University of Tehran
Department of ECE
Neural Networks & Deep Learning
MP3

Members: Milad Mohammadi & Tohid Abdi
IDs: 810100462 & 810100410

Date: June 18, 2022

Questions List Table

1 SGAN **3**

1.1 3

1.2 3

1.3 10

1.4 11

2 DCGAN **19**

2.1 20

2.2 20

2.3 32

3 Cycle GAN **33**

3.1 33

3.2 33

3.3 33

3.4 50

1 SGAN

1.1

The Semi-Supervised GAN, or sometimes SGAN for short, is an extension of the Generative Adversarial Network architecture for addressing semi-supervised learning problems.

The discriminator in a traditional GAN is trained to predict whether a given image is real (from the dataset) or fake (generated), allowing it to learn features from unlabeled images. The discriminator can then be used via transfer learning as a starting point when developing a classifier for the same dataset, allowing the supervised prediction task to benefit from the unsupervised training of the GAN.

In the Semi-Supervised GAN, the discriminator model is updated to predict K+1 classes, where K is the number of classes in the prediction problem and the additional class label is added for a new "fake" class. It involves directly training the discriminator model for both the unsupervised GAN task and the supervised classification task simultaneously.

Training in unsupervised mode allows the model to learn useful feature extraction capabilities from a large unlabeled dataset, whereas training in supervised mode allows the model to use the extracted features and apply class labels.

The result is a classifier model that can achieve state-of-the-art results on standard problems such as MNIST when trained on very few labeled examples, such as tens, hundreds, or one thousand. Additionally, the training process can also result in better quality images output by the generator model.

1.2

Import essential libraries:

[1]:
```
from numpy import expand_dims
from numpy import zeros
from numpy import ones
from numpy import asarray
from numpy import prod
from numpy import reshape
from numpy.random import randn
from numpy.random import randint
from keras.datasets.mnist import load_data
import tensorflow as tf
import keras.optimizers
from keras.models import Model
from keras.layers import Input
from keras.layers import Dense
from keras.layers import Reshape
from keras.layers import Flatten
from keras.layers import Conv2D
from keras.layers import Conv2DTranspose
from keras.layers import LeakyReLU
```

```
from keras.layers import Dropout
from keras.layers import Lambda
from keras.layers import Activation
from keras.layers import BatchNormalization
from keras.layers import MaxPooling2D
from matplotlib import pyplot
from keras import backend
```

Custom activation Function:

[2]:
```
def custom_activation(output):
        logexpsum = backend.sum(backend.exp(output), axis=-1, keepdims=True)
        result = logexpsum / (logexpsum + 1.0)
        return result
```

Standalone supervised and unsupervised discriminator models:

Starting with the standard GAN discriminator model, we can update it to create two models that share feature extraction weights.

Specifically, we can define one classifier model that predicts whether an input image is real or fake, and a second classifier model that predicts the class of a given model.

- Binary Classifier Model. Predicts whether the image is real or fake, sigmoid activation function in the output layer, and optimized using the binary cross entropy loss function.
- Multi-Class Classifier Model. Predicts the class of the image, softmax activation function in the output layer, and optimized using the sparse categorical cross entropy loss function.

Both models have different output layers but share all feature extraction layers. This means that updates to one of the classifier models will impact both models.

[3]:
```
def define_discriminator(in_shape=(28,28,1), n_classes=10):
  # image input
        in_image = Input(shape=in_shape)
        # downsample
        fe = Conv2D(128, (3,3), strides=(2,2), padding='same')(in_image)
        fe = LeakyReLU(alpha=0.2)(fe)
        # downsample
        fe = Conv2D(128, (3,3), strides=(2,2), padding='same')(fe)
        fe = LeakyReLU(alpha=0.2)(fe)
        # downsample
        fe = Conv2D(128, (3,3), strides=(2,2), padding='same')(fe)
        fe = LeakyReLU(alpha=0.2)(fe)
        # flatten feature maps
        fe = Flatten()(fe)
        # dropout
        fe = Dropout(0.4)(fe)
        # output layer nodes
        fe = Dense(n_classes)(fe)
        # supervised output
```

```
        c_out_layer = Activation('softmax')(fe)
        # define and compile supervised discriminator model
        c_model = Model(in_image, c_out_layer)
        c_model.compile(loss='sparse_categorical_crossentropy', optimizer=tf.
↪keras.optimizers.Adam(lr=0.0002, beta_1=0.5), metrics=['accuracy'])
        # unsupervised output
        d_out_layer = Lambda(custom_activation)(fe)
        # define and compile unsupervised discriminator model
        d_model = Model(in_image, d_out_layer)
        d_model.compile(loss='binary_crossentropy', optimizer=tf.keras.
↪optimizers.Adam(lr=0.0002, beta_1=0.5))
        return d_model, c_model
```

Define the standalone generator models:

The generator model will take as input a point in the latent space and will use transpose convolutional layers to output a 28×28 grayscale image. The generator model will be fit via the unsupervised discriminator model.

[4]:
```
def define_generator(latent_dim):
        # image generator input
        in_lat = Input(shape=(latent_dim,))
        # foundation for 7x7 image
        n_nodes = 128 * 7 * 7
        gen = Dense(n_nodes)(in_lat)
        gen = LeakyReLU(alpha=0.2)(gen)
        gen = Reshape((7, 7, 128))(gen)
        # upsample to 14x14
        gen = Conv2DTranspose(128, (4,4), strides=(2,2), padding='same')(gen)
        gen = LeakyReLU(alpha=0.2)(gen)
        # upsample to 28x28
        gen = Conv2DTranspose(128, (4,4), strides=(2,2), padding='same')(gen)
        gen = LeakyReLU(alpha=0.2)(gen)
        # output
        out_layer = Conv2D(1, (7,7), activation='tanh', padding='same')(gen)
        # define model
        model = Model(in_lat, out_layer)
        return model
```

Define the combined generator and discriminate model, for updating the generator:

We will use the composite model architecture, common to training the generator model when implemented in Keras. Specifically, weight sharing is used where the output of the generator model is passed directly to the unsupervised discriminator model, and the weights of the discriminator are marked as not trainable.

The define_gan() function below implements this, taking the already-defined generator and discriminator models as input and returning the composite model used to train the weights of the generator model.

[5]:
```
def define_gan(g_model, d_model):
        # make weights in the discriminator not trainable
        d_model.trainable = False
        # connect image output from generator as input to discriminator
        gan_output = d_model(g_model.output)
        # define gan model as taking noise and outputting a classification
        model = Model(g_model.input, gan_output)
        # compile model
        opt = tf.keras.optimizers.Adam(lr=0.0002, beta_1=0.5)
        model.compile(loss='binary_crossentropy', optimizer=opt)
        return model
```

Load the images:

Load the training dataset and scale the pixels to the range [-1, 1] to match the output values of the generator model.

[6]:
```
def load_real_samples():
        # load dataset
  (trainX, trainy), (testX, testy) = load_data()
  # expand to 3d, e.g. add channels
  trainX = expand_dims(trainX, axis=-1)
  testX = expand_dims(testX, axis=-1)
  # convert from ints to floats
  trainX = trainX.astype('float32')
  testX = testX.astype('float32')
  # scale from [0,255] to [-1,1]
  trainX = (trainX - 127.5) / 127.5
  testX = (testX - 127.5) / 127.5

  return [trainX, trainy], [testX, testy]
```

Select a supervised subset of the dataset with balanced classes:

Select a subset of the training dataset in which we keep the labels and train the supervised version of the discriminator model.

This function is careful to ensure that the selection of examples is random and that the classes are balanced. The number of labeled examples is parameterized and set at 100, meaning that each of the 10 classes will have 10 randomly selected examples.

[7]:
```
def select_supervised_samples(dataset, n_samples= 100, n_classes=10):
        X, y = dataset
        X_list, y_list = list(), list()
        n_per_class = int(n_samples / n_classes)
        for i in range(n_classes):
                # get all images for this class
                X_with_class = X[y == i]
                # choose random instances
```

```
            ix = randint(0, len(X_with_class), n_per_class)
            # add to list
            [X_list.append(X_with_class[j]) for j in ix]
            [y_list.append(i) for j in ix]
        return asarray(X_list), asarray(y_list)
```

Select real samples:

A sample of images and labels is selected, with replacement. This same function can be used to retrieve examples from the labeled and unlabeled dataset, later when we train the models. In the case of the "unlabeled dataset", we will ignore the labels.

[8]:
```
def generate_real_samples(dataset, n_samples):
        # split into images and labels
        images, labels = dataset
        # choose random instances
        ix = randint(0, images.shape[0], n_samples)
        # select images and labels
        X, labels = images[ix], labels[ix]
        # generate class labels
        y = ones((n_samples, 1))
        return [X, labels], y
```

Generate points in latent space as input for the generator:

First, the generate_latent_points() function will create a batch worth of random points in the latent space that can be used as input for generating images. The generate_fake_samples() function will call this function to generate a batch worth of images that can be fed to the unsupervised discriminator model or the composite GAN model during training.

[9]:
```
def generate_latent_points(latent_dim, n_samples):
        # generate points in the latent space
        z_input = randn(latent_dim * n_samples)
        # reshape into a batch of inputs for the network
        z_input = z_input.reshape(n_samples, latent_dim)
        return z_input
```

Use the generator to generate n fake examples, with class labels:

[10]:
```
def generate_fake_samples(generator, latent_dim, n_samples):
        # generate points in latent space
        z_input = generate_latent_points(latent_dim, n_samples)
        # predict outputs
        images = generator.predict(z_input)
        # create class labels
        y = zeros((n_samples, 1))
        return images, y
```

Generate samples for drawing plot:

[11]:
```
def summarize_performance(step, g_model, c_model, latent_dim, dataset,␣
↪n_samples=100):
        # prepare fake examples
        global X
        X, _ = generate_fake_samples(g_model, latent_dim, n_samples)
        # scale from [-1,1] to [0,1]
        X = (X + 1) / 2.0
        # evaluate the classifier model
        X, y = dataset
        _, acc = c_model.evaluate(X, y, verbose=0)
        print('Classifier Accuracy: %.3f%%' % (acc * 100))
```

Train the generator and discriminator:

We can define a function to train the models. The defined models and loaded training dataset are provided as arguments, and the number of training epochs and batch size are arbitrary.

The chosen model configuration was found to overfit the training dataset quickly, hence the relatively smaller number of training epochs. Increasing the epochs to 100 or more results in much higher-quality generated images, but a lower-quality classifier model. Balancing these two concerns might make a fun extension.

First, the labeled subset of the training dataset is selected, and the number of training steps is calculated.

A single cycle through updating the models involves first updating the supervised discriminator model with labeled examples, then updating the unsupervised discriminator model with unlabeled real and generated examples. Finally, the generator model is updated via the composite model.

The shared weights of the discriminator model get updated with 1.5 batches worth of samples, whereas the weights of the generator model are updated with one batch worth of samples each iteration. Changing this so that each model is updated by the same amount might improve the model training process.

[12]:
```
def train(g_model, d_model, c_model, gan_model, dataset, latent_dim, n_epochs,␣
↪n_batch=100):
        # select supervised dataset
        X_sup, y_sup = select_supervised_samples(dataset)
        print(X_sup.shape, y_sup.shape)
        # calculate the number of batches per training epoch
        bat_per_epo = int(dataset[0].shape[0] / n_batch)
        # calculate the number of training iterations
        n_steps = bat_per_epo * n_epochs
        # calculate the size of half a batch of samples
        half_batch = int(n_batch / 2)
        print('n_epochs=%d, n_batch=%d, 1/2=%d, b/e=%d, steps=%d' % (n_epochs,␣
↪n_batch, half_batch, bat_per_epo, n_steps))
        # manually enumerate epochs
```

```
        for i in range(n_steps):
                # update supervised discriminator (c)
                [Xsup_real, ysup_real], _ = generate_real_samples([X_sup,␣
↪y_sup], half_batch)
                c_loss, c_acc = c_model.train_on_batch(Xsup_real, ysup_real)
                # update unsupervised discriminator (d)
                [X_real, _], y_real = generate_real_samples(dataset, half_batch)
                d_loss1 = d_model.train_on_batch(X_real, y_real)
                X_fake, y_fake = generate_fake_samples(g_model, latent_dim,␣
↪half_batch)
                d_loss2 = d_model.train_on_batch(X_fake, y_fake)
                # update generator (g)
                X_gan, y_gan = generate_latent_points(latent_dim, n_batch),␣
↪ones((n_batch, 1))
                g_loss = gan_model.train_on_batch(X_gan, y_gan)
                # evaluate the model performance every so often
                if (i+1) % (bat_per_epo * 1) == 0:
                        summarize_performance(i, g_model, c_model, latent_dim,␣
↪dataset)
```

Size of the latent space:

[13]:
```
latent_dim = 100
```

Create the discriminator models:

[14]:
```
d_model, c_model = define_discriminator()
```

```
/usr/local/lib/python3.7/dist-packages/keras/optimizer_v2/adam.py:105:
UserWarning: The `lr` argument is deprecated, use `learning_rate` instead.
  super(Adam, self).__init__(name, **kwargs)
```

Create the generator:

[15]:
```
g_model = define_generator(latent_dim)
```

Create the gan:

[16]:
```
gan_model = define_gan(g_model, d_model)
```

```
/usr/local/lib/python3.7/dist-packages/keras/optimizer_v2/adam.py:105:
UserWarning: The `lr` argument is deprecated, use `learning_rate` instead.
  super(Adam, self).__init__(name, **kwargs)
```

Load image data:

[17]:
```
train_dataset, test_dataset = load_real_samples()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-
datasets/mnist.npz
```

```
11493376/11490434 [==============================] - 0s 0us/step
11501568/11490434 [==============================] - 0s 0us/step
```

Train model:

[18]:
```
train(g_model, d_model, c_model, gan_model, train_dataset, latent_dim, 5)
```

```
(100, 28, 28, 1) (100,)
n_epochs=5, n_batch=100, 1/2=50, b/e=600, steps=3000
Classifier Accuracy: 84.958%
Classifier Accuracy: 90.307%
Classifier Accuracy: 91.758%
Classifier Accuracy: 92.840%
Classifier Accuracy: 93.195%
```

[19]:
```
_, acc = c_model.evaluate(test_dataset[0], test_dataset[1], verbose=0)
print('Accuracy: %.3f%%' % (acc * 100))
```

```
Accuracy: 93.860%
```

1.3

[20]:
```
def model_compile(loss, optimizer, batch_s):

    X_train, y_train = select_supervised_samples(train_dataset)

    model.compile(loss = loss,
                  optimizer = optimizer,
                  metrics = ['accuracy'])

    global model_history
    model_history = model.fit(X_train, y_train, epochs=500, batch_size = batch_s)
```

[21]:
```
model = keras.models.Sequential()
model.add(Conv2D(filters=16, kernel_size=(3,3), strides=1, activation='relu',␣
 ↪padding='same', input_shape=(28, 28, 1)))
model.add(Conv2D(filters=16, kernel_size=(3,3), strides=1, activation='relu',␣
 ↪padding='same'))
model.add(BatchNormalization())
model.add(MaxPooling2D(2, 2))
model.add(Dropout(0.2))

model.add(Conv2D(filters=32, kernel_size=(3,3), strides=1, activation='relu',␣
 ↪padding='same'))
model.add(Conv2D(filters=32, kernel_size=(3,3), strides=1, activation='relu',␣
 ↪padding='same'))
model.add(BatchNormalization())
model.add(MaxPooling2D(2, 2))
```

```
model.add(Dropout(0.2))

model.add(Conv2D(filters=64, kernel_size=(3,3), strides=1, activation='relu',␣
 ↪padding='same'))
model.add(Conv2D(filters=64, kernel_size=(3,3), strides=1, activation='relu',␣
 ↪padding='same'))
model.add(BatchNormalization())
model.add(MaxPooling2D(2, 2))
model.add(Dropout(0.2))


model.add(keras.layers.Flatten())
model.add(keras.layers.Dense(64, activation = 'relu'))
model.add(BatchNormalization())
model.add(Dropout(0.2))
model.add(keras.layers.Dense(10, activation = 'softmax'))
```

[22]:
```
model_compile('sparse_categorical_crossentropy', 'adam', 32)
```

```
Epoch 1/500
4/4 [==============================] - 2s 70ms/step - loss: 3.0396 - accuracy:
0.0800
...
Epoch 500/500
4/4 [==============================] - 0s 7ms/step - loss: 0.0172 - accuracy:
1.0000
```

CNN accuracy:

[23]:
```
_, acc_cnn = model.evaluate(test_dataset[0], test_dataset[1], verbose=0)
print('Accuracy: %.3f%%' % (acc_cnn * 100))
```

```
Accuracy: 87.180%
```

Table 1: Classifier accuracy

Examples	CNN	SGAN
100	0.8718	0.9386

1.4

VAE:

Two variables to hold the size and number of channels in the image.

[24]:
```
img_size = 28
num_channels = 1
```

[25]:
```
# create the input layer of the encoder
x = Input(shape=(img_size, img_size, num_channels), name="encoder_input")

# combination of 2d convolution, batch normalization, and leakyReLu
encoder_conv_layer1 = Conv2D(filters=1, kernel_size=(3, 3), padding="same",␣
 ↪strides=1, name="encoder_conv_1")(x)
encoder_norm_layer1 =␣
 ↪BatchNormalization(name="encoder_norm_1")(encoder_conv_layer1)
encoder_activ_layer1 = LeakyReLU(name="encoder_leakyrelu_1")(encoder_norm_layer1)

# other combinations of conv-norm-relu layers.
encoder_conv_layer2 = Conv2D(filters=32, kernel_size=(3,3), padding="same",␣
 ↪strides=1, name="encoder_conv_2")(encoder_activ_layer1)
encoder_norm_layer2 =␣
 ↪BatchNormalization(name="encoder_norm_2")(encoder_conv_layer2)
encoder_activ_layer2 =␣
 ↪LeakyReLU(name="encoder_activ_layer_2")(encoder_norm_layer2)

encoder_conv_layer3 = Conv2D(filters=64, kernel_size=(3,3), padding="same",␣
 ↪strides=2, name="encoder_conv_3")(encoder_activ_layer2)
encoder_norm_layer3 =␣
 ↪BatchNormalization(name="encoder_norm_3")(encoder_conv_layer3)
encoder_activ_layer3 =␣
 ↪LeakyReLU(name="encoder_activ_layer_3")(encoder_norm_layer3)

encoder_conv_layer4 = Conv2D(filters=64, kernel_size=(3,3), padding="same",␣
 ↪strides=2, name="encoder_conv_4")(encoder_activ_layer3)
encoder_norm_layer4 =␣
 ↪BatchNormalization(name="encoder_norm_4")(encoder_conv_layer4)
encoder_activ_layer4 =␣
 ↪LeakyReLU(name="encoder_activ_layer_4")(encoder_norm_layer4)

encoder_conv_layer5 = Conv2D(filters=64, kernel_size=(3,3), padding="same",␣
 ↪strides=1, name="encoder_conv_5")(encoder_activ_layer4)
encoder_norm_layer5 =␣
 ↪BatchNormalization(name="encoder_norm_5")(encoder_conv_layer5)
encoder_activ_layer5 =␣
 ↪LeakyReLU(name="encoder_activ_layer_5")(encoder_norm_layer5)

# hold the shape of the result before being flattened, in order to decode the␣
 ↪result successfully.
shape_before_flatten = tf.keras.backend.int_shape(encoder_activ_layer5)[1:]
encoder_flatten = Flatten()(encoder_activ_layer5)

# create two layers mean and variance.
encoder_mu = Dense(units=latent_dim, name="encoder_mu")(encoder_flatten)
```

```
encoder_log_variance = Dense(units=latent_dim,␣
 ↪name="encoder_log_variance")(encoder_flatten)

encoder_mu_log_variance_model = Model(x, (encoder_mu, encoder_log_variance),␣
 ↪name="encoder_mu_log_variance_model")

def sampling(mu_log_variance):
    mu, log_variance = mu_log_variance
    epsilon = tf.keras.backend.random_normal(shape=tf.keras.backend.shape(mu),␣
 ↪mean=0.0, stddev=1.0)
    random_sample = mu + tf.keras.backend.exp(log_variance/2) * epsilon
    return random_sample

encoder_output = Lambda(sampling, name="encoder_output")([encoder_mu,␣
 ↪encoder_log_variance])

# create a model that associates the input layer to the output layer of the␣
 ↪encoder
encoder = Model(x, encoder_output, name="encoder_model")
```

Building the Decoder:

In the previous section, the encoder accepted an input of shape (28, 28) and returned a vector of length 2. In this section, the decoder should do the reverse: accept an input vector of length 2, and return a result of shape (28, 28).

[26]:
```
# create a layer which holds the input
decoder_input = Input(shape=(latent_dim), name="decoder_input")

# expand the length of the vector from 2 to the value specified into the␣
 ↪shape_before_flatten variable
decoder_dense_layer1 = Dense(units=prod(shape_before_flatten),␣
 ↪name="decoder_dense_1")(decoder_input)

# reshape the result from a vector to a matrix
decoder_reshape =␣
 ↪Reshape(target_shape=shape_before_flatten)(decoder_dense_layer1)

# add a number of layers that expand the shape until reaching the desired shape␣
 ↪of the original input
decoder_conv_tran_layer1 = Conv2DTranspose(filters=64, kernel_size=(3, 3),␣
 ↪padding="same", strides=1, name="decoder_conv_tran_1")(decoder_reshape)
decoder_norm_layer1 =␣
 ↪BatchNormalization(name="decoder_norm_1")(decoder_conv_tran_layer1)
decoder_activ_layer1 = LeakyReLU(name="decoder_leakyrelu_1")(decoder_norm_layer1)
```

```
decoder_conv_tran_layer2 = Conv2DTranspose(filters=64, kernel_size=(3, 3),␣
 ↪padding="same", strides=2, name="decoder_conv_tran_2")(decoder_activ_layer1)
decoder_norm_layer2 =␣
 ↪BatchNormalization(name="decoder_norm_2")(decoder_conv_tran_layer2)
decoder_activ_layer2 = LeakyReLU(name="decoder_leakyrelu_2")(decoder_norm_layer2)

decoder_conv_tran_layer3 = Conv2DTranspose(filters=64, kernel_size=(3, 3),␣
 ↪padding="same", strides=2, name="decoder_conv_tran_3")(decoder_activ_layer2)
decoder_norm_layer3 =␣
 ↪BatchNormalization(name="decoder_norm_3")(decoder_conv_tran_layer3)
decoder_activ_layer3 = LeakyReLU(name="decoder_leakyrelu_3")(decoder_norm_layer3)

decoder_conv_tran_layer4 = Conv2DTranspose(filters=1, kernel_size=(3, 3),␣
 ↪padding="same", strides=1, name="decoder_conv_tran_4")(decoder_activ_layer3)
decoder_output = LeakyReLU(name="decoder_output")(decoder_conv_tran_layer4 )

# create a model that links the input and output of the decoder.
decoder = Model(decoder_input, decoder_output, name="decoder_model")
```

Implementation of the loss function:

[27]:
```
def loss_func(encoder_mu, encoder_log_variance):
    def vae_reconstruction_loss(y_true, y_predict):
        reconstruction_loss_factor = 1000
        reconstruction_loss = tf.keras.backend.mean(tf.keras.backend.
 ↪square(y_true-y_predict), axis=[1, 2, 3])
        return reconstruction_loss_factor * reconstruction_loss

    def vae_kl_loss(encoder_mu, encoder_log_variance):
        kl_loss = -0.5 * tf.keras.backend.sum(1.0 + encoder_log_variance - tf.
 ↪keras.backend.square(encoder_mu) - tf.keras.backend.exp(encoder_log_variance),␣
 ↪axis=1)
        return kl_loss

    def vae_kl_loss_metric(y_true, y_predict):
        kl_loss = -0.5 * tf.keras.backend.sum(1.0 + encoder_log_variance - tf.
 ↪keras.backend.square(encoder_mu) - tf.keras.backend.exp(encoder_log_variance),␣
 ↪axis=1)
        return kl_loss

    def vae_loss(y_true, y_predict):
        reconstruction_loss = vae_reconstruction_loss(y_true, y_predict)
        kl_loss = vae_kl_loss(y_true, y_predict)

        loss = reconstruction_loss + kl_loss
        return loss
```

```
    return vae_loss
```

Building the VAE:

Build a third model that combines previous models.

[28]:
```
# an input layer representing the input to the VAE (which is identical to that␣
 ↪of the encoder).
vae_input = Input(shape=(img_size, img_size, num_channels), name="VAE_input")

vae_encoder_output = encoder(vae_input)
# The output of the encoder is then connected to the decoder to reconstruct the␣
 ↪input.
vae_decoder_output = decoder(vae_encoder_output)

# the model of the VAE that links the encoder to the decoder
vae = Model(vae_input, vae_decoder_output, name="VAE")

vae.compile(optimizer=tf.keras.optimizers.Adam(lr=0.0005),␣
 ↪loss=loss_func(encoder_mu, encoder_log_variance))
```

```
/usr/local/lib/python3.7/dist-packages/keras/optimizer_v2/adam.py:105:
UserWarning: The `lr` argument is deprecated, use `learning_rate` instead.
  super(Adam, self).__init__(name, **kwargs)
```

Training the VAE:

[29]:
```
X_train, y_train = select_supervised_samples(train_dataset)
```

[30]:
```
vae.fit(train_dataset[0], train_dataset[0], epochs=5, batch_size=32,␣
 ↪shuffle=True, validation_data=(test_dataset[0], test_dataset[0]))
```

```
Epoch 1/5
1875/1875 [==============================] - 15s 7ms/step - loss: 57.9733 -
val_loss: 30.4919
Epoch 2/5
1875/1875 [==============================] - 14s 8ms/step - loss: 28.5795 -
val_loss: 27.0838
Epoch 3/5
1875/1875 [==============================] - 14s 7ms/step - loss: 25.8092 -
val_loss: 25.5870
Epoch 4/5
1875/1875 [==============================] - 14s 7ms/step - loss: 24.5672 -
val_loss: 24.4876
Epoch 5/5
1875/1875 [==============================] - 13s 7ms/step - loss: 23.8311 -
val_loss: 23.3687
```

[30]:
```
<keras.callbacks.History at 0x7eff787a9ed0>
```

Encode and then decode data with VAE:

[31]:
```
encoded_data = encoder.predict(test_dataset[0])
decoded_data = decoder.predict(encoded_data)
```

Accuracy of VAE:

[32]:
```
_, acc_vae = c_model.evaluate(decoded_data, test_dataset[1], verbose=0)
print('Accuracy: %.3f%%' % (acc_vae * 100))
```

```
Accuracy: 93.680%
```

Table 2: Classifier accuracy

Examples	CNN	SGAN	VAE
100	0.8718	0.9386	0.9368

Outputs of SGAN and VAE:

[33]:
```
pyplot.figure(figsize=(15,15))
for i in range(100):
        # define subplot
        pyplot.subplot(10, 10, 1 + i)
        # turn off axis
        pyplot.axis('off')
        # plot raw pixel data
        pyplot.imshow(X[i, :, :, 0], cmap='gray')
```

