IMAGE INPAINTING

A PREPRINT

Peter Butler[*]
Department of Computer Engineering
City College of New York

Shravan Dhakal[†]
Department of Computer Science
City College of New York

Ted Chen
Department of Computer Science
City College of New York

May 17, 2019

ABSTRACT

Image inpainting has been a problem that many have wished to solve for many years. Images with randomly masked out holes are presented to the network model and the model is responsible for filling up the holes effectively. Through the use of Generative Adversarial Networks and custom defined loss functions: Contextual and Perceptual, we were able to produce decent results with the Celebrity dataset.

[*]Source Code: `https://github.com/peterdbutler/LoC_FSA_1935_Scraper`
[†]Source code: `https://github.com/shravan097/PythonMLPractice/tree/master/DCGanScript`

1 Introduction

1.1 Motivation

During the 1930s, the Farm Security Administration (FSA) was created in an attempt to mitigate the effects of the Great Depression on farmers and their families in rural America.[1] As an extension of this program, a photography program was initiated in 1935. Ostensibly an effort to provide good public relations to the broader FSA's efforts, it also provided out of work photographers and technicians with gainful employment. The photography program successfully and famously documented rural and urban life in America of the late 30s. When America entered World War II in 1941, the program was rolled into the Office of War Information, where it's declared focus transitioned to documenting the nation's domestic war efforts.

The program was led by a man named Roy Stryker. While Styker is and should be credited for the program's enormous success and historical import, historians have criticized him for his ruthless and destructive editing practices. He and others under his charge would take a hole punch to negatives of images deemed to be aesthetically inadequate and physically destroy them. While some of these images are undoubtedly of low technical quality (i.e. over- or under-exposed, out of focus, etc), many are perfectly fine images.

In the early 2010s, the Library of Congress (LoC) completed the digitization of the original negatives, and made the images available online (the collection contains over 175,000 unique images). Using programs such as Adobe Photoshop or the GNU Image Manipulation Program (GIMP), many individuals have since attempted to perform a kind of digital conservation by manually inpainting the hole-punched damaged images. We propose to leverage machine learning and develop and train a model capable of inpainting these damaged regions automatically and at scale.

1.2 Background

Through detailed research of similar inpainting problems, we found that there was a wide breadth of solutions from which we might draw from in finding a solution specific to our problem. A majority of these machine learning solutions were either implemented with the use of encoders/decoders, Convolutional Neural Networks (CNNs), Generative Adversarial Networks (GANs), or a combination thereof. GANs are a relatively new concept of deep learning and many research papers that employ such networks report delivering exceptional results. GANs can be described simply as a network that tries to learn its input's data distribution. While there exists a broad spectrum of different classes of networks, in general terms, the architecture is the same. As a whole it is constructed from two separate networks, a Generator Network (G) and a Discriminator Network (D).[2]

The discriminator network is a type of classifier network that determines if a given input is 'real' or 'fake'. It will train itself to be best at catching any 'fake' data that does not come from the original data distribution. To train the network (D), we feed in training batch of size m where half of the training data comes from $p_{origialData}$ where $p_{originalData} \in originalDataDisribution$ and half of the other data comes from p_{model} where $p_{model} \in G(z)$. We label all the $p_{originalData}$ as 'real' and all the data coming from p_{model} as 'fake'. The 'real' and 'fake' can be simply represented as 1s and 0s in binary. The discriminator will optimize itself by minimizing the following loss function:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} [\log D(x^i) + \log(1 - D(G(z^i)))] \tag{1}$$

In equation (1), we see that the loss function is defined in two parts. The first part,$\log(D(x))$ eventually will be high whenever the $D(x)$ is close to zero and low when $D(x)$ is close to 1. $D(x)$ is close to zero whenever it sees a 'fake' image and close to 1 whenever it sees 'real' image. The second part to the loss function,

$\log(1 - D(G(z)))$, is the generator's loss that will be discussed in detail later on. In short, this part will be high whenever, the discriminator misclassifies the input image, ie. it classifies 'fake' input image as 'real'; and low whenever the discriminator gets it right, ie. it classifies 'fake' image as 'fake'.

The generator network (G) utilizes a deep learning architecture and is a differential function with z as a input. The z is a random vector such that $z \in R^n$. Vector z is also called a latent vector. The goal of the generator is to fool the discriminator as much as possible by picking a good latent vector z such that $G(z) \sim p_{originalData}$. We attain a better generator by minimizing the following loss function (1) and move toward the gradient.

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} [\log(1 - D(G(z^i)))] \tag{2}$$

In equation (2), we are taking average of all the loss in the training batch of size m.

The loss function is defined with the function $\log(1 - D(G(z)))$. The loss function attains high value whenever the generator does poor job in fooling the discriminator. If the generator fools the discriminator proficiently then the loss function also outputs low value.

The overall algorithm is a minimax game where the discriminator is trying to maximize its reward, $V(D, G)$ (3),and the generator is trying to minimize discrimator's reward.

$$\min_G \max_D V(D, G) = \mathbb{E}_{x\ p_{data}(x)}[\log D(x)] + \mathbb{E}_{x\ p_z(z)}[\log 1 - D(G(x))] \tag{3}$$

As a first step objective, we sought to train a GAN to inpaint a specific portion of an image, without evaluating how realistic the generated content may or may not appeared to be. Being that the use of GANs are a relatively new trend in machine learning circles, we further believed that there would be plenty of room to experiment and learn.

1.3 Related Work

Image inpainting has always been an interesting and challenging problem in computer science, image processing, and computer-vision related applications. Before the advent and popular rise of deep machine learning, researchers offered algorithms that were non-learning based in nature. Bertalmilo et al. applied 2 mathematical concepts and models from fluid dynamics, partial differential equations to solve the inpainting problem.

