Introduction - IMAGE TO SKETCH CONVERSION

Objective:

To convert image to sketch using GAN and contrastive learning.

Import Necessary Libraries

```
import numpy as np
import tensorflow as tf
import keras
from keras.layers import Dense, Conv2D, MaxPool2D, UpSampling2D,
Dropout, Input
from tensorflow.keras.utils import img_to_array
import matplotlib.pyplot as plt
import cv2
from tqdm import tqdm
import os
import re
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision.datasets as datasets
import torchvision.transforms as transforms
from torch.utils.data import DataLoader
from torchvision.utils import save_image
from PIL import Image

#Set random seed for reproducibility
torch.manual_seed(0)
#Set device
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

Load data

This dataset consist of 188 image and their corresponding sketches. As these images aren't enough for training our autoencoder model, we have augmented them using open cv library. After Augmentation we have got around 1500 images, these 1500 images. These images are converted into array and are stored in the list.

```
# to get the files in proper order
def sorted_alphanumeric(data):
    convert = lambda text: int(text) if text.isdigit() else
text.lower()
    alphanum_key = lambda key: [convert(c) for c in re.split('([0-
9]+)',key)]
    return sorted(data,key = alphanum_key)
```

```
# defining the size of image
SIZE = 256

image_path = '/kaggle/input/cuhk-face-sketch-database-cufs-2/photos'
img_array = []

sketch_path =
'/kaggle/input/cuhk-face-sketch-database-cufs-2/sketches'
sketch_array = []

image_file = sorted_alphanumeric(os.listdir(image_path))
sketch_file = sorted_alphanumeric(os.listdir(sketch_path))


for i in tqdm(image_file):
    image = cv2.imread(image_path + '/' + i,1)

    # as opencv load image in bgr format converting it to rgb
    image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

    # resizing images
    image = cv2.resize(image, (SIZE, SIZE))

    # normalizing image
    image = image.astype('float32') / 255.0

    #appending normal normal image
    img_array.append(img_to_array(image))
    # Image Augmentation

    # horizontal flip
    img1 = cv2.flip(image,1)
    img_array.append(img_to_array(img1))
     #vertical flip
    img2 = cv2.flip(image,-1)
    img_array.append(img_to_array(img2))
     #vertical flip
    img3 = cv2.flip(image,-1)
    # horizontal flip
    img3 = cv2.flip(img3,1)
    img_array.append(img_to_array(img3))
    # rotate clockwise
    img4 = cv2.rotate(image, cv2.ROTATE_90_CLOCKWISE)
    img_array.append(img_to_array(img4))
    # flip rotated image
    img5 = cv2.flip(img4,1)
    img_array.append(img_to_array(img5))
     # rotate anti clockwise
    img6 = cv2.rotate(image, cv2.ROTATE_90_COUNTERCLOCKWISE)
```

```
    img_array.append(img_to_array(img6))
    # flip rotated image
    img7 = cv2.flip(img6,1)
    img_array.append(img_to_array(img7))


for i in tqdm(sketch_file):
    image = cv2.imread(sketch_path + '/' + i,1)

    # as opencv load image in bgr format converting it to rgb
    image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

    # resizing images
    image = cv2.resize(image, (SIZE, SIZE))

    # normalizing image
    image = image.astype('float32') / 255.0
    # appending normal sketch image
    sketch_array.append(img_to_array(image))

    #Image Augmentation
    # horizontal flip
    img1 = cv2.flip(image,1)
    sketch_array.append(img_to_array(img1))
     #vertical flip
    img2 = cv2.flip(image,-1)
    sketch_array.append(img_to_array(img2))
     #vertical flip
    img3 = cv2.flip(image,-1)
    # horizontal flip
    img3 = cv2.flip(img3,1)
    sketch_array.append(img_to_array(img3))
    # rotate clockwise
    img4 = cv2.rotate(image, cv2.ROTATE_90_CLOCKWISE)
    sketch_array.append(img_to_array(img4))
    # flip rotated image
    img5 = cv2.flip(img4,1)
    sketch_array.append(img_to_array(img5))
     # rotate anti clockwise
    img6 = cv2.rotate(image, cv2.ROTATE_90_COUNTERCLOCKWISE)
    sketch_array.append(img_to_array(img6))
    # flip rotated image
    img7 = cv2.flip(img6,1)
    sketch_array.append(img_to_array(img7))
```

```
100%|██████████| 188/188 [00:02<00:00, 84.97it/s]
100%|██████████| 188/188 [00:02<00:00, 86.47it/s]
```

```
print("Total number of sketch images:",len(sketch_array))
print("Total number of images:",len(img_array))
```

```
Total number of sketch images: 1504
Total number of images: 1504
```

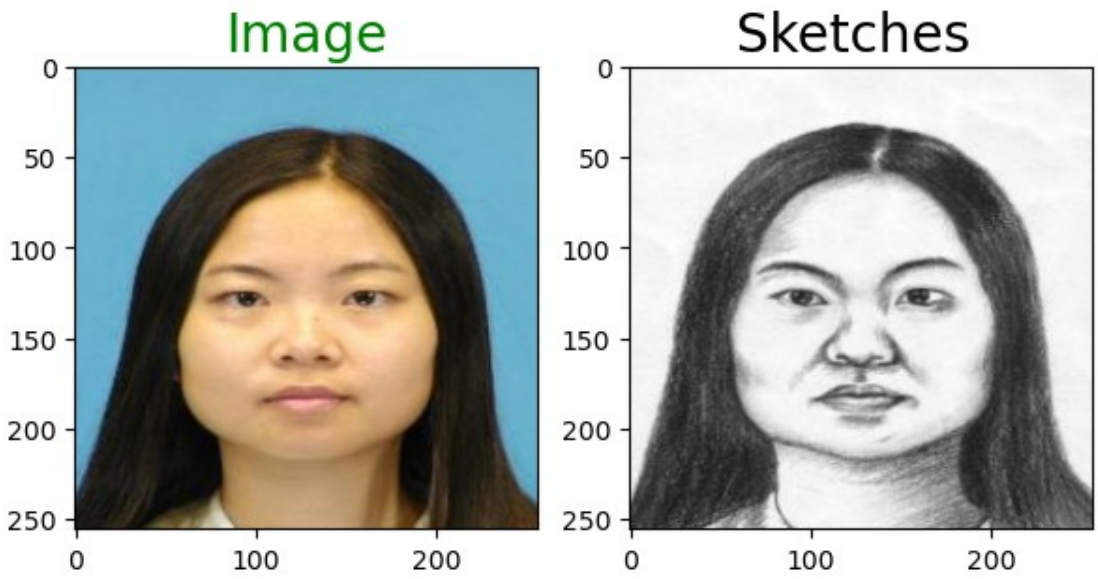
Visualizing images

Here we have plotted all augmented images and its augmented sketches

```
# defining function to plot images pair
def plot_images(image, sketches):
    plt.figure(figsize=(7,7))
    plt.subplot(1,2,1)
    plt.title('Image', color = 'green', fontsize = 20)
    plt.imshow(image)
    plt.subplot(1,2,2)
    plt.title('Sketches ', color = 'black', fontsize = 20)
    plt.imshow(sketches)

    plt.show()

ls = [i for i in range(0,65,8)]
for i in ls:
    plot_images(img_array[i],sketch_array[i])
```



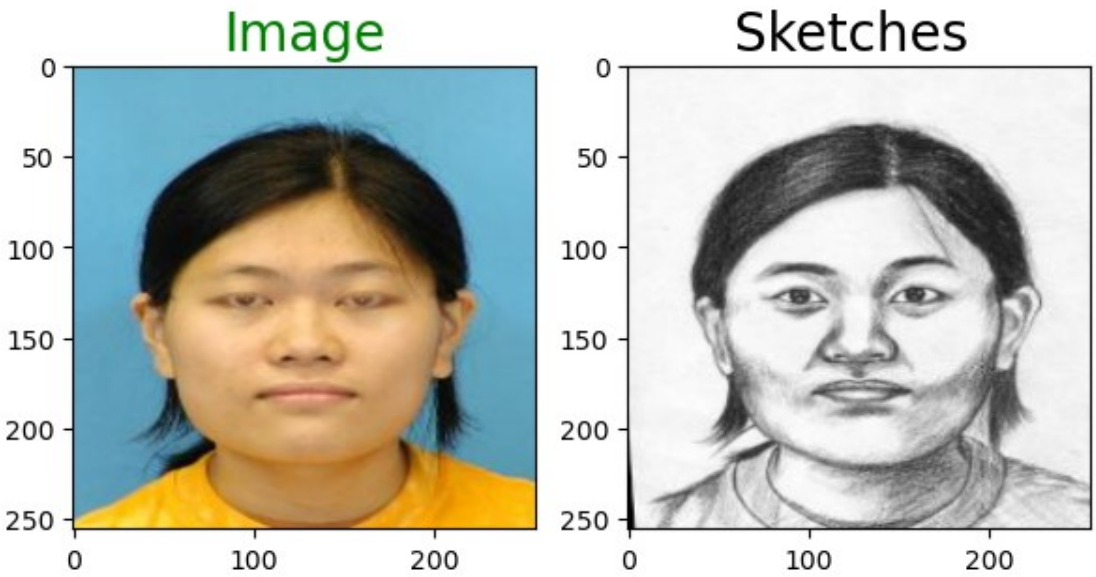


Image
Sketches
0
50
100
150
200
250
0
100
200

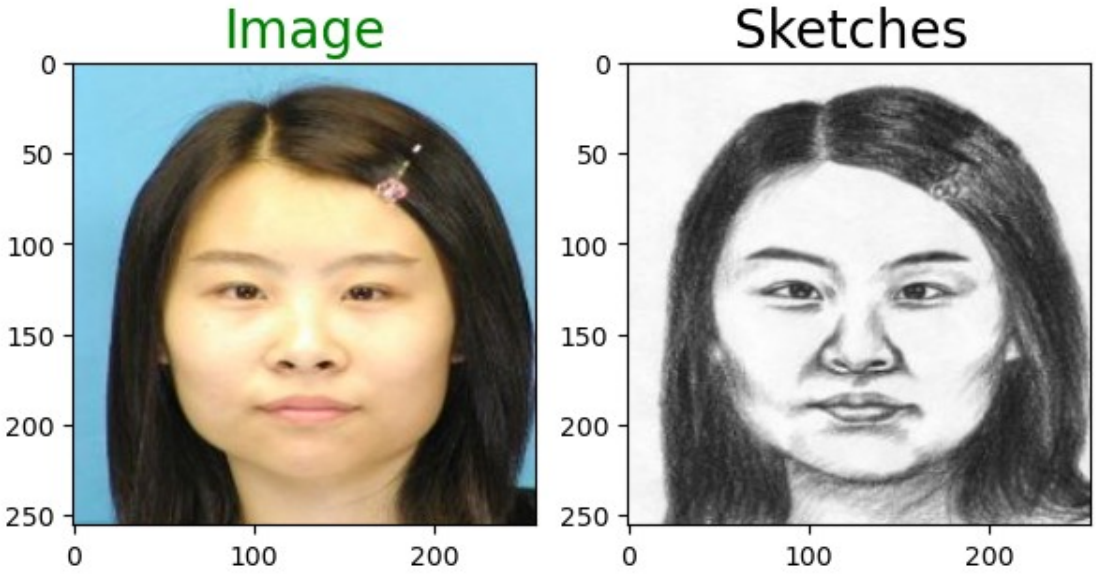


Image
Sketches
0
50
100
150
200
250
0
100
200

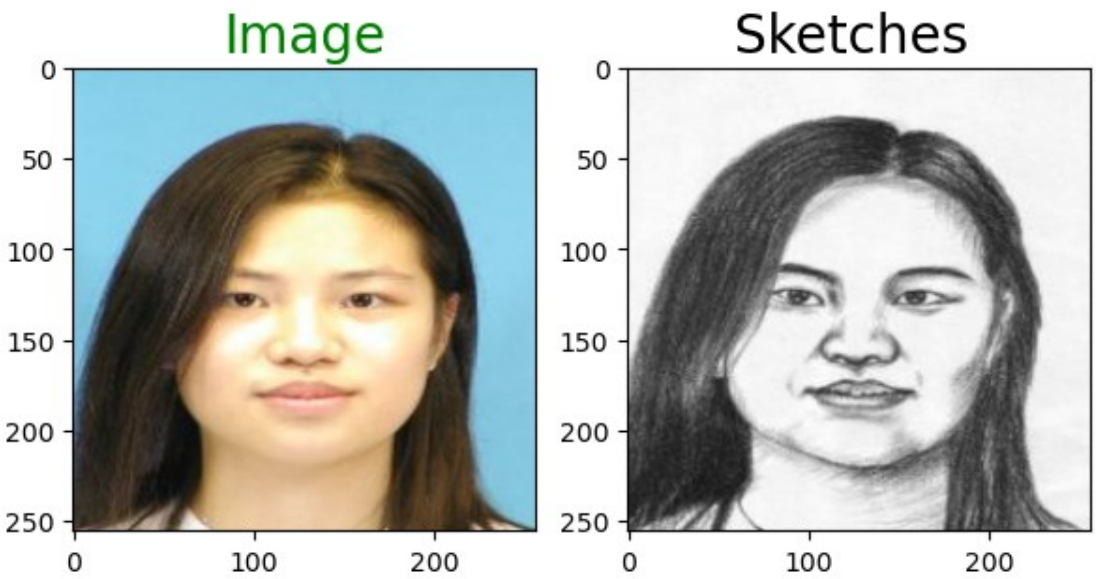


Image
Sketches
0
50
100
150
200
250
0
100
200

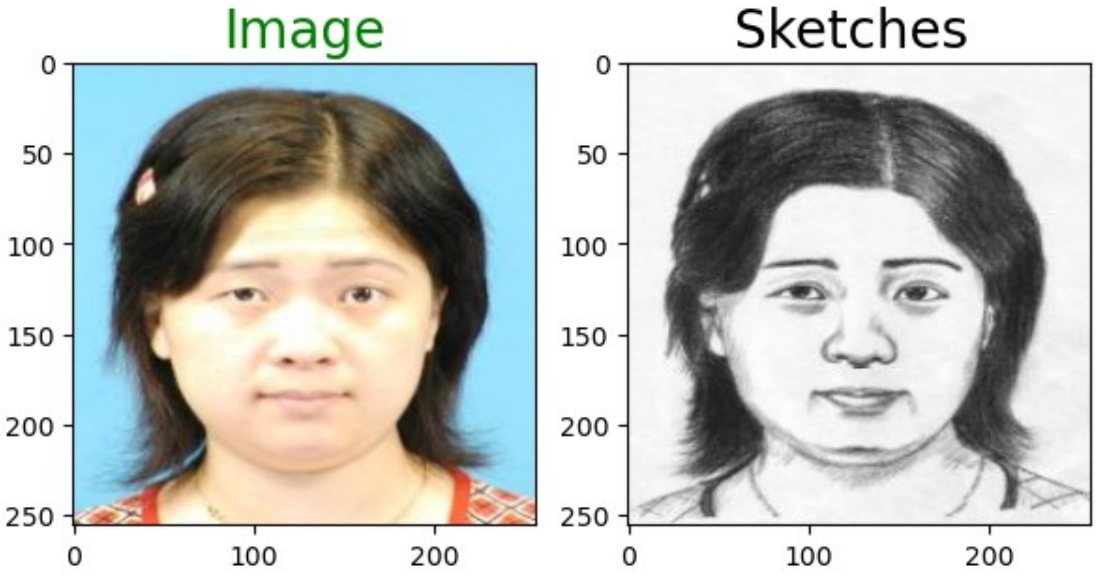


Image
Sketches
0
50
100
150
200
250
0
100
200

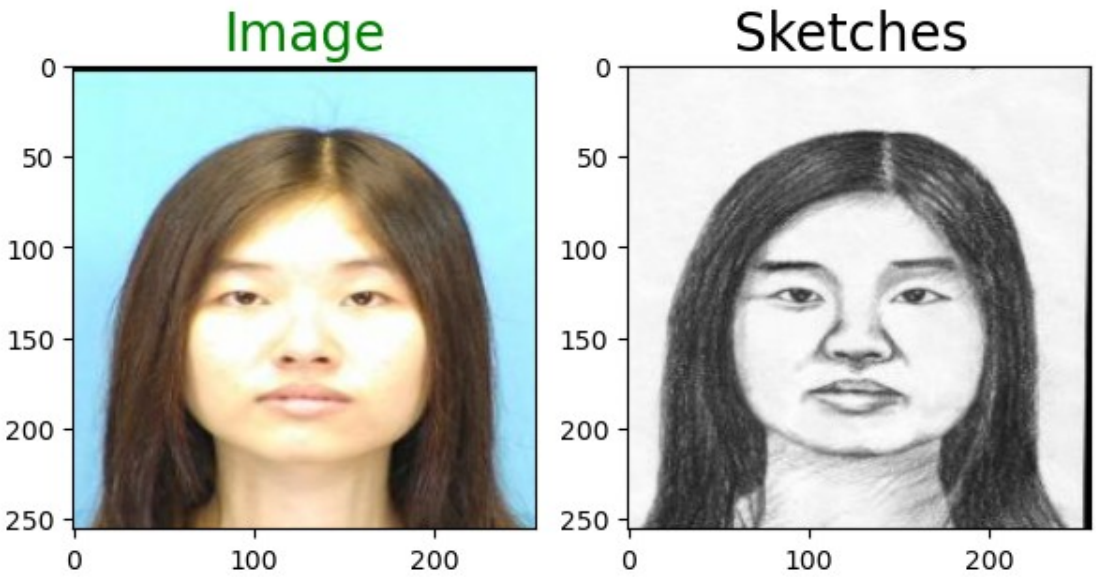


Image
Sketches
0
50
100
150
200
250
0
100
200

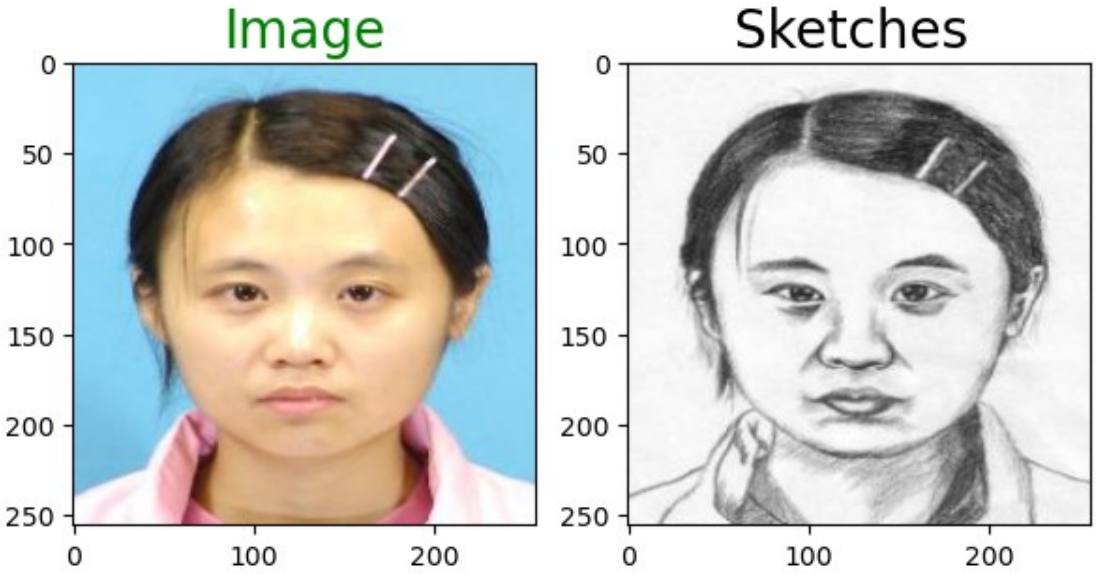
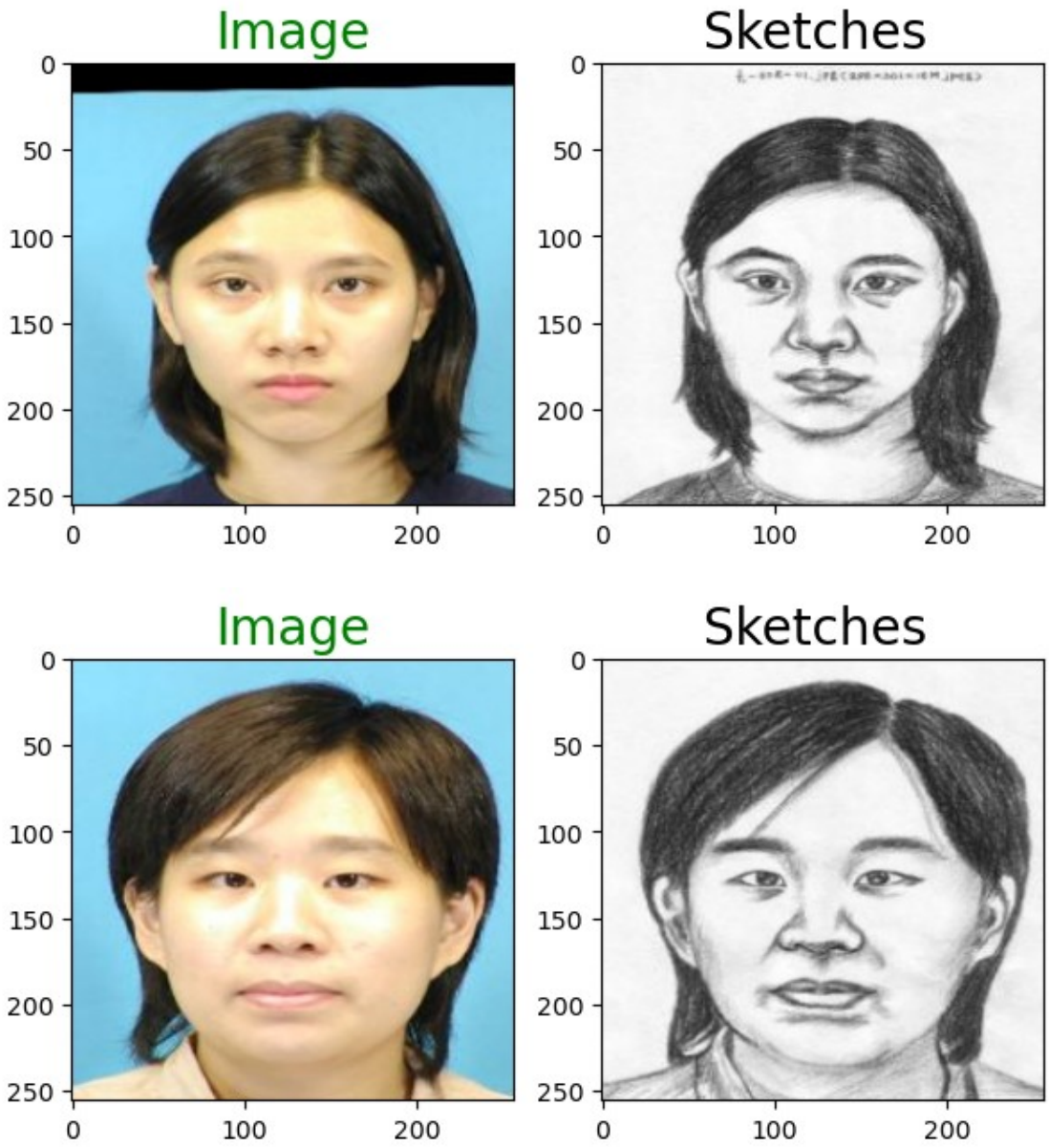


Image
Sketches
0
50
100
150
200
250
0
100
200



Slicing and reshaping

Out of 1504 images We have sliced them to two part. train images consist 1400 images while test images contains 104 images. After slicing image array, we reshaped them so that images can be fed directly into our encoder network

```
train_sketch_image = sketch_array[:1400]
train_image = img_array[:1400]
test_sketch_image = sketch_array[1400:]
test_image = img_array[1400:]
# reshaping
train_sketch_image = np.reshape(train_sketch_image,
(len(train_sketch_image),SIZE,SIZE,3))
```

```
train_image = np.reshape(train_image, (len(train_image),SIZE,SIZE,3))
print('Train color image shape:',train_image.shape)
test_sketch_image = np.reshape(test_sketch_image,
(len(test_sketch_image),SIZE,SIZE,3))
test_image = np.reshape(test_image, (len(test_image),SIZE,SIZE,3))
print('Test color image shape',test_image.shape)
```

```
Train color image shape: (1400, 256, 256, 3)
Test color image shape (104, 256, 256, 3)
```

Generator Network

```
# Define the generator network
class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()
        self.conv1 = nn.Conv2d(1, 64, kernel_size=3, stride=1,
padding=1)
        self.bn1 = nn.BatchNorm2d(64)
        self.conv2 = nn.Conv2d(64, 128, kernel_size=3, stride=2,
padding=1)
        self.bn2 = nn.BatchNorm2d(128)
        self.conv3 = nn.Conv2d(128, 256, kernel_size=3, stride=2,
padding=1)
        self.bn3 = nn.BatchNorm2d(256)
        self.conv4 = nn.Conv2d(256, 512, kernel_size=3, stride=2,
padding=1)
        self.bn4 = nn.BatchNorm2d(512)
        self.convT1 = nn.ConvTranspose2d(512, 256, kernel_size=4,
stride=2, padding=1)
        self.bnT1 = nn.BatchNorm2d(256)
        self.convT2 = nn.ConvTranspose2d(256, 128, kernel_size=4,
stride=2, padding=1)
        self.bnT2 = nn.BatchNorm2d(128)
        self.convT3 = nn.ConvTranspose2d(128, 64, kernel_size=4,
stride=2, padding=1)
        self.bnT3 = nn.BatchNorm2d(64)
        self.convT4 = nn.ConvTranspose2d(64, 1, kernel_size=3,
stride=1, padding=1)

    def forward(self, x):
        # Encoder
        x = self.conv1(x)
        x = nn.functional.relu(self.bn1(x))
        x = self.conv2(x)
        x = nn.functional.relu(self.bn2(x))
        x = self.conv3(x)
        x = nn.functional.relu(self.bn3(x))
        x = self.conv4(x)
        x = nn.functional.relu(self.bn4(x))
        # Decoder
```

```
        x = self.convT1(x)
        x = nn.functional.relu(self.bnT1(x))
        x = self.convT2(x)
        x = nn.functional.relu(self.bnT2(x))
        x = self.convT3(x)
        x = nn.functional.relu(self.bnT3(x))
        x = self.convT4(x)
        x = torch.tanh(x)
        return x
```

Discriminator Network

```
# Define the discriminator network
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.conv1 = nn.Conv2d(2, 64, kernel_size=4, stride=2,
padding=1)
        self.conv2 = nn.Conv2d(64, 128, kernel_size=4, stride=2,
padding=1)
        self.bn2 = nn.BatchNorm2d(128)
        self.conv3 = nn.Conv2d(128, 256, kernel_size=4, stride=2,
padding=1)
        self.bn3 = nn.BatchNorm2d(256)
        self.conv4 = nn.Conv2d(256, 512, kernel_size=4, stride=2,
padding=1)
        self.bn4 = nn.BatchNorm2d(512)
        self.conv5 = nn.Conv2d(512, 1, 4, padding=1)

    def forward(self, x, y):
    # Concatenate the input image and the generated image
        x = torch.cat([x, y], dim=1)
        x = nn.functional.leaky_relu(self.conv1(x),
negative_slope=0.2)
        x = nn.functional.leaky_relu(self.bn2(self.conv2(x)),
negative_slope=0.2)
        x = nn.functional.leaky_relu(self.bn3(self.conv3(x)),
negative_slope=0.2)
        x = nn.functional.leaky_relu(self.bn4(self.conv4(x)),
negative_slope=0.2)
        x = self.conv5(x)
        return x
```

Contrastive Loss

```
#Define the contrastive loss
  class ContrastiveLoss(nn.Module):
    def init(self, margin=2.0):
      super(ContrastiveLoss, self).init()
      self.margin = margin
    def forward(self, x, y, label):
```

```
        dist = torch.sqrt(torch.sum((x - y) ** 2, dim=1))
        loss = torch.mean((1 - label) * torch.pow(dist, 2) + label *
torch.pow(torch.clamp(self.margin - dist, min=0.0), 2))
        return loss
```

Compiling and Fitting our model

Here we have used Adam optimizer and mean_squared_error as loss and have trained model.

```
#Define the generator model
generator = Generator().to(device)

#Define the discriminator model
discriminator = Discriminator().to(device)

#Define the contrastive loss function
criterion = ContrastiveLoss().to(device)

#Define the optimizers
optimizer_g = optim.Adam(generator.parameters(), lr=0.0002,
betas=(0.5, 0.999))
optimizer_d = optim.Adam(discriminator.parameters(), lr=0.0002,
betas=(0.5, 0.999))

#Define the transforms for the dataset
transform = transforms.Compose([
transforms.Resize((256, 256)),
transforms.ToTensor(),
transforms.Normalize(mean=[0.5], std=[0.5])
])

#Define the data loader
dataloader = DataLoader(dataset, batch_size=1, shuffle=True)

#Start the training
num_epochs = 50

for epoch in range(num_epochs):
  for i, (x_real, _) in enumerate(dataloader):
    # Generate a fake image from the input image
    x_real = x_real.to(device)
    x_fake = generator(x_real)
    # Train the discriminator
    optimizer_d.zero_grad()
    y_real = torch.ones(x_real.size(0), 1, 30, 30).to(device)
    y_fake = torch.zeros(x_real.size(0), 1, 30, 30).to(device)
    d_real = discriminator(x_real, x_fake.detach())
    d_fake = discriminator(x_real, x_fake)
```

```
    d_loss_real = criterion(d_real, y_real,
label=torch.ones_like(y_real))
    d_loss_fake = criterion(d_fake, y_fake,
label=torch.ones_like(y_fake))
    d_loss = d_loss_real + d_loss_fake
    d_loss.backward()
    optimizer_d.step()

    # Train the generator
    optimizer_g.zero_grad()
    y_real = torch.ones(x_real.size(0), 1, 30, 30).to(device)
    x_fake = generator(x_real)
    d_fake = discriminator(x_real, x_fake)
    g_loss = criterion(d_fake, y_real, label=torch.ones_like(y_real))
    g_loss.backward()
    optimizer_g.step()

model.compile(optimizer = tf.keras.optimizers.Adam(learning_rate =
0.0001), loss = 'mean_absolute_error',
              metrics = ['acc'])

model.fit(train_image, train_sketch_image, epochs = 10000, verbose =
0)
```

```
2023-04-01 03:48:30.179659: E
tensorflow/core/grappler/optimizers/meta_optimizer.cc:954] layout
failed: INVALID_ARGUMENT: Size of values 0 does not match size of
permutation 4 @ fanin shape
inmodel/sequential_6/dropout/dropout/SelectV2-2-TransposeNHWCToNCHW-
LayoutOptimizer
```

```
<keras.callbacks.History at 0x7f4bc0243ad0>
```

```
model.save('/kaggle/working/final_model.h5')
#os.remove("/kaggle/working/file_name.csv")
```

Evaluating our model

```
prediction_on_test_data = model.evaluate(test_image,
test_sketch_image)
print("Loss: ", prediction_on_test_data[0])
print("Accuracy: ", np.round(prediction_on_test_data[1] * 100,1))
```

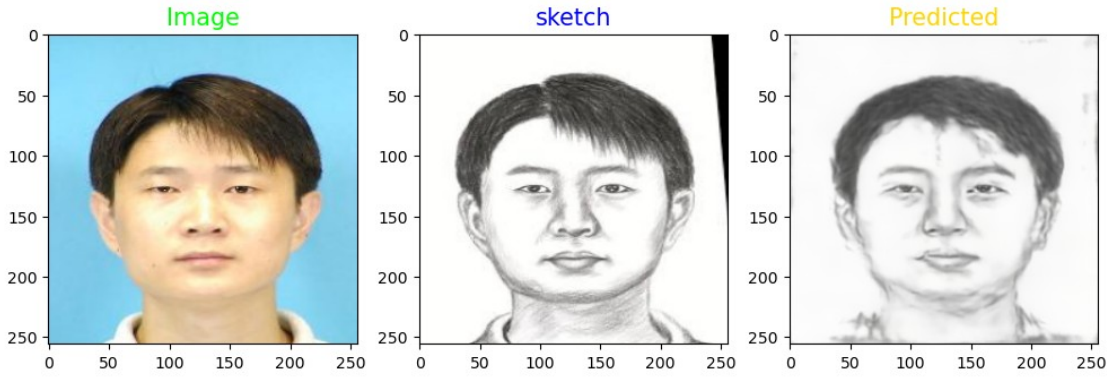
```
4/4 [==============================] - 1s 62ms/step - loss: 0.0813 -
acc: 0.6714
Loss:  0.08133046329021454
Accuracy:  67.1
```

Plotting our predicted sketch along with real sketch

```
def show_images(real,sketch, predicted):
    plt.figure(figsize = (12,12))
    plt.subplot(1,3,1)
    plt.title("Image",fontsize = 15, color = 'Lime')
    plt.imshow(real)
    plt.subplot(1,3,2)
    plt.title("sketch",fontsize = 15, color = 'Blue')
    plt.imshow(sketch)
    plt.subplot(1,3,3)
    plt.title("Predicted",fontsize = 15, color = 'gold')
    plt.imshow(predicted)

ls = [i for i in range(0,95,8)]
for i in ls:
    predicted
=np.clip(model.predict(test_image[i].reshape(1,SIZE,SIZE,3)),0.0,1.0).
reshape(SIZE,SIZE,3)
    show_images(test_image[i],test_sketch_image[i],predicted)
```

```
1/1 [==============================] - 0s 380ms/step
1/1 [==============================] - 0s 28ms/step
1/1 [==============================] - 0s 25ms/step
1/1 [==============================] - 0s 26ms/step
1/1 [==============================] - 0s 26ms/step
1/1 [==============================] - 0s 28ms/step
1/1 [==============================] - 0s 27ms/step
1/1 [==============================] - 0s 26ms/step
1/1 [==============================] - 0s 26ms/step
1/1 [==============================] - 0s 26ms/step
1/1 [==============================] - 0s 26ms/step
1/1 [==============================] - 0s 25ms/step
```



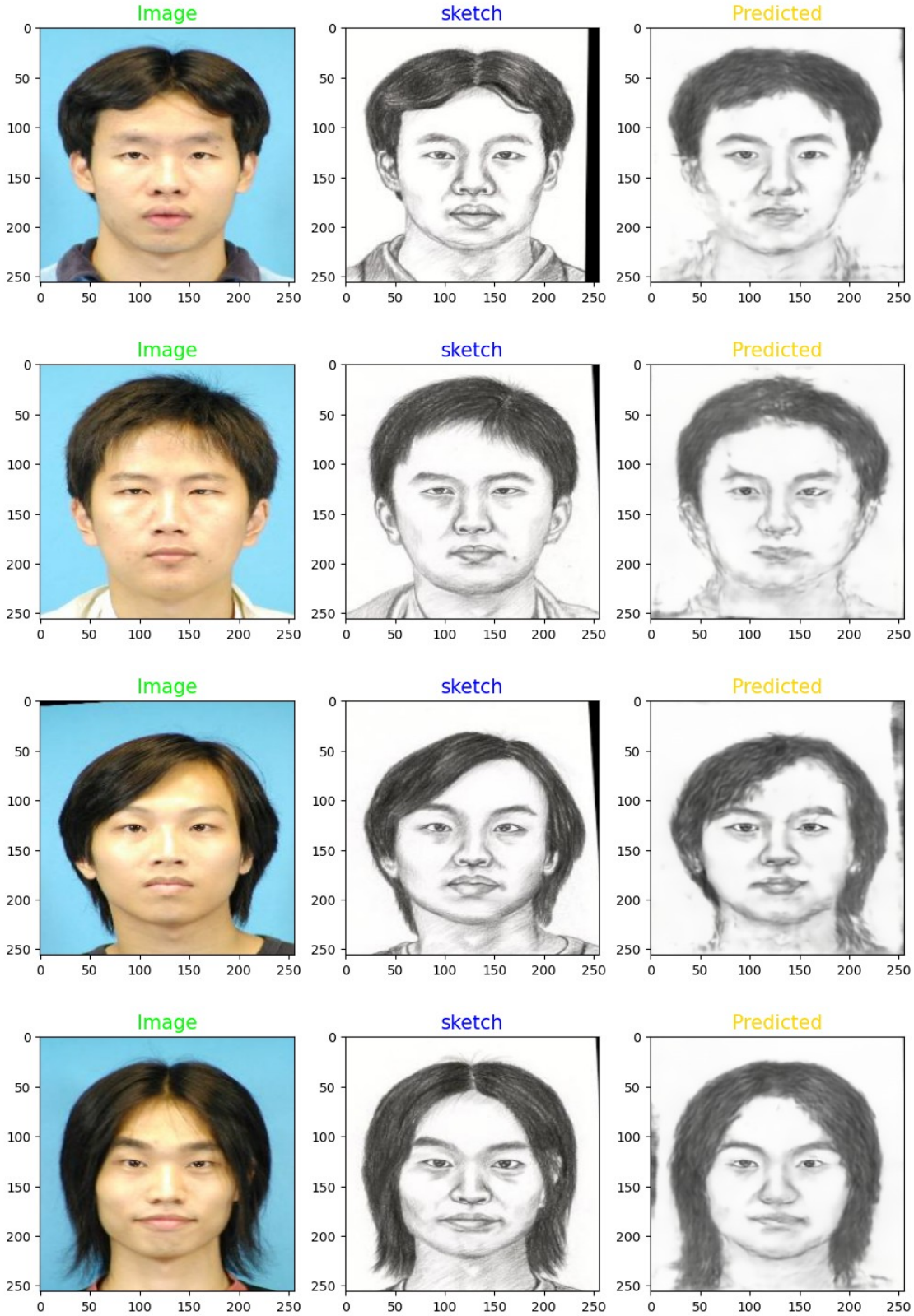


Image
sketch
Predicted
Image
sketch
Predicted
Image
sketch
Predicted
Image
sketch
Predicted

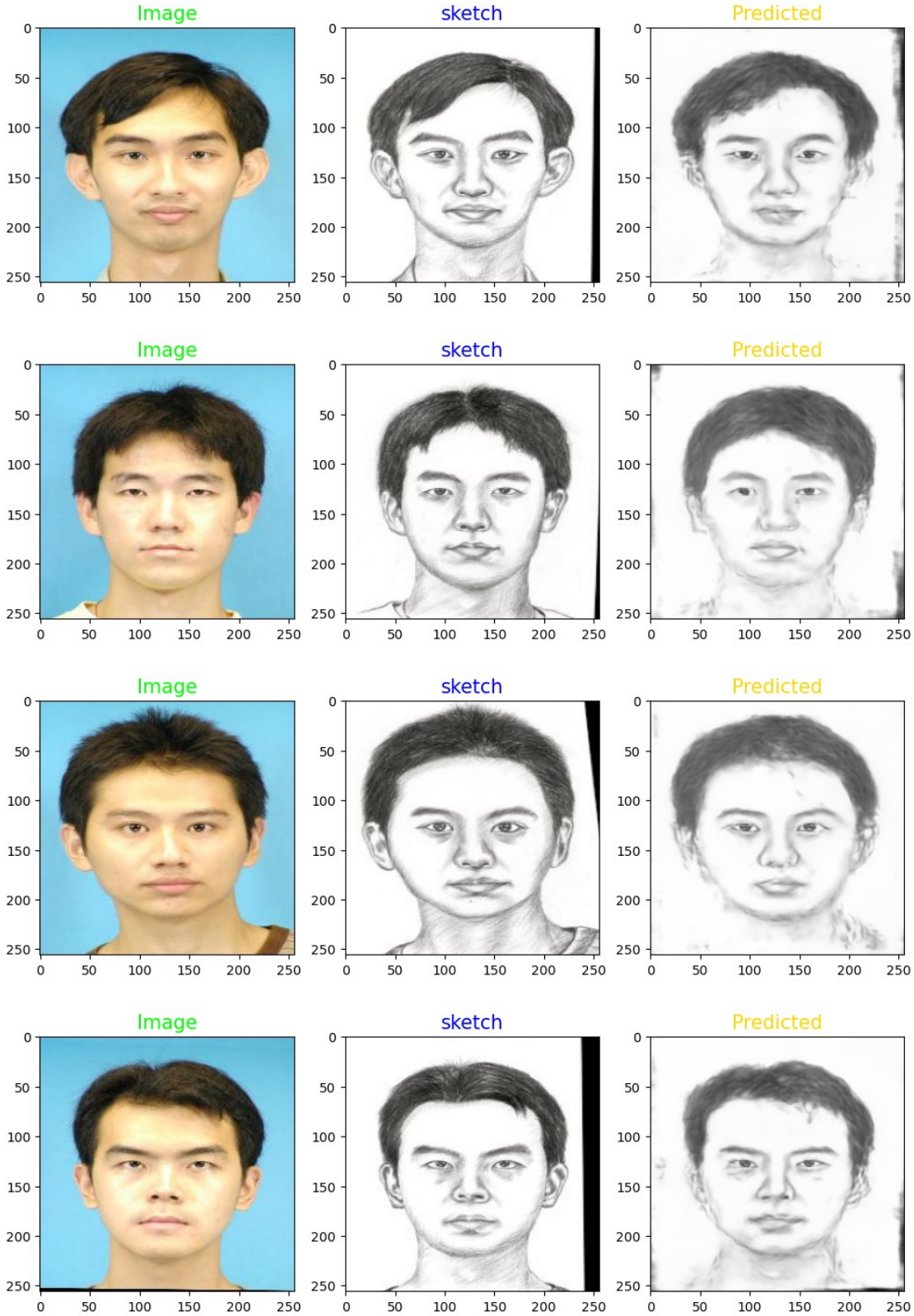


Image
sketch
Predicted
Image
sketch
Predicted
Image
sketch
Predicted
Image
sketch
Predicted

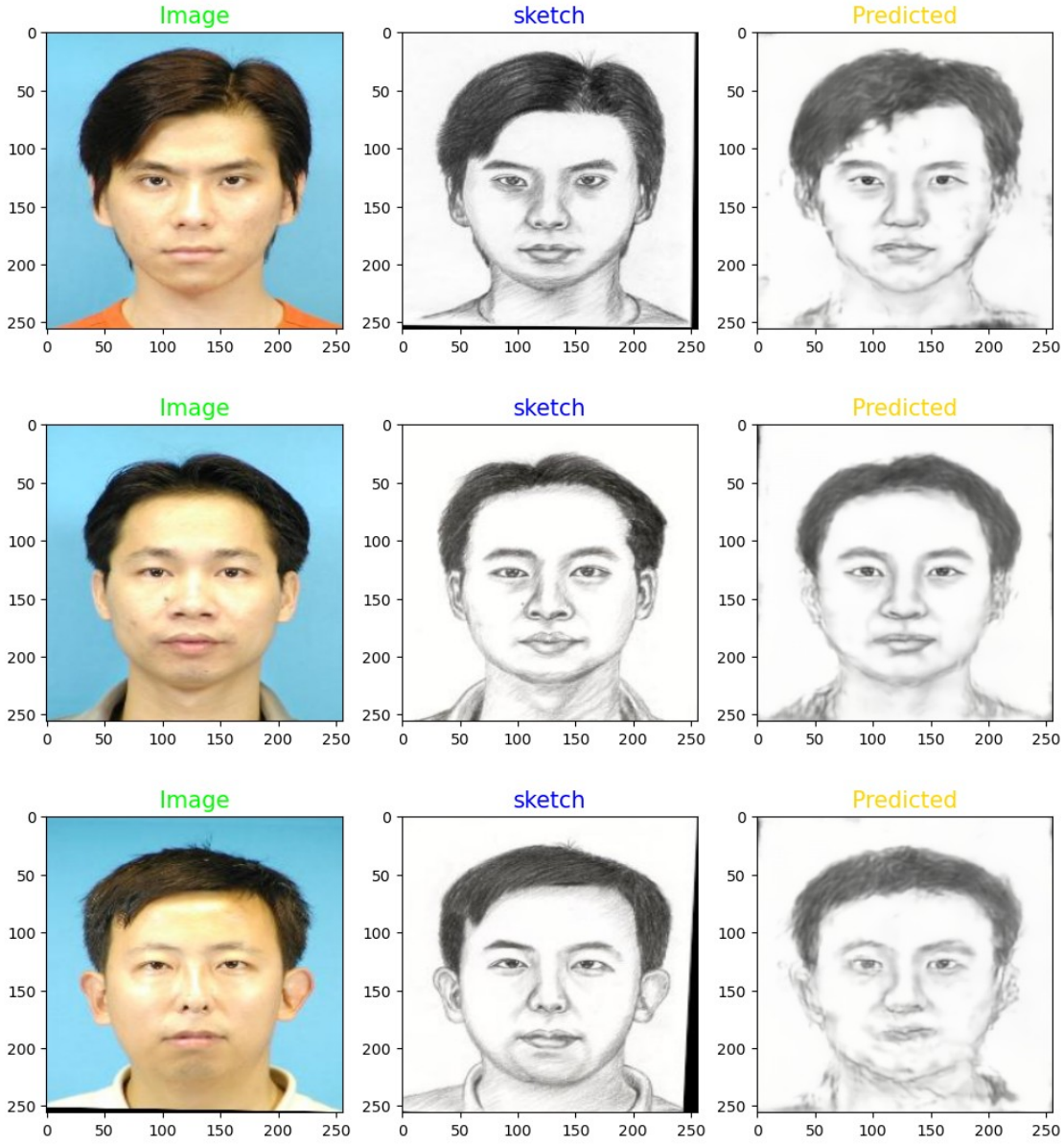


Image
sketch
Predicted
Image
sketch
Predicted
Image
sketch
Predicted