Ch 14.2 Pix2Pix Gan and Cycle Gan

Madhu Sanjeevi (Mady) Follow
Mar 15 · 10 min read

Welcome back to the chapter 14 GAN's series, this is the 3rd story connected to the previous 2 stories.

I hope you have gone through the last stories or you have already an idea about GAN's and it's types a bit.

In this story, I mainly wanna talk about different new ideas like Pix2Pix, CycleGAN's with it's Math and Training.

I wanna share the author's views from ground so you can train these for your problems or do a little more work around as part of your research.

Just a little

> *Recap from last stories*

→ Gans learn the distribution of the training data from random distribution

→ Discriminator is an X to Y mapping (Y → {0,1}), while Generator is a mapping from a fixed random noise vector to X (training data).

→ CGAN's take some vector as an input along with random noise at the Generator network.

$$L_D^{CGAN} = E\left[\log\left(D(x,c)\right)\right] + E\left[\log\left(1 - D(G(z),c\ )\right)\right]$$
$$L_G^{CGAN} = E\left[\log\left(D(G(z)\ ,c)\right)\right]$$

c is the condition vector

Okay I hope you are comfortable of understanding Gan's , lets start with

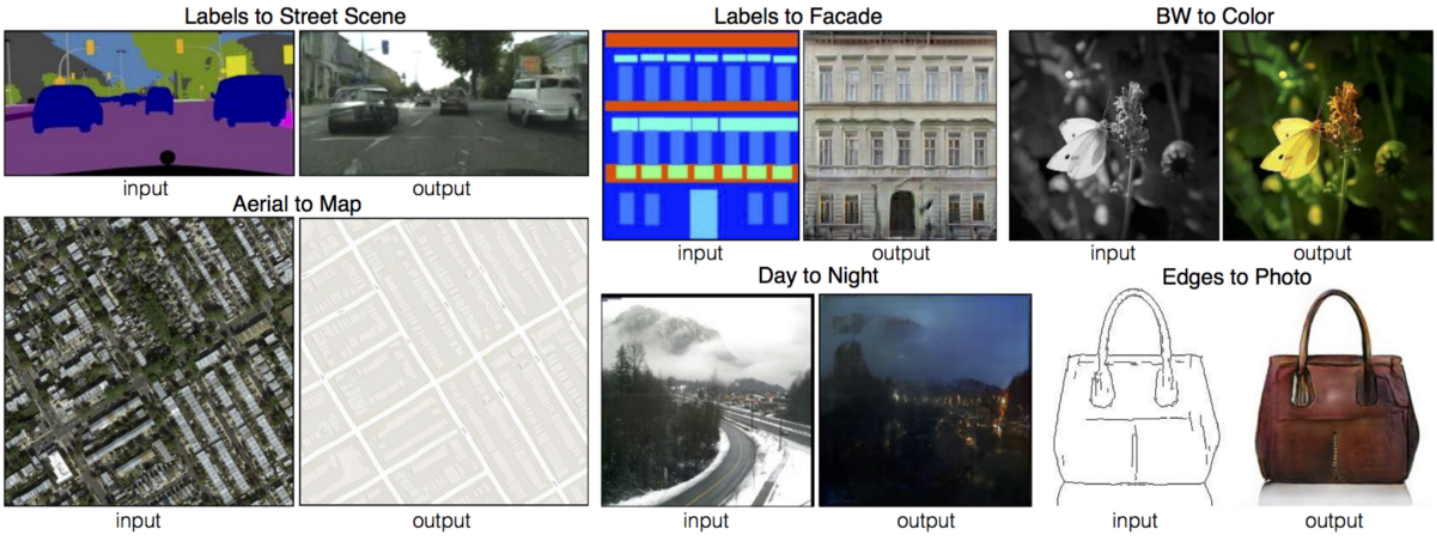
Pix2Pix

The name itself says "Pixel to Pixel" which means , in an image it takes a pixel, then convert that into another pixel.

The goal of this model is to convert from one image to another image, in other words the goal is to learn the mapping from an input image to an output image.

but why and what application we can think of ??

well, there are tons of applications we can think of



pix2pix

These are the main ones but you can think of a lot (litterly you can transform from one world to another world).

And the reason why we use GAN's for this is to synthesize these photos from one space to another.

