Image Denoising using CNN-based Autoencoders 💡

Autoencoders are class of artificial neural networks designed for unsupervised learning, specifically CNN-based autoencoders are used for this image denoising task here. This model performs the task of improving the overall quality of an image making it clearer and more visually appealing. Image denoising is the process of removing random dots and specks called noise, photos taken in low light are generally noisy. CNN-based autoencoders outperforms traditional denoising methods like Non-Local Means because of its ability to capture spatial relationship and hierarchical representations in images and also it generalized well.

Introduction

In today's digital era, the quality and reliability of images play a crucial role in various domains, including medical imaging, surveillance, and satellite imaging. However, the presence of noise, characterized by random variations in brightness and color, hides important details and reduces the interpretability of an image. Traditional methods for image denoising, such as simple filters and Non-Local Means, often fall short in effectively removing noise while preserving image details. To overcome this my project leverages deep learning techniques, particularly CNN-based autoencoders, for image denoising tasks. It can learn the efficient encoding of input data, and then decoding this encoding to reconstruct the original image. for image denoising, an autoencoder can be trained to transform noisy images into their denoised version. Training is done on the dataset of noisy and clean image pairs, aiming to learn a mapping from noisy to denoised images.

Methodology

1. Data Preparation

Data Loading: The dataset comprises two directories: `low_dir` and `high_dir`. The `low_dir` contains low-resolution images, and the `high_dir` contains corresponding high-resolution images. Given the large size of the dataset, loading all images into memory at once could be impractical. To address this, the images are read and processed in smaller, manageable chunks.

Chunking Images: To facilitate efficient memory usage, a function was implemented to read and sort images numerically from the directories. This numerical sorting ensures that images are processed in a logical order. Once sorted, the images are read in groups of a specified size (e.g., 100 images per chunk). These chunks of images are then serialized and saved as `.pkl` files. This approach not only helps in managing memory but also makes it easier to load the images in subsequent steps.

Loading Chunks: For further processing, the serialized chunks of images are loaded back into memory. A function was created to deserialize the `.pkl` files and normalize the images by scaling pixel values to a range between 0 and 1. Normalization is a crucial preprocessing step that ensures the pixel values are within a consistent range, which can improve the performance of machine learning models.

```
def numerical_sort(value):
    numeric_part = ''.join(filter(str.isdigit, value))
    if numeric_part:
        return int(numeric_part)
    else:
        return float('inf')

def save_images_in_chunks(low_dir, high_dir, chunk_size=100):
    low_images = []
    high_images = []
    low_files = sorted(os.listdir(low_dir), key=numerical_sort)
    high_files = sorted(os.listdir(high_dir), key=numerical_sort)

    for i, (low_file, high_file) in enumerate(zip(low_files, high_files)):
        if low_file.endswith('.png') and high_file.endswith('.png'):
            low_img_path = os.path.join(low_dir, low_file)
            high_img_path = os.path.join(high_dir, high_file)
            low_images.append(imageio.v2.imread(low_img_path))
            high_images.append(imageio.v2.imread(high_img_path))

        if (i + 1) % chunk_size == 0 or (i + 1) == len(low_files):
            chunk_index = i // chunk_size
            with open(f'low_images_chunk_{chunk_index}.pkl', 'wb') as f:
                pickle.dump(low_images, f)
            with open(f'high_images_chunk_{chunk_index}.pkl', 'wb') as f:
                pickle.dump(high_images, f)
            print(f'Saved chunk {chunk_index}')
            low_images = []
            high_images = []
def load_images_from_chunks(low_chunk_files, high_chunk_files):
    low_images = []
    high_images = []
# pickle.load() is used to load previously serialized (pickled) images from
chunk files.
    for low_chunk_file, high_chunk_file in zip(low_chunk_files,
high_chunk_files):
        with open(low_chunk_file, 'rb') as f:
            chunk_low_images = pickle.load(f)
            low_images.extend(chunk_low_images)
        with open(high_chunk_file, 'rb') as f:
            chunk_high_images = pickle.load(f)
            high_images.extend(chunk_high_images)

    low_images_normalized = np.array(low_images) / 255.0
    high_images_normalized = np.array(high_images) / 255.0

    return low_images_normalized, high_images_normalized

low_dir = "/kaggle/input/train-dataset-main/low"
high_dir = "/kaggle/input/train-dataset-main/high"
save_images_in_chunks(low_dir, high_dir, chunk_size=100)

low_chunk_files = [f'low_images_chunk_{i}.pkl' for i in
range(len(os.listdir(low_dir)) // 100 + 1)]
high_chunk_files = [f'high_images_chunk_{i}.pkl' for i in
range(len(os.listdir(high_dir)) // 100 + 1)]
```

```
low_images_normalized, high_images_normalized =
load_images_from_chunks(low_chunk_files, high_chunk_files)
print(low_images_normalized.shape, high_images_normalized.shape)
```

2. Data Splitting

Splitting the Dataset: Once the images are loaded and normalized, the next step is to split the dataset into training, validation, and test sets. This is achieved using the `train_test_split` function from the `sklearn` library. The dataset is divided such that 70% of the images are used for training, 20% for validation, and 10% for testing. This stratification ensures that each subset of data is representative of the entire dataset, which is essential for building a robust model. The splitting function ensures that each set contains a diverse and representative sample of the data.

The function `split_in_chunks` was used to perform this split. This function iterates over each array (in this case, the low-resolution and high-resolution images) and splits them into training, validation, and test sets according to the specified proportions. By using a consistent random state, the function ensures reproducibility in the splits.

```
from sklearn.model_selection import train_test_split
def split_in_chunks(arrays, val_size=0.2, test_size=0.1, random_state=None):
    train_arrays = []
    val_arrays = []
    test_arrays = []
    for array in arrays:
        train_array, remaining_array = train_test_split(array,
test_size=val_size+test_size, random_state=random_state)
        val_array, test_array = train_test_split(remaining_array,
test_size=test_size/(val_size+test_size), random_state=random_state)
        train_arrays.append(train_array)
        val_arrays.append(val_array)
        test_arrays.append(test_array)
    return tuple(train_arrays), tuple(val_arrays), tuple(test_arrays)
arrays = [low_images_normalized, high_images_normalized]
train_arrays, val_arrays, test_arrays = split_in_chunks(arrays, val_size=0.2,
test_size=0.1, random_state=42)

train_low_images, train_high_images = train_arrays
val_low_images, val_high_images = val_arrays
test_low_images, test_high_images = test_arrays


print(f"Train low images shape: {train_low_images.shape}")
print(f"Validation low images shape: {val_low_images.shape}")
print(f"Test low images shape: {test_low_images.shape}")
print(f"Train high images shape: {train_high_images.shape}")
print(f"Validation high images shape: {val_high_images.shape}")
print(f"Test high images shape: {test_high_images.shape}")
```

3. Model Architecture

Autoencoder Definition: The core of the project is an autoencoder model designed for image denoising. An autoencoder consists of two main components: an encoder and a decoder.

Encoder: The encoder compresses the input image into a lower-dimensional representation. It comprises several convolutional layers with ReLU activation functions, which introduce non-linearity to capture complex patterns in the data. BatchNormalization layers are included to normalize the outputs of the convolutional layers, which helps stabilize and accelerate the training process. MaxPooling layers are used to progressively reduce the spatial dimensions of the image, effectively downsampling the input and retaining essential features.

Decoder: The decoder reconstructs the image from the encoded lower-dimensional representation. It mirrors the architecture of the encoder, using convolutional layers with ReLU activation functions and BatchNormalization. UpSampling layers are employed to restore the spatial dimensions of the image, essentially performing the reverse operation of MaxPooling. The final layer of the decoder uses a sigmoid activation function to ensure the output pixel values are between 0 and 1, which corresponds to the normalized input range.

```
# encoder
encoder_input = Input(shape = train_low_images.shape[1:])
x = Conv2D(32, (3,3), activation = 'relu', padding = 'same')(encoder_input)
x = BatchNormalization()(x)
x = MaxPool2D(pool_size = (2,2), padding = 'same')(x)
x = Conv2D(32, (3,3), activation = 'relu', padding = 'same')(x)
x = BatchNormalization()(x)
encoded = MaxPool2D(pool_size = (2,2), padding = 'same')(x)

# decoder
x = Conv2D(32, (3,3), activation = 'relu', padding = 'same')(encoded)
x = BatchNormalization()(x)
x = UpSampling2D()(x)
x = Conv2D(32, (3,3), activation = 'relu', padding = 'same')(x)
x = BatchNormalization()(x)
x = UpSampling2D()(x)
decoded = Conv2D(3, (3,3), activation = 'sigmoid', padding = 'same')(x)


autoencoder = Model(encoder_input, decoded, name = 'Denoising_Model')
autoencoder.summary()
```

```
Model: "Denoising_Model"
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━┓
┃ Layer (type)                         ┃ Output Shape                ┃         Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━┩
│ input_layer_1 (InputLayer)           │ (None, 400, 600, 3)         │               0 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ conv2d_5 (Conv2D)                    │ (None, 400, 600, 32)        │             896 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ batch_normalization_4                │ (None, 400, 600, 32)        │             128 │
│ (BatchNormalization)                 │                             │                 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ max_pooling2d_2 (MaxPooling2D)       │ (None, 200, 300, 32)        │               0 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ conv2d_6 (Conv2D)                    │ (None, 200, 300, 32)        │           9,248 │
```

