GNR 638 Mini Project 2

Kunal Randad
Roll Number: 20d070049

Ayush Raisoni
Roll Number: 200100041

March 2024

Problem-1: Image Deblurring

1 Introduction

Image deblurring is a crucial task in computer vision, aiming to restore sharpness and clarity to blurred images. This project addresses the challenge of designing a neural network architecture capable of effectively deblurring images, with a constraint on model complexity.

The dataset consists of two sets of images: Set A contains sharp images, while Set B comprises the same images with varying levels of blur introduced through Gaussian filters of different kernel sizes and sigma values. Set A serves as the ground truth for evaluation, while Set B is used as input for deblurring.

Initially, images from Set A are downscaled to (256,448) resolution for uniform processing. This was done using python and the resized images were saved locally. Set B is generated by applying Gaussian filters with kernel sizes of 3x3, 7x7, and 11x11, and corresponding sigma values of 0.3, 1, and 1.6, respectively. This diverse dataset provides a challenging training environment.

The objective is to design a neural network capable of deblurring images from Set B to resemble their sharp counterparts in Set A. However, model complexity is restricted to 15 million parameters to ensure efficiency.

2 Assumption

Resized images have height and width as 256 and 448 respectively (rather than 448 and 256) to preserve aspect ratio.

3 Pre-Processing

3.1 Data Compression

Initially running the models on personal laptop could not take place due to slow GPU and limited RAM. So, it was necessary to shift to Google Colab. The dataset was of size 32 GB and could not be processed on Google Colab due to limited memory. The compressed dataset was of size about 4.5 GB for sharp images and filtered images. Thus, a total memory of 18 GB would be required if all images were to be

uploded which is not feasible.
To solve this, I explored the sub folders. There were 240 subfolders and each sub-folder contained 100 image frames which were taken with a very small time-gap. Out of these 100 images, 1'st, 40'th and 80'th frames were chosen. Thus, only 3 out of 100 images were taken actually for training.

4 Models Tried

In this section, we discuss the various models explored for the image deblurring task. We experimented with different architectures to find the most suitable one for our problem.

4.1 Baseline Convolutional Neural Network (CNN)

We began our exploration with a baseline Convolutional Neural Network (CNN) architecture. The baseline CNN consisted of several convolutional layers. Max-Pooling was not used as Image dimensions had to be conserved. Thus, we had to use big kernels so that local spatial information could play a role in deblurring. However, because of larger kernel size, the number of parameters blew up and it took a lot of steps to train. There were visible artefacts as well, for eg, lines at edges. Some images also turned out greyish.

4.2 U-Net

Next, we experimented with the U-Net architecture, which is widely used for image segmentation tasks. U-Net consists of an encoder-decoder structure with skip connections between corresponding encoder and decoder layers. These skip connections allow the network to preserve spatial information during upsampling. Despite its success in segmentation tasks, we found that U-Net struggled to deblurr images effectively.

4.3 Residual Networks (ResNet)

We also investigated the use of Residual Networks (ResNet) for image deblurring. ResNet introduces residual connections, allowing for deeper networks to be trained without encountering the vanishing gradient problem. We experimented with different depths of ResNet architectures. As Max Pooling was not used here, same problem was faced as earlier.

4.4 Custom Architectures

Finally, we explored designing custom architectures tailored specifically for the image deblurring task. These architectures incorporated elements from the baseline CNN, U-Net, and ResNet architectures, along with additional modifications.

U-Net seemed to be a good approach but it was not much succesful. I investigated the structure of U-Net and found that they contain several Up-Sampling layers, some close to output layer. This could have been a cause of blurred output as Up-Sampling the image causes blurring and it seems difficult to deblur an image in 1 Convolutional Layer.

Thus, to tackle this problem, a few only convolutional layers were added before the output. As the input had gone through multiple convolution, pooling and upsampling, it was no longer a simple representation of input image but rather a representation of its features.

