IMAGE DENOISING USING RESNET BASED MODEL

INTRODUCTION:

Image denoising is a fundamental task in image processing aimed at removing noise from an image to recover the original clean image. This process is crucial in various applications, including medical imaging, photography, and remote sensing, where image quality is paramount. Recently, deep learning techniques, particularly Convolutional Neural Networks (CNNs), have shown remarkable success in image denoising tasks. Among these, Residual Networks (ResNets) have emerged as a powerful tool due to their ability to efficiently train deep architectures.

MODEL ARCHITECTURE:

```python
# Building class for model building with required functions
class ChannelAttention(Layer):
    def __init__(self, filters, ratio=16):
        super(ChannelAttention, self).__init__()
        self.filters = filters
        self.ratio = ratio
        self.avg_pool = layers.GlobalAveragePooling2D()
        self.max_pool = layers.GlobalMaxPooling2D()
        self.fc1 = layers.Conv2D(filters // ratio, 1, activation='relu', padding='same')
        self.fc2 = layers.Conv2D(filters, 1, padding='same')

    def call(self, x):
        avg_out = self.fc2(self.fc1(tf.expand_dims(tf.expand_dims(self.avg_pool(x), 1), 1)))
        max_out = self.fc2(self.fc1(tf.expand_dims(tf.expand_dims(self.max_pool(x), 1), 1)))
        return x * tf.sigmoid(avg_out + max_out)

class SpatialAttention(Layer):
    def __init__(self, kernel_size=7):
        super(SpatialAttention, self).__init__()
        self.conv = layers.Conv2D(1, kernel_size, activation='sigmoid', padding='same')

    def call(self, x):
        avg_out = tf.reduce_mean(x, axis=-1, keepdims=True)
        max_out = tf.reduce_max(x, axis=-1, keepdims=True)
        return x * self.conv(tf.concat([avg_out, max_out], axis=-1))

def conv_block(x, filters, kernel_size, strides, activation='relu'):
    x = layers.Conv2D(filters, kernel_size, strides=strides, padding='same')(x)
    if activation:
        x = layers.Activation(activation)(x)
    return x

def residual_block(x, filters):
    shortcut = x
    x = conv_block(x, filters, 3, 1, activation='relu')
    x = conv_block(x, filters, 3, 1, activation=None)
    x = ChannelAttention(filters)(x)
    x = SpatialAttention()(x)
    x = layers.Add()([x, shortcut])
    x = layers.Activation('relu')(x)
    return x
```

Above is a code snippet of the model architecture I used in this project and I'll

Step by step explain how this model architecture works.

Class: `ChannelAttention`

Function:

The Channel Attention mechanism enhances the model's ability to focus on important features across different channels (filters) of the input tensor.

Detailed Breakdown:

1. **Global Average Pooling (`avg_pool`)**: Computes the average value for each channel across all spatial locations. This produces a feature vector summarizing the global context of each channel.
    - Input: A tensor of shape `(batch_size, height, width, channels)`.
    - Output: A tensor of shape `(batch_size, channels)`.
2. **Global Max Pooling (`max_pool`)**: Computes the maximum value for each channel across all spatial locations. This also produces a feature vector summarizing the most prominent features of each channel.

- Input: Same as above.
- Output: Same as above.

3. **Fully Connected Layers (`fc1` and `fc2`)**:

- `fc1`: Reduces the number of channels by a factor of `ratio` (to reduce complexity and focus on the most significant features).
- `fc2`: Restores the number of channels back to the original count.
- Both pooling outputs (average and max) are passed through these layers independently.

4. **Combining Average and Max Features**:

- The outputs of `fc2` from both the average and max pathways are added together.
- A sigmoid activation is applied to produce an attention map with values between 0 and 1, indicating the importance of each channel.

5. **Element-wise Multiplication**:

- The input tensor `x` is multiplied element-wise with the attention map, effectively enhancing important channels and suppressing less important ones.

Class: `SpatialAttention`

Function:

The Spatial Attention mechanism enhances the model's ability to focus on important spatial locations within the feature maps.

Detailed Breakdown:

1. **Computing Channel-wise Statistics**:
    - `avg_out`: Computes the average across the channel dimension for each spatial location.
    - `max_out`: Computes the maximum across the channel dimension for each spatial location.
    - Input: A tensor of shape `(batch_size, height, width, channels)`.
    - Output: Two tensors of shape `(batch_size, height, width, 1)`.
2. **Concatenation**:

- The two tensors (`avg_out` and `max_out`) are concatenated along the channel dimension.
- Output: A tensor of shape `(batch_size, height, width, 2)`.

