Monet Painting Style Transfer Using CycleGAN

Matthew O'Neill
University of Massachusetts Amherst
Amherst, MA 01003
moneill@umass.edu

Adhrit Srivastav
University of Massachusetts Amherst
Amherst, MA 01003
adhritsrivas@umass.edu

Prasad Adhiyaman
University of Massachusetts Amherst
Amherst, MA 01003
padhiyaman@umass.edu

Abstract

This paper explores and employs the generative abilities of General Adversarial Networks (GANs) to project Monet painting Style Transfer onto photographed images. In particular, CycleGANs, the marriage of two GANs, were used in order to effectively capture, learn, and project Monet's style of painting.

1. Introduction

The goal of this project is to explore the problem of training style transfer models for image generators. We also want to explore image generation itself and create the most accurate generated images that we can. We are doing so by exploring and testing a Generative Adversarial Network architecture called CycleGANs as the primary method for training style transfer models.

1.1. Generative Adversarial Network(GAN)

A generative adversarial network, or GAN, is a neural networks strategy for generating images to be used to achieve some goal (GANs are incredibly useful for methods like data augmentation). It consists of training two neural network models, a generator and a discriminator, the first of which is trained to generate images which fit some criteria and the second is trained to evaluate the output of the first model on staying accurate to the criteria. The criteria for the generator in our GAN will be generating images which match the style of some set of images.

1.2. Style Transfer

Style Transfer is a methodology that aims to capture the style of a target style image, and to maintain the content of the target content image. In the context of our project, this means to employ convolutional networks to extract style representations from the Monet paintings. It aims to capture the dotted, smooth, single colored style of Monet's paintings. The Style Transfer algorithm uses a Style Loss function which searches for high correlation between outputted images and the target style image in terms of style. The algorithm also uses a Content Loss function which measures how much outputs differ from the target content image in terms of content. By minimizing both of these losses together, the algorithm can effectively create realistic transfers of styles.

1.3. CycleGAN

A CycleGAN [7] is a type of GAN that couples two GANs together – two generator networks, and two discriminator networks. The first generator network aims to generate images that can fool the discriminator model. The discriminator model's goal is to accurately discern if an image is real or fake. This is where CycleGAN comes in. It adds another generator that will convert the first generated image back into the original image, and calculate the cycle-consistency loss between them. The second discriminator then discerns if the second generated image is real or fake. This enables style transfer and image generation that doesn't rely on pairs of images and can create impressively accurate transfers of style upon independent images.

2. Thesis

Our goal is to try and improve CycleGAN in some meaningful way and explore many core neural network concepts in the process, especially those we learned in class. Namely, we will first attempt to build our own Generator and Discriminator architecture which will give us an opportunity to

apply what we have learned in class to CycleGAN. In addition, we plan to experiment with different fine tuning decisions, such as the loss functions and optimizer functions chosen, to come to a conclusion on which might be the best for CycleGAN, why some are better than others in this context, and if the introduction of new loss functions could even improve the accuracy and performance of CycleGAN.

2.1. Motivation

The primary motivation for this project was to gain exposure to data augmentation. Acquiring image data can be extremely difficult, expensive, and time consuming. By leveraging technologies like GANs to generate mathematically accurate images, image datasets can be augmented and models can be trained to be even more robust. Along with an interest in data augmentation and its practical applications, we simply have a passion for art. As people who enjoy spending time in the quiet serenity of museums and art galleries, this seemed like a really fun project to marry our passions for artificial intelligence, and art.

2.2. Practical Approach

To achieve this goal, we absorbed as many of the concepts from the original CycleGAN paper as we could and worked closely with public Kaggle Notebooks [4] from a Kaggle competition [2] for style transfer, which implement CycleGAN to give us a foundation for experimenting with improvement strategies.This allowed us to spend more time focusing on ways we could iterate on the model and less time reinventing fundamental parts of CycleGAN.

2.3. Roadmap

At the genesis of our project, we wanted to build the Style Transfer GAN on our own, from scratch. This was before we were assigned the homeworks regarding them and far before we even understood how to achieve Style Transfer or how GANs worked. Upon completion of these homework assignments, it became clear how these individual components worked, and it became lucid that one GAN alone would not achieve our goal and that CycleGAN was needed. We had been told prior by Professor Gan that CycleGANs are difficult, but given the necessity of CycleGANs to achieve our goal, we began working closely with public Kaggle Notebooks which implement CycleGAN to give us a solid platform upon which we could explore and experiment with the algorithm. This allowed us to focus on ways we can implement the methodologies we learned in class, apply tactics based on our own inference, and gain an understanding of CycleGANs by simply working with its various mechanisms.

