

The Chinese University of Hong Kong

CSCI4999 - MHW1903

Final Year Project II

Term 2 Report

Supervisor: Prof. WONG Man Hon

Students: CHOI Ki Fung



TSANG Ka Hung



Prepared by: CHOI Ki Fung



Table of Content

1. Introduction.....	6
1.1. Motivation.....	6
1.1.1. Emoji.....	6
1.1.2. Chinese Calligraphy.....	7
1.2. Background.....	9
1.2.1. Generative Adversarial Network (GAN).....	9
1.2.2. Conditional GAN (CGAN).....	10
1.2.3. Image-to-Image Translation with Conditional Adversarial Nets (Pix2Pix).....	11
1.2.4. Cycle-Consistent Adversarial Networks (Cycle GAN).....	13
1.2.5. Previous work.....	15
2. Data Gathering.....	16
2.1. Emoji.....	16
Unicode.....	16
Images.....	16
Description.....	17
2.2. Chinese Calligraphy.....	18
3. Data Preprocessing.....	21
3.1. Emoji.....	21

<i>Preprocessing the images of emojis</i>	21
3.1.1. CGAN Data Preprocessing.....	22
Labels.....	22
3.2. Chinese Calligraphy.....	24
<i>Example of the inconsistent dimension of raw collected images</i>	26
<i>Example of the preprocessed images</i>	27
4. Neural Network Architecture Design	29
4.1. Overview of Generative Adversarial Network (GAN).....	29
Overview of Generator	29
Generator Detail.....	30
Overview of Discriminator	31
Discriminator Detail.....	31
GAN Result.....	34
4.2. Overview of Conditional Generative Adversarial Network (CGAN)	37
Overview of Generator	38
Generator Detail.....	39
Overview of Discriminator	40
Discriminator Detail.....	41
CGAN Result and Improvement.....	42
14 conditions CGAN.....	43

58 conditions CGAN.....	54
4.3. Overview of Pix2Pix.....	60
Pix2Pix: Generator.....	61
Pix2Pix: Discriminator.....	62
Pix2Pix Methodology	63
Encoder-decoder network.....	63
U-net	64
PatchGAN.....	65
Loss function.....	66
Adversarial loss (MiniMax Loss)	66
Problem of Pix2Pix.....	69
4.4. Overview of Cycle-Consistent Adversarial Network (Cycle GAN).....	71
Model design.....	72
Cycle GAN: Generator (5 ResNet version)	72
Cycle GAN: Discriminator	75
Cycle GAN Methodology	77
Residual block (ResNet).....	77
DenseNet.....	78
Loss function.....	78
Cycle Consistency Loss	80

Cycle GAN Encoder + Decoder Only Version Result	81
Cycle GAN U-net Version Result.....	83
Cycle GAN Resnet Version Result.....	85
Cycle GAN ResNet + U-net Version Result.....	88
Cycle GAN DenseNet Version Result.....	90
Non-Chinese character Result.....	91
5. Difficulties and Solution.....	93
The transparency layer of the image.....	93
Either generator or discriminator is too good	94
Lack of Data.....	96
Future Work.....	99
Learning the overall structure of Chinese character and font style without supervised learning	99
Learn multiple font styles in one network	99
Solve real-life problem	99
Division of labor	101
Reference	105

1. Introduction

1.1. Motivation

This project is to explore the different possibilities of Generative Adversarial Network. Initially, we aimed to generate new emojis for daily communication. However, the number of data we can collect for emoji is very limited. Then, we switched our focus on Chinese Calligraphy, which has far more data and various styles. Below, we will explain the motivations why we choose these two domains of problem.

1.1.1. Emoji

With the rise of smartphones in the last decade, emojis were added to several operating systems. People use instant messaging apps, like WhatsApp and Telegram, more frequently than SMS and call. According to Quito (2019), “Numerous studies suggest that 55% of human communication is through body language—gestures, posture, facial expression—and 38% is conveyed by a speaker’s tone and inflection.” However, it is limited in digital messaging. Thus, emojis were used as ideograms to encode the sender’s emotional and social cues, hoping that the recipient understands what he/ she is trying to convey.

The total number of emojis is 3,019 as of September 2019 (Unicode, 2019), but some emojis use the same appearance or different skin tones. The Unicode adds about a hundred of new emojis each year. With the development of neural networks, we wish to use GAN to generate synthetic emojis for people to use.

1.1.2. Chinese Calligraphy

On top of modern emoji, another thing we desire to explore is Chinese Calligraphy. It is one of the crucial elements of Chinese art and one of the Four Arts of The Chinese Scholar. Besides, the history of Chinese Calligraphy can be traced back to the Oracle bone script, the first form of Chinese characters, in 1300 B.C. Then, Chinese Calligraphy derived into different stream and style, for example, Clerical, Cursive, Semi-cursive, Regular script (隸書、草書、行書、楷書).

However, in the modern world, people have been using computers and mobile phones. We seldom write on paper, let alone calligraphy. Fewer and fewer people know how to calligraph, and fewer and fewer calligrapher.

Moreover, some of the greatest Chinese calligrapher's artworks have passed through centuries down to this day. However, there is only a limited number and character. What if Wang Xizhi(王羲之) can write a Fai Chun for me? Or Su Shi(蘇軾)?

Locally, we have seen Chinese Calligraphy every day from the street sign and minibus sign. Sadly, some of the authors of these signs are passed away, and this kind of art is diminishing. For example, Uncle Lee¹ who was a street calligrapher with his work at every corner, Tsang Tsou Choi², a.k.a King of Kowloon, famous

¹ <https://www.facebook.com/pg/leehonhk>

² https://en.wikipedia.org/wiki/Tsang_Tsou_Choi

for his calligraphy graffiti, and Mr. Mak³, the last minibus sign designer in Hong Kong.



Replicated work of Uncle Lee



The last minibus sign designer

Therefore, we wish to use GAN to capture the style of these calligraphers and regenerate them with different possibilities.

3

<https://www.eldage.com/pages/%E6%89%8B%E5%AF%AB%E5%B0%8F%E5%B7%B4%E7%89%8C%E5%B7%A5%E4%BD%9C%E5%9D%8A>

1.2. Background

1.2.1. Generative Adversarial Network (GAN)

GAN is a deep neural network proposed by Goodfellow, et al. in 2014. GAN comprises two neural networks, a generator, and a discriminator. The generator takes a random number from the latent space, where features lie, to generate a sample. It could be an image, audio, text, etc. The generated samples are then mixed with the real samples to form batches of training data and feed into the discriminator. For each of the input samples, the discriminator has to classify that it is a real-world sample or a sample generated by the generator. Then, a loss function uses the result to calculate the loss and update the models. If the discriminator successfully identifies a sample, it will be rewarded for recognizing the generator's flaws. Similarly, if the generator successfully fakes a sample without being caught, it will be rewarded for generating more samples like that one.

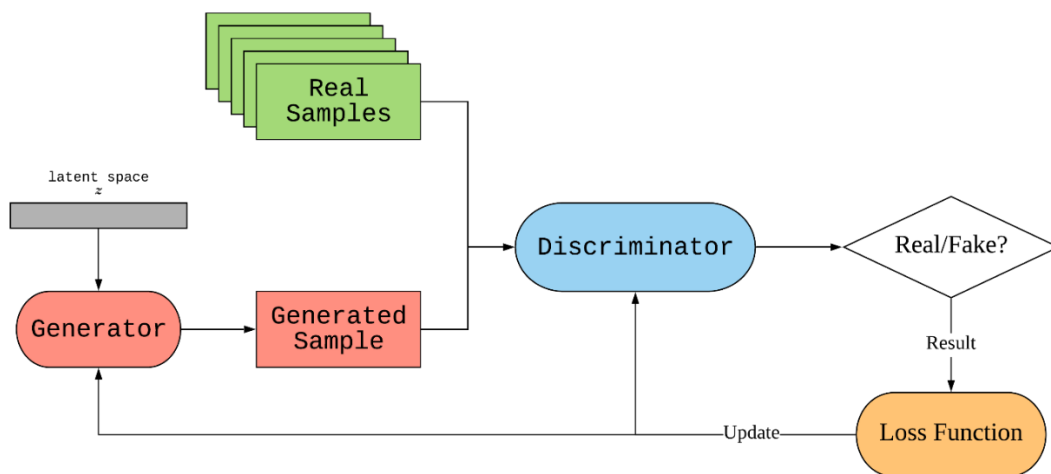


Illustration of GAN

1.2.2. Conditional GAN (CGAN)

Though GAN produces a plausible and similar result inferring from a given dataset, we may want a concise outcome in real life instead of a random, uncontrollable one. Conditional Generative Adversarial Nets, CGAN, is an extension of GAN with the ability to generate sample on a class label (Mirza & Osindero, 2014). CGAN is merely adding the labels as an additional layer, but the modification has to be carried out on both generator and discriminator for balance. The generator takes a random number from latent space z given a label y and outputs a synthetic sample. The discriminator takes a sample x , which can be from the generator or real-world, given the label y and predict the authenticity of x .

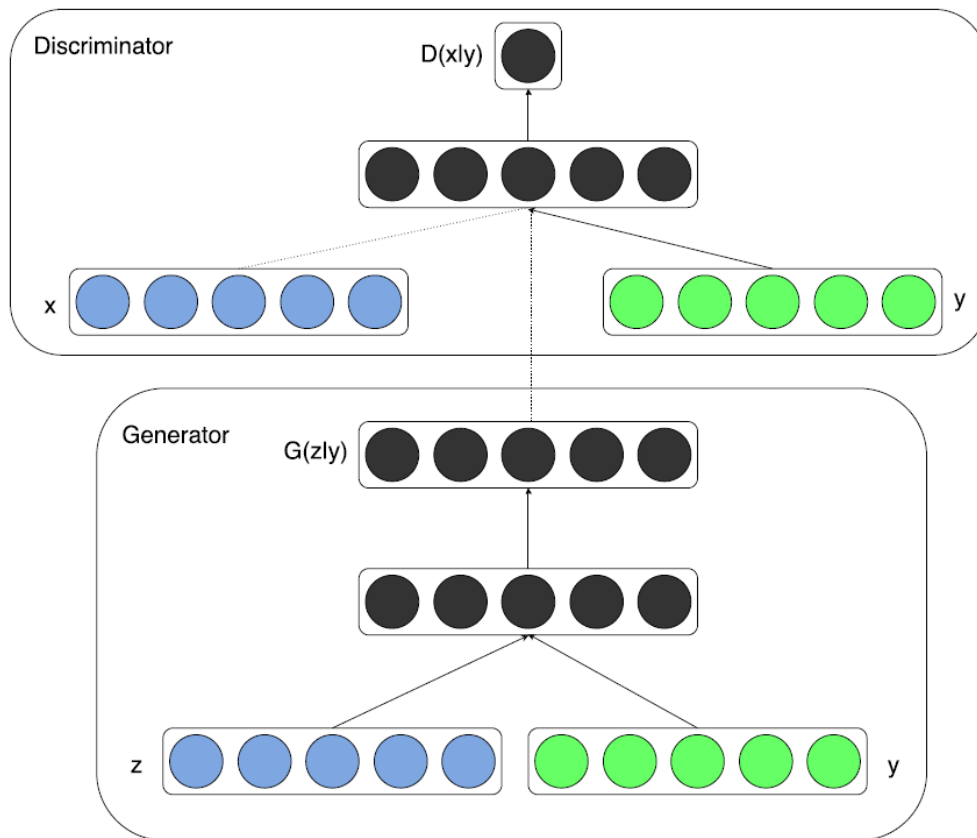
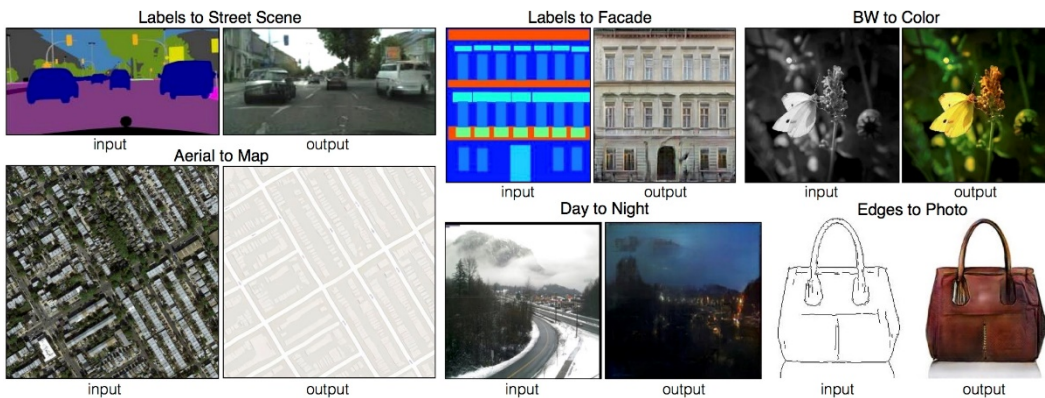


Illustration of CGAN. (Mirza & Osindero, 2014)

1.2.3. Image-to-Image Translation with Conditional Adversarial Nets (Pix2Pix)

Pix2Pix is an astonishing network that solves the image-to-image translation problem (Isola, Zhu, Zhou, & Efros, 2017). After the network is trained, it developed a generic loss function that is dedicated to a specific domain of image mapping, such as mapping a satellite image to a map. When switching to another domain, the only thing to do is just training on a different set of data, and the architecture of the network does not need to be changed.



Example usage of image-to-image translation problems. (Isola, Zhu, Zhou, & Efros, 2017)

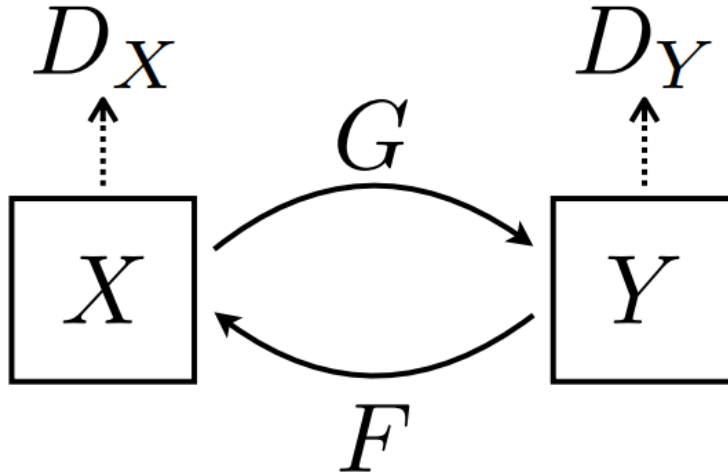
Pix2Pix is derived from CGAN, where, in this time, the synthetic image is based on the input image. In the training phase, the network generator is fed with the source image. In this time, the discriminator is not only provided the generated image. Both the source and the generated images are given to increase the power of the discriminator. The generator and decimator battled again each other, in an adversarial manner.

Although Pix2Pix based on CGAN, Pix2Pix tweak the internal component of the network to make it more robust and giving more promising result.

1.2.4. Cycle-Consistent Adversarial Networks (Cycle GAN)

In Pix2Pix, image-to-image translation is trained using a set of paired training images, e.g., a set of photography(input) and a set of corresponding desired painting(output). However, most of the time, our training data is unpaired; for example, the paintings do not depict the same scene in the photographs. One of the reasons is that it is merely expensive to prepare the paired dataset, e.g., hiring someone to do the painting. Another reason is that the data is in the past, e.g., a painting by Van Gogh.

Cycle GAN is an unsupervised learning network to solve this unpaired image-to-image translation (Zhu, Park, Isola, & Efros, 2017). Cycle GAN is able to extract general characteristics from two sets of unrelated images and exploit that in the image translation process. Traditional GAN has a problem called mode collapse, which is the generator found out an especially plausible output that can fool the discriminator and generate only that output. The variety of output of the GAN is lost in mode collapse. There is a possibility that the generator generates an output that has no correlation with the input while fooling the discriminator. To alleviate this problem, Cycle GAN uses a technique called cycle consistency.



The basic structure of Cycle GAN. (Zhu, et al., 2017)

The idea is to implement two GANs (G & F). The newly introduced GAN F is to translate the output image back to the input image. Because if we can revert to the input image, it proves that the G is not generating an arbitrary image but actually doing the translation.

1.2.5. Previous work

We found user ‘anoff’ on GitHub has developed a deep emoji generative adversarial network about generating new emoji⁴.

This project provides some different network designs and hyperparameters to try to generate new emojis with DCGAN. Here is his final sample:

Final sample



We found that this project’s final sample looks like the same. Most of them contain dot eyes and a smile/poker face. Moreover, some of the quality is bad. We want to improve this project by designing a new GAN network. Also, trying to provide some condition on the emoji so we can somehow control what we expect the emoji look like.

⁴ <https://github.com/anoff/deep-emoji-gan>

2. Data Gathering

A well begun is half done. The quality and quantity of data are undoubtedly the factors of yielding a good result. Therefore, here we identify the critical data that needs to be collected here and the reason why.

2.1. Emoji

Before collecting any data, we identified three types of data are needed. First is the Unicode that defines an emoji. Second is the raw images of emojis, which have to in a consistent dimension. The third is the description of an emoji, which is essential to construct a CGAN.

Unicode

The Unicode is the unique value to identify an emoji, e.g., 🍌 = 1F600. We discovered that Dabbas(2019) has already maintained an up-to-date emoji database in CSV format with 3827 entries in Kaggle⁵ which is an online website to find and publish data sets. Each emoji is paired with a Unicode value and groups. The Unicode value can be used to extract the corresponding emoji image in another repository.

Images

Though the Unicode Standard is the unified consortium to design or approve the new emoji, the Unicode characters are more about the “identity” rather than the

⁵ <https://www.kaggle.com/eliasdabbas/emoji-data-descriptions-codepoints>

appearance, in consequence, there are different sets of emoji from each company, just like different fonts for English characters.







Downcast face emojis from Apple, Facebook, Google, Messenger, and Twitter.

A GitHub repository⁶ has collected groups of emoji photos and provided a way to parse emoji data and images easily. It contains a JSON file about the meta-data of the emojis, like the image file name of the emoji, and whether a company has published an emoji.

Description

The description of an emoji is useful for building up the neural network. However, there is no secondhand data which was made by other people before. We have to input the description of an emoji one by one. Using the database from Kaggle, we add a new column about the characteristic of the emojis.

emoji	name	codepoints	labels
	grinning face	1F600	face, grinning, smiling, joy, teeth, pleasure, cheer, open eyes, open mouth
	grinning face with big eyes	1F603	face, grinning, smiling, joy, teeth, big eyes, oval eyes, open mouth
	grinning face with smiling eyes	1F604	face, grinning, smiling, joy, laughing, smiling eyes, squinting eyes, open mouth, teeth
	beaming face with smiling eyes	1F601	face, beaming, smiling, joy, laughing, smiling eyes, Cheese, teeth, warm, gratified, proud, amused

Labeled Excel Data

⁶ <https://github.com/iamcal/emoji-data>

2.2. Chinese Calligraphy

There are three parts for the data about the Unicode, font, and handwritten word.

Unicode

The Unicode is the unique value to identify different Chinese characters, e.g., 一 = `\u4e00`. We discovered that most of the traditional Chinese characters are in the range of `\u4e00` to `\u9fd5`, which contains about 20949 of traditional Chinese characters. We make a JSON file to store all those Unicode for further usage.

Font

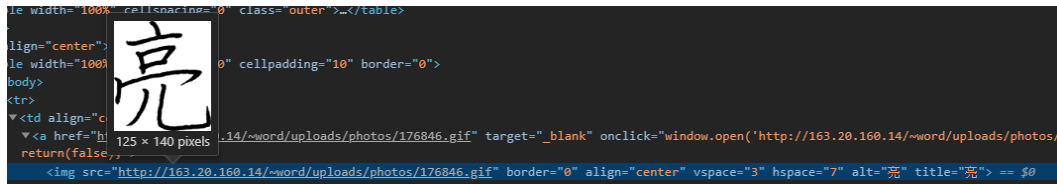
In a modern computer operating system, there are many build-in fonts on the OS. For Windows 10, there are two font styles which are related to traditional Chinese characters which are MingLiU(新細明體) and DFKai-SB(標楷體). We can directly get their TrueType file(.ttf), which is the standard format for fonts on the OS. Also, we collect some more font on the internet⁷, which is 王羲之書法字體 (we name it hzwong).

By collecting fonts, we can convert the font characters into images. This significantly reduces the effort required to prepare a paired dataset for training the network. The details of the operation will be elaborated in the Methodology section below.

⁷ <http://www.diyiziti.com/>

Handwritten word

There are two types of handwritten words. First one is scraped from 「陳忠建字庫」⁸. Unfortunately, unlike emoji, there is currently no related database or repository that archives Chinese calligraphy. 「陳忠建字庫」 is a personal website of a calligrapher, Mr. Chen Chung Chien, which stores over 210,000 of his own handwritten Chinese calligraphy images with various style. We use a python package called Beautiful Soup to pull the data from the website. First, we go to the album page of the website and extract all the images page by page. In Figure 1, the source of the image and the text label are encapsulated in the tags.



Example of the image and the corresponding DOM element

However, as our program hit the server too frequently, the webserver crashed and banned Hong Kong IP addresses for a period of time. At this point, we scraped about 14,000 images. We deem that it is enough for training the model referring to the last experiment we did in the last term.

The second type of data is the handwritten Chinese characters written by us. We pick around 1000 commonly used Chinese characters from CUHK HUMANUM⁹

⁸ <http://163.20.160.14/>

⁹ <https://humanum.arts.cuhk.edu.hk/Lexis/lexi-can/faq.php>

for our project and write those Chinese characters on Windows Paint and paper with size 256x256.



Handwritten word by using paint with size 256x256



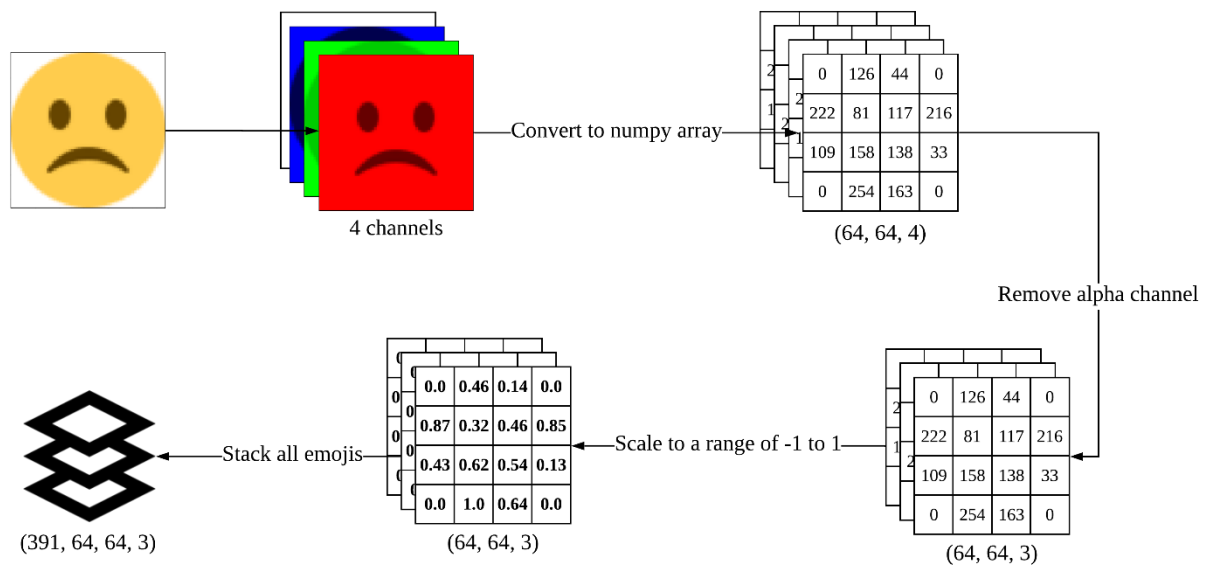
Handwritten word on paper with size 256x256

3. Data Preprocessing

The data we collected from the web or prepared by us need to be adjusted for consistency before training. For instance, the dimension, format, and position of the images.

3.1. Emoji

The images extracted from the GitHub repository we mentioned earlier are filtered by the Unicode we specified and turned into a NumPy array. Since the images were encoded in PNG format, the transparency channel should be removed, i.e., turning the transparent pixel to a white background. Then, scale down the image pixel value from 0 ~ 255 to -1 ~ 1, which is generally a common trick to normalize an image in building a neural network.



Preprocessing the images of emojis

3.1.1. CGAN Data Preprocessing

Labels

The images of the emojis and the labels have to be fed into the neural network, but the network cannot directly take as inputs. We used Pandas¹⁰, a data manipulation and analysis tool, to preprocess the data. We turned the description into One-Hot Encode data using Pandas, resulting in a 91 rows x 420 columns table.

First, we use Pandas to read our CSV file and save it to a dataframe(*df*). We select the first 91 emojis, which are face emojis only.

```
In [1]: import pandas as pd
import numpy as np
from sklearn.preprocessing import MultiLabelBinarizer

In [2]: df = pd.read_csv("emoji_df_excel.csv")

In [4]: df = df[:91]
df.head()
```

Out[4]:

	emoji	name	group	sub_group	codepoints	labels
0	😊	grinning face	Smileys & Emotion	face-smiling	1F600	face, grinning, smiling, joy, teeth, pleasure,...
1	😄	grinning face with big eyes	Smileys & Emotion	face-smiling	1F603	face, grinning, smiling, joy, teeth, big eyes,...
2	😁	grinning face with smiling eyes	Smileys & Emotion	face-smiling	1F604	face, grinning, smiling, joy, laughing, smilin...
3	😆	beaming face with smiling eyes	Smileys & Emotion	face-smiling	1F601	face, beaming, smiling, joy, laughing, smiling...
4	😏	grinning squinting face	Smileys & Emotion	face-smiling	1F606	face, grinning, squinting, joy, laughing, broa...

Then, we transform the “labels” column, which describes the emoji by words into one-hot encoding.

¹⁰ <https://pandas.pydata.org/>

```
In [5]: one_hot = df.labels.str.split('\s*',\s*', expand=True) \
        .stack() \
        .str.get_dummies() \
        .sum(level=0)
one_hot.head()
```

```
Out[5]:
```

	\$	><	Cheese	ROFL	X-shaped eyes	XD	abashed	admiration	adoration	adventure	...	winking	woozy	worried	wow	ya
0	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0
3	0	0	1	0	0	0	0	0	0	0	...	0	0	0	0	0
4	0	1	0	0	1	1	0	0	0	0	...	0	0	0	0	0

5 rows × 419 columns

Next, we drop the original “labels” column and concatenate our newly created one-hot labels to the dataframe. Finally, we drop the “name,” “group,” “sub_group,” and “codepoints” columns, which are unnecessary for training the neural network.

```
In [7]: df.iloc[:5, np.r_[0, 5:len(df.columns)]]
```

```
Out[7]:
```

	emoji	\$	><	Cheese	ROFL	X-shaped eyes	XD	abashed	admiration	adoration	...	winking	woozy	worried	wow	yawnin
0	😄	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0
1	😄	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0
2	😄	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0
3	😄	0	0	1	0	0	0	0	0	0	...	0	0	0	0	0
4	😄	0	1	0	0	1	1	0	0	0	...	0	0	0	0	0

5 rows × 420 columns

Final and neat labels of the emojis

3.2. Chinese Calligraphy

Two preprocessing tasks have to be carried out for the font from .ttf file and the image collected on the Internet.

