Summary

Monet CycleGAN Model – Advanced Deep Learning Project

Our model was developed after reading the article "Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks" authored by the Berkeley AI Research (BAIR) laboratory, which delves into the topic of CycleGAN.

Selection of model parameters:

Image size was fixed at 320x320x3 as per the project requirements.

The Batch Size initially set to 1, following the article's recommendation. Over time and with the hardware capabilities (RTX 4090), we increased it to 4 to enhance training speed, ensuring each batch runs through the entirety of the Monet's image dataset.

The chosen number of epochs was initially 200, following the article's recommendation. However, based on our model's learning capabilities, we gradually reduced it to 100 epochs for the final model.

We chose our Residual Block to start as 7 and then changed it to 12.

Adam optimizer was chosen due to its popularity among practitioners in this field. The initial learning rate was set to 0.0002, as recommended in the article, with beta_1 set to 0.5 for consistency. Other values remained default. Adam remained our optimizer throughout all the model components.

Datasets:

For training, we utilized 7036(2 images less than the full data set for it to be divided by the batch size) regular images from the competition dataset. We augmented Monet's image dataset to 1028 images (link provided in the relevant section of the file) to train the model on a larger image dataset (compared to only 300 images in the competition dataset).

Model Architecture:

To maximize code readability, we decided to work with generic blocks that tie both the Generator and the Discriminator, each tailored to our requirements.

1. Encoder Block: Compresses the image, the operation depends on how the block is externally triggered regarding decreasing input quantity (a variable passed to the block). It consists of a single Conv2D layer with ReLU activation function, chosen for its suitability for learning complex features, and optionally Instance Normalization layer, added later during experimentation (detailed explanation in a dedicated section).

2. Residual Block: Addresses the "Vanishing Gradient" problem during backpropagation in deep learning models. It prevents degradation of weights in deep layers. The number of blocks of this type varied in our experiments, detailed later. This block contains two Conv2D layers, with another ReLU activation function layer. After few more experiments two ReflectionPadding2D layers were added before every Conv2D layer (detailed later).

3. Decoder Block: This block is the reverse of the Encoder Block in its role. It performs upsampling using the Conv2DTranspose layer. Instance Normalization can also be applied here.

4. Other block details such as Bias, Strides, Padding were chosen according to the article recommendations, providing quality performance during experiments.

5. Generator:

   Takes input size (320x320x3) and the chosen number of Residual Blocks. This parameter allows us to conduct various experiments with different numbers of Residual layers without substantial code changes. Next, the input passes through 3 Encoder blocks with an increasing number of inputs, doubling from 64 onwards. Then, the input traverses through X Residual blocks.
   Finally, the process reverses into compression using 2 Decoder blocks with decreasing input sizes, halving from 128 onwards, with a Concatenate layer between them. The Concatenate layer's role is to "stitch" different inputs from various Residual blocks into one large tensor. This action is crucial as it consolidates inputs from different Residual blocks back into a single input at the end of the generation process.

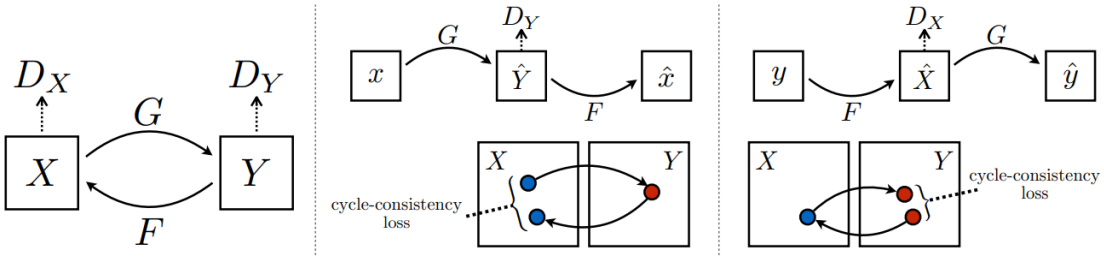
6. Discriminator:

   Initially, the input passes through a RandomCrop layer, which randomly extracts a portion of the input image before entering the discriminator stages. The goal is to enhance its performance, as it may frequently receive partially missing input images randomly and still be required to correctly identify and classify them (whether they are regular images or paintings by Monet). Afterward, the input goes through 4 Decoder blocks, as explained above.

