

# CycleGAN Implementation for Photograph to Monet-style Painting Transformation

*“Every artist dips his brush in his own soul and paints his own nature into his pictures”*

**San Jose State University**

**CMPE257 - Spring 2021**

**Group 11:**

Hashmitha Katta

Hisaam Hashim

Manasa Bobba

Poorna Srinivas Gutta

Rohit Sai Chowdary Potluri

# 1. Introduction

For our project we have taken an active Kaggle competition titled “I’m Something of a Painter Myself”. The objective of this competition is to generate paintings in the style of Monet. The intent is to create a model that can generate a Monet style painting from a photograph.

We identify the works of artists through their unique style, such as color choices or brush strokes. The “je ne sais quoi” of artists like Claude Monet can now be imitated with algorithms thanks to generative adversarial networks (GANs). Generative adversarial networks is a deep learning technique in which two models, a generative model and a discriminative model, are trained simultaneously. The generator tries to mimic examples from a training dataset. The discriminator receives a sample, but it does not know where the sample comes from. Its role is to predict whether it is a data sample or a synthetic sample. The discriminator is trained to make accurate predictions, and the generator is trained to output samples that fool the discriminator into thinking they came from the data distribution.

GANs are generally used for unsupervised machine learning tasks. GANs are applied for modelling natural images. CycleGAN is an image to image translation model. The model learns mapping between input and output images using unpaired dataset. CycleGAN uses a cycle consistency loss to enable training without the need for paired data. In other words, it can translate from one domain to another without a one-to-one mapping between the source and target domain. We need to convert images from one domain to another domain using unpaired images hence, we are using CycleGAN for this problem statement.

## 2. Literature Review / Related Work

### **Tutorial Notebook**

A tutorial was provided in the ‘description’ section of the kaggle competition that provided basic concepts like loading data from TFRecords, using TPUs etc.

### **Generative Adversarial Nets**

*Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, Yoshua Bengio*

This paper introduced the concept of Adversarial Nets and proposed a new framework for estimating generative models via an adversarial process, in which two models are simultaneously trained : a generative model  $G$  that captures the data distribution, and a discriminative model  $D$  that estimates the probability that a sample came from the training data rather than  $G$ .

<https://arxiv.org/pdf/1406.2661>

### **Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks**

*Jun-Yan Zhu, Taesung Park, Phillip Isola, Alexei A. Efros*

This paper is the one we followed to implement the architecture of CycleGAN. This paper is written to address the problem of unpaired image-to-image translation. The goal of this paper is to learn a mapping  $G : X \rightarrow Y$  such that the distribution of images from  $G(X)$  is indistinguishable from the distribution  $Y$  using an adversarial loss. Because this mapping is highly under-constrained, they coupled it with an inverse mapping  $F : Y \rightarrow X$  and introduce a cycle consistency loss to push  $F(G(X)) \sim X$  ( and vice-versa ). This method works on several tasks where paired training data does not exist, including collection style transfer, object transfiguration, season transfer, photo enhancement etc.

<https://arxiv.org/pdf/1703.10593>

### **Image-to-Image Translation with Conditional Adversarial Networks**

*Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, Alexei A. Efros*

This paper investigates conditional adversarial networks as a general-purpose solution to image-to-image translation problems. These networks not only learn the mapping from input image to output image, but also learn a loss function to train this mapping. This paper focuses on paired image-to-image translation (our project is unpaired image-to-image translation). Even though we didn't use any techniques mentioned in this paper explicitly in our project, this paper helped us understand the concepts of loss associated with image translation a little better.

<https://arxiv.org/pdf/1611.07004>

### 3. DATA EXPLORATION AND DATA PROCESSING

#### 3.1 Data Exploration

The 'monet\_jpg' dataset has 300 Monet paintings sized 256x256 in JPEG format, 'monet\_tfrec' dataset has 300 Monet paintings sized 256x256 in TFRecord format, 'photo\_jpg' dataset has 7028 photos sized 256x256 in JPEG format and 'photo\_tfrec' dataset has 7028 photos sized 256x256 in TFRecord format. We are using both the TFRecords datasets to train the model. The monet images are mostly for the model to learn patterns from and apply the transformation to the photos. The output of this competition will be the output generated from the first Generator (gives monet style paintings when normal photographs are given)

#### Files

- monet\_jpg - 300 Monet paintings sized 256×256 in JPEG format
- monet\_tfrec - 300 Monet paintings sized 256×256 in TFRecord format
- photo\_jpg - 7028 photos sized 256×256 in JPEG format
- photo\_tfrec - 7028 photos sized 256×256 in TFRecord format

