**Deep Image Prior**

**Introduction**

This report presents a novel approach to JPEG restoration using a Convolutional Neural Network (CNN) with Deep Image Prior (DIP). This method exploits the inherent structural properties of CNNs to enhance image quality without necessitating extensive training datasets. The algorithm is implemented in Python, utilizing libraries such as NumPy, Matplotlib, OpenCV, PyTorch, and Scikit-Image for image processing and neural network construction.

Deep Image Prior (DIP) leverages the CNN architecture as an implicit regularizer for solving inverse problems in image processing. Unlike traditional methods that require large datasets for training, DIP enables image reconstruction by fitting a CNN directly to a single corrupted image. This approach is particularly effective for tasks such as denoising, inpainting, and super-resolution.

**Workflow Overview:**

1. True Image: This is the original, uncorrupted image that we aim to reconstruct through the denoising process.

2. Noisy Image: This is the input image with gaps and noise that we aim to clean using the CNN.

3. Loss Function: The loss function is used to compare the output of the CNN with the noisy image. Specifically, the mean squared error (MSE) is calculated by comparing each pixel in the original image with the corresponding pixel in the noisy image. This discrepancy is minimized during the training process.

4. CNN (Deep Image Prior Model): The CNN is supplied with a fixed, random input image `z`. Through training, this input is mapped to an output that approximates the true, denoised image. The network learns to fill in gaps and remove noise by optimizing the weights to minimize the loss function.

5. Libraries Used:

NumPy: Core Python library for handling N-dimensional arrays and scientific programming.

Matplotlib: Used for plotting images and visualizing the results.

OpenCV (cv2): Utilized here primarily for waiting, to allow Matplotlib time to plot images.

Scikit-Image (skimage): Used for image processing tasks, such as resizing the image.

6. Resizing: The image is resized to 128x128 pixels and scaled by a factor of 100 to adjust the noise level by manipulating the amplitude.

**Setting Up the CNN Using PyTorch**

1. Inherit from `torch.nn.Module`: The CNN class is derived from `torch.nn.Module`, which is the base class for all neural network modules in PyTorch.

2. `nn.Sequential`: This is used to define a sequential container to set up the sequence of layers in the CNN model.

3. Layers:

First Layer: Takes a single-channel input image and applies multiple kernels to produce `n\_chan` output channels. Each kernel is of size 3x3 with padding of 1 pixel on all sides.

PReLU (Parametric ReLU): A type of activation function that allows for learning the parameter of the rectifier.

Intermediate Layers: Consist of convolutional layers with `n\_chan` input and output channels, followed by PReLU activations.

Final Layer: Outputs a single-channel image, matching the single-channel input.

4. Input Image: A random 2D array created using `torch.rand` is used as the initial input to the CNN. This array is trainable and adjusted during the training process.

5. Noisy Version: A noisy version of the original image is created using a Poisson distribution, which is then used to train the CNN.

**Normalizing the RMS Value Using `nrmse\_fn`**

The Normalized Root Mean Squared Error (NRMSE) function (`nrmse\_fn`) is used to evaluate the quality of the reconstruction by comparing the reconstructed image to the true image. The NRMSE is calculated as follows:

Numerator: The squared differences between the reconstructed and true images.

Denominator: The squared values of the true image.

Normalization: The NRMSE is scaled to a percentage by multiplying by 100.

**Training Loop**

1. Optimizer: An Adam optimizer is used to update the CNN's weights based on the gradients calculated from the loss function.

The loss function in this JPEG restoration algorithm is critical as it guides the training of the Convolutional Neural Network (CNN) to improve image quality. Here, we use the Mean Squared Error (MSE) as the primary component of the loss function, which compares the output of the CNN with the noisy input image. Additionally, we normalize the error using the Normalized Root Mean Squared Error (NRMSE) function to provide a more interpretable metric.

#### Mean Squared Error (MSE)

The MSE is a common loss function used in regression tasks, including image restoration. It measures the average squared difference between the predicted values (output of the CNN) and the actual values (noisy image). The formula for MSE is:

Where:

* N is the total number of pixels in the image.
* ​ is the true pixel value at position
* is the predicted pixel value at position

In the context of our algorithm:

* ​ corresponds to the pixel values of the noisy image.
* corresponds to the pixel values of the reconstructed image generated by the CNN.

#### Normalized Root Mean Squared Error (NRMSE)

The NRMSE provides a normalized version of the error, making it easier to interpret the reconstruction quality. It scales the error by the magnitude of the true image, converting it into a percentage. The formula for NRMSE is:

Where:

* The numerator is the Root Mean Squared Error (RMSE), which is the square root of the MSE.
* The denominator is the root mean square of the true image's pixel values.

The NRMSE is calculated in the `nrmse\_fn` function as follows:

1. **Numerator**: Compute the squared differences between the reconstructed and true images, then take the mean of these differences.
2. **Denominator**: Compute the mean of the squared true image values.
3. **Normalization**: Scale the RMSE by the square root of the mean squared true image values and multiply by 100 to express the error as a percentage.

2. Training Steps:

Zero Gradients: The gradients are zeroed before each step.

Forward Pass: The CNN processes the input image to produce an output.

Loss Calculation: The loss is calculated by comparing the output to the noisy image using the NRMSE function.

Backpropagation: The loss is backpropagated to calculate gradients.

Optimization Step: The optimizer updates the weights of the CNN.

3. Tracking Best Epoch: The best reconstruction and corresponding epoch are tracked based on the lowest NRMSE value observed so far.

4. Visualization: Periodically, the current state of the training process is visualized:

Clearing Axes: The axes are cleared to update the plots.

Plotting: The reconstructed images and error metrics are plotted using Matplotlib.

5. Waiting: `cv2.waitKey(1)` is used to introduce a slight delay, allowing time for the plots to update.

This process iteratively refines the CNN's output, reducing the noise and filling in gaps, resulting in a high-quality reconstruction of the original image.

### **Conclusion and Outcomes**

The approach using Deep Image Prior (DIP) and CNNs demonstrates significant potential for JPEG restoration, effectively reconstructing high-quality images from noisy inputs without extensive training datasets. The methodology harnesses the implicit regularization properties of CNNs, making it particularly effective for image denoising, inpainting, and super-resolution tasks. The implementation in Python, leveraging libraries such as PyTorch, ensures efficient neural network construction and training.

Key outcomes from this research include:

* Successful application of DIP for high-quality image reconstruction.
* Efficient training and optimization using PyTorch, achieving substantial noise reduction and gap filling.
* Validation of the methodology through NRMSE metrics, showing improved reconstruction quality.
* Visualization and tracking of training progress, ensuring the model's performance meets the desired criteria.

This report underscores the feasibility and effectiveness of using CNNs with DIP for JPEG restoration, providing a robust framework for future research and practical applications in image processing.

Top of Form

Bottom of Form