batch_normalization_5 (BatchNormalization)	(None, 200, 300, 32)	128
max_pooling2d_3 (MaxPooling2D)	(None, 100, 150, 32)	0
conv2d_7 (Conv2D)	(None, 100, 150, 32)	9,248
batch_normalization_6 (BatchNormalization)	(None, 100, 150, 32)	128
up_sampling2d_2 (UpSampling2D)	(None, 200, 300, 32)	0
conv2d_8 (Conv2D)	(None, 200, 300, 32)	9,248
batch_normalization_7 (BatchNormalization)	(None, 200, 300, 32)	128
up_sampling2d_3 (UpSampling2D)	(None, 400, 600, 32)	0
conv2d_9 (Conv2D)	(None, 400, 600, 3)	867

```
 Total params: 30,019 (117.26 KB)
 Trainable params: 29,763 (116.26 KB)
 Non-trainable params: 256 (1.00 KB)
```

4. Model Training

Training Process: The autoencoder model is compiled using the `binary_crossentropy` loss function and the `adam` optimizer. The binary crossentropy loss function is appropriate for this task as it measures the discrepancy between the input images and the reconstructed output images on a pixel-by-pixel basis. The `adam` optimizer is selected for its efficiency and ability to handle sparse gradients on noisy data. The model is trained on the training dataset, with periodic evaluations on the validation dataset to monitor its performance and prevent overfitting.

```
from keras.callbacks import ModelCheckpoint
checkpoint = ModelCheckpoint("denoising_model.keras", save_best_only=True,
save_weights_only=False, verbose=1)
history = autoencoder.fit(train_low_images, train_high_images, batch_size=2,
epochs=10, callbacks=[checkpoint], validation_data=(val_low_images,
val_high_images), verbose=2)
```

5. Model Evaluation

Evaluation Metrics: To assess the quality of the denoised images, several metrics are used:

- **Peak Signal-to-Noise Ratio (PSNR):** PSNR is a widely used metric for measuring the quality of reconstructed images. It compares the similarity between the original high-resolution images and the denoised images. Higher PSNR values indicate better quality and less distortion.

- **Mean Squared Error (MSE):** MSE measures the average squared difference between the original and denoised images. Lower MSE values indicate that the denoised images are closer to the original images.
- **Mean Absolute Error (MAE):** MAE calculates the average absolute differences between the original and denoised images. Similar to MSE, lower MAE values signify better reconstruction quality.

The evaluation is performed on both the training and validation datasets, providing insights into the model's effectiveness and generalization ability.

```
def calculate_psnr(original, denoised):
    psnr_values = [psnr(original[i], denoised[i],
data_range=original[i].max() - original[i].min()) for i in
range(len(original))]
    average_psnr = np.mean(psnr_values)
    print(f"Average PSNR for training dataset is : {average_psnr}")
    return psnr_values
psnr_value_train = calculate_psnr(train_high_images,train_pred)
# Calculate Mean Squared Error (MSE)
mse = mean_squared_error(train_high_images.flatten(),train_pred.flatten())

# Calculate Mean Absolute Error (MAE)
mae = mean_absolute_error(train_high_images.flatten(),train_pred.flatten())

print("Mean Squared Error on the training dataset is:",mse)
print("Mean absolute Error on the training dataset is:",mae)
Average PSNR for training dataset is : 17.111931260656362
Mean Squared Error on the training dataset is: 0.024261177003964636
Mean absolute Error on the training dataset is: 0.123963512857033
```

6. Model Saving

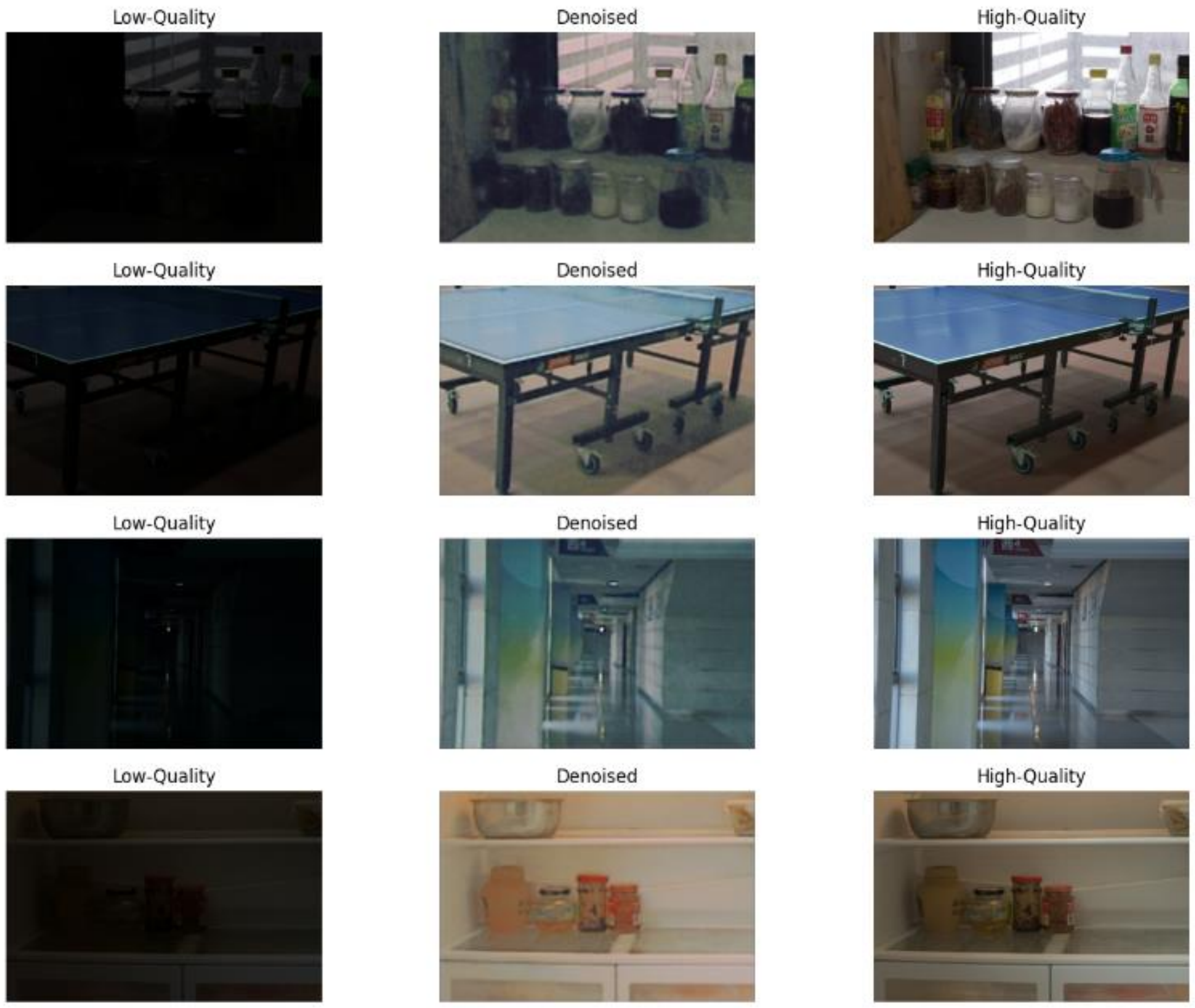
Saving the Model: After training, the model is saved for future use and deployment. This ensures that the trained model can be loaded and used without retraining, saving computational resources and time. The `keras.models.save_model` function is used to serialize the trained model and save it to disk.

```
from keras.models import load_model
autoencoder = load_model("denoising_model.keras")
autoencoder.summary()
```