<u>CycleGan Class:</u>

- Architecture:

  We created the architecture of the CycleGan exactly as described by the main



  article.

- Train Step: to train our model, we had to override the train_step function found in the Keras library. Our train_step function follows the exact description in the diagram above. From generating the images to the cycle loss. Each train step calculates all the losses and sums them together:
  - Discriminator Losses(binary cross-entropy, comparing to a tensor of ones or zeros(real or fake))
  - Generator Losses(binary cross-entropy, comparing the generated images to a tensor of ones)
  - Identity Losses(MAE)
  - Cycle Consistency Loss(MSE)*This is the only loss different than the main article.
- Gradient Calculation and Optimization: Gradients of the total loss are computed for each generator and discriminator, and optimization steps are performed to update their weights.

Experiments and Lessons Learned:

The process of conducting experiments until reaching the final model involved several steps:

1. To measure the model's learning progress and improvement during training, four random regular images were converted into Monet-style images at the end of each epoch, and printed. Initially, we encountered an issue where each generated Monet-style image exhibited a frame that did not match the surrounding pixel characteristics. Below is an example:

After researching the issue, we discovered that it is a common phenomenon. When our neural network processes an image, it utilizes padding to maintain the image's dimensions after convolution operations. We initially chose a very basic form of padding (a value defined in the Keras library within the Conv2D layer) that is not suitable for such a complex model. This led to the formation of unwanted bands at the borders of the image. To address this phenomenon, we added a new layer called ReflectionPadding2D, which exists in the PyTorch library and is implemented based on information found in the Keras documentation. This layer was inserted at a critical stage in image construction, specifically before each Cov2D layer within the Residual block. This layer essentially performs a reflective padding process, where it "reflects" similar pixels surrounding the pixel that needs to be completed. This helps to minimize the formation of prominent artifacts unrelated to the image structure, such as the bands.

2. After implementing this action, we encountered a new issue: the images generated by the Generator did not sufficiently match the style of Monet (for example, the brush strokes). In fact, the Discriminator managed to distinguish Monet-style images from regular ones effectively, but the Generator failed to produce them with high quality. Below is an example:

In order to enhance our model, we utilized the paper "Instance Normalization: The Missing Ingredient for Fast Stylization" (https://arxiv.org/pdf/1607.08022.pdf). This paper effectively presents the tool we chose to employ, which assists models in learning the reflective style from one image and applying it to a new image. We decided to integrate the Instance Normalization layer taken from the TensorFlow Addons library. This layer was incorporated at the end of each block, specifically after each Residual block.

The goal is to perform normalization in the following manner:
For each instance in the batch, Instance Normalization computes the mean and variance of that instance's channels. It then uses these statistics to normalize the instance's channels to have a mean of zero and a variance of one. After normalization, it applies a scale (gamma) and an offset (beta), which are learned parameters, allowing the normalized output to be scaled and shifted in a way that the network learns is most beneficial.

3. After implementing both of these improvements, we observed an enhancement in the quality of the images generated by the model. The images produced during the advanced training stages became more similar to Monet's style, but around Epoch 50, we noticed a decline in image quality. To address this issue, we proposed reducing the learning rate linearly from a certain stage of training.
   To achieve this, we implemented the LinearLRScheduler function. Starting from a defined epoch, this function gradually decreases the learning rate based on the progress of training (i.e., the number of epochs). As it operates in more advanced training stages, the reduction in the learning rate becomes more drastic.

