CycleGAN Image Conversion

Yushuo (Shawn) Han

July 2020

1 Introduction

Visual arts is an effective mean of story-telling. It is expressive in nature and easy to forge a connection to the audience, and thus is commonly used as a mean of persuasion by injecting pathos in its role. In the field of conservation, visual arts are crucial in raising public awareness in endangered species and habitat protection. The modern approach of "Artivism", is using visual arts to create change in culture and shift in public opinion. One of the successful cases of "Artivism" is the documentary film by Shawn Heinrichs, a cinematographer focusing on filming sharks and rays slaughtering. His documentaries, played to the locals in West Papua, Indonesia, successfully transitioned a hunting culture into eco-tourism, and eventually managed to create the first marine "*Provinsi Konservasi*", or conservation province, in the world. A wide variety of visual arts can fulfill the purpose of artivism. The idea of this project is to provide style-mapping to images using deep learning to create novelty, bestow new layers of meanings, and elicit empathy.

A Generative Adversarial Network (GAN) is a generative network that is first introduced in 2014 by Ian Goodfellow, et al (DOI: arXiv:1406.2661). The generator generates simple random variables, and transforms them into complex ones as output. The generator is then paired with a discrimator which tries to distinguish the generator's output from the samples in the target domain. The generator and discriminator are then trained adversarially to enhance their performance until the equilibrium where the generator can completely deceive the discriminator.

The idea of CycleGAN is developed in 2017 by Jun-Yan Zhu, et al. in their research paper *Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks* (DOI: arXiv:1703.10593). In the cycle-consistent adversarial networks, two generators, *AtoB* and *BtoA*, map domain A to domain B, and domain B to domain A, respectively. Both generators are paired with a discrimator, and are trained adversarially to generate realistic outputs. Moreover, outputs from generator *AtoB* are fed into *BtoA* to map the output back to domain A. This "cycled" output is then compared with domain A by the discriminator, and the difference ("cycle loss") is used to punish the networks. This design enables the generative neural nets to only apply styles/target features to images without changing the main features of the input.

In this project, a CycleGAN is built in Keras to take winter landscape photos as input, and output a summer-styled version of them, and vice versa. The networks are trained on approximately 1000 summer photos and another 1000 winter photos scraped from the internet, where all photo samples are resized to 256 x 256. The network is trained using a Tesla V100 GPU on Vast.ai.

For future, the two desired domains, summer and winter, are abbreviated as A and B, respectively.

2 Data Preprocessing

First, import and load datasets.

[1]:
```
import os

import numpy as np
from matplotlib import pyplot as plt

from keras.preprocessing.image import img_to_array, load_img
from keras_contrib.layers.normalization.instancenormalization import␣
 ↪InstanceNormalization
from keras.initializers import RandomNormal
from keras.layers import Input, Conv2D, Conv2DTranspose, LeakyReLU, Activation,␣
 ↪Concatenate
from keras import Model
from keras.optimizers import Adam

import random

def load_dir(src):
    photos = []

    for pic in os.listdir(src):
        pic = load_img(src + '/' + pic, target_size=(256, 256))
        pic = img_to_array(pic)
        photos.append(pic)

    return np.asarray(photos)

trainA = load_dir('resizedSeasonDataset/trainA')
testA = load_dir('resizedSeasonDataset/testA')
trainB = load_dir('resizedSeasonDataset/trainB')
testB = load_dir('resizedSeasonDataset/testB')

A = np.vstack((trainA, testA))
B = np.vstack((trainB, testB))
print('Shape of A: ', A.shape)
print('Shape of B: ', B.shape)

np.savez_compressed('resizedSeasonDataset/dataAB.npz', A, B)
print('Data saved to resizedSeasonDataset/dataAB.npz')
```

```
Shape of A:  (1096, 256, 256, 3)
Shape of B:  (1090, 256, 256, 3)
Data saved to resizedSeasonDataset/dataAB.npz
```

In the following, the samples from two domains are briefly examined. It can be observed that the summer samples are more colourful in style, comparing to the majority of whiteness in winter samples. The variety in the colour feature, as a hypothetic note, would create inconsistent style mapping from winter domain to summer domain, and if time permits, the summer photos could have been carefully prepared with a consistent style (e.g. green).

[2]:
```
dataset = np.load('resizedSeasonDataset/dataAB.npz')
A, B = dataset['arr_0'], dataset['arr_1']

plt.figure(figsize=(12, 5))
for i in range (5):
    plt.subplot(2, 5, 1 + i)
    plt.axis('off')
    plt.imshow(A[i].astype('uint8'))
plt.show()

plt.figure(figsize=(12, 5))
for i in range (5):
    plt.subplot(2, 5, 1 + i)
    plt.axis('off')
    plt.imshow(B[i].astype('uint8'))
plt.show()
```

3 Models: Generators and Discriminators

The discriminator is a PatchGAN 70x70 discriminator built using convolutional and LeakyReLU layers, based on the paper. The optimization uses Mean Squared Error (MSE). Moreover, the author recommands to use loss_weight = 0.5 because it is desirable to make the updates of the discriminator slower than the generator during training for more robust generators.

The generator is an encoder-decoder model. Besides the downsampling and upsampling, the model uses ResNet blocks to interpret the downsampled/encoded data. Matching the research paper, 9 such ResNet layers are applied, and the filter size of the convolutional layers in ResNet blocks are all set to be 3x3.

[3]:
```
def discriminator(shape):

    init = RandomNormal(stddev=0.02)

    # Input
    inputLayer = Input(shape=shape)

    # Convolution layers: 64, 128, 256, 512, 512
    model = Conv2D(filters=64, kernel_size=(4, 4), strides=(2, 2),␣
↪padding='same',
                   kernel_initializer=init)(inputLayer)
    model = LeakyReLU(alpha=0.2)(model)

    model = Conv2D(filters=128, kernel_size=(4, 4), strides=(2, 2),␣
↪padding='same',
                   kernel_initializer=init)(model)
    model = InstanceNormalization(axis=-1)(model)
    model = LeakyReLU(alpha=0.2)(model)

    model = Conv2D(filters=256, kernel_size=(4, 4), strides=(2, 2),␣
↪padding='same',
                   kernel_initializer=init)(model)
    model = InstanceNormalization(axis=-1)(model)
    model = LeakyReLU(alpha=0.2)(model)

    model = Conv2D(filters=512, kernel_size=(4, 4), strides=(2, 2),␣
↪padding='same',
                   kernel_initializer=init)(model)
    model = InstanceNormalization(axis=-1)(model)
    model = LeakyReLU(alpha=0.2)(model)

    model = Conv2D(filters=512, kernel_size=(4, 4), strides=(1, 1),␣
↪padding='same',
                   kernel_initializer=init)(model)
    model = InstanceNormalization(axis=-1)(model)
    model = LeakyReLU(alpha=0.2)(model)
```

```
    # output
    output = Conv2D(filters=1, kernel_size=(4, 4), strides=(1, 1),␣
↪padding='same',
                    kernel_initializer=init)(model)
    model = Model(inputLayer, output)

    # Compile
    model.compile(loss='mse', optimizer=Adam(lr=0.0002, beta_1=0.5),␣
↪loss_weights=[0.5])

    return model
```

[4]:
```
def resnetGen(n_filters, inputLayer):
    init = RandomNormal(stddev=0.02)

    # convolutional layer 1
    block = Conv2D(filters=n_filters, kernel_size=(3, 3), padding='same',
                   kernel_initializer=init)(inputLayer)
    block = InstanceNormalization(axis=-1)(block)
    block = Activation('relu')(block)

    # convolutional layer 2
    block = Conv2D(filters=n_filters, kernel_size=(3, 3), padding='same',
                   kernel_initializer=init)(block)
    block = InstanceNormalization(axis=-1)(block)

    # output
    block = Concatenate()([block, inputLayer])
    return block

def generator(shape):
    """
    According to the article, choose the network with 9 residual blocks
    c7s1-64, d128, d256, r256x9, u128, u64, c7s1-3
    """
    init = RandomNormal(stddev=0.02)

    # Input
    inputLayer = Input(shape=shape)

    # c7s1-64
    model = Conv2D(filters=64, kernel_size=(7, 7), padding='same',
                   kernel_initializer=init)(inputLayer)
    model = InstanceNormalization(axis=-1)(model)
    model = Activation('relu')(model)
```

```
    # d128
    model = Conv2D(filters=128, kernel_size=(3, 3), strides=(2, 2),␣
↪padding='same',
                   kernel_initializer=init)(model)
    model = InstanceNormalization(axis=-1)(model)
    model = Activation('relu')(model)

    # d256
    model = Conv2D(filters=256, kernel_size=(3, 3), strides=(2, 2),␣
↪padding='same',
                   kernel_initializer=init)(model)
    model = InstanceNormalization(axis=-1)(model)
    model = Activation('relu')(model)

    # R256x9
    for i in range(9):
        model = resnetGen(256, model)

    # u128
    model = Conv2DTranspose(filters=128, kernel_size=(3, 3), strides=(2, 2),␣
↪padding='same',
                            kernel_initializer=init)(model)
    model = InstanceNormalization(axis=-1)(model)
    model = Activation('relu')(model)

    # u64
    model = Conv2DTranspose(filters=64, kernel_size=(3, 3), strides=(2, 2),␣
↪padding='same',
                            kernel_initializer=init)(model)
    model = InstanceNormalization(axis=-1)(model)
    model = Activation('relu')(model)

    # c7s1-3
    model = Conv2D(filters=3, kernel_size=(7, 7), strides=(1, 1), padding='same',
                            kernel_initializer=init)(model)
    model = InstanceNormalization(axis=-1)(model)

    # output
    output = Activation('tanh')(model)
    model = Model(inputLayer, output)
    return model
```

