Report

INTRO:

The goal of this project is to develop an image denoising algorithm utilizing deep learning techniques, with a particular emphasis on convolutional neural networks (CNNs). Image denoising is an essential preprocessing step in numerous image processing applications, intended to eliminate noise from images while maintaining crucial details.

Data Information

Our dataset comprises pairs of noisy and clean images. Each pair includes an image that has been corrupted with noise and its corresponding clean version.

Libraries and Modules

- TensorFlow/Keras: For building and training the model.
- skimage.metrics: For calculating PSNR.
- matplotlib: For plotting graphs.
- NumPy: For numerical operations.

Specification

- Framework: TensorFlow/Keras
- Layers: Multiple convolutional layers with activation functions, upsampling layers for increasing image dimensions back to the original size.
- Optimizer: Adam optimizer
- Loss Function: Binary-cross entropy

Architecture

In this project, we employed a convolutional neural network (CNN) architecture, which is widely used in image processing tasks for its ability to capture spatial hierarchies in images. Our CNN model includes multiple convolutional layers, each followed by activation and pooling layers, to progressively extract and condense image features.

- Convolutional Layers: Extract features from the input images.
- LeakyReLU Activation: Allows a small, non-zero gradient for negative inputs, preventing the "dying ReLU" problem and enhancing gradient flow.
- Pooling Layers: Downsample the feature maps to reduce spatial dimensions and computational load.
- Upsampling Layers: Restore the spatial dimensions of the feature maps to match the original image size.

Code snippets for model creation

Data Preprocessing and Cleaning

```
train_noisy, train_noisy, train_clean, train_clean = train_test_split(
    noisy_images, clean_images, test_size=0.2, random_state=42
)
```

✓ 0.1s

```
def preprocess_image(image):
    return tf.image.convert_image_dtype(image, tf.float32)

# Create TensorFlow datasets
train_noisy_ds = tf.data.Dataset.from_tensor_slices(train_noisy).map(preprocess_image).batch(32)
train_clean_ds = tf.data.Dataset.from_tensor_slices(train_clean).map(preprocess_image).batch(32)
test_noisy_ds = tf.data.Dataset.from_tensor_slices(test_noisy).map(preprocess_image).batch(32)
test_clean_ds = tf.data.Dataset.from_tensor_slices(test_clean).map(preprocess_image).batch(32)

# Combine noisy and clean images for training
train_ds = tf.data.Dataset.zip((train_noisy_ds, train_clean_ds))
test_ds = tf.data.Dataset.zip((test_noisy_ds, test_clean_ds))
```

✓ 53.9s

We have divided the data into train and test data, with train : test split being 80% : 20% respectively, and then created train_ds and test_ds which are used to train our custom built model with a batch size of 32.

- Model Layers and Parameters

Layer (type)	Output Shape	Param #
conv2d_5 (Conv2D)	(None, 128, 128, 64)	1,792
max_pooling2d_2 (MaxPooling2D)	(None, 64, 64, 64)	0
conv2d_6 (Conv2D)	(None, 64, 64, 32)	18,464
max_pooling2d_3 (MaxPooling2D)	(None, 32, 32, 32)	0
conv2d_7 (Conv2D)	(None, 32, 32, 32)	9,248
up_sampling2d_2 (UpSampling2D)	(None, 64, 64, 32)	0
conv2d_8 (Conv2D)	(None, 64, 64, 64)	18,496
up_sampling2d_3 (UpSampling2D)	(None, 128, 128, 64)	0
conv2d_9 (Conv2D)	(None, 128, 128, 3)	1,731

Total params: 49,731 (194.26 KB)

Trainable params: 49,731 (194.26 KB)

Non-trainable params: 0 (0.00 B)

The model consists of 5 convolutional layers, 2 max pooling layers, and 2 sampling layers, which are essential components of the architecture. This setup is designed to remove noise from low-quality noisy images, converting them into higher quality images that closely resemble the original clean, noise-free versions.

- Model Creation, Optimisers and CallBacks

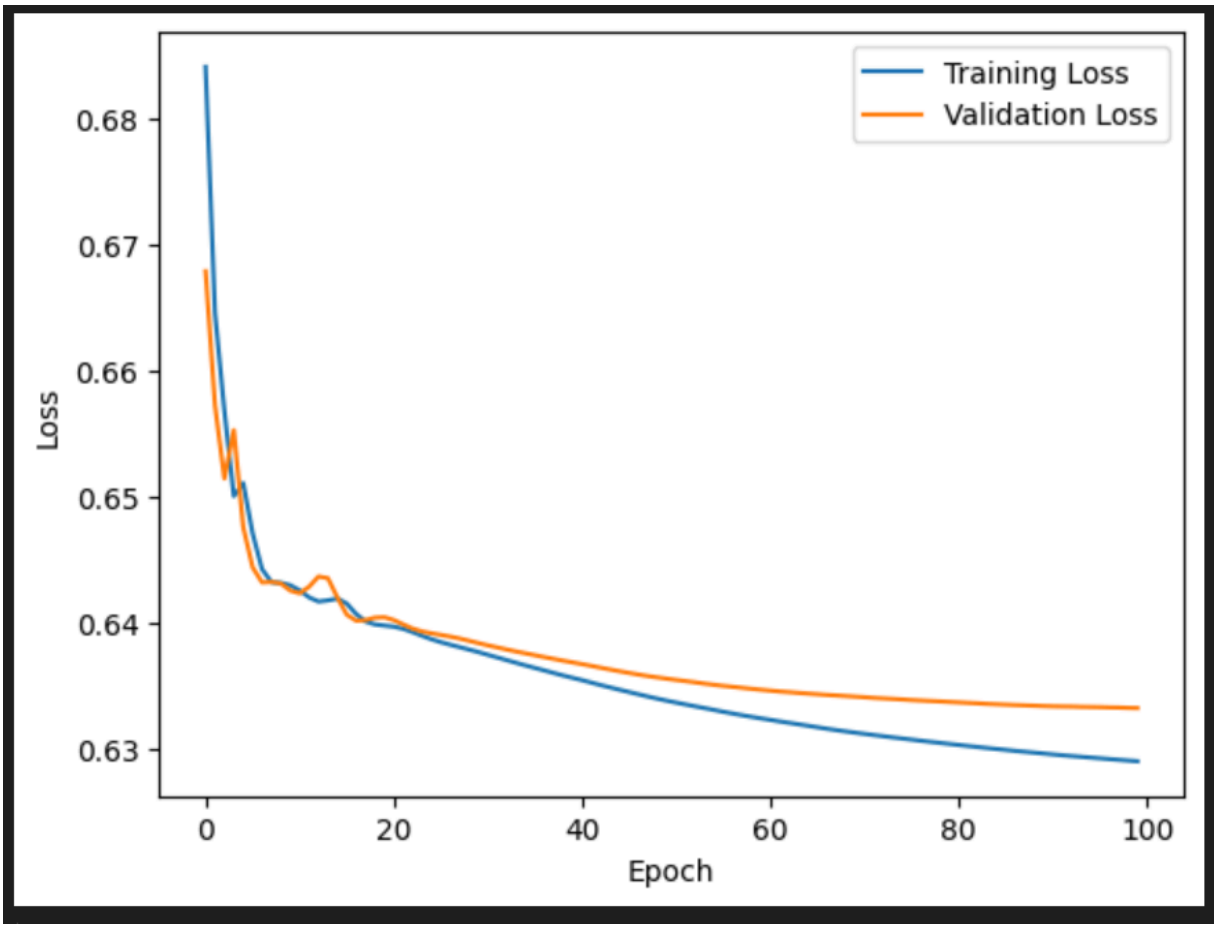
```python
# Define the denoising model
def create_denoising_model(input_shape):
    model = Sequential()
    model.add(layers.Input(shape=input_shape))
    model.add(layers.Conv2D(64, (3, 3), activation=LeakyReLU(), padding='same'))
    model.add(layers.MaxPooling2D((2, 2), padding='same'))
    model.add(layers.Conv2D(32, (3, 3), activation=LeakyReLU(), padding='same'))
    model.add(layers.MaxPooling2D((2, 2), padding='same'))
    model.add(layers.Conv2D(32, (3, 3), activation=LeakyReLU(), padding='same'))
    model.add(layers.UpSampling2D((2, 2)))
    model.add(layers.Conv2D(64, (3, 3), activation=LeakyReLU(), padding='same'))
    model.add(layers.UpSampling2D((2, 2)))
    model.add(layers.Conv2D(3, (3, 3), activation='sigmoid', padding='same'))
    return model

# Example usage
input_shape = (128, 128, 3)
model = create_denoising_model(input_shape)
# Compile with Adam optimizer and monitor PSNR during training
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=[psnr])
model.summary()

# Assuming you have training and validation datasets: train_ds and test_ds
val_data = next(iter(test_ds))  # Extract a batch from the validation dataset

# Instantiate the custom PSNR callback with validation data
psnr_callback = PSNRCallback(validation_data=val_data)
```

To monitor the PSNR values after each epoch and ensure the highest PSNR score, we utilized callbacks. Binary cross-entropy was employed as the loss function, with Adam serving as the optimizer.

Results and Findings from Cleaned and Denoised Images

We calculated the loss using the binary cross-entropy loss function and mean squared error (MSE) by varying the number of epochs and batch size. The binary cross-entropy yielded better results, with a lower loss value, when using 100 epochs, a batch size of 32, and a train-to-test ratio of 8:2.

```
Epoch 97/100
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 186ms/stepep - loss: 0.6326 - psnr: 17.80
Epoch 97: val_psnr = 17.468788146972656
13/13 ━━━━━━━━━━━━━━━━━━━━ 6s 458ms/step - loss: 0.6324 - psnr: 17.8073 - val_loss: 0.6333 - val_psnr: 17.4688
Epoch 98/100
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 176ms/stepep - loss: 0.6326 - psnr: 17.81
Epoch 98: val_psnr = 17.470565795898438
13/13 ━━━━━━━━━━━━━━━━━━━━ 6s 456ms/step - loss: 0.6323 - psnr: 17.8134 - val_loss: 0.6333 - val_psnr: 17.4706
Epoch 99/100
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 182ms/stepep - loss: 0.6325 - psnr: 17.81
Epoch 99: val_psnr = 17.47403335571289
13/13 ━━━━━━━━━━━━━━━━━━━━ 6s 453ms/step - loss: 0.6323 - psnr: 17.8194 - val_loss: 0.6333 - val_psnr: 17.4740
Epoch 100/100
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 180ms/stepep - loss: 0.6324 - psnr: 17.82
Epoch 100: val_psnr = 17.47610855102539
13/13 ━━━━━━━━━━━━━━━━━━━━ 6s 458ms/step - loss: 0.6322 - psnr: 17.8251 - val_loss: 0.6333 - val_psnr: 17.4761
```



This plot displays the training and validation loss over the epochs, providing a visualization of the model's performance throughout the training process.

PSNR Calculation

- Binary-cross entropy: The psnr() function calculates the loss between the clean and denoised images.
- Peak Signal-to-Noise Ratio (PSNR): Once loss is computed, PSNR is derived from it. PSNR measures the quality of the denoised image by comparing it to the original clean image. Higher PSNR values indicate higher fidelity and less noise in the denoised image.

Calculation of average PSNR

For Training Data

18.03 [loss-binary cross-entropy,epoch=100]

17.35 [loss-binary cross-entropy,epoch=50]

For Testing Data

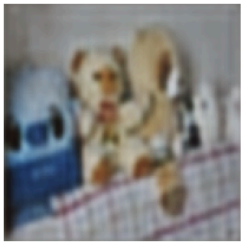
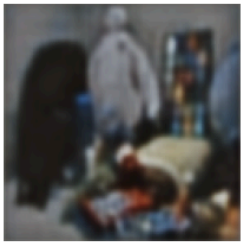
17.74 [loss-binary cross-entropy,epoch=100]

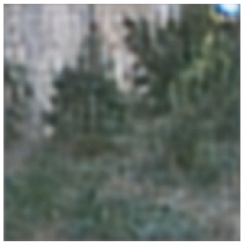
17.21 [loss-binary cross-entropy,epoch=50]

17.3698 [loss-MSE,epoch=50]

Results and Discussions

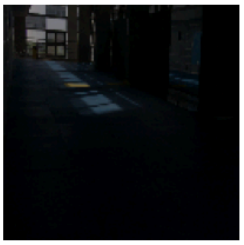
Predicted image:



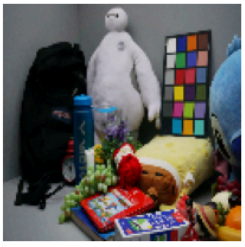
Noisy images:




Clean images:





Sample Images:

(noisy)

(predicted)

(clean)

PSNR Score

The PSNR value obtained for the denoised images using our CNN model is

For the Test Data : 17.737285614013672 dB

For the Train Data : 18.038999557495117 dB.

This indicates a significant reduction in noise and a good preservation of image details.

Scope for improvement

Data Augmentation: Introduce additional variations in the training data (e.g., rotation, scaling, flipping) to improve the model's robustness.

Advanced Architectures: Investigate more advanced models like U-Net, DnCNN, or GAN-based architectures for potentially enhanced denoising performance.

Hyperparameter Tuning: Experiment with various learning rates, batch sizes, and numbers of epochs to optimize the model's performance.

Transfer Learning: Utilize pre-trained models and fine-tune them for the denoising task to take advantage of pre-learned features.