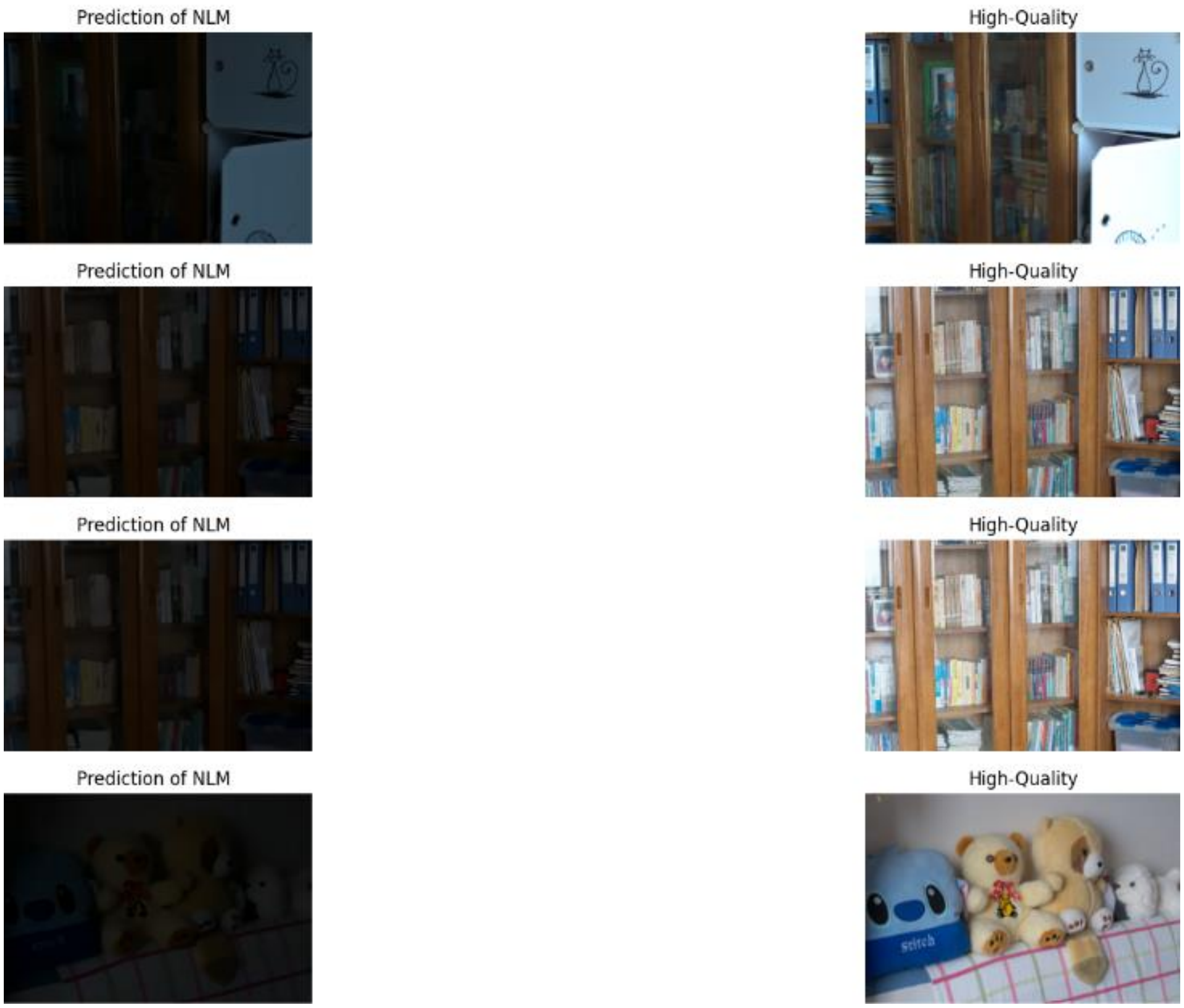
Result and Evaluation metrices

I have solved and submitted the results of denoising task with two methods of Non-local Means and by using CNN based Autoencoder, Non-local Means being the traditional approach gives the psnr of 7.68 whereas the use of neural network for the task of denoising improves the model very well and gives the training psnr value of 17.11. The prediction of both the models given the input denoised image are shown below:

CNN Autoencoders:



Non-local Means:



Resources/References

https://www.youtube.com/watch?v=ebPq0cILZV8

https://dergipark.org.tr/en/download/article-file/3535772

Author

Name: Mohit Sharma

Email-ID : mohit_s@ch.iitr.ac.in