   Executing this action resulted in an improvement in the quality of the generated images. For example, at Epoch 50:

Before

After

4. As part of our efforts to improve the quality of the generated images, we added data augmentation techniques to the dataset used for the training. We decided to apply Horizontal Flip augmentation for both classes, as recommended by the article.
   However, we added an additional augmentation specifically for images labeled as "Normal". This additional augmentation involves randomly cropping the images, aiming to challenge the model by introducing variations in the image boundaries, thus facilitating its improvement.

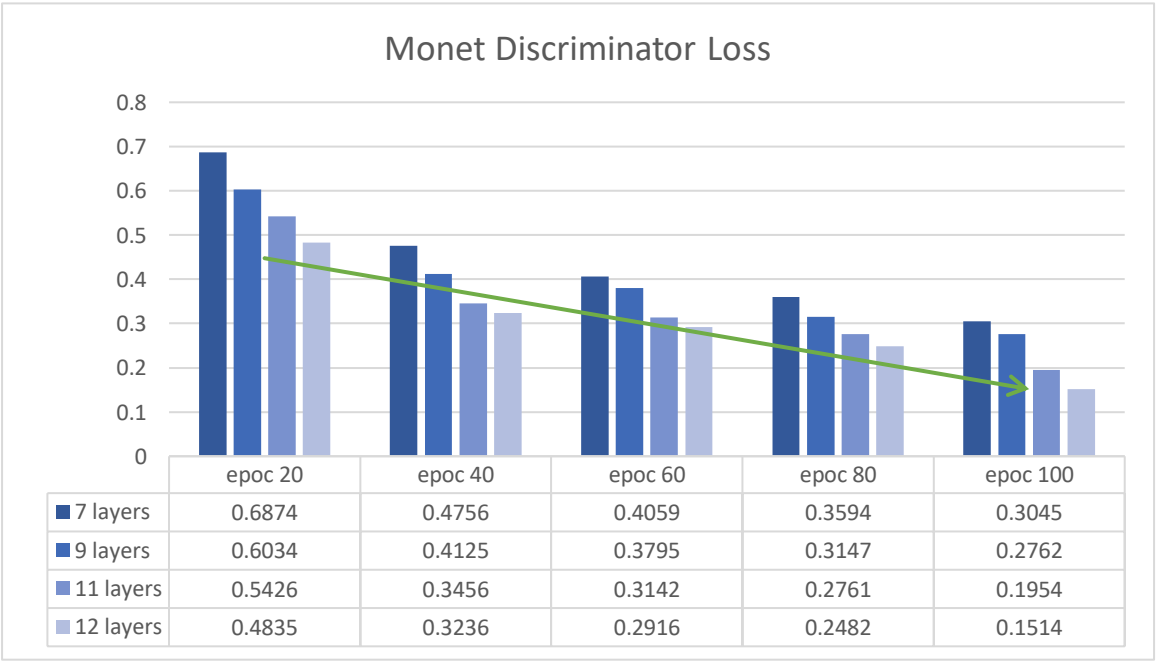
5. When running our Model using the cycle consistency loss (MAE) found in the article, we didn't see any drastic changes in early epochs and only saw changes after around epoch 150 but by then the images would begin to distort. According to this article https://www.tongzhouwang.info/better_cycles/report.pdf Tongzhou Wang

and Yihan Lin found that enforcing a “weaker” notion of cycle consistency can create more drastic changes to the output. They used an L1 loss on the CNN features extracted by the corresponding discriminator, we tried a simpler approach of changing the MAE loss into an MSE loss. After changing this loss, we could already see good looking images from epoch 20.