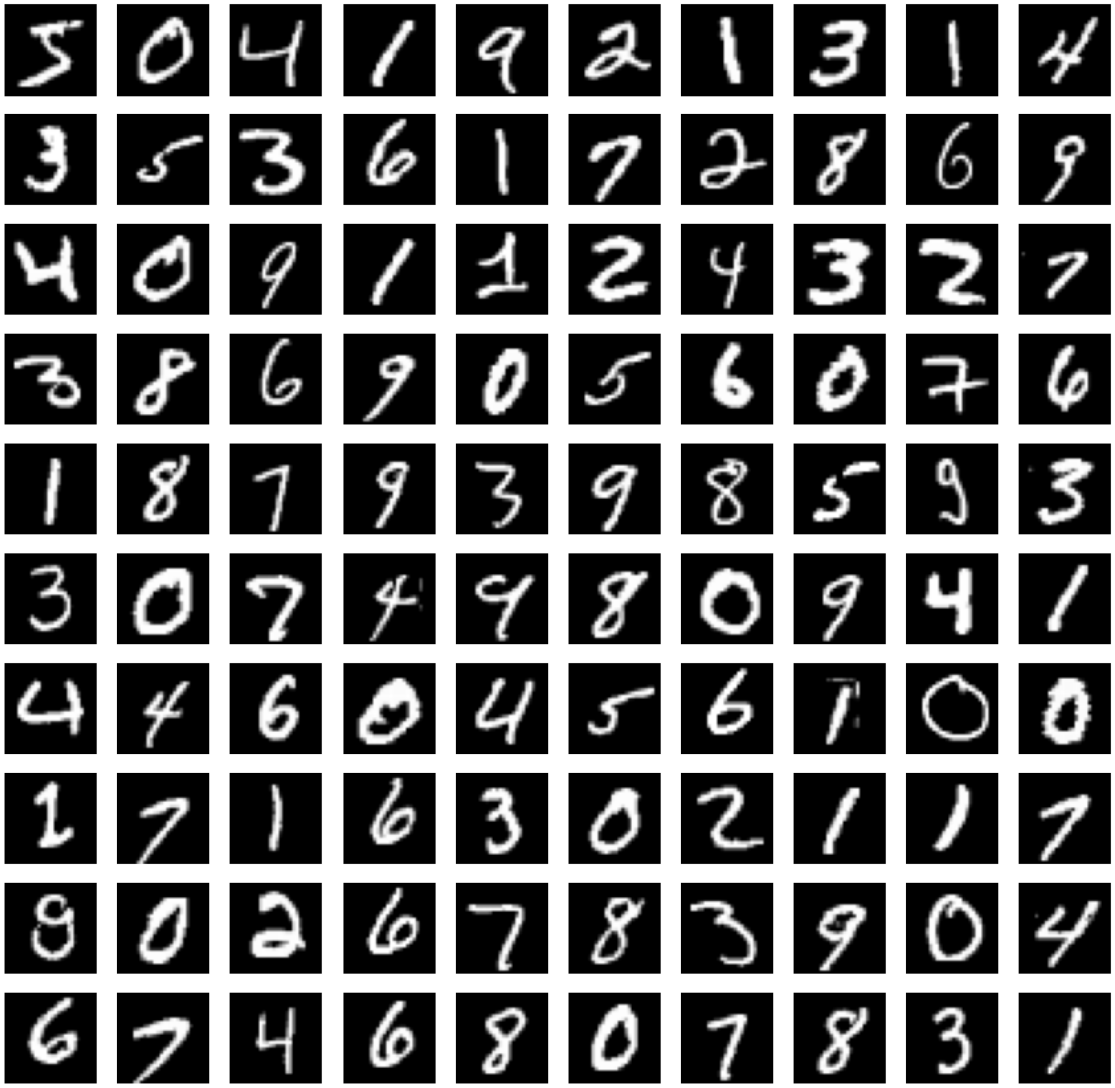

Figure 1: Output samples from SGAN after 5 MNIST epochs

[34]:
```
pyplot.figure(figsize=(15,15))
for i in range(100):
        # define subplot
        pyplot.subplot(10, 10, 1 + i)
        # turn off axis
        pyplot.axis('off')
        # plot raw pixel data
        pyplot.imshow(decoded_data[i, :, :, 0], cmap='gray')
```

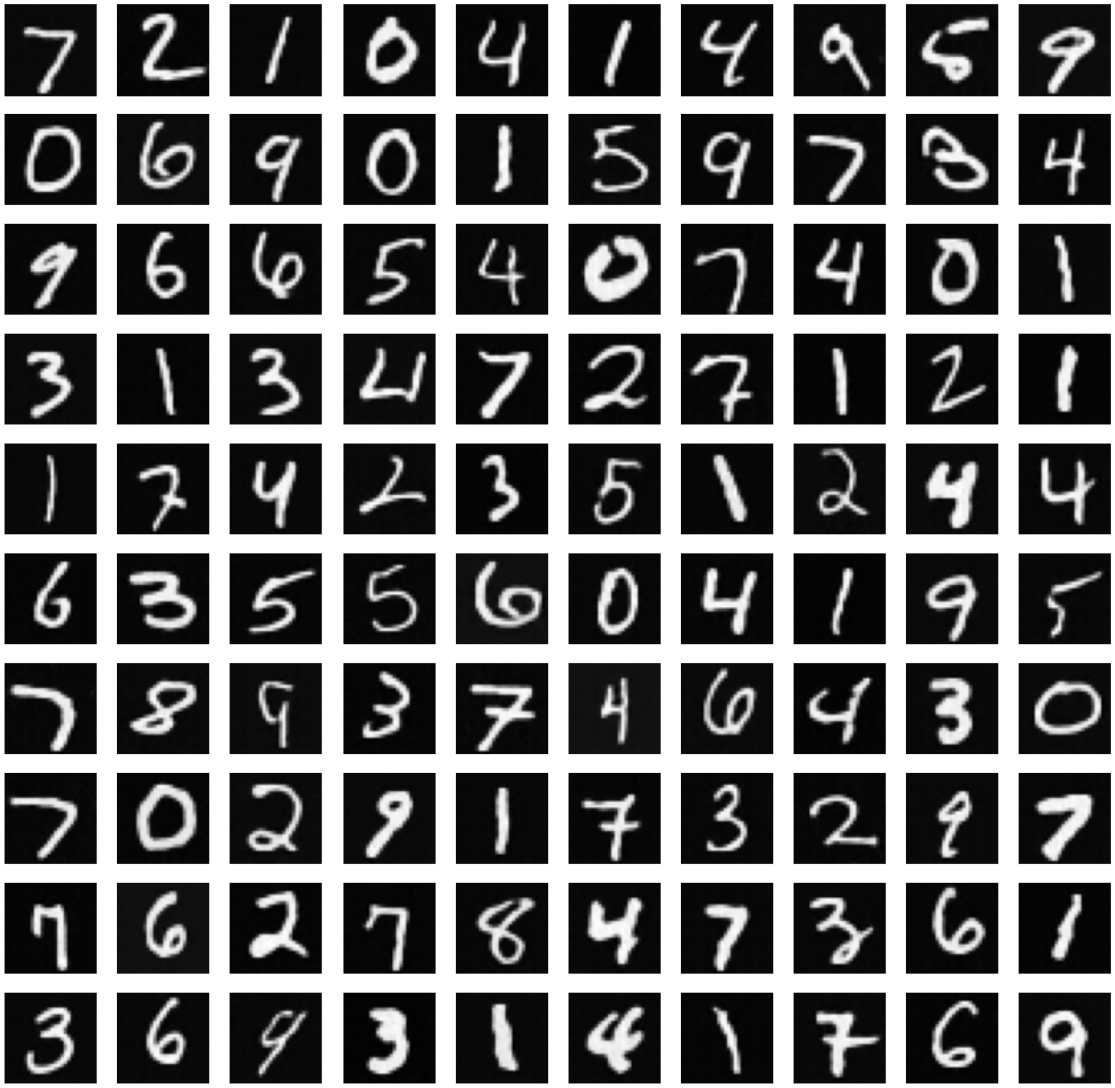

Figure 2: Output samples from VAE after 5 MNIST epochs

References:

https://machinelearningmastery.com/semi-supervised-generative-adversarial-network/

https://keras.io/examples/generative/vae/

2 DCGAN

[41]:
```
! pip install kaggle
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-
wheels/public/simple/
Requirement already satisfied: kaggle in /usr/local/lib/python3.7/dist-packages
(1.5.12)
Requirement already satisfied: certifi in /usr/local/lib/python3.7/dist-packages
(from kaggle) (2022.5.18.1)
Requirement already satisfied: tqdm in /usr/local/lib/python3.7/dist-packages
(from kaggle) (4.64.0)
Requirement already satisfied: urllib3 in /usr/local/lib/python3.7/dist-packages
(from kaggle) (1.24.3)
Requirement already satisfied: python-dateutil in /usr/local/lib/python3.7/dist-
packages (from kaggle) (2.8.2)
Requirement already satisfied: python-slugify in /usr/local/lib/python3.7/dist-
packages (from kaggle) (6.1.2)
Requirement already satisfied: requests in /usr/local/lib/python3.7/dist-
packages (from kaggle) (2.23.0)
Requirement already satisfied: six>=1.10 in /usr/local/lib/python3.7/dist-
packages (from kaggle) (1.15.0)
Requirement already satisfied: text-unidecode>=1.3 in
/usr/local/lib/python3.7/dist-packages (from python-slugify->kaggle) (1.3)
Requirement already satisfied: chardet<4,>=3.0.2 in
/usr/local/lib/python3.7/dist-packages (from requests->kaggle) (3.0.4)
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.7/dist-
packages (from requests->kaggle) (2.10)
```

[42]:
```
! mkdir ~/.kaggle
```

```
mkdir: cannot create directory ‘/root/.kaggle’: File exists
```

[43]:
```
! cp kaggle.json ~/.kaggle/
```

[44]:
```
! chmod 600 ~/.kaggle/kaggle.json
```

[45]:
```
from kaggle.api.kaggle_api_extended import KaggleApi
api = KaggleApi()
api.authenticate()
```

[46]:
```
api.dataset_download_files('bryanb/abstract-art-gallery')
```

[47]:
```
from zipfile import ZipFile
from __future__ import print_function
#%matplotlib inline
import argparse
```

```
import os
import random
import torch
import torch.nn as nn
import torch.nn.parallel
import torch.backends.cudnn as cudnn
import torch.optim as optim
import torch.utils.data
import torchvision.datasets as dset
import torchvision.transforms as transforms
import torchvision.utils as vutils
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from IPython.display import HTML

# Set random seed for reproducibility
manualSeed = random.randint(1, 10000)
print("Random Seed: ", manualSeed)
random.seed(manualSeed)
torch.manual_seed(manualSeed)
```

[48]:
```
zf = ZipFile('abstract-art-gallery.zip')
zf.extractall()
zf.close()
```

2.1

A DCGAN is a direct extension of the GAN, except that it explicitly uses convolutional and convolutional-transpose layers in the discriminator and generator, respectively. The discriminator is made up of strided convolution layers, batch norm layers, and LeakyReLU activations. The input is a image and the output is a scalar probability that the input is from the real data distribution. The generator is comprised of convolutional-transpose layers, batch norm layers, and ReLU activations. The input is a latent vector, zz, that is drawn from a standard normal distribution and the output is a RGB image. The strided conv-transpose layers allow the latent vector to be transformed into a volume with the same shape as an image. In the paper, the authors also give some tips about how to setup the optimizers, how to calculate the loss functions, and how to initialize the model weights.

2.2

[50]:
```
# Root directory for dataset
dataroot = "/content/Abstract_gallery"

# Number of workers for dataloader
workers = 2
```

```
# Batch size during training
batch_size = 128

# Spatial size of training images. All images will be resized to this
#   size using a transformer.
image_size = 64

# Number of channels in the training images. For color images this is 3
nc = 3

# Size of z latent vector (i.e. size of generator input)
nz = 100

# Size of feature maps in generator
ngf = 64

# Size of feature maps in discriminator
ndf = 64

# Number of training epochs
num_epochs = 200

# Learning rate for optimizers
lr = 0.0002

# Beta1 hyperparam for Adam optimizers
beta1 = 0.5

# Number of GPUs available. Use 0 for CPU mode.
ngpu = 1
```

[51]:
```
# Create the dataset
dataset = dset.ImageFolder(root=dataroot,
                           transform=transforms.Compose([
                               transforms.Resize(image_size),
                               transforms.CenterCrop(image_size),
                               transforms.ToTensor(),
                               transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5,␣
 ↪0.5)),
                           ]))
# Create the dataloader
dataloader = torch.utils.data.DataLoader(dataset, batch_size=batch_size,
                                         shuffle=True, num_workers=workers)

# Decide which device we want to run on
device = torch.device("cuda:0" if (torch.cuda.is_available() and ngpu > 0) else␣
 ↪"cpu")
```

```
# Plot some training images
real_batch = next(iter(dataloader))
plt.figure(figsize=(15,15))
plt.axis("off")
plt.title("Training Images")
plt.imshow(np.transpose(vutils.make_grid(real_batch[0].to(device)[:64],␣
 ↪padding=2, normalize=True).cpu(),(1,2,0)))
```

[51]: <matplotlib.image.AxesImage at 0x7f1e7ec9e190>

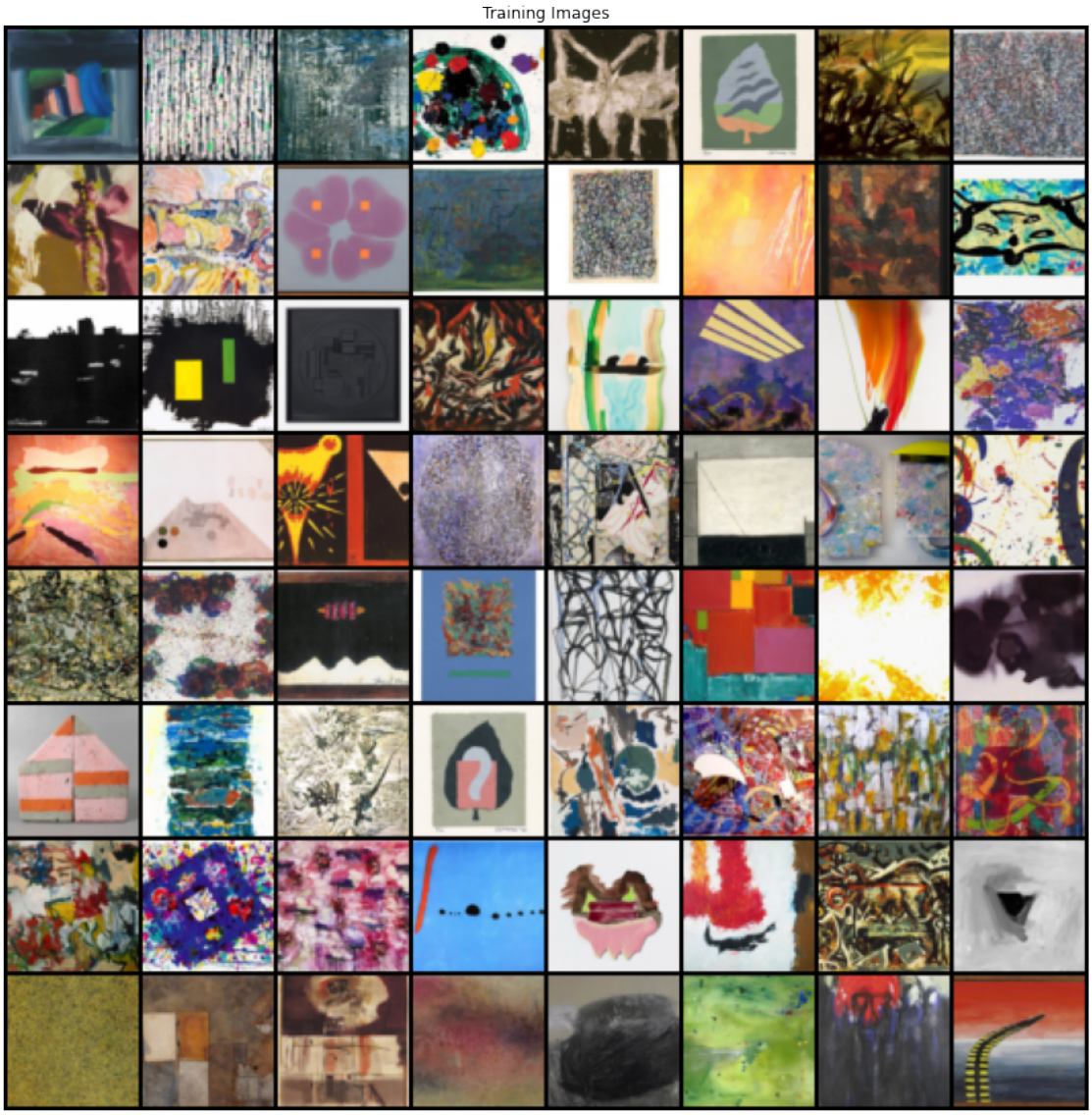

Figure 3: Some of training images

Weight Initialization:

From the DCGAN paper, the authors specify that all model weights shall be randomly initialized from a Normal distribution with $mean = 0$, $stdev = 0.02$.

[52]:
```
# custom weights initialization called on netG and netD
def weights_init(m):
    classname = m.__class__.__name__
    if classname.find('Conv') != -1:
        nn.init.normal_(m.weight.data, 0.0, 0.02)
    elif classname.find('BatchNorm') != -1:
        nn.init.normal_(m.weight.data, 1.0, 0.02)
        nn.init.constant_(m.bias.data, 0)
```

Generator:

The generator, GG, is designed to map the latent space vector (zz) to data-space. Since our data are images, converting zz to data-space means ultimately creating a RGB image with the same size as the training images. In practice, this is accomplished through a series of strided two dimensional convolutional transpose layers, each paired with a 2d batch norm layer and a relu activation. The output of the generator is fed through a tanh function to return it to the input data range of $[-1, 1][-1, 1]$. It is worth noting the existence of the batch norm functions after the conv-transpose layers, as this is a critical contribution of the DCGAN paper. These layers help with the flow of gradients during training. An image of the generator from the DCGAN paper is shown below.

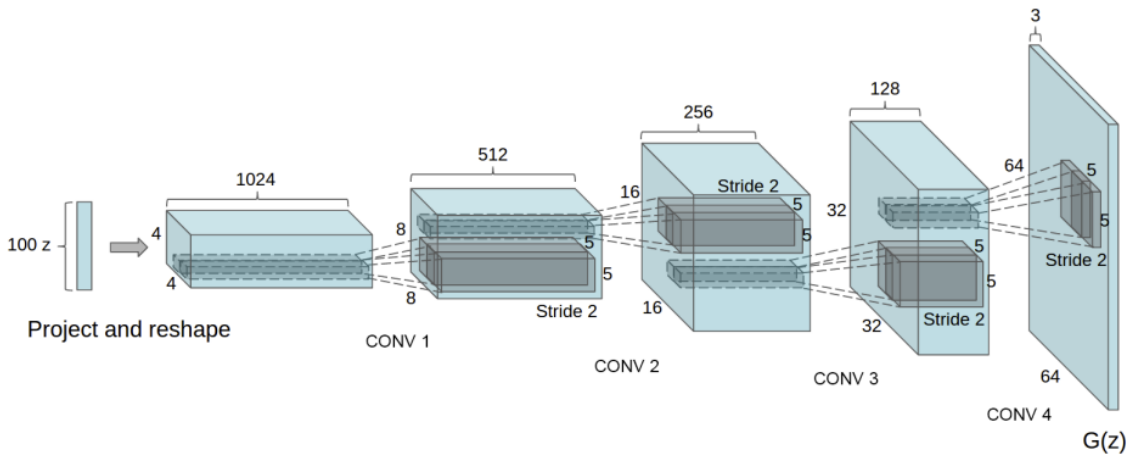

Figure 4: Architecture of generator

[53]:
```
# Generator Code

class Generator(nn.Module):
    def __init__(self, ngpu):
        super(Generator, self).__init__()
        self.ngpu = ngpu
        self.main = nn.Sequential(
            # input is Z, going into a convolution
            nn.ConvTranspose2d( nz, ngf * 8, 4, 1, 0, bias=False),
            nn.BatchNorm2d(ngf * 8),
            nn.ReLU(True),
            # state size. (ngf*8) x 4 x 4
            nn.ConvTranspose2d(ngf * 8, ngf * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 4),
            nn.ReLU(True),
            # state size. (ngf*4) x 8 x 8
            nn.ConvTranspose2d( ngf * 4, ngf * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 2),
            nn.ReLU(True),
            # state size. (ngf*2) x 16 x 16
            nn.ConvTranspose2d( ngf * 2, ngf, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf),
            nn.ReLU(True),
            # state size. (ngf) x 32 x 32
            nn.ConvTranspose2d( ngf, nc, 4, 2, 1, bias=False),
            nn.Tanh()
            # state size. (nc) x 64 x 64
        )

    def forward(self, input):
        return self.main(input)
```

[54]:
```
# Create the generator
netG = Generator(ngpu).to(device)

# Handle multi-gpu if desired
if (device.type == 'cuda') and (ngpu > 1):
    netG = nn.DataParallel(netG, list(range(ngpu)))

# Apply the weights_init function to randomly initialize all weights
#  to mean=0, stdev=0.02.
netG.apply(weights_init)

# Print the model
print(netG)
```

```
Generator(
```

```
  (main): Sequential(
    (0): ConvTranspose2d(100, 512, kernel_size=(4, 4), stride=(1, 1),
bias=False)
    (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): ConvTranspose2d(512, 256, kernel_size=(4, 4), stride=(2, 2),
padding=(1, 1), bias=False)
    (4): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (5): ReLU(inplace=True)
    (6): ConvTranspose2d(256, 128, kernel_size=(4, 4), stride=(2, 2),
padding=(1, 1), bias=False)
    (7): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (8): ReLU(inplace=True)
    (9): ConvTranspose2d(128, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1,
1), bias=False)
    (10): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (11): ReLU(inplace=True)
    (12): ConvTranspose2d(64, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1,
1), bias=False)
    (13): Tanh()
  )
)
```

Discriminator:

The discriminator, DD, is a binary classification network that takes an image as input and outputs a scalar probability that the input image is real (as opposed to fake). There is significance to the use of the strided convolution, BatchNorm, and LeakyReLUs. The DCGAN paper mentions it is a good practice to use strided convolution rather than pooling to downsample because it lets the network learn its own pooling function. Also batch norm and leaky relu functions promote healthy gradient flow which is critical for the learning process of both GG and DD.

[55]:
```python
class Discriminator(nn.Module):
    def __init__(self, ngpu):
        super(Discriminator, self).__init__()
        self.ngpu = ngpu
        self.main = nn.Sequential(
            # input is (nc) x 64 x 64
            nn.Conv2d(nc, ndf, 4, 2, 1, bias=False),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (ndf) x 32 x 32
            nn.Conv2d(ndf, ndf * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 2),
            nn.LeakyReLU(0.2, inplace=True),
```

```
            # state size. (ndf*2) x 16 x 16
            nn.Conv2d(ndf * 2, ndf * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 4),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (ndf*4) x 8 x 8
            nn.Conv2d(ndf * 4, ndf * 8, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 8),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (ndf*8) x 4 x 4
            nn.Conv2d(ndf * 8, 1, 4, 1, 0, bias=False),
            nn.Sigmoid()
        )

    def forward(self, input):
        return self.main(input)
```

[56]:
```
# Create the Discriminator
netD = Discriminator(ngpu).to(device)

# Handle multi-gpu if desired
if (device.type == 'cuda') and (ngpu > 1):
    netD = nn.DataParallel(netD, list(range(ngpu)))

# Apply the weights_init function to randomly initialize all weights
#  to mean=0, stdev=0.2.
netD.apply(weights_init)

# Print the model
print(netD)
```

```
Discriminator(
  (main): Sequential(
    (0): Conv2d(3, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1),
bias=False)
    (1): LeakyReLU(negative_slope=0.2, inplace=True)
    (2): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1),
bias=False)
    (3): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (4): LeakyReLU(negative_slope=0.2, inplace=True)
    (5): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1),
bias=False)
    (6): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (7): LeakyReLU(negative_slope=0.2, inplace=True)
    (8): Conv2d(256, 512, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1),
bias=False)
```

```
    (9): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (10): LeakyReLU(negative_slope=0.2, inplace=True)
    (11): Conv2d(512, 1, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (12): Sigmoid()
  )
)
```

Loss Functions and Optimizers:

With DD and GG setup, we can specify how they learn through the loss functions and optimizers. We will use the Binary Cross Entropy loss (BCELoss) function.

Next, we define our real label as 1 and the fake label as 0. These labels will be used when calculating the losses of DD and GG, and this is also the convention used in the original GAN paper. Finally, we set up two separate optimizers, one for DD and one for GG. As specified in the DCGAN paper, both are Adam optimizers with learning rate 0.0002 and Beta1 = 0.5. For keeping track of the generator's learning progression, we will generate a fixed batch of latent vectors that are drawn from a Gaussian distribution (i.e. fixed_noise) . In the training loop, we will periodically input this fixed_noise into GG, and over the iterations we will see images form out of the noise.

[57]:
```
# Initialize BCELoss function
criterion = nn.BCELoss()

# Create batch of latent vectors that we will use to visualize
#  the progression of the generator
fixed_noise = torch.randn(64, nz, 1, 1, device=device)

# Establish convention for real and fake labels during training
real_label = 1.
fake_label = 0.

# Setup Adam optimizers for both G and D
optimizerD = optim.Adam(netD.parameters(), lr=lr, betas=(beta1, 0.999))
optimizerG = optim.Adam(netG.parameters(), lr=lr, betas=(beta1, 0.999))
```

Training:

Training is split up into two main parts. Part 1 updates the Discriminator and Part 2 updates the Generator.

Part 1 - Train the Discriminator

The goal of training the discriminator is to maximize the probability of correctly classifying a given input as real or fake. In terms of Goodfellow, we wish to "update the discriminator by ascending its stochastic gradient". Practically, we want to maximize $log(D(x)) + log(1 - D(G(z)))log(D(x)) + log(1 - D(G(z)))$. We will calculate this in two steps. First, we will construct a batch of real samples from the training set, forward pass through DD, calculate the loss $(log(D(x))log(D(x)))$, then calculate the gradients in a backward pass. Secondly, we will construct a batch of fake samples with the current generator, forward pass this batch through DD,

calculate the loss ($log(1 - D(G(z)))log(1 - D(G(z)))$), and accumulate the gradients with a backward pass. Now, with the gradients accumulated from both the all-real and all-fake batches, we call a step of the Discriminator's optimizer.

Part 2 - Train the Generator

We want to train the Generator by minimizing $log(1 - D(G(z)))log(1 - D(G(z)))$ in an effort to generate better fakes. we wish to maximize $log(D(G(z)))log(D(G(z)))$. We accomplish this by: classifying the Generator output from Part 1 with the Discriminator, computing G's loss using real labels as GT, computing G's gradients in a backward pass, and finally updating G's parameters with an optimizer step.

The training statistics reported are:

- Loss_D - discriminator loss calculated as the sum of losses for the all real and all fake batches ($log(D(x)) + log(1 - D(G(z)))log(D(x)) + log(1 - D(G(z)))$).
- Loss_G - generator loss calculated as $log(D(G(z)))log(D(G(z)))D(x)$ - the average output (across the batch) of the discriminator for the all real batch.
- D(G(z)) - average discriminator outputs for the all fake batch. The first number is before D is updated and the second number is after D is updated.