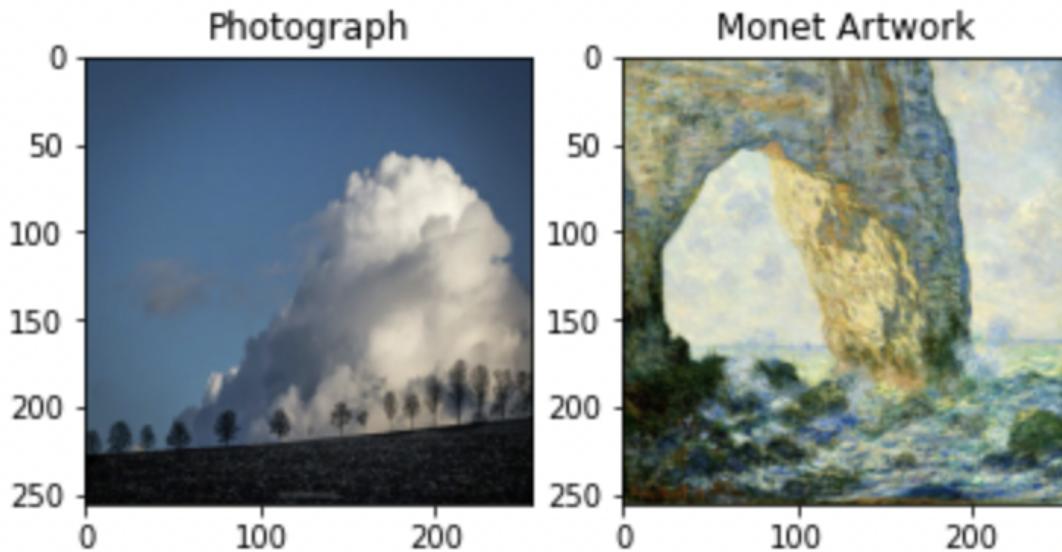
```
num_records_monet = sum(1 for record in monet_images_dataset)
num_records_photographs = sum(1 for record in photographs_images_dataset)
print("# of monet images to train with: %s" % num_records_monet)
print("# of photographs to predict: %s" % num_records_photographs)
```

```
2021-12-01 15:39:07.370320: W ./tensorflow/core/distributed_runtime/eager/destroy_tensor_handle_node.h:57] Ignoring an error encountered when deleting remote tensors handles: Invalid argument: Unable to find the relevant tensor remote_handle: Op ID: 329, Output num: 0
Additional GRPC error information from remote target /job:worker/replica:0/task:0:
{"created": "@1638373147.367042033", "description": "Error received from peer ipv4:10.0.0.2:8470", "file": "external/com_github_grpc_grpc/src/core/lib/surface/call.cc", "file_line": 1056, "grpc_message": "Unable to find the relevant tensor remote_handle: Op ID: 329, Output num: 0", "grpc_status": 3}
```

```
# of monet images to train with: 300
# of photographs to predict: 7038
```

There are more photographs for the prediction than there are monet images. This is good, as the photographs transformations will be very useful in training and validating. The monet images are mostly for the model to learn patterns from and apply the transformation to the photos.

## **SAMPLE DATASET**



These are the sample images from each of the datasets. The first image is from photo\_tfrec format and the second is from monet\_tfrec.

## 3.2 Data Preprocessing

We're rescaling the pixel values from -1 to +1. Most of the images were already in the required shape. Just to be sure, we're resizing all the images to 256X256 shape. There's not much preprocessing to be done as these are images and also, the competition focuses on our ability to make the CycleGAN work instead of mostly giving more attention to data preprocessing. Hence all the images are mostly ready to use without much preparation.

The below functions are the ones we used for loading the tfrecord format datasets and preprocessing.

```
def decode_image(photo):
    photo = tf.image.decode_jpeg(photo, channels=3)
    photo = (tf.cast(photo, tf.float32) / 127.5) - 1
    photo = tf.reshape(photo, [256, 256, 3])
    return photo

def tfrecord_read(sample):
    tfrecord_format = {"image_name": tf.io.FixedLenFeature([], tf.string),
                       "image": tf.io.FixedLenFeature([], tf.string),
                       "target": tf.io.FixedLenFeature([], tf.string)}
    sample = tf.io.parse_single_example(sample, tfrecord_format)
    image = decode_image(sample["image"])
    return image

def load_dataset(names, labeled=True, ordered=False):
    ds = tf.data.TFRecordDataset(names)
    ds = ds.map(tfrecord_read, num_parallel_calls = AUTOTUNE)
    return ds
```

These are the most commonly used functions for loading any tfrecord dataset with images as the data items.

