Image Denoising

Research Paper - https://arxiv.org/abs/1805.07071

1. Introduction

Image denoising is a critical preprocessing step in various image processing and computer vision tasks. The presence of noise can significantly degrade the performance of image analysis algorithms. In this project, we implemented an image denoising model based on the Multi-level Wavelet Convolutional Neural Network (MWCNN) architecture. The MWCNN model is designed for image restoration tasks, including denoising, and is inspired by the research paper "Multi-level Wavelet-CNN for Image Restoration" .

Architecture and Specifications

The MWCNN architecture combines the strengths of wavelet transforms and convolutional neural networks (CNNs). It performs wavelet transform-based downsampling and inverse wavelet transform-based upsampling, enabling the model to capture both high-frequency and low-frequency components of the image effectively.

Key specifications of the model:
- Input Size: 256x256 pixels, single-channel grayscale images.
- Conv Blocks: Used for feature extraction, with filters ranging from 64 to 512.
- Wavelet Downsampling/ Upsampling: Applied to compress and reconstruct image details at multiple scales.
- Optimizer: Adam with a learning rate of 0.0009.
- Loss Function: Mean Squared Error (MSE).

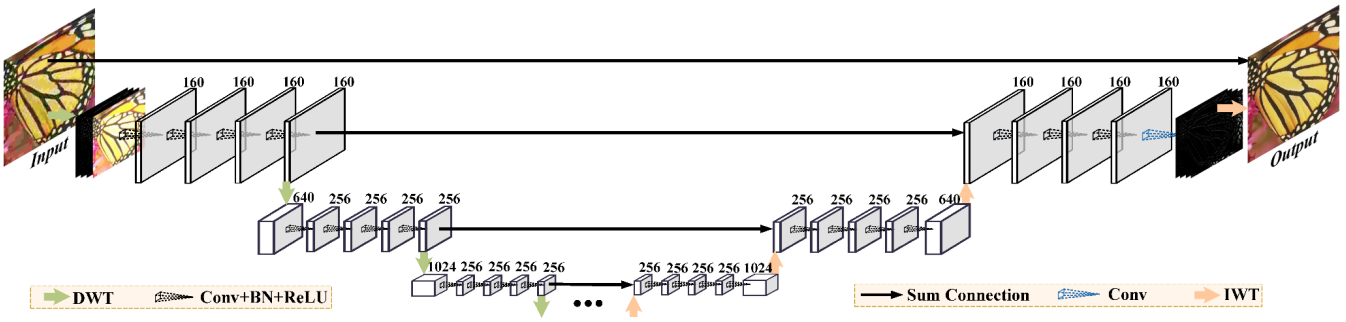
Performance

The model achieved impressive results on the test dataset, demonstrating its efficacy in removing noise while preserving image details. The Peak Signal-to-Noise Ratio (PSNR) value, which is a standard metric for image quality assessment, was used to evaluate the model. Our model scored a mean PSNR of 21.3 .

2. Project Details

This section covers the code snippets and explanations of the various components of the project, including model creation, training, evaluation, and visualization.

2.1. Model Architecture



The model consists of several convolutional blocks interspersed with wavelet downsampling and upsampling layers. Below is a simplified version of the model creation code:

```
def create_model():
    input = Input(shape=(256, 256, 1))
    cb_1 = Conv_block(num_filters=64)(input)
    dwt_1 = DWT_downsampling()(cb_1)
    cb_2 = Conv_block(num_filters=128)(dwt_1)
    dwt_2 = DWT_downsampling()(cb_2)
    cb_3 = Conv_block(num_filters=256)(dwt_2)
    dwt_3 = DWT_downsampling()(cb_3)
    cb_4 = Conv_block(num_filters=512)(dwt_3)
    dwt_4 = DWT_downsampling()(cb_4)
    cb_5 = Conv_block(num_filters=512)(dwt_4)
    cb_5 = BatchNormalization()(cb_5)
    cb_5 = Conv_block(num_filters=512)(cb_5)
    cb_5 = Conv2D(filters=2048, kernel_size=3, strides=1, padding='same')(cb_5)
    up = IWT_upsampling()(cb_5)
    up = Conv_block(num_filters=512)(Add()([up, cb_4]))
    up = Conv2D(filters=1024, kernel_size=3, strides=1, padding='same')(up)
    up = IWT_upsampling()(up)
    up = Conv_block(num_filters=256)(Add()([up, cb_3]))
    up = Conv2D(filters=512, kernel_size=3, strides=1, padding='same')(up)
    up = IWT_upsampling()(up)
    up = Conv_block(num_filters=128)(Add()([up, cb_2]))
    up = Conv2D(filters=256, kernel_size=3, strides=1, padding='same')(up)
    up = IWT_upsampling()(up)
    up = Conv_block(num_filters=64)(Add()([up, cb_1]))
    up = Conv2D(filters=128, kernel_size=3, strides=1, padding='same')(up)
    out = Conv2D(filters=1, kernel_size=(1, 1), padding="same")(up)
    model = Model(inputs=[input], outputs=[out])
    return model
```