Ingredients required

1. Training data pairs (**x** and **y** where **x** is the input image and **y** is the output image)
2. Pix2Pix uses the **conditional GAN (CGAN)** → G : {x, z} → y. (z → noise vector, x → input image, y → output image)
3. Generator Network (Encode- decode architecture) as an image is the input , we wanna learn the deep representation and decode it. and Discriminaor Network (PatchGAN #willdiscuss).
4. CGAN loss function and L1 or L2 distance.

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

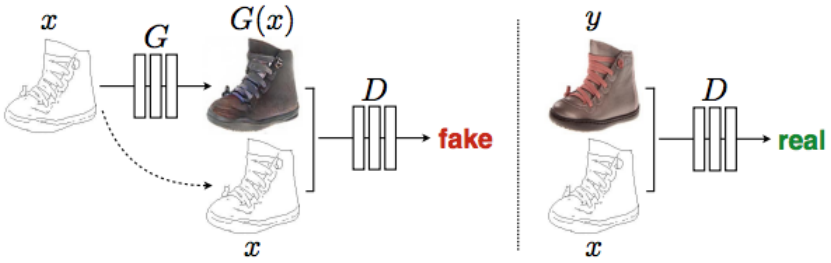
$$\mathcal{L}_{L1}(G) = \mathbb{E}_{x,y,z}[\|y - G(x,z)\|_1]. \qquad \mathcal{L}_{L_2}(G) = \mathbb{E}_{x,y,z}\left(y - G(x,z)\right)^2$$

Training Process

→ The Generator G takes x and z then it produces y, the goal of the G is to produce output images that cannot be distinguished from “real” images by the discriminator.

→ The discriminator D takes the pair (x and y) from both real images and fake images. The goal of the D is to distinguish between fake and real inputs.

This image illustrates that.



Here noise vector ignored

Training pix2pix gan as same as training any normal Gan except a little modification that is being done at the generator’s loss function.

The generator G is not only trying to reduce the loss from discriminator but also trying to move the fake distribution close to real distribution by using L1 or L2 loss

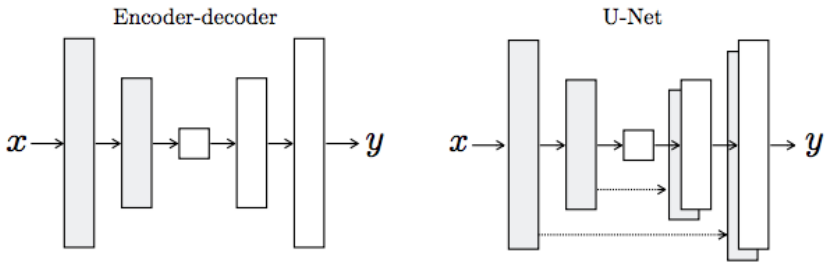
The loss fuction of generator network is

$$G^* = \arg\min_G \max_D \mathcal{L}_{cGAN}(G, D) + \lambda \mathcal{L}_{L1}(G).$$

Let’s talk about Networks Architectures.

As we know the generator is an encoder-decoder network (first a series of down sampling layers then we have bottle neck layer then a series of upsampling layers)

The authors used the “U-Net” architechure with skip connections as the E-D network.



The discriminator network uses the PatchGAN network (which was termed by the authors) ,

instead of predicting the whole image as fake or real at the discriminator, the model takes a N*N patch image and predicts every pixel in that patch if its real or fake.

That's the patchGAN.

because of every pixel giving a label, the pix2pix can produce sharp images with rich details in the images.

To know more about Network Params, Evaluation and Results , Please take a look at the paper.

Let me quickly explain the code w t r the changes.

You can find the full code here in my **github**.

```
x = tf.placeholder(tf.float32, shape=(None, img_size, img_size, channels))
y = tf.placeholder(tf.float32, shape=(None, img_size, img_size, channels))
```

Two place holders x and y as a pair (img_size is 256 and channels is 3)

```
def generator(x, isTrain=True, reuse=False):

    with tf.variable_scope('generator', reuse=reuse):

        # encoder
        conv1 = lrelu(tf.layers.conv2d(x, 64, [4, 4], strides=(2, 2), padding='same', kernel_initializer=w_init,
        conv2 = lrelu(tf.layers.batch_normalization(tf.layers.conv2d(conv1, 128, [4, 4], strides=(2, 2), padding
        conv3 = lrelu(tf.layers.batch_normalization(tf.layers.conv2d(conv2, 256, [4, 4], strides=(2, 2), padding
        conv4 = lrelu(tf.layers.batch_normalization(tf.layers.conv2d(conv3, 512, [4, 4], strides=(2, 2), padding
        conv5 = lrelu(tf.layers.batch_normalization(tf.layers.conv2d(conv4, 512, [4, 4], strides=(2, 2), padding
        conv6 = lrelu(tf.layers.batch_normalization(tf.layers.conv2d(conv5, 512, [4, 4], strides=(2, 2), padding
        conv7 = lrelu(tf.layers.batch_normalization(tf.layers.conv2d(conv6, 512, [4, 4], strides=(2, 2), padding
        conv8 = tf.nn.relu(tf.layers.conv2d(conv7, 512, [4, 4], strides=(2, 2), padding='same', kernel_initializ

        # decoder and skip connections
        deconv1 = tf.nn.dropout(tf.layers.batch_normalization(tf.layers.conv2d_transpose(conv8, 512, [4, 4], str
        deconv1 = tf.nn.relu(tf.concat([deconv1, conv7], 3))

        deconv2 = tf.nn.dropout(tf.layers.batch_normalization(tf.layers.conv2d_transpose(deconv1, 512, [4, 4], s
        deconv2 = tf.nn.relu(tf.concat([deconv2, conv6], 3))

        deconv3 = tf.nn.dropout(tf.layers.batch_normalization(tf.layers.conv2d_transpose(deconv2, 512, [4, 4], s
        deconv3 = tf.nn.relu(tf.concat([deconv3, conv5], 3))

        deconv4 = tf.layers.batch_normalization(tf.layers.conv2d_transpose(deconv3, 512, [4, 4], strides=(2, 2),
        deconv4 = tf.nn.relu(tf.concat([deconv4, conv4], 3))

        deconv5 = tf.layers.batch_normalization(tf.layers.conv2d_transpose(deconv4, 256, [4, 4], strides=(2, 2),
        deconv5 = tf.nn.relu(tf.concat([deconv5, conv3], 3))

        deconv6 = tf.layers.batch_normalization(tf.layers.conv2d_transpose(deconv5, 128, [4, 4], strides=(2, 2),
        deconv6 = tf.nn.relu(tf.concat([deconv6, conv2], 3))

        deconv7 = tf.layers.batch_normalization(tf.layers.conv2d_transpose(deconv6, 64, [4, 4], strides=(2, 2),
        deconv7 = tf.nn.relu(tf.concat([deconv7, conv1], 3))

        deconv8 = tf.nn.tanh(tf.layers.conv2d_transpose(deconv7, 3, [4, 4], strides=(2, 2), padding='same', kerr

        return deconv8
```

The generator network is based on the U-net model (a set of downsampling layers, then bottleneck layer, then a set of upsampling layers)

```
def discriminator(x, y, isTrain=True, reuse=False):

    with tf.variable_scope('discriminator', reuse=reuse):

        cat1 = tf.concat([x, y], 3)
        conv1 = lrelu(tf.layers.conv2d(cat1, 64, [4, 4], strides=(2, 2), padding='same', kernel_initializer=w_ir
        conv2 = lrelu(tf.layers.batch_normalization(tf.layers.conv2d(conv1, 128, [4, 4], strides=(2, 2), padding

        conv3 = tf.layers.batch_normalization(tf.layers.conv2d(conv2, 256, [4, 4], strides=(2, 2), padding='same
        conv3 = lrelu(tf.pad(conv3, [[0, 0], [1, 1], [1, 1], [0, 0]], mode="CONSTANT"))

        conv4 = tf.layers.batch_normalization(tf.layers.conv2d(conv3, 512, [4, 4], strides=(1, 1), padding='vali
        conv4 = lrelu(tf.pad(conv4, [[0, 0], [1, 1], [1, 1], [0, 0]], mode="CONSTANT"))

        conv5 = tf.layers.conv2d(conv4, 1, [4, 4], strides=(1, 1), padding='valid', kernel_initializer=w_init, b

        out = tf.nn.sigmoid(conv5) #gives 30*30 patchGAN

        return out, conv5
```

The discriminator is a simple network (a bunch of downsampling layers and at the end we get the patchGAN, a grid of 30*30 values or pixels where each pixel is classified how much fake or real (between 0 and 1).

Observe that, here we concatnate the x and y.

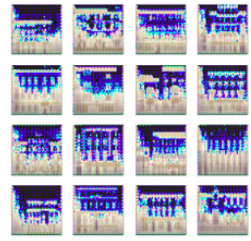
```
G_x = generator(x)
D_real_outputs, D_real_logits = discriminator(x, y)
D_fake_outputs, D_fake_logits = discriminator(x, G_x, reuse=True)


D_loss_real = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(logits=D_real_logits,
                                                                     labels=tf.ones_like(D_real_logits)))
D_loss_fake = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(logits=D_fake_logits,
                                                                     labels=tf.zeros_like(D_fake_logits)))
D_loss = (D_loss_real + D_loss_fake)

G_loss_gan = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(logits=D_fake_logits,
                                                                    labels=tf.ones_like(D_fake_logits)))
G_l1_loss = tf.reduce_mean(tf.reduce_sum(tf.abs(G_x - y), 3))

G_loss = G_loss_gan + G_l1_loss * l1_weight
```

So as usual, we feed the fake x to the discriminator along with the real x and calculate the D loss.

The G loss has now 2 components, one is the normal generator loss and another one is the weighted L1 loss between generated x (fake image) and real y.

The below snippet is for training

```
for it in range(200):

    epoch_start_time = time.time()

    for iter in range(train.shape[0] // batch_size):

        train_data = train.next_batch()

        train_x = norm(train_data[:, :, img_size:, :])
        train_y = norm(train_data[:, :, 0:img_size, :])

        _, D_loss_curr = sess.run([D_solver, D_loss], feed_dict={x: train_x, y: train_y})

        _, G_loss_curr = sess.run([G_solver, G_loss], feed_dict={x: train_x, y: train_y})
```



Epoch 1

```
Iter: 0
D loss: 0.3862
G_loss: 114.4

The total time for epoch0 is49.39186334609985
```

Note: I did not really bother about the results and did not fine tune the parameters , I just explained it to understand the topic, you can try the original author code to see the results.

Well, that's all about the pix2pix gan → Input pairs supervised, Simple GAN and Easy Process.

Alright! so we can do image-image translations when we have paired dataset, thats cool but …

> *some researchers are never satisifed*

Cycle GAN's

The same researchers came up with another idea later that year, they call

"Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks"

The outcome is → Given any two unordered image collections X and Y , the new algorithm learns to automatically "translate" an image from one into the other and vice versa.

in the above gif, not only the horse (x) can be mapped to a zebra(y) but also the zebra can be mapped back to the horse.

And this is achieved by unpaired training data unlike the above. (if pix2pix , we would have given the horse(x) and the zebra(y) as a pair)

This is cool because often we may lack of paired training data.

In a paired dataset, we give **x** and **y** as a pair such that they have some representation in common and they share some features.

so we make neural nets learn that mapping/understanding between the both images therotically and practically with the loss (L1).

but in an unpaired dataset ,there may no meaningful transformation that we can learn from x to y.

so the question is how can we make the net learn some mapping b/w X and Y and do the image to image translation.

Ok let's talk about how this is being done and the concepts.

> *Ingredients*

→ The training data consists of two different sets of images (based on the problem) One set of images called "**Domain X**", another set of images called "**Domain Y**".

eg: one set full of horses while another set full of zebras (no ordered pairing).

→ Two Generators with two different functions/processes (G(x), and F(G(x))) and Two Discriminators.

→GAN loss along with Cycle Consistent loss (new).

> *Key ideas and concepts*

Let's first talk about the model generators and discriminatiors,

- The first generator G(x) takes an input image x from **Domain X**, gives a generated image(output) y'

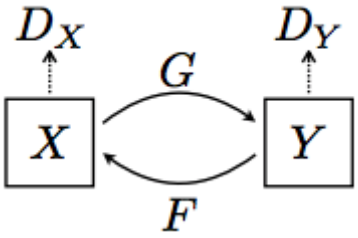
Eg: Horse to Zebra

G(x) → y' *(which should be indistinguishable from* ***Domain Y*** *images)*

- The second generator F(y) takes an input image y from **Domain Y**, gives a generated image x'

Eg: Zebra to Horse

F(y) →x' (*which should be indistinguishable from* ***Domain X*** *images)*



The **DX** and **DY** are the discriminators,

- The **DY** verifies the input image from **G(x)** if it looks like **Domain Y** images (Eg: is it same as zebras??)
- The **DX** verifies the input image from **F(y)** if it looks like **Domain X** images (Eg: is it same as horses??)

If I take an input image from Domain X → run the first generator → take that as the input and run the second generator → I expect to get the same image I started with

x → G(x) → y' → F(y') = x

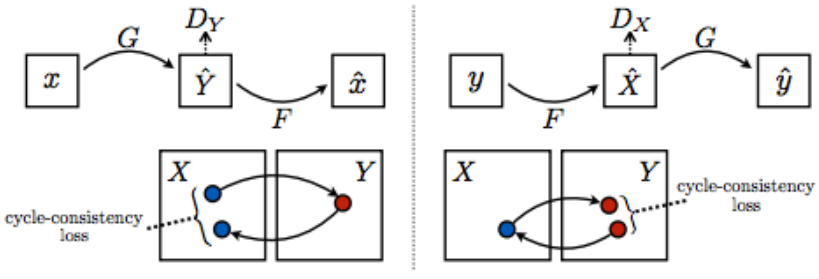
y → F(y) → x' → G(x') = y

x ∈ X and **y ∈ Y**

x' ≈ Y and **y' ≈ X**

F(G(x) = F(y') and **G(F(y)) = G(x')**

This intution is called **cycle consistent**.



Picture says it all.

Well, what I have explained here is only the idea or the "should be" case

Now let's talk about losses and training process.

Since we have two discriminators , we will have two normal GAN losses

x → G(x) → y' *(DY discriminator fake)* and **y** *(DY discriminator real)*

$$\begin{aligned}\mathcal{L}_{\text{GAN}}(G, D_Y, X, Y) =& \mathbb{E}_{y \sim p_{\text{data}}(y)}[\log D_Y(y)] \\ &+ \mathbb{E}_{x \sim p_{\text{data}}(x)}[\log(1 - D_Y(G(x))],\end{aligned} \tag{1}$$

y → F(y) → x' *(DX discriminator fake)* and *x (DX discriminator real)*

$$\begin{aligned}\mathcal{L}_{\text{GAN}}(G, D_X, X, Y) =& \mathbb{E}_{x \sim p_{\text{data}}(x)}[\log D_X(x)] \\ &+ \mathbb{E}_{y \sim p_{\text{data}}(y)}[\log(1 - D_X(G(y))],\end{aligned} \tag{2}$$

and as I mentioned above, it introduces a new loss called "Cycle loss"

which is just like, take an image x from Domain X and do this process

x → G(x) → y' → F(y') = x

you should get x back so we can simply calculate L1 loss between x and F(y') which is the cycle loss

same for **y → F(y) → x' → G(x') = y**

$$\begin{aligned}\mathcal{L}_{\text{cyc}}(G, F) =& \mathbb{E}_{x \sim p_{\text{data}}(x)}[\|F(G(x)) - x\|_1] \\ &+ \mathbb{E}_{y \sim p_{\text{data}}(y)}[\|G(F(y)) - y\|_1].\end{aligned} \tag{3}$$

The final loss = GAN loss + Cycle loss

$$\begin{aligned}\mathcal{L}(G, F, D_X, D_Y) =& \mathcal{L}_{\text{GAN}}(G, D_Y, X, Y) \\ &+ \mathcal{L}_{\text{GAN}}(F, D_X, Y, X) \\ &+ \lambda \mathcal{L}_{\text{cyc}}(G, F),\end{aligned} \tag{4}$$

