Analysis Of Semantic And Stylistic Image Generation

Merly Thomas, Kenrick Fernandes, Jerome Nicholas

Professor, Student, Student
Computer Engineering,
Fr. CRCE, Mumbai, India.

Abstract : Semantic Image Generation refers to the task of generating photorealistic images conditioning on some input data. This task is carried out by a specific set of neural networks called Generative Adversarial Networks. These are a set of neural networks which work opposed to each other, evolving from each other's successes.
The generator is a convolution network that outputs some image, while the discriminator is a network that classifies said image. The job of the discriminator is to perfectly identify an image as fake or real, while the generator's job is to try to produce realistic images.
Combining Convolutional Neural Networks with a technique called Spatially Adaptive Normalization (which is similar to Batch Normalization), the results of semantic image generation tend to be less washed out, owing to the semantic information being "retained" throughout the network.
Stylistic transfer is added by using Variational Auto Encoders, which take as input an image and breaks it down into its latent space. The latent space is then used to transfer the input image's style to the output of the semantic image generator. The results of our project have tended towards photorealism after about 50 epochs, with further training promising even better results. Transferring the learned networks to a mobile app, we will be able to allow users to create content of cities with minimal effort.

Keywords: Semantic Image Generation, Style Transfer, GANs.

I. INTRODUCTION

The conditional generative adversarial network, or cGAN for short, is a type of GAN that involves the conditional generation of images by a generator model.

Image generation can be conditional on a class label, if available, allowing the targeted generated of images of a given type. A specific form of conditional image synthesis is used, which is converting a semantic segmentation mask to a photorealistic image. This form has a wide range of applications such as content generation, image editing, interior designing, architectural designing etc. The model built to perform the task is a stacked Convolutional Neural Network, paired with Spatially Adaptive De-Normalization and Non-linearity layers (RELU specifically). Spatially Adaptive De-Normalization is a conditional normalization layer that modulates the activations using input semantic layouts through a spatially adaptive, learned transformation and can effectively propagate the semantic information throughout the network. This experiment was conducting by training the model in Google Colab over several weeks, using a backend with GPUs. Tensorflow is used to build the input pipeline, the model and its output. The dataset used for this task is the 'COCO-Stuff' dataset, available at http://cocodataset.org/

II. PROBLEM STATEMENT

To generate photo-realistic images from a basic solid colored drawing in a mobile platform. The images will belong to a set of objects dictated by the COCO-stuff dataset.

The goal of the project is photo-realism to the naked human eye. To achieve this, Spatially Adaptive Normalization is combined with Generative Adversarial Networks. Previous methods include stacking convolutional, normalization, and non linearity layers. This is at best sub-optimal, because the normalization layers tend to "wash away" information in input semantic masks.

To tackle this, the model built to perform the task is a stacked Convolutional Neural Network, paired with Spatially Adaptive De-Normalization and Non-linearity layers (RELU specifically). By using these deep learning technologies, we propose a system that will be able to generate photo-realistic images to the user's satisfaction.

III. OBJECTIVES

The aim of this project is multipurpose:

- To study the new and hot topic in Deep Learning – Generative Adversarial Networks.
- To study Spatially Adaptive Normalization and the effects it has on the outputs.
- To study Variational Auto Encoders and how efficient it is in image style transfer.
- To study how computationally effective the trained generator would be in an android app for semantic image generation.
- To study how certain variations in implementations can affect the outcome of the model – using different generator architectures, using SPADE with different GAN architectures and comparing them to GAN architectures without SPADE, using different kernel sizes in convolutions etc.

IV. LITERATURE REVIEW

The papers used in this project are named:

[1] Taesung Park, Ming-Yu Liu,Ting-Chun Wang, Jun-Yan Zhu. Semantic Image Synthesis with Spatially-Adaptive Normalization arxiv 2019. arXiv preprint arXiv:1903.07291v1 [cs.CV] 18 Mar 2019.

[2] Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, Alexei A. Efros. Image-to-Image Translation with Conditional Adversarial Networks 2018. arXiv preprint arXiv:1611.07004v3 [cs.CV] 26 Nov 2018 .

[3] Ting-Chun Wang, Ming-Yu Liu, Jun-Yan Zhu, Andrew Tao, Jan Kautz, Bryan Catanzaro. High-Resolution Image Synthesis and Semantic Manipulation with Conditional GANs arxiv 2018. arXiv preprint : arXiv:1711.11585v2 [cs.CV] 20 Aug 2018.

[1], published by Nvidia, describes a novel technique of normalization that produces photorealism better compared to any other method tried before. This normalization technique is called Spatially Adaptive deNormalization.

It is similar to Batch Normalization – In SPADE, the input semantic information is convolved with two trainable parameters, beta and gamma. The activation at each layer is standardized using the mean and standard deviation of each channel, and then convolved with the beta and gamma parameters.

This allows for the semantic information to be spread across the neural network, thus improving the “de-washing” capabilities of the networks.

SPADE ensures that no information in the input is lost at any stage or layer of the neural network, thus enabling the generator to not only create far realer images, but also images that mimic the input semantic information to a higher degree.

[2], published by UC Berkley, investigates conditional adversarial networks as a general-purpose solution to image-to-image translation problems. These networks not only learn the mapping from input image to output image, but also learn a loss function to train this mapping. This makes it possible to apply the same generic approach to problems that traditionally would require very different loss formulations.

They demonstrate that this approach is effective at synthesizing photos from label maps, reconstructing objects from edge maps, and colorizing images, among other tasks. Indeed, since the release of the pix2pix software associated with this paper, a large number of internet users (many of them artists) have posted their own experiments with the system, further demonstrating its wide applicability and ease of adoption without the need for parameter tweaking. As a community, we no longer hand-engineer our mapping functions, and this work suggests we can achieve reasonable results without hand-engineering our loss functions either.

