End of Internship Report

Name of Intern	Vico Lee Zheng Yuan
Name of Supervisor	Dr. Teow Loo Nin
Date of Internship	06 January 2020 to 27 March 2020
Project Aim	1) Exploration of Generative Adversarial Networks (GANs) 2) To mitigate bias in datasets using GANs

Contents

Introduction

Generative adversarial network (GAN) is a class of machine learning systems invented by Ian Goodfellow et al. It involves building two neural networks to contest with each other in a game. Given a training set, this technique learns to generate new data with the same statistics as the training set.

GANs have since been used in countless industries ranging from generating photos of imaginary fashion models to improving astronomical images.

However, as with all technologies, along with the rise of the benefits of GANs came exploitations of this emerging technology, such as usage of Deepfakes to create fake news on the media and spread misinformation and uncertainty in our society.

In this project, I have set out to achieve two main objectives:

1. To deepen my understanding of various GAN models
2. To replicate the results of Fairness GAN: Generating Datasets with Fairness Properties using Generative Adversarial Network by Sattigeri et. al.

Implementation Details

In the following sections, I will be exploring the use of various GAN models in generating images after training on datasets like MNIST, CIFAR-10, and CelebA. To build my models, I used Keras (version 2.2.4) with Tensorflow-GPU (version 1.15) running as backend. I highly recommend the use of Functional APIs when using Keras rather than the Sequential method as it provides greater flexibility such as creation of ResBlocks in Fairness GAN. To manipulate matrices and structured data, I used libraries like Numpy and Pandas. My models were trained on the Linux environment using one Nvidia's GeForce GTX 1080 graphics card.

Vanilla Generative Adversarial Networks

To explore GANs, I first implemented basic architectures like the Vanilla GAN to train on simple datasets like MNIST, a dataset with handwritten digits.

The architecture of the Vanilla GAN is as follows:

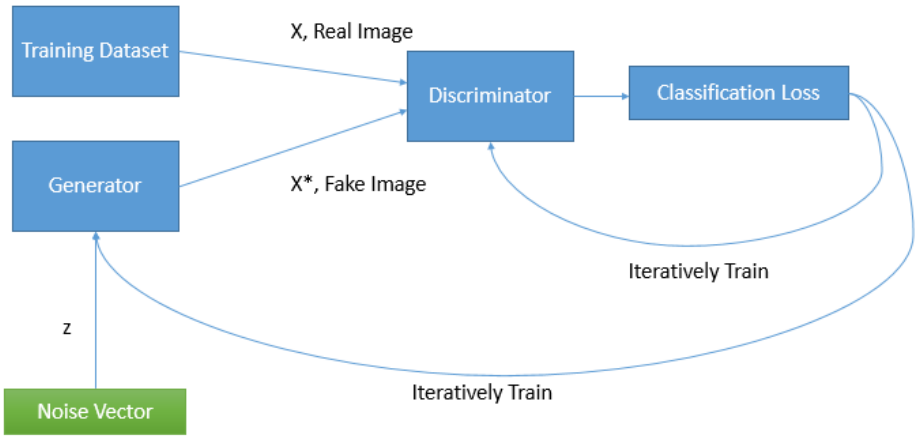
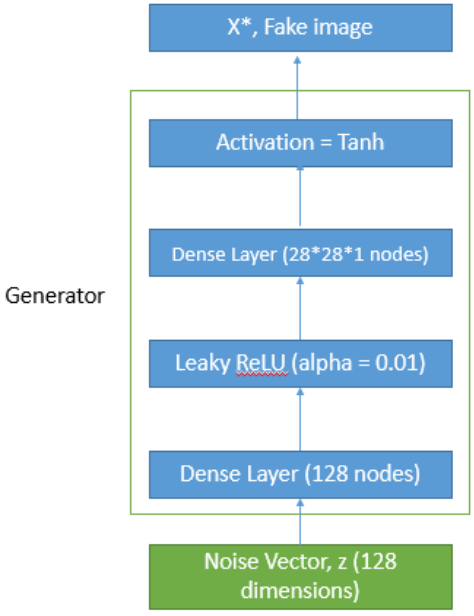


Figure 1: High level implementation of Generative Adversarial Networks.

In the GAN architecture diagram above, both the Generator and the Discriminator are trained using the Discriminator's loss. The Discriminator strives to minimize the loss; the Generator seeks to maximize the loss for the fake examples it produces.

Below shows the architecture used in Vanilla GAN to train on MNIST datasets for both the Generator and Discriminator:



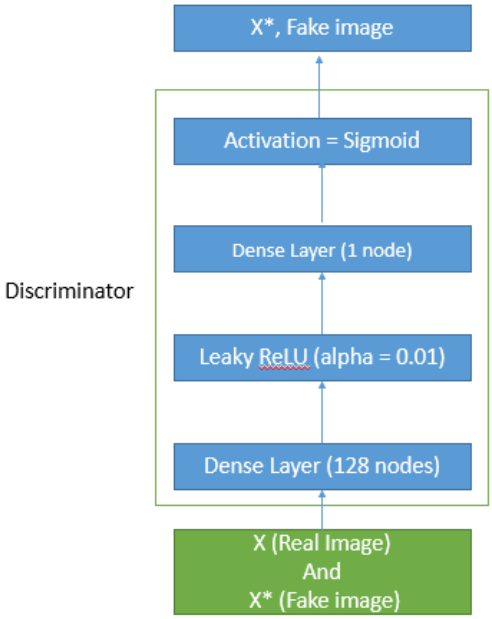
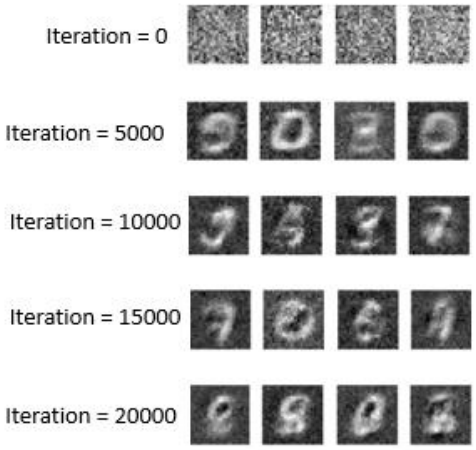


Figure 2 (on the left): Architecture used by the Generator in Vanilla GAN.
Figure 3 (on the right): Architecture used by the Discriminator in Vanilla GAN.