here is the optimal objective function for cycle gan

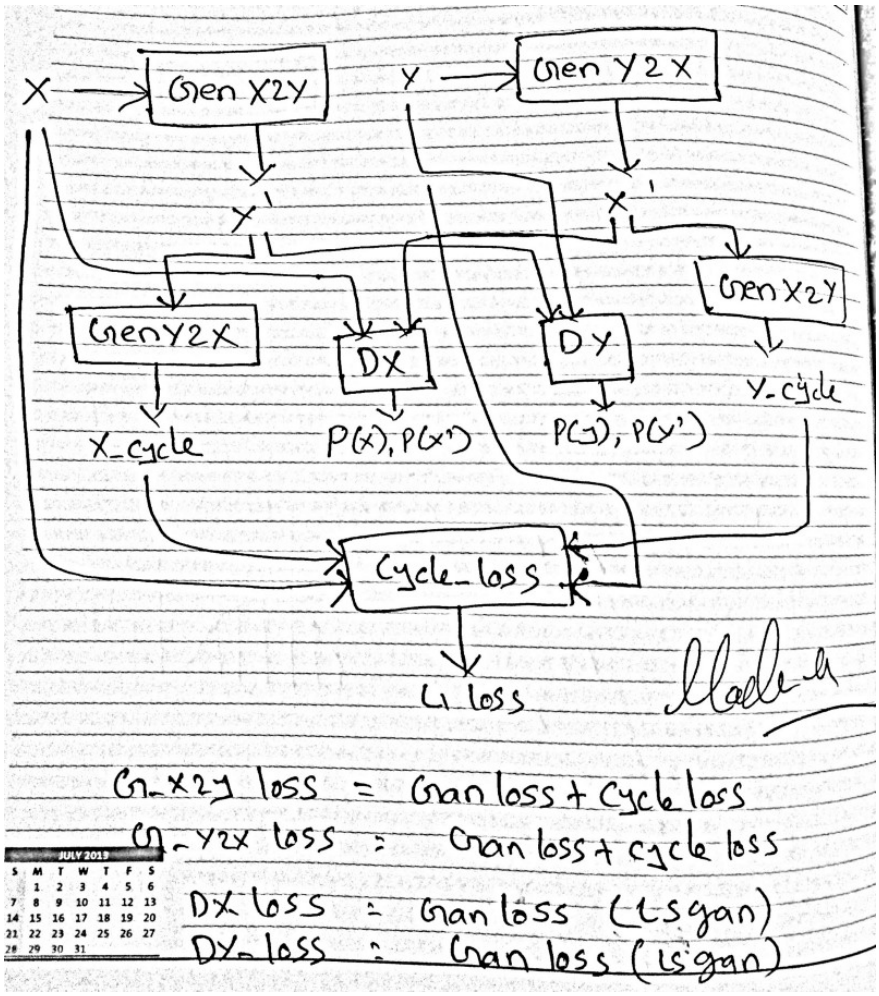
$$G^*, F^* = \arg\min_{G,F} \max_{D_x, D_Y} \mathcal{L}(G, F, D_X, D_Y). \quad (5)$$

so we train the model with this objective function,

in English,The training process is a 7 step process.

1. Take the 2 images x and y (1 from Domain X and 1 from Domain Y)
2. Run the two generators (x2y and y2x) → generate 2 fake images(y',x')
3. Run the two dicriminators (DX and DY) DX takes x and x' , DY takes y and y'
4. Calucalte the discriminators losses from above equations
5. Run again the two generators, x2y (x' as the input) and y2x (y' is the input) generates two cycle images (y_cycle, and x_cycle)
6. Calculate the cycle l1 loss from (x,x_cycle and y,y_cycle)
7. Finally calcualte the generator loss

Here is the picture drawn by me to help understand the process.



I am not going to explain the G and D networks architectures as I assume by now you can understand it by glancing the paper or code.

okay let me quickly explain the code w.r.t the changes

You can find the full code on my github **here**

```
y_fake = generator(x,scope='G_x2y') # x --> G(x) --> y'
x_fake = generator(y, scope='G_y2x') # y --> F(y) --> x'


#should be getting same as x and y
x_cycle = generator(y_fake,reuse=True,scope='G_y2x') # x --> G(x) --> y' --> F(y') = x
y_cycle = generator(x_fake,reuse=True,scope='G_x2y') # y --> F(y) --> x' --> G(x') = y
```

```
DX_real = discriminator(x,scope='DX')
DY_real = discriminator(y,scope='DY')
DX_fake = discriminator(x_fake, reuse=True,scope='DX')
DY_fake = discriminator(y_fake, reuse=True,scope='DY')
```

First we get two (y fake and x fake) images when run the same generator network with different scopes.

Then we give those outputs to the same generators scopes but switching the inputs to produce the cycle images.

