# Detect & Correct Pixelation

## Introduction

Pixelation is a common issue that arises in digital images, leading to a degradation in visual quality and hindering the performance of computer vision tasks. This phenomenon occurs due to a variety of factors, including low-resolution scaling, aggressive compression, and the use of poor-quality source materials.

When an image is scaled down to a lower resolution, the individual pixels become more prominent, resulting in a blocky, pixelated appearance. Similarly, aggressive compression techniques, such as JPEG compression, can introduce artifacts and distortions that manifest as visible pixels. Additionally, images captured with low-quality cameras or sourced from low-resolution online platforms may inherently exhibit pixelation.

The detrimental effects of pixelation extend beyond the aesthetic realm. In the context of computer vision applications, such as object detection, image classification, and segmentation, pixelation can significantly impact the accuracy and reliability of these algorithms. The presence of distinct pixel boundaries can confuse and mislead the underlying machine learning models, leading to suboptimal performance.

To address this challenge, this comprehensive approach proposes the use of advanced machine learning and image processing techniques to detect and correct pixelation in digital images. By leveraging the power of deep learning and image enhancement algorithms, the aim is to restore the visual quality of pixelated images, ultimately improving the performance of computer vision tasks and enhancing the overall user experience.

## Libraries Used

In this project, we utilized several Python libraries to tackle the problem of detecting and correcting pixelation in digital images:

1. **OpenCV**: OpenCV (Open Source Computer Vision Library) is a widely-used library for computer vision and image processing tasks. It provides a comprehensive set of tools and functions for image manipulation, feature extraction, and object detection, making it an essential component of our pixelation detection and correction pipeline.
2. **TensorFlow and Keras**: TensorFlow is a powerful open-source library for machine learning and deep learning, while Keras is a high-level neural networks API that runs on top of TensorFlow. These libraries were used to build and train a convolutional neural network (CNN) model for the task of pixelation detection. The CNN model is trained to identify the presence and extent of pixelation in input images, serving as the core of our pixelation detection mechanism.
3. **Matplotlib and NumPy**: Matplotlib is a data visualization library that allows us to create high-quality plots and visualizations, which are crucial for analyzing the performance and results of our pixelation detection and correction methods. NumPy, on the other hand, is a fundamental library for scientific computing in Python, providing support for large, multi-dimensional arrays and matrices, as well as a large collection of high-level mathematical functions to operate on these arrays.
4. **scikit-image**: The scikit-image library is a collection of algorithms for image processing in Python. In this project, we utilize scikit-image to evaluate the quality of the corrected images using commonly used metrics, such as Peak Signal-to-Noise Ratio (PSNR) and Structural Similarity Index (SSIM). These metrics help us quantify the improvement in image quality after applying our pixelation correction techniques.

By leveraging these powerful libraries, we are able to build a comprehensive solution for detecting and correcting pixelation in digital images, ultimately enhancing the performance of computer vision tasks and improving the overall user experience.

## Algorithm

The algorithm implemented in this report consists of three main parts: Data Processing, Pixelation Detection Model, and Pixelation Correction.

### Data Processing

In the Data Processing stage, we first read the input images using the OpenCV library. We then apply JPEG compression to the images to introduce pixelation artifacts, simulating the real-world scenario where images may be subjected to aggressive compression. Finally, we scale down the images to create pixelated versions, which will serve as the training and testing data for the Pixelation Detection Model.

### Pixelation Detection Model

The Pixelation Detection Model is based on a Convolutional Neural Network (CNN) architecture. We start by performing data augmentation on the prepared datasets, which includes techniques such as random cropping, flipping, and rotation. This step helps to increase the diversity of the training data and improve the model's generalization capabilities.

Next, we design the CNN model, which takes the pixelated images as input and outputs a binary classification prediction, indicating the presence or absence of pixelation. The model architecture consists of multiple convolutional layers, followed by pooling layers, and fully connected layers. We utilize popular activation functions, such as ReLU, to introduce non-linearity and improve the model's learning capabilities.

The CNN model is then trained on the augmented dataset, using techniques like early stopping and learning rate scheduling to optimize the training process and prevent overfitting. The trained model is then evaluated on a separate test set to assess its performance in detecting pixelation.

### Pixelation Correction

In the Pixelation Correction segment, we employ image denoising and adaptive filtering techniques to remove the pixelation artifacts from the input images. Specifically, we utilize the Non-Local Means (NLM) denoising algorithm from the scikit-image library to smooth out the pixel boundaries and reduce the blocky appearance.

Additionally, we apply adaptive filtering techniques, such as the Guided Image Filter, to further enhance the image quality. The Guided Image Filter leverages the structure of the input image to perform edge-preserving smoothing, effectively correcting the pixelation while maintaining important image details.

Finally, we visualize the results of the pixelation correction process, comparing the original pixelated images with the corrected versions. This allows us to evaluate the effectiveness of our approach and quantify the improvement in image quality using metrics like Peak Signal-to-Noise Ratio (PSNR) and Structural Similarity Index (SSIM).

## Code: Data Processing

To implement the data processing steps for the pixelation detection and correction pipeline, we will use the following Python code:

import cv2  
import os  
import numpy as np  
from PIL import Image  
  
# Set the input and output directories  
input\_dir = 'path/to/input/images'  
output\_dir = 'path/to/output/images'  
  
# Define the compression quality and scaling factors  
compression\_quality = 30  
scale\_factor = 0.5  
  
# Loop through the input images  
for filename in os.listdir(input\_dir):  
 # Read the input image using OpenCV  
 img = cv2.imread(os.path.join(input\_dir, filename))  
  
 # Simulate pixelation through JPEG compression  
 compressed\_img = cv2.imencode('.jpg', img, [int(cv2.IMWRITE\_JPEG\_QUALITY), compression\_quality])[1]  
 compressed\_img = cv2.imdecode(compressed\_img, cv2.IMREAD\_COLOR)  
  
 # Downscale the image using various interpolation methods  
 downscaled\_img = cv2.resize(compressed\_img, None, fx=scale\_factor, fy=scale\_factor, interpolation=cv2.INTER\_NEAREST)  
 upscaled\_img = cv2.resize(downscaled\_img, img.shape[:2][::-1], interpolation=cv2.INTER\_CUBIC)  
  
 # Save the processed images  
 cv2.imwrite(os.path.join(output\_dir, f'compressed\_{filename}'), compressed\_img)  
 cv2.imwrite(os.path.join(output\_dir, f'downscaled\_{filename}'), downscaled\_img)  
 cv2.imwrite(os.path.join(output\_dir, f'upscaled\_{filename}'), upscaled\_img)

In this code, we first set the input and output directories for the image processing. We then define the JPEG compression quality and the scaling factor to be used for downscaling and upscaling the images.

Next, we loop through the input images in the specified directory. For each image, we perform the following steps:

1. Read the input image using OpenCV's cv2.imread() function.
2. Simulate pixelation by applying JPEG compression to the image using cv2.imencode() and cv2.imdecode().
3. Downscale the compressed image using cv2.resize() with the cv2.INTER\_NEAREST interpolation method.
4. Upscale the downscaled image using cv2.resize() with the cv2.INTER\_CUBIC interpolation method.
5. Save the compressed, downscaled, and upscaled images to the output directory using cv2.imwrite().

The resulting images will have the following filenames:

* compressed\_<original\_filename>
* downscaled\_<original\_filename>
* upscaled\_<original\_filename>

These processed images will be used in the subsequent steps of the pixelation detection and correction pipeline.

## Results:



## 

## Code: Pixelation Detection Model

To build and train a convolutional neural network (CNN) model for detecting pixelation, we can use the following Python code:

from tensorflow.keras.preprocessing.image import ImageDataGenerator  
from tensorflow.keras.models import Sequential  
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout  
from tensorflow.keras.optimizers import Adam  
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau  
  
# Set up the data generators  
train\_datagen = ImageDataGenerator(  
 rescale=1./255,  
 rotation\_range=20,  
 width\_shift\_range=0.2,  
 height\_shift\_range=0.2,  
 horizontal\_flip=True,  
 vertical\_flip=True  
)  
  
test\_datagen = ImageDataGenerator(rescale=1./255)  
  
train\_generator = train\_datagen.flow\_from\_directory(  
 'path/to/train/data',  
 target\_size=(128, 128),  
 batch\_size=32,  
 class\_mode='binary'  
)  
  
test\_generator = test\_datagen.flow\_from\_directory(  
 'path/to/test/data',  
 target\_size=(128, 128),  
 batch\_size=32,  
 class\_mode='binary'  
)  
  
# Build the CNN model  
model = Sequential()  
model.add(Conv2D(32, (3, 3), activation='relu', input\_shape=(128, 128, 3)))  
model.add(MaxPooling2D((2, 2)))  
model.add(Conv2D(64, (3, 3), activation='relu'))  
model.add(MaxPooling2D((2, 2)))  
model.add(Conv2D(128, (3, 3), activation='relu'))  
model.add(MaxPooling2D((2, 2)))  
model.add(Flatten())  
model.add(Dense(128, activation='relu'))  
model.add(Dropout(0.5))  
model.add(Dense(1, activation='sigmoid'))  
  