4 Loss

Define four losses: * **Adversial Loss**: Feed an image from A to G1, and compare the loss of the output against target domain B. * **Identity Loss**: Feed an image from B to G1, and compare the loss of the output against ifself (target domain B). * **Forward Cycle Loss**: Feed an image from A to G1, then feed the output to G2, and compare the loss of the final output against source domain A. * **Backward Cycle Loss**: Feed an image from B to G2, then feed the output to G1, and compare the loss of the final output against target domain B.

Matching the research paper, the total loss is a weigted average of the four losses above, with weight distributed 1, 5, 10, 10, respectively, and that they are defined as mean squared error, mean absolute error, mean absolute error, and mean absolute error, respectively.

[5]:
```
def compositeModel(gen1, dis1, gen2, shape):

    gen1.trainable = True
    dis1.trainable = False
    gen2.trainable = False

    # Adversial
    inputLayer = Input(shape=shape)
    gen1Out = gen1(inputLayer)
    dis1Out = dis1(gen1Out)

    # Identity
    identityInputLayer = Input(shape=shape)
    identityGen1Out = gen1(identityInputLayer)

    # Forward Cycle
    forwardOut = gen2(gen1Out)

    # Backward Cycle
    gen2Out = gen2(identityInputLayer)
    backwardOut = gen1(gen2Out)

    # Define model
    model = Model([inputLayer, identityInputLayer],
                  [dis1Out, identityGen1Out, forwardOut, backwardOut])
    model.compile(loss=['mse', 'mae', 'mae', 'mae'],
                  loss_weights=[1, 5, 10, 10],
                  optimizer=Adam(lr=0.0002, beta_1=0.5))

    return model
```

Each composite model is used to train only one generator (as above, only for the input “gen1” generator), and thus in the end two composite models are needed, one for each generator.

5 Load Data and Batch

The getRealSample function randomly selects images from one domain, and generates the target labels for the patchGAN discriminator model (ones). The output of the patchGAN will be a 256 = 16x16x1 activation map. The getFakeSample function uses the generator to generate a fake sample, but the target labels are set to fake (zeros).

[6]:
```
def loadSamples(src):
    #load
    dataset = np.load(src)
    A, B = dataset['arr_0'], dataset['arr_1']

    # Normalize and return
    A = (A - 127.5) / 127.5
    B = (B - 127.5) / 127.5
    return [A, B]

def getRealSample(dataset, sampleSize, shape):

    indices = np.random.randint(0, dataset.shape[0], sampleSize)
    X = dataset[indices]
    y = np.ones((sampleSize, shape, shape, 1))

    return X, y

def getFakeSample(gen, dataset, shape):

    X = gen.predict(dataset)
    y = np.zeros((len(X), shape, shape, 1))

    return X, y
```

Typically, the loss of the GAN models does not converge. The only indication is that an equilibrium is reached between the generator and the discriminator. And thus, in every certain amount of epochs, the weights would have to be saved and used to train new images.

[30]:
```
def saveModel(step, gen1, gen2):
    gen1.save('gen1_{}saved.h5'.format(step+1))
    gen2.save('gen2_{}saved.h5'.format(step+1))
```

The viewGenerated function defined below demonstrates some generated translated images, and saves the plot.

