# notebook6a77936090

October 6, 2024

[321]:
```
# This Python 3 environment comes with many helpful analytics libraries␣
 ↪installed
# It is defined by the kaggle/python Docker image: https://github.com/kaggle/
 ↪docker-python
# For example, here's several helpful packages to load

import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)

# Input data files are available in the read-only "../input/" directory
# For example, running this (by clicking run or pressing Shift+Enter) will list␣
 ↪all files under the input directory

import os
for dirname, _, filenames in os.walk('/kaggle/input'):
    for filename in filenames:
        print(os.path.join(dirname, filename))

# You can write up to 20GB to the current directory (/kaggle/working/) that␣
 ↪gets preserved as output when you create a version using "Save & Run All"
# You can also write temporary files to /kaggle/temp/, but they won't be saved␣
 ↪outside of the current session
```

```
/kaggle/input/image-
colorization/inception_resnet_v2_weights_tf_dim_ordering_tf_kernels.h5

/kaggle/input/image-colorization/ab/ab/ab3.npy

/kaggle/input/image-colorization/ab/ab/ab1.npy

/kaggle/input/image-colorization/ab/ab/ab2.npy

/kaggle/input/image-colorization/l/gray_scale.npy
```

[322]:
```
!pip install torchsummary
```

```
Requirement already satisfied: torchsummary in /opt/conda/lib/python3.10/site-
packages (1.5.1)
```

[323]:
```
import os

from pathlib import Path

import glob

import gc

import cv2

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import matplotlib

matplotlib.style.use('fivethirtyeight')
%matplotlib inline

import PIL

from PIL import Image

from skimage.color import rgb2lab, lab2rgb

import torch

from torch import nn,optim

from torchvision import transforms

from torch.utils.data import Dataset, DataLoader

from torch.autograd import Variable

from torchvision import models

from torch.nn import functional as F

import torch.utils.data

from torchvision.models.inception import inception_v3

from scipy.stats import entropy
```

```
from torchsummary import summary

from tqdm import tqdm
```

[324]:
```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

[325]:
```
ab_path = "/kaggle/input/image-colorization/ab/ab/ab1.npy"

L_path = "/kaggle/input/image-colorization/l/gray_scale.npy"
```

[326]:
```
ab_df = np.load(ab_path)[0:100]

L_df = np.load(L_path)[0:100]

dataset = (L_df,ab_df)
gc.collect()
```

[326]:
```
126596
```

[327]:
```
class ImageColorizationDataset(Dataset):
    ''' Black and White (L) Images and corresponding A&B Colors'''
    def __init__(self, dataset, transform=None):
        '''
        :param dataset: Tuple containing L and ab datasets.
        :param transform: Optional transform to be applied on sample
        '''
        self.dataset = dataset
        self.transform = transform

    def __len__(self):
        return len(self.dataset[0])

    def __getitem__(self, idx):
        L = np.array(self.dataset[0][idx]).reshape((224, 224, 1))
        L = transforms.ToTensor()(L)

        ab = np.array(self.dataset[1][idx])
        ab = transforms.ToTensor()(ab)

        return ab, L

# Example usage
batch_size = 5

# Prepare the Datasets
train_dataset = ImageColorizationDataset(dataset=(L_df,ab_df))
test_dataset = ImageColorizationDataset(dataset=(L_df,ab_df))
```

```
# Build DataLoaders
train_loader = DataLoader(dataset=train_dataset, batch_size=batch_size,␣
 ↪shuffle=True,num_workers=2)
test_loader = DataLoader(dataset=test_dataset, batch_size=batch_size,␣
 ↪shuffle=False,num_workers=2)
```

[ ]:
```
class UNet(nn.Module):

    def unet_conv(self, ch_in, ch_out, is_leaky, dropout_rate=0):
        if is_leaky:
            return nn.Sequential(
                nn.Conv2d(ch_in, ch_out, 3, padding=1),
                nn.BatchNorm2d(ch_out),
                nn.LeakyReLU(0.2),
                nn.Dropout(dropout_rate),
                nn.Conv2d(ch_out, ch_out, 3, padding=1),
                nn.BatchNorm2d(ch_out),
                nn.LeakyReLU(0.2),
                nn.Dropout(dropout_rate)
            )
        else:
            return nn.Sequential(
                nn.Conv2d(ch_in, ch_out, 3, padding=1),
                nn.BatchNorm2d(ch_out),
                nn.ReLU(),
                nn.Dropout(dropout_rate),
                nn.Conv2d(ch_out, ch_out, 3, padding=1),
                nn.BatchNorm2d(ch_out),
                nn.ReLU(),
                nn.Dropout(dropout_rate)
            )

    def up(self, ch_in, ch_out):
        return nn.Sequential(
            nn.ConvTranspose2d(ch_in, ch_out, 2, stride=2),
            nn.ReLU()
        )

    def __init__(self, is_leaky, dropout_rate=0.2):
        super(UNet, self).__init__()

        # Add an additional depth layer (2048 channels)
        self.conv1 = self.unet_conv(1, 64, is_leaky, dropout_rate)
        self.conv2 = self.unet_conv(64, 128, is_leaky, dropout_rate)
        self.conv3 = self.unet_conv(128, 256, is_leaky, dropout_rate)
        self.conv4 = self.unet_conv(256, 512, is_leaky, dropout_rate)
```

```
        self.conv5 = self.unet_conv(512, 1024, is_leaky, dropout_rate)
        self.conv6 = self.unet_conv(1024, 2048, is_leaky, dropout_rate)  # New␣
↪additional layer

        self.pool = nn.MaxPool2d(2)

        # Upsampling layers
        self.up1 = self.up(2048, 1024)  # Adjust to match new depth
        self.up2 = self.up(1024, 512)
        self.up3 = self.up(512, 256)
        self.up4 = self.up(256, 128)
        self.up5 = self.up(128, 64)

        # Decoding layers
        self.conv7 = self.unet_conv(2048, 1024, False)  # Adjust for new depth
        self.conv8 = self.unet_conv(1024, 512, False)
        self.conv9 = self.unet_conv(512, 256, False)
        self.conv10 = self.unet_conv(256, 128, False)
        self.conv11 = self.unet_conv(128, 64, False)

        # Output layer
        self.conv12 = nn.Conv2d(64, 2, kernel_size=1)

    def forward(self, x):
        # Encoding path
        x1 = self.conv1(x)
        x2 = self.conv2(self.pool(x1))
        x3 = self.conv3(self.pool(x2))
        x4 = self.conv4(self.pool(x3))
        x5 = self.conv5(self.pool(x4))
        x6 = self.conv6(self.pool(x5))  # New additional encoding layer

        # Decoding path
        x = self.conv7(torch.cat((x5, self.up1(x6)), 1))
        x = self.conv8(torch.cat((x4, self.up2(x)), 1))
        x = self.conv9(torch.cat((x3, self.up3(x)), 1))
        x = self.conv10(torch.cat((x2, self.up4(x)), 1))
        x = self.conv11(torch.cat((x1, self.up5(x)), 1))

        x = self.conv12(x)
        return nn.Tanh()(x)
```

[ ]:
```
class DNet(nn.Module):

    def unet_conv(self, ch_in, ch_out):
        """
        Construct a convolutional unit comprising of two conv layers
```

```
        followed by a batch normalization layer and Leaky ReLU.
        """
        return nn.Sequential(
            nn.Conv2d(ch_in, ch_out, kernel_size=3, padding=1),
            nn.BatchNorm2d(ch_out),
            nn.LeakyReLU(0.2),
            nn.Conv2d(ch_out, ch_out, kernel_size=3, padding=1),
            nn.BatchNorm2d(ch_out),
            nn.LeakyReLU(0.2)
        )

    def __init__(self):
        super(DNet, self).__init__()

        # First layer
        self.conv1 = self.unet_conv(3, 64)
        # Second layer
        self.conv2 = self.unet_conv(64, 128)
        # Third layer
        self.conv3 = self.unet_conv(128, 256)
        # Fourth layer
        self.conv4 = self.unet_conv(256, 512)
        # Fifth layer (new)
        self.conv5 = self.unet_conv(512, 1024)
        # Sixth layer (new)
        self.conv6 = self.unet_conv(1024, 2048)

        # Adaptive average pooling layer
        self.pool = nn.AdaptiveAvgPool2d((1, 1))
        self.fc = nn.Linear(2048, 1)  # Adjusted for 2048 channels after conv6

    def forward(self, x):
        """
        An input tensor of a colored image from either the generator or source
        is accepted and passed through the model. The probability of the image
        belonging to the source domain is returned as the result.
        """
        x1 = self.conv1(x)
        x2 = self.conv2(self.pool(x1))
        x3 = self.conv3(self.pool(x2))
        x4 = self.conv4(self.pool(x3))
        x5 = self.conv5(self.pool(x4))  # Added layer
        x6 = self.conv6(self.pool(x5))  # Added layer

        # Adaptive pooling to ensure output size of (1, 1)
        x = self.pool(x6)  # Output will be [N, 2048, 1, 1]
```

```
        # Flatten to remove spatial dimensions
        x = x.view(x.size(0), -1)  # Shape: [N, 2048]

        # Pass through the final layer
        x = self.fc(x)  # Shape: [N, 1]

        # Apply sigmoid activation to get a probability score
        x = torch.sigmoid(x)  # Shape: [N, 1]

        return x
```

[332]:
```
from skimage import color


def display_progress(cond, real, fake, current_epoch=0, figsize=(20, 15)):
    """
    Display cond (L channel), real (original AB), and generated (fake AB)␣
 ↪images in LAB color space,
    and convert them to RGB for visualization.
    """
    print(f'Epoch: {current_epoch}')

    # Detach from GPU and move to CPU, then convert to numpy
    cond = cond.detach().cpu().numpy()
    real = real.detach().cpu().numpy()
    fake = fake.detach().cpu().numpy()

    # Normalize the L channel (cond) to the LAB scale
    cond = cond * 100  # Scale L from 0 to 100 (expected range in LAB color␣
 ↪space)

    # Normalize AB channels (real and fake) to the LAB scale
    real = (real - 0.5) * 2 * 128  # Scale from [0,1] to [-128,128]
    fake = (fake - 0.5) * 2 * 128

    # **Clamp AB channels** to avoid out-of-range values
    fake = np.clip(fake, -128, 128)  # Limit fake AB values between -128 and 128

    # Create LAB images by concatenating the L (cond) and AB channels (real and␣
 ↪fake)
    real_lab = np.concatenate([cond, real], axis=0).transpose(1, 2, 0)  # LAB␣
 ↪for real
    fake_lab = np.concatenate([cond, fake], axis=0).transpose(1, 2, 0)  # LAB␣
 ↪for fake

    # Expand cond (L channel) to match dimensions of real and fake (add AB␣
 ↪channels as zeros)
```

```
    cond_expanded = np.concatenate([cond, np.zeros((2, 224, 224))], axis=0).
↪transpose(1, 2, 0)  # L + AB=0 for input

    # Convert LAB to RGB
    try:
        cond_rgb = color.lab2rgb(cond_expanded)
        real_rgb = color.lab2rgb(real_lab)
        fake_rgb = color.lab2rgb(fake_lab)
    except ValueError as e:
        print(f"Error converting LAB to RGB: {e}")
        return

    # Plotting
    images = [cond_rgb, real_rgb, fake_rgb]
    titles = ['Input (L)', 'Real (RGB)', 'Generated (RGB)']

    fig, ax = plt.subplots(1, 3, figsize=figsize)
    for idx, img in enumerate(images):
        ax[idx].imshow(img)
        ax[idx].axis("off")
        ax[idx].set_title(titles[idx])
    plt.show()
```

[333]:
```
from torch import cat

# Initialize generator and discriminator
generator = UNet(True)
discriminator = DNet()

# Check if CUDA (GPU) is available
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Move models to the selected device (GPU or CPU)
generator = generator.to(device)
discriminator = discriminator.to(device)

# Define optimizers and loss functions
d_optimizer = optim.Adam(discriminator.parameters(), betas=(0.5, 0.999), lr=0.
↪0002)
g_optimizer = optim.Adam(generator.parameters(), betas=(0.5, 0.999), lr=0.0002)

# Optionally use learning rate scheduler
lr_scheduler_d = torch.optim.lr_scheduler.StepLR(d_optimizer, step_size=10,␣
↪gamma=0.5)
lr_scheduler_g = torch.optim.lr_scheduler.StepLR(g_optimizer, step_size=10,␣
↪gamma=0.5)
```

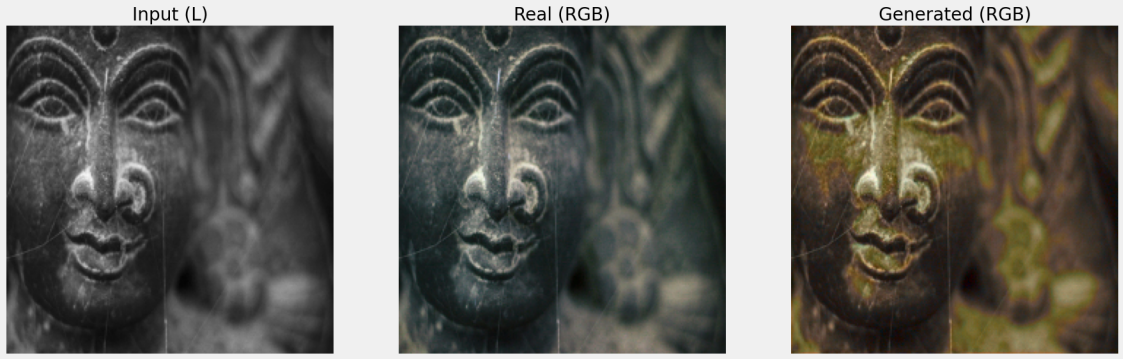
```
d_criterion = nn.BCELoss()  # For discriminator
g_criterion_1 = nn.BCELoss()  # For generator adversarial loss
g_criterion_2 = nn.L1Loss()  # For pixel-wise L1 loss

def train_():
    g_lambda = 100 # Weight for pixel-wise loss
    smooth = 0.1   # Label smoothing

    # Loop over the dataset multiple times.
    for epoch in range(150):
        d_running_loss = 0.0
        g_running_loss = 0.0

        for i, (c_images, l_images) in enumerate(train_loader):
            batch_size = l_images.shape[0]

