Ishmael Obeso
December 30, 2019

**Udacity Machine Learning Engineer Capstone Report**

*Generating Images with Brushstrokes using Generative Adversarial Networks*

# I. Definition

## I. Overview

The purpose of this project is to develop a way to generate images with brushstrokes similar to human artists. In order to accomplish this, I trained Generative Adversarial Networks (GANs) to generate brushstrokes, then I trained them to "paint" a series of brushstrokes on a canvas in order to imitate an input image. This process is similar to a human artist painting a subject that is sitting in front of them onto a canvas, both the human artist and the Brushstroke GAN output an image that is not identical to the input image, rather it is an artistic impression of the input image. It is useful to think of this as the difference between a sunflower and how Vincent Van Gogh paints a sunflower; the GAN will take an input image of a sunflower and output a painting of the flower.

*DCGAN | Original | Brushstroke*

I wanted to develop this project because I was interested in GANs, but it seemed that the traditional Deep Convolutional GAN (DCGAN) architecture was inadequate for generating aesthetically pleasing images. DCGANs are traditionally better at imitating an input image, but the output images tend to have artifacting and pixel static that make the output images seem unnatural. This project was designed to create a method for generating aesthetically pleasing images using simple GAN architectures and small datasets, without resorting to large GAN architectures such as StyleGAN that require huge amounts of computing power.

The only datasets utilized were relatively small compared to traditional GAN-training datasets such as CIFAR-10 and ImageNet. The first dataset used was the set containing the brushstroke-training data. It consists of 100,000 example brushstrokes which the Brushstroke GAN was trained to imitate. The second dataset was a set of 8,683 images used for training the DCGAN and Brushstroke GAN models to generate images. Both GANs trained on a single image at a time for a set amount of epochs and the outputs were compared to see which GAN was able to create a more aesthetically pleasing and accurate output image.

The final product is a Flask Web app that a user can submit images to and get back an image from the Brushstroke GAN as well as a gif of the GAN painting the image.

## II. Problem Statement

With the recent advances in deep learning we are seeing more and more computer-generated images being used in things like advertising, web design, and fashion. However, it is still relatively easy to spot when a computer-generated image is being used, especially when creating painting-like images. These images tend to be created using style transfer and GAN's that build the image pixel-by-pixel, but if we want to create convincing computer-generated paintings we need to start using human-like techniques. In this way, neural painters imitate human artists by using brushstrokes on a canvas to create convincing art.

However, neural painters based on Variational AutoEncoders(VAE's) have not yielded good results, as they are simply not able to recreate the fine details of brushstrokes. In addition, since this breakdown happens even on simple images such as the ones in the MNIST database, a new method will

need to be used to create neural painters that can recreate complex images such as paintings and pictures.

This new method will be implemented using GAN's trained to imitate brushstrokes. These "Brushstroke GANs" will then be given an input image to train train on, with the goal being to optimize the brushstrokes painted on the canvas to accurately imitate the input image. These Brushstroke GANs will be able to quickly imitate an input image, but the outputs will never be a direct copy of the input image as the brushstrokes used will create a painting-like style in the output image. However, even with this limitation in mind, the Brushstroke GANs will still be able to produce more accurate and aesthetically pleasing output images than a traditional image-generation GAN, such as a DCGAN.

### III. Metrics

Evaluating the performance of GANs has historically been difficult, as there are not many ways to quantifiably measure the quality of images. Comparision between two GAN models is usually measured by visual inspection of the output images. However, I believe I have found an appropriate evaluation metric for this problem, Perceptual Loss (aka Feature Loss). This loss is based upon research done in the paper Losses for Real-Time Style Transfer and Super-Resolution, and the implementation in Python is based on the FastAI course v3 Lesson 7 implementation.

It works by feeding an input image and a target image into a pretrained image classifier, such as the Vgg16 network I used in this project, and then taking the activations of the network at different layers and comparing the activations between the images. This essentially takes the "features" that the network sees in the two images, and compares the features of the two images. The Feature Loss is thus calculated by taking the L1 losses of the feature maps generated by the Vgg16 network at each layer between the input and target image and adding them together.

feat_losses = l1_loss(input_image_features, target_image_features) for input_image_features, target_image_features in Vgg16.layer.weights

Feature Loss = sum(feat_losses)

References:

-*Fast.ai MOOC Lesson 7: Resnets from scratch; U-net; Generative (adversarial) networks*. https://course.fast.ai/videos/?lesson=7 ; Notebook: https://nbviewer.jupyter.org/github/fastai/course-v3/blob/master/nbs/dl1/lesson7-superres.ipyn

-Perceptual Losses for Real-Time Style Transfer and Super-Resolution - Justin Johnson et al

## II. Analysis

### I. Data Exploration

This project makes use of two relatively small datasets. The first is a dataset containing 100,000 images and parameters of brushstrokes. The second is a dataset containing 8,683 paintings by famous artists.

