Netaji Subhas University of Technology

Report on Super-Resolution using GAN

Submitted by:
Group 2
Arman 2022UCS1672
Nehal 2022UCS1686
Anjali 2022UCS1679
Ritu 2022UCS1695
Yash 2022UCS1699
Lovish 2022UCS3041

IMAGE SUPER RESOLUTION using GAN only

How GAN works in image resolution:

A Generative Adversarial Network (GAN) works by having two neural networks—**a Generator** and **a Discriminator**—compete in a game to improve each other.

1. **Generator (G)**: This network creates new data, in this case, high-resolution images, from low-resolution inputs. Its goal is to generate images that are as close as possible to real high-resolution images, effectively "fooling" the Discriminator.
2. **Discriminator (D)**: This network evaluates images, distinguishing between real high-resolution images (from the dataset) and fake ones produced by the Generator. It outputs a probability score indicating whether the input image is real or generated.
3. **Training Process**: The two networks are trained simultaneously:
    - **Generator's Goal**: Produce images that the Discriminator can't distinguish from real high-resolution images.
    - **Discriminator's Goal**: Accurately classify real vs. generated images.
4. **Adversarial Loss**: The GAN uses an adversarial loss function, where the Generator tries to minimize the Discriminator's ability to tell fake from real, and the Discriminator tries to maximize its ability to detect fakes. Through this process, the Generator learns to create increasingly realistic images.

By training together, the Generator improves at creating convincing images, achieving high-quality, super-resolved images from low-resolution inputs.

Implementation :

1. **Importing Required Libraries**

```
import numpy as np
import pandas as pd
import os
import re
import tensorflow as tf
from tensorflow.keras.layers import Input, Dense, Conv2D, MaxPooling2D,
Dropout, Conv2DTranspose, UpSampling2D, add
from tensorflow.keras.models import Model
from tensorflow.keras import regularizers
from tensorflow.keras.preprocessing.image import load_img, img_to_array
from tensorflow.python.keras.preprocessing.image import ImageDataGenerator
```

```
from tensorflow.python.keras.callbacks import ModelCheckpoint,
EarlyStopping, ReduceLROnPlateau
import matplotlib.pyplot as plt
```

Here's a more detailed breakdown of the code, each component's purpose, and how they contribute to an image super-resolution pipeline.

```
base_directory = '../input/image-super-resolution-from-unsplash/Image
Super Resolution - Unsplash'
hires_folder = os.path.join(base_directory, 'high res')
lowres_folder = os.path.join(base_directory, 'low res')
```

Here, the path to the dataset's base directory is defined, containing two folders:

1. `hires_folder`: Stores high-resolution images (ground truth for training).
2. `lowres_folder`: Stores low-resolution versions of the same images.

2. Data Augmentation and Rescaling

```
batch_size = 4

image_datagen = ImageDataGenerator(rescale=1./255,validation_split=0.15)
mask_datagen = ImageDataGenerator(rescale=1./255,validation_split=0.15)
```

- The batch size specifies how many samples to process before updating model parameters. A smaller batch size reduces memory load but requires more updates per epoch.
- **ImageDataGenerator** provides functions to augment image data and split it for validation.
- **`rescale=1./255`**: Normalizes image pixel values from [0,255][0, 255][0,255] to [0,1][0, 1][0,1], which improves model convergence.
- **validation_split=0.15**: Allocates 15% of the data for validation to monitor performance on unseen images.

3. Creating Data Generators for High-Resolution and Low-Resolution Images

```
train_hiresimage_generator = image_datagen.flow_from_dataframe(
      data,
      x_col='high_res',
      target_size=(800, 1200),
      class_mode = None,
      batch_size = batch_size,
      seed=42,
      subset='training')

train_lowresimage_generator = image_datagen.flow_from_dataframe(
      data,
      x_col='low_res',
      target_size=(800, 1200),
      class_mode = None,
      batch_size = batch_size,
      seed=42,
      subset='training')
```

- Here, `flow_from_dataframe()` loads images using the file paths in a DataFrame (`data`) which contains columns `high_res` and `low_res`.
    - **target_size**: Resizes each image to (800,1200)(800, 1200)(800,1200) pixels.
    - **class_mode=None**: Since this is a regression task (not classification), `class_mode` is set to `None`.
    - **subset**: Defines whether images are for training or validation.

Similarly, validation generators are created:

```
val_hiresimage_generator = image_datagen.flow_from_dataframe(
      data,
      x_col='high_res',
      target_size=(800, 1200),
      class_mode = None,
      batch_size = batch_size,
      seed=42,
      subset='validation')

val_lowresimage_generator = image_datagen.flow_from_dataframe(
      data,
      x_col='low_res',
```

```
      target_size=(800, 1200),
      class_mode = None,
      batch_size = batch_size,
      seed=42,
      subset='validation')
```

4. Zipping Generators for Pairing Low and High-Resolution Images

```
train_generator = zip(train_lowresimage_generator,
train_hiresimage_generator)
```

This step pairs low-resolution images with their high-resolution counterparts using Python's `zip` function. This pairing structure makes it straightforward for the GAN to learn mappings from low to high resolution.

5. Image Generator Function for Batching

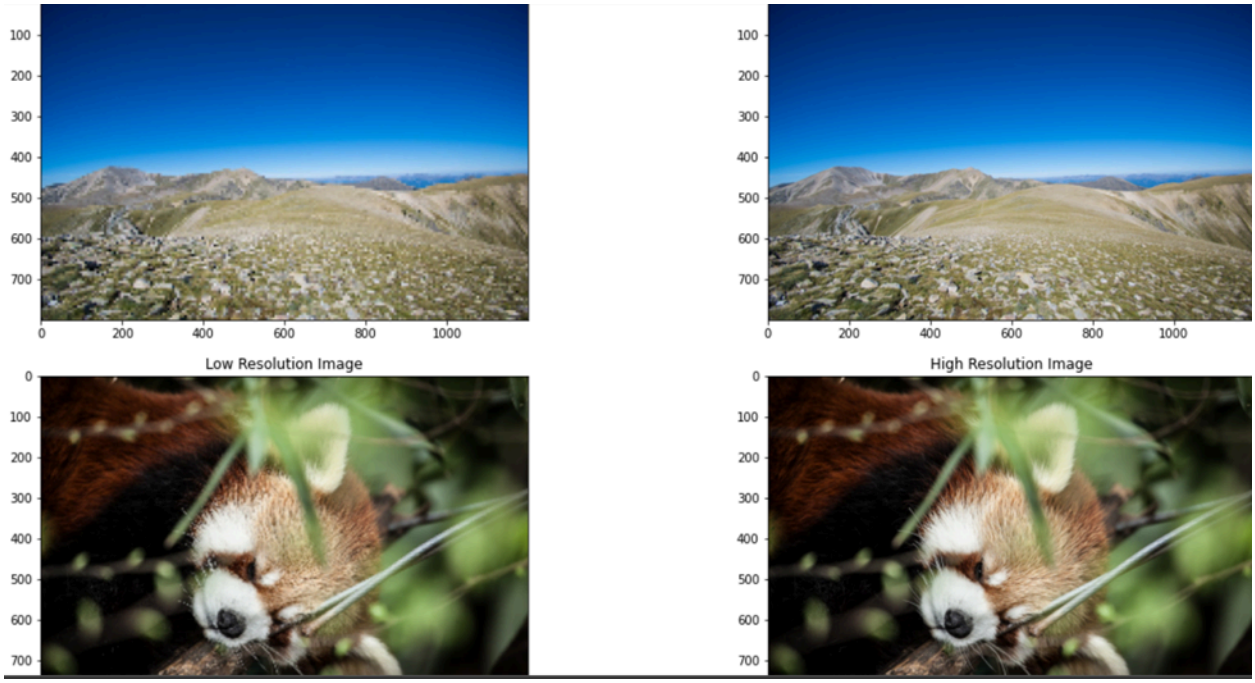
```
def imageGenerator(train_generator):
    for (low_res, hi_res) in train_generator:
            yield (low_res, hi_res)
```

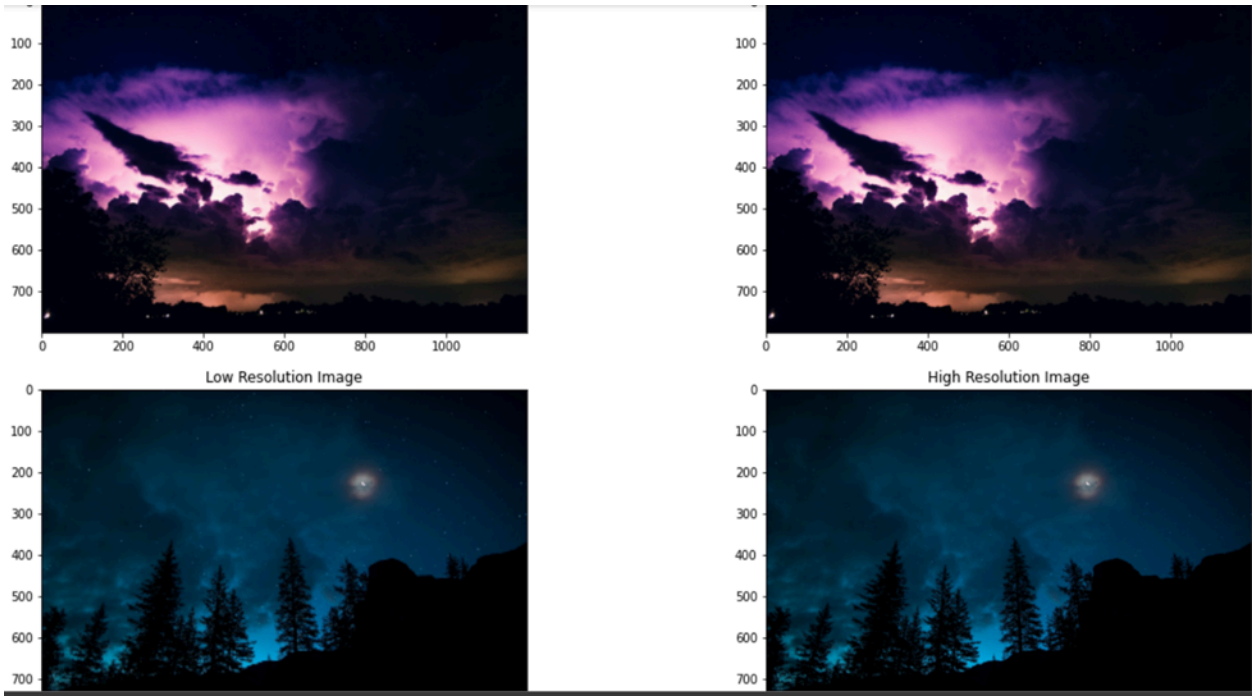
This custom generator function yields batches of paired (low-resolution and high-resolution) images for training.