            # Move to GPU if available
            l_images = l_images.to(device)
            c_images = c_images.to(device)

            # Train the discriminator
            d_optimizer.zero_grad()

            # Generate fake images with generator
            with torch.no_grad():  # No gradients needed for generator here
                fake_images = generator(l_images).detach()

            # Discriminator loss for real and fake images
            logits_real = discriminator(torch.cat([l_images, c_images], 1))
            d_real_loss = d_criterion(logits_real, ((1 - smooth) * torch.
 ↪ones(batch_size, 1)).to(device))

            logits_fake = discriminator(torch.cat([l_images, fake_images], 1))
            d_fake_loss = d_criterion(logits_fake, torch.zeros(batch_size, 1).
 ↪to(device))

            d_loss = d_real_loss + d_fake_loss
            d_loss.backward()
            d_optimizer.step()

            # Train the generator more frequently (e.g., 2 updates per␣
 ↪discriminator update)
            for _ in range(2):
                g_optimizer.zero_grad()
                fake_images = generator(l_images)
```

```
                fake_logits = discriminator(torch.cat([l_images, fake_images],␣
↪1))
                g_fake_loss = g_criterion_1(fake_logits, torch.ones(batch_size,␣
↪1).to(device))

                g_image_distance_loss = g_lambda * g_criterion_2(fake_images,␣
↪c_images)
                g_loss = g_fake_loss + g_image_distance_loss

                g_loss.backward()
                g_optimizer.step()

            # Accumulate loss for printing statistics
            d_running_loss += d_loss.item()
            g_running_loss += g_loss.item()

            # Print statistics at intervals
            if i % 10 == 0:
                print('[%d, %5d] d_loss: %.5f g_loss: %.5f' %
                      (epoch + 1, i + 1, d_running_loss / 10, g_running_loss /␣
↪10))
                d_running_loss = 0.0
                g_running_loss = 0.0

            # Clear cache after each batch
            torch.cuda.empty_cache()

        # Learning rate step
        lr_scheduler_d.step()
        lr_scheduler_g.step()

        # Display progress every 5 epochs
        if (epoch + 1) % 5 == 0:
            display_progress(l_images[0], c_images[0], fake_images[0],␣
↪current_epoch=epoch + 1)

        # Optionally save model state after each epoch
        #torch.save(generator.state_dict(), 'generator.pth')
        #torch.save(discriminator.state_dict(), 'discriminator.pth')
```

[334]:
```
train_()
print('Finished Training')
```

```
[1,     1] d_loss: 0.14570 g_loss: 0.41010

[1,    11] d_loss: 1.41467 g_loss: 2.84493
```

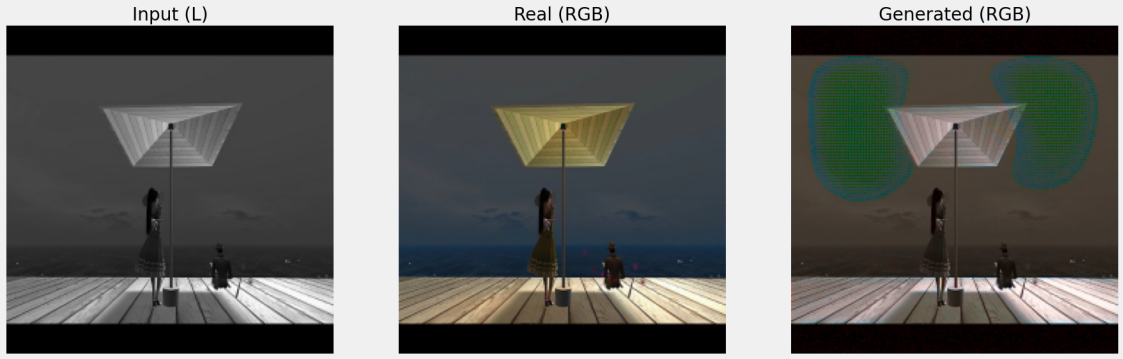
```
[2,     1] d_loss: 0.13816 g_loss: 0.16315
[2,    11] d_loss: 1.37613 g_loss: 1.57859
[3,     1] d_loss: 0.13619 g_loss: 0.14945
[3,    11] d_loss: 1.35468 g_loss: 1.53019
[4,     1] d_loss: 0.13609 g_loss: 0.15540
[4,    11] d_loss: 1.34954 g_loss: 1.48614
[5,     1] d_loss: 0.13535 g_loss: 0.14598
[5,    11] d_loss: 1.35242 g_loss: 1.51359
Epoch: 5
```



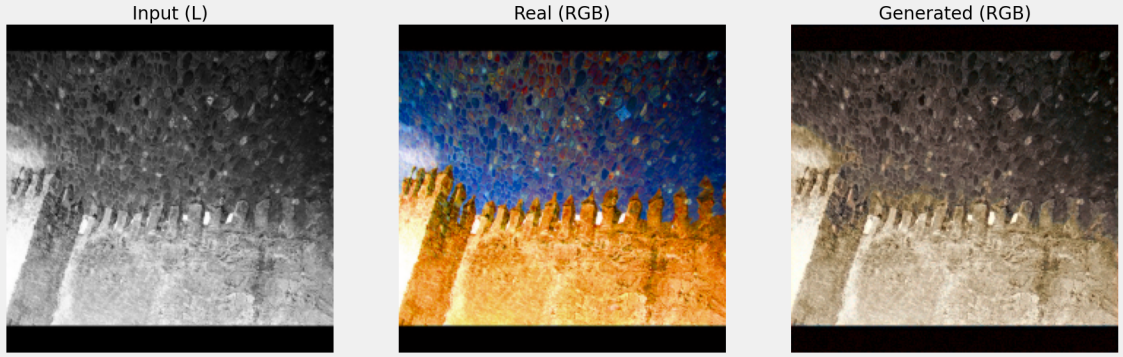

```
[6,     1] d_loss: 0.13498 g_loss: 0.13344
[6,    11] d_loss: 1.34880 g_loss: 1.51783
[7,     1] d_loss: 0.13511 g_loss: 0.14665
[7,    11] d_loss: 1.34375 g_loss: 1.46826
[8,     1] d_loss: 0.13309 g_loss: 0.16157
[8,    11] d_loss: 1.33804 g_loss: 1.46944
[9,     1] d_loss: 0.13207 g_loss: 0.15620
[9,    11] d_loss: 1.33444 g_loss: 1.45921
[10,     1] d_loss: 0.13138 g_loss: 0.15887
```

```
[10,    11] d_loss: 1.33704 g_loss: 1.50379
```

```
Epoch: 10
```

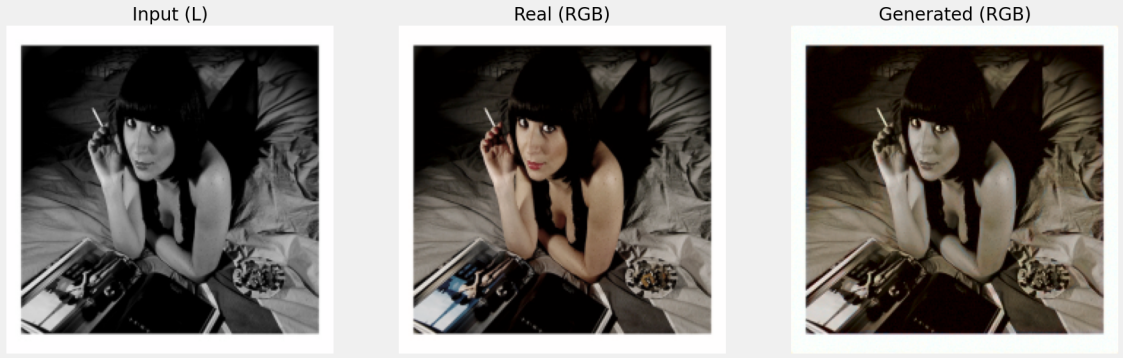



```
[11,     1] d_loss: 0.12981 g_loss: 0.17727
[11,    11] d_loss: 1.32185 g_loss: 1.43124
[12,     1] d_loss: 0.13410 g_loss: 0.12848
[12,    11] d_loss: 1.32248 g_loss: 1.43246
[13,     1] d_loss: 0.13518 g_loss: 0.12501
[13,    11] d_loss: 1.30723 g_loss: 1.48215
[14,     1] d_loss: 0.13403 g_loss: 0.14802
[14,    11] d_loss: 1.29528 g_loss: 1.49004
[15,     1] d_loss: 0.13136 g_loss: 0.12982
[15,    11] d_loss: 1.28008 g_loss: 1.53092
Epoch: 15
```




```
[16,     1] d_loss: 0.12516 g_loss: 0.12186
[16,    11] d_loss: 1.29153 g_loss: 1.50048
[17,     1] d_loss: 0.12494 g_loss: 0.16309
[17,    11] d_loss: 1.27275 g_loss: 1.52337
[18,     1] d_loss: 0.12825 g_loss: 0.13993
[18,    11] d_loss: 1.28679 g_loss: 1.52606
[19,     1] d_loss: 0.13098 g_loss: 0.15220
[19,    11] d_loss: 1.30771 g_loss: 1.44554
[20,     1] d_loss: 0.12562 g_loss: 0.17781
[20,    11] d_loss: 1.28016 g_loss: 1.55147
Epoch: 20
```

```
[21,     1] d_loss: 0.13407 g_loss: 0.12287
[21,    11] d_loss: 1.30997 g_loss: 1.44910
[22,     1] d_loss: 0.13489 g_loss: 0.13610
[22,    11] d_loss: 1.31314 g_loss: 1.48137
[23,     1] d_loss: 0.13059 g_loss: 0.14606
[23,    11] d_loss: 1.29496 g_loss: 1.49000
[24,     1] d_loss: 0.13571 g_loss: 0.11660
[24,    11] d_loss: 1.28823 g_loss: 1.51603
[25,     1] d_loss: 0.13560 g_loss: 0.15169
[25,    11] d_loss: 1.32201 g_loss: 1.48429
Epoch: 25
```



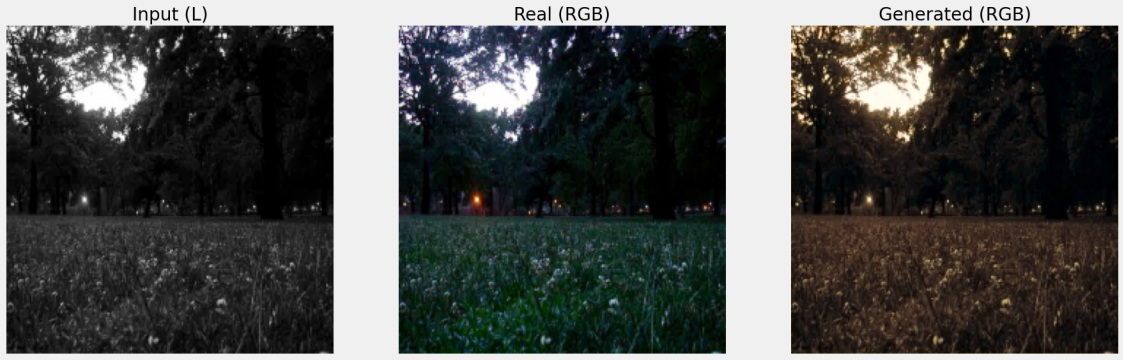

```
[26,     1] d_loss: 0.12368 g_loss: 0.14856
[26,    11] d_loss: 1.30141 g_loss: 1.46803
[27,     1] d_loss: 0.12588 g_loss: 0.15723
[27,    11] d_loss: 1.26681 g_loss: 1.50885
[28,     1] d_loss: 0.12839 g_loss: 0.17616
[28,    11] d_loss: 1.30045 g_loss: 1.50750
[29,     1] d_loss: 0.12757 g_loss: 0.14552
```

```
[29,    11] d_loss: 1.28514 g_loss: 1.50257
[30,     1] d_loss: 0.12374 g_loss: 0.13189
[30,    11] d_loss: 1.28218 g_loss: 1.51217
Epoch: 30
```


```
[31,     1] d_loss: 0.13498 g_loss: 0.13678
[31,    11] d_loss: 1.31502 g_loss: 1.50900
[32,     1] d_loss: 0.12410 g_loss: 0.16320
[32,    11] d_loss: 1.31748 g_loss: 1.48314
[33,     1] d_loss: 0.13328 g_loss: 0.14514
[33,    11] d_loss: 1.28783 g_loss: 1.48102
[34,     1] d_loss: 0.13011 g_loss: 0.13589
[34,    11] d_loss: 1.29947 g_loss: 1.55110
[35,     1] d_loss: 0.13405 g_loss: 0.15874
[35,    11] d_loss: 1.24389 g_loss: 1.57490
Epoch: 35
```

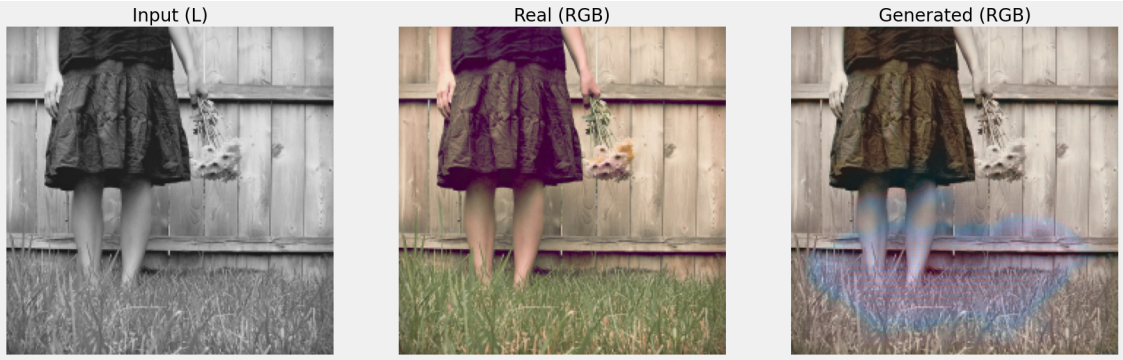

```
[36,     1] d_loss: 0.12869 g_loss: 0.16415
[36,    11] d_loss: 1.25580 g_loss: 1.50630
[37,     1] d_loss: 0.13710 g_loss: 0.13820
[37,    11] d_loss: 1.28852 g_loss: 1.47493
[38,     1] d_loss: 0.13173 g_loss: 0.13737
[38,    11] d_loss: 1.29803 g_loss: 1.47072
[39,     1] d_loss: 0.11962 g_loss: 0.17820
[39,    11] d_loss: 1.21824 g_loss: 1.56577
[40,     1] d_loss: 0.13529 g_loss: 0.12875
[40,    11] d_loss: 1.24566 g_loss: 1.51349
Epoch: 40
```

```
[41,     1] d_loss: 0.12208 g_loss: 0.14519
[41,    11] d_loss: 1.23342 g_loss: 1.59834
[42,     1] d_loss: 0.12370 g_loss: 0.13493
[42,    11] d_loss: 1.26148 g_loss: 1.56868
[43,     1] d_loss: 0.13641 g_loss: 0.15398
[43,    11] d_loss: 1.28132 g_loss: 1.48776
[44,     1] d_loss: 0.12640 g_loss: 0.16428
[44,    11] d_loss: 1.23833 g_loss: 1.54390
[45,     1] d_loss: 0.12558 g_loss: 0.15747
[45,    11] d_loss: 1.26993 g_loss: 1.47252
Epoch: 45
```




```
[46,     1] d_loss: 0.12925 g_loss: 0.13931
[46,    11] d_loss: 1.25353 g_loss: 1.56106
[47,     1] d_loss: 0.12017 g_loss: 0.17794
[47,    11] d_loss: 1.22795 g_loss: 1.57757
[48,     1] d_loss: 0.12184 g_loss: 0.15755
[48,    11] d_loss: 1.22534 g_loss: 1.58663
[49,     1] d_loss: 0.12760 g_loss: 0.14689
```

```
[49,    11] d_loss: 1.21882 g_loss: 1.53010
[50,     1] d_loss: 0.11232 g_loss: 0.18094
[50,    11] d_loss: 1.24158 g_loss: 1.52790
Epoch: 50
```




```
[51,     1] d_loss: 0.11625 g_loss: 0.17425
[51,    11] d_loss: 1.20135 g_loss: 1.59779
[52,     1] d_loss: 0.12458 g_loss: 0.14196
[52,    11] d_loss: 1.21037 g_loss: 1.58971
[53,     1] d_loss: 0.12861 g_loss: 0.14370
[53,    11] d_loss: 1.26014 g_loss: 1.52947
[54,     1] d_loss: 0.12688 g_loss: 0.14180
[54,    11] d_loss: 1.24833 g_loss: 1.52798
[55,     1] d_loss: 0.11505 g_loss: 0.16559
[55,    11] d_loss: 1.20021 g_loss: 1.61232
Epoch: 55
```

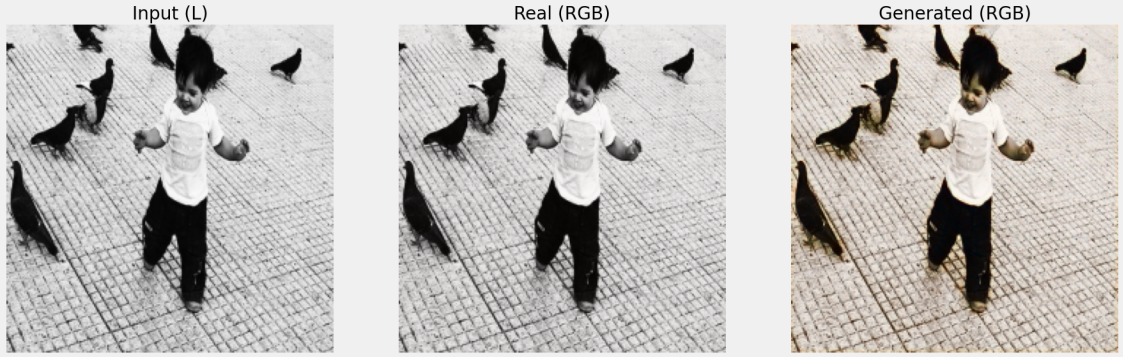

```
[56,     1] d_loss: 0.13261 g_loss: 0.12094
[56,    11] d_loss: 1.20548 g_loss: 1.57479
[57,     1] d_loss: 0.12108 g_loss: 0.17046
[57,    11] d_loss: 1.23140 g_loss: 1.51678
[58,     1] d_loss: 0.12759 g_loss: 0.15605
[58,    11] d_loss: 1.25138 g_loss: 1.52100
[59,     1] d_loss: 0.12010 g_loss: 0.14422
[59,    11] d_loss: 1.23544 g_loss: 1.59589
[60,     1] d_loss: 0.11634 g_loss: 0.16230
[60,    11] d_loss: 1.24147 g_loss: 1.50448
Epoch: 60
```

```
[61,     1] d_loss: 0.11269 g_loss: 0.16579
[61,    11] d_loss: 1.24135 g_loss: 1.55204
[62,     1] d_loss: 0.11505 g_loss: 0.15823
[62,    11] d_loss: 1.22492 g_loss: 1.56559
[63,     1] d_loss: 0.11920 g_loss: 0.15952
[63,    11] d_loss: 1.23588 g_loss: 1.46909
[64,     1] d_loss: 0.13095 g_loss: 0.14622
[64,    11] d_loss: 1.24536 g_loss: 1.50454
[65,     1] d_loss: 0.12953 g_loss: 0.14086
[65,    11] d_loss: 1.18872 g_loss: 1.62955
Epoch: 65
```


```
[66,     1] d_loss: 0.13220 g_loss: 0.13546
[66,    11] d_loss: 1.17890 g_loss: 1.56476
[67,     1] d_loss: 0.11326 g_loss: 0.17825
[67,    11] d_loss: 1.23270 g_loss: 1.49486
[68,     1] d_loss: 0.12793 g_loss: 0.13509
[68,    11] d_loss: 1.27164 g_loss: 1.49153
[69,     1] d_loss: 0.12036 g_loss: 0.15014
```

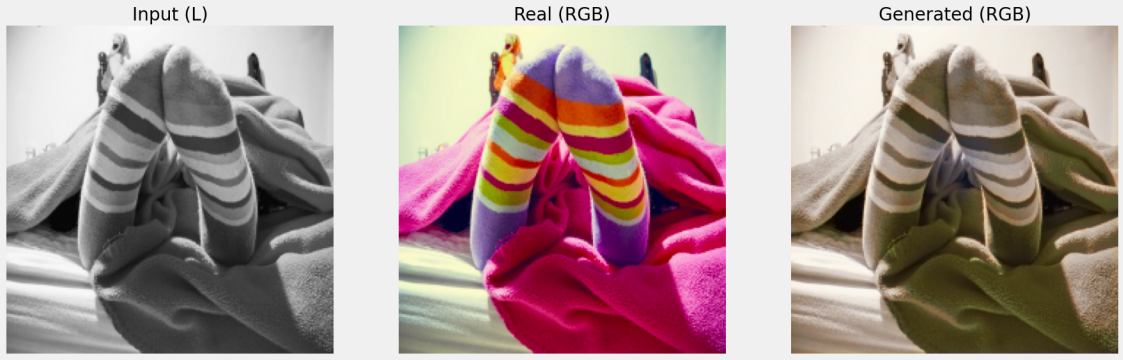
```
[69,    11] d_loss: 1.23524 g_loss: 1.52831
[70,     1] d_loss: 0.11617 g_loss: 0.14880
[70,    11] d_loss: 1.22621 g_loss: 1.58426
Epoch: 70
```


```
[71,     1] d_loss: 0.13911 g_loss: 0.12905
[71,    11] d_loss: 1.20950 g_loss: 1.61240
[72,     1] d_loss: 0.12437 g_loss: 0.14805
[72,    11] d_loss: 1.22600 g_loss: 1.56529
[73,     1] d_loss: 0.11652 g_loss: 0.15592
[73,    11] d_loss: 1.25844 g_loss: 1.55757
[74,     1] d_loss: 0.13835 g_loss: 0.14160
[74,    11] d_loss: 1.21549 g_loss: 1.52458
[75,     1] d_loss: 0.11063 g_loss: 0.18954
[75,    11] d_loss: 1.24811 g_loss: 1.57065
Epoch: 75
```



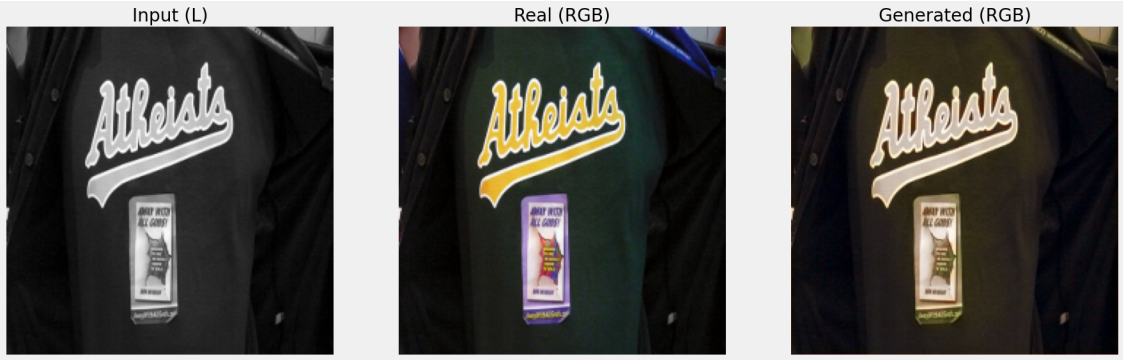

```
[76,     1] d_loss: 0.12161 g_loss: 0.15556
[76,    11] d_loss: 1.20947 g_loss: 1.57578
[77,     1] d_loss: 0.10815 g_loss: 0.16386
[77,    11] d_loss: 1.18250 g_loss: 1.56348
[78,     1] d_loss: 0.13003 g_loss: 0.15005
[78,    11] d_loss: 1.22546 g_loss: 1.55149
[79,     1] d_loss: 0.11652 g_loss: 0.15210
[79,    11] d_loss: 1.20325 g_loss: 1.63850
[80,     1] d_loss: 0.11516 g_loss: 0.15980
[80,    11] d_loss: 1.22875 g_loss: 1.52107
Epoch: 80
```

```
[81,     1] d_loss: 0.12827 g_loss: 0.15021
[81,    11] d_loss: 1.20205 g_loss: 1.60875
[82,     1] d_loss: 0.10914 g_loss: 0.15597
[82,    11] d_loss: 1.23643 g_loss: 1.56129
[83,     1] d_loss: 0.12320 g_loss: 0.14693
[83,    11] d_loss: 1.21837 g_loss: 1.59550
[84,     1] d_loss: 0.11951 g_loss: 0.17131
[84,    11] d_loss: 1.23443 g_loss: 1.55460
[85,     1] d_loss: 0.11932 g_loss: 0.17095
[85,    11] d_loss: 1.23213 g_loss: 1.52752
Epoch: 85
```




```
[86,     1] d_loss: 0.13124 g_loss: 0.13482
[86,    11] d_loss: 1.21629 g_loss: 1.62964
[87,     1] d_loss: 0.11950 g_loss: 0.15561
[87,    11] d_loss: 1.22469 g_loss: 1.57535
[88,     1] d_loss: 0.13792 g_loss: 0.14697
[88,    11] d_loss: 1.24329 g_loss: 1.53013
[89,     1] d_loss: 0.12565 g_loss: 0.14917
```

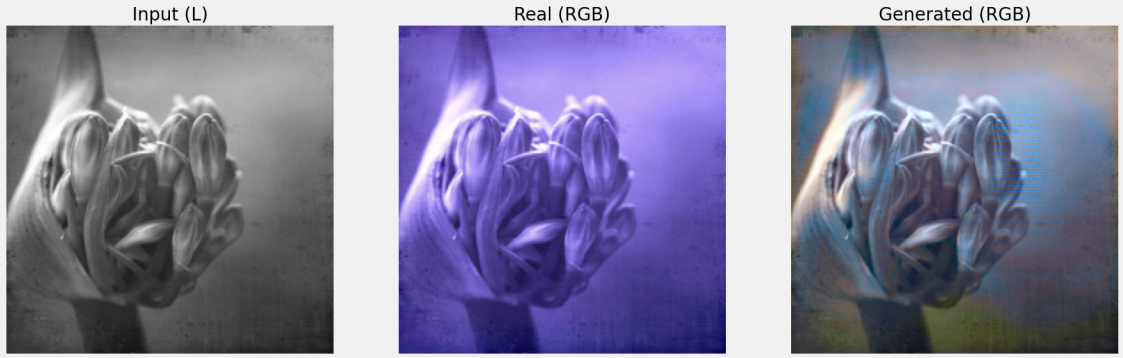
```
[89,    11] d_loss: 1.23592 g_loss: 1.59754
[90,     1] d_loss: 0.11399 g_loss: 0.18693
[90,    11] d_loss: 1.23256 g_loss: 1.57220
Epoch: 90
```




```
[91,     1] d_loss: 0.13447 g_loss: 0.11936
[91,    11] d_loss: 1.24061 g_loss: 1.57395
[92,     1] d_loss: 0.11451 g_loss: 0.16204
[92,    11] d_loss: 1.22127 g_loss: 1.58580
[93,     1] d_loss: 0.11522 g_loss: 0.15526
[93,    11] d_loss: 1.23517 g_loss: 1.56413
[94,     1] d_loss: 0.13141 g_loss: 0.13520
[94,    11] d_loss: 1.26596 g_loss: 1.51894
[95,     1] d_loss: 0.11868 g_loss: 0.16676
[95,    11] d_loss: 1.23552 g_loss: 1.49680
Epoch: 95
```

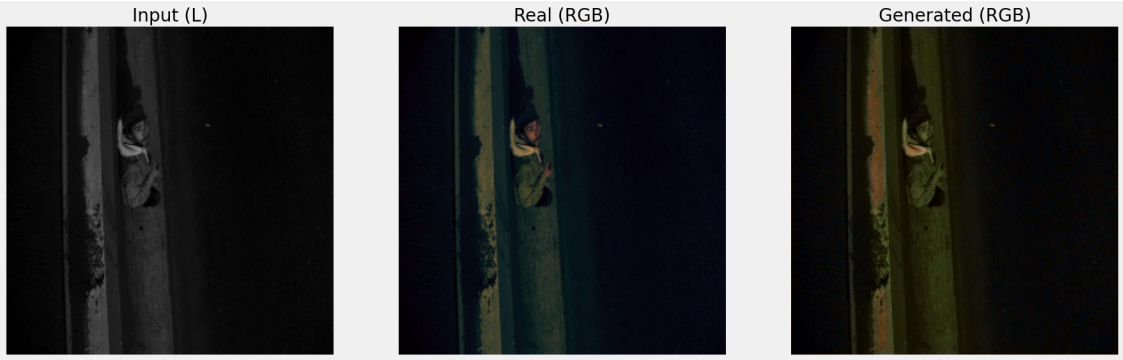

```
[96,     1] d_loss: 0.12551 g_loss: 0.15873
[96,    11] d_loss: 1.24683 g_loss: 1.56940
[97,     1] d_loss: 0.13001 g_loss: 0.16135
[97,    11] d_loss: 1.24494 g_loss: 1.47963
[98,     1] d_loss: 0.11390 g_loss: 0.14855
[98,    11] d_loss: 1.24228 g_loss: 1.56221
[99,     1] d_loss: 0.12761 g_loss: 0.13797
[99,    11] d_loss: 1.24137 g_loss: 1.54484
[100,     1] d_loss: 0.12248 g_loss: 0.16054
[100,    11] d_loss: 1.20020 g_loss: 1.59253
Epoch: 100
```

```
[101,     1] d_loss: 0.11900 g_loss: 0.18064
[101,    11] d_loss: 1.21883 g_loss: 1.57398
[102,     1] d_loss: 0.13841 g_loss: 0.11485
[102,    11] d_loss: 1.24076 g_loss: 1.51356
[103,     1] d_loss: 0.12924 g_loss: 0.12558
[103,    11] d_loss: 1.20590 g_loss: 1.57327
[104,     1] d_loss: 0.12110 g_loss: 0.15976
[104,    11] d_loss: 1.19467 g_loss: 1.59273
[105,     1] d_loss: 0.11486 g_loss: 0.15186
[105,    11] d_loss: 1.23007 g_loss: 1.54706
Epoch: 105
```

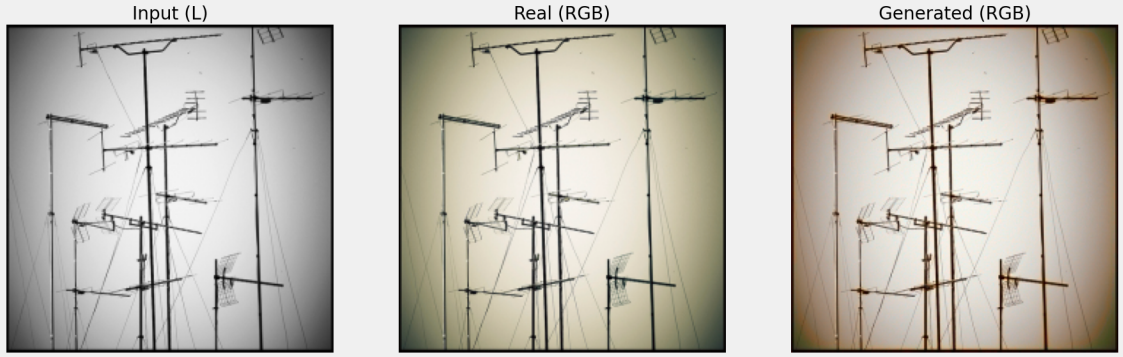



```
[106,     1] d_loss: 0.11884 g_loss: 0.17135
[106,    11] d_loss: 1.22064 g_loss: 1.54960
[107,     1] d_loss: 0.11422 g_loss: 0.18581
[107,    11] d_loss: 1.22643 g_loss: 1.55307
[108,     1] d_loss: 0.11011 g_loss: 0.16108
[108,    11] d_loss: 1.23620 g_loss: 1.58129
[109,     1] d_loss: 0.12332 g_loss: 0.14663
```

```
[109,    11] d_loss: 1.23340 g_loss: 1.52347
[110,     1] d_loss: 0.12773 g_loss: 0.13370
[110,    11] d_loss: 1.27411 g_loss: 1.53201
Epoch: 110
```

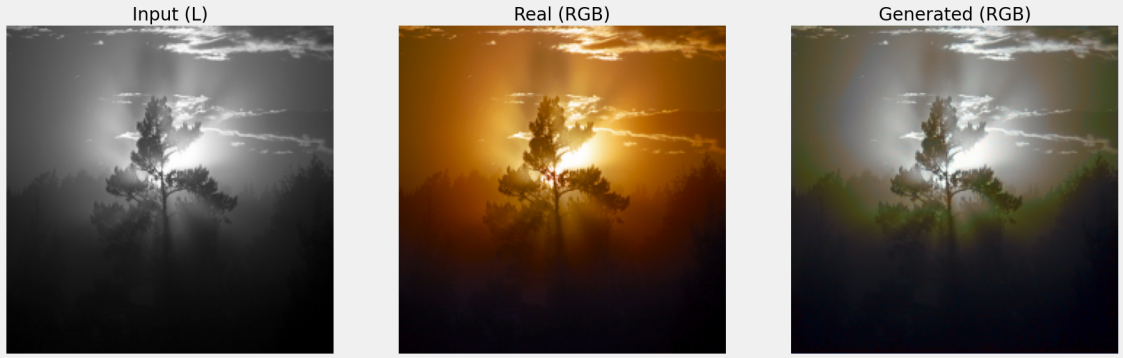



```
[111,     1] d_loss: 0.12963 g_loss: 0.15594
[111,    11] d_loss: 1.23303 g_loss: 1.53288
[112,     1] d_loss: 0.10840 g_loss: 0.15757
[112,    11] d_loss: 1.21477 g_loss: 1.53647
[113,     1] d_loss: 0.12553 g_loss: 0.14962
[113,    11] d_loss: 1.20177 g_loss: 1.56482
[114,     1] d_loss: 0.12700 g_loss: 0.13281
[114,    11] d_loss: 1.20225 g_loss: 1.61406
[115,     1] d_loss: 0.11960 g_loss: 0.16883
[115,    11] d_loss: 1.21355 g_loss: 1.60858
Epoch: 115
```




```
[116,     1] d_loss: 0.11724 g_loss: 0.15033
[116,    11] d_loss: 1.29872 g_loss: 1.47159
[117,     1] d_loss: 0.11859 g_loss: 0.15106
[117,    11] d_loss: 1.23668 g_loss: 1.54646
[118,     1] d_loss: 0.11503 g_loss: 0.18333
[118,    11] d_loss: 1.24650 g_loss: 1.49244
[119,     1] d_loss: 0.12874 g_loss: 0.16829
[119,    11] d_loss: 1.22059 g_loss: 1.60981
[120,     1] d_loss: 0.12528 g_loss: 0.14221
[120,    11] d_loss: 1.26014 g_loss: 1.53562
Epoch: 120
```

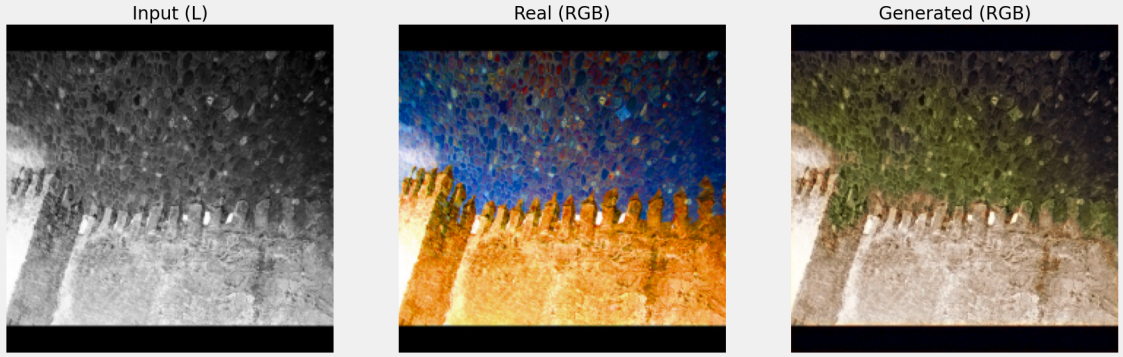
```
[121,     1] d_loss: 0.12838 g_loss: 0.14008
[121,    11] d_loss: 1.18269 g_loss: 1.61561
[122,     1] d_loss: 0.11849 g_loss: 0.15619
[122,    11] d_loss: 1.24202 g_loss: 1.50203
[123,     1] d_loss: 0.11644 g_loss: 0.16619
[123,    11] d_loss: 1.23587 g_loss: 1.52983
[124,     1] d_loss: 0.12403 g_loss: 0.15247
[124,    11] d_loss: 1.20757 g_loss: 1.58314
[125,     1] d_loss: 0.12522 g_loss: 0.15760
[125,    11] d_loss: 1.26575 g_loss: 1.48990
Epoch: 125
```




```
[126,     1] d_loss: 0.12799 g_loss: 0.14878
[126,    11] d_loss: 1.21775 g_loss: 1.57284
[127,     1] d_loss: 0.13287 g_loss: 0.14122
[127,    11] d_loss: 1.21364 g_loss: 1.60940
[128,     1] d_loss: 0.12758 g_loss: 0.15019
[128,    11] d_loss: 1.22437 g_loss: 1.55268
[129,     1] d_loss: 0.13088 g_loss: 0.14121
```

```
[129,    11] d_loss: 1.28478 g_loss: 1.46763
[130,     1] d_loss: 0.11094 g_loss: 0.18139
[130,    11] d_loss: 1.25352 g_loss: 1.53997
Epoch: 130
```


```
[131,     1] d_loss: 0.13249 g_loss: 0.14750
[131,    11] d_loss: 1.22369 g_loss: 1.52366
[132,     1] d_loss: 0.11999 g_loss: 0.17327
[132,    11] d_loss: 1.24871 g_loss: 1.58722
[133,     1] d_loss: 0.11754 g_loss: 0.16356
[133,    11] d_loss: 1.25234 g_loss: 1.53355
[134,     1] d_loss: 0.11557 g_loss: 0.18759
[134,    11] d_loss: 1.22311 g_loss: 1.62774
[135,     1] d_loss: 0.12444 g_loss: 0.14750
[135,    11] d_loss: 1.20709 g_loss: 1.58508
Epoch: 135
```

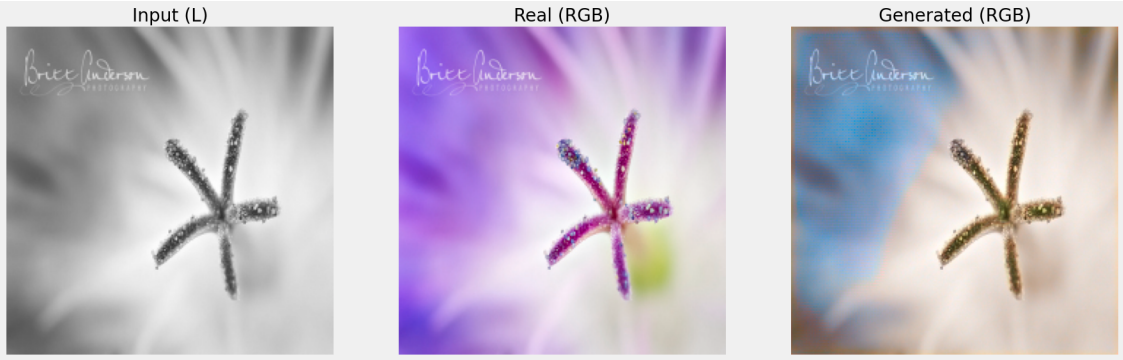



```
[136,     1] d_loss: 0.12601 g_loss: 0.14880
[136,    11] d_loss: 1.24034 g_loss: 1.55587
[137,     1] d_loss: 0.11343 g_loss: 0.18270
[137,    11] d_loss: 1.21697 g_loss: 1.59663
[138,     1] d_loss: 0.13026 g_loss: 0.16267
[138,    11] d_loss: 1.25812 g_loss: 1.54617
[139,     1] d_loss: 0.11621 g_loss: 0.15765
[139,    11] d_loss: 1.23307 g_loss: 1.54434
[140,     1] d_loss: 0.12900 g_loss: 0.14142
[140,    11] d_loss: 1.23169 g_loss: 1.51953
Epoch: 140
```

```
[141,     1] d_loss: 0.12833 g_loss: 0.16170
[141,    11] d_loss: 1.25761 g_loss: 1.48859
[142,     1] d_loss: 0.11999 g_loss: 0.14766
[142,    11] d_loss: 1.27088 g_loss: 1.54241
[143,     1] d_loss: 0.11864 g_loss: 0.15128
[143,    11] d_loss: 1.22575 g_loss: 1.55868
[144,     1] d_loss: 0.13110 g_loss: 0.13492
[144,    11] d_loss: 1.23457 g_loss: 1.56588
[145,     1] d_loss: 0.11822 g_loss: 0.13925
[145,    11] d_loss: 1.22259 g_loss: 1.52402
Epoch: 145
```


```
[146,     1] d_loss: 0.11273 g_loss: 0.15193
[146,    11] d_loss: 1.23397 g_loss: 1.58691
[147,     1] d_loss: 0.12202 g_loss: 0.15379
[147,    11] d_loss: 1.22008 g_loss: 1.55154
[148,     1] d_loss: 0.11842 g_loss: 0.15265
[148,    11] d_loss: 1.20527 g_loss: 1.55918
[149,     1] d_loss: 0.12794 g_loss: 0.14990
```

```
[149,    11] d_loss: 1.21737 g_loss: 1.60289
[150,     1] d_loss: 0.13174 g_loss: 0.14699
[150,    11] d_loss: 1.24608 g_loss: 1.54302
Epoch: 150
```




```
Finished Training
```

[335]:
```
l_criterion=nn.L1Loss()
```

[336]:
```
def eval_():

    # Loop over the dataset multiple times.
    running_loss = 0.0
    num_steps = 0.0

    # Set the model to evaluation mode
    generator.eval()

    with torch.no_grad():  # Disable gradient calculations
        for i, (c_images, l_images) in enumerate(test_loader):
            num_steps += 1
            batch_size = l_images.shape[0]

            # Move to GPU
            l_images = l_images.cuda()
            c_images = c_images.cuda()

            # Generate fake images
            fake_images = generator(l_images)
```

```
            # Calculate the image distance loss (MAE) pixelwise between the␣
↪images
            loss = l_criterion(fake_images, c_images)

            # Detach the loss from the computational graph and move to CPU
            running_loss += loss.detach().cpu().item()

    # Calculate and print the mean absolute error (MAE)
    print('Mean Absolute Error (MAE):', running_loss / num_steps)
```

[337]:
```
eval_()
```

```
Mean Absolute Error (MAE): 0.05456062443554401
```