For the font collected from the .ttf file. We need to make Chinese characters become images and feed into the neural network. We use the pillow(PIL) library¹¹, which is a Python Image Processing Library to make a simple program to generate those images. For each Unicode, we create a new image with a size 256x256 match with different font. Store the image into separate folders categorize by the font family.

```
# for all unicode with chinese character
for i in range(len(CN_T_CHARSET)):
    img = Image.new("RGB", (256, 256), (255, 255, 255))
    draw = ImageDraw.Draw(img)

    # set the font and font size
    font = ImageFont.truetype(TTF_PATH, FONT_SIZE)
    # put the Chinese character with specify font into the image
    draw.text((0, 0), CN_T_CHARSET[i], (0, 0, 0), font=font)
    # save the image
    img.save(IMAGES_DIR+"1_" + CN_T_CHARSET[i]+".jpg")
```

Code of generating Chinese Characters by font

After that, by manually checking all the generated images, some images are non-Chinese characters or blank image because the font does not have a design for that Unicode. We handpicked it up and deleted it.

¹¹pillow(PIL) library <https://pillow.readthedocs.io>



Non-Chinese Characters and Blanked Generated image

As a result, MingLiU has 18554 images, DFKai-SB has 18542 images, and hzwong has 6861 images.



MingLiU font Chinese Characters

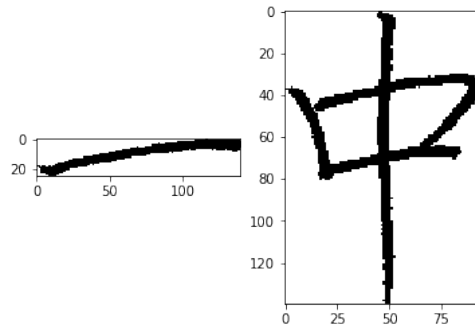


DFKai-SB font Chinese Characters

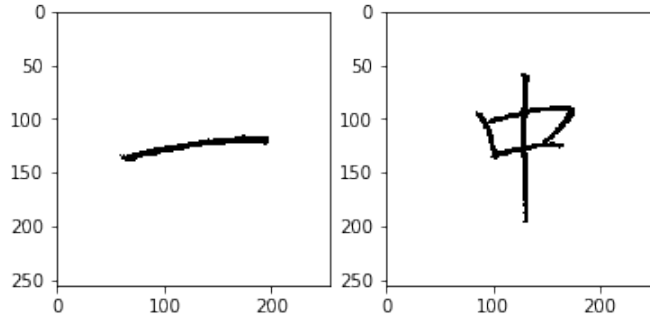


hzwong font Chinese Characters

For the image collected from the Internet. The images of calligraphy that we collected come with different dimensions. The neural network normally has a fixed size in the input layer, so we have to resize the image to a consistent resolution.



Example of the inconsistent dimension of raw collected images



Example of the preprocessed images

Because of the large number of images, it is inefficient for us to resize each of them one by one. We used an open-source software called ImageMagick¹² to do batch image processing. Since the raw images are not in the 1:1 aspect ratio, we have to first scale up the images uniformly. Then, set the canvas size to 256x256 and shift the images to the center. Finally, fill the unoccupied area with white color to act as a background.

Before the image is loaded into the network. For all Chinese character images, we do a process called random jitter, which means we enlarge the image from 256x256x3 to a larger scale. Then random crop the image back to 256x256x3 and randomly flip the image horizontally. This process will improve model training and make the model not easy to overfit.

¹² <https://imagemagick.org/>

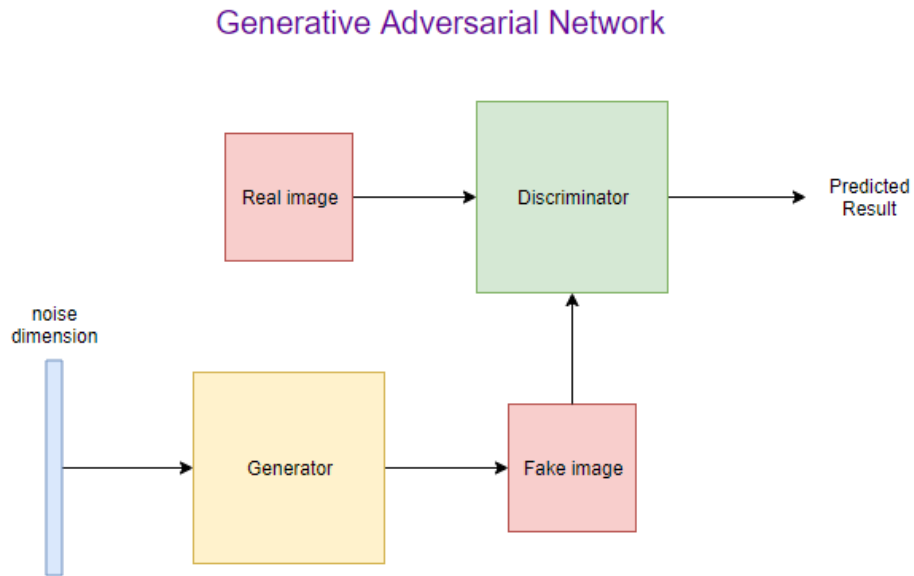
```
def random_jitter(image):  
    # resizing to 286 x 286 x 3  
    image = tf.image.resize(image, [286, 286],  
                             method=tf.image.ResizeMethod.NEAREST_NEIGHBOR)  
    # randomly cropping to 256 x 256 x 3  
    image = random_crop(image)  
    # random mirroring  
    image = tf.image.random_flip_left_right(image)  
    return image
```

Code of random jittering

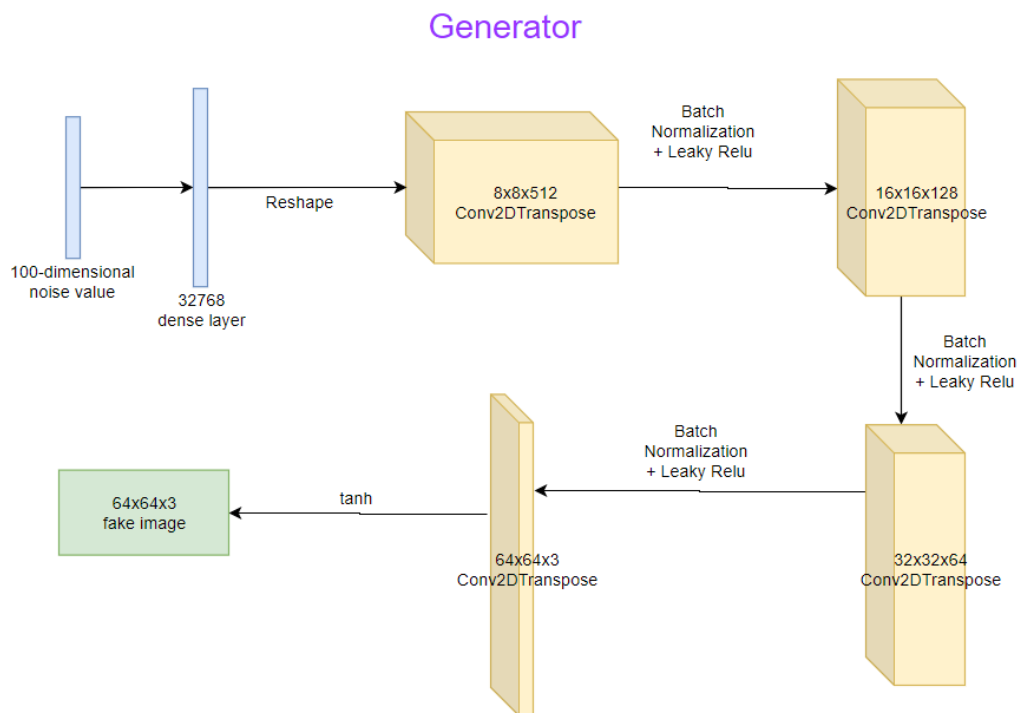
After that, we do the same thing like emoji images to normalize the image from image pixel value from 0 ~ 255 to -1 ~ 1.

4. Neural Network Architecture Design

4.1. Overview of Generative Adversarial Network (GAN)



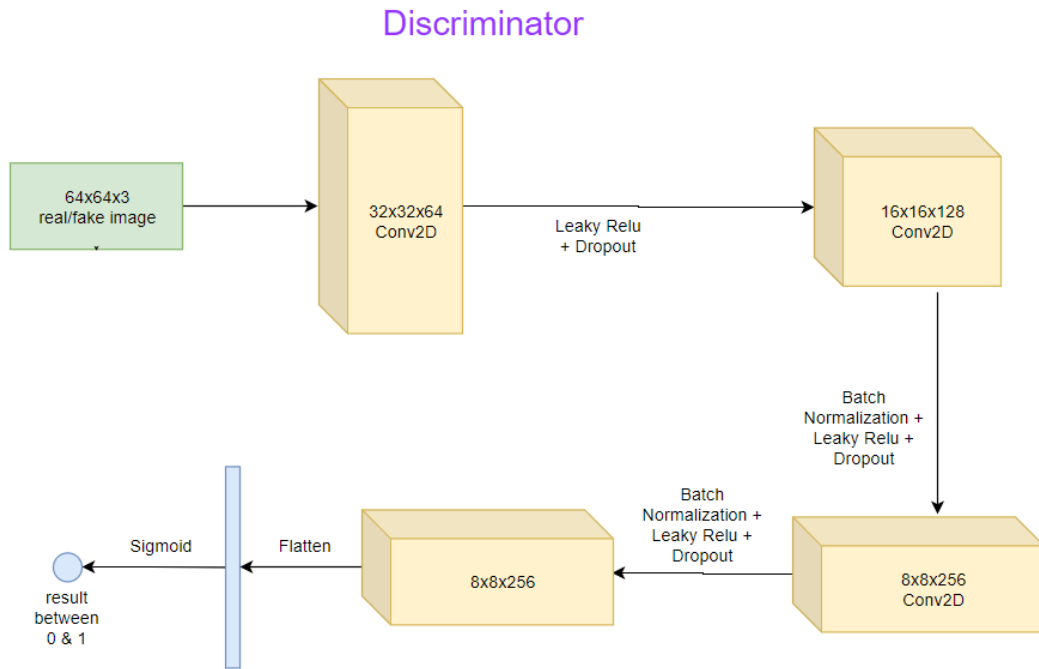
Overview of Generator



Generator Detail

noise_dim: Input Layer	input:	(None, 100)
	output:	(None, 100)
dense: Dense Layer	input:	(None, 100)
	output:	(None, 32768)
reshape: Reshape Layer	input:	(None, 32768)
	output:	(None, 8, 8, 512)
conv2d_transpose_1: Conv2DTranspose Layer	input:	(None, 8, 8, 512)
	output:	(None, 16, 16, 128)
batch_normalization: BatchNormalization Layer	input:	(None, 16, 16, 128)
	output:	(None, 16, 16, 128)
leaky_re_lu: LeakyReLU Layer	input:	(None, 16, 16, 128)
	output:	(None, 16, 16, 128)
conv2d_transpose_1: Conv2DTranspose Layer	input:	(None, 16, 16, 128)
	output:	(None, 32, 32, 64)
batch_normalization_1: BatchNormalization Layer	input:	(None, 32, 32, 64)
	output:	(None, 32, 32, 64)
leaky_re_lu_1: LeakyReLU Layer	input:	(None, 32, 32, 64)
	output:	(None, 32, 32, 64)
conv2d_transpose_2: Conv2DTranspose Layer	input:	(None, 32, 32, 64)
	output:	(None, 64, 64, 3)
tanh activation: Activation Layer	input:	(None, 64, 64, 3)
	output:	(None, 64, 64, 3)

Overview of Discriminator



Discriminator Detail

images: Input Layer	input:	(None, 64, 64, 3)
	output:	(None, 64, 64, 3)
conv2d: Conv2D Layer	input:	(None, 64, 64, 3)
	output:	(None, 32, 32, 64)
leaky_re_lu_2: LeakyReLU Layer	input:	(None, 32, 32, 64)
	output:	(None, 32, 32, 64)
dropout: Dropout Layer	input:	(None, 32, 32, 64)
	output:	(None, 32, 32, 64)

conv2d_1: Conv2D Layer	input:	(None, 32, 32, 64)
	output:	(None, 16, 16, 128)
batch_normalization_2: BatchNormalization Layer	input:	(None, 16, 16, 128)
	output:	(None, 16, 16, 128)
leaky_re_lu_3: LeakyReLU Layer	input:	(None, 16, 16, 128)
	output:	(None, 16, 16, 128)
dropout_1: Dropout Layer	input:	(None, 16, 16, 128)
	output:	(None, 16, 16, 128)
conv2d_2: Conv2D Layer	input:	(None, 16, 16, 128)
	output:	(None, 8, 8, 256)
batch_normalization_3: BatchNormalization Layer	input:	(None, 8, 8, 256)
	output:	(None, 8, 8, 256)
leaky_re_lu_4: LeakyReLU Layer	input:	(None, 8, 8, 256)
	output:	(None, 8, 8, 256)
dropout_2: Dropout Layer	input:	(None, 8, 8, 256)
	output:	(None, 8, 8, 256)
flatten: Flatten Layer	input:	(None, 8, 8, 256)
	output:	(None, 16384)
dense_1: Dense Layer (activation: sigmoid)	input:	(None, 16384)
	output:	(None, 1)

Loss function:

We tried two different loss functions to try to build the network. They are Minimax loss and Wasserstein loss.

Minimax Loss

$$E_x[\log(D(x))] + E_z[\log(1 - D(G(z)))]$$

$D(x)$ is the probability of the discriminator's estimate the real image x is real

$G(z)$ is the generator's output when given noise z

$D(G(z))$ is the probability of the discriminator's estimate the fake image x is real

E_x is the expected value of all the real images

E_z is the expected value of all the fake images

The generator aims to minimize this function while the discriminator aims to maximize this function.

In the GAN that we build with minimax loss function, the activation of the output layer in the discriminator is sigmoid, and the optimizer is Adam.

Wasserstein Loss

Wasserstein loss in discriminator is not output the range (0, 1) to classify the image is real or not. It outputs a number only to try to make the number of real images is much bigger than the number of fake images.

Discriminator loss: $D(x) - D(G(z))$

The discriminator tries to maximize the value of this function

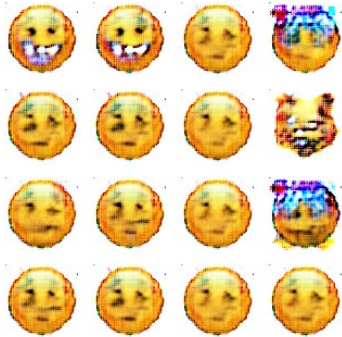
Generator loss: $D(G(z))$

The generator tries to maximize the value of this function

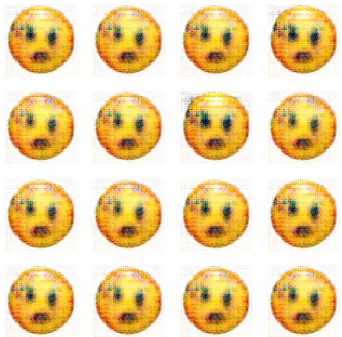
We follow the suggestion in the paper of Wasserstein GAN (Arjovsky, Martin, Soumith, & Léon. , 2017), the activation of the output layer in the discriminator is linear, and the optimizer is RMSProp.

GAN Result

Using the GAN architecture similar to ‘anoff’ one (<https://github.com/anoff/deep-emoji-gan>):



Epoch 2505, using minimax loss function (first_gan.py)



Epoch 25305, using Wasserstein loss function (cwgan.py)

Using the GAN model mention above using minimax loss function (DCGAN.ipynb):



Epoch 8900

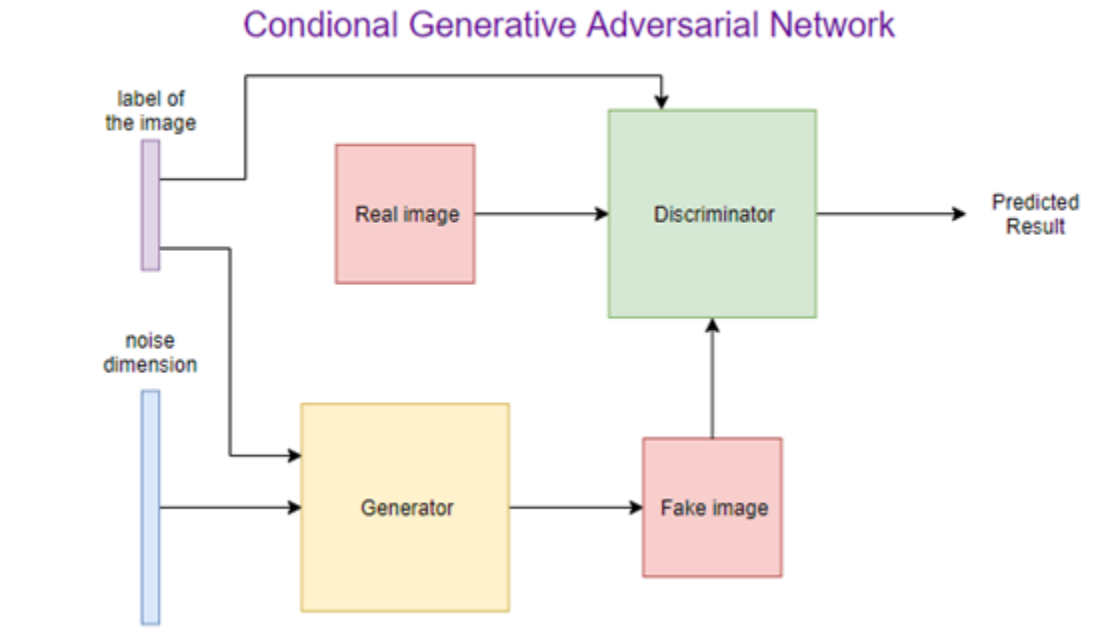


Epoch 9700

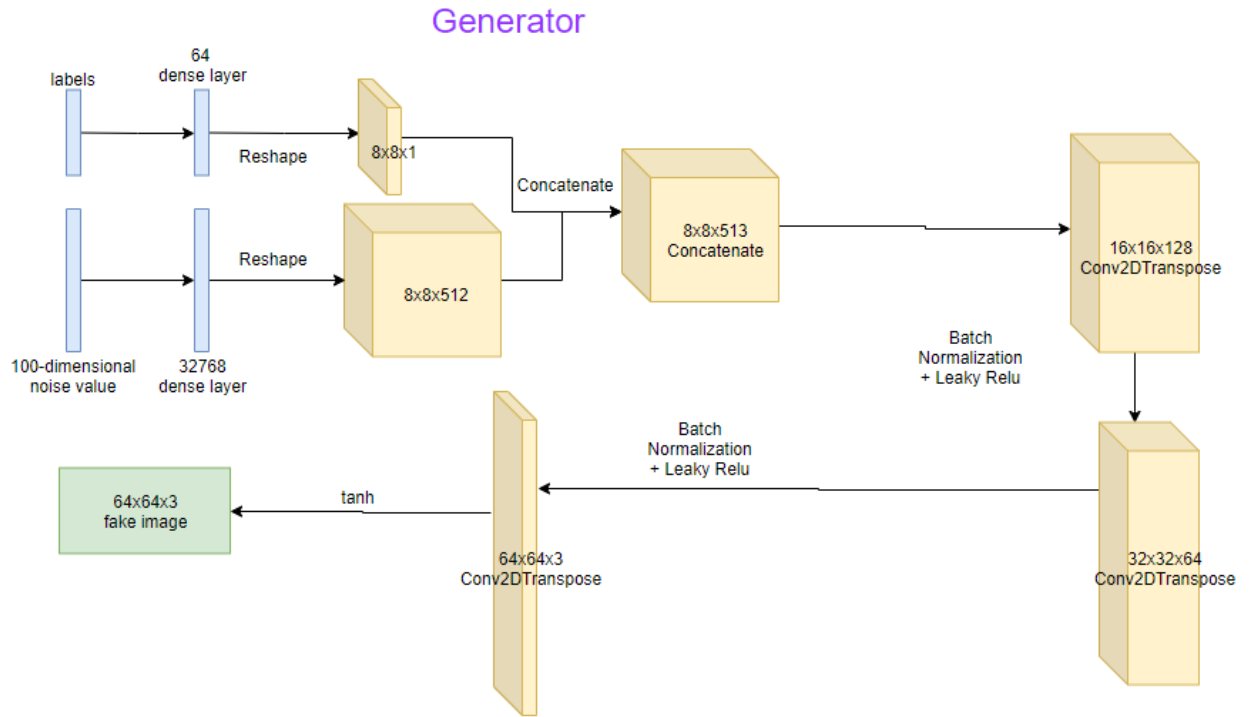
The above result of different loss functions shows that minimax loss function seems to have a better result with less training epoch. Therefore, when we build our conditional-GAN, we would like to use the minimax loss function in our model.

4.2. Overview of Conditional Generative Adversarial Network (CGAN)

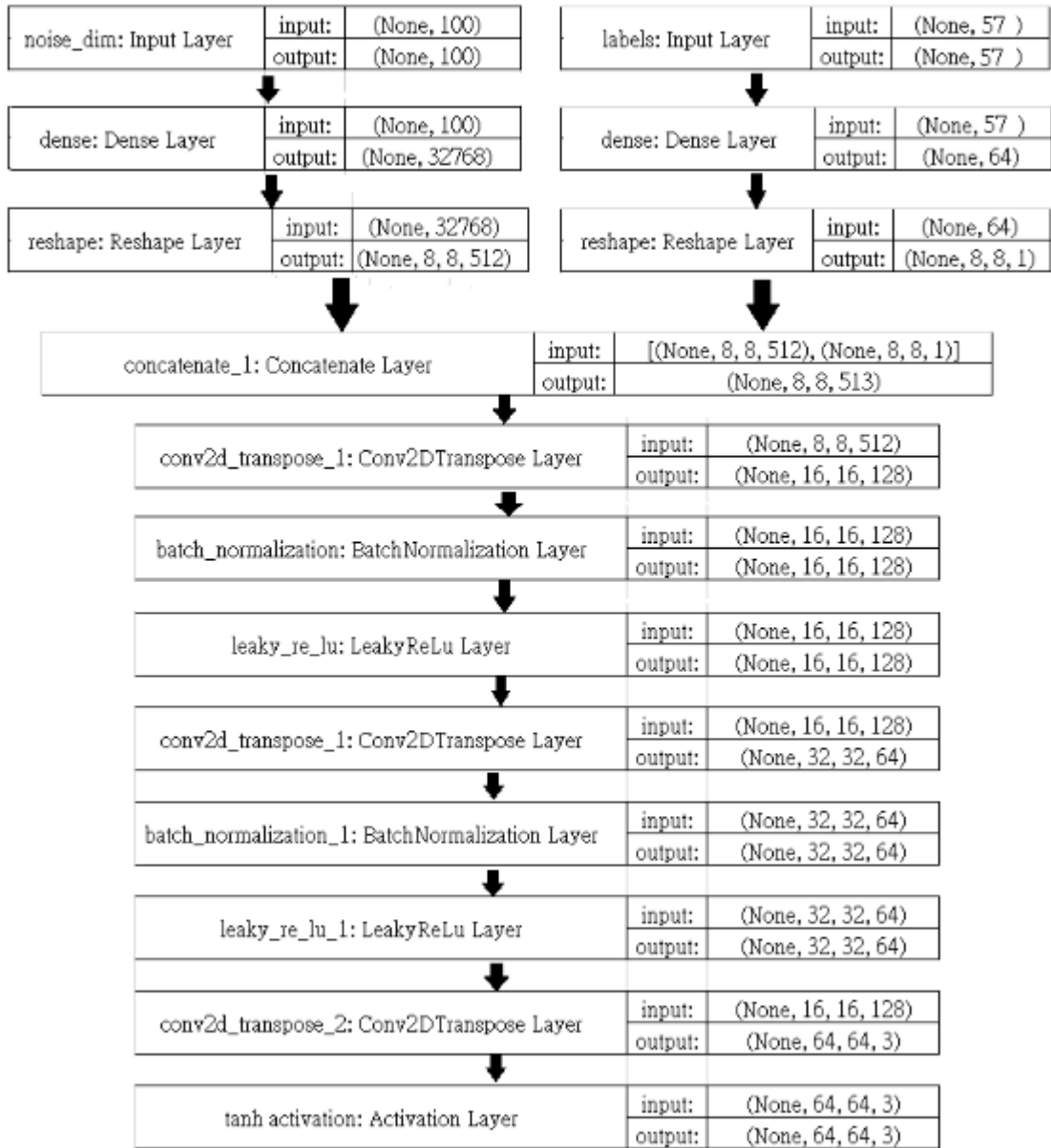
We used the concept of DCGAN as our model base (Radford, Metz & Chintala, 2016). And followed the idea provided by Conditional Generative Adversarial Nets (Mirza & Osindero, 2014)



