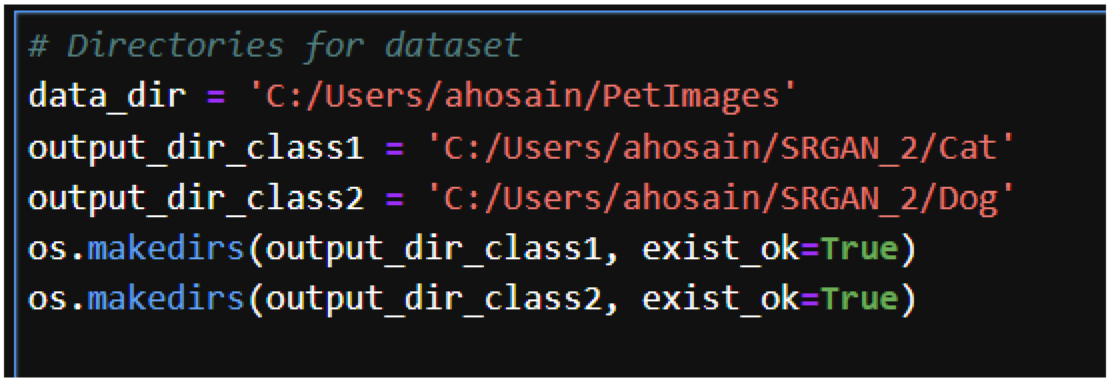
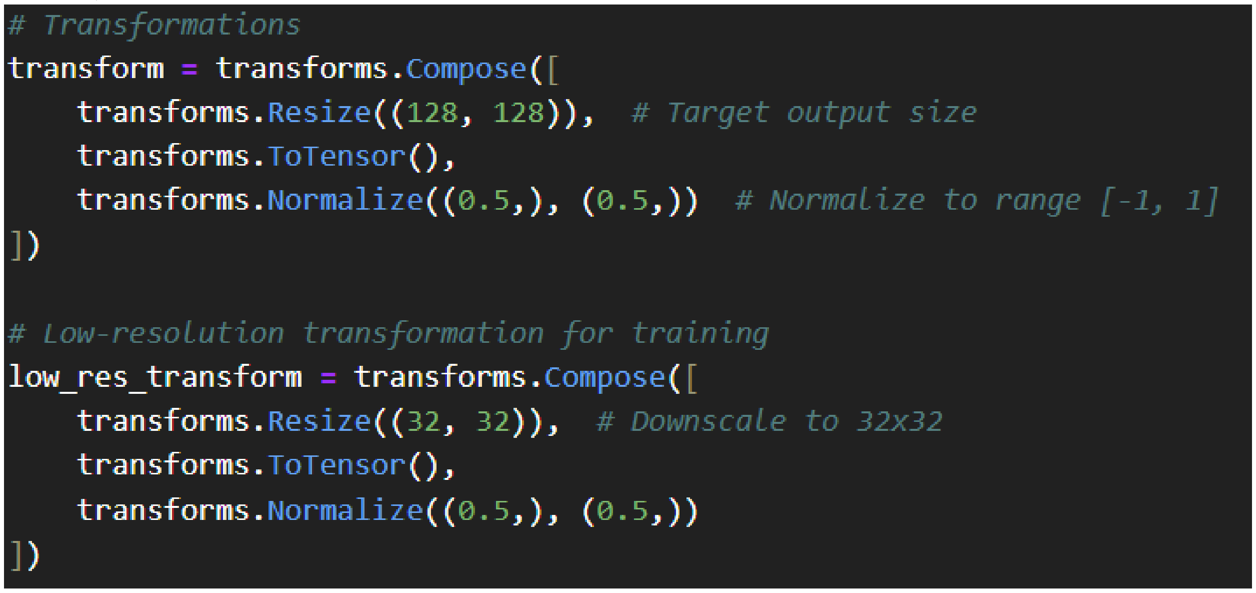
SRGAN:

1. Setting path to image directories



```python
# Directories for dataset
data_dir = 'C:/Users/ahosain/PetImages'
output_dir_class1 = 'C:/Users/ahosain/SRGAN_2/Cat'
output_dir_class2 = 'C:/Users/ahosain/SRGAN_2/Dog'
os.makedirs(output_dir_class1, exist_ok=True)
os.makedirs(output_dir_class2, exist_ok=True)
```

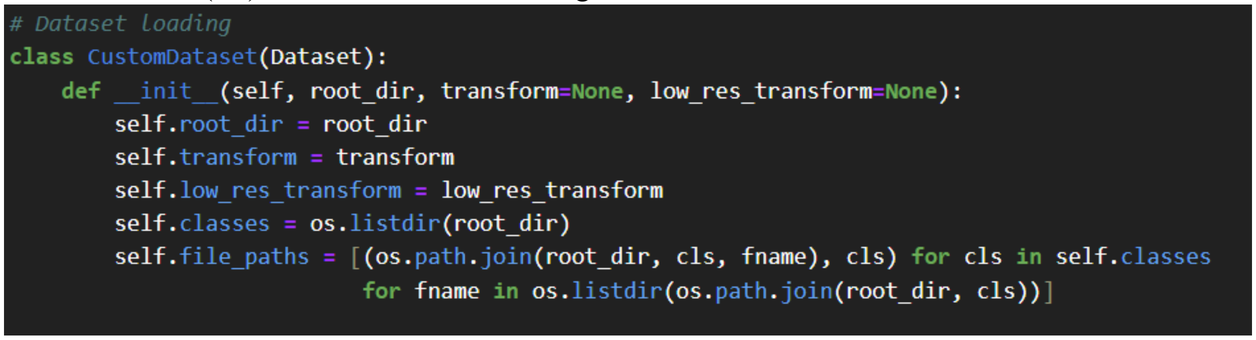
2. transform: For high-resolution images, we resize to 128x128, convert to tensors, and normalize.
3. low_res_transform: For low-resolution images, we downscale to 32x32, convert to tensors, and normalize.



```python
# Transformations
transform = transforms.Compose([
    transforms.Resize((128, 128)),  # Target output size
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))  # Normalize to range [-1, 1]
])

# Low-resolution transformation for training
low_res_transform = transforms.Compose([
    transforms.Resize((32, 32)),  # Downscale to 32x32
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])
```

4. root_dir: The directory containing the dataset. Each class should have its own subfolder within this directory.
5. transform and low_res_transform: Transformation functions for high-resolution and low-resolution images. These are applied to the images to create HR and LR versions.
6. self.classes: Lists all subdirectories (classes) inside root_dir. Each subdirectory represents a different class.
7. self.file_paths: This creates a list of tuples, each containing:

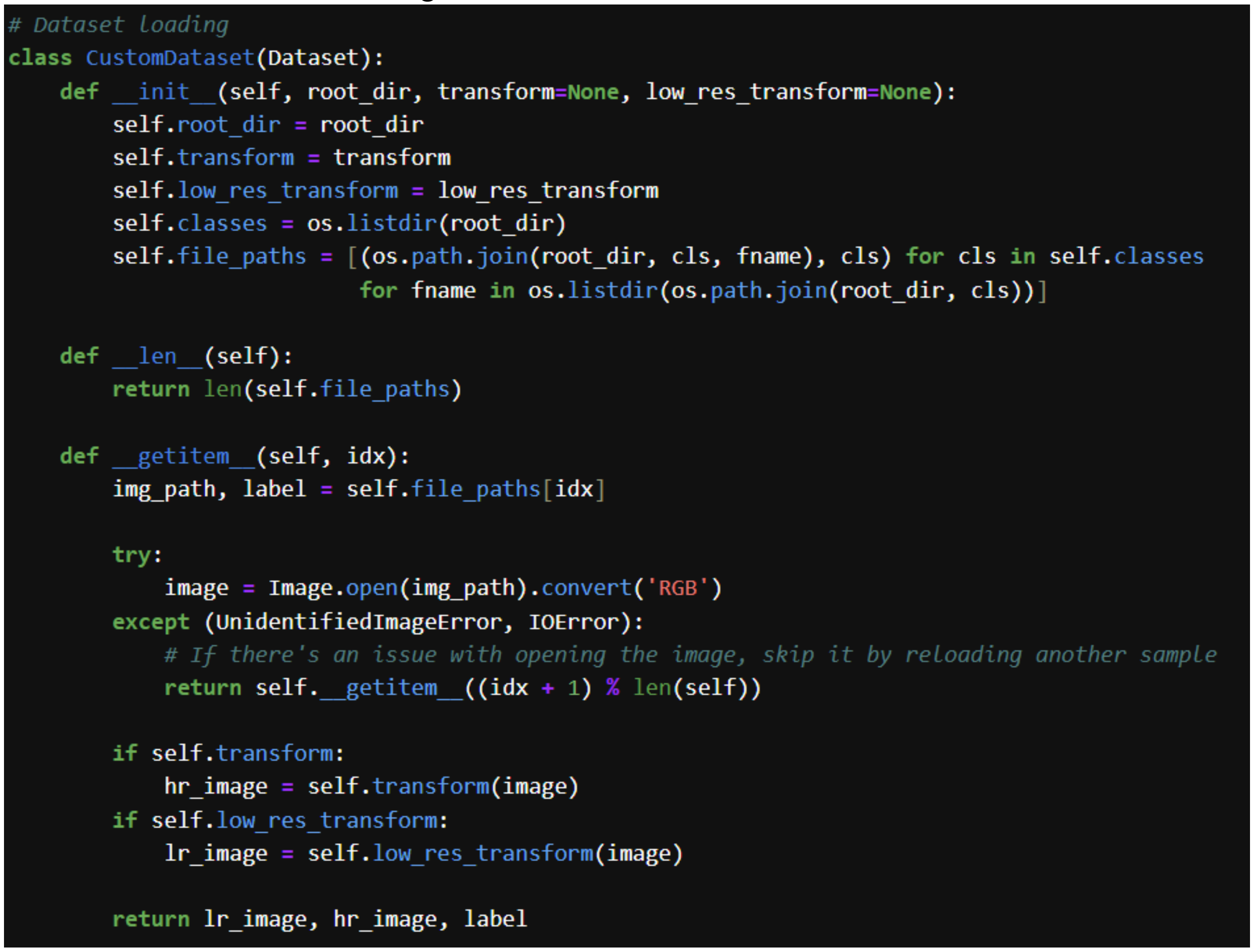
8. The full path to an image (os.path.join(root_dir, cls, fname)), constructed by combining the root directory, class folder, and filename.
9. The class label (cls) associated with that image.



```python
# Dataset loading
class CustomDataset(Dataset):
    def __init__(self, root_dir, transform=None, low_res_transform=None):
        self.root_dir = root_dir
        self.transform = transform
        self.low_res_transform = low_res_transform
        self.classes = os.listdir(root_dir)
        self.file_paths = [(os.path.join(root_dir, cls, fname), cls) for cls in self.classes
                           for fname in os.listdir(os.path.join(root_dir, cls))]
```

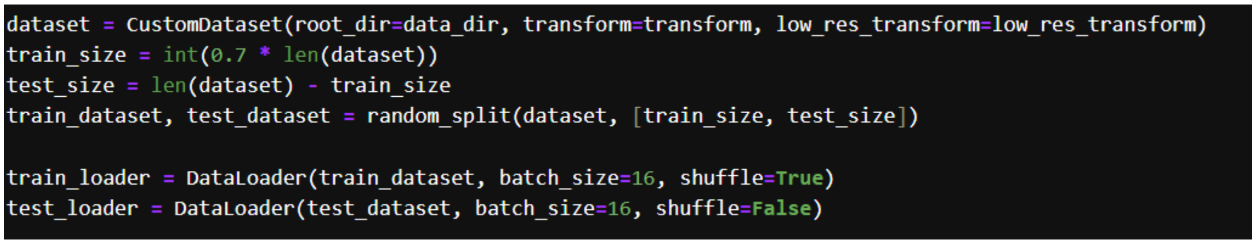
10. img_path and label are taken from self.file_paths[idx], corresponding to the index provided.
11. hr_image: The high-resolution version is created by applying self.transform to the loaded image.
12. lr_image: The low-resolution version is created by applying self.low_res_transform, which downscales the image (e.g., to 32x32 pixels).
13. Return: This method returns a tuple of three elements:
14. lr_image: The low-resolution version of the image.
15. hr_image: The high-resolution version of the image.

16. label: The class label for the image.



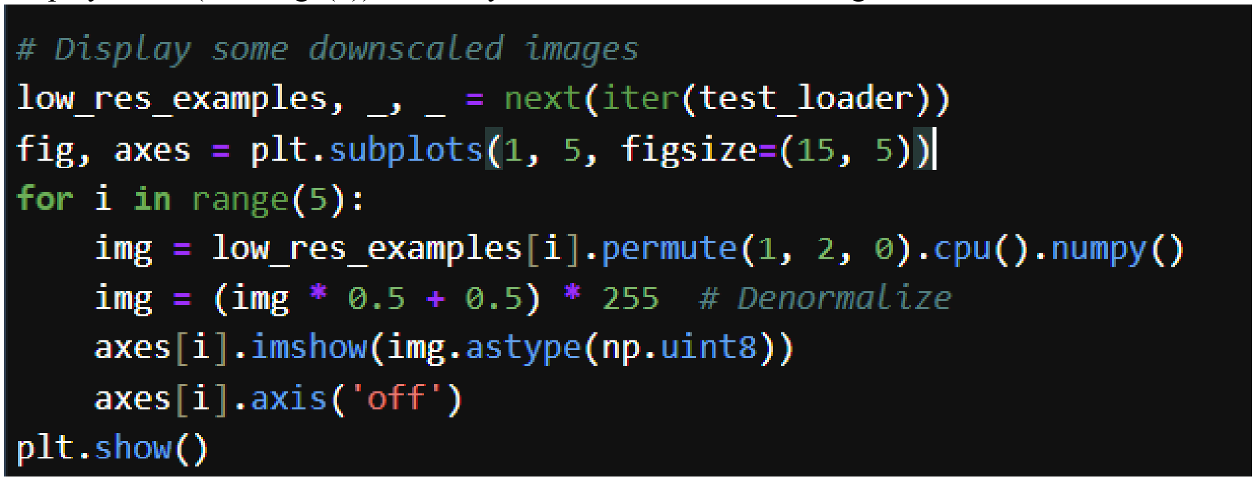
```python
# Dataset loading
class CustomDataset(Dataset):
    def __init__(self, root_dir, transform=None, low_res_transform=None):
        self.root_dir = root_dir
        self.transform = transform
        self.low_res_transform = low_res_transform
        self.classes = os.listdir(root_dir)
        self.file_paths = [(os.path.join(root_dir, cls, fname), cls) for cls in self.classes
                           for fname in os.listdir(os.path.join(root_dir, cls))]

    def __len__(self):
        return len(self.file_paths)

    def __getitem__(self, idx):
        img_path, label = self.file_paths[idx]

        try:
            image = Image.open(img_path).convert('RGB')
        except (UnidentifiedImageError, IOError):
            # If there's an issue with opening the image, skip it by reloading another sample
            return self.__getitem__((idx + 1) % len(self))

        if self.transform:
            hr_image = self.transform(image)
        if self.low_res_transform:
            lr_image = self.low_res_transform(image)

        return lr_image, hr_image, label
```

17. The dataset is split into 70% training and 30% testing, and we create data loaders for each.



```python
dataset = CustomDataset(root_dir=data_dir, transform=transform, low_res_transform=low_res_transform)
train_size = int(0.7 * len(dataset))
test_size = len(dataset) - train_size
train_dataset, test_dataset = random_split(dataset, [train_size, test_size])

train_loader = DataLoader(train_dataset, batch_size=16, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=16, shuffle=False)
```

18. Displays 5 for (i in range(5)) randomly chosen low-resolution images from the test set.



```python
# Display some downscaled images
low_res_examples, _, _ = next(iter(test_loader))
fig, axes = plt.subplots(1, 5, figsize=(15, 5))
for i in range(5):
    img = low_res_examples[i].permute(1, 2, 0).cpu().numpy()
    img = (img * 0.5 + 0.5) * 255  # Denormalize
    axes[i].imshow(img.astype(np.uint8))
    axes[i].axis('off')
plt.show()
```

Figure: Downscaled image sample

SRGAN Structure:

The SRGAN model consists of two primary components: the Generator and the Discriminator. Together, these components work in an adversarial framework where the generator tries to produce high-resolution images, and the discriminator aims to distinguish real images from generated images.

***Generator**:*

Initial Feature Extraction Layer:

Conv2d(3, 64, kernel_size=9, padding=4): A 2D convolution layer with a 9x9 kernel that takes an RGB image as input (3 channels) and outputs 64 feature maps.

PReLU: A Parametric ReLU activation function for non-linearity.

Residual Blocks: 16 Residual Blocks: Each block has:

Conv2d(64, 64, kernel_size=3, padding=1): A 3x3 convolution that maintains the spatial size of the image.

BatchNorm2d(64): Batch normalization to stabilize learning.

ReLU: A ReLU activation function for non-linearity.

These residual blocks allow the generator to learn high-frequency details without losing the original structure of the input.

Upsampling Layers: Two upsampling blocks to increase spatial resolution. Each block consists of:

Conv2d(64, 256, kernel_size=3, padding=1): Convolution with increased depth to 256 channels.

PixelShuffle(2): Rearranges the 256 channels to create a 2x upsampled image.

PReLU: Activation function for non-linearity.

These layers perform a 4x upsampling in total (e.g., from 32x32 to 128x128).

Final Output Layer: Conv2d(64, 3, kernel_size=9, padding=4): A 9x9 convolution that outputs a 3-channel (RGB) image with the same dimensions as the upsampled output.

Tanh: Scales the output to the range [-1, 1], matching the expected image data range.

***Discriminator**:*

Convolutional Layers: These layers progressively reduce the spatial dimensions of the input while increasing the depth, learning complex features to identify real vs. fake images.

Each convolutional layer is followed by LeakyReLU activation, with batch normalization applied from the second layer onward.

Flatten Layer: Flattens the output from the convolutional layers into a 1D vector.

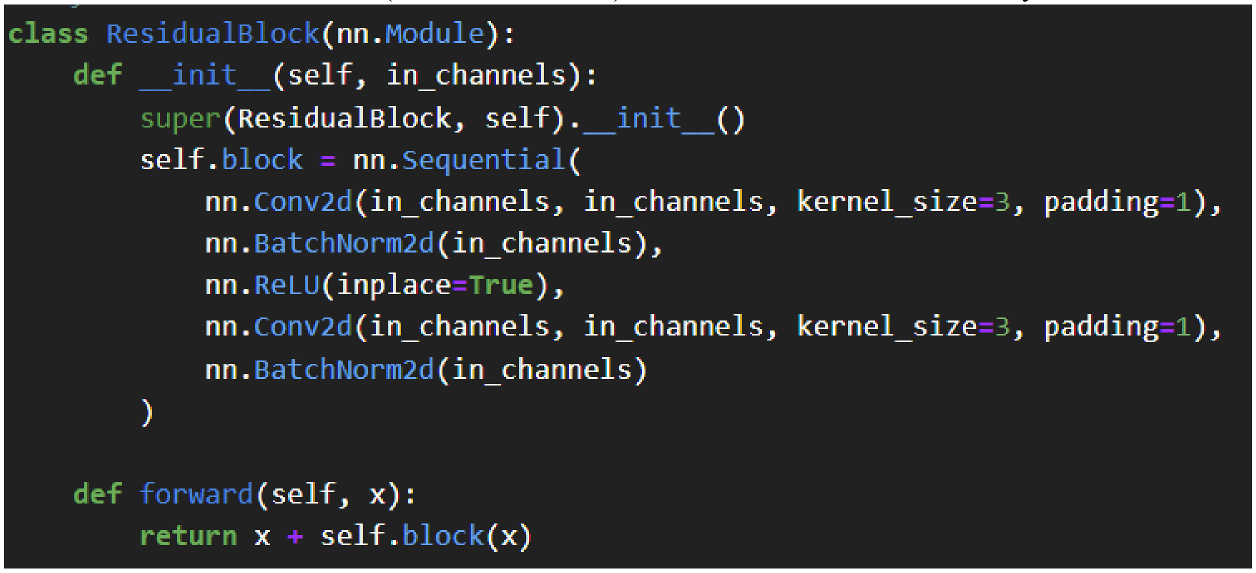
Fully Connected Layer:

*Linear(256 * 16 * 16, 1): A dense layer that outputs a single value.*

Sigmoid: Applies a sigmoid activation to produce a probability between 0 and 1, representing the discriminator's confidence in the image being real.

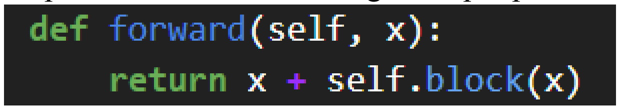
19. in_channels: the number of input channels for this block
20. self.block: This defines a sequential set of layers, which form the core of the residual block.
21. First Convolution Layer (nn.Conv2d): A 2D convolutional layer with in_channels as both input and output channels. Uses a 3x3 kernel with padding of 1, keeping the output size the same as the input.
22. Batch Normalization (nn.BatchNorm2d): Normalizes the output of the convolutional layer across the batch.
23. ReLU Activation (nn.ReLU(inplace=True)): Applies a ReLU activation function

24. Second Convolution Layer (nn.Conv2d) Another 3x3 convolutional layer with the same number of input and output channels. The second convolutional layer is applied after the ReLU activation.
25. Second Batch Normalization (nn.BatchNorm2d): A final batch normalization layer.



```python
class ResidualBlock(nn.Module):
    def __init__(self, in_channels):
        super(ResidualBlock, self).__init__()
        self.block = nn.Sequential(
            nn.Conv2d(in_channels, in_channels, kernel_size=3, padding=1),
            nn.BatchNorm2d(in_channels),
            nn.ReLU(inplace=True),
            nn.Conv2d(in_channels, in_channels, kernel_size=3, padding=1),
            nn.BatchNorm2d(in_channels)
        )

    def forward(self, x):
        return x + self.block(x)
```

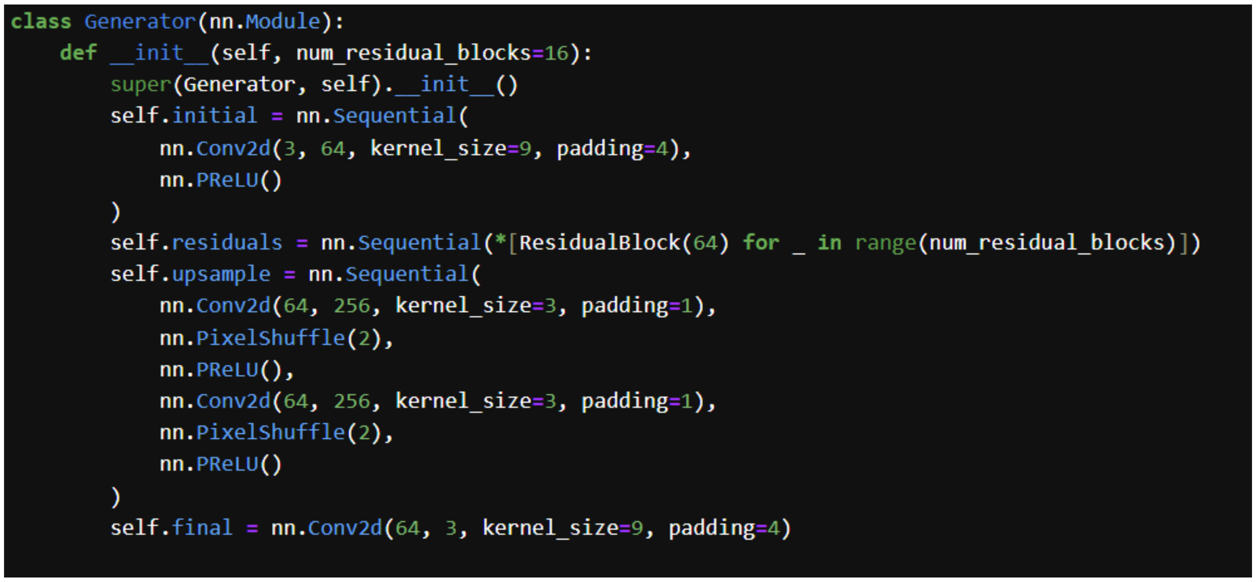
26. x: This is the input to the residual block.
27. self.block(x): Passes the input x through the sequential layers defined in self.block.
28. x + self.block(x): Adds the original input x to the output of self.block(x). This addition creates a "shortcut" path, allowing the network to "skip" over the block if needed. The output of the block is the original input plus the transformed features.



```python
def forward(self, x):
    return x + self.block(x)
```

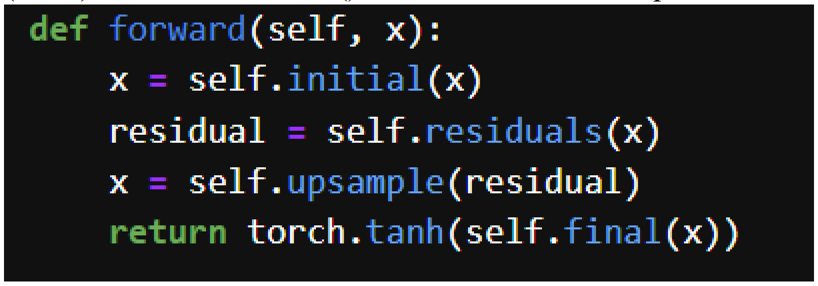
29. num_residual_blocks: 16 residual blocks are used in SRGANs.
30. super(Generator, self).__init__(): This calls the parent class (nn.Module) constructor to initialize it.
31. nn.Conv2d(3, 64, kernel_size=9, padding=4): A convolutional layer taking 3-channel input and outputs 64 feature maps. The kernel size is 9x9, and padding is 4
32. nn.PReLU(): A Parametric ReLU activation function
33. *[ResidualBlock(64) for _ in range(num_residual_blocks)]: Creates a list of num_residual_blocks (default 16) residual blocks, each with 64 input and output channels.
34. nn.Conv2d(64, 256, kernel_size=3, padding=1): A convolutional layer with 64 input channels and 256 output channels.
35. nn.PixelShuffle(2): PixelShuffle rearranges the elements of a 256-channel tensor into a 64-channel tensor with twice the height and width.
36. nn.PReLU(): A PReLU activation function to introduce non-linearity

37. nn.Conv2d(64, 3, kernel_size=9, padding=4): A convolutional layer that reduces the channels from 64 back to 3 RGB channels



```python
class Generator(nn.Module):
    def __init__(self, num_residual_blocks=16):
        super(Generator, self).__init__()
        self.initial = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=9, padding=4),
            nn.PReLU()
        )
        self.residuals = nn.Sequential(*[ResidualBlock(64) for _ in range(num_residual_blocks)])
        self.upsample = nn.Sequential(
            nn.Conv2d(64, 256, kernel_size=3, padding=1),
            nn.PixelShuffle(2),
            nn.PReLU(),
            nn.Conv2d(64, 256, kernel_size=3, padding=1),
            nn.PixelShuffle(2),
            nn.PReLU()
        )
        self.final = nn.Conv2d(64, 3, kernel_size=9, padding=4)
```

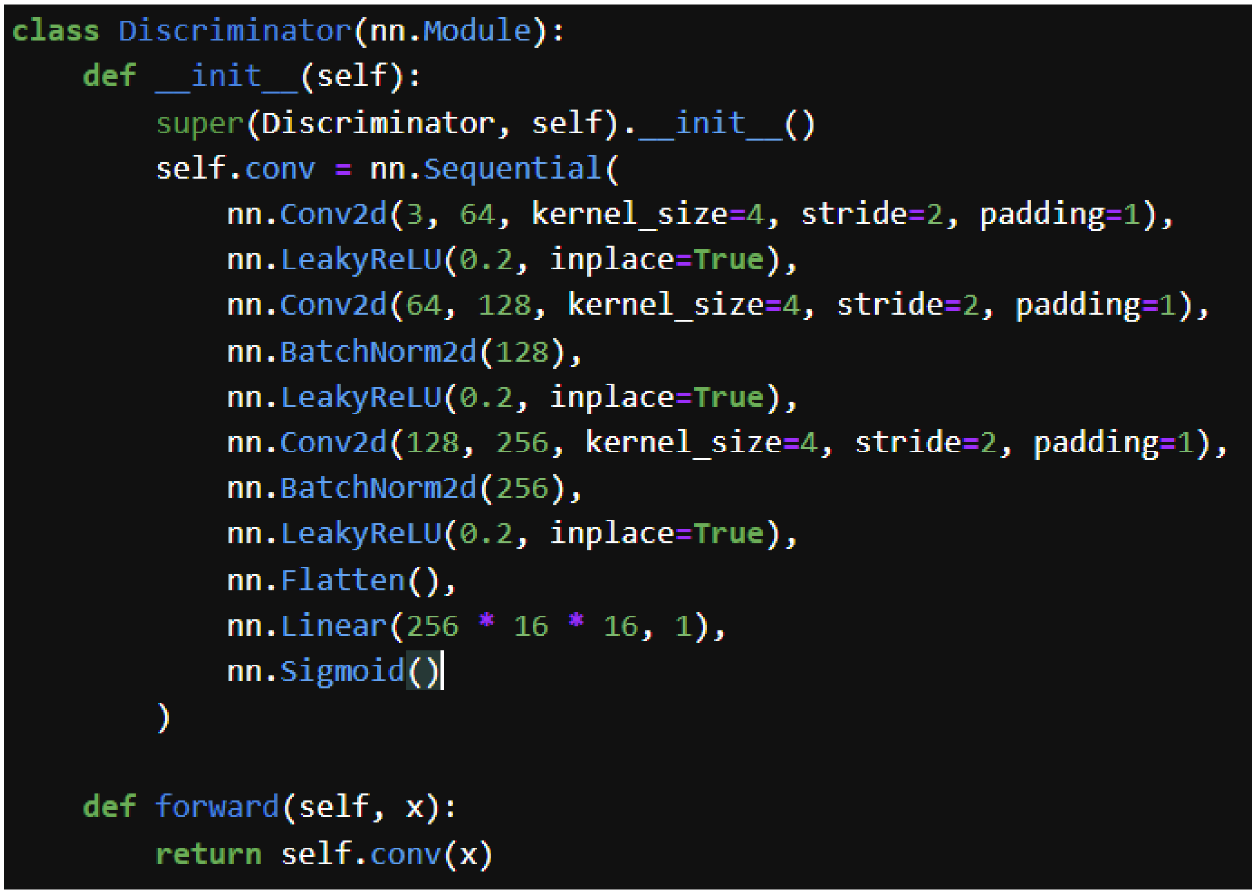
38. self.initial(x): The input image x is first processed by the initial feature extraction layer.
39. self.residuals(x): The result is then passed through the residual blocks.
40. self.upsample(residual): The residual feature map is then upsampled through the upsampling layers
41. torch.tanh(self.final(x)): The final convolutional layer outputs an image with 3 channels (RGB), and the torch.tanh() function scales the output to the range [-1, 1]



```python
def forward(self, x):
    x = self.initial(x)
    residual = self.residuals(x)
    x = self.upsample(residual)
    return torch.tanh(self.final(x))
```

42. nn.Conv2d(3, 64, kernel_size=4, stride=2, padding=1): Takes an RGB image with 3 channels as input. Outputs 64 feature maps. Kernel size of 4x4 with a stride of 2 and padding of 1 reduces the spatial size of the input by half.
43. nn.LeakyReLU(0.2, inplace=True): Applies the leaky ReLU activation function with a negative slope (0.2).
44. inplace=True modifies the data in-place for memory efficiency.
45. Second Convolutional Block: nn.Conv2d(64, 128, kernel_size=4, stride=2, padding=1): Takes the 64 feature maps from the previous layer and outputs 128 feature maps. Uses a 4x4 kernel with a stride of 2, again reducing spatial dimensions by half.
46. nn.LeakyReLU(0.2, inplace=True): Applies leaky ReLU activation with a small negative slope (0.2).
47. Third Convolutional Block

48. nn.Conv2d(128, 256, kernel_size=4, stride=2, padding=1): Takes the 128 feature maps and outputs 256 feature maps. The 4x4 kernel with a stride of 2 further reduces the spatial dimensions by half.
49. nn.BatchNorm2d(256): Normalizes the output to improve convergence and reduce internal covariate shift.
50. nn.LeakyReLU(0.2, inplace=True): Applies leaky ReLU activation.
51. Flatten Layer:nn.Flatten(): Converts the 3D tensor output from the convolutional layers into a 1D vector. After the convolutional blocks, the feature map size is expected to be 256 x 16 x 16. Flattening transforms this into a vector of size 256 * 16 * 16 = 65536.
52. Fully Connected Layer
53. nn.Linear(256 * 16 * 16, 1): This layer reduces the 65536-dimensional vector into a single value.
54. nn.Sigmoid(): Applies a sigmoid activation function to produce a value between 0 and 1. A value close to 1 indicates the discriminator believes the image is real, and close to 0 indicates it's fake.

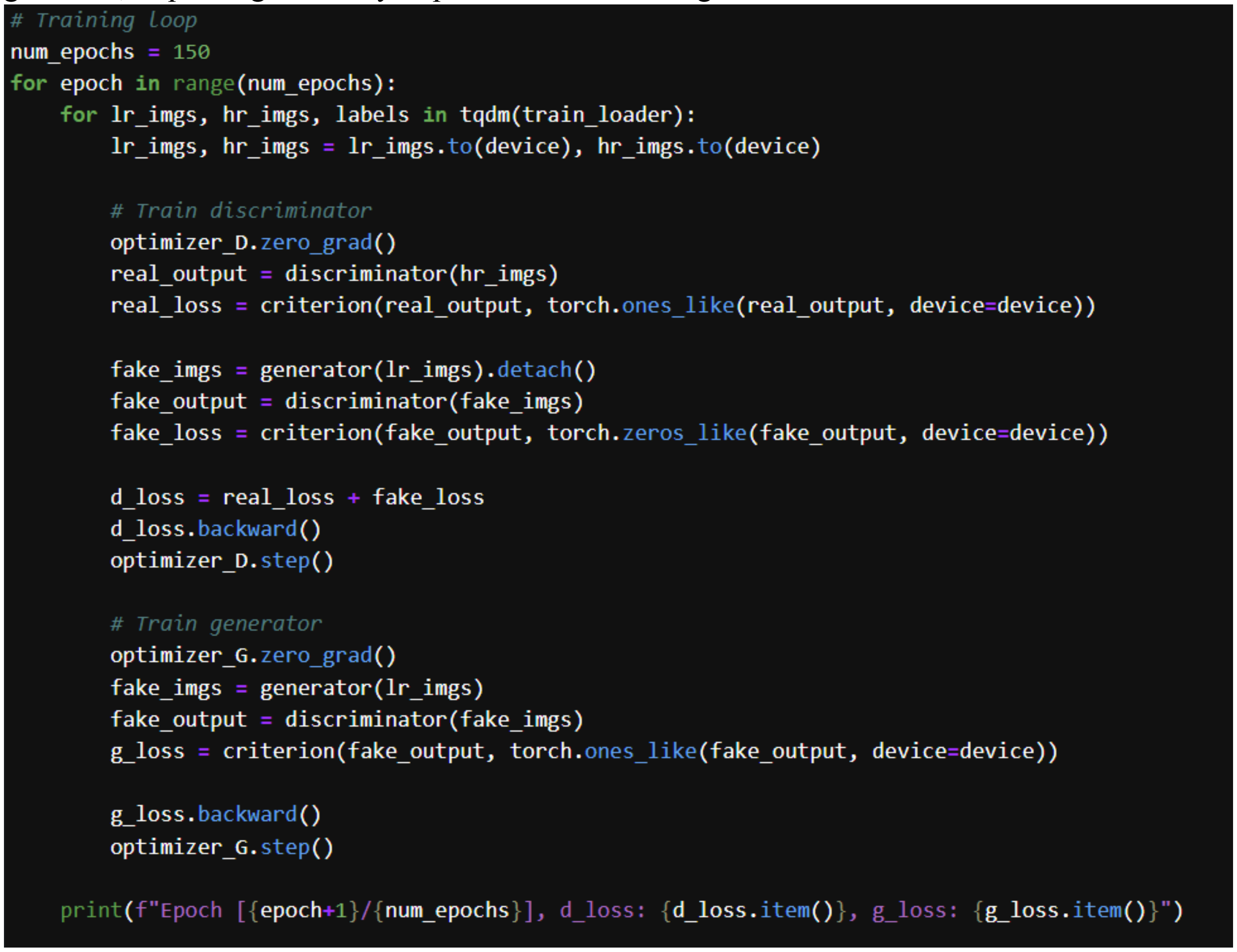


```python
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.conv = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=4, stride=2, padding=1),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(64, 128, kernel_size=4, stride=2, padding=1),
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(128, 256, kernel_size=4, stride=2, padding=1),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Flatten(),
            nn.Linear(256 * 16 * 16, 1),
            nn.Sigmoid()
        )

    def forward(self, x):
        return self.conv(x)
```

55. num_epochs: The total number of training epochs is set to 150.
56. for lr_imgs, hr_imgs, labels in tqdm(train_loader): This inner loop iterates over batches of images from the train_loader, which contains:
57. lr_imgs: Low-resolution images, the input to the generator.
58. hr_imgs: High-resolution images, the target for the generator's output.

59. optimizer_D.zero_grad(): Resets the gradients for the discriminator’s optimizer to avoid accumulation from previous batches.
60. real_output = discriminator(hr_imgs): The discriminator processes the high-resolution (real) images, outputting a prediction (close to 1 for real).
61. torch.ones_like(real_output, device=device): Creates a tensor filled with 1s, representing "real" labels.
62. real_loss = criterion(real_output, torch.ones_like(real_output, device=device)): Calculates the BCE loss for real images
63. fake_imgs = generator(lr_imgs).detach(): The generator creates fake high-resolution images from low-resolution inputs. .detach() prevents gradients from flowing back to the generator during the discriminator’s training.
64. fake_output = discriminator(fake_imgs): The discriminator evaluates the generated (fake) images, outputting a prediction (close to 0 for fake).
65. torch.zeros_like(fake_output, device=device): Creates a tensor filled with 0s, representing "fake" labels.
66. fake_loss = criterion(fake_output, torch.zeros_like(fake_output, device=device)): Calculates the BCE loss for fake images, encouraging the discriminator to predict values close to 0 for fake images.
67. d_loss = real_loss + fake_loss: Combines the loss for real and fake images to get the total discriminator loss.
68. d_loss.backward(): Computes the gradients of d_loss with respect to the discriminator’s parameters.
69. optimizer_D.step(): Updates the discriminator’s parameters based on the computed gradients, improving its ability to differentiate real from fake images.
70. optimizer_G.zero_grad(): Resets the gradients for the generator’s optimizer to avoid accumulation from previous batches.
71. Generate Fake Images:
72. fake_imgs = generator(lr_imgs): The generator takes low-resolution images as input and produces high-resolution (fake) images.
73. fake_output = discriminator(fake_imgs): The discriminator evaluates the generated images, outputting a prediction (close to 0 or 1).
74. Calculate Generator Loss:
75. torch.ones_like(fake_output, device=device): Creates a tensor filled with 1s, indicating that the generator wants the discriminator to classify its output as real.
76. g_loss = criterion(fake_output, torch.ones_like(fake_output, device=device)): Calculates the BCE loss for the generator.
77. Backpropagation and Update:
78. g_loss.backward(): Computes the gradients of g_loss with respect to the generator’s parameters.

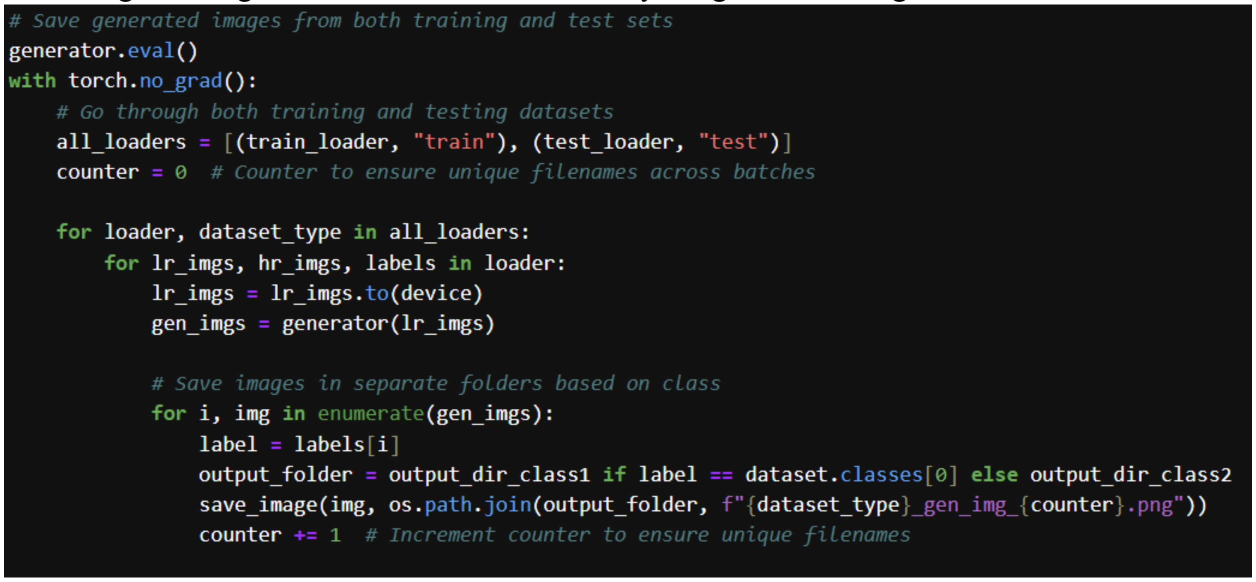
79. optimizer_G.step(): Updates the generator’s parameters based on the computed gradients, improving its ability to produce realistic images.



```python
# Training loop
num_epochs = 150
for epoch in range(num_epochs):
    for lr_imgs, hr_imgs, labels in tqdm(train_loader):
        lr_imgs, hr_imgs = lr_imgs.to(device), hr_imgs.to(device)

        # Train discriminator
        optimizer_D.zero_grad()
        real_output = discriminator(hr_imgs)
        real_loss = criterion(real_output, torch.ones_like(real_output, device=device))

        fake_imgs = generator(lr_imgs).detach()
        fake_output = discriminator(fake_imgs)
        fake_loss = criterion(fake_output, torch.zeros_like(fake_output, device=device))

        d_loss = real_loss + fake_loss
        d_loss.backward()
        optimizer_D.step()

        # Train generator
        optimizer_G.zero_grad()
        fake_imgs = generator(lr_imgs)
        fake_output = discriminator(fake_imgs)
        g_loss = criterion(fake_output, torch.ones_like(fake_output, device=device))

        g_loss.backward()
        optimizer_G.step()

    print(f"Epoch [{epoch+1}/{num_epochs}], d_loss: {d_loss.item()}, g_loss: {g_loss.item()}")
```

80. generator.eval(): Sets the generator to evaluation mode, which disables dropout and batch normalization layers’ training behavior.
81. with torch.no_grad(): Deactivates gradient tracking
82. all_loaders: A list containing both the training and test data loaders
83. counter = 0: Initializes a counter to ensure unique filenames, preventing overwriting when saving images across batches.
84. gen_imgs = generator(lr_imgs): Passes the low-resolution images through the generator to create high-resolution (generated) images.
85. label = labels[i]: Retrieves the class label for the corresponding generated image.
86. Determine output folder:
87. output_dir_class1 if label == dataset.classes[0] else output_dir_class2: Uses the label to decide which output folder to save the image in:
88. output_dir_class1 if the image belongs to the first class.
89. output_dir_class2 if it belongs to the second class.
90. Saving the Image:
91. save_image(img, os.path.join(output_folder, f"{dataset_type}_gen_img_{counter}.png")):

92. Saves the generated image to the specified output folder with a unique filename. We are saving the images to train our model to classify the generated images.



```
# Save generated images from both training and test sets
generator.eval()
with torch.no_grad():
    # Go through both training and testing datasets
    all_loaders = [(train_loader, "train"), (test_loader, "test")]
    counter = 0  # Counter to ensure unique filenames across batches

    for loader, dataset_type in all_loaders:
        for lr_imgs, hr_imgs, labels in loader:
            lr_imgs = lr_imgs.to(device)
            gen_imgs = generator(lr_imgs)

            # Save images in separate folders based on class
            for i, img in enumerate(gen_imgs):
                label = labels[i]
                output_folder = output_dir_class1 if label == dataset.classes[0] else output_dir_class2
                save_image(img, os.path.join(output_folder, f"{dataset_type}_gen_img_{counter}.png"))
                counter += 1  # Increment counter to ensure unique filenames
```

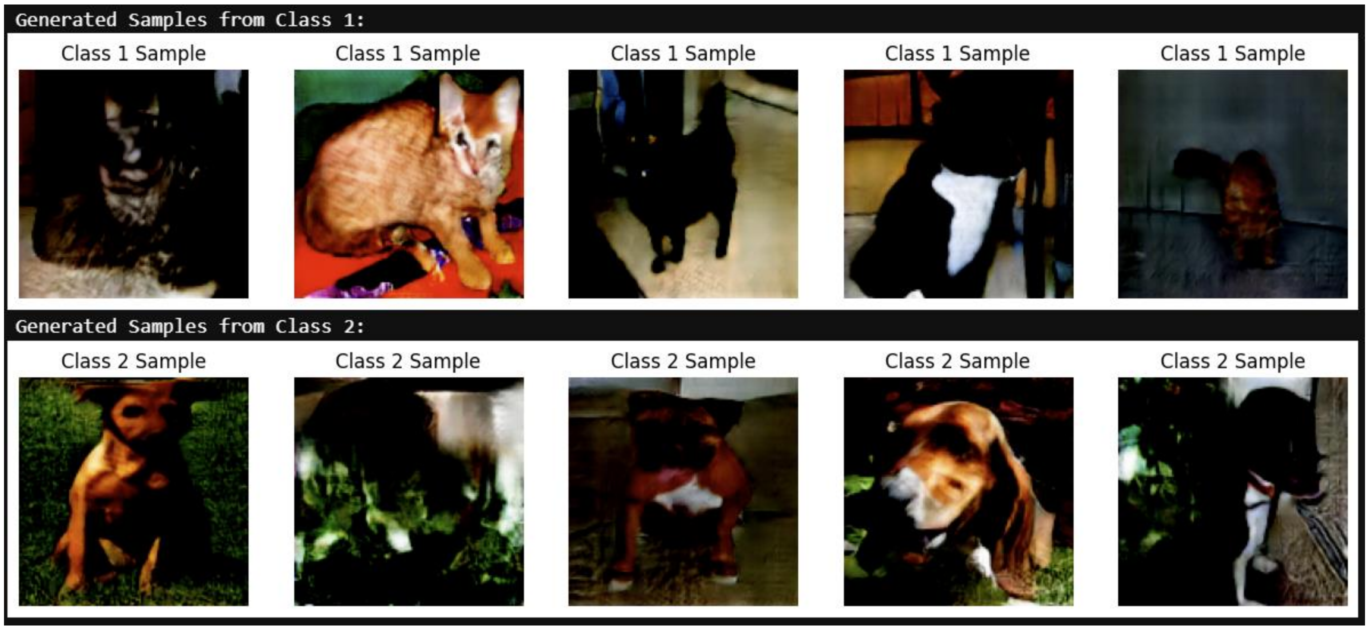


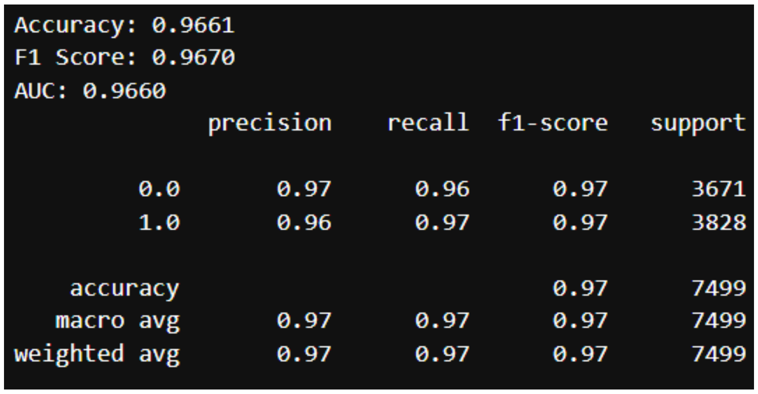
Figure: Generated image samples

Model performance comparison:

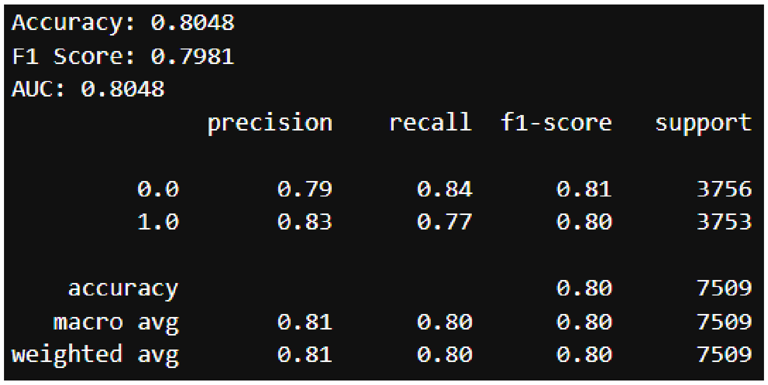
Model 1 is trained on original dataset and model 2 is trained on generated images from SRGAN dataset. We have used VGG 16 as our base model for both cases and the structure and parameters of both the models are kept the same. The complete model structure is as follows:

- *Feature Extraction (from VGG16, frozen)*
- *Classification Head (custom layers for binary classification):*
- *Flatten layer*
- *Fully Connected (4096, ReLU)*
- *Dropout (0.5)*
- *Fully Connected (128, ReLU)*
- *Dropout (0.5)*
- *Fully Connected (1, Sigmoid)*

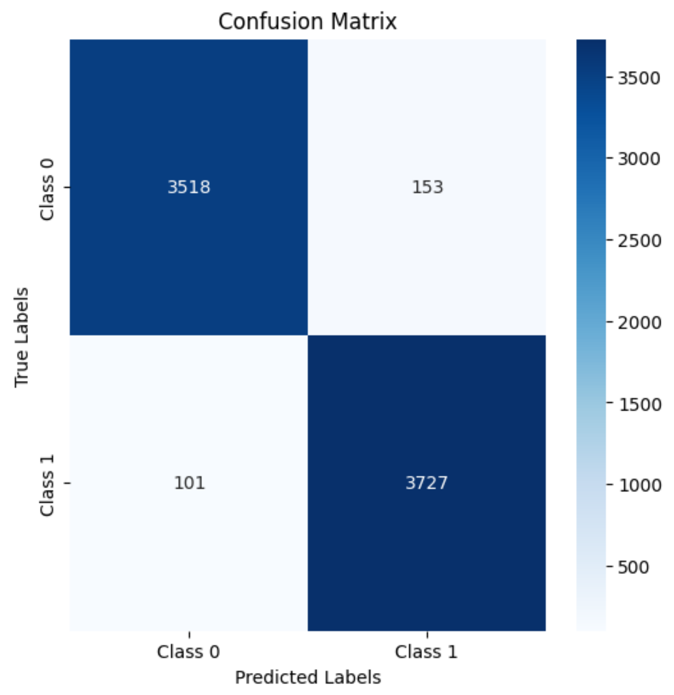
Model 1 performance:


```
Accuracy: 0.9661
F1 Score: 0.9670
AUC: 0.9660
              precision    recall  f1-score   support

         0.0       0.97      0.96      0.97      3671
         1.0       0.96      0.97      0.97      3828

    accuracy                           0.97      7499
   macro avg       0.97      0.97      0.97      7499
weighted avg       0.97      0.97      0.97      7499
```

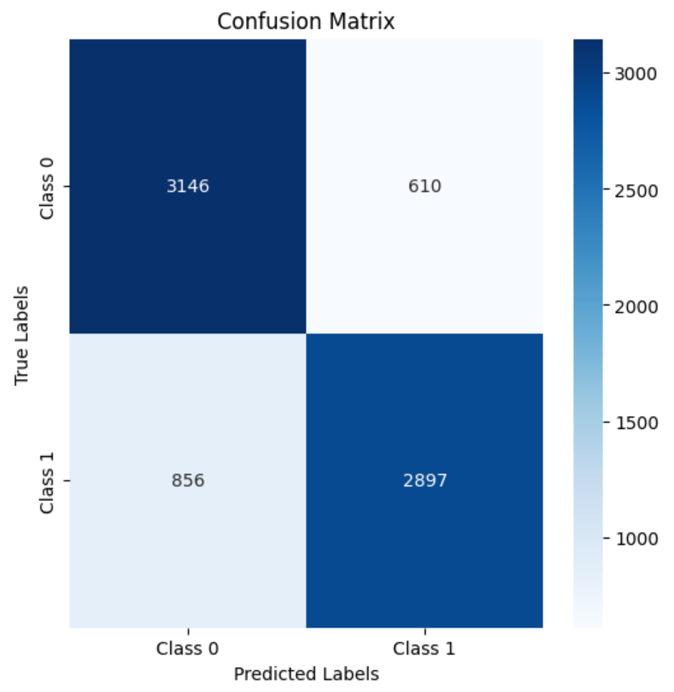
Model 2 performance:


```
Accuracy: 0.8048
F1 Score: 0.7981
AUC: 0.8048
              precision    recall  f1-score   support

         0.0       0.79      0.84      0.81      3756
         1.0       0.83      0.77      0.80      3753

    accuracy                           0.80      7509
   macro avg       0.81      0.80      0.80      7509
weighted avg       0.81      0.80      0.80      7509
```

Model 1 Confusion Matrix on Orginal test set:



Model 2 Confusion Matrix on Generated image test set:



It is evident from the performance matrix that Model 1, which is trained on the Original dataset, is performing better than Model 2 (trained on generated image dataset).