Ravi Ranjan Kumar

8 Followers About Follow

Sketch to Color Anime: An application of Conditional GAN.

Ravi Ranjan Kumar Feb 29, 2020 · 10 min read

Don't you think it seems magical?

Introduction

On the stage of Deep Learning, things look like magical but the idea that works behind the curtain is totally conceptual. In this post we will be focusing over **Generative**

Table of contents:

1. Prerequisites

This post assumes familiarity with basic knowledge of Deep Learning techniques like CNN, python syntax, some libraries like numpy, pillow,keras, etc.

2. About Data

Data has been collected from kaggle.

Danbooru Sketch Pair 128x	

The Data provided is of size 10GB which contains approx 300K sketch-color pair. We will sample approx 100K sketch-color pair images for training which is approx 3GB of data. Each data provided is in form of sketch-color pair having shape (128x128x3).

3. Sampling Data

After unzipping the downloaded file, we got “danbooru-sketch-pair-128x” folder, the hierarchy tree of this folder is as follow:-

```
danbooru-sketch-pair-128x/
                          color/
                                sketch/
                                      0000/
                                           *.png ...
                                      ...
                                      0150/
                                           *.png ...
                                src/
                                      0000/
                                           *.png ...
                                      ...
                                      0150/
                                           *.png ...
                          gray/
                                sketch/
                                      0000/
                                           *.png ...
                                      ...
                                      0150/
                                           *.png ...
                                src/
                                      0000/
                                           *.png ...
                                      ...
                                      0150/
                                           *.png ...
```

Folder “gray” contains the gray shades images so we drop folder “gray” as we want to work only with colorful sketch-color pair.

data set, no need to keep those sub folders.

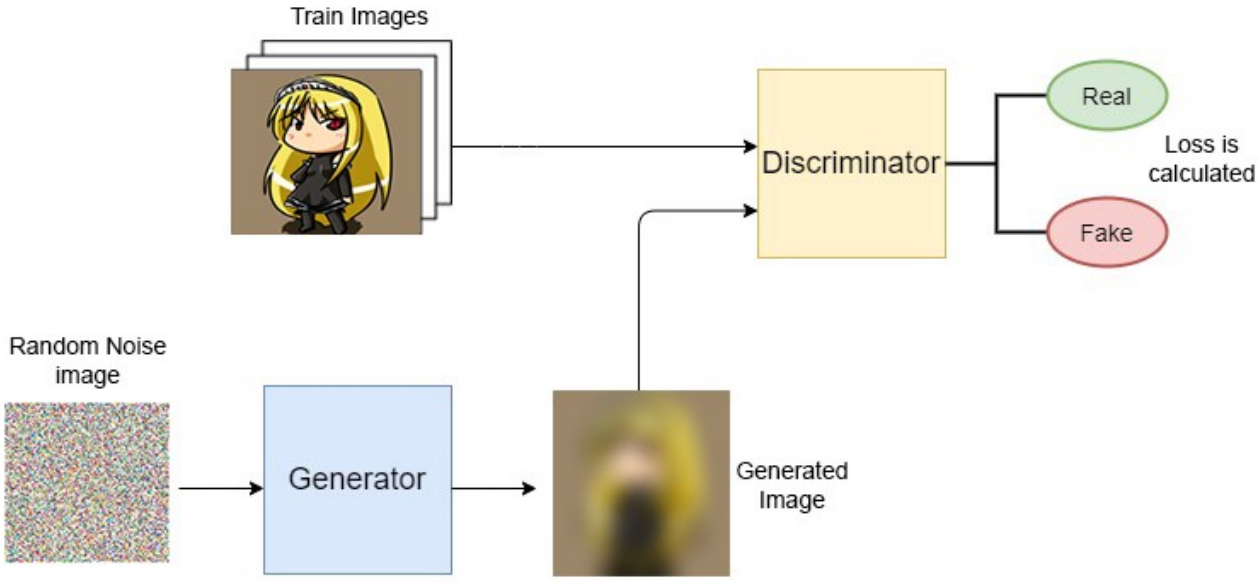
So, first we will create a dictionary having key as sub-folder name and value as list of names of images inside it, then we will sample image names from this dictionary and create our train and test set.

```
def sample_data(path):
    '''
    This Function take a path where data is stored
    example:- ".\\danbooru-sketch-pair-128x\\color\\"
    it creates two folder train_set, and test_set with sketch-color pair images
    '''
    # storing all the names of images in a dictionary where key is sobfolder name
    # and value is a list of names of images in that sub folder
    folder_dict = dict()
    for d in os.listdir(path+"/sketch"):
        folder_dict[d] = [img_name for img_name in os.listdir(path+"/sketch/"+d)]

    # sampling 700 images from each sub-folder
    sample_train = dict()
    for key, val in folder_dict.items():
        index = random.sample(range(0, len(val)), 700)
        sample_train[key] = [val[i] for i in index]

    # moving 700 images from each folder of sketch directory to train_set
    split_data(sample_train,path,"sketch","train_set")
    # moving 700 images from each folder of src directory to train_set
    split_data(sample_train,path,"src","train_set")

    # sampling 40 images from each folder of train_set directory to test_set
    sample_test = dict()
    for key, val in sample_train.items():
        index = random.sample(range(0, 700), 40)
        sample_test[key] = [val[i] for i in index]

    # moving 40 images from each folder of sketch directory to test_set
    split_train_data(sample_test,"train_set","sketch","test_set")
    # moving 40 images from each folder of src directory to test_set
    split_train_data(sample_test,"train_set","src","test_set")

```

Now our data is ready to process, let's see the modeling part.

4. Generative Adversarial Network (GAN)

The idea of GAN was proposed by **Ian J. Goodfellow** in june, 2014. The idea is to generate fake data after learning from the original data. GAN is a combination of two networks, the **Generator network** and the **Discriminator network.**



An Overview of GAN.