3. **Convolution Layer (`conv`)**:

- Applies a convolution with a 1x1 kernel and sigmoid activation to produce a spatial attention map.
- The attention map has values between 0 and 1, indicating the importance of each spatial location.
- Output: A tensor of shape `(batch_size, height, width, 1)`.

4. **Element-wise Multiplication**:

- The input tensor `x` is multiplied element-wise with the spatial attention map, enhancing important spatial locations and suppressing less important ones.

Function: `conv_block`

Function:

The `conv_block` function simplifies the creation of a convolutional layer followed by an optional activation function, used to extract features from the input tensor.

Detailed Breakdown:

1. **Convolution Layer**:
    - Applies a 2D convolution with specified parameters (`filters`, `kernel_size`, `strides`).
    - Pads the input to maintain spatial dimensions (padding='same').
2. **Activation Function**:

- If an activation function is specified, it is applied to the convolution output.

Function: `residual_block`

Function:

The `residual_block` function defines a residual block, combining convolutional layers, channel attention, spatial attention, and a skip connection. This block is essential for building deep networks that can learn complex mappings while mitigating issues like vanishing gradients.

Now we'll look at how the workflow of this architecture is, how input 'x'(noisy) is taken to 'x_denoised' through different steps in this residual block which is created using Channel_attention, spatial_attention and conv_block as described above.

The input tensor `x` is stored in `shortcut` to be added back later (residual connection).

- Applies a convolution followed by ReLU activation.
- Input: Tensor `x`.

- Output: Intermediate feature map.

- Applies another convolution without activation.
- Input: Intermediate feature map from the first block.

- Output: Feature map ready for attention mechanisms.

- Enhances the feature map by focusing on important channels.
- Input: Feature map from the second convolution block.

- Output: Channel-attended feature map.

- Further enhances the feature map by focusing on important spatial locations.
- Input: Channel-attended feature map.

- Output: Spatially attended feature map.

- Adds the original input tensor (`shortcut`) to the attended feature map.

- This addition helps the network learn residuals, which are easier to optimize.

- Applies a ReLU activation to the combined feature map.

- Output: The final output of the residual block, ready to be passed to the next layer.

We'll also take a look at how this model performed in actual research paper published:

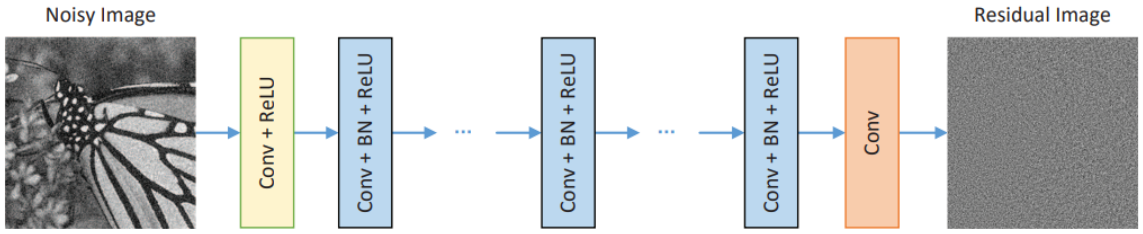
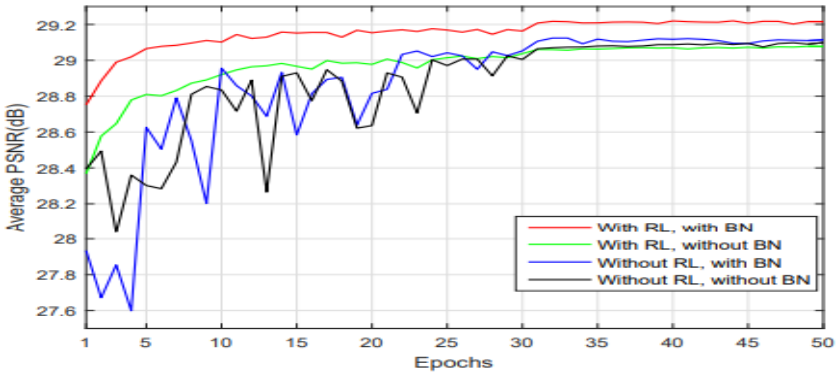


Fig. 1. The architecture of the proposed DnCNN network.

This shows the workflow of this model and essentially I've made some small changes to this and used only 10 residual blocks because of less computation power.



(b) Adam

This is the reason why I used adam optimizer instead of SGD(stochastic gradient descent), because adam converges much faster.

