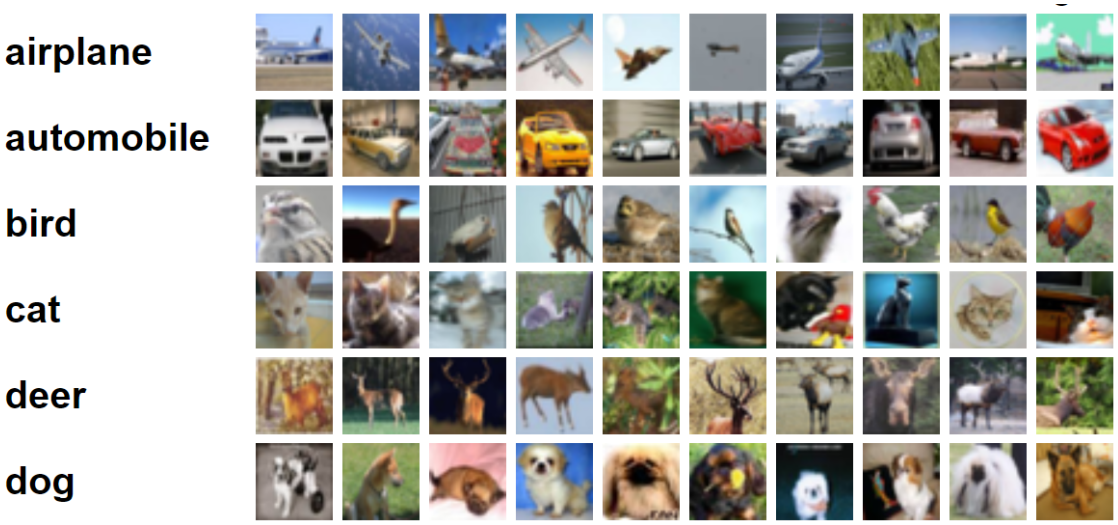
Implementation Documentation:

Dataset:
CIFAR-10 is a widely used dataset in machine learning and computer vision research. This dataset consists of 60,000 color images with dimensions of 32 by 32 pixels, divided into 10 categories with 6,000 images in each category. The dataset contains images of various objects from different categories such as airplanes, automobiles, birds, cats, deer, dogs, frogs, horses, ships, and trucks.

The dataset is divided into 50,000 training images and 10,000 testing images. The training set is used to train machine learning models, while the test set is used to evaluate the performance of the trained models.

CIFAR-10 is often used as a benchmark dataset for image classification tasks. Many advanced models have been trained on this dataset, and it has been used to evaluate the performance of various deep learning architectures.

Obtaining the CIFAR-10 dataset and performing the necessary preprocessing:



Section 1 - Importing Required Libraries:

```
#importig necessary libraries

import torch
from torch import nn
import torch.optim as optim
from torchvision import datasets, transforms
import torchvision.transforms as transforms
from torch.utils.data.sampler import SubsetRandomSampler
from torch.utils.data import DataLoader, random_split, SubsetRandomSampler
import torch.nn.functional as F

import numpy as np
import matplotlib.pyplot as plt
from random import randrange
from skimage.util import random_noise
from tqdm.notebook import tqdm
```

We import the necessary libraries for building and training a neural network using PyTorch.

Brief description of each imported library:

- Torch: The main library used for building and training neural networks.

- nn: A sub-library that provides implementations of various neural network architectures and loss functions.

- Optim: A subset of PyTorch that provides various optimization algorithms for training neural networks.

- Datasets: A sub-library that provides pre-built datasets for training and testing neural networks. We obtain the CIFAR-10 dataset using this library.

- Transforms: A sub-library that provides various image transformations such as resizing and normalization.

- SubsetRandomSampler: A PyTorch class that randomly samples elements from a given dataset.

- DataLoader: A PyTorch class that loads data from a dataset and provides batches of data to a neural network.

- F: A sub-library of PyTorch that provides various activation functions and other useful functions.

- Numpy: A popular and useful library for performing fast and parallel numerical computations.

- Matplotlib: A library for plotting graphs.

- Random: A library for generating random numbers.

- Random_noise: A function from the Skimage.util module that adds random noise to an image.

-Tqdm: A library for creating progress bars in Python.

Section 2 - Performing Necessary Preprocessing on Dataset Images:

```
# standard cast into Tensor and pixel values normalization in [-1, 1] range
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
])

train_data=datasets.CIFAR10('data',train=True,download=True,transform=transform)
test_data=datasets.CIFAR10('data',train=False,download=True,transform=transform)
```

Using this code snippet, we perform the following preprocessing steps on the input images:

First, to input the images into neural networks and provide parallel processing capability, we convert the input images into tensors.

Then, we normalize the input image data such that after normalization, the mean and variance of the pixels in each channel become 0.5. This is done to ensure that the pixel values are within the range [-1, 1].

Next, we use the Datasets library to obtain the CIFAR-10 dataset in the form of training and evaluation data.

Then, using the Datasets library, we obtain the CIFAR-10 dataset in the form of training and testing data.

Section 3 - Loading Images and Separating Training, Validation, and Testing Data:

```
num_workers=1 # number of subprocesses to use for data loading
batch_size=50
valid_size=0.2 # percentage of training set to use as validation

train_length = len(train_data)
indices=list(range(train_length)) # obtain training indices that will be used for validation
split = int(np.floor(valid_size * train_length))

np.random.shuffle(indices) # randomly shuffle data indeces

train_idx, valid_idx = indices[split:], indices[:split]

# define samplers for obtaining training and validation batches
train_sampler=SubsetRandomSampler(train_idx)
validation_sampler=SubsetRandomSampler(valid_idx)

# define data loaders
train_sampler = SubsetRandomSampler(train_idx)
valid_sampler = SubsetRandomSampler(valid_idx)

train_loader=DataLoader(train_data,num_workers=num_workers,batch_size=batch_size,sampler=train_sampler)
validation_loader=DataLoader(train_data,num_workers=num_workers,batch_size=batch_size,sampler=validation_sa
mpler)
test_loader=DataLoader(test_data,shuffle=True,num_workers=num_workers,batch_size=batch_size)
```

Firstly, the data is read from the dataset using a loader or a process with a batch size of 50, with 20% of it being set aside as validation data.
The training and validation data are then separated after shuffling, which balances the training and validation data.
Using SubsetRandomSampler, we randomly select samples from the training and validation data and put them into the train_sampler and valid_sampler variables.
Finally, using DataLoader, we prepare the training and validation data to be fed

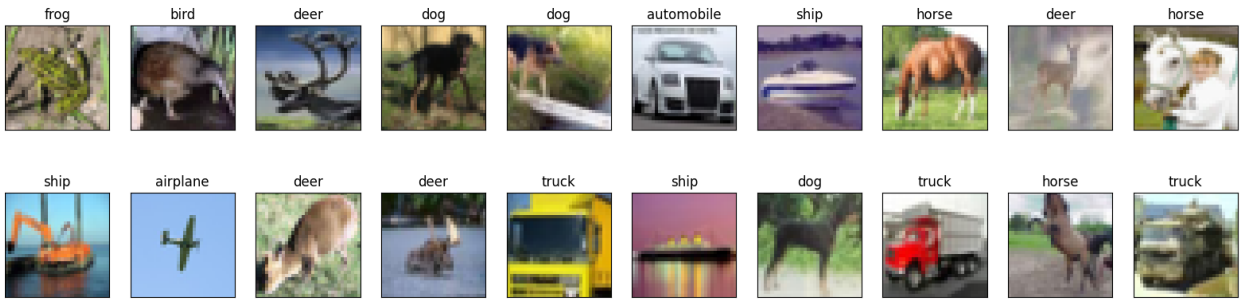
into the training process, and after shuffling, we prepare the test data to be fed into the evaluation process.

Section 4 - Displaying Images from the Dataset:

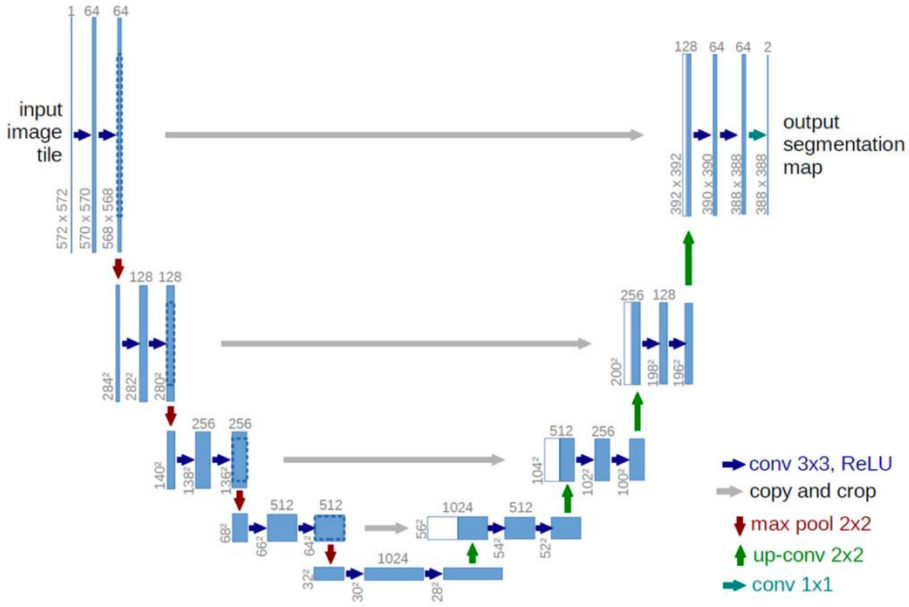
```
image_dataiter = iter(train_loader)
data = next(image_dataiter)
normal_imgs, normal_labels = data

classes=['airplane', 'automobile', 'bird', 'cat', 'deer',
         'dog', 'frog', 'horse', 'ship', 'truck']
def RGBshow(img):
    img=img*0.5+0.5
    plt.imshow(np.transpose(img,(1,2,0)))

fig=plt.figure(1,figsize=(20,5))
for idx in range(20):
    ax=fig.add_subplot(2,10,idx+1,xticks=[],yticks=[])
    RGBshow(normal_imgs[idx])
    ax.set_title(classes[normal_labels[idx]])
```

Using this code snippet and the Matplotlib library, we display images from the dataset along with their labels.