Operation	Kernel	Strides	Feature maps	BN?	Dropout	Nonlinearity
G(z) - 128 x 1 x 1 input						
Linear	N/A	N/A	128	No	0.0	Leaky ReLU
Linear	N/A	N/A	784	No	0.0	Tanh
D(x) - 28 x 28 x 1 input						
Linear	N/A	N/A	128	No	0.0	Leaky ReLU
Linear	N/A	N/A	1	No	0.0	Sigmoid
Generator Optimizer	Adam (learning rate = 0.001, beta 1 = 0.9, beta 2 = 0.999)					
Discriminator Optimizer	Adam (learning rate = 0.001, beta 1 = 0.9, beta 2 = 0.999)					
Batch size	128					
Iterations	20000					
Leaky ReLU slope	0.01					
Weight, bias initialization	Glorot Uniform, Constant(0)					

Table 1: Hyperparameters used in constructing the Vanilla GAN to generate handwritten digits.

Below are the MNIST images generated by the Vanilla GAN at 5000 iteration intervals.



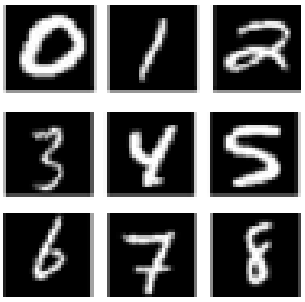


Figure 4 (on the left): MNIST images generated by the Vanilla GAN at 5000 iteration intervals.
Figure 5 (on the right): Sample images of various digits from the real MNIST dataset.

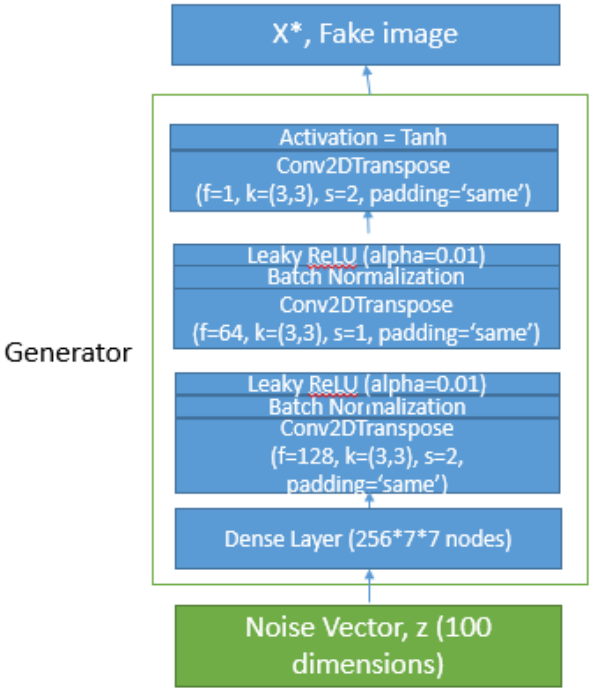
Conclusion

Although the images my Vanilla GAN generated are far from perfect, many of them are somewhat recognizable as real numerals. Perhaps by increasing the complexity of the architecture used, the Generator and Discriminator would be able to learn to produce images of higher resolutions.

Deep Convolutional Generative Adversarial Networks (DCGANs)

To improve the quality of images generated, I attempted to use a more powerful network architecture: DCGAN. For DCGAN, I had to use batch normalization to reduce the sensitivity of the training process to the scale of the features. The overall model architecture for DCGAN is same as that of Vanilla GAN as shown in Figure 1. The only differences are in the architectures of the Generator and Discriminator itself.

Below shows the architecture of the DCGAN used for MNIST:



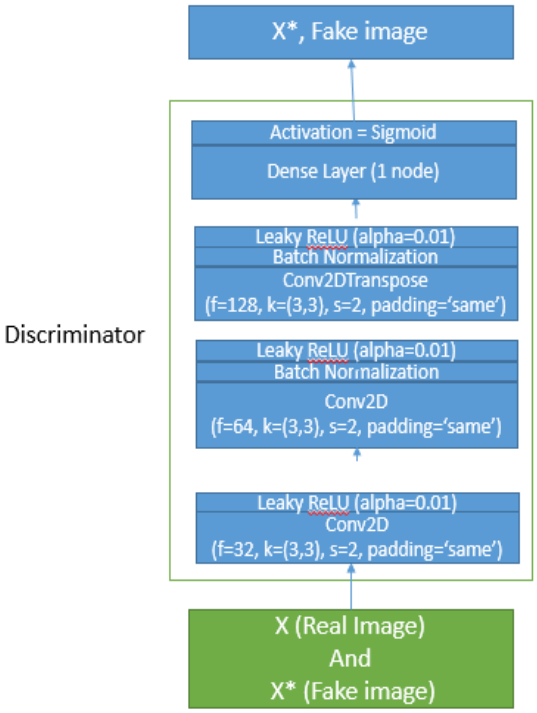


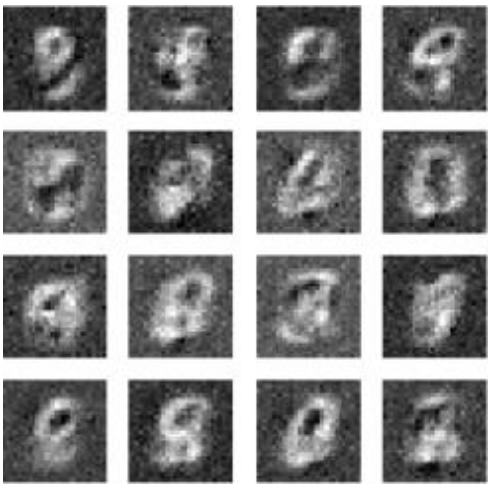
Figure 6 (on the left): Architecture used by the Generator in DCGAN to train on MNIST.
Figure 7 (on the right): Architecture used by the Discriminator in DCGAN to train on MNIST.

Note that convolutional layers were used instead of the Vanilla GAN's linear layers. Batch Normalization was also added in this architecture.

