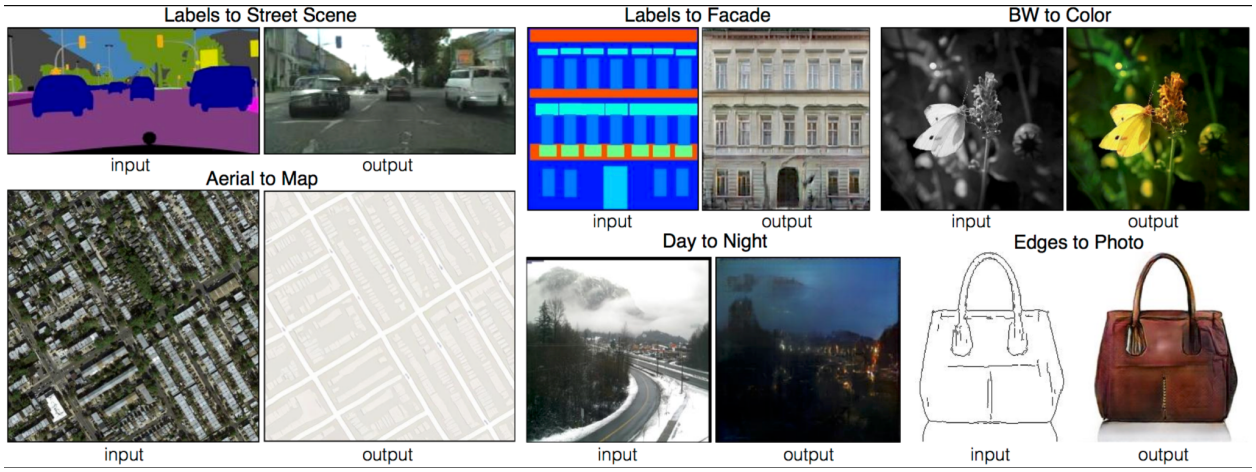
IE 590 Project: Using Pix2Pix for generating street-view images from satellite images

Made by Khush Garg

Part 1: Introduction

A concept may be expressed in either English or Spanish. Similarily, a scene may be rendered as an RGB image, a gradient field, an edge map, a semantic label map, etc. In analogy to automatic language translation, we define **Image to Image Translation** as the following:

It is a framework of conditional generation that transforms images into different styles i.e taking an image and transforming it to get a different image of a different style but maintaining the contents of the original image. For example, converting a black and white old image of a woman into a colorized version, converting a segmented map to a realistic image, converting from low resolution to high resolution.



Paired Image-to-Image Translation:

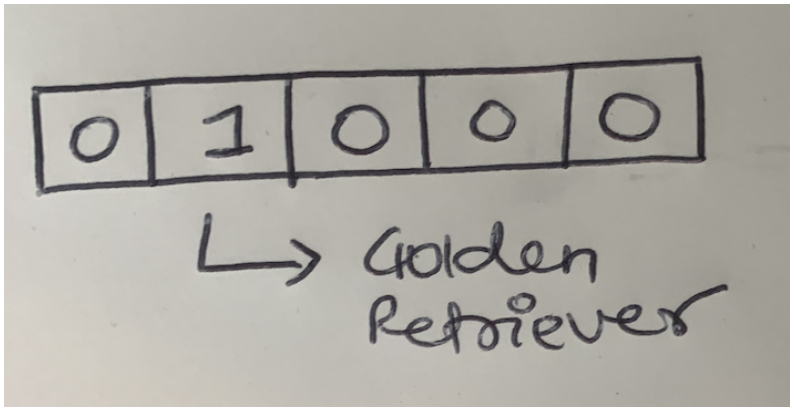
In your training dataset, for every single input example, you have a corresponding output image that contains the contents of the input image with a different style. You have input-output pairs that map exactly onto each other as the training dataset.
For example, if the goal is to create street map view images from satellite image, your training dataset would have pairs of satellite-streetmap images.
Pix2Pix is a type of conditional GAN which will be used to perform Paired Image-to-Image Translation.

Conditional Generative Adversial Networks:

Conditional Generative Adversial networks, also known as cGANs, are used as a solution to solve image-to-image translation problems. These networks are an extension of the GANs model. These networks inherently learn the **loss function. Loss Function** is described as the distance between the algorithm's current output and its expected output. This measurement is used as a feedback signal to adjust the way the algorithm works.
For example, Log Loss is used as a loss function for classification problems. MSE and MAE are used as loss functions for regression problems. However, when you are predicting pixels from pixels in image translation problems, you require the loss function to make the output image indistinguishable from reality. cGANs learns a loss that tries to classify if the output image is real or fake through a discriminator while it simultaneously trains the generative model to minimize this loss. A major advantage of using cGANs is that they learn a loss that adapts to the data and hence, it can be used for multiple tasks including reconstructing objects from edge maps, colorizing images etc. Convolutional Neural Networks are also used for tackling image translation problems but a lot of manual effort is require to design effective loss functions.

Conditional vs Unconditional Generation:
Understanding the difference between conditional and unconditional generation is crucial as it involves having different inputs for the model to be trained on. In the case of conditional generation, you need to have a labelled training dataset to train your cGANs and learn to generate examples from the desired classes. However, in the case of unconditional generation, the training dataset need not be labelled to generate examples from random classes.

In a conditional GAN, **the input to the generator involves a concatenated vector of both the noise vector and the one-hot class information.** For conditional generation, you need a vector that tells the generator from which class the generated examples should come from. This vector is called **one-hot vector.** This one-hot vector has a series of 0's and 1's where the position of 1 corresponds to the desired class. For example: This one-hot vector has different dog breeds and the second cell here corresponds to a golden retriever.

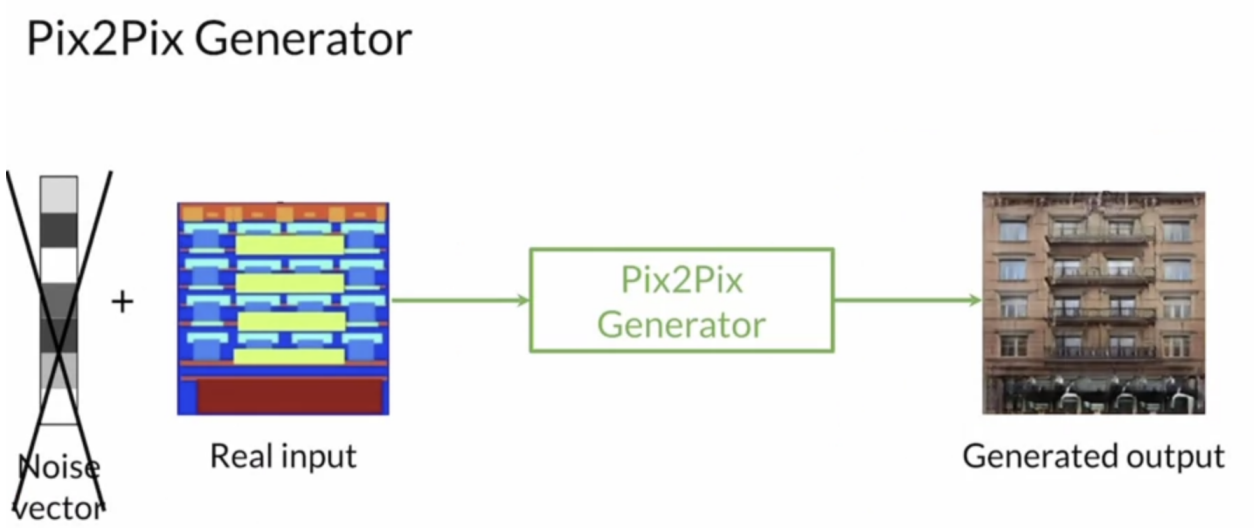


This one-hot vector would allow the generator to specifically produce images of golden retrievers. Concatenating this vector with the random noise vector would let you generate a diverse set of examples within the certain class or conditioned on that certain class.
For the discriminator, the class information is appended as one-hot matrices and the discriminator classifies based on if the images look real from that specific class.

Part 2: Pix2Pix

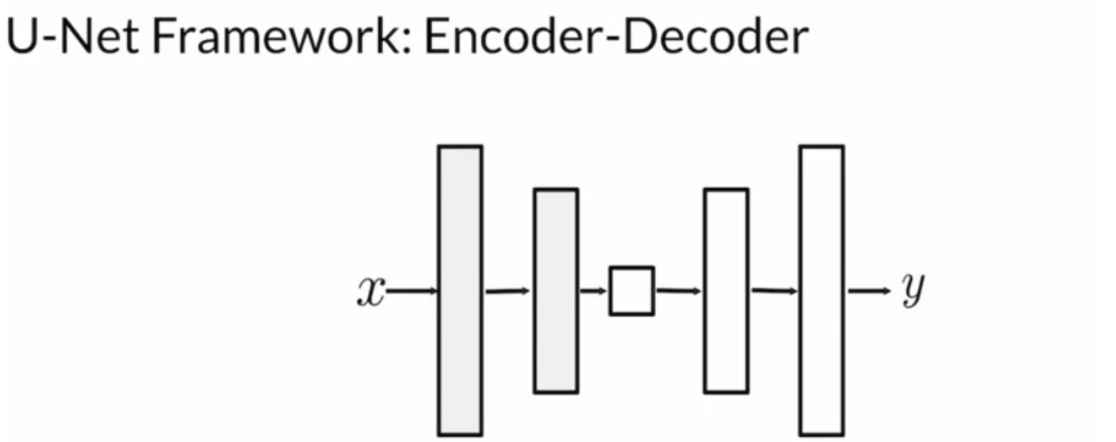
Pix2Pix is a unique conditional GAN which is used for image to image translation problems due to its upgraded generator and discriminator architectures.

A) Architecture of Pix2Pix Generator:

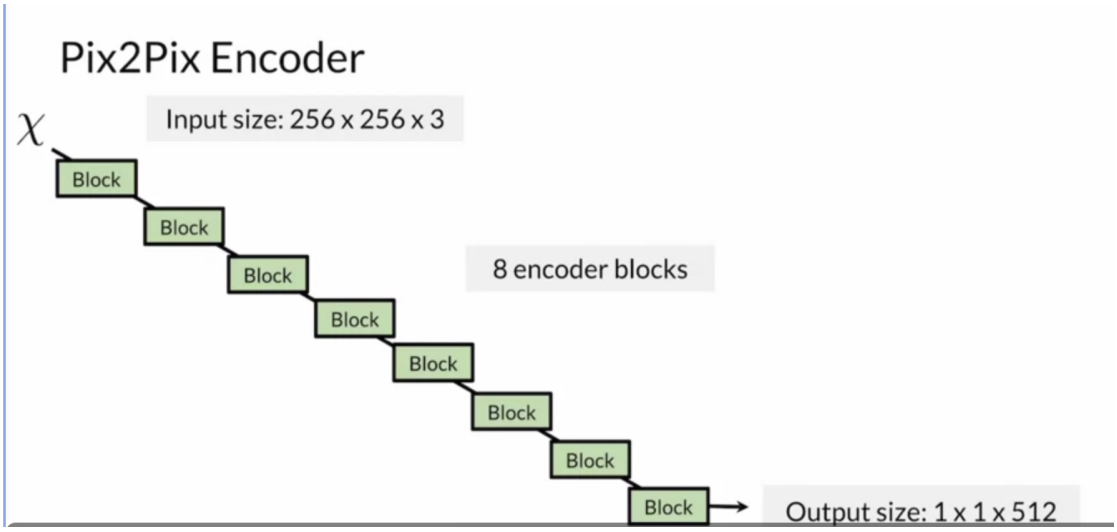


Instead of passing the one-hot class vector as the input to the generator, we use a real image as an input. For example, here we use the segmented image of a building as the input to the model. The noise vector is crossed out here as an input as the authors felt that the noise vector didn't make much of a difference in the context of the generated output.

It has a U-Net architecture framework for the generator architecture. U-Net has been a very successful computer vision model for image segmentation. It's an encoder-decoder model which uses skip connections in between.