[31]:
```
def viewGenerated(step, gen, X_train, name):

    # Get a sample from a domain
    X_in, _ = getRealSample(X_train, 5, 0)
```

```
    # Translate using generator and normalize.
    X_out, _ = getFakeSample(gen, X_in, 0)
    X_in = (X_in + 1) / 2
    X_out = (X_out + 1) / 2

    # plot real
    plt.figure(figsize=(12, 5))
    for i in range(5):
        plt.subplot(2, 5, 1 + i)
        plt.axis('off')
        plt.imshow(X_in[i])

    # plot fake
    plt.figure(figsize=(12, 5))
    for i in range(5):
        plt.subplot(2, 5, 6 + i)
        plt.axis('off')
        plt.imshow(X_out[i])

    # Save plot
    plt.savefig('{0}StepGeneratedPlot({1}).png'.format(step + 1, name))
    plt.close()
```

In order to manage and monitor the learning/updating speed of the discriminator model, the paper's original author proposed the method of maintaining a pool of generated fake images for each discriminator. This is implemented as a pool of 50 fake images stored in a python list, and probabilistically either add a new fake image or replace one of the existing fake image in the list using the updatePool function below.

[32]:
```
def updatePool(pool, imgs):

    selectedImgs = list()
    for img in imgs:

        # Adds to the pool if the pool is less than 50 imgs.
        if len(pool) < 50:
            pool.append(img)
            selectedImgs.append(img)

        # 50% of the remaining case, use the image but do not add to pool
        elif random.random() < 0.5:
            selectedImgs.append(img)

        # 50% of the remaining case, use the image and add to pool
        else:
            idx = np.random.randint(0, len(pool))
            selectedImgs.append(pool[idx])
```

```
            pool[idx] = img

    return np.asarray(selectedImgs)
```

6 Training

In the following, the training hyperparameters are set according to the paper. Specifically, the batchsize is set to 1, and the number of epochs is set to be 200. However, due to various constraints, the model is trained on only 30 epochs.

First, a batch of real images are randomly selected from each domain, then they are used by the generators to generate the fake images. The fake images are used to update each discriminator's fake image pool. Next, both generators are updated via their respective composite model. Finally, each weighted average losses are used to update each generator and are reported.

[33]:
```
def train(disA, disB, genA, genB, compA, compB, dataset):

    # Set up hyperparams and data structures
    BATCHSIZE = 1
    EPOCH = 30 # should be 200
    PATCH = disA.output_shape[1]
    trainA, trainB = dataset
    poolA, poolB = [], []
    BATCH_PER_EPOCH = int(len(trainA) / BATCHSIZE)
    STEPS = BATCH_PER_EPOCH * EPOCH

    print("Total steps: {}".format(STEPS))
    # train over epochs
    for i in range (STEPS):

        # Randomly select real samples
        X_realA, y_realA = getRealSample(trainA, BATCHSIZE, PATCH)
        X_realB, y_realB = getRealSample(trainB, BATCHSIZE, PATCH)

        # Use real samples to generate fake samples
        X_fakeA, y_fakeA = getFakeSample(genB, X_realB, PATCH)
        X_fakeB, y_fakeB = getFakeSample(genA, X_realA, PATCH)

        # update pool for each discriminator
        X_fakeA = updatePool(poolA, X_fakeA)
        X_fakeB = updatePool(poolB, X_fakeB)

        # update genB via composite model
        genBLoss, _,_,_,_ = compB.train_on_batch([X_realB, X_realA],
                                                 [y_realA, X_realA, X_realB,␣
↪X_realA])
```

