Mounting Google Drive (if using Colab):

- This part assumes you're using Google Colab, a cloud platform for running Python code.
- The `drive.mount('/content/drive')` line mounts your Google Drive to a directory within Colab, allowing you to access your data.

Navigating to the notebook directory:

- It sets the `notebook_directory` variable to the path where your notebook is stored on your mounted drive (assuming it's in `/content/drive/My Drive/Colab Notebooks`).
- It then uses `os.chdir` to change the current working directory to the location of your notebook.
- Finally, it prints the current directory to confirm the successful change.

Switching to the test data directory:

- It defines `test_directory` as the name of the folder containing your test data.
- Similar to step 2, it uses `os.chdir` to navigate to the test directory.
- It again verifies the current directory using `os.getcwd()`.

Defining paths to high and low-quality images:

- It sets variables for `high_quality_dir` and `low_quality_dir` which likely contain the subdirectories within the test directory where your high-quality (clean) and low-quality (noisy) images are stored.
- An `if` statement checks if both directories exist using `os.path.exists`.
- If they exist, it prints the absolute paths of both directories for confirmation.

- If not, it provides an error message indicating you might need to adjust the directory paths.

Finding Image Paths:

- It uses `glob.glob` to search for all image files with the `.png` extension (assuming PNG format) within the directories defined by `high_quality_dir` and `low_quality_dir`.
- `os.path.join` combines the directory path with the wildcard pattern `*.png` to create the search string for `glob.glob`.
- The `sorted` function ensures the order of the filenames is consistent, which might be important for pairing later.

Verifying Image Count:

- An `assert` statement checks if the lengths of `high_quality_images` and `low_quality_images` are equal.
- This is crucial because we need the same number of high-quality (clean) and low-quality (noisy) images to create corresponding pairs for training.
- If the lengths differ, an error message is displayed indicating an inconsistency.

Printing Image Counts:

- It prints the number of images found in both directories for confirmation.
- This gives you an idea of how much data you're working with.

load_image_pairs Function:

- This function takes lists containing the paths to high and low-quality images as input.

- It initializes an empty list `pairs` to store the image pairs.
- It uses `zip` to iterate through both lists simultaneously, matching each high-quality image path with its corresponding low-quality image path.
- For each pair of paths:
    - It uses `cv2.imread` to read the image files, specifying `cv2.IMREAD_COLOR` to load them in color mode.
    - An `if` statement checks if reading either image fails (returns `None`). This can happen due to incorrect paths, invalid image formats, or other issues.
        - If there's an error, it prints the corresponding file paths for troubleshooting.
    - If both images are read successfully, it appends a tuple containing the high-quality image and the low-quality image (as NumPy arrays) to the `pairs` list.
- Finally, the function returns the list of image pairs.

Loading Image Pairs:

- The script calls the `load_image_pairs` function with the lists of high-quality and low-quality image paths obtained earlier.
- The result, which is a list of image pairs, is stored in the `image_pairs` variable.

Checking Loaded Pairs:

- An `if` statement checks if `image_pairs` is empty (no pairs loaded).
- If empty, it indicates a potential problem with file paths, image formats, or the `load_image_pairs` function itself.

- If not empty, it prints a success message indicating the number of image pairs loaded, which is essential for training or evaluating the denoising model.

normalize_image function:

- This function takes a single image as input (either high-quality or low-quality).
- It converts the image data type to `np.float32` (likely because the model might work better with floating-point numbers).
- Then, it divides all pixel values by 255.0 (assuming the image data is in the 0-255 range). This normalization step scales the pixel intensities between 0.0 and 1.0, which can be beneficial for training some deep learning models.

Normalize all image pairs:

- This line uses a list comprehension to iterate through each pair of images in `image_pairs`.
- Inside the loop, it applies the `normalize_image` function to both the high-quality and low-quality images in each pair.
- The result is a new list `normalized_pairs` containing pairs of normalized images.

Check normalization:

- It prints the shapes of the first normalized high-quality and low-quality image in the `normalized_pairs` list.

- This helps verify that the normalization process resulted in the expected data format (usually with pixel values between 0.0 and 1.0).

Extract high-quality and low-quality images from pairs:

- This line uses `zip(*normalized_pairs)` to unpack the normalized image pairs into separate lists.
- The first element of the resulting unpacked lists becomes `high_quality_images` containing the normalized high-quality images.
- Similarly, the second element becomes `low_quality_images` containing the normalized low-quality images.

Split the dataset:

- The script uses `train_test_split` from `sklearn.model_selection` to split the data into training, validation, and test sets.
- The first split (`train_test_split`) divides the combined high-quality (`high_quality_images`) and low-quality (`low_quality_images`) images into training and testing sets with a test size of 20% (`test_size=0.2`).
- It uses a random seed (`random_state=42`) to ensure consistent splitting behavior if you run the code multiple times.
- This results in four separate lists: `hq_train`, `hq_test`, `lq_train`, and `lq_test`.
- The second split (`train_test_split`) further splits the training data (`hq_train` and `lq_train`) into training and validation sets with a validation size of 25% (`test_size=0.25`).
- This ensures you have a set of data to evaluate the model's performance during training (validation set) besides the final test set.

Convert lists to NumPy arrays:

- This section converts the six split data lists (`hq_train`, `hq_val`, `hq_test`, `lq_train`, `lq_val`, and `lq_test`) from Python lists to NumPy arrays.
- NumPy arrays are generally more efficient for numerical computations compared to lists, which is important for training deep learning models.

Print dataset sizes:

- Finally, the code prints the sizes of the training, validation, and test sets for each data type (high-quality and low-quality).
- This allows you to confirm that the splitting process resulted in a balanced distribution of data across the sets.

ImagePairDataset Class:

- This class inherits from `torch.utils.data.Dataset`.
- It serves as a blueprint for creating a dataset specifically designed to handle pairs of high-quality and low-quality images.
- `__init__` function:
  - This function initializes the object when you create an instance of the class.
  - It takes two arguments: `hq_images` (a NumPy array containing the high-quality images) and `lq_images` (a NumPy array containing the low-quality images).
  - Inside the function, it assigns these arguments to the object's attributes `self.hq_images` and `self.lq_images`, essentially storing the data within the class instance.
- `__len__` function:

        - This function defines the length of the dataset.
        - It simply returns the length of the `hq_images` list (assuming both lists have the same length).
        - This tells PyTorch how many data items (image pairs) are in the dataset.
    - `__getitem__` function:
        - This function defines how to access a specific item (image pair) in the dataset.
        - It takes an index (`idx`) as input.
        - It retrieves the high-quality image (`hq_img`) and low-quality image (`lq_img`) at the corresponding index from the `self.hq_images` and `self.lq_images` lists, respectively.
        - It then converts both images from NumPy arrays to PyTorch tensors using `torch.from_numpy`.
        - An important step is that it transposes the image data using `transpose((2, 0, 1))`. This is because PyTorch expects the channel dimension (RGB) to be at the first index by default, whereas NumPy arrays typically have it as the last dimension.
        - Finally, the function returns a tuple containing the processed high-quality and low-quality image tensors.

Create Data Loaders:

- `batch_size` variable defines the number of image pairs to process together in a single batch during training.
- `ImagePairDataset` instances are created for training (`train_dataset`), validation (`val_dataset`), and testing (`test_dataset`) using the corresponding split data (high-quality and low-quality images).

    - `DataLoader` instances are created:
        - Each `DataLoader` takes a dataset object and other configuration options.
        - `batch_size` argument specifies the number of image pairs to include in each batch.
        - `shuffle=True` is set for the training loader to randomly shuffle the data order in each epoch (training iteration) for better generalization.
        - `shuffle=False` is used for validation and test loaders to maintain the original data order for consistent evaluation.

Class Definition:

- `DenoisingCNN(nn.Module)`: This line defines a class named `DenoisingCNN` that inherits from `nn.Module` in PyTorch's `nn` (neural network) module. The `nn.Module` class is the foundation for building neural networks in PyTorch.

Initialization (`__init__` function):

- `super(DenoisingCNN, self).__init__()`: This line calls the `__init__` method of the parent class (`nn.Module`) to initialize the basic functionalities inherited from that class.
- `self.conv1 = nn.Conv2d(3, 64, kernel_size=3, padding=1)`:
    - This line defines a convolutional layer using `nn.Conv2d`.

