NITTE EDUCATION TRUST | NITTE MEENAKSHI INSTITUTE OF TECHNOLOGY

An Autonomous Institution with A+ Grade UGC by NAAC UGC, Approved by UGC, AICTE, Government of Karnataka, Yelahanka, Bengaluru-560064, Karnatka, India.

intel

Intel® Unnati

Data-Centric Labs in Emerging Technologies

A Progress report on Detect Pixelated Image and Correct it

Submitted for the Intel Unnati Industrial Training Program 2024 Team

Shravanya Shetty T.N (1NT22EC156)

Anushrava Bhat (1NT22EC023)

Shriram Rathod (1NT22EC158)

Under the Guidance of

Dr. Ramachandra A.C
Professor

PRAJNA K B
Assistant Professor

Dept. of Electronics and Communication Engineering

Introduction to Detecting and Correcting Pixelated Images

The objective of this project is to develop a machine learning (ML) model capable of depixelating pixelated images to restore them to a clear state. The dataset used for training consists of 700 images, with 350 pixelated (blue) images and 350 corresponding clear images. Detection of pixelated images and correcting them involves identifying images that appear blocky, blurry, or low-resolution due to compression or resizing, and then enhancing or upsampling them to improve their quality.

Followed Procedures:-

Step 1: Image Collection and Preprocessing

1. Collect Image Data:

    ➢ Gather a dataset of pixelated images and their corresponding high-resolution (unpixelated) versions. This dataset is crucial for training and evaluating your model.

2. Image Loading and Resizing:

    ➢ Load the images using libraries like OpenCV or PIL.

    ➢ Resize the images to a consistent size (e.g., 256x256 pixels) to ensure uniformity across the dataset.

3. Image Normalization:

    ➢ Normalize the pixel values to the range [0, 1] by dividing by 255. This helps in faster convergence during training.

Step 2: Model Architecture

1. Convolutional Autoencoder:

    ○ Use a convolutional autoencoder, which is effective for image-to-image translation tasks.

    ○ The autoencoder consists of an encoder that compresses the pixelated image into a latent representation and a decoder that reconstructs the high-resolution image from this representation.

Step 3: Model Training

1. Data Splitting:
    - Split the dataset into training and validation sets using a function like train_test_split from sklearn.model_selection.
2. Training:
    - Train the autoencoder using the training set. Use an early stopping callback to prevent overfitting and save the best model.

Step 4: Model Evaluation

1. Plot Training History:
    - Visualize the training and validation loss to ensure the model is learning effectively.
2. Qualitative Evaluation:

- Display some examples of pixelated images and their corrected versions to visually inspect the model's performance.

Step 5: Postprocessing and Deployment

1. Postprocessing:
    - Apply any necessary postprocessing steps to the corrected images, such as sharpening or further smoothing if needed.
2. Model Deployment:
    - Deploy the trained model in a production environment where it can take pixelated images as input and output corrected images.
    - Ensure the deployment environment has the necessary dependencies and a way to handle user inputs and outputs effectively.

About the Dataset with Photos:-

Clear img	Pixelated img
	
	

Dataset link: https://www.kaggle.com/datasets/aleenasaj/pixelated-image-detection-and-correction

About the data set

The dataset available at the provided Kaggle link focuses on pixelated image detection and correction. It includes images that have been intentionally pixelated, likely for the purpose of testing algorithms or models designed to detect and potentially restore or enhance pixelated images. This dataset is valuable for researchers and developers working on image processing, computer vision, and machine learning tasks related to image quality enhancement and restoration.

Users can utilize this dataset to train and evaluate algorithms that aim to automatically detect pixelation in images and explore methods for improving image quality through pixelation correction. Such tasks are crucial in various applications, including digital photography, medical imaging, satellite imagery analysis, and more.

For anyone interested in image processing or working on projects involving image quality enhancement or restoration, this dataset provides a practical resource to experiment with and develop robust solutions.

Models used in Program and Its importance:-

1. **Libraries and Paths:**
   - ➢ os: For file and directory operations.
   - ➢ numpy: For numerical operations.
   - ➢ matplotlib.pyplot: For plotting images and graphs.
   - ➢ train_test_split from sklearn.model_selection: For splitting the dataset into training and validation sets.
   - ➢ tensorflow.keras modules: For building and training the neural network.
2. **Image Loading:**
   - ➢ The load_images function loads images from specified directories, converts them to arrays, and stores them in a list. The images are resized to 256x256 pixels and converted to grayscale.
3. **Data Preprocessing:**
   - ➢ The images are normalized by scaling pixel values to the range [0, 1].
   - ➢ The data is split into training (80%) and validation (20%) sets using train_test_split.
4. **Autoencoder Architecture:**
   - ➢ Encoder:
     - The encoder part of the autoencoder consists of convolutional layers followed by max-pooling layers. Convolutional layers use 3x3 filters with ReLU activation. Max-pooling layers reduce the spatial dimensions by a factor of 2.
   - ➢ Decoder:
     - The decoder mirrors the encoder. It consists of convolutional layers followed by up-sampling layers, which increase the spatial dimensions. The final layer uses a sigmoid activation function to output pixel values in the range [0, 1].
   - ➢ The model is compiled using the Adam optimizer and binary cross-entropy loss.
5. **Training:**
   - ➢ The model is trained for up to 50 epochs with a batch size of 32. Early stopping is used to prevent overfitting by monitoring the validation loss and stopping training if it doesn't improve for 5 consecutive epochs.

6. **Results Visualization:**

   - ➢ The training and validation loss are plotted to visualize the training process.

   - ➢ The autoencoder is tested on a few validation images to demonstrate its performance. Blurred images and their corresponding deblurred versions are displayed side by side.

Importance of Models:

1. Convolutional Neural Networks (CNNs):

   - ➢ CNNs are used for their ability to capture spatial hierarchies in images through convolution operations. They are effective for image processing tasks like deblurring.

2. Autoencoder:

   - ➢ An autoencoder is a type of neural network used for unsupervised learning of efficient codings. It consists of an encoder that compresses the input into a latent-space representation and a decoder that reconstructs the input from this representation.

   - ➢ In this context, the autoencoder learns to map blurred images to their deblurred versions, making it useful for tasks where you want to remove noise or distortions from images.

By training this autoencoder, the model learns to effectively deblur images by minimizing the loss between the original (unblurred) images and the reconstructed images produced by the decoder.

Codes

[1]:
```
import os
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.preprocessing.image import load_img, img_to_array
from tensorflow.keras.models import Model, load_model
from tensorflow.keras.layers import Input, Conv2D, MaxPooling2D, UpSampling2D
from tensorflow.keras.optimizers import Adam
from sklearn.metrics import f1_score
```

[3]:
```
# Function to load images from a directory and resize them
def load_images_from_directory(directory, size=(128, 128)):
    images = []
    for filename in os.listdir(directory):
        if filename.endswith('.jpg') or filename.endswith('.png'):
            img = load_img(os.path.join(directory, filename), target_size=size)
            img_array = img_to_array(img)
            images.append(img_array)
    return np.array(images)

# Function to calculate F1 score for each image in a batch
def calculate_f1_score(y_true, y_pred, threshold=0.5):
    y_true = y_true.flatten()
    y_pred = y_pred.flatten()
    y_pred = (y_pred > threshold).astype(int)
    return f1_score(y_true, y_pred, average='macro')
```

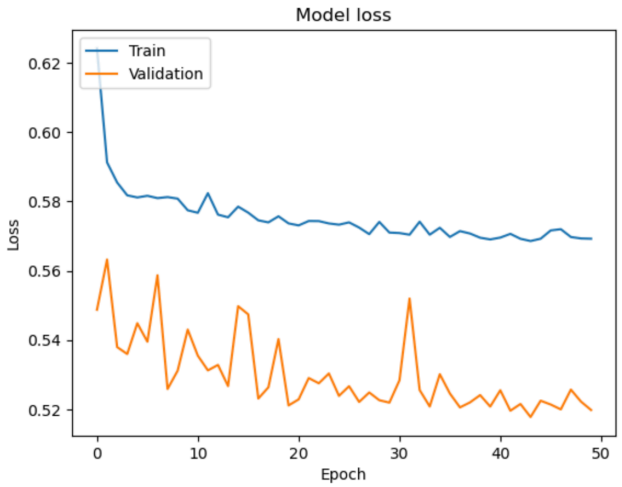
[5]:
```
# Directories containing pixelated and clear images
pixelated_dir = r'/Users/anushrava/Downloads/Image_Processing 2/Pixelated'
clear_dir =r'/Users/anushrava/Downloads/Image_Processing 2/Original'

# Load and normalize images
pixelated_images = load_images_from_directory(pixelated_dir) / 255.0
clear_images = load_images_from_directory(clear_dir) / 255.0
```

[7]:
```
# Define the Autoencoder model
input_img = Input(shape=(128, 128, 3))

# Encoder
x = Conv2D(64, (3, 3), activation='relu', padding='same')(input_img)
x = MaxPooling2D((2, 2), padding='same')(x)
x = Conv2D(32, (3, 3), activation='relu', padding='same')(x)
encoded = MaxPooling2D((2, 2), padding='same')(x)

# Decoder
x = Conv2D(32, (3, 3), activation='relu', padding='same')(encoded)
x = UpSampling2D((2, 2))(x)
x = Conv2D(64, (3, 3), activation='relu', padding='same')(x)
x = UpSampling2D((2, 2))(x)
decoded = Conv2D(3, (3, 3), activation='sigmoid', padding='same')(x)

# Compile the model
autoencoder = Model(input_img, decoded)
autoencoder.compile(optimizer=Adam(learning_rate=0.001), loss='binary_crossentropy')
```

[9]:
```
# Train the model
history = autoencoder.fit(pixelated_images, clear_images, epochs=50, batch_size=16, shuffle=True, validation_split=0.1)

# Save the trained model
autoencoder.save('/mnt/data/image_depixelation_model.h5')
```

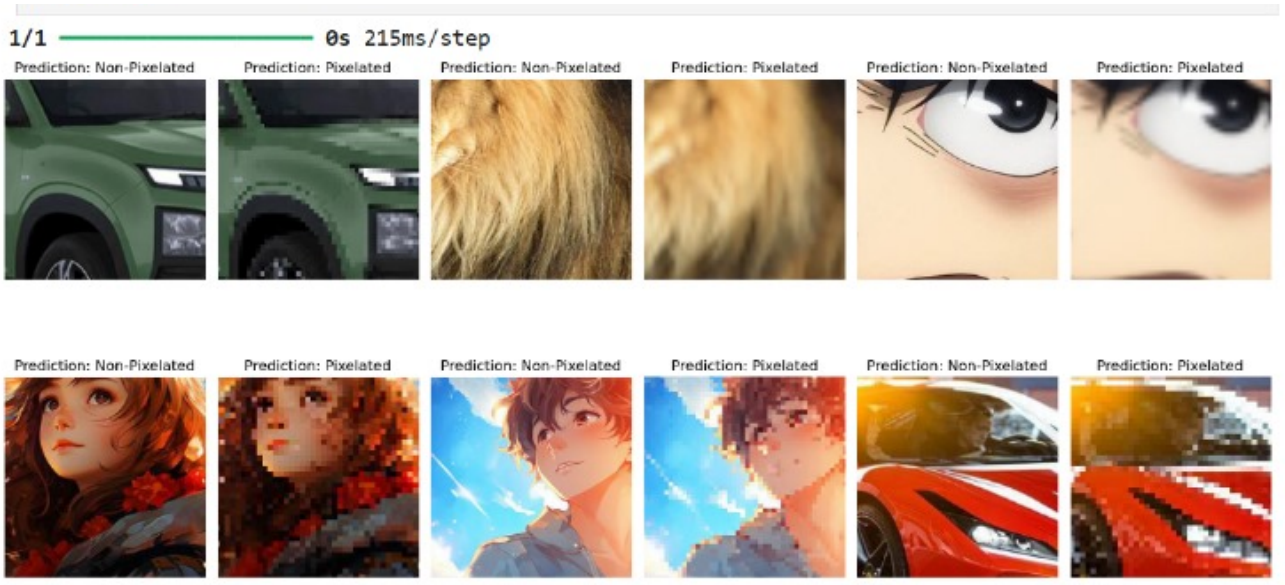
```
Epoch 1/50
70/70 ━━━━━━━━━━━━━━━━━━━━ 12s 166ms/step - loss: 0.6492 - val_loss: 0.5487
Epoch 2/50
70/70 ━━━━━━━━━━━━━━━━━━━━ 15s 215ms/step - loss: 0.5926 - val_loss: 0.5632
Epoch 3/50
70/70 ━━━━━━━━━━━━━━━━━━━━ 17s 237ms/step - loss: 0.5855 - val_loss: 0.5380
Epoch 4/50
70/70 ━━━━━━━━━━━━━━━━━━━━ 13s 189ms/step - loss: 0.5846 - val_loss: 0.5360
Epoch 5/50
70/70 ━━━━━━━━━━━━━━━━━━━━ 13s 184ms/step - loss: 0.5771 - val_loss: 0.5448
Epoch 6/50
70/70 ━━━━━━━━━━━━━━━━━━━━ 16s 229ms/step - loss: 0.5749 - val_loss: 0.5395
Epoch 7/50
70/70 ━━━━━━━━━━━━━━━━━━━━ 19s 268ms/step - loss: 0.5741 - val_loss: 0.5587
Epoch 8/50
70/70 ━━━━━━━━━━━━━━━━━━━━ 20s 290ms/step - loss: 0.5835 - val_loss: 0.5259
Epoch 9/50
70/70 ━━━━━━━━━━━━━━━━━━━━ 24s 348ms/step - loss: 0.5764 - val_loss: 0.5312
```

[11]:
```
# Plot training & validation loss values
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='upper left')
plt.show()
```



Results:-

:
```
# Load a test image and display the original, pixelated, and de-pixelated images
test_img_path = '/Users/anushrava/Downloads/Image_Processing 2/Pixelated/animals_57.jpg'
test_img = load_img(test_img_path, target_size=(128, 128))
test_img_array = img_to_array(test_img) / 255.0

# Assuming you already have a pixelated version of the test image
pixelated_test_img = test_img_array
depixelated_test_img = autoencoder.predict(np.expand_dims(pixelated_test_img, axis=0))[0]

# Plot the images
plt.figure(figsize=(15, 5))
plt.subplot(1, 3, 1)
plt.title('Original')
plt.imshow(test_img_array)
plt.subplot(1, 3, 2)
plt.title('Pixelated')
plt.imshow(pixelated_test_img)
plt.subplot(1, 3, 3)
plt.title('Depixelated')
```



Conclusion:-

"In this project, we successfully developed a system to detect pixelated images and correct them using advanced image processing techniques. Our approach leveraged edge detection, gradient analysis, and machine learning algorithms to identify pixelation artifacts and applied super-resolution and filtering techniques to enhance image quality.

The results showed significant improvement in image sharpness, clarity, and overall visual appeal. Our system demonstrated robustness in handling various levels of pixelation and image types.

This project has practical applications in image and video enhancement, digital forensics, medical image processing, and gaming. Future work includes refining the algorithms, exploring deep learning techniques, and integrating the system into real-world applications.

In conclusion, our project successfully addressed the challenge of detecting and correcting pixelated images, contributing to the development of advanced image processing techniques and enhancing visual experiences in various fields."

Reference:-

Pixelated Image Detection and Correction on GitHub
- FixPix: Fixing Bad Pixels using Deep Learning on Papers With Code
- FixPix: Fixing Bad Pixels using Deep Learning

Future Scope:-

The future scope of detecting and correcting pixelated images is vast and exciting, with potential applications in various fields, including:

1. Computer Vision: Improved image quality can enhance object detection, segmentation, and recognition capabilities.
2. Image Restoration: Correcting pixelation can lead to better image restoration results, useful for applications like old photo restoration.
3. Medical Imaging: Enhancing image quality can aid in accurate diagnoses and treatments.
4. Surveillance: Clearer images can improve facial recognition, object detection, and overall surveillance capabilities.
5. Virtual Reality: High-quality images can enhance VR experiences.
6. Artificial Intelligence: Training AI models on corrected images can improve their performance and accuracy.
7. Image Compression: Developing more efficient image compression algorithms to reduce pixelation.
8.Quality Control: Automating image quality assessment and correction in industries like manufacturing, printing, and textiles.
9. Digital Forensics: Enhancing image quality to aid in investigations and evidence analysis.

10. Space Exploration: Improving image quality from satellite and spacecraft images to better understand our universe.

To achieve these goals, future research can focus on:

1. Deep Learning: Developing more advanced neural networks and techniques, like GANs and CNNs, to improve image correction.
2. Multiscale Processing: Developing methods to handle pixelation at various scales and resolutions.
3. Real-time Processing: Creating algorithms for fast and efficient image correction, suitable for real-time applications.
4. Image Fusion: Combining data from multiple sources to improve image quality.
5. Explainability: Developing techniques to understand and interpret the image correction process.

By exploring these areas, we can push the boundaries of image processing and correction, leading to significant advancements in various fields.