Brushstroke Data-

```
#load brushstroke data into memory
data = np.load('brushstroke_data/' + 'episodes_42.npz')
stroke_data = data['strokes']
action_data = data['actions']
```

```
# explore action data
print(f'The dataset contains: {action_data.shape[0]} examples of brushstroke actions')
print(f'Each example contains: {action_data.shape[1]} parameters for a single brushstroke')
print(f'Example of a single actions parameters: {action_data[0]}')

# each action vector contains 12 parameters for a single brushstroke
# these parameters consist of - Start and End Pressure (param 1,2)
#                            - Brush Size Radius (param 3)
#                            - RGB Color (param 4,5,6)
#                            - Brush Coordinates, Start, End, Intermediate (param (7,8),(9,10),(11,12))
```

```
The dataset contains: 100000 examples of brushstroke actions
Each example contains: 12 parameters for a single brushstroke
Example of a single actions parameters: [0.33052152 0.24929554 0.63813394 0.6562682  0.53990434 0.43563418
 0.62354892 0.49426813 0.38073039 0.89612729 0.92475446 0.46671069]
```

```
# visualize stroke data
# total number of strokes in dataset
print(f'This dataset contains {stroke_data.shape[0]} strokes')
print(f'Each stroke contains {stroke_data.shape[1]} by {stroke_data.shape[2]} pixels and {stroke_data.shape[3]} colo

# visualizing a stroke in dataset
example_stroke = stroke_data[5][:,:]
plt.grid(False)
plt.imshow(example_stroke, cmap='Greys')
plt.show
```

```
This dataset contains 100000 strokes
Each stroke contains 64 by 64 pixels and 3 color channels
```

*Brushstroke Data Exploration*

Painting Data -

```
# explore Image Dataset

# batch size
batch_size = 1
root = 'painting_data'

# transform
transform = transforms.Compose([
        transforms.Resize(64),
        transforms.CenterCrop(64),
        transforms.ToTensor(),
        ])

# create dataset from images in ImageFolder
data = torchvision.datasets.ImageFolder(root=root, transform=transform)

#create sampler
sampler = RandomSampler(data, replacement=True, num_samples=100)

# create dataloader for data
dataloader = torch.utils.data.DataLoader(data, batch_size = batch_size, sampler=sampler)
```

```
# print out info on data and dataloader
print(f'There are {len(data)} images in this dataset')
print(f'We have taken a sample of {len(dataloader)} images from this dataset')
```

```
There are 8683 images in this dataset
We have taken a sample of 100 images from this dataset
```

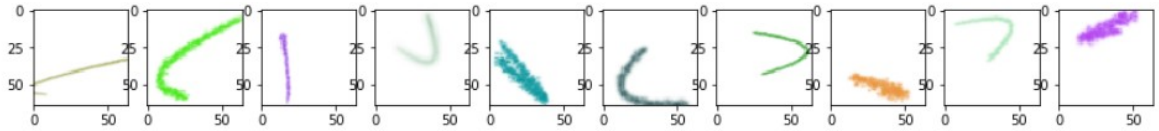
*Painting Data Exploration*

Sources: https://www.kaggle.com/reiinakano/mypaint_brushstrokes
https://www.kaggle.com/ikarus777/best-artworks-of-all-time

## II. Exploratory Visualization

-Brushstroke dataset visualization

```
# visualizing multiple strokes in dataset
num_strokes = 10
random_sample = np.random.choice(stroke_data.shape[0], num_strokes)
stroke_subset = []

# take a random sample of num_strokes strokes
for i in random_sample:
    stroke_subset.append(stroke_data[i])

# plot the brushstrokes
fig = plt.figure(figsize=(30, 10))

i=0
for stroke in stroke_subset:
    i += 1
    fig.add_subplot(1, 20, i+1)
    plt.grid(False)
    plt.imshow(stroke)

plt.show()
```



