print(stderr.decode())
    else:
        print("Inference completed successfully.")
```

Handle Base64

```
import base64
import numpy as np
import time


def decode(base64_string):
    header, data = base64_string.split(';base64,')
    ext = header.split('/')[-1]
    with open("../database/old.png", 'wb') as f:
        f.write(base64.b64decode(data))

    return "../database/old.png"


def encode(path):
    base64_image = base64.encode(path)
    return base64_image
```

4.4) Server

```python
from flask import Flask, request, send_file, jsonify
import os
from Utils import inpaint_inference, deblur_inference, denoise_inference, handle_base64
import matplotlib.pyplot as plt
from flask_cors import CORS
import base64
import numpy as np
from datetime import datetime

app = Flask(__name__)
CORS(app)

@app.route("/denoise", methods=["POST", "GET"])
def denoise():
    base64_string = request.form["base64_string"]
    print(base64_string)
    image_path = handle_base64.decode(base64_string)
    print("denoising... Start")
    output_image_noisy, output_image_clear = denoise_inference.predict(image_path)
    print("denoising... Done")
    print(output_image_clear.shape)
    # # response.headers.add('Access-Control-Allow-Origin', '*')
    denoised_image_path = "F:/machine-learning/Image_restoration_UI/database/denoised.jpg"
    plt.imsave(denoised_image_path, output_image_clear)
    print(jsonify({'path': "database/denoised.jpg"}))
    return jsonify({'path': "database/denoised.jpg"})

@app.route("/deblur", methods=["POST", "GET"])
def deblur():
    base64_string = request.form["base64_string"]
    print(base64_string)
    image_path = handle_base64.decode(base64_string)
    print("deblur... Start")
    output_image_blurry, output_image_clear = deblur_inference.predict(image_path)
    print("deblur... Done")
    deblurred_image_path = "F:/machine-learning/Image_restoration_UI/database/deblurred.jpg"
    plt.imsave(deblurred_image_path, output_image_clear)
    print(jsonify({'path': "database/deblurred.jpg"}))
    return jsonify({'path': "database/deblurred.jpg"})

@app.route("/inpaint", methods=["POST", "GET"])
def inpaint():
    base64_string = request.form["base64_string"]
    print(base64_string)
    image_path = handle_base64.decode(base64_string)
    print("Inpainting... Start")
    print(image_path[:-8])
    inpaint_inference.run_inference(image_path[:-8], )
    print("Inpainting... Done")
    print(jsonify({'path': "database/output/final_output/old.png"}))
    return jsonify({'path': "database/output/final_output/old.png"})

