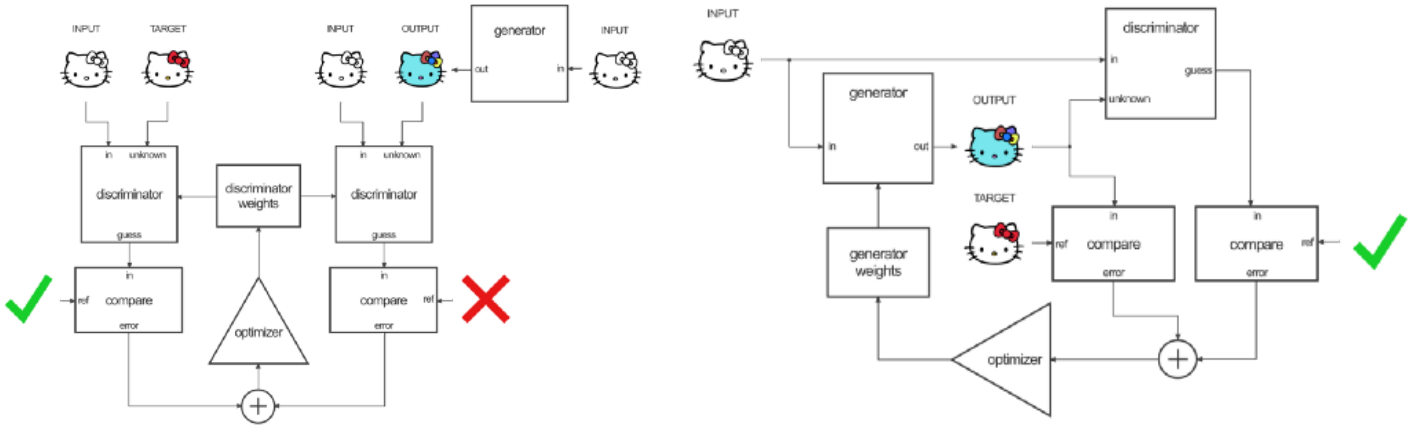


Discriminator and Generator training

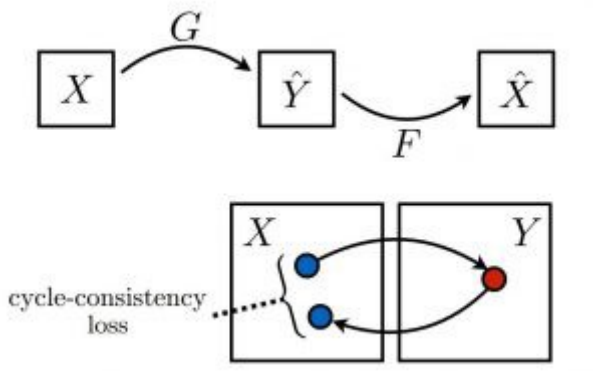
CycleGANs

Original CycleGAN paper

While PIX2PIX can produce truly magical results, the challenge is in training data. The two image spaces that you wanted to learn to translate between needed to be pre-formatted into a single X/Y image that held both tightly-correlated images. This could be time-consuming, infeasible, or even impossible based on what two image types you were trying to translate between (for instance, if you didn't have one-to-one matches between the two image profiles). This is where the CycleGAN comes in.

The key idea behind CycleGANs is that they can build upon the power of the PIX2PIX architecture, but allow you to point the model at two discrete, *unpaired collections* of images. For example, one collection of images, Group X, would be full of sunny beach photos while Group Y would be a collection of overcast beach photos. The CycleGAN model can learn to translate the images between these two aesthetics without the need to merge tightly correlated matches together into a single X/Y training image.

The way CycleGANs are able to learn such great translations without having explicit X/Y training images involves introducing the idea of a *full translation cycle* to determine how good the entire translation system is, thus improving both generators at the same time.



This approach is the clever power that CycleGANs brings to image-to-image translations and how it enables better translations among non-paired image styles.

The original CycleGANs paper, “Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks”, was published by Jun-Yan Zhu, et al.

Loss functions

The power of CycleGANs is in how they set up the loss function, and use the full cycle loss as an additional optimization target.

As a refresher: we’re dealing with 2 generators and 2 discriminators.

Generator Loss

Let’s start with the generator’s loss functions, which consist of 2 parts.

Part 1: The generator is successful if fake (generated) images are so good that discriminator can not distinguish those from real images. In other words, the discriminator’s output for fake images should be as close to 1 as possible. In TensorFlow terms, the generator would like to minimize:

```
g_loss_G_disc = tf.reduce_mean((discY_fake —
tf.ones_like(discY_fake)) ** 2)
```

```
g_loss_F_dicr = tf.reduce_mean((discX_fake —
tf.ones_like(discX_fake)) ** 2)
```

Note: the “**” symbol above is the power operator in Python.

Part 2: We need to capture **cyclic loss**: as we go from one generator back to the original space of images using another generator, the difference between the original image (where we started the cycle) and the cyclic image should be minimized.

```
g_loss_G_cycle = tf.reduce_mean(tf.abs(real_X – genF_back))
+ tf.reduce_mean(tf.abs(real_Y – genG_back))
```

```
g_loss_F_cycle = tf.reduce_mean(tf.abs(real_X – genF_back))
+ tf.reduce_mean(tf.abs(real_Y – genG_back))
```

Finally, the generator loss is the sum of these two terms:

g_loss_G = g_loss_G_disc + g_loss_G_cycle

Because cyclic loss is so important we want to multiply its effect. We used an L1_lambda constant for this multiplier (in the paper the value 10 was used).

Now the grand finale of the generator loss looks like:

g_loss_G = g_loss_G_disc + L1_lambda * g_loss_G_cycle

g_loss_F = g_loss_F_disc + L1_lambda * g_loss_F_cycle

Discriminator Loss

The Discriminator has 2 decisions to make:

1. Real images should be marked as real (recommendation should be as close to 1 as possible)
2. The discriminator should be able to recognize generated images and thus predict 0 for fake images.

```
DY_loss_real = tf.reduce_mean((DY – tf.ones_like(DY))** 2)
```

```
DY_loss_fake = tf.reduce_mean((DY_fake_sample –
tf.zeros_like(DY_fake_sample)) ** 2)
```

```
DY_loss = (DY_loss_real + DY_loss_fake) / 2
```

```
DX_loss_real = tf.reduce_mean((DX – tf.ones_like(DX)) ** 2)
```

```
DX_loss_fake = tf.reduce_mean((DX_fake_sample —
tf.zeros_like(DX_fake_sample)) ** 2)
```

```
DX_loss = (DX_loss_real + DX_loss_fake) / 2
```