Noise Level	$\sigma = 50$												
BM3D [2]	26.13	29.69	26.68	25.04	25.82	25.10	25.90	29.05	27.22	26.78	26.81	26.46	26.722
WNNM [13]	26.45	**30.33**	26.95	25.44	26.32	25.42	26.14	29.25	**27.79**	26.97	26.94	26.64	27.052
EPLL [33]	26.10	29.12	26.80	25.12	25.94	25.31	25.95	28.68	24.83	26.74	26.79	26.30	26.471
MLP [24]	26.37	29.64	26.68	25.43	26.26	25.56	26.12	29.32	25.24	27.03	27.06	26.67	26.783
TNRD [16]	26.62	29.48	27.10	25.42	26.31	25.59	26.16	28.93	25.70	26.94	26.98	26.50	26.812
DnCNN-S	**27.03**	30.00	27.32	25.70	26.78	25.87	**26.48**	**29.39**	26.22	27.20	**27.24**	26.90	27.178
DnCNN-B	**27.03**	30.02	**27.39**	**25.72**	**26.83**	**25.89**	**26.48**	29.38	26.38	**27.23**	27.23	**26.91**	**27.206**

ResNet performs much better on higher gaussian noise (sigma = 50).

Our initial data psnr is 7.76 which is considerably low, and as ResNet performs better on higher noise and can handle many types of noise, this is why I opted for ResNet.

Now I'll be explaining the code snippets.

EXPLAINING CODE SNIPPETS:

```
#Loading Libraries
import os
import numpy as np
from PIL import Image
import matplotlib.pyplot as plt
import cv2
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.layers import Layer
from tensorflow.keras import layers, models, optimizers, losses
from tensorflow.keras.layers import Conv2D, BatchNormalization, MaxPooling2D, UpSampling2D, Dropout, Concatenate, Input , Add, Activation
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau
from patchify import patchify, unpatchify
```

In this we imported all the libraries necessary for data handling, model building and training and then evaluation.

```
[ ]  # Defining function to extract images from traget folder
     def load_images_from_folder(folder):
         images = []
         for filename in os.listdir(folder):
             if filename.endswith(".jpg") or filename.endswith(".png"):
                 img_path = os.path.join(folder, filename)
                 img = Image.open(img_path)
                 img = img.resize((256,256))
                 img = img.convert('RGB')
                 img = np.array(img)
                 images.append(img)

         images = np.array(images)
         images = images/255
         return images
```

```
[ ]  #Location of image folder
     noisy = r"C:/Users/prajw/OneDrive/Documents/DenoisingProject/All/low"
     clean = r"C:/Users/prajw/OneDrive/Documents/DenoisingProject/All/high"
```

```
[ ]  # x-->Low and noisy images
     # y-->High and non-noisy target images
     x = load_images_from_folder(noisy)
     y = load_images_from_folder(clean)
```

First we loaded the data using load_images_from_folder function which is valid for .jpg and .png formats. Then we resized the images to 256X256 and kept in RGB format which will give us a numpy.ndarry format of 3 dimensions for RGB. And then we saved them in x(noisy) and y(clean).

```
[ ]  # Creating more data images by making patches
     # In this code each image is divided into 16 images
     patch_size = 64

     noisy_patches = np.vstack([patchify(img, (patch_size, patch_size, 3), step=patch_size) for img in x])
     clean_patches = np.vstack([patchify(mask, (patch_size, patch_size, 3), step=patch_size) for mask in y])

     noisy_patches = noisy_patches.reshape(-1, patch_size, patch_size, 3)
     clean_patches = clean_patches.reshape(-1, patch_size, patch_size, 3)
```

Then we used patchify library to create patches of size 64X64. I chose patches over data augmentation because then the model will be able to focus on the whole image and incorporate all the features.

```
[ ]  # Model Building
     def model_creation(input_shape=(None, None, 3), num_residual_blocks=10):
         inputs = layers.Input(shape=input_shape)
         x = conv_block(inputs, 64, 3, 1, activation='relu')

         for _ in range(num_residual_blocks):
             x = residual_block(x, 64)

         x = conv_block(x, 64, 3, 1, activation='relu')
         x = layers.Conv2D(3, 3, strides=1, padding='same')(x)

         outputs = layers.Add()([x, inputs])
         model = models.Model(inputs, outputs)
         return model
```