[58]:
```
# Training Loop

# Lists to keep track of progress
img_list = []
G_losses = []
D_losses = []
iters = 0

print("Starting Training Loop...")
# For each epoch
for epoch in range(num_epochs):
    # For each batch in the dataloader
    for i, data in enumerate(dataloader, 0):

        ############################
        # (1) Update D network: maximize log(D(x)) + log(1 - D(G(z)))
        ###########################
        ## Train with all-real batch
        netD.zero_grad()
        # Format batch
        real_cpu = data[0].to(device)
        b_size = real_cpu.size(0)
        label = torch.full((b_size,), real_label, dtype=torch.float,␣
 ↪device=device)
        # Forward pass real batch through D
        output = netD(real_cpu).view(-1)
        # Calculate loss on all-real batch
        errD_real = criterion(output, label)
```

```
        # Calculate gradients for D in backward pass
        errD_real.backward()
        D_x = output.mean().item()

        ## Train with all-fake batch
        # Generate batch of latent vectors
        noise = torch.randn(b_size, nz, 1, 1, device=device)
        # Generate fake image batch with G
        fake = netG(noise)
        label.fill_(fake_label)
        # Classify all fake batch with D
        output = netD(fake.detach()).view(-1)
        # Calculate D's loss on the all-fake batch
        errD_fake = criterion(output, label)
        # Calculate the gradients for this batch, accumulated (summed) with␣
↪previous gradients
        errD_fake.backward()
        D_G_z1 = output.mean().item()
        # Compute error of D as sum over the fake and the real batches
        errD = errD_real + errD_fake
        # Update D
        optimizerD.step()

        ############################
        # (2) Update G network: maximize log(D(G(z)))
        ###########################
        netG.zero_grad()
        label.fill_(real_label)  # fake labels are real for generator cost
        # Since we just updated D, perform another forward pass of all-fake␣
↪batch through D
        output = netD(fake).view(-1)
        # Calculate G's loss based on this output
        errG = criterion(output, label)
        # Calculate gradients for G
        errG.backward()
        D_G_z2 = output.mean().item()
        # Update G
        optimizerG.step()

        # Output training stats
        if i % 50 == 0:
            print('[%d/%d][%d/%d]\tLoss_D: %.4f\tLoss_G: %.4f\tD(x): %.
↪4f\tD(G(z)): %.4f / %.4f'
                  % (epoch, num_epochs, i, len(dataloader),
                     errD.item(), errG.item(), D_x, D_G_z1, D_G_z2))

        # Save Losses for plotting later
```

```
        G_losses.append(errG.item())
        D_losses.append(errD.item())

        # Check how the generator is doing by saving G's output on fixed_noise
        if (iters % 500 == 0) or ((epoch == num_epochs-1) and (i ==␣
↪len(dataloader)-1)):
            with torch.no_grad():
                fake = netG(fixed_noise).detach().cpu()
            img_list.append(vutils.make_grid(fake, padding=2, normalize=True))

        iters += 1
```

```
Starting Training Loop...
[0/200][0/22]   Loss_D: 1.7763  Loss_G: 4.5358  D(x): 0.3643    D(G(z)): 0.4109
/ 0.0142
[1/200][0/22]   Loss_D: 0.3179  Loss_G: 6.4311  D(x): 0.8033    D(G(z)): 0.0327
/ 0.0030
[2/200][0/22]   Loss_D: 0.5099  Loss_G: 19.2765 D(x): 0.7031    D(G(z)): 0.0000
/ 0.0000
[3/200][0/22]   Loss_D: 1.7704  Loss_G: 3.6306  D(x): 0.2905    D(G(z)): 0.0030
/ 0.0511
[4/200][0/22]   Loss_D: 2.1836  Loss_G: 2.8799  D(x): 0.2356    D(G(z)): 0.0120
/ 0.1086
[5/200][0/22]   Loss_D: 0.4552  Loss_G: 3.9982  D(x): 0.7749    D(G(z)): 0.1435
/ 0.0252
...
[196/200][0/22] Loss_D: 0.1117  Loss_G: 4.2317  D(x): 0.9512    D(G(z)): 0.0562
/ 0.0248
[197/200][0/22] Loss_D: 0.1054  Loss_G: 4.6048  D(x): 0.9407    D(G(z)): 0.0402
/ 0.0205
[198/200][0/22] Loss_D: 0.1696  Loss_G: 3.6391  D(x): 0.8958    D(G(z)): 0.0507
/ 0.0475
[199/200][0/22] Loss_D: 2.7705  Loss_G: 8.7728  D(x): 0.9951    D(G(z)): 0.8414
/ 0.0014
```

Results:

[59]:
```
plt.figure(figsize=(10,5))
plt.title("Generator and Discriminator Loss During Training")
plt.plot(G_losses,label="G")
plt.plot(D_losses,label="D")
plt.xlabel("iterations")
plt.ylabel("Loss")
plt.legend()
plt.show()
```

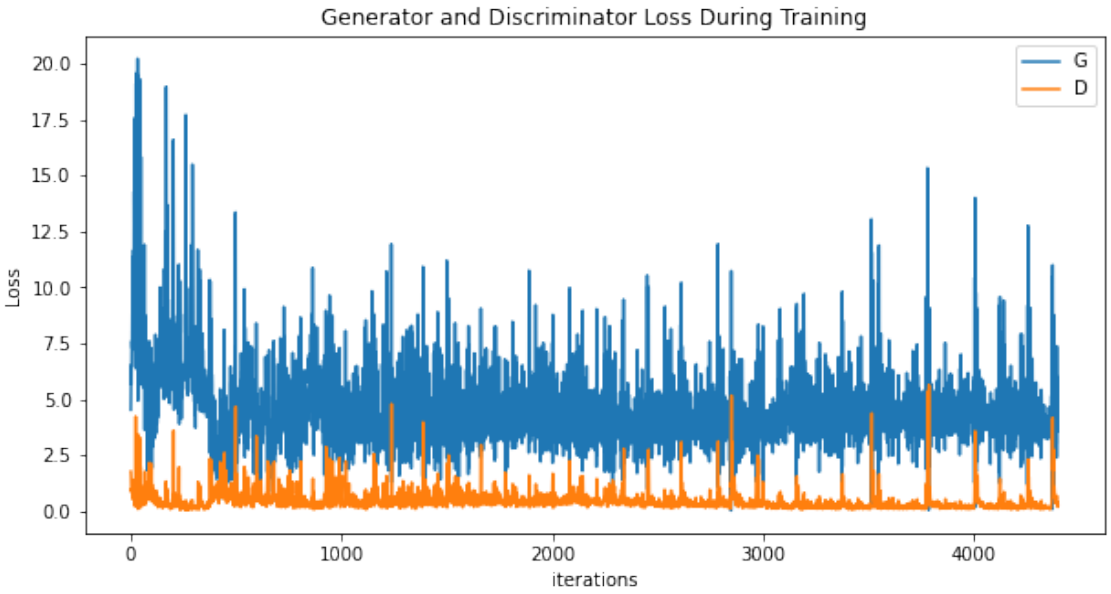

Figure 5: Generator and Discriminator loss during training

[63]:
```
# Grab a batch of real images from the dataloader
real_batch = next(iter(dataloader))

# Plot the real images
plt.figure(figsize=(25,20))
plt.subplot(1,2,1)
plt.axis("off")
plt.title("Real Images")
plt.imshow(np.transpose(vutils.make_grid(real_batch[0].to(device)[:64],␣
 ↪padding=5, normalize=True).cpu(),(1,2,0)))

# Plot the fake images from the last epoch
plt.subplot(1,2,2)
plt.axis("off")
plt.title("Fake Images")
plt.imshow(np.transpose(img_list[-1],(1,2,0)))
plt.show()
```

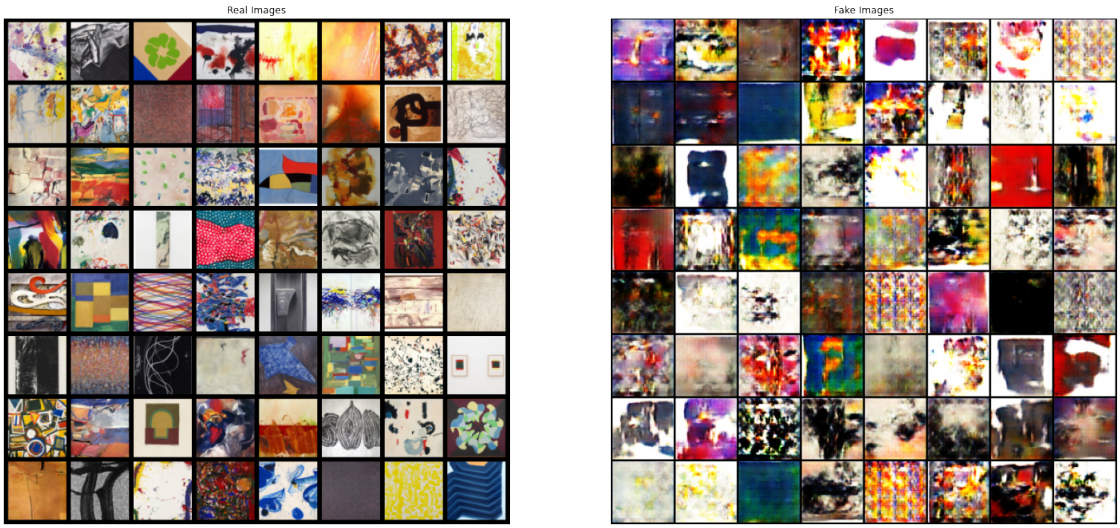

Figure 6: Examples of real images and images that are generated by DCGAN

2.3

There are two ways in which our generator may fail to improve, where the improvement is taken with respect to the quality of samples it produces. In the first case, though a solution may exist, the dynamics of the gradient descent training algorithm prevents the neural nets from reaching their optimal parameter values.

In the second case, which is a special case of the first failure, the gradient along which the generator must train is diminished to the point that the generator cannot usefully learn from it. This is known as the vanishing gradient problem, or the saturation problem. Goodfellow et al. (2014) claims that this problem is caused by the discriminator successfully rejecting generator samples with high confidence, so that the generator's gradient vanishes. This suggests that we ought to avoid over-training the discriminator, and instead carefully interplay discriminator and generator improvements.

Mode collapse is a problem that occurs when the generator learns to produce only a limited range of samples from the real data distribution. It does so by mapping several different input values $z \sim p_z$ to the same output point $G(z)$. The name 'mode collapse' comes from the fact that, when trying to learn a multimodal distribution, the generator only outputs samples from a select number of these modes.

For resolving these problems, we can use Wasserstein GAN (WGAN), which uses Wasserian distance as loss function of GAN network.

3 Cycle GAN

3.1

CycleGAN is a model that aims to solve the image-to-image translation problem. The goal of the image-to-image translation problem is to learn the mapping between an input image and an output image using a training set of aligned image pairs.

The CycleGAN is an extension of the GAN architecture that involves the simultaneous training of two generator models and two discriminator models.

One generator takes images from the first domain as input and outputs images for the second domain, and the other generator takes images from the second domain as input and generates images for the first domain. Discriminator models are then used to determine how plausible the generated images are and update the generator models accordingly.

This extension alone might be enough to generate plausible images in each domain, but not sufficient to generate translations of the input images, so The CycleGAN uses an additional extension to the architecture called cycle consistency. This is the idea that an image output by the first generator could be used as input to the second generator and the output of the second generator should match the original image. The reverse is also true: that an output from the second generator can be fed as input to the first generator and the result should match the input to the second generator.

The CycleGAN encourages cycle consistency by adding an additional loss to measure the difference between the generated output of the second generator and the original image, and the reverse. This acts as a regularization of the generator models, guiding the image generation process in the new domain toward image translation.

Adversarial training can learn mappings G and F that produce outputs identically distributed as target domains Y and X respectively. However, with large enough capacity, a network can map the same set of input images to any random permutation of images in the target domain, where any of the learned mappings can induce an output distribution that matches the target distribution. Thus, adversarial losses alone cannot guarantee that the learned function can map an individual input x_i to a desired output y_i. For each image x from domain X, the image translation cycle should be able to bring x back to the original image, i.e., $x \rightarrow G(x) \rightarrow F(G(x)) \approx x$. We call this forward cycle consistency. Similarly, for each image y from domain Y , G and F should also satisfy backward cycle consistency: $y \rightarrow F(y) \rightarrow G(F(y)) \approx y$.

This behavior incentivized using a cycle consistency loss.

3.2

For the discriminator networks, CycleGAN will use 70×70 PatchGANs which aim to classify whether 70×70 overlapping image patches are real or fake. Such a patch-level discriminator architecture has fewer parameters than a full-image discriminator and can work on arbitrarilysized images in a fully convolutional fashion.

3.3

Download dataset from Kaggle:

[1]:
```
! mkdir ~/.kaggle
```

```
mkdir: cannot create directory ‘/root/.kaggle’: File exists
```

[2]:
```
! cp kaggle.json ~/.kaggle/
```

[3]:
```
! chmod 600 ~/.kaggle/kaggle.json
```

[4]:
```
from kaggle.api.kaggle_api_extended import KaggleApi
api = KaggleApi()
api.authenticate()
```

[5]:
```
api.dataset_download_files('balraj98/summer2winter-yosemite')
```

Import essential libraries:

[6]:
```
! pip install tensorflow_addons
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-
wheels/public/simple/
Requirement already satisfied: tensorflow_addons in
/usr/local/lib/python3.7/dist-packages (0.17.1)
Requirement already satisfied: typeguard>=2.7 in /usr/local/lib/python3.7/dist-
packages (from tensorflow_addons) (2.7.1)
Requirement already satisfied: packaging in /usr/local/lib/python3.7/dist-
packages (from tensorflow_addons) (21.3)
Requirement already satisfied: pyparsing!=3.0.5,>=2.0.2 in
/usr/local/lib/python3.7/dist-packages (from packaging->tensorflow_addons)
(3.0.9)
```

[58]:
```
from zipfile import ZipFile
import glob
import tensorflow as tf
import time
import concurrent.futures
import numpy as np
import matplotlib.pyplot as plt
import tensorflow_addons as tfa
from keras.models import Model, Sequential
from keras import layers
```

Import dataset:

[8]:
```
zf = ZipFile('summer2winter-yosemite.zip')
zf.extractall()
zf.close()
```

Variables initilization:

[9]:
```
# Load the dataset
train_A_paths = glob.glob('/content/trainA/*.jpg')
train_B_paths = glob.glob('/content/trainB/*.jpg')
test_A_paths = glob.glob('/content/testA/*.jpg')
test_B_paths = glob.glob('/content/testB/*.jpg')

# Define the standard image size.
IMSIZE = 256

OUTPUT_CHANNELS = 3

loss_object = tf.keras.losses.BinaryCrossentropy(from_logits=True)
LAMBDA = 7

Epochs = 20
```

Preproccess images:

[10]:
```
def read_img(image):
    img = tf.keras.preprocessing.image.load_img(image, color_mode='rgb',␣
 ↪target_size=(IMSIZE, IMSIZE))
    return img
```

[11]:
```
def prepare_dataset(namelist):
    imgs = []
    with concurrent.futures.ThreadPoolExecutor(max_workers = 16) as executor:
        i = 0
        for value in executor.map(read_img, namelist):
            i+=1
            imgs.append(value)
        imgs = np.stack(imgs)
        imgs = tf.convert_to_tensor(imgs)
    return imgs
```