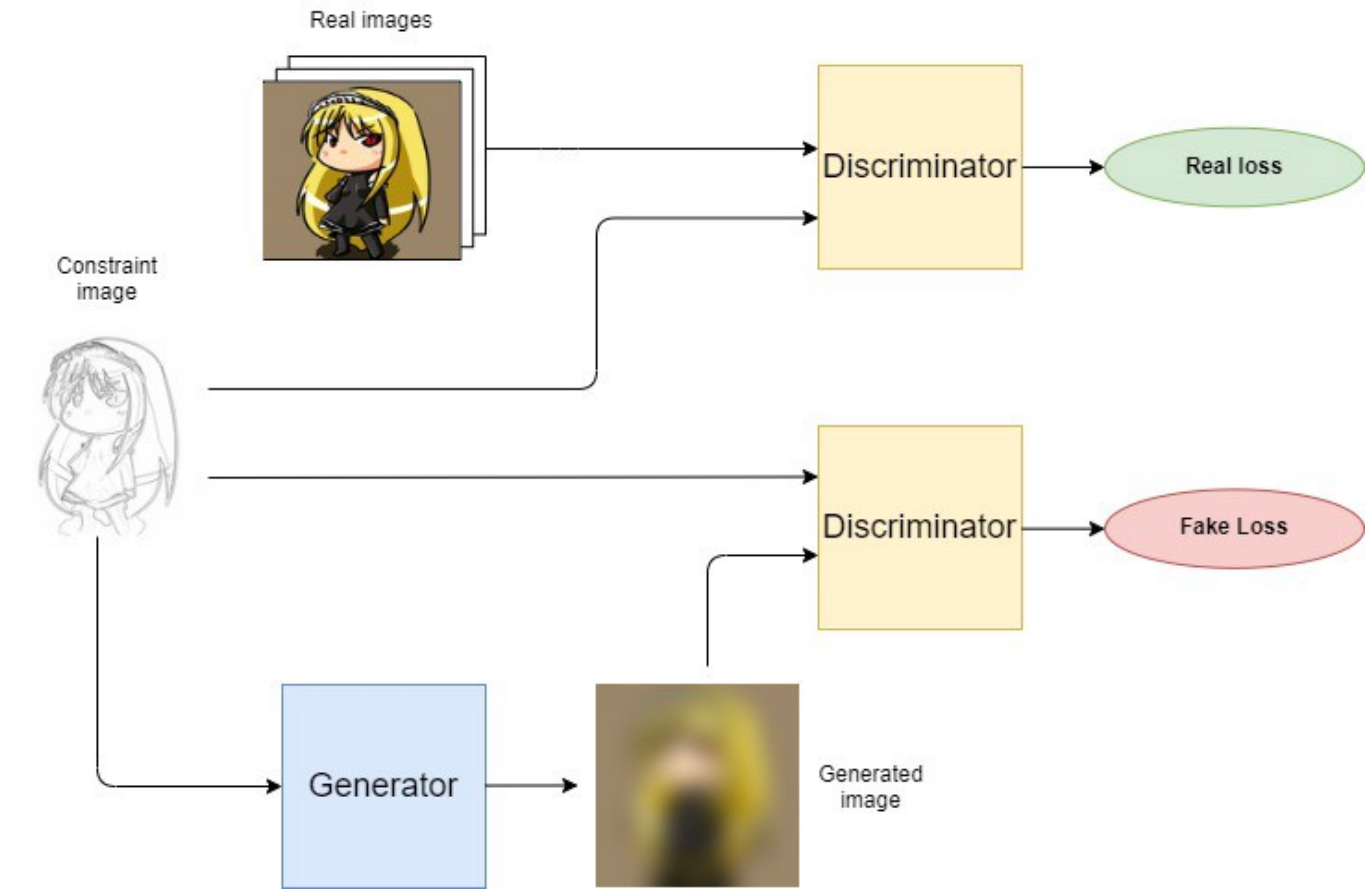
The generator takes a random noise image and tries to generate fake images similar to those which present in training set. The discriminator takes both the original image and the generated image. It tries to differentiate between real and generated image, if the output is **fake** i.e. the generated image is different from the real one then the loss is computed and the weights of discriminator is updated. This loop is repeated until we get our desired result, i.e. the discriminator should not be able to differentiate between real and generated image, in such case my generator is trained so well that it can easily fool the discriminator.

GAN comes under **Unsupervised Learning**, which uses a Supervised loss as a part of training. For understanding GAN in depth, you should read the ***original GAN paper***.

5. Conditional Generative Adversarial Network (CGAN)

constraint over generator to generate images according to the constraint provided.

CGAN also comes under **Unsupervised Learning.** I found the best reason ***here.***



An overview of Conditional GAN.

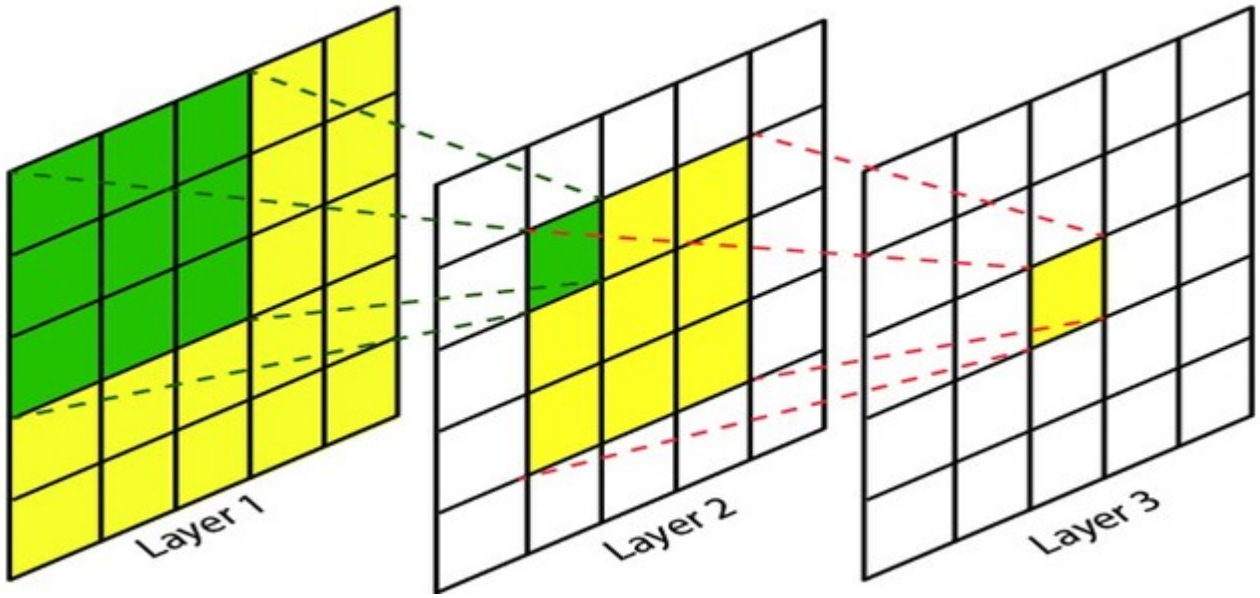
> *Instead of random noise vector, we are providing sketch to the generator, it is like an extra information about the distribution of training data.*

Real loss is calculated between real image and its corresponding sketch image. Fake loss is calculated between generated image and its corresponding sketch image.

Our problem falls under the category of Conditional GAN. Please go through the research paper ***Image-to-Image Translation with Conditional Adversarial Networks*** by **Phillip Isola** and his team, you will get a detailed idea about Conditional GAN and its wide range of applications.

Now, let's see the architecture of Discriminator and Generator in detail.

into the architecture we need to understand about **effective receptive field.** We can define it as the amount of cross-section covered by each pixel of output grid.