This meant we need to introduce the initial image once again so that the information in image is used and this was, our output is guaranteed to be at-least as good as the input. This was done through concatenation. Input Image and output of first convolution layer was concatenated just before the output layer. Finally, this archotecture worked!

5 Final Model Architecture

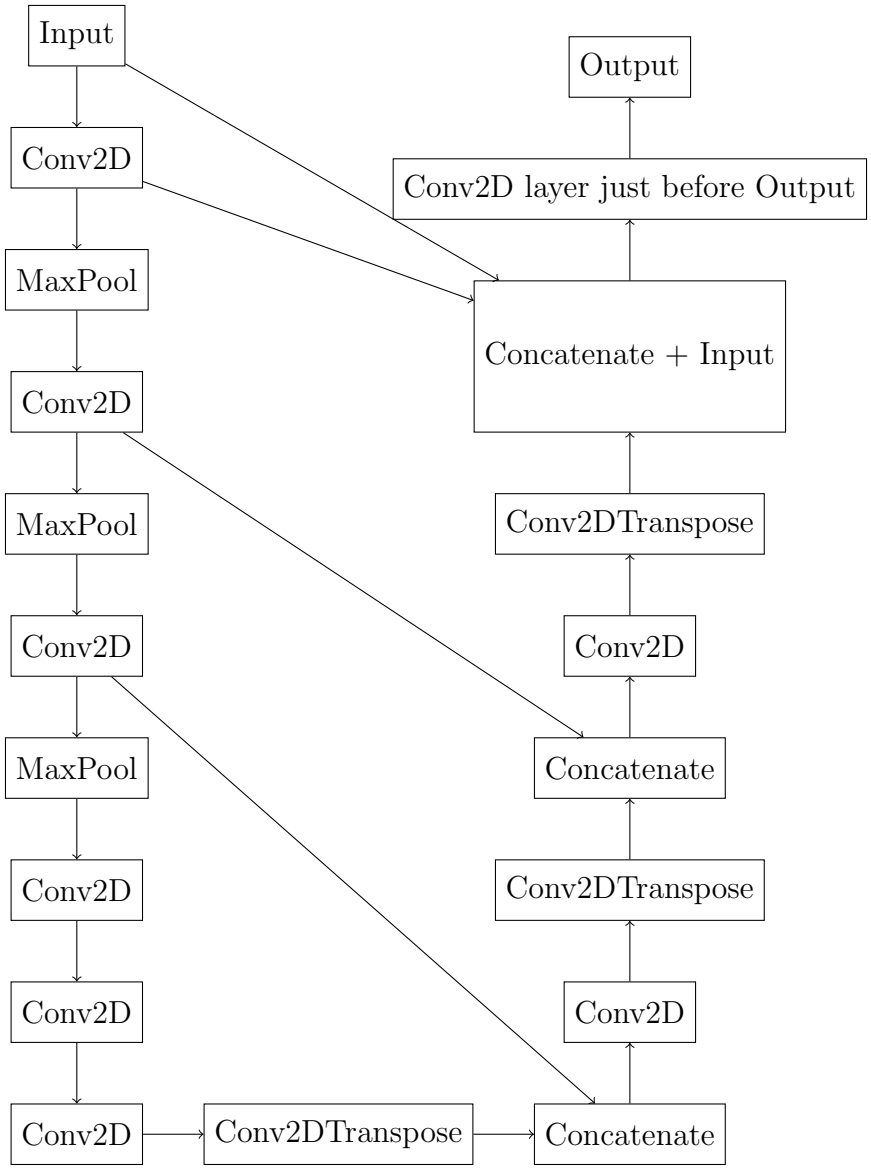

Figure 1: UNet Model Block Diagram

Table 1: Model: model

Layer (type)	**Output Shape**	**Param #**	**Connected to**
InputLayer	(None, 256, 448, 3)	0	[]
Conv2D	(None, 256, 448, 64)	1792	['input_1[0][0]']
Conv2D	(None, 256, 448, 64)	36928	['conv2d[0][0]']
MaxPooling2D	(None, 128, 224, 64)	0	['conv2d_1[0][0]']
Conv2D	(None, 128, 224, 128)	73856	['max_pooling2d[0][0]']
Conv2D	(None, 128, 224, 128)	147584	['conv2d_2[0][0]']
MaxPooling2D	(None, 64, 112, 128)	0	['conv2d_3[0][0]']
Conv2D	(None, 64, 112, 256)	295168	['max_pooling2d_1[0][0]']
Conv2D	(None, 64, 112, 256)	590080	['conv2d_4[0][0]']
MaxPooling2D	(None, 32, 56, 256)	0	['conv2d_5[0][0]']
Conv2D	(None, 32, 56, 512)	1180160	['max_pooling2d_2[0][0]']
Conv2D	(None, 32, 56, 512)	2359808	['conv2d_6[0][0]']
Conv2D	(None, 32, 56, 512)	2359808	['conv2d_7[0][0]']
Conv2DTranspose	(None, 64, 112, 256)	524544	['conv2d_8[0][0]']
Concatenate	(None, 64, 112, 512)	0	['conv2d_tr.[0][0]', 'conv2d_5[0][0]']
Conv2D	(None, 64, 112, 256)	1179904	['concatenate[0][0]']
Conv2D	(None, 64, 112, 256)	590080	['conv2d_9[0][0]']
Conv2DTranspose	(None, 128, 224, 128)	131200	['conv2d_10[0][0]']
Concatenate	(None, 128, 224, 256)	0	['conv2d_tr._1[0][0]', 'conv2d_3[0][0]']