*Brushstroke Data Visualization*

-Painting dataset visualization

```
# visualizing single image from dataset

def imshow(imgs):
    imgs = torchvision.utils.make_grid(imgs)
    npimgs = imgs.numpy()
    plt.figure(figsize=(8,8))
    plt.imshow(np.transpose(npimgs, (1,2,0)), cmap='Greys_r')
    plt.xticks([])
    plt.yticks([])
    plt.show

dataIter = iter(dataloader)

imgs, labels = dataIter.next()

imshow(imgs)

print(imgs.shape)
```

```
torch.Size([1, 3, 64, 64])
```

*Painting Data Visualization*

## III. Algorithms and Techniques

This project makes use of four main techniques and algorithms in training the GAN models: Binary Cross-Entropy, L1 loss, MSE loss, RMS prop, and ADAM.

- *BCE Loss*

- This loss is used in training of the DCGAN, and is used as the adversarial loss function . During training of the DCGAN, the discriminator and the generator are competing, the discriminator is trying to classify whether an image is original or generated by the Generator while the Generator is trying to create images that trick the discriminator into thinking they are original. BCE loss is used to measure how accurately the discriminator classifies images are fake or original. Using this BCE loss, the discriminator is then able to optimize itself to better classify the images. At the same time, the generator uses the BCE loss to judge how well it was able to trick the discriminator, then it optimizes itself to better generate images that trick the discriminator.

- BCE Loss takes as input the predicted class probabilities, and compares them to the real classes. Then it outputs the negative log of the probabilities, and computes the mean of all the losses. The formula is defined by

$$H_p(q) = -\frac{1}{N}\sum_{i=1}^{N} y_i \cdot log(p(y_i)) + (1 - y_i) \cdot log(1 - p(y_i))$$

where y is the label and p(y) is the predicted probability of the data being the label y.

- *L1 Loss*

- This loss was used in training the Brushstroke GAN to imitate brushstrokes initially, but it failed to capture the features of the brushstrokes. As a result the GAN did not imitate the brushstrokes, and instead output black boxes.

- The L1 loss function is the Least Absolute Deviations function, where it is used to minimize the error by summing up all the absolute differences between the true value and the predicted value. In this instance, the L1 loss took in as input all the pixel values from the generated image, and compared them to the pixel values of the original image. However, this function was not able to correctly train the GAN.

$$L1LossFunction = \sum_{i=1}^{n} |y_{true} - y_{predicted}|$$

-*MSE Loss*

- This loss was used in training the Brushstroke GAN to imitate brushstroke after the L1 loss produced inadequate results. It was able to train the GAN to properly imitate brushstrokes from the brushstroke dataset. However, its weakness is that it takes a couple of hours of training to do this properly. A better loss function might result in faster training.

- The MSE Loss stands for the mean squared error loss, which measures the average of the squares of the errors between the estimated value and the actual value. In this case, it measures the pixel values between our output and original images and calculates the MSE loss from those values.

$$MSE = \frac{\sum_{i=1}^{n} (y_i - y_i^p)^2}{n}$$

Where yi is the original image pixel value and yi(p) is the output image pixel value.

- *RMSprop*

- RMSprop was used as an optimizer function in this project. It was used after the Brushstroke GAN learned to imitate brushstrokes, and it optimized the brushstroke parameters (actions) in order to imitate an input image. So essentially, it was used to tell the model which brushstrokes it should put on the canvas in order to imitate an image.

-RMSprop is an optimizer function that uses the average of squared gradients to normalize the gradient created by the GAN. In this instance it is not used to update the weights of the GAN, but the action space of the brushstrokes placed on the canvas.

$$For\ each\ Parameter\ w^j$$
$$(j\ subscript\ dropped\ for\ clarity)$$

$$\nu_t = \rho\nu_{t-1} + (1-\rho) * g_t^2$$
$$\Delta\omega_t = -\frac{\eta}{\sqrt{\nu_t + \epsilon}} * g_t$$
$$\omega_{t+1} = \omega_t + \Delta\omega_t$$

$\eta : Initial\ Learning\ rate$
$\nu_t : Exponential\ Average\ of\ squares\ of\ gradients$
$g_t : Gradient\ at\ time\ t\ along\ \omega^j$

-*ADAM*

-Adam was used as an optimizer for the generator and discriminator of the Brushstroke GAN during training on the brushstroke dataset, and it was also used as an optimizer for the generator and discriminator for the DCGAN through the whole project.