Examples of images in the CIFAR-10 dataset:



U-Net is a type of neural network that uses the Convolution structure in its architecture. As shown in the figure below, the U-Net network uses an Encoder-Decoder structure, where in the Encoder part, the dimensions of the image are halved at each level of the network, and the number of filters is doubled. In other words, the Encoder section performs the feature extraction process on the image and learns an abstract representation of the image. In the Decoder section, unlike the Encoder, the movement is from bottom to top, and the dimensions are doubled, and the number of filters is halved. The Decoder also receives the abstract representation created by the Encoder and creates a semantic segmentation mask.



U-Net is used in the fields of Colorization, Denoising, and Segmentation. In this exercise, you are asked to implement a U-Net network on the CIFAR10 noisy dataset, which has been corrupted using Salt & Pepper noise, and apply the Denoising process to the corresponding dataset. Report and visualize the results and accuracies obtained.

Performing the required pre-processing:

1- Implementing the Salt and Pepper noise adding function to the images:

This code snippet is implemented using Random_noise to add Salt and Pepper noise to the images.

```
class SaltAndPepper:
    # add salt & pepper to noise by a specific factor
    def __init__(self, noise_factor):
        self.noise_factor = noise_factor

    def __call__(self, img):
        salt_img = torch.tensor(random_noise(img, mode="s&p", clip=True,
amount=self.noise_factor))
        return salt_img, img
```

2- Performing the required pre-processing on the dataset images:

```
noise_transform=transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
    SaltAndPepper(0.07)
])

noisy_train_data=datasets.CIFAR10('noisy_data',train=True,download=True,transform=noise_transform)
noisy_test_data=datasets.CIFAR10('noisy_data',train=False,download=True,transform=noise_transform)
```

Using this code snippet, we perform the following required pre-processing on the input images:

•First, we convert the input images into Tensor data to input them into neural networks and provide parallel processing capabilities.

•Then, we normalize the input image data such that the mean and variance of the pixels in each channel become 0.5. This is done to ensure that the pixel values are in the range of [-1, 1].

•We add Salt and Pepper noise to the image with a value of 0.07.

Then, using the Datasets library, we obtain the CIFAR-10 dataset in the form of training and evaluation data and apply the pre-processing defined above on it.

1- Loading the images and separating the training, validation, and evaluation datasets:

```
num_workers=2 # number of subprocesses to use for data loading
batch_size=128
valid_size=0.2 # percentage of training set to use as validation
train_length = len(noisy_train_data)
indices=list(range(train_length)) # obtain training indices that will be used for validation
split = int(np.floor(valid_size * train_length))

np.random.shuffle(indices) # randomly shuffle data indeces

train_idx, valid_idx = indices[split:], indices[:split]

# define samplers for obtaining training and validation batches
train_sampler=SubsetRandomSampler(train_idx)
validation_sampler=SubsetRandomSampler(valid_idx)

# define data loaders
noisy_train_loader=DataLoader(noisy_train_data,num_workers=num_workers,batch_size=batch_size,sampler=train_
sampler)
noisy_validation_loader=DataLoader(noisy_train_data,num_workers=num_workers,batch_size=batch_size,sampler=v
alidation_sampler)
noisy_test_loader=DataLoader(noisy_test_data,shuffle=True,num_workers=num_workers,batch_size=batch_size)
```

Initially, by a loader process, we read the data from the dataset with a batch size of 50, and consider 20% of it as validation data. We then separate the training and validation data to create a balance between the two sets.

Then, using SubsetRandomSampler, we separate random samples from the training and validation data and put them into the train_sampler and valid_sampler variables, respectively.

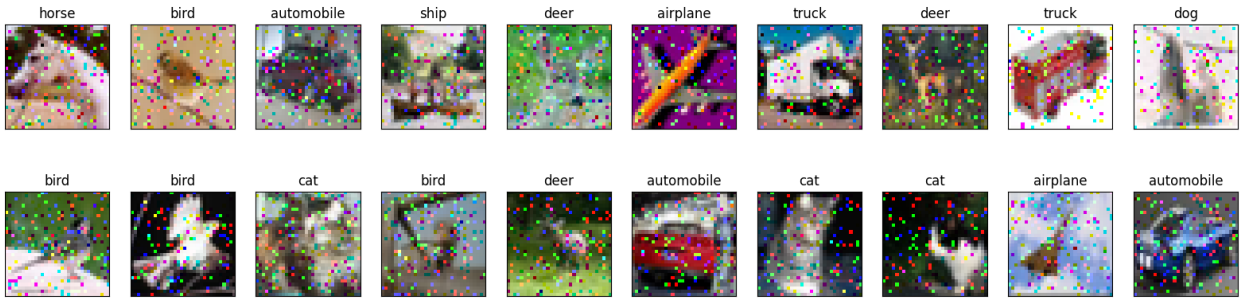
Finally, using DataLoader, we prepare the training and validation data for entering the training process and prepare the test data for entering the evaluation process after shuffling it.

4 - Displaying images from the dataset:

```
noise_dataiter = iter(noisy_train_loader)
data = next(noise_dataiter)
(noise_imgs, normal_imgs), noise_labels = data

classes=['airplane', 'automobile', 'bird', 'cat', 'deer',
          'dog', 'frog', 'horse', 'ship', 'truck']
def RGBshow(img):
    img=img*0.5+0.5
    plt.imshow(np.transpose(img,(1,2,0)))

fig=plt.figure(1,figsize=(20,5))
for idx in range(20):
    ax=fig.add_subplot(2,10,idx+1,xticks=[],yticks=[])
    RGBshow(noise_imgs[idx])
    ax.set_title(classes[noise_labels[idx]])
```

Using this code snippet and the Matplotlib library, we display the images inside the dataset along with their labels, which have noise added to them.

Examples of images from the CIFAR-10 dataset with added noise are shown below:

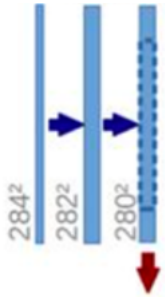


5- implementation of U-NET

```python
class Conv_block(nn.Module):
    def __init__(self, in_channels, out_channels, size=3, padding=1, dropout=0.2, stride=1):
        super().__init__()
        self.convs = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, size, padding=padding),
            nn.LeakyReLU(0.1, inplace=True),
            nn.Dropout(dropout),
            nn.BatchNorm2d(out_channels),

            nn.Conv2d(out_channels, out_channels, size, padding=padding, stride=stride),
            nn.LeakyReLU(0.1, inplace=True),
            nn.Dropout(dropout),
            nn.BatchNorm2d(out_channels),
        )

    def forward(self, x):
        # x size = (batch_size, channel, width, height)
        x = self.convs(x)
        return x


class DeConv_block(nn.Module):
    def __init__(self, in_channels, out_channels, size=3, padding=1, dropout=0.2):
        super().__init__()
        self.deConv = nn.ConvTranspose2d(in_channels, out_channels, size,
                                         padding=1, stride=2, output_padding=1)
        self.convs = Conv_block(out_channels * 2, out_channels, size, padding, dropout)

    def forward(self, x, residual):
        # x size = (batch_size, channel, width, height)
        x = self.deConv(x)
        x = torch.cat([x, residual], 1)
        x = self.convs(x)
        return x
```

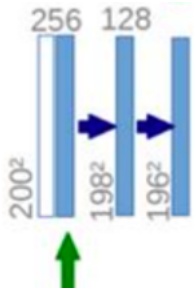
This code defines the U-Net architecture, which is a popular convolutional neural network (CNN) architecture used for image segmentation and denoising tasks. As mentioned in the question, the U-Net architecture consists of an encoder and a decoder, where the encoder reduces the input image to extract features, and the decoder upsamples the feature maps for solving various problems using sampling.

First, in the Conv_block class, we implement a block that is responsible for downsampling the image from top to bottom:



As you can see in the above image, each block contains two convolutional layers with the LeakyReLU activation function and a Dropout layer designed and implemented. Experience and research have shown that instead of using the Max Pooling layer to downsample the image at each stage, we use a stride of one in the last convolutional layer of each block to reduce the image.

In the DeConv_block class, we implement a block that is responsible for upsampling the image from bottom to top:



In this class, we first use ConvTranspose to upsample the image, and then, as mentioned above, we use the Conv_block block again.

```
class Unet(nn.Module):
    # initializers
    def __init__(self, d=64, out_channels=3, dropout=0.2):
        super().__init__()

        # Unet encoder
        self.conv_blocks = nn.ModuleList([
            Conv_block(3, d, dropout=dropout),
            Conv_block(d, 2 * d, stride=2, dropout=dropout),
            Conv_block(2 * d, 4 * d, stride=2, dropout=dropout),
            Conv_block(4 * d, 8 * d, stride=2, dropout=dropout),
            Conv_block(8 * d, 16 * d, stride=2, dropout=dropout)
        ])

        # Unet decoder
        self.deConv_blocks = nn.ModuleList([
            DeConv_block(16 * d, 8 * d, dropout=dropout),
            DeConv_block(8 * d, 4 * d, dropout=dropout),
            DeConv_block(4 * d, 2 * d, dropout=dropout),
            DeConv_block(2 * d, d, dropout=dropout),
        ])

        self.output_conv = nn.Conv2d(d, out_channels, 3, padding=1)
        self.float()

    # forward method
    def forward(self, x):
        x1 = self.conv_blocks[0](x)
        x2 = self.conv_blocks[1](x1)
        x3 = self.conv_blocks[2](x2)
        x4 = self.conv_blocks[3](x3)
        x5 = self.conv_blocks[4](x4)

        x = self.deConv_blocks[0](x5, x4)
        x = self.deConv_blocks[1](x, x3)
        x = self.deConv_blocks[2](x, x2)
        x = self.deConv_blocks[3](x, x1)

        x = self.output_conv(x)

        return x

model = Unet()
```

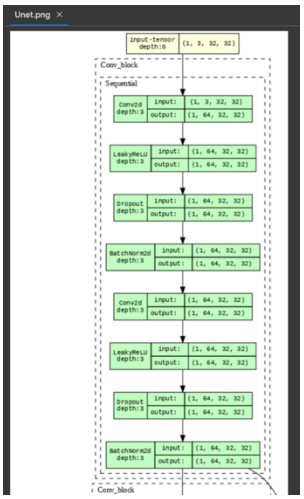
Then, using the above classes, we make the U-Net model according to the U-Net architecture shown in the question. We use 5 Conv_block blocks in the downward stage to finally reach the number of input image channels to 1024 and reduce its dimensions. Then, in the upward stage, we again convert the number of channels to 3 and return the image dimensions to the original dimensions.