Conv2D	(None, 128, 224, 128)	295040	['concatenate_1[0][0]']
Conv2D	(None, 128, 224, 128)	147584	['conv2d_11[0][0]']
Conv2D	(None, 128, 224, 128)	147584	['conv2d_12[0][0]']
Conv2DTranspose	(None, 256, 448, 64)	32832	['conv2d_13[0][0]']
Concatenate	(None, 256, 448, 128)	0	['conv2d_tr._2[0][0]', 'conv2d_1[0][0]']
Conv2D	(None, 256, 448, 64)	73792	['concatenate_2[0][0]']
Conv2D	(None, 256, 448, 64)	36928	['conv2d_14[0][0]']
Conv2D	(None, 256, 448, 64)	36928	['conv2d_15[0][0]']
Conv2D	(None, 256, 448, 64)	36928	['conv2d_16[0][0]']
Conv2D	(None, 256, 448, 64)	36928	['conv2d_17[0][0]']
Conv2D	(None, 256, 448, 64)	36928	['conv2d_18[0][0]']
Conv2D	(None, 256, 448, 64)	36928	['conv2d_19[0][0]']
Concatenate	(None, 256, 448, 67)	0	['conv2d_20[0][0]', 'input_1[0][0]']
Conv2D	(None, 256, 448, 128)	77312	['concatenate_3[0][0]']
Conv2D	(None, 256, 448, 128)	147584	['conv2d_21[0][0]']
Conv2D	(None, 256, 448, 3)	387	['conv2d_22[0][0]']
Total params	10,577,667 (40.35 MB)		

6 Training Details

Given Batch Size 10, the data loader returns 10 random blurred and their corresponding sharp images. As there were 3 kernels given, images are chosen corresponding to a kernel randomly and uniformly. Out of the 240 sub-folders, the sub-folder is also selected randomly and uniformly. From the provided code, the following training details are utilized:

- **Batch Size**: 10
- **Epochs**: 40
- **Loss Function**: Mean Squared Error (MSE)
- **Optimizer**: Adam
- **Steps per Epoch**: 64 (Number of gradient steps taken per epoch)

Callbacks:

- **ModelCheckpoint**: Used to save the best model during training.