- Adam is an optimizer function that is based on stochiastic gradient descent, but incorporates some important improvements. In particular, it is able to dynamically change the parameter learning rates based on the moving averages of the gradient and the squared gradient. This results in faster convergence and good results, which is why it is popular to use.

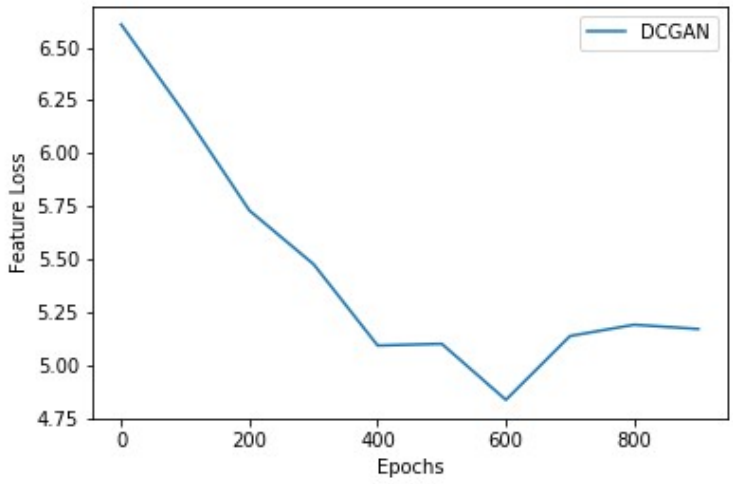
$$For\ each\ Parameter\ w^j$$
$$(j\ subscript\ dropped\ for\ clarity)$$

$$\nu_t = \beta_1 * \nu_{t-1} - (1-\beta_1) * g_t$$
$$s_t = \beta_2 * s_{t-1} - (1-\beta_2) * g_t^2$$
$$\Delta\omega_t = -\eta\frac{\nu_t}{\sqrt{s_t + \epsilon}} * g_t$$
$$\omega_{t+1} = \omega_t + \Delta\omega_t$$

$\eta : Initial\ Learning\ rate$
$g_t : Gradient\ at\ time\ t\ along\ \omega^j$
$\nu_t : Exponential\ Average\ of\ gradients\ along\ \omega_j$
$s_t : Exponential\ Average\ of\ squares\ of\ gradients\ along\ \omega_j$
$\beta_1, \beta_2 : Hyperparameters$

## IV. Benchmark

The benchmark model in this project is a traditional GAN architecture known as a Deep Convolutional GAN. These types of GANs are used to generate images and they are relatively simple. However, the training of these GANs is often unstable, as the discriminator and the generator have a tendency to get stuck in local minima, stopping training or leading to odd outputs. In addition, even when training goes well, the output images are often slightly off to the human eve and typically contain pixel “snow” or other artifacting that is easy to notice. For this reason, it is a good benchmark as it is commonly used and easy to train. I will use the DCGAN output image, and compare it to the input image using the FeatureLoss metric explained earlier. If my Brushstroke GAN model can score lower on average in the FeatureLoss metric than the DCGAN, I will call it a success since that means the Brushstroke GAN image features are closer to the original image features than the DCGAN.




*Feature Loss of DCGAN at every hundredth epoch*

## III. Methodology

### I. Data Preprocessing

- Brushstroke Data Preprocessing

The Brushstroke Dataset needs to be split into seperate “stroke” and “actions” datasets. Then the actions and strokes are stacked together to form two seperate tensors.These two tensors are loaded into a Pytorch dataset and subsequent dataloader. During training, this data will also need to be transformed so the RGB pixel data is in a standard format.

```
#load data into memory
data = np.load('brushstroke_data/' + 'episodes_42.npz')
stroke_data = data['strokes']

# define batch size
batch_size = 256

# define dataloaders and dataset used in training
actions = data['actions']
actions = torch.stack([torch.from_numpy(action).float() for action in actions])
strokes = torch.stack([torch.from_numpy(stroke).permute(2,0,1) for stroke in stroke_data])

# use generic pytorch datasets and dataloader classes
dataset = TensorDataset(strokes, actions)
dataloader = DataLoader(dataset, batch_size = 256, shuffle=False)

# define stroke transformer we will use during training
strokes_tfms = transforms.Compose([transforms.Lambda(lambda x: x.float()),
                                   transforms.Lambda(lambda x: x.div(255))
                                  ])
```

*Brushstroke Data Preprocessing*

- Painting Dataset Preprocessing

Before it can be used in training, the Painting dataset needs to be resized, cropped, and then turned into a tensor by a Pytorch transformer. Then the images are loaded into a pytorch ImageFolder dataset, and random sample of size num_samples of images are taken from this dataset and placed into a dataloader for training. In my case I have chosen 100 random images to be loaded into the dataloader for training.

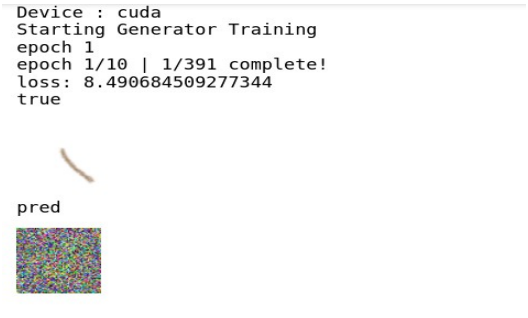
```
# batch size
batch_size = 1
root = 'painting_data'

# transform
transform = transforms.Compose([
        transforms.Resize(64),
        transforms.CenterCrop(64),
        transforms.ToTensor(),
        ])

# create dataset from images in ImageFolder
data = torchvision.datasets.ImageFolder(root=root, transform=transform)

#create sampler
sampler = RandomSampler(data, replacement=True, num_samples=100)

# create dataloader for data
dataloader = torch.utils.data.DataLoader(data, batch_size = batch_size, sampler=sampler)
```

*Painting Data Preprocessing*

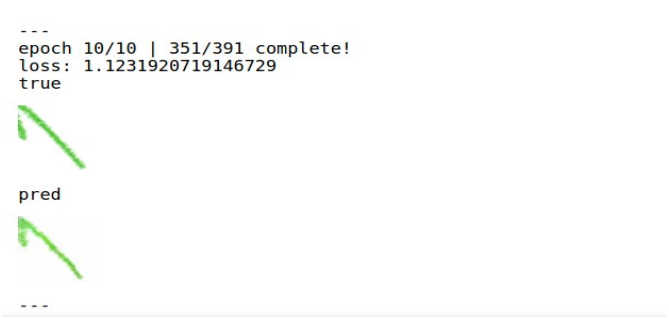
## II. Implementation

The implementation of this project was done in six steps.

1. Training Brushstroke Generator Non-Adversarially
    1. First, a generator model was created by combining the fast.ai basic_generator model with a fully connected layer. Then the model was trained non-adversarially by using mse_loss and the ADAM optimizer to train the generator to imitate the brushstrokes from the dataset. This resulted in fast convergence toward good imitations of the brushstrokes. Then the model was saved. This process is documented in the notebook “1. GAN_train_non_adversarial”.