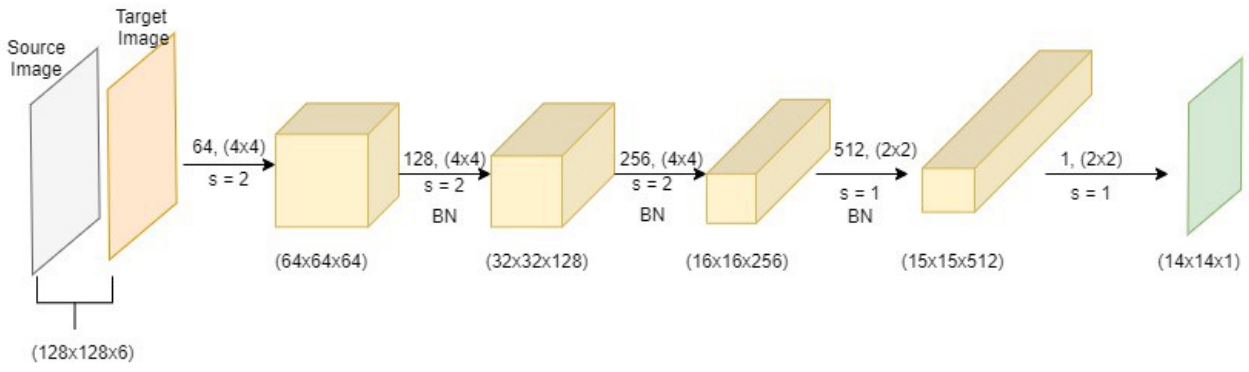


Layer 1 : Input Layer | Layer 2 : Hidden Layer | Layer 3: Output Layer

Kernel (3x3) is convolved over (5x5) matrix in layer 1, therefore a single pixel in layer 2 will see (3x3) cross-section in layer 1, similarly in layer 3 a single pixel will see (3x3) in layer 2 and (5x5) in layer 1. So in layer 3 a single pixel will cover (5x5) cross section from Layer 1. This (5x5) is effective receptive field. We need to back propagate to calculate effective receptive field for a network. Formula is:-

Receptive field = (output size - 1) * stride + kernel size

Discriminator Architecture.



The BN is Batch Normalization Layer, and the output is (14x14x1) grid, each cell of output layer sees (38x38) grid from the input layer. ***How?*** Each cell of output layer stores probability value that each (38x38) grid from input layer is fake or real. This technique is termed as **PatchGAN.**

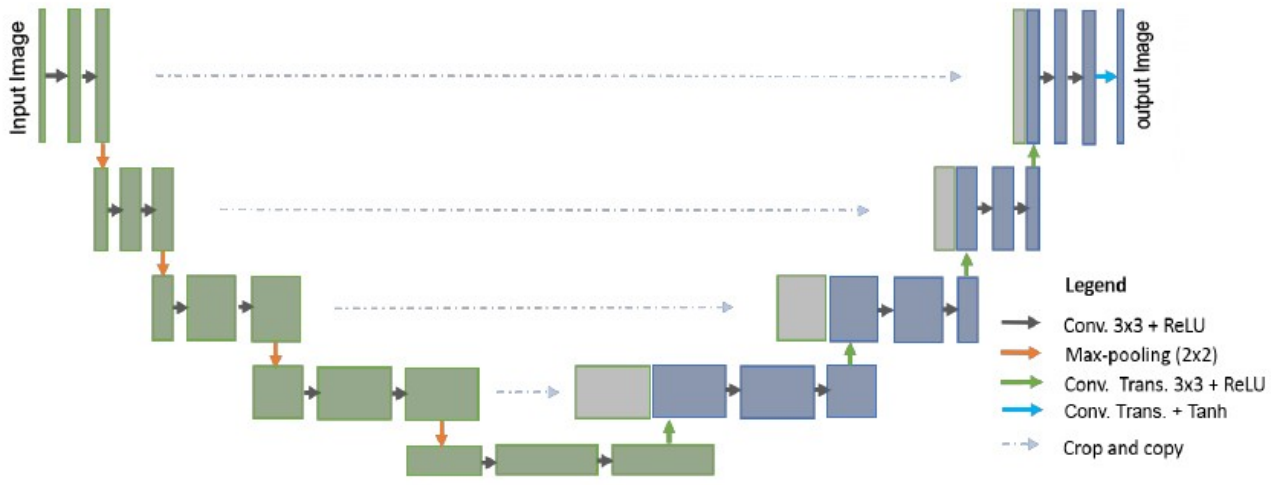
```
def discriminator(img_shape):
    '''
    This function takes the shape of image as input and returns the discriminator as PatchGAN.
    '''

    # source image
    source_image = Input(shape=img_shape)
    # target image
    target_image = Input(shape=img_shape)

    # defining kernel_initializer
    k_in = RandomNormal(stddev=0.02)

    # concatenating images
    con = Concatenate()([source_image, target_image])

    dis = Conv2D(64, (4,4), strides=(2,2), padding='same', kernel_initializer=k_in)(con) #64x64x
    dis = LeakyReLU(alpha=0.2)(dis)

    dis = Conv2D(128, (4,4), strides=(2,2), padding='same', kernel_initializer=k_in)(dis) #32x32
    dis = BatchNormalization()(dis)
    dis = LeakyReLU(alpha=0.2)(dis)


    dis = Conv2D(256, (4,4), strides=(2,2), padding='same', kernel_initializer=k_in)(dis) #16x16
    dis = BatchNormalization()(dis)
    dis = LeakyReLU(alpha=0.2)(dis)


    # second last layer
    dis = Conv2D(512, (2,2), kernel_initializer=k_in)(dis) #15x15x512
    dis = BatchNormalization()(dis)
    dis = LeakyReLU(alpha=0.2)(dis)

    # last layer
    out = Conv2D(1, (2,2), activation = "sigmoid", padding='valid', kernel_initializer=k_in)(dis
```

```
    # defining model
    model = Model([source_image, target_image], final_out)
    model.compile(loss='binary_crossentropy', optimizer=Adam(lr=0.0002, beta_1=0.5), loss_weight
    return model


# checking model
image_shape = (128,128,3)
model = discriminator(image_shape)

# summary
model.summary()
```

discriminator.py hosted with ♡ by GitHub view raw

7. The Generator Architecture

We are using a special architecture called U-net for Generator model. U-net is an Encoder-Decoder network with skip connections.



The U-Net Architecture | Research Paper on U-Net by Olaf Ronneberger, **click here** to read

The U-net takes input and tries to reduce it with a series of encoders (green color boxes in image) into a much smaller representation. The idea is that by compressing it this way we hopefully have a higher level representation of the data after the final encode layer.