[3], published by Nvidia and UC Berkley, details a new method for synthesizing high resolution photo-realistic images from semantic label maps using conditional generative adversarial networks (conditional GANs).

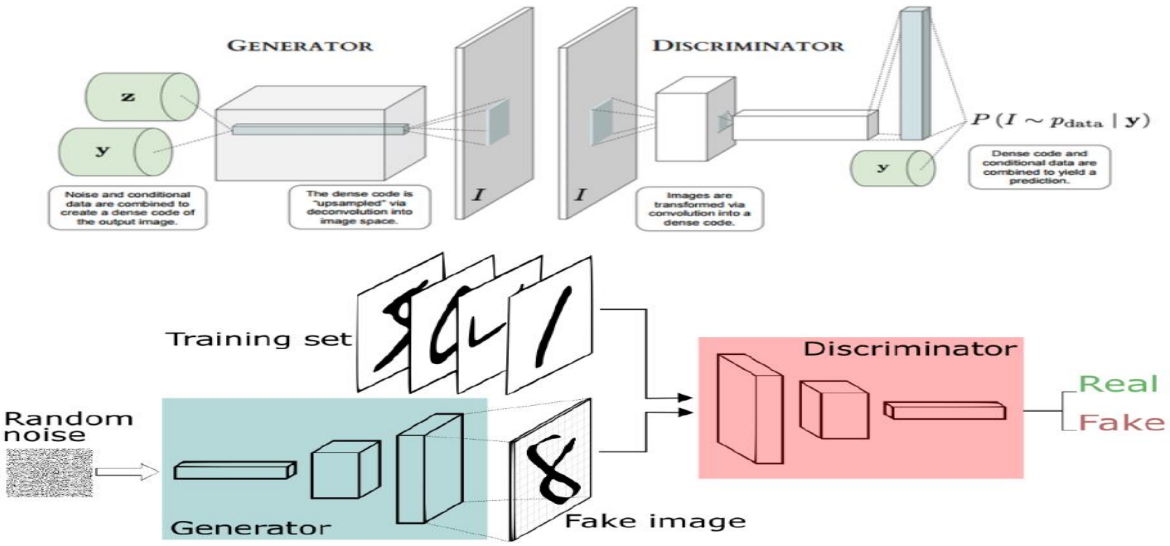
Conditional GANs have enabled a variety of applications, but the results are often limited to low resolution and still far from realistic. In this work, they generate 2048 × 1024 visually appealing results with a novel adversarial loss, as well as new multi-scale generator and discriminator architectures.

Furthermore, they extend a framework to interactive visual manipulation with two additional features. First, they incorporate object instance segmentation information, which enables object manipulations such as removing/adding objects and changing the object category.

Second, they propose a method to generate diverse results given the same input, allowing users to edit the object appearance interactively. Human opinion studies demonstrate that our method significantly outperforms existing methods, advancing both the quality and the resolution of deep image synthesis and editing.

V. PROPOSED SYSTEM

Data Design for Generative Adversarial Network -



The generator is inputted with random noise that gets convolved and normalized using SPADE to produce a fake image. The original training set is concatenated with the fake image set and inputted to the Discriminator. The Discriminator then calculates the logprob of the image being Real or Fake.

VI. ACQUIRING DATA

Our first task was to obtain the dataset. The dataset that we have used to train the network is COCO Stuff provided at http://cocodataset.org/

VII. PREPROCESSING DATA

The various techniques used to pre-process the images are as follows -
•Resize – Resize the image to a size (height x width) as inputted by the user.
•Scale Width – Scale the width of the image to a target width as inputted by the user.
•Scale Short side – Find out the shorter dimension of the image and scale that to a target size inputted by the user.
•Crop – Crop the image to capture the most important features. .
•Flip – Flip the image i.e. use matrix transpose to flip left and right side of the image.

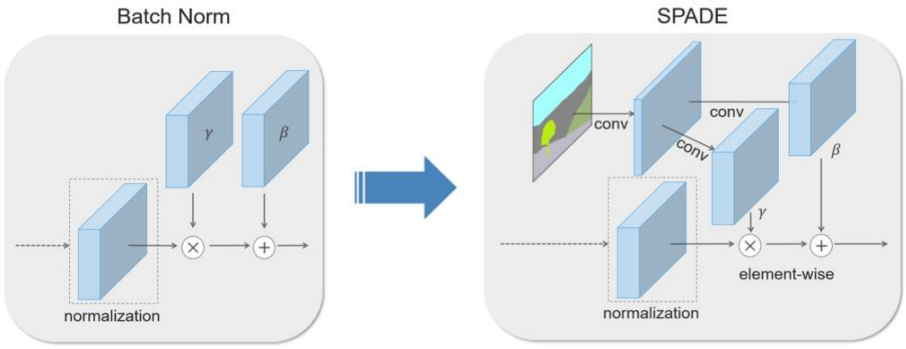
These preprocessing techniques will allow the Deep Learning model to be more robust and not just generalize around the data. Not all the preprocessing techniques were used in this iteration of the project. Our goal is to test the model with all the techniques used.

IX. CREATING THE MODEL

SPADE is the cornerstone of the project. Spatially Adaptive De-Normalization is a conditional normalization layer that modulates the activations using input semantic layouts through a spatially adaptive, learned transformation and can effectively propagate the semantic information throughout the network. Using TensorFlow, the gamma and beta parameterized convolutions were created. The functionality of SPADE comes from this formula –

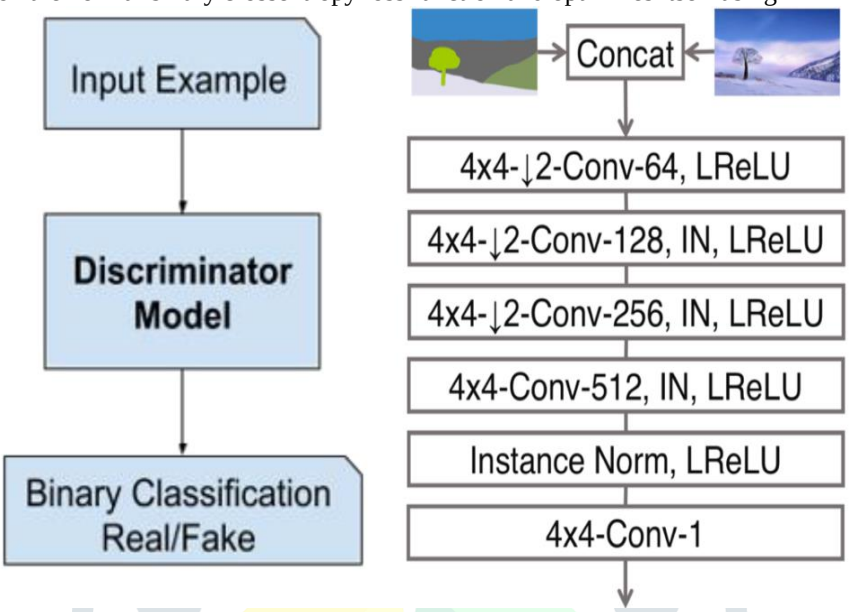
$$\gamma^i_{c,y,x}(\mathbf{m})\frac{h^i_{n,c,y,x}-\mu^i_c}{\sigma^i_c}+\beta^i_{c,y,x}(\mathbf{m})$$

This can be visualized in the model as such -

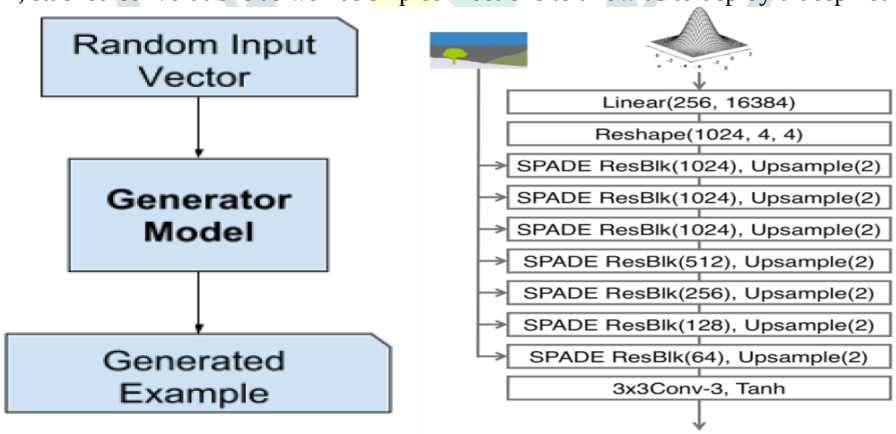


X. Implementing Generator, Encoder And Discriminator

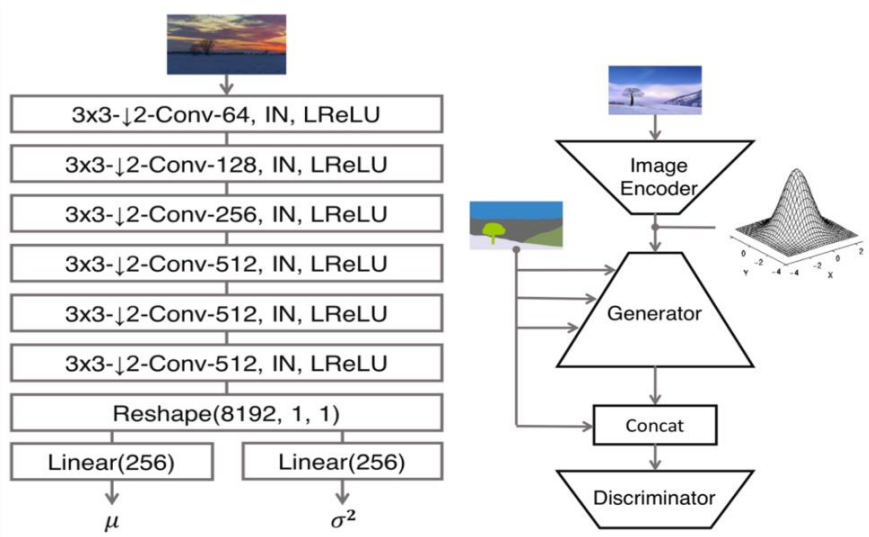
The discriminator model takes an example from the domain as input (real or generated) and predicts a binary class label of real or fake (generated). It runs on the normal binary crossentropy loss function and optimizes itself using ADAM .



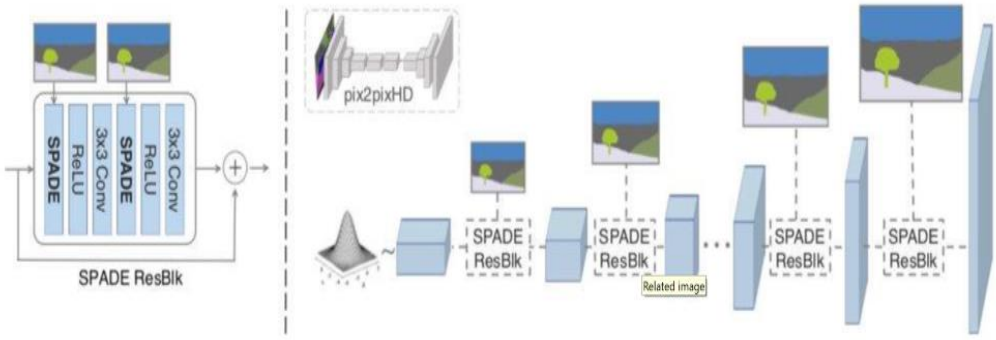
The generator model takes a fixed-length random vector as input and generates a sample in the domain. The generator is constructed using SPADE, stacked convolutions as well as skip connections to allow us to deploy a deep neural network.