    - The first argument (`3`) specifies the number of input channels (assuming the input images have 3 channels for RGB).
    - The second argument (`64`) specifies the number of output channels (filters) for this layer. This means the convolution will learn 64 filters to extract features from the input.
    - `kernel_size=3` defines the size of the filter (a 3x3 kernel in this case).
    - `padding=1` specifies padding of 1 around the input, which helps maintain the image dimensions throughout the convolution.
- Similar lines (`self.conv2` and `self.conv3`) define two more convolutional layers with the following changes:
    - `self.conv2` takes 64 input channels (same as the output of the previous layer) and again produces 64 output channels.
    - `self.conv3` takes 64 input channels and produces 3 output channels, likely representing the denoised RGB image.
- `self.relu = nn.ReLU()`: This line defines a ReLU (Rectified Linear Unit) activation function using `nn.ReLU`. This activation function introduces non-linearity into the network, helping it learn more complex patterns.

Forward Pass (`forward` function):

- `def forward(self, x)`: This function defines how the data propagates through the network (forward pass). It takes an input tensor `x` (likely representing a batch of noisy images).
- `x = self.relu(self.conv1(x))`: This line performs the first convolutional operation followed by the ReLU activation.
    - `self.conv1(x)` applies the first convolutional layer to the input `x`.

    - `self.relu( ... )` applies the ReLU activation to the output of the convolution.
- Similar lines (`x = self.relu(self.conv2(x))` and `x = self.conv3(x)`) perform convolutions with ReLU activations for the second and third layers, respectively.
- `return x`: Finally, the function returns the output of the last convolutional layer (`x`), which is expected to be the denoised image.

This code defines a basic 3-layer CNN architecture for image denoising. The convolutional layers with ReLU activations aim to learn features from the noisy input images and reconstruct a clean version at the output. It's important to note that this is a simple example, and more complex architectures with additional layers and techniques can be used for better denoising performance.

Function Definition:

- `def initialize_model():`: This line defines a function named `initialize_model`. This function likely gets called at the beginning of your training script to set up the core elements for training.

Model Creation:

- `model = DenoisingCNN()`: This line creates an instance of the `DenoisingCNN` class defined earlier. This class represents the architecture of your convolutional neural network for image denoising.

Loss Function:

- `criterion = nn.MSELoss()`: This line defines the loss function using `nn.MSELoss` from PyTorch's `nn` module. As explained previously, the loss function calculates the error between the model's predictions (denoised images) and the actual clean (high-quality) images. Here, it uses Mean Squared Error (MSE).

Optimizer:

- `optimizer = optim.Adam(model.parameters(), lr=0.001)`: This line defines the optimizer using `optim.Adam` from PyTorch's `optim` module. The optimizer updates the model's parameters (weights and biases) to minimize the loss function during training.
    - `model.parameters()`: This retrieves all the trainable parameters of the model instance created earlier.
    - `lr=0.001`: This sets the learning rate to 0.001, which controls the magnitude of updates to the model's parameters based on the calculated loss.

Returning Values:

- `return model, criterion, optimizer`: This line returns three values as a tuple:
    - `model`: The created instance of the `DenoisingCNN` class representing your image denoising model.
    - `criterion`: The defined loss function (`nn.MSELoss`) used to measure the error between predictions and ground truth.
    - `optimizer`: The optimizer (`optim.Adam`) used to update the model's parameters during training.

By calling this function, you can conveniently initialize all the necessary components (model, loss function, and optimizer) in one place, promoting code organization and reusability in your training script.

The provided code defines two functions related to data loading for your image denoising model using PyTorch's `DataLoader`:

1. `ImagePairDataset` **Class:**

This class, defined earlier, serves as a blueprint for creating a custom dataset specifically designed to handle pairs of high-quality and low-quality images. It's already been explained in detail in previous responses.

2. `create_dataloaders` **Function:**

- `def create_dataloaders(hq_train, lq_train, hq_val, lq_val, batch_size=16):`: This function takes several arguments:
    - `hq_train`: A NumPy array containing the high-quality training images.
    - `lq_train`: A NumPy array containing the low-quality training images.
    - `hq_val`: A NumPy array containing the high-quality validation images.
    - `lq_val`: A NumPy array containing the low-quality validation images.
    - `batch_size` (optional): The number of image pairs to include in each batch (defaults to 16).

Steps Performed:

1. **Dataset Creation:**

- `train_dataset = ImagePairDataset(hq_train, lq_train)`: This line creates an instance of the `ImagePairDataset` class for the training data. It passes the training high-quality and low-quality image arrays to the `__init__` function of the class.
- `val_dataset = ImagePairDataset(hq_val, lq_val)`: Similarly, this line creates another instance of the `ImagePairDataset` class for the validation data using the validation high-quality and low-quality image arrays.

2. **Dataloader Creation:**
   - `train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)`: This line creates a `DataLoader` object for the training data. It takes the training dataset (`train_dataset`), the desired batch size (`batch_size`), and sets `shuffle=True`. This instructs the `DataLoader` to shuffle the data order in each epoch (training iteration) to prevent the model from overfitting to a specific order of examples.
   - `val_loader = DataLoader(val_dataset, batch_size=batch_size, shuffle=False)`: This line creates a `DataLoader` object for the validation data. It follows the same configuration as the training loader except for `shuffle=False`. Shuffling is not necessary for validation data as you want to evaluate the model's performance on the same set of images consistently.
3. **Return Values:**
   - `return train_loader, val_loader`: Finally, the function returns a tuple containing the two `DataLoader` objects:
     - `train_loader`: Used to iterate through the training data in batches during training.

        - `val_loader`: Used to iterate through the validation data in batches during model validation.

This function promotes code organization and reusability by encapsulating the data loader creation process within a function. You can call this function with your training and validation data to easily generate data loaders for your image denoising model.

1. Function Definition:

- `def train_model(model, train_loader, val_loader, criterion, optimizer, num_epochs=3, device=torch.device('cuda' if torch.cuda.is_available() else 'cpu')):`
    - This line defines the `train_model` function. It takes several arguments:
        - `model`: The PyTorch model instance representing your denoising CNN.
        - `train_loader`: The `DataLoader` object for iterating through the training data in batches.
        - `val_loader`: The `DataLoader` object for iterating through the validation data in batches.
        - `criterion`: The loss function (e.g., `nn.MSELoss`) used to measure the error between predictions and ground truth.
        - `optimizer`: The optimizer (e.g., `optim.Adam`) used to update the model's parameters based on the loss.

        - `num_epochs` (optional): The number of training epochs (defaults to 3).
        - `device` (optional): The device to use for training (defaults to 'cuda' if available, otherwise 'cpu').

2. Move Model to Device:

- `model.to(device)`: This line moves the model to the specified device (`device`). This allows the model to utilize GPU acceleration if a CUDA-enabled GPU is available, potentially speeding up training.

3. Training Loop (Epochs):

- `for epoch in range(num_epochs)`: This loop iterates over the specified number of epochs (training iterations).

  Training Phase (Per Epoch):
  - `model.train()`: Sets the model to training mode. This might affect some layers' behavior (e.g., dropout) for better training.
  - `train_loss = 0.0`: Initializes a variable to store the accumulated training loss for the current epoch.

    Training Batch Loop:
    - `for inputs, targets in train_loader` This loop iterates over the training data batches provided by the `train_loader`.
      - `inputs`: Represents a batch of low-quality images.
      - `targets`: Represents a batch of corresponding high-quality (clean) images.
    - **It is working** (This line seems like a placeholder comment and can be removed)

        - `inputs, targets = inputs.to(device), targets.to(device)`: Moves the current batch of data (inputs and targets) to the specified device (`device`).
        - `optimizer.zero_grad()`: Sets the gradients of the model's parameters to zero before calculating the loss for the current batch. This ensures proper gradient accumulation for each batch.
        - `outputs = model(inputs)`: Performs a forward pass through the model using the current batch of low-quality images (`inputs`). This results in the predicted denoised images (`outputs`).
        - `loss = criterion(outputs, targets)`: Calculates the loss between the predicted denoised images (`outputs`) and the actual clean images (`targets`) using the loss function (`criterion`).
        - `loss.backward()`: Performs backpropagation to calculate the gradients of the loss function with respect to the model's parameters.
        - `optimizer.step()`: Updates the model's parameters based on the calculated gradients using the optimizer (`optimizer`).
        - `train_loss += loss.item() * inputs.size(0)`: Accumulates the current batch's loss to the total training loss for the epoch. `inputs.size(0)` represents the batch size.
    - `train_loss /= len(train_loader.dataset)`: Calculates the average training loss for the epoch by dividing the accumulated loss by the total number of training examples.