*Generator Output at train start*

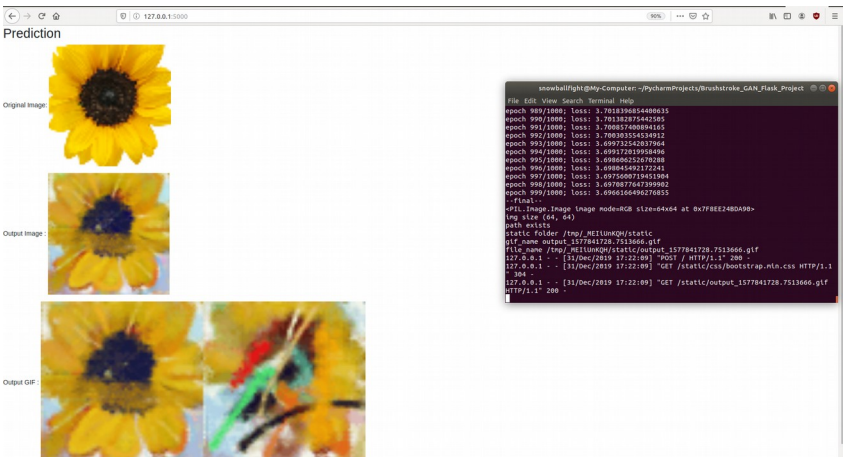



*Generator Output at train end*

2. Training Brushstroke Discriminator Non-Adversarially
    1. Then a discriminator model was created by using the fast.ai gan_critic model. This discriminator was then fed images of original brushstrokes and images of brushstrokes generated by the pretrained generator from the first step. The discriminator was tasked with classifying whether a given brushstroke was real or generated, and used BCE loss and the ADAM optimizer to train over the given dataset. This resulted in a discriminator that was able to consistently differentiate between real and fake brushstrokes. Although in order to reach this level of discrimination took multiple hours of training time. This process is documented in the notebook “2. Critic_train_non_adversarial”.
3. Training Brushstroke GAN adversarially
    1. After pretraining both the discriminator and generator, they were then trained adversarially like a traditional GAN in order to fine-tune their outputs. This resulted in the generator and discriminator “fighting” each other and creating better outputs. The generated brushstrokes ended up looking much better and captured more of the original brushstrokes features. The discriminator ended up not being able to tell the generated strokes from the original strokes easily, meaning that the generator was producing better strokes than before. This process is documented in the notebook “3. GAN_train_adversarial”
4. Creating DCGAN and Analyzing Outputs
    1. This step was not strictly necessary for implementation. I used it in order to analyze how DCGAN outputs normally look when trained on an image. I concluded that a DCGAN is typically pretty unstable, and the training fails often because either the discriminator or generator will get stuck in a local minima and stop training properly. This is known as a “failure mode” when training GANS. I also concluded that the DCGAN image outputs were not aesthetically pleasing to the human eye, and often contained “pixel snow” and odd segmentation lines that resulted in me easily being able to tell that the image was generated. This process is documented in the notebook “4. DCGAN_analyze_outputs”

*Example DCGAN Output (left) Original (right)*

5. Comparing DCGAN Outputs with Brushstroke GAN Outputs
    1. After creating and training both the DCGAN and Brushstroke GAN, I then trained them both on a set of images at the same time, and compared their output images. These output images were compared to the original image using FeatureLoss to quantify how well each GAN imitated the input image. I found that my Brushstroke GAN was able to generate better, more aesthetically pleasing images than the DCGAN. I then took the average FeatureLoss of each GAN over every image at every hundredth epoch and output those feature losses into a chart. This chart clearly shows how the Brushstroke GAN creates better images than the DCGAN. This process is documented in the notebook "5. Compare_DCGAN_with_Brushstroke_GAN".
6. Creating a Flask App
    1. After I created a Brushstroke GAN model I was happy with, I then went on to create a Flask web app where I could submit an image to the flask server and receive back a generated image from the Brushstroke GAN as well as a gif of the GAN painting the image stroke-by-stroke.
    2. I wanted to host the server on a remote server, so I would simply link to the website, but the models were too computation-heavy to run smoothly on a rented server. I then tried to host the model with AWS services, but since the model takes longer than 60 seconds to create an output image, the AWS lambda service would time out. As a result of this, I decided to simply host the flask server on my own machine, and then bundle the flask app into an executable so that anyone can simply run the executable and have the flask server run on their own machine. In doing this, I ran into multiple complications with conflicting libraries, missing dll's and a host of other problems getting the flask app to bundle itself into an executable properly. However, I was able to finally get everything working smoothly, and created executables for my flask app in both Windows 10 and Ubuntu 18.04.




## III. Refinement

My project thankfully needed little refinement in the algorithms and techniques used. The most important refinement in this whole project was switching from L1 loss while training to the MSE loss. This allowed my Brushstroke GAN to properly generate brushstrokes, as opposed to black boxes.

Most of the refinement of the models used in this project was done by increasing the training time. I initially used 25 training epochs per model while training the Brushstroke GAN, but later

increased this to 100 train epochs, which increased the accuracy of the generated brushstrokes. This resulted in brushstrokes that were less uniform and resulted in better imitation of the features of the original brushstrokes.

## IV. Results

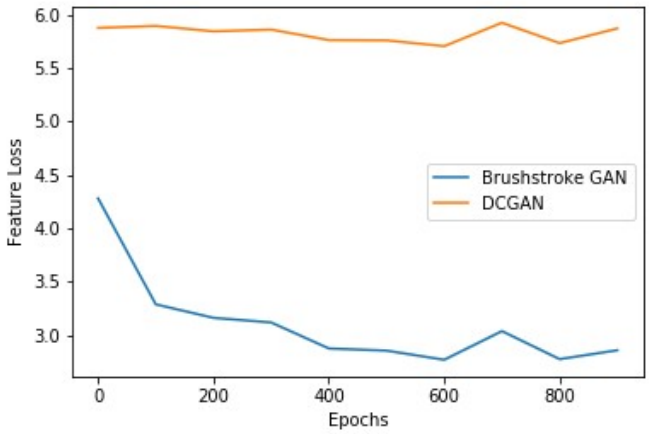
### I. Model Evaluation and Validation

The final model used in this project is a Brushstroke GAN generator, which I use to generate images from input images for my flask app. However, since the model is a type of Neural Network, discussion and evaluation of the models parameters is difficult and unintuitive. However, there are some qualities which can be discussed, such as model size and memory usage. The models parameter file is 145.6MB large, which is the largest model I have worked with to date. As a result, it uses a significant amount of memory, about 1.8GB of Vram.

Additionally, this model is much more robust than the DCGAN. The DCGAN has a tendency to fail to train, resulting in the outputs being simply static, or the output images will simply get stuck in a local minima and not train after a certain epoch. In analyzing the 100 output images, the DCGAN failed to train on >30% of the images. In contrast, the Brushstroke GAN solution is much more robust. It failed to train on 0 images. Every single image it was able to successfully train to some relatively good output. Therefore, I believe that my model is much more robust than the traditional DCGAN models used for image generation.

### II. Justification

After training on 100 images, I gathered the average FeatureLoss of both the DCGAN and Brushstroke GAN at every hundreth epoch and found that the FeatureLoss of the Brushstroke GAN was consistently lower than the DCGAN. This shows that the output images generated by the Brushstroke GAN contain more of the features present in the input images than the images generated by the DCGAN. As a result, I believe my Brushstroke GAN model is better at creating accurate imitations of input images, and those output images are also more aesthetically pleasing than the DCGAN images. The DCGAN images, in addition to being less feature accurate than the Brushstroke GAN images, also feature heavy artifacting and static. Many of the images also failed to train, so the Brushstroke GAN is more stable than the DCGAN as well.




*Brushstroke GAN FeatureLoss is significantly lower than DCGAN FeatureLoss*

Output Image Examples: DCGAN Output(left) Original Image(center) Brushstroke GAN Output(right)

## V. Conclusion

After a visual inspection of the output images, I believe my model adequately solves the problem statement I laid out. The Brushstroke GAN is able to better generate aesthetically pleasing and accurate images than a traditional image-generating DCGAN. In addition, it is also more stable and lacks any weird artifacting and static present in the DCGAN images.

However, it can use some improvements. Small features such as eyes and hands/feet or things of that nature are hard for the model to accurately paint. It might be a good improvement to increase the

amount of strokes it is allowed to paint with, although that would increase memory usage. Also, I believe that using the FeatureLoss during training might increase performance as well. As it is, this model does solve the problem, and I think it is a very interesting solution to a unique problem.

On a personal note, I think it's very interesting to watch the Brushstroke GAN paint its images stroke-by-stroke, which can be seen after launching the flask app on a machine.

## VI. References


**1)** *Decrappification, DeOldification, and Super Resolution.* Jason Antic (Deoldify), Jeremy Howard (fast.ai), and Uri Manor (Salk Institute) https://www.fast.ai/2019/05/03/decrappify/ , 2019.

**2)** *Fast.ai MOOC Lesson 7: Resnets from scratch; U-net; Generative (adversarial) networks*. https://course.fast.ai/videos/?lesson=7 ; Notebook: https://nbviewer.jupyter.org/github/fastai/course-v3/blob/master/nbs/dl1/lesson7-superres.ipynb

**3)** *Perceptual Losses for Real-Time Style Transfer and Super-Resolution* Justin Johnson, Alexandre Alahi, Li Fei-Fei https://arxiv.org/abs/1603.08155 , 2016

**4)** *Neural Painters: A Learned Differentiable Constraint for Generating Brushstroke Paintings*. Reiichiro Nakano arXiv preprint arXiv:1904.08410, 2019.

***5)*** *The Joy of Neural Painting https://medium.com/libreai/the-joy-of-neural-painting-e4319282d51f*