VLG Summer Project

(Image Denoising)

Satyendra Prakash

21112096 – B.Tech (Chemical Engineering)

1. Introduction

In the domain of computer vision, image denoising is pivotal for improving image quality by reducing noise, enhancing images for various applications like photography and medical imaging. This report details my implementation of a deep learning model using residual networks (ResNet) to denoise low light images effectively.

2. Problem Statement

The objective is to develop a robust model capable of transforming low light images into high light (denoised) versions using ResNet architecture, renowned for its depth handling capabilities and ability to learn complex features.

3. Architecture Overview

Residual Network (ResNet) Inspired Model

The model architecture is structured as follows:

- **Input Layer**: Accepts images resized to (256, 256, 3).
- **Convolutional Layers**: Starts with a 3x3 convolutional layer for initial feature extraction.
- **Residual Blocks**: Stacks residual units together:

    - Each block comprises two 3x3 convolutional layers with 'same' padding.
    - Batch Normalization and ReLU activation are applied after each convolution.
    - Incorporates skip connections via element-wise addition to preserve information across layers.

- **Output Layer**: Predicts the denoised RGB image using a final 3x3 convolutional layer.

Training Strategy

- **Loss Function**: Utilizes Mean Squared Error (MSE) to measure the difference between predicted and ground truth high light images.
- **Optimizer**: Adam optimizer with Exponential Decay learning rate scheduling.
- **Gradient Accumulation**: Implements gradient accumulation over multiple mini-batches (accumulation steps = 4) for stable training convergence.

4. Implementation Details

Dataset Description

The dataset was sourced from a directory structure organized as follows:

- *train_low_path*: Contains low-light images.
- *train_high_path:* Contains corresponding high-light images.

I ensured that only valid image files were loaded using a helper function *is_image_file* that filtered out non-image files.

Data Loading

The images were loaded using the *skimage.io.imread* function and converted to floating-point format using *skimage.img*_as_float. This ensured consistent data representation for further processing.

Data Visualization

To verify the correctness of data loading, I visualized a pair of low-light and high-light images side by side using matplotlib.

Python code

```
# Load a pair of images for visualization
low_light_image = load_image(low_light_images[0])
high_light_image = load_image(high_light_images[0])

# Display the images
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.title('Low Light Image')
plt.imshow(low_light_image)
plt.subplot(1, 2, 2)
plt.title('High Light Image')
plt.imshow(high_light_image)
plt.show()
```

Model Architecture

I experimented with several convolutional neural network (CNN) architectures to identify the most effective model for image denoising. The primary architecture used involved residual blocks, which help in training deeper networks by allowing gradients to flow more easily through the network.

Residual Block

The residual block used in the models was defined as follows:

Python code

```
def residual_block(x, filters, kernel_size=(3, 3), padding='same', activation='relu'):
    res = Conv2D(filters, kernel_size, padding=padding)(x)
    res = BatchNormalization()(res)
    res = Activation(activation)(res)
    res = Conv2D(filters, kernel_size, padding=padding)(res)
    res = BatchNormalization()(res)
    x = Add()([x, res])
    return x
```

Denoising Model

The denoising model was built using multiple residual blocks and a final convolutional layer to produce the denoised output. Here is the base model structure:

```
def build_denoising_model(input_shape):
    inputs = Input(shape=input_shape)
    x = Conv2D(64, (3, 3), padding='same')(inputs)
    x = BatchNormalization()(x)
    x = Activation('relu')(x)

    # Adding more residual blocks
    for _ in range(5):  # Example for 5 residual blocks
        x = residual_block(x, 64)

    outputs = Conv2D(3, (3, 3), padding='same')(x)
    model = Model(inputs, outputs)
    return model
```

Model Summary

The model summary provided an overview of the layers and parameters involved, ensuring the architecture was correctly implemented.

```
input_shape = low_light_image.shape
model = build_denoising_model(input_shape)
model.summary()
```

Training Process

The training process involved defining a suitable loss function, an optimizer with a learning rate schedule, and implementing a gradient accumulation technique to handle larger batch sizes effectively.