Operation	Kernel	Strides	Feature maps	BN?	Dropout	Nonlinearity
G(z) - 100 x 1 x 1 input						
Linear	N/A	N/A	12544	No	0.0	NIL
Transposed Convolution	3 x 3	2 x 2	128	Yes	0.0	Leaky ReLU
Transposed Convolution	3 x 3	1 x 1	64	Yes	0.0	Leaky ReLU
Transposed Convolution	3 x 3	2 x 2	1	No	0.0	Tanh
D(x) - 28 x 28 x 1 input						
Convolution	3 x 3	2 x 2	32	No	0.0	Leaky ReLU
Convolution	3 x 3	2 x 2	64	Yes	0.0	Leaky ReLU
Convolution	3 x 3	2 x 2	128	Yes	0.0	Leaky ReLU
Linear	N/A	N/A	1	No	0.0	Sigmoid
Generator Optimizer	Adam (learning rate = 0.001, beta 1 = 0.9, beta 2 = 0.999)					
Discriminator Optimizer	Adam (learning rate = 0.001, beta 1 = 0.9, beta 2 = 0.999)					
Batch size	128					
Iterations	20000					
Leaky ReLU slope	0.01					
Weight, bias initialization	Glorot Uniform, Constant(0)					

Table 2: Hyperparameters used in constructing the DCGAN to generate handwritten digits.

Below are the MNIST images generated by the DCGAN at 20000 iteration:



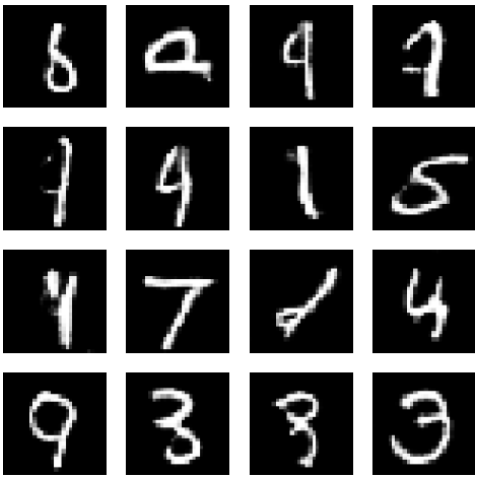


Figure 8 (on the left): MNIST images generated by the Vanilla GAN at 20,000 iterations.
Figure 9 (on the right): MNIST images generated by the DCGAN at 20,000 iterations.

Conclusion:

As evident by the preceding figures, many of the images of handwritten digits (MNIST) that the network produces after being fully trained (20,000 iterations) are virtually indistinguishable from the ones written by a human hand.

I also noticed that by using batch normalization, the covariate shift is limited. This reduces unwanted interdependence between parameters across layers by normalizing outputs of each layer before they are passed as inputs to the next layer, helping to speed up the training process and increase the robustness.

While DCGAN's ability to produce realistic-looking handwritten digits is impressive, I was unable to control what it produces, say, the number '7' as opposed to the number '9' at any given time. To improve the capability of GANs, I needed the neural network to learn to match labels to their respective images, allowing for selective generation of datasets.

Conditional Generative Adversarial Networks (CGANs)

As seen from my Vanilla GAN and DCGAN above, while the Generators from these networks were able to generate impressive images, they were unable to generate images by choice. Rather, the images generated were largely random. To overcome this issue, I will now be exploring Conditional GANs (CGANs) on the same handwritten digits dataset used above.

CGAN is a generative adversarial network whose Generator and Discriminator are conditioned during training using some additional information. This auxiliary information could be anything, such as class label, a set of tags, or even a written description. During training, the Generator learns to produce realistic examples for each label in the training dataset, and the Discriminator learn to distinguish fake example-label pairs from the real example-label pairs. The Discriminator learns to reject wrongly paired image-labels while it also learn to reject all image-label pairs in which the image is fake, even if the label matches the image.

Accordingly, it is not enough for the Generator to produce realistic-looking data. It also needs to generate images which matches their labels.

The architecture for CGAN is as follows:

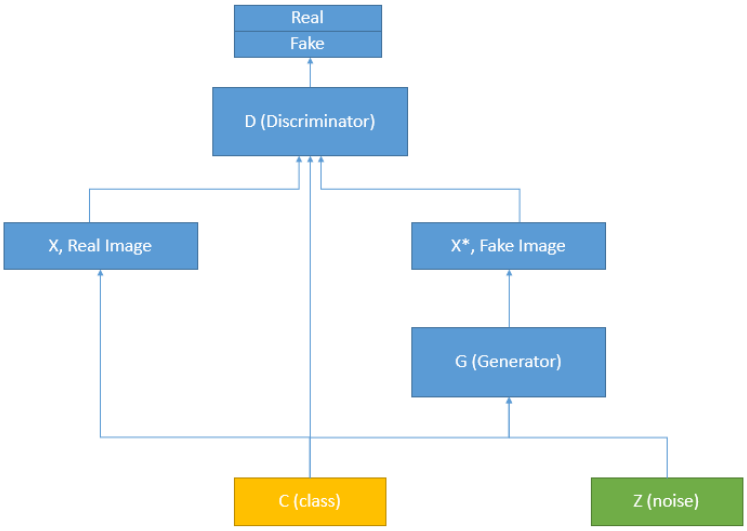
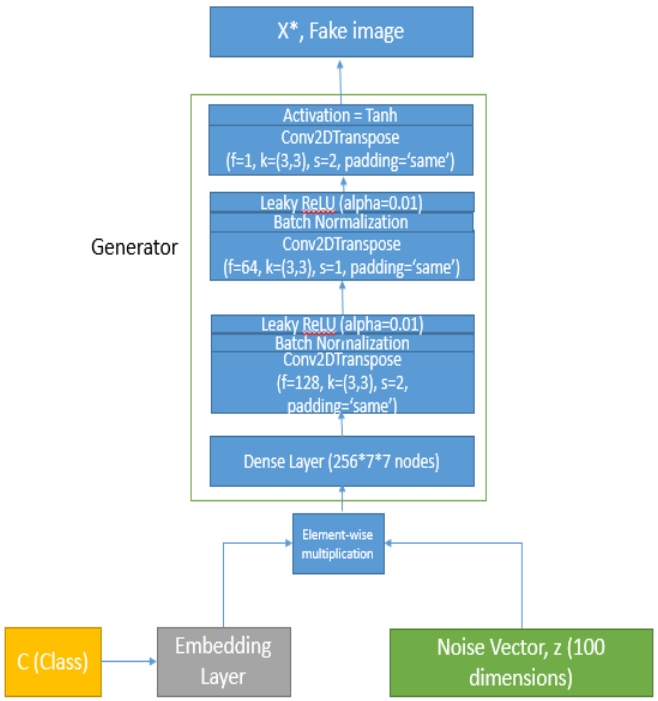


Figure 10: High-level architecture of CGAN.

