# srgan

November 21, 2023

### 0.0.1 Libraries

[1]:
```
import numpy as np
import pandas as pd
import os, math, sys
import glob, itertools
import argparse, random

import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.autograd import Variable
from torchvision.models import vgg19
import torchvision.transforms as transforms
from torch.utils.data import DataLoader, Dataset
from torchvision.utils import save_image, make_grid

import plotly
import plotly.express as px
import plotly.graph_objects as go
import matplotlib.pyplot as plt

from PIL import Image
from tqdm import tqdm_notebook as tqdm
from sklearn.model_selection import train_test_split

random.seed(42)
import warnings
warnings.filterwarnings("ignore")
```

### 0.0.2 Settings

[2]:
```
load_pretrained_models = True
n_epochs = 2
dataset_path = "../input/celeba-dataset/img_align_celeba/img_align_celeba"
batch_size = 16
lr = 0.00008
b1 = 0.5
```

```
b2 = 0.999
decay_epoch = 100
n_cpu = 8
hr_height = 256
hr_width = 256
channels = 3

os.makedirs("images", exist_ok=True)
os.makedirs("saved_models", exist_ok=True)

cuda = torch.cuda.is_available()
hr_shape = (hr_height, hr_width)
```

### 0.0.3 Define Dataset Class

[3]:
```
mean = np.array([0.485, 0.456, 0.406])
std = np.array([0.229, 0.224, 0.225])

class ImageDataset(Dataset):
    def __init__(self, files, hr_shape):
        hr_height, hr_width = hr_shape
        self.lr_transform = transforms.Compose(
            [
                transforms.Resize((hr_height // 4, hr_height // 4), Image.
 ↪BICUBIC),
                transforms.ToTensor(),
                transforms.Normalize(mean, std),
            ]
        )
        self.hr_transform = transforms.Compose(
            [
                transforms.Resize((hr_height, hr_height), Image.BICUBIC),
                transforms.ToTensor(),
                transforms.Normalize(mean, std),
            ]
        )
        self.files = files

    def __getitem__(self, index):
        img = Image.open(self.files[index % len(self.files)])
        img_lr = self.lr_transform(img)
        img_hr = self.hr_transform(img)

        return {"lr": img_lr, "hr": img_hr}

    def __len__(self):
        return len(self.files)
```

### 0.0.4 Get Train/Test Dataloaders

[5]:
```
train_paths, test_paths = train_test_split(sorted(glob.glob(dataset_path + "/*.
 ↪*")), test_size=0.02, random_state=42)
train_dataloader = DataLoader(ImageDataset(train_paths, hr_shape=hr_shape),␣
 ↪batch_size=batch_size, shuffle=True, num_workers=n_cpu)
test_dataloader = DataLoader(ImageDataset(test_paths, hr_shape=hr_shape),␣
 ↪batch_size=int(batch_size*0.75), shuffle=True, num_workers=n_cpu)
```

### 0.0.5 Define Model Classes

[15]:
```
class FeatureExtractor(nn.Module):
    def __init__(self):
        super(FeatureExtractor, self).__init__()
        vgg19_model = vgg19(pretrained=True)
        self.feature_extractor = nn.Sequential(*list(vgg19_model.features.
 ↪children())[:18])

    def forward(self, img):
        return self.feature_extractor(img)


class ResidualBlock(nn.Module):
    def __init__(self, in_features):
        super(ResidualBlock, self).__init__()
        self.conv_block = nn.Sequential(
            nn.Conv2d(in_features, in_features, kernel_size=3, stride=1,␣
 ↪padding=1),
            nn.BatchNorm2d(in_features, 0.8),
            nn.PReLU(),
            nn.Conv2d(in_features, in_features, kernel_size=3, stride=1,␣
 ↪padding=1),
            nn.BatchNorm2d(in_features, 0.8),
        )

    def forward(self, x):
        return x + self.conv_block(x)


class GeneratorResNet(nn.Module):
    def __init__(self, in_channels=3, out_channels=3, n_residual_blocks=16):
        super(GeneratorResNet, self).__init__()

        # First layer
        self.conv1 = nn.Sequential(nn.Conv2d(in_channels, 64, kernel_size=9,␣
 ↪stride=1, padding=4), nn.PReLU())
```

```
        # Residual blocks
        res_blocks = []
        for _ in range(n_residual_blocks):
            res_blocks.append(ResidualBlock(64))
        self.res_blocks = nn.Sequential(*res_blocks)

        # Second conv layer post residual blocks
        self.conv2 = nn.Sequential(nn.Conv2d(64, 64, kernel_size=3, stride=1,␣
↪padding=1), nn.BatchNorm2d(64, 0.8))

        # Upsampling layers
        upsampling = []
        for out_features in range(2):
            upsampling += [
                # nn.Upsample(scale_factor=2),
                nn.Conv2d(64, 256, 3, 1, 1),
                nn.BatchNorm2d(256),
                nn.PixelShuffle(upscale_factor=2),
                nn.PReLU(),
            ]
        self.upsampling = nn.Sequential(*upsampling)

        # Final output layer
        self.conv3 = nn.Sequential(nn.Conv2d(64, out_channels, kernel_size=9,␣
↪stride=1, padding=4), nn.Tanh())

    def forward(self, x):
        out1 = self.conv1(x)
        out = self.res_blocks(out1)
        out2 = self.conv2(out)
        out = torch.add(out1, out2)
        out = self.upsampling(out)
        out = self.conv3(out)
        return out


class Discriminator(nn.Module):
    def __init__(self, input_shape):
        super(Discriminator, self).__init__()

        self.input_shape = input_shape
        in_channels, in_height, in_width = self.input_shape
        patch_h, patch_w = int(in_height / 2 ** 4), int(in_width / 2 ** 4)
        self.output_shape = (1, patch_h, patch_w)

        def discriminator_block(in_filters, out_filters, first_block=False):
            layers = []
```

```
            layers.append(nn.Conv2d(in_filters, out_filters, kernel_size=3,␣
↪stride=1, padding=1))
            if not first_block:
                layers.append(nn.BatchNorm2d(out_filters))
            layers.append(nn.LeakyReLU(0.2, inplace=True))
            layers.append(nn.Conv2d(out_filters, out_filters, kernel_size=3,␣
↪stride=2, padding=1))
            layers.append(nn.BatchNorm2d(out_filters))
            layers.append(nn.LeakyReLU(0.2, inplace=True))
            return layers

        layers = []
        in_filters = in_channels
        for i, out_filters in enumerate([64, 128, 256, 512]):
            layers.extend(discriminator_block(in_filters, out_filters,␣
↪first_block=(i == 0)))
            in_filters = out_filters

        layers.append(nn.Conv2d(out_filters, 1, kernel_size=3, stride=1,␣
↪padding=1))

        self.model = nn.Sequential(*layers)

    def forward(self, img):
        return self.model(img)
```

### 0.0.6 Train Super Resolution GAN (SRGAN)

[17]:
```
# Initializing generator and discriminator
generator = GeneratorResNet()
discriminator = Discriminator(input_shape=(channels, *hr_shape))
feature_extractor = FeatureExtractor()

# Setting feature extractor to inference mode
feature_extractor.eval()

# Losses calculation
criterion_GAN = torch.nn.MSELoss()
criterion_content = torch.nn.L1Loss()

if cuda:
    generator = generator.cuda()
    discriminator = discriminator.cuda()
    feature_extractor = feature_extractor.cuda()
    criterion_GAN = criterion_GAN.cuda()
    criterion_content = criterion_content.cuda()
```

```
# Load pretrained models
if load_pretrained_models:
    generator.load_state_dict(torch.load("../input/
 ↪single-image-super-resolution-gan-srgan-pytorch/saved_models/generator.pth"))
    discriminator.load_state_dict(torch.load("../input/
 ↪single-image-super-resolution-gan-srgan-pytorch/saved_models/discriminator.
 ↪pth"))

# Optimizers
optimizer_G = torch.optim.Adam(generator.parameters(), lr=lr, betas=(b1, b2))
optimizer_D = torch.optim.Adam(discriminator.parameters(), lr=lr, betas=(b1,␣
 ↪b2))

Tensor = torch.cuda.FloatTensor if cuda else torch.Tensor
```

[18]:
```
train_gen_losses, train_disc_losses, train_counter = [], [], []
test_gen_losses, test_disc_losses = [], []
test_counter = [idx*len(train_dataloader.dataset) for idx in range(1,␣
 ↪n_epochs+1)]

for epoch in range(n_epochs):

    ### Training
    gen_loss, disc_loss = 0, 0
    tqdm_bar = tqdm(train_dataloader, desc=f'Training Epoch {epoch} ',␣
 ↪total=int(len(train_dataloader)))
    for batch_idx, imgs in enumerate(tqdm_bar):
        generator.train(); discriminator.train()
        # Configure model input
        imgs_lr = Variable(imgs["lr"].type(Tensor))
        imgs_hr = Variable(imgs["hr"].type(Tensor))
        # Adversarial ground truths
        valid = Variable(Tensor(np.ones((imgs_lr.size(0), *discriminator.
 ↪output_shape))), requires_grad=False)
        fake = Variable(Tensor(np.zeros((imgs_lr.size(0), *discriminator.
 ↪output_shape))), requires_grad=False)

        ### Train Generator
        optimizer_G.zero_grad()
        # Generate a high resolution image from low resolution input
        gen_hr = generator(imgs_lr)
        # Adversarial loss
        loss_GAN = criterion_GAN(discriminator(gen_hr), valid)
        # Content loss
        gen_features = feature_extractor(gen_hr)
```

```
        real_features = feature_extractor(imgs_hr)
        loss_content = criterion_content(gen_features, real_features.detach())
        # Total loss
        loss_G = loss_content + 1e-3 * loss_GAN
        loss_G.backward()
        optimizer_G.step()

        ### Train Discriminator
        optimizer_D.zero_grad()
        # Loss of real and fake images
        loss_real = criterion_GAN(discriminator(imgs_hr), valid)
        loss_fake = criterion_GAN(discriminator(gen_hr.detach()), fake)
        # Total loss
        loss_D = (loss_real + loss_fake) / 2
        loss_D.backward()
        optimizer_D.step()

        gen_loss += loss_G.item()
        train_gen_losses.append(loss_G.item())
        disc_loss += loss_D.item()
        train_disc_losses.append(loss_D.item())
        train_counter.append(batch_idx*batch_size + imgs_lr.size(0) +␣
↪epoch*len(train_dataloader.dataset))
        tqdm_bar.set_postfix(gen_loss=gen_loss/(batch_idx+1),␣
↪disc_loss=disc_loss/(batch_idx+1))

    # Testing
    gen_loss, disc_loss = 0, 0
    tqdm_bar = tqdm(test_dataloader, desc=f'Testing Epoch {epoch} ',␣
↪total=int(len(test_dataloader)))
    for batch_idx, imgs in enumerate(tqdm_bar):
        generator.eval(); discriminator.eval()
        # Configure model input
        imgs_lr = Variable(imgs["lr"].type(Tensor))
        imgs_hr = Variable(imgs["hr"].type(Tensor))
        # Adversarial ground truths
        valid = Variable(Tensor(np.ones((imgs_lr.size(0), *discriminator.
↪output_shape))), requires_grad=False)
        fake = Variable(Tensor(np.zeros((imgs_lr.size(0), *discriminator.
↪output_shape))), requires_grad=False)

        ### Eval Generator
        # Generate a high resolution image from low resolution input
        gen_hr = generator(imgs_lr)
        # Adversarial loss
        loss_GAN = criterion_GAN(discriminator(gen_hr), valid)
        # Content loss
```

```
        gen_features = feature_extractor(gen_hr)
        real_features = feature_extractor(imgs_hr)
        loss_content = criterion_content(gen_features, real_features.detach())
        # Total loss
        loss_G = loss_content + 1e-3 * loss_GAN

        ### Eval Discriminator
        # Loss of real and fake images
        loss_real = criterion_GAN(discriminator(imgs_hr), valid)
        loss_fake = criterion_GAN(discriminator(gen_hr.detach()), fake)
        # Total loss
        loss_D = (loss_real + loss_fake) / 2

        gen_loss += loss_G.item()
        disc_loss += loss_D.item()
        tqdm_bar.set_postfix(gen_loss=gen_loss/(batch_idx+1),␣
 ↪disc_loss=disc_loss/(batch_idx+1))

        # Saving image grid with upsampled inputs and SRGAN outputs
        if random.uniform(0,1)<0.1:
            imgs_lr = nn.functional.interpolate(imgs_lr, scale_factor=4)
            imgs_hr = make_grid(imgs_hr, nrow=1, normalize=True)
            gen_hr = make_grid(gen_hr, nrow=1, normalize=True)
            imgs_lr = make_grid(imgs_lr, nrow=1, normalize=True)
            img_grid = torch.cat((imgs_hr, imgs_lr, gen_hr), -1)
            save_image(img_grid, f"images/{batch_idx}.png", normalize=False)

    test_gen_losses.append(gen_loss/len(test_dataloader))
    test_disc_losses.append(disc_loss/len(test_dataloader))

    # Saving model checkpoints
    if np.argmin(test_gen_losses) == len(test_gen_losses)-1:
        torch.save(generator.state_dict(), "saved_models/generator.pth")
        torch.save(discriminator.state_dict(), "saved_models/discriminator.pth")
```

```
HBox(children=(FloatProgress(value=0.0, description='Training Epoch 0 ',␣
 ↪max=12410.0, style=ProgressStyle(desc…
```

```
HBox(children=(FloatProgress(value=0.0, description='Testing Epoch 0 ', max=338.
 ↪0, style=ProgressStyle(descrip…
```

```
HBox(children=(FloatProgress(value=0.0, description='Training Epoch 1 ',␣
 ↪max=12410.0, style=ProgressStyle(desc…
```

```
HBox(children=(FloatProgress(value=0.0, description='Testing Epoch 1 ', max=338.
 ↪0, style=ProgressStyle(descrip…
```

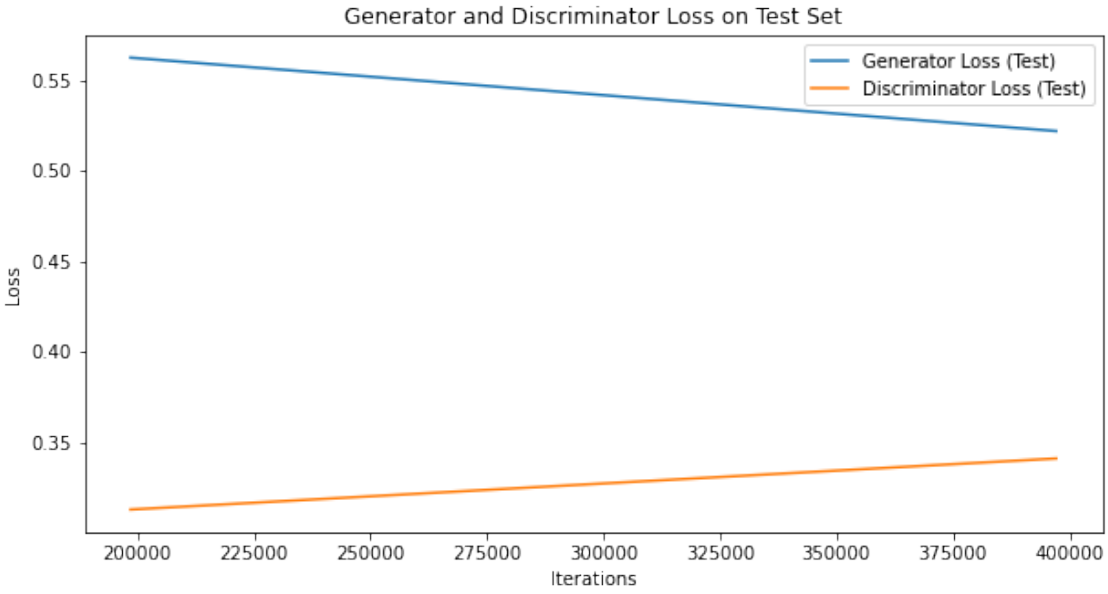
[19]:
```
import plotly.graph_objects as go

fig = go.Figure()

# Train Generator Loss
fig.add_trace(go.Scatter(x=train_counter, y=train_gen_losses, mode='lines',␣
 ↪name='Train Generator Loss'))

# Test Generator Loss
fig.add_trace(go.Scatter(x=test_counter, y=test_gen_losses,
                         mode='markers', marker=dict(symbol='star-diamond',␣
 ↪color='orange', size=9, line_width=1),
                         name='Test Generator Loss'))

fig.update_layout(
    width=1000,
    height=500,
    title="Train vs. Test Generator Loss",
    xaxis_title="Number of training examples seen",
    yaxis_title="Adversarial + Content Loss",
    legend=dict(x=0, y=1, traceorder='normal', orientation='h'),
)

fig.show()
```

[20]:
```
import plotly.graph_objects as go

fig = go.Figure()

# Train Discriminator Loss
fig.add_trace(go.Scatter(x=train_counter, y=train_disc_losses, mode='lines',␣
 ↪name='Train Discriminator Loss'))

# Test Discriminator Loss
fig.add_trace(go.Scatter(x=test_counter, y=test_disc_losses,
                         mode='markers', marker=dict(symbol='star-diamond',␣
 ↪color='orange', size=9, line_width=1),
                         name='Test Discriminator Loss'))

fig.update_layout(
    width=1000,
    height=500,
```

```
    title="Train vs. Test Discriminator Loss",
    xaxis_title="Number of training examples seen",
    yaxis_title="Adversarial Loss",
    legend=dict(x=0, y=1, traceorder='normal', orientation='h'),
)

fig.show()
```

[28]:
```
plt.figure(figsize=(10, 5))
plt.plot(train_counter, train_gen_losses, label='Generator Loss')
plt.plot(train_counter, train_disc_losses, label='Discriminator Loss')
plt.legend()
plt.title('Generator and Discriminator Loss Over Training')
plt.xlabel('Iterations')
plt.ylabel('Loss')
plt.show()
```


[29]:
```
plt.figure(figsize=(10, 5))
plt.plot(test_counter, test_gen_losses, label='Generator Loss (Test)')
plt.plot(test_counter, test_disc_losses, label='Discriminator Loss (Test)')
plt.legend()
plt.title('Generator and Discriminator Loss on Test Set')
plt.xlabel('Iterations')
plt.ylabel('Loss')
plt.show()
```

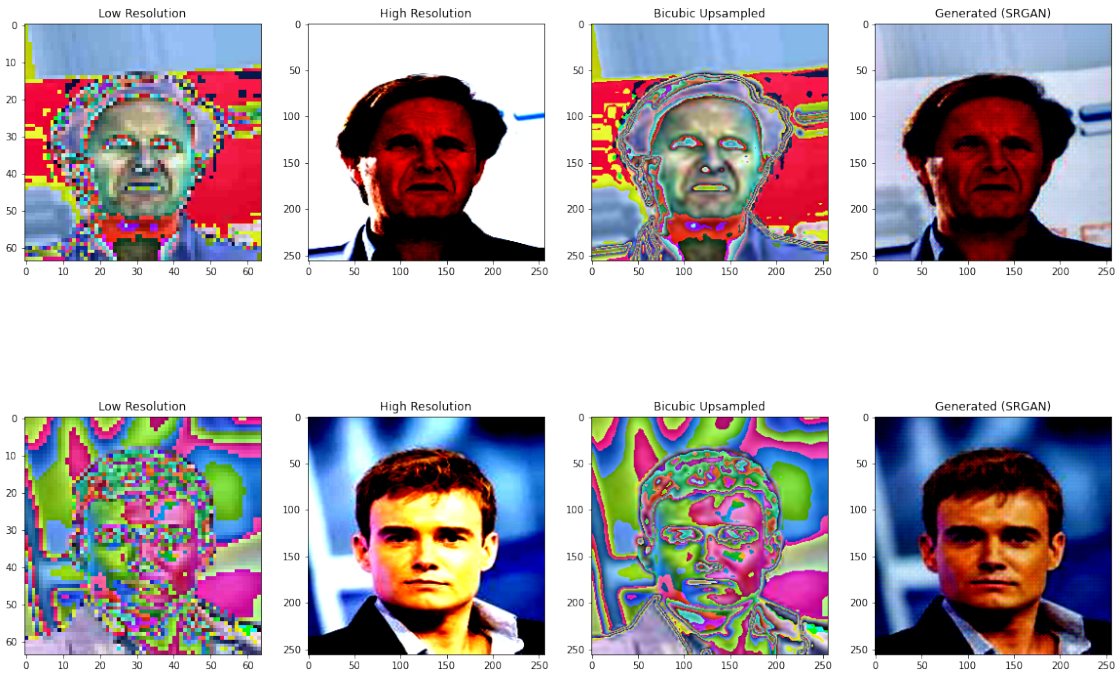


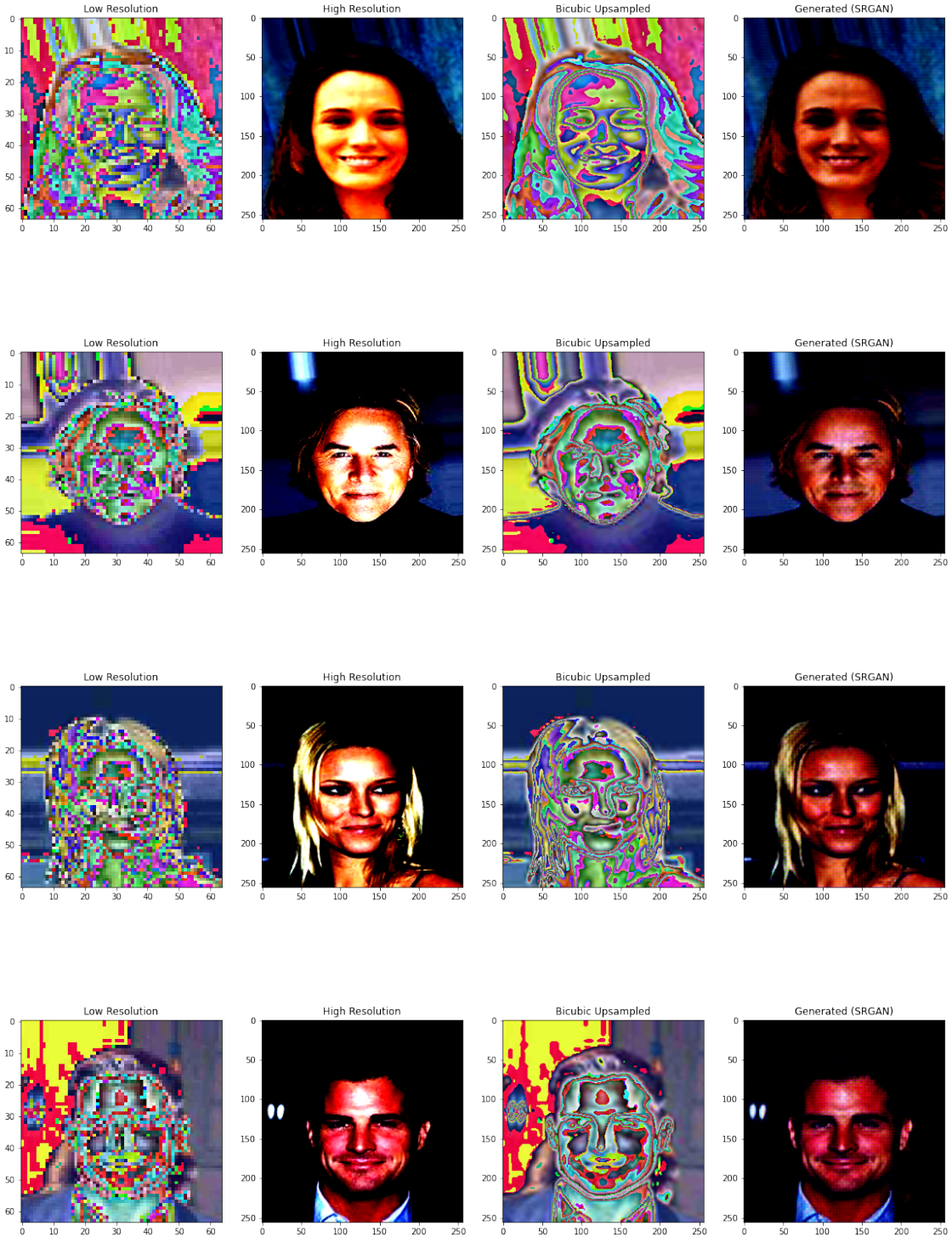

[53]:
```
from skimage.metrics import peak_signal_noise_ratio
import pandas as pd

psnr_details = []

for epoch in range(n_epochs):
    tqdm_bar = tqdm(test_dataloader, desc=f'Testing Epoch {epoch} ',␣
 ↪total=int(len(test_dataloader)))
    for batch_idx, imgs in enumerate(tqdm_bar):
        generator.eval(); discriminator.eval()
        # Configure model input
        imgs_lr = Variable(imgs["lr"].type(Tensor))
        imgs_hr = Variable(imgs["hr"].type(Tensor))
        # Adversarial ground truths
        valid = Variable(Tensor(np.ones((imgs_lr.size(0), *discriminator.
 ↪output_shape))), requires_grad=False)
        fake = Variable(Tensor(np.zeros((imgs_lr.size(0), *discriminator.
 ↪output_shape))), requires_grad=False)

        ### Eval Generator
        # Generate a high resolution image from low resolution input
        gen_hr = generator(imgs_lr)
        # Adversarial loss
        loss_GAN = criterion_GAN(discriminator(gen_hr), valid)
        # Content loss
        gen_features = feature_extractor(gen_hr)
        real_features = feature_extractor(imgs_hr)
```

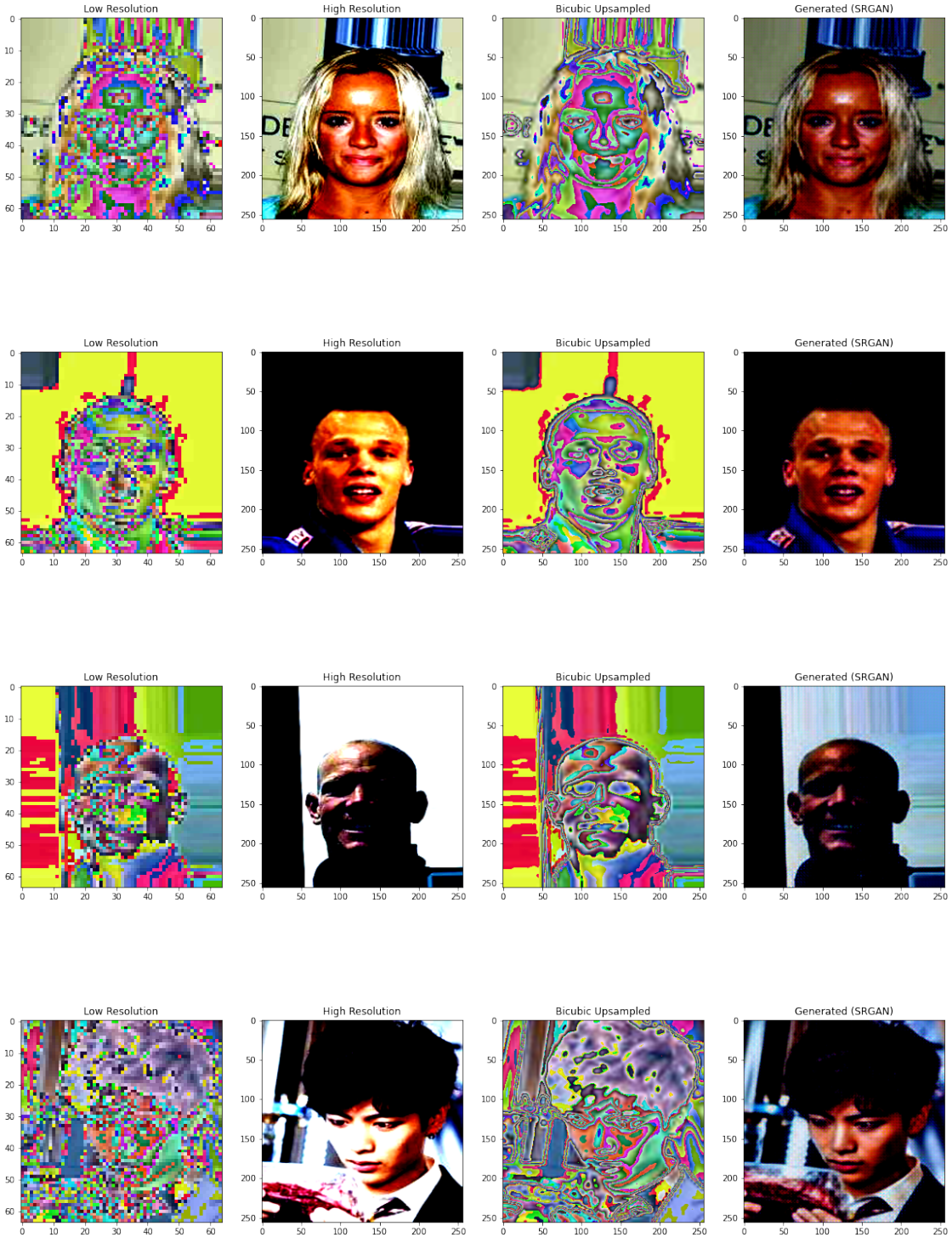
```
        loss_content = criterion_content(gen_features, real_features.detach())
        # Total loss
        loss_G = loss_content + 1e-3 * loss_GAN

        ### Eval Discriminator
        # Loss of real and fake images
        loss_real = criterion_GAN(discriminator(imgs_hr), valid)
        loss_fake = criterion_GAN(discriminator(gen_hr.detach()), fake)
        # Total loss
        loss_D = (loss_real + loss_fake) / 2

        tqdm_bar.set_postfix(gen_loss=loss_G.item(), disc_loss=loss_D.item())

        # Ensure images are in the range [0, 1] before calculating PSNR
        imgs_hr = imgs_hr.clamp(0, 1)
        gen_hr = gen_hr.clamp(0, 1)

        # Calculate PSNR with data_range=1.0
        psnr = peak_signal_noise_ratio(imgs_hr.cpu().detach().numpy(), gen_hr.
↪cpu().detach().numpy(), data_range=1.0)

        # Store details in the list
        psnr_details.append({
            'Epoch': epoch,
            'Batch': batch_idx,
            'PSNR': psnr,
            'HR_Min': imgs_hr.min().item(),
            'HR_Max': imgs_hr.max().item(),
            'Gen_Min': gen_hr.min().item(),
            'Gen_Max': gen_hr.max().item()
        })

        if random.uniform(0, 1) < 0.1:
        # Choose a random image index
            random_index = random.randint(0, len(test_paths) - 1)

    # Bicubic upsampling
            imgs_lr_bicubic = nn.functional.interpolate(imgs_lr,␣
↪scale_factor=4, mode='bicubic')

    # Display the low-resolution, high-resolution, bicubic-upsampled, and␣
↪generated images
            plt.figure(figsize=(20, 5))

            plt.subplot(1, 4, 1)
            plt.imshow(transforms.ToPILImage()(imgs_lr.cpu()[0]))
            plt.title('Low Resolution')
```

```
            plt.subplot(1, 4, 2)
            plt.imshow(transforms.ToPILImage()(imgs_hr.cpu()[0]))
            plt.title('High Resolution')

            plt.subplot(1, 4, 3)
            plt.imshow(transforms.ToPILImage()(imgs_lr_bicubic.cpu()[0]))
            plt.title('Bicubic Upsampled')

            plt.subplot(1, 4, 4)
            plt.imshow(transforms.ToPILImage()(gen_hr.cpu()[0]))
            plt.title('Generated (SRGAN)')

            plt.show()
# Convert the list of dictionaries to a Pandas DataFrame
psnr_df = pd.DataFrame(psnr_details)

# Display the Pandas DataFrame
print(psnr_df)
```

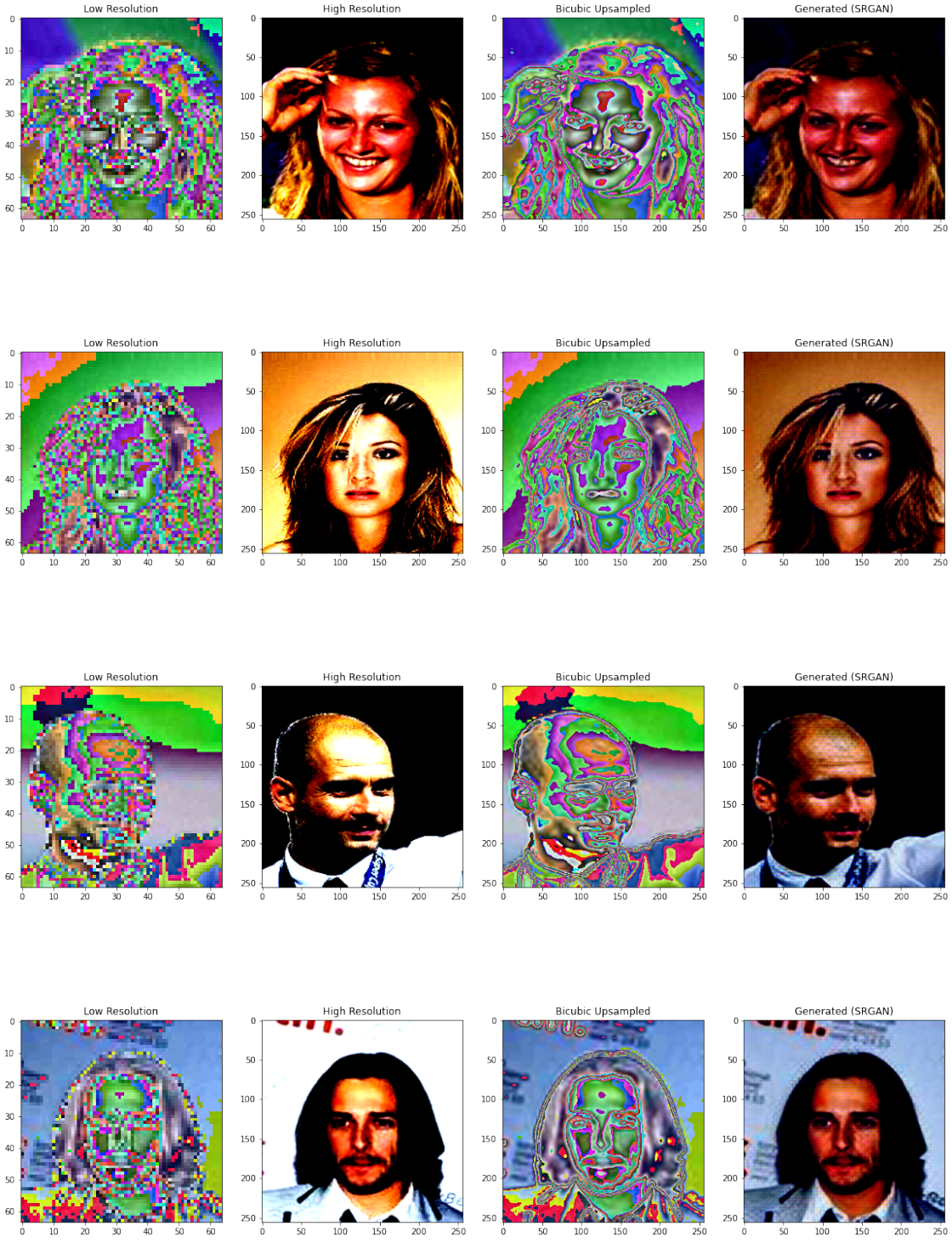
```
HBox(children=(FloatProgress(value=0.0, description='Testing Epoch 0 ', max=338.
  ↪0, style=ProgressStyle(descrip…
```

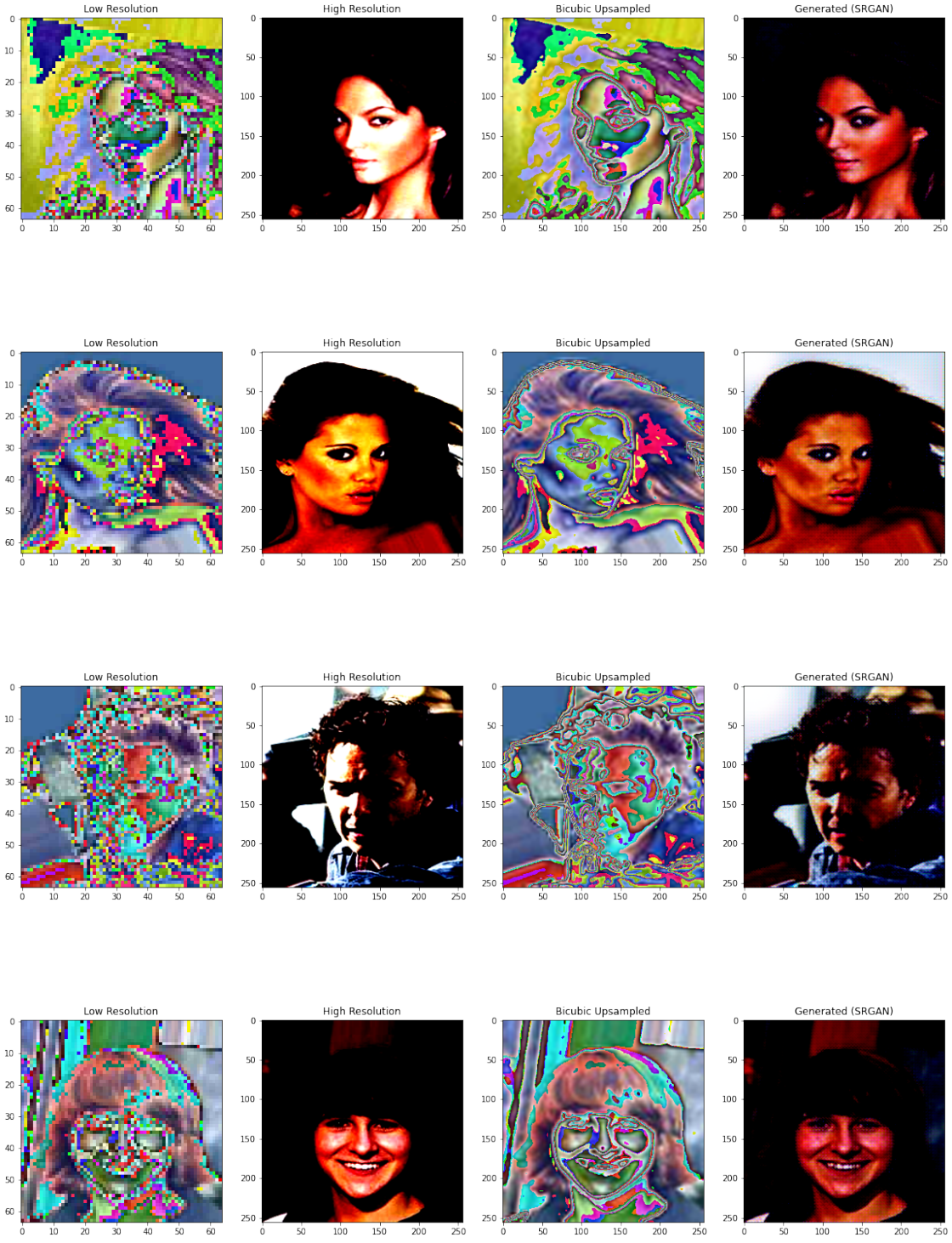


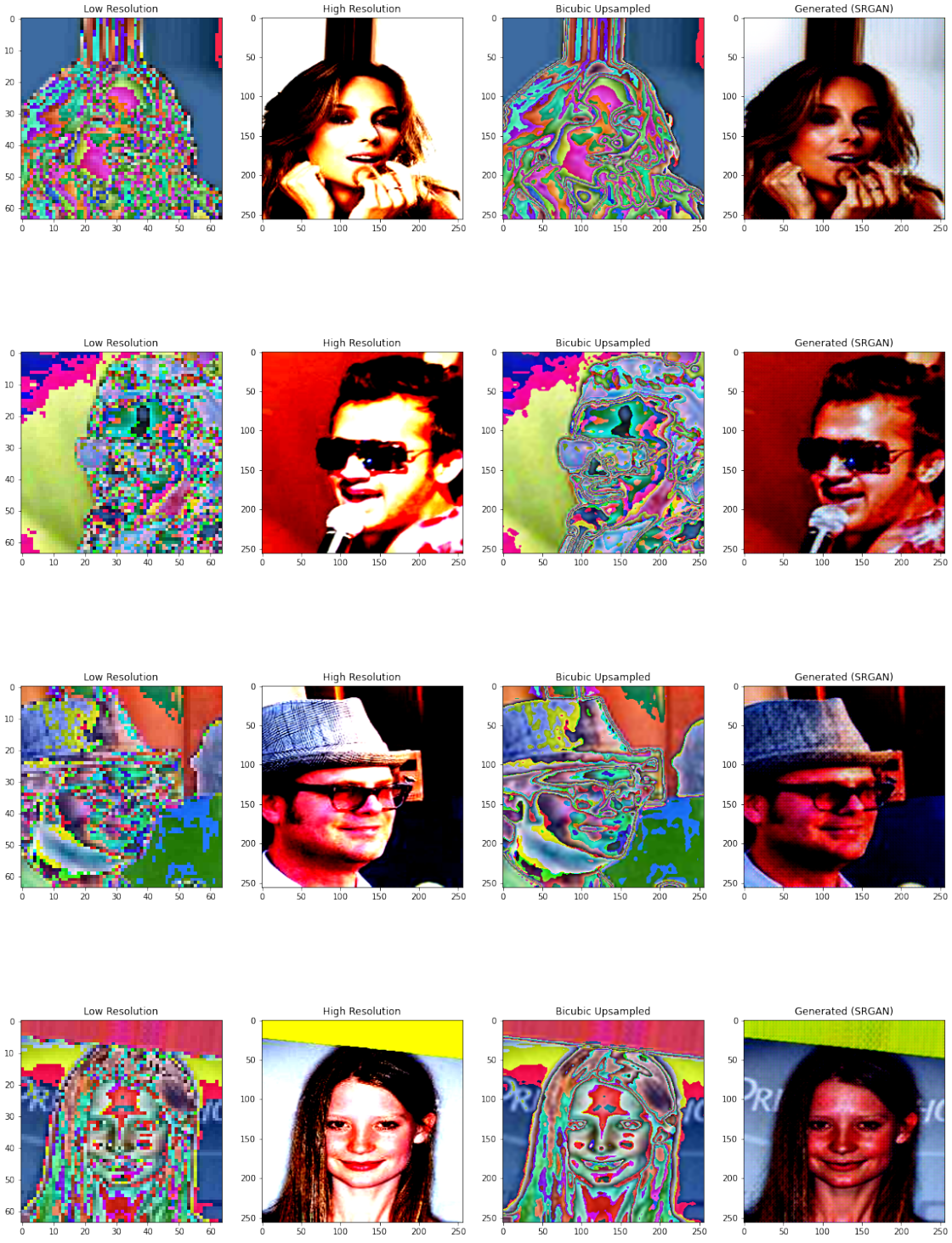

Low Resolution
High Resolution
Bicubic Upsampled
Generated (SRGAN)
Low Resolution
High Resolution
Bicubic Upsampled
Generated (SRGAN)
Low Resolution
High Resolution
Bicubic Upsampled
Generated (SRGAN)
Low Resolution
High Resolution
Bicubic Upsampled
Generated (SRGAN)

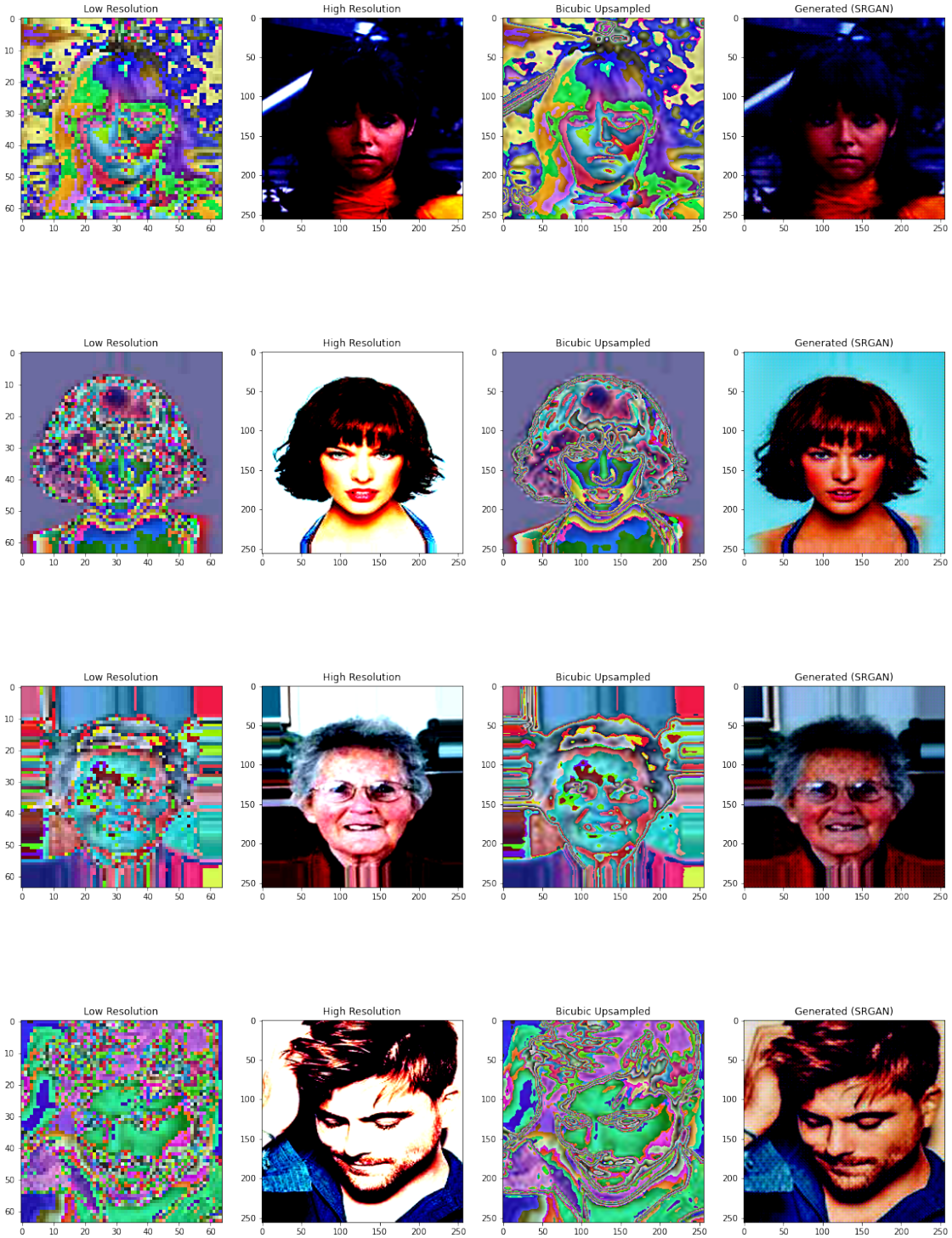

Low Resolution
High Resolution
Bicubic Upsampled
Generated (SRGAN)
Low Resolution
High Resolution
Bicubic Upsampled
Generated (SRGAN)
Low Resolution
High Resolution
Bicubic Upsampled
Generated (SRGAN)
Low Resolution
High Resolution
Bicubic Upsampled
Generated (SRGAN)

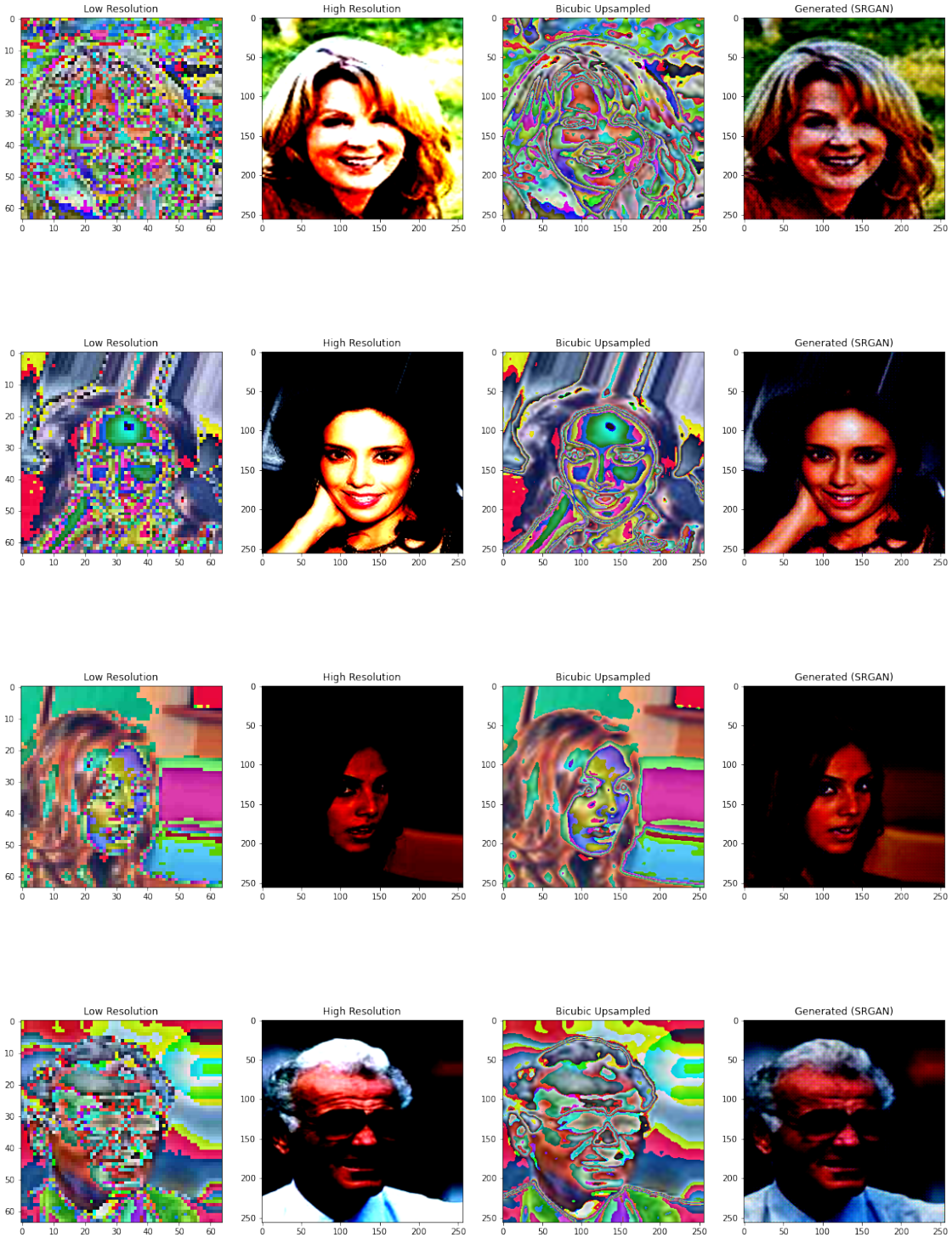

Low Resolution
High Resolution
Bicubic Upsampled
Generated (SRGAN)
Low Resolution
High Resolution
Bicubic Upsampled
Generated (SRGAN)
Low Resolution
High Resolution
Bicubic Upsampled
Generated (SRGAN)
Low Resolution
High Resolution
Bicubic Upsampled
Generated (SRGAN)

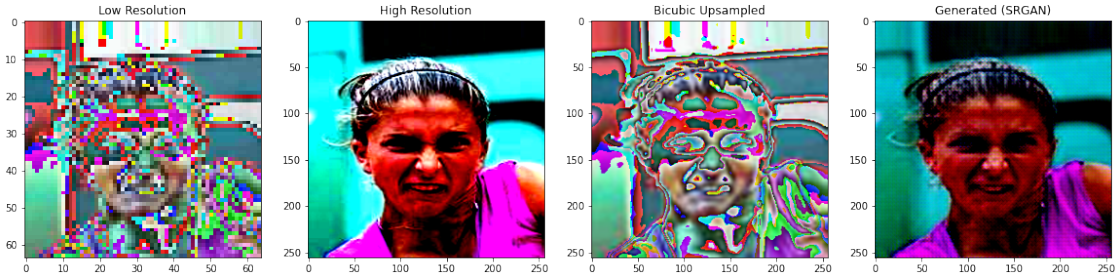

Low Resolution
High Resolution
Bicubic Upsampled
Generated (SRGAN)
Low Resolution
High Resolution
Bicubic Upsampled
Generated (SRGAN)
Low Resolution
High Resolution
Bicubic Upsampled
Generated (SRGAN)
Low Resolution
High Resolution
Bicubic Upsampled
Generated (SRGAN)

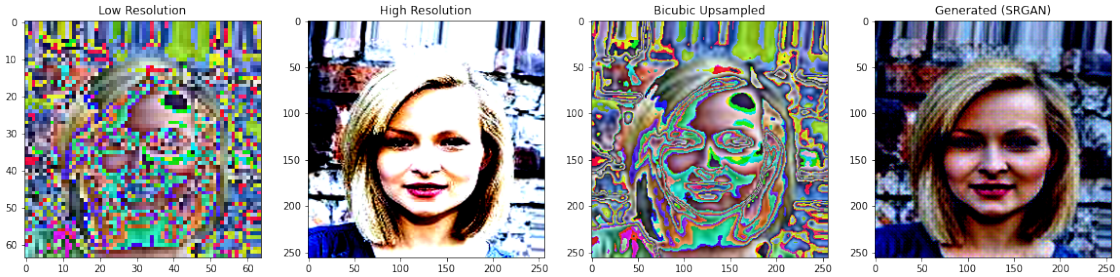

Low Resolution
High Resolution
Bicubic Upsampled
Generated (SRGAN)
Low Resolution
High Resolution
Bicubic Upsampled
Generated (SRGAN)
Low Resolution
High Resolution
Bicubic Upsampled
Generated (SRGAN)
Low Resolution
High Resolution
Bicubic Upsampled
Generated (SRGAN)

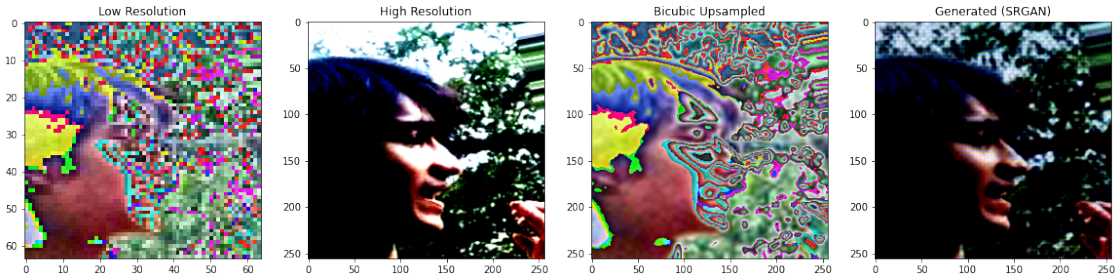

Low Resolution
High Resolution
Bicubic Upsampled
Generated (SRGAN)
Low Resolution
High Resolution
Bicubic Upsampled
Generated (SRGAN)
Low Resolution
High Resolution
Bicubic Upsampled
Generated (SRGAN)
Low Resolution
High Resolution
Bicubic Upsampled
Generated (SRGAN)


Low Resolution
High Resolution
Bicubic Upsampled
Generated (SRGAN)
Low Resolution
High Resolution
Bicubic Upsampled
Generated (SRGAN)
Low Resolution
High Resolution
Bicubic Upsampled
Generated (SRGAN)
Low Resolution
High Resolution
Bicubic Upsampled
Generated (SRGAN)

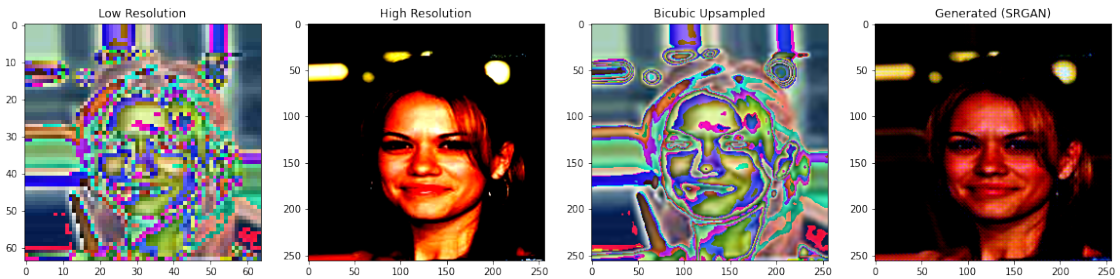

Low Resolution
High Resolution
Bicubic Upsampled
Generated (SRGAN)
Low Resolution
High Resolution
Bicubic Upsampled
Generated (SRGAN)
Low Resolution
High Resolution
Bicubic Upsampled
Generated (SRGAN)
Low Resolution
High Resolution
Bicubic Upsampled
Generated (SRGAN)

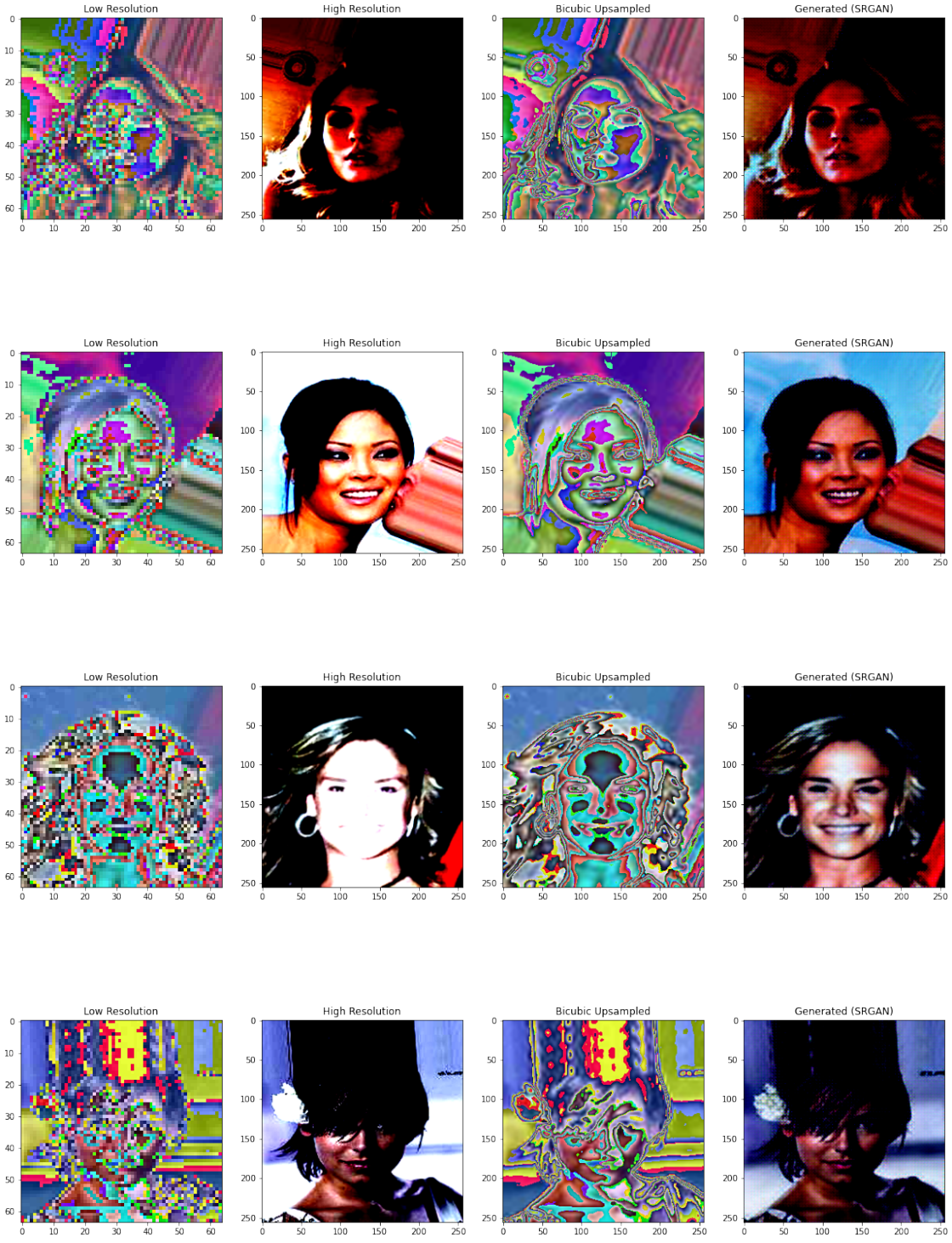


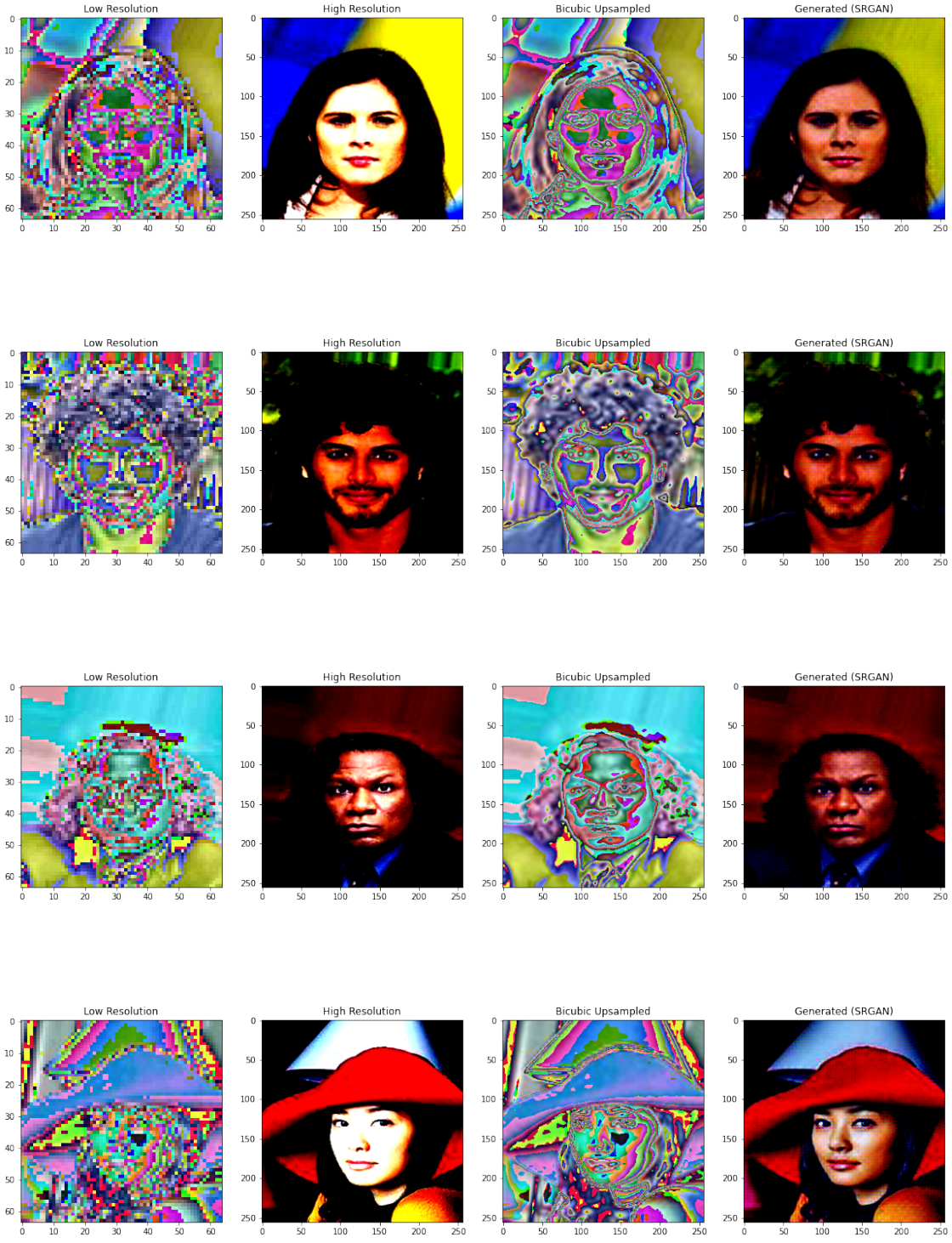



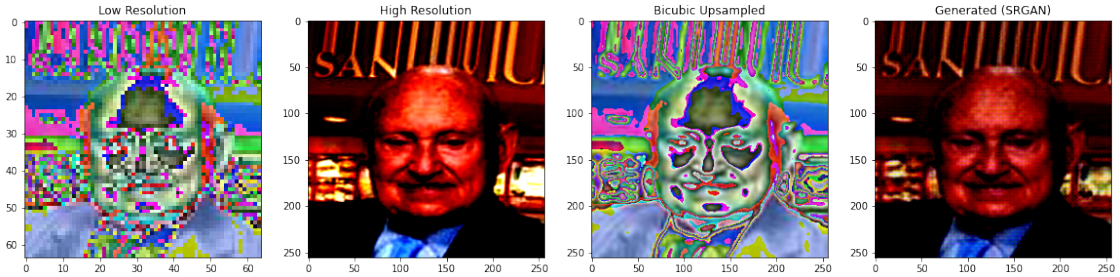

```
HBox(children=(FloatProgress(value=0.0, description='Testing Epoch 1 ', max=338.
↪0, style=ProgressStyle(descrip…
```

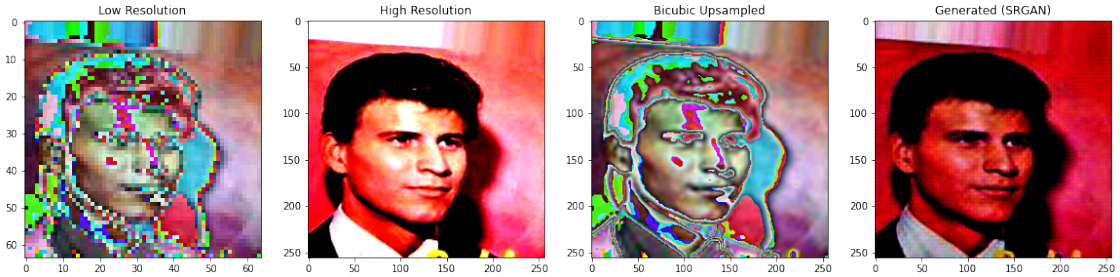


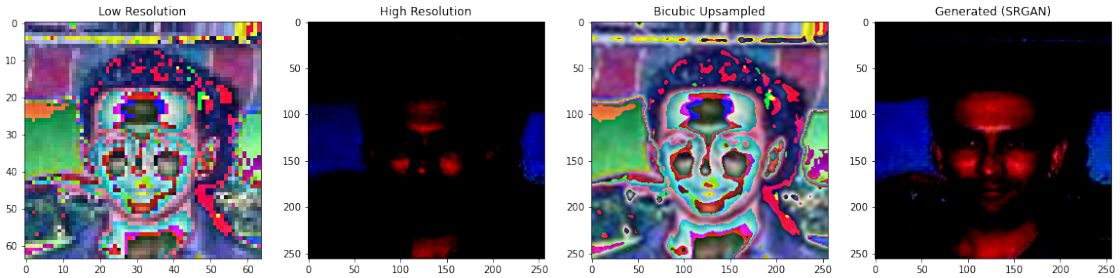

Low Resolution
High Resolution
Bicubic Upsampled
Generated (SRGAN)
Low Resolution
High Resolution
Bicubic Upsampled
Generated (SRGAN)
Low Resolution
High Resolution
Bicubic Upsampled
Generated (SRGAN)
Low Resolution
High Resolution
Bicubic Upsampled
Generated (SRGAN)

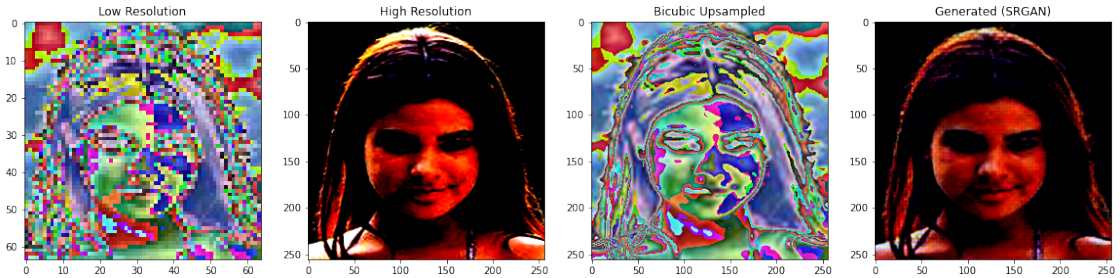

Low Resolution
High Resolution
Bicubic Upsampled
Generated (SRGAN)
Low Resolution
High Resolution
Bicubic Upsampled
Generated (SRGAN)
Low Resolution
High Resolution
Bicubic Upsampled
Generated (SRGAN)
Low Resolution
High Resolution
Bicubic Upsampled
Generated (SRGAN)

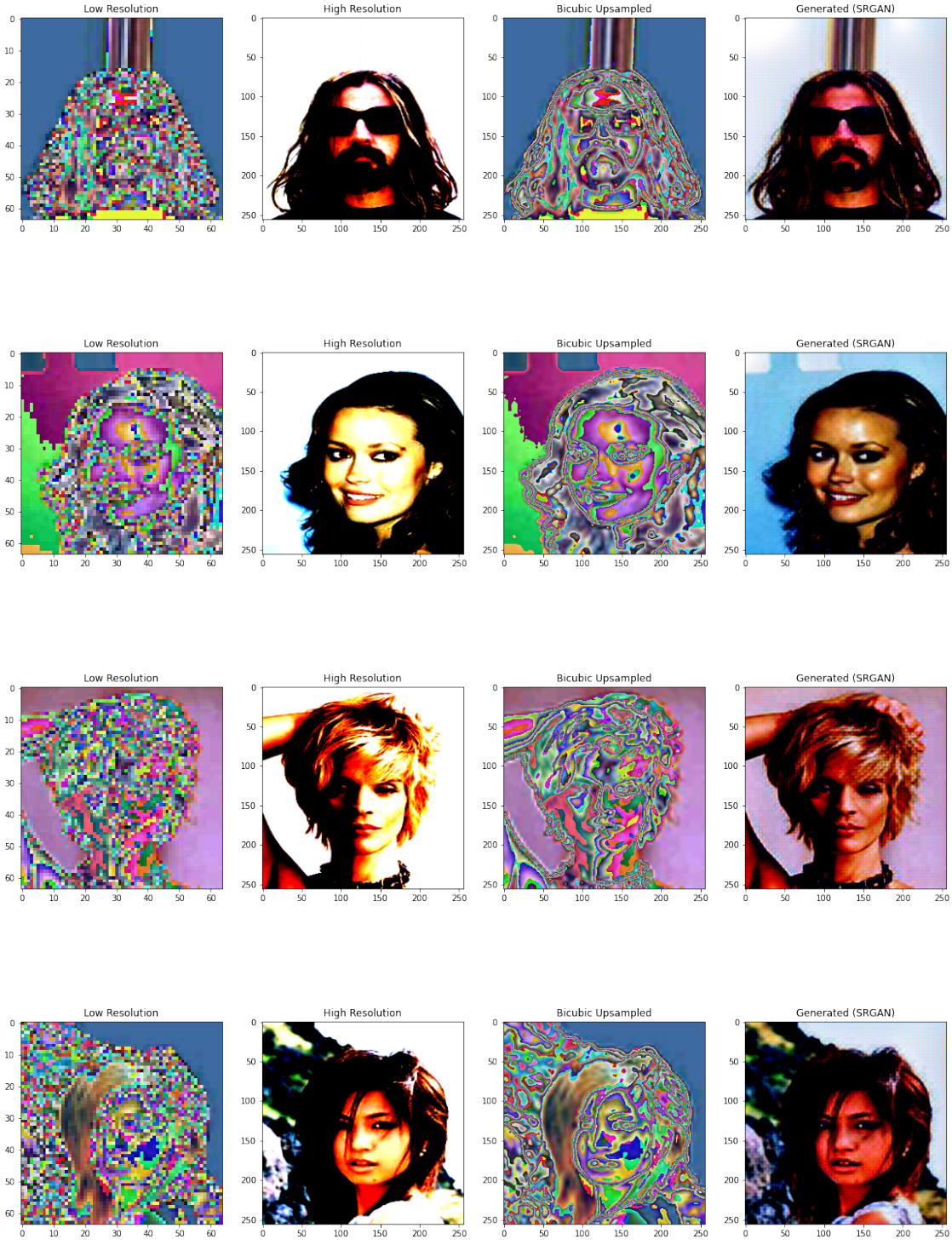

Low Resolution
High Resolution
Bicubic Upsampled
Generated (SRGAN)
Low Resolution
High Resolution
Bicubic Upsampled
Generated (SRGAN)
Low Resolution
High Resolution
Bicubic Upsampled
Generated (SRGAN)
Low Resolution
High Resolution
Bicubic Upsampled
Generated (SRGAN)


Low Resolution
High Resolution
Bicubic Upsampled
Generated (SRGAN)

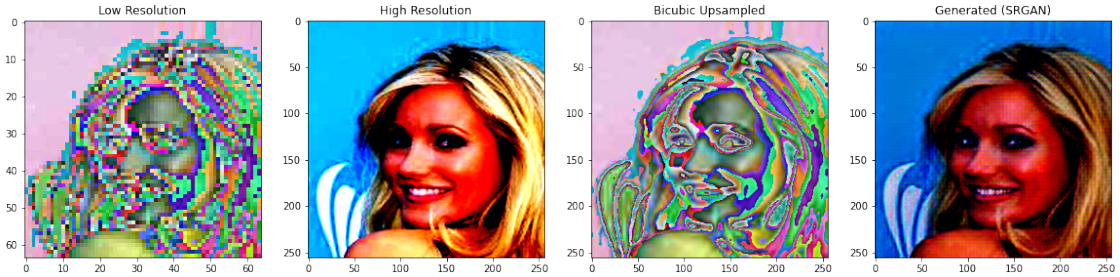

Low Resolution
High Resolution
Bicubic Upsampled
Generated (SRGAN)

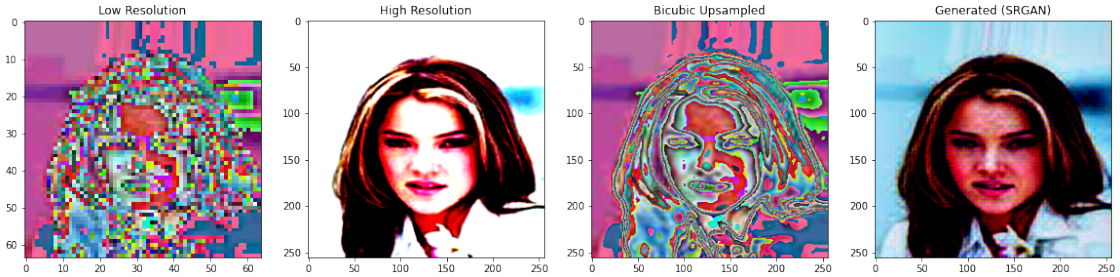

Low Resolution
High Resolution
Bicubic Upsampled
Generated (SRGAN)

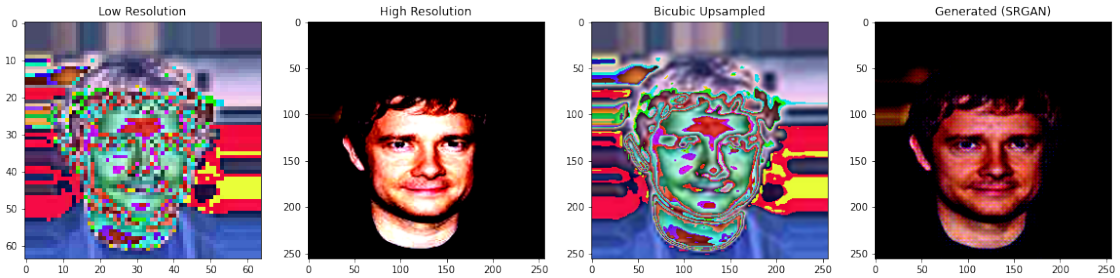

Low Resolution
High Resolution
Bicubic Upsampled
Generated (SRGAN)

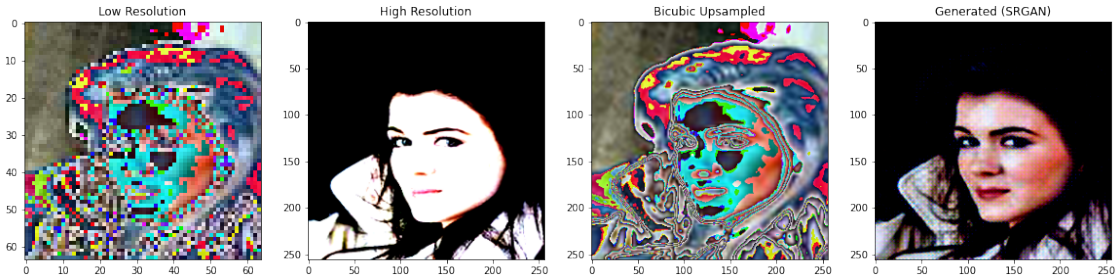

Low Resolution
High Resolution
Bicubic Upsampled
Generated (SRGAN)
Low Resolution
High Resolution
Bicubic Upsampled
Generated (SRGAN)
Low Resolution
High Resolution
Bicubic Upsampled
Generated (SRGAN)
Low Resolution
High Resolution
Bicubic Upsampled
Generated (SRGAN)


Low Resolution
High Resolution
Bicubic Upsampled
Generated (SRGAN)
Low Resolution
High Resolution
Bicubic Upsampled
Generated (SRGAN)
Low Resolution
High Resolution
Bicubic Upsampled
Generated (SRGAN)
Low Resolution
High Resolution
Bicubic Upsampled
Generated (SRGAN)



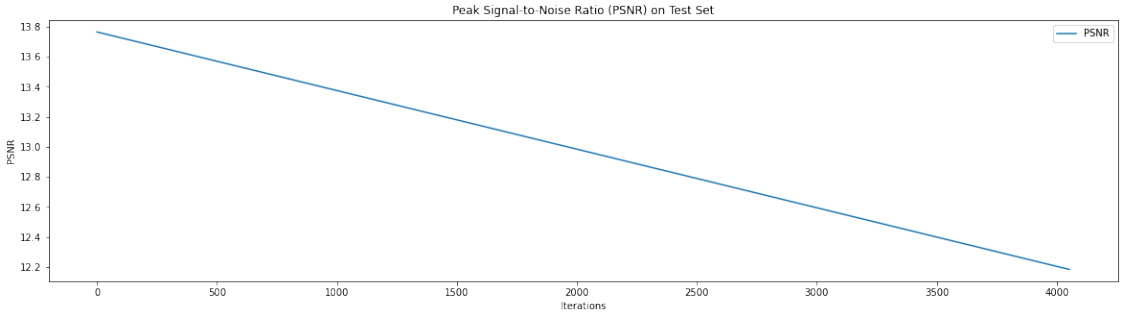




| | Epoch | Batch | PSNR | HR_Min | HR_Max | Gen_Min | Gen_Max |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 12.057614 | 0.0 | 1.0 | 0.0 | 1.000000 |
| 1 | 0 | 1 | 11.630188 | 0.0 | 1.0 | 0.0 | 1.000000 |

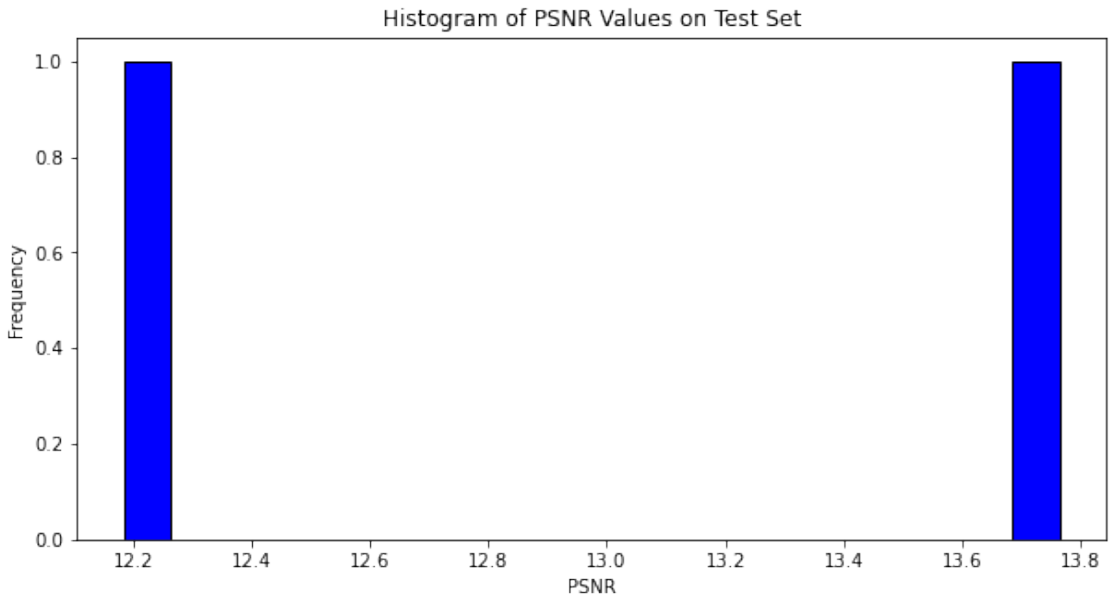
```
2        0      2  10.940614     0.0     1.0      0.0  0.999924
3        0      3  12.354712     0.0     1.0      0.0  1.000000
4        0      4  12.143303     0.0     1.0      0.0  1.000000
..     ...    ...        ...     ...     ...      ...       ...
671      1    333  12.566619     0.0     1.0      0.0  1.000000
672      1    334  11.238737     0.0     1.0      0.0  1.000000
673      1    335  12.206124     0.0     1.0      0.0  0.999938
674      1    336  12.686282     0.0     1.0      0.0  1.000000
675      1    337   9.856307     0.0     1.0      0.0  1.000000

[676 rows x 7 columns]
```

[54]:
```python
import matplotlib.pyplot as plt

# Ensure test_counter and psnr_values have the same length
min_length = min(len(test_counter), len(psnr_values))
test_counter = test_counter[:min_length]
psnr_values = psnr_values[:min_length]


# Adjust the figure size based on the total number of iterations
plt.figure(figsize=(20, 5))

plt.plot(test_counter, psnr_values, label='PSNR')
plt.legend()
plt.title('Peak Signal-to-Noise Ratio (PSNR) on Test Set')
plt.xlabel('Iterations')
plt.ylabel('PSNR')
plt.show()
```




[55]:
```python
plt.figure(figsize=(10, 5))
plt.hist(psnr_values, bins=20, color='blue', edgecolor='black')
plt.title('Histogram of PSNR Values on Test Set')
plt.xlabel('PSNR')
plt.ylabel('Frequency')
```

```
plt.show()
```



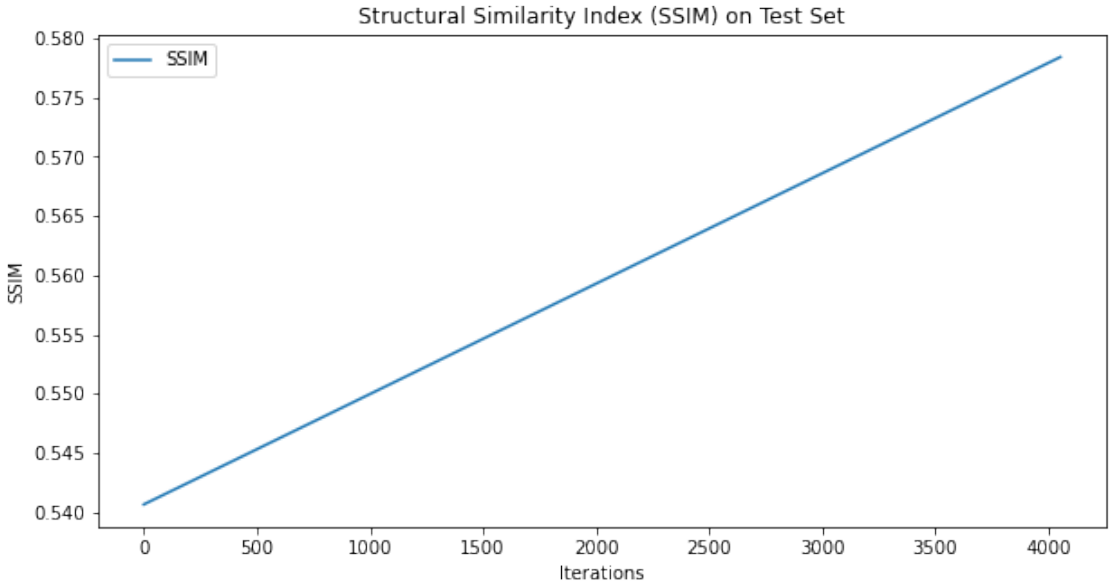

[56]:
```
from skimage.metrics import structural_similarity as ssim
import numpy as np

# ...

# Create lists to store SSIM values
ssim_values = []

for epoch in range(n_epochs):
    tqdm_bar = tqdm(test_dataloader, desc=f'Testing Epoch {epoch} ',␣
 ↪total=int(len(test_dataloader)))
    for batch_idx, imgs in enumerate(tqdm_bar):
        generator.eval(); discriminator.eval()
        # Configure model input
        imgs_lr = Variable(imgs["lr"].type(Tensor))
        imgs_hr = Variable(imgs["hr"].type(Tensor))
        # Adversarial ground truths
        valid = Variable(Tensor(np.ones((imgs_lr.size(0), *discriminator.
 ↪output_shape))), requires_grad=False)
        fake = Variable(Tensor(np.zeros((imgs_lr.size(0), *discriminator.
 ↪output_shape))), requires_grad=False)

        ### Eval Generator
        # Generate a high resolution image from low resolution input
        gen_hr = generator(imgs_lr)
```

```
        # Ensure images are in the range [0, 1]
        imgs_hr = imgs_hr.clamp(0, 1)
        gen_hr = gen_hr.clamp(0, 1)

        # Convert PyTorch tensors to NumPy arrays and change channel order if␣
↪necessary
        img1 = np.transpose(imgs_hr.cpu().detach().numpy(), (0, 2, 3, 1))[0]
        img2 = np.transpose(gen_hr.cpu().detach().numpy(), (0, 2, 3, 1))[0]

        # Calculate SSIM
        ssim_value, _ = ssim(img1, img2, full=True, multichannel=True)

        # Append SSIM value to the list
        ssim_values.append(ssim_value)

        tqdm_bar.set_postfix(gen_loss=loss_G.item(), disc_loss=loss_D.item(),␣
↪ssim=ssim_value)
```

```
HBox(children=(FloatProgress(value=0.0, description='Testing Epoch 0 ', max=338.
↪0, style=ProgressStyle(descrip…
```

```
HBox(children=(FloatProgress(value=0.0, description='Testing Epoch 1 ', max=338.
↪0, style=ProgressStyle(descrip…
```

[57]:
```
min_length_ssim = min(len(test_counter), len(ssim_values))
test_counter_ssim = test_counter[:min_length_ssim]
ssim_values = ssim_values[:min_length_ssim]

# Plotting SSIM values
plt.figure(figsize=(10, 5))
plt.plot(test_counter_ssim, ssim_values, label='SSIM')
plt.legend()
plt.title('Structural Similarity Index (SSIM) on Test Set')
plt.xlabel('Iterations')
plt.ylabel('SSIM')
plt.show()
```


Structural Similarity Index (SSIM) on Test Set
SSIM
0.580
0.575
0.570
0.565
0.560
0.555
0.550
0.545
0.540
SSIM
0
500
1000
1500
2000
2500
3000
3500
4000
Iterations