- **Validation Phase (Per Epoch):**
    - `model.eval()`: Sets the model to evaluation mode. This might disable certain layers used during training (e.g., dropout).
    - `val_loss = 0.0`: Initializes a variable to store the accumulated validation loss for the current epoch.

      Validation Batch Loop (with torch.no_grad):
        - `with torch.no_grad()`: This context manager disables gradient calculation during validation to improve efficiency since we don't need gradients for updating parameters during validation.
            - `for inputs_val, targets_val in val_loader`: This loop iterates over the validation data batches provided by the `val_loader`. Similar to the training loop.
            - `inputs_val, targets_val = inputs_

1. Example Usage:

- **Comments:** These lines assume you have already defined NumPy arrays named `hq_train`, `lq_train`, `hq_val`, and `lq_val` containing your high-quality and low-quality training and validation images, respectively.

2. Create Dataloaders:

- `batch_size = 16`: Sets the batch size for data loading (16 images per batch in this case).

- `train_loader, val_loader = create_dataloaders(hq_train, lq_train, hq_val, lq_val, batch_size=batch_size)`: This line calls the `create_dataloaders` function to create the data loaders for training and validation using your image data and the specified batch size.

3. Initialize Model, Criterion, Optimizer:

- `model, criterion, optimizer = initialize_model()`: This line calls the `initialize_model` function to create an instance of your denoising CNN model (`model`), define the loss function (`criterion`), and set up the optimizer (`optimizer`) for training.

4. Train the Model:

- `trained_model = train_model(model, train_loader, val_loader, criterion, optimizer, num_epochs=2)`: This line calls the `train_model` function to train the model. It passes the model instance, data loaders, loss function, optimizer, and the desired number of training epochs (2 in this case). The function trains the model and returns the trained model (`trained_model`).

5. Save the Trained Model:

- `torch.save(trained_model.state_dict(), '/content/denoising_model.pth')`: This line saves the trained model. It uses `torch.save` to save only the model's state dictionary (containing the learned weights and biases) to a file named `denoising_model.pth` in the `/content` directory (likely a Google Colab environment).

1. Load the Saved Model:

- `model = DenoisingCNN()`: This line creates a new instance of the `DenoisingCNN` class, essentially defining the model architecture.
- `model.load_state_dict(torch.load('denoising_model.pth'))`: This line loads the weights and biases (parameters) of the trained model from the saved file `denoising_model.pth` using `torch.load`. It essentially fills the new model instance with the learned parameters from your training process.

2. Evaluation Mode:

- `model.eval()`: This line sets the model to evaluation mode. This might disable certain layers used during training (e.g., dropout) to ensure a more deterministic evaluation process.

3. Evaluate on Test Set:

- **Assuming `test_loader` is defined:** This code snippet assumes you have already created a `test_loader` using the `create_dataloaders` function for your test data (high-quality and low-quality image pairs).
- `test_loss = 0.0`: Initializes a variable to store the accumulated test loss.

Test Batch Loop (with torch.no_grad):

- `with torch.no_grad()`: This context manager disables gradient calculation during evaluation for efficiency since we don't need gradients to update parameters during testing.
    - `for inputs_test, targets_test in test_loader`: This loop iterates over the test data batches provided by the `test_loader`.

        - `inputs_test`: Represents a batch of low-quality images from the test set.
        - `targets_test`: Represents a batch of corresponding high-quality (clean) images from the test set.
    - `outputs_test = model(inputs_test.float())`: Performs a forward pass through the loaded model on the current batch of low-quality test images (`inputs_test`). The `inputs_test.float()` converts the input tensor to float data type if necessary.
    - `loss = criterion(outputs_test, targets_test.float())`: Calculates the loss between the predicted denoised images (`outputs_test`) and the actual clean images (`targets_test`) from the test set using the loss function (`criterion`).
    - `test_loss += loss.item() * inputs_test.size(0)`: Accumulates the current batch's loss to the total test loss.