MAE at epoch 160

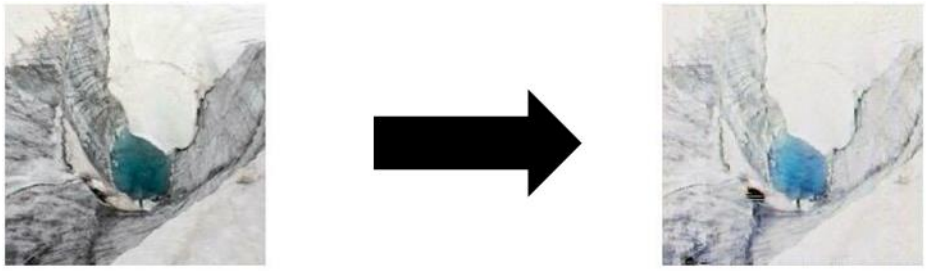
MSE at epoch 60

6. The last experiment conducted involved varying the number of Residual blocks integrated into the Generator. We started with 7 layers and then incrementally increased to 9, 11, and 12 layers.
   Below are the results:

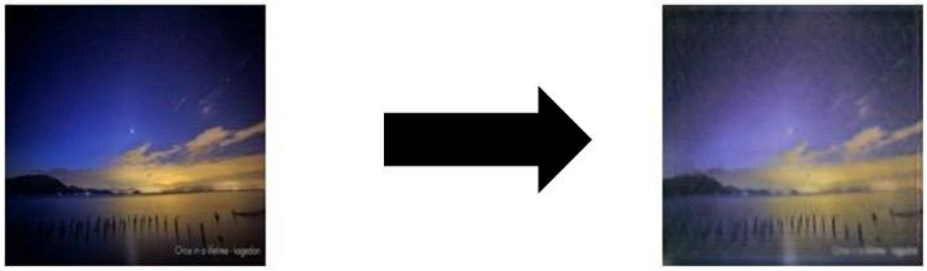


	epoc 20	epoc 40	epoc 60	epoc 80	epoc 100
7 layers	0.6874	0.4756	0.4059	0.3594	0.3045
9 layers	0.6034	0.4125	0.3795	0.3147	0.2762
11 layers	0.5426	0.3456	0.3142	0.2761	0.1954
12 layers	0.4835	0.3236	0.2916	0.2482	0.1514

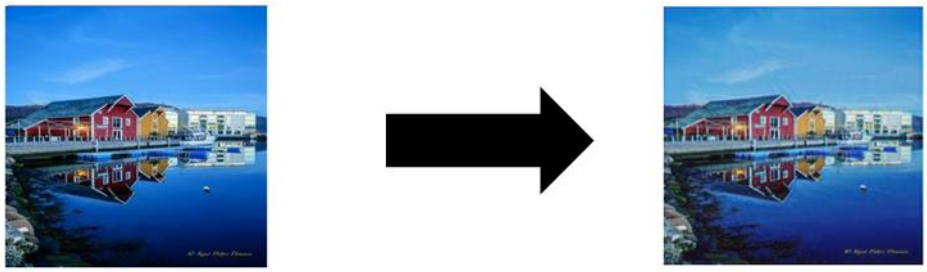
Epoch 45, 7 Residual blocks:



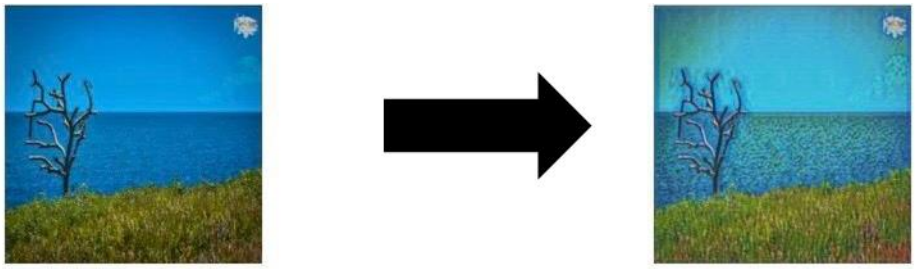
Epoch 45, 9 Residual blocks:



Epoch 45, 11 Residual blocks:



Epoch 45, 12 Residual blocks:



Training Example: