Weather Wizard: transforming weather conditions of a landscape

By Anpandoh

Source: https://medium.com/@anpandoh/weather-wizard-transforming-weather-conditions-of-a-landscape-1e70c4966660

By Aneesh Pandoh → mailto:Pandoh@wisc.edu, Harrison Smith → mailto:smithht@wofford.edu, Ethan Xi → mailto:eyxi@wisc.edu,

Intro

The weather is constantly changing and something that we as humans cannot control. It would be great if you could visually see what a specific area or view may look like under different weather conditions, weather (amazing pun) it's snow, rain, sun, or something as crazy and made up as a dystopian scene. Through the use of neural networks, our project will attempt to transform the weather conditions of any given landscape.

Our Neural Network

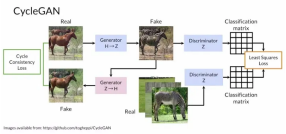
Credits: CycleGAN paper → https://arxiv.org/pdf/1703.10593.pdf, Tensorflow CycleGAN → https://www.tensorflow.org/tutorials/generative/cyclegan, ResNET Youtube Video → https://www.youtube.com/watch?v=VzIO5_R9XEM&ab_channel=DigitalSreeni

What is CycleGAN and why did we choose it? Why is it better suited than other NN such as ConditionalGAN?

The neural network that we will be implementing to perform the task of transforming weather conditions of a landscape is called CycleGAN. CycleGAN is a deep learning model that can learn to transform images from one domain to another and back again. But what makes CycleGAN better than other, more traditional GAN models? Typically, GAN models such as ConditionalGAN need paired data to train the model. For example, a set of images, and then a paired set of the same images with, in our case, rain, snow, or whatever weather condition desired. Paired datasets are often hard to obtain and find. However, with a CycleGAN we can train our model without the need for paired images. This allows us to more easily collect datasets and gives us a larger amount of possible image transformations to implement.

How do CycleGANs work?

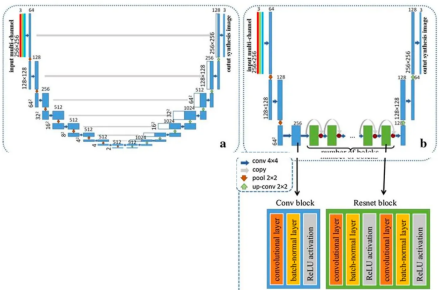
The CycleGAN model consists of two generators and two discriminators. During training, the generators try to produce images that can fool the discriminators, while the discriminators would try to distinguish between real and fake images. The cycle consistency loss would encourage the generators to produce images that, when translated back and forth between the two domains, are as close as possible to the original ones. For example, in our example of transforming daytime/sunny images to be snowy, we would train the model on a dataset of day-time images, and snowy images. The model would then learn to transform day-time images to snowy images and vice versa, by passing them through the generators. Once the model is trained, we can use it to transform new sunny images into snowy images, by passing them through the generator. The generator will produce a snowy version of the image, which should be as similar as possible to a real snowy landscape image. This is a good image of how they work:



Generators and Discriminators:

The generators and discriminators in the CycleGAN for our model are modified resnet generators, with the discriminators to go with them.

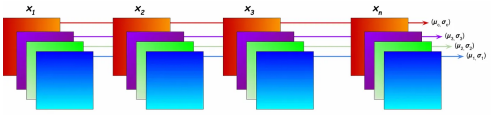
The tutorial on Tensorflow for cycleGANs used a Unet generator for simplicity but, suggested, similar to the paper, to use a resnet generator. Both of these generator types use convolutional layers with instance normalization and a ReLu activation layer to downsample(encoder) and downsample(decoder) the images, but the resnet generator includes resnet blocks that use 2 similar convolution blocks that do not change the amount of filters or size of kernel in conv2D layer once they are downsampled.In our case we included 9 resnet blocks to get better results.



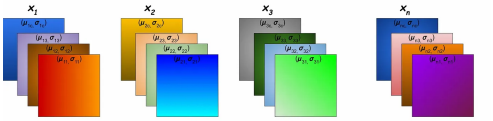
It is also important to note that we are using instance normalization instead of batch normalization which are used in traditional pix2pix models, the difference being that batch normalization calculates the

mean and variance *across all samples and both spatial dimensions* (x,y) while instance normalization calculates the same thing for *each individual sample across both spatial dimensions*.