Alexandru Telea's work similarly relies on partial differentials and surface normals, yet provides another, faster algorithm for inpainting a region of an image, utilizing the Fast Marching Method.[3] The simple algorithm is as follows: take a given point p on the boundary edge $\partial\Omega$ of a region to be inpainted,Ω . The nearby region of p , $B_\epsilon(p)$, is defined by the size ϵ . For a sufficiently small ϵ , we may define the first order approximation of this region to be the point q . We then inpaint p with the normalized sum of values for all q in the region $B_\epsilon(p)$, moving the boundary edge $\partial\Omega$ inward towards the center of Ω at each iteration, until the center is inpainted. Implementations of both algorithms are included in the standard OpenCV package, and comparison results using Telea's algorithm are included in later sections.

In Radford, Metz, and Chintala [4], Researchers at Facebook Artificial Intelligence and Indico Research introduced Deep Convolutional Generative Adversarial Networks (DCGAN). A fundamental challenge to training GANs is the imbalance in training difficulties associated with the generator network vs. the discriminator. The researchers proposal was more stable to train and was able to generate, on average, better results from a given sample space. DC GAN is a novel approach for unsupervised learning. It makes use of convolution neural networks (CNN) without pooling and fully connected layers. The primary differences between a regular GAN and a

DCGAN were that pooling layers in the discriminator and generator were replaced with strided convolutions and fractional-strided convolutions respectively. Both networks used batch normalization in each layer, which was crucial in stabilizing the training.

Context-Conditional Generative Adversarial Networks (CCGANs), introduced in 2016 by Denton et al., are essentially a modification of the DCGAN architecture mentioned above. Denton et al. propose a change in the model by introducing an extra parameter into the generator: the masked image in need of inpainting. Denton et al.proposed that the added parameter would allow the generator to generalize samples well because both the generator and discriminator would be conditioned to fill in the image effectively using the unmasked areas of the input image parameter.[5]

The Globally and Locally Consistent Image Completion (GLCIC) network architecture was introduced by Iizuka et al. in SIGGRAPH, 2017. The GLCIC mode features three networks: a fully convolutional completion network and two auxiliary global and local discriminator networks.[6] The main goal of this architecture is to train the completion network to inpaint the images and have the local discriminator evaluate the region inpainted and the global discriminator to evaluate the entire image as a whole. The completion network, unlike other convolutional nets, do not use max-pooling layers and instead, uses strided convolutions in order to decrease the size of images. The paper also mentions the use of dilated convolution kernels which is able to increase the area of which the completion network uses as information to inpaint the image.

Finally, DeepfillV1, a model introduced in 2018 by Yu, et al.[7] (with DeepfillV2, a modification and extension of the DeepfillV1 model released in 2019[8]), proposes several modifications to a DCGAN, as well as incorporations of many of the concepts introduced by the network classes listed above. The first, and most major modification is the introduction of a coarse-to-fine network architecture. This architecture utilizes two loss functions, a contextualized (fine) loss, and a broader (coarse) loss. Similar to a CCGAN, Deepfillv1 takes as input the original, context image, as well as the mask, indicating the region needing inpainting. The network's loss function is built from the global loss of the original input image (vis-a-vis the generated output image), as well as the contextual loss over the masked region. Second, the discriminator network uses the Earth-Mover(Wassersterin-1) distance for classifying inputs as real or fake. Finally, the loss function utilized by the author's is spatially discounted: the loss for generated pixels nearer to a boundary edge are weighted more heavily than those more inward. Incredible results have bebeen presented by the author's paper, and comparative examples using the FSA dataset on several pre-trained Deepfillv1 models have been provided below.

2 Approach

2.1 The Language and Library

We decided from the outset to utilize Python to implement our model. Python was an easy choice, thanks in part to its versatility in terms of data pre-processing and it's wide popularity and use for and in machine learning applications. Other choices such as C++, Java, and JavaScript do not offer as many exhaustive and extensive libraries or frameworks that support machine learning applications.

Python has libraries like Tensorflow, Keras, PyTorch, and scikit-learn – all of which are highly regarded and widely used in research and machine learning model development. Tensorflow is an open source library originally provided by Google and is often used for lower level machine learning applications. Given the scope of our problem, it would make sense to use Tensorflow but as a library we found it to be very verbose, requiring extensive knowledge in machine learning to pilot correctly.

Another leading contender for our development library was Keras. Since Keras made things a lot more simple when compared to Tensorflow, it was also easier to learn and easier to read. For the most part, Keras was able to implement the same machine learning application that one would find in Tensorflow. The only real difference was that we would be abstracted from being able to tweak specific hyperparameters or modify specific layers or change very low-level related things in our network architecture. Being that Keras was a wrapper, this was all expected but we believed the pros outweigh the cons when compared with Tensorflow. Despite believing that Keras was more suitable for our needs, we still progressed on to look for the most suitable library.

PyTorch was not our initial choice for machine learning libraries but we became aware, from reading various research papers, that PyTorch was actually very popular in research. PyTorch was first introduced by Facebook Artificial Intelligence Research and became popular due to their support of dynamic auto-gradient graphs and Pythonic language style, making the code easier to understand and write. Additionally, PyTorch has an extensive standard library that we found rivals Tensorflow and Keras. By looking at the scope of our problem and online resources for learning Tensorflow, Keras, and PyTorch, we deemed that PyTorch was just as easy to learn as Keras but was also more flexible as it was not a wrapper. PyTorch was capable of more lower-level work when compared to Keras but did not require as extensive knowledge to the library as it did for Tensorflow. Because of all those observations, we decided that PyTorch would be the most suitable library for us to use in our project.

2.1.1 Google Cloud Vs. Amazon AWS

Google Cloud and Amazon AWS are the two largest and most reputable cloud services that were in our sights. Based on market share statistics, the three largest cloud services are Amazon AWS, Microsoft Azure, and Google Cloud, respectively.[9] However, we decided to only consider Amazon AWS and Google Cloud services because they were highly suggested platforms mentioned in lecture and also suggested by Stanford's class CS231n which were heavy inspirations for the tech stack chosen for the project.[10] We set up virtual instances using the guides posted by Stanford which show detailed step by step instructions in using and setting up instances on Amazon AWS and Google Cloud.[] One of the largest discrepancies between the two services was that Google Cloud offered GPU's in their instances while Amazon AWS only had CPU as an option. In terms of machine learning training, the difference between a CPU and a GPU in compute power can well be over tens or even hundreds of times faster for the GPU. Adding into consideration that our project would likely train on three channel images and may need to be trained on a large dataset, we concluded that we would need the strongest compute power possible due to time constraints.

Aside from having stronger compute power, Google Cloud also had a specific subset of virtual machines that were designed and tailored specifically for machine learning applications. Google Cloud offered virtual instances that had PyTorch, Keras,and Tensorflow dependencies pre-installed so their virtual machines were essentially ready to go without having to tinker with installing anything else. This was very attractive because we would be able to start implementing and running machine learning applications without worrying about our workspace environment.

Even though we chose PyTorch as the library that we wanted to use, none of us had any previous experience using the library. As a result, PyTorch had to be learned in order to make progress with our project. The great thing about PyTorch is that PyTorch provides an extensive set of tutorials that helps people get ramped up with PyTorch.[11] We used many of the tutorials found under the "Getting Started" section in order to understand PyTorch syntax, how to save and load model states, and even experiment with various machine learning algorithms using PyTorch. There was even a tutorial on DCGAN implementation using the CelebA as the dataset which was a great introduction to the entire workflow machine learning with PyTorch. Lastly, the pythonic nature of PyTorch allowed us to get things ramped up really quickly – we were able to run different machine learning algorithms and tinker with various loss functions.

2.2 Image Inpainting with a custom loss function

2.2.1 Loss Function

This idea was proposed by Yeh et al from the University of Illinois at Urbana-Champaign [12]. This method makes use of a generative network to fill the missing pixels. First, we trained our own generator and discriminator for five epochs using our training set from celebrity dataset on google cloud virtual machine using NVIDIA Tesla K80 GPU. The training took approximately 2 hours. Then we used a simple feed-forward neural network to optimize the custom loss function with respect to z vector, element of $\mathbb{R}^{100}$. The custom loss function consists of the contextual loss function and perceptual loss function. They are defined as the following:

$$L_{contextual}(z|y, M) = ||M \odot (G(z) - y)||_1 \tag{4}$$

$$L_{perceptual}(z) = -\lambda \log(D(G(z))) \tag{5}$$

$$M_i = \begin{cases} 0 & imagePixel_i = hole \\ 1 & imagePixel_i \neq hole \end{cases} \tag{6}$$

The loss function, $L_{contextual}$ (4) , utilizes the contextual information of the original picture with the hole. It helps us narrow down the search accurately. It takes y and M to find z vector where y represents the original image with a hole, M represents the mask,defined in (6) , and z is the random vector. It finds the difference between the generated image and original image with the hole and then multiplies with the mask to get the contextual pixels. Finally, we find the L_1 norm which is the sum of the absolute value of the pixels. Note that this loss function penalizes any image with high difference in contextual information between the generated image and original image with a hole.

The loss function, $L_{perceptual}$ (5), checks to see if our generated image looks real and authentic. It is essentially a cross entropy function as depicted in Figure 1, with a hyperparameter λ This loss function penalizes heavily on any misleading image that has low probability as outputted by the discriminator, $D(x)$, which gives probability of how realistic the image is . We set the hyperparameter $\lambda = 0.003$ according to Yeh et al. paper's suggestion which was based on their cross-validation set results. Our overall loss function is defined as following:

$$L(z) = L_{contextual}(z) + \lambda L_{perceptual}(z) \tag{7}$$

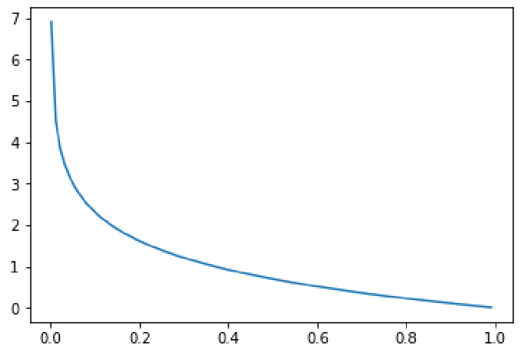


Figure 1: Pereceptual Loss

2.2.2 Optimizing *Z*

Our biggest challenge was to constrain the output image of the generator given a random vector z . Constraining the generator is an ongoing research topic and a lot of progress has been made in the research community. Many of the techniques were complex and out of the scope of the project.

Ultimately, due to inability to advance further in experiments where we constrain the vector z, we decided to perform a search in the generator's latent image space and back-propagate to minimize the loss function with respect to the z vector. Our architecture was straightforward to solve this minimization problem. We have a feed-forward neural network with a 100-dimensional input layer and a 100-dimensional output layer with a sigmoid activation function applied on the output layer. Adam optimizer is used to minimize the error. The feedforward network runs for 100 epochs.

3 Experiments and Result

3.1 Dataset

To preface the image inpainting problem, it is important to define the problem in the scope of our datasets. Throughout the entire research and experimentation process, we worked primarily on two datasets: images scanned from the Farm Security Administration film negatives ('FSA dataset') and the Celebrity Attribute Faces dataset ('CelebA').[13] The two datasets are massive in both structure and content. As a result, each yielded different results when we applied various machine learning algorithms and techniques. The following sections will seek to explore the various differences, problems, and benefits that each dataset has to the inpainting task.

The FSA Dataset was assembled by downloading image files of scanned digital negatives, provided by the Library of Congress. As both the original photographic work and their digitized derivatives were publicly funded by the library, the rights usage is public domain. For each unique image, five image files are typically available: three lower resolution jpegs, and two full resolution tiffs of different bit depth. Scanning resolution appears to have been fixed, and image resolution is therefore dependent on the size of the original film stock used. Film stocks are standardized formats of 35mm and 120mm roll stocks, or 4- by 5- inch sheet film.