Then we have the discriminators networks (fake x vs real x and fake y vs real y).

```
# Discriminator loss (we use LSGAN loss)
Gan_loss_DX = (tf.reduce_mean(tf.square(DX_real - tf.ones_like(DX_real))) +
               tf.reduce_mean(tf.square(DX_fake)))/ 2.0
Gan_loss_DY = (tf.reduce_mean(tf.square(DY_real - tf.ones_like(DY_real))) +
               tf.reduce_mean(tf.square(DY_fake))) / 2.0
```

```
# Cycle consistent loss
l1 = 10
cycle_loss = tf.reduce_mean(l1 * tf.abs(x - x_cycle)) + tf.reduce_mean(l1 * tf.abs(y - y_cycle))
```

```
# Generator loss
Gan_loss_GX = tf.reduce_mean(tf.square(DY_fake - tf.ones_like(DY_fake))) + cycle_loss
Gan_loss_FY = tf.reduce_mean(tf.square(DX_fake - tf.ones_like(DX_fake))) + cycle_loss
```

Losses as explained above (observe we use LSGAN loss function, to understand it you can refer my previous story **here**)

```
DX_vars = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES, 'DX')
DY_vars = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES, 'DY')
G_x2y_vars = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES, 'G_x2y')
G_y2x_vars = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES, 'G_y2x')
```

```
DX_solver = tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(Gan_loss_DX, var_list=DX_vars)
DY_solver = tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(Gan_loss_DY, var_list=DY_vars)
G_x2y_solver = tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(Gan_loss_GX, var_list=G_x2y_vars)
G_y2x_solver = tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(Gan_loss_FY, var_list=G_y2x_vars)
```

```
for it in range(20):

    epoch_start_time = time.time()

    for iter in range(len(trainA) // batch_size):

        batch_start_time = time.time()

        X = trainA[iter*batch_size:(iter+1)*batch_size]
        Y = trainB[iter*batch_size:(iter+1)*batch_size]

        X_fake, Y_fake = sess.run([x_fake,y_fake],feed_dict={x: X,y: Y})

        # Optimize G Networks
        _,loss_G_x2y = sess.run([Gan_loss_GX,G_x2y_solver], feed_dict={x:X,y:Y})
        _,loss_G_y2x = sess.run([Gan_loss_FY,G_y2x_solver], feed_dict={x:X,y:Y})

        # Optimize D Networks
        _,loss_DY = sess.run([Gan_loss_DY,DY_solver],feed_dict={x: X,y: Y})
        _,loss_DX = sess.run([Gan_loss_DX,DX_solver],feed_dict={x: X,y: Y})

    epoch_end_time = time.time()
    per_epoch_ptime = epoch_end_time - epoch_start_time
    print("The total time for epoch{0} is{1} ".format(it,per_epoch_ptime))
```

This is the training script to train the generators and discriminators

That's pretty much about the cycle GAN.

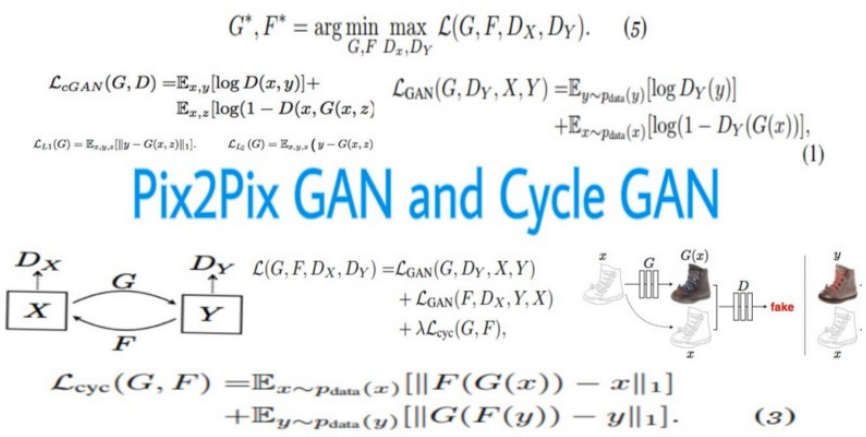
> ***Summary***

→ Image to Image translation is what we have been focused on.

→Pix2Pix takes pairs of images (X and Y) to be able to learn the translation from one image X to another Y.

→Cycle GAN don't require the pair of images as inputs, it does the learning from taking the images from one Domain (A) and produces

the another domain images in a way that there should be the cycle representation.



Overview

If anyone has doubts/thoughts/suggestions, feel free to ask and if i can help I will definitely help.

> ***Original Papers and Credit to the Authors***

Image-to-Image Translation with Conditional Adversarial Networks

Cycle GAN

Tensorflow Cycle GAN