3. Custom Generators and Discriminator

Our first experiment was to design our own architecture for the discriminator and generator for CycleGAN. We initially had done something similar before the mid way point of the project when we were still learning about GANs in general and created a simple architecture that produced some output while working with LINK Kaggle notebook. However, after we each finished the third assignment, we realized that performing style transfer with GANs was going to be challenging and we would need to work with CycleGANs [6].

3.1. First Attempt

Working in REF as a playground, we created Generator and Discriminator models with a shallow set of layers, most of which were some of the most common types of layers that we've seen all semester. Normalized raw photos would be passed into the Generator starting with a Convolutional layer followed immediately by anonlinear ReLU layer. Output here was followed another convolutional layer and a dropout layer with a p-value of 0.3, and finally we performed batchnorm before using a convolutional layer to produce an output.

3.2. Concept Application and Expectations

The Generator model is generating images in order to preserve the style of Monet paintings which can have many distinct features in a single image. He has a particular way of drawing different parts of nature such as trees, fields, or mountains and his way of depicting the sky is remarkable. These distinct features make his paintings an ideal candidate for convolutional neural networks, which are designed to find distinct features within an image. From a high level, we want a generator to learn what a "Monet tree" or a "Monet sky" is, so that when it is tasked with generating a real photo in the style of Monet, it just have to look for those same distinct features. As such, convolutional layers are big part of our model. Nonlinear layers are big part of any good neural network, but they are especially helpful when working with learning Monet's style of painting, as Monet does not use sharp edges and shapes in his paintings which makes it even more difficult for models to detect features in his work. Finally, our first attempt included a Batchnorm layer which we learned in class is a very useful strategy of processing data during training to improve gradient flow and reduce dependence on the initialization of the weights.

3.3. First CycleGAN Output

We ran this model for 50 epochs and graphed the loss in Figure 1. The discriminator loss remained relatively low and constantly across all epochs and we weren't able to improve the generator much with our basic model. We checked the

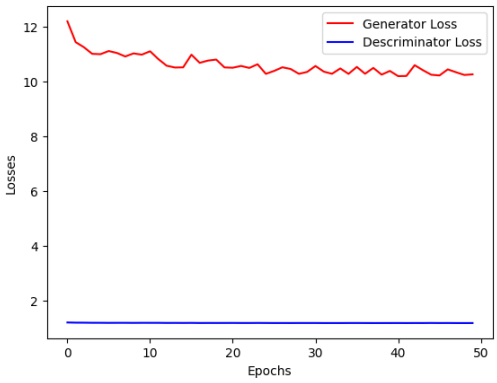

Figure 1. Plot of losses over 50 epochs for first iteration of Generator and Discriminator architecture.

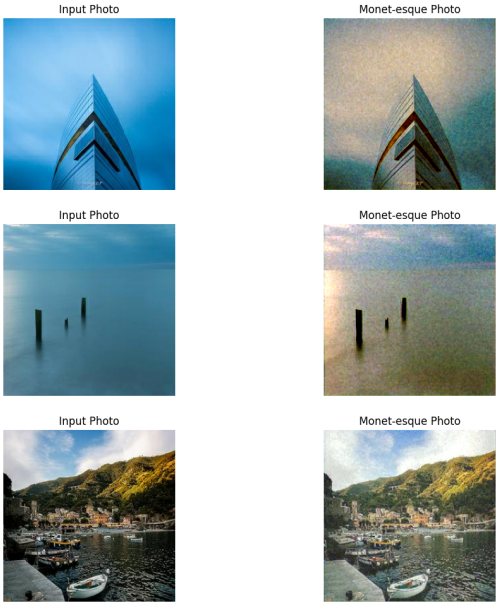

Figure 2. First Monet Style Transfer Output

generated outputs for real images in Monet style and got the results shown in Figure 2.

For a first run at creating our own CycleGAN model architecture, we weren't totally unhappy with the output of the generator. From a style transfer standpoint, the generator was able to maintain the content of the original image well, but the color scale was a bit off and there was not much to speak on when it comes to seeing a "Monet influence" on the outputs.

3.4. Applying Upsampling and Downsampling

One of the very interesting concepts we learned in class was using the structure of a fully connected network, and applying it to an image processing network to make a fully convolutional network. This approach to the network essentially processes all the pixels at once and makes predictions for each of them all at the same time. The issue with just stacking convolutional layers was that receptive field size is linear in the convolutional layers, and convolution computation on high resolution images is very expensive. The proposed solution is to downsample using strided convolutions and pooling and upsample using techniques like unpooling and biliear interpolation.

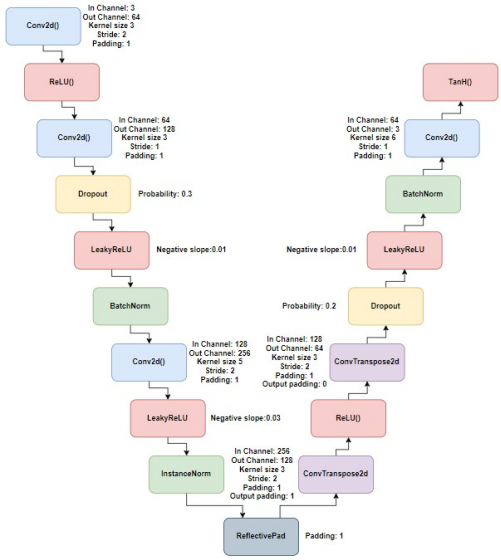

Figure 3. Improved Generator Model Architecture

3.5. Understanding Autoencoders

Autoencoders are a type of unsupervised learning networks that learns data through the downsampling layers which are primarily convolutional layers. Later the original shape is brought back by using upsampling layers such as ConvTranspose2d layers. Autoencoders perform image to image translations, which is very much suitable for our project of implementing a style transfer. They are a smart way to compress input data, distill patterns and use those patterns to produce a compressed version of the input data, reduce the noise in the data, and encourage the focus to be placed only on areas of relevance. This architecture has been shown to work effectively in CycleGANs and we wanted to try our hands at it by implementing our own autoencoder architecture for our generator model.

3.6. Sophisticated Generator Model

The downsampling layers consists of 3 Conv2d layers that increases the input channel size from 3 to 64 in the first convolutional layer and 64 to 128 in the second convolutional layer and 128 to 256 in the third convolutional layer. Each convolutional layer was followed by a ReLU or a

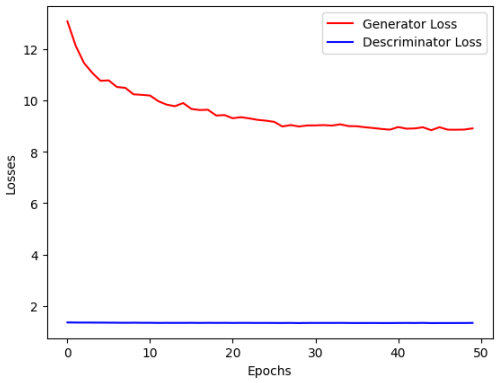

Figure 4. Loss Across 50 Epochs

LeakyReLU activation functions to give the model non-linearity. The ReLU layers where preferred due to their ability to prevent vanishing gradient and LeakyReLU to handle negative inputs as well. This is essential as the images are normalized before loading it to the model. Normalization techniques applied on the images include subtracting each channel in the image by 0.5 and dividing by 0.5. This helps in having the pixel values range from [-1,1]. We added a reflectionpad layer of size 1 on each side to increase the receptive field. After that we added upsampling layers which consists of two ConvTranspose2d layers that decreases the input channel size from 256 to 128 in the first layer and 128 to 64 in the second layer. Finally a convolutional layer has been added to change the channel size from 64 to 3 which is equal to the number of channels in the original image. The dropout layers were added during downsampling and upsampling to regularize the values in the tensors. Activations functions such as ReLU(), LeakyReLU() and Tanh() functions were added to provide non-linearity.

This model helped us reduce our loss and saw improvements which were more stable, but the loss was still fairly high and the output were not in the realm we wanted them to be in. We initially used Adagrad as our optimizer for this training session and we decided to try RMSProp next.

3.7. Iterate Discriminator and RMSProp

The role of a discriminator is to classify whether the images generated by the generator are real or fake. It plays the role of a classifier which tells the generator about how real and fake an image is. If the discriminator is very strong, no matter how good the generated image is the discriminator will always say it is fake which leads to no learning for the generator. And if a discriminator is bad, then the discriminator will always say its real no matter how bad the generated image is.

We decided to add an additional convolutional layer to our discriminator as well as two additional nonlinear layers in hopes of creating a discriminator which knows what

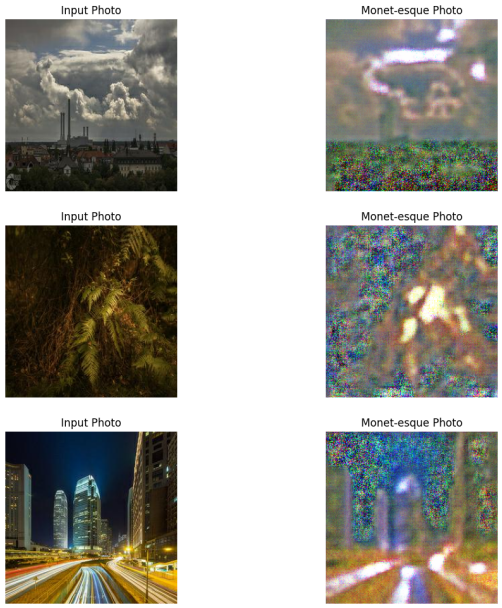

Figure 5. Updated Generator Output

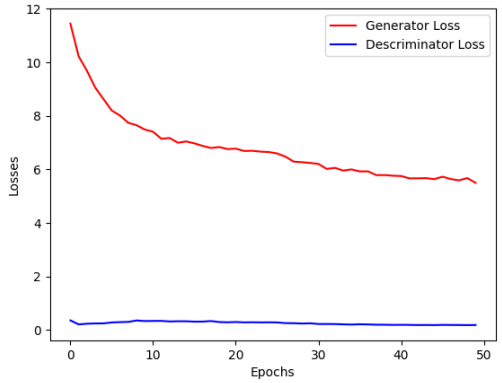

Figure 6. Loss after expanding Generator and Discriminator models

"Monet style" is and would be more informative when calling outputs as real or fake. Thus our overall discriminator discriminator model now has 3 Conv2d layers where the first one increases the input channel from 3 to 64, the second one increase the channels from 64 to 128 and third one reduced the channels from 128 to 1 to do a binary classification. The layers were activated using ReLU and LeakyReLU activation functions, the dropout layers and Batchnorm layers were added to regularize and generalize the tensors. We also decided to use RMSProp as our optimizer after the Adagrad results for this iteration where still not where we wanted them to be.

??

Generator loss made a big improvement after fleshing out the discriminator but more importantly, the actual out-

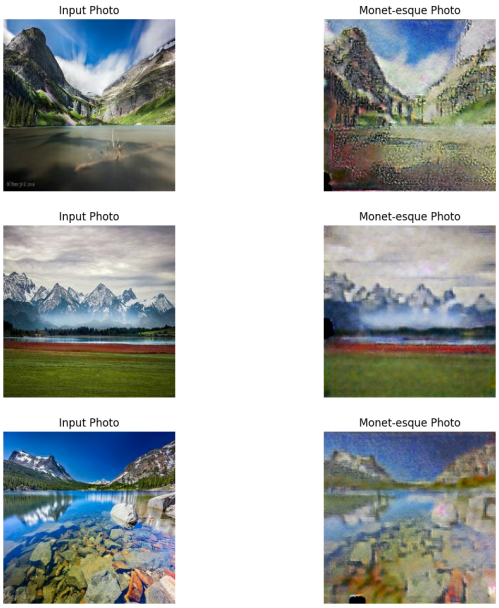

Figure 7. Updated Model Output

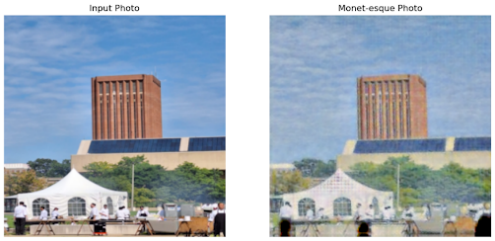

Figure 8. Test Output of Photo of Du Bios Library

puts were showing quite a bit of successful Monet style transfer. When we were happy with our output, we tried a handful of real photos that we took and feed them into our generator. Our best one was this photo of the Du Bois library which we felt was one of the best generated outputs which was great to see.

3.8. Learning Opportunities

We spent a lot of time on this portion of the project from debugging model creation, to learning autoencoder and upsampling more deeply, and finally to training for 50 epochs each. Creating our architecture from scratch helped us understand the CycleGAN paper much more deeply as well as some of the more complex concepts from the CNN part of the course. Although we were not able to improve on the results of CycleGAN with Monet paintings, we are proud of the results we were able to generate with our own generator and discriminator architectures and feel we were able to preserve the Monet painting style. However, since we were not able to make an improvement here, we wanted to move on to explore other avenues of improvement for CycleGAN.

4. Integrating Style Loss

Our next choice was looking at the loss function from the original paper and seeing what would happen if we introduced an additional loss function which was targeted at preserving the style of Monet paintings.

4.1. CycleGAN Loss

The CycleGAN Loss is the combination of three different losses: Adversarial Loss, Cycle Consistency Loss, and Identity Mapping Loss. One of the reasons GANs alone can be so successful is largely due to its Adversarial Loss. The Adversarial Loss in essence is the difference between generated images and real images. Minimizing the adversarial loss thus effectively pushes the generated images to be as realistic and mathematically similar to the real images. One of the purposes of the second generator is to take the first generated image, feed it into the second generator, and attempt to recreate the original image. This is where Cycle Consistency Loss comes in. It compares the original image with the second generator's generated image and aims to drive the differences between them down by minimizing the cycle consistency loss. This is one of the most important aspects of the CycleGAN because this loss function enables training to be executed without the necessity of image pairs. Image pairs are two separate images that have the same exact structure – imagine a hand-drawn image of Mount Everest, and then a photograph of Mount Everest from the exact same position and angle. In traditional style transfer, image pairs are needed. CycleGANs effectively remove this necessity with the cycle consistency loss, which is an incredibly huge dependency removed. The final loss function is Identity Mapping Loss. Identity mapping is the concept of an input and an output having the same identity, or being the exact same. In the context of CycleGANs, the generator tends to confuse lightings like sunset and daytime which have a slightly different filter to humans, but can go relatively undetected by adversarial and cycle consistency loss functions. By introducing identity mapping loss, the generators of the CycleGAN are further bolstered in terms of the realism of their generations.

4.2. Style Transfer Background

Although our focus for this project was to work with GANs, there is a more pointed neural network strategy for working with style transfer specifically [1]. We covered neural style transfer in class, especially during homework 3, and we even worked with generating real photos in the style of various painters. Neural Style Transfer makes used of Gram matrices to calculate the loss of the style and content of generated images in order to train a model to perform style transfer.

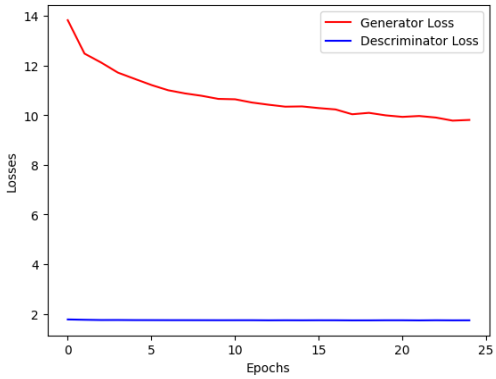

Figure 9. Model loss with Style Loss included

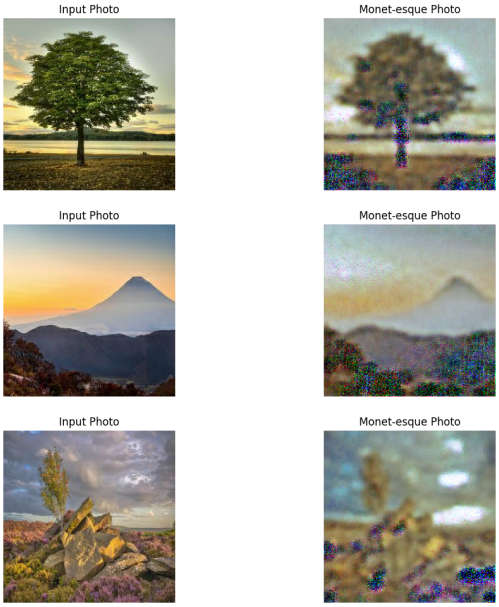

Figure 10. Model Output with Style Loss included

4.3. Expanding the CycleGAN Loss Function

We were curious how our model may perform if we relied on the three original parts of the CycleGAN loss function for preserving the content of the original real images and trying to preserve style by expanding the loss function to include the style loss function.

4.4. Results

The loss took a noticeable step back compared to our best model as it rose back up above 10 and the outputs were much more noisy with a dull color palette.

4.5. Conceptual Understanding of Failure

The failure of this change demonstrates how carefully the loss function in CycleGAN was put together and that there may not be a need to refine it . Additionally, pulling the style-loss function out of the context of strict neural style transfer is messy since style loss is best utilized when paired with context loss and total variation loss. Ultimately, adding this loss to the model was an experiment to see how expanding the loss function would impact our output, and the most logical loss function we thought to apply was style loss. After learning quite a lot about GANs and style transfer to this point, we see now that incorporating style loss to CycleGAN would almost defeat the purpose. CycleGAN can do the work of neural style transfer while being broad to perform other tasks accurately as well.

5. Optimizer Function Experimentation

Optimization in neural networks is the process of searching for the best set of parameters for a model to minimize its loss and predict as accurately and consistently as it can. Optimizers execute this task by aiding in the search for the best weights and biases for the model to predict most accurately. After finding the variation of our model architecture which produced the best results, we began to experiment with different optimization functions and documented how they affected loss and the output.

5.1. RMSProp

RSMProp, or Root Mean Squared Propagation is much like other gradient descent algorithms in that it calculates gradients based off of the loss functions and uses the gradients to tune model parameters as it descends in search for the local and global minima that will minimize the model's losses. The way RMSProp differs from classic gradient descent is that it uses a moving average of squared gradients to to scale the learning rate of each parameter. This effectively contributes to further stabilizing the learning process and can prevent vacillations in the optimization process. RMSProp is actually a tweak/adjustment to AdaGrad that attempts to reduce its aggressive, monotonically decreasing learning rate. RMSProp was the optimizer function we used during all of our attempts at creating a useful CycleGAN model with our own architecture of Generator and Discriminator layers. We originally chose this optimizer as our baseline in order to take a separate approach from the one taken in the original Kaggle notebook, but it also ended up being the one which produced the lowest loss and the best qualitative generated outputs.

5.2. Stochastic Gradient Descent

Stochastic Gradient Descent, or SGD, is an iteration of classic gradient descent in that it evaluates the gradients for one example and then updates the weights. Classic gradient descent goes through the entire dataset before it calculates gradients and updates the weights. Naturally, that is very computationally expensive. Another update provided by SGD

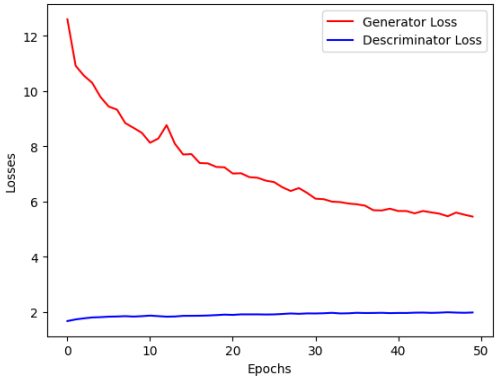

Figure 11. Loss for Model trained with SGD

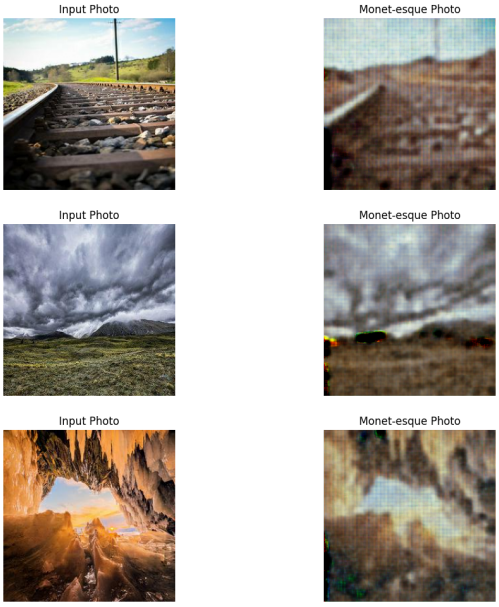

Figure 12. Output trained with SGD

is the inclusion of physics via momentum in descent. Momentum and physics brings the idea of potential energy, and the force felt by the particle is the negative gradient of the loss function. Because force = mass * acceleration, the negative gradient is proportional to the negative acceleration of the particle. With these updates, Stochastic Gradient Descent provides a clearly faster solution to descending upon minima. When we trained our model using SGD as our optimizer function, we got the results in figure .

SGD did a great job at minimizing loss, but it did not to a very good job of creating outputs which demonstrated the style of Monet paintings. We suspect the reason that SGD sees a decline in qualitative results compared to RSMProp is that CycleGAN benefits from RMSProps inclusion of a moving average and the basic SGD quite sophisticated enough to handle the demands of CycleGAN

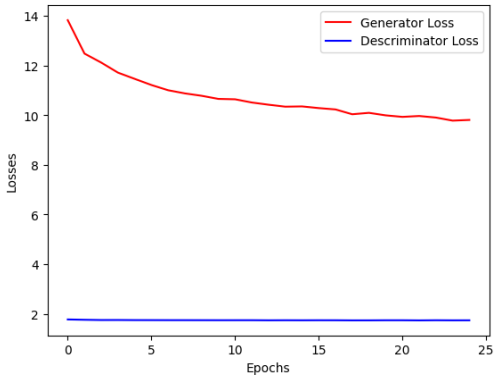

Figure 13. AdaGrad Loss

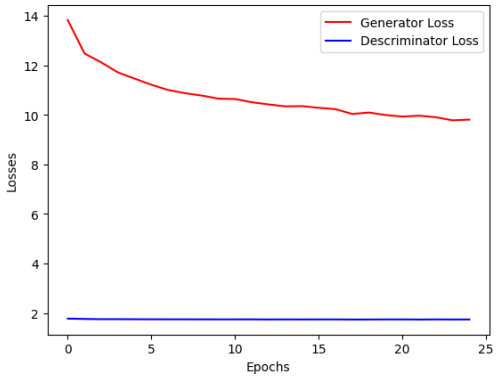

Figure 14. AdaGrad Output

5.3. AdaGrad

AdaGrad, or Adaptive Gradient, is an iteration of Stochastic Gradient Descent. It develops upon its parent optimizer by adapting the learning rates for each feature depending on the estimated geometry of the task at hand. It assigned smaller learning rates to frequently occurring features, and larger learning rates to more infrequently occurring features. This effectively ensures parameter/weight updates occur based on relevance, not so much frequency. Another way to say this, is that weights that receive high gradients will have their learning rate reduced. Weights that receive small or infrequent updates will have their learning rate increased.

The loss for our mode using AdaGrad is much higher than SGD, but the color palette of these outputs is about as close to what we expect of Monet painting as we had with SGD. The content is noisy, like SGD, and falls short to our trained model of RMSProp.

5.4. Adam

Adam is another iteration of Stochastic Gradient Descent that can effectively serve as an alternative to SGD. It is similar to RMSProp, but it uses a smoothed version of the gra-

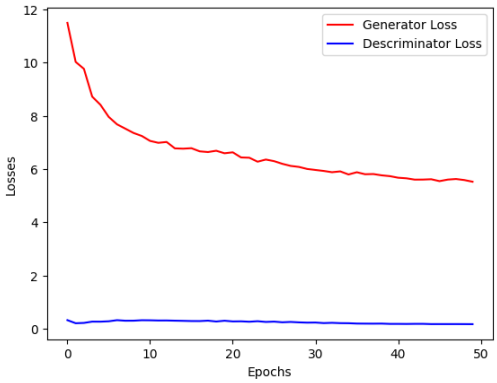

Figure 15. Adam Optimizer Loss

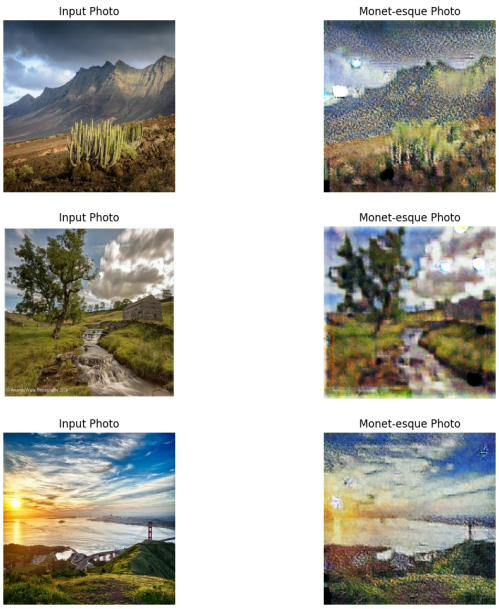

Figure 16. Adam Optimizer Output

dients and includes a bias correction mechanism which corrects for initialized values that bias toward zero in the first few steps of descent. These updates have shown to help the optimizer in generalization. It is currently one of the most widely used optimizers.

Adam Optimizer achieved low loss and had outputs that looked better qualitatively compared to SGD and AdaGrad. The original Kaggle notebook used Adam as their optimizing function and got better results than these with their model architecture but color palette here is very Monet-esque. The main issue here is the results are a bit more noisy than our results from RMSProp and the shape features are a bit more sharp compared to what Monet paintings are like.

5.5. Optimizer Summary

Each of these optimizer functions performed fairly similarly in terms of loss and the outputs saw much less drastic output variations than the ones we saw from training two different model architectures, demonstrating that while optimizer functions are important, the quality and coherence of the generator and discriminator models you point together may matter more.

6. Frechet Inception Distance Evaluation

The Frechet Inception Distance [3] is a strong evaluation method for comparing the output of generative models to their target. The Frechet Inception Distance combines the concept of the Inception distance which is a method for evaluation output of generative models by feeding outputs into a pretrained model, and of Frechet Distance which compares two gaussians. FID calculates the covariance of samples of real and fake images and uses it as the sigma for a gaussian in a Frechet Distance calculation. It sought to improve on simply using the Inception score by reducing the variance seen by the pre-trained model while still maintaining the inception distance's usefulness in modeling human interpretation of results being "good".

6.1. Challenges with FID in Practice

Originally in our project proposal we had planned to use the Frechet Inception Distance as our quantitative evaluation metric for the strength of the model we created. However, after we were satisfied with our model and began to explore FID in code, we came across challenges getting a solution built out. We tried to implement the method from scratch [5] using Numpy and SciPy but this proved to be both conceptually complicated as well as very memory intensive, as our testing environments would frequently crash while trying to implement this solution. Although we were not able to implement Frechet Inception Distance successfully by the deadline, we learned quite a bit in the process and felt it would still be worth recording our effort.

7. Conclusion and Takaways

Generative AI is something that was deeply interesting to all three of us early on in the semester because it has such an important practical application to the ever changing landscape of machine learning, while also being fairly new. We weren't fully aware about how complicated working with CycleGAN would be when we started this project but we're all glad we chose this avenue as we learned a tremendous amount from our various experiments. The challenging nature of working with this model gave us a lot of room to explore, experiment, apply concepts, and learn.

7.1. Creating our one Architecture

Although generating an improvement to CycleGAN by training our own generator and discriminator models from scratch was unrealistic, we wanted to start there so that we would not only get an opportunity to apply concepts we learned in class to our own model, but also so that our later experiments and analysis would be built off a foundational model architecture that we created. We were able to iterate on the model to a point of generating results which were Monet-like and strengthened the foundation of our model design skills which we worked on all semester. Additionally we had an opportunity to apply concepts we hadn't used as much in homework assignments such as autoencoding and upsampling/downsampling, which we took a lot of value from.

7.2. Experimented with Expanded Loss Function

Expanding the loss function was our first idea of an area of CycleGAN which could potentially be improved on and our intuition was to integrate the style loss function as an expansion. After we got the results from this experiment however, it became clear that the loss function was carefully crafted to begin with, and we would need a very specific reason to change the loss function in any way in order to improve it. Additionally we understood that the style loss may not be as effective outside of the explicit context of neural style transfer, even though our focus was preserving style transfer here as well.

7.3. Comparing Optimizer Functions

Comparing optimizer functions was both an exercise in fine tuning our model to see what works best, as well as recording our results and making comparisons between functions. Our expectations were somewhat subverted during this process as we originally expected the Adam optimizer to perform the best, since it seems to be very commonly used and was even the function chosen by the authors of the original Kaggle notebook we worked with. That did not end up being the case however, and we got some insight into how different optimizer functions can be more of less useful under different model architectures. Additionally, in order to perform the analysis we had to spend some time reading up on the pros and cons or each function and trying to apply those concepts to the results we were seeing, which helped us flesh out our concept of what an optimizer is and why it's so useful.

References

[1] 14.12. neural style transfer. 2021. 5

[2] I'm something of a painter myself., 2023. Kaggle Competition of Monet Style Transfer. 2

[3] Martin Heusel, Hubert Ramsauer, Thomas Unterthiner, Bernhard Nessler, and Sepp Hochreiter. Gans trained by a two time-scale update rule converge to a local nash equilibrium, 2018. 8

[4] ROBIN LUTTER. Painting with gans, 2023. CycleGAN Notebook on Kaggle. 2

[5] Alexander Mathiasen and Frederik Hvilshøj. Backpropagating through fréchet inception distance, 2021. 8

[6] Filipa M. Ramos. Optimizing cyclegan in the context of monet painting generation. *16th Doctoral Symposium in Informatics Engineering.*, 2021. 2

[7] Jun-Yan Zhu, Taesung Park, Phillip Isola, and Alexei A Efros. Unpaired image-to-image translation using cycle-consistent adversarial networks. In *Computer Vision (ICCV), 2017 IEEE International Conference on*, 2017. 1