4. Calculate and Print Average Test Loss:

- `test_loss /= len(test_loader.dataset)`: Calculates the average test loss by dividing the accumulated loss by the total number of test examples.
- `print(f'Test Loss: {test_loss:.4f}')`: Prints the calculated average test loss, giving you an indication of how well the trained model performs on unseen data (the test set).

1. Imports:

- `import torch`: Imports the PyTorch library for deep learning operations.

- `from skimage.metrics import peak_signal_noise_ratio, structural_similarity`: Imports the `peak_signal_noise_ratio` and `structural_similarity` functions from the `skimage.metrics` module for image quality assessment.

2. `calculate_metrics` Function:

- `def calculate_metrics(denoised, original):`: This function takes two arguments:
    - `denoised`: A NumPy array representing the denoised image.
    - `original`: A NumPy array representing the corresponding clean (original) image.
- `psnr_value = peak_signal_noise_ratio(original, denoised)`: This line calculates the PSNR between the denoised and original images using the `peak_signal_noise_ratio` function. PSNR is a common metric in image restoration, measuring the peak signal compared to the noise level. Higher PSNR indicates better denoising.
- `ssim_value, _ = structural_similarity(original, denoised, win_size=5, full=True, multichannel=True)`: This line calculates the SSIM between the denoised and original images using the `structural_similarity` function. SSIM considers both structural similarity and luminance information. It outputs two values: the SSIM index (between 0 and 1, higher is better) and additional information (discarded here using `_`).
- `return psnr_value, ssim_value`: The function returns both the PSNR and SSIM values for the image pair.

3. PSNR and SSIM Calculation on Test Set:

- `psnr_values = []`: This line initializes an empty list to store PSNR values for all test images.
- `ssim_values = []`: This line initializes an empty list to store SSIM values for all test images.

4. Test Batch Loop (with torch.no_grad):

- `with torch.no_grad()`: This context manager disables gradient calculation during evaluation for efficiency since we don't need gradients to update parameters during testing.
    - `for idx, (lq_tensor, original_img) in enumerate(test_loader)`: This loop iterates over the test data batches provided by the `test_loader`.
        - `idx`: The index of the current batch.
        - `lq_tensor`: A PyTorch tensor representing the low-quality image from the test set.
        - `original_img`: A PyTorch tensor representing the corresponding high-quality (clean) image from the test set.
    - `denoised_tensor = model(lq_tensor.float())`: Performs a forward pass through the model on the current low-quality test image (`lq_tensor`). `.float()` converts the input tensor to float data type if necessary.
    - `denoised_img = denoised_tensor.squeeze().cpu().numpy()`: Converts the output tensor (`denoised_tensor`) from the model to a

NumPy array (`denoised_img`) suitable for metric calculation. `.squeeze()` removes any unnecessary dimensions (e.g., batch dimension) and `.cpu()` moves the data to CPU memory if it was on GPU.

- `original_img = original_img.squeeze().cpu().numpy()`: Converts the original image tensor (`original_img`) to a NumPy array (`original_img`) for metric calculation.

:

1. `skimage.transform.resize` **Function:**

This function from the scikit-image library resizes an image to a desired output shape.

2. Resizing Images:

- `denoised_img = skimage.transform.resize(denoised_img, (500, 700))`: This line resizes the `denoised_img` (which is a NumPy array representing the denoised image) to a new shape of (500, 700). This means the image will have a height of 500 pixels and a width of 700 pixels after resizing.
- `original_img = skimage.transform.resize(original_img, (500, 700))`: This line performs the same operation on the `original_img` (representing the clean image), also resizing it to (500, 700).

3. Purpose of Resizing:

There could be a few reasons for resizing the images:

- **Standardization:** Resizing all images to a uniform size can be helpful for some downstream tasks or comparisons where consistent image dimensions are preferred.
- **Visualization:** If you're planning to display the images together (denoised and original), resizing them to the same size ensures they appear at the same scale for easier visual comparison.

4. Important Note:

Resizing can introduce artifacts or information loss depending on the resizing method used by `skimage.transform.resize`. If preserving image details is crucial, consider using interpolation methods like `bilinear` or `cubic` during resizing (these are not explicitly specified in the provided code snippet, so check the default behavior or documentation for `skimage.transform.resize`).

.