With respect to figure 10, notice that for each fake example, the same label y is passed to both the Generator and the Discriminator. Also, note that the Discriminator is never explicitly trained to reject mismatched pairs by being trained on real examples with mismatching labels; its ability to identify mismatched pairs is a by-product of being trained to accept only real matching pairs.

The CGAN Discriminator receives fake labelled examples (x*|y, y) produced by the Generator and real labelled examples (x, y), and it learns to tell whether a given example-label is real or fake.

Below shows the architecture used to train the Generator and Discriminator on MNIST dataset for CGAN:



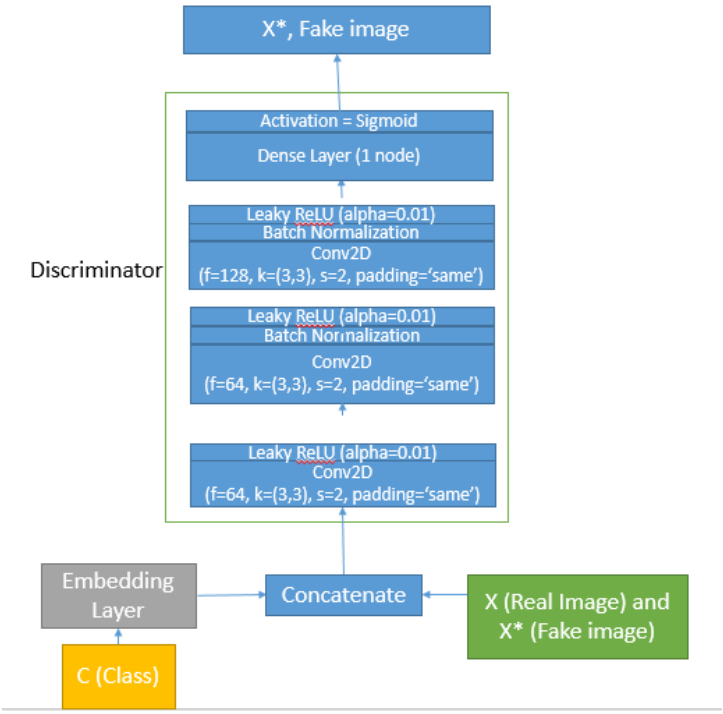


Figure 11 (on the left): Architecture of the Generator used to train on MNIST. Figure 12 (on the right): Architecture of the Discriminator used to train on MNIST.

Operation	Kernel	Strides	Feature maps	BN?	Dropout	Nonlinearity
G(z) - 100 x 1 x 1 input						
Linear	N/A	N/A	12544	No	0.0	NIL
Transposed Convolution	3 x 3	2 x 2	128	Yes	0.0	Leaky ReLU
Transposed Convolution	3 x 3	1 x 1	64	Yes	0.0	Leaky ReLU
Transposed Convolution	3 x 3	2 x 2	1	No	0.0	Tanh
D(x) - 28 x 28 x 1 input						
Convolution	3 x 3	2 x 2	64	No	0.0	Leaky ReLU
Convolution	3 x 3	2 x 2	64	Yes	0.0	Leaky ReLU
Convolution	3 x 3	2 x 2	128	Yes	0.0	Leaky ReLU
Linear	N/A	N/A	1	No	0.0	Sigmoid
Generator Optimizer	Adam (learning rate = 0.001, beta 1 = 0.9, beta 2 = 0.999)					
Discriminator Optimizer	Adam (learning rate = 0.001, beta 1 = 0.9, beta 2 = 0.999)					
Batch size	32					
Iterations	12000					
Leaky ReLU slope	0.01					
Weight, bias initialization	Glorot Uniform, Constant(0)					

Table 3: Hyperparameters used in constructing the CGAN to train on handwritten digits.

Figure 13: Images of handwritten digits produced by CGAN Generator after it is fully trained at 12,000 iterations and batch size of 32. Notice that now we are able to select which handwritten digit should be produced by inputting the desired labels into the Generator.

Conclusion:

Below shows images of digits produced by the Generator. At each row, we instruct the Generator to synthesize a different numeral. Notice that each numeral is rendered in a different writing style, attesting to CGAN's ability not only to learn to produce examples matching labels from the training dataset, but also to capture the full diversity of the training data.

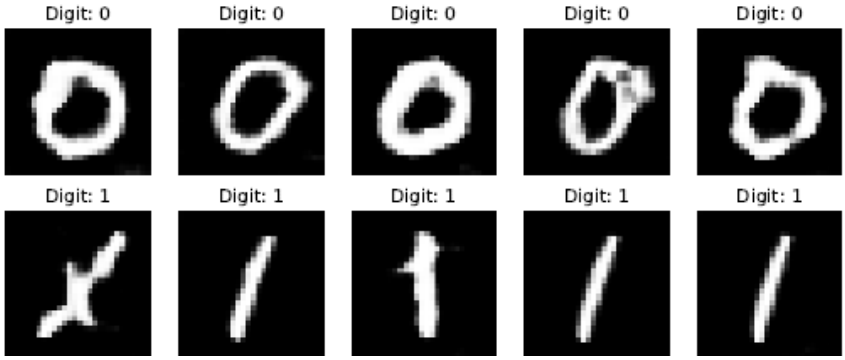


Figure 14: Images of handwritten digits (0 and 1) produced by the CGAN Generator at 12,000 iterations.

Auxiliary Classifier Generative Adversarial Networks (AC-GANs):

The CGAN framework used in the previous section supplies both the generator and discriminator with class labels in order to produce class conditional samples. However, in AC-GANs, instead of feeding side information to the discriminator, one can task the discriminator with reconstructing side information. This is done by modifying the discriminator to contain an auxiliary decoder network that outputs the class labels of the training data. The motivation behind doing so is that by forcing the model to perform additional tasks, it will improve its performance on the original task. Thus, AC-GANs were proposed such that it is class conditional but with an auxiliary decoder that is tasked with reconstructing class labels.

Below shows the architecture used by the AC-GANs on the CIFAR-10 dataset:

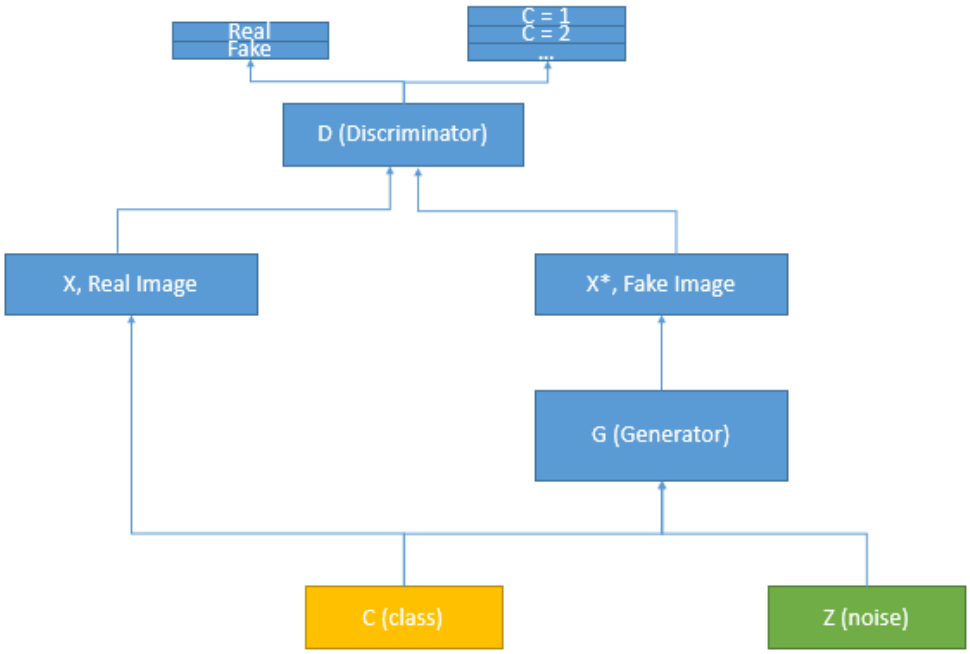


Figure 15: Architecture used by AC-GAN to train on CIFAR-10 datasets.

Notice that it varies from CGAN such that instead of feeding side information (in this case, class) to the discriminator, it tasks the discriminator to reconstruct the side information.

Operation	Kernel	Strides	Feature maps	BN?	Dropout	Nonlinearity
$G_x(z)$ – $110 \times 1 \times 1$ input						
Linear	N/A	N/A	384	×	0.0	ReLU
Transposed Convolution	5×5	2×2	192	√	0.0	ReLU
Transposed Convolution	5×5	2×2	96	√	0.0	ReLU
Transposed Convolution	5×5	2×2	3	×	0.0	Tanh
$D(x)$ – $32 \times 32 \times 3$ input						
Convolution	3×3	2×2	16	×	0.5	Leaky ReLU
Convolution	3×3	1×1	32	√	0.5	Leaky ReLU
Convolution	3×3	2×2	64	√	0.5	Leaky ReLU
Convolution	3×3	1×1	128	√	0.5	Leaky ReLU
Convolution	3×3	2×2	256	√	0.5	Leaky ReLU
Convolution	3×3	1×1	512	√	0.5	Leaky ReLU
Linear	N/A	N/A	11	×	0.0	Soft-Sigmoid
Generator Optimizer	Adam ($\alpha = [0.0001, 0.0002, 0.0003]$, $\beta_1 = 0.5$, $\beta_2 = 0.999$)					
Discriminator Optimizer	Adam ($\alpha = [0.0001, 0.0002, 0.0003]$, $\beta_1 = 0.5$, $\beta_2 = 0.999$)					
Batch size	100					
Iterations	100000					
Leaky ReLU slope	0.2					
Activation noise standard deviation	$[0, 0.1, 0.2]$					
Weight, bias initialization	Isotropic gaussian ($\mu = 0$, $\sigma = 0.02$), Constant(0)					

Table 4: Hyperparameters used in constructing the AC-GAN and training it on the CIFAR-10 dataset.

Figure 16 (on the left): Images from the actual CIFAR-10 dataset of certain classes (Airplanes, Automobiles, Horses, Trucks).
Figure 17 (on the right): Images produced by the AC-GAN Generator for the respective classes of the CIFAR-10 dataset after training for 100,000 iterations.

Conclusion:

Looking at the dataset, while the generated images can still be discernible to a certain extent, these images remain blur. Notice that the fake images of airplanes seem to be clearest amongst other generated images like the automobile. It is possible that this is because the background of the training dataset for airplanes have lesser variances and are fairly constant throughout (blue sky), whereas the backgrounds of other images like trucks and horses from CIFAR-10 dataset appears to vary largely across the dataset, potentially resulting in a greater difficulty for the Generator to learn to produce clearer images.

Fairness Generative Adversarial Networks (FGANs)

In this section, I will discuss about my attempt at my second objective of replicating the results of the paper - Fairness GAN: Generating Datasets with Fairness Properties Using a Generative Adversarial Network by Sattigeri et. al. The dataset which I chose to train my GAN on was CelebA.

Note that the Fairness GAN research paper did not explicitly state the hyperparameters used and only briefly glossed over the architecture used. As such, the architecture which I use might not be exactly the same as those used in the Fairness GAN research paper. Also note that Fairness GAN was built on the concept of AC-GAN which was discussed in the previous section.

As mentioned in the research paper, the researchers manually cropped and resized the CelebA images to 64 x 64 pixels. However, I was unable to find the cropped and resized images online and had to pre-process the images myself. In the downloaded CelebA dataset, there are coordinates of the bounding box for each image. Using these coordinates, I managed to crop the images of the celebrities but later noticed that the resulting images often showed a small portion of the celebrity's face and found the bounding boxes coordinates to be erroneous.

Thus, I came out with a method to crop the images. Referring to the landmark coordinates provided by the CelebA dataset, which gave coordinates of key features like noses and eyes of a human face, I found that using the left eye coordinate and translating the coordinate by 15 pixels leftwards and upwards gives a good representation of the left corner of the celebrity's face. Thus, I used that point as the top left corner coordinate to crop out a 64x64 pixel region using the package PIL.

I also followed the research paper and extracted the class labels C, in this case whether the celebrity is a male, and the outcome Y, in this case whether the celebrity is attractive, from the CelebA dataset using the Pandas library.