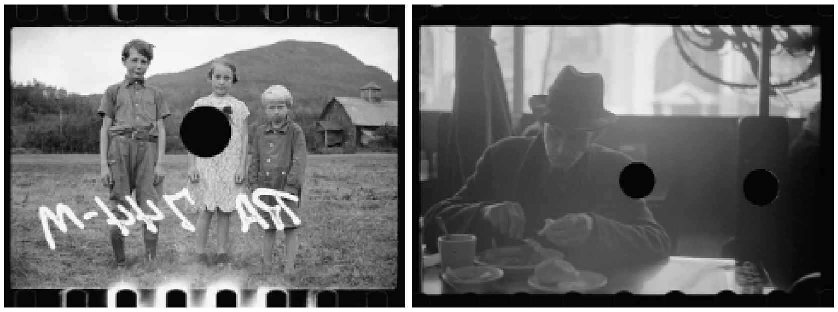


Figure 2: FSA Holed Images

While some color images do exist, we chose to restrict our images to black and white ones. Consequently, all images provided are single channel grayscale images.

Due to the enormity of the dataset, we chose to create a subset of images found by using the search term '1935' in the Library's web search. This yielded approximately 53,000 images. Images and associated metadata were scrapped from the using several Python scripts written using the Scrapy framework and BeautifulSoup HTML-parsing library. The script is hosted on github and can be accessed at `https://github.com/peterdbutler/LoC_FSA_1935_Scraper`.

We discovered early on that while the LoC website claimed images containing hole punches would yield a subset of results, this subset was incomplete, and damaged images still populated our training and cross-validation sets. In an initial review of the images, we discovered a wide latitude in hole size, complicating our plans for automatic hole detection. The issue results from two primary factors, examples of which are presented below. Firstly, because of the non-uniformity in film stock, and the semi-constant size of the hole punch used, for smaller film stocks (such as 35 mm), the hole appears larger, while for larger stocks the hole punch is much smaller (Figure 2). Second, there are some small variations in the size of the hole puncher used. These differences are too small to be viewed when comparing differing film stocks, but do become apparent when comparing like stocks to each other.

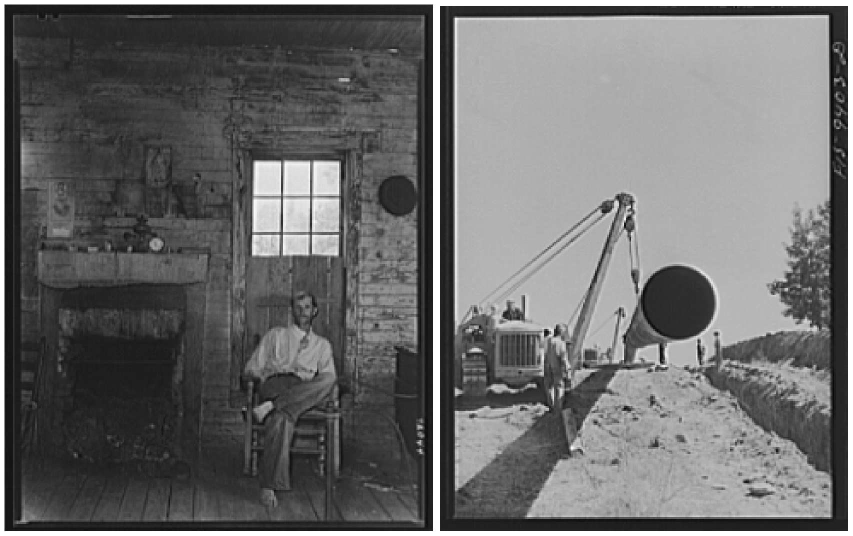


Figure 3: FSA Holed Images

There exists two broad approaches to feature detection we felt applicable to our problem: template matching and application of the Hough Circle Transform. Due to the enormous variations in size, and the general scale invariance of template matching, we immediately dismissed this approach. Further, we found that the Hough Circle Transfrom presented its own difficulties. We determined early on that an effort to develop an algorithm using the Hough Circle Transform solution was deemed to be too brittle and unreliable to find most target images.Further, we found that there were multiple cases of false positives(see Figure 3). As such the dataset was manually sorted into damaged and undamaged images.

The images consist predominantly of scenes from rural life in America of 1935. It is extremely varied, containing images of landscapes, people, animals, or combinations thereof. Further, by using the search term ‘1935’, undated images given the generalization ‘1935- 1945’ are included. Many of these images include subjects such as the domestic war effort or from urban scenes in depression-era America.

The CelebA dataset, on the other hand, is an extensive dataset that contains over two hundred thousand images and over ten thousand unique identities.[13] The greatest attribute of the CelebA dataset is that all the images are in a pretty uniform format – each image is normalized to contain the general head to lower shoulder region of a person. Furthermore, most images in the dataset have the person in the photo looking at a position relatively close to the camera. As a result, there are many similar looking photos in terms of style and structure. Even so, CelebA is also considered to be widely diverse. As mentioned earlier, there are over ten thousand unique celebrities in the dataset which allows for generative algorithms to generate diverse results that make relatively good sense. Depending on the scope of the problem, this can be seen as either a positive or a negative. In the scope of image inpainting, this is really beneficial because the generated sample could be used as a good foundational image for some image inpainting algorithms. Thus, it became clear that the CelebA dataset would be very useful for us to experiment with; it would bring stable results and act as a control ground for us when comparing results from the FSA dataset, particularly damaged images of persons.

3.2 Training

This training section is pretty straightforward and isn't really different from the way that many other researchers employ their training. For most of our training, we split the FSA dataset using the conventional 80/20 Train/Test split that most people employ. For the CelebA dataset, we split the dataset using a 99/1 Train/Test split because the dataset is so massive that even 1% of the dataset equated to over two thousand images for testing. Due to nature of our task, we want more training data than testing data because the lack of training data results in a faulty generator that may not produce realistic looking images and will fail to learn the distribution of our training images. Furthermore, our dataset is very large and deals with images and therefore 99/1 split was the best fit for this task. Afterward, we would write the scripts necessary to run and save the models on a virtual machine. Using the free credit service from Google Cloud Platform, we would train our models for some unset time on the virtual machines and save the state of the models. With PyTorch, we can easily load the saved models into our own local machines and perform whatever tweaking or experiments to the dataset and model. We repeat this process and gather whatever results we find throughout the entire process.

3.3 Testing and Results

3.3.1 OpenCV Inpainting

As a comparative example, Telea's inpainting algorithm, implemented by the OpenCV library utilized on several FSA Images are presented below. Examples include two sets of progressively larger inpainting areas. The limitations of this unguided approach are immediately obvious. While a naive Telea operation might yield passable results for smaller hole punches, acceptable texturing and detail are lacking for larger areas needing inpainting.

3.3.2 DCGAN on FSA Dataset

One of our first algorithms implemented was the DCGAN introduced earlier. For the most part, the training aspect of the DCGAN implementation was pretty straightforward because the architecture and hyperparameters of our DCGAN has been tried and tested for the best performance as suggested by the research paper. Despite the generator and discriminator converging at a reasonable value (see Figure 4) , the images generated through the FSA dataset was still left to be desired.

As we can see from the Figure 4, it appears that the loss on the generator can no longer be further reduced. The discriminator converged at a notably lower value which isn't ideal for us because the discriminator converging too quickly would halt further progress from the generator.

Training the DCGAN for much longer iterations,Figure 5, confirms the hypothesis that the generator can no longer perform better. The FSA dataset proves to be too varied and contains too many different classes within the dataset for the generator to find a good enough distribution within the dataset. We can further corroborate

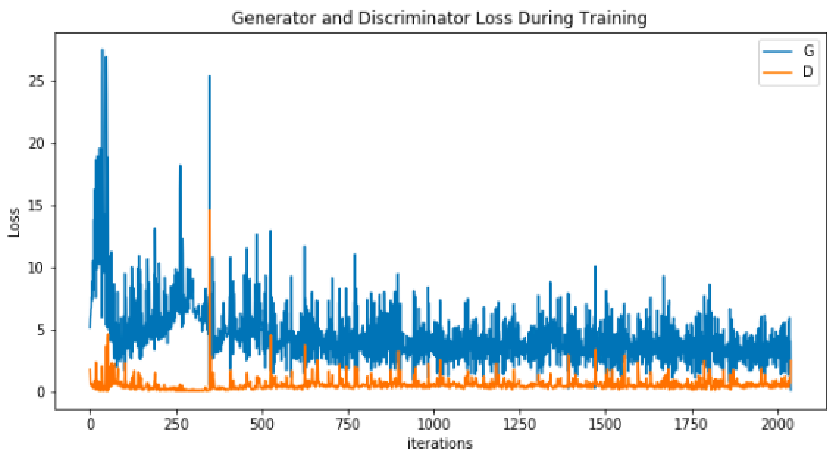


Figure 4: Generator and Discriminator Loss graphed

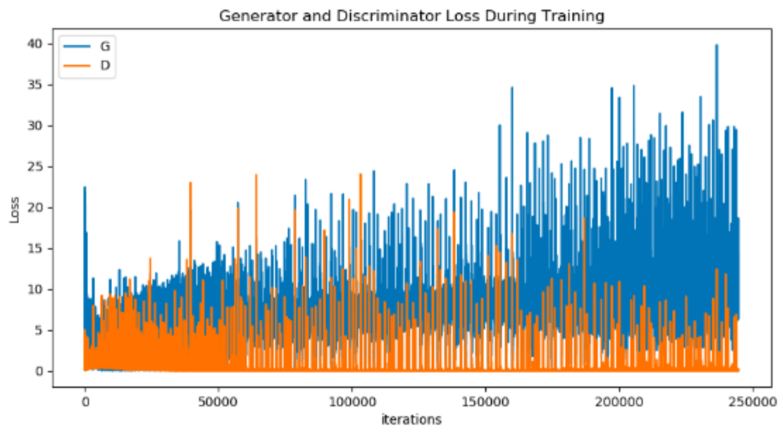


Figure 5: Generator and Discriminator graphed over 250k iterations

this by viewing some of the images,Figure 6, that generator created after such long training.

3.3.3 Algorithm from 2.2

The inpainting algorithm described in 2.2 performed fairly well in most of the testing dataset. We see that the middle two images were inpainted almost perfectly and the algorithm was able to generate similar fake celebrity, Figure 7 (c), to patch the original picture. However, it does have some performance issues on some test

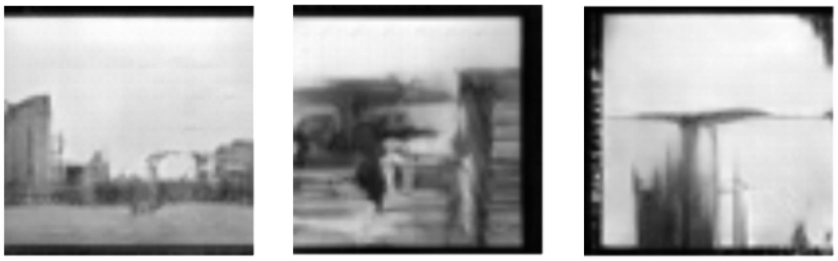


Figure 6: Three FSA sample images generated by the generator

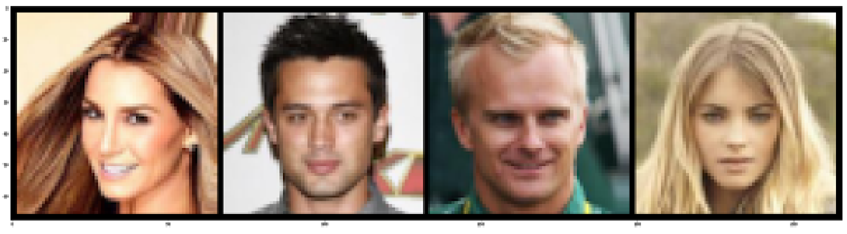
cases such as the image on the far left and the far right in Figure 7. We see that the algorithm fails to generate a good fake image to inpaint and the blending of the patch from generated images does not match the original mask.