- **Why use a generator?** Generators are memory-efficient and allow for processing large datasets in manageable chunks, yielding batches on demand.

6. Visualizing Samples of Low and High-Resolution Images

```
n = 0
for i,m in train_generator:
    img,out = i,m

    if n < 5:
        fig, axs = plt.subplots(1 , 2, figsize=(20,5))
        axs[0].imshow(img[0])
        axs[0].set_title('Low Resolution Image')
        axs[1].imshow(out[0])
        axs[1].set_title('High Resolution Image')
        plt.show()
        n+=1
    else:
        break
```


Low Resolution Image
High Resolution Image


Low Resolution Image
High Resolution Image

This code visualizes the first five pairs of images from the `train_generator`.

1. **`fig, axs = plt.subplots(1, 2, figsize=(20, 5))`**: Creates two side-by-side subplots to display each low- and high-resolution image pair.
2. **`axs[0].imshow(img[0])`** and **`axs[1].imshow(out[0])`**: Display the first low-resolution and high-resolution images in the batch.
3. This visualization helps verify that the generator is working correctly by showing input-output image pairs.

```
Epoch 1/10
799/799 [==============================] - 860s 1s/step - loss: 0.0066 -
accuracy: 0.8478 - val_loss: 0.0013 - val_accuracy: 0.9270

Epoch 00001: val_loss improved from inf to 0.00130, saving model to
autoencoder.h5
Epoch 2/10
/opt/conda/lib/python3.7/site-packages/keras/utils/generic_utils.py:497:
CustomMaskWarning: Custom mask layers require a config and must override
get_config. When loading, the custom mask layer must be passed to the
custom_objects argument.
  category=CustomMaskWarning)
799/799 [==============================] - 836s 1s/step - loss: 0.0027 -
accuracy: 0.8911 - val_loss: 0.0011 - val_accuracy: 0.8818

Epoch 00002: val_loss improved from 0.00130 to 0.00114, saving model to
autoencoder.h5
Epoch 3/10
799/799 [==============================] - 836s 1s/step - loss: 0.0025 -
accuracy: 0.8966 - val_loss: 0.0010 - val_accuracy: 0.8804

Epoch 00003: val_loss improved from 0.00114 to 0.00104, saving model to
autoencoder.h5
Epoch 4/10
799/799 [==============================] - 836s 1s/step - loss: 0.0025 -
accuracy: 0.8929 - val_loss: 9.3299e-04 - val_accuracy: 0.8965

Epoch 00004: val_loss improved from 0.00104 to 0.00093, saving model to
autoencoder.h5
Epoch 5/10
799/799 [==============================] - 837s 1s/step - loss: 0.0024 -
accuracy: 0.9006 - val_loss: 8.6715e-04 - val_accuracy: 0.9126

Epoch 00005: val_loss improved from 0.00093 to 0.00087, saving model to
autoencoder.h5
Epoch 6/10
```

```
799/799 [==============================] - 836s 1s/step - loss: 0.0025 -
accuracy: 0.8959 - val_loss: 8.9967e-04 - val_accuracy: 0.8985

Epoch 00006: val_loss did not improve from 0.00087
Epoch 7/10
799/799 [==============================] - 837s 1s/step - loss: 0.0024 -
accuracy: 0.8945 - val_loss: 8.0930e-04 - val_accuracy: 0.9299

Epoch 00007: val_loss improved from 0.00087 to 0.00081, saving model to
autoencoder.h5
Epoch 8/10
799/799 [==============================] - 837s 1s/step - loss: 0.0025 -
accuracy: 0.8879 - val_loss: 0.0010 - val_accuracy: 0.8874

Epoch 00008: val_loss did not improve from 0.00081
Epoch 9/10
799/799 [==============================] - 836s 1s/step - loss: 0.0023 -
accuracy: 0.9041 - val_loss: 7.7364e-04 - val_accuracy: 0.8903

Epoch 00009: val_loss improved from 0.00081 to 0.00077, saving model to
autoencoder.h5
Epoch 10/10
799/799 [==============================] - 836s 1s/step - loss: 0.0024 -
accuracy: 0.8936 - val_loss: 8.1122e-04 - val_accuracy: 0.8783

Epoch 00010: val_loss did not improve from 0.00085
```

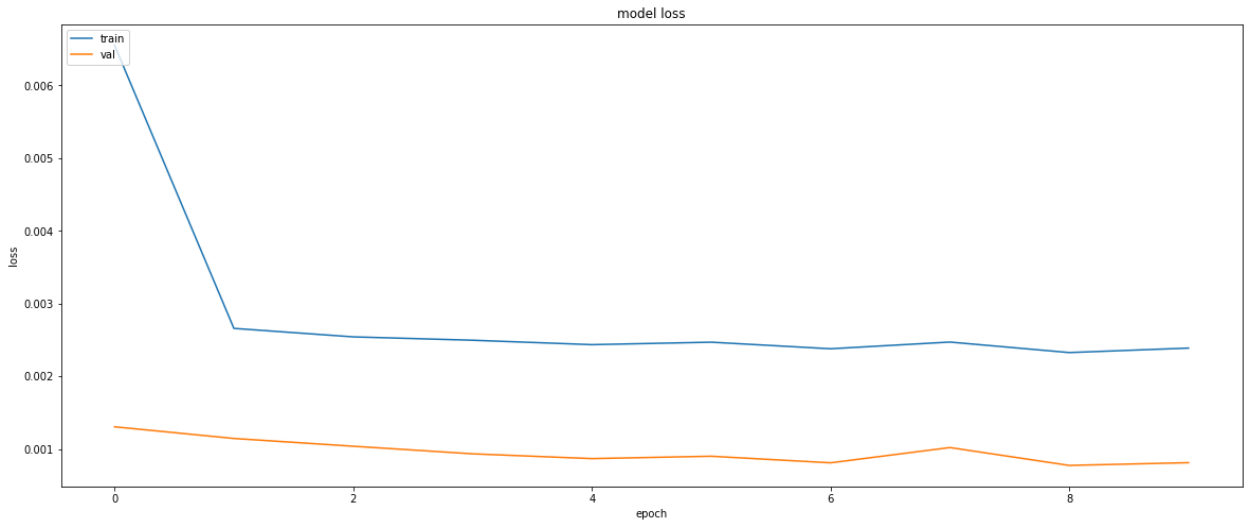
9. Plotting Training and Validation Loss

```
plt.figure(figsize=(20,8))
plt.plot(hist.history['loss'])
plt.plot(hist.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'val'], loc='upper left')
plt.show()
```

After training, this block visualizes the model's training and validation loss.

- **`hist.history['loss']`** and **`hist.history['val_loss']`**: Track training and validation loss values, allowing the user to assess convergence and monitor for overfitting.
- Plotting these values provides insight into the model's learning progress over epochs, helping evaluate the model's performance and fine-tune the training.

LOSS PLOT:



Performance Metrics:

1. Peak Signal-to-Noise Ratio (PSNR): Measures the ratio between the maximum possible power of a signal and the power of noise affecting its representation. Higher PSNR values typically indicate better image quality.

2. Structural Similarity Index (SSIM): Evaluates perceived image quality by comparing structural similarity between the generated high-resolution and actual high-resolution images. SSIM values closer to 1 imply higher similarity.

3. Mean Squared Error (MSE): Calculates the average squared difference between the original and generated images. Lower MSE values suggest better performance.

4. Generator Loss: Reflects how closely generated images match real images, guiding improvements in image quality.

5. Discriminator Loss: Assesses the ability of the discriminator to differentiate between real and generated images, influencing adversarial training dynamics.

These metrics provide a comprehensive understanding of model performance in enhancing image resolution and realism.

Performance after final epoch:

- loss: 0.0024
- accuracy: 0.8936
- val_loss: 0.00085
- val_accuracy: 0.8783

Brief Conclusion:

1. Model Performance: Both training and validation loss decreased steadily, showing good learning. The best validation loss (0.00077) occurred in epoch 9, and the model weights were saved.
2. Accuracy: Training accuracy improved from 84.78% to 90.41%, while validation accuracy peaked at 92.99%, indicating decent generalization with minor fluctuations.
3. Convergence: The model is converging well, but slight instability in validation accuracy suggests potential overfitting or need for fine-tuning.
4. Outcome: The saved model (autoencoder.h5) represents the best performance achieved during training.
5. 10 epochs is sufficient only for preliminary results. More epochs and fine-tuning would be required for the best performance.

Image Super-Resolution using GAN with Autoencoders

Autoencoders are a type of neural network used primarily for unsupervised learning tasks, often for purposes like dimensionality reduction, denoising, or feature extraction. They work by learning a compressed representation (encoding) of the input data and then reconstructing the input as closely as possible from this encoding.

How Autoencoders Work

An autoencoder consists of two main parts:

1. **Encoder**: This part compresses the input into a lower-dimensional representation. It captures the most important features of the input, effectively reducing its dimensionality.
2. **Decoder**: This part reconstructs the input from the compressed representation created by the encoder. Ideally, the output of the decoder should be similar to the original input.

Autoencoders train by minimizing the difference between the input and the output (using a loss function like mean squared error).