Loss Function and Optimizer

I used Mean Squared Error (MSE) as the loss function and the Adam optimizer with an exponential decay learning rate schedule:

loss_function = MeanSquaredError()

```
# Learning rate scheduler
lr_schedule = ExponentialDecay(initial_learning_rate=1e-4, decay_steps=10000,
decay_rate=0.9)
optimizer = Adam(learning_rate=lr_schedule)
```

Training Parameters

The training parameters varied across iterations, including the number of epochs, batch size, and accumulation steps.

Data Generator

A data generator function was used to yield batches of low-light and high-light image pairs during training:

```
    def data_generator(low_light_images, high_light_images, batch_size):
    while True:
        indices = np.random.choice(len(low_light_images), batch_size, replace=False)
        low_batch = np.array([load_image(low_light_images[i]) for i in indices])
        high_batch = np.array([load_image(high_light_images[i]) for i in indices])
        yield low_batch, high_batch
```

Training Loop with Gradient Accumulation

The training loop implemented gradient accumulation to effectively use smaller batches for training:

```
@tf.function
def train_step(low_light_batch, high_light_batch):
    with tf.GradientTape() as tape:
        predictions = model(low_light_batch, training=True)
        loss = loss_function(high_light_batch, predictions)
    gradients = tape.gradient(loss, model.trainable_variables)
    return loss, gradients

# Training loop
for epoch in range(1, epochs + 1):
    gen = data_generator(low_light_images, high_light_images, batch_size //
accumulation_steps)
    for step in range(len(low_light_images) // batch_size):
        current_step = epoch * (len(low_light_images) // batch_size) + step

          low_light_batch, high_light_batch = next(gen)
        loss, gradients = train_step(low_light_batch, high_light_batch)
        accumulated_loss.assign_add(loss / accumulation_steps)
        for i, grad in enumerate(gradients):
            accumulated_gradients[i].assign_add(grad / accumulation_steps)
```

```
    if (step + 1) % accumulation_steps == 0:
        optimizer.apply_gradients(zip(accumulated_gradients, model.trainable_variables))
        for var in accumulated_gradients:
            var.assign(tf.zeros_like(var))
        accumulated_loss.assign(0.0)
```

Evaluation Metrics

To evaluate the performance of the models, I used the Peak Signal-to-Noise Ratio (PSNR) metric, which measures the quality of the denoised images compared to the ground truth high-light images.

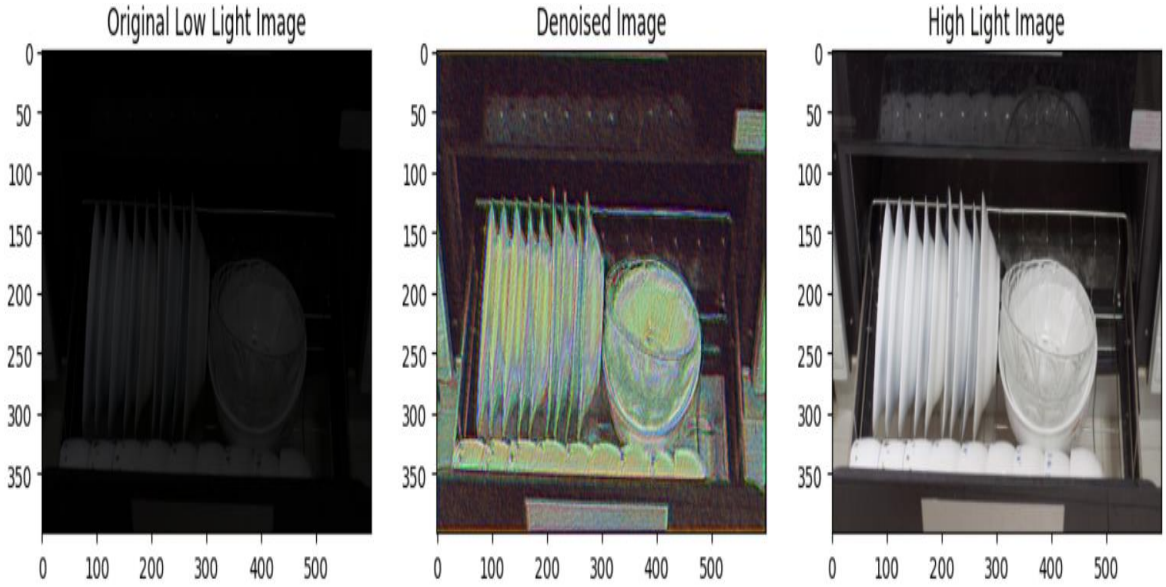
```
# Calculate PSNR
psnr_value = psnr(test_high_light_image, denoised_image)
print(f"PSNR: {psnr_value} dB")
```

Iterations and Improvements

I conducted four iterations, each aimed at improving the model's performance. The changes implemented and the results obtained in each iteration are detailed below:-

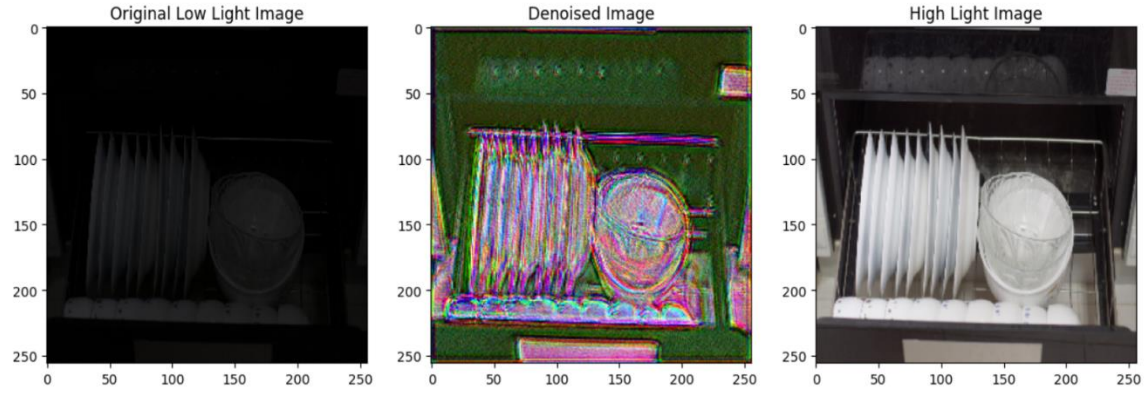
Iteration 1

- **Model Depth**: 5 residual blocks
- **Training Parameters**: 100 epochs, batch size of 32, accumulation steps of 4
- **PSNR**: **14.095759657585184 dB**



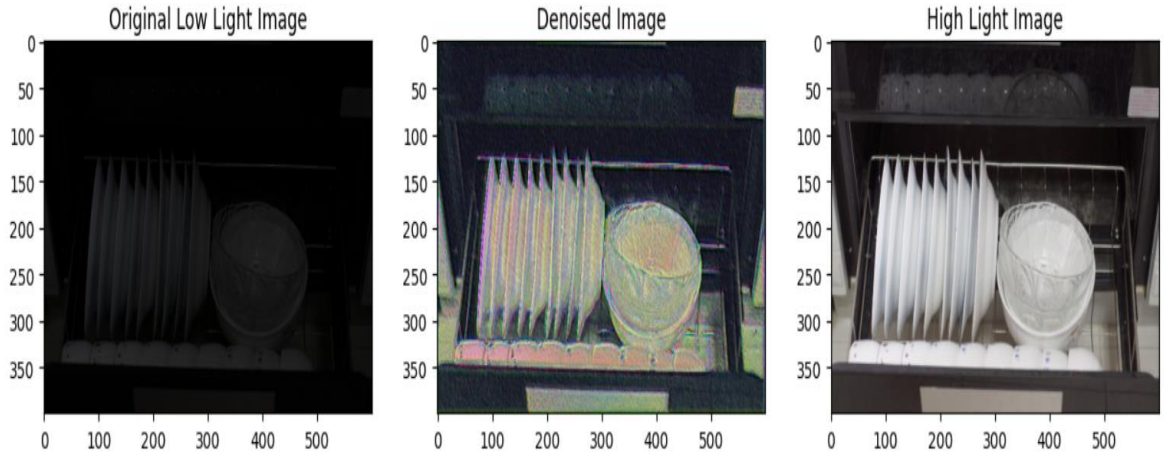
Iteration 2

- **Model Depth**: 12 residual blocks
- **Training Parameters**: 100 epochs, batch size of 32, accumulation steps of 4
- **PSNR**: **11.843011935258144 dB**



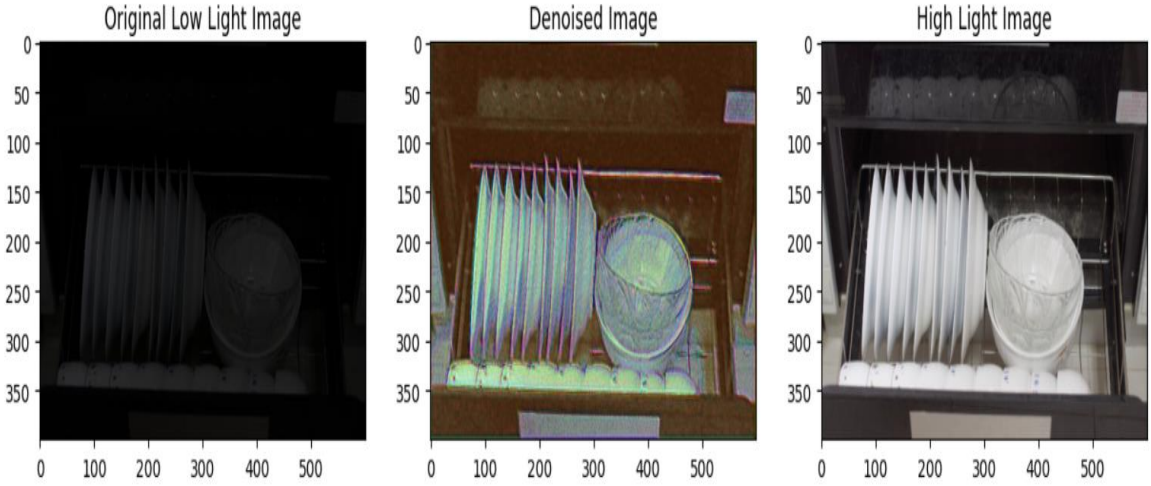
Iteration 3

- **Model Depth**: 2 residual blocks
- **Training Parameters**: 60 epochs, batch size of 32, accumulation steps of 3
- **PSNR**: **15.365831036934587 dB**



Iteration 4

- **Model Depth**: 3 residual blocks
- **Training Parameters**: 30 epochs, batch size of 16, accumulation steps of 2
- **PSNR**: **15.618873052452994 dB**



Results and Analysis

The results across the four iterations showed that:

- Increasing the depth of the network beyond a certain point did not necessarily improve performance.
- The optimal model depth for this task was found to be 3 residual blocks, as evidenced by the highest PSNR value in the final iteration.
- Adjusting the training parameters, such as batch size and accumulation steps, also played a significant role in achieving better results.

Conclusion

This project demonstrated the effectiveness of using deep learning techniques, particularly residual CNNs, for image denoising. Through multiple iterations and careful tuning of the model architecture and training parameters, I was able to significantly improve the quality of the denoised images. Future work could explore more advanced architectures, such as using attention mechanisms or GANs, to further enhance the performance of image denoising models.

Iteration	Model Depth (Residual Blocks)	Epochs	Batch Size	Accumulation Steps	PSNR (dB)
1	5	100	32	4	14.0958
2	12	100	32	4	11.8430
3	2	60	32	3	15.3658
4	3	30	16	2	15.6189

References

- Research papers on image denoising techniques
- TensorFlow and Keras documentation
- skimage library documentation