U-net is used for learning image segmentation, it catches fine details about pixels and boundary regions, we also need to segment the different regions of sketch image and provide a proper color to it, this is the reason of using U-net network.

```
def generator(img_shape):

    '''
    This function takes the shape of image as input and returns the generator model as U-net arc
    '''

    k_in = RandomNormal(stddev=0.02)

    # image input
    sketch_img = Input(shape=img_shape)

    # encoder
    gen_en1 = Conv2D(64, (4,4), strides=(2,2), padding='same', kernel_initializer=k_in)(sketch_i
    gen_en1 = LeakyReLU(alpha=0.2)(gen_en1)

    gen_en2 = Conv2D(128, (4,4), strides=(2,2), padding='same', kernel_initializer=k_in)(gen_en1
    gen_en2 = BatchNormalization()(gen_en2, training=True)
    gen_en2 = LeakyReLU(alpha=0.2)(gen_en2)

    gen_en3 = Conv2D(256, (4,4), strides=(2,2), padding='same', kernel_initializer=k_in)(gen_en2
    gen_en3 = BatchNormalization()(gen_en3, training=True)
    gen_en3 = LeakyReLU(alpha=0.2)(gen_en3)

    gen_en4 = Conv2D(512, (4,4), strides=(2,2), padding='same', kernel_initializer=k_in)(gen_en3
    gen_en4 = BatchNormalization()(gen_en4, training=True)
    gen_en4 = LeakyReLU(alpha=0.2)(gen_en4)

    gen_en5 = Conv2D(512, (4,4), strides=(2,2), padding='same', kernel_initializer=k_in)(gen_en4
    gen_en5 = BatchNormalization()(gen_en5, training=True)
    gen_en5 = LeakyReLU(alpha=0.2)(gen_en5)

    gen_en6 = Conv2D(512, (4,4), strides=(2,2), padding='same', kernel_initializer=k_in)(gen_en5
    gen_en6 = BatchNormalization()(gen_en6, training=True)
    gen_en6 = LeakyReLU(alpha=0.2)(gen_en6)

```

```
    # decoder
    gen_de1 = Conv2DTranspose(512, (4,4), strides=(2,2), padding='same', kernel_initializer=k_in
    gen_de1 = BatchNormalization()(gen_de1, training=True)
    gen_de1 = Dropout(0.5)(gen_de1, training=True)
    gen_de1 = Concatenate()([gen_de1, gen_en6])
    gen_de1 = Activation('relu')(gen_de1)

    gen_de2 = Conv2DTranspose(512, (4,4), strides=(2,2), padding='same', kernel_initializer=k_in
    gen_de2 = BatchNormalization()(gen_de2, training=True)
    gen_de2 = Dropout(0.5)(gen_de2, training=True)
    gen_de2 = Concatenate()([gen_de2, gen_en5])
    gen_de2 = Activation('relu')(gen_de2)

    gen_de3 = Conv2DTranspose(512, (4,4), strides=(2,2), padding='same', kernel_initializer=k_in
    gen_de3 = BatchNormalization()(gen_de3, training=True)
    gen_de3 = Dropout(0.5)(gen_de3, training=True)
    gen_de3 = Concatenate()([gen_de3, gen_en4])
    gen_de3 = Activation('relu')(gen_de3)

    gen_de4 = Conv2DTranspose(256, (4,4), strides=(2,2), padding='same', kernel_initializer=k_in
    gen_de4 = BatchNormalization()(gen_de4, training=True)
    gen_de4 = Concatenate()([gen_de4, gen_en3])
    gen_de4 = Activation('relu')(gen_de4)

    gen_de5 = Conv2DTranspose(128, (4,4), strides=(2,2), padding='same', kernel_initializer=k_in
    gen_de5 = BatchNormalization()(gen_de5, training=True)
    gen_de5 = Concatenate()([gen_de5, gen_en2])
    gen_de5 = Activation('relu')(gen_de5)

    gen_de6 = Conv2DTranspose(64, (4,4), strides=(2,2), padding='same', kernel_initializer=k_in)
    gen_de6 = BatchNormalization()(gen_de6, training=True)
    gen_de6 = Concatenate()([gen_de6, gen_en1])
    gen_de6 = Activation('relu')(gen_de6)


    # output
    gen_col_img = Conv2DTranspose(3, (4,4), strides=(2,2), activation= 'tanh',padding='same', ke

    # define model
    model = Model(sketch_img, gen_col_img)
    return model
```

```
84  model = generator(image_shape)
85
86  # summary
87  model.summary()
```

generator.py hosted with ♡ by GitHub view raw

The network take an image as input and after processing through encoder and and decoder network it outputs an image.

8. Different Losses

The loss is something which we want to reduce over time with the help of optimization algorithm.

> ***Discriminator and Generator loss***

The Discriminator model tries to predict a patch from input image is real or fake. So it is trying to reduce **binary cross entropy** loss, either 0 or 1.

The loss function that is described in the research paper is:-

$$\mathcal{L}_{cGAN}(G, D) = \mathbb{E}_{x,y}[\log D(x, y)] + \mathbb{E}_{x,z}[\log(1 - D(x, G(x, z))],$$

The Conditional GAN loss. | G: Generator, D:Discriminator

'x' is sketch image or the constraint image, **'y'** is real color image, **'z'** is random image vector, In our case we are providing only **'x'** as the input to generator to generate fake images. The Discriminator tries to **maximize** the the probability of assigning the correct label to both training examples and generated samples. Generator tries to **minimize** the **'log(1 − D(x, G(x, z))' .**Both models are in a tug of war situation in which both have to win by reaching an **equilibrium state.**

Generator tries to **minimize** this loss function.

To get good result over our problem we introduce more losses like L1 loss, L2 loss, Total Variation loss, let's see them in detail.

L1 loss

Popularly known as Manhattan distance. For each pixel of real color image '**y**' and generated image '**G(x, z)**', we find the average of distance between them . It is also known as **Pixel Level Loss.**

$$\mathcal{L}_{L1}(G) = \mathbb{E}_{x,y,z}\left[\|y - G(x,z)\|_1\right].$$

L1 loss

```
def pixel_loss(true, pred):
    '''
    parameter "true" is real color image, parameter "pred" is generated image,
    For each pixel of real color image and generated image it returns the average
    of distance between them.
    '''
    def p_loss(y_true, y_pred):
        return K.mean( K.abs( true - pred ) )
    return p_loss
```

pixel_loss.py hosted with ♡ by GitHub view raw

L2 loss

Popularly known as euclidean distance. We get the feature map from the pretrained VGG 16 net and calculate the euclidean distance between feature map of real color image '**y**' and generated image '**G(x, z)**', It is also known as **Feature Level Loss.** We choose output of 4th layer of VGG 16 net to calculate L2 distance.

φj is feature map layer, for our case j=4

```
def feature_loss(true, pred):
    '''
    parameter "true" is feature map of real color image,
    parameter "pred" is feature map of generated image,
    it returns the euclidean distance between feature map
    of real color image and generated image.
    '''
    def f_loss(y_true, y_pred):
        return K.mean( K.sqrt( K.sum( K.square( true - pred ))))
    return f_loss
```

feature_level_loss.py hosted with ♡ by GitHub view raw

Total Variation Loss

It is calculated only for generated image **'G(x, z)'.** It is square root of sum of differences between neighboring pixels, it is done to avoid the noise and provide smoothness in the image.

$$L_{tv} = \sqrt{(y_{i+1,j} - y_{i,j})^2 + (y_{i,j+1} - y_{i,j})^2}$$

We combine all loss together with their own weights to generate our final loss function.

```
def variation_loss(pred):
    '''
    parameter "pred" is generated image,
    it returns the square root of sum of differences between neighbouring pixels
    of geberated image
    '''
    def v_loss(y_pred):
        return K.sqrt(K.sum(K.square(pred[:, 1:, :, :] - pred[:, :-1, :, :]))\
                            + K.sum(K.square(pred [:, :, 1:, :] - pred [:, :, :-1, :])))
    return v_loss
```

$$L = w_p L_P + w_f L_f + w_G L_G + w_{tv} L_{tv}$$

We train our model over this loss function. You can read about these losses in ***this research paper*** by **Yifan Liu** and his team.

9. The GAN Model

Combining both generator and discriminator results in the formation of GAN model. While training GAN we freeze the weights of Discriminator i.e. the weights of Discriminator is not updated. This is done because we are calling both discriminator and generator in the function GAN and we can't update the weights of both network together as discriminator is using a loss function which is opposite of the generator's loss function. I got a good explanation of ***this question on Quora.***

```
def define_gan(g_model, d_model, image_shape):

    '''
    Takes generator, discriminator model and image_shape as input, and returns the GAN model as (
    '''

    def gan_loss(y_true, y_pred):
        '''
        returns the loss function of GAN.
        '''
        return tf.keras.losses.binary_crossentropy(y_true, y_pred) + pixel_loss_weight * pixel_l(
               variation_loss_weight * variation_loss_out(y_pred) + feature_loss_weight * featur(

    # Extracting intermediate layer features keras : https://keras.io/applications/#vgg16

    vgg_1 = Model(inputs=vgg16.input, outputs=ReLU()(vgg16.get_layer('block2_conv2').output))
    vgg_2 = Model(inputs=vgg16.input, outputs=ReLU()(vgg16.get_layer('block2_conv2').output))

    pixel_loss_weight = 100
    variation_loss_weight = .0001
    feature_loss_weight = .01

```

```
26      sketch_image = Input(image_shape)
27      gen_output = g_model([sketch_image])
28
29      #Discriminator
30      dis_output = d_model([sketch_image, gen_output])
31
32      #Pixel Loss
33      color_image = Input(image_shape)
34      pixel_loss_out = pixel_loss(color_image, gen_output)
35
36      #Variation Loss
37      variation_loss_out = variation_loss(gen_output)
38
39      #Feature Loss
40      vgg1_out = vgg_1([tf.image.resize(color_image, (128,128), tf.image.ResizeMethod.BILINEAR)])
41      vgg2_out = vgg_2([tf.image.resize(gen_output, (128,128), tf.image.ResizeMethod.BILINEAR)])
42
43      feature_loss_out = feature_loss(vgg1_out,vgg2_out)
44
45      #Final Model
46      model = Model(inputs=[sketch_image, color_image], outputs=dis_output)
47
48      #Single output multiple loss functions in keras : https://stackoverflow.com/a/51705573/907909
49
50      model.compile(loss= gan_loss, optimizer= Adam(lr=0.0002, beta_1=0.5))
51
52      return model
53
54
55  # checking the model
56  image_shape = (128,128,3)
57
58  g_model = generator(image_shape)
59
60  d_model = discriminator(image_shape)
61
62  gan_model = define_gan(g_model, d_model,image_shape)
63  gan_model.summary()
```

GAN.py hosted with ♡ by GitHub view raw

“generate_real_images”, this function will read the images from folders and return the actual sketch-color pair with class label as “1” which denotes real images.

```
def generate_real_images(sketch_path, color_image_path, n_sample):
  '''
    This function will read the data from paths provided and return the
    actual sketch-color pair along with class label as "1" which denotes real images.
  '''

    index = np.random.randint(0, total_img, n_sample)
    sketch_image = []
    color_image = []

    for sketch, img in zip(sketch_path[index], color_image_path[index]):
        sketch_image.append(np.array(Image.open(sketch).convert('RGB')))
        color_image.append(np.array(Image.open(img).convert('RGB')))

    # Normalizing the values to be between [-1, 1].
    sketch_image = (np.array(sketch_image, dtype='float32')/127.5 - 1)
    color_image = (np.array(color_image, dtype='float32')/127.5 - 1)

    y_real = np.ones((n_sample,1))

    return sketch_image, color_image, y_real
```

generate_real_images.py hosted with ♡ by GitHub view raw

We also load fake images. The function “generate_fake_images” read only sketch image from folder and generate fake images over sketch images and return fake sketch-color pair along with class label as “0” which denotes fake images.

```
def generate_fake_images(g_model, sketch_path, n_sample , seed_sketch=None, seed_color=None):
    '''
    This function will read only sketch data, and generate fake data using generator and return
    fake sketch-color pair along with class label as "0" which denotes fake images.
    '''

    sketch_image = []
    fake_color_image = []
```

```
        return seed_color, fake_color_image

    else:
        index = np.random.randint(0, total_img, n_sample)

        for sketch in sketch_path[index]:
            sketch_image.append(np.array(Image.open(sketch).convert('RGB')))

        # Normalizing the values to be between [-1, 1].
        sketch_image = (np.array(sketch_image, dtype='float32')/127.5 - 1)

        fake_color_image = g_model.predict(sketch_image)
        y_fake = np.zeros((n_sample, 1))

        return sketch_image, fake_color_image, y_fake
```

generate_fake_images.py hosted with ♡ by GitHub view raw

11. Training the Model

Training the model is tricky part, we randomly sample real images and fake images in batch and pass it for training over discriminator model. We need to compute real and fake loss for that we have to call discriminator twice for each generator call, this will cause my discriminator to over learn. So we call discriminator alternatively half of times and half of times together with half batch of data. This will train my discriminator model.

Now we sample real data and call GAN model over it, it will train the generator model, and here we freeze the discriminator weights. We display results and save model for each epoch.

```
def train(g_model, d_model, gan_model, sketch_path, color_image_path, seed_sketch, seed_color, n
    '''
    This function train the model over randomly sampled batch of real images
    and fake images. Both discriminator and generator is trained alternatively.
    Results are displayed and generator model is saved after each epoch.
    '''
    batch_per_epoch = int(total_img / n_batch)
    half_batch = int(n_batch / 2) # this is created for discriminator as we are calling it twice
```

```
        generator_loss = []
        discriminator_loss = []

        for j in range(batch_per_epoch):

            # discriminator model needs to be called twice
            if not j%2: # train only for odd value of j, to avoid over training

                # Discriminator real loss
                real_sketch, real_col_image, y_real = generate_real_images(sketch_path, color_im
                d_loss_real = d_model.train_on_batch([real_sketch, real_col_image], y_real * .9)

            if not j%3: # train only when j is multiple of 3. to avoid over training

                # Discriminator fake loss
                fake_sketch, fake_col_image, y_fake = generate_fake_images(g_model, sketch_path,
                d_loss_fake = d_model.train_on_batch([fake_sketch, fake_col_image], y_fake)

            total_dis_loss = d_loss_real + d_loss_fake

            #GAN loss
            real_sketch, real_col_image, y_real = generate_real_images(sketch_path, color_image_
            gan_loss = gan_model.train_on_batch([real_sketch, real_col_image], y_real)

            # saving loss per epoch
            discriminator_loss.append(total_dis_loss)
            generator_loss.append(gan_loss)


            # output after 100 iter
            if not j % 100:
                print("epoch>{}, {}/{}, d_real={:.3f}, d_fake={:.3f}, gan={:.3f}".format(i+1, j+
                                                                                    d_loss_


        # writing to tensorboard
        write_log(disc_callback, 'discriminator_loss', np.mean(discriminator_loss), i+1, (i+1)%3
        write_log(gen_callback, 'generator_loss', np.mean(generator_loss), i+1, (i+1)%3==0)

        #Summary after every epoch.
        display.clear_output(True)
        print('Time for epoch {} : {}'.format(i+1, datetime.now()-start2))
```

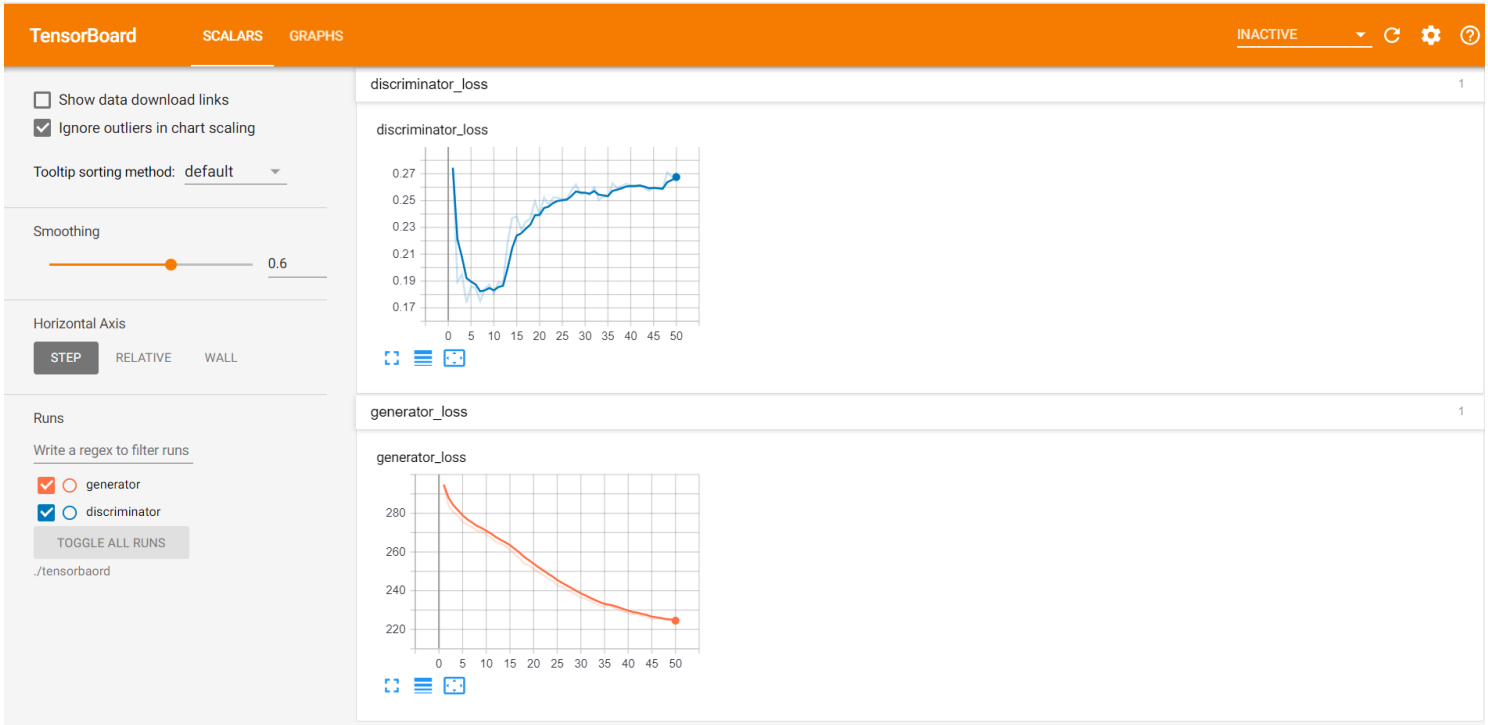
```
56
57          # saving model
58          save_model(i,g_model)
59
60      # Final summary
61      display.clear_output(True)
62      print('epoch>{}, {}/{}, d_real={:.3f}, d_fake={:.3f}, gan={:.3f}'.format(i+1, j+1, batch_per
63                                                                              d_loss_fake, gan_lo
64      summary_save_plot(i, g_model, sketch_path, seed_sketch, seed_color, seed_color.shape[0])
65
```

train.py hosted with ♡ by GitHub view raw

We are training on approx 100K sketch-color pairs. Total time taken for **50 epoch** with **batch size 32** is approz **25 hours** on my system with 16 GB RAM, 6GB NVIDIA GeForce GTX 1060.

12. Results

> *Tensorboard Graph*



somewhat learns the distribution then the loss is increasing, and that's what we want, to increase my overall discrimination loss.

Generator loss is decreasing all way, which shows that my generator is learning all the way, which is good.

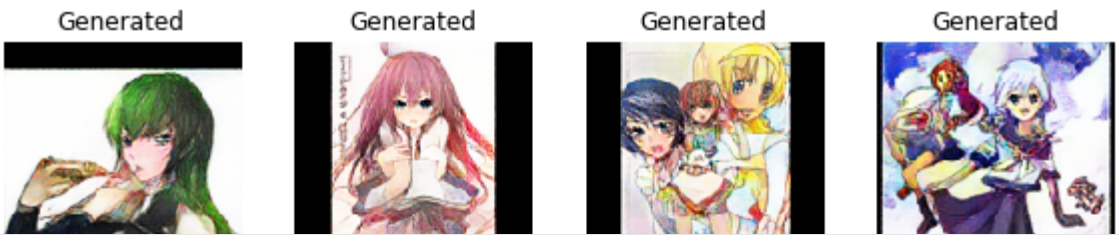
> *Train and test results*

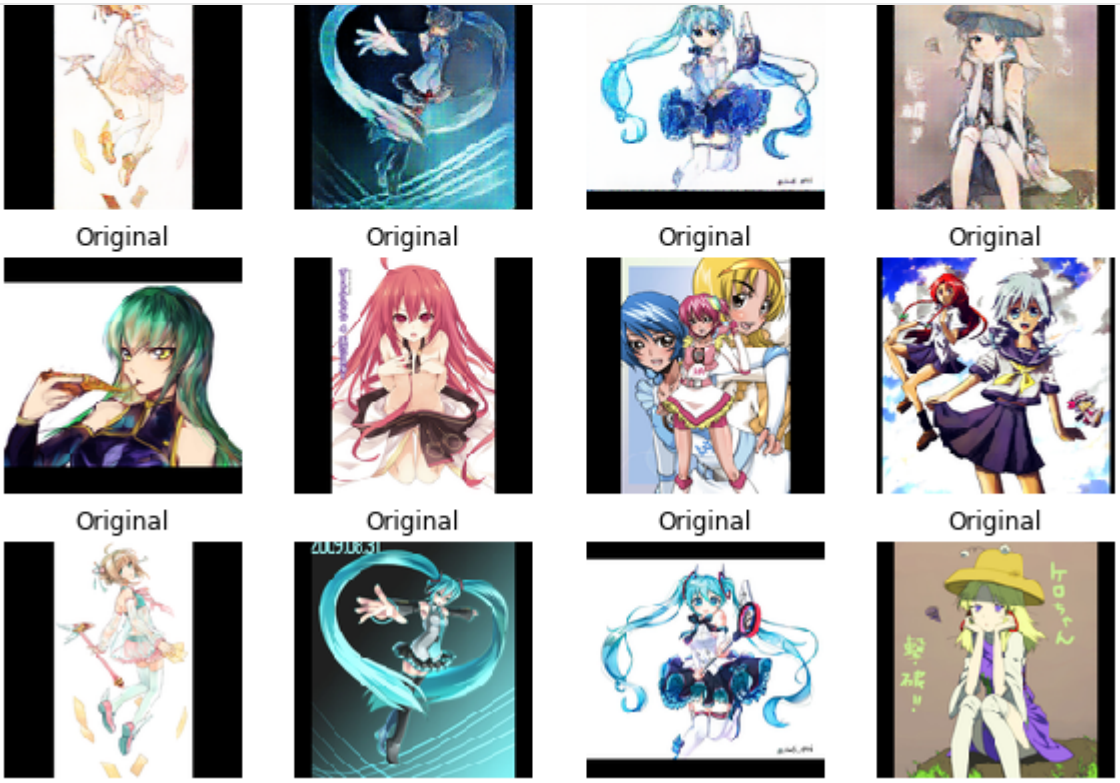
Now its time to check how good you are with image matching skills. Image with title **'Generated'** are fake images generated using generator and image with title **'Original'** is its corresponding real image. Let's have some work out for your eyes.

Results after different epochs from training set is:-

Result after 30 epochs.

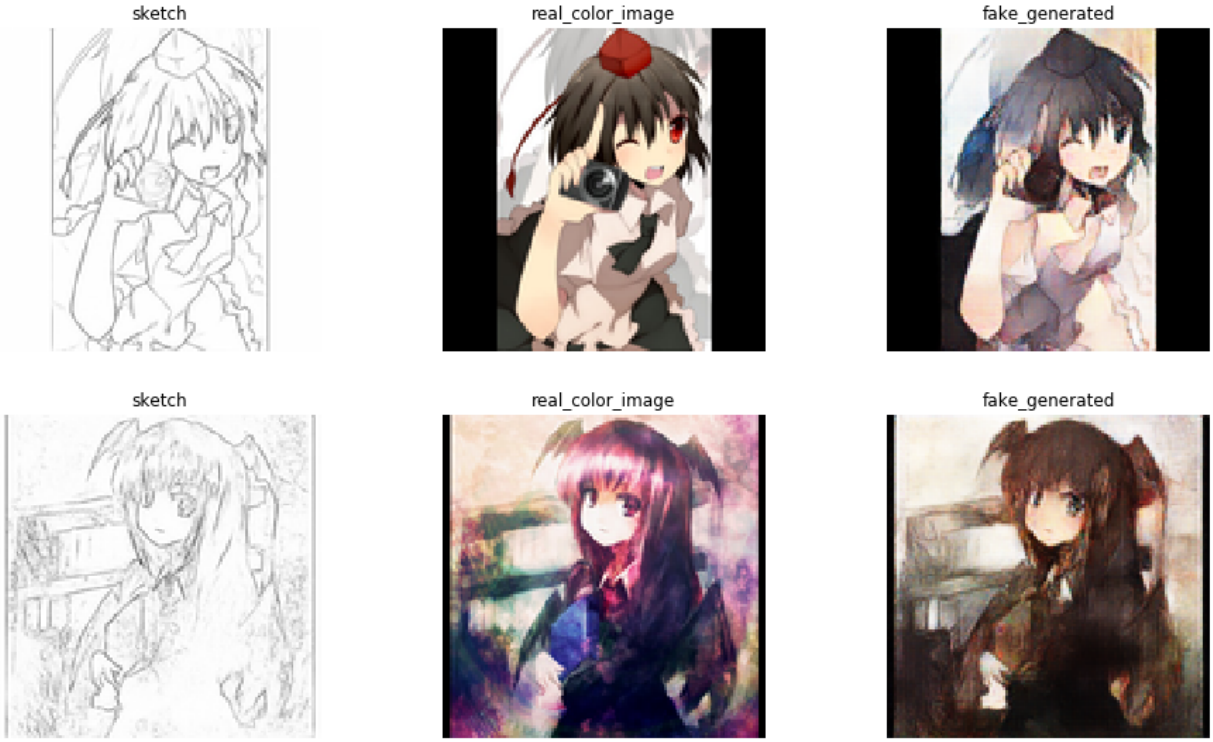
Try to observe the progress in learning as epoch increases.





Result after epoch 50

Now let's see results from the Unseen test set of data.



sketch

fake_generated

sketch

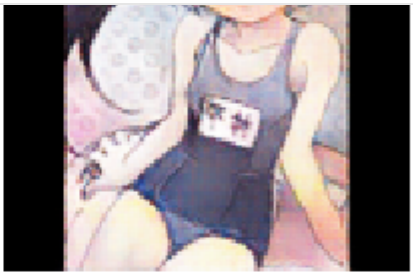
fake_generated

sketch

fake_generated

sketch

fake_generated



sketch

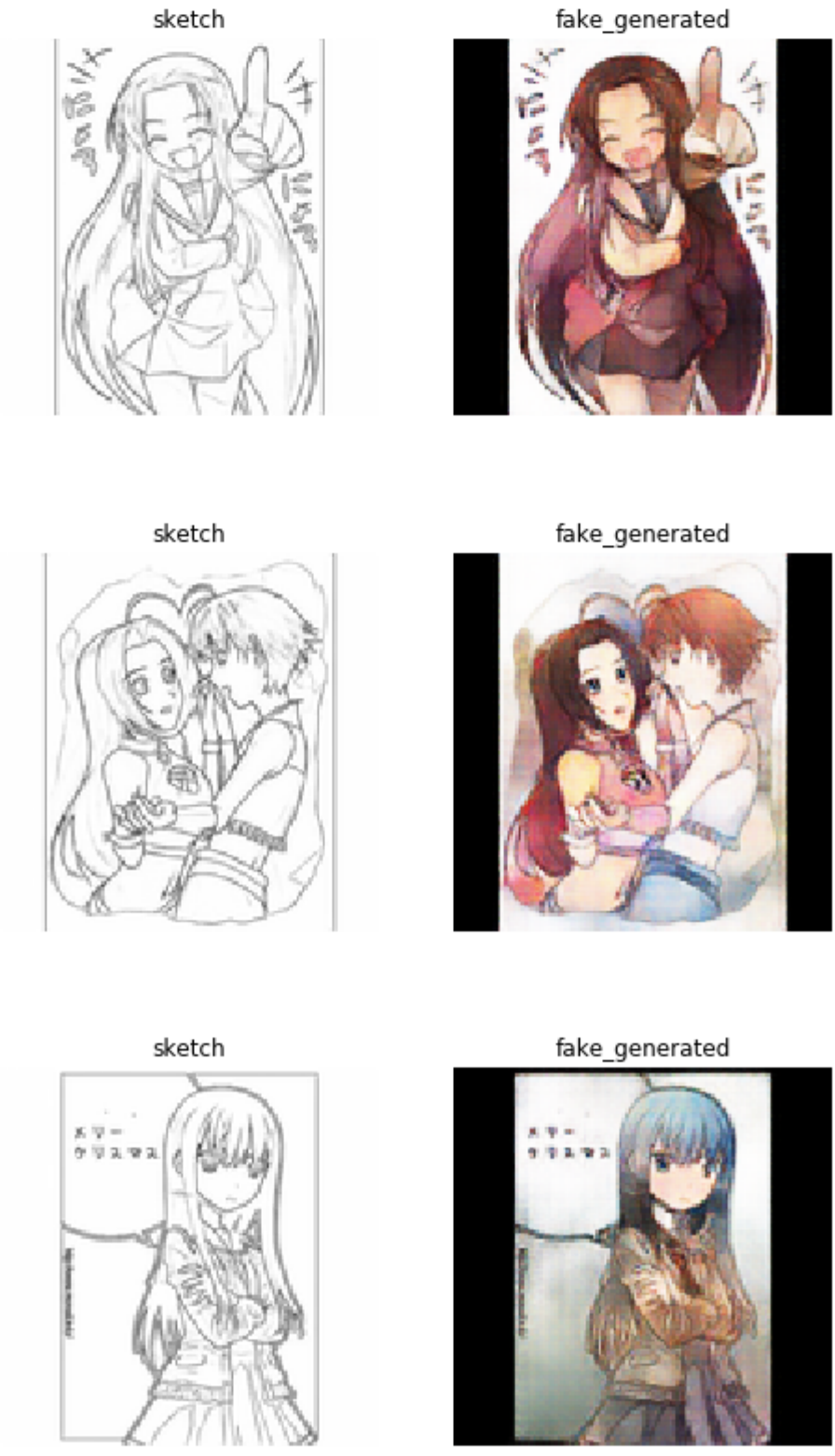
fake_generated

sketch

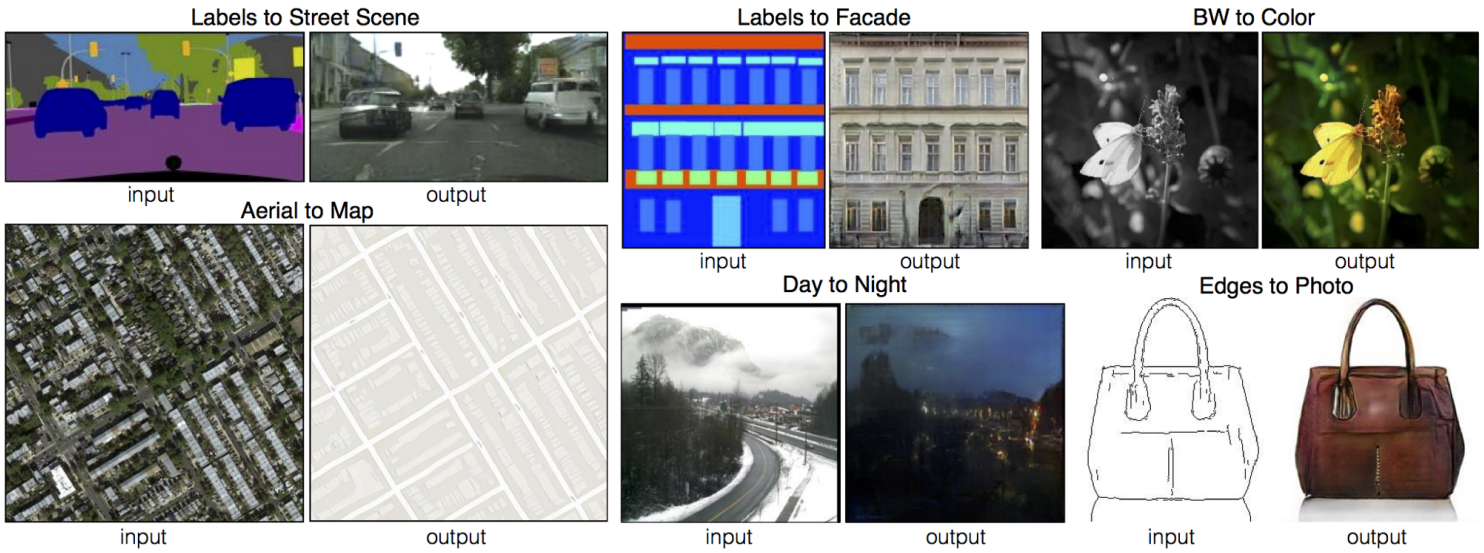
fake_generated

sketch

fake_generated



Result over randomly chosen image from the test set.



Source : Image-to-Image Translation with Conditional Adversarial Nets

Here input is the constraint image, and output is the desired result which is similar to ground truth image i.e. real image.

Get full ***code on github*** with detailed explanation in form of **jupyter notebook.**

Suggestions are welcome. Thanks a lot for reading… 💗

14. References

1. Appliedaicourse
2. Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, Alexei A. Efros : Image-to-Image Translation with Conditional Adversarial Networks
3. Yifan Liu, Zengchang Qin, Zhenbo Luo, Hua Wang : Auto-painter: Cartoon Image Generation from Sketch by Using Conditional Generative Adversarial Networks
4. Are GAN's unsupervised or supervised?
5. TensorBoard Profile: Profiling basic training metrics in Keras
6. https://phillipi.github.io/pix2pix/

You can reach me at:-

Github : https://github.com/ravi-1634063/

Gans Pix2pix Generative Adversarial Cnn Deep Learning