# A Closer Look at a cGAN Model for Universal Image-to-Image Transformation with MNIST Dataset and Potential Improvement on its Stochasticity

**Sainan Liu**
A13291871
sal131@eng.ucsd.edu

**Hao-en Sung**
A53204772
wrangle1005@gmail.com

**Shiwei Song**
A53206591
shs163@eng.ucsd.edu

**Haifeng Huang**
A53208823
hah086@eng.ucsd.edu

## Abstract

We have replicated the results for five proposed loss functions from Lua [11] to Keras [7] based on the Image-to-Image Translation with Conditional Adversarial Networks paper [5]. To evaluate the results, visual evaluation of the generated images, FCN(Fully Convolutional Networks for Semantic Segmentation)-score[9] and color distributions of CIELAB colors [4] are used. Some discrepancies were found between the two code versions and the problems with the Keras code are discussed in this report. To better understand the paper, we have added Lua code on top of the original model [6] to reconstruct image from every convolutional layer, and run the model through a couple of new datasets. The reconstruction showed us that the transformation is more complicated and global. We also trained the models on some new datasets. The datasets are all variations of MNIST [10] image transformations. By showing that the network can transform images from MNIST dataset to variations of fonts as well as images of Chinese number characters, we have further proved that the model is truly versatile and is very powerful for multiple-to-one transformations. Additionally, we explored and analyzed the stochasticity of the network based on the output images, the histogram images of all pixel values per image, and the PCA figures of different outputs from models using the same input image. We verified that higher dropout rate do tend to increase the stochasticity, and it can be even further improved by adding a random activation layer. However, there is trade-off effect between the stochasticity and the realistic-looks of the image.

## 1 Introduction

In the real world, we always encounter image-to-image translation problems. Imagine you are an architect and have designed the outline of a building for your client, but the client are not sure whether your design has a good look in the real world. It's apparent that the client is not willing to pay for the money to build it according to your design but finally finds that the appearance is what not they want, so you need to translate your design blueprint to real world images to satisfy your client. That is where our motivation comes in. If we can just draw a outline of the building and make the network help us figure out what it may looks like in the real world, we can save a lot of time and money to recruit a drawer to make the design realistic. There are many kinds of image-to-image translation tasks, for example, we can translate segmented images to real world images, black and white images to colored images, maps to aerial photographs, day images to night images, and etc. The other direction is also applicable.

Normal neural networks are often designed to solve specific problems, so they are not general enough to tackle all these problems with the same structure. The Image-to-Image Translation paper [5] proposed an interesting conditional Generative Adversarial Networks (cGAN) as a general-purpose solution to image-to-image translation problems. It is interesting to us because it was able to use a generalized neural network to tackle many types of problems in image processing and computer vision, such as translate GPS maps to google maps, turn black and white image to a colorful image, and even turn segmented image to a photo-realistic image, etc. These problems are difficult because they normally require different loss functions, and the translation is not one-to-one as is mentioned in [5]. Hence it is very important for us to analyze and learn more about the mechanism of this network and what we can do to improve its performance.

We want to first replicate the model in Keras code from the original Lua code and verify the figures from the paper on the Facades dataset [15].

After test the capabilities of cGAN network, we want to further study this model and use it to test its capability with the MNIST dataset. Although MNIST dataset is easily approachable, this dataset is usually used for classification tasks. Studies have rarely used it for image to image transformations. We think it is a simple but powerful experiment to prove the capability of the model for the multiple to one relationship, which has not yet been extensively explored in the original paper.

Additionally, the paper has mentioned in order to capture the full entropy of the conditional distributions that the cGAN models, we need to design a cGAN that produces stochastic output. This allows the same conditional input to produce different image content instead of one deterministic output. This area has not been explored much, so we wanted to come up with some method to visualize this problem better, and potentially improve the stochasticity of the output.

## 2 Background

### 2.1 Previous Studies

Generative Adversarial Network (GAN) is a kind of system consisting of two neural networks invented by Ian Goodfellow [3]. One is a generator and another is a discriminator. In learning procedures, they will compete against each other and improve their results. GAN are widely used in image generation. GAN has a feature that it treats each output pixels as conditionally independent with other pixels given input images, in other words, GAN is learning a unstructured loss.

GAN networks have been proven to be much more effective than previous models in generating realistic images, and have been widely studied in past few years[16] [17] [18]. To improve the performance,Conditional Generative Adversarial Network (cGAN) [21] is often used in most recent years. It is the conditional version of GAN. The discriminator would also take the source image as input to make its decision, the performance is often better than GAN model alone [21].

Some groups have studied the image-to-image transformation problem with different methods. In [19], they used multi-scale auto-regression to learn filters for the task. However, the application can only be restricted to filters. In [20], they used a locally affine model to transform single outdoor photo to photos at different times of a day. However, their model required a large amount of data and it is restricted to the specific task. People also have applied cGAN to some fields. For example, in [21], they used cGAN for tagging images. In [22], they used cGAN for video frame prediction. The problem with all of these frameworks is that they were designed for one specific task.

### 2.2 For Replication

The paper we based our project on is "Image-to-Image Translation Using Conditional Adversarial Networks" [5]. It is based on a cGAN model. The main achievement of the paper [5] is that the network automatically learns a loss function for their generative part of the model, which allows the model to be versatile enough to all type of image-to-image translations. Therefore, we will try to replicate the figures related to the different types of general loss functions they have compared in the paper

### 2.3 For Model Understanding

Additionally, previous study has shown that reconstructing images from convolutional layer can help review simple local transformations [2]. Hence we have decided to use a similar approach to take a deeper look at each layers of this network.

This paper tries to achieve universal tasks, such as labeled image to street scenes, BW image to color image and edges to photos. These are mostly one image to one image transformations, we would like to explore the multiple-to-one transformations using a variations of the MNIST dataset [10]. Unlike using the MNIST dataset for classification tasks, we will use it as an input and create as set of 10 target images per experiment to achieve many-to-one mapping and see how well this network can perform in this image-to-image translation process.

### 2.4 For Stochasticity

In previous research, [16] has tried to add Gaussian noise as an input to the generator in addition to x, but both [5] and [12] found that the generator would simply learn to ignore the noise. Therefore, [5] proposed to provide noise only in the form of dropout on several layers of the generator at both training and test time. However, they observe very minor stochasticity in the output of their nets. Hence we decided to build on top of their network, and further explore ways to improve the stochasticity of the output for cGAN. We would like to further explore the concept of stochasticity in terms of manipulating the dropout layers and potentially try to improve it.

## 3 Model

### 3.1 Introduction to cGANs

The conditional Generative Adversarial Network consists of two neural networks: a generator **G** and a discriminator **D**. In this image-to-image translation problem, given a sample pair $(X, T)$, where we call them input $X$ and target $T$ for example. The generator will take a sample $X$ and a random noise $Z$ as input and try to generate an output $Y$ that is as close to $T$ as possible. This process can be simplified as Equation 1. The goal is to train a good generator **G** that can do translations such as from segmented images to photos.

$$Y = G(X, Z) \tag{1}$$

The discriminator will take a pair $(X, Y)$ and try to figure out whether $Y$ is a real target or a fake target generated by the generator as is shown in Equation 2

$$D(X,Y) = \begin{cases} 1 & D \text{ regards } Y \text{ as a real target} \\ 0 & D \text{ regards } Y \text{ as a fake target} \end{cases} \tag{2}$$

The loss function we will use to train the two networks are:

$$L_D = \sum -t \log(D(x,y)) - (1-t)\log(1 - D(x, G(x,z))) \tag{3}$$

$$L_G = \sum -log(D(x, G(x,z))) \tag{4}$$

We will try to find $G^* = argmin_G max_D L_{cGAN}(G, D)$, where

$$L_{cGAN} = E_{x,y \sim p_{data}(x,y)}[logD(x,y)] + E_{x \sim p_{data}(x), z \sim p_z(z)}[log(1 - D(x, G(x,z)))] \tag{5}$$

The general network that is used in [5] can be simplified as shown in Figure 1

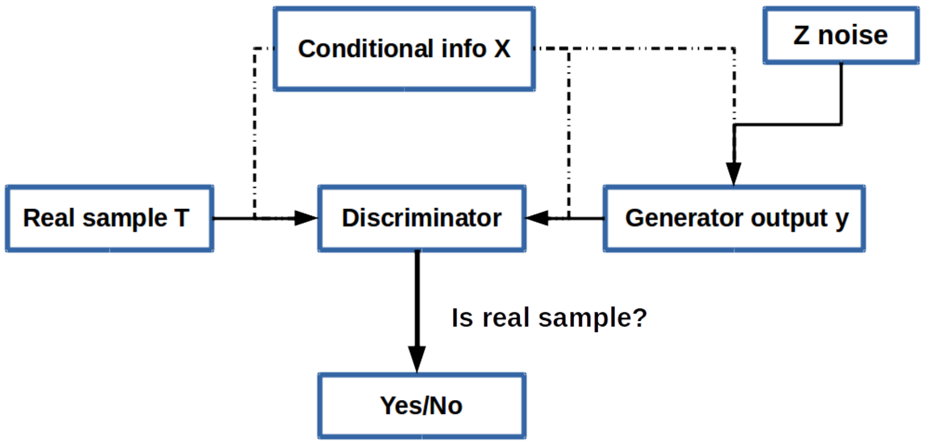


Figure 1: Generalized cGAN model

The loss functions we use for G and D are binary cross entropy. When adding L1 loss on top of G part of the cGAN model, we use mean absolute error as the loss function. These are consistent with [5].

### 3.2 Structure Visualization

The two networks both are consist of a general structure: convolution-BatchNorm-ReLu.

For the generator **G**, it has a tanh activation function in the output layer. We will use the U-Net model described in [13]. So layer $n - i$ will have a connection with layer $i$ as shown below. In the network, both of the input and output dimensions are $128 \times 128 \times 3$ scaled images with pixel values scaled to zero-mean between -1 to 1. The detailed model graph is shown in Appendix A and in the code section.

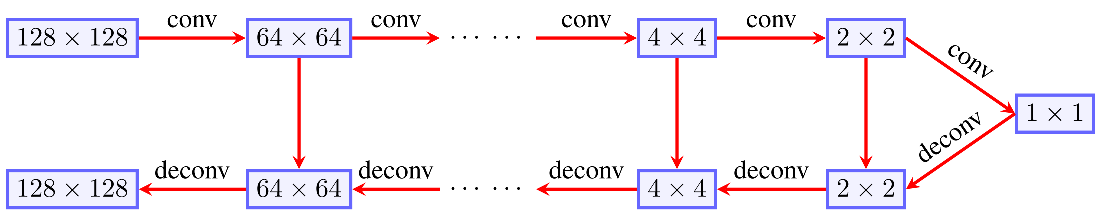


Figure 2: Structure of the generator network

For the discriminator, **D**, it will have a sigmoid activation function in the output layer. So the output would be the probability of $Y$ as a real target. The detailed model graph is shown in Appendix B and in the code section.

$$\begin{tikzcd}\boxed{128 \times 128} \arrow[r, "\text{conv}"] & \boxed{64 \times 64} \arrow[r, "\text{conv}"] & \cdots\ \cdots \arrow[r, "\text{conv}"] & \boxed{4 \times 4} \arrow[r, "\text{sigmoid}"] & \boxed{1 \times 1}\end{tikzcd}$$

Figure 3: Structure of the discriminator network

The entire cGAN model is shown in Appendix C.

### 3.3 Datasets

Two datasets will be used in this report. The summary is shown in Table 1. The CMP Facade Database [15] will be used to replicate three figures from the paper as well as for network understanding and stochasticity improvement. The new sets of MNIST dataset will be used to better understand and explore the visual translation capabilities of this network.

Table 1: Dataset summary.

| # Dataset Name | Task | Train No. | Test No. | Input Dimension |
|---|---|---|---|---|
| CMP Facades[15] | Architectural labels $\leftrightarrow$ photo | 378 | 228 | $256 \times 256 \times 3$ |
| MNIST | Hand-written $\leftrightarrow$ Thick font | 1000 | 200 | $128 \times 128 \times 3$ |
| MNIST | Hand-written $\leftrightarrow$ Thin font | 1000 | 200 | $128 \times 128 \times 3$ |
| MNIST | Hand-written $\leftrightarrow$ Chinese number | 1000 | 200 | $128 \times 128 \times 3$ |

#### 3.3.1 CMP Facades

The paper [5] used the Cityscapes dataset [1] for most of the main findings. However the datasets provided by the authors in [6] contain noise in their labeled images, which may be introduced by converting image format from png to jpg. This made it impossible for us to replicate the FCN-scores, hence we have decided to obtain and use images from the original source. However, the original Cityscapes dataset requires manual permission to be used for research and we were not able to obtain that in time. Hence, we eventually used Facade dataset instead. The Facade dataset is the only other dataset the paper worked on that is a task of transformation from labels to photos, which we could use to reproduce the FCN-scores.

After scaling, the pixel values are further preprocessed to be between -1 and 1.

The data is labeled with 12 classes including:

- facade
- molding
- cornice
- pillar
- window
- door
- sill
- blind
- balcony
- shop
- deco
- background

There are two sets of original data: CMP facade DB base (378 images), and CMP facade DB extended (228 images). The paper used 400 images for training and 100 images for validation and 106 images for testing. We have decided to use the 378 CMP facade DB base for training, and the 228 CMP facade DB extended for testing for easy mangement between 4 teammates.

An image sample from the facades dataset has been shown in 4. The image on the right is the photo of the city that we are trying to generate, the labeled segmented image on the left will be our input information.

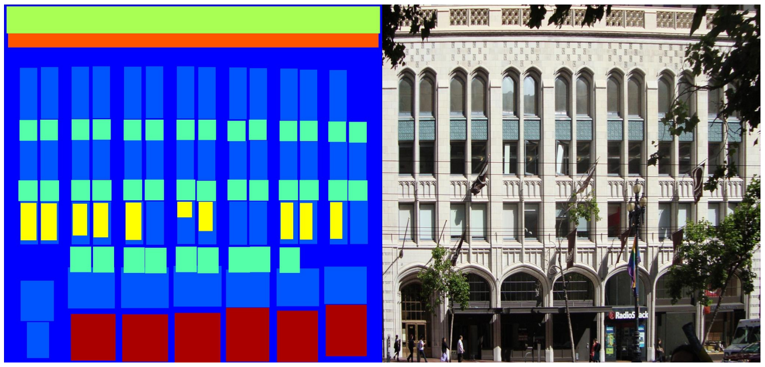


Figure 4: Sample image from Facades. The mapping we are trying to replicate here is from left to right.

### 3.3.2 New MNIST

We have used the original MNIST dataset as the input, and created four sets of outputs to test out the network's capability of visual font formatting and language translation. The input and target image samples are shown as follows in Figure 5.

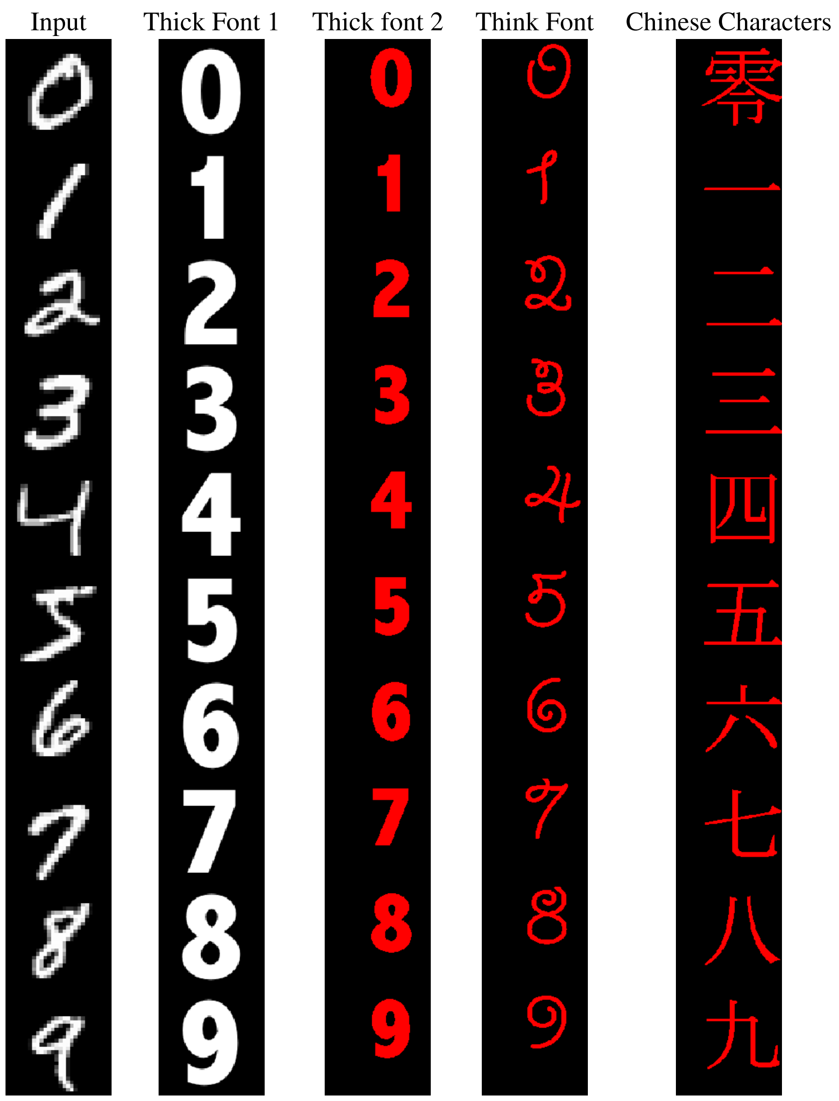


Figure 5: Datasets used for MNIST training. The inputs are the original MNIST hand-writings. The right three columns are four types of targets we are trying to acquire through the network on.

### 3.4 Software

The original code was written in Lua [11] with the Torch library [14], which is available publicly at [6]. We duplicated their structure in Keras [7] shown in the code section.

## 4 Evaluation Methods

### 4.1 For Model Replication

#### 4.1.1 Visual Appearance

We will generate real-looking images based on 228 labeled test Facade image across five loss functions. The visual appearances of blurriness, realistic features, details of the output among different models in Lua and Keras code will be compared.

### 4.1.2 FCN-score

FCN-score is proposed in Fully Convolutional Networks for Semantic Segmentation [9]. To calculate the FCN-score, we need to use a segmentation network which takes real image as input and label image as teaching signal. The structure of that network is shown as Figure 6.

We found that the upper part of FCN is exactly the same as VGG16 network, except two fully-connected layers fc6 and fc7 are replaced by fully convolutional network. It is interesting to notice that the weight retrieved from pretrained VGG16 network can be used here with proper network surgery. The lower part of FCN is especially designed for segmentation recognition. They use *Fuse* layers to merge output from two different layers and use *Upscore* layers to deconvolute the image. At the end, there is a *Softmax* layer to generate image label pixel-wisely.

There are three kinds of scores that we are curious about: per-pixel accuracy, per-class accuracy, and IoU(Intersection over Union) accuracy, which can be measured by taking the semantic segmentation of our predicted images and ground truth images to construct a confusion matrix. Suppose after segmentation, there are $n$ labels for each pixel, confusion matrix is a $n \times n$ matrix, where its $(i, j)$ element $C_{i,j}$ is the number of pixels whose true label is $i$ and predicted label is $j$. We denote by $G_i = \sum_{j=1}^{n} C_{i,j}$, the total number of pixels labelled with $i$, where n is the number of classes, and by $P_j = \sum_{i=1}^{n} C_{i,j}$, the total number of pixels whose prediction is $j$.

Per-pixel accuracy is the proportion of pixels whose label prediction is correct, so it's equal to the trace of confusion matrix divided by the sum of all entries in confusion matrix.

$$\text{per-pixel} = \frac{\sum_{i=1}^{n} C_{ii}}{\sum_{i=1}^{n} G_i} \tag{6}$$

Per-class accuracy measures the proportion of correctly labelled pixels for each class and then averages over the classes.

$$\text{per-class} = \frac{1}{N} \sum_{i=1}^{n} \frac{C_{ii}}{G_i} \tag{7}$$

IoU accuracy measures the intersection over the union of the labelled segments for each class and averages over the classes.

$$\text{IoU} = \frac{1}{N} \sum_{i=1}^{n} \frac{C_{ii}}{G_i + P_i - C_{ii}} \tag{8}$$

The detailed procedure of evaluating FCN-score can be summarized as follows.

1. Train on 378 image pairs (real image, label image) from *Based Facade Images* for 30 epochs.
2. Predict on 228 real images from *Extended Facade Images* to produce predicted label images. Then, evaluate the accuracy between predicted label images and corresponding label images as ground truth score.
3. Predict on 228 synthetic real images generated from different models to produce predicted label images. Then, evaluate the accuracy between predicted label images and corresponding label images as model score.

### 4.1.3 Color Distribution

The author mentioned that, L1 loss tended to be minimized by choosing the median conditional probability density function when it's uncertain which color it should take, so it will cause the image to have some average and grayish effects. On contrast, cGAN will distinguish the grayish color from real color, thus it will encourage the output to have as closest color distribution as the ground truth.

Images with high similarity should have similar color distribution. Thus, it is an intuitive way to measure the model performance. We specifically measure the distribution in CIELAB color space [4], which consists of *L*, *a*, and *b*.

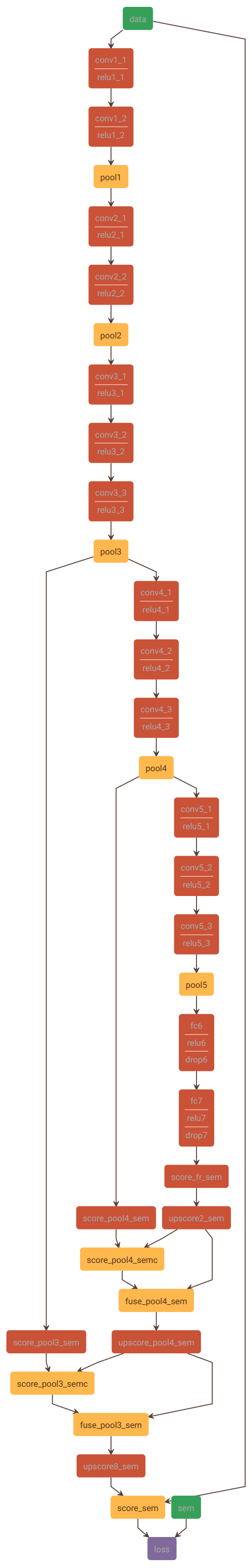


Figure 6: Fully Convolutional Network for Image Segmentation

### 4.2 For Model Understanding

For Image reconstruction and the output of the set of new datasets for MNIST transformations, we only evaluated them based on their visual appearance.

### 4.3 For Stochasticity

Stochasticity is hard to measure. We found no literature mentioning how to systematically and quantitatively evaluate how stochastic a GAN image output is so far. To evaluate the stochasticity, we need to feed the same image to the generator several times and get a series of generated images with same source. We came up with three methods of our own designs. The most simple way is by looking at generated images with same source to figure out their variations. This method could provide some intuition about the randomness of outputs. The two other methods are using histogram and PCA.

#### 4.3.1 Histogram

We plotted the histogram for pixel values of the generated images with same source on the same figure. This enabled us to get a distribution of pixel values for each generated images. By plotting them on the same histogram, we can get a clear view of the variation. In this experiment, we generated 5 images for the same input sample. For the pixel values, we combined all RGB channels and the values are in the range of $[0, 255]$ which are separated into 16 bins for the histogram.

#### 4.3.2 PCA

We used PCA to reduce the dimension of generated images from $256 \times 256 \times 3$ to 2 in order to visualize the distribution of these generated images in 2D space. By looking at the clustering of different models, we can see if the spread of the data has increased or not. If the spread is small, we think this represents the two images are closer to each other in the higher dimensions, thus not much stochasticity are included; if the spread is large, then the models have increased the stochasticity.

## 5 Experiment & Results

### 5.1 For Model Replication

To replicate the major accomplishment of this paper, we have covered the FCN-scores, the output images, and color distributions of [5] to verify our replication code. To make it comparable, we used the Facades dataset instead of the Cityscapes dataset as is shown from [5].

To be consistent wit the paper, we will run the same set of 5 types of loss functions.

- generator + L1 loss ('mean_absolute_error') alone
- GAN(Generative adversarial networks)
- cGAN
- GAN + l1
- cGAN + l1

#### 5.1.1 Visual Appearance

We use the Facades dataset with size of $256 \times 256 \times 3$ as inputs. The batch size is set to be 32 for faster process (original paper used 8), and the results are acquired after 200 epochs for both Lua and Keras code. All other hyper-parameters remain consistent with the original paper.

The generated images from the original Lua code for the Facades dataset are compared with images from the Keras implementation.

- The result of Figure 7 shows that it confirms the findings from the paper. The generator alone with L1 loss encourages averaging and produces a blurred grayish result, whereas using a cGAN pushes the output distribution closer to the ground truth.

- The result of Figure 8 shows what we have acquired from the our Keras model. The resuls are not what we have expected, it seems like with L1 turned on in our model, all results look similar with or without the discriminator part of the network, whereas by setting L1 loss weight to 0 in the GAN model, we get almost no feedback information from the target images.

This difference indicates that the way we are routing our generator and discriminator together in the GAN model does not seem to be working as expected.

1. First, We have generated 228 images from the test set use Lua code shown here in Figure 7 after 200 epoch.

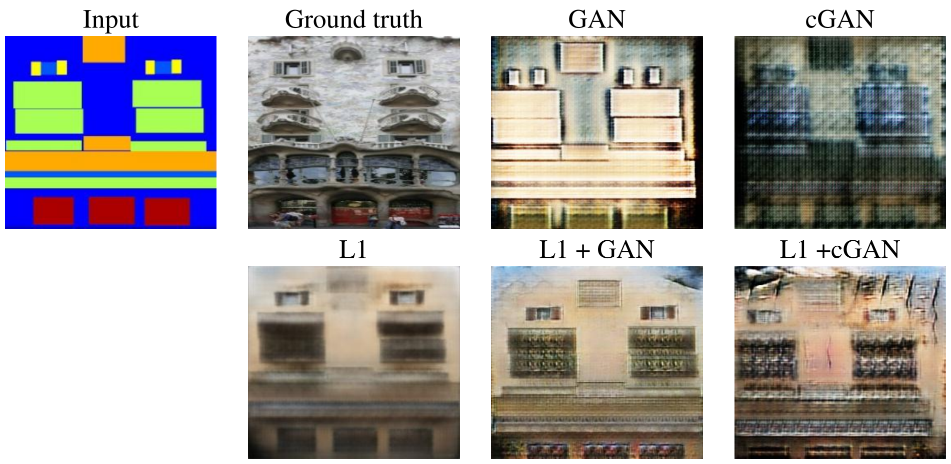


Figure 7: Test result for different loss functions from their Lua model. Each column shows results trained under a different loss. The models used were trained over 200 epochs

2. Then, we have acquired the same sets of results for our Keras code in Figure 8.

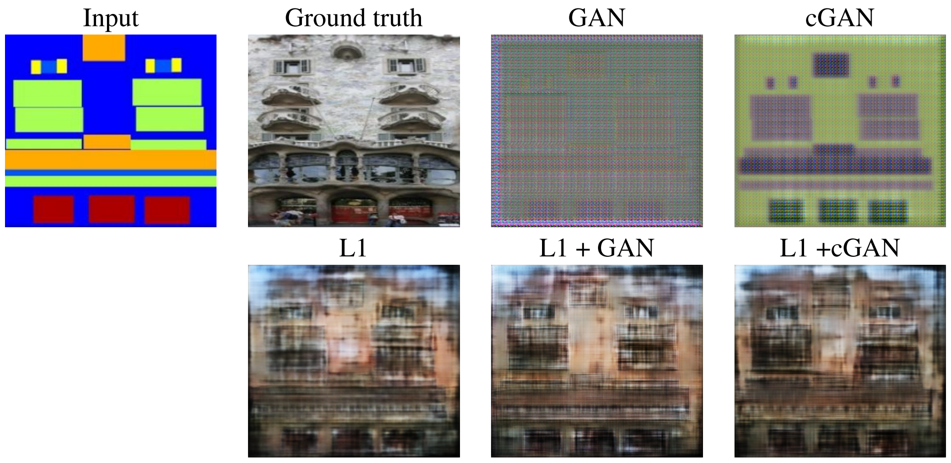


Figure 8: Test result for different loss functions from our Keras model. Each column shows results trained under a different loss. The models used were trained over 200 epochs

### 5.1.2 FCN-score

Here, we compare the results between 5 models — **L1**, **GAN**, **cGAN**, **L1+GAN**, **L1+cGAN** implemented in both lua and Keras, as recorded in Table 2 and 3, respectively.

| Loss | Per-pixel acc. | Per-class acc. | Class IOU |
|---|---|---|---|
| **L1** | 0.42 | 0.14 | 0.08 |
| **GAN** | 0.36 | 0.13 | 0.07 |
| **cGAN** | 0.35 | 0.11 | 0.07 |
| **L1+GAN** | **0.43** | **0.18** | **0.11** |
| **L1+cGAN** | **0.43** | **0.18** | **0.11** |
| Ground Truth | 0.52 | 0.32 | 0.22 |

Table 2: FCN-score For Facade (modeled in lua)

| Loss | Per-pixel acc. | Per-class acc. | Class IOU |
|---|---|---|---|
| **L1** | 0.47 | 0.24 | 0.15 |
| **GAN** | 0.25 | 0.13 | 0.06 |
| **cGAN** | 0.39 | 0.14 | 0.07 |
| **L1+GAN** | **0.47** | **0.27** | **0.16** |
| **L1+cGAN** | 0.43 | 0.26 | 0.14 |
| Ground Truth | 0.52 | 0.32 | 0.22 |

Table 3: FCN-score For Facade (modeled in keras)

### 5.1.3 Color Distribution

We ran the algorithm in Lua and Keras and put experiments on 5 kinds of models: generator + L1 alone, cGAN, GAN, GAN + L1 and cGAN + L1. Figures 9 and 10 show the color distribution of lua and Keras respectively. In the figures, we compare the color distribution of generator + L1 alone, cGAN, GAN, GAN + L1 and cGAN + L1 to see if our model's outputs are close to ground truth. We can see that L1 leads to narrower color distribution and cGAN (GAN) has the opposite effect: it makes the outputs more colorful.

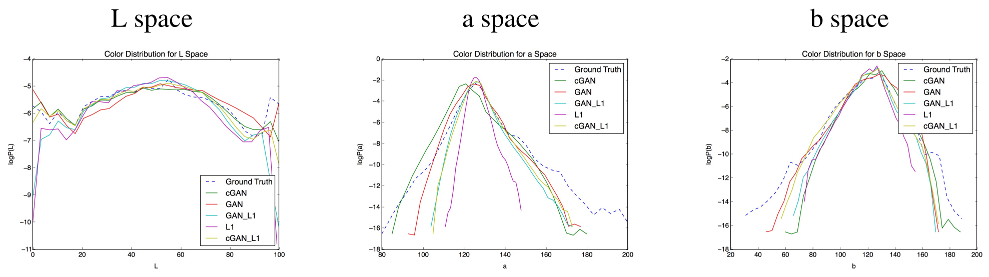


Figure 9: Lab color distribution for lua

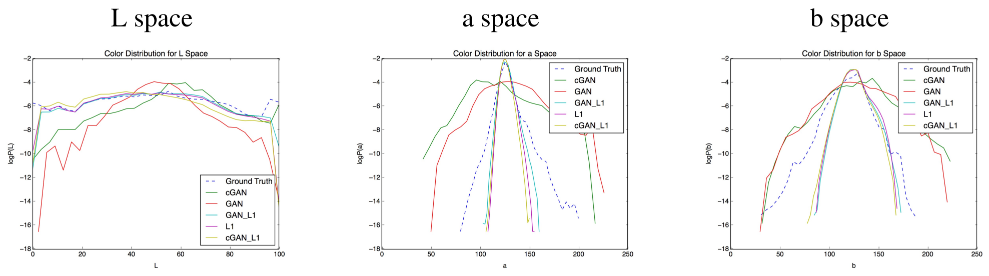


Figure 10: Lab color distribution for Keras

### 5.2 For Model Understanding

We want to try to understand how this model works. Therefore, we acquired some image-reconstruction techniques from [2] to visualize the internal layers. Additionally, we designed a couple new sets of MNIST transformations to further test out the capabilities of this network.

#### 5.2.1 Reconstruction

To get a better understanding of how the networks work, we reconstructed the images at different layers through the generator model to visualize the results at internal layers.

Here, after the model is trained, the reconstruction is defined as finding the image that will produce close activation at the layer as the sample. Say we want to reconstruct at layer 5. We first chose one sample image and fed it to the trained network to get the activation at layer 5. This activation was used as the target. Then we initialized a white noise image and forwarded it to layer 5. The mean-squared-error was calculated compared to the target. Then we fixed the weights of the networks and back-propagated the error to update the noise image.After a number of iterations, we gained the reconstructed image at that layer.

We reconstructed at layer 1, 5, 8, 11 and 14 for both label to image and image to label model trained on the facades dataset. Figure 11 and Figure 12 shows some of the reconstructed images.

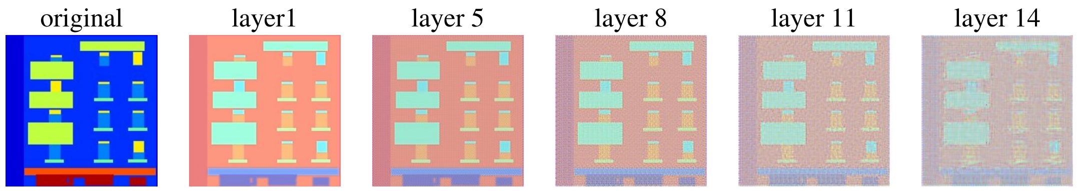


Figure 11: Reconstruction results from different layers for label as input

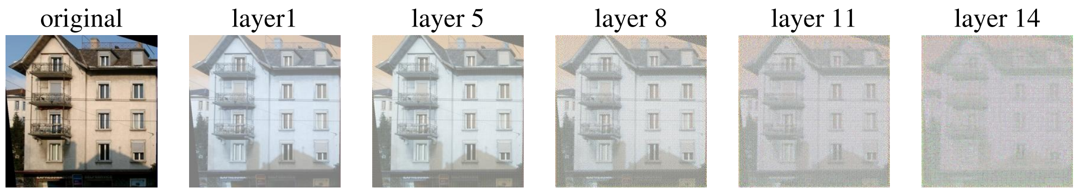


Figure 12: Reconstruction results from different layers for image as ainput

#### 5.2.2 New MNIST dataset

The reason why we choose to transform MNIST dataset is because is it easily available, and it contains minimal features which normally requires less training time. Traditionally, MNIST dataset is often used for classification evaluations, here we use it for an image-to-image task to open up new opportunities for this network. If this transformation works seamlessly, it can potentially be used to transform bad information to better information to better assist other complicated machine learning tasks.

We decided to transform the MNIST dataset into two different type of fonts because we wanted to test two types of transformation, one is thickening the digits to fill a space near the lines since the task is to map the thin hand-written digits to a thicker font. The other one is to re-configure the image since it is mapping to a thinner font. Then we decided to tune up the difficulty level so that the mapping becomes similar to digits translation between languages. For Think Target 1 and 2, we are comparing the transformation of structure with or without color a the same time.

We have run the network on the new MNIST datasets and acquired the following images after 100 epoch in Figure 13. It seems like the model performed really well on all datasets.

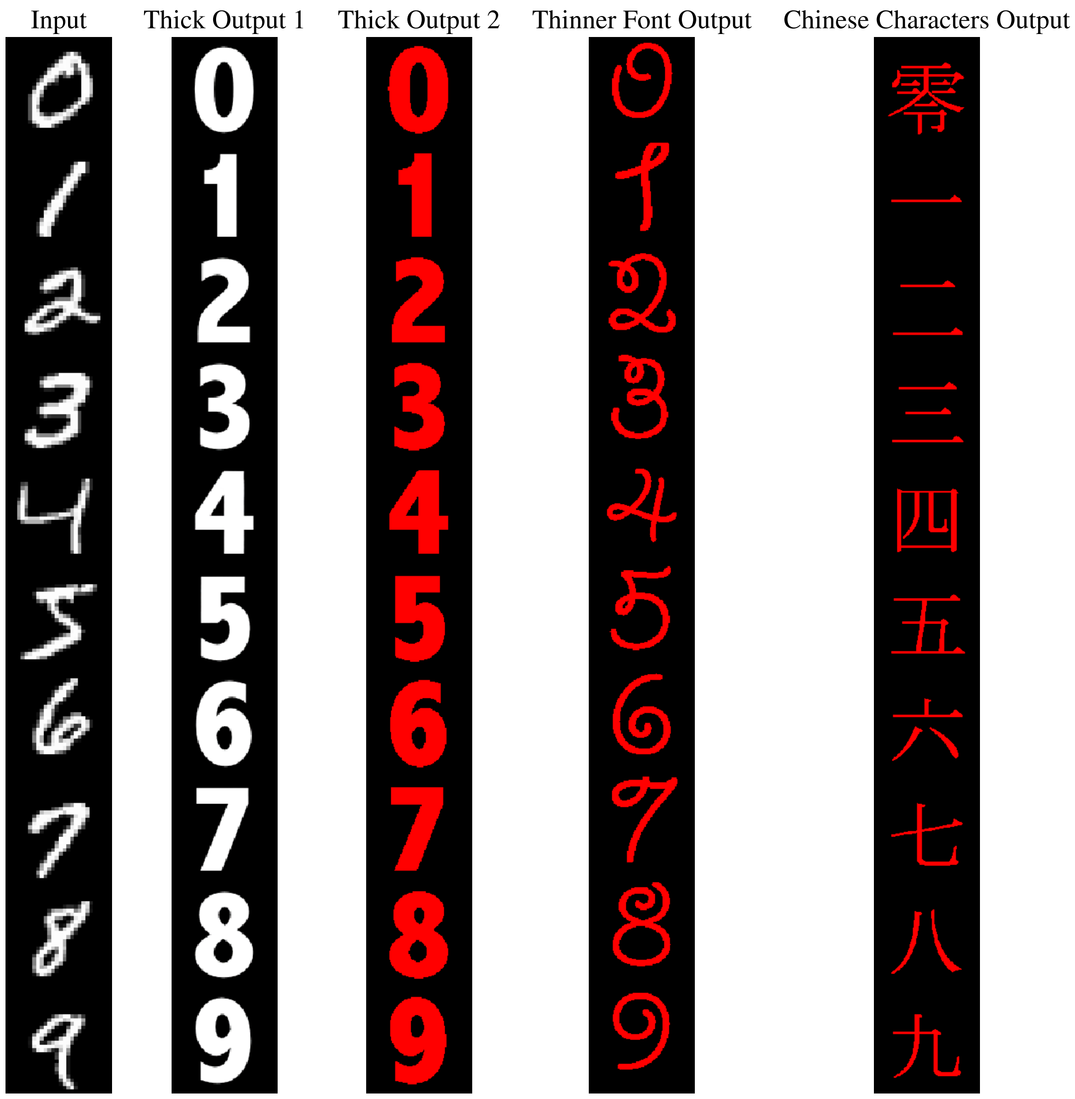


Figure 13: Datasets used for MNIST training. The inputs are the original MNIST hand-writings. The right four columns are four types of outputs we have generated after 100 epoch.

### 5.3 For Stochasticity

This task was mentioned in the paper, and it was a general problem in the field of generative models. Therefore, we have decided to explore it further and see if we can make improvement. For improving the stochasticity, we made the following experiments.

- [5] mentioned that their random noise is introduced in the form of dropout layers. Therefore, in order to increase the stochasticity, we wanted to first increase the dropout rate further to see if that helps with the performance. The dropout rates are: no dropout, 0.5, 0.8.
- Instead of dropout layer which will zero out inputs, we randomize the inputs based on certain distribution to increase randomness. The baseline is a uniform distribution ranged from [-1, 1] of the original value. The randomize rate is set to 0.5.
- Additionally, we used a normal distribution with adjustable variance to do randomization. We tried with mean 0 and standard deviation of 0.2 and 0.4.

The results for stochasticity are shown as follows:

### 5.3.1 Generated Images

For the input image and target image Figure 14, we generated 5 images with each of the models mentioned above. The results are shown in Figure 15.

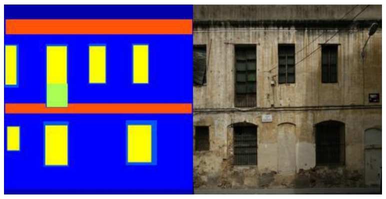


Figure 14: Input label and target image for testing stochasticity

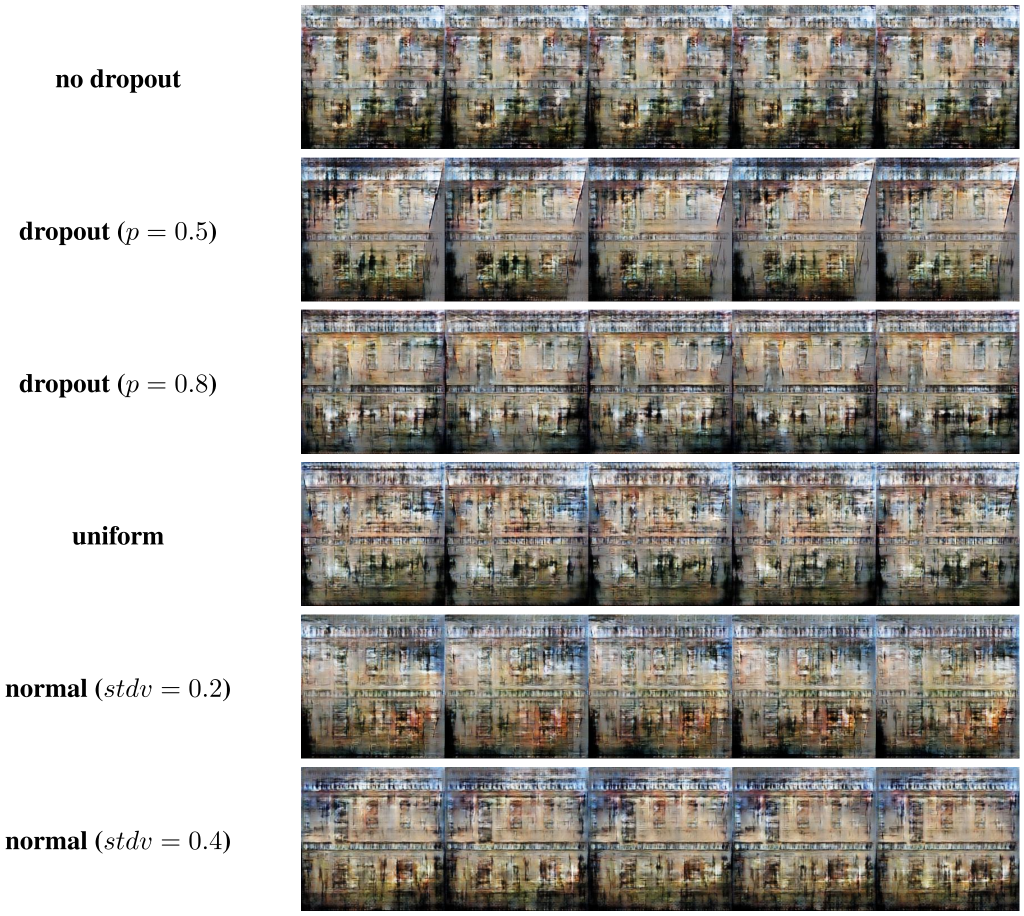


Figure 15: Generated images from the same source with different models for testing stochasticity

The quality of the generated images may look a little poor. The reason is limited time and resources so that we don't have enough time to train the models to generate good looking results. Given more time, the quality of the generated images could increase. The quality could be compared with a relative criterion. The dropout ($p = 5$) model is the default model used in [5]. So all models generate comparable quality images with the original model. In this part, we will focus more on the stochasticity.

As we can see, no dropout model generated 5 same images. All other models generate images with some extent of variations.

### 5.3.2 Histogram

The reason why we chose Histogram over pixel information is because it is fairly easy way to visualize if there is any difference in the images we are looking at. Admittedly, this method loses a lot of information in high dimensional space. Hence we have decided to try dimensionality reduction as our next experiment on this topic.

As mentioned above, we plotted the histogram of pixel values for the 5 images generated from the same source. The results are shown as follow.

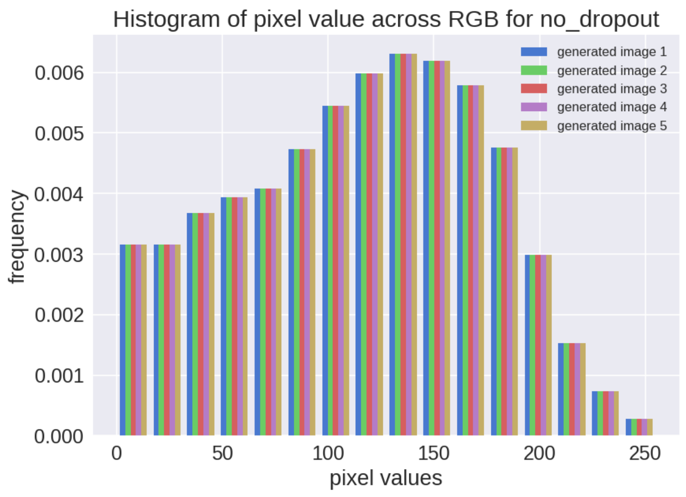


Figure 16: Histogram of pixel values for 5 generated images for no dropout model

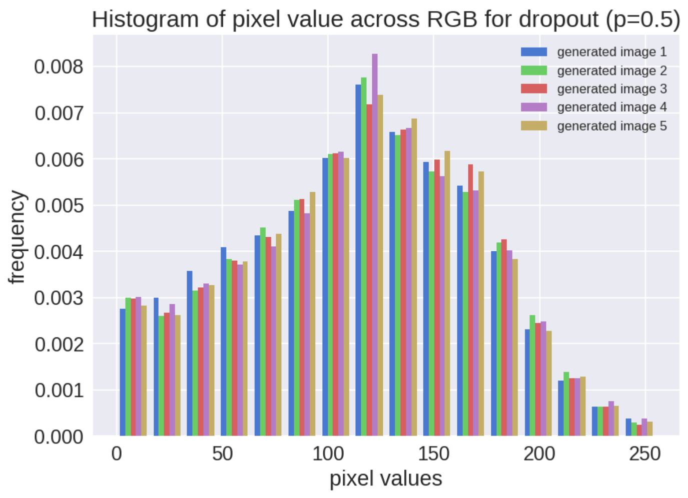


Figure 17: Histogram of pixel values for 5 generated images for dropout ($p = 0.5$) model

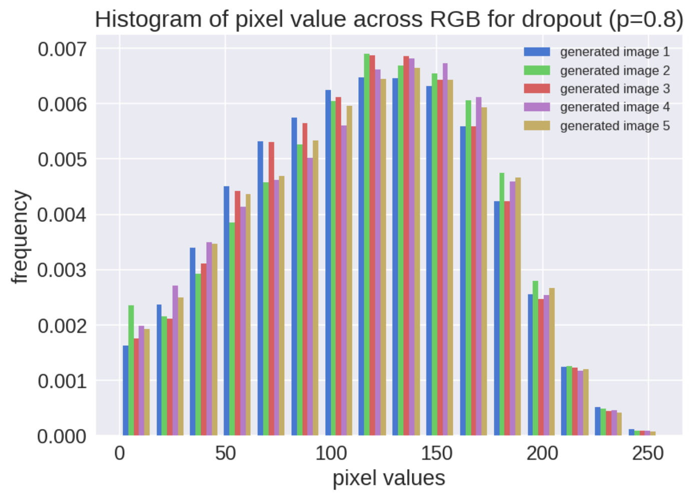


Figure 18: Histogram of pixel values for 5 generated images for dropout ($p = 0.8$) model

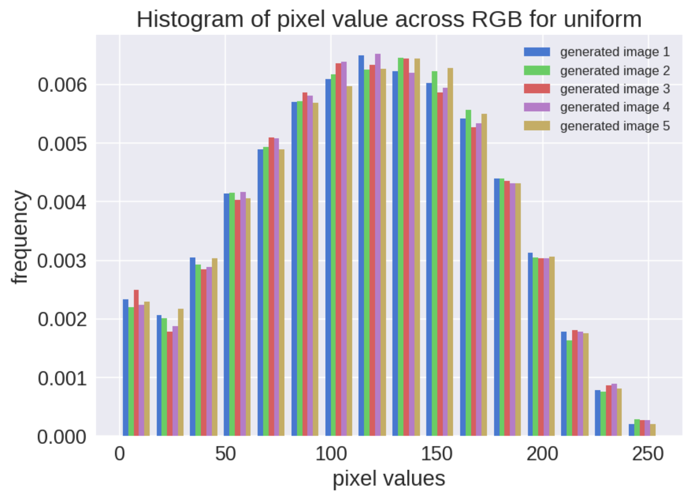


Figure 19: Histogram of pixel values for 5 generated images for uniform model

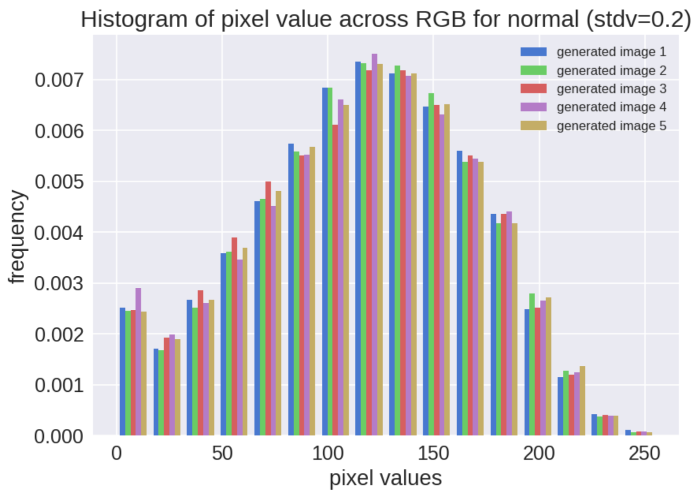


Figure 20: Histogram of pixel values for 5 generated images for normal ($stdv = 0.2$) model

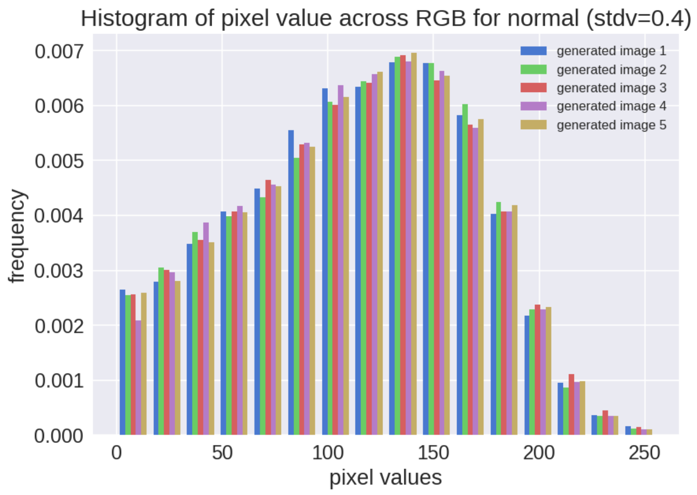


Figure 21: Histogram of pixel values for 5 generated images for normal ($stdv = 0.4$) model

### 5.3.3 PCA

Because our previous Histogram result did not show much difference between different models, and we are looking at the variance between images, PCA as a dimension-reduction tool that maximizes variance is our first choice for dimension reduction. It helps us to project information onto a 2D space.

In this experiment, we generated 30 images for each model with the same original image. They are then dimension-reduced to 2-dim points. The results are shown in Figure 22.

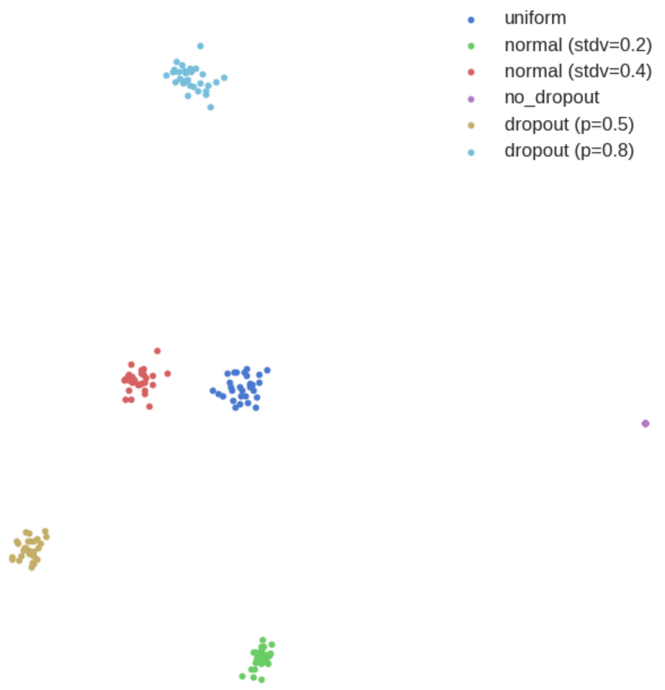


Figure 22: Generated images plot in 2-dimensional space

## 6 Discussion

### 6.1 For Model Replication

#### 6.1.1 Visual appearance

We have duplicated the results for the Facades dataset using both the original Lua code and our Keras model. Here are some thoughts on why our code did not perform as well as the Lua code.

First of all, we notice that in the original Lua code, the backpropagation from the discriminator to generator is truncated, so that the derivatives for the conditional labeled image will not be combined with the derivatives for the target image. We concated our result the same way the paper did, but due to the nature of Keras, we did not have enough time to look at the Keras backend and to make sure the same derivaties have been routed the same way. This may be contributing to the reason why there appears no target information in the output of our GAN and cGAN models.

Secondly, we have turned the weight for the generator loss to be 0(no L1 loss) or 100(with L1 loss) in the final GAN model, which is similar to what they did in the paper. However, for our GAN and cGAN model it seems like there is no target information being use for the adjustment, whereas there seem to be no difference among L1, L1+GAN and L1+cGAN models. This indicates the Keras weight system may behave differently than what we have expected. Additionally, they have added their 100 weight onto the descriminator part of the derivative for the generator, whereas the weight we assigned might have been added at the final loss of the discriminator. This explains why the L1 effect might be over-powering through for GAN and cGAN models as well.

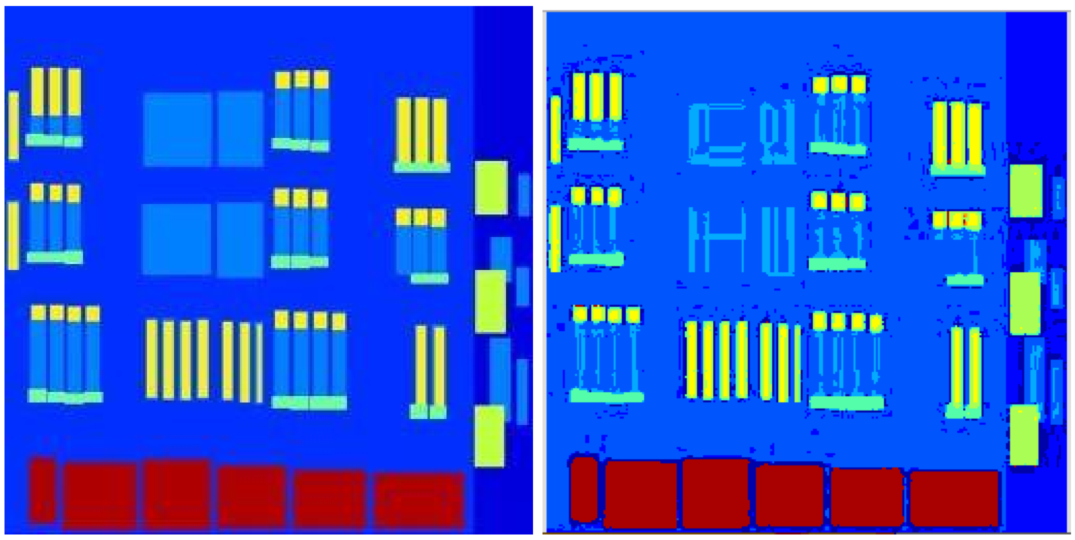
In summary, to fix or verify these hypothesis for potential bugs in the Keras code, we need to explore the backend of Keras further to make sure everything is passed through as what we expected, and we add weights at the location that we wanted for back-propagation. Unfortunately, we did not have that time to explore further.

#### 6.1.2 FCN-score

For cGAN and some of its variances, they regard label images as input and real images as teaching signals while training. In the testing phase, the generative model will generate some synthetic real images as output, whose performance will then be evaluated by FCN model. On the other hand, FCN model takes real images as input and label images as teaching signals in training phase and evaluate synthetic real images by segmenting those real images. If the segmented real images have high similarity to origin label images, then we can conclude that cGAN model does generate some reliable synthetic real images.

This framework seems robust and reasonable to us in the beginning. However, we find that the labeled images in the given dataset contain too much noise and not well-formed, which makes FCN model impossible to learn and evaluate the cGAN model performance.

In view of this, we try to use a universal way to denoise these noisy label images, which are stored in $RGB$ mode with 3 channels, and map them back to the correct label space, which consists of only 1 channel. It is noticeable that the denoise procedure must be consistent for all images, which kept us from using Gaussian or other filters to smooth the image; otherwise, we will still have trouble in mapping images back to label space. At the end, we try to do the direct mapping by picking the closest label with the smallest Euclidean distance error. Unfortunately, the image is still too noisy for us to recover. One of the noisy label images and its corresponding recovered label image are shown in Figure 23a and Figure 23b.



(a) Noisy Label Image (b) Recovered Label Image

Figure 23: Trial of Recovering Image with Smallest Euclidean Distance

With the failure to recover label image, we at the end need to download the data from official websites and do all image preprocessing works, i.e. image resizing to (256 $\times$ 256 pixels) by ourselves.

From the results, one can tell that L1 regularization plays a very important role here. Without L1 term, the accuracy for both **GAN** and **cGAN** model drops significantly. Combined the scores from both tables, we can see the general model performance trend as: **L1+GAN** $\geq$ **L1+cGAN** $>$ **L1** $>$ **cGAN** $\geq$ **GAN**, which is aligned to the results from previous paper.

When compared the results generated from lua and Keras, we can see a clear trend that Keras generated images have higher scores than lua ones, which is then straightforward for us to infer Keras models have better performance. However, when visualizing the image, we find out an interesting phenomenon: image generated from Keras are more similar to label, which might greatly enhance its FCN-score. It is then crucial for us to also take the results of color distribution into consideration. More details about the difference between Keras and lua models will be discussed in the following sections.

#### 6.1.3 Color Distribution

We can see from Figures 9 and 10 that L1 has the effect to narrow the spectral distribution of output images when it's uncertain which value to take when there are several values, so it will choose the median and make the network simpler.

Interestingly, we can see from the a and b color space distribution that cGAN leads to wider color distribution than ground truth, which means cGAN will make the output images more colorful, even though it doesn't exist in the original images.

Combined the results of color distribution with the results of FCN-scores, we can conclude that, because L1 regularization has the effect to narrow the spectral distribution and cGAN (GAN) will make the image more colorful, we need to combine cGAN or GAN with L1 regularization to get better performance.

Compare the result of color distribution of lua and Keras, we can see that for a and b color space, Keras produces wider spectral properties for GAN and cGAN but on the other hand narrower spectral properties for L1, cGAN+L1 and GAN+L1. So GAN and cGAN of Keras produces more colorful and distinguishable outputs than lua, Keras has lower FCN-scores for GAN and cGAN. While L1, cGAN+L1 and GAN+L1 for Keras produces narrower and closer color distribution to ground truth, thus L1, cGAN+L1 and GAN+L1 for Keras have higher scores than lua.

### 6.2 For Model Understanding

#### 6.2.1 Reconstruction

Reconstruction provided us with a way to visualize the internal representations of the networks. From the reconstruction results we found that as the network goes deeper, the reconstructed image becomes more randomized. As discussed in the model part, we are using an encoder-decoder model with u-net structure. So as the encoder goes deeper, the network learns its own internal representation of the image. The u-net connection from higher layer to lower layer helps the network to keep a good representation of the position and structure of the original image for generating the output image.

The reconstruction result is consistent with the reconstruction result of real images in this paper [2]. This verifies that the type of transformation: label to photos is a much more complicated transformation than artistic styles. We cannot simply visualize the networks by reconstructing the image from different convolutional layers. Because the transformation is in a global scale, we were not able to visualize simple local transformations.

#### 6.2.2 MNIST Dataset

The MNIST results in 13 shows that all 4 scales of transformations have been accomplished by the network with minimal effort. This verifies that the network is great at performing multiple-to-one transformation. Due to lack of stochasticity, we took advantage of the deterministic nature of the network, and successfully transformed the given hand-written digits to different fonts(fattening or slimming) as well as characters with a complete different shape (Chinese characters) in a different color. This also verifies that the transformation is not one-to-one, and it is really versatile. At the beginning of the network we can observe that after the first epoch with 32 images per batch as shown in 24, the output is mostly capturing the input; however, after 7 epochs as shown in 25, we can see that the network is trying to head to the direction of translating both of the color and shape towards the target. After roughly 10 epoch or so, the majority of the test output is already able to capture the target pretty accurately.

Additionally, it doesn't seem like the network behaves differently among the three tasks we have designed here. We think this may indicate that multiple-to-one mapping is too simple a task for the network. The difficulty level of the image does not really matter for this type of task, especially when the number of colors in the images are very limited.

These results are encouraging and provide evidence that this network can be potentially used for data preprocessing for other more complicated machine learning problems.

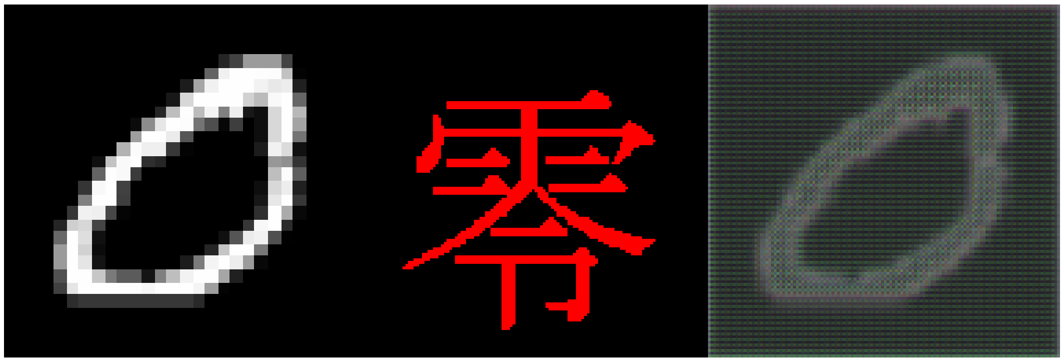


Figure 24: Images from first epoch of samples in batch of 32, and size of $128 \times 128 \times 3$. The order of the images are from left to right: Input, Target, Output.

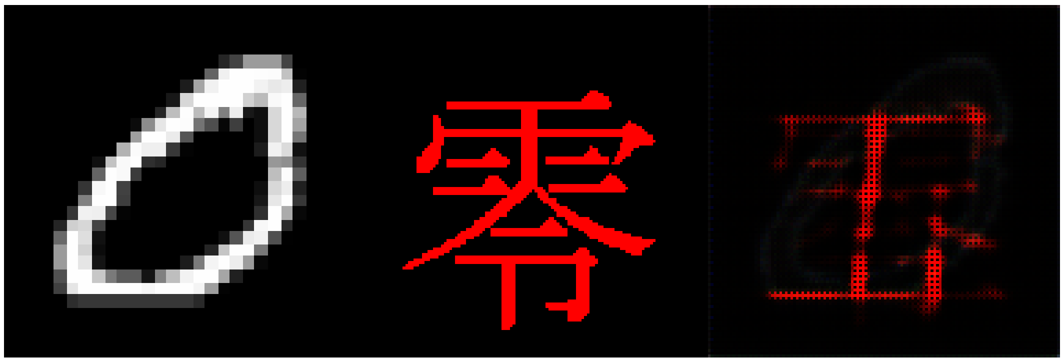


Figure 25: Images from 7th epoch of samples in batch of 32, and size of $128 \times 128 \times 3$. The order of the images are from left to right: Input, Target, Output.

The MNIST dataset is a very small dataset, which is much easier for the network to capture the transformation of this task; however this proves the potential of the network in the field of many-to-one mapping. This specific experiment proves that this network can be further used for language visual translation with much more complicated datasets.

## 6.3 For Stochasticity

### 6.3.1 Histogram

From the histograms, we get a general information about the pixel value distribution of the generated images. However, we don't see a significant difference in the extent of variations for all models except the no dropout one. So, we conclude that histogram of pixel values may not be a proper evaluation for stochasticity of the model.

### 6.3.2 PCA

As we can see from Fig 22, the images generated by the no dropout model all mapped into one point. We ranked them in spreading size: dropout ($p = 0.8$) $\geq$ uniform $\geq$ normal ($stdv = 0.4$) $>$ dropout ($p = 0.5$) $\geq$ normal ($stdv = 0.2$) $>$ no dropout. This reflects the stochasticity of the models to some extent. Combined this result with the visual results, we found that the randomizing layer we designed helps to increase the stochasticity of the model. Different distribution models with different parameters could lead to different results. The detailed difference needs more experiments and studies to discover.

## 7 Concept and/or Innovation

1. We have attached image visualization pipeline to the original model to visualize what is underneath the hood of this universal image-to-image transformation network.

2. We have applied the network on a new MNIST dataset to evaluate multiple-image-to-one capability of this model, and proved that it is truly powerful, and it has much more potentials in foreign language translations, etc. This result has shown that the network can be potentially used to translate bad data input (noisy information) into better images. Hence it can potentially help other tasks in the field of Machine Learning.

3. Additionally, we did implement the model in Keras, and improved the stochasticity with a newly innovated layer added to the model.

## 8 Summary

In this report, we explored the capabilities of cGAN network in the image-to-image translation problems and compared the performance of different loss functions: L1, GAN, cGAN, GAN+L1, cGAN+L1. We used several different methods to evaluate the performance of the network: visual appearance, FCN-scores, color distribution. From the FCN-scores, the L1+GAN model generated best results. To get more understanding of what the network is doing, we reconstruct the activations in different layers. We also used the cGAN network to do some multiple image to one image tasks on some newly created MNIST transformation datasets. To increase the stochasticity, we have tried different dropout rate, add randomized layers to the models. To evaluate the stochasticity, we compared the outputs visually, drew histograms and used PCA to visualize the randomness of the generated images. The randomized layer increased the stochasticity of the model.

## 9 Individual Contributions

Hao-en and Haifeng are in charge of FCN-score and Color Distribution measurement. Shiwei and Sainan are in charge of duplicating Keras code and test out the stochasticity improvement.

## Acknowledgments

We would like to thank Amazon for the resources and the Professor and the teaching staff's help throughout our project.

## References

[1] Cordts,M., Omran,M., Ramos, S.,Rehfeld, T.,Enzweiler,M., Benenson,R. ,Franke, U., Roth, S. & Schiele, B. (2016). *The cityscapes dataset for semantic urban scene understanding.* CVPR

[2] Gatys, L.A., Ecker, A.S., Bethge, M. (2015). *A Neural Algorithm of Artistic Style*.

[3] Goodfellow, I. J., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., Bengio, Y. (2014). *Generative Adversarial Networks*

[4] Hoffmann, G. (2013). *CIELab Color Space*
`http://docs-hoffmann.de/cielab03022003.pdf`

[5] Isola, P., Zhu, J., Zhou, T. & Efros, A.A., (2016). *Image-to-Image Translation with Conditional Adversarial Networks*.
`https://arxiv.org/pdf/1611.07004v1.pdf`
`https://github.com/shelhamer/fcn.berkeleyvision.org`

[6] Isola, P. Zhu, J., Zhou, T. & Efros, A.A. (2016). *pix2pix datasets*,
`http://people.eecs.berkeley.edu/ tinghuiz/projects/pix2pix/datasets/`

[7] Keras, Deep learning library for Theano and Tensorflow
`https://Keras.io/`

[8] Laurens, M. & Hinton, G. (2008) *Visualizing Data using t-SNE*

[9] Long, J, Shelhamer, E. & Darrell, T. (2015) *Fully convolutional networks for semantic segmentation." Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*CVPR, pages 3431 3440.
`https://github.com/shelhamer/fcn.berkeleyvision.org`

[10] LeCun, Y., Cortes, C., Burges, C. J.C. (accessed on 2017) *The MNIST database of handwritten digits*

[11] Lua, an embeddable scripting language.
`http://www.lua.org/`

[12] Mathieu,M., Couprie, C.,& LeCun, Y. (2016) *Deep multi-scale video prediction beyond mean square error* ICLR

[13] Ronneberger,O., Fischer, P. & Brox, T. (2015)*U-net: Convolutional networks for biomedical image segmentation.* MICCAI, pages 234241. Springer

[14] Torch, a scientific computing framework for LUAJIT
`http://torch.ch/`

[15] Tylecek, R., Sara, R. .(2013) *Spatial Pattern Templates for Recognition of Objects with Regular Structure.* CVPR

[16] Wang, X. & Gupta,A.(2016) *Generative image modeling using style and structure adversarial networks* ECCV

[17] Denton, E.L., Chintala, S. Fergus, et al. (2015)*Deep generative image models using a laplacian pyramid of adversarial networks.* NIPS

[18] Radford, A. Metz, L. and Chintala S. (2015) *Unsupervised representation leraning with deep convolutional generative adversarial networks.*

[19] A. Hertzmann, C. E. Jacobs, N. Oliver, B. Curless, and D. H. Salesin. (2001) *Image analogies.* In SIGGRAPH, pages 327340. ACM.

[20] Y. Shih, S. Paris, F. Durand, and W. T. Freeman. (2013) *Data-driven hallucination of different times of day from a single outdoor photo.* ACM Transactions on Graphics (TOG), 32(6):200.

[21] M. Mirza and S. Osindero. (2014) *Conditional generative adversarial nets.*

[22] M. Mathieu, C. Couprie, and Y. LeCun. (2016) *Deep multi-scale video prediction beyond mean square error.* ICLR.

# Appendices

## A Generator Model Detail

Model graph for Generator:

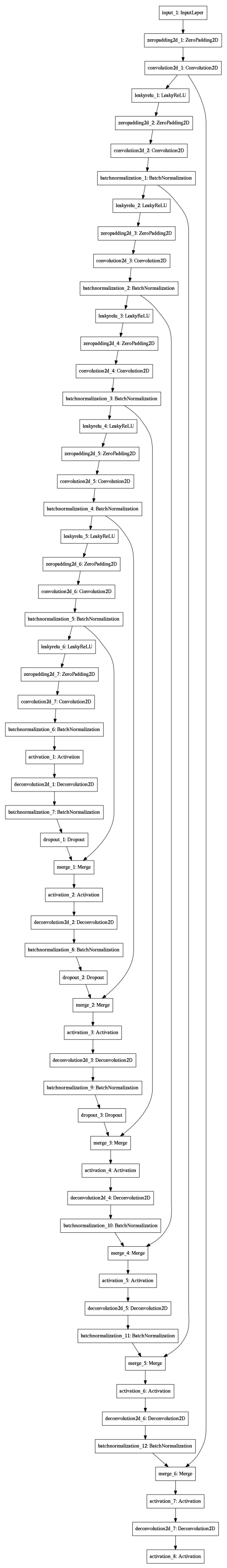

input_1: InputLayer
zeropadding2d_1: ZeroPadding2D
convolution2d_1: Convolution2D
leakyrelu_1: LeakyReLU
zeropadding2d_2: ZeroPadding2D
convolution2d_2: Convolution2D
batchnormalization_1: BatchNormalization
leakyrelu_2: LeakyReLU
zeropadding2d_3: ZeroPadding2D
convolution2d_3: Convolution2D
batchnormalization_2: BatchNormalization
leakyrelu_3: LeakyReLU
zeropadding2d_4: ZeroPadding2D
convolution2d_4: Convolution2D
batchnormalization_3: BatchNormalization
leakyrelu_4: LeakyReLU
zeropadding2d_5: ZeroPadding2D
convolution2d_5: Convolution2D
batchnormalization_4: BatchNormalization
leakyrelu_5: LeakyReLU
zeropadding2d_6: ZeroPadding2D
convolution2d_6: Convolution2D
batchnormalization_5: BatchNormalization
leakyrelu_6: LeakyReLU
zeropadding2d_7: ZeroPadding2D
convolution2d_7: Convolution2D
batchnormalization_6: BatchNormalization
activation_1: Activation
deconvolution2d_1: Deconvolution2D
batchnormalization_7: BatchNormalization
dropout_1: Dropout
merge_1: Merge
activation_2: Activation
deconvolution2d_2: Deconvolution2D
batchnormalization_8: BatchNormalization
dropout_2: Dropout
merge_2: Merge
activation_3: Activation
deconvolution2d_3: Deconvolution2D
batchnormalization_9: BatchNormalization
dropout_3: Dropout
merge_3: Merge
activation_4: Activation
deconvolution2d_4: Deconvolution2D
batchnormalization_10: BatchNormalization
merge_4: Merge
activation_5: Activation
deconvolution2d_5: Deconvolution2D
batchnormalization_11: BatchNormalization
merge_5: Merge
activation_6: Activation
deconvolution2d_6: Deconvolution2D
batchnormalization_12: BatchNormalization
merge_6: Merge
activation_7: Activation
deconvolution2d_7: Deconvolution2D
activation_8: Activation

Figure 26: Keras model for generator **G** - model1

## B Discriminator Detail

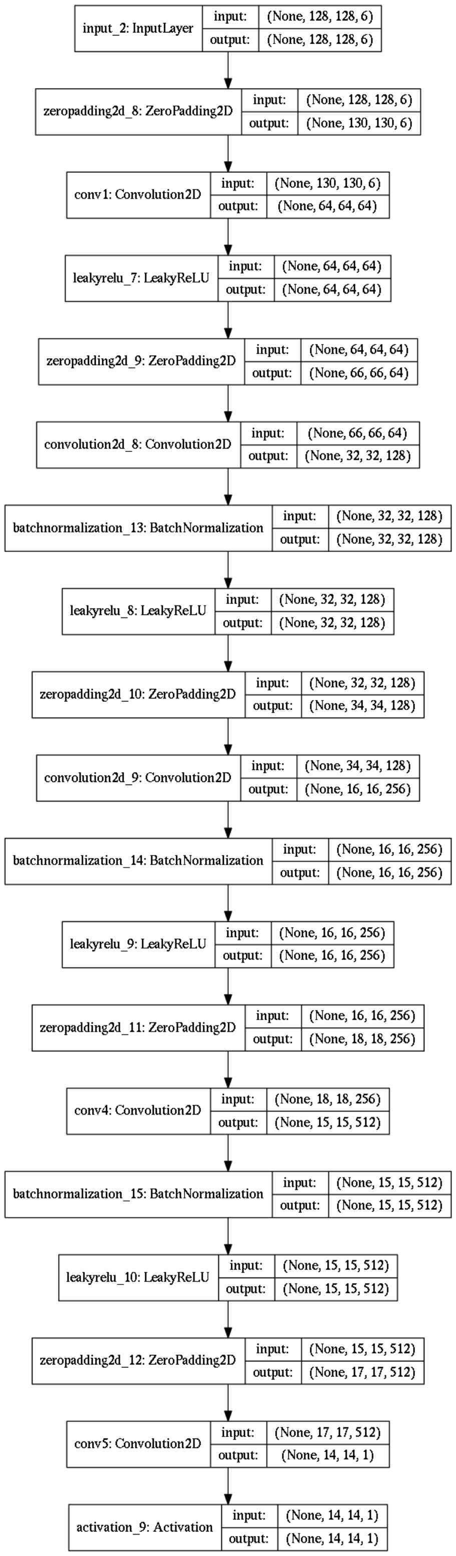
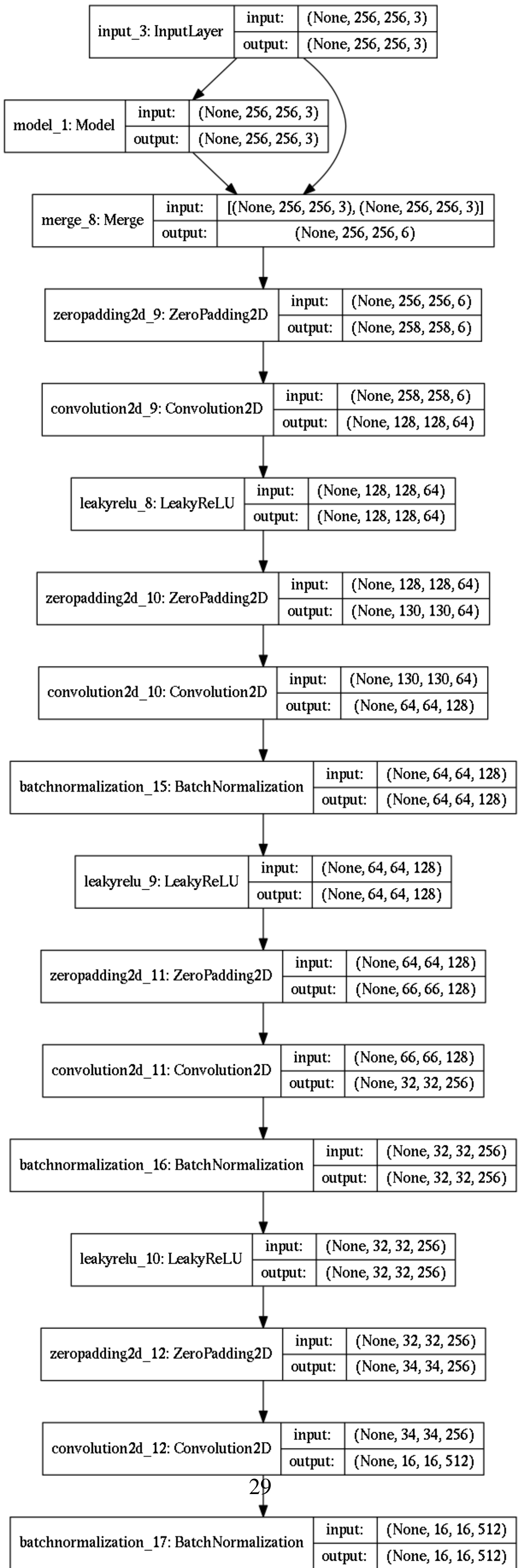

input_2: InputLayer
input: (None, 128, 128, 6)
output: (None, 128, 128, 6)
zeropadding2d_8: ZeroPadding2D
input: (None, 128, 128, 6)
output: (None, 130, 130, 6)
conv1: Convolution2D
input: (None, 130, 130, 6)
output: (None, 64, 64, 64)
leakyrelu_7: LeakyReLU
input: (None, 64, 64, 64)
output: (None, 64, 64, 64)
zeropadding2d_9: ZeroPadding2D
input: (None, 64, 64, 64)
output: (None, 66, 66, 64)
convolution2d_8: Convolution2D
input: (None, 66, 66, 64)
output: (None, 32, 32, 128)
batchnormalization_13: BatchNormalization
input: (None, 32, 32, 128)
output: (None, 32, 32, 128)
leakyrelu_8: LeakyReLU
input: (None, 32, 32, 128)
output: (None, 32, 32, 128)
zeropadding2d_10: ZeroPadding2D
input: (None, 32, 32, 128)
output: (None, 34, 34, 128)
convolution2d_9: Convolution2D
input: (None, 34, 34, 128)
output: (None, 16, 16, 256)
batchnormalization_14: BatchNormalization
input: (None, 16, 16, 256)
output: (None, 16, 16, 256)
leakyrelu_9: LeakyReLU
input: (None, 16, 16, 256)
output: (None, 16, 16, 256)
zeropadding2d_11: ZeroPadding2D
input: (None, 16, 16, 256)
output: (None, 18, 18, 256)
conv4: Convolution2D
input: (None, 18, 18, 256)
output: (None, 15, 15, 512)
batchnormalization_15: BatchNormalization
input: (None, 15, 15, 512)
output: (None, 15, 15, 512)
leakyrelu_10: LeakyReLU
input: (None, 15, 15, 512)
output: (None, 15, 15, 512)
zeropadding2d_12: ZeroPadding2D
input: (None, 15, 15, 512)
output: (None, 17, 17, 512)
conv5: Convolution2D
input: (None, 17, 17, 512)
output: (None, 14, 14, 1)
activation_9: Activation
input: (None, 14, 14, 1)
output: (None, 14, 14, 1)

Figure 27: Keras model for discriminator **D** - model 2

## C cGAN Detail



## Codes

### Codes developed so far

Listing 1: model.py for generator discriminator and combined GAN model

```
from keras.models import Model, Sequential
from keras.layers import Activation, Input, Reshape, merge, Lambda, Dropout, Flatten, Dense
from keras.layers.convolutional import Convolution2D, Deconvolution2D, ZeroPadding2D, Cropping
from keras.layers.advanced_activations import LeakyReLU
from keras.layers.normalization import BatchNormalization
from keras.utils.visualize_util import plot

import keras.backend as K
import numpy as np

def conv(x, nf, norm=True):
    y = LeakyReLU(0.2)(x)
    y = ZeroPadding2D(padding=(1, 1))(y)
    y = Convolution2D(nf, 4, 4, subsample=(2, 2))(y)
    if norm:
        y = BatchNormalization(mode=2)(y)
    return y

def deconv(x, nf, output_size, norm=True):
    y = Activation('relu')(x)
    y = Deconvolution2D(nf, 4, 4, output_shape=(None, output_size+2, output_size+2, nf), subsa
    y = Cropping2D(cropping=((1, 1), (1, 1)))(y)
    if norm:
        y = BatchNormalization(mode=2)(y)
    return y

def generator(input_size=128, nf=64, p=0.5):
    g_inputs = Input(shape=(input_size, input_size, 3))

    # 128 * 128 * 3
    e1 = ZeroPadding2D(padding=(1, 1))(g_inputs)
    e1 = Convolution2D(nf, 4, 4, subsample=(2, 2))(e1)

    # 64 * 64 * nf
    e2 = conv(e1, nf * 2)
    # 32 * 32 * (nf * 2)
    e3 = conv(e2, nf * 4)
    # 16 * 16 * (nf * 4)
    e4 = conv(e3, nf * 8)
    e5 = conv(e4, nf * 8)
    # 8 * 8 * (nf * 8)
    # 4 * 4 * (nf * 8)
    e6 = conv(e5, nf * 8)
    # 2 * 2 * (nf * 8)
    e7 = conv(e6, nf * 8)
    # 1 * 1 * (nf * 8)
    g1 = deconv(e7, nf * 8, output_size=2)
    g1 = Dropout(p)(g1)
    g1 = merge([g1, e6], mode='concat', concat_axis=-1)
    # 2 * 2 * (nf * 8 * 2)
    g2 = deconv(g1, nf * 8, output_size=4)
    g2 = Dropout(p)(g2)
    g2 = merge([g2, e5], mode='concat', concat_axis=-1)
    # 4 * 4 * (nf * 8 * 2)
    g3 = deconv(g2, nf * 8, output_size=8)
    g3 = Dropout(p)(g3)
    g3 = merge([g3, e4], mode='concat', concat_axis=-1)
    # 8 * 8 * (nf * 8 * 2)
    g4 = deconv(g3, nf * 4, output_size=16)
    g4 = merge([g4, e3], mode='concat', concat_axis=-1)
    # 16 * 16 * (nf * 4 * 2)
    g5 = deconv(g4, nf * 2, output_size=32)
    g5 = merge([g5, e2], mode='concat', concat_axis=-1)
    # 32 * 32 * (nf * 2 * 2)
    g6 = deconv(g5, nf, output_size=64)
    g6 = merge([g6, e1], mode='concat', concat_axis=-1)
    # 64 * 64 * (nf * 2)
    g7 = deconv(g6, 3, output_size=128, norm=False)
    g_outputs = Activation('tanh')(g7)

    G = Model(input=g_inputs, output=g_outputs)
    plot(G, to_file='../model/model_G.png', show_shapes=True)
    return G

def discriminator(input_size=128, nf=64, n_layers=3):
    d_inputs = Input(shape=(input_size, input_size, 6))
    # 128 * 128 * 6
    d_hidden = ZeroPadding2D(padding=(1, 1))(d_inputs)
    d_hidden = Convolution2D(nf, 4, 4, subsample=(2, 2))(d_hidden)
    # 64 * 64 * nf
    ndf = nf
    for i in range(n_layers -1):
        ndf = min(nf * 8, ndf * 2)
        d_hidden = conv(d_hidden, ndf)
    ndf = min(nf * 8, ndf * 2)
    d_hidden = LeakyReLU(0.2)(d_hidden)
    d_hidden = ZeroPadding2D(padding=(1, 1))(d_hidden)
    d_hidden = Convolution2D(ndf, 4, 4, subsample=(1, 1))(d_hidden)
    d_hidden = BatchNormalization(mode=2)(d_hidden)
    d_hidden = LeakyReLU(0.2)(d_hidden)
    d_hidden = ZeroPadding2D(padding=(1, 1))(d_hidden)
    d_hidden = Convolution2D(1, 4, 4, subsample=(1, 1))(d_hidden)
    d_outputs = Activation('sigmoid')(d_hidden)
    # d2 = conv(d1, nf * 2)
    # # 32 * 32 * (nf * 2)
    # d3 = conv(d2, nf * 4)
    # # 16 * 16 * (nf * 4)
    # d4 = conv(d3, nf * 8)
    # # 8 * 8 * (nf * 8)
    # d5 = conv(d4, nf * 8)
    # # 4 * 4 * (nf * 8)
    # d6 = conv(d5, nf * 8)
    # # 2 * 2 * (nf * 8)
    # d7 = conv(d6, nf * 8)
    # # 1 * 1 * (nf * 8)
    # d8 = Flatten()(d7)
    # d8 = Dense(1)(d8)
    # d_outputs = Activation('sigmoid')(d8)

    D = Model(input=d_inputs, output=d_outputs)
    plot(D, to_file='../model/model_D.png', show_shapes=True)
    return D

def cGAN(G, D, input_size=128):
    gan_inputs = Input(shape=(input_size, input_size, 3))
    g_outputs = G(gan_inputs)
    d_inputs = merge([gan_inputs, g_outputs], mode='concat', concat_axis=-1)
    d_hidden = D.layers[1](d_inputs)
    for i in range(2, len(D.layers)):
        d_hidden = D.layers[i](d_hidden)

    GAN = Model(input=gan_inputs, output=[g_outputs, d_hidden])
    return GAN

```

```
if __name__ == "__main__":
    G = generator()
    D = discriminator()
    cGAN = cGAN(G, D)
```

Listing 2: preprocess.py for data preprocessing

```
from PIL import Image
import numpy as np
from os.path import isdir
from os import makedirs

def preprocess(x):
    # Scale input 0 to 255 to -1 to 1.
    func = np.vectorize(lambda x: x/127.5-1)
    return func(x)

def deprocess(x):
    # Scale output -1 to 1 back to 0 to 255.
    func = np.vectorize(lambda x: (x+1)*127.5)
    return func(x)

def readData(dataset='../data/cityscapes/', size=128, n_train=1024, n_valid=128):
    print("Reading Data...")
    x_train = np.empty([n_train, size, size, 3])
    y_train = np.empty([n_train, size, size, 3])

    img = Image.open(dataset + 'train/1.jpg')
    w, h = img.size
    for i in range(1, n_train+1):
        s = dataset + "train/" + str(i) + ".jpg"
        img = Image.open(s)
        limg = img.crop((0, 0, w/2, h)).resize((size, size))
        rimg = img.crop((w/2, 0, w, h)).resize((size, size))

        x_train[i-1] = np.array(rimg.getdata()).reshape(size, size, 3)
        y_train[i-1] = np.array(limg.getdata()).reshape(size, size, 3)

    x_test = np.empty([n_valid, size, size, 3])
    y_test = np.empty([n_valid, size, size, 3])

    for i in range(1, n_valid+1):
        s = dataset + "val/" + str(i) + ".jpg"
        img = Image.open(s)
        limg = img.crop((0, 0, w/2, h)).resize((size, size))
        rimg = img.crop((w/2, 0, w, h)).resize((size, size))

        x_test[i-1] = np.array(rimg.getdata()).reshape(size, size, 3)
        y_test[i-1] = np.array(limg.getdata()).reshape(size, size, 3)

    x_train = preprocess(x_train)
    y_train = preprocess(y_train)
    x_test = preprocess(x_test)
    y_test = preprocess(y_test)

    print("Finish")
    print("Training samples: %d\nTesting samples: %d"%(len(x_train), len(x_test)))

    return x_train, y_train, x_test, y_test


def showImage(x, y, gx, title='image', size=128):
    img = Image.new("RGB", (size*3, size))
    im = np.append(x, y, axis=1)
    im = np.append(im, gx, axis=1)
    im = list(im.reshape(size * size * 3, 3))
    im = [map(int, z) for z in im]
    im = map(tuple, im)
    img.putdata(im)
    img.show(title)

def saveImage(x, y, gx, title='', saveDir='', size=128):
    img = Image.new("RGB", (size*3, size))
    im = np.append(x, y, axis=1)
    im = np.append(im, gx, axis=1)
    im = list(im.reshape(size * size * 3, 3))
    im = [map(int, z) for z in im]
    im = map(tuple, im)
    img.putdata(im)
    if not isdir(saveDir):
        makedirs(saveDir)
    img.save(saveDir + str(title) + '.png')

if __name__ == '__main__':
    readData()
```

Listing 3: train.py for training sample code

```
from tqdm import *
from model import generator, discriminator, cGAN
from preprocess import readData, showImage, saveImage, preprocess, deprocess
from keras.models import Model
from keras.optimizers import Adam
import numpy as np
from PIL import Image
import math
import keras.backend as K


EPOCH = 50
BATCH_SIZE = 8
N_FILTER = 64
N_LAYER = 4
L1_WEIGHT = 100
N_TRAIN = 2968
N_VALID = 256
INPUT_SIZE = 128
DATASET = '../datasets/cityscapes/'

G = generator(nf=N_FILTER)
D = discriminator(nf=N_FILTER, n_layers=N_LAYER)
GAN = cGAN(G, D)

G_optim = Adam(lr=0.0002, beta_1=0.5)
G.compile(optimizer=G_optim, loss='binary_crossentropy')
G.summary()

D_optim = Adam(lr=0.0002, beta_1=0.5)
D.compile(optimizer=D_optim, loss='binary_crossentropy')
D.summary()

GAN.compile(optimizer=G_optim, loss=['mean_absolute_error', 'binary_crossentropy'], loss_weig

x_train, y_train, x_valid, y_valid = readData(dataset=DATASET, n_train=N_TRAIN, n_valid=N_VALI

patch_size = D.output_shape[1]

real_label = np.array([1] * BATCH_SIZE * patch_size * patch_size).reshape(BATCH_SIZE, patch_si
fake_label = np.array([0] * BATCH_SIZE * patch_size * patch_size).reshape(BATCH_SIZE, patch_si

real_vlabel = np.array([1] * len(x_valid) * patch_size * patch_size).reshape(len(x_valid), pat
fake_vlabel = np.array([0] * len(x_valid) * patch_size * patch_size).reshape(len(x_valid), pat

print(”Start␣training”)
print(”Batch␣size␣=␣%d”%(BATCH_SIZE))
print(”#␣of␣filter␣for␣first␣layer␣=␣%d”%(N_FILTER))
print(”#␣of␣layer␣for␣discriminator␣=␣%d”%(N_LAYER))
print(”lambda␣for␣L1␣=␣%d”%(L1_WEIGHT))

for epo in range(1, EPOCH+1):
    print(”Epoch␣%d/%d”%(epo, EPOCH))
    train_size = len(x_train)
    valid_size = len(x_valid)
    n_batch = train_size / BATCH_SIZE

    # train on batch
    for i in tqdm(range(n_batch)):
        x_batch = x_train[i*BATCH_SIZE:(i+1)*BATCH_SIZE]
        y_batch = y_train[i*BATCH_SIZE:(i+1)*BATCH_SIZE]

        gen = G.predict_on_batch(x_batch)
        fake = np.append(x_batch, gen, axis=3)
        real = np.append(x_batch, y_batch, axis=3)

        D.trainable = True
        D.train_on_batch(real, real_label)
        D.train_on_batch(fake, fake_label)

        D.trainable = False
        GAN.train_on_batch(x_batch, [y_batch, real_label])

    # test on validation set
    gen = G.predict_on_batch(x_valid)
    fake = np.append(x_valid, gen, axis=3)
    real = np.append(x_valid, y_valid, axis=3)

    lossD_real = D.test_on_batch(real, real_vlabel)
    lossD_fake = D.test_on_batch(fake, fake_vlabel)
    errG = GAN.test_on_batch(x_valid, [y_valid, real_vlabel])
    lossD = (lossD_fake + lossD_real) / 2
    print(”On␣validation␣set,␣lossD␣=␣%f,␣L1_G␣=␣%f,␣lossG␣=␣%f”%(lossD, errG[0], errG[1]))
    for i in range(5):
        #showImage(deprocess(x_valid[i]), deprocess(y_valid[i]), deprocess(fake[i,:,:,3:]), st
        saveImage(deprocess(x_valid[i]), deprocess(y_valid[i]), deprocess(fake[i,:,:,3:]), str
    # save weights for every 10 epoch
    if epo % 10 == 0:
        G.save_weights(”../model/G_weight.h5”)
        D.save_weights(”../model/D_weight.h5”)

G.save_weights(”../model/G_weight.h5”)
D.save_weights(”../model/D_weight.h5”)
```