Below shows the high-level architecture I used in trying to replicate the results of the Fairness GAN:

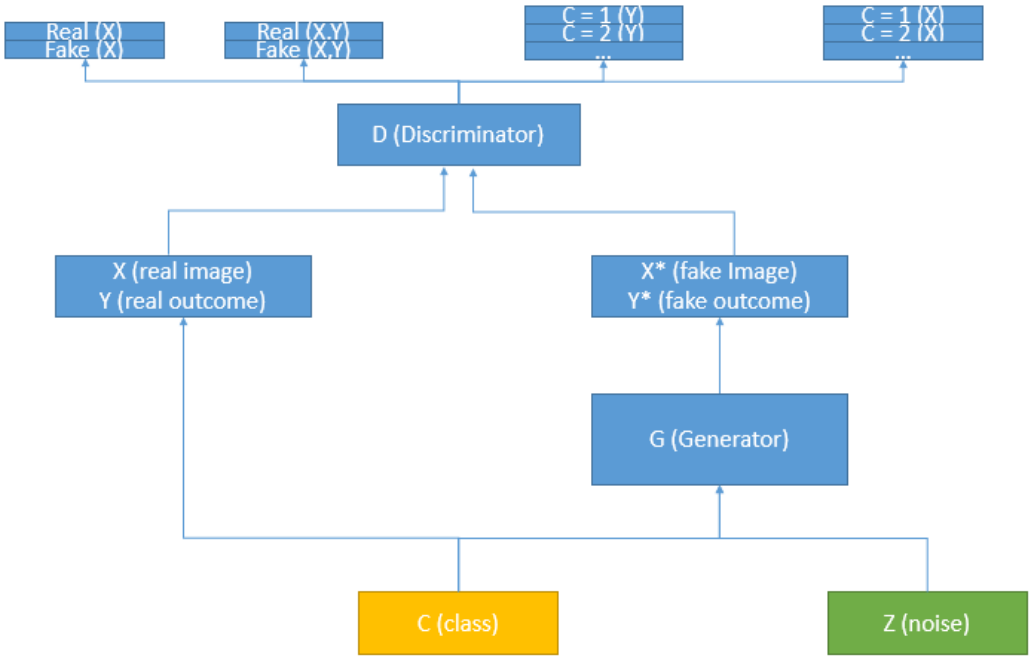


Figure 18: High-level architecture of Fairness GAN model.

Below is the architecture used for the Generator:

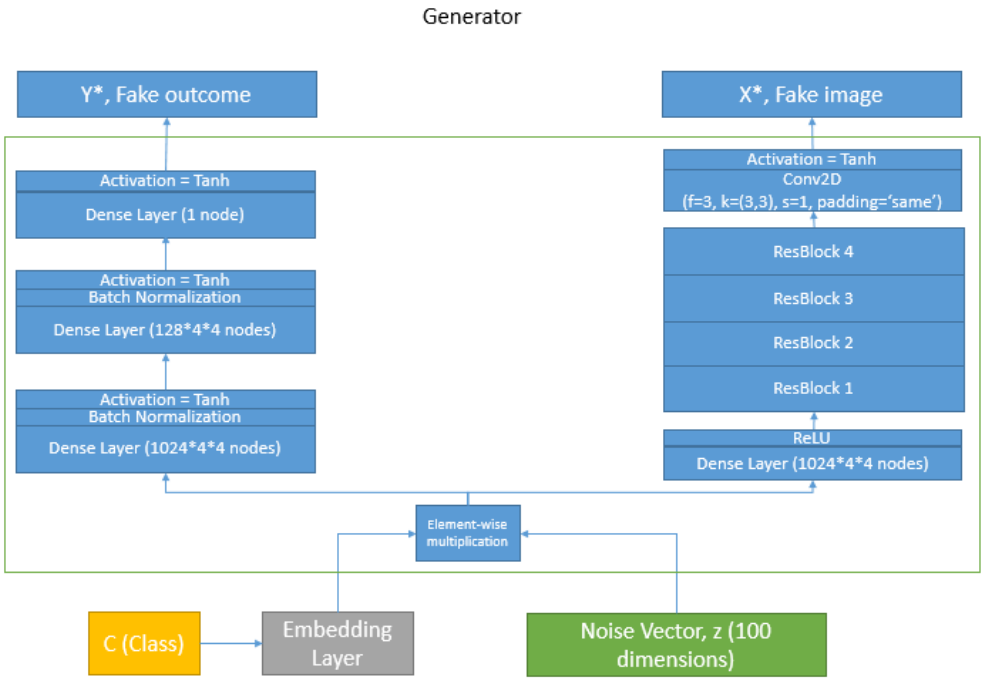


Figure 19: Architecture of Generator used for Fairness GAN.

Below shows the architecture used for the Discriminator:

