**DETECT PIXELATED IMAGE & CORRECT IT**

A PROJECT REPORT

*Submitted by*

*Team:* **INTELLITECHS**

*Team Members:* **SATVIK SHARMA & ROHAN KUPPILI**

**CHAPTER 1**

**INTRODUCTION TO THE PROBLEM STATEMENT**

In today's digital world, the quality of images is paramount across various fields such as social media, digital marketing, surveillance, and personal photography. However, a common issue encountered is the pixelation of images, which leads to a noticeable degradation in visual quality. Pixelation occurs when images lose their fine details and appear blocky or blurry, often due to compression, resizing, or poor network conditions. This problem can severely impact the user experience and the effectiveness of visual communication. Therefore, this project, titled "Detect Pixelated Image and Correct It," focuses on developing an AI model capable of identifying and rectifying pixelated images, restoring them to their original quality.

A pixelated image is characterized by visible blocks or pixels, which occur due to a significant loss of resolution. This loss typically results from improper resizing, excessive compression, or intentional manipulation to obscure details. When an image is pixelated, it loses the smooth transitions and fine details that are crucial for clarity and visual appeal. The degradation is often most noticeable in areas with fine details or sharp edges, where the loss of information is more apparent.

Several real-life scenarios contribute to the occurrence of pixelated images. One common situation is low bandwidth streaming, where high-quality videos are streamed over poor or limited internet connections. To accommodate the available bandwidth, streaming services often downgrade the image quality, leading to pixelation. Another scenario is the zooming into low-resolution images, where enlarging a small image reveals the individual pixels, resulting in a blocky appearance. Additionally, excessive compression of images to reduce file size for storage or transmission can lead to pixelation, as important visual information is lost in the process.

Correcting pixelated images is crucial for maintaining the integrity and quality of visual content. In fields like digital marketing, clear and high-quality images are essential for attracting and engaging customers. In surveillance, pixelated images can hinder the ability to identify important details, potentially compromising security. For personal photography, pixelation can ruin cherished memories by degrading the visual quality of photos. By developing an AI model that can detect and correct pixelated images, this project aims to enhance the overall quality of digital images, ensuring they meet the high standards required in various applications.

**CHAPTER 2**

**ABOUT OUR SOLUTION**

The project, "Detect Pixelated Image and Correct It," addresses the prevalent issue of image pixelation by developing an advanced AI model capable of detecting pixelated images and correcting them. The solution leverages deep learning techniques, specifically utilizing a modified VGG16 architecture, to restore the quality of pixelated images. This approach ensures the preservation of fine details and enhances the overall visual appeal of the images.

**Key Components and Methodology:**

* Data Preparation

The initial step in our project involves preparing the dataset. We start by creating a dataset of pixelated images through a pixelation process applied to a set of high-resolution images. The pixelation is achieved by downscaling the images and then upscaling them back to their original size, simulating common real-life scenarios where pixelation occurs.

* Image Preprocessing

Preprocessing is crucial for ensuring that the input images are standardized and suitable for model training. This involves resizing images to a consistent resolution, converting colour spaces, and normalizing pixel values. Our preprocessing pipeline ensures that all images meet the minimum resolution requirements and are properly formatted for the neural network.

* Custom VGG16 Model

The core of our solution is a custom VGG16-based model. We use a pre-trained VGG16 model, leveraging its robust feature extraction capabilities while making necessary modifications to suit our specific task. The model is truncated to reduce complexity and enhance performance, followed by upsampling layers to restore the resolution of pixelated images. Additionally, a Gaussian blur layer is incorporated to smoothen the output, further improving the visual quality.

* Training and Optimization

The model is trained using a large dataset of pixelated and original images. We employ mean squared error (MSE) as the loss function, optimizing the model's performance in restoring the pixelated images to their original form. Gradient accumulation is used to effectively manage memory during training, allowing for larger batch sizes and improved training stability. Quantization techniques are applied to the model to enhance its efficiency, making it suitable for deployment on resource-constrained devices.

* Evaluation Metrics

To evaluate the effectiveness of our solution, we use several performance metrics including accuracy, precision, recall, and F1 score. These metrics provide a comprehensive assessment of the model's ability to detect and correct pixelated images, ensuring high-quality results.

**Benefits and Unique Features:**

* Quality Restoration

Our solution significantly improves the quality of pixelated images, restoring fine details and enhancing visual clarity. The use of a modified VGG16 model ensures that the restored images retain the essential features of the original images, making them almost indistinguishable from high-resolution counterparts.

* Efficiency and Scalability

The application of quantization techniques and model pruning optimizes the model for efficient deployment. This allows the solution to run on various devices, including those with limited computational resources, without compromising performance.

* Versatility

The solution is designed to handle a wide range of image types and resolutions. It can be applied to different domains such as digital marketing, social media, surveillance, and personal photography, making it a versatile tool for improving image quality across various applications.

* Robust Preprocessing Pipeline

Our comprehensive preprocessing pipeline ensures that input images are consistently prepared, which is crucial for the model's performance. This pipeline handles variations in image resolution and format, making the solution robust and reliable.

* Usefulness and Applications

The ability to detect and correct pixelated images has numerous practical applications. In digital marketing, high-quality images are essential for engaging customers and conveying messages effectively. Our solution ensures that marketing materials maintain their visual appeal, even when sourced from lower-quality images.

In surveillance, the clarity of images is critical for identifying details and ensuring security. By restoring pixelated images, our solution enhances the effectiveness of surveillance systems, allowing for better identification and analysis.

For personal photography, our solution preserves the quality of cherished memories, ensuring that photos remain clear and detailed. This is particularly useful for old or compressed images that may have suffered from pixelation over time.

Overall, our project presents a comprehensive and efficient solution to the problem of image pixelation, offering significant benefits and practical applications across various fields.

**CHAPTER 3**

**PROJECT DATA**

**Dataset for images:**  
<https://www.kaggle.com/datasets/evgeniumakov/images4k>

**Code:**import os

import cv2

import torch

import torch.nn as nn

import torch.optim as optim

import numpy as np

from torch.utils.data import DataLoader, Dataset

from torchvision import transforms, models

from sklearn.metrics import accuracy\_score, precision\_score, recall\_score, f1\_score

from torch.quantization import quantize\_dynamic, get\_default\_qconfig

import torch.nn.utils.prune as prune

# Constants and parameters

data\_dir = 'D:\\Code'

pixelated\_dir = 'D:\\Code\\pixelated'

output\_dir = 'D:\\Code\\output'

img\_extensions = ['jpeg', 'jpg', 'bmp', 'png']

min\_resolution\_width = 1920

min\_resolution\_height = 1080

min\_width, min\_height = 256, 256

batch\_size = 4

accumulation\_steps = 16

# Function to pixelate images

def pixelate\_images(folder\_path, scale\_factor, output\_folder):

os.makedirs(output\_folder, exist\_ok=True)

pixelated\_images = []

for filename in os.listdir(folder\_path):

image\_path = os.path.join(folder\_path, filename)

if not filename.lower().endswith((".jpg", ".jpeg", ".png")):

continue # Skip non-image files

img = cv2.imread(image\_path)

if img is None:

print(f"Failed to read image: {image\_path}, skipping pixelation.")

continue

try:

# Downscale the image

small = cv2.resize(img, None, fx=scale\_factor, fy=scale\_factor, interpolation=cv2.INTER\_LINEAR)

# Upscale the image back to the original size

pixelated = cv2.resize(small, (img.shape[1], img.shape[0]), interpolation=cv2.INTER\_NEAREST)

pixelated\_images.append(pixelated)

# Construct the output file path

output\_path = os.path.join(output\_folder, f"pixelated\_{filename}")

# Save the pixelated image

if cv2.imwrite(output\_path, pixelated):

print(f"Pixelated image saved to {output\_path}")

else:

print(f"Failed to save image: {output\_path}")

except Exception as e:

print(f"Error processing image: {image\_path} - {e}")

return pixelated\_images

# Paths and parameters for pixelation

pr\_train\_folder = 'D:\\Code\\pr\_train'

scale\_factor = 0.4

pixelated\_output\_folder = 'D:\\Code\\pixelated'

# Pixelate images in pr\_train folder

pixelate\_images(pr\_train\_folder, scale\_factor, pixelated\_output\_folder)

# Function to preprocess image

def preprocess\_image(image\_path):

try:

image = cv2.imread(image\_path)

if image is None:

print(f"Failed to read image: {image\_path}")

return None

height, width, \_ = image.shape

if width < min\_resolution\_width or height < min\_resolution\_height:

print(f"Skipping image {image\_path}: resolution {width}x{height} is less than {min\_resolution\_width}x{min\_resolution\_height}")

return None

# Resize original image

image = cv2.resize(image, (min\_width, min\_height), interpolation=cv2.INTER\_LINEAR)

image = cv2.cvtColor(image, cv2.COLOR\_BGR2RGB)

image = image / 255.0

return image.astype(np.float32)

except Exception as e:

print(f"Error processing image {image\_path}: {e}")

return None

# Custom dataset class

class ImageDataset(Dataset):

def \_init\_(self, folder\_path, transform=None):

self.image\_files = [os.path.join(folder\_path, f) for f in os.listdir(folder\_path) if os.path.isfile(os.path.join(folder\_path, f))]

self.transform = transform

def \_len\_(self):

return len(self.image\_files)

def \_getitem\_(self, idx):

image\_path = self.image\_files[idx]

image = preprocess\_image(image\_path)

if image is not None:

if self.transform:

image = self.transform(image)

return image

return None

def collate\_fn(batch):

batch = list(filter(lambda x: x is not None, batch))

if len(batch) == 0:

print("Warning: Batch is empty after filtering None values.")

return None # Handle this case appropriately

try:

return torch.stack(batch)

except Exception as e:

print(f"Error stacking tensors: {e}")

return None # Handle this case appropriately

# Function to build a custom model with VGG16

class CustomVGG16Model(nn.Module):

def \_init\_(self):

super(CustomVGG16Model, self).\_init\_()

vgg16 = models.vgg16(pretrained=True)

self.features = vgg16.features[:23] # Use fewer layers from VGG16

self.upsample = nn.Upsample(scale\_factor=8, mode='bilinear', align\_corners=False)

self.gaussian\_blur = GaussianBlur(kernel\_size=5, sigma=1.0)

self.decoder = nn.Sequential(

nn.Conv2d(512, 128, kernel\_size=3, stride=1, padding=1),

nn.ReLU(inplace=True),

nn.Conv2d(128, 64, kernel\_size=3, stride=1, padding=1),

nn.ReLU(inplace=True),

nn.Conv2d(64, 3, kernel\_size=3, stride=1, padding=1),

nn.Sigmoid()

)

def forward(self, x):

x = self.features(x)

x = self.upsample(x)

x = self.gaussian\_blur(x)

x = self.decoder(x)

return x

# Gaussian blur module

class GaussianBlur(nn.Module):

def \_init\_(self, kernel\_size, sigma):

super(GaussianBlur, self).\_init\_()

self.kernel\_size = kernel\_size

self.sigma = sigma

# Create a Gaussian kernel

x = torch.arange(kernel\_size).float()

x -= (kernel\_size - 1) / 2

gauss = torch.exp(-(x \*\* 2) / (2 \* sigma \*\* 2))

gauss = gauss / gauss.sum()

self.gaussian\_kernel = gauss[:, None] \* gauss[None, :]

def forward(self, x):

# Apply Gaussian blur

channels = x.shape[1]

kernel = self.gaussian\_kernel.expand(channels, 1, -1, -1).to(x.device)

padding = self.kernel\_size // 2

x = nn.functional.conv2d(x, kernel, padding=padding, groups=channels)

return x

# Main function for training and evaluation

def main():

# Paths to training and validation image folders

train\_folder\_path = pixelated\_dir

val\_folder\_path = pixelated\_dir

# Define transformations

transform = transforms.Compose([

transforms.ToTensor(),

])

# Load datasets

train\_dataset = ImageDataset(train\_folder\_path, transform=transform)

val\_dataset = ImageDataset(val\_folder\_path, transform=transform)

train\_loader = DataLoader(train\_dataset, batch\_size=batch\_size, shuffle=True, num\_workers=0, collate\_fn=collate\_fn)

val\_loader = DataLoader(val\_dataset, batch\_size=batch\_size, shuffle=False, num\_workers=0, collate\_fn=collate\_fn)

# Build and compile the model

device = torch.device('cuda' if torch.cuda.is\_available() else 'cpu')

model = CustomVGG16Model().to(device)

model.qconfig = get\_default\_qconfig('fbgemm')

torch.quantization.prepare(model, inplace=True)

criterion = nn.MSELoss()

optimizer = optim.Adam(model.parameters(), lr=1e-4)

# Training loop with gradient accumulation

num\_epochs = 10

for epoch in range(num\_epochs):

model.train()

running\_loss = 0.0

optimizer.zero\_grad()

for i, images in enumerate(train\_loader):

if images is None:

continue # Skip None batches

images = images.to(device)

with torch.cuda.amp.autocast():

outputs = model(images)

if outputs.size() != images.size():

outputs = nn.functional.interpolate(outputs, size=images.size()[2:], mode='bilinear', align\_corners=False)

loss = criterion(outputs, images)

loss.backward()

if (i + 1) % accumulation\_steps == 0:

optimizer.step()

optimizer.zero\_grad()

running\_loss += loss.item() \* images.size(0)

epoch\_loss = running\_loss / len(train\_loader.dataset)

print(f'Epoch {epoch+1}/{num\_epochs}, Loss: {epoch\_loss:.4f}')

# Apply quantization

torch.quantization.convert(model, inplace=True)

# Save the model

torch.save(model.state\_dict(), os.path.join(output\_dir, 'satvik\_rohan.pth'))

print("Model training and saving completed successfully.")

# Evaluation

model.eval()

all\_preds = []

all\_labels = []

with torch.no\_grad():

for images in val\_loader:

if images is None:

continue # Skip None batches

images = images.to(device)

with torch.cuda.amp.autocast():

outputs = model(images)

outputs = nn.functional.interpolate(outputs, size=images.size()[2:], mode='bilinear', align\_corners=False)

all\_preds.append(outputs.cpu().numpy())

all\_labels.append(images.cpu().numpy())

# Convert lists to numpy arrays and flatten them

all\_preds = np.concatenate(all\_preds).flatten()

all\_labels = np.concatenate(all\_labels).flatten()

accuracy = accuracy\_score(all\_labels, (all\_preds > 0.5).astype(int))

precision = precision\_score(all\_labels, (all\_preds > 0.5).astype(int), average='binary')

recall = recall\_score(all\_labels, (all\_preds > 0.5).astype(int), average='binary')

f1 = f1\_score(all\_labels, (all\_preds > 0.5).astype(int), average='binary')

print(f'Validation Accuracy: {accuracy:.4f}')

print(f'Validation Precision: {precision:.4f}')

print(f'Validation Recall: {recall:.4f}')

print(f'Validation F1 Score: {f1:.4f}')

# Ensure the script runs when executed directly

if \_name\_ == '\_main\_':

main()

**CHAPTER 4**

**FEATURES AND ADVANTAGES**

**Technical Points:**

- High-Quality Restoration: Utilizes a modified VGG16 architecture to restore pixelated images to near-original quality.

- Efficient Processing: Incorporates quantization and pruning techniques for optimized performance on resource-constrained devices.

- Comprehensive Preprocessing Pipeline: Ensures standardized input images through resizing, color space conversion, and normalization.

- Upsampling and Gaussian Blur Integration: Enhances visual quality by smoothening the output images.

- Scalable Solution: Can handle various image types and resolutions, making it adaptable to different applications.

- Robust Evaluation Metrics: Employs accuracy, precision, recall, and F1 score for comprehensive performance assessment.

- Gradient Accumulation: Manages memory effectively during training, allowing for larger batch sizes and improved stability.

- Dynamic Quantization: Enhances model efficiency without compromising the quality of the output images.

**General Points:**

- Versatile Applications: Suitable for digital marketing, social media, surveillance, and personal photography.

- User-Friendly: Simplifies the process of improving image quality for non-technical users.

- Enhanced User Experience: Improves the visual appeal of images, leading to better engagement and satisfaction.

- Preserves Memories: Restores old or compressed photos to maintain their clarity and detail.

- Security Enhancement: Improves the effectiveness of surveillance systems by providing clearer images for analysis.

- Market Competitiveness: Ensures high-quality marketing materials that stand out in a crowded marketplace.

- Broad Accessibility: Can be deployed on various devices, ensuring wide accessibility and usability.

- Improved Communication: Enhances the clarity of visual content, facilitating more effective communication.

**CHAPTER 5**

**TECHNOLOGIES USED**

1. Python: Python is the primary programming language used for its simplicity, versatility, and strong ecosystem of libraries, making it ideal for rapid development in AI and machine learning projects.

2. OpenCV: OpenCV (Open Source Computer Vision Library) is a powerful library designed for computer vision tasks. It provides functions to read, write, and manipulate images (`cv2.imread`, `cv2.resize`, etc.), essential for image processing tasks in your project.

3. PyTorch: PyTorch is a popular deep learning framework known for its flexibility and ease of use. It provides tools for building and training neural networks (`torch.nn`, `torch.optim`), handling tensors, and GPU acceleration, which are crucial for implementing deep learning models like the one in your project.

4. torchvision: torchvision is a package in PyTorch that provides access to popular datasets, model architectures, and image transformations. In your project, `torchvision.transforms` is used to preprocess images (`ToTensor`), and `torchvision.models` is used to access pretrained models like VGG16.

5. scikit-learn: scikit-learn is a machine learning library that provides simple and efficient tools for data analysis and modeling. In your project, `sklearn.metrics` is used to compute evaluation metrics such as accuracy, precision, recall, and F1 score, which are crucial for assessing the performance of your image correction model.

6. NumPy: NumPy is a fundamental package for numerical computing in Python. It provides support for large, multi-dimensional arrays and matrices, along with a collection of mathematical functions to operate on these arrays. In your project, NumPy (`numpy`) is used for array operations and data handling tasks.

These technologies collectively empower our project to handle image data, implement deep learning models, evaluate model performance, and ensure efficient computation throughout the development and deployment phases.

**CHAPTER 6**

**CONCLUSION AND FUTURE ENHANCEMENTS**

The project "Detect Pixelated image and Correct It" leverages a combination of advanced technologies to address the challenge of identifying and correcting pixelated images. Python served as the foundational programming language, enabling seamless integration of libraries like OpenCV for image processing and PyTorch for deep learning model development. Through the implementation of a custom VGG16-based neural network, the project achieves automated image correction by pixelating and subsequently refining images to enhance clarity.

**Future Possible Enhancements:**

1. Advanced Model Architectures: Explore more sophisticated neural network architectures beyond VGG16 to potentially improve accuracy and performance in image correction tasks.

2. Dataset Augmentation: Introduce techniques such as data augmentation to diversify the training data, enhancing the model's robustness to various types of pixelation and image degradation.

3. Real-Time Processing: Implement real-time image correction capabilities for applications requiring instantaneous feedback, leveraging optimized inference pipelines and hardware acceleration.

4. Interactive User Interface: Develop a user-friendly interface to allow users to interactively upload and correct pixelated images, potentially integrating with cloud services for scalability and accessibility.

5. Transfer Learning: Experiment with transfer learning techniques by fine-tuning pretrained models on specific types of pixelated images, potentially reducing the need for extensive training data.

6. Deployment Optimization: Optimize model deployment for different platforms and environments, ensuring efficient performance across diverse hardware configurations.

These enhancements aim to expand the project's capabilities, improve user experience, and extend its applicability to broader use cases, ultimately advancing the state-of-the-art in automated image enhancement and correction.

**TEAM INTELLITECHS**

The team comprises of

**Satvik Sharma**, who is a student of B.Tech in CSE with specialisation in Computer Networking at SRM Institute of Science and Technology, Kattankulathur, Chennai, Tamil Nadu- 603203, India

and

**Rohan Kuppili**, who is a student of B.Tech in CSE with specialisation in Cloud Computing at SRM Institute of Science and Technology, Kattankulathur, Chennai, Tamil Nadu- 603203, India.

Satvik Sharma: [ss3930@srmist.edu.in](mailto:ss3930@srmist.edu.in)

Rohan Kuppili: [rk6257@srmist.edu.in](mailto:rk6257@srmist.edu.in)