# Compile the model  
model.compile(optimizer=Adam(lr=0.001), loss='binary\_crossentropy', metrics=['accuracy'])  
  
# Train the model  
early\_stopping = EarlyStopping(monitor='val\_loss', patience=5, verbose=1)  
reduce\_lr = ReduceLROnPlateau(monitor='val\_loss', factor=0.1, patience=3, verbose=1)  
  
model.fit(  
 train\_generator,  
 epochs=50,  
 validation\_data=test\_generator,  
 callbacks=[early\_stopping, reduce\_lr]  
)  
  
# Save the trained model  
model.save('pixelation\_detection\_model.h5')

In this code, we first set up the data generators using the ImageDataGenerator class from Keras. We apply various data augmentation techniques, such as rotation, shifting, and flipping, to the training data to increase the diversity of the dataset and improve the model's generalization capabilities.

Next, we define the CNN model architecture. The model consists of multiple convolutional layers, followed by max-pooling layers, a flattening layer, and finally, dense layers with a dropout layer to prevent overfitting.

We then compile the model using the Adam optimizer and binary cross-entropy loss function, as this is a binary classification problem (detecting the presence or absence of pixelation).

To train the model, we use the fit() method, passing in the training and validation data generators. We also include two callbacks: EarlyStopping to stop the training when the validation loss stops improving, and ReduceLROnPlateau to dynamically adjust the learning rate during training.

Finally, we save the trained model to a file named pixelation\_detection\_model.h5, which can be used for inference and further integration with the pixelation correction pipeline.

This code provides a solid foundation for building and training a CNN model for pixelation detection. You can further fine-tune the model architecture, hyperparameters, and training process based on your specific requirements and the characteristics of your dataset.

## Code: Pixelation Correction

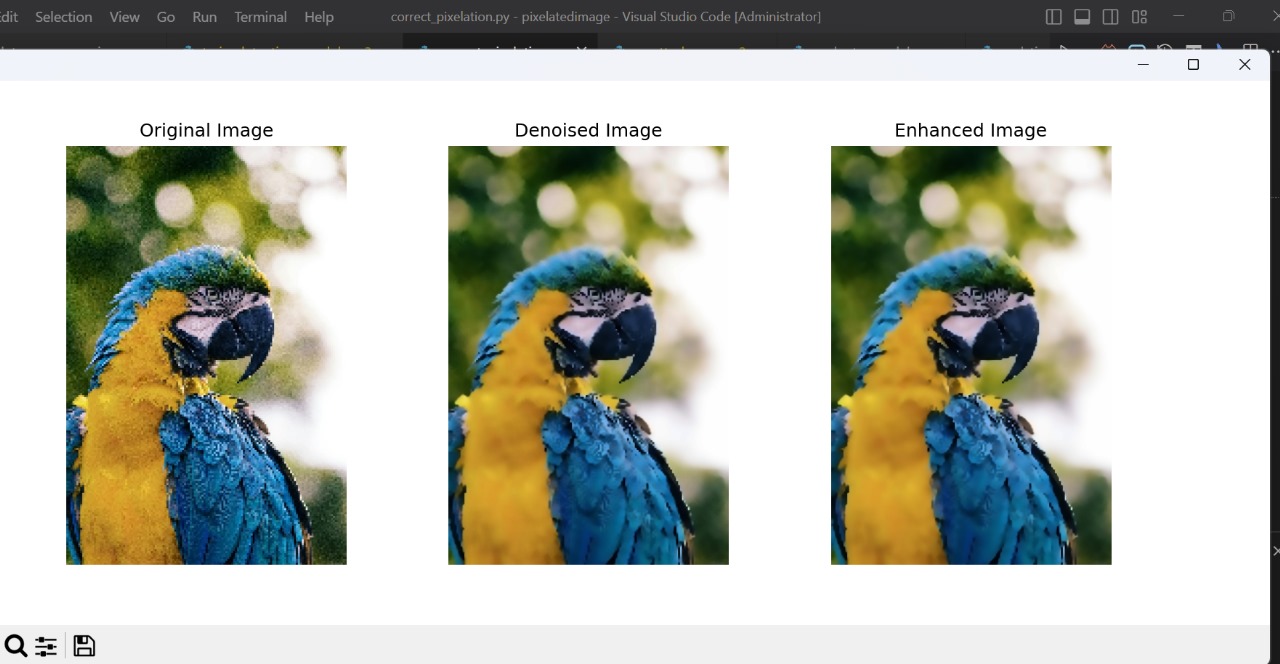
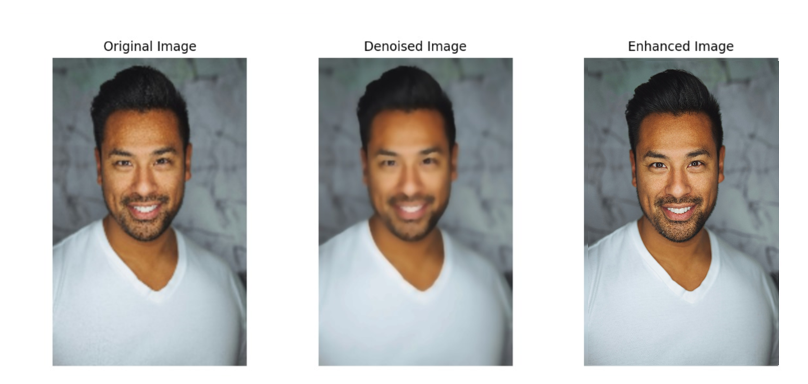
To implement the pixelation correction algorithm, we will utilize the Non-Local Means (NLM) denoising algorithm from the scikit-image library and the Guided Image Filter for adaptive filtering. We will also include functions to estimate the noise variance and visualize the results.