In this we built the model using 10 residual blocks.

```
[ ]  # Defining model, optimizers and loss.
     resnet_1 = model_creation()
     resnet_1.compile(optimizer=optimizers.Adam(learning_rate=1e-4), loss=losses.MeanSquaredError())

     WARNING:tensorflow:From c:\Users\prajw\AppData\Local\Programs\Python\Python311\Lib\site-packages\keras\src\backen

[ ]  # Variable learning rate and early stopping
     callbacks = [
         tf.keras.callbacks.ReduceLROnPlateau(monitor='loss', factor=0.5, patience=5, min_lr=1e-6),
         tf.keras.callbacks.EarlyStopping(monitor='loss', patience=10, restore_best_weights=True)
     ]

[ ]  # Training the data to the model
     resnet_1.fit(noisy_patches,clean_patches, epochs=25, callbacks=callbacks,batch_size=16, validation_split=0.1)
```

I initialised my first model with name resnet_1, with adam optimiser and learning rate of 10^(-4). Then I implemented ReduceLOR with patience of 5 and reducing factor of 0.5. Then I also implemented Earlystopping which monitors loss and a patience of 10 and restore weights as True. Trained model with 25 epochs.

```
     # Prediction
     x_denoised = resnet_1.predict(x)

     16/16 ━━━━━━━━━━━━━━━━━━━━ 103s 6s/step

[ ]  # Define function for psnr
     def psnr(target, ref):
         target_data = np.array(target, dtype=np.float32)
         ref_data = np.array(ref, dtype=np.float32)
         diff = ref_data - target_data
         rmse = np.sqrt(np.mean(diff ** 2))
         psnr_value = 20 * np.log10(1.0 / rmse)
         return psnr_value

[ ]  # Calculate PSNR for x
     psnr_values = [psnr(y[i], x[i]) for i in range(len(y))]
     average_psnr = np.mean(psnr_values)
     print(f'Average PSNR: {average_psnr}')

     Average PSNR: 7.765406973920432

[ ]  # Calculate PSNR for x_denoised
     psnr_values = [psnr(y[i], x_denoised[i]) for i in range(len(y))]
     average_psnr = np.mean(psnr_values)
     print(f'Average PSNR: {average_psnr}')

     Average PSNR: 21.966737012502627
```

In this part I saved my output in x_denoised, and wrote a function to calculate psnr and then I calculated average psnr which came out as 21.96.

```
[ ]  # Creating more data images by making patches
     # In this code each image is divided into 64 images
     patch_size_2 = 32
     x_denoised_patches = np.vstack([patchify(img, (patch_size_2, patch_size_2, 3), step=patch_size_2) for img in x_denoised])
     x_denoised_patches = x_denoised_patches.reshape(-1, patch_size_2, patch_size_2, 3)
     y_patches_2 = np.vstack([patchify(mask, (patch_size_2, patch_size_2, 3), step=patch_size_2) for mask in y])
     y_patches_2 = y_patches_2.reshape(-1, patch_size_2, patch_size_2, 3)

[ ]  # Passing the x_denoised again in the same moel with new features and optimizers
     # Defining the new model
     resnet_2 = model_creation()
     resnet_2.compile(optimizer=optimizers.Adam(learning_rate=1e-4),
                      loss=losses.MeanSquaredError())

[ ]  resnet_2.fit(x_denoised_patches,y_patches_2, epochs=35, callbacks=callbacks,batch_size=16, validation_split=0.1)
```

Then I fed my x_denoised to another initialised model of same architecture but this time I changed patch size to 32 and ran 35 epochs with validation split of 0.1.

CONCLUSION:

Now for the Final PSNR:

```
# Calculating psnr for final denoised image i.e. x_denoised_2
psnr_values = [psnr(y[i], x_denoised_2[i]) for i in range(len(y))]
average_psnr = np.mean(psnr_values)
print(f'Average PSNR: {average_psnr}')

Average PSNR: 23.87530208335023
```

```
# Calculating psnr for final denoised image i.e. x_denoised_2
psnr_values = [psnr(y[i], x_denoised_2[i]) for i in range(len(y))]
average_psnr = np.mean(psnr_values)
print(f'Average PSNR: {average_psnr}')

Average PSNR: 23.73658494010944
```

```
# Calculating psnr for final denoised image i.e. x_denoised_2
psnr_values = [psnr(y[i], x_denoised_2[i]) for i in range(len(y))]
average_psnr = np.mean(psnr_values)
print(f'Average PSNR: {average_psnr}')

Average PSNR: 22.40630827001062
```

I've run the model three times and got PSNR1 as 23.87, PSNR2 as 23.72 and PSNR3 as 22.4. Now we'll be taking average of these values.

FINAL PSNR = 23.66

To further improve this project I would suggest using FFDNet but it takes a lot of RAM and VRAM and is very computationally stressing, so I used ResNet.

THIS IS THE END OF MY REPORT AND I’VE TRIED A LOT OF DIFFERENT MODELS AND GAVE MY BEST!!!

THANK YOU