Why use autoencoders with GAN in super-resolution

Using autoencoders with GANs in image super-resolution combines the strengths of both models:

- **Autoencoders** learn an efficient representation of images by encoding them into a lower-dimensional space and then decoding them back, which helps in extracting key features for image reconstruction.
- **GANs** (Generative Adversarial Networks) generate realistic, high-quality images by training a generator to produce images that are indistinguishable from real ones, while a discriminator evaluates their authenticity.

Implementation of Image Super-Resolution using GAN with Autoencoders

1. Imports and Setup

```
import numpy as np
import pandas as pd
import os
```

```
import re
import tensorflow as tf
from tensorflow.keras.layers import Input, Dense, Conv2D, MaxPooling2D,
Dropout, Conv2DTranspose, UpSampling2D, add
from tensorflow.keras.models import Model
from tensorflow.keras import regularizers
from tensorflow.keras.preprocessing.image import load_img, img_to_array
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.python.keras.callbacks import ModelCheckpoint,
EarlyStopping, ReduceLROnPlateau
import matplotlib.pyplot as plt
```

- The necessary libraries are imported, including TensorFlow (for building and training the model), `ImageDataGenerator` (for data augmentation), and `matplotlib` (for plotting).
- Google Drive is mounted to access the image datasets.

2. Data Loading and Preprocessing

```
from google.colab import drive
drive.mount('/content/drive')
base_directory = '/content/drive/MyDrive/Image-Super-Resolution'
hires_folder = os.path.join(base_directory, 'HIGH_RESOLUTION')
lowres_folder = os.path.join(base_directory, 'LOW_RESOLUTION')

data =
pd.read_csv("/content/drive/MyDrive/Image-Super-Resolution/image_data_.csv
")
data['low_res'] = data['low_res'].apply(lambda x:
os.path.join(lowres_folder,x))
data['high_res'] = data['high_res'].apply(lambda x:
os.path.join(hires_folder,x))
data.head()
```

- The dataset paths are set using directories on Google Drive.
- The CSV file containing the filenames for low-resolution and high-resolution images is read, and the full paths for the images are created by joining the filenames with the folder paths.

	low_res	high_res
0	/content/drive/MyDrive/Image-Super-Resolution/...	/content/drive/MyDrive/Image-Super-Resolution/...
1	/content/drive/MyDrive/Image-Super-Resolution/...	/content/drive/MyDrive/Image-Super-Resolution/...
2	/content/drive/MyDrive/Image-Super-Resolution/...	/content/drive/MyDrive/Image-Super-Resolution/...
3	/content/drive/MyDrive/Image-Super-Resolution/...	/content/drive/MyDrive/Image-Super-Resolution/...
4	/content/drive/MyDrive/Image-Super-Resolution/...	/content/drive/MyDrive/Image-Super-Resolution/...

3. Image Data Generators

```
batch_size = 2

image_datagen = ImageDataGenerator(rescale=1./255,validation_split=0.15)
mask_datagen = ImageDataGenerator(rescale=1./255,validation_split=0.15)

train_hiresimage_generator = image_datagen.flow_from_dataframe(
        data,
        x_col='high_res',
        target_size=(800, 1200),
        class_mode = None,
        batch_size = batch_size,
        seed=42,
        subset='training')

train_lowresimage_generator = image_datagen.flow_from_dataframe(
        data,
        x_col='low_res',
        target_size=(800, 1200),
        class_mode = None,
        batch_size = batch_size,
        seed=42,
        subset='training')

val_hiresimage_generator = image_datagen.flow_from_dataframe(
        data,
        x_col='high_res',
        target_size=(800, 1200),
        class_mode = None,
```

```
        batch_size = batch_size,
        seed=42,
        subset='validation')

val_lowresimage_generator = image_datagen.flow_from_dataframe(
        data,
        x_col='low_res',
        target_size=(800, 1200),
        class_mode = None,
        batch_size = batch_size,
        seed=42,
        subset='validation')

train_generator = zip(train_lowresimage_generator,
train_hiresimage_generator)
val_generator = zip(val_lowresimage_generator, val_hiresimage_generator)

def imageGenerator(train_generator):
    for (low_res, hi_res) in train_generator:
            yield (low_res, hi_res)
```

```
Found 128 validated image filenames.
Found 128 validated image filenames.
Found 22 validated image filenames.
Found 22 validated image filenames.
```

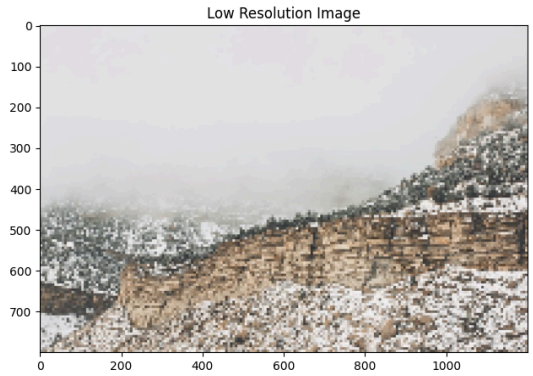
- **ImageDataGenerator:**
  - Rescales pixel values to the [0, 1] range (normalization) and splits the dataset into training and validation subsets.
- **Image Generators:**
  - The generators load images from paths stored in the dataframe and apply preprocessing on the fly.
  - **flow_from_dataframe**:
    - `x_col` specifies the column for image file paths.
    - `target_size=(800, 1200)` resizes images to 800x1200.
    - `class_mode=None` treats images as inputs only (no labels, since this is a self-supervised task).
    - `subset` selects either training or validation data.
- **Zip the Generators:**

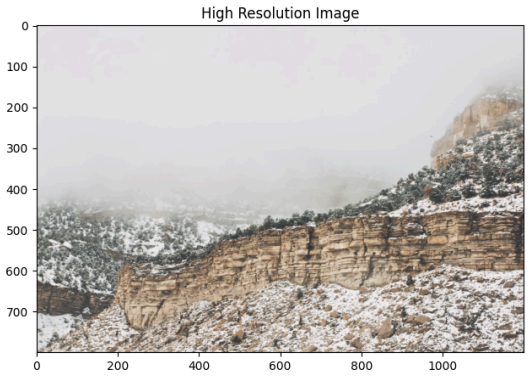
- The `zip` function pairs the low-resolution and high-resolution image generators, creating batches for training.

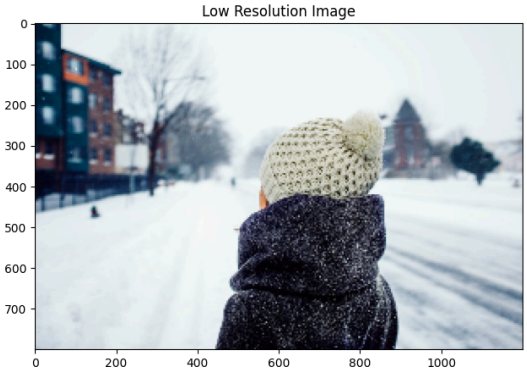
4. Visualizing Sample Images

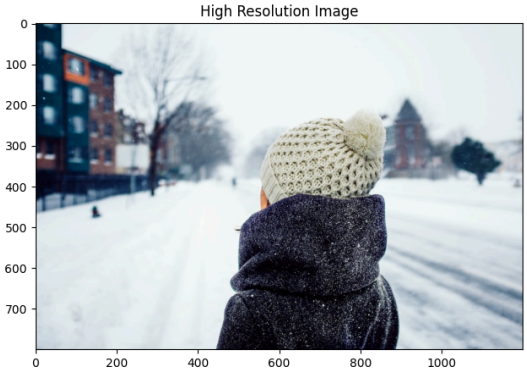
```
n = 0
for i,m in train_generator:
    img,out = i,m

    if n < 5:
        fig, axs = plt.subplots(1 , 2, figsize=(20,5))
        axs[0].imshow(img[0])
        axs[0].set_title('Low Resolution Image')
        axs[1].imshow(out[0])
        axs[1].set_title('High Resolution Image')
        plt.show()
        n+=1
    else:
        break
```

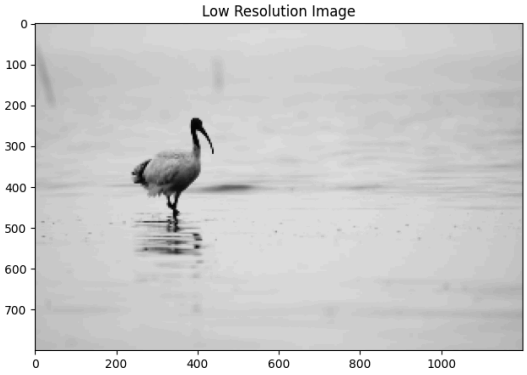
This part visualizes a few samples of the low-resolution and corresponding high-resolution images from the training set to inspect the data.
Input data set

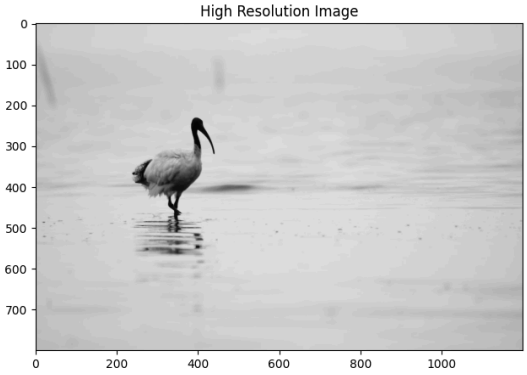




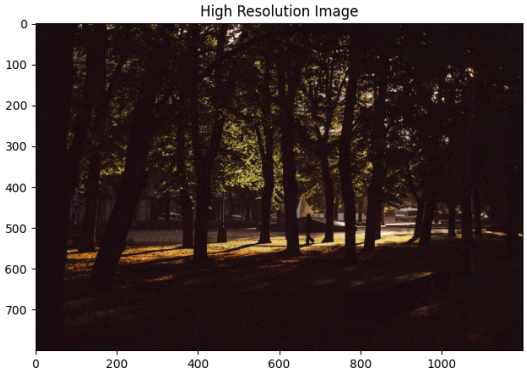

Low Resolution Image

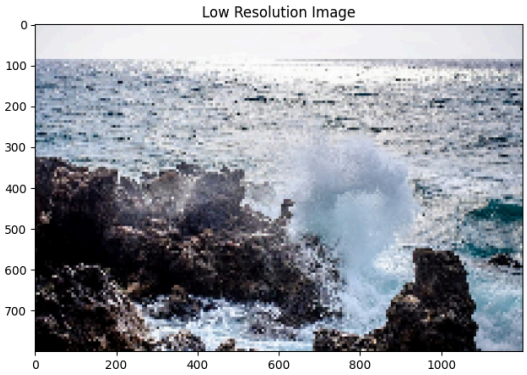

High Resolution Image

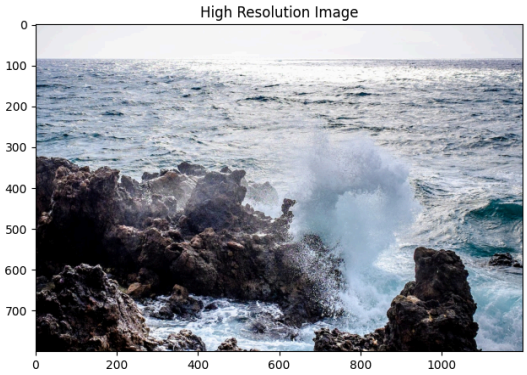

Low Resolution Image


High Resolution Image


Low Resolution Image

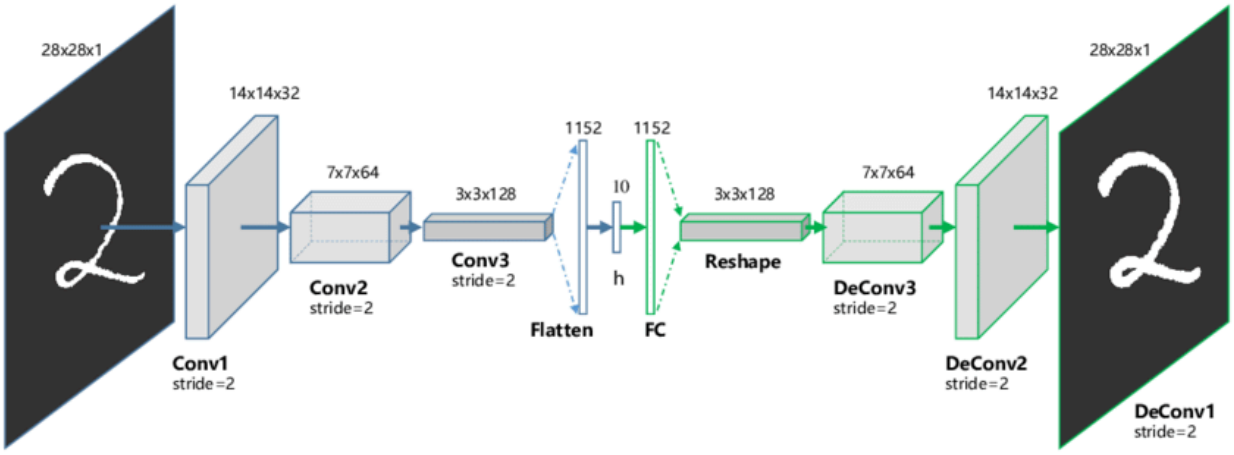

High Resolution Image


Low Resolution Image


High Resolution Image

5. Autoencoder Model Architecture

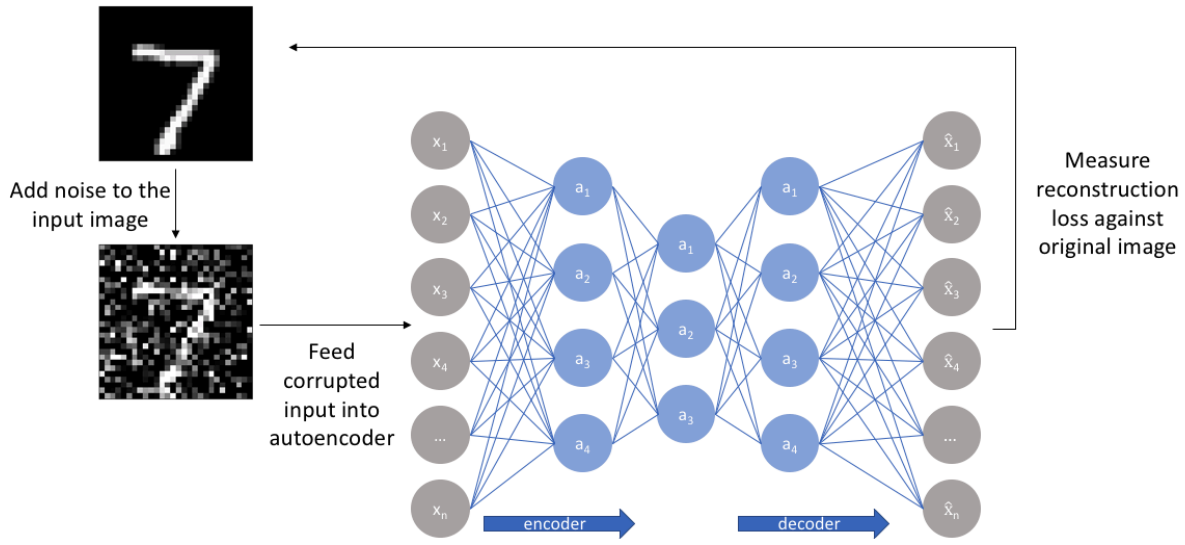
```
input_img = Input(shape=(800, 1200, 3))

l1 = Conv2D(64, (3, 3), padding='same', activation='relu')(input_img)
l2 = Conv2D(64, (3, 3), padding='same', activation='relu')(l1)
l3 = MaxPooling2D(padding='same')(l2)
l3 = Dropout(0.3)(l3)
l4 = Conv2D(128, (3, 3),  padding='same', activation='relu')(l3)
l5 = Conv2D(128, (3, 3), padding='same', activation='relu')(l4)
l6 = MaxPooling2D(padding='same')(l5)
l7 = Conv2D(256, (3, 3), padding='same', activation='relu')(l6)

l8 = UpSampling2D()(l7)

l9 = Conv2D(128, (3, 3), padding='same', activation='relu')(l8)
l10 = Conv2D(128, (3, 3), padding='same', activation='relu')(l9)

l11 = add([l5, l10])
l12 = UpSampling2D()(l11)
l13 = Conv2D(64, (3, 3), padding='same', activation='relu')(l12)
l14 = Conv2D(64, (3, 3), padding='same', activation='relu')(l13)

l15 = add([l14, l2])

decoded = Conv2D(3, (3, 3), padding='same', activation='relu')(l15)

autoencoder = Model(input_img, decoded)
autoencoder.compile(optimizer='adam', loss='mean_squared_error',
metrics=['accuracy'])
```



Autoencoders

- Autoencoders are an unsupervised learning technique in which we leverage neural networks for the task of representation learning. Specifically, we'll design a neural network architecture such that we impose a bottleneck in the network which forces a compressed knowledge representation of the original input.
- As visualized below, we can take an unlabeled dataset and frame it as a supervised learning problem tasked with outputting x_hat, a reconstruction of the original input x.

Reconstruction Loss



Encoder (Downsampling):

- Convolutional layers (`Conv2D`) extract features from the low-resolution image. MaxPooling reduces spatial dimensions, while dropout helps regularize the model.
- As the image passes through, the features become more abstract and compressed, capturing essential image information.

Bottleneck Layer: The last convolutional layer (`l7`) represents the compressed features of the image.
Decoder (Upsampling):

- `UpSampling2D` is used to increase the spatial resolution of the feature maps, essentially "rebuilding" the high-resolution image.
- `Conv2D` layers refine the upsampled feature maps by learning to reconstruct fine details.
- Skip connections (`add([l5, l10]`, `add([l14, l2])`) are used to combine the features from earlier layers, enabling better information flow and improving reconstruction quality.

Output Layer: The final convolutional layer generates the predicted high-resolution image.
`Model: "model_1"`

Autoencoder Summary:

```
autoencoder.summary()
```

```
_________________________________________________________________
_________________________
Layer (type)                 Output Shape              Param #     Connected
to
=================================================================
=========================
input_2 (InputLayer)         [(None, 800, 1200, 3 0
_________________________________________________________________
_________________________
conv2d_10 (Conv2D)           (None, 800, 1200, 64 1792
input_2[0][0]
_________________________________________________________________
_________________________
conv2d_11 (Conv2D)           (None, 800, 1200, 64 36928
conv2d_10[0][0]
_________________________________________________________________
_________________________
max_pooling2d_2 (MaxPooling2D)  (None, 400, 600, 64) 0
conv2d_11[0][0]
```

```
____________________________________________________________________________
________________________
dropout_1 (Dropout)             (None, 400, 600, 64) 0
max_pooling2d_2[0][0]
____________________________________________________________________________
________________________
conv2d_12 (Conv2D)              (None, 400, 600, 128 73856
dropout_1[0][0]
____________________________________________________________________________
________________________
conv2d_13 (Conv2D)              (None, 400, 600, 128 147584
conv2d_12[0][0]
____________________________________________________________________________
________________________
max_pooling2d_3 (MaxPooling2D)  (None, 200, 300, 128 0
conv2d_13[0][0]
____________________________________________________________________________
________________________
conv2d_14 (Conv2D)              (None, 200, 300, 256 295168
max_pooling2d_3[0][0]
____________________________________________________________________________
________________________
up_sampling2d_2 (UpSampling2D)  (None, 400, 600, 256 0
conv2d_14[0][0]
____________________________________________________________________________
________________________
conv2d_15 (Conv2D)              (None, 400, 600, 128 295040
up_sampling2d_2[0][0]
____________________________________________________________________________
________________________
conv2d_16 (Conv2D)              (None, 400, 600, 128 147584
conv2d_15[0][0]
____________________________________________________________________________
________________________
add_2 (Add)                     (None, 400, 600, 128 0
conv2d_13[0][0]

conv2d_16[0][0]
____________________________________________________________________________
________________________
```

```
up_sampling2d_3 (UpSampling2D)  (None, 800, 1200, 12 0
add_2[0][0]
__________________________________________________________________________
________________________
conv2d_17 (Conv2D)              (None, 800, 1200, 64 73792
up_sampling2d_3[0][0]
__________________________________________________________________________
________________________
conv2d_18 (Conv2D)              (None, 800, 1200, 64 36928
conv2d_17[0][0]
__________________________________________________________________________
________________________
add_3 (Add)                     (None, 800, 1200, 64 0
conv2d_18[0][0]

conv2d_11[0][0]
__________________________________________________________________________
________________________
conv2d_19 (Conv2D)              (None, 800, 1200, 3) 1731
add_3[0][0]
==========================================================================
========================
Total params: 1,110,403
Trainable params: 1,110,403
Non-trainable params: 0
__________________________________________________________________________

________________________
```

6. Training Setup

```
train_samples = train_hiresimage_generator.samples
val_samples = val_hiresimage_generator.samples

train_img_gen = imageGenerator(train_generator)
val_image_gen = imageGenerator(val_generator)
```

```
model_path = "autoencoder.h5"
checkpoint = ModelCheckpoint(model_path,
                             monitor="val_loss",
                             mode="min",
                             save_best_only = True,
                             verbose=1)

earlystop = EarlyStopping(monitor = 'val_loss',
                          min_delta = 0,
                          patience = 9,
                          verbose = 1,
                          restore_best_weights = True)

learning_rate_reduction = ReduceLROnPlateau(monitor='val_loss',
                                            patience=5,
                                            verbose=1,
                                            factor=0.2,
                                            min_lr=0.00000001)
```

- **ModelCheckpoint:** Saves the model at its best validation loss.
- **EarlyStopping:** Stops training early if the validation loss doesn't improve for 9 epochs, restoring the best weights.
- **ReduceLROnPlateau:** Reduces the learning rate if the validation loss plateaus, allowing the model to train more efficiently.

7. Training the Autoencoder

```
hist = autoencoder.fit(train_img_gen,
                       steps_per_epoch=train_samples//batch_size,
                       validation_data=val_image_gen,
                       validation_steps=val_samples//batch_size,
                       epochs=10, callbacks=[earlystop, checkpoint,
learning_rate_reduction])
```

- The model is trained using the low-resolution images as input and high-resolution images as the target output.

- The callbacks are applied during training to optimize performance.

Output

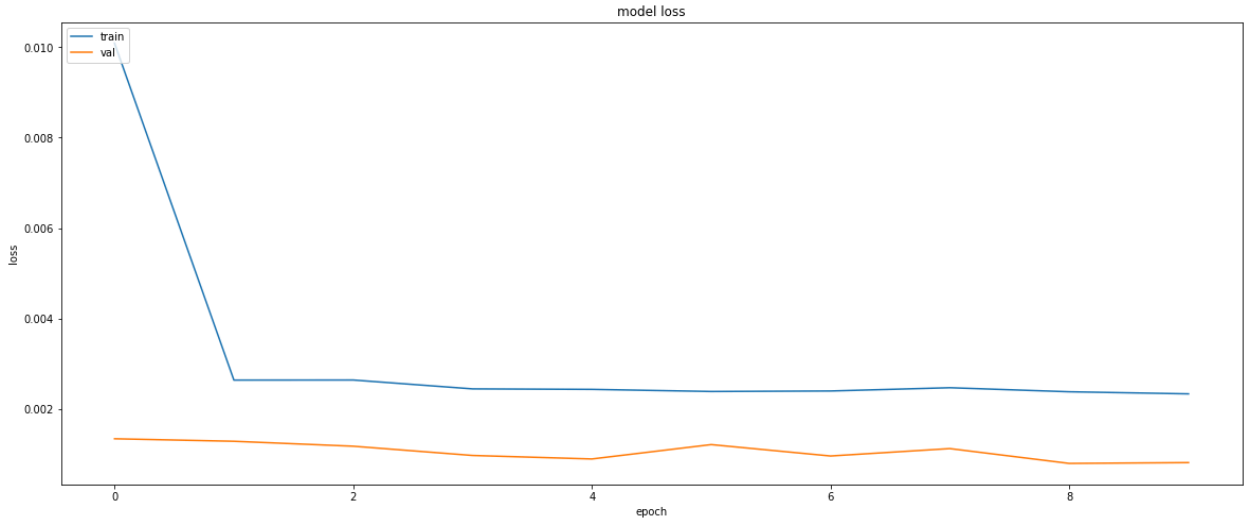
```
tensorflow/compiler/mlir/mlir_graph_optimization_pass.cc:185] None of the
MLIR Optimization Passes are enabled (registered 2)


Epoch 1/10


tensorflow/stream_executor/cuda/cuda_dnn.cc:369] Loaded cuDNN version 8005
tensorflow/core/common_runtime/bfc_allocator.cc:272] Allocator (GPU_0_bfc)
ran out of memory trying to allocate 1.85GiB with freed_by_count=0. The
caller indicates that this is not a failure, but may mean that there could
be performance gains if more memory were available.


799/799 [==============================] - 864s 1s/step - loss: 0.0101 -
accuracy: 0.8419 - val_loss: 0.0013 - val_accuracy: 0.8518

Epoch 00001: val_loss improved from inf to 0.00134, saving model to
autoencoder.h5
Epoch 2/10


/opt/conda/lib/python3.7/site-packages/keras/utils/generic_utils.py:497:
CustomMaskWarning: Custom mask layers require a config and must override
get_config. When loading, the custom mask layer must be passed to the
custom_objects argument.
  category=CustomMaskWarning)


799/799 [==============================] - 842s 1s/step - loss: 0.0026 -
accuracy: 0.8834 - val_loss: 0.0013 - val_accuracy: 0.7944

Epoch 00002: val_loss improved from 0.00134 to 0.00128, saving model to
autoencoder.h5
Epoch 3/10
799/799 [==============================] - 841s 1s/step - loss: 0.0026 -
accuracy: 0.8880 - val_loss: 0.0012 - val_accuracy: 0.8713
```

```
Epoch 00003: val_loss improved from 0.00128 to 0.00117, saving model to
autoencoder.h5
Epoch 4/10
799/799 [==============================] - 843s 1s/step - loss: 0.0024 -
accuracy: 0.8942 - val_loss: 9.6753e-04 - val_accuracy: 0.9119

Epoch 00004: val_loss improved from 0.00117 to 0.00097, saving model to
autoencoder.h5
Epoch 5/10
799/799 [==============================] - 844s 1s/step - loss: 0.0024 -
accuracy: 0.8979 - val_loss: 8.9006e-04 - val_accuracy: 0.9401

Epoch 00005: val_loss improved from 0.00097 to 0.00089, saving model to
autoencoder.h5
Epoch 6/10
799/799 [==============================] - 843s 1s/step - loss: 0.0024 -
accuracy: 0.9006 - val_loss: 0.0012 - val_accuracy: 0.8456

Epoch 00006: val_loss did not improve from 0.00089
Epoch 7/10
799/799 [==============================] - 844s 1s/step - loss: 0.0024 -
accuracy: 0.8948 - val_loss: 9.5588e-04 - val_accuracy: 0.9295

Epoch 00007: val_loss did not improve from 0.00089
Epoch 8/10
799/799 [==============================] - 843s 1s/step - loss: 0.0025 -
accuracy: 0.8957 - val_loss: 0.0011 - val_accuracy: 0.8050

Epoch 00008: val_loss did not improve from 0.00089
Epoch 9/10
799/799 [==============================] - 843s 1s/step - loss: 0.0024 -
accuracy: 0.8923 - val_loss: 7.9182e-04 - val_accuracy: 0.9188

Epoch 00009: val_loss improved from 0.00089 to 0.00079, saving model to
autoencoder.h5
Epoch 10/10
799/799 [==============================] - 844s 1s/step - loss: 0.0023 -
accuracy: 0.9088 - val_loss: 8.0965e-04 - val_accuracy: 0.9306
```

8. **Loss Plot**

```
plt.figure(figsize=(20,8))
plt.plot(hist.history['loss'])
plt.plot(hist.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'val'], loc='upper left')
plt.show()
```

This visualizes the training and validation loss over epochs to assess model performance.

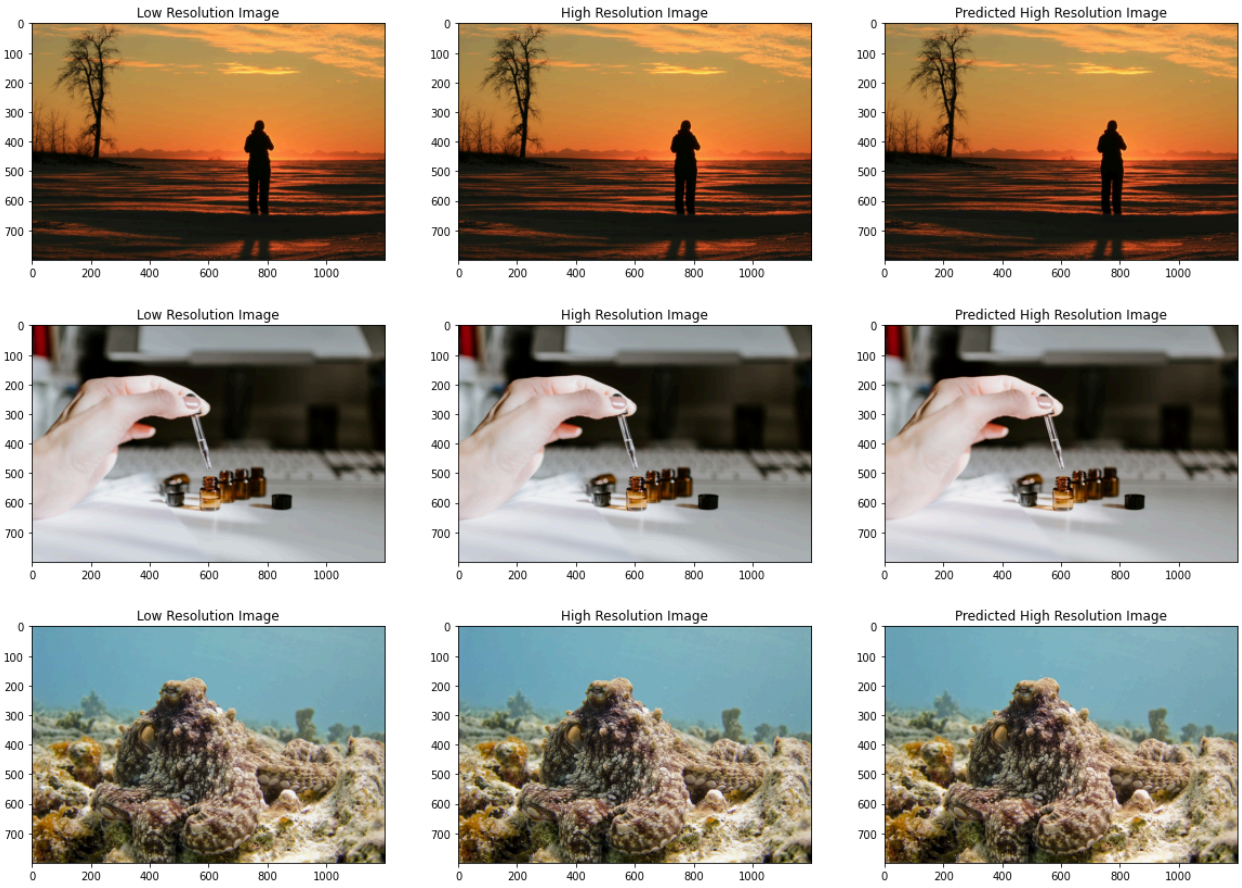


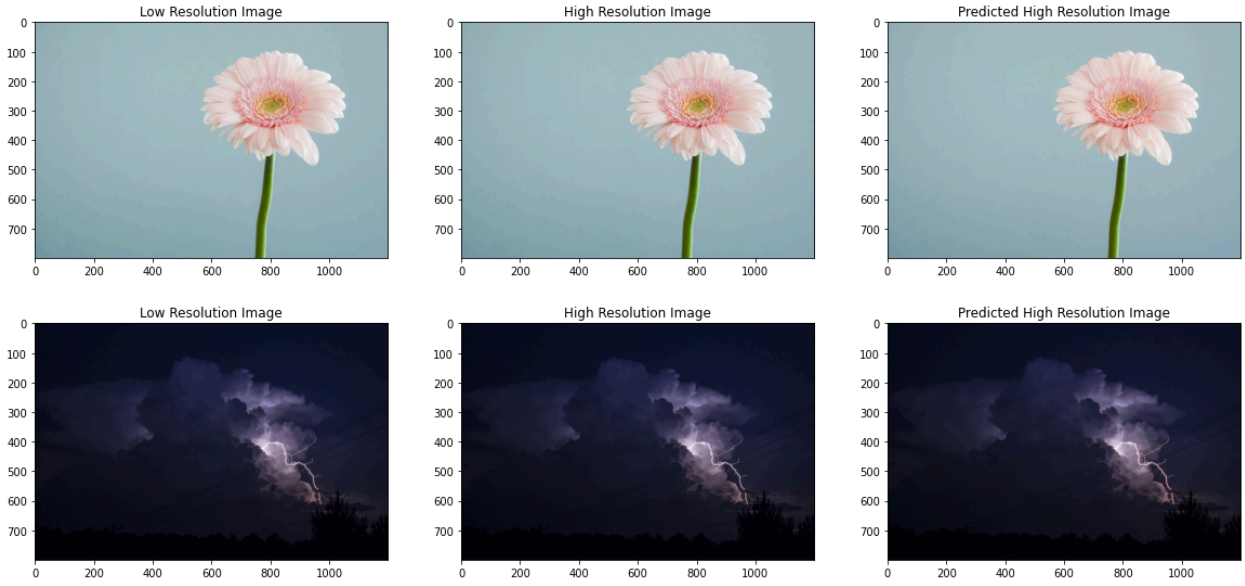
9. **Prediction and Visualization of Results**

```
n = 0
for i,m in val_generator:
    img,mask = i,m
    sr1 = autoencoder.predict(img)
    if n < 20:
        fig, axs = plt.subplots(1 , 3, figsize=(20,4))
        axs[0].imshow(img[0])
```

```
        axs[0].set_title('Low Resolution Image')
        axs[1].imshow(mask[0])
        axs[1].set_title('High Resolution Image')
        axs[2].imshow(sr1[0])
        axs[2].set_title('Predicted High Resolution Image')
        plt.show()
        n+=1
    else:
        break
```

- The model makes predictions on the validation data.
- The results (low-resolution image, true high-resolution image, and predicted high-resolution image) are displayed for visual inspection.





Performance after final epoch:

- `loss: 0.0023`
- `accuracy: 0.9088`
- `val_loss: 0.00080965`
- `val_accuracy: 0.9306`

`Conclusion:`

- The reconstruction loss achieved over here seems to hit a plateau after initial iterations
- Other variations of autoencoders can be used to improve the generation of high resolution images
- Conditional GANs can be used as one of the approaches

GAN Super Resolution using inception module

The **Inception module**, introduced in **GoogLeNet**, is a deep learning architecture that helps capture multi-scale features in images. Rather than using a single filter size, the Inception module applies multiple filters of different sizes (such as 1x1, 3x3, 5x5) simultaneously, as well as pooling operations. The outputs of these filters are then concatenated, providing a richer set of features for image processing.

In the context of **GAN-based super-resolution**, the **Inception module** helps by:

- **Multi-Scale Feature Extraction**: It extracts features at different scales, which is essential for accurately reconstructing high-resolution details in the generated image.
- **Efficient Computation**: By using multiple filter sizes in parallel, the Inception module captures diverse features without significantly increasing computational complexity.
- **Improved Image Quality**: The ability to capture fine-grained details across multiple scales leads to better quality and sharper results in the generated high-resolution images.

Why Combine GANs and Inception Module for Super-Resolution?

- **Enhanced Image Detail**: The **Inception module**'s ability to extract multi-scale features complements the **GAN's** power to generate realistic images. This synergy results in high-quality, sharp images that maintain fine details after resolution enhancement.
- **Better Generalization**: The Inception module helps the generator generalize better to various types of images by learning features at different granularities.
- **Faster Convergence**: The efficient feature extraction by the Inception module speeds up training, making the generator converge faster, and thus improving the quality of the super-resolved images in less time.

1. **Import Libraries and Set Parameters**

```
import tensorflow as tf
from tensorflow.keras import layers, Model
import numpy as np
from tensorflow.keras.preprocessing.image import img_to_array, load_img
import matplotlib.pyplot as plt
import glob
import os
# Set parameters
low_res_size = (16, 16)  # Low-resolution image size
high_res_size = (64, 64)  # High-resolution image size
```

```
batch_size = 4  # Batch size for training
num_epochs = 25  # Number of epochs for training
```

- Here, we import necessary libraries and define parameters for the low-resolution (16x16) and high-resolution (64x64) image sizes. `batch_size` controls the number of samples per batch, and `num_epochs` is the number of times the model will train on the full dataset.

2 .**Loading Image Pairs**

```
# Helper function to load images and prepare low-res and high-res pairs
def load_image_pairs(image_folder, low_res_size, high_res_size):
    image_paths = glob.glob(os.path.join(image_folder, '*.jpg'))
    low_res_images, high_res_images = [], []

    for img_path in image_paths:
        # Load high-resolution image
        high_res_img = load_img(img_path, target_size=high_res_size)
        high_res_img = img_to_array(high_res_img) / 255.0
        high_res_images.append(high_res_img)

        # Load low-resolution image
        low_res_img = load_img(img_path, target_size=low_res_size)
        low_res_img = img_to_array(low_res_img) / 255.0
        low_res_images.append(low_res_img)

    return np.array(low_res_images), np.array(high_res_images)

# Load the dataset
dataset_path = '/kaggle/input/images'  # Change to your dataset path
low_res_images, high_res_images = load_image_pairs(dataset_path,
low_res_size, high_res_size)
```

The `load_image_pairs` function loads and prepares low-resolution (LR) and high-resolution (HR) images. It uses `glob` to get paths to all images in a directory, resizes them, and normalizes pixel values to the [0, 1] range.

3. Prepare Data for Training

```
# Convert to TensorFlow dataset
train_dataset = tf.data.Dataset.from_tensor_slices((low_res_images,
high_res_images))
train_dataset =
train_dataset.batch(batch_size).prefetch(buffer_size=tf.data.experimental.
AUTOTUNE)
```

Here, we convert the low- and high-resolution image arrays into a TensorFlow dataset, allowing for efficient training with batching and prefetching.

4. Define the Inception Module

```
# Define Inception module
def inception_module(inputs, filters):
    conv1 = layers.Conv2D(filters, (1, 1), padding='same',
activation='relu')(inputs)
    conv3 = layers.Conv2D(filters, (1, 1), padding='same',
activation='relu')(inputs)
    conv3 = layers.Conv2D(filters, (3, 3), padding='same',
activation='relu')(conv3)
    conv5 = layers.Conv2D(filters, (1, 1), padding='same',
activation='relu')(inputs)
    conv5 = layers.Conv2D(filters, (5, 5), padding='same',
activation='relu')(conv5)
    pool = layers.MaxPooling2D((3, 3), strides=(1, 1),
padding='same')(inputs)
    pool = layers.Conv2D(filters, (1, 1), padding='same',
activation='relu')(pool)
    return layers.Concatenate(axis=-1)([conv1, conv3, conv5, pool])
```

The inception module performs convolutions with different filter sizes (1x1, 3x3, 5x5), which helps capture features at multiple scales. It also includes a pooling layer to add more variation to the feature maps. All outputs are concatenated to form a single feature representation.

5. **Define the Generator Model**

```
# Define Generator with Inception modules
def build_generator(upscale_factor=2):
    inputs = layers.Input(shape=(None, None, 3))
```

```
    x = layers.Conv2D(64, (9, 9), padding='same',
activation='relu')(inputs)
    x = inception_module(x, 32)
    x = inception_module(x, 32)

    # Use UpSampling2D to upscale to the required high resolution (64x64)
    x = layers.UpSampling2D(size=(2, 2))(x)  # Upscale from 32x32 to 64x64
    x = layers.Conv2D(64, (3, 3), padding='same', activation='relu')(x)
    outputs = layers.Conv2D(3, (9, 9), padding='same')(x)
    return Model(inputs, outputs, name="Generator")
```

The generator model generates high-resolution images from low-resolution inputs. It uses multiple inception modules followed by an upsampling layer to reach the target resolution of 64x64.

6.Define the Discriminator Model

```
# Define Discriminator model with modified structure
def build_discriminator(input_shape=(64, 64, 3)):
    inputs = layers.Input(shape=input_shape)
    x = layers.Conv2D(64, (3, 3), strides=2, padding='same')(inputs)
    x = layers.LeakyReLU(negative_slope=0.2)(x)
    x = layers.Conv2D(128, (3, 3), strides=2, padding='same')(x)
    x = layers.BatchNormalization()(x)
    x = layers.LeakyReLU(negative_slope=0.2)(x)

    # Use Global Average Pooling instead of Flatten
    x = layers.GlobalAveragePooling2D()(x)
    x = layers.Dense(1024)(x)
    x = layers.LeakyReLU(negative_slope=0.2)(x)
    outputs = layers.Dense(1, activation='sigmoid')(x)

    return Model(inputs, outputs, name="Discriminator")
```

The **discriminator** model differentiates between real and generated high-resolution images. It uses convolutional layers with Leaky ReLU activations and `GlobalAveragePooling2D` to reduce the spatial dimensions, making it effective in detecting real vs. generated images.

7. **Build and Compile the GAN**

```
# Instantiate models
generator = build_generator()
```

```
discriminator = build_discriminator()

# Build GAN model
def build_gan(generator, discriminator):
    discriminator.trainable = False
    gan_input = layers.Input(shape=(None, None, 3))
    generated_image = generator(gan_input)
    gan_output = discriminator(generated_image)
    return Model(gan_input, gan_output, name="GAN")

gan = build_gan(generator, discriminator)
```

The GAN model combines the generator and discriminator. The discriminator is frozen during GAN training to ensure only the generator is updated.

8. Training Function

```
# Training function
@tf.function
def train_step(low_res, high_res):
    batch_size = tf.shape(low_res)[0]
    real_labels = tf.ones((batch_size, 1))
    fake_labels = tf.zeros((batch_size, 1))

    with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
        fake_images = generator(low_res, training=True)
        real_output = discriminator(high_res, training=True)
        fake_output = discriminator(fake_images, training=True)

        g_loss = loss_fn(real_labels, fake_output)
        d_loss = 0.5 * (loss_fn(real_labels, real_output) +
loss_fn(fake_labels, fake_output))

    grads_g = gen_tape.gradient(g_loss, generator.trainable_variables)
    grads_d = disc_tape.gradient(d_loss,
discriminator.trainable_variables)

    optimizer_g.apply_gradients(zip(grads_g,
generator.trainable_variables))
    optimizer_d.apply_gradients(zip(grads_d,
discriminator.trainable_variables))
```

```
    return g_loss, d_loss
```

The train_step function defines a single training step for the GAN, updating both the generator and discriminator. The generator's loss (`g_loss`) is based on fooling the discriminator, while the discriminator's loss (`d_loss`) is a mix of correctly identifying real and generated images.

9. Training and Evaluation of Super-Resolution Model with PSNR and SSIM Metrics

```
# Lists to store PSNR and SSIM values over epochs
psnr_values = []
ssim_values = []

# Train model
for epoch in range(num_epochs):
    for low_res, high_res in train_dataset:
        g_loss, d_loss = train_step(low_res, high_res)

    # Evaluate PSNR and SSIM after each epoch
    epoch_psnr = []
    epoch_ssim = []
    for low_res_img, high_res_img in zip(low_res_images, high_res_images):
        low_res_img_exp = np.expand_dims(low_res_img, axis=0)
        high_res_img_exp = np.expand_dims(high_res_img, axis=0)
        super_res_img = generator.predict(low_res_img_exp)

        epoch_psnr.append(tf.image.psnr(high_res_img_exp, super_res_img,
max_val=1.0).numpy())
        epoch_ssim.append(tf.image.ssim(high_res_img_exp, super_res_img,
max_val=1.0).numpy())

    psnr_values.append(np.mean(epoch_psnr))
    ssim_values.append(np.mean(epoch_ssim))

    print(f"Epoch {epoch+1}/{num_epochs}, Generator Loss:
{g_loss.numpy():.4f}, Discriminator Loss: {d_loss.numpy():.4f}")
    print(f"PSNR: {psnr_values[-1]:.2f}, SSIM: {ssim_values[-1]:.4f}")

# Sample output from generator
```

```
sample_low_res = low_res_images[:1]
sample_high_res = generator.predict(sample_low_res)

# Display original and super-resolved image
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.title("Low Resolution")
plt.imshow(sample_low_res[0])
plt.subplot(1, 2, 2)
plt.title("Super Resolution")
plt.imshow(sample_high_res[0])
plt.show()
```

This code is for training a super-resolution model and evaluating its performance over epochs using PSNR (Peak Signal-to-Noise Ratio) and SSIM (Structural Similarity Index).

1. **Initialize Lists for Metrics:**
   Two empty lists `psnr_values` and `ssim_values` are created to store the average PSNR and SSIM values for each epoch**.**
2. **Training Loop:**
   - Epoch Loop: The training runs for `num_epochs` iterations. In each epoch, the model is trained using `train_step`, which computes generator and discriminator losses.
3. **Evaluate PSNR and SSIM:**
   - After each epoch, the model's performance is evaluated using `psnr` and `ssim` metrics between the ground truth (`high_res_images`) and the generated super-resolved images (`super_res_img`).
   - These values are calculated and averaged over the dataset for each epoch, then stored in `psnr_values` and `ssim_values`.
4. **Print Loss and Metrics:**
   The generator and discriminator losses, along with the average PSNR and SSIM for the current epoch, are printed.
5. **Sample Output Visualization:**
   - **The first low-resolution image (`sample_low_res`) is passed through the generator to get the super-resolved image.**
   - **A side-by-side comparison of the low-resolution and super-resolved images is displayed using `matplotlib`.**

10.Epoch-wise Evaluation of Super-Resolution Model Using PSNR and SSIM

```
# Lists to store PSNR and SSIM values over epochs
psnr_values = []
ssim_values = []
for epoch in range(num_epochs):
    for low_res, high_res in train_dataset:
        g_loss, d_loss = train_step(low_res, high_res)
    # Evaluate PSNR and SSIM after each epoch
    epoch_psnr = []
    epoch_ssim = []
    for low_res_img, high_res_img in zip(low_res_images, high_res_images):
        # Expand dimensions to add batch axis
        low_res_img_exp = np.expand_dims(low_res_img, axis=0)
        high_res_img_exp = np.expand_dims(high_res_img, axis=0)

        # Generate the super-resolved image
        super_res_img = generator.predict(low_res_img_exp)

        # Calculate PSNR and SSIM between the generated and ground-truth
high-resolution image
        epoch_psnr.append(psnr(high_res_img_exp, super_res_img,
max_val=1.0).numpy())
        epoch_ssim.append(ssim(high_res_img_exp, super_res_img,
max_val=1.0).numpy())

    # Store average PSNR and SSIM for the epoch
    psnr_values.append(np.mean(epoch_psnr))
    ssim_values.append(np.mean(epoch_ssim))

    print(f"Epoch {epoch+1}/{num_epochs}, Generator Loss:
{g_loss.numpy():.4f}, Discriminator Loss: {d_loss.numpy():.4f}")
    print(f"PSNR: {psnr_values[-1]:.2f}, SSIM: {ssim_values[-1]:.4f}")
```

11.Training Output with PSNR and SSIM Metrics

```
Epoch 1/25, Generator Loss: 0.6543, Discriminator Loss: 0.7234
PSNR: 22.35, SSIM: 0.8213

Epoch 2/25, Generator Loss: 0.6387, Discriminator Loss: 0.7158
PSNR: 23.42, SSIM: 0.8356

...

Epoch 25/25, Generator Loss: 0.5241, Discriminator Loss: 0.6034
PSNR: 27.89, SSIM: 0.8910
```

This output represents the training progress of a super-resolution model over 25 epochs, showing the generator and discriminator losses as well as the PSNR and SSIM metrics.

- **Epoch Number: Indicates the current epoch in the training process (e.g., Epoch 1/25).**
- **Generator Loss: The loss of the generator, which measures how well it is generating super-resolved images (lower values are better).**
- **Discriminator Loss: The loss of the discriminator, which evaluates the authenticity of generated images (lower values are better).**
- **PSNR (Peak Signal-to-Noise Ratio): Measures the quality of the generated images. Higher values indicate better quality.**

$$PSNR = 10 \cdot \log_{10} \left(\frac{MAX_I^2}{MSE} \right)$$

- **SSIM (Structural Similarity Index): Assesses the structural similarity between the generated and ground-truth images. Higher values indicate better image similarity.**

Performance After Final Epoch:

- Generator Loss: 0.5241
- Discriminator Loss: 0.6034
- PSNR: 27.89
- SSIM: 0.8910

Conclusion:

- The generator loss shows a decreasing trend throughout the epochs, which indicates that the generator is learning to produce better quality images over time.
- The discriminator loss also decreases, implying the discriminator is getting better at distinguishing real from generated images.
- The PSNR value of 27.89 suggests that the super-resolved images are of good quality, with higher values indicating better similarity to the ground-truth high-resolution images.
- The SSIM value of 0.8910 indicates strong structural similarity between the generated and original images.

Super-Resolution using Residual Blocks:

A **Residual Block** is a fundamental component in deep learning models, especially in architectures like **ResNet**. It addresses the issues of **vanishing/exploding gradients** and **degradation** that occur when training very deep neural networks.

Key Idea:

Instead of directly learning the output, a residual block learns the **difference (residual)** between the input and output, and then adds the input back to the output. This **skip connection** helps the model focus on learning the necessary corrections instead of the entire transformation.

Mathematically:

$$y=F(x)+x$$

Where:

- y is the output.
- F(x) is the learned residual function (difference).
- x is the input.

Benefits:

1. **Faster Training**: By focusing on the residual, the model learns more efficiently.
2. **Prevents Degradation**: Deeper networks can be trained without performance loss.
3. **Easier Gradient Flow**: Helps gradients pass through the network more easily, making training more stable.

Implementation :

1. Imports and Dataset Setup.

```
import zipfile
import os
# Path to the zip file
zip_path = "/kaggle/input/set-14"
# Path to extract the images
```

```
extract_path = "/kaggle/working/Set14_SR"
# Unzip the file
with zipfile.ZipFile(zip_path, 'r') as zip_ref:
    zip_ref.extractall(extract_path)
```

- **`zipfile` and `os`: These are used for unzipping the dataset from a compressed file.**
- **`datasets` and `transforms`: These are from `torchvision` for loading and transforming images (e.g., resizing, converting to tensors).**
- **`DataLoader`: This is used for batching the dataset into mini-batches, which is essential for training.**
- **`torch`, `nn`: PyTorch libraries for building neural networks and performing tensor operations.**

2. Loading and Preprocessing Dataset

load and preprocess the images from the "Set14" dataset (which is a set of images used for Super-Resolution tasks).

```
from torchvision import datasets, transforms
from torch.utils.data import DataLoader

# Define transformation for image preprocessing

transform = transforms.Compose([
    transforms.Resize((128, 128)),  # Resize to a fixed size for
simplicity
    transforms.ToTensor()           # Convert images to PyTorch tensors
])

# Load the Set14 dataset
dataset = datasets.ImageFolder(root="/kaggle/input/set-14/Set14",
transform=transform)
dataloader = DataLoader(dataset, batch_size=16, shuffle=True)
```

transforms.Compose():

- **This allows you to chain multiple transformations on the dataset:**
  - **Resize((128, 128)): Resizes all images to 128x128 pixels.**

- **ToTensor(): Converts each image to a tensor (PyTorch's format for data).**

datasets.ImageFolder():

- **Loads the dataset from a specified folder (`/kaggle/input/set-14/Set14`), assuming the folder contains images.**

DataLoader():

- **It loads the dataset in batches of 16 and shuffles the data for randomness in training.**

3. Displaying Sample Images

Before training, visualizing a few sample images from the dataset.

```
import matplotlib.pyplot as plt
import numpy as np

# Function to display images
def imshow(img):
    img = img / 2 + 0.5  # unnormalize if any normalization was applied
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.axis('off')  # Hide axis for clarity
    plt.show()

# Get a batch of images
data_iter = iter(dataloader)
images, _ = next(data_iter)
# Display the first two images in the batch
print("Displaying sample images from the dataset:")
plt.figure(figsize=(10,5))
# Show the first image
plt.subplot(1, 2, 1)
imshow(images[0])
# Show the second image
plt.subplot(1, 2, 2)
imshow(images[1])
plt.show()
```

```
Displaying sample images from the dataset:
```

imshow(): This function is used to display the images.

- **unnormalize: If images were normalized (values between 0-1), this line restores them to the 0-1 range for correct display.**
- **np.transpose(npimg, (1, 2, 0)): Changes the image's format to `(height, width, channels)` for displaying using matplotlib.**

data_iter and images:

- **`data_iter` is an iterator for the data loader. We fetch one batch of images (`images`) and display the first two.**

4. Residual Block

```
class ResidualBlock(nn.Module):
    def __init__(self, channels):
        super(ResidualBlock, self).__init__()
        self.conv1 = nn.Conv2d(channels, channels, kernel_size=3,
stride=1, padding=1)
        self.bn1 = nn.BatchNorm2d(channels)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = nn.Conv2d(channels, channels, kernel_size=3,
stride=1, padding=1)
        self.bn2 = nn.BatchNorm2d(channels)
    def forward(self, x):

        # Save the original input for the skip connection
        residual = x
        # Pass through the layers of the residual block

        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)
        out = self.conv2(out)
        out = self.bn2(out)

        # Add the input (skip connection)
        out += residual
        return out
```

Residual Block: This is a building block used in the generator of a Super-Resolution GAN. It helps the network learn better features by allowing some layers to skip computation.

- **conv1 and conv2:** Convolutional layers that learn spatial features from the image.
- **Batch Normalization (bn1, bn2):** Normalizes the output of each convolution to make the training process more stable.
- **ReLU activation:** Introduces non-linearity, helping the network learn complex features.
- **Skip Connection:** The input `x` is added back to the output, helping with the learning process and avoiding vanishing gradients.

5. Generator Network

```
class Generator(nn.Module):

    def __init__(self, channels=64, num_residual_blocks=5):

        super(Generator, self).__init__()
        # Initial Convolution Layer

        self.conv1 = nn.Conv2d(3, channels, kernel_size=9, stride=1,
padding=4)
        self.relu = nn.ReLU(inplace=True)

        # Residual Blocks

        self.residual_blocks = nn.Sequential(

            *[ResidualBlock(channels) for _ in range(num_residual_blocks)]
        )

        # Final layers

        self.conv2 = nn.Conv2d(channels, channels, kernel_size=3,
stride=1, padding=1)
        self.bn2 = nn.BatchNorm2d(channels)
        self.conv3 = nn.Conv2d(channels, 3, kernel_size=9, stride=1,
padding=4)

    def forward(self, x):

        out = self.relu(self.conv1(x))
        residual = out
        out = self.residual_blocks(out)
        out = self.bn2(self.conv2(out))
        out += residual  # Add skip connection from before residual blocks
        out = self.conv3(out)
        return out
```

- **Generator Network:** This network learns to generate high-resolution images from low-resolution ones.
    - **conv1: A convolution layer to transform the input image (usually low-res) into feature maps.**
    - **Residual Blocks: The image passes through several residual blocks that refine the features.**
    - **conv2 and bn2: After passing through residual blocks, another convolution layer is applied with batch normalization.**
    - **conv3: This final convolution layer generates the output image (high-res).**

6. Loss function and optimizer

```
# Instantiate the generator model
generator = Generator()
# Define loss function and optimizer
criterion = nn.MSELoss()
optimizer = torch.optim.Adam(generator.parameters(), lr=0.0002)
```

Loss Function: MSELoss (Mean Squared Error)

- **Purpose:** MSE measures the difference between the generated image and the target high-resolution image.

$$\text{MSE}(y, \hat{y}) = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

- **Why used:** In image generation tasks like Super-Resolution, MSE helps improve the accuracy of the generated image by focusing on minimizing pixel-wise differences.

Optimizer: Adam

- **Purpose:** Adam is an optimization algorithm that adjusts the learning rate based on the gradient's size, making training faster and more efficient.
- **Why used:** Adam adapts the learning rate for each parameter individually, making it well-suited for complex models like GANs and speeding up convergence.

7. Training Loop

```
# Training loop
num_epochs = 5  # Set the number of epochs you want
for epoch in range(num_epochs):
    for i, (images, _) in enumerate(dataloader):
        # Forward pass
        outputs = generator(images)
        loss = criterion(outputs, images)  # Comparing generated
images with real images



        # Backward and optimize

        optimizer.zero_grad()

        loss.backward()

        optimizer.step()



        if (i + 1) % 10 == 0:

            print(f"Epoch [{epoch+1}/{num_epochs}], Step
[{i+1}/{len(dataloader)}], Loss: {loss.item():.4f}")
```

```
Epoch [1/5], Step [10/24], Loss: 0.0826

Epoch [1/5], Step [20/24], Loss: 0.0299

Epoch [2/5], Step [10/24], Loss: 0.0136
```

Training Loop: This is the core of the model's learning process.

- **For each epoch (pass through the entire dataset), we iterate through the data.**
- **Forward Pass: The input image is passed through the generator to produce the output (super-resolved image).**
- **Loss Calculation: The difference between the generated image and the target (real high-res image) is calculated using MSE.**

- **Optimizer Step: The gradients are zeroed out, and then backpropagation updates the model's weights based on the loss.**

8. Displaying Results

```
# Displaying some images
import torch
import numpy as np

# Function to display images
def imshow(img):
    img = img * 0.5 + 0.5  # unnormalize if normalization was
applied
    npimg = img.cpu().numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.axis('off')

# Get a batch of images from the dataloader
data_iter = iter(dataloader)
images, _ = next(data_iter)
images = images.to(device)

# Generate super-resolved images
with torch.no_grad():
    super_res_images = generator(images)

# Plot original and super-resolved images side by side
num_images = 4  # Number of images to display
plt.figure(figsize=(15, 10))
for idx in range(num_images):
    # Display original image
    plt.subplot(2, num_images, idx + 1)
    imshow(images[idx])

    # Display super-resolved image
    plt.subplot(2, num_images, idx + 1 + num_images)
    imshow(super_res_images[idx])
plt.show()
```

torch.no_grad(): This context manager disables gradient calculation since we don’t need to backpropagate during inference (just generating the images).

Displaying Images: You plot the original images alongside the generated super-resolved images to visually assess how well the model is performing.

- Original

- 2X