Training Procedure:

```
from tensorflow.keras.callbacks import ModelCheckpoint
import os

batch_size = 10
epochs = 40
checkpoint = ModelCheckpoint('/content', verbose=1, save_best_only=True)
model = create_unet_model2()
model.compile(optimizer='adam', loss='mean_squared_error')

from tensorflow.keras.callbacks import ModelCheckpoint, Callback, History
class LossPlotter(Callback):
    def on_train_begin(self, logs={}):
        self.losses = []

    def on_epoch_end(self, epoch, logs={}):
        self.losses.append(logs.get('loss'))
        plt.plot(self.losses)
        plt.title('Model Loss')
        plt.xlabel('Epoch')
        plt.ylabel('Loss')
        plt.show()

# loss_plotter = LossPlotter()
print(model.summary())
model.fit(dataloader(blurred_image_dir_1, blurred_image_dir_2, blurred_image_dir_3,
downscaled_image_dir, batch_size),
            batch_size=batch_size,
            epochs=epochs,
            steps_per_epoch=64,
            callbacks=[checkpoint])
```

7 Training Curves

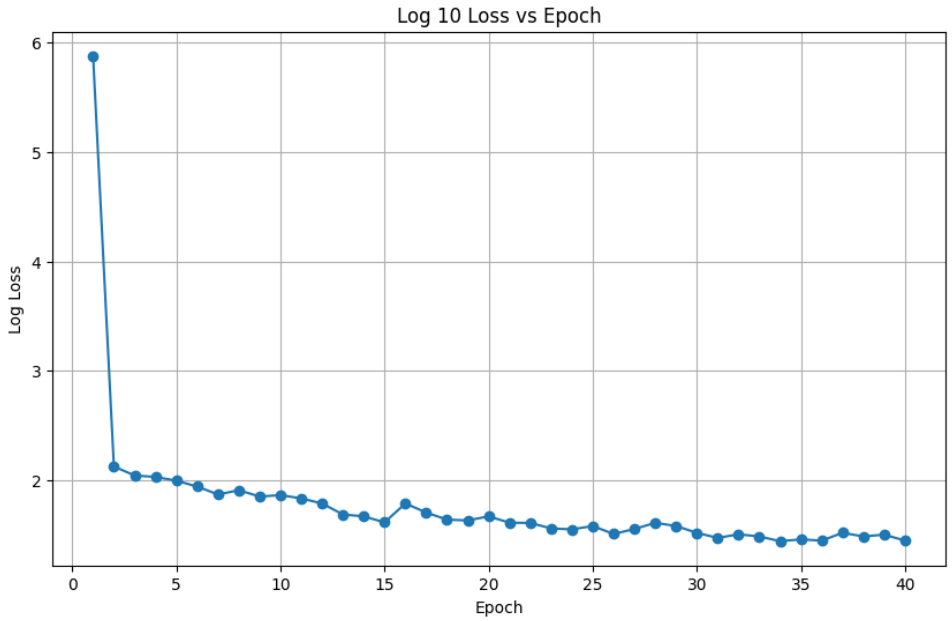

Figure 2: Log to the base 10 of Training Loss

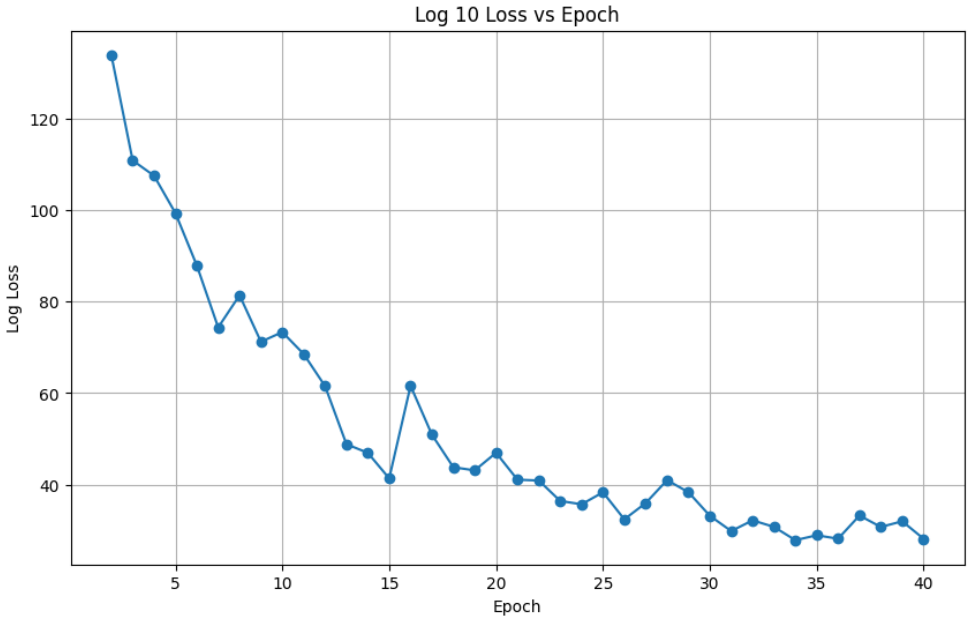

Figure 3: Training Loss vs Epoch starting from epoch 2

8 Results

8.1 Qualitative Results

8.1.2 Qualitative Description of Results