## **4. Problem Formulation and Model Selection**

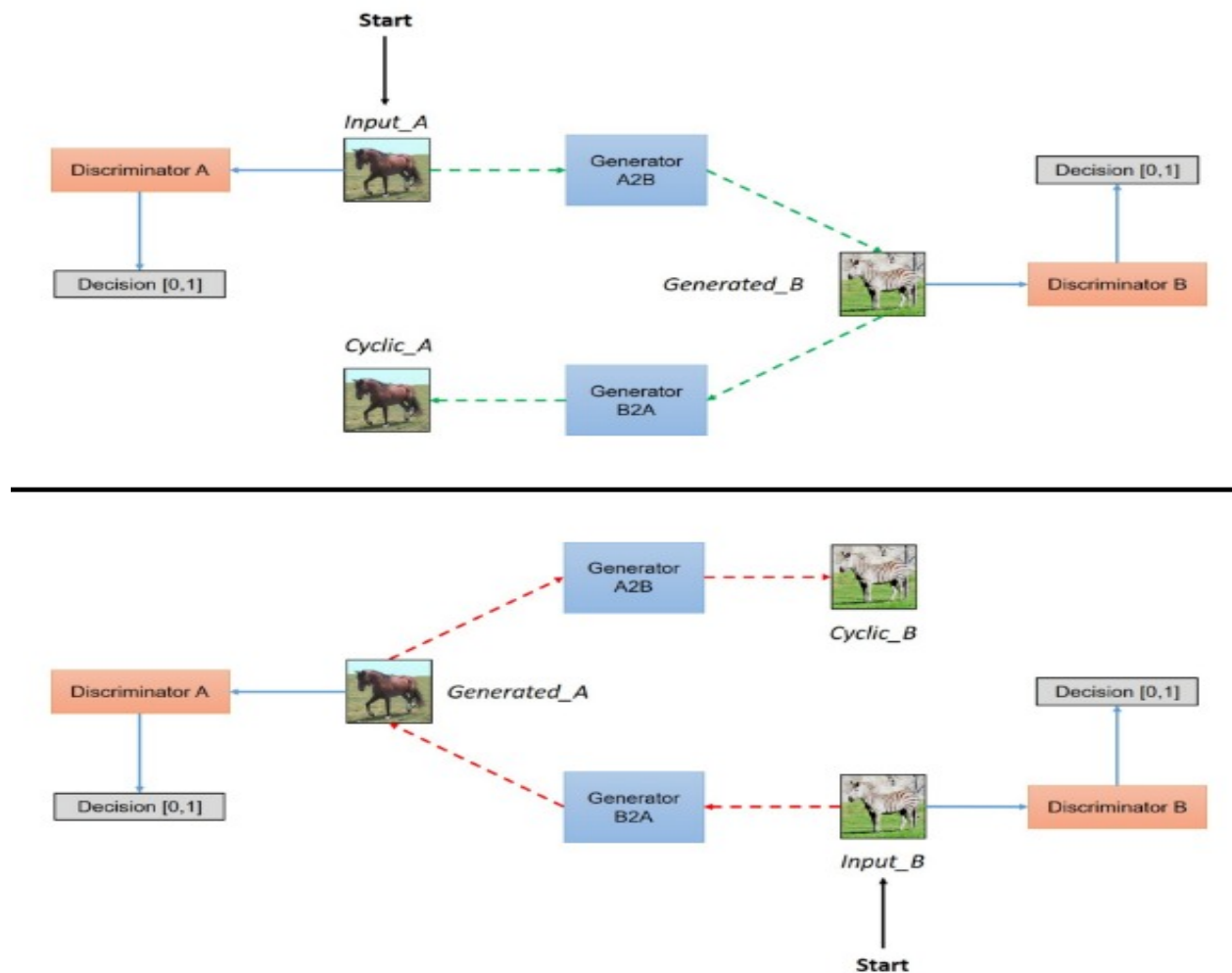
### **4.1 Problem Formulation**

In order to perform Monet to Photo translation, we know that we had to choose an image generator model that could be trained. In order to verify the “closeness” of the generated image to the “expected” image, two requirements would need to be fulfilled: Classification Model layers fed from the Generator model output to compare if image is real or fake, and input images with photo with exact paired monet painting version for training the model for each source image. Looking at the dataset, the training dataset consisted of two types: source photo and monet photo. The Source photos had over 1000 images while the Monet photos had around 300 images. Neither of the images in these datasets were matching pairs. Due to the fact that we needed a model capable of being trained in an unsupervised way where no matching pairs of training images were needed, this reduced the available models we could use, and with the Kaggle competition requirements page description, this would need to be some variation of a CycleGAN. As mentioned before on the requirements, the Classification model for the images is simply the Discriminator model and the dataset provided by Kaggle is already compatible with the model and doesn't need to have preprocessing performed on it, except for scaling all images to the same size during model training and model prediction.

### **4.2 Model Selection**

For the model, we didn't have to explore other options as this is a classic unpaired image-to-image translation problem statement and there are no better techniques other than CycleGAN which perform exceptionally well on these kinds of problem statements. It was also mentioned in the description of the Kaggle competition that it is advised to use CycleGANs for this problem. So, we went with this model.

The CycleGAN high level architecture looks like below ([source](#)):



As seen above, the CycleGAN architecture consists of two generator models and two discriminator models entwined together in a special way such that the training photo and training monet image (unpaired) undergo a forward pass and backward pass generation process to be fed into various loss functions used to train the generator and discriminator models on each pass. Both the generator and the discriminator models are CNNs primarily focused on image training, so there are many layers with Conv2D blocks, padding blocks and other familiar image training blocks used by typical CNNs. However, in addition to this, there are now upsampling and downsampling blocks as well. The four models are trained by updating weights through finding gradients based on loss function outputs compared to actual current training weights for the models. The four loss functions for the architecture are the cycle loss, identity loss, generator loss and discriminator loss. These functions are shown below.



### Cycle Consistency Loss:

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

```
def cycle_loss(real_image, cycled_image, lamda):  
    loss = tf.reduce_mean(tf.abs(real_image - cycled_image))  
    cyc_loss = loss * lamda  
    return cyc_loss
```

### Identity Loss:

$$\mathcal{L}_{\text{identity}}(G, F) = \mathbb{E}_{y \sim p_{\text{data}}(y)} [\|G(y) - y\|_1] \\ + \mathbb{E}_{x \sim p_{\text{data}}(x)} [\|F(x) - x\|_1]$$

This loss added to the training weights adjustment phase helps preserve color and tint in the images.

```
def identity_loss(real_image, same_image, lamda1, lamda2):  
    loss = tf.reduce_mean(tf.abs(real_image - same_image))  
    id_loss = loss * lamda1 * lamda2  
    return id_loss
```

### Generator Loss:

Performs binary cross entropy for 1s matrix of fed generator output to generator output, obtaining the loss.

```
def generator_loss(generated_image):  
    gen_loss = keras.losses.BinaryCrossentropy(from_logits=True,  
                                                reduction=tf.keras.losses.Reduction.NONE)(tf.ones_like(generated_image), generated_image)  
    return gen_loss
```

### Discriminator Loss:

Performs binary cross entropy for 1s matrix of fed discriminated real image output to discriminated real image output and another binary cross entropy for the discriminated

generated image output. These are then summed together and multiplied by a scalar value to obtain the loss.