if __name__ == "__main__":
    app.run(port=5000)
```

Chapter 5

Proposed system

5.1) Description

5.2) Datasets

5.3) System Results

5.4) System Performance

5.5) Tools and Hardware used

5.6) Future Work

5.7) System Snapshots

5.1) Description

The image restoration system is designed to enhance and restore images by performing tasks such as deblurring, denoising, and inpainting. The system is built around a custom neural network architecture, NAFNet, and follows a modular pipeline for processing images. Key components of the system include:

Preprocessing: Converts input images into tensors suitable for model inference.

Model Inference: Uses Our NAFNet model | Microsoft's Model to process and restore images.

Post-processing: Converts the output tensors back into image format for display or storage.

5.2) Datasets

The NAFNet Model trained and tested on various publicly available image restoration datasets. Examples include:

1) Smartphone Image Denoising Dataset (SIDD)

- **Description**: SIDD consists of thousands of noisy and clean image pairs captured in real-world scenarios using different smartphone cameras under various lighting conditions.
- **Purpose**: Used for training and evaluating the Denoising capability of the image restoration system.

2) GoPro Dataset

- **Description**: The GoPro Dataset consists of high-resolution blurred and corresponding sharp image pairs captured using a GoPro camera. The images are taken in diverse real-world scenarios, including various types of motion and lighting conditions.
- **Purpose**: Used for training and evaluating the Deblurring capability of the image restoration system.

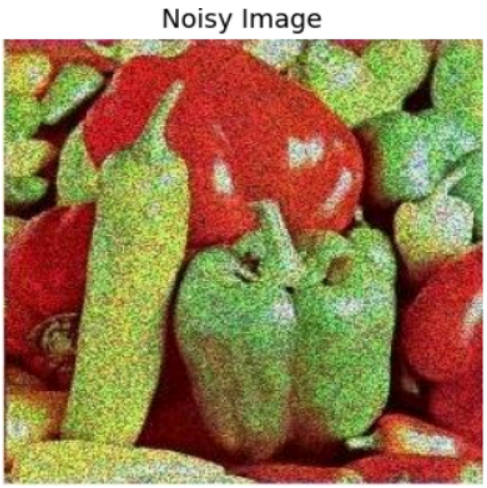
3) DIV2K Dataset

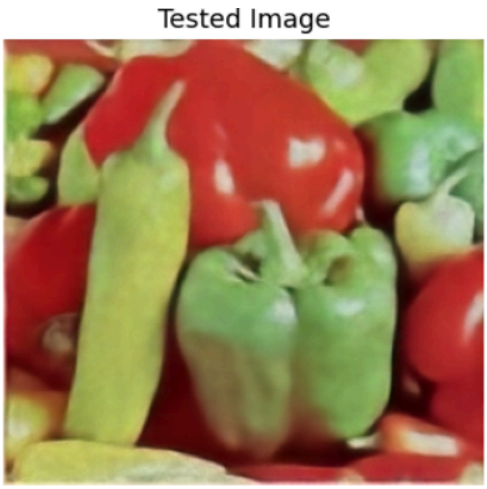
- **Description**: The DIV2K dataset contains high-resolution images for image super-resolution tasks, with images available in both low-resolution and high-resolution formats.
- **Purpose**: Used for training and evaluating the Denoising & Deblurring capabilities of the system.

For demonstration, a custom set of noisy, blurred, and corrupted images can also be used.

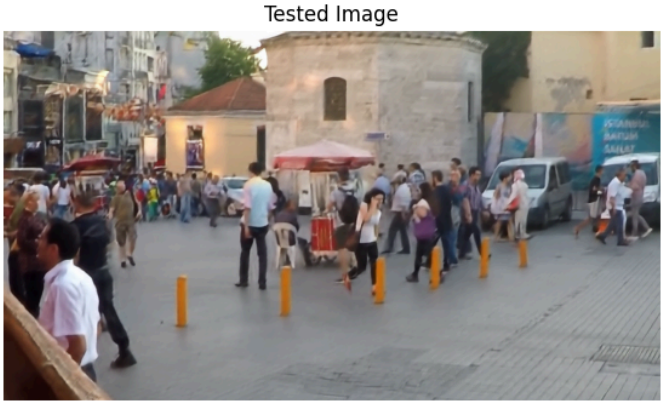
5.3) System Results

Denoising: The system effectively removes noise, improving the clarity and detail of the image.





Deblurring: The system successfully reduces motion blur, enhancing the sharpness of the image.



Inpainting: The system fills in missing or corrupted parts of the image, creating a seamless restoration.

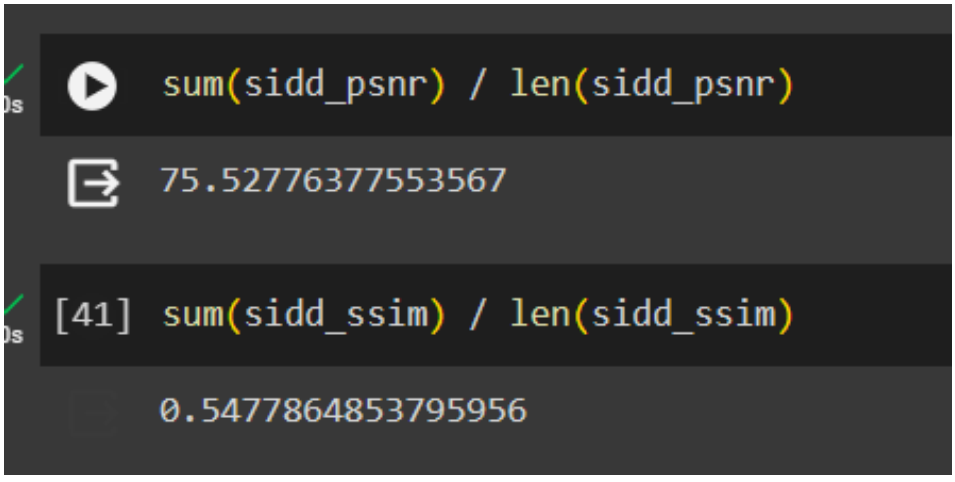
5.4) System Performance

The system's performance is evaluated based on its ability to restore images to a high quality. Metrics used for evaluation may include:

- PSNR (Peak Signal-to-Noise Ratio): Measures the quality of the restored image compared to the original.
- SSIM (Structural Similarity Index): Measures the similarity between the restored and original images.
- MSE (Mean Squared Error) Loss: Measures the average squared difference between the restored and original image pixels. Lower values indicate better restoration quality.

Denoising

PSNR | SSIM (Test set):

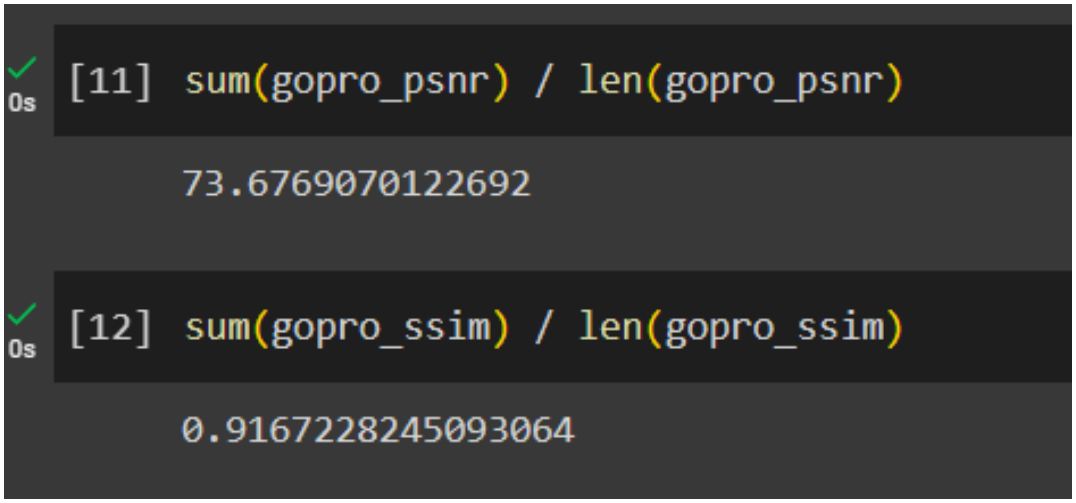


Loss:

1	0.000590088	0.0001	SIDD
15	0.000138525	0.00009	SIDD
30	0.00011624	0.00001	SIDD
45	0.007370933739	0.0001	Div2k
68	0.003343170568	0.0001	Combined (SIDD & DIV2K)

Deblurring

PSNR | SSIM (Test set):



Loss:

1	0,003683623	0,0001
15	0,001813571	0,0001
30	0,001189649	0,0001

45	0,0009301421095	0,0001

59	0,000758353632	0,0001

5.5) Tools and Hardware

Programming Language: Python

Libraries and Frameworks:

- PyTorch: For deep learning model implementation and inference.
- PIL (Python Imaging Library): For image loading and processing.
- torchvision.transforms: For image transformations.
- Flask: For building and deploying the web application.
- matplotlib: For visualizing images and results.
- numpy: For numerical operations and array manipulations.
- base64: For encoding and decoding image data.

Software:

Frontend => React.js

Backend => Flask

Hardware:

Laptop: Intel Core i3 10110u Processor (4M Cache, up to 4.10 GHz), with 12GB RAM, with MX130 2GB VRAM GPU and Windows 10 with 64-bit operating system.

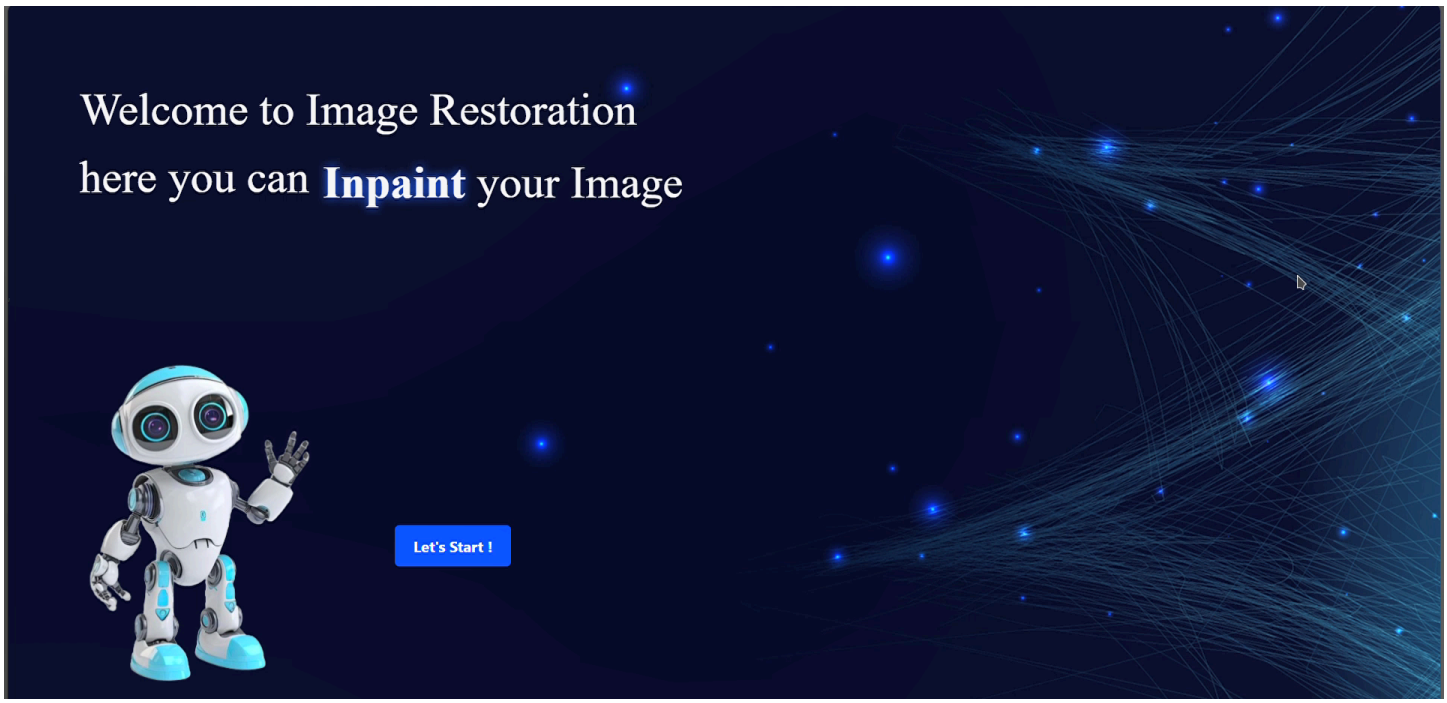
5.6) Future Work:

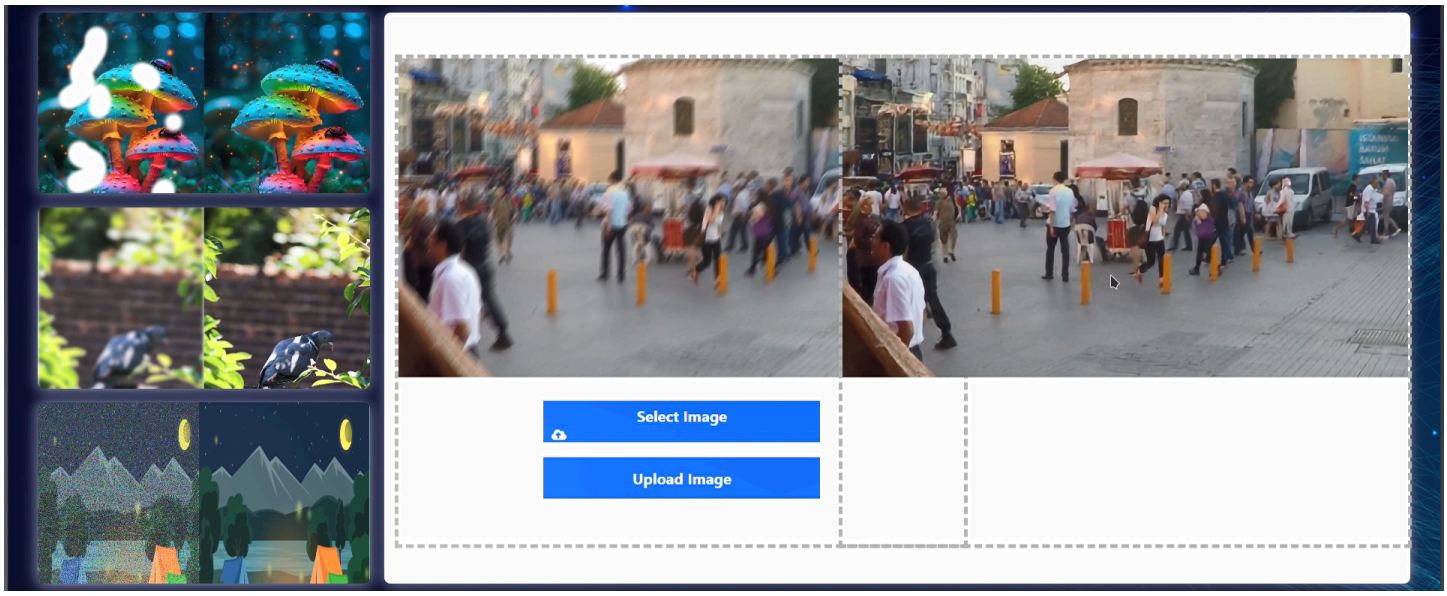
Future improvements and extensions of the system might include:

- Model Optimization: Enhancing the model for better performance and faster inference.
- Real-time Processing: Implementing real-time image restoration for video applications.
- Extended Dataset: Training the model on larger and more diverse datasets to improve generalization.
- Fine-tune Inpainting Model using Masked-images generated based on DIV2K dataset

5.7) System Snapshots

Below are examples of the system in action, including input images, restored outputs, and the model's processing stages.





References

[13] Han, Q., Fan, Z., Dai, Q., Sun, L., Cheng, M.M., Liu, J., Wang, J.: Demystifying local vision transformer: Sparse connectivity, weight sharing, and dynamic weight. arXiv preprint arXiv:2106.04263 (2021)

[22] Liu, Z., Lin, Y., Cao, Y., Hu, H., Wei, Y., Zhang, Z., Lin, S., Guo, B.: Swin transformer: Hierarchical vision transformer using shifted windows. In: Proceedings of the IEEE/CVF International Conference on Computer Vision. pp. 10012–10022 (2021)