A Project Report Submitted

for Machine Learning (UML501)

by

Batch 3 COE 2

102103035 Ankit Bhardwaj

102103053 Aaryan Gupta

Submitted to

Dr. Anjula Mehto

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

THAPAR INSTITUTE OF ENGINEERING AND TECHNOLOGY (A DEEMED TO BE UNIVERSITY), PATIALA, PUNJAB

INDIA

August-December 2023

Introduction

In the realm of computer vision, image colorization has emerged as a fascinating and challenging task. With the rapid advancement of machine learning techniques, the ability to automatically add color to grayscale images has become a reality, revolutionizing the way we perceive and interact with visual content.

Color plays a pivotal role in our perception of the world, evoking emotions, conveying information, and enhancing the overall aesthetic appeal of images. However, historical limitations in technology restricted the capturing of color information, leaving us with countless grayscale photographs that fail to fully capture the vibrancy and richness of the original scenes. Image colorization seeks to bridge this gap, enabling us to breathe life into these monochromatic snapshots, bringing them into the realm of vividness and realism.

The journey of colorizing images involves a complex interplay of techniques from computer vision and machine learning. From pixel-level regression to deep learning-based approaches, researchers have tirelessly explored and fine-tuned algorithms to tackle this challenging task. By harnessing the power of convolutional neural networks (CNNs), recurrent neural networks (RNNs), and generative adversarial networks (GANs), image colorization algorithms have achieved remarkable results, surpassing human expectations.

Beyond the realm of art and aesthetics, image colorization has found practical applications in various domains. It has proven to be invaluable in historical photo restoration, aiding in preserving and visualizing moments from the past with enhanced accuracy and realism. Additionally, in fields such as fashion, design, and advertising, image colorization serves as a powerful tool to create captivating visuals that resonate with audiences.

Image colorization represents a fascinating intersection of computer vision and machine learning, enabling us to breathe life into grayscale images and enrich our visual experiences. By understanding the underlying techniques and advancements in this field, we can unlock new possibilities for enhancing the way we perceive and interact with the world of imagery.

Problem Statement

Image colorization involves the task of automatically adding color to grayscale images. It is generally framed as an inverse problem in computer vision, where the missing color of a grayscale pixel needs to be reconstructed by selecting the best color from a set of possibilities. The challenge lies in accurately predicting the appropriate colors for each pixel, considering the content, context, and artistic intent of the original image.

Image colorization is a multimodal problem, as a single grayscale image can correspond to multiple plausible colored images. Traditional approaches often required significant user input alongside the grayscale image, while recent advancements in deep neural networks have shown remarkable success in automatic image colorization without additional human input. These deep learning models leverage semantic information and capture the underlying patterns and features of the image to generate colorized images.

The problem becomes more complex when dealing with textured images, as geometric similarities may not be reliable for colorization. Additionally, the subjective nature of color perception adds another layer of complexity, as there can be multiple valid color combinations for a grayscale image. Efficient and scalable algorithms are required to handle large datasets and real-time applications.

Image colorization involves developing algorithms and models that can accurately predict the appropriate colors for grayscale images, considering the content, context, and artistic intent of the original image. This problem requires addressing the multimodal nature of colorization, leveraging deep learning techniques, handling textured images, and accounting for subjective color perception.

Data Set

We have used a data set that contains RGB images of various scenes/places.

The link to the data set is as follows:

https://drive.google.com/drive/folders/18ODl_6aMBWLxiOoq2U2mTuG_gKtQuxPl

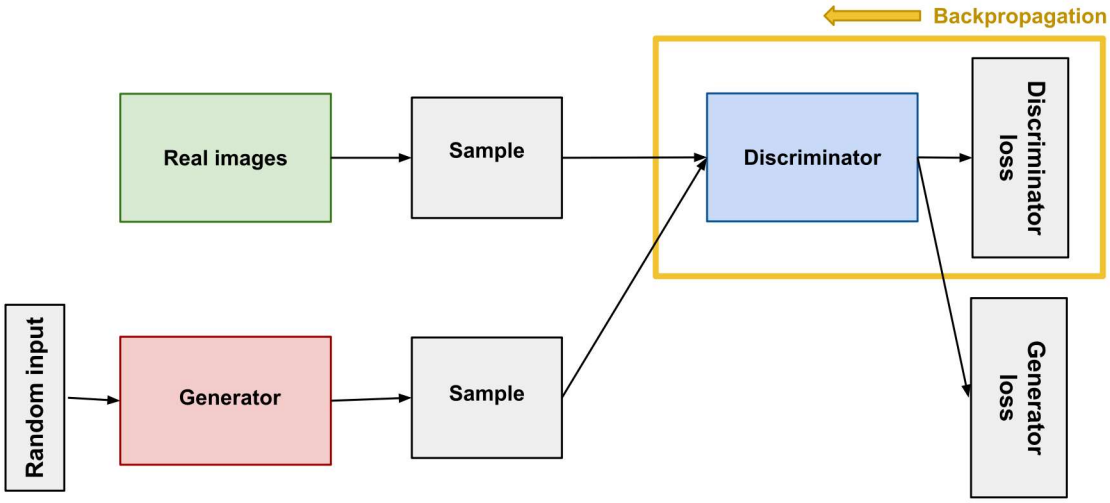
It contains 2791 RGB Images.

This data set is not our own and we came by this data set while researching on the various methods to colorize an image. The owner of the data set has used the data set to train a GAN. We have used this data set to implement the pix2pix method explained in great details in the paper: Isola, P., Zhu, J.-Y., Zhou, T., & Efros, A. A. (2016). Image-to-Image Translation with Conditional Adversarial Networks. Retrieved from [Source Link]

Methodology

The Discriminator

The discriminator learns to distinguish the generator's fake data from real data. The discriminator penalizes the generator for producing implausible results.




```
def disc(img_shape):
  init=RandomNormal(stddev=0.02)
  in_src_img=Input(shape=img_shape)
  in_tar_img=Input(shape=img_shape)
  merged=Concatenate()([in_src_img,in_tar_img])
  d = Conv2D(64, (4,4), strides=(2,2), padding='same', kernel_initializer=init)(merged)
  d = LeakyReLU(alpha=0.2)(d)
  d = Conv2D(128, (4,4), strides=(2,2), padding='same', kernel_initializer=init)(d)
  d = BatchNormalization()(d)
  d = LeakyReLU(alpha=0.2)(d)
  d = Conv2D(256, (4,4), strides=(2,2), padding='same', kernel_initializer=init)(d)
  d = BatchNormalization()(d)
  d = LeakyReLU(alpha=0.2)(d)
  d = Conv2D(512, (4,4), strides=(2,2), padding='same', kernel_initializer=init)(d)
  d = BatchNormalization()(d)
  d = LeakyReLU(alpha=0.2)(d)
  d = Conv2D(512, (4,4), padding='same', kernel_initializer=init)(d)
  d = BatchNormalization()(d)
  d = LeakyReLU(alpha=0.2)(d)
```

```
d = Conv2D(1, (4,4), padding='same', kernel_initializer=init)(d)
patch_out = Activation('sigmoid')(d)
model = Model([in_src_img, in_tar_img], patch_out)
opt = Adam(learning_rate=0.0002, beta_1=0.5)
model.compile(loss='binary_crossentropy', optimizer=opt, loss_weights=[0.5])
return model
```

The function called `disc` defines a discriminator model for image-to-image translation using conditional adversarial networks. The discriminator is responsible for distinguishing between real and generated images.

Here is a breakdown of the code:

1. The function takes `img_shape` as input, which represents the shape of the input images.

2. It initializes the weights of the model using the `RandomNormal` initializer with a standard deviation of 0.02.

3. Two input layers, `in_src_img` and `in_tar_img`, are created with the specified `img_shape`.

4. The input images are concatenated using the `Concatenate` layer.

5. Several convolutional layers with different parameters are added to the model. Each convolutional layer is followed by a batch normalization layer and a leaky ReLU activation function with an alpha value of 0.2.

6. The final convolutional layer outputs a single-channel feature map.

7. The `Activation` layer with the sigmoid activation function is used to produce the final output of the discriminator, which represents the probability of the input image being real or fake.