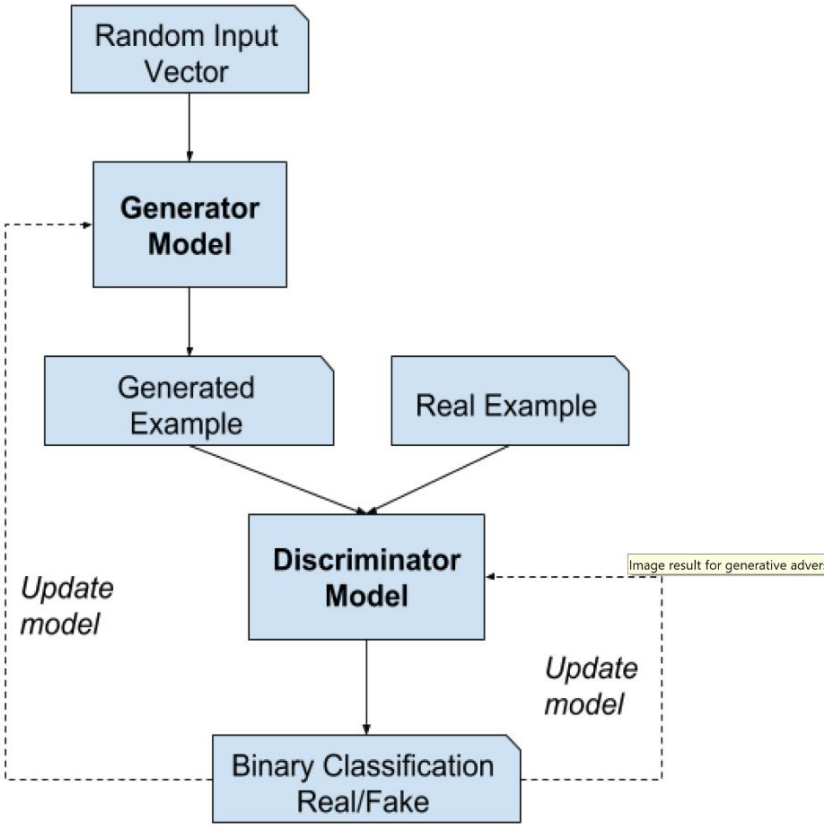
The image encoder is a Variational Auto Encoder used to transfer an images style to another image. This is basically a stacked layer of convolutions which embeds the input image to a latent space characterized by its standard deviation and mean; used to create images of the same latent variables.



XI. PIPELINES WORKFLOW



The Discriminator is pre-trained on the original input distribution. The generator is inputted with a random noise vector. The generator convolves the input and uses the SPADE Resblock to learn the modulation parameters from the activation of that layer. The Image encoder plays a key role in making the neural network perceive the input images. The Discriminator is trained on cross entropy classification loss The generator is trained to mimic the input distribution, fooling the discriminator. The image encoder takes as input an image, embeds it into a latent space, and that latent space is used to transfer its style to the output image of the generator. The result is a stylized image generated from the users input segmentation mask.



XII. Training Configuration

Criteria	Original Paper	Our Implementation
Num Filters in SPADE	128	128
COCO Image Size	512x256	256x256
Batch Size	20x8 GPU	1 GPU
No.Upsampling layers in generator	7	4
No.Epochs	200	68
Filters in Discriminator	More	Half Of Original
GPU Used	NVIDIA V100	Tesla T4

Table 1: Training Configuration Parameters

XIII. Training Epochs

The model was trained using Google Colab's GPU Environment. An epoch took about 5000 seconds to complete, and hence we were limited to training the model for only 68 epochs. Each training step took about 1.5 seconds, and each epoch was limited to 5000 iterations.