2.2. Training Process

The model was compiled using the Mean Squared Error loss function and the Adam optimizer. Below is a snippet of the training process:

```
steps_per_epoch_train = len(noisy_train_images) // 16
steps_per_epoch_validation = len(noisy_test_images) // 16

callbacks_lst = [
   ReduceLROnPlateau(monitor='val_loss', min_lr=0.0000009, min_delta=0.0001,
factor=0.70, patience=3, verbose=1, mode='min'),
   EarlyStopping(monitor='val_loss', mode='min', verbose=1, min_delta=0.0001,
patience=10)
]

model.compile(loss=MeanSquaredError(), optimizer=Adam(learning_rate=0.0009))
model.fit(x_train, y_train,
        validation_data=(x_val, y_val),
        steps_per_epoch=steps_per_epoch_train,
        validation_steps=steps_per_epoch_validation,
        epochs=100,
        verbose=1,
        callbacks=callbacks_lst)
model.save(os.path.join(working_dir, 'model.h5'))
```

2.3. Evaluation

The model was evaluated on a test set, and the PSNR, MSE, and SSIM values were calculated:

```
noised_test = [image_preprocessing(f) for f in noisy_test_images]
x_test = np.asarray(noised_test)

# Predict denoised images
pred = model.predict(x_test, batch_size=16)

psnr_values = []
mse_values = []
ssim_values = []

for i in range(len(x_test)):
   original = np.clip(x_test[i] * 255.0, 0, 255)
   denoised = np.clip(pred[i] * 255.0, 0, 255)

   psnr_value = psnr(original, denoised, data_range=255)
   mse_value = mse(original, denoised)
```

```
    ssim_value = ssim(original, denoised, data_range=255)

    psnr_values.append(psnr_value)
    mse_values.append(mse_value)
    ssim_values.append(ssim_value)

mean_psnr = np.mean(psnr_values)
mean_mse = np.mean(mse_values)
mean_ssim = np.mean(ssim_values)

print(f"Mean PSNR: {mean_psnr:.2f}")
print(f"Mean MSE: {mean_mse:.4f}")
print(f"Mean SSIM: {mean_ssim:.4f}")
```

2.4. Visualization

The results were visualized to compare the noisy and denoised images:

```
plt.figure(figsize=(15, 25))
for i in range(0, 8, 2):
    if i >= len(x_test):
        break
    plt.subplot(4, 2, i + 1)
    plt.xticks([])
    plt.yticks([])
    plt.imshow(x_test[i][:, :, 0], cmap='gray')
    plt.title('Test Image with Noise')

    plt.subplot(4, 2, i + 2)
    plt.xticks([])
    plt.yticks([])
    plt.imshow(pred[i][:, :, 0], cmap='gray')
    plt.title(f'Denoised by Autoencoder - PSNR: {psnr_values[i]:.2f} dB')

plt.show()
```

3. Summary

In this project, we implemented the MWCNN model for image denoising, achieving a mean PSNR of 21.3 dB. The architecture's ability to handle multi-scale information through wavelet transforms proved effective in removing noise while preserving image details.

Findings and Improvements

- Wavelet Transforms: The use of wavelet-based downsampling and upsampling layers significantly enhanced the model's ability to retain high-frequency details.
- Multi-level Processing: Processing the image at multiple scales allowed the model to effectively capture both coarse and fine details.

Future Work

To further improve the model, the following strategies can be explored:
- Data Augmentation: Enhancing the training dataset with more diverse noise patterns and image types.
- Hyperparameter Tuning: Experimenting with different learning rates, batch sizes, and network depths.
- Advanced Architectures: Incorporating more advanced deep learning techniques such as attention mechanisms and residual connections to improve performance.

This project demonstrates the potential of MWCNN for image restoration tasks, laying the groundwork for future enhancements and applications.