The results are impressive! We can see a significant improvement in quality of blurred images. The images blurred by kernels 2 and 3 of size 7*7 and 11*11 respectively are significantly improved. The images blurred by kernel of size 3*3 does not have much improvement. In fact, their PSNR has decreased but this is because PSNR is extremely sensitive for closely related images. So, a small deviation is enough to reduce PSNR of the model's prediction and thus, its reliability is questionable in this range.

8.2 Quantitative Results

8.2.1 Kernel 1

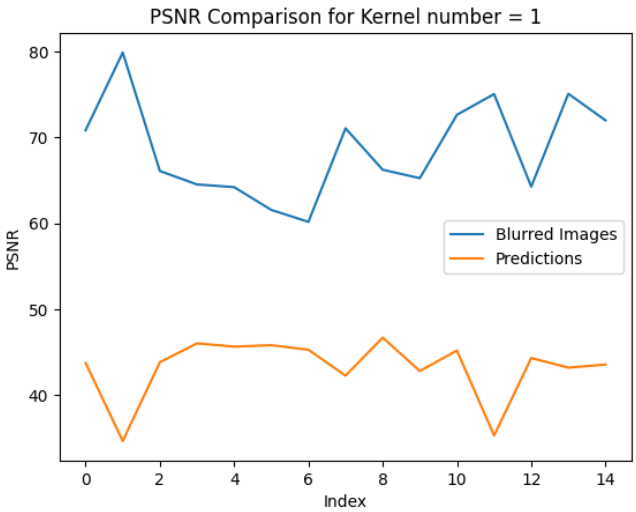

Figure 4: Prediction and Blurred Image PSNR for 15 random samples

- Average PSNR of Predicted Images: 43.24
- Average PSNR Improvement: -25.34

The Average PSNR value is although higher than kernel 2 and 3, it is lower than PSNR of the blurred image. This is because the kernel size and sigma for the Gaussian are very small. The PSNR formula is very sensitive for closely related images and is thus unreliable in this range.

8.2.2 Kernel 2

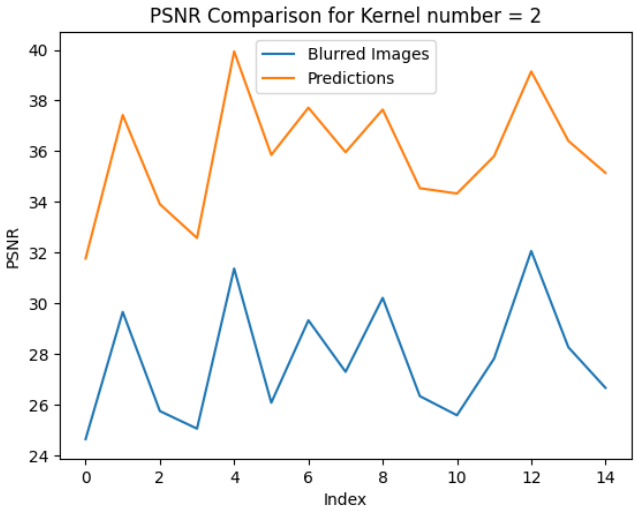

Figure 5: Prediction and Blurred Image PSNR for 15 random samples

- Average PSNR of Predicted Images: 35.88
- Average PSNR Improvement: 8.14

The results are very good in this category. There is a significant improvement in PSNR value.

