**Image Resolution Enhancer Using Autoencoders: An ML Model for Enhanced Image Quality**

**By: Chirag Sharma**

**Abstract:**

The Image Resolution Enhancer ML model based on autoencoders is a cutting-edge solution for enhancing image quality. Autoencoders, a type of neural network architecture, are leveraged to learn efficient image representations and restore lost image details caused by low resolution. This report presents a comprehensive description of the model, highlighting its methodology, key components, and advantages. The proposed image resolution enhancer model demonstrates promising results, offering a viable solution for a wide range of applications in image processing and computer vision.

**Introduction:**

The field of image processing continually seeks methods to enhance image quality. The Image Resolution Enhancer ML model addresses this challenge using the powerful concept of autoencoders. By leveraging deep learning techniques, the model aims to recover fine-grained details and enhance the visual appeal of low-resolution images.

**Methodology:**

The Image Resolution Enhancer ML model employs an autoencoder-based architecture consisting of an encoder and a decoder. Autoencoders are unsupervised learning models that are trained to reconstruct the input data. In the case of image resolution enhancement, the model is trained to recover high-resolution images from their low-resolution counterparts.

**Model Components:**

**3.1 Encoder:**

The encoder component of the model takes low-resolution images as input and learns a compact representation of the image data. It consists of several convolutional layers, which extract high-level features from the input images. The extracted features are then compressed into a lower-dimensional latent space.

**3.2 Decoder:**

The decoder component aims to reconstruct high-resolution images from the learned latent representation. It consists of a series of deconvolutional (also known as transposed convolutional) layers, which upsample the encoded features. The decoder's objective is to generate output images that are visually indistinguishable from the original high-resolution images.

**Training Process:**

The model is trained using a dataset of paired low-resolution and high-resolution images. During training, the model learns to minimize the difference between the reconstructed high-resolution images and the original high-resolution images. This process is typically achieved by optimizing a loss function, such as mean squared error (MSE) or perceptual loss, which captures both pixel-level and perceptual differences.

**Advantages:**

**5.1 Enhanced Image Quality:**

The Image Resolution Enhancer ML model offers significant improvements in image quality, successfully restoring fine details and textures that are lost in low-resolution images.

**5.2 Generalization:**

The model can generalize well to unseen low-resolution images, making it applicable to a wide range of image enhancement tasks.

**5.3 Computational Efficiency:**

The model architecture and training process are designed to ensure computational efficiency, allowing for real-time or near-real-time performance on various platforms.

**Applications:**

The Image Resolution Enhancer ML model has diverse applications in multiple domains, including:

Surveillance systems: Enhancing low-resolution surveillance images for improved object recognition.

Medical imaging: Enhancing medical scans to aid in diagnosis and analysis.

Satellite imagery: Improving the quality of satellite images for enhanced analysis and interpretation.

Digital forensics: Enhancing low-quality images for better identification and analysis of evidence.

**Conclusion:**

The Image Resolution Enhancer ML model based on autoencoders demonstrates its potential as an effective solution for enhancing image quality. By leveraging the power of deep learning and autoencoder-based architectures, the model provides a reliable method for image resolution enhancement across various domains and applications. Continued research and development in this field are expected to yield further advancements in image processing and computer vision.

**Code Statement:**

**Importing Libraries:**

import numpy as np

import cv2

import glob

import tensorflow as tf

from tensorflow.keras import Model, Input, regularizers

from tensorflow.keras.layers import InputLayer,Dense, Conv2D, MaxPool2D, UpSampling2D, Add, Dropout,Flatten

from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint

from tensorflow.keras import layers, Input

from tensorflow.keras.models import Sequential, Model

from keras.preprocessing import image

import matplotlib.pyplot as plt

from sklearn.model\_selection import train\_test\_split

import pickle

import random

import os

import matplotlib.image as mpimg

%matplotlib inline

**Importing Image Dataset:**

import PIL

import os

import os.path

from PIL import Image

f = r'C:\Users\win10\ImageEnhancer\Imageset'

for file in os.listdir(f):

f\_img = f+"/"+file

img = Image.open(f\_img)

img = img.resize((80,80))

img.save(f\_img)

**Rescaling the Image set for training of model:**

plt.figure(figsize=(80,80))

test\_folder=r'C:\Users\win10\ImageEnhancer\Imageset'

for i in range(5):

file = random.choice(os.listdir(test\_folder))

image\_path= os.path.join(test\_folder, file)

img=mpimg.imread(image\_path)

ax=plt.subplot(1,5,i+1)

ax.title.set\_text(file)

plt.imshow(img)

**Preprocessing the image dataset:**

def create\_dataset(test\_folder):

img\_data\_array=[]

class\_name=[]

for dir1 in os.listdir(test\_folder):

for file in os.listdir(os.path.join(test\_folder, dir1)):

image\_path= os.path.join(test\_folder, dir1, file)

image= cv2.imread( image\_path, cv2.COLOR\_BGR2RGB)

image=np.array(image)

image = image.astype('float32')

image /= 255

img\_data\_array.append(image)

class\_name.append(dir1)

return img\_data\_array, class\_name

# extract the image array and class name

img\_data, class\_name =create\_dataset(r'C:\Users\win10\ImageEnhancer')

with open('img\_data','wb') as f:pickle.dump(img\_data, f)

print(len(img\_data))

**Train and Test splitting:**

all\_images = np.array(img\_data)

print(all\_images.shape)

#Split test and train data. all\_images will be our output images

train\_x, val\_x = train\_test\_split(all\_images, random\_state = 32, test\_size=0.2)

print(train\_x.shape)

print(val\_x.shape)

**Interpolation:**

#now we will make input images by lowering resolution without changing the size

def pixalate\_image(image, scale\_percent = 40):

width = int(image.shape[1] \* scale\_percent / 100)

height = int(image.shape[0] \* scale\_percent / 100)

dim = (width, height)

small\_image = cv2.resize(image, dim, interpolation = cv2.INTER\_AREA)

# scale back to original size

width = int(small\_image.shape[1] \* 100 / scale\_percent)

height = int(small\_image.shape[0] \* 100 / scale\_percent)

dim = (width, height)

low\_res\_image = cv2.resize(small\_image, dim, interpolation = cv2.INTER\_AREA)

return low\_res\_image

**Generating distorted images for training and validation set:**

train\_x\_px = []

for i in range(train\_x.shape[0]):

print(train\_x.shape)

temp = pixalate\_image(train\_x[i,:,:,:])

train\_x\_px.append(temp)

train\_x\_px = np.array(train\_x\_px) #Distorted images

# get low resolution images for the validation set

val\_x\_px = []

for i in range(val\_x.shape[0]):

temp = pixalate\_image(val\_x[i,:,:,:])

val\_x\_px.append(temp)

val\_x\_px = np.array(val\_x\_px) #Distorted images

**Building the model:**

Input\_img = Input(shape=(80, 80, 3))

#encoding architecture

x1 = Conv2D(64, (3, 3), activation='sigmoid', padding='same')(Input\_img)

x2 = Conv2D(64, (3, 3), activation='sigmoid', padding='same')(x1)

x3 = MaxPool2D(padding='same')(x2)

x4 = Conv2D(128, (3, 3), activation='sigmoid', padding='same')(x3)

x5 = Conv2D(128, (3, 3), activation='sigmoid', padding='same')(x4)

x6 = MaxPool2D(padding='same')(x5)

encoded = Conv2D(256, (3, 3), activation='sigmoid', padding='same')(x6)

#encoded = Conv2D(64, (3, 3), activation='relu', padding='same')(x2)

# decoding architecture

x7 = UpSampling2D()(encoded)

x8 = Conv2D(128, (3, 3), activation='sigmoid', padding='same')(x7)

x9 = Conv2D(128, (3, 3), activation='sigmoid', padding='same')(x8)

x10 = Add()([x5, x9])

x11 = UpSampling2D()(x10)

x12 = Conv2D(64, (3, 3), activation='sigmoid', padding='same')(x11)

x13 = Conv2D(64, (3, 3), activation='sigmoid', padding='same')(x12)

x14 = Add()([x2, x13])

# x3 = UpSampling2D((2, 2))(x3)

# x2 = Conv2D(128, (3, 3), activation='relu', padding='same')(x3)

# x1 = Conv2D(256, (3, 3), activation='relu', padding='same')(x2)

decoded = Conv2D(3, (3, 3), padding='same',activation='sigmoid', kernel\_regularizer=regularizers.l1(10e-10))(x14)

autoencoder = Model(Input\_img, decoded)

autoencoder.compile(optimizer='adam', loss='mse', metrics=['accuracy'])

**Autoencoder summary:**

autoencoder.summary()

**Training the model:**

early\_stopper = EarlyStopping(monitor='val\_loss', min\_delta=0.01, patience=50, verbose=1, mode='min')

model\_checkpoint = ModelCheckpoint('superResolution\_checkpoint3.h5', save\_best\_only = True)

history = autoencoder.fit(train\_x\_px,train\_x,

epochs=500,

validation\_data=(val\_x\_px, val\_x),

callbacks=[early\_stopper, model\_checkpoint])

**Predicting Accuracy of the model:**

results = autoencoder.evaluate(val\_x\_px, val\_x)

print('val\_loss, val\_accuracy', results)

**Printing Results:**

predictions = autoencoder.predict(val\_x\_px)

n = 4

plt.figure(figsize= (20,10))

for i in range(n):

ax = plt.subplot(3, n, i+1)

plt.imshow(val\_x\_px[i+20])

ax.get\_xaxis().set\_visible(False)

ax.get\_yaxis().set\_visible(False)

ax = plt.subplot(3, n, i+1+n)

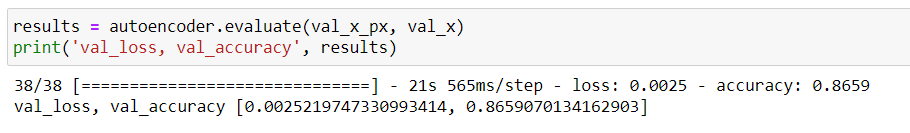
plt.imshow(predictions[i+20])

ax.get\_xaxis().set\_visible(False)

ax.get\_yaxis().set\_visible(False)

plt.show()

**Results:**

****