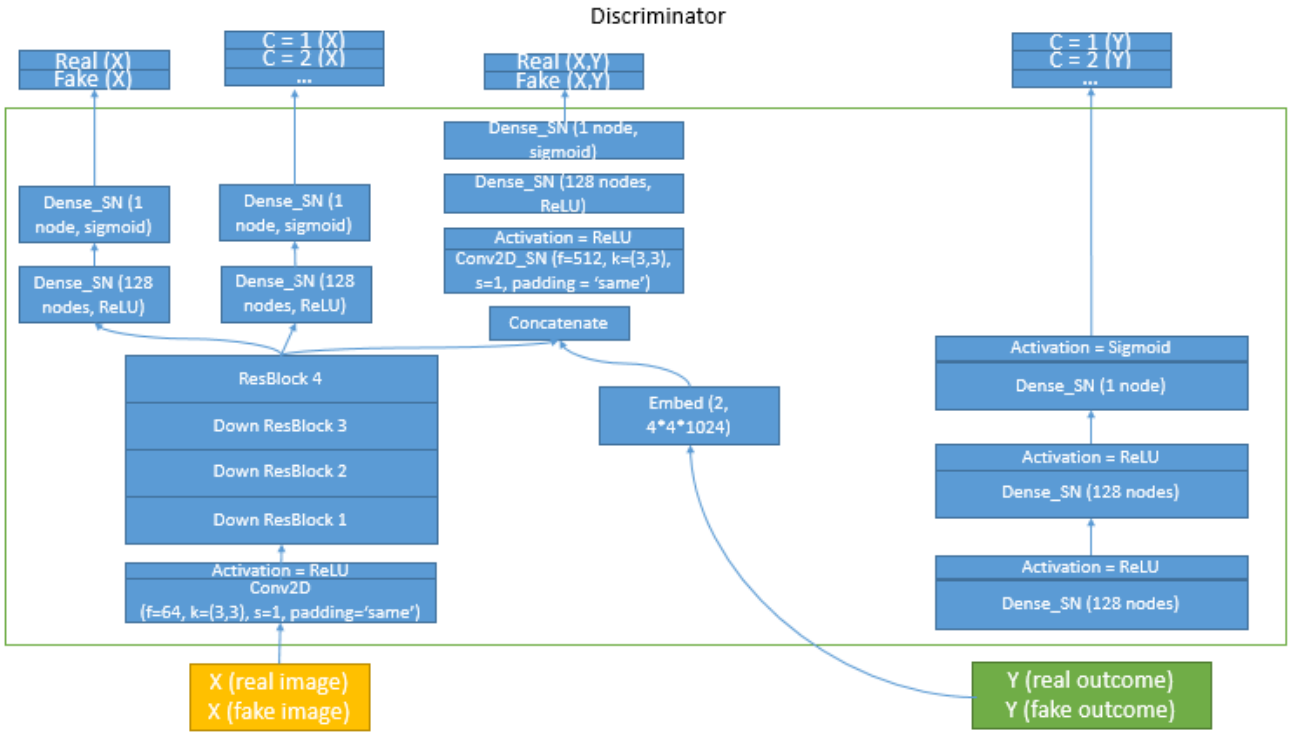


Figure 20: Architecture of Discriminator used in Fairness GAN.

Generator ResBlock architecture:

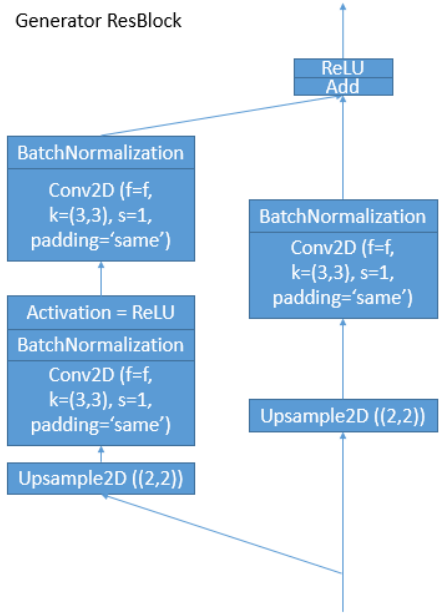


Figure 21: Generator ResBlock – Filter sizes for each block are 1024, 512, 256, 128 respectively.

Discriminator ResBlock architecture:

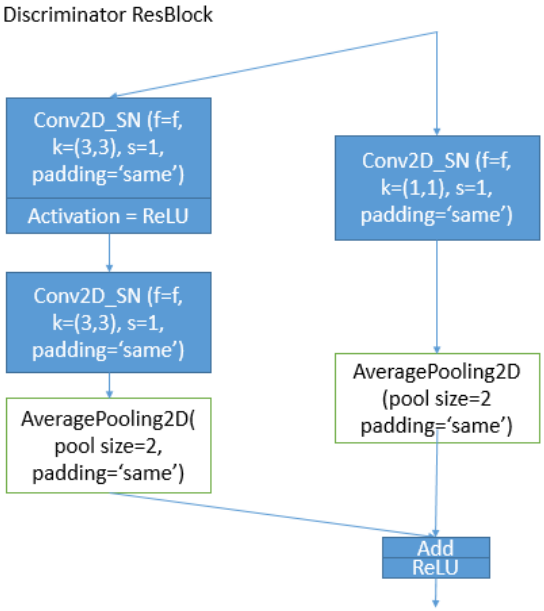


Figure 22: Discriminator ResBlock – Filter sizes for each block are 128, 256, 512, 1028 respectively. Note that the average pooling was only applied to the blocks 1, 2, and 3.

	FGAN - Generator						
	Operation	Kernel	Strides	Feature maps	BN?	Dropout	Nonlinearity
	G(z) - 100 x 1 x 1 input						
Image Path	Linear	N/A	N/A	16384	No	0.0	ReLU
	ResBlock 1						
	(Identity) Upsample2D	2 x 2	N/A	1024 x 8 x 8	No	0.0	NIL
	(Identity) Convolution	3 x 3	1 x 1	1024	Yes	0.0	NIL
	(Shortcut) Upsample2D	2 x 2	N/A	1024 x 8 x 8	No	0.0	NIL
	(Shortcut) Convolution	3 x 3	1 x 1	1024	Yes	0.0	ReLU
	(Shortcut) Convolution	3 x 3	1 x 1	1024	Yes	0.0	NIL
	(Add) Identity & Shortcut	N/A	N/A	(Element-Wise Addition)	No	0	ReLU
	ResBlock 2						
	(Identity) Upsample2D	2 x 2	N/A	512 x 16 x 16	No	0.0	NIL
	(Identity) Convolution	3 x 3	1 x 1	512	Yes	0.0	NIL
	(Shortcut) Upsample2D	2 x 2	N/A	512 x 16 x 16	No	0.0	NIL
	(Shortcut) Convolution	3 x 3	1 x 1	512	Yes	0.0	ReLU
	(Shortcut) Convolution	3 x 3	1 x 1	512	Yes	0.0	NIL
	(Add) Identity & Shortcut	N/A	N/A	(Element-Wise Addition)	No	0	ReLU
	ResBlock 3						
	(Identity) Upsample2D	2 x 2	N/A	256 x 32 x 32	No	0.0	NIL
	(Identity) Convolution	3 x 3	1 x 1	256	Yes	0.0	NIL
	(Shortcut) Upsample2D	2 x 2	N/A	256 x 32 x 32	No	0.0	NIL
	(Shortcut) Convolution	3 x 3	1 x 1	256	Yes	0.0	ReLU
	(Shortcut) Convolution	3 x 3	1 x 1	256	Yes	0.0	NIL
	(Add) Identity & Shortcut	N/A	N/A	(Element-Wise Addition)	No	0	ReLU
	ResBlock 4						
	(Identity) Upsample2D	2 x 2	N/A	128 x 64 x 64	No	0.0	NIL
	(Identity) Convolution	3 x 3	1 x 1	128	Yes	0.0	NIL
	(Shortcut) Upsample2D	2 x 2	N/A	128 x 64 x 64	No	0.0	NIL
	(Shortcut) Convolution	3 x 3	1 x 1	128	Yes	0.0	ReLU
	(Shortcut) Convolution	3 x 3	1 x 1	128	Yes	0.0	NIL
	(Add) Identity & Shortcut	N/A	N/A	(Element-Wise Addition)	No	0	ReLU
	End of ResBlocks						
	Convolution	3 x 3	1 x 1	3	No	0.0	Tanh
Outcome Model	Linear	N/A	N/A	16384	Yes	0.0	Tanh
	Linear	N/A	N/A	2048	Yes	0.0	Tanh
	Linear	N/A	N/A	1	No	0.0	Sigmoid
	Generator Optimizer	Adam (learning rate = 0.001, beta 1 = 0.9, beta 2 = 0.999)					
	Weight, bias initialization	Glorot Uniform, Constant(0)					

Table 5: Hyperparameters used for Generator in FGAN.

Below shows the hyperparameters used by the Discriminator in the FGAN model:

FGAN - Discriminator						
Operation	Kernel	Strides	Feature maps	BN/SN?	Dropout	Nonlinearity
D(x, y) - 64 x 64 x 3 Image Input (x) - 1 Outcome Input (y)						
Convolution	3 x 3	1 x 1	64	No	0.0	ReLU
ResBlock 1						
(Identity) Convolution	1 x 1	1 x 1	128	SN	0.0	ReLU
(Identity) Average Pooling	2 x 2	N/A	128 x 32 x 32	No	0.0	NIL
(Shortcut) Convolution	3 x 3	1 x 1	128	SN	0.0	ReLU
(Shortcut) Convolution	3 x 3	1 x 1	128	SN	0.0	ReLU
(Shortcut) Average Pooling	2 x 2	N/A	128 x 32 x 32	No	0.0	NIL
(Add) Identity & Shortcut	N/A	N/A	(Element-Wise Addition)	No	0.0	ReLU
ResBlock 2						
(Identity) Convolution	1 x 1	1 x 1	256	SN	0.0	ReLU
(Identity) Average Pooling	2 x 2	N/A	256 x 16 x 16	No	0.0	NIL
(Shortcut) Convolution	3 x 3	1 x 1	256	SN	0.0	ReLU
(Shortcut) Convolution	3 x 3	1 x 1	256	SN	0.0	ReLU
(Shortcut) Average Pooling	2 x 2	N/A	256 x 16 x 16	No	0.0	NIL
(Add) Identity & Shortcut	N/A	N/A	(Element-Wise Addition)	No	0.0	ReLU
ResBlock 3						
(Identity) Convolution	1 x 1	1 x 1	512	SN	0.0	ReLU
(Identity) Average Pooling	2 x 2	N/A	512 x 8 x 8	No	0.0	NIL
(Shortcut) Convolution	3 x 3	1 x 1	128	SN	0.0	ReLU
(Shortcut) Convolution	3 x 3	1 x 1	128	SN	0.0	ReLU
(Shortcut) Average Pooling	2 x 2	N/A	512 x 8 x 8	No	0.0	NIL
(Add) Identity & Shortcut	N/A	N/A	(Element-Wise Addition)	No	0.0	ReLU
ResBlock 4						
(Identity) Convolution	1 x 1	1 x 1	1024	SN	0.0	ReLU
(Identity) Average Pooling	2 x 2	N/A	1024 x 4 x 4	No	0.0	NIL
(Shortcut) Convolution	3 x 3	1 x 1	1024	SN	0.0	ReLU
(Shortcut) Convolution	3 x 3	1 x 1	1024	SN	0.0	ReLU
(Shortcut) Average Pooling	2 x 2	N/A	1024 x 4 x 4	No	0.0	NIL
(Add) Identity & Shortcut	N/A	N/A	(Element-Wise Addition)	No	0.0	ReLU
End of ResBlocks						
Shared Architecture for SJ_XY, SX_X, C_X						
SX_X (Input						
Linear	N/A	N/A	128	SN	0.0	ReLU
Linear	N/A	N/A	1	SN	0.0	Sigmoid
C_X						
Linear	N/A	N/A	128	SN	0.0	ReLU
Linear	N/A	N/A	1	SN	0.0	Sigmoid
SJ_XY						
Convolution	3 x 3	1 x 1	512	SN	0.0	ReLU
Linear	N/A	N/A	128	SN	0.0	ReLU
Linear	N/A	N/A	1	SN	0.0	Sigmoid
C_Y (Input Y)						
Linear	N/A	N/A	128	SN	0.0	ReLU
Linear	N/A	N/A	128	SN	0.0	ReLU
Linear	N/A	N/A	1	SN	0.0	Sigmoid
Discriminator Optimizer	Adam (learning rate = 0.001, beta 1 = 0.9, beta 2 = 0.999)					
Weight, bias initialization	Glorot Uniform, Constant(0)					

Table 6: Hyperparameters used by Discriminator in FGAN.

Below shows the images produced by the FGAN at 10,000 iterations.

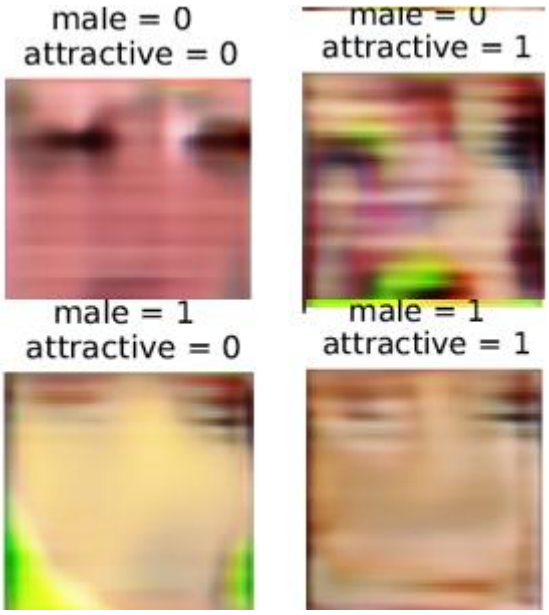


Figure 23: Images produced by FGAN at 10,000 iterations with batch size 64.

Conclusion

Looking at the above images generated, the GAN seem not to be learning, producing extremely blur images. This could perhaps be due to the occlusion in the method used in cropping and pre-processing the celebrity faces. Due to the lack of time, I will be unable to test out this hypotheses by the end of the internship. In addition, I was unable to implement inner product as proposed by the research paper and used concatenation instead. While I was able to find Spectral Normalization (for discriminator) implementations online for Keras, I was unable to find similar sources for Conditional Batch Normalization which was supposed to be used on the Generator. As such, I replaced Conditional Batch Normalization with the normal Batch Normalization. These could also be reasons for the poor quality of images produced by the Generator.