8.2.3 Kernel 3

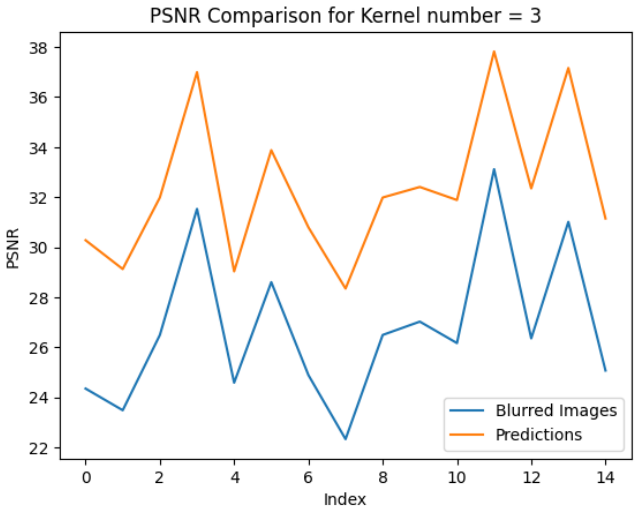

Figure 6: Prediction and Blurred Image PSNR for 15 random samples

- Average PSNR of Predicted Images: 32.35
- Average PSNR Improvement: 5.58

The results are decent in this category. There is a good improvement in PSNR value.

9 Evaluation on Test Data

- Average PSNR of Blurred Images: 26.68
- Average PSNR of Predicted Images: 31.06
- Average PSNR Improvement: 4.38

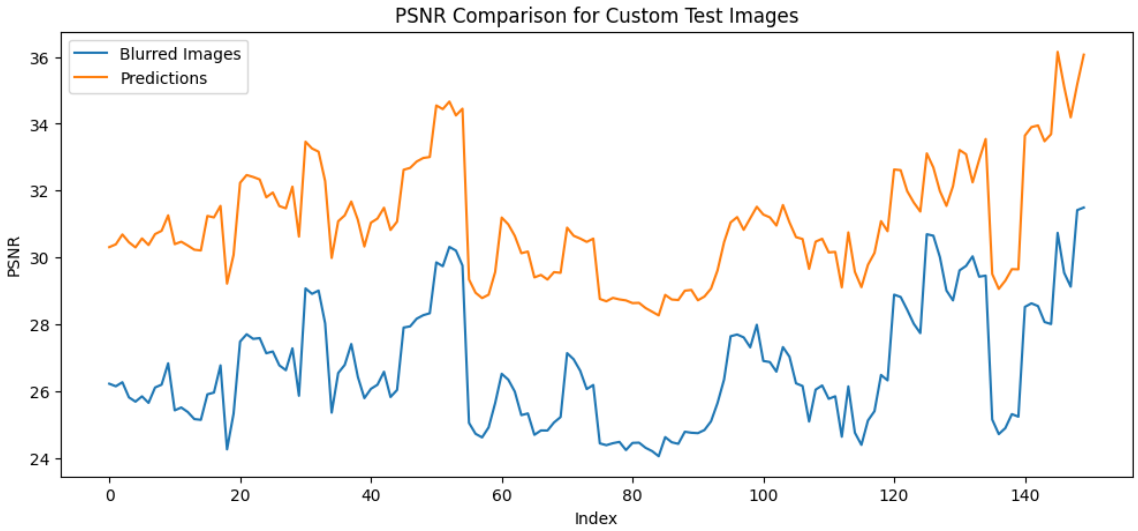

Figure 7: Prediction and Blurred Image PSNR for all Test Data

Figure 8: Prediction and Blurred Images for 4 random samples