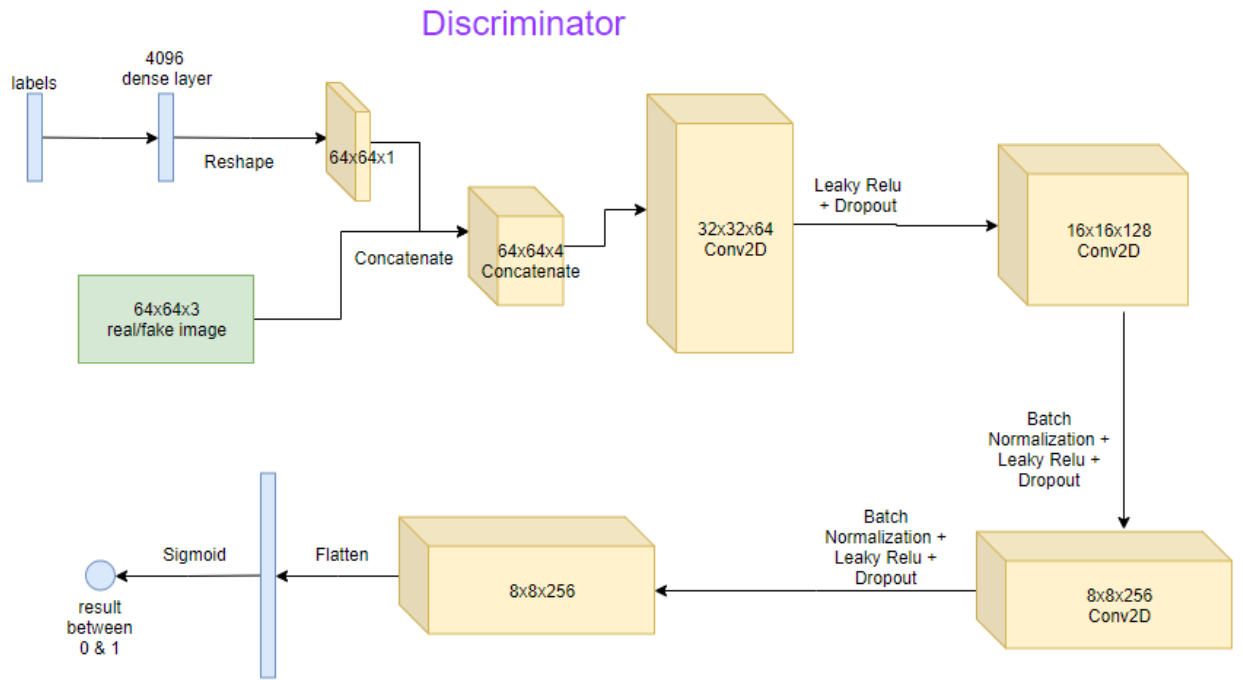
Overview of Generator



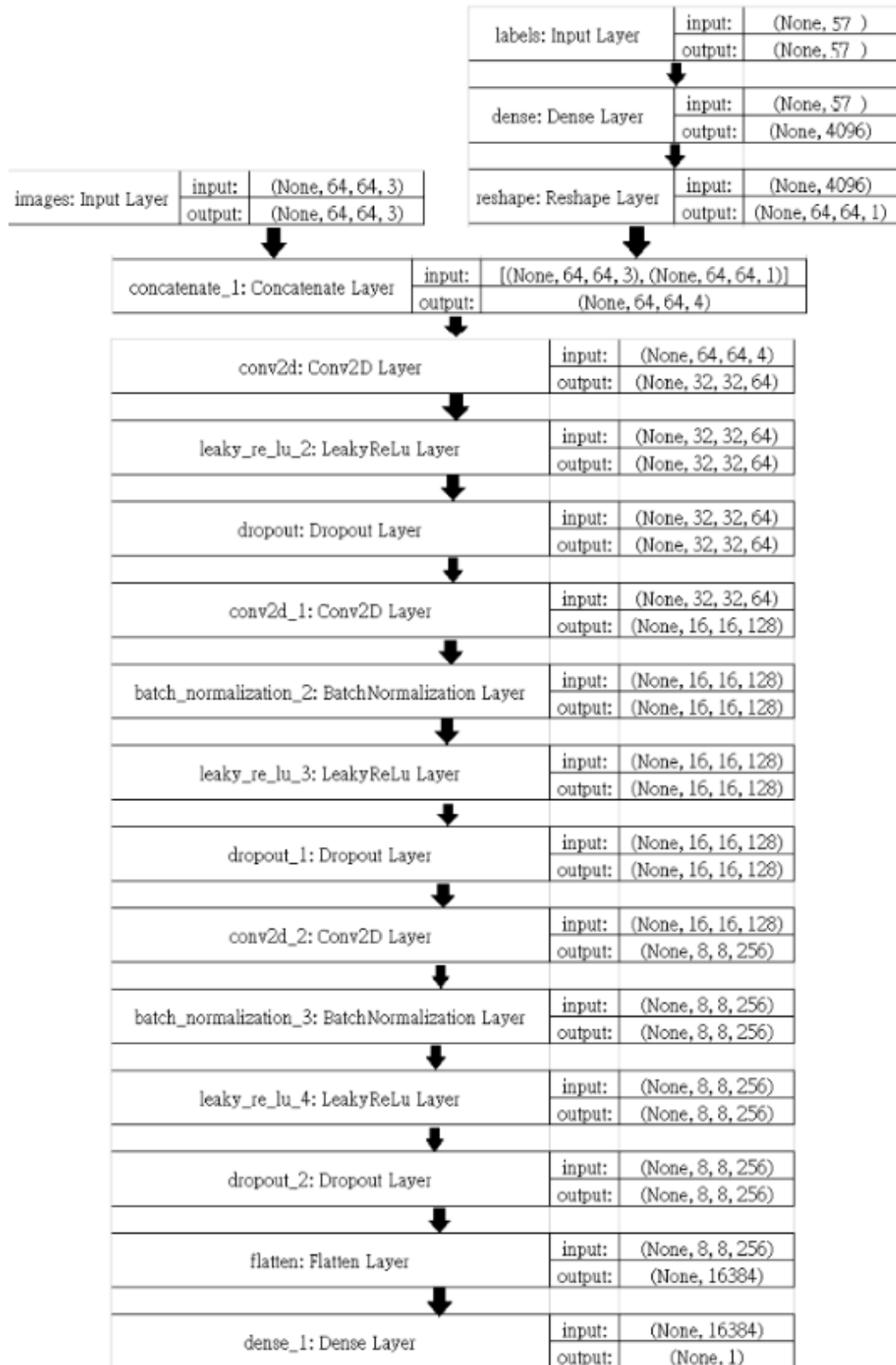
Generator Detail



Overview of Discriminator




Discriminator Detail



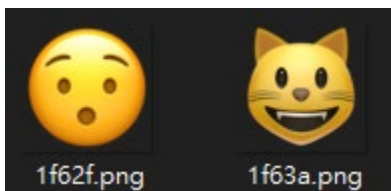
CGAN Result and Improvement

When we used the preprocessing method above to make the label as the condition, the generated emoji was not following the label. It was no different from the normal GAN, and we cannot control the generated emoji we wanted. The problem of the initially provided label in the dataset is that it contains too much label. It contains a total of 418 labels. If we took an in-depth look into it, those labels are too specific.

For example, this emoji  contains [face, grinning, smiling, joy, teeth, pleasure, cheer, open eyes, open mouth]. That is a total of 9 labels for one emoji. This is way too much and shows that most of the labels are similar.

Therefore, we decided to make the label of each emoji by ourselves. We are not just lowering the amount of the label, but also making the label more general and not similar to each other.







We marked the label focusing on three main parts of the emoji – eye, mouth, and other special effects. The example of other special effect is like these two emoji:











1f62f.png contains the eyebrow. 1f63a.png is a cat.

14 conditions CGAN

To start with something simple to test it works. We picked the most general label as our condition. Here is the table of our labeling, there is a total of 14 labels:

Eye			
heart_eye	m_eye	sad_eye	dot_eye
			
poker_eye	Xd_eye		
			

Mouth			
smile	tougue	laugh	o_mouth
			
poker_face	sad_mouth		
			

Other			
eye_brow	cat		
			

As we just selected a few conditions, it let some emoji cannot fit the condition. Therefore, we just provided a total of 48 different emojis and a total of 236 samples to train the conditional-GAN.

As we self-defined our CSV file (face_emoji/emoji_onehot.csv), we needed to modify our preprocessing method to handle the label that can feed into our model.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1	emoji	smile	tongue	laugh	o_mouth	poker_face	sad_mouth	heart_eye	m_eye	sad_eye	dot_eye	poker_eye	xd_eye	eye_brow	cat
2	1f60a	1	0	0	0	0	0	0	1	0	0	0	0	0	0
3	1f60b	1	0	0	0	0	0	0	1	0	0	0	0	0	0
4	1f60d	0	0	1	0	0	0	1	0	0	0	0	0	0	0
5	1f61b	0	1	0	0	0	0	0	0	0	1	0	0	0	0
6	1f61d	0	1	0	0	0	0	0	0	0	0	0	1	0	0
7	1f61e	0	0	0	0	0	1	0	0	1	0	0	0	0	0
8	1f61f	0	0	0	0	0	1	0	0	0	1	0	0	1	0
9	1f62b	0	0	0	0	0	1	0	0	0	0	0	1	1	0
10	1f62e	0	0	0	1	0	0	0	0	0	1	0	0	0	0
11	1f62f	0	0	0	1	0	0	0	0	0	1	0	0	1	0
12	1f63a	0	0	1	0	0	0	0	0	0	1	0	0	0	1
13	1f63b	0	0	1	0	0	0	1	0	0	0	0	0	0	1
14	1f63c	1	0	0	0	0	0	0	0	1	0	0	0	1	1
15	1f63e	0	0	0	0	0	1	0	0	0	1	0	0	1	1
16	1f600	0	0	1	0	0	0	0	0	0	1	0	0	0	0
17	1f601	0	0	1	0	0	0	0	1	0	0	0	0	0	0
18	1f602	0	0	1	0	0	0	0	1	0	0	0	0	1	0
19	1f603	0	0	1	0	0	0	0	0	0	1	0	0	0	0
20	1f604	0	0	1	0	0	0	0	1	0	0	0	0	0	0
21	1f605	0	0	1	0	0	0	0	1	0	0	0	0	0	0
22	1f606	0	0	1	0	0	0	0	0	0	0	0	1	0	0
23	1f607	1	0	0	0	0	0	0	1	0	0	0	0	0	0
24	1f609	1	0	0	0	0	0	0	0	0	1	1	0	1	0
25	1f610	0	0	0	0	1	0	0	0	0	1	0	0	0	0
26	1f611	0	0	0	0	1	0	0	0	0	0	1	0	0	0
27	1f612	0	0	0	0	0	1	0	0	0	1	1	0	1	0
28	1f613	0	0	0	0	0	1	0	1	0	0	1	0	0	0
29	1f614	0	0	0	0	1	0	0	0	1	0	0	0	1	0
30	1f615	0	0	0	0	1	1	0	0	0	1	0	0	0	0
31	1f620	0	0	0	0	0	1	0	0	0	1	0	0	1	0
32	1f622	0	0	0	0	0	1	0	0	0	1	0	0	1	0
33	1f623	0	0	0	0	0	1	0	0	0	0	0	1	1	0
34	1f624	0	0	0	0	1	1	0	0	0	0	1	0	1	0
35	1f625	0	0	0	0	0	1	0	0	0	1	0	0	1	0
36	1f626	0	0	0	1	0	0	0	0	0	1	0	0	0	0
37	1f627	0	0	0	1	0	0	0	0	0	1	0	0	1	0
38	1f628	0	0	0	1	0	0	0	0	0	1	0	0	1	0
39	1f630	0	0	0	1	0	1	0	0	0	1	0	0	1	0

First, we loaded all the image paths from the file paths and stored them into a list.

Then we filtered the image paths by their filename is existed in our CSV file.

Furthermore, we also needed their file name to match our CSV file, so we stored those names into another list.

```

▶ paths = [
    "/content/drive/My Drive/Colab Notebooks/face_emoji/img-apple-64",
    "/content/drive/My Drive/Colab Notebooks/face_emoji/img-facebook-64",
    "/content/drive/My Drive/Colab Notebooks/face_emoji/img-google-64",
    "/content/drive/My Drive/Colab Notebooks/face_emoji/img-messenger-64",
    "/content/drive/My Drive/Colab Notebooks/face_emoji/img-twitter-64"
]
filepaths = [] #store all the image path that we used
labels = [] #store the file name
filtered = [] #store the image path that exist in our csv file
for p in paths:
    image_paths = glob.glob(p + "/*.png") #get all the image paths
    for p in image_paths:
        for i in range(df.index.stop):
            if p.split("/")[-1].split('.')[0] in df['emoji'][i]: #check the image file name is in our csv file
                filtered.append(p)
for f in filtered:
    labels.append(f.split('/')[-1].split('.')[0]) #store the image file name
filepaths = filtered

```

Next, we needed to create one-hot encoding labels. We made a Numpy array that the size is matching our total number of the label.

```

▶ one_hot = np.empty((1, 14), dtype="int32")
for label in labels:
    label = df.loc[df['emoji'] == label].drop(columns=['emoji']) #drop the firsts columns which define the emoji file name
    label = label.to_numpy()
    one_hot = np.concatenate((one_hot, label)) #put the csv file data into the numpy array
one_hot = one_hot[1:]

```

After that, we needed to change the label input shape of our generator and discriminator. In this case, the input shape is 14.

```

# The generator takes noise as input and generates imgs
z = Input(shape=(self.latent_dim,))
z_label = Input(shape=(14, ))
img = self.generator([z, z_label])

```

```

def build_generator(self):
    noise = Input(shape=(self.latent_dim,), name='noise_in')
    labels = Input(shape=(14,), name='labels_in')

```

```

def build_discriminator(self):
    labels = Input(shape=(14,), name='labels_in')

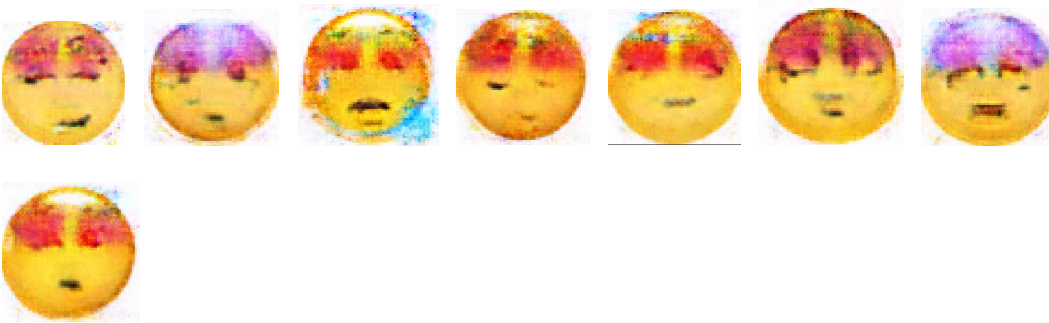
```

Here is some result after training 5000 epochs:

label: dot_eye, laugh



label: heart_eye, o_mouth



label: poker_eye, poker_face



label: sad_eye, sad_mouth, cat



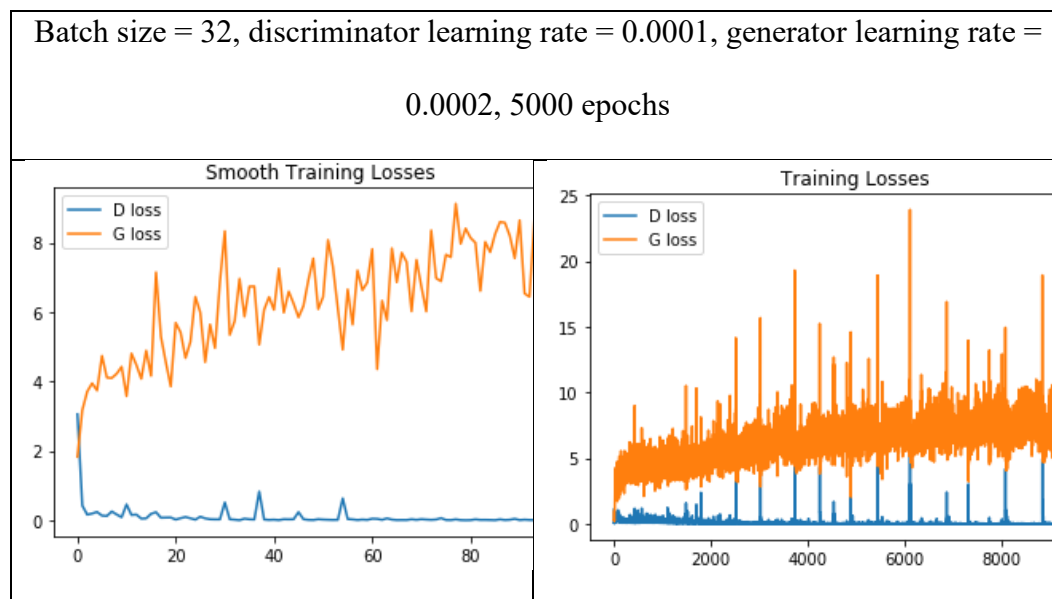
label: xd_eye, smile, eye_brow



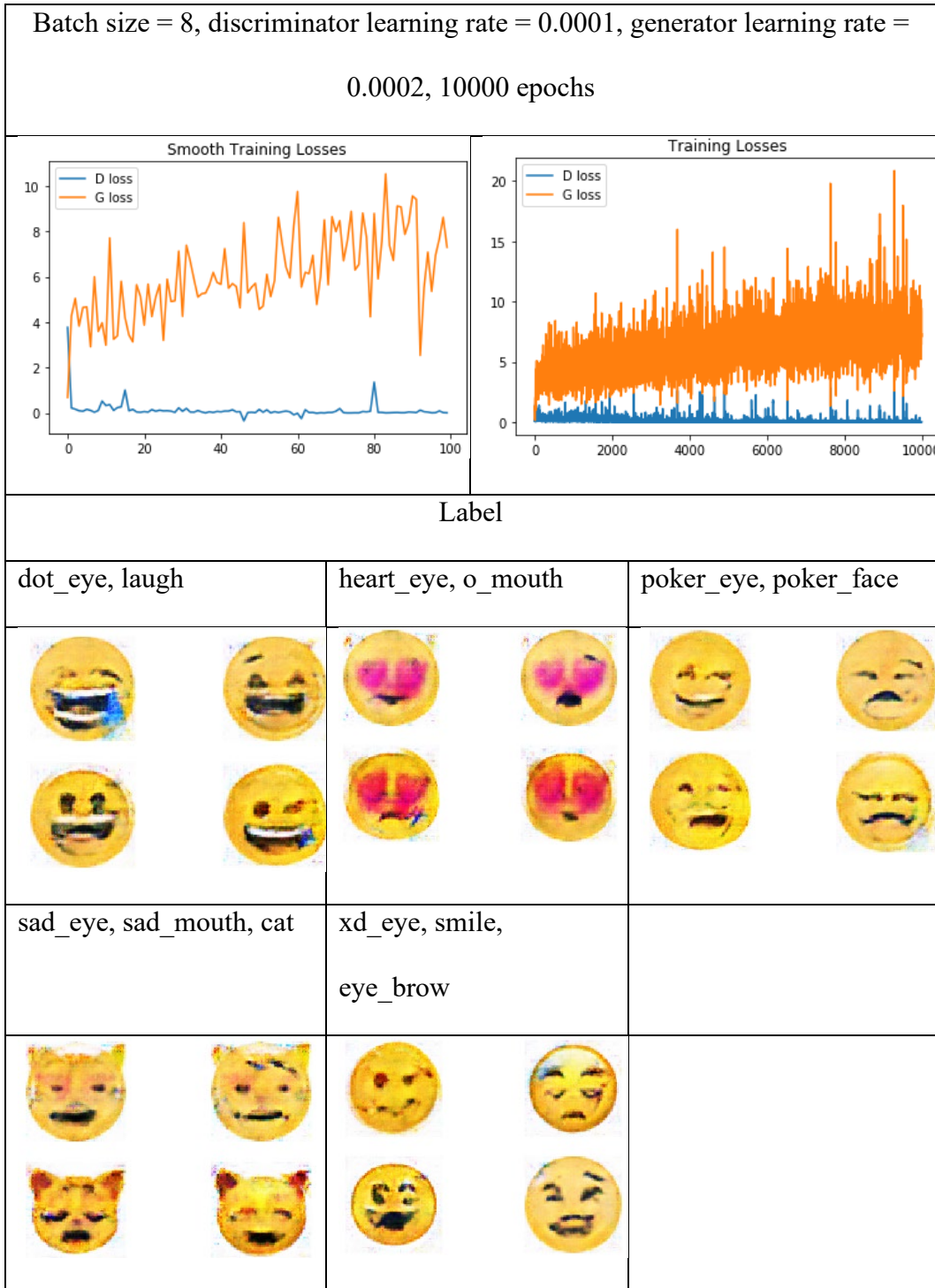
We found that some of the generated emoji can produce an emoji that matches the label. However, there was some mistake in matching the label. For example, xd_eye

cannot show on our model, and some emoji label with the cat did not look like a cat. The model was not stable enough to produce the emoji that matched our label.

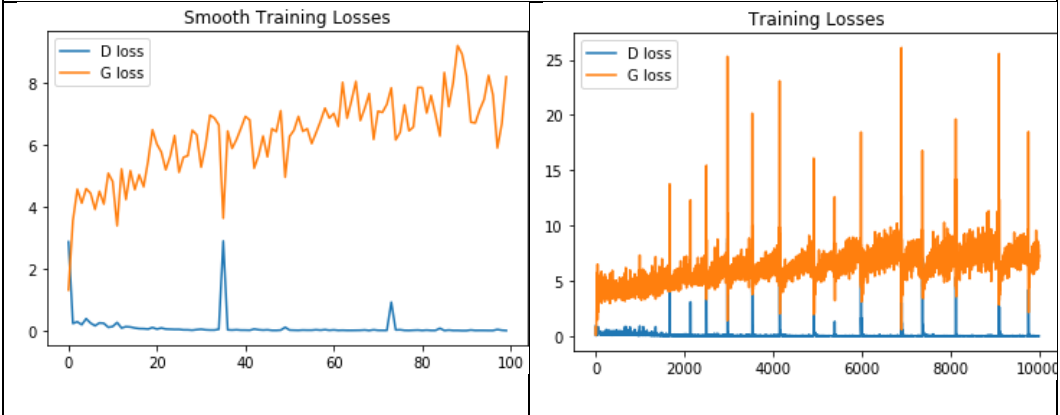
After checking the training loss, we found that the generator loss kept increasing while the discriminator kept its loss low. That meant the discriminator is dominating the generator.



Therefore, we tried changing different parameters to see if the generator loss can be better, and the image generated can be better to match the label.



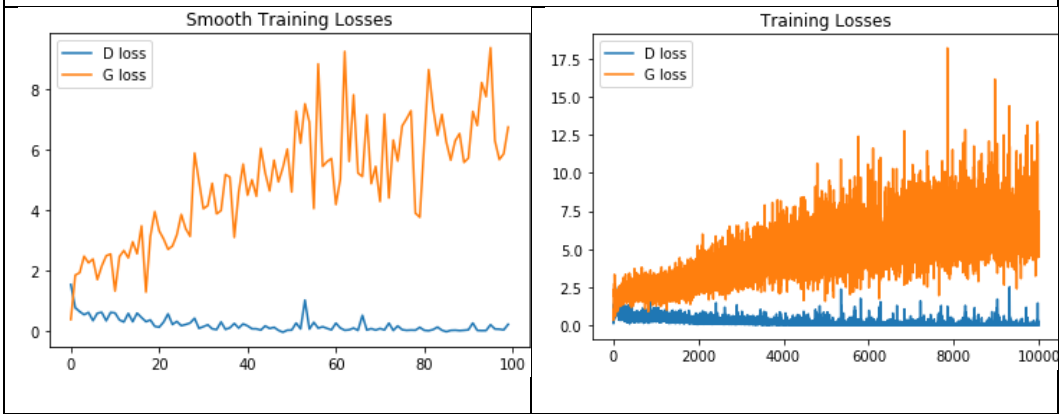
Batch size = 128, discriminator learning rate = 0.0001, generator learning rate = 0.0002, 10000 epochs



Label

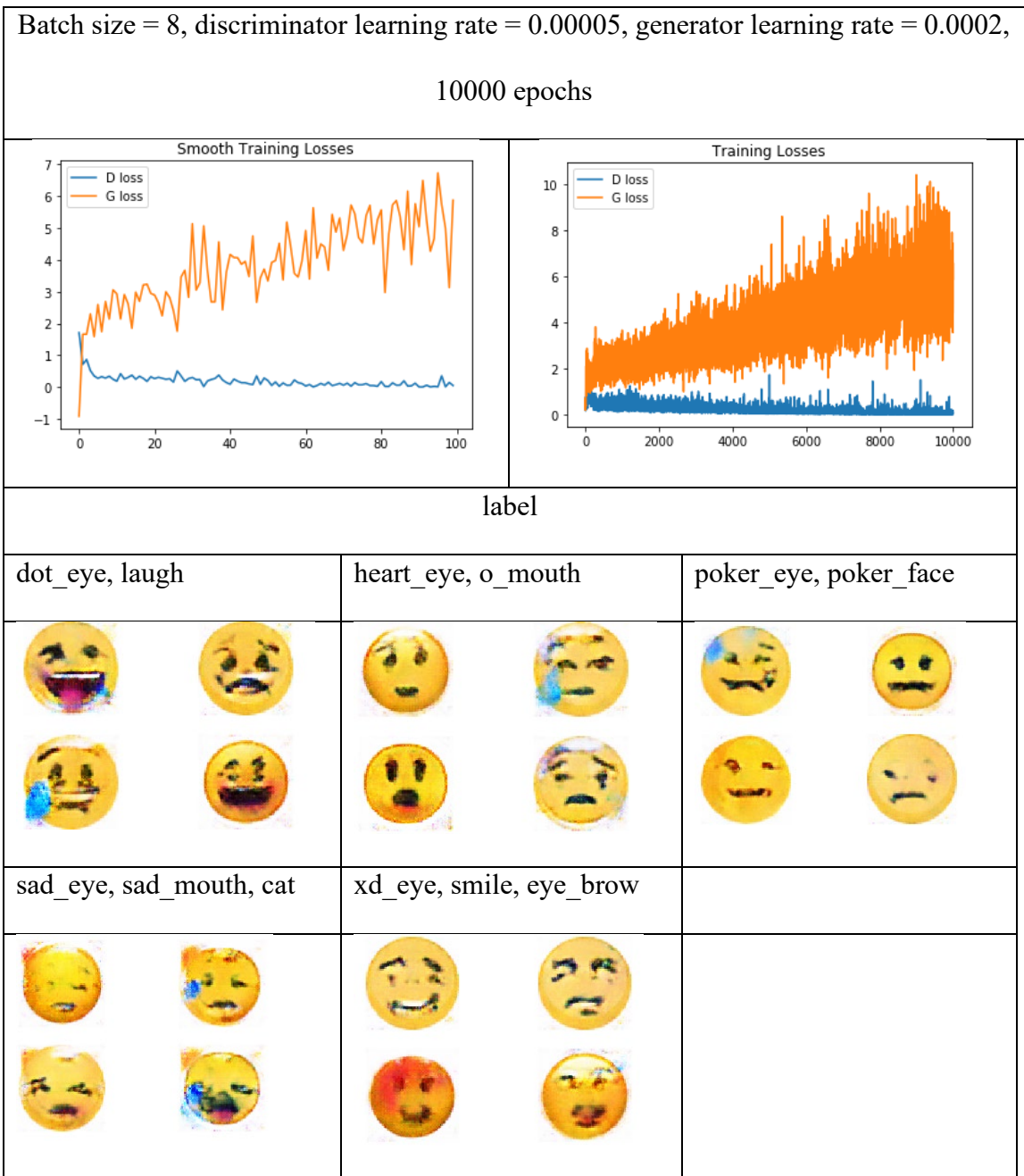
dot_eye, laugh		heart_eye, o_mouth		poker_eye, poker_face	
sad_eye, sad_mouth, cat		xd_eye, smile, eye_brow			

Batch size = 8, discriminator learning rate = 0.0001, generator learning rate = 0.001, 10000 epochs



label

dot_eye, laugh		heart_eye, o_mouth		poker_eye, poker_face	
sad_eye, sad_mouth, cat		xd_eye, smile, eye_brow			

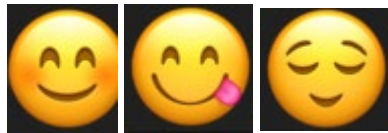


The above result shows that batch size = 8 have a better-generated emoji because those label condition we tested seems to have more correctness than batch size = 128. For example, batch size = 8 can generate sad_mouth as expected but batch size = 128 cannot. Adjusting the learning rate of the generator higher seems to have not

much different while adjusting the learning rate of discriminator lower makes the result worse. For example, the model even cannot generate a cat or heart_eye while others can.

As a result, we can just modify the batch size smaller to make a slightly better result, but the result was still not good enough.

Also, we found out that no matter how we modify the parameters. The generated emoji with labels dot_eye and laugh always can be generated as expected. By checking our inputted emojis, we found out that dot_eye and laugh had many emojis that were labeled by them. Moreover, when we marked the label in the CSV file, we made similar eye/mouth batch together in one label. For example, these















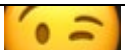










three emojis we all labeled them with a smile, but actually, they have different smiles.









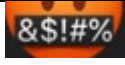







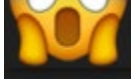


Therefore, we thought that if we increase the number of our training data and make the label more specific, we can make other labels can be generated as expected too.

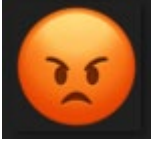



58 conditions CGAN

As a result, in our second try, we designed a total of 58 labels:

Eye			
m_eye	m_eye_small	small_w_eye	heart_eye
			
sunglass	tadpole_eye	bold_one_eye	dot_eye
			
o_eye	XD	eight_eye	cry
			
cat_eye	laugh_tear	twinkie_eye	dash_eye
			
white_eye	x_eye	star_eye	
			

Mouth			
smile	small_smile	l_smile	l_smile_v2
			
laugh	laugh_R	3_mouth	n_mouth

			
small_n_mouth	big_n_mouth	open_n_mouth	o_mouth
			
o_mouth	big_o_mouth	cruse_mouth	dash_mouth
			
small_dash_mouth	ww_mouth	tougue	side_tougue
			
teeth	laugh_teeth	laugh_teeth_full	omg
			
mask	open_small_n_mouth		
			

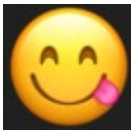
Other			
red_face	eye_brow	nose_water	cat
			
tear	swearing	angle	devil

			
heart	sad	blue_head	sleep
			
Shy			
			

Based on the above labels, we provided a total of 68 different emoji and a total of 331 samples to train the conditional-GAN. Some of the emojis are labeled as the following:



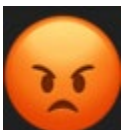
m_eye, smile, shy



m_eye, side_tougue



small_w_eye, small_smile, eye_brow



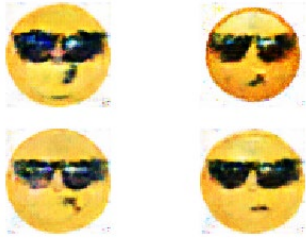
dot_eye, small_n_mouth, eye_brow, red_face

Then, we made a one-hot encoding CSV file that labeled each face emoji one-by-one. Here is part of the original data in the file: (face_emoji/emoji_onehot_v2.csv)

	A	B	C	D	E	F	G	H	I	J	K	L	M
1	emoji												
2	1f60a	m_eye, smile, sh	m_eye	m_eye_small	small_w_eye	heart_eye	sunglass	sperm_eye	bold_one_eye	dot_eye	0_eye	XD	eight_eye
3	1f60b	m_eye, side, touk	1	0	0	0	0	0	0	0	0	0	0
4	1f60c	small_w_eye, str	0	0	1	0	0	0	0	0	0	0	0
5	1f60d	laugh, heart_eye	0	0	0	1	0	0	0	0	0	0	0
6	1f60e	sunglass, smile	0	0	0	0	1	0	0	0	0	0	0
7	1f60f	sperm_eye, eye	0	0	0	0	0	1	0	0	0	0	0
8	1f61a	3_mouth, shy, bo	0	0	0	0	0	0	1	0	0	0	0
9	1f61b	touque, dot_eye	0	0	0	0	0	0	0	1	0	0	0
10	1f61d	touque, XD	0	0	0	0	0	0	0	0	0	1	0
11	1f61e	small_n_mouth, t	0	0	0	0	0	0	0	0	0	0	1
12	1f61f	n_mouth, dot_ey	0	0	0	0	0	0	0	1	0	0	0
13	1f62a	small_w_eye, no	0	0	1	0	0	0	0	0	0	0	0
14	1f62b	XD laugh_R_eyr	0	0	0	0	0	0	0	0	0	1	0
15	1f62c	dot_eye, teath	0	0	0	0	0	0	0	1	0	0	0
16	1f62d	cry, 0_mouth, ey	0	0	0	0	0	0	0	0	0	0	0
17	1f62e	dot_eye, o_mout	0	0	0	0	0	0	0	1	0	0	0
18	1f62f	dot_eye, o_mout	0	0	0	0	0	0	0	1	0	0	0
19	1f63a	cat_laugh, cat_ei	0	0	0	0	0	0	0	0	0	0	0
20	1f63b	cat, heart_eye, le	0	0	0	1	0	0	0	0	0	0	0
21	1f63c	cat, L_smile, cat_L	0	0	0	0	0	0	0	0	0	0	0
22	1f63d	cat, 3_mouth, bo	0	0	0	0	0	0	1	0	0	0	0
23	1f63e	cat, n_mouth, cat	0	0	0	0	0	0	0	0	0	0	0
24	1f63f	cat, small_n_mou	0	0	0	0	0	0	0	0	0	0	0
25	1f63c	red_face, dot_ey	0	0	0	0	0	0	0	1	0	0	0
26	1f600	dot_eye, laugh_th	0	0	0	0	0	0	0	1	0	0	0
27	1f601	m_eye, laugh_la	1	0	0	0	0	0	0	0	0	0	0
28	1f602	m_eye, small_lak	0	1	0	0	0	0	0	0	0	0	0
29	1f603	0_eye, laugh_la	0	0	0	0	0	0	0	0	1	0	0
30	1f604	m_eye, laugh_la	1	0	0	0	0	0	0	0	0	0	0
31	1f605	m_eye, laugh_la	1	0	0	0	0	0	0	0	0	0	0
32	1f606	XD laugh_teath	0	0	0	0	0	0	0	0	0	1	0
33	1f607	m_eye, smile, an	1	0	0	0	0	0	0	0	0	0	0
34	1f608	devil, smile, eye_	0	0	0	0	0	0	0	1	0	0	0
35	1f609	twinkle_eye, sme	0	0	0	0	0	0	0	0	0	0	0
36	1f610	dot_eye, dash_tr	0	0	0	0	0	0	0	1	0	0	0
37	1f611	dash_eye, dash_L	0	0	0	0	0	0	0	0	0	0	0
38	1f612	sperm_eye, smal	0	0	0	0	0	1	0	0	0	0	0
39	1f613	m_eye, small, str	0	0	1	0	0	0	0	0	0	0	0
40	1f614	small_w_eye, str	0	0	0	0	0	0	0	0	0	0	0
41	1f615	dot_eye, L_smile,	0	0	0	0	0	0	0	1	0	0	0
42	1f616	XD_wv_mouth	0	0	0	0	0	0	0	0	0	1	0

We used the same method as 14 conditions CGAN to preprocess the data and input to our model. Here were some results after training 10000 epochs. We found out that some of the results look accurate based on our set condition. For example:

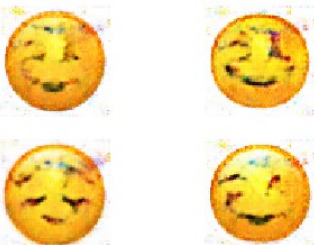
Label: sunglass, 3_mouth



Label: dot_eye, tongue, blue_head



Label: star_eye, smile, shy



We observed that some of the results are not what we expected. After checking our self-made one-hot CSV file, we found out that some of the labels just used on one

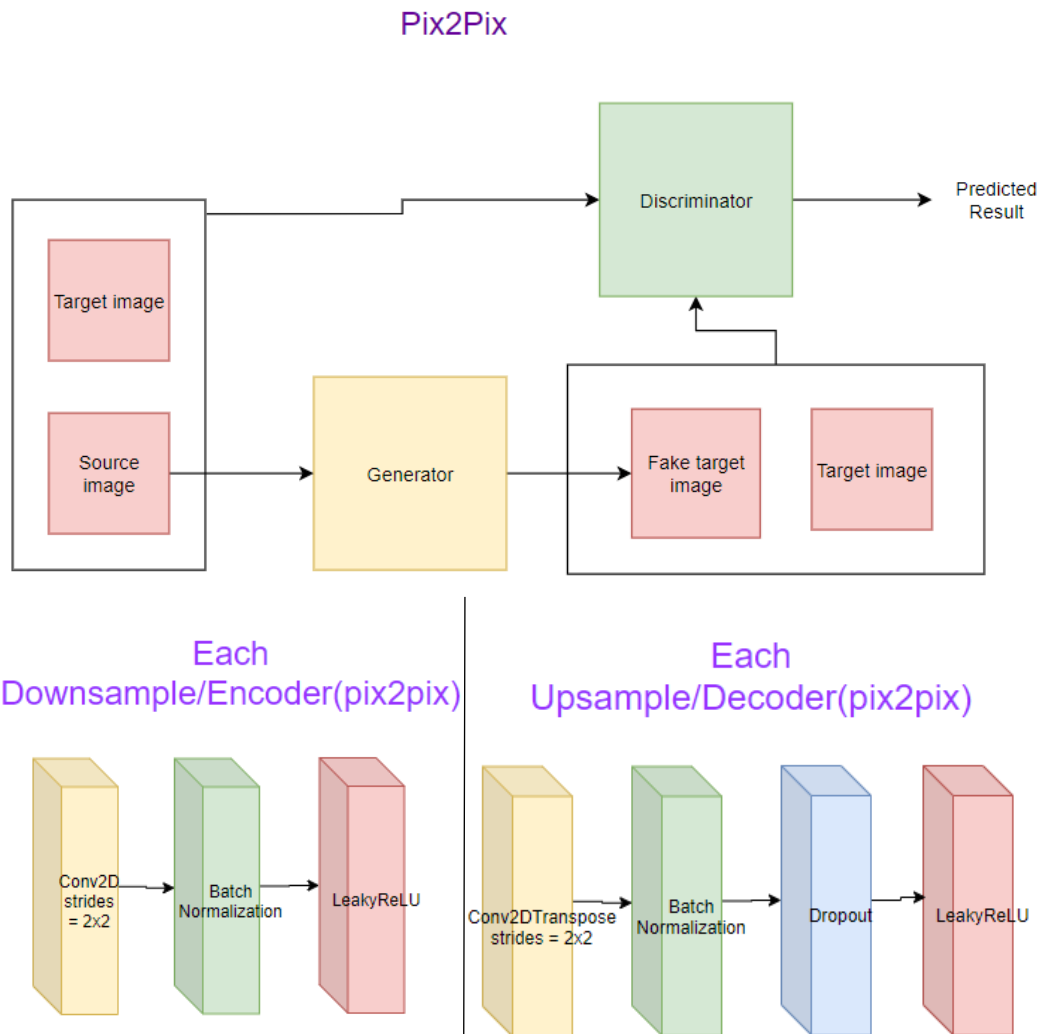


emoji only. For example, star_eye just used on 1f929.png 1f929.png.

Therefore, we got the same problem when we are testing 14 conditions CGAN. We thought that the lack of training data is our leading problem that made our CGAN hard to find the pattern of condition.

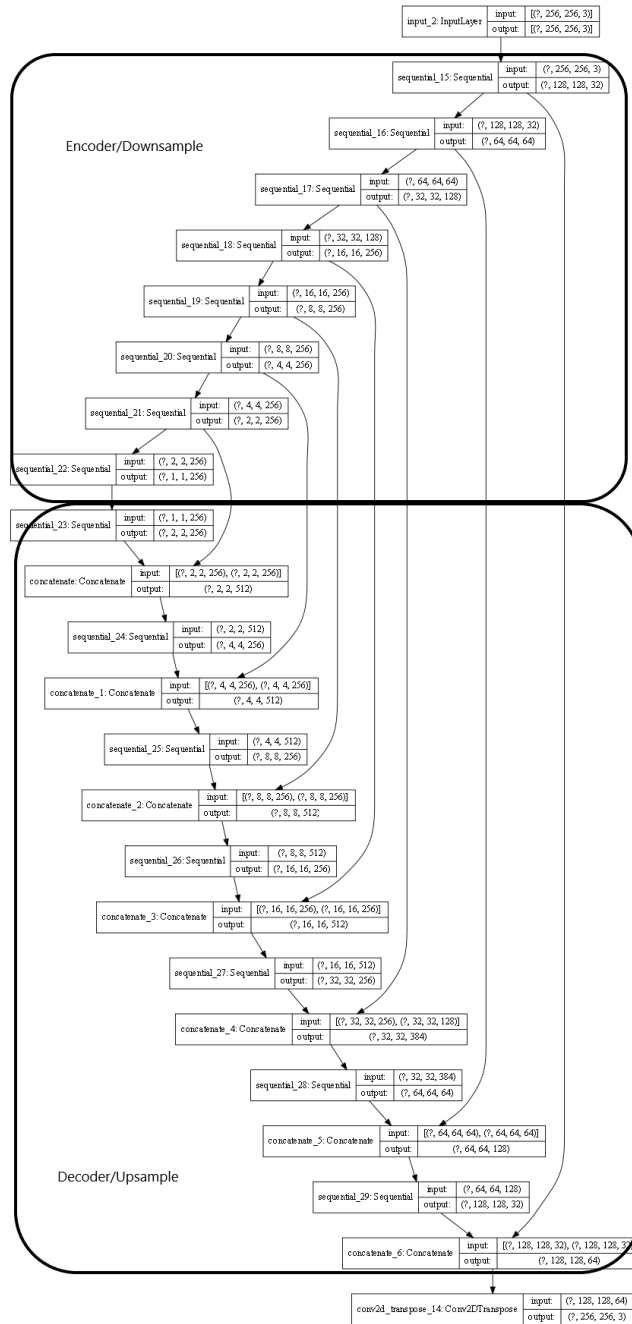
4.3. Overview of Pix2Pix

Based on the knowledge of the CGAN and Pix2Pix. We built the GAN model, as shown. The general structure between CGAN and Pix2Pix is similar except the input of Pix2Pix is an image not a point from the latent space, and the label of Pix2Pix becomes a target image, not an integer array.



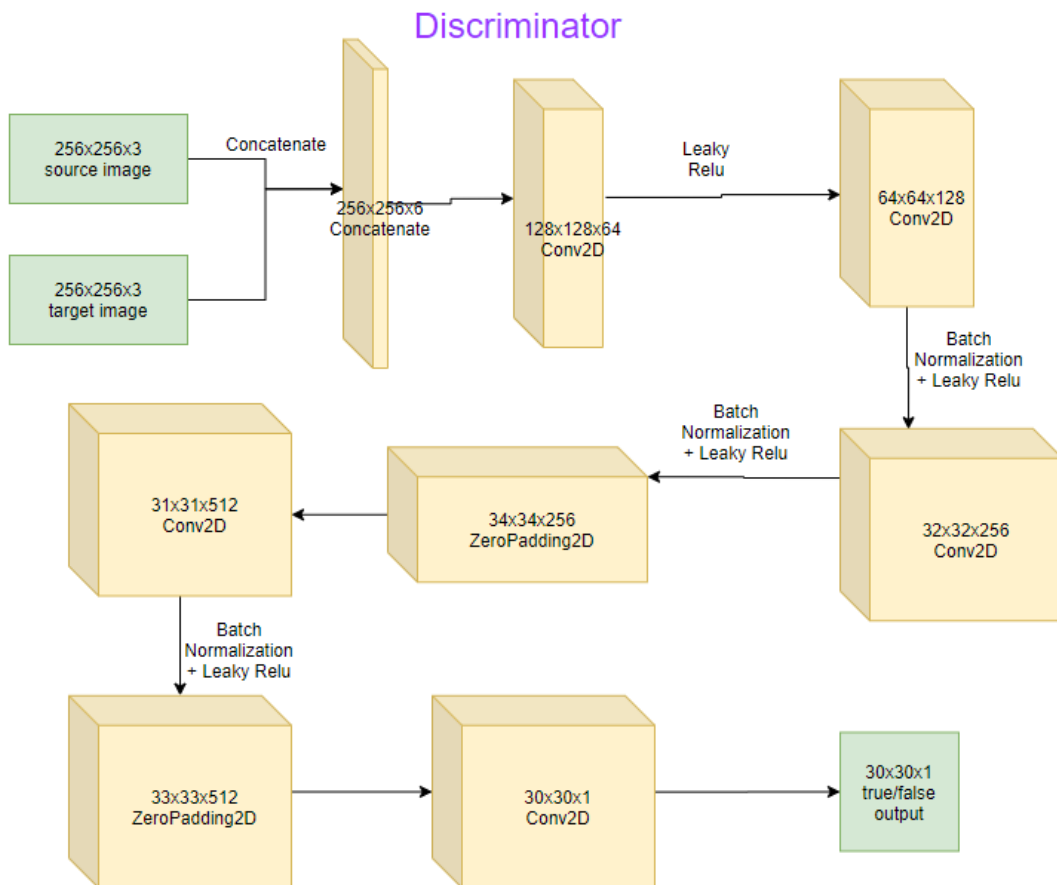
Pix2Pix: Generator

Below is the U-net structure where the first half is the encoder, and the second half is the decoder. The layer with the same output size in the encoder and the decoder has a connection between causing the skip connections.



Pix2Pix: Discriminator

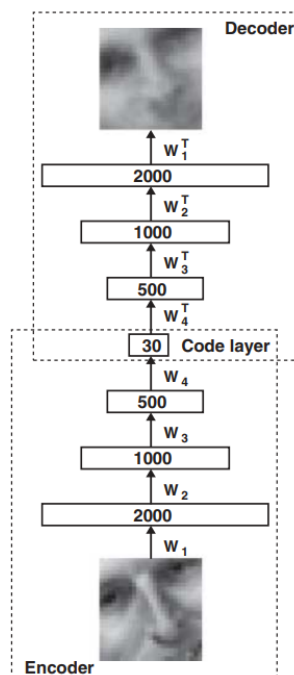
Based on the pix2pix paper (Isola, Zhu, Zhou, & Efros, 2017), the discriminator is a PatchGAN where the output is a (batch size, 30, 30, 1) shape. Each 30x30 patch of the output classifies a 70x70 portion of the input image, which means that we manually chopped up the image into 70x70 overlapping patches, run a regular discriminator over each patch, and averaged the results.



Pix2Pix Methodology

Encoder-decoder network

Based on the paper of the encoder-decoder network (Hinton, 2006). The encoder takes the images as input. Then each layer is gradually downsampled until the last layer, which is the bottleneck layer, and it outputs a feature vector. On the other hand, there is a decoder with the exact network architecture as encoder but in a reverse direction. On the contrary, the decoder tries to output as closely related as the original input as possible.

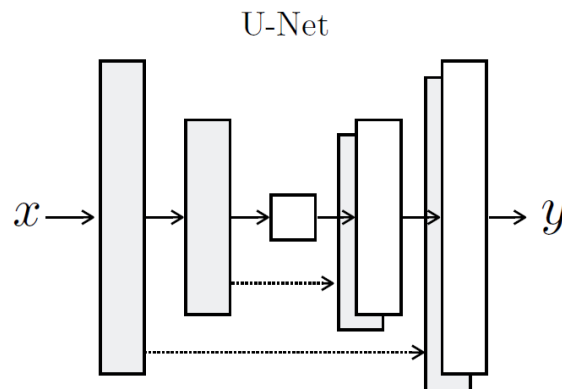


Encoder-decoder. (Hinton, 2006)

Nevertheless, in real-life applications, researchers usually do not use it to reconstruct the original input. Instead, it is used to map and translate to the certain desired output, for instance, highlighting a particular object in an image.

U-net

U-net is the generator in Pix2Pix adopts (Ronneberger, Fischer, & Brox, 2015). In essence, U-Net is an improved version of encoder-decoder. U-Net has the same architecture of encoder-decoder but with the addition of skip connections. Isola et al. (2017) stated that it is specially picked to handle translation problems, “there is a great deal of low-level information shared between the input and output, and it would be desirable to shuttle this information directly across the net.” As mentioned in the previous paragraph, encoder-decoder has a bottleneck layer in the middle, and some information might be lost during the process.



U-Net. (Isola, Zhu, Zhou, & Efros, 2017)

In the above figure, the layers with the same size in encoder and decoder are linked together. Those links allow the information to circumvent the bottleneck. Because of this, U-Net yields a better result than encoder-decoder as tested in the author’s paper.



U-Net produces a much better result. (Isola, Zhu, Zhou, & Efros, 2017)

PatchGAN

Isola et al. introduced a new network called PatchGAN to enhance the discriminator. In a normal GAN discriminator, a deep convolutional neural network is used to do the classification. As the name suggests, the PatchGAN classify patches of the input images as fake or real instead of the whole image. In detail, the image is split into $N \times N$ patch or grid. The discriminator will determine each patch as fake or real convolutionally. The output is a feature map that consists of predictions that can map to a specific size of the source image. Then, averaging all the patches yields the final output of the discriminator. The advantage of PatchGAN is that it can take arbitrary sized images as input. Besides, Isola et al. (2017) found that a 70×70 patch size performed well across various image-to-image translation problems.

Loss function

Adversarial loss (MiniMax Loss)

It is similar to the original GAN loss function, but some of the input parameters is different.

$$E_x[\log(D(x, y))] + E_y[\log(1 - D(x, G(x)))]$$

x is the real input image

y is the ground truth/target image

$D(x, y)$ is the probability of the discriminator's estimate the real image x and ground truth image pair is real

$G(x)$ is the generator's output when given image x

$D(x, G(x))$ is the probability of the discriminator's estimate the fake image $G(x)$ and ground truth image pair is real

E_x is the expected value of all the real images and ground truth image pair

E_y is the expected value of all the fake images and ground truth image pair

The generator aims to minimize this function while the discriminator aims to maximize this function.

Pix2Pix Result

As the training of the pix2pix needs to match the same image of the source image and the target image (ground truth). We made a program to combine two same Chinese characters of different font/handwritten words to one image. Here are some examples of the images:

(the left character is the ground truth; the right character is the source image)



Combined image with handwritten image as input, hzwong as ground truth



Combined image with DFKai-SB image as input, hzwong as ground truth

Then we put these images into the train folder for training the network and the test folder for output some samples for us to evaluate.

Our first training is using DFKai-SB font as the source image and MingLiU font as ground truth to test our model is really working as these two fonts look very similar in style. Here is some result after 50 epochs with batch size 1: (left image is the source image, the middle is ground truth and the right is generated image)

糴 糴 糴 笈 笈 笈 答 答 答

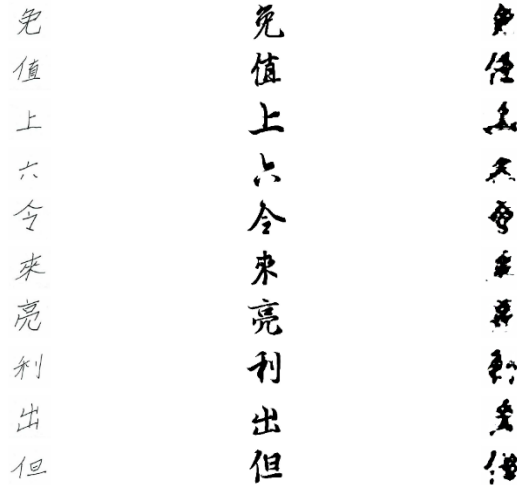
We can observe that the generated image is very similar to the ground truth. It shows that our network is working.

Then we use DFKai-SB font as the source image and hzwong font as ground truth to put into the same network to test our model, and the result is like this:

港	港	港
千	千	千
星	星	星
甚	甚	甚
牛	牛	牛
這	這	這
矛	矛	矛
火	火	火
庫	庫	庫
級	級	級

By just observing the photo, we can notice that the generated image become not similar to the ground truth, and some of the generated images cannot recognize the character such as 「牛」 「級」.

When we use our handwritten word as the source image and hzwong font as ground truth. We cannot recognize any of the generated image's Chinese characters. The result gets worse than before:



免 值 上 六 令 來 亮 利 出 但

免 值 上 六 令 來 亮 利 出 但

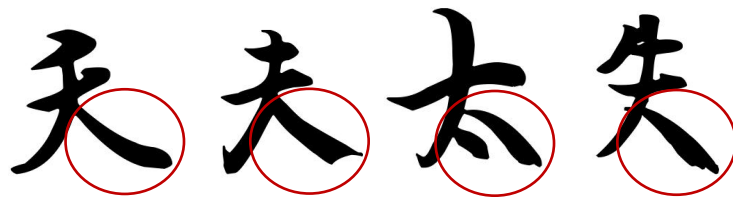
免 值 上 六 令 來 亮 利 出 但

Problem of Pix2Pix

We think the problem is about the similarity of different Chinese characters in the same font. For the font like DFKai-SB and MingLiU. Their structure of the Chinese characters is precise and rigid. For example, the Chinese characters of DFKai-SB 課誰. We can notice that the radical 「言」 of these two images are in exactly the same place. Moreover, their stroke (e.g. 橫 豎 撇 捺 點) on different Chinese characters looks the same on their font. For example, in the below figure, these 4 Chinese characters in DFKai-SB font have four same stroke styles of 捺.



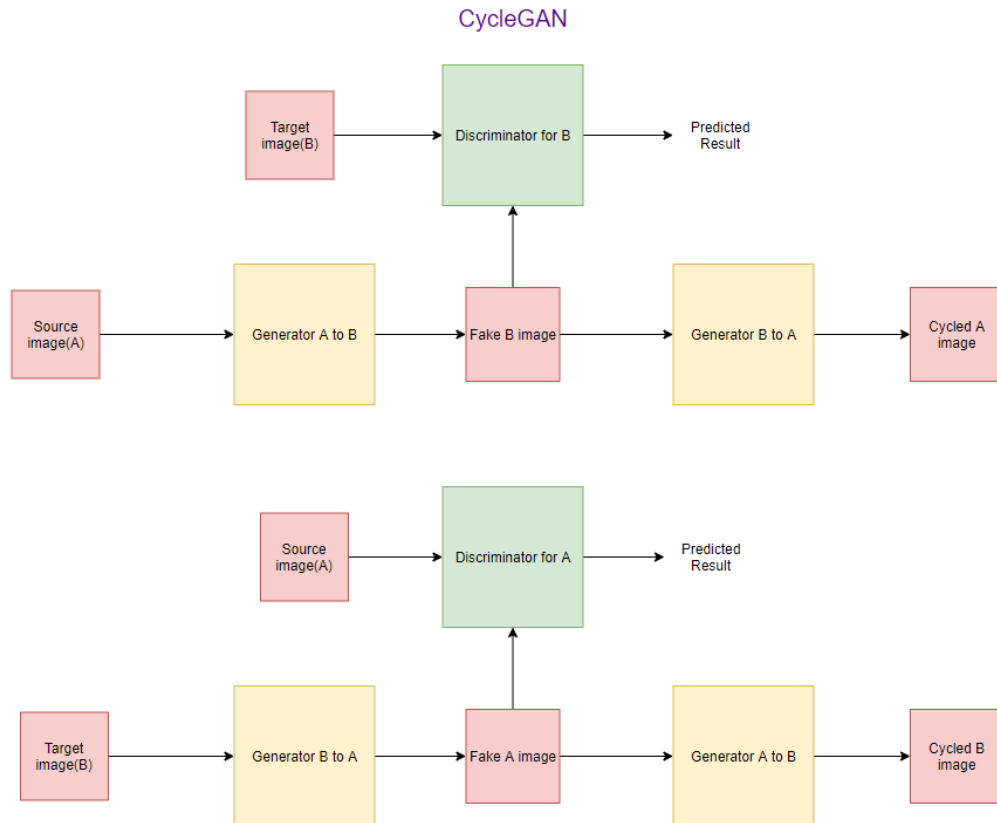
In the case of hzwong font and handwritten Chinese character, because both of them are handwritten characters. We cannot make the character precise and rigid on the pixel level. So it causes each character to have a slightly different style, but the model cannot generalize the difference. For example, these 4 Chinese characters in hzwong font have four different stroke styles of 捺.



As a result, we need to make the model more general to learn the style of the font. We need to make the model not learning by the one-to-one Chinese character font, but learn by all the Chinese characters of the font. That means we need an unpair image for training.

4.4. Overview of Cycle-Consistent Adversarial Network (Cycle GAN)

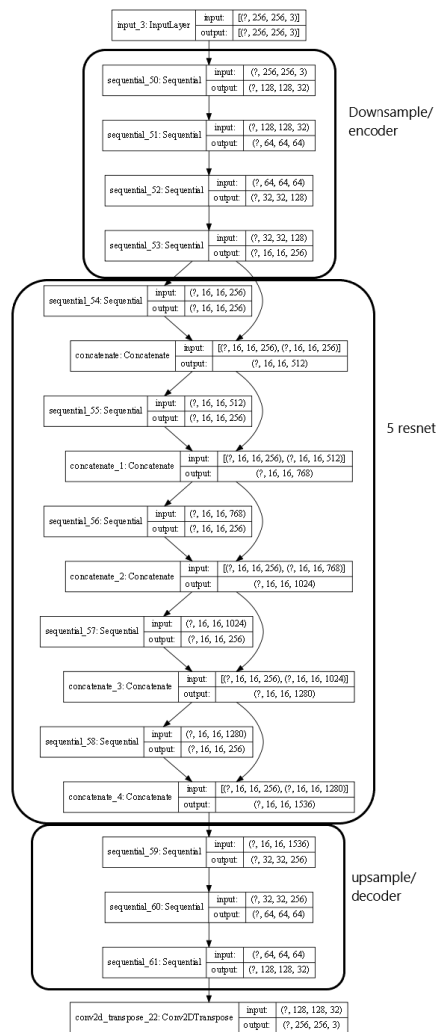
We based on the Pix2pix network and followed the idea provided by Cycle-Consistent Adversarial Networks (Zhu et al. 2017) