INTEL UNNATI 2024

PIXELATED IMAGE DETECTION AND CORRECTION

DETECTION MODEL ALGORITHM AND CODE BREAKDOWN:

DETECTION MODEL CODE:

```
import tensorflow as tf
import numpy as np
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Input, Conv2D, MaxPooling2D, Flatten,   Dense
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau
from tensorflow.keras.preprocessing import image

# Define the input shape
img_height, img_width = 150, 150

# Data Augmentation (Optional)
train_datagen = ImageDataGenerator(
    rescale=1./255,  # Normalize pixel values (0-1)
    shear_range=0.2,  # Optional: Random shear images for data augmentation
    zoom_range=0.2,  # Optional: Randomly zoom images for data augmentation
    horizontal_flip=True  # Optional: Randomly flip images horizontally
)

# Define paths to your training and test data directories (replace with your actual paths)
train_data_dir = r"/Users/praveen/Desktop/annotated images/train_data"
test_data_dir = r"/Users/praveen/Desktop/annotated images/test data"

# Load training and test data using flow_from_directory
train_generator = train_datagen.flow_from_directory(
    train_data_dir,
    target_size=(img_height, img_width),
    batch_size=32,  # Adjust batch size based on your hardware limitations
    class_mode='binary'  # Assuming binary classification (pixelated vs non-pixelated)
)
```

```
test_datagen = ImageDataGenerator(rescale=1./255)  # Inherit normalization from
train_datagen
test_generator = test_datagen.flow_from_directory(
    test_data_dir,
    target_size=(img_height, img_width),
    batch_size=32,  # Adjust batch size based on your hardware limitations
    class_mode='binary'  # Assuming binary classification
)

# Define CNN model
# Create the Sequential model
model = Sequential([
    Input(shape=(img_height, img_width, 3)),  # Specify input shape here
    Conv2D(32, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    Conv2D(64, (3, 3), activation='relu'),  # Add another convolutional layer
    MaxPooling2D((2, 2)),
    Flatten(),
    Dense(64, activation='relu'),
    Dense(1, activation='sigmoid')  # Output layer with sigmoid for binary classification
])

# Compile the model
model.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=['accuracy'])

# Define callbacks for early stopping and learning rate reduction
early_stopping = EarlyStopping(monitor='val_loss', patience=3)
reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.5, patience=2,
min_lr=0.0001)

# Train the model
model.fit(train_generator,
          epochs=10,  # Adjust number of epochs based on your data and validation
          validation_data=test_generator,
          callbacks=[early_stopping, reduce_lr])

# Save the model for future use
model.save('/Users/praveen/Desktop/annotated_images/model/
pixelation_detector_model.h5')

# Use the trained model to predict on a new image
```

```
def predict_image(image_path):
    # Load and preprocess the image
    img = image.load_img(image_path, target_size=(img_height, img_width))
    img = image.img_to_array(img)
    img = np.expand_dims(img, axis=0)  # Add a new dimension for batch size
    img = img / 255.0  # Normalize the image

    # Make prediction on the preprocessed image
    prediction = model.predict(img)

    # Interpret the prediction (round the sigmoid output to 0 or 1)
    predicted_class = np.round(prediction[0][0])
    if predicted_class == 0:
        print("Image is classified as non-pixelated.")
    else:
        print("Image is classified as pixelated.")

# Example usage (replace with your image path)
image_path = "/Users/praveen/Downloads/WhatsApp Image 2024-07-05 at
19.44.28.jpeg"
predict_image(image_path)
```

DETECTION MODEL ALGORITHM:

Step - 1: Data Preprocessing:

- Load training and test images from directories.
- Resize images to a fixed size (e.g., 150x150 pixels).
- Normalize pixel values (typically from 0-255 to 0-1).
- (Optional) Apply data augmentation techniques (e.g., random shear, zoom, flip) to increase training data variety.

Step - 2: Model Definition:

- Create a CNN model using the Sequential API from TensorFlow.
- The model typically consists of convolutional layers for feature extraction, pooling layers for dimensionality reduction, and fully-connected layers for classification.
- In this example, the model has:
  - An input layer that takes a 3-channel image (RGB) of the specified size (150x150).

- Two convolutional layers with 32 and 64 filters (learnable kernels) of size 3x3, followed by ReLU activation for non-linearity.
- Two max pooling layers with a pool size of 2x2 for down sampling.
- A flattening layer to transform the 2D feature maps into a 1D vector.
- A fully-connected layer with 64 neurons and ReLU activation.
- An output layer with one neuron and sigmoid activation for binary classification (pixelated vs non-pixelated).

Step - 3: Model Compilation:

- Choose an optimizer (e.g., Adam) that updates the model's weights based on the loss function.
- Define a loss function (e.g., binary cross-entropy) that measures the difference between the model's predictions and true labels.
- Specify a metric (e.g., accuracy) to track the model's performance during training.

Step - 4: Model Training:

- Feed the preprocessed training data (images and labels) into the model in batches.
- During each iteration (epoch):
  - The model calculates the loss for the current batch.
  - The optimizer updates the model's weights based on the calculated loss to improve its predictions.
  - The model's performance on the validation data (a subset of the training data) is evaluated using the chosen metric (e.g., accuracy).

Step - 5: Model Evaluation (Optional):

- After training, evaluate the model's performance on the unseen test data using the chosen metric.
- Analyze the results to assess the model's effectiveness in classifying pixelated images.

Step - 6: Prediction (Optional):

- Preprocess a new image following the same steps as for training data.
- Feed the preprocessed image into the trained model.
- The model outputs a prediction for the image class (pixelated or non-pixelated in this case).

DETECTION SOURCE CODE BREAKDOWN:

Part 1: Imports and Data Setup

Python

```
import tensorflow as tf

import numpy as np

from tensorflow.keras.preprocessing.image import ImageDataGenerator

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Input, Conv2D, MaxPooling2D, Flatten, Dense

from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau

from tensorflow.keras.preprocessing import image
```

- This section imports necessary libraries for image processing, model building, training, and prediction.
    - tensorflow: The core library for building and training machine learning models.
    - numpy: Used for numerical computations and array manipulation.
    - ImageDataGenerator: Used for data augmentation (optional technique to increase training data variety).
    - Sequential: A class for building models layer by layer.
    - Convolutional and pooling layers (Conv2D, MaxPooling2D): Key components of Convolutional Neural Networks (CNNs) for image classification.
    - Flatten, Dense: Layers used to transform the data into a format suitable for the output layer.
    - EarlyStopping, ReduceLROnPlateau: Callbacks to prevent overfitting and adjust learning rate during training.
    - image: Utility functions for loading and pre-processing images.

Part 2: Data Augmentation (Optional)

Python

```
train_datagen = ImageDataGenerator(

  rescale=1./255, # Normalize pixel values (0-1)

  shear_range=0.2, # Optional: Random shear images for data augmentation

  zoom_range=0.2, # Optional: Randomly zoom images for data augmentation
```

```
  horizontal_flip=True # Optional: Randomly flip images horizontally
)
```

- This section defines a data generator object for training data.
- rescale=1./255: Normalizes pixel values in images from 0-255 range to 0-1 range, a common pre-processing step for image data.
- The remaining arguments (shear_range, zoom_range, horizontal_flip) are optional data augmentation techniques. These randomly modify training images (shear, zoom, flip) to create more variations and improve model generalization.

Part 3: Data Loading

Python

```
# Define paths to your training and test data directories (replace with your actual paths)
train_data_dir = r"/Users/praveen/Desktop/annotated images/train_data"
test_data_dir = r"/Users/praveen/Desktop/annotated images/test data"

# Load training and test data using flow_from_directory
train_generator = train_datagen.flow_from_directory(
  train_data_dir,
  target_size=(img_height, img_width),
  batch_size=32, # Adjust batch size based on your hardware limitations
  class_mode='binary' # Assuming binary classification (pixelated vs non-pixelated)
)

test_datagen = ImageDataGenerator(rescale=1./255) # Inherit normalization from train_datagen
test_generator = test_datagen.flow_from_directory(
  test_data_dir,
  target_size=(img_height, img_width),
```

```
  batch_size=32, # Adjust batch size based on your hardware limitations

  class_mode='binary' # Assuming binary classification

)
```

- This section defines paths to your training and test data directories (replace these with your actual paths).
- flow_from_directory function from ImageDataGenerator is used to load images from directories while automatically applying the defined transformations (normalization in this case).
- target_size specifies the expected image size for the model (150x150 in this example).
- batch_size defines the number of images processed together during training (adjust based on your hardware capabilities).
- class_mode='binary' indicates binary classification (pixelated vs non-pixelated images).

Part 4: Model Definition

Python

```
# Define CNN model

# Create the Sequential model

model = Sequential([

  Input(shape=(img_height, img_width, 3)), # Specify input shape here

  Conv2D(32, (3, 3), activation='relu'),

  MaxPooling2D((2, 2)),

  Conv2D(64, (3, 3), activation='relu'), # Add another convolutional layer

  MaxPooling2D
```

- **Forward Pass:** During each training iteration (epoch), the model takes a batch of training images and their labels as input.
- **Prediction and Loss Calculation:** The model predicts a class probability for each image. The loss function then calculates the difference between these predictions and the true labels.
- **Backpropagation and Weight Update:** The calculated loss is used in a process called backpropagation to update the weights and biases of the model's layers.

- **Minimization Goal:** The goal of the optimizer (e.g., Adam used in your code) is to adjust these weights and biases in a way that minimizes the overall loss function.

CORRECTION OF PIXELETED IMAGES (DE-PIXELETION)

ALGORITHM:

Step 1: Import necessary modules.
Step 2: Define a Unet architecture with Encoder and Decoder layers for extracting feature from the image and enhancing it.
Step 3: Use transforms to resize the image and to transform into tensors.
Step 4: Give directories for pixelated images and corresponding High resolute images.
Step 5: Load the directories using the ImagePairDataset class.
Step 6: Forward pass the pixelated images in the network.
Step 7: Compute the loss between corrected output and the actual high resoluted image.
Step 8: Backpropogate to update weight and bias.
Step 9: Save the model.
Step 10: Test the model with an unseen image.

SOURCE CODE:

```
# Define the EnhancedUNet model
class EnhancedUNet(nn.Module):
    def __init__(self):
        super(EnhancedUNet, self).__init__()
        self.encoder = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(64, 64, kernel_size=3, padding=1),
            nn.ReLU(inplace=True)
        )
        self.middle = nn.Sequential(
            nn.Conv2d(64, 128, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(128, 128, kernel_size=3, padding=1),
            nn.ReLU(inplace=True)
        )
        self.decoder = nn.Sequential(
            nn.Conv2d(128, 64, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(64, 64, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(64, 3, kernel_size=3, padding=1)
        )

    def forward(self, x):
        x1 = self.encoder(x)
        x2 = self.middle(x1)
        x3 = self.decoder(x2)
        return x3
```

```
# Model, Loss, and Optimizer
model = EnhancedUNet().to(device)
criterion = nn.L1Loss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Training loop
num_epochs = 1000
for epoch in range(num_epochs):
    model.train()
    epoch_loss = 0
    for i, (pixelated_imgs, high_res_imgs) in enumerate(data_loader):
        pixelated_imgs = pixelated_imgs.to(device)
        high_res_imgs = high_res_imgs.to(device)

        optimizer.zero_grad()

        outputs = model(pixelated_imgs)

        loss = criterion(outputs, high_res_imgs)

        loss.backward()
        optimizer.step()

        epoch_loss += loss.item()

    print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {epoch_loss/len(data_loader)}')

# Save the trained model
torch.save(model.state_dict(), 'enhanced_unet_model.pth')
print("Model saved as 'enhanced_unet_model.pth'")
```

This is the code to rectify a pixeled image and produce its high-resolution version for the base provided by the Unet architecture; it produces an optimal result. The enhanced UNet model implemented is an extension of the traditional U-Net model, which is highly recognized as effective in the field of translational tasks from image to image.

• Encoder: This is composed of convolutional layers followed by a ReLU activation with the goal of extracting
features of an input image.

• Middle Layers: Additional convolutional layers that retain and further fine-tune the extracted features.

• Decoder: This comprises convolutional layers with ReLU activations, which aims at reconstructing the high-resolution images from the extracted features so that it retains pixelation details.

Its architecture ensures the propagation of information through the network while maintaining relevant spatial information that is important for restorations in image quality.

Dataset: It utilizes a custom dataset with pixelated images matched against their high-resolution counterparts. If needed, integrate data augmentation into the transforms function.

• Loss Function: It uses L1 loss, which is the mean absolute error between the model's output and the ground truth high-resolution image, on a pixel-by-pixel basis.

• Optimizer: Adam optimizer is used for efficient gradient-based optimization of model parameters.

• Training Loop: This module trains the model iteratively for the number of epochs specified—each comprising one batch of data fed through the network. It learns to decrease pixel image and high-resolution image differences through backpropagation and updates in its parameters.

The model is then tested on a test image after training. Usec Matplotlib to visually compare the depixelated output to its pixelated original input, thus it provides insight qualitatively into how good or bad the model is.

Sources of Data: Commonly used public datasets like DIV2K or BSDS500, or any others available on Kaggle that are related to image processing and computer vision, are used to train and test pixelated images derived from them.

This finally leads to an EnhancedUNet model, very promising in image depixelation, and could actually restore high-resolution details from pixelated inputs. Further work will be in the optimization of the model architecture by exploring additional loss functions and increasing the size of the dataset in order to ensure generalization and robustness.

OUTPUT: