Name: Kusuma Veerapalli Student id : 801360989

Super-Resolution Generative Adversarial Networks (SRGAN) Project Documentation

Step 1: Dataset Preparation

- Task: Collect and prepare the dataset for the SRGAN project.

Step 2: Identify the source of your dataset, whether it's from public sources, surveys, experiments, or other means.
Download, gather, or create the dataset, making sure it's relevant to your problem and properly stored in a structured format.
Data Preprocessing:
Inspect the dataset for data quality issues like missing values, duplicates, or outliers. Clean and handle these issues as necessary.
Step 3: Labelling Assign labels to the data if it's a supervised learning task. Labels represent the target variable or classes you aim to predict.
Data Splitting: Divide the dataset into training and testing subsets. The common split is 70% for training and 30% for testing. Ensure that the split is representative and preserves the distribution of data.

```
dataset_path = './dogs-vs-cats/train/train'

image_size = (128, 128)
sample_size = 1000
test_size = 0.3

sampled_files = random.sample(os.listdir(dataset_path), sample_size)

# Lists to store sampled images and labels
images = []
labels = []

# Load and preprocess the sampled images
for filename in sampled_files:
    if filename.startswith("cat"):
        label = 0  # "cat"
    else:
        label = 1  # "dog"

    # Load and resize the image
    img = load_img(os.path.join(dataset_path, filename), target_size=image_size)
    img = img_to_array(img) / 255.0  # Normalize pixel values to [0, 1]

    images.append(img)
    labels.append(label)

# Convert the lists to numpy arrays
X = np.array(images)
y = np.array(labels)

# Split the subset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=test_size, random_state=42)

# Display the number of samples in each split
```

Step 2: Build and Train the SRGAN

- Task: Build the SRGAN model and train it on the dataset.

1. Network Architecture:

SRGAN typically consists of two main components: a Generator and a Discriminator.

The Generator aims to produce high-resolution images from low-resolution inputs. It often uses convolutional neural networks (CNNs) with upsampling layers.

The Discriminator serves as an adversary to the Generator. It tries to distinguish between real high-resolution images and the ones generated by the Generator. This adversarial training improves the quality of generated images.

2. Data Preparation:

You need a dataset of images for training. These images should include pairs of low-resolution and high-resolution versions. The low-resolution images will be used as inputs, and the high-resolution images as ground truth for training the Generator.

Dataset preprocessing may involve resizing all images to a consistent low-resolution size (e.g., 32x32), which aligns with the input size of the Generator.

3. Loss Functions:

SRGAN employs several loss functions, including:

Perceptual Loss: A measure of how similar the high-resolution output from the Generator is to the ground truth. This is usually computed using a pretrained network like VGG.

Adversarial Loss: The loss used by the Discriminator to distinguish between real and generated images. The Generator aims to minimize this loss.

Content Loss: Encourages the generated image to have similar content to the ground truth.

Feature Loss: Measures the difference between the feature maps of the generated and real images.

4. Training Procedure:

SRGAN uses a GAN training procedure. It alternates between training the Generator and the Discriminator.

The Generator is trained to minimize the Perceptual Loss and Adversarial Loss. It generates high-resolution images from low-resolution inputs.

The Discriminator is trained to improve its ability to distinguish real images from generated ones.

The training process continues for a large number of epochs (e.g., 150 or more), with the aim of achieving better and better image quality over time.

Save the Generator's weights periodically during training, allowing you to resume training from specific checkpoints if needed.

5. Hyperparameter Tuning:

Experiment with various hyperparameters, such as learning rates, batch sizes, and architecture choices, to optimize the training process.

6. Evaluation:

After training, evaluate the performance of the SRGAN on a separate test dataset to ensure that it can effectively enhance image resolution.

7. Inference:

Use the trained SRGAN to upscale low-resolution images to high-resolution versions. This can be done by feeding the low-resolution image into the Generator.

Training an SRGAN is a computationally intensive process and often benefits from GPU acceleration. Additionally, you may need to fine-tune the architecture and parameters based on the specific requirements of your project and dataset. Once trained, the SRGAN can be a powerful tool for generating high-quality, super-resolved images.

```
Layer (type)                 Output Shape              Param #   Connected to
==================================================================================================
input_1 (InputLayer)         [(None, 128, 128, 3)]     0         []

conv1_pad (ZeroPadding2D)    (None, 134, 134, 3)       0         ['input_1[0][0]']

conv1_conv (Conv2D)          (None, 64, 64, 64)        9472      ['conv1_pad[0][0]']

conv1_bn (BatchNormalizati   (None, 64, 64, 64)        256       ['conv1_conv[0][0]']
on)

conv1_relu (Activation)      (None, 64, 64, 64)        0         ['conv1_bn[0][0]']

pool1_pad (ZeroPadding2D)    (None, 66, 66, 64)        0         ['conv1_relu[0][0]']

pool1_pool (MaxPooling2D)    (None, 32, 32, 64)        0         ['pool1_pad[0][0]']

conv2_block1_1_conv (Conv2   (None, 32, 32, 64)        4160      ['pool1_pool[0][0]']
D)

conv2_block1_1_bn (BatchNo   (None, 32, 32, 64)        256       ['conv2_block1_1_conv[0][0]']
rmalization)

conv2_block1_1_relu (Activ   (None, 32, 32, 64)        0         ['conv2_block1_1_bn[0][0]']
ation)

conv2_block1_2_conv (Conv2   (None, 32, 32, 64)        36928     ['conv2_block1_1_relu[0][0]']
D)

conv2_block1_2_bn (BatchNo   (None, 32, 32, 64)        256       ['conv2_block1_2_conv[0][0]']
rmalization)

conv2_block1_2_relu (Activ   (None, 32, 32, 64)        0         ['conv2_block1_2_bn[0][0]']
ation)

conv2_block1_0_conv (Conv2   (None, 32, 32, 256)       16640     ['pool1_pool[0][0]']
D)

conv2_block1_3_conv (Conv2   (None, 32, 32, 256)       16640     ['conv2_block1_2_relu[0][0]']
D)

conv2_block1_0_bn (BatchNo   (None, 32, 32, 256)       1024      ['conv2_block1_0_conv[0][0]']
rmalization)
```

Step 3: Show Examples of Scaled Images in Jupyter Notebook (JNB)

- Task: Visualize and display images upscaled from 32x32 to 128x128 using the SRGAN.

```python
import tensorflow as tf
from tensorflow.keras.layers import Input, Conv2D, BatchNormalization, LeakyReLU, UpSampling2D
from tensorflow.keras.models import Model

# Define the generator network for 32x32 images
def build_generator_32x32():
    input_lr = Input(shape=(32, 32, 3))

    # Initial convolution block
    x = Conv2D(64, 3, padding='same')(input_lr)
    x = LeakyReLU(0.2)(x)

    # Upsampling blocks
    x = Conv2D(64, 3, padding='same')(x)
    x = UpSampling2D(size=(2, 2))(x)  # Upsample to match 32x32 size
    x = LeakyReLU(0.2)(x)

    # Output layer
    x = Conv2D(3, 3, activation='tanh', padding='same')(x)  # Output size is 32x32 for low-resolutio

    return Model(input_lr, x)

# Build the 32x32 generator
generator_32x32 = build_generator_32x32()
generator_32x32.summary()
```

output

```
Model: "model_26"
```

Step 4: Utilize Images Generated by SRGAN to Train a New Model (Model B)
- Task: Use high-resolution images generated by SRGAN to create a new dataset for another task. Train a new model (Model B) for the specified task.

```
Step 4: Utilize Images Generated by SRGAN to Train a New Model (Model B)

import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers

num_classes = 10


# Define your Model B architecture (classification example)
model_b = keras.Sequential([
    layers.Input(shape=(128, 128, 3)),  # Adjust input shape based on your high-res images
    layers.Conv2D(32, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.Flatten(),
    layers.Dense(64, activation='relu'),
    layers.Dense(num_classes, activation='softmax')
])

# Compile the model
model_b.compile(optimizer='adam',
                loss='categorical_crossentropy',  # Adjust loss function for your task
                metrics=['accuracy'])

# Train the model using your high-resolution dataset
```

Step 5: Divide the Dataset into Training and Testing Sets
- Task: Split the dataset into a training set (70%) and a testing set (30%) for Classifier A and Model B.

```
# Split the dataset into training (70%) and testing (30%) sets
X_train_A, X_test_A, y_train_A, y_test_A = train_test_split(X_classifier_A, y_classifier_A, test_size=0.3, random_state=42)


# for model B
X_train_B, X_test_B, y_train_B, y_test_B = train_test_split(X_model_B, y_model_B, test_size=0.3, random_state=42)


]
```

Step 6: Apply Normalization and Image Transformation
- Task: Apply image transformations and normalisation to the dataset to prepare it for training.
-

```
# Image Augmentation
from keras.preprocessing.image import ImageDataGenerator

datagen = ImageDataGenerator(
    rotation_range=40,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,
    fill_mode='nearest'
)

# Fit the generator on your training data
datagen.fit(X_train_A_normalized)
```

```
# Resizing
from keras.preprocessing.image import img_to_array, array_to_img
from keras.preprocessing.image import load_img

target_size = (128, 128)
X_train_A_resized = np.array([img_to_array(load_img(img, target_size=target_size)) for img in X_train_A_normal
X_test_A_resized = np.array([img_to_array(load_img(img, target_size=target_size)) for img in X_test_A_normaliz
```

Normalization:

Normalization is the process of scaling pixel values in images to a standardized range. This ensures that all images have consistent pixel value distributions, making it easier for machine learning models to learn patterns and features from the data. Common techniques for normalization include:
Image Transformation:
Image transformations are operations that alter the appearance of images, either to augment the dataset or to improve model generalization. Common image transformations include:
Rotation:
Rotating images by a certain angle (e.g., 90 degrees) to introduce variations in orientation. This helps the model generalize better.
Flipping:
Horizontally or vertically flipping images to create mirror images. This can help the model learn from different viewpoints.
Cropping:
Removing parts of the image to focus on specific regions of interest. Cropping can improve the model's ability to recognize objects in different positions.
Brightness and Contrast Adjustments:
Altering the brightness and contrast of images to simulate variations in lighting conditions. This can make the model more robust to real-world lighting changes.

Step 7: Compare Model Performance
- Task: Train Classifier A and Model B on their respective datasets. Evaluate model performance using F1 score, accuracy, and AUC. Compare the performance of the two models for the specific tasks.

```python
# evaluating the perfomances
from sklearn.metrics import f1_score, accuracy_score, roc_auc_score

# Evaluate Classifier A
y_pred_A = classifier_A.predict(X_test_A_resized)
f1_A = f1_score(y_test_A, y_pred_A)
accuracy_A = accuracy_score(y_test_A, y_pred_A)
auc_A = roc_auc_score(y_test_A, y_pred_A)

# Evaluate Model B
y_pred_B = model_B.predict(X_test_B)
f1_B = f1_score(y_test_B, y_pred_B)
accuracy_B = accuracy_score(y_test_B, y_pred_B)
auc_B = roc_auc_score(y_test_B, y_pred_B)
```

Evaluate each model's performance on the testing dataset using the chosen evaluation metrics.
Here's a brief explanation of some common metrics:
Accuracy measures the proportion of correctly classified instances.
Precision quantifies the ratio of true positive predictions to the total predicted positives, useful for tasks where false positives are costly.
Recall (Sensitivity) calculates the ratio of true positives to the total actual positives, important for tasks where missing actual positives is costly.
F1 Score is the harmonic mean of precision and recall, balancing precision and recall.
AUC (Area Under the ROC Curve) measures the model's ability to distinguish between classes.
For regression tasks, MSE and MAE indicate the prediction error's magnitude.

Step 8: Save Models After Each N Epoch

- Task: Save models (Classifier A and Model B) at regular intervals during training using the ModelCheckpoint callback.

```python
checkpoint_dir = "model_checkpoints/"

import os
if not os.path.exists(checkpoint_dir):
    os.makedirs(checkpoint_dir)

classifier_checkpoint_filename = checkpoint_dir + "classifier_A_epoch_{epoch:02d}.h5"
model_b_checkpoint_filename = checkpoint_dir + "model_B_epoch_{epoch:02d}.h5"

# Define the ModelCheckpoint callbacks
```