Below is a graphic depicting iterations within epoch 3, depicting the losses calculated after each iteration and the time taken to complete an iteration.

```
Epoch: [ 3] [ 1725/ 5000] time: 9227.4518 d_loss: 1.57648921, g_loss: 15.96718311
Epoch: [ 3] [ 1726/ 5000] time: 9228.4449 d_loss: 0.42006695, g_loss: 16.95077515
Epoch: [ 3] [ 1727/ 5000] time: 9229.3877 d_loss: 0.89508855, g_loss: 16.22129631
Epoch: [ 3] [ 1728/ 5000] time: 9230.3392 d_loss: 1.23623860, g_loss: 16.49508286
Epoch: [ 3] [ 1729/ 5000] time: 9231.3125 d_loss: 2.15958166, g_loss: 18.81159019
Epoch: [ 3] [ 1730/ 5000] time: 9232.2600 d_loss: 0.75763416, g_loss: 16.83553696
Epoch: [ 3] [ 1731/ 5000] time: 9233.2826 d_loss: 1.37750304, g_loss: 18.09132385
Epoch: [ 3] [ 1732/ 5000] time: 9234.2494 d_loss: 1.09959745, g_loss: 15.39035988
Epoch: [ 3] [ 1733/ 5000] time: 9235.2002 d_loss: 1.42712724, g_loss: 16.94935417
Epoch: [ 3] [ 1734/ 5000] time: 9236.1533 d_loss: 1.43191707, g_loss: 15.73798561
Epoch: [ 3] [ 1735/ 5000] time: 9237.1880 d_loss: 1.53728795, g_loss: 15.44825172
Epoch: [ 3] [ 1736/ 5000] time: 9238.1695 d_loss: 1.13327193, g_loss: 15.44215775
Epoch: [ 3] [ 1737/ 5000] time: 9239.2239 d_loss: 1.66768718, g_loss: 16.48461914
Epoch: [ 3] [ 1738/ 5000] time: 9240.1891 d_loss: 1.65717578, g_loss: 16.98602486
Epoch: [ 3] [ 1739/ 5000] time: 9241.1536 d_loss: 0.70468009, g_loss: 17.74939346
Epoch: [ 3] [ 1740/ 5000] time: 9242.1920 d_loss: 1.13893771, g_loss: 16.27427483
Epoch: [ 3] [ 1741/ 5000] time: 9243.1318 d_loss: 0.56304580, g_loss: 13.19570827
Epoch: [ 3] [ 1742/ 5000] time: 9244.0786 d_loss: 1.75879931, g_loss: 18.10719872
Epoch: [ 3] [ 1743/ 5000] time: 9245.1335 d_loss: 0.06242421, g_loss: 17.10961533
Epoch: [ 3] [ 1744/ 5000] time: 9246.1638 d_loss: 0.89282334, g_loss: 13.98987579
Epoch: [ 3] [ 1745/ 5000] time: 9247.1942 d_loss: 1.75388658, g_loss: 18.16321945
Epoch: [ 3] [ 1746/ 5000] time: 9248.1387 d_loss: 0.51961529, g_loss: 17.68027496
Epoch: [ 3] [ 1747/ 5000] time: 9249.1830 d_loss: 1.62220800, g_loss: 15.04948330
Epoch: [ 3] [ 1748/ 5000] time: 9250.1219 d_loss: 2.12890577, g_loss: 15.41708565
Epoch: [ 3] [ 1749/ 5000] time: 9251.1525 d_loss: 1.74715400, g_loss: 15.37134743
Epoch: [ 3] [ 1750/ 5000] time: 9252.0931 d_loss: 0.97964847, g_loss: 18.34981155
Epoch: [ 3] [ 1751/ 5000] time: 9253.0574 d_loss: 1.42623925, g_loss: 15.50289345
Epoch: [ 3] [ 1752/ 5000] time: 9254.0165 d_loss: 2.28261042, g_loss: 16.92005920
Epoch: [ 3] [ 1753/ 5000] time: 9255.0756 d_loss: 0.93993592, g_loss: 15.24379349
Epoch: [ 3] [ 1754/ 5000] time: 9256.0296 d_loss: 0.58856678, g_loss: 15.27865601
Epoch: [ 3] [ 1755/ 5000] time: 9257.0821 d_loss: 2.48501754, g_loss: 14.28540802
Epoch: [ 3] [ 1756/ 5000] time: 9258.1054 d_loss: 0.93596137, g_loss: 16.26085281
Epoch: [ 3] [ 1757/ 5000] time: 9259.0592 d_loss: 0.91887963, g_loss: 16.45780563
Epoch: [ 3] [ 1758/ 5000] time: 9260.1684 d_loss: 1.18697119, g_loss: 16.37429810
Epoch: [ 3] [ 1759/ 5000] time: 9261.1349 d_loss: 1.61051607, g_loss: 17.13753700
Epoch: [ 3] [ 1760/ 5000] time: 9262.0968 d_loss: 1.32238829, g_loss: 15.27259254
Epoch: [ 3] [ 1761/ 5000] time: 9263.1307 d_loss: 1.83663297, g_loss: 16.77178955
Epoch: [ 3] [ 1762/ 5000] time: 9264.1623 d_loss: 1.96604896, g_loss: 18.36571121
Epoch: [ 3] [ 1763/ 5000] time: 9265.1967 d_loss: 1.93057907, g_loss: 15.42958736
Epoch: [ 3] [ 1764/ 5000] time: 9266.2433 d_loss: 1.25376976, g_loss: 16.66280365
```

XIII. TENSORBOARD VISUALIZATIONS

We have used Tensorboard to visualize the various losses calculated with reference to the model's performance.

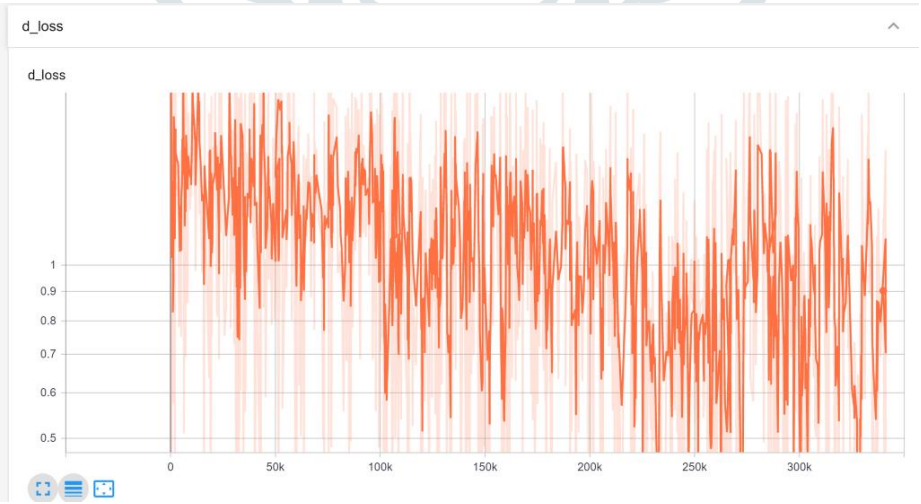
•D(x) is the critic's output for a real instance.
•G(z) is the generator's output when given noise z.
•D(G(z)) is the critic's output for a fake instance.
•The output of critic D does not have to be between 1 and 0.