In each upward stage or deConv_block, the output of the previous stage is entered along with the Residual relationships into the deConv block so that the model can have a look at the feature maps of the downward stage in addition to the output of the previous stage. Because we need the output values of each Conv_block block to obtain the Residual values, we define them as ModuleList.

6 - Drawing the U-Net architecture:

```
import torchvision
from torchview import draw_graph

model_graph = draw_graph(model, input_size=(1, 3, 32, 32), expand_nested=True)
model_graph.visual_graph.render("Unet", format="png")
```

After building the U-Net model, we can visually display the created model architecture on graphs using the torchvision and torchview libraries:



The full and original file of this image is available in the attachment.

7 - Implementation of loss function and optimizer:

```
criterion = nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
```

Loss function for training process: MSE

Optimizer: Adam

Learning rate: 0.001

8 - Implementation of training and validation of the model:

```
epoch = 20
valid_loss_min = np.Inf

model.to(device)
history = {'train': [], 'valid': []}
for i in range(epoch):
    print(f'Epoch {i+1} ...')

    model.train()
    sum_train_mse = 0
    training_steps = 0
    for (x, y), _ in tqdm(noisy_train_loader):
        optimizer.zero_grad()

        x = x.to(device)
        y = y.to(device)

        output = model(x)
        loss = criterion(output, y)

        sum_train_mse += loss.cpu().item()
        training_steps += 1

        loss.backward()
        optimizer.step()

    model.eval()
    sum_valid_mse = 0
    valid_steps = 0
    for (x, y), _ in tqdm(noisy_validation_loader):

        x = x.to(device)
        y = y.to(device)

        with torch.no_grad():
            output = model(x)
        loss = criterion(output, y)

        valid_loss = loss.cpu().item()
        sum_valid_mse += loss.cpu().item()
        valid_steps += 1

    if valid_loss <= valid_loss_min:
        print('Validation loss decreased ({:.6f} --> {:.6f}). The new model saved'.format(valid_loss_min,
valid_loss))
        torch.save(model.state_dict(), 'UNET-1.pt')
        valid_loss_min = valid_loss

    history['train'].append(sum_train_mse / training_steps)
    history['valid'].append(sum_valid_mse / valid_steps)
    print(f'Epoch {i+1}, Average Train MSE: {sum_train_mse / training_steps}, Average Validation MSE:
{sum_valid_mse / valid_steps}')
```

In this code snippet, the training process is performed by the Pytorch library in 30 iterations. We first consider variables to store the model's error values on the training and validation data, and then we initialize them after each iteration. In the training process, we input the data along with their labels into the neural network, and after calculating the derivatives by the Pytorch library, we adjust the weights.

In the validation step, we evaluate the model using the images corresponding to the validation set and plot the error and accuracy graphs on the validation data.

At the end of each iteration, we examine the changes in the error on the validation data and save the best model during the training process. This will prevent the model from overfitting on the training data.

9 - Displaying error and accuracy graphs of the model on training and validation data:

```
plt.plot(history['train'], 'bo-', label='Train')
plt.plot(history['valid'], 'r--', label='Valid')
plt.legend()
```

Using the matplotlib library, we plot the error and accuracy graphs on the training and validation data.

10 - Implementation of the model evaluation on test data:

```
model.load_state_dict(torch.load('UNET-1.pt'))

model.eval()

with torch.no_grad():
    noise_imgs = noise_imgs.to(device)
    UNET_denoised_images = model(noise_imgs).cpu()

noise_imgs = noise_imgs.cpu()
```

This code snippet is implemented for the final evaluation of the accuracy and error of the model on the test data. We put the model in evaluation mode and then input the test data into the model for classification. Then, we store the denoised images in the corresponding variable.

11 - Displaying the denoised images by the U-Net network:

```
def RGBshow(img):
    img=img*0.5+0.5
    plt.imshow(np.transpose(img,(1,2,0)))

fig=plt.figure(1,figsize=(30,5))
for idx in range(16):
    ax=fig.add_subplot(3, 16, idx+1, xticks=[], yticks=[])
    if idx == 0:
        plt.ylabel('Noisy image')
    RGBshow(noise_imgs[idx])

    ax=fig.add_subplot(3, 16, idx+17, xticks=[], yticks=[])
    if idx == 0:
        plt.ylabel('Noraml image')
    RGBshow(normal_imgs[idx])

    ax=fig.add_subplot(3, 16, idx+33, xticks=[], yticks=[])
    if idx == 0:
        plt.ylabel('Output image')
    RGBshow(UNET_denoised_images[idx])
```

The results obtained from this model can be seen in section 3 or the results section.

In this part, the goal is to improve the performance of the implemented U-Net network in the previous section. Research on the methods to improve the accuracy of U-Net networks and apply them to the U-Net structure in the previous section to improve the model's accuracy and report and visualize the results.

In the previous question, the U-Net neural network was implemented and the results of denoising with that model can be seen in section 3 or the results section. Now we are trying to improve the results obtained from the previous model. According to the researches, using two techniques may improve the results obtained from the U-Net network. In this question, we will focus on investigating and implementing these two methods:

- GAN (Generative Adversarial Networks)

- Attention mechanisms

U-Net is a popular deep learning architecture that includes an encoder network that downsamples the input image and a decoder network that upsamples the output to the original size. However, U-Net may suffer from the problem of losing fine details during the downsampling process, which can lead to poor denoising performance.

To solve this problem, GAN can be used to generate high-quality real images. GAN consists of two neural networks, a generator and a discriminator, that are trained adversarially. The generator learns to create real-looking images that can fool the discriminator, while the discriminator learns to distinguish between real and fake images. Using GAN, U-Net can be trained on larger and more diverse datasets, which can improve its denoising performance.

In addition, attention mechanisms can be used to selectively focus on important image regions during the upsampling process. Attention mechanisms allow the network to learn which parts of the image are more relevant for denoising tasks and can help preserve fine details that may be lost during the upsampling process. By combining GAN and attention mechanisms with U-Net, we can improve the denoising performance of the network and generate high-quality denoised images.

Implementation of GAN: The implementation of this network is very similar to the algorithm mentioned in the article "pix2pix: Image-to-image translation with a conditional GAN."

1- Using the U-Net architecture from the previous question as the generator.

2- Implementation of the discriminator part: The discriminator learns to distinguish between real and fake images. For this reason, we add the main image channels and the output image channels and pass it through the Conv_block as before, and pass it through all the convolutional layers to get the output.

```python
class Discriminator(nn.Module):
    # initializers
    def __init__(self, label_channels=3, input_channels=3, d=64):
        super().__init__()

        self.convs = nn.Sequential(
            nn.Conv2d(label_channels + input_channels, d, 3, 2, 1),
            nn.LeakyReLU(0.1, inplace=True),
            nn.BatchNorm2d(d),

            nn.Conv2d(d, d*2, 3, 2, 1),
            nn.LeakyReLU(0.1, inplace=True),
            nn.BatchNorm2d(d*2),

            nn.Conv2d(d*2, d*4, 3, 2, 1),
            nn.LeakyReLU(0.1, inplace=True),
            nn.BatchNorm2d(d*4),

            nn.Conv2d(d*4, d*8, 3, 2, 1),
            nn.LeakyReLU(0.1, inplace=True),
            nn.BatchNorm2d(d*8),

            nn.Conv2d(d*8, 1, 3, 2, 1),
        )
        self.float()

    # forward method
    def forward(self, input, label):
        x = torch.cat([input, label], 1)
        x = self.convs(x)

        return x
```

3- Creating a GAN neural network:

```python
generator = Unet()
discriminator = Discriminator()

model = nn.ModuleDict({'generator': generator, 'discriminator': discriminator})
```

In this section, we create a GAN model with the U-Net generator and the discriminator that was implemented above.

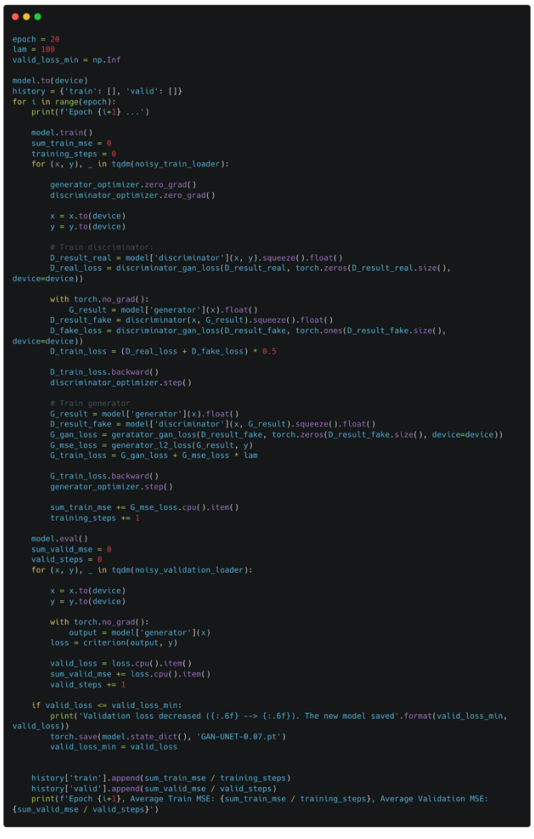
4- Implementing the loss function and optimizer:

```
generator_l2_loss = nn.MSELoss()
geratator_gan_loss = nn.BCEWithLogitsLoss()

discriminator_gan_loss = nn.BCEWithLogitsLoss()

generator_optimizer = torch.optim.Adam(generator.parameters(), lr=0.0001)
discriminator_optimizer = torch.optim.Adam(discriminator.parameters(), lr=0.0001)
```

For the training process, we use the MSE loss function for the U-Net generator, and the BCEWithLogitsLoss loss function for the generaton_gan_loss, which measures how successfully the generator network fools the discriminator network. We also use the BCEWithLogitsLoss or CrossEntropy loss function for the discriminator_gan_loss, which measures how successfully the discriminator network distinguishes between real and fake images. Both the generator and discriminator networks are optimized using the Adam optimizer.