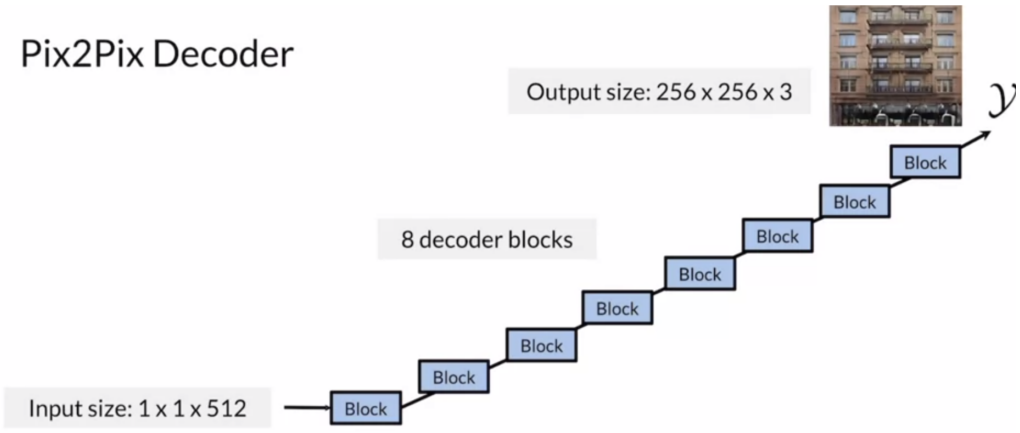


Instead of taking a random noise vector along with the class vector, the generator here takes an image as an input, represented by x in the figure above. First, all of the information of the input image is encoded and embedded in the bottleneck layer before it is decoded to give an output image, represented by y in the figure above. To avoid overfitting, U-Net introduces skip connections from the encoder to the decoder. This is extremely useful as it allows to retrieve information that might have been lost during the encoding stage. Having skip connections allow for information flow to the decoder as well as improves gradient flow to the encoder.



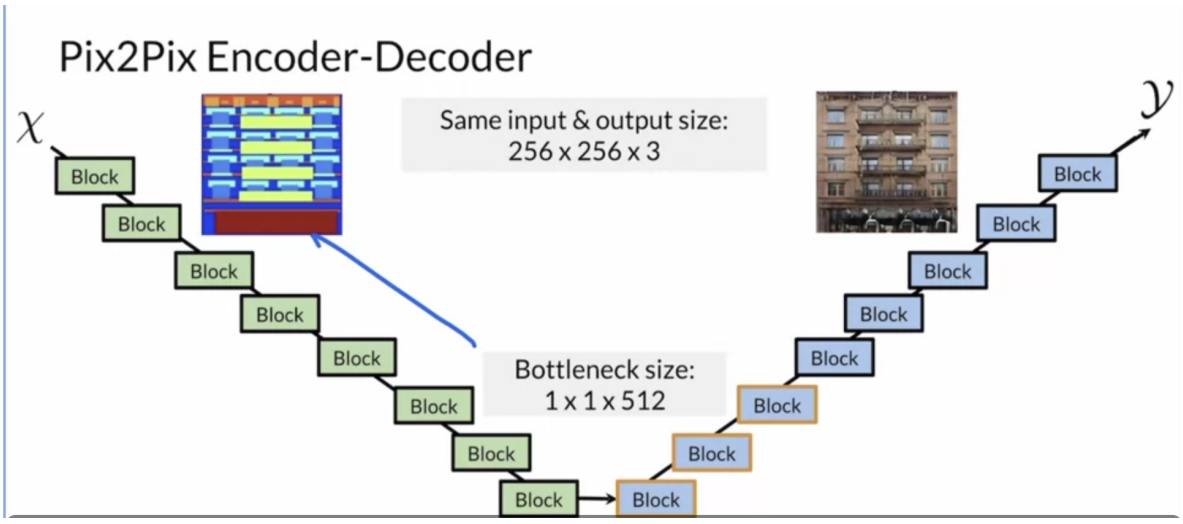
For example, you take an input image of size 256*256 which goes through eight encoder blocks to finally have a size of 1*1. Each block downsamples the spatial size by a factor of 2. Each of these blocks contain a convolution layer, BatchNorm layer, and a leakyReLU activation layer.

The encoder structure is: C64 - C128 - C256 - C512 - C512- C512 - C512- C512



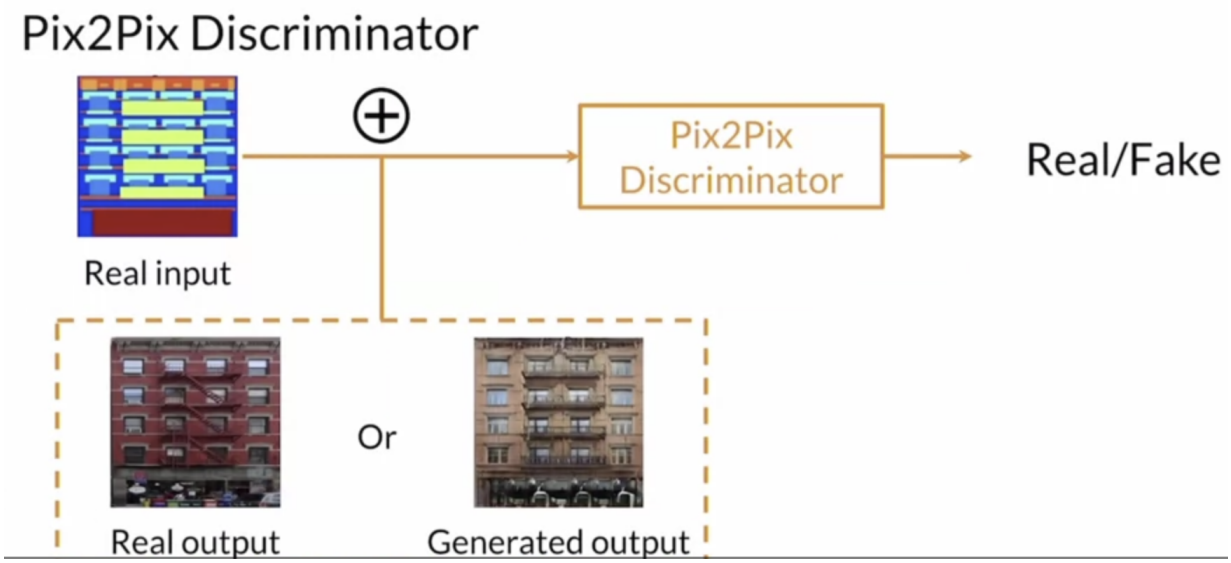
On the decoder side, now you have the input image with the size of 1*1 and 8 decoder blocks to generate the output image of the same size as the input image to the generator. Each decoder block is composed of a transposed convolution followed by BatchNorm and a ReLU activation function.

The Decoder structure is: CD512- CD512- CD512 - CD512 - C512 - C256 - C128 - C64



The figure above shows the entire encoder-decoder model for Pix2Pix Generator.

B) Architecture of Pix2Pix Discriminator:



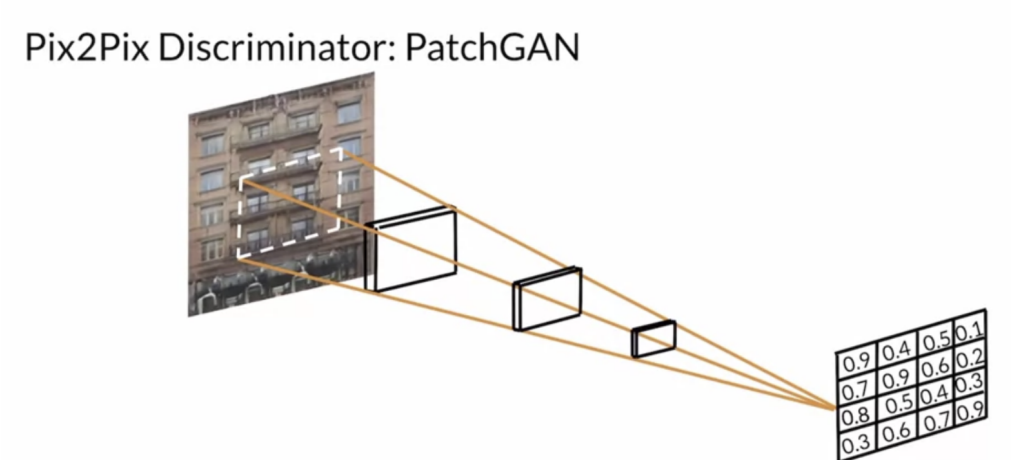
Here, the **real output** image is an image of the actual building that forms the segmentation map(**Real Input**). **In simpler words, the real output is the ground truth image of the real input.** The **generated output** image has been generated by the Pix2Pix generator based on what the generator was conditioned on. The **real input** image is concatenated with both of these images: **Real Output and Generated Output** and fed to the **Pix2Pix Discriminator.** The discriminator tries to classify if each N * N patch in an image is real or fake as opposed to classifying an entire image.

The Pix2Pix discriminator has a PatchGAN architecture. A 70*70 PatchGAN forces output images to be more sharp according to the FCN score metric. It is carefully designed so that each output prediction of the model maps to a 70×70 square or patch of the input image.

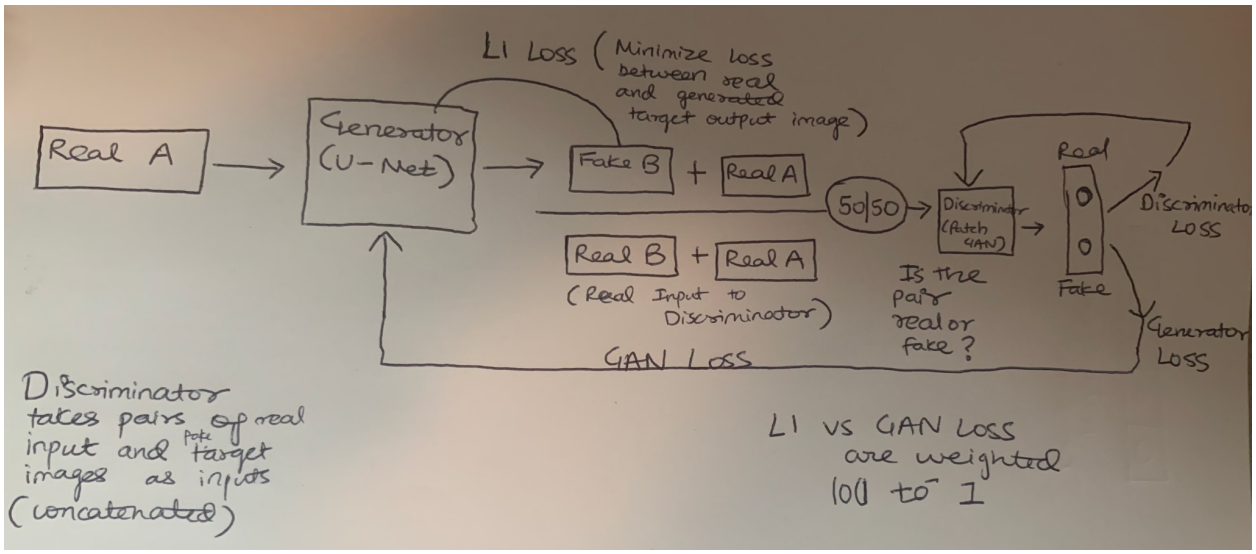
70*70 PatchGAN Architecture: C64 - C128 - C256- C512

All ReLUs are leaky with slope = 0.2

The PatchGAN architecture involves outputting a matrix of classification values as opposed to a single value, where each value is between 0 and 1(0 indicating fake classification and 1 indicating real classification).



PatchGAN gives a feedback on each patch of the image by outputting the probability of each patch being fake or real. A fake image would have a matrix filled with all zeros. Similarly, a real image would have a matrix filled with all ones.



Summary: The figure above shows the entire architecture of Pix2Pix along with the loss functions

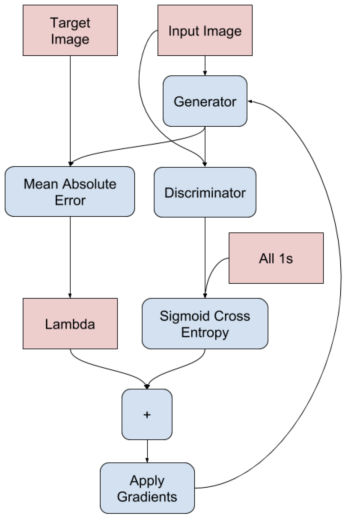
C) Loss Functions:

Pix2Pix Generator Loss

$$\text{BCE Loss} + \lambda \sum_{i=1}^{n} \left| \textit{generated\_output} - \textit{real\_output} \right|$$

Pixel Distance Loss(L1 Loss) calculates the difference between the fake and real target outputs. **Pix2Pix Generator Loss** uses this Pixel Distance Loss in addition to the **BCE Loss.**

L1 and GAN loss are weighted 100 to 1. Using only L1 Loss gives reasonable but blurry results. However, using only GAN loss gives superior results with few errors. Hence, using a combination of both losses is preferred for best results.



The figure above summarises the manner in which the loss functions are used. Sigmoid Cross Entropy(GANLoss) and Mean Absolute Error(L1 Loss) are weighted 100 to 1. Lambda is equal to 100.

Part 3: Problem Statement

Given the Google Satellite Maps, we aim to construct accurate street-view maps using the Pix2Pix software described above.
The dataset was retrieved from the following website:
`http://efrosgans.eecs.berkeley.edu/pix2pix/datasets/maps.tar.gz`

The above dataset has pairs of satellite-streetview images. There are 1096 pairs in the dataset.

A) **Method:**

The define_discriminator() function below implements the 70×70 PatchGAN discriminator model as per the design of the model in the paper. For the discriminator model, you define **in_src_image** as the variable that stores the source image(Real Satellite images) and **in_target_image** as the variable that stores the target image(Street View). Both are concatenated and follow the **C64- C128- C256- C512 architecture.** Batch Normalization is not a part of the first convolution layer. For rest of the layers, it exists. **We use a batch size of 1 and a really small learning rate of 0.0002 and beta = 0.5.** The model is optimized using binary cross entropy, and a weighting is used so that updates to the model have half (0.5) the usual effect. The authors of Pix2Pix recommend this weighting of model updates to slow down changes to the discriminator, relative to the generator model during training.

The define_generator() function below implements the U-Net encoder-decoder generator model. It uses the **define_encoder_block()** helper function to create blocks of layers for the encoder and the **decoder_block()** function to create blocks of layers for the decoder. The tanh activation function is used in the output layer, meaning that pixel values in the generated image will be in the range [-1,1]. T**he architecture of C64 - C128 - C256 - C512 - C512- C512 - C512- C512 (encoder) and CD512- CD512- CD512 - CD512 - C512 - C256 - C128 - C64(decoder) is followed.**
Each block in the encoder is: Convolution -> Batch normalization -> Leaky ReLU. Each block in the decoder is: Transposed convolution -> Batch normalization -> Dropout -> ReLU. Skip connections are made between the encoder and decoder.

Now, the we define the combined generator and discriminator model **where the discriminant layers are set to untrainable.** The same learning rate is used. The generator model is trained via the discriminator model. It is updated to minimize the loss predicted by the discriminator for generated images marked as “real.” The generator is also updated to minimize the L1 loss between the generated image and the target image. Weight of 1:100 is used (BCE:L1 Loss)
The **define_gan()** function implements this.

The **generate_real_samples() function** below will prepare a batch of random pairs of images from the training dataset, and the corresponding discriminator label of class=1 to indicate they are real. The variable **trainA** stores the satellite images and the variable **trainB** stores the corresponding real street-view images. Now, you select a batch and store in the variable named **generate_fake_samples.** The fake generated images from the generator are stored in the variable named **X. You assign them values of 0.**

We can then review the generated images at the end of training and use the image quality to choose a final model. The **summarize_performance() function** implements this.

Once we are done with the function definitions, we proceed towards training. The paper suggests to use a batch size of 1. Number of epochs =100. You create a batch of real and fake samples, you train the discriminator on the real samples, then you train the discriminator on the fake set(Real Satellite concatenated on Fake generated image). You update the generator and then print the loss functions.

The loss is reported each training iteration, including the discriminator loss on real examples (d1), discriminator loss on generated or fake examples (d2), and generator loss, which is a weighted average of adversarial and L1 loss (g).

Models are saved every 10 epochs and saved to a file with the training iteration number. Additionally, images are generated every 10 epochs and compared to the expected target images. These plots are assessed at the end of the run and used to select a final generator model based on generated image quality.

B) Code:

```
### Importing the necessary packages

from os import listdir
from numpy import asarray, load
from numpy import vstack
from keras.preprocessing.image import img_to_array
from keras.preprocessing.image import load_img
from numpy import savez_compressed
from matplotlib import pyplot
import numpy as np

from numpy import zeros
from numpy import ones
```

```
from numpy.random import randint
from tensorflow.keras.optimizers import Adam
from keras.initializers import RandomNormal
from keras.models import Model
from keras.models import Input
from keras.layers import Conv2D
from keras.layers import Conv2DTranspose
from keras.layers import LeakyReLU
from keras.layers import Activation
from keras.layers import Concatenate
from keras.layers import Dropout
from keras.layers import BatchNormalization
from matplotlib import pyplot as plt
from tensorflow.keras.utils import plot_model


# load all images in a directory into memory
def load_images(path, size=(256,512)):
  src_list, tar_list = list(), list()
  # enumerate filenames in directory, assume all are images
  for filename in listdir(path):
     # load and resize the image
     pixels = load_img(path + filename, target_size=size)
     # convert to numpy array
     pixels = img_to_array(pixels)
     # split into satellite and map
     sat_img, map_img = pixels[:, :256], pixels[:, 256:]
     src_list.append(sat_img)
     tar_list.append(map_img)
  return [asarray(src_list), asarray(tar_list)]

# dataset path
path = "C:/Users/khushgarg/Documents/maps/train/"
# load dataset
[src_images, tar_images] = load_images(path)
print('Loaded: ', src_images.shape, tar_images.shape)


n_samples = 3
for i in range(n_samples):
  pyplot.subplot(2, n_samples, 1 + i)
  pyplot.axis('off')
```

```
  pyplot.imshow(src_images[i].astype('uint8'))
# plot target image
for i in range(n_samples):
  pyplot.subplot(2, n_samples, 1 + n_samples + i)
  pyplot.axis('off')
  pyplot.imshow(tar_images[i].astype('uint8'))
pyplot.show()

#####################################

def define_discriminator(image_shape):
   # weight initialization
   init = RandomNormal(stddev=0.02)  # As described in the original paper

   # source image input
   in_src_image = Input(shape=image_shape)  # Image we want to convert to another
image
   # target image input
   in_target_image = Input(shape=image_shape)  # Image we want to generate after
training.

   # concatenate images, channel-wise
   merged = Concatenate()([in_src_image, in_target_image])

   # C64: 4x4 kernel Stride 2x2
   d = Conv2D(64, (4, 4), strides=(2, 2), padding='same',
kernel_initializer=init)(merged)
   d = LeakyReLU(alpha=0.2)(d)
   # C128: 4x4 kernel Stride 2x2
   d = Conv2D(128, (4, 4), strides=(2, 2), padding='same', kernel_initializer=init)(d)
   d = BatchNormalization()(d)
   d = LeakyReLU(alpha=0.2)(d)
   # C256: 4x4 kernel Stride 2x2
   d = Conv2D(256, (4, 4), strides=(2, 2), padding='same', kernel_initializer=init)(d)
   d = BatchNormalization()(d)
   d = LeakyReLU(alpha=0.2)(d)
   # C512: 4x4 kernel Stride 2x2
   d = Conv2D(512, (4, 4), padding='same', kernel_initializer=init)(d)
   d = BatchNormalization()(d)
   d = LeakyReLU(alpha=0.2)(d)
   # patch output
   d = Conv2D(1, (4, 4), padding='same', kernel_initializer=init)(d)
```

```
    patch_out = Activation('sigmoid')(d)
    # define model
    model = Model([in_src_image, in_target_image], patch_out)
    # compile model
    # The loss for the discriminator is weighted by 50% for each model update.

    opt = Adam(lr=0.0002, beta_1=0.5)
    model.compile(loss='binary_crossentropy', optimizer=opt, loss_weights=[0.5])
    return model


################
# Now define the U-NET generator
# define an encoder block to be used in generator
def define_encoder_block(layer_in, n_filters, batchnorm=True):
    # weight initialization
    init = RandomNormal(stddev=0.02)
    # add downsampling layer
    g = Conv2D(n_filters, (4, 4), strides=(2, 2), padding='same',
kernel_initializer=init)(layer_in)
    # conditionally add batch normalization
    if batchnorm:
        g = BatchNormalization()(g, training=True)
    # leaky relu activation
    g = LeakyReLU(alpha=0.2)(g)
    return g


# define a decoder block to be used in generator
def decoder_block(layer_in, skip_in, n_filters, dropout=True):
    # weight initialization
    init = RandomNormal(stddev=0.02)
    # add upsampling layer
    g = Conv2DTranspose(n_filters, (4, 4), strides=(2, 2), padding='same',
kernel_initializer=init)(layer_in)
    # add batch normalization
    g = BatchNormalization()(g, training=True)
    # conditionally add dropout
    if dropout:
        g = Dropout(0.5)(g, training=True)
    # merge with skip connection
    g = Concatenate()([g, skip_in])
```

```
    # relu activation
    g = Activation('relu')(g)
    return g


# define the standalone generator model - U-net
def define_generator(image_shape=(256, 256, 3)):
    # weight initialization
    init = RandomNormal(stddev=0.02)
    # image input
    in_image = Input(shape=image_shape)
    # encoder model: C64-C128-C256-C512-C512-C512-C512-C512
    e1 = define_encoder_block(in_image, 64, batchnorm=False)
    e2 = define_encoder_block(e1, 128)
    e3 = define_encoder_block(e2, 256)
    e4 = define_encoder_block(e3, 512)
    e5 = define_encoder_block(e4, 512)
    e6 = define_encoder_block(e5, 512)
    e7 = define_encoder_block(e6, 512)
    # bottleneck, no batch norm and relu
    b = Conv2D(512, (4, 4), strides=(2, 2), padding='same',
kernel_initializer=init)(e7)
    b = Activation('relu')(b)
    # decoder model: CD512-CD512-CD512-C512-C256-C128-C64
    d1 = decoder_block(b, e7, 512)
    d2 = decoder_block(d1, e6, 512)
    d3 = decoder_block(d2, e5, 512)
    d4 = decoder_block(d3, e4, 512, dropout=False)
    d5 = decoder_block(d4, e3, 256, dropout=False)
    d6 = decoder_block(d5, e2, 128, dropout=False)
    d7 = decoder_block(d6, e1, 64, dropout=False)
    # output
    g = Conv2DTranspose(image_shape[2], (4, 4), strides=(2, 2), padding='same',
kernel_initializer=init)(
        d7)  # Modified
    out_image = Activation('tanh')(g)  # Generates images in the range -1 to 1. So
change inputs also to -1 to 1
    # define model
    model = Model(in_image, out_image)
    return model
```

```
# define the combined generator and discriminator model, for updating the generator
def define_gan(g_model, d_model, image_shape):
   for layer in d_model.layers:
       if not isinstance(layer, BatchNormalization):
           layer.trainable = False  # Discriminator layers set to untrainable in the
combined GAN but
           # standalone descriminator will be trainable.

   # define the source image
   in_src = Input(shape=image_shape)
   # supply the image as input to the generator
   gen_out = g_model(in_src)
   # supply the input image and generated image as inputs to the discriminator
   dis_out = d_model([in_src, gen_out])
   # src image as input, generated image and disc. output as outputs
   model = Model(in_src, [dis_out, gen_out])
   # compile model
   opt = Adam(lr=0.0002, beta_1=0.5)

   # Total loss is the weighted sum of adversarial loss (BCE) and L1 loss (MAE)
   # Authors suggested weighting BCE vs L1 as 1:100.
   model.compile(loss=['binary_crossentropy', 'mae'],
                 optimizer=opt, loss_weights=[1, 100])
   return model


# select a batch of random samples, returns images and target
def generate_real_samples(dataset, n_samples, patch_shape):
   # unpack dataset
   trainA, trainB = dataset
   # choose random instances
   ix = randint(0, trainA.shape[0], n_samples)
   # retrieve selected images
   X1, X2 = trainA[ix], trainB[ix]
   # generate 'real' class labels (1)
   y = ones((n_samples, patch_shape, patch_shape, 1))
   return [X1, X2], y


# generate a batch of images, returns images and targets
```

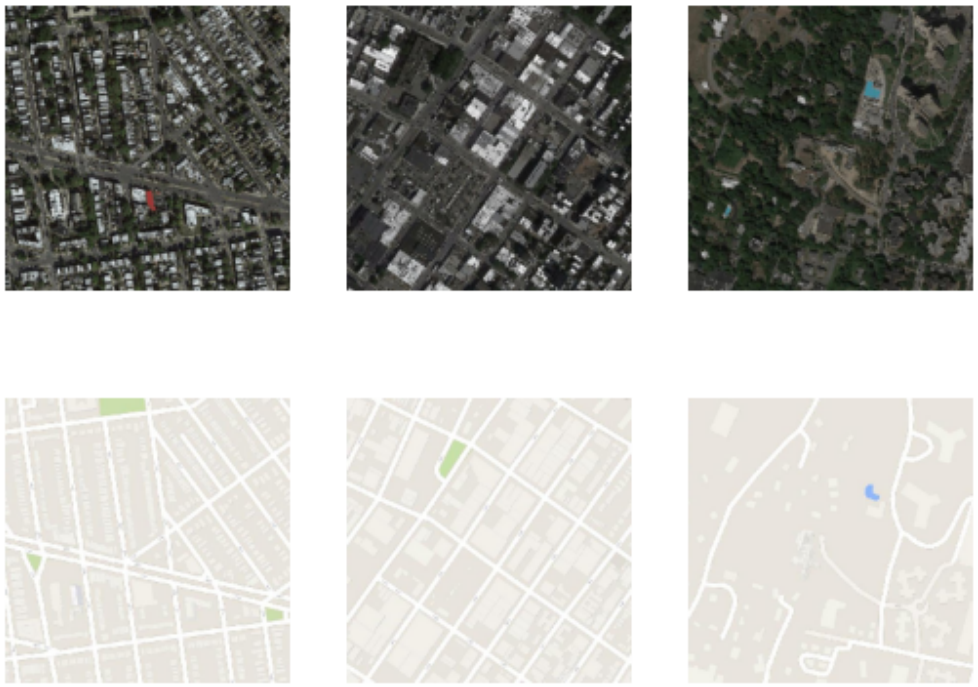
```
def generate_fake_samples(g_model, samples, patch_shape):
   # generate fake instance
   X = g_model.predict(samples)   ##generator makes these images
   # create 'fake' class labels (0)
   y = zeros((len(X), patch_shape, patch_shape, 1))
   return X, y


# save the generator model and check how good the generated image looks.
def summarize_performance(step, g_model, dataset, n_samples=3):
   # select a sample of input images
   [X_realA, X_realB], _ = generate_real_samples(dataset, n_samples, 1)
   # generate a batch of fake samples
   X_fakeB, _ = generate_fake_samples(g_model, X_realA, 1)
   # scale all pixels from [-1,1] to [0,1]
   X_realA = (X_realA + 1) / 2.0
   X_realB = (X_realB + 1) / 2.0
   X_fakeB = (X_fakeB + 1) / 2.0
   # plot real source images
   for i in range(n_samples):
       plt.subplot(3, n_samples, 1 + i)
       plt.axis('off')
       plt.imshow(X_realA[i])
   # plot generated target image
   for i in range(n_samples):
       plt.subplot(3, n_samples, 1 + n_samples + i)
       plt.axis('off')
       plt.imshow(X_fakeB[i])
   # plot real target image
   for i in range(n_samples):
       plt.subplot(3, n_samples, 1 + n_samples * 2 + i)
       plt.axis('off')
       plt.imshow(X_realB[i])
   # save plot to file
   filename1 = 'plot_%06d.png' % (step + 1)
   plt.savefig(filename1)
   plt.close()
   # save the generator model
   filename2 = 'model_%06d.h5' % (step + 1)
   g_model.save(filename2)
   print('>Saved: %s and %s' % (filename1, filename2))
```

```
# training
def train(d_model, g_model, gan_model, dataset, n_epochs=100, n_batch=1):
   # determine the output square shape of the discriminator
   n_patch = d_model.output_shape[1]
   # unpack dataset
   trainA, trainB = dataset
   # calculate the number of batches per training epoch
   bat_per_epo = int(len(trainA) / n_batch)
   # calculate the number of training iterations
   n_steps = bat_per_epo * n_epochs
   # manually enumerate epochs
   for i in range(n_steps):
       # select a batch of real samples
       [X_realA, X_realB], y_real = generate_real_samples(dataset, n_batch, n_patch)
       # generate a batch of fake samples
       X_fakeB, y_fake = generate_fake_samples(g_model, X_realA, n_patch)
       # update discriminator for real samples
       d_loss1 = d_model.train_on_batch([X_realA, X_realB], y_real)
       # update discriminator for generated samples
       d_loss2 = d_model.train_on_batch([X_realA, X_fakeB], y_fake)
       # update the generator
       g_loss, _, _ = gan_model.train_on_batch(X_realA, [y_real, X_realB])
       # summarize performance
       print('>%d, d1[%.3f] d2[%.3f] g[%.3f]' % (i + 1, d_loss1, d_loss2, g_loss))
       # summarize model performance
       if (i + 1) % (bat_per_epo * 10) == 0:
           summarize_performance(i, g_model, dataset)



# define input shape based on the loaded dataset
image_shape = src_images.shape[1:]
# define the discriminator and generator models
d_model = define_discriminator(image_shape)
g_model = define_generator(image_shape)
# define the composite model
gan_model = define_gan(g_model, d_model, image_shape)

#Define data
# load and prepare training images
data = [src_images, tar_images]
```

```
def preprocess_data(data):
  # load compressed arrays
  # unpack arrays
  X1, X2 = data[0], data[1]
  # scale from [0,255] to [-1,1]
  X1 = (X1 - 127.5) / 127.5
  X2 = (X2 - 127.5) / 127.5
  return [X1, X2]

dataset = preprocess_data(data)

from datetime import datetime
start1 = datetime.now()

train(d_model, g_model, gan_model, dataset, n_epochs=10, n_batch=1)



stop1 = datetime.now()
#Execution time of the model
execution_time = stop1-start1
print("Execution time is: ", execution_time)

#R

##############################################

#Test trained model on a few images...

from keras.models import load_model
from numpy.random import randint
model = load_model('saved_model_10epochs.h5')

# plot source, generated and target images
def plot_images(src_img, gen_img, tar_img):
  images = vstack((src_img, gen_img, tar_img))
  # scale from [-1,1] to [0,1]
  images = (images + 1) / 2.0
  titles = ['Source', 'Generated', 'Expected']
  # plot images row by row
  for i in range(len(images)):
     # define subplot
```

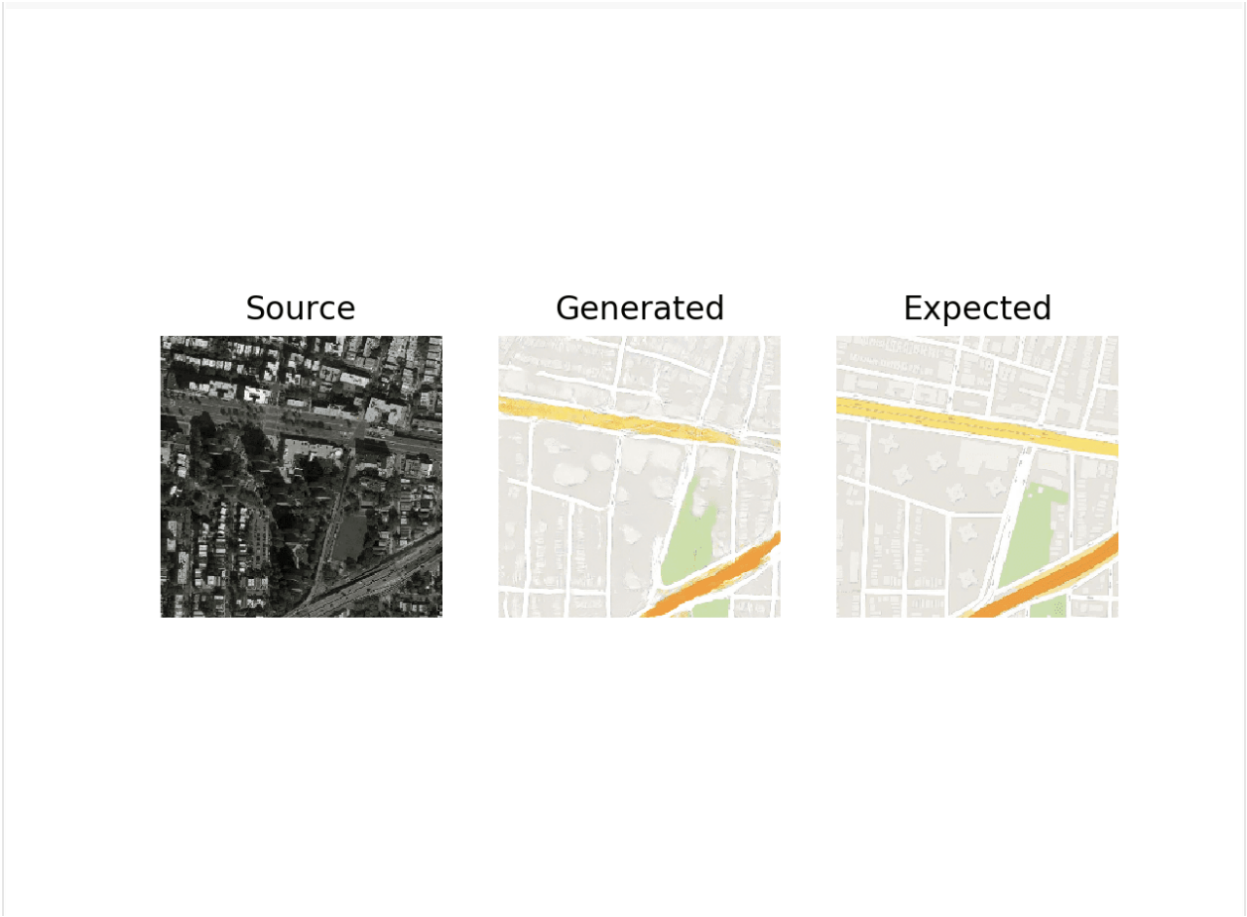
```
    pyplot.subplot(1, 3, 1 + i)
    # turn off axis
    pyplot.axis('off')
    # plot raw pixel data
    pyplot.imshow(images[i])
    # show title
    pyplot.title(titles[i])
  pyplot.show()



[X1, X2] = dataset
# select random example
ix = randint(0, len(X1), 1)
src_image, tar_image = X1[ix], X2[ix]
# generate image from source
gen_image = model.predict(src_image)
# plot all three images
plot_images(src_image, gen_image, tar_image)
```

C) Results:



The above figure displays the real pairs of satellite-street view images. These are of 256 * 256 pixels resolution. The code above makes sure that all these pairs have the following size: 256 *256. Having the pairs in this form makes it easier to use this for the discriminator model. The images are normalized in the range of [-1,1]

After the first 10 epochs, map images are generated that look plausible but have blurry results. Generated images after about 50 training epochs begin to look very realistic.



This figure shows the source real satellite image on the left, generated street-view image in the middle, and real street-view image on the right.