The discriminator loss is calculated as follows:

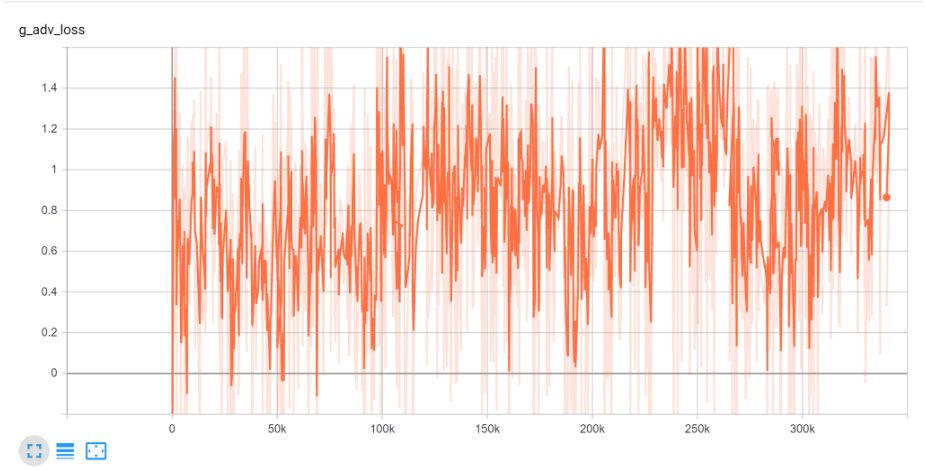
$$D(x) - D(G(z)) \quad (1)$$

The discriminator tries to maximize this function. In other words, it tries to maximize the difference between its output on real instances and its output on fake instances.

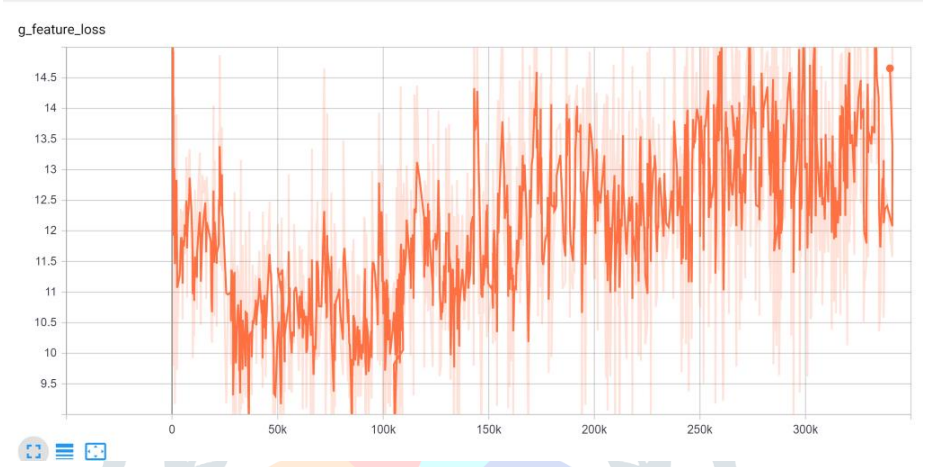
The discriminator loss is shown below.



The Generator Adversarial Loss is shown below.



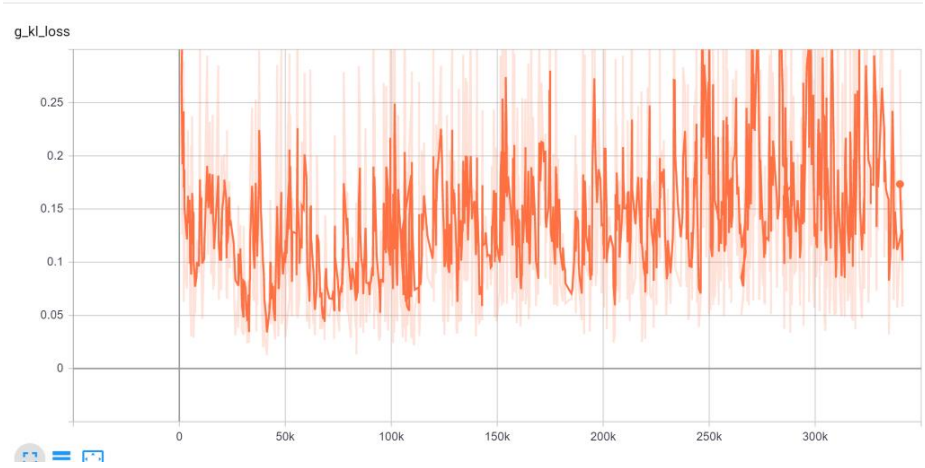
The Generator Feature Loss is shown below.