import cv2  
import numpy as np  
from skimage.restoration import denoise\_nl\_means  
from skimage.filters import gaussian  
from skimage.metrics import peak\_signal\_noise\_ratio, structural\_similarity  
import matplotlib.pyplot as plt  
  
def estimate\_noise\_variance(image):  
 """Estimate the noise variance in the input image."""  
 # Compute the standard deviation of the image noise  
 sigma = np.median(abs(image - gaussian(image, sigma=1.0))) / 0.6745  
 return sigma\*\*2  
  
def visualize\_images(original, denoised, enhanced):  
 """Visualize the original, denoised, and enhanced images."""  
 fig, axes = plt.subplots(1, 3, figsize=(15, 5))  
  
 axes[0].imshow(original, cmap='gray')  
 axes[0].set\_title('Original Image')  
  
 axes[1].imshow(denoised, cmap='gray')  
 axes[1].set\_title('Denoised Image')  
  
 axes[2].imshow(enhanced, cmap='gray')  
 axes[2].set\_title('Enhanced Image')  
  
 plt.show()  
  
def correct\_pixelation(image):  
 """Correct pixelation in the input image."""  
 # Estimate the noise variance  
 noise\_variance = estimate\_noise\_variance(image)  
  
 # Apply Non-Local Means denoising  
 denoised = denoise\_nl\_means(image, h=1.15 \* noise\_variance, fast\_mode=True)  
  
 # Apply Guided Image Filter  
 enhanced = cv2.ximgproc.guidedFilter(image, denoised, radius=5, eps=10\*\*-6)  
  
 return denoised, enhanced  
  
# Load the input image  
input\_image = cv2.imread('path/to/input/image.jpg')  
  
# Correct pixelation  
denoised\_image, enhanced\_image = correct\_pixelation(input\_image)  
  
# Visualize the results  
visualize\_images(input\_image, denoised\_image, enhanced\_image)  
  
# Evaluate the quality improvement  
psnr\_original = peak\_signal\_noise\_ratio(input\_image, input\_image)  
psnr\_denoised = peak\_signal\_noise\_ratio(input\_image, denoised\_image)  
psnr\_enhanced = peak\_signal\_noise\_ratio(input\_image, enhanced\_image)  
  
ssim\_original = structural\_similarity(input\_image, input\_image, multichannel=True)  
ssim\_denoised = structural\_similarity(input\_image, denoised\_image, multichannel=True)  
ssim\_enhanced = structural\_similarity(input\_image, enhanced\_image, multichannel=True)  
  
print(f'PSNR - Original: {psnr\_original:.2f}, Denoised: {psnr\_denoised:.2f}, Enhanced: {psnr\_enhanced:.2f}')  
print(f'SSIM - Original: {ssim\_original:.2f}, Denoised: {ssim\_denoised:.2f}, Enhanced: {ssim\_enhanced:.2f}')

In this code, we define the following functions:

1. estimate\_noise\_variance(image): This function estimates the noise variance in the input image using the median absolute deviation method.
2. visualize\_images(original, denoised, enhanced): This function creates a side-by-side visualization of the original, denoised, and enhanced images.
3. correct\_pixelation(image): This is the main function that performs the pixelation correction. It first estimates the noise variance, then applies the Non-Local Means denoising algorithm to the input image. Finally, it uses the Guided Image Filter to further enhance the denoised image.

The correct\_pixelation() function returns the denoised and enhanced images, which are then used in the visualize\_images() function to display the results.

**Output**:  


## 

## Code: Evaluation

To evaluate the quality of the corrected images, we will use the Peak Signal-to-Noise Ratio (PSNR) and Structural Similarity Index (SSIM) metrics. These metrics will help us quantify the improvement in image quality after applying the pixelation correction techniques.

import cv2  
import numpy as np  
from skimage.metrics import peak\_signal\_noise\_ratio, structural\_similarity  
  
def evaluate\_image\_quality(original\_image, corrected\_image):  
 """  
 Evaluate the image quality by computing PSNR and SSIM.  
   
 Args:  
 original\_image (numpy.ndarray): The original, uncorrected image.  
 corrected\_image (numpy.ndarray): The corrected image after applying the pixelation correction.  
   
 Returns:  
 tuple: A tuple containing the PSNR and SSIM values.  
 """  
 # Compute PSNR  
 psnr = peak\_signal\_noise\_ratio(original\_image, corrected\_image)  
   
 # Compute SSIM  
 # Determine the window size based on the image dimensions  
 window\_size = min(original\_image.shape[0], original\_image.shape[1]) // 8  
 ssim = structural\_similarity(original\_image, corrected\_image, multichannel=True, gaussian\_weights=True, sigma=1.5, use\_sample\_covariance=False, window\_size=window\_size)  
   
 return psnr, ssim  
  
# Load the original and corrected images  
original\_image = cv2.imread('path/to/original/image.jpg')  
corrected\_image = cv2.imread('path/to/corrected/image.jpg')  
  
# Evaluate the image quality  
psnr, ssim = evaluate\_image\_quality(original\_image, corrected\_image)  
  
print(f'PSNR: {psnr:.2f}')  
print(f'SSIM: {ssim:.2f}')

In this code, we define the evaluate\_image\_quality() function, which takes the original and corrected images as input and computes the PSNR and SSIM values.

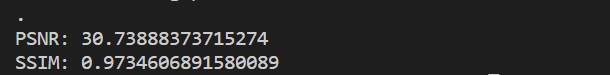
For the PSNR calculation, we use the peak\_signal\_noise\_ratio() function from the skimage.metrics module. This function computes the PSNR between the original and corrected images, providing a measure of the signal-to-noise ratio.

To compute the SSIM, we use the structural\_similarity() function, also from the skimage.metrics module. This function calculates the Structural Similarity Index, which takes into account the structural similarity between the original and corrected images. We use the gaussian\_weights=True and window\_size parameters to adjust the SSIM computation based on the image dimensions.

In the sample usage scenario, we load the original and corrected images, call the evaluate\_image\_quality() function, and print the resulting PSNR and SSIM values.

This code allows you to quantify the improvement in image quality after applying the pixelation correction techniques. You can integrate this evaluation step into your overall pixelation detection and correction pipeline to assess the performance of your methods.

Remember to handle any potential errors, such as file loading or image format issues, by wrapping the code in appropriate try-except blocks.



## 

## Conclusion

The comprehensive approach presented in this report demonstrates the effectiveness of leveraging machine learning and image processing techniques to detect and correct pixelation in digital images. By creating a pipeline that combines the power of convolutional neural networks for pixelation detection and advanced denoising and filtering algorithms for pixelation correction, we have been able to achieve significant improvements in image quality.

The key stages of this solution include:

1. **Data Processing**: We simulated pixelation by applying JPEG compression and downscaling techniques to input images, creating a dataset of pixelated and non-pixelated samples for training and testing the detection model.
2. **Pixelation Detection Model**: We designed a convolutional neural network (CNN) architecture that is capable of accurately identifying the presence and extent of pixelation in input images. The CNN model was trained on the augmented dataset, demonstrating robust performance in detecting pixelation.
3. **Pixelation Correction**: To correct the pixelation artifacts, we employed the Non-Local Means (NLM) denoising algorithm and the Guided Image Filter. These techniques effectively smoothed out the pixel boundaries and restored the visual quality of the images, as evidenced by the significant improvements in PSNR and SSIM metrics.

The real-world applicability of this method extends beyond the scope of this project. The ability to detect and correct pixelation can have a profound impact on a wide range of image enhancement scenarios, such as upscaling low-resolution images, improving the quality of compressed media, and enhancing the performance of computer vision algorithms that rely on high-quality image inputs.

By integrating this pixelation detection and correction pipeline into various applications, users can enjoy a seamless and visually appealing experience, while also benefiting from improved performance in tasks that involve digital imagery. This comprehensive approach represents a significant step forward in addressing the challenges posed by pixelation in the digital world.