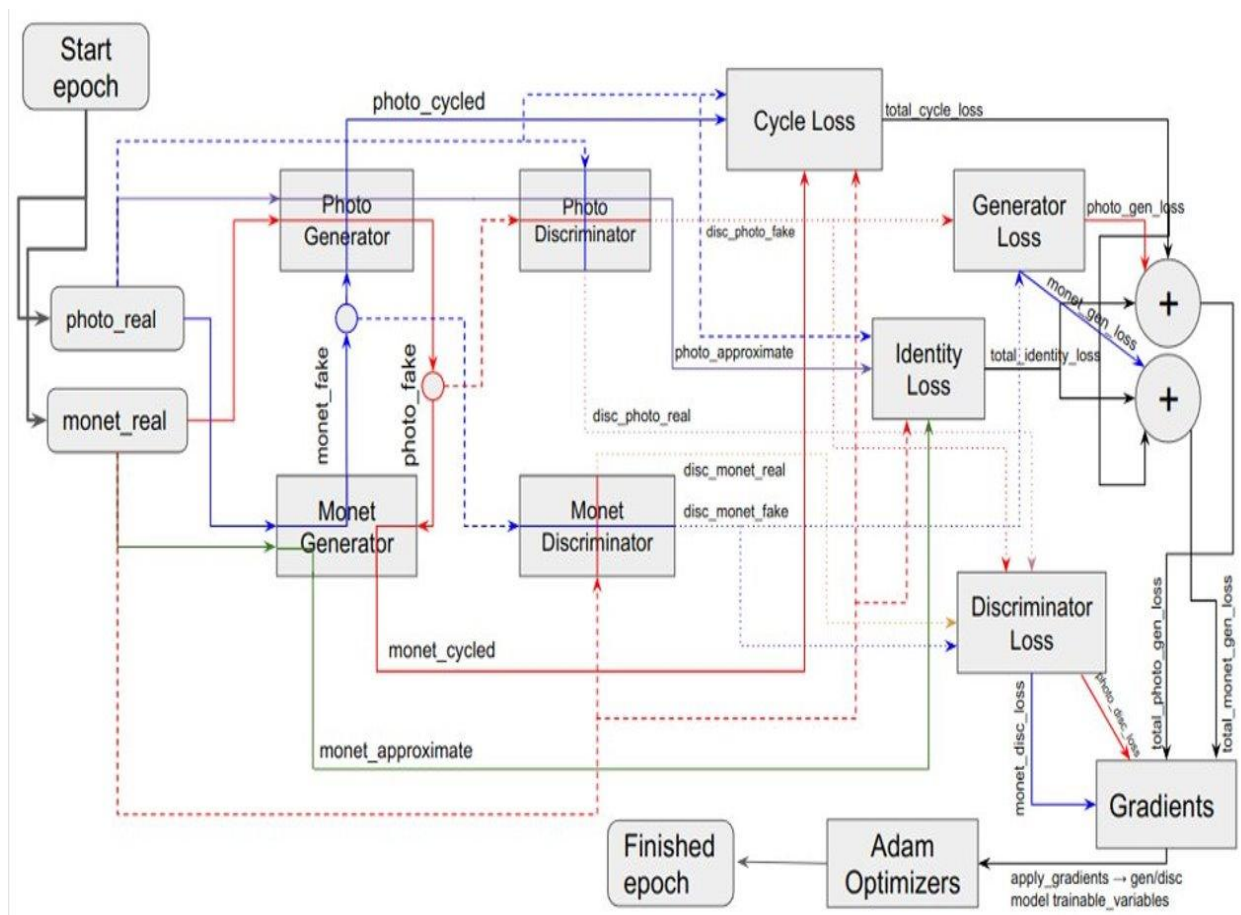
```
def discriminator_loss(real_image, generated_image):  
    loss_real = keras.losses.BinaryCrossentropy(from_logits=True,  
                                                reduction=tf.keras.losses.Reduction.NONE)(tf.ones_like(real_image), real_image)  
    loss_fake = keras.losses.BinaryCrossentropy(from_logits=True,  
                                                reduction=tf.keras.losses.Reduction.NONE)(tf.zeros_like(generated_image), generated_image)  
  
    total_discriminator_loss = (loss_real + loss_fake) * 0.5  
    return total_discriminator_loss
```

### Explanation:

At the beginning of each training session, the model selects a training photo and a training monet image and starts generating images in a forward pass and then generates images in the backward pass. In the forward pass, the training photo is fed into the Monet Generator (G2), saves the resulting image, and then feeds that image back to the Photo Generator (G1) and also saves the image. In the backwards pass, the training monet photo is fed into the Photo Generator (G1), saves the image and then feeds the generated image back through the Monet Generator (G2) and saves that image as well. By this point, four new generated images were saved in the current training session: monet\_generated, photo\_generated, monet\_cycled, photo\_cycled. Additionally, the model also uses the same two generators to feed the training photo through the Photo Generator (G1) to obtain an approximate copy of the same photo for some later loss evaluation called photo\_approximate. Similarly, the same is done with the training monet image through the Monet Generator (G2) called monet\_approximate. After these images are generated, the discriminator models are utilized to generate several outputs used for the loss functions outputs for weights adjustments. The monet discriminator model (D1) is used twice to generate outputs with the following images: monet\_generated and training monet image. These results from these images are saved to be used in later loss functions called disc\_monet\_generated and disc\_monet\_real. Likewise, the photo discriminator model (D2) is used to do the same to the images photo\_generated and training photo image, producing disc\_photo\_generated and disc\_photo\_real. It should be noted here that the discriminator models produce output vectors which describe the differences in the inputs found to the type of image desired. For example, the monet discriminator output returns an output on how close the input image is to a monet photo, and likewise the photo discriminator returns an output to how close the input is to a photo. All of the outputs from the generator and discriminator models are then fed into four main loss functions: cycle\_loss, generator\_loss, identity\_loss and discriminator\_loss. These losses compare

the training photo and training monet to the outputs we obtained earlier and return a decimal value output where the lower the number the better the model is at the specific transformation. For example, photo cycle\_loss is fed the training photo and photo\_cycled. The output value tells how “bad” the photo\_generator is at generating realistic photos when fed a real photo. If the output is less than 1 or higher, the generator is not very good. If instead the output is something like 0.6 the model is better but could need some improvements. All of the loss functions are then used together with keras\_training\_variables to obtain the gradients for the model updates. Finally these gradients are fed to the apply\_gradients() function for the model to update all the model weights to train and improve the model on every subsequent pass. In our experience, 50 training passes (50 epochs) are the minimum for getting a somewhat believable image translation from a photograph to a monet image output.

Mapping out the whole training step described above, we obtain the flow chart shown below:



### Selected Model:

For the main model we used, we chose a scalar value of 0.5 for the discriminator loss function and used the most popular ResNet generator model structure. Despite this, the generator model was very complex while the discriminator model was more simple.

The final activation function for the Generator model was a “tanh” function as it allowed the generator neural network to not miss any features found that are positive or negative to the classification of the image type (photo or generator).

### Snapshot of the Generator Code Block:

---

```
image_input = layers.Input(shape=(256,256,3), name=image_layer_name)

# Define the Relu activation layer.
relu_activation_layer = layers.Activation("relu")

model_layers = PaddingConstant2D(padding=(3,3))(image_input)
model_layers = layers.Conv2D(filters, (7,7), kernel_initializer=kernel_initializer, use_bias=False)(model_layers)
model_layers = tf.nn.instance_normalization(gamma_initializer=gamma_initializer)(model_layers)
model_layers = relu_activation_layer(model_layers)

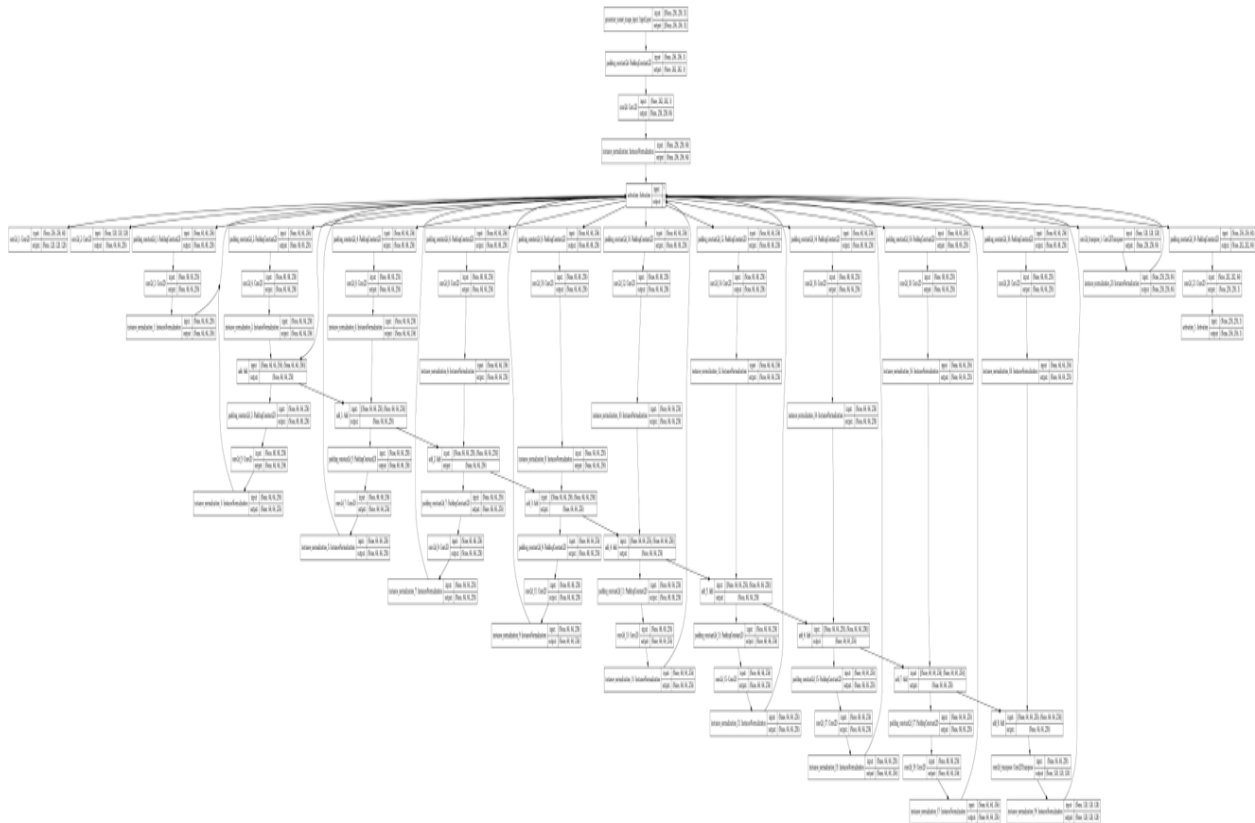
# Add Downsampling layers
for _ in range(num_downsampling_blocks):
    filters *= 2
    model_layers = downsampling_layers(model_layers, filters=filters, activation=relu_activation_layer)

# Add Residual block layers
for _ in range(num_residual_blocks):
    model_layers = residual_block_layers(model_layers, activation=relu_activation_layer)

# Add Upsampling layers
for _ in range(num_upsampling_blocks):
    filters //= 2
    model_layers = upsampling_layers(model_layers, filters=filters, activation=relu_activation_layer)

# Final layers with Tanh activation.
model_layers = PaddingConstant2D(padding=(3,3))(model_layers)
model_layers = layers.Conv2D(3, (7,7), padding="valid")(model_layers)
model_layers = layers.Activation("tanh")(model_layers)
```

## ResNet Generator Model with over 20 layers:



The final activation function for the discriminator model is a Conv2D block.

```
image_input = layers.Input(shape=(256,256,3), name=image_layer_name)

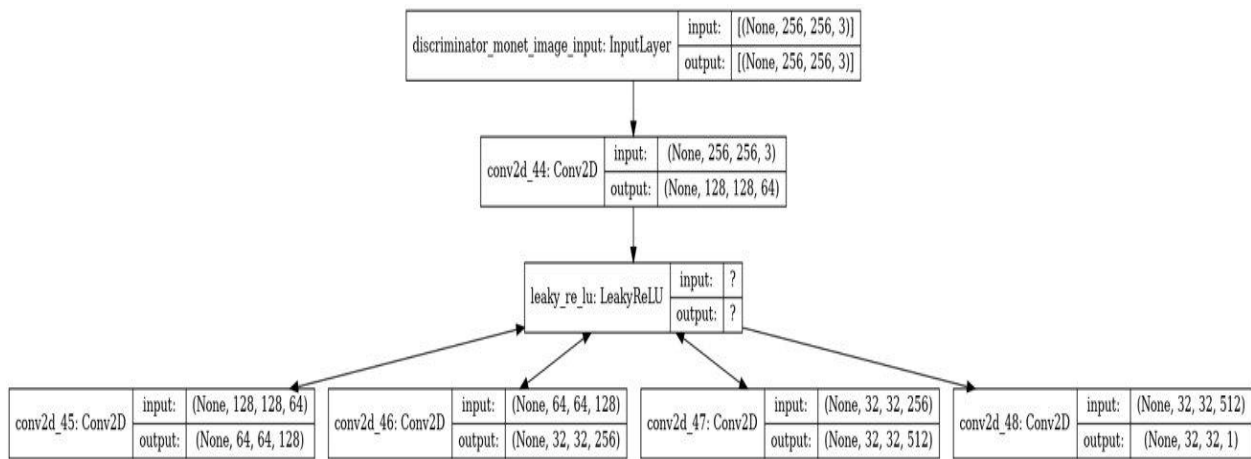
# Define the leaky relu activation layer.
leaky_relu_activation_layer = layers.LeakyReLU(0.2)

# Add a convolution layer with 2x2 strides followed by a leaky relu activation layer.
model_layers = layers.Conv2D(filters=(4,4), strides=(2,2), padding="same",
                              kernel_initializer=kernel_initializer)(image_input)
model_layers = leaky_relu_activation_layer(model_layers)

# Add Downsampling layers.
num_filters = filters
for num_downsample_block in range(3):
    num_filters *= 2
    if num_downsample_block < 2:
        model_layers = downsampling_layers(model_layers, filters=num_filters, activation=leaky_relu_activation_layer,
                                             kernel_size=(4,4), strides=(2,2))
    else:
        model_layers = downsampling_layers(model_layers, filters=num_filters, activation=leaky_relu_activation_layer,
                                             kernel_size=(4,4), strides=(1,1))

# Finally add the convolution layer with 1x2 stride at the end of the model.
model_layers = layers.Conv2D(1, (4,4), strides=(1,1), padding="same",
                              kernel_initializer=kernel_initializer)(model_layers)
```

**Discriminator Model with 7 layers:**



## 5. EVALUATION, RESULT ANALYSIS, VISUALIZATION

### 5.1 Model Justification

We used Cycle GANs for this specific problem because CycleGANs perform exceptionally well on unpaired image-to-image translation. And also, as this was a Kaggle competition, it was given in the description section that CycleGANs are the way to go for this kind of problem statements.

### 5.2 Model Evaluation

#### MiFID

Submissions are evaluated on MiFID (Memorization-informed Fréchet Inception Distance), which is a modification from [Fréchet Inception Distance \(FID\)](#).

The smaller MiFID is, the better your generated images are.

#### What is MiFID (Memorization-informed FID)?

In addition to FID, Kaggle takes training sample memorization into account.

The memorization distance is defined as the minimum cosine distance of all training samples in the feature space, averaged across all user generated image samples. This distance is thresholded, and it's assigned to 1.0 if the distance exceeds a pre-defined epsilon.

In mathematical form:

$$d_{ij} = 1 - \cos(f_{gi}, f_{rj}) = 1 - \frac{f_{gi} \cdot f_{rj}}{\|f_{gi}\| \|f_{rj}\|}$$

where  $f_g$  and  $f_r$  represent the generated/real images in feature space (defined in pre-trained networks); and  $f_{gi}$  and  $f_{rj}$  represent the  $i^{th}$  and  $j^{th}$  vectors of  $f_g$  and  $f_r$ , respectively.

$$d = \frac{1}{N} \sum_i \min_j d_{ij}$$

defines the minimum distance of a certain generated image ( $i$ ) across all real images ( $j$ ), then averaged across all the generated images.



$$d_{thr} = \begin{cases} d, & \text{if } d < \epsilon \\ 1, & \text{otherwise} \end{cases}$$

defines the threshold of the weight only applies when the ( $d$ ) is below a certain empirically determined threshold.

Finally, this memorization term is applied to the FID:

$$MiFID = FID \cdot \frac{1}{d_{thr}}$$

## Kaggle Prediction:

59	Hisaam Hashim	</> monet2photo - Cyc...		61.86613
71	Rohit Sai Chowdary			91.47916

There were around 80+ teams at the time and the best ranking that we got is shown above. Although 59 doesn't seem like a great result, it was a challenging competition and most of the concepts were too advanced for this course and we had to do immense research just to get our model working. Making GANs work is one of the most challenging tasks that we had to do. And most people participating in this competition are experts as the problem statement wasn't something that most people entering the field of Data Science and Machine Learning can just participate and do well in.

## **Model Fitting and Final Losses:**

### **Train the CycleGAN Model with 50 epochs**

The model fitting will be stuck at Epoch 1/50 for sometime while loading to the TPU, after sometime, the TPU will kickstart and the training will begin.

With the TPU, each epoch takes approximately 2 minutes. This means 50 epochs takes about 120 minutes or about 2 hours.

```
: history = cycle_gan_model.fit(tf.data.Dataset.zip((monet_images_dataset, photographs_images_dataset)),  
                                epochs=50)
```

We fit the model with the `monet_images_dataset` and `photographs_dataset` and made it run for 50 epochs. We could run for more epochs but we had some constraints.

- Each epoch runs for around 2 mins and it almost takes 2 hours for 50 epochs to run.
- We have to train the model on TPUs only as this is an extremely intensive task but the free usage of TPU granted was for 20 hours per week.



- Even if we made small changes in parameters and tried to train the model again, it took 2 hours just to see if we got better results or not. So, we had to constrain ourselves to 50 epochs only.

```
Epoch 46/50
300/300 [=====] - 97s 325ms/step - monet_gen_loss: 5.4031 -
photo_gen_loss: 5.1777 - monet_disc_loss: 0.5529 - photo_disc_loss: 0.6175
Epoch 47/50
300/300 [=====] - 97s 325ms/step - monet_gen_loss: 5.4876 -
photo_gen_loss: 5.2851 - monet_disc_loss: 0.5570 - photo_disc_loss: 0.6165
Epoch 48/50
300/300 [=====] - 97s 324ms/step - monet_gen_loss: 5.3511 -
photo_gen_loss: 5.1627 - monet_disc_loss: 0.5560 - photo_disc_loss: 0.6042
Epoch 49/50
300/300 [=====] - 97s 325ms/step - monet_gen_loss: 5.2566 -
photo_gen_loss: 5.0579 - monet_disc_loss: 0.5687 - photo_disc_loss: 0.6183
Epoch 50/50
300/300 [=====] - 97s 324ms/step - monet_gen_loss: 5.2053 -
photo_gen_loss: 5.0713 - monet_disc_loss: 0.5876 - photo_disc_loss: 0.6125
```

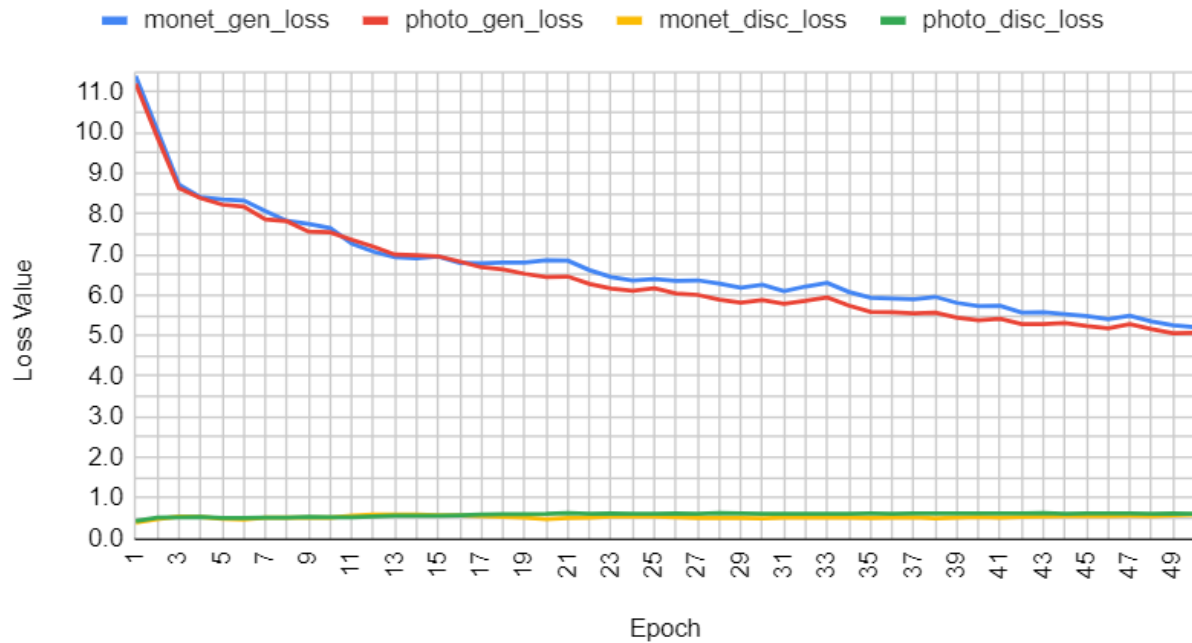
These are the last 5 epochs of our training and the final losses that we got. We essentially had 4 losses and they are:

- Monet Generator Loss
- Photo Generator Loss
- Monet Discriminator Loss
- Photo Discriminator Loss

The final result on kaggle is evaluated on these 4 losses and also the output images that were generated. This competition required us to generate 7000-10000 images and we generated 7038 images finally. It took a lot of time for all the 7038 images to be generated. Hence we were constrained by this factor as well.

Plotting all the losses:

### Loss vs Epoch (CycleGAN Model)



- The above graph is of our best result on Kaggle.
- From the above graph, we see that monet\_gen\_loss and photo\_gen\_loss go down as the number of epochs is increasing. We can increase the number of epochs and the losses might still go down. But we had limited resources as mentioned above.
- Both the discriminators had lesser values from the start and it maintained steady for all the 50 epochs.

```
Generated Monet of photograph for photo = 5151/7038
Generated Monet of photograph for photo = 5252/7038
Generated Monet of photograph for photo = 5353/7038
Generated Monet of photograph for photo = 5454/7038
Generated Monet of photograph for photo = 5555/7038
Generated Monet of photograph for photo = 5656/7038
Generated Monet of photograph for photo = 5757/7038
Generated Monet of photograph for photo = 5858/7038
Generated Monet of photograph for photo = 5959/7038
Generated Monet of photograph for photo = 6060/7038
Generated Monet of photograph for photo = 6161/7038
Generated Monet of photograph for photo = 6262/7038
Generated Monet of photograph for photo = 6363/7038
Generated Monet of photograph for photo = 6464/7038
Generated Monet of photograph for photo = 6565/7038
Generated Monet of photograph for photo = 6666/7038
Generated Monet of photograph for photo = 6767/7038
Generated Monet of photograph for photo = 6868/7038
Generated Monet of photograph for photo = 6969/7038
```

The above screenshot is of the number of images that were generated. We transformed 7038 normal photos to monet-style paintings as per the competition's requirements.

It took around 1 min for every 100 images to be generated. So, in total this action took around 70-80 minutes to generate all the images required.

Some output images that we generated:

Photograph Input



Generated Monet Output



Photograph Input



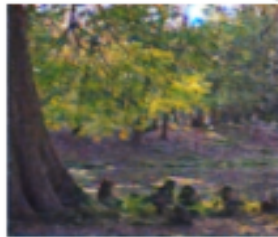
Generated Monet Output



Photograph Input



Generated Monet Output



Photograph Input



Generated Monet Output



Photograph Input



Generated Monet Output



Photograph Input



Generated Monet Output



Photograph Input



Generated Monet Output



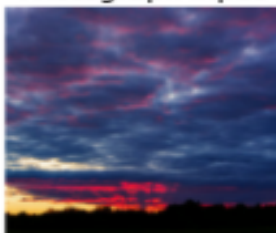
Photograph Input



Generated Monet Output



Photograph Input



Generated Monet Output



Photograph Input



Generated Monet Output



## 6. Conclusion, Future Work and Improvement

### Conclusion:

- Began by loading in the TFRecords format dataset of photographs and monet-style paintings.
- Did basic preprocessing required for these images (didn't require much because all the images were already resized and given).
- Build Generator and Discriminator models.
- Defined loss function required which are generator loss, discriminator loss, cycle loss and identity loss which are the core losses for CycleGAN models.
- Built the CycleGAN model and trained it with both the datasets ( normal photos and monet-style images ) for 50 epochs.
- The final losses after 50 epochs were

**Monet\_Generator\_Loss - 5.2053**

**Photo\_Generator\_Loss - 5.0713**

**Monet\_Discriminator\_Loss - 0.5876**

**Photo\_Discriminator\_Loss - 0.6125**

### Future Work and Improvement:

- Our final model's performance can be improved if we trained it for more epochs but we were constrained by resources. If we had TPUs available for more than 20 hours per week, we could've gotten better results. Also, if we had more training data for both the datasets, then the model's performance could likely be improved.
- The loss functions used can be custom-implemented. But these concepts are too advanced and implementing our own loss functions was extremely difficult to do. So, we had to limit ourselves to using basic loss functions taught in the coursework like Binary Cross Entropy and Categorical Cross Entropy.
- Keep an eye on current research trends in the field of image generation and also follow publications very closely to explore further advanced techniques which can deal with problem statements like this.
- The topic of Generative Adversarial Networks is fast advancing and there are hundreds of GAN architectures available till date and the number of research papers with the name "GAN" in their titles are increasing exponentially. If the research keeps going at this pace, we could come up with a more sophisticated model than a cycleGAN in a year or two at the most.