The Kullback–Leibler divergence is the directed divergence between two distributions. It is calculated as follows:

$$D_KL(P||Q) = \Sigma P(x)\log(P(x)/Q(x)) \quad (2)$$

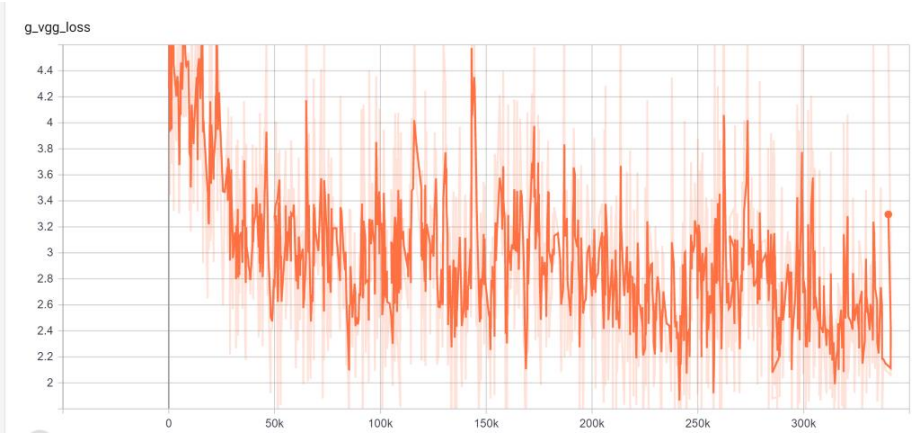
The Generator KLD Loss is shown below.



The Generator VGG Loss is the Perceptual Loss that measures the perceptual differences in content and style between images.

$$W = \text{argmin}_W E_{x,y}[\Sigma \lambda_i \text{Loss}_i(f_W(x), y_i)] \quad (3)$$

The Generator VGG Loss is shown below.

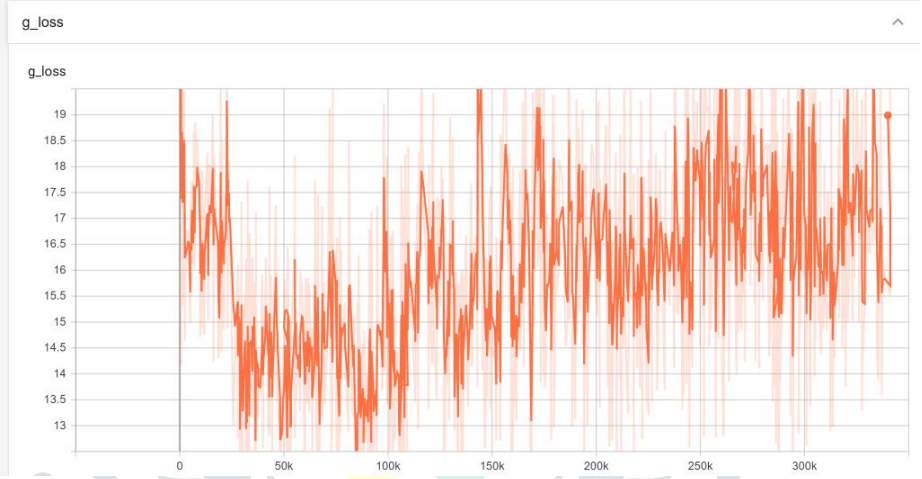


The Generator Loss is calculated as :

$$D(G(z)) \quad (4)$$

The generator tries to maximize this function. In other words, It tries to maximize the discriminator's output for its fake instances.

The Generator Loss is shown below.



XV. Hardware Requirement

The training part of the project was performed in Google Colab. A python 3.0 Backend GPU was deployed from the google cloud, with 13 GB of RAM and 350 GB of disk space. The RAM was used to hold the preprocessed images to be delivered to the networks for training. The disk space was used to contain the entire dataset, the python and tensorflow code and the Deep Learning Models.

The application part of the project was carried out using Android Studio. The Deep Learning Models were converted to TFlite and deployed in the Android App. As such, to hold the TFlite model as well as the app, a minimum Disk Space of 1Gb will be required, 2 GB RAM is recommended for smooth workflow, and a 2 Ghz Snapdragon/Apple processor is required to run the Deep Learning Model.

XVI. Software Requirement

Python 3.0 and Tensorflow are the dependencies required to train the Deep Learning Models on Google Colab. No Software dependency is required to run the application on the users end. All training, testing and conversions have been done on the google cloud. A user simply has to download the app on the Play Store to utilize it.