[12]:
```
with tf.device('/cpu:0'):
    train_a= prepare_dataset(train_A_paths)
    train_b= prepare_dataset(train_B_paths)
    test_a= prepare_dataset(test_A_paths)
    test_b= prepare_dataset(test_B_paths)
print("Training A tensor shape", train_a.shape)
print("Training B tensor shape", train_b.shape)
print("Testing A tensor shape", test_a.shape)
print("Testing B tensor shape", test_b.shape)
```

```
Training A tensor shape (1231, 256, 256, 3)
Training B tensor shape (962, 256, 256, 3)
Testing A tensor shape (309, 256, 256, 3)
Testing B tensor shape (238, 256, 256, 3)
```

[13]:
```
# Map values in the range [-1, 1]
def map(image):

    image = tf.cast(image, tf.float32)
    image = image / 255
    image = image * 2 - 1

    return image
```

[14]:
```
train_a = tf.data.Dataset.from_tensor_slices(train_a)
train_b = tf.data.Dataset.from_tensor_slices(train_b)
test_a = tf.data.Dataset.from_tensor_slices(test_a)
test_b = tf.data.Dataset.from_tensor_slices(test_b)
```

[15]:
```
AUTOTUNE = tf.data.AUTOTUNE
BUFFER_SIZE = 200

# Apply the preprocessing operations to data
train_a = train_a.map(map, num_parallel_calls=AUTOTUNE).shuffle(BUFFER_SIZE).
 ↪batch(4).prefetch(AUTOTUNE)
train_b = train_b.map(map, num_parallel_calls=AUTOTUNE).cache().
 ↪shuffle(BUFFER_SIZE).batch(4).prefetch(AUTOTUNE)
test_a = test_a.map(map, num_parallel_calls=AUTOTUNE).cache().
 ↪shuffle(BUFFER_SIZE).batch(4).prefetch(AUTOTUNE)
test_b = test_b.map(map, num_parallel_calls=AUTOTUNE).cache().
 ↪shuffle(BUFFER_SIZE).batch(4).prefetch(AUTOTUNE)
```

[16]:
```
train_dataset = tf.data.Dataset.zip((train_a, train_b))
test_dataset = tf.data.Dataset.zip((test_a, test_b))
```

Visualize some samples:

[17]:
```
plt.figure(figsize=(12, 6))
for imgs_A, imgs_B in test_dataset.take(1):
    plt.subplot(1,2,1)
    plt.imshow((imgs_A[0]+1)/2)
    plt.subplot(1,2,2)
    plt.imshow((imgs_B[0]+1)/2)
```

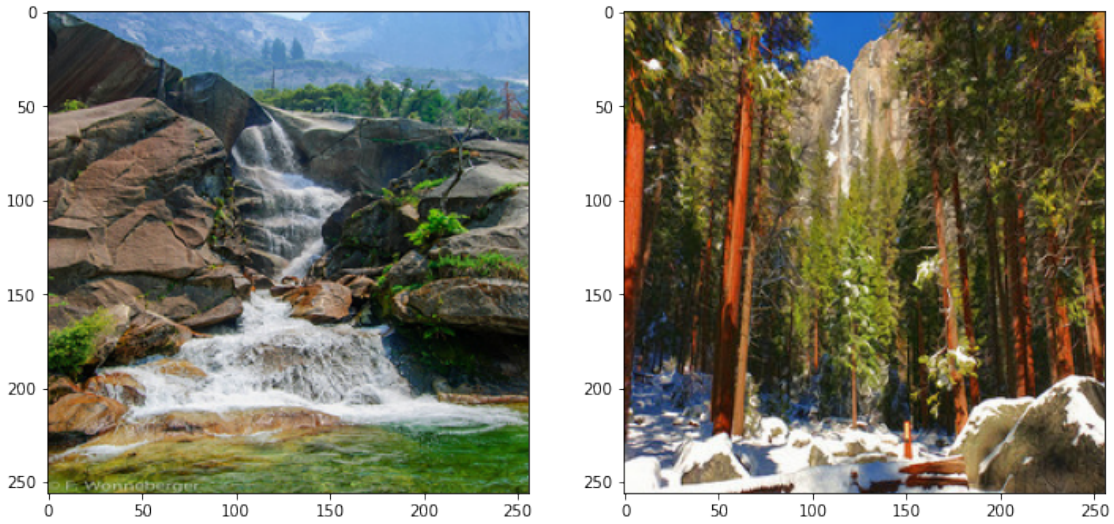

Figure 7: Sample images of summer (left) and winter (right) from dataset

Building blocks used in the CycleGAN generators and discriminators:

[18]:
```
def downsample(filters, size, apply_batchnorm=True):

    result = tf.keras.Sequential()
    result.add(tf.keras.layers.Conv2D(filters, size, strides=2, padding='same',␣
 ↪use_bias=False))

    if apply_batchnorm:
        result.add(tfa.layers.InstanceNormalization())

    result.add(tf.keras.layers.LeakyReLU())

    return result
```

[19]:
```
def upsample(filters, size, apply_dropout=False):

    result = tf.keras.Sequential()
    result.add(tf.keras.layers.Conv2DTranspose(filters, size, strides=2,␣
 ↪padding='same', use_bias=False))

    result.add(tfa.layers.InstanceNormalization())

    if apply_dropout:
        result.add(tf.keras.layers.Dropout(0.5))
```

```
    result.add(tf.keras.layers.ReLU())

    return result
```

Build the generators:

The generator consists of downsampling blocks: nine residual blocks and upsampling blocks.

[20]:
```
def Generator():
    inputs = tf.keras.layers.Input(shape=[256,256,3])

    down_stack = [
        downsample(64, 4, apply_batchnorm=False), # (bs, 128, 128, 64)
        downsample(128, 4), # (bs, 64, 64, 128)
        downsample(256, 4), # (bs, 32, 32, 256)
        downsample(512, 4), # (bs, 16, 16, 512)
        downsample(512, 4), # (bs, 8, 8, 512)
        downsample(512, 4), # (bs, 4, 4, 512)
        downsample(512, 4), # (bs, 2, 2, 512)
        downsample(512, 4), # (bs, 1, 1, 512)
    ]

    up_stack = [
        upsample(512, 4, apply_dropout=True), # (bs, 2, 2, 1024)
        upsample(512, 4, apply_dropout=True), # (bs, 4, 4, 1024)
        upsample(512, 4, apply_dropout=True), # (bs, 8, 8, 1024)
        upsample(512, 4), # (bs, 16, 16, 1024)
        upsample(256, 4), # (bs, 32, 32, 512)
        upsample(128, 4), # (bs, 64, 64, 256)
        upsample(64, 4), # (bs, 128, 128, 128)
    ]

    last = tf.keras.layers.Conv2DTranspose(OUTPUT_CHANNELS, 4, strides=2,␣
↪padding='same', activation='tanh') # (bs, 256, 256, 3)

    x = inputs

    # Downsampling through the model
    skips = []
    for down in down_stack:
        x = down(x)
        skips.append(x)

    skips = reversed(skips[:-1])

    # Upsampling and establishing the skip connections
    for up, skip in zip(up_stack, skips):
```

```
        x = up(x)
        x = tf.keras.layers.Concatenate()([x, skip])

    x = last(x)

    return tf.keras.Model(inputs=inputs, outputs=x)
```

[21]:
```
generator_x = Generator()   # a-->o
generator_y = Generator()   # o-->a
tf.keras.utils.plot_model(generator_x, show_shapes=True, dpi=48)
```

[21]:

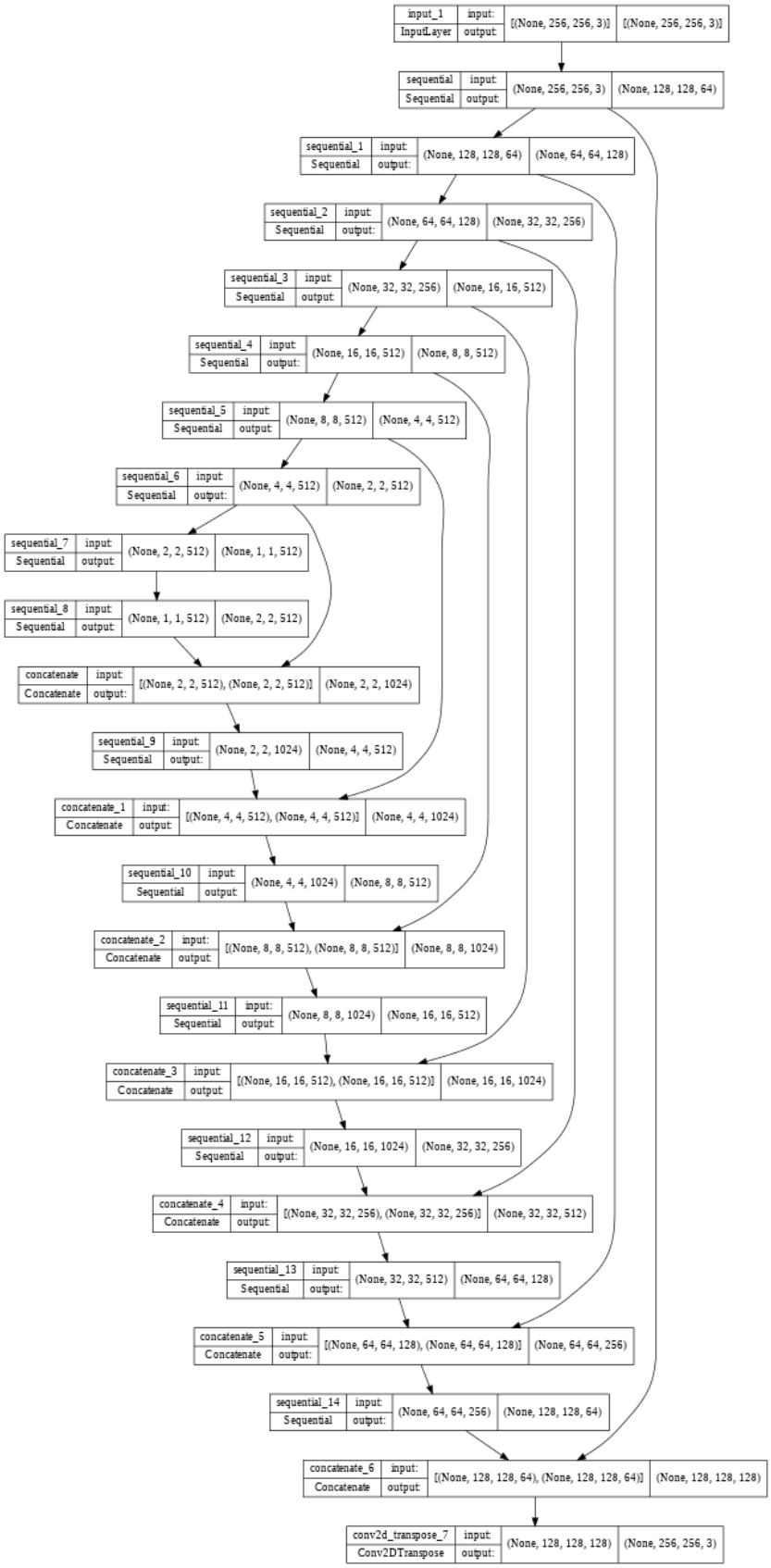

Figure 8: Architecture of generator

Build the discriminators:

The discriminators implement the following architecture: C64->C128->C256->C512

[22]:
```
def Discriminator():

    inp = tf.keras.layers.Input(shape=[256, 256, 3], name='input_image')

    down1 = downsample(64, 4, False)(inp) # (bs, 128, 128, 64)
    down2 = downsample(128, 4)(down1) # (bs, 64, 64, 128)
    down3 = downsample(256, 4)(down2) # (bs, 32, 32, 256)

    zero_pad1 = tf.keras.layers.ZeroPadding2D()(down3)  # (bs, 34, 34, 256)
    conv = tf.keras.layers.Conv2D(512, 4, strides=1,use_bias=False)(zero_pad1) ␣
↪# (bs, 31, 31, 512)

    norm1 = tfa.layers.InstanceNormalization()(conv)

    leaky_relu = tf.keras.layers.LeakyReLU()(norm1)

    zero_pad2 = tf.keras.layers.ZeroPadding2D()(leaky_relu)  # (bs, 33, 33, 512)

    last = tf.keras.layers.Conv2D(1, 4, strides=1)(zero_pad2)  # (bs, 30, 30, 1)

    return tf.keras.Model(inputs=inp, outputs=last)
```

[23]:
```
discriminator_x = Discriminator()   # discriminator  a
discriminator_y = Discriminator()   # discriminator  o
tf.keras.utils.plot_model(discriminator_x, show_shapes=True, dpi=64)
```

[23]:

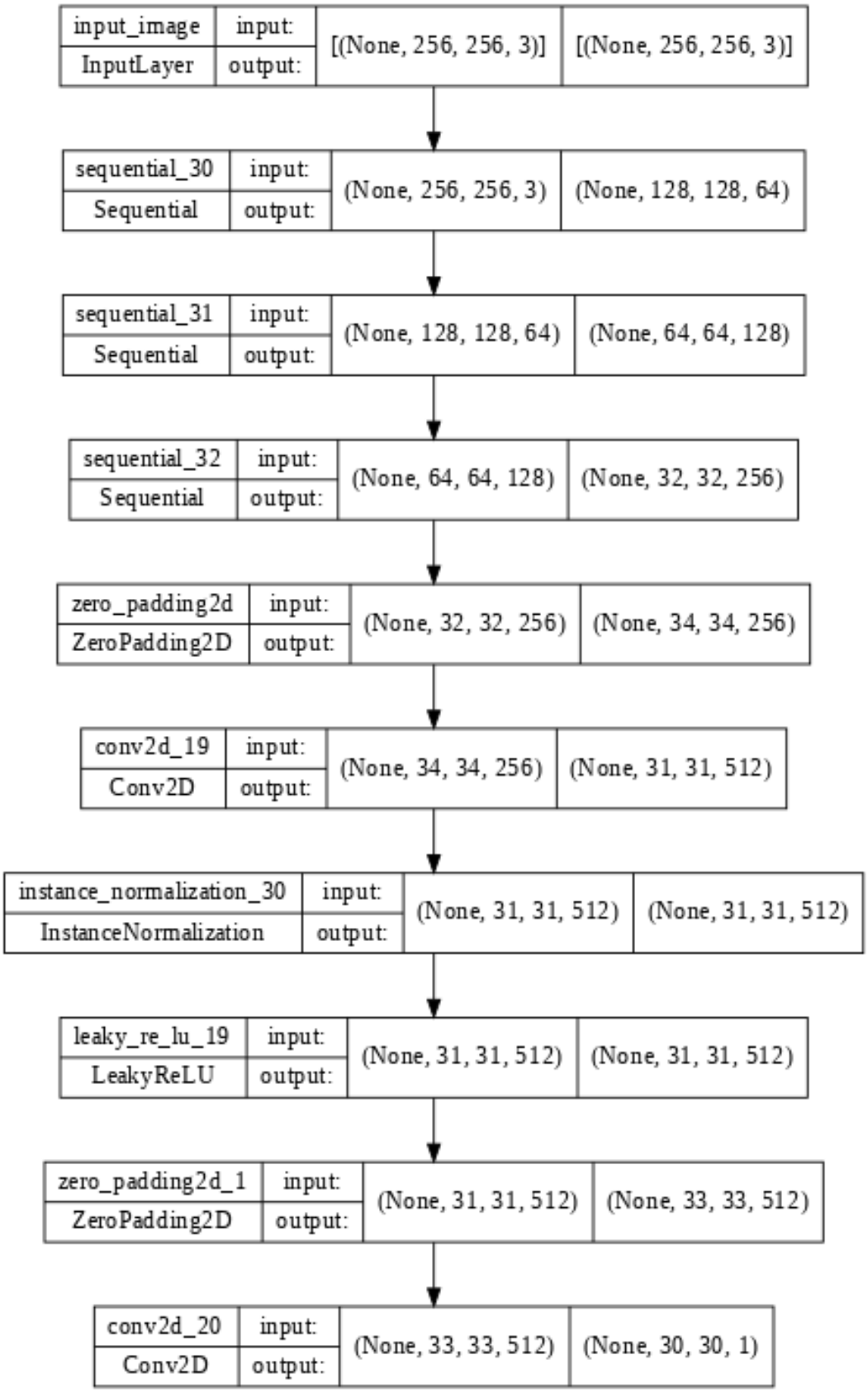

Figure 9: Architecture of discriminator

Visualizing model input and output:

[24]:
```
for example_input, example_target in test_dataset.take(1):
    pass
```

[25]:
```
def image_show():

    title=['Source domain','Map']
    rendered_pictures = generator_x(example_input, training=False)

    fig = plt.figure(figsize=(12, 24))

    plt.subplot(4,2,1)
    plt.imshow((example_input[0].numpy()+1)/2)
    plt.title(title[0])

    plt.subplot(4,2,2)
    plt.imshow((rendered_pictures[0].numpy()+1)/2)
    plt.title(title[1])

    plt.subplot(4,2,3)
    plt.imshow((example_input[1].numpy()+1)/2)
    plt.title(title[0])

    plt.subplot(4,2,4)
    plt.imshow((rendered_pictures[1].numpy()+1)/2)
    plt.title(title[1])

    plt.subplot(4,2,5)
    plt.imshow((example_input[2].numpy()+1)/2)
    plt.title(title[0])

    plt.subplot(4,2,6)
    plt.imshow((rendered_pictures[2].numpy()+1)/2)
    plt.title(title[1])

    plt.subplot(4,2,7)
    plt.imshow((example_input[3].numpy()+1)/2)
    plt.title(title[0])

    plt.subplot(4,2,8)
    plt.imshow((rendered_pictures[3].numpy()+1)/2)
    plt.title(title[1])
```

```
    plt.show()

    return rendered_pictures
```

[26]:
```
pred = image_show()
```

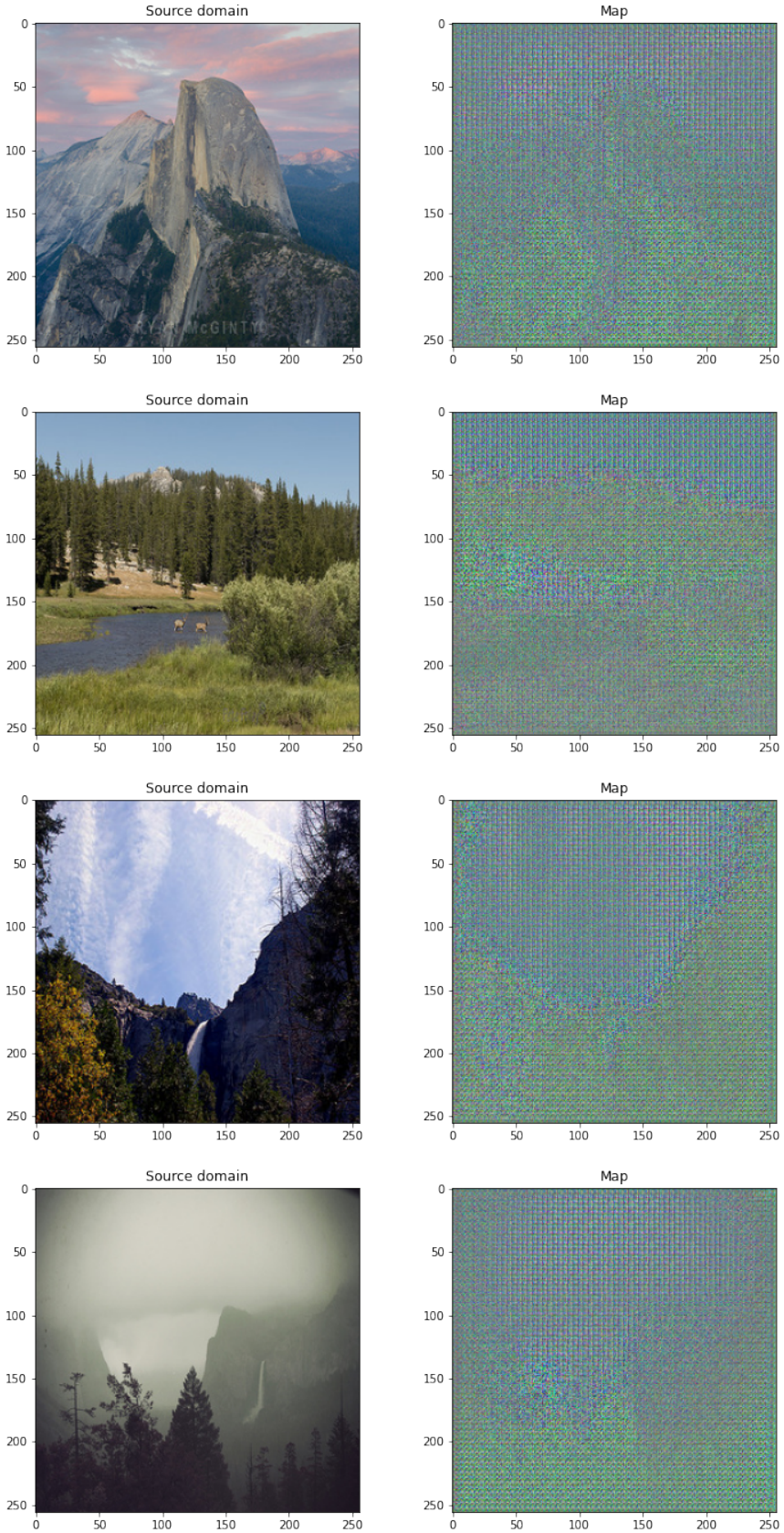

Figure 10: Sample images of summer and mapped pictures of them before training

Loss Calculation:

We need to calculate different kinds of losses for the generators and discriminators.

[27]:
```
def discriminator_loss(disc_real_output, disc_fake_output):
    real_loss = loss_object(tf.ones_like(disc_real_output), disc_real_output)
    fake_loss = loss_object(tf.zeros_like(disc_fake_output), disc_fake_output)
    total_disc_loss = real_loss + fake_loss
    return total_disc_loss
```

[28]:
```
def generator_loss(disc_fake_output):
    gen_loss = loss_object(tf.ones_like(disc_fake_output), disc_fake_output)
    return gen_loss
```

[29]:
```
def calc_cycle_loss(real_image, cycled_image):
    loss = tf.reduce_mean(tf.abs(real_image - cycled_image))
    return LAMBDA * loss
```

Set the optimizers:

[30]:
```
generator_x_optimizer = tf.keras.optimizers.Adam(2e-4, beta_1=0.5)
generator_y_optimizer = tf.keras.optimizers.Adam(2e-4, beta_1=0.5)
discriminator_x_optimizer = tf.keras.optimizers.Adam(2e-4, beta_1=0.5)
discriminator_y_optimizer = tf.keras.optimizers.Adam(2e-4, beta_1=0.5)
```

Train the end-to-end model:

[31]:
```
@tf.function
def train_discriminator(image_a, image_b):
    with tf.GradientTape(persistent=True) as discriminator_tape:
        #A->B->A
        fake_b = generator_x(image_a, training=True)
        cycled_a = generator_y(fake_b, training=True)
        #B->A->B
        fake_a = generator_y(image_b, training=True)
        cycled_b = generator_x(fake_a, training=True)
        #discriminator
        disc_real_a = discriminator_x(image_a, training=True)
        disc_real_b = discriminator_y(image_b, training=True)
        disc_fake_a = discriminator_x(fake_a, training=True)
        disc_fake_b = discriminator_y(fake_b, training=True)

        # Define the loss function for the discriminators
        discriminator_x_loss = discriminator_loss(disc_real_a, disc_fake_a)
        discriminator_y_loss = discriminator_loss(disc_real_b, disc_fake_b)
```

```
    discriminator_x_gradients = discriminator_tape.
↪gradient(discriminator_x_loss, discriminator_x.trainable_variables)
    discriminator_y_gradients = discriminator_tape.
↪gradient(discriminator_y_loss, discriminator_y.trainable_variables)

    discriminator_x_optimizer.
↪apply_gradients(zip(discriminator_x_gradients,discriminator_x.
↪trainable_variables))
    discriminator_y_optimizer.
↪apply_gradients(zip(discriminator_y_gradients,discriminator_y.
↪trainable_variables))
```

[32]:
```
@tf.function
def train_generator(image_a, image_b):
    with tf.GradientTape(persistent=True) as generator_tape:
        #A->B->A
        fake_b = generator_x(image_a, training=True)
        cycled_a = generator_y(fake_b, training=True)
        #B->A->B
        fake_a = generator_y(image_b, training=True)
        cycled_b = generator_x(fake_a, training=True)
        #discriminator
        disc_fake_a = discriminator_x(fake_a, training=True)
        disc_fake_b = discriminator_y(fake_b, training=True)

        # Define the loss function for the generators
        gen_x_loss = generator_loss(disc_fake_b)
        gen_y_loss = generator_loss(disc_fake_a)

        total_cycle_loss = calc_cycle_loss(image_a, cycled_a) +␣
↪calc_cycle_loss(image_b, cycled_b)

        total_gen_x_loss = gen_x_loss + total_cycle_loss
        total_gen_y_loss = gen_y_loss + total_cycle_loss

    generator_x_gradients = generator_tape.gradient(total_gen_x_loss,␣
↪generator_x.trainable_variables)
    generator_y_gradients = generator_tape.gradient(total_gen_y_loss,␣
↪generator_y.trainable_variables)

    generator_x_optimizer.apply_gradients(zip(generator_x_gradients, generator_x.
↪trainable_variables))
    generator_y_optimizer.apply_gradients(zip(generator_y_gradients, generator_y.
↪trainable_variables))
```

Compile the model:

[33]:
```
for epoch in range(Epochs):
    start = time.time()
    i = 0
    print ('\nEpoch {}/{} '.format(epoch+1, Epochs))
    for img_a, img_b in train_dataset:

        train_discriminator(img_a, img_b)
        train_generator(img_a, img_b)

        percent = float(i+1) * 100 / len(train_dataset)
        arrow   = '-' * int(percent/100 * 10 - 1) + '>'
        spaces  = ' ' * (10 - len(arrow))
        print('\rTraining: [%s%s] %d %% '% (arrow, spaces, percent), end='',␣
 ↪flush=True)
        i += 1
    print(" -", int(time.time()-start), "s", end="")
    print()

cache = image_show()
```

```
Epoch 1/20
Training: [---------->] 100 %  - 209 s

...

Epoch 20/20
Training: [---------->] 100 %  - 161 s
```

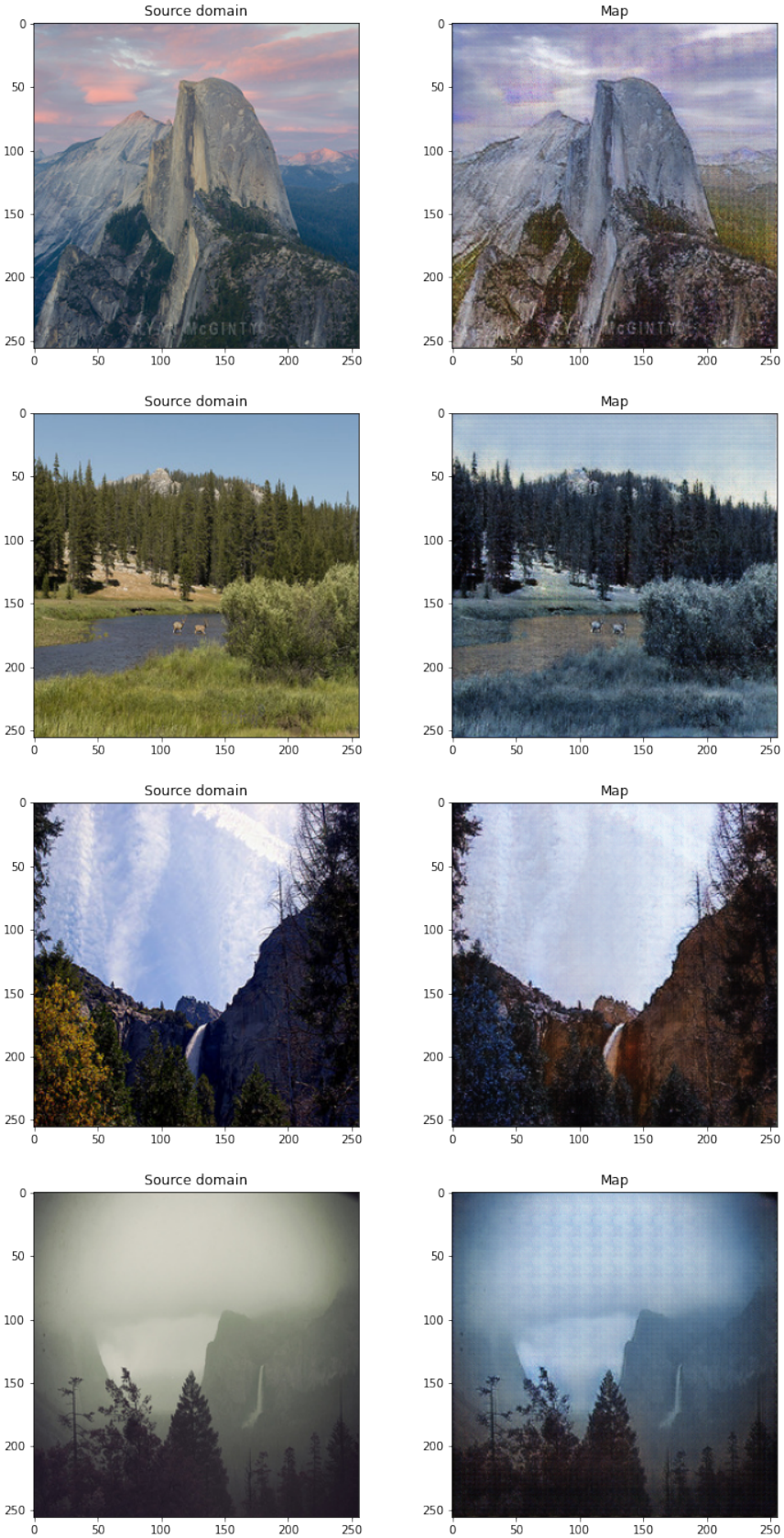

Figure 11: Sample images of summer and mapped pictures of them after training

3.4

Create U_net:

[60]:
```
def unet_generator():
    inputs = layers.Input(shape=(256, 256, 3))

    ### [First half of the network: downsampling inputs] ###

    # Entry block
    x = layers.Conv2D(32, 3, strides=2, padding="same")(inputs)
    x = layers.BatchNormalization()(x)
    x = layers.Activation("relu")(x)

    previous_block_activation = x  # Set aside residual

    # Blocks 1, 2, 3 are identical apart from the feature depth.
    for filters in [64, 128, 256]:
        x = layers.Activation("relu")(x)
        x = layers.SeparableConv2D(filters, 3, padding="same")(x)
        x = layers.BatchNormalization()(x)

        x = layers.Activation("relu")(x)
        x = layers.SeparableConv2D(filters, 3, padding="same")(x)
        x = layers.BatchNormalization()(x)

        x = layers.MaxPooling2D(3, strides=2, padding="same")(x)

        # Project residual
        residual = layers.Conv2D(filters, 1, strides=2, padding="same")(
            previous_block_activation
        )
        x = layers.add([x, residual])  # Add back residual
        previous_block_activation = x  # Set aside next residual

    ### [Second half of the network: upsampling inputs] ###

    for filters in [256, 128, 64, 32]:
        x = layers.Activation("relu")(x)
        x = layers.Conv2DTranspose(filters, 3, padding="same")(x)
        x = layers.BatchNormalization()(x)

        x = layers.Activation("relu")(x)
```

```
        x = layers.Conv2DTranspose(filters, 3, padding="same")(x)
        x = layers.BatchNormalization()(x)

        x = layers.UpSampling2D(2)(x)

        # Project residual
        residual = layers.UpSampling2D(2)(previous_block_activation)
        residual = layers.Conv2D(filters, 1, padding="same")(residual)
        x = layers.add([x, residual])  # Add back residual
        previous_block_activation = x  # Set aside next residual

    # Add a per-pixel classification layer
    outputs = layers.Conv2D(3, 3, activation="softmax", padding="same")(x)

    # Define the model
    model = Model(inputs, outputs)
    return model
```

[61]:
```
generator_x_unet = unet_generator()   # a-->o
generator_y_unet = unet_generator()   # o-->a
tf.keras.utils.plot_model(generator_x_unet, show_shapes=True, dpi=48)
```

[61]:

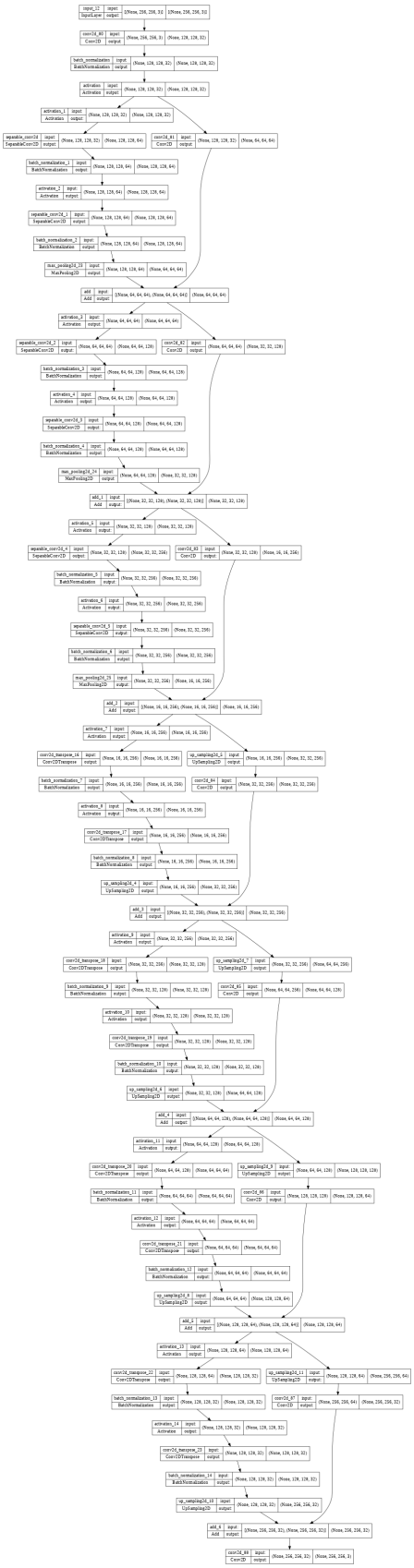

Figure 12: Architecture of U-Net encoder decoder

Train new model:

[62]:
```
@tf.function
def train_discriminator(image_a, image_b):
    with tf.GradientTape(persistent=True) as discriminator_tape:
        #A->B->A
        fake_b = generator_x_unet(image_a, training=True)
        cycled_a = generator_y_unet(fake_b, training=True)
        #B->A->B
        fake_a = generator_y_unet(image_b, training=True)
        cycled_b = generator_x_unet(fake_a, training=True)
        #discriminator
        disc_real_a = discriminator_x(image_a, training=True)
        disc_real_b = discriminator_y(image_b, training=True)
        disc_fake_a = discriminator_x(fake_a, training=True)
        disc_fake_b = discriminator_y(fake_b, training=True)

        # Define the loss function for the discriminators
        discriminator_x_loss = discriminator_loss(disc_real_a, disc_fake_a)
        discriminator_y_loss = discriminator_loss(disc_real_b, disc_fake_b)

    discriminator_x_gradients = discriminator_tape.
↪gradient(discriminator_x_loss, discriminator_x.trainable_variables)
    discriminator_y_gradients = discriminator_tape.
↪gradient(discriminator_y_loss, discriminator_y.trainable_variables)

    discriminator_x_optimizer.
↪apply_gradients(zip(discriminator_x_gradients,discriminator_x.
↪trainable_variables))
    discriminator_y_optimizer.
↪apply_gradients(zip(discriminator_y_gradients,discriminator_y.
↪trainable_variables))
```

[63]:
```
@tf.function
def train_generator(image_a, image_b):
    with tf.GradientTape(persistent=True) as generator_tape:
        #A->B->A
        fake_b = generator_x_unet(image_a, training=True)
        cycled_a = generator_y_unet(fake_b, training=True)
        #B->A->B
        fake_a = generator_y_unet(image_b, training=True)
        cycled_b = generator_x_unet(fake_a, training=True)
        #discriminator
```

```
        disc_fake_a = discriminator_x(fake_a, training=True)
        disc_fake_b = discriminator_y(fake_b, training=True)

        # Define the loss function for the generators
        gen_x_loss = generator_loss(disc_fake_b)
        gen_y_loss = generator_loss(disc_fake_a)

        total_cycle_loss = calc_cycle_loss(image_a, cycled_a) +␣
↪calc_cycle_loss(image_b, cycled_b)

        total_gen_x_loss = gen_x_loss + total_cycle_loss
        total_gen_y_loss = gen_y_loss + total_cycle_loss

    generator_x_gradients = generator_tape.gradient(total_gen_x_loss,␣
↪generator_x_unet.trainable_variables)
    generator_y_gradients = generator_tape.gradient(total_gen_y_loss,␣
↪generator_y_unet.trainable_variables)

    generator_x_optimizer.apply_gradients(zip(generator_x_gradients,␣
↪generator_x_unet.trainable_variables))
    generator_y_optimizer.apply_gradients(zip(generator_y_gradients,␣
↪generator_y_unet.trainable_variables))
```

Compile new model:

[64]:
```
for epoch in range(Epochs):
    start = time.time()
    i = 0
    print ('\nEpoch {}/{} '.format(epoch+1, Epochs))
    for img_a, img_b in train_dataset:

        train_discriminator(img_a, img_b)
        train_generator(img_a, img_b)

        percent = float(i+1) * 100 / len(train_dataset)
        arrow   = '-' * int(percent/100 * 10 - 1) + '>'
        spaces  = ' ' * (10 - len(arrow))
        print('\rTraining: [%s%s] %d %% '% (arrow, spaces, percent), end='',␣
↪flush=True)
        i += 1
    print(" -", int(time.time()-start), "s", end="")
    print()

cache = image_show()
```

```
Epoch 1/20
```

```
Training: [---------->] 100 %  - 145 s

...

Epoch 20/20
Training: [---------->] 100 %  - 118 s
```

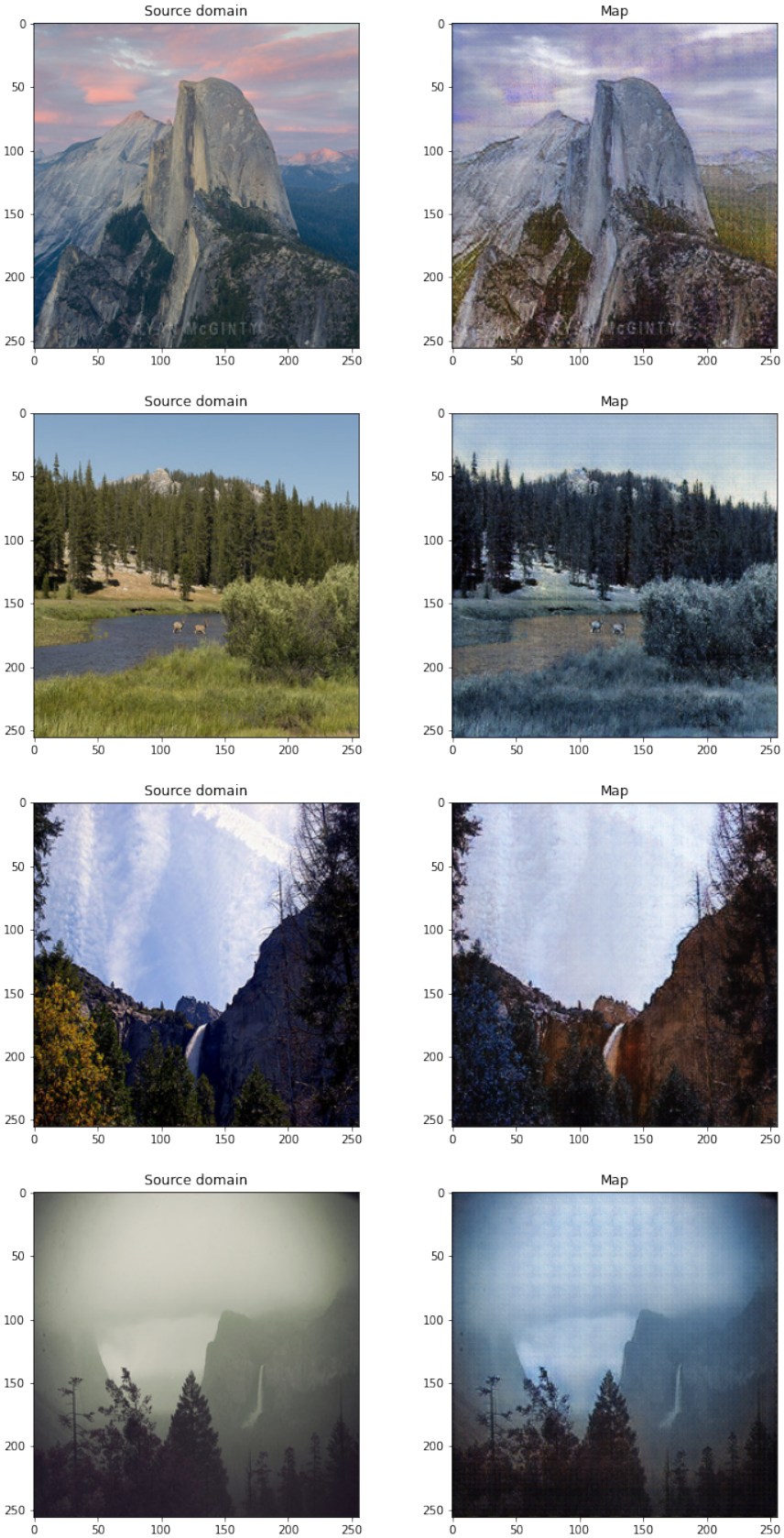

Figure 13: Sample images of summer and mapped pictures of them after training with CycleGan with U-Net generator