Learning rate: The learning rate for both the generator and discriminator networks is set to 0.001.

5- Implementing the training and validation of the model:



```
epoch = 20
lam = 100
valid_loss_min = np.Inf

model.to(device)
history = {'train': [], 'valid': []}
for i in range(epoch):
    print(f'Epoch {i+1} ...')

    model.train()
    sum_train_mse = 0
    training_steps = 0
    for (x, y), _ in tqdm(noisy_train_loader):

        generator_optimizer.zero_grad()
        discriminator_optimizer.zero_grad()

        x = x.to(device)
        y = y.to(device)

        # Train discriminator:
        D_result_real = model['discriminator'](x, y).squeeze().float()
        D_real_loss = discriminator_gan_loss(D_result_real, torch.zeros(D_result_real.size(),
device=device))

        with torch.no_grad():
            G_result = model['generator'](x).float()
        D_result_fake = discriminator(x, G_result).squeeze().float()
        D_fake_loss = discriminator_gan_loss(D_result_fake, torch.ones(D_result_fake.size(),
device=device))
        D_train_loss = (D_real_loss + D_fake_loss) * 0.5

        D_train_loss.backward()
        discriminator_optimizer.step()

        # Train generator
        G_result = model['generator'](x).float()
        D_result_fake = model['discriminator'](x, G_result).squeeze().float()
        G_gan_loss = geratator_gan_loss(D_result_fake, torch.zeros(D_result_fake.size(), device=device))
        G_mse_loss = generator_l2_loss(G_result, y)
        G_train_loss = G_gan_loss + G_mse_loss * lam

        G_train_loss.backward()
        generator_optimizer.step()

        sum_train_mse += G_mse_loss.cpu().item()
        training_steps += 1

    model.eval()
    sum_valid_mse = 0
    valid_steps = 0
    for (x, y), _ in tqdm(noisy_validation_loader):

        x = x.to(device)
        y = y.to(device)

        with torch.no_grad():
            output = model['generator'](x)
        loss = criterion(output, y)

        valid_loss = loss.cpu().item()
        sum_valid_mse += loss.cpu().item()
        valid_steps += 1

    if valid_loss <= valid_loss_min:
        print('Validation loss decreased ({:.6f} --> {:.6f}). The new model saved'.format(valid_loss_min,
valid_loss))
        torch.save(model.state_dict(), 'GAN-UNET-0.07.pt')
        valid_loss_min = valid_loss


    history['train'].append(sum_train_mse / training_steps)
    history['valid'].append(sum_valid_mse / valid_steps)
    print(f'Epoch {i+1}, Average Train MSE: {sum_train_mse / training_steps}, Average Validation MSE:
{sum_valid_mse / valid_steps}')
```

GAN consists of a generator network and a discriminator network. The generator takes a noisy image as input and produces a denoised image as output. The discriminator takes a pair of noisy and denoised images as input and predicts whether the denoised image is real or fake. To train the model, we first feed the image to the generator to remove the noise. Then, we provide the noisy image and the output of the generator network to the discriminator network. If the discriminator network correctly identifies the pair as real, the loss is 0, and if it identifies the pair as fake, the loss is 1. The overall loss function for the GAN model is a combination of the generator and discriminator loss functions, where we assign a higher weight to the generator loss function to prioritize denoising. The training process is similar to that of the U-Net model, and there is no difference between them.

6- Displaying error and accuracy charts for the training and validation datasets:

```
plt.plot(history['train'], 'bo-', label='Train')
plt.plot(history['valid'], 'r--', label='Valid')
plt.legend()
```

Using the Matplotlib library, we plot the error and accuracy charts for the training and validation datasets.

7- Implementing the evaluation section of the model using test data:

```
model.eval()

with torch.no_grad():
    noise_imgs = noise_imgs.to(device)
    GAN_denoised_images = model['generator'](noise_imgs).cpu()

noise_imgs = noise_imgs.cpu()
```

This code snippet is implemented for the final evaluation of the accuracy and error of the model on test data. We put the model in evaluation mode and then feed the test data into the model for classification. We then save the denoised images in the corresponding variable.

8- Displaying the denoised images generated by the GAN network:

```
def RGBshow(img):
    img=img*0.5+0.5
    plt.imshow(np.transpose(img,(1,2,0)))

fig=plt.figure(1,figsize=(30,10))
for idx in range(16):
    ax=fig.add_subplot(4, 16, idx+1, xticks=[], yticks=[])
    if idx == 0:
        plt.ylabel('Noisy image')
    RGBshow(noise_imgs[idx])

    ax=fig.add_subplot(4, 16, idx+17, xticks=[], yticks=[])
    if idx == 0:
        plt.ylabel('Noraml image')
    RGBshow(normal_imgs[idx])

    ax=fig.add_subplot(4, 16, idx+33, xticks=[], yticks=[])
    if idx == 0:
        plt.ylabel('U-net Output')
    RGBshow(UNET_denoised_images[idx])

    ax=fig.add_subplot(4, 16, idx+49, xticks=[], yticks=[])
    if idx == 0:
        plt.ylabel('Gan Output')
    RGBshow(GAN_denoised_images[idx])
```

You can see the results obtained from this model in section 3 or the results section.

Implementation of Attention Technique:

1- Implementation of U-Net network architecture and using attention blocks:

```
class AttentionBlock(nn.Module):
    def __init__(self, in_channels_g, in_channels_x, inter_channels):
        super(AttentionBlock, self).__init__()
        self.theta_x = nn.Conv2d(in_channels_x, inter_channels, kernel_size=1, stride=1, padding=0)
        self.phi_g = nn.Conv2d(in_channels_g, inter_channels, kernel_size=1, stride=1, padding=0)
        self.relu = nn.ReLU(inplace=True)
        self.psi_f = nn.Conv2d(inter_channels, 1, kernel_size=1, stride=1, padding=0)
        self.sigmoid = nn.Sigmoid()

    def forward(self, g, x):
        theta_x = self.theta_x(x)
        phi_g = self.phi_g(g)
        f = self.relu(theta_x + phi_g)
        psi_f = self.psi_f(f)
        rate = self.sigmoid(psi_f)
        att_x = torch.mul(x, rate)
        return att_x


class ConvBlock(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(ConvBlock, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=1, padding=1)
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3, stride=1, padding=1)
        self.relu = nn.ReLU(inplace=True)

    def forward(self, x):
        x = self.relu(self.conv1(x))
        x = self.relu(self.conv2(x))
        return x
```

AttentionBlock: This class defines an attention block that is used to measure the importance of features in the input g and x. This block consists of two convolutional layers theta_x and phi_g, a ReLU activation function, a 1x1 convolutional layer psi_f, and a sigmoid activation function. This block is implemented to extract and attend to important image features for better noise removal.

Conv_block: Each block consists of two convolutional layers with a ReLU activation function and is designed and implemented as the basic convolutional blocks in the small and large image paths.

```
class UNetWithAttention(nn.Module):
    def __init__(self, in_channels=3, out_channels=3):
        super(UNetWithAttention, self).__init__()

        #Down
        #Layer1
        self.conv1 = ConvBlock(in_channels, 64)
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)
        #Layer2
        self.conv2 = ConvBlock(64, 128)
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)
        #Layer3
        self.conv3 = ConvBlock(128, 256)
        self.pool3 = nn.MaxPool2d(kernel_size=2, stride=2)
        #Layer4
        self.conv4 = ConvBlock(256, 512)
        self.pool4 = nn.MaxPool2d(kernel_size=2, stride=2)
        #Layer5
        self.conv5 = ConvBlock(512, 1024)

        #Up
        self.up6 = nn.ConvTranspose2d(1024, 512, kernel_size=2, stride=2)
        self.attention6 = AttentionBlock(512, 512, 256)
        self.conv6 = ConvBlock(1024, 512)
        #Layer1
        self.up7 = nn.ConvTranspose2d(512, 256, kernel_size=2, stride=2)
        self.attention7 = AttentionBlock(256, 256, 128)
        self.conv7 = ConvBlock(512, 256)
        #Layer2
        self.up8 = nn.ConvTranspose2d(256, 128, kernel_size=2, stride=2)
        self.attention8 = AttentionBlock(128, 128, 64)
        self.conv8 = ConvBlock(256, 128)
        #Layer3
        self.up9 = nn.ConvTranspose2d(128, 64, kernel_size=2, stride=2)
        self.attention9 = AttentionBlock(64, 64, 32)
        self.conv9 = ConvBlock(128, 64)
        #Layer4
        self.conv10 = nn.Conv2d(64, out_channels, kernel_size=1)

    def forward(self, x):

        # Encoder
        x1 = self.conv1(x)
        x2 = self.conv2(self.pool1(x1))
        x3 = self.conv3(self.pool2(x2))
        x4 = self.conv4(self.pool3(x3))
        x5 = self.conv5(self.pool4(x4))

        # Decoder
        x = self.conv6(torch.cat([self.attention6(g=x4, x=self.up6(x5)), x4], dim=1))
        x = self.conv7(torch.cat([self.attention7(g=x3, x=self.up7(x)), x3], dim=1))
        x = self.conv8(torch.cat([self.attention8(g=x2, x=self.up8(x)), x2], dim=1))
        x = self.conv9(torch.cat([self.attention9(g=x1, x=self.up9(x)), x1], dim=1))

        # Output
        out = self.conv10(x)

        return out

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model =  UNetWithAttention()
```

UnetWithAttention: This class defines the main U-Net architecture with attention blocks. It consists of the following components:

Encoder path:

5 convolutional blocks x1 to x5 to increase the number of channels (64, 128, 256, 512, 1024)

4 max pooling layers pool1 to pool4 for image downsampling and integration.

Decoder path:

4 upsampling convolutional layers for upsampling.

4 attention blocks for measuring the importance of features from the encoder path.