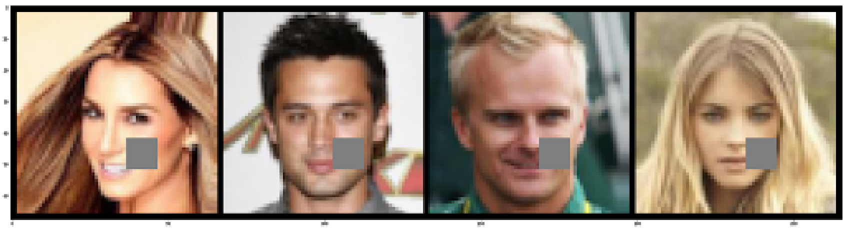
We can also visualize the loss function in Figure 8 when the inpainting algorithm was performed in Figure 7 images. We can see that the algorithm does a good job in minimizing the loss function but does not converge fully to zero. As expected, Image 1 and Image 4, the far right and far left image, struggles to minimize probably. We can see the error oscillating until it levels off. Hence, we see poor inpainting of Image 1 and Image 4 as compared to Image 2 and Image 3 where errors smoothly decrease.

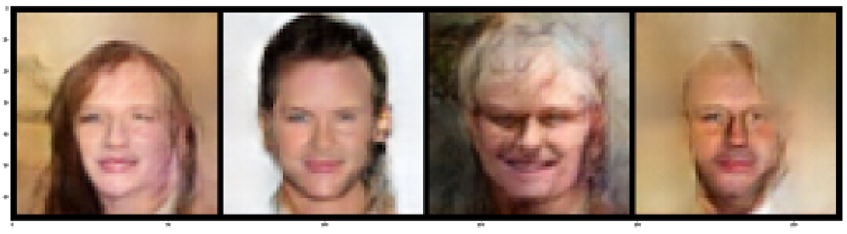
3.3.4 DeepfillV1 Pre-trained Models

The authors of DeepfillV1 have made several pre-trained models available at a github repository associated with the published paper. These models are trained on CelebA and CelebA-HQ (a higher resolution instance of the CelebA dataset), ImageNet, and Places. ImageNet is a dataset consisting of over 14 million images while Places is an MIT provided dataset of 2.5 million images.

Using a Google Cloud ML instance, these models pre-trained on the CelebHD (a higher resolution version of the celebA), and imageNet (another dataset widely well regarded for it's breadth and variation) were given as inputs several images from the FSA datasets, examples of which can be viewed below. Prior to feeding images as


(a) Original Images from Testing Dataset


(b) Original Image with mask applied


(c) Generator's attempt to generate similar image

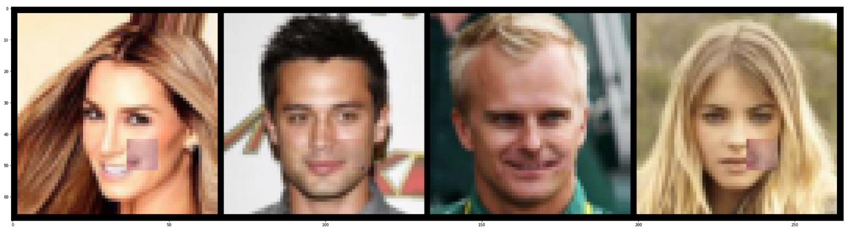

(d) After inpainting

Figure 7: Testing results from Celebrity dataset

inputs into the model, images were cropped to a square format, and expanded to three channel images from their original single channel grayscale format. These images are also compared to results of inpainting using the Telea algorithm, implemented in the openCV library.

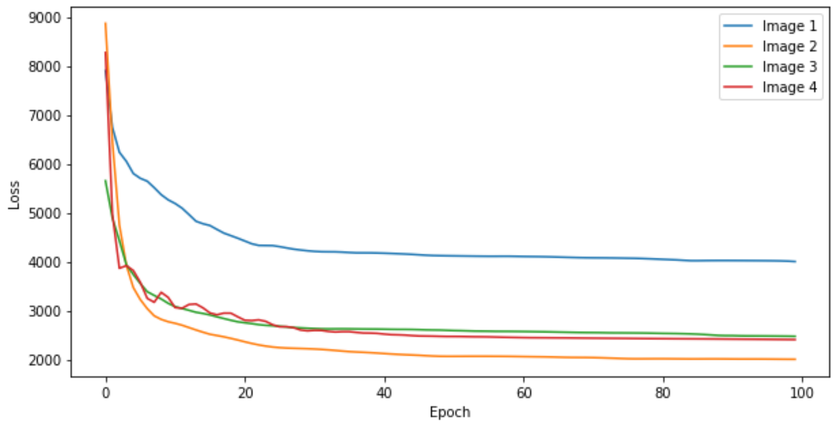
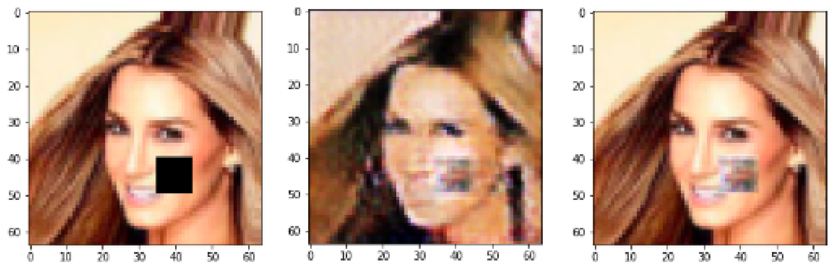


Figure 8: Loss Function of Figure 7 images after training. Labeled respectively starting from the left in Figure 3.

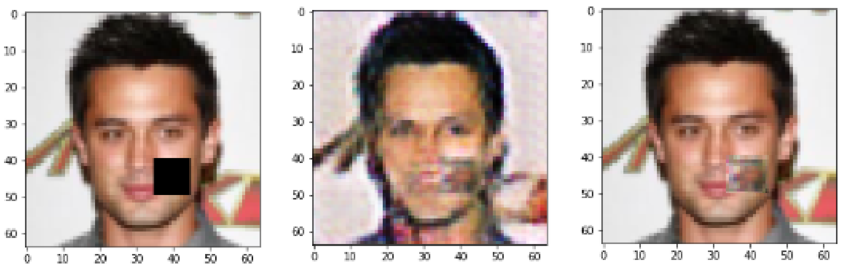
Finally, in the interest of completion, we applied the same Telea algorithm and pretrained (celebHD and imageNet) deepfill models on the same CelebA images, who's results we shared above in section 3.3.3. The results of these inpainting techniques can be viewed below in Figure 11. We note the disparaity in the image size. Deepfill's atchitecture calls for input images 256 x 256 pixels square. Our model was developed for 64 x 64 pixels.

3.3.5 Context Conditional GAN (CCGAN)

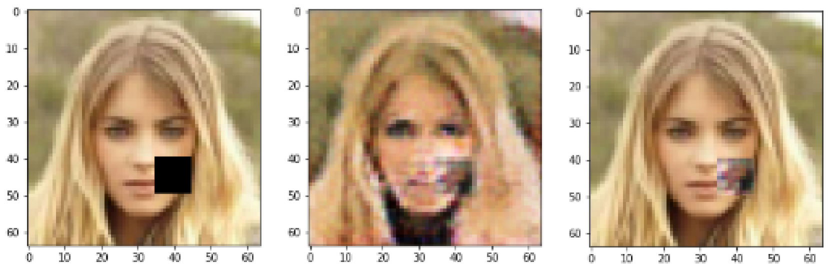
As previously discussed in our related works section, CCGANs are a special type of GANs where the objective is no longer to generate an entire image from the sample data but rather, to generate a portion that is originally masked out. The discriminator of the network is now no longer to determine whether an image is real or fake but rather, whether the generated portion of the image contextually makes sense. The network is able to learn whether the generated context makes sense by modifying the input of the generator. In previous works of GANs, the generator only received one argument as input: the noise vector **Z**. In the case of CCGANs, the generator now receives some downsampled image and the noise vector **Z** as input. The point of the downsampled image is to allow the generator to learn the context of the image and the purpose of downsampling the image is to ensure the generator does not overfit to the original image. This type of architecture is an indirect method in constraining the noise vector **Z**, a problem that will need to be looked into when dealing with a dataset as diverse as the FSA. Results are shown in **Figure 9**.



(a)



(b)



(c)

Figure 9: CCGAN result samples from test images(Figure 7a). Masked, Generated, Inpainted Result

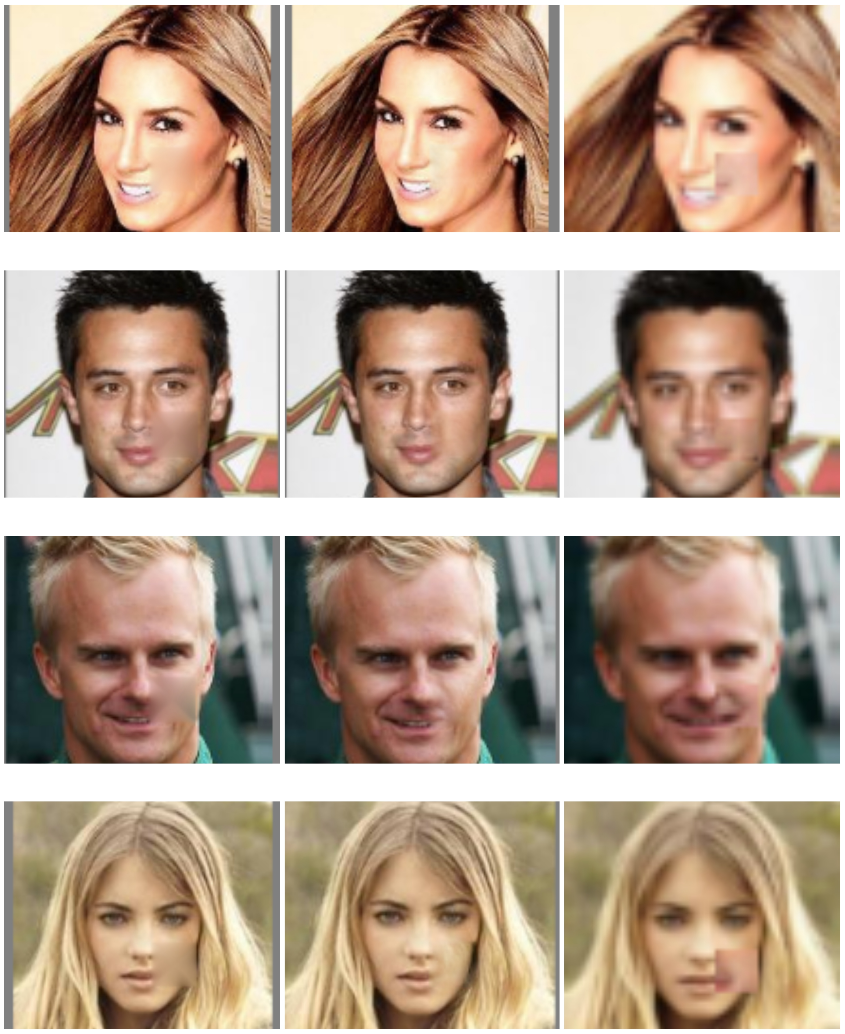


Figure 10: Comparison of Telea(OpenCV), DeepfillV1 and Custom Loss respectively

4 Conclusion

4.1 Future Works and Improvments

Looking back, we experimented with various different algorithms and garnered various results. Despite being able to develop an approach to successfully inpaint images within the CelebA dataset, there is still plenty more to improve on. For instance, many inpainting algorithms we saw used Poisson Blending in order to

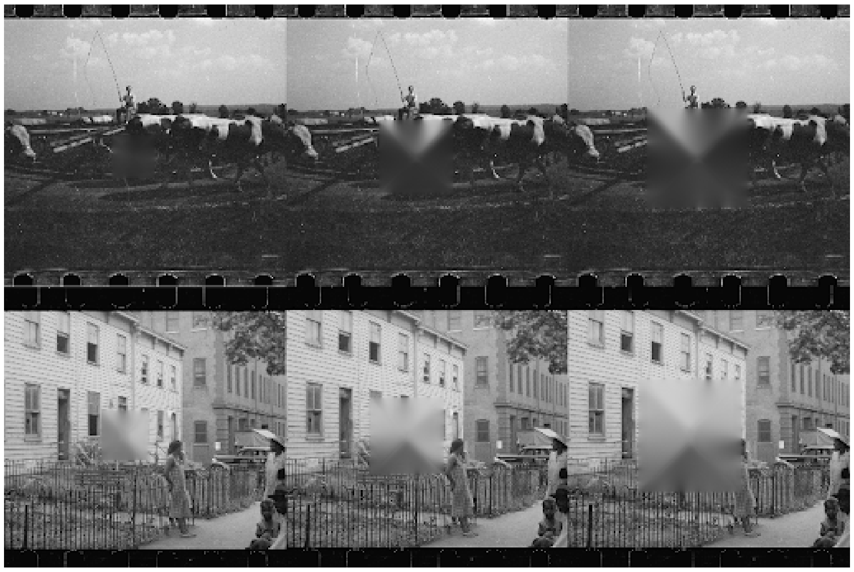


Figure 11: Comparison of Telea(OpenCV), DeepfillV1(celebrity), Deepfill(ImageNet) on FSA dataset respectively.

smoothly inpaint the masked image.[14] As seen from the results posted in earlier sections, the inpainted region of the image looks rough and patchy. Poisson Blending solves this problem by smoothing out the gradients of the image in the inpainted region. As a result, the outcome garners a more appealing and realistic result.

Aside from improvements with Poisson Blending, there are countless other research in image inpainting that yield decent results. For instance, researchers at Nvidia were able to produce a really robust image inpainting model using convolutional neural networks.[15] This is a completely different approach from ours and it goes to show how there are many more different methods that may solve the inpainting problem.

4.2 Final Remark

Our initial goal was to be able to restore the hole punched images on the FSA dataset through the use of an inpainting model and as evidenced, we were unable to do so. Despite that, we still learned a lot throughout the entire process. We would have never realized the difficulties of cultivating one's own dataset had we not have to manually sort through some of the fifty-thousand images in our FSA dataset. We realized how important data normalization is and how much we took it for granted since many of the datasets that we can find online are already nicely packaged for us. In the end, though unsuccessful with the FSA dataset, we were able to come up with a model to inpaint images from the Celebrity dataset. We knew that our original goal was ambitious but to be able to walk away knowing that we were successful with the Celebrity dataset is still an achievement in the end.

References

[1] Farm security administration/office of war information black-and-white negatives - background and scope. `http://www.loc.gov/pictures/collection/fsa/background.html`, Jan 1970.

[2] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative Adversarial Networks. *arXiv e-prints*, page arXiv:1406.2661, Jun 2014.

[3] Alexandru Telea. An image inpainting technique based on the fast marching method. *Eindhoven University of Technology*, Vol. 9, No. 1: 25—36.

[4] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks. *arXiv e-prints*, page arXiv:1511.06434, Nov 2015.

[5] Emily Denton, Sam Gross, and Rob Fergus. Semi-Supervised Learning with Context-Conditional Generative Adversarial Networks. *arXiv e-prints*, page arXiv:1611.06430, Nov 2016.

[6] Satoshi Iizuka, Edgar Simo-Serra, and Hiroshi Ishikawa. Globally and locally consistent image completion. `http://iizuka.cs.tsukuba.ac.jp/projects/completion/data/completion_sig2017.pdf`, Jul 2017.

[7] Jiahui Yu, Zhe Lin, Jimei Yang, Xiaohui Shen, Xin Lu, and Thomas S. Huang. Generative Image Inpainting with Contextual Attention. *arXiv e-prints*, page arXiv:1801.07892, Jan 2018.

[8] Jiahui Yu, Zhe Lin, Jimei Yang, Xiaohui Shen, Xin Lu, and Thomas S. Huang. Free-Form Image Inpainting with Gated Convolution. *arXiv e-prints*, page arXiv:1806.03589, Jun 2018.

[9] Cloud market share q4 2018 and full year 2018. `https://www.canalys.com/newsroom/cloud-market-share-q4-2018-and-full-year-2018`.

[10] `http://cs231n.github.io/setup-instructions/`.

[11] Welcome to pytorch tutorials. `https://pytorch.org/tutorials/`.

[12] Raymond A. Yeh, Chen Chen, Teck Yian Lim, Alexander G. Schwing, Mark Hasegawa-Johnson, and Minh N. Do. Semantic Image Inpainting with Deep Generative Models. *arXiv e-prints*, page arXiv:1607.07539, Jul 2016.

[13] `http://mmlab.ie.cuhk.edu.hk/projects/CelebA.html`.

[14] Andrew Blake Patrick Perez, Michel Gangnet. Poisson image editing. `https://www.cs.virginia.edu/~connelly/class/2014/comp_photo/proj2/poisson.pdf`.

[15] Guilin Liu, Fitsum A. Reda, Kevin J. Shih, Ting-Chun Wang, Andrew Tao, and Bryan Catanzaro. Image Inpainting for Irregular Holes Using Partial Convolutions. *arXiv e-prints*, page arXiv:1804.07723, Apr 2018.