Batch:



Instance:



Implementation:

```
# Resnet blocks to be used in the generator
# residual block that contains two 3 × 3 convolution-
al layers with the same number of filters on both
layers.
def resnet_block(n_filters, input_layer):
  print(input_layer)
 # weight initialization
  init = RandomNormal(stddev=0.02)
 # first convolutional layer
  g = Conv2D(n_filters, (3,3), padding='same', ker-
nel_initializer=init)(input_layer)
  g = InstanceNormalization(axis=-1)(g)
  g = Activation('relu')(g)
 # second convolutional layer
  g = Conv2D(n_filters, (3,3), padding='same', ker-
nel_initializer=init)(g)
  g = InstanceNormalization(axis=-1)(g)
 # concatenate merge channel-wise with input layer
  g = Concatenate()([g, input_layer])
  return g
#Generator
def define_generator(image_shape, n_resnet=9):
 # weight initialization
 init = RandomNormal(stddev=0.02)
 # image input
 in_image = Input(shape=image_shape)
 # 64 filters, 49 size kernel
  g = Conv2D(64, (7,7), padding='same', kernel_ini-
tializer=init)(in_image)
 g = InstanceNormalization(axis=-1)(g)
 g = Activation('relu')(g)
 # 128 filters , 9 size kernel
    g    =    Conv2D(128,    (3,3),    strides=(2,2),
padding='same', kernel_initializer=init)(g)
 g = InstanceNormalization(axis=-1)(g)
```

```
 g = Activation('relu')(g)
 # 256  filters, 9 size kernel
    g    =    Conv2D(256,    (3,3),    strides=(2,2),
padding='same', kernel_initializer=init)(g)
 g = InstanceNormalization(axis=-1)(g)
 g = Activation('relu')(g)
 # Resnet at 256 filters
 for _ in range(n_resnet):
  g = resnet_block(256, g)
 # 128  filters, 9 size kernel
 g = Conv2DTranspose(128, (3,3), strides=(2,2), pad-
ding='same', kernel_initializer=init)(g)
 g = InstanceNormalization(axis=-1)(g)
 g = Activation('relu')(g)
 # 64 filters
 g = Conv2DTranspose(64, (3,3), strides=(2,2), pad-
ding='same', kernel_initializer=init)(g)
 g = InstanceNormalization(axis=-1)(g)
 g = Activation('relu')(g)
 # 3  filters
 g = Conv2D(3, (7,7), padding='same', kernel_initial-
izer=init)(g)
 g = InstanceNormalization(axis=-1)(g)
 out_image = Activation('tanh')(g)
 # define model
 model = Model(in_image, out_image)
 return model
```

The discriminator is pretty straightforward as it uses the same block style with a convolutional layer, instance normalization, and ReLu, in order to classify if an image patch from the generator output or from the training data is real or not real.

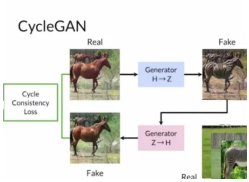
Implementation:

```
#After the last layer, conv to 1-dimensional output,
followed by a Sigmoid function.
def define_discriminator(image_shape):
 # weight initialization
 init = RandomNormal(stddev=0.02)
 # source image input
 in_image = Input(shape=image_shape)
 # 64 filters, 16 size kernel
 d = Conv2D(64, (4,4), strides=(2,2), padding='same',
kernel_initializer=init)(in_image)
 d = LeakyReLU(alpha=0.2)(d)
 # 128 filters 16 size kernel
 d = Conv2D(128, (4,4), strides=(2,2),
padding='same', kernel_initializer=init)(d)
 d = InstanceNormalization(axis=-1)(d)
 d = LeakyReLU(alpha=0.2)(d)
 # 256 filter 16 size
 d = Conv2D(256, (4,4), strides=(2,2),
padding='same', kernel_initializer=init)(d)
 d = InstanceNormalization(axis=-1)(d)
 d = LeakyReLU(alpha=0.2)(d)
 # 512 filters 16 size kernel
 d = Conv2D(512, (4,4), padding='same', kernel_ini-
tializer=init)(d)
 d = InstanceNormalization(axis=-1)(d)
 d = LeakyReLU(alpha=0.2)(d)
 # patch output
 patch_out = Conv2D(1, (4,4), padding='same', ker-
nel_initializer=init)(d)
 # define model
 model = Model(in_image, patch_out)
 # compile model
 model.compile(loss='mse', optimizer=Adam(lr=0.0002,
beta_1=0.5), loss_weights=[0.5])
 return model
```

Cycle Loss

The main factor that differentiates CycleGANs from regular GANs besides the generator and discriminator architecture is cycle consistency loss

Cycle Consistency: After a cycle of generating a fake, snowy image from the original real sunny image, we then pass it back through the other generator to get something close to the original image. We then compare the 2 using mean absolute error to get loss.



> For example, if one translates a sentence from English to French, and then translates it back from French to English, then the resulting sentence should be the same as the original sentence. — Tensorflow people.

Identity Loss

If trying to go from a sunny image to a snowy image, and a snowy image is inputted, the generated image should not be drastically changed.

Similar to the identity property for addition (+0) and multiplication (x1) where the input should not be changed.

Generator Loss

Sum of real loss and generated loss.

Real loss: sigmoid cross entropy loss of real images compared against an array of 1's

Generated loss: sigmoid cross entropy loss of generated images compared against an array of 0's.

Discriminator Loss

Sum of Gan loss, L1 loss, and lambda

Gan loss: sigmoid cross entropy loss of generated images and an array of 1's

L1 loss: mean absolute error between generated image and the target image.

lambda = 10 in our project

Data Collection and Preprocessing:

The data set(s) for a CycleGAN model is really up to the user, since the desired type of image translation is in their hands. For our specific runs, we had the goal of transforming a sunny landscape into snowy, and vice versa. Therefore, our data → https://www.cs.ccu.edu.tw/~wtchu/projects/Weather/index.html consists of 1400 sunny images, and 1200 snowy images. To get the images ready for training, we reshaped the images to be 256x256 pixels and we added some random mirroring and normalized from [-1,1].

Training:

The process of training our data consists of four basic steps:

1. Get the predictions
2. Calculate the loss
3. Calculated the gradients using backpropagation
4. Apply the gradient to the optimizer

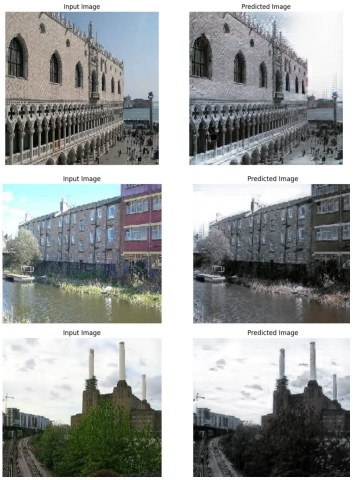
```
def train_step(real_x, real_y):
  # persistent is set to True because the tape is
used more than
  # once to calculate the gradients.
  with tf.GradientTape(persistent=True) as tape:
    # Generator G translates X -> Y
    # Generator F translates Y -> X.
        fake_y = generator_g(real_x, training=True)
    cycled_x = generator_f(fake_y, training=True)
       fake_x = generator_f(real_y, training=True)
    cycled_y = generator_g(fake_x, training=True)
         # same_x and same_y are used for identity
loss.
    same_x = generator_f(real_x, training=True)
    same_y = generator_g(real_y, training=True)
        disc_real_x = discriminator_x(real_x, train-
ing=True)
       disc_real_y  =  discriminator_y(real_y,  train-
ing=True)
        disc_fake_x = discriminator_x(fake_x, train-
ing=True)
       disc_fake_y  =  discriminator_y(fake_y,  train-
ing=True)
       # calculate the loss
    gen_g_loss = generator_loss(disc_fake_y)
    gen_f_loss = generator_loss(disc_fake_x)
              total_cycle_loss = calc_cycle_loss(re-
al_x, cycled_x) + calc_cycle_loss(real_y, cycled_y)
            # Total generator loss = adversarial loss
+ cycle loss
     total_gen_g_loss = gen_g_loss + total_cycle_loss
+ identity_loss(real_y, same_y)
     total_gen_f_loss = gen_f_loss + total_cycle_loss
+ identity_loss(real_x, same_x)
```

```
        disc_x_loss = discriminator_loss(disc_real_x,
disc_fake_x)
       disc_y_loss  =  discriminator_loss(disc_real_y,
disc_fake_y)
        # Calculate the gradients for generator and
discriminator
     generator_g_gradients   =   tape.gradient(total_-
gen_g_loss,
                                               generator_g.-
trainable_variables)
     generator_f_gradients   =   tape.gradient(total_-
gen_f_loss,
                                               generator_f.-
trainable_variables)
            discriminator_x_gradients  =  tape.gradi-
ent(disc_x_loss,
                                                 discrimi-
nator_x.trainable_variables)
     discriminator_y_gradients  =  tape.gradient(dis-
c_y_loss,
                                                 discrimi-
nator_y.trainable_variables)
       # Apply the gradients to the optimizer
    generator_g_optimizer.apply_gradients(zip(genera-
tor_g_gradients,
                                                  genera-
tor_g.trainable_variables))
       generator_f_optimizer.apply_gradients(zip(gen-
erator_f_gradients,
                                                  genera-
tor_f.trainable_variables))
               discriminator_x_optimizer.apply_gradi-
ents(zip(discriminator_x_gradients,
                                                    dis-
criminator_x.trainable_variables))
```

```
                discriminator_y_optimizer.apply_gradi-
ents(zip(discriminator_y_gradients,
                                                dis-
criminator_y.trainable_variables))
```

Results:

Sunny to Snowy



Snowy to Sunny

Overall, we are pretty pleased with how our model turned out. A high point that stood out to us included how our model would sometimes generate an actual sun when transforming an image into a sunny image. Similarly, it would also generate snow when asked to transform an image into a snowy image. However, if we had more time, there are certainly some areas we would look to improve. The first would be finding a better dataset. When transforming to snowy images, we noticed a lot of our predicted images turned out dark and almost looked black and white. Some of the more poor results:

This is likely due to the fact that a lot of the snowy images from our dataset are originally dark and almost seem black and white. Additionally, we would have loved to have more time to simply train more data, and produce more accurate results. We also had the idea to transform the landscape to look more dystopian or Martian in the future. This model has many applications due to the fact that the data can be acquired so easily.

Our Code:

Snowy To Sunny → https://colab.research.google.com/drive/1PVjBZv7UCEsg0GqvLduFFq_QG5Ha9RYU#scrollTo=0FMYgY_mPfTi

Sunny to Snowy → https://colab.research.google.com/drive/1ZX5hKUKghBQbVc_AC26oYK1hY30NtGGP#scrollTo=35vwbCLfK-z5