XVII. Results And Conclusions

After training the model for a few months, we reached an epoch level of 68. The results of our training have been fruitful in producing nearly photorealistic images. The training process is shown below –

```
Epoch: [26] [ 2254/ 5000] time: 2316.7047 d_loss: 0.88836586, g_loss: 15.85569191
Epoch: [26] [ 2255/ 5000] time: 2317.6797 d_loss: 2.37825727, g_loss: 15.42190933
Epoch: [26] [ 2256/ 5000] time: 2318.6356 d_loss: 1.33399975, g_loss: 12.54837608
Epoch: [26] [ 2257/ 5000] time: 2319.6813 d_loss: 1.55737758, g_loss: 14.85847950
Epoch: [26] [ 2258/ 5000] time: 2320.8169 d_loss: 0.16404612, g_loss: 16.47517967
Epoch: [26] [ 2259/ 5000] time: 2321.8373 d_loss: 0.22974265, g_loss: 11.67346382
Epoch: [26] [ 2260/ 5000] time: 2322.7873 d_loss: 2.42915106, g_loss: 12.75806618
Epoch: [26] [ 2261/ 5000] time: 2323.7423 d_loss: 1.23304915, g_loss: 16.93707657
Epoch: [26] [ 2262/ 5000] time: 2324.6935 d_loss: 1.25949001, g_loss: 13.75324440
Epoch: [26] [ 2263/ 5000] time: 2325.7156 d_loss: 1.18878734, g_loss: 13.70453262
Epoch: [26] [ 2264/ 5000] time: 2326.7849 d_loss: 0.33212137, g_loss: 17.20745850
Epoch: [26] [ 2265/ 5000] time: 2327.7410 d_loss: 1.64839435, g_loss: 10.84650135
Epoch: [26] [ 2266/ 5000] time: 2328.7548 d_loss: 1.13975811, g_loss: 15.19740772
Epoch: [26] [ 2267/ 5000] time: 2329.7286 d_loss: 1.73636806, g_loss: 15.88940334
Epoch: [26] [ 2268/ 5000] time: 2330.6820 d_loss: 1.53951335, g_loss: 13.58553600
Epoch: [26] [ 2269/ 5000] time: 2331.6338 d_loss: 0.50042880, g_loss: 13.32257271
Epoch: [26] [ 2270/ 5000] time: 2332.7232 d_loss: 0.97756749, g_loss: 12.52379894
Epoch: [26] [ 2271/ 5000] time: 2333.6756 d_loss: 1.14740920, g_loss: 18.14331055
Epoch: [26] [ 2272/ 5000] time: 2334.7096 d_loss: 1.08294952, g_loss: 14.49746132
Epoch: [26] [ 2273/ 5000] time: 2335.7971 d_loss: 1.27840924, g_loss: 15.40806580
Epoch: [26] [ 2274/ 5000] time: 2336.9072 d_loss: 1.19388473, g_loss: 14.14816380
Epoch: [26] [ 2275/ 5000] time: 2337.9489 d_loss: 1.29029346, g_loss: 13.89641666
Epoch: [26] [ 2276/ 5000] time: 2338.9149 d_loss: 0.72051752, g_loss: 13.95492172
Epoch: [26] [ 2277/ 5000] time: 2339.8732 d_loss: 0.32879597, g_loss: 16.43352509
Epoch: [26] [ 2278/ 5000] time: 2340.8990 d_loss: 0.39289087, g_loss: 14.59524536
Epoch: [26] [ 2279/ 5000] time: 2341.9394 d_loss: 0.71166462, g_loss: 11.17766953
Epoch: [26] [ 2280/ 5000] time: 2342.8921 d_loss: 1.00687528, g_loss: 13.98381805
Epoch: [26] [ 2281/ 5000] time: 2343.8483 d_loss: 1.16779625, g_loss: 15.60788155
Epoch: [26] [ 2282/ 5000] time: 2344.8083 d_loss: 1.20772636, g_loss: 14.95472240
Epoch: [26] [ 2283/ 5000] time: 2345.7627 d_loss: 0.91674125, g_loss: 13.44414711
Epoch: [26] [ 2284/ 5000] time: 2346.8737 d_loss: 0.78444707, g_loss: 16.27453041
Epoch: [26] [ 2285/ 5000] time: 2347.8359 d_loss: 0.16004391, g_loss: 14.71691799
Epoch: [26] [ 2286/ 5000] time: 2348.7902 d_loss: 0.57241720, g_loss: 18.92914963
Epoch: [26] [ 2287/ 5000] time: 2349.8172 d_loss: 1.67548847, g_loss: 17.15609932
Epoch: [26] [ 2288/ 5000] time: 2350.7661 d_loss: 1.69638634, g_loss: 14.01230717
Epoch: [26] [ 2289/ 5000] time: 2351.7140 d_loss: 0.40628117, g_loss: 16.03540993
Epoch: [26] [ 2290/ 5000] time: 2352.7350 d_loss: 1.41439867, g_loss: 13.56789780
Epoch: [26] [ 2291/ 5000] time: 2353.6907 d_loss: 0.11379062, g_loss: 19.61034012
Epoch: [26] [ 2292/ 5000] time: 2354.6452 d_loss: 0.27490667, g_loss: 12.14905262
Epoch: [26] [ 2293/ 5000] time: 2355.6887 d_loss: 0.49680448, g_loss: 15.67325020
Epoch: [26] [ 2294/ 5000] time: 2356.6352 d_loss: 2.37666345, g_loss: 16.19764709
```

As shown above, the GAN loss, VGG Loss and the Discriminators probabilities of fake and real images have been recorded. It is observed that the Discriminator is almost perfect in the start of the training, owing to the generator not being able to mimic the input distribution. But, with time, the Generator performs better and better, eventually fooling the discriminator into believing the input image belongs to the original distribution.

Our project has demonstrated that this new field of Deep Learning – Generative Adversarial Networks are very efficient at mimicking data distributions. This will be very important in the near future, where we will be able to mimic data for the use of DL, AI, ML and Big Data Application.

Our project dealt entirely with image data. The results observed were photorealistic and non- blurry. This itself is an indication of how effective Generative Adversarial Networks are in conditional image synthesis. The utilization of Spatially Adaptive de-Normalization affected the results in a profound way. It was observed that without SPADE, photorealism would not have been possible.

We therefore conclude that, SPADE in tandem with GANs were effective, efficient, but also computationally heavy in the synthesis of photorealistic images.

The entire Android App can be found on the github page linked : https://github.com/jeromenicholas07/sketchola_flask

REFERENCES

[1] Taesung Park, Ming-Yu Liu,Ting-Chun Wang, Jun-Yan Zhu. Semantic Image Synthesis with Spatially-Adaptive Normalization arxiv 2019. arXiv preprint arXiv:1903.07291v1 [cs.CV] 18 Mar 2019.

[2] Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, Alexei A. Efros. Image-to-Image Translation with Conditional Adversarial Networks 2018. arXiv preprint arXiv:1611.07004v3 [cs.CV] 26 Nov 2018 .

[3] Ting-Chun Wang, Ming-Yu Liu, Jun-Yan Zhu, Andrew Tao, Jan Kautz, Bryan Catanzaro. High-Resolution Image Synthesis and Semantic Manipulation with Conditional GANs arxiv 2018. arXiv preprint: arXiv:1711.11585v2 [cs.CV] 20 Aug 2018. Web Pages: Nvidias SPADE page - https://nvlabs.github.io/SPADE/