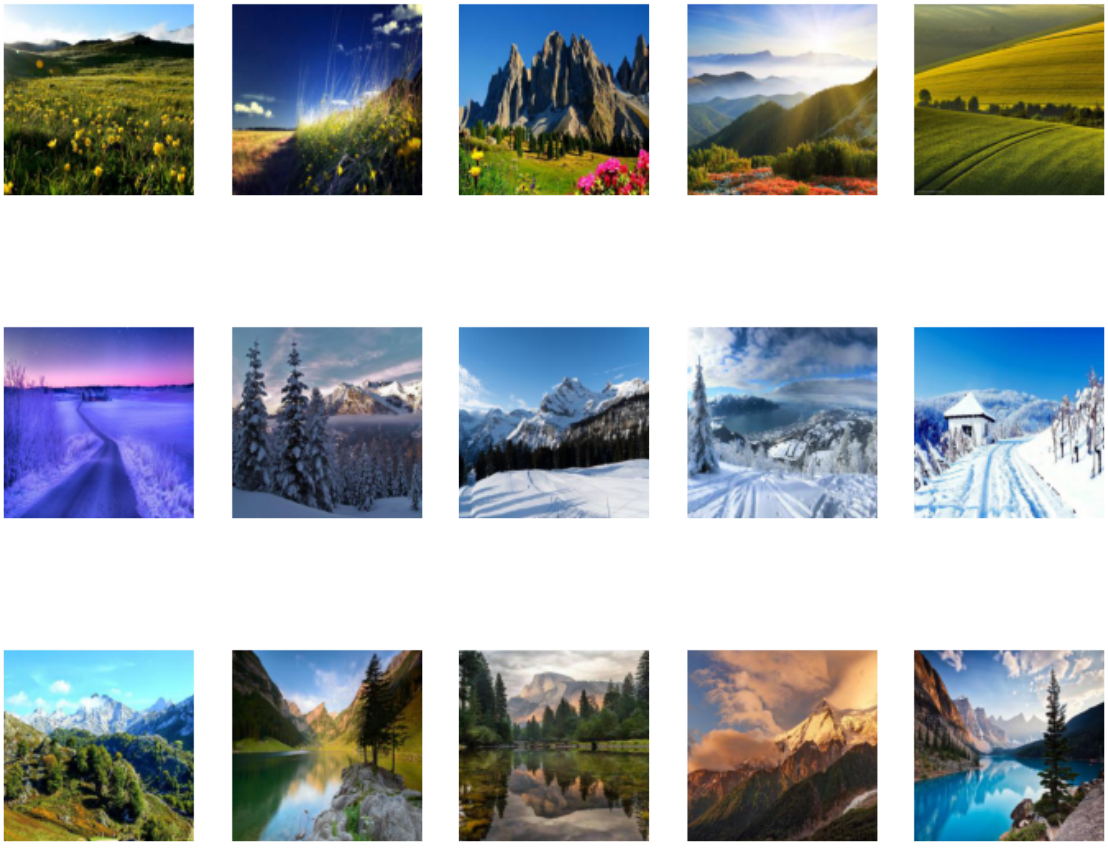
```
        # update disA via the real/fake classification result
        disALoss1 = disA.train_on_batch(X_realA, y_realA)
        disALoss2 = disA.train_on_batch(X_fakeA, y_fakeA)

        # update genA via composite model
        genALoss, _,_,_,_ = compA.train_on_batch([X_realA, X_realB],
                                                 [y_realB, X_realB, X_realA,␣
↪X_realB])

        # update disB via the real/fake classification result
        disBLoss1 = disB.train_on_batch(X_realB, y_realB)
        disBLoss2 = disB.train_on_batch(X_fakeB, y_fakeB)

        # Summarize
        print('>%d, dA[%.3f,%.3f] dB[%.3f,%.3f] g[%.3f,%.3f]' % (
            i+1, disALoss1, disALoss2, disBLoss1, disBLoss2, genALoss, genBLoss))

        # Evaluate
        if (i + 1) % (BATCH_PER_EPOCH * 1) == 0:
            viewGenerated(i, genA, trainA, 'AtoB')
            viewGenerated(i, genB, trainB, 'BtoA')

        if (i + 1) % (BATCH_PER_EPOCH * 5) == 0:
            saveModel(i, genA, genB)
```

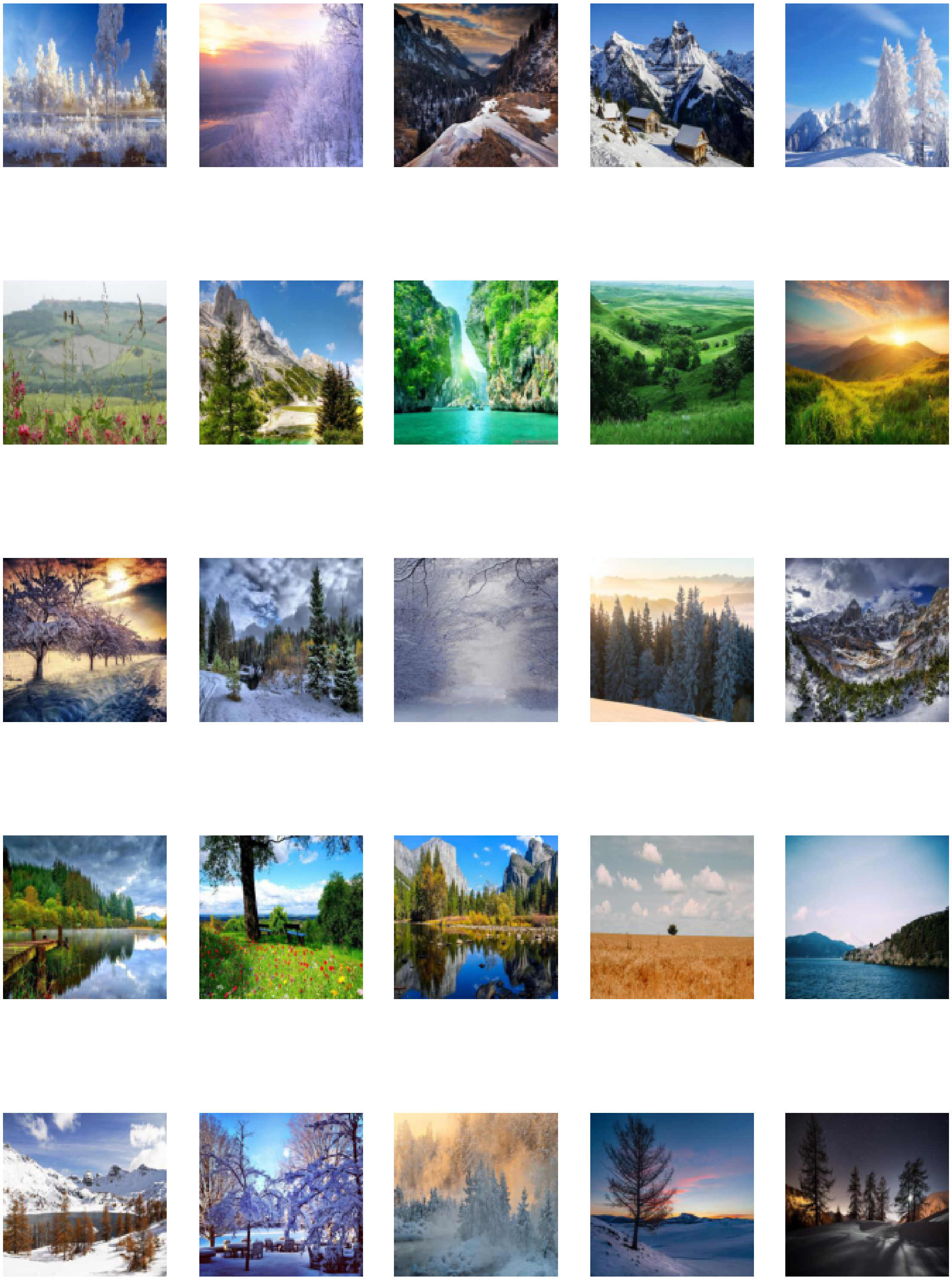
Now, load data and start training. The data is trained on a Tesla V100 GPU on Vast.ai. On the machine, the training speed is around 80 seconds per 100 steps, or 0.8s/step.

[34]:
```
# Load data
dataset = loadSamples('./resizedSeasonDataset/dataAB.npz')
shape = dataset[0].shape[1:]

# models
genA = generator(shape)
genB = generator(shape)
disA = discriminator(shape)
disB = discriminator(shape)
compA = compositeModel(genA, disB, genB, shape)
compB = compositeModel(genB, disA, genA, shape)

# start training
train(disA, disB, genA, genB, compA, compB, dataset)
```

```
Total steps: 32880
>1, dA[0.554,1.300] dB[0.594,1.809] g[17.106,18.077]
>2, dA[3.824,1.390] dB[5.443,1.301] g[17.912,18.418]
>3, dA[11.737,0.956] dB[8.010,1.535] g[23.230,19.280]
>4, dA[4.996,1.329] dB[2.669,4.955] g[18.507,18.362]
```

```
>5, dA[3.209,1.039] dB[1.781,2.037] g[18.304,17.500]
>6, dA[4.750,0.926] dB[1.757,4.372] g[21.767,17.445]
>7, dA[8.536,0.806] dB[0.973,3.814] g[20.899,15.737]
>8, dA[2.345,0.681] dB[3.521,2.237] g[21.668,15.176]
>9, dA[1.174,0.624] dB[2.910,1.385] g[15.637,14.221]
>10, dA[1.111,0.684] dB[1.024,1.180] g[14.357,12.179]
>5451, dA[0.042,0.181] dB[0.005,0.027] g[7.696,6.903]
>5452, dA[0.025,0.009] dB[0.012,0.007] g[7.758,7.672]
>5453, dA[0.288,0.009] dB[0.140,0.026] g[3.086,4.628]
>5454, dA[0.140,0.190] dB[0.064,0.170] g[3.671,3.778]
>5455, dA[0.178,0.530] dB[0.008,0.063] g[4.590,4.307]
>5456, dA[0.161,0.032] dB[0.185,0.058] g[4.124,4.036]
>5457, dA[0.029,0.034] dB[0.007,0.019] g[6.431,6.269]
>5458, dA[0.070,0.012] dB[0.005,0.158] g[6.561,7.763]
>5459, dA[0.046,0.047] dB[0.108,0.015] g[6.614,6.178]
>5460, dA[0.128,0.152] dB[0.384,0.072] g[4.845,4.262]
```





Save the ultimate model.

[35]:
```
def saveModel(gen1, gen2):
    gen1.save('./gen1saved.h5')
    gen2.save('./gen2saved.h5')

saveModel(genA, genB)
```

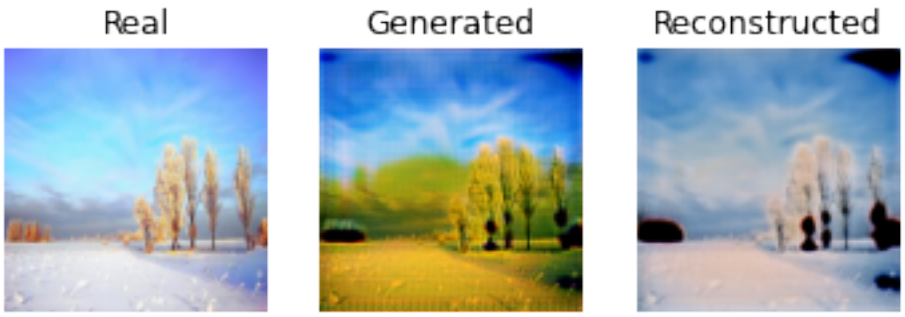
7 Predicting a new datapoint

In the following, a datapoint is randomly selected from each domain. Then, the two generators are loaded and are used to transform the images respectively.

[36]:
```
from keras.models import load_model

# Load data
def getRealTestSample(filename):
    # load data
    data = np.load(filename)
    X1, X2 = data['arr_0'], data['arr_1']

    # scaling
    X1 = (X1 - 127.5) / 127.5
    X2 = (X2 - 127.5) / 127.5
    return [X1, X2]
A, B = getRealTestSample('./resizedSeasonDataset/dataAB.npz')

# Load model
theLayer = {'InstanceNormalization': InstanceNormalization}
model_AtoB = load_model('gen1saved.h5', theLayer)
model_BtoA = load_model('gen2saved.h5', theLayer)

# select a sample
def selectData(dataset, num):
    idx = np.random.randint(0, dataset.shape[0], num)
    return dataset[idx]

# Show result
def showResult(imagesX, imagesY1, imagesY2):
    images = np.vstack((imagesX, imagesY1, imagesY2))
    titles = ['Real', 'Generated', 'Reconstructed']

    # scale from [-1,1] to [0,1]
    images = (images + 1) / 2.0

    # plot
```

```
    for i in range(len(images)):
        plt.subplot(1, len(images), 1 + i)
        plt.axis('off')
        plt.imshow(images[i])
        plt.title(titles[i])
    plt.show()

# A to B to A
AReal = selectData(A, 1)
BGenerated = model_AtoB.predict(AReal)
ARegenerated = model_BtoA.predict(BGenerated)
showResult(AReal, BGenerated, ARegenerated)

# B to A to B
BReal = selectData(B, 1)
AGenerated = model_BtoA.predict(BReal)
BRegenerated = model_AtoB.predict(AGenerated)
showResult(BReal, AGenerated, BRegenerated)
```

```
WARNING:tensorflow:No training configuration found in the save file, so the
model was *not* compiled. Compile it manually.
WARNING:tensorflow:No training configuration found in the save file, so the
model was *not* compiled. Compile it manually.
```



The first summer sample is taken in South Island, NZ. It can be observed that the summer-to-winter generator transforms the photo by adding white/grey colours to the foreground and the sky. The bright blue colour of the lake is also reduced to a more pale colour. This output is then transformed by the winter-to-summer generator, which applies green colour to the flowers and mountains.

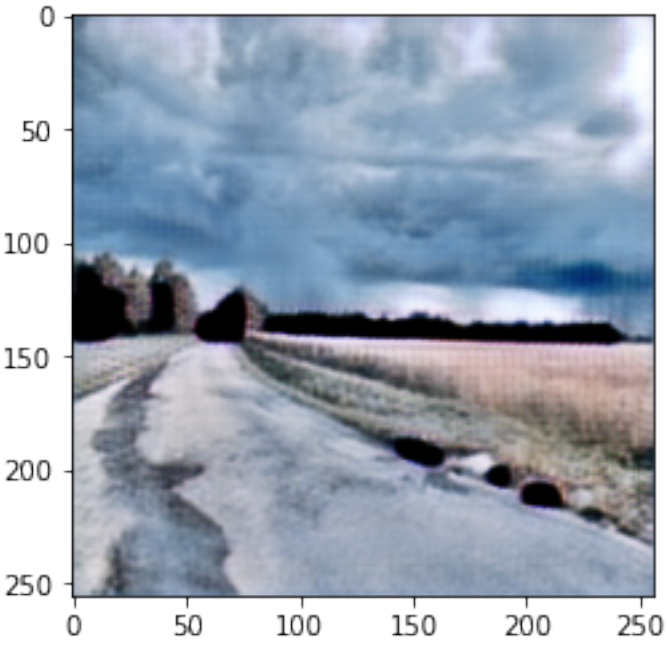
The second winter sample is more successful in style mapping. The generator, however, mistakens the cloudy sky as mountains and also applies green colour to it. However, in general, the generated and reconstructed photos are quite realistic in appearance.

It is worth mentioning that the generator identifies certain features to apply style (grassland, trees, etc) and others to not apply style/apply other style (lake, sky, etc). This is an exceptional ability of cycleGAN, making this a potentially high-prospect approach for artistic image processing.

In the following, a single generator is used to transform a single datapoint from one domain (summer) to another (winter). It can be observed that, although not perfect, the generator successfully applies winter elements to the photo. It can be postulated that with more epochs of training, the style mapping would be more successful and realistic.

[44]:
```
from PIL import Image
from IPython.display import display

def seeImage(src):
    pixels = load_img(src, target_size=(256, 256))
    pixels = img_to_array(pixels)
    pixels = np.expand_dims(pixels, 0)
    pixels = (pixels - 127.5) / 127.5
    return pixels

# load
theImg = seeImage('./resizedSeasonDataset/trainA/24.jpg')
theLayer = {'InstanceNormalization': InstanceNormalization}
model_AtoB = load_model('gen1saved.h5', theLayer)

# See original
orig = Image.open('./resizedSeasonDataset/trainA/24.jpg')
display(orig)

# Transform using generator and see result.
result = model_AtoB.predict(theImg)
result = (result + 1) / 2.0
plt.imshow(result[0])
plt.show()
```

```
WARNING:tensorflow:No training configuration found in the save file, so the
model was *not* compiled. Compile it manually.
```


0
50
100
150
200
250
0
50
100
150
200
250

8 Conclusion

Visual arts is of high value to the field of conservation. In this project, a deep learning approach of artivism is explored using cycleGAN, the cycle-consistent generative adversarial networks. With the design of cycle loss, the CycleGAN is designed to apply winter style to summer landscape photos and vice versa. The CycleGAN is trained on web-scraped winter and summer photos, approximately 1000 each, and a Tesla V100 GPU. The networks are implemented in Keras. Although only trained for a limited amount of 30 epochs, the trained networks showed high ability to apply seasonal styles from one domain to another. It is also observed that the trained generators are able to distinguish different features in the photo and apply corresponding colours/styles accordingly. The result thus shows high potential of well-trained cycleGAN in image style mapping, giving it great artistic value in image processing and thus making it a valuable potential approach for environemental artivism.

9 Acknowledgement

This notebook is based on the research paper *Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks* by Jun-Yan Zhu, et al. and an article by Jason Brownlee. Their work is greatly appreciated.