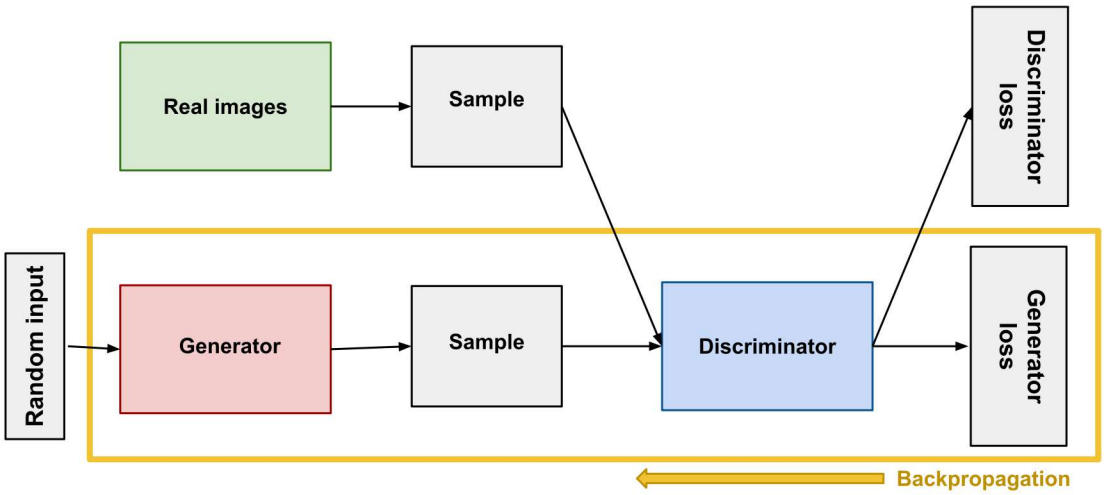
8. The model is defined using the `Model` class from Keras, with the input layers and the output layer.

9. The Adam optimizer with a learning rate of 0.0002 and a beta value of 0.5 is used, and the model is compiled with the binary cross-entropy loss function.

10. The function returns the compiled discriminator model.

The Generator

The generator learns to generate plausible data. The generated instances become negative training examples for the discriminator.




```
def enc(layer_in,n_filters,batchnorm=True):
    init = RandomNormal(stddev=0.02)
    g = Conv2D(n_filters, (4,4), strides=(2,2), padding='same', kernel_initializer=init)(layer_in)
    if batchnorm:
        g = BatchNormalization()(g, training=True)
    g = LeakyReLU(alpha=0.2)(g)
    return g


def dec(layer_in,skip_in,n_filters,dropout=True):
    init = RandomNormal(stddev=0.02)
    g = Conv2DTranspose(n_filters, (4,4), strides=(2,2), padding='same',
kernel_initializer=init)(layer_in)
    g = BatchNormalization()(g, training=True)
    if dropout:
        g = Dropout(0.5)(g, training=True)
    g = Concatenate()([g, skip_in])
    g = Activation('relu')(g)
    return g


def gen(img_shape=(256,256,3)):
    init = RandomNormal(stddev=0.02)
    in_image = Input(shape=img_shape)
    e1 = enc(in_image, 64, batchnorm=False)
```

```
    e2 = enc(e1, 128)
    e3 = enc(e2, 256)
    e4 = enc(e3, 512)
    e5 = enc(e4, 512)
    e6 = enc(e5, 512)
    e7 = enc(e6, 512)
    b = Conv2D(512, (4,4), strides=(2,2), padding='same', kernel_initializer=init)(e7)
    b = Activation('relu')(b)
    d1 = dec(b, e7, 512)
    d2 = dec(d1, e6, 512)
    d3 = dec(d2, e5, 512)
    d4 = dec(d3, e4, 512, dropout=False)
    d5 = dec(d4, e3, 256, dropout=False)
    d6 = dec(d5, e2, 128, dropout=False)
    d7 = dec(d6, e1, 64, dropout=False)
    g = Conv2DTranspose(img_shape[2], (4,4), strides=(2,2), padding='same',
kernel_initializer=init)(d7)
    out_image = Activation('tanh')(g)
    model = Model(in_image, out_image)
    return model
```

1. The `enc` function defines an encoder block. It takes the input layer (`layer_in`), the number of filters (`n_filters`), and an optional flag for batch normalization (`batchnorm`). It performs a convolution operation, followed by batch normalization (if `batchnorm` is `True`), and a leaky ReLU activation function. The output of the encoder block is returned.

2. The `dec` function defines a decoder block. It takes the input layer (`layer_in`), a skip connection from the corresponding encoder block (`skip_in`), the number of filters (`n_filters`), and an optional flag for dropout (`dropout`). It performs a transposed convolution operation, followed by batch normalization, dropout (if `dropout` is `True`), concatenation with the skip connection, and a ReLU activation function. The output of the decoder block is returned.

3. The `gen` function defines the generator model. It takes an input shape (`img_shape`) and initializes the weights using the `RandomNormal` initializer.

4. An input layer (`in_image`) is created with the specified `img_shape`.

5. The input image is passed through a series of encoder blocks (`e1` to `e7`), with increasing numbers of filters.

6. A bottleneck layer (`b`) is created using a convolutional layer with a stride of 2 and a ReLU activation function.

7. The bottleneck layer is passed through a series of decoder blocks (`d1` to `d7`), with decreasing numbers of filters.

8. The output of the last decoder block is passed through a transposed convolutional layer to generate the output image (`out_image`).

9. The output image is passed through an activation function (`tanh`) to ensure pixel values are within the range of -1 to 1.

10. The model is defined using the `Model` class from Keras, with the input layer and the output layer.

11. The generator model is returned.

The GAN

```
def gan(g_model,d_model,img_shape):
  for layer in d_model.layers:
    if not isinstance(layer,BatchNormalization):
      layer.trainable=False
  in_src=Input(shape=img_shape)
  gen_out=g_model(in_src)
  dis_out=d_model([in_src,gen_out])
  model=Model(in_src,[dis_out,gen_out])
  opt=Adam(learning_rate=0.0002,beta_1=0.5,beta_2=0.999)
  model.compile(loss=['binary_crossentropy','mae'],optimizer=opt,loss_weights=[1,100])
  return model
```

This function takes three arguments: `g_model` (the generator model), `d_model` (the discriminator model), and `img_shape` (the shape of the input images). It creates a composite model that combines the generator and discriminator models.

Here's how the code works:

1. It freezes the weights of all layers in the discriminator model except for the BatchNormalization layers. This is done to prevent the discriminator from being trained during the generator training process.

2. It defines an input layer `in_src` with the specified `img_shape`.

3. It passes the input `in_src` through the generator model `g_model` to obtain the generated output `gen_out`.

4. It passes both the input `in_src` and the generated output `gen_out` through the discriminator model `d_model` to obtain the discriminator output `dis_out`.

5. It creates a composite model `model` that takes `in_src` as input and outputs both `dis_out` and `gen_out`.

6. It uses the Adam optimizer with a learning rate of 0.0002, beta_1 of 0.5, and beta_2 of 0.999.

7. It compiles the model with two loss functions: binary cross-entropy for the discriminator output and mean absolute error for the generated output. The loss weights are set to [1, 100] to give more importance to the generator loss.

8. Finally, it returns the composite model.

Additional Function:

Sample Generating Functions

```
def gen_real_samp(dataset,n_samples,patch_shape):
  trainA,trainB=dataset
  ix=randint(0,trainA.shape[0],n_samples)
  X1,X2=trainA[ix],trainB[ix]
  y=ones((n_samples,patch_shape,patch_shape,1))
  return [X1,X2],y


def gen_fake_samp(g_model,samples,patch_shape):
  X=g_model.predict(samples)
  y=zeros((len(X),patch_shape,patch_shape,1))
  return X,y
```

gen_real_samp(dataset, n_samples, patch_shape)

This function generates real samples from a given dataset.

1. The function takes three parameters: `dataset`, `n_samples`, and `patch_shape`.

2. The `dataset` parameter is expected to be a tuple containing two arrays, `trainA` and `trainB`.

3. The function generates random indices (`ix`) within the range of the number of samples in `trainA` using the `randint` function.

4. It then selects the corresponding samples from `trainA` and `trainB` using the generated indices.

5. The function creates a target array `y` filled with ones, with the shape `(n_samples, patch_shape, patch_shape, 1)`.

6. Finally, the function returns a list containing the selected samples `[X1, X2]` and the target array `y`.

gen_fake_samp(g_model, samples, patch_shape)

This function generates fake samples using a given generator model.

1. The function takes three parameters: `g_model`, `samples`, and `patch_shape`.

2. The `g_model` parameter represents a generator model.

3. The function predicts fake samples (`X`) using the generator model and the provided `samples`.

4. It creates a target array `y` filled with zeros, with the shape `(len(X), patch_shape, patch_shape, 1)`.

5. Finally, the function returns the predicted fake samples `X` and the target array `y`.

Performance Function

```
def perf(step,g_model,dataset,n_samples=3):
  [X_realA,X_realB],_=gen_real_samp(dataset,n_samples,1)
  X_fakeB,_=gen_fake_samp(g_model,X_realA,1)
  X_realA=(X_realA+1)/2
  X_realB=(X_realB+1)/2
  X_fakeB=(X_fakeB+1)/2
  for i in range(n_samples):
    plt.subplot(3, n_samples, 1 + i)
    plt.axis('off')
    plt.imshow(X_realA[i])
  for i in range(n_samples):
    plt.subplot(3, n_samples, 1 + n_samples + i)
    plt.axis('off')
    plt.imshow(X_fakeB[i])
  # plot real target image
  for i in range(n_samples):
    plt.subplot(3, n_samples, 1 + n_samples*2 + i)
    plt.axis('off')
    plt.imshow(X_realB[i])
```

```
  filename1 = 'plot_%06d.png' % (step+1)
  plt.savefig(filename1)
  plt.close()
  filename2 = 'model_%06d.h5' % (step+1)
  g_model.save(filename2)
  print('>Saved: %s and %s' % (filename1, filename2))
```

The function `perf`' performs several tasks related to saving and plotting images during the training process of a model.

1. The function takes four parameters: `step`, `g_model`, `dataset`, and `n_samples`.

2. Inside the function, it calls the `gen_real_samp` function to generate real samples from the given dataset, `dataset`, with the specified number of samples, `n_samples`, and patch shape of 1.

3. It assigns the generated real samples to the variables `X_realA` and `X_realB`, and ignores the target array.

4. Next, it calls the `gen_fake_samp` function to generate fake samples using the generator model, `g_model`, and the real samples `X_realA` as input. The generated fake samples are assigned to the variable `X_fakeB`, and the target array is ignored.

5. The function then performs some preprocessing on the real and fake samples by scaling them from the range [-1, 1] to the range [0, 1] using the `(X + 1) / 2` operation.

6. It uses a loop to plot the real samples, fake samples, and real target images using the `plt.imshow` function. The `n_samples` parameter is used to determine the number of subplots.

7. After plotting the images, the function saves the figure as a PNG file named `plot_%06d.png` using the `plt.savefig` function. The filename is constructed based on the `step` parameter.

8. The function then saves the generator model to an HDF5 file named `model_%06d.h5` using `g_model.save`. Similarly, the filename is constructed based on the `step` parameter.

9. Finally, it prints a message indicating the files that have been saved.

Train Function

```
def train(d_model,g_model,gan_model,dataset,n_epochs=100,n_batch=1):
  n_patch=d_model.output_shape[1]
  trainA,trainB=dataset
  bat_per_epo=int(len(trainA)/n_batch)
  n_steps=bat_per_epo*n_epochs
  for i in range(n_steps):
    [X_realA,X_realB],y_real=gen_real_samp(dataset,n_batch,n_patch)
    X_fakeB,y_fake=gen_fake_samp(g_model,X_realA,n_patch)
    d_loss1=d_model.train_on_batch([X_realA,X_realB],y_real)
    d_loss2=d_model.train_on_batch([X_realA,X_fakeB],y_fake)
    g_loss,_,_=gan_model.train_on_batch(X_realA,[y_real,X_realB])
    print('>%d,d1[%.3f] d2[%.3f] g[%.3f]'%(i+1,d_loss1,d_loss2,g_loss))
    if (i+1)%(500)==0:
      perf(i,g_model,dataset)
```

The function `train` trains the Conditional GAN model.

1. The function takes six parameters: `d_model`, `g_model`, `gan_model`, `dataset`, `n_epochs`, and `n_batch`.

2. It determines the shape of the output patches of the discriminator model, `d_model`, and assigns it to the variable `n_patch`.

3. It unpacks the dataset into `trainA` and `trainB`.

4. It calculates the number of batches per epoch by dividing the length of `trainA` by the batch size, `n_batch`, and assigns it to the variable `bat_per_epo`.

5. It calculates the total number of steps for training by multiplying `bat_per_epo` with the number of epochs, `n_epochs`, and assigns it to the variable `n_steps`.

6. It enters a loop that iterates `n_steps` times.

7. Inside the loop, it calls the `gen_real_samp` function to generate real samples from the dataset with the specified batch size, `n_batch`, and patch shape, `n_patch`. The generated real samples are assigned to the variables `X_realA`, `X_realB`, and `y_real`.

8. It calls the `gen_fake_samp` function to generate fake samples using the generator model, `g_model`, and the real samples `X_realA` as input. The generated fake samples are assigned to the variables `X_fakeB` and `y_fake`.

9. It trains the discriminator model, `d_model`, twice using the `train_on_batch` method. The first training step uses the real samples `X_realA` and `X_realB` as input and the corresponding real labels `y_real`. The second training step uses the real samples `X_realA` and the generated fake samples `X_fakeB` as input and the corresponding fake labels `y_fake`. The losses from both training steps are assigned to the variables `d_loss1` and `d_loss2`, respectively.

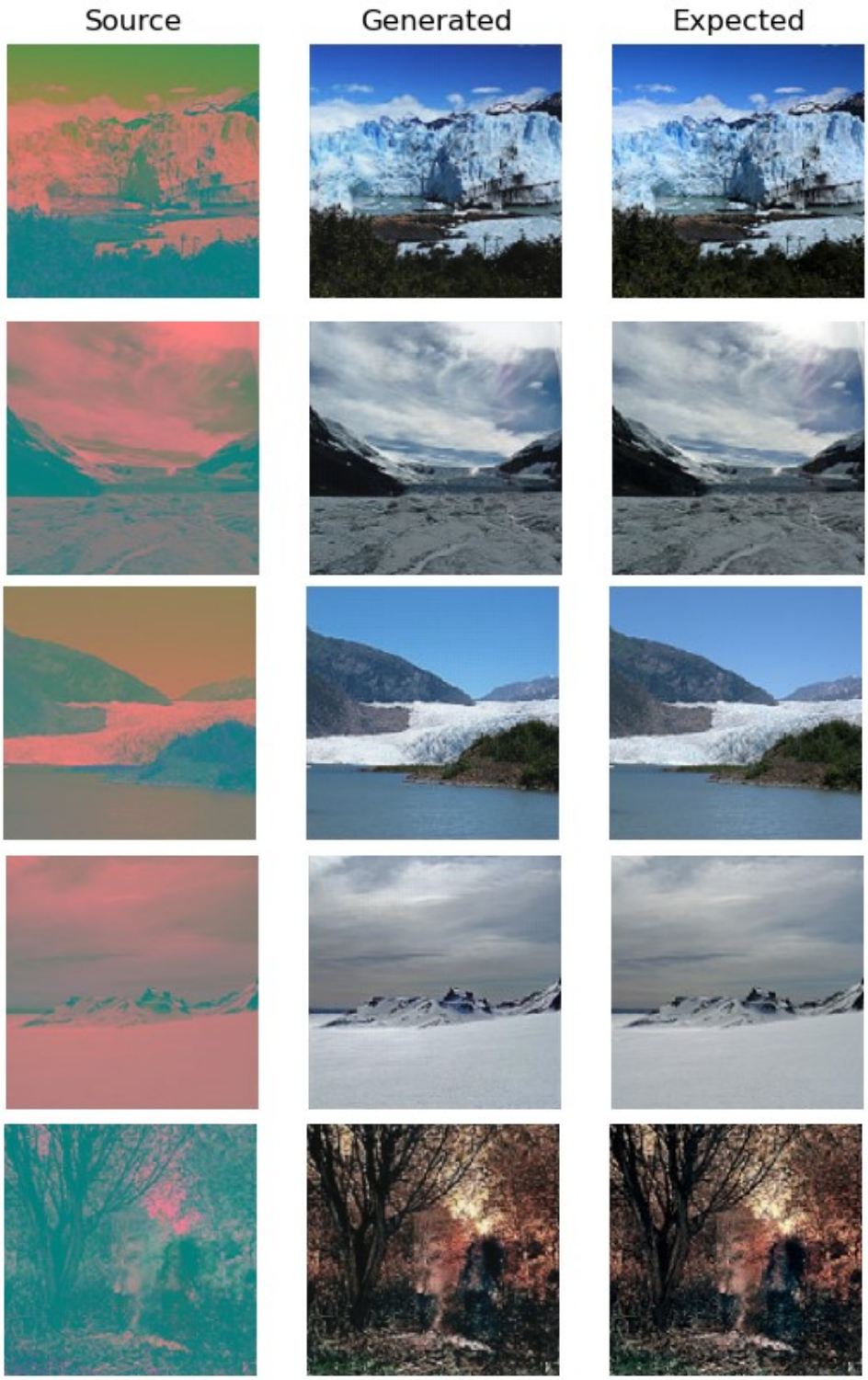
10. It trains the generator model, `g_model`, and the combined GAN model, `gan_model`, using the `train_on_batch` method. The generator model is trained using the real samples `X_realA` as input and the real labels `y_real` and the target images `X_realB`. The loss from the generator model training is assigned to the variable `g_loss`.

11. It prints the discriminator and generator losses for each training step.

12. If the current step is a multiple of 500, it calls the `perf` function to save and plot images using the generator model, `g_model`, and the dataset.

13. After the loop ends, the training is complete, and the function returns.

Final Outcome



GitHub Link: https://github.com/Barbaaryan/MLProject