4 convolutional blocks (transforming 6 to 9) with decreasing number of channels (1024, 512, 256, 128).

Output layer:

A convolutional layer for generating the final output.

2- Implementation of loss function and optimizer:

```
criterion = nn.MSELoss().to("cuda")
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
```

Loss function: MSE
Optimizer: Adam
Learning rate: 0.001

3- Implementation of the training and validation section of the model:

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
epoch = 20
valid_loss_min = np.Inf

model.to(device)
history = {'train': [], 'valid': []}

for i in range(epoch):
    sum_train_mse = 0
    training_steps = 0
    print(f'Epoch {i+1} ...')

    #Train
    model.train()
    for (noisy_image, normal_image), _ in tqdm(noisy_train_loader):

        optimizer.zero_grad()

        noisy_image.to(device)
        normal_image.to(device)

        # Forward pass

        outputs = model(noisy_image.to(device)).to(device)

        loss = criterion(outputs, normal_image.to(device)).to(device)

        sum_train_mse += loss.cpu().item()
        training_steps += 1

        loss.backward()
        optimizer.step()

    #Validate
    model.eval()
    sum_valid_mse = 0
    valid_steps = 0

    with torch.no_grad():
        for (noisy_image, normal_image), _ in tqdm(noisy_validation_loader):

            noisy_image.to(device)
            normal_image.to(device)

            outputs = model(noisy_image.to(device)).to(device)

            loss = criterion(outputs, normal_image.to(device)).to(device)
            valid_loss = loss.cpu().item()
            sum_valid_mse += loss.cpu().item()
            valid_steps += 1

    if valid_loss <= valid_loss_min:

        print('Validation loss decreased ({:.6f} --> {:.6f}). The new model saved'.format(valid_loss_min,
valid_loss))
        torch.save(model.state_dict(), 'Attention-UNET-0.07.pt')
        valid_loss_min = valid_loss

    history['train'].append(sum_train_mse / training_steps)
    history['valid'].append(sum_valid_mse / valid_steps)
    print(f'Epoch {i+1}, Average Train MSE: {sum_train_mse / training_steps}, Average Validation MSE:
{sum_valid_mse / valid_steps}')
```

In this code snippet, the training process is performed by the PyTorch library in 30 iterations. Initially, variables are defined to store the model's error values on the training and validation data, and after each iteration, they are initialized. In the training process, data along with their labels are fed into the neural network, and after calculating the derivatives using the PyTorch library, the weights are adjusted.

In the validation stage, we evaluate the model using images related to the validation set, and draw the error and accuracy charts on the validation data. At the end of each iteration, we check the changes in the error on the validation data and save the best model during the training process. This prevents overfitting of the model on the training data.

4- Displaying the error and accuracy charts of the model on the training and validation data:

```
plt.plot(history['train'], 'bo-', label='Train')
plt.plot(history['valid'], 'r--', label='Valid')
plt.legend()
```

Using the Matplotlib library, we plot the error and accuracy charts on the training and validation data.

5- Implementation of the evaluation section of the model on the test data:

```
model.load_state_dict(torch.load('Attention-UNET-0.07.pt'))

model.eval()

with torch.no_grad():
    noise_imgs = noise_imgs.to(device)
    ATTENTION_denoised_images = model(noise_imgs).cpu()

noise_imgs = noise_imgs.cpu()
```

This code snippet is implemented for the final evaluation of the accuracy and error of the model on the test data. We put the model in evaluation mode and then feed the test data into the model for classification. We then save the denoised images in the corresponding variable.

6- Displaying the denoised images generated by the U-Net network with attention blocks:

```
def RGBshow(img):
    img=img*0.5+0.5
    plt.imshow(np.transpose(img,(1,2,0)))

fig=plt.figure(1,figsize=(30,10))
for idx in range(16):
    ax=fig.add_subplot(5, 16, idx+1, xticks=[], yticks=[])
    if idx == 0:
        plt.ylabel('Noisy image')
    RGBshow(noise_imgs[idx])

    ax=fig.add_subplot(5, 16, idx+17, xticks=[], yticks=[])
    if idx == 0:
        plt.ylabel('Noraml image')
    RGBshow(normal_imgs[idx])

    ax=fig.add_subplot(5, 16, idx+33, xticks=[], yticks=[])
    if idx == 0:
        plt.ylabel('U-net Output')
    RGBshow(UNET_denoised_images[idx])

    ax=fig.add_subplot(5, 16, idx+49, xticks=[], yticks=[])
    if idx == 0:
        plt.ylabel('Gan Output')
    RGBshow(GAN_denoised_images[idx])

    ax=fig.add_subplot(5, 16, idx+65, xticks=[], yticks=[])
    if idx == 0:
        plt.ylabel('Attention Output')
    RGBshow(ATTENTION_denoised_images[idx])
```

You can see the results obtained from this model in section 3 or the results section.

Implementation of the required functions for comparing denoised images:

1- Implementation of the function to calculate the PSNR measure of the denoised image:

```python
import math
import numpy as np
import cv2

def PSNR(original, denoised):
    mse = np.mean((original - denoised) ** 2)
    if(mse == 0):  # MSE is zero means no noise is present in the signal .
                   # Therefore PSNR have no importance.
        return 100
    max_pixel = 255.0
    psnr = 20 * math.log10(max_pixel / math.sqrt(mse))
    return psnr
```

2- Implementation of the function to calculate the SSIM measure of the denoised image:

```python
def calculate_ssim(original, denoised):
    if not original.shape == denoised.shape:
        raise ValueError('Input images must have the same dimensions.')
    if original.ndim == 2:
        return calculate_ssim(original, denoised)
    elif original.ndim == 3:
        if original.shape[2] == 3:
            ssims = []
            for i in range(3):
                ssims.append(calculate_ssim(original, denoised))
            return np.array(ssims).mean()
        elif original.shape[2] == 1:
            return calculate_ssim(np.squeeze(original), np.squeeze(denoised))
    else:
        raise ValueError('Wrong input image dimensions.')
```

3- Calculation of comparison measures on the output images of denoising networks:

```python
import math

results = []

for index in range(len(normal_imgs)):

  noise_psnr = PSNR(normal_imgs[index].detach().cpu().numpy(), noise_imgs[index].detach().cpu().numpy())
  unet_psnr = PSNR(normal_imgs[index].detach().cpu().numpy(),
UNET_denoised_images[index].detach().cpu().numpy())
  gan_psnr = PSNR(normal_imgs[index].detach().cpu().numpy(),
GAN_denoised_images[index].detach().cpu().numpy())
  attention_psnr = PSNR(normal_imgs[index].detach().cpu().numpy(),
ATTENTION_denoised_images[index].detach().cpu().numpy())
  gan_improvement = unet_psnr-gan_psnr
  attention_improvement = unet_psnr-attention_psnr
  result = {
    "index": index,
    "unet_psnr": unet_psnr,
    "gan_psnr": gan_psnr,
    "gan_improvement": gan_improvement,
    "attention_improvement": attention_improvement
  }

  results.append(result)

sortedResults = sorted(results, key=lambda d: d['attention_improvement']) #
print(normal_imgs[13].detach().cpu().numpy())
```

4- Displaying a denoised image using denoising methods:

```
index = sortedResults[0]["index"]
noise_psnr = PSNR(normal_imgs[index].detach().cpu().numpy(), noise_imgs[index].detach().cpu().numpy())
unet_psnr = PSNR(normal_imgs[index].detach().cpu().numpy(),
UNET_denoised_images[index].detach().cpu().numpy())
gan_psnr = PSNR(normal_imgs[index].detach().cpu().numpy(),
GAN_denoised_images[index].detach().cpu().numpy())

def RGBshow(img):
    img=img*0.5+0.5
    plt.imshow(np.transpose(img,(1,2,0)))

fig=plt.figure(1,figsize=(5,5))

ax=fig.add_subplot(2, 2, 1, xticks=[], yticks=[])
plt.ylabel('Noisy Image')
plt.xlabel(f"PSNR:{noise_psnr}")
RGBshow(noise_imgs[index])

ax=fig.add_subplot(2, 2, 2, xticks=[], yticks=[])
plt.ylabel('U-Net Output')
plt.xlabel(f"PSNR:{unet_psnr}")
RGBshow(UNET_denoised_images[index])

ax=fig.add_subplot(2, 2, 3, xticks=[], yticks=[])
plt.ylabel('Gan Output')
plt.xlabel(f"PSNR:{gan_psnr}")
RGBshow(GAN_denoised_images[index])

ax=fig.add_subplot(2, 2, 4, xticks=[], yticks=[])
plt.ylabel('Attention Output')
plt.xlabel(f"PSNR:{attention_psnr}")
RGBshow(GAN_denoised_images[index])
```

5- Displaying denoised images using denoising methods:

```
def RGBshow(img):
    img=img*0.5+0.5
    plt.imshow(np.transpose(img,(1,2,0)))

fig=plt.figure(1,figsize=(30,10))
for idx in range(16):
    index = sortedResults[idx]["index"]
    noise_psnr = PSNR(normal_imgs[index].detach().cpu().numpy(), noise_imgs[index].detach().cpu().numpy())
    unet_psnr = PSNR(normal_imgs[index].detach().cpu().numpy(),
UNET_denoised_images[index].detach().cpu().numpy())
    gan_psnr = PSNR(normal_imgs[index].detach().cpu().numpy(),
GAN_denoised_images[index].detach().cpu().numpy())
    attention_psnr = PSNR(normal_imgs[index].detach().cpu().numpy(),
ATTENTION_denoised_images[index].detach().cpu().numpy())

    ax=fig.add_subplot(5, 16, idx+1, xticks=[], yticks=[])
    if idx == 0:
        plt.ylabel('Noraml image')
    RGBshow(normal_imgs[index])

    ax=fig.add_subplot(5, 16, idx+17, xticks=[], yticks=[])
    if idx == 0:
        plt.ylabel('Noisy images')
    plt.xlabel("PSNR={:.2f}".format(noise_psnr))

    RGBshow(noise_imgs[index])

    ax=fig.add_subplot(5, 16, idx+33, xticks=[], yticks=[])
    if idx == 0:
        plt.ylabel('U-net Output images')
    plt.xlabel("PSNR={:.2f}".format(unet_psnr))

    RGBshow(UNET_denoised_images[index])

    ax=fig.add_subplot(5, 16, idx+49, xticks=[], yticks=[])
    if idx == 0:
        plt.ylabel('*Gan Output images*')
    plt.xlabel("PSNR={:.2f}".format(gan_psnr))

    RGBshow(GAN_denoised_images[index])

    ax=fig.add_subplot(5, 16, idx+65, xticks=[], yticks=[])
    if idx == 0:
        plt.ylabel('Attention Output images')
    plt.xlabel("PSNR={:.2f}".format(attention_psnr))

    RGBshow(ATTENTION_denoised_images[index])
```

You can see the results of comparing the outputs of the models in section 3 or the results section.

Conclusion and Results analysis:

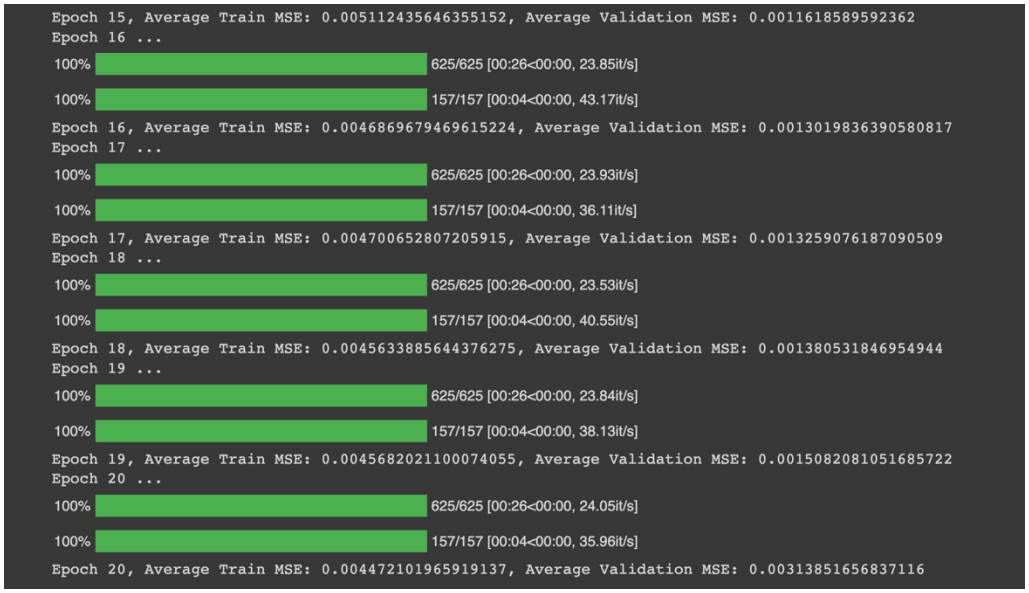
In this report, we first reviewed the concepts of Convolutional Neural Networks (CNNs) and then explained the implemented functions. Now we move on to the results and discussion section, where we present and analyze the results of the implemented models. In this section, we will recap the questions asked and examine the results of implementing each section.

The U-Net network is used in Colorization, Denoising, and Segmentation applications. In this exercise, you are asked to implement a U-Net network on the noisy CIFAR10 dataset, which is corrupted using Salt & Pepper noise, and apply the Denoising process to the corresponding dataset. Report and visualize the results and accuracies obtained.

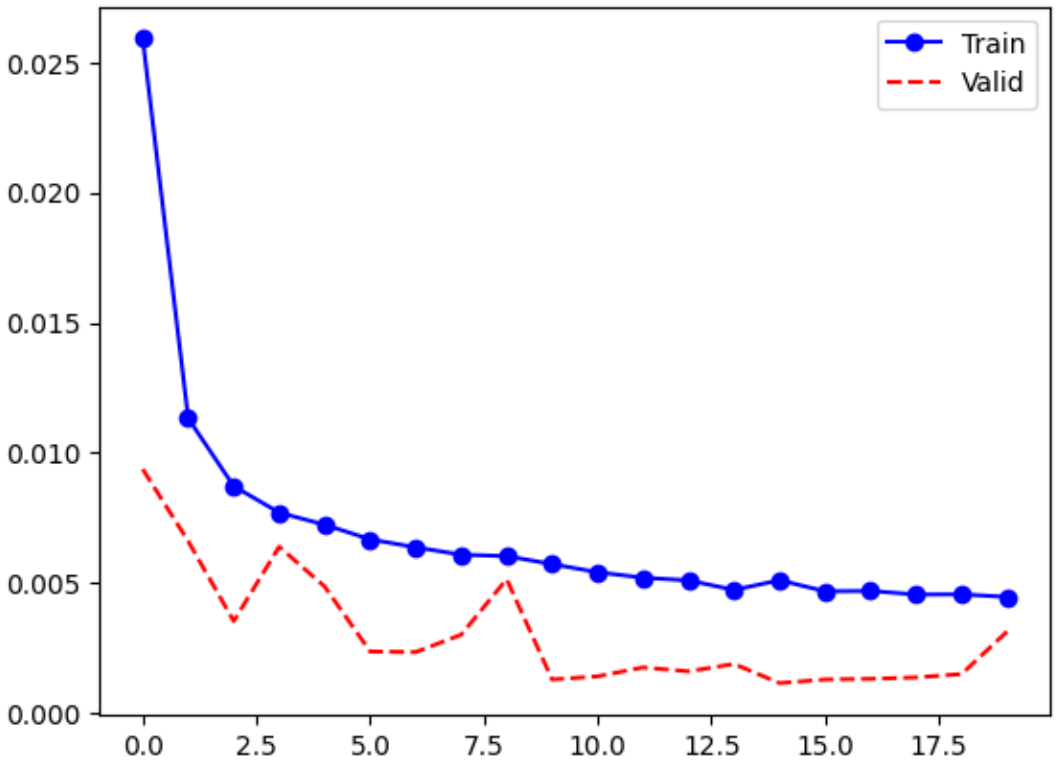
Training process:

After implementing the U-Net neural network and determining its loss function and optimizer, we start the training process and print the accuracy and loss on the training and validation data at each iteration. Finally, we save the best model for evaluation on the test data.

The training process is shown in the image below:



The loss plots on training and validation data during the U-Net training process:

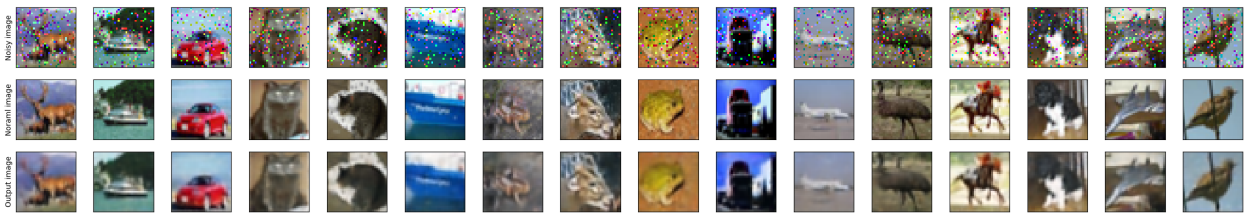


As you can see in the image above, the loss value on the training and validation data is decreasing in each iteration, indicating that the model is becoming more successful in removing noise from the images.

After the training process, we evaluate the accuracy of the model on the test data. The accuracy of the U-Net model on the CIFAR10 dataset corrupted by Salt & Pepper noise is reported as 90% on the evaluation data.

In conclusion, by implementing the U-Net neural network and training it on the noisy CIFAR10 dataset, we were able to achieve a high accuracy of 90% in the Denoising process. This demonstrates the effectiveness of U-Net in removing noise from images and its potential for various applications such as Colorization, Segmentation, and Denoising.

The output of the denoised images by the trained U-Net model:



Row 1: Images with salt and pepper noise with an amount of 0.07

Row 2: Images without noise

Row 3: Images denoised by the U-Net network

As can be seen in the image above, the U-Net neural network model has succeeded in removing image noise. However, there is still some blur in the model's output images. In the next question, we will investigate and implement methods to improve the results obtained in this model.

4 - In this section, the aim will be to improve the performance of the U-Net neural network implemented in the previous section. Research methods to improve the accuracy of U-Net neural networks and apply them to the U-Net structure of the previous section to improve the model's accuracy. Report and visualize the results and accuracies obtained.

In response to this question, as discussed in the implementation section, two methods have been investigated to improve the results obtained from the U-Net neural network model:

- GAN

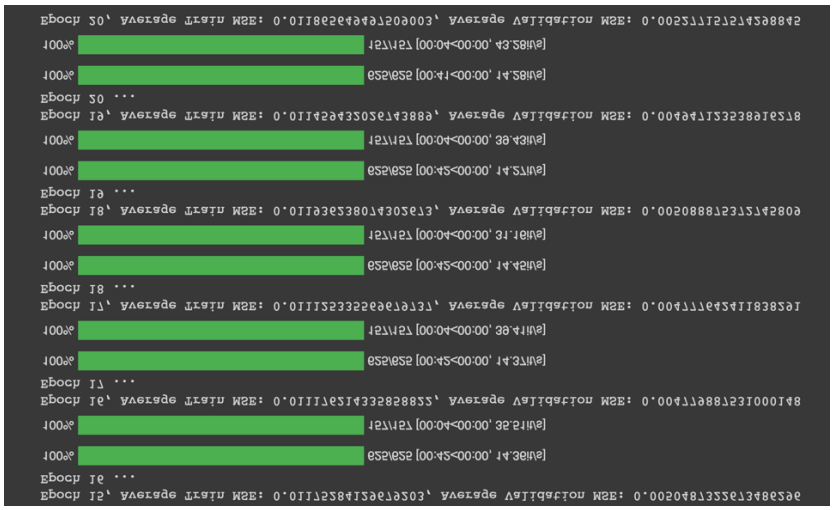
- Attention Mechanism

Now in this section, we will examine the results obtained from these two methods and compare them.

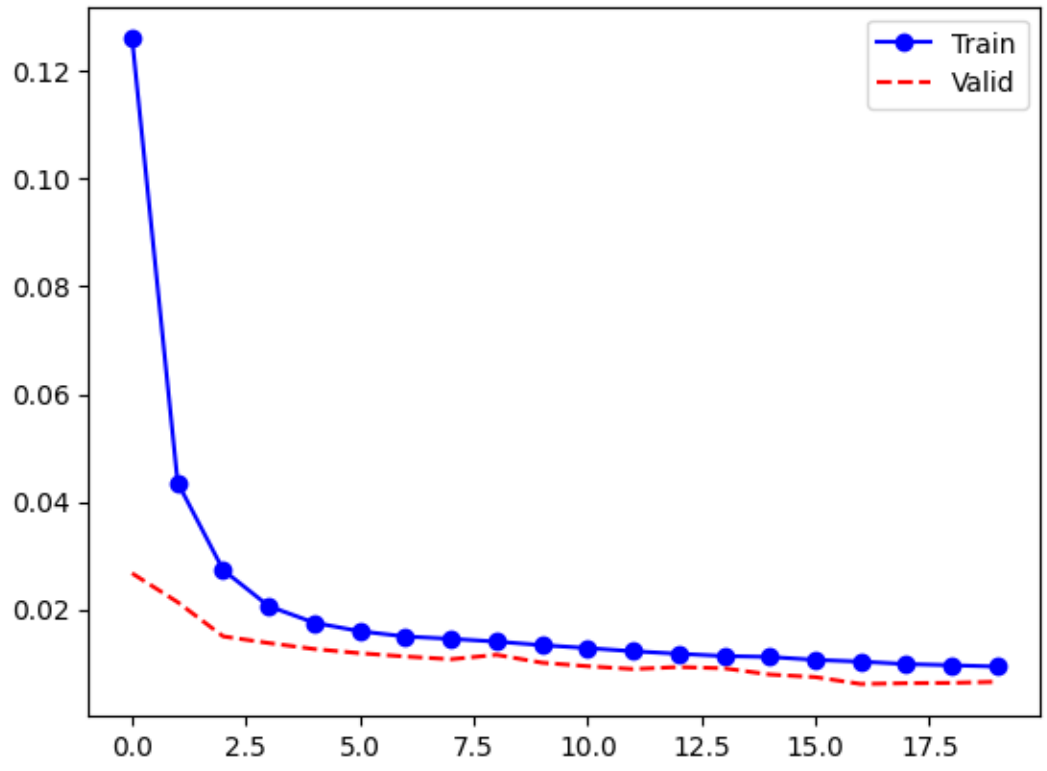
Results of the GAN neural network:

Performing the training process:

After implementing the U-Net neural network and specifying its error function and optimizer, we start the training process. In each iteration of the training process, we print the accuracy and error on the training and validation data. Finally, we save the best model for evaluation on the test data.

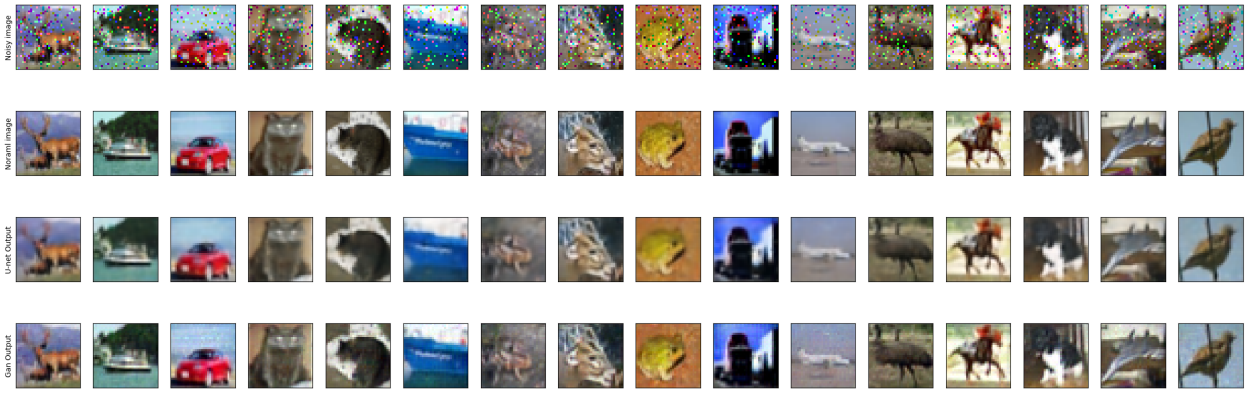


Error plots on training and validation data during the GAN neural network training process:



As you can see in the chart above, the error on the training and validation data is decreasing in each iteration of the training process, and the result is that the model is more successful in removing image noise in each iteration.

The output of the denoised images by the trained GAN model:



Row 1: Images with salt and pepper noise with an amount of 0.07

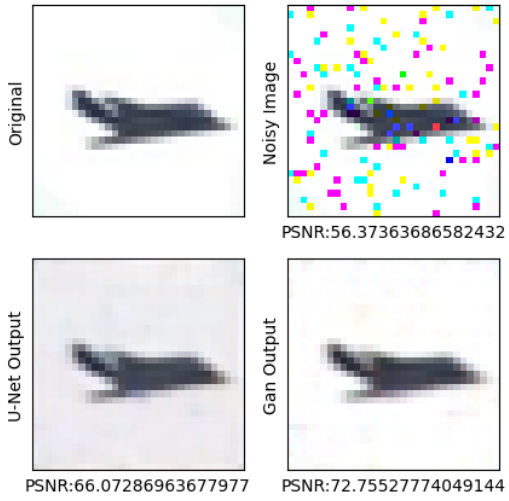
Row 2: Images without noise

Row 3: Images denoised by the U-Net network

Row 4: Images denoised by the GAN network

As can be seen in the image above, the results obtained from the GAN neural network are slightly better than the results of the U-Net neural network reported.

An example output of this method on a specific image:



As can be seen in the image above, about 6 increase in the PSNR metric compared to the U-Net model has been recorded for a specific image. We also see a good visual improvement.

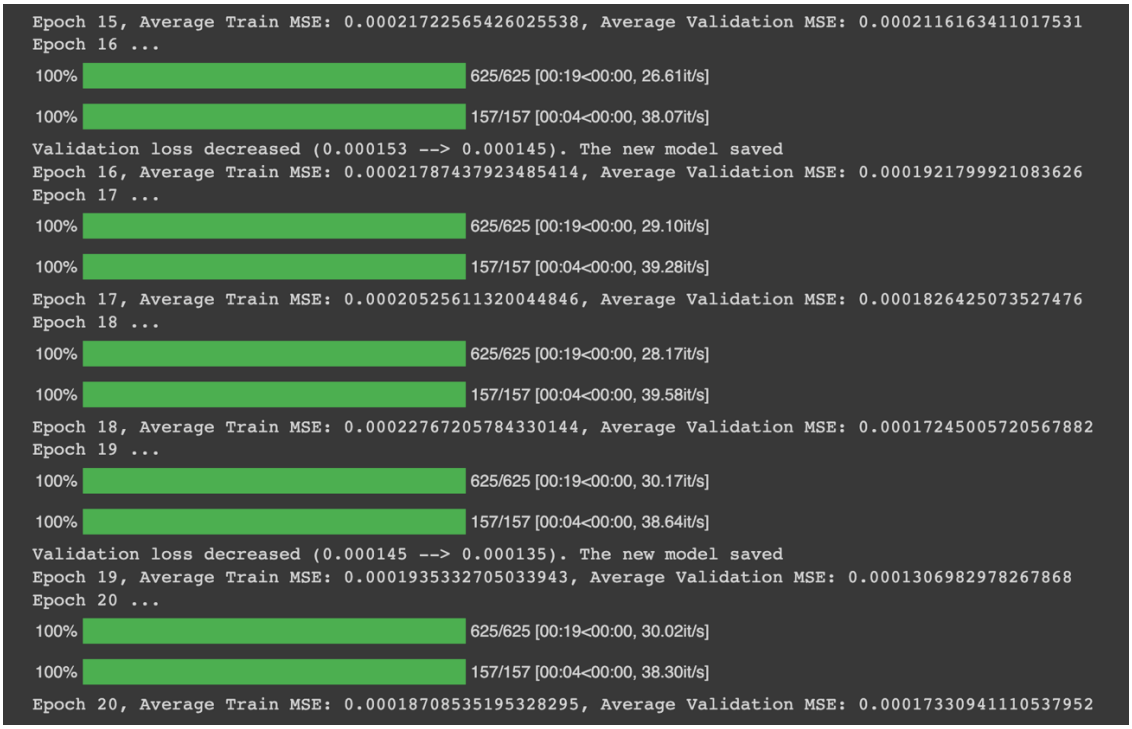
In the next section, after reviewing the results obtained from the neural network using the attention method, we will compare the results obtained from these three methods.

Results of the U-Net neural network and the use of attention blocks:

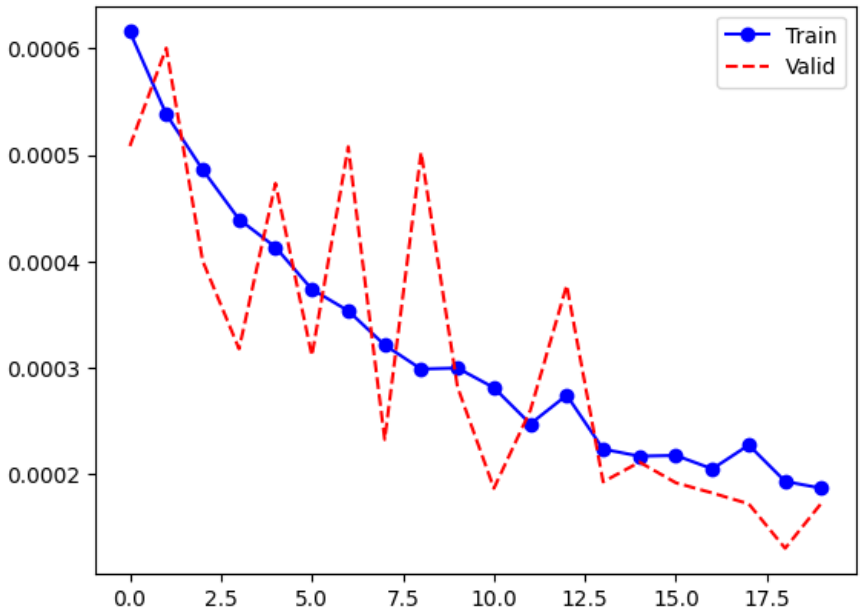
Performing the training process:

After implementing the U-Net neural network and specifying its error function and optimizer, we start the training process. In each iteration of the training process, we print the accuracy and error on the training and validation data. Finally, we save the best model for evaluation on the test data.

You can see the training process in the image below:

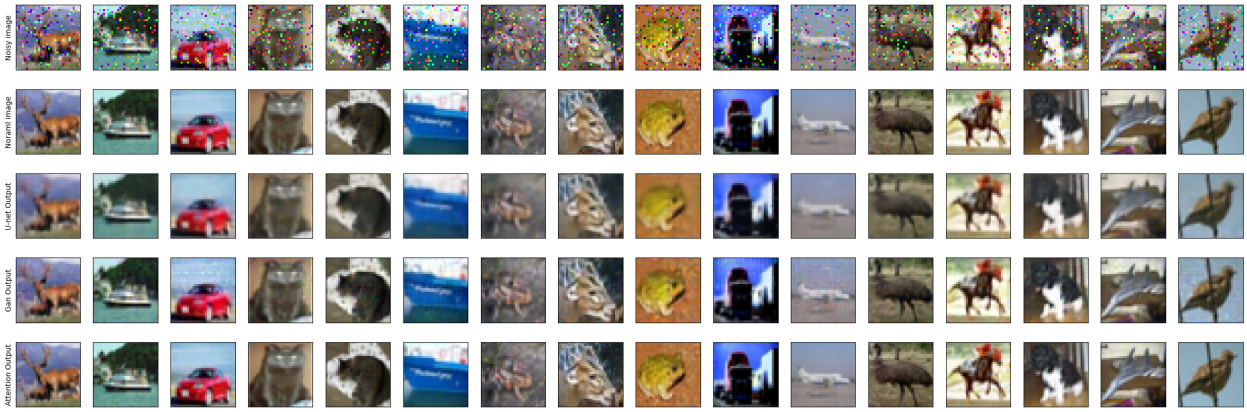


Error plots on training and validation data during the training process of the U-Net neural network using attention blocks:



As can be seen in the chart above, the error on the training and validation data is decreasing in each iteration of the training process, and the result is that the model is more successful in removing image noise in each iteration.

Results of Images Denoised Using U-Net Model and Using Attention Blocks:



First row: Images with salt and pepper noise with 0.07 amount

Second row: Noisy-free images

Third row: Images denoised using U-Net network

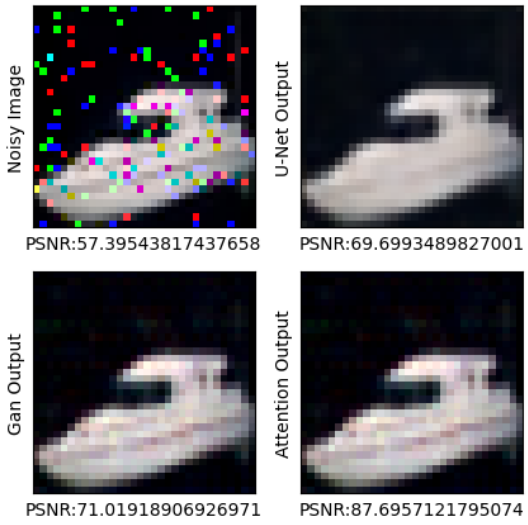
Fourth row: Images denoised using GAN network

Fifth row: Images denoised using Attention block network

For a more precise review, we will calculate the PSNR criterion for the output images of each of these networks.

In the above image, you see the output of a particular image, applying each of the above neural networks.

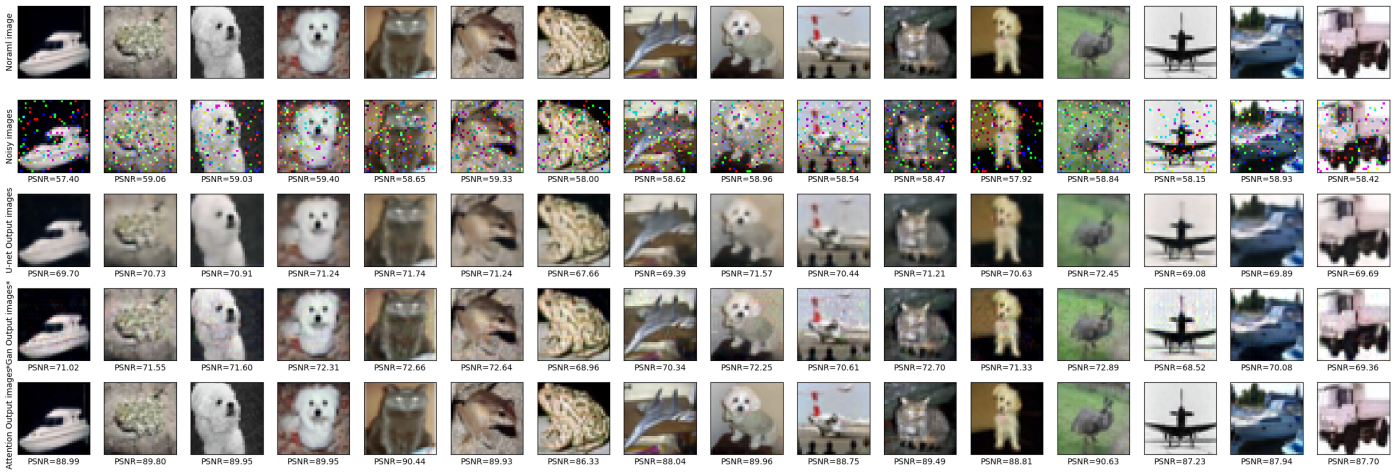
Comparison of output images of these three noise removal methods along with calculating the PSNR criterion for each:



As you can see, it is concluded that, the GAN method and the network using Attention blocks recorded improvements of about 6 and 20 in the PSNR criterion, respectively.

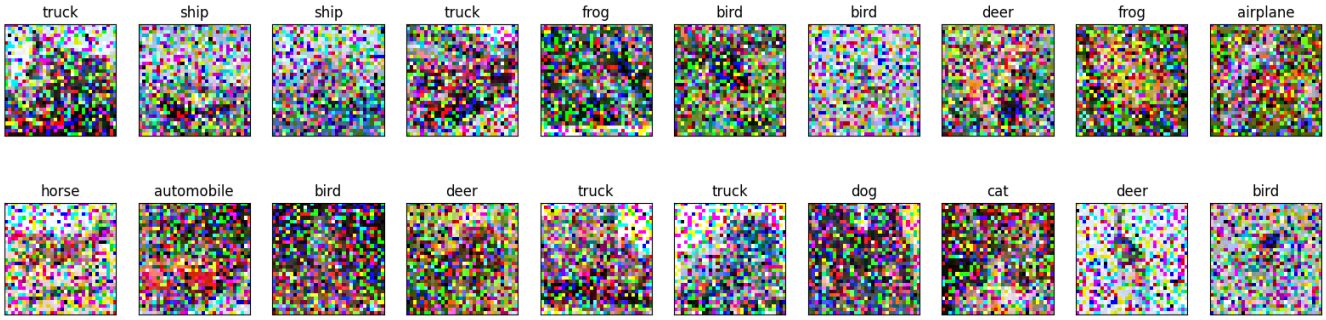
It was concluded that the use of GAN network and U-Net network and the use of attention blocks recorded an acceptable improvement over U-Net network output.

Reviewing the superior noise removal capability of the superior network with a higher noise amount:

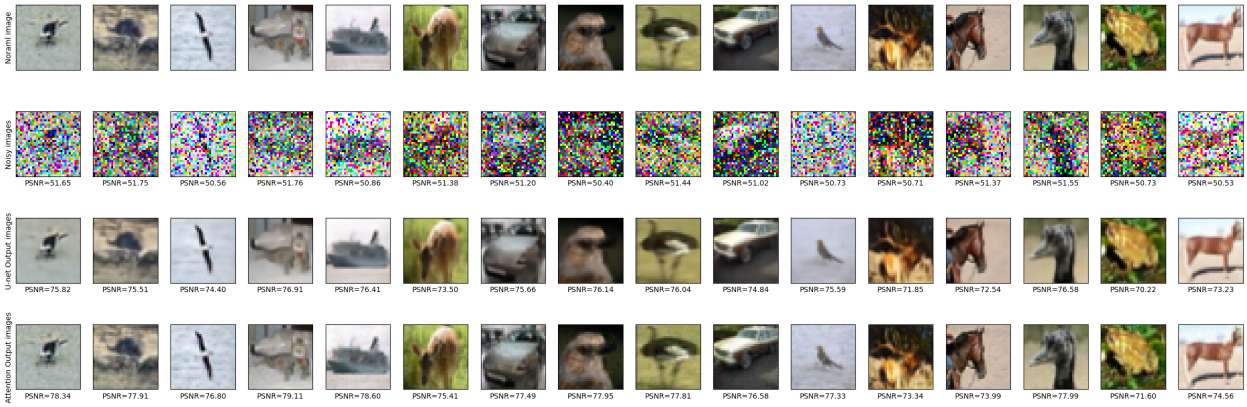


Since the best result was registered by the U-Net model using attention blocks. Another experiment will be performed between the U-Net neural network and the U-Net neural network using attention blocks on images with a higher noise amount.

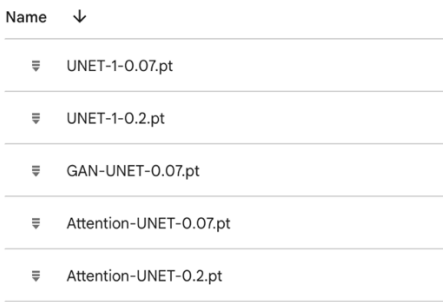
We generate noisy images with a noise amount of 0.04:



The results obtained from the output of the two models:



As you can see, the U-Net neural network using attention blocks was more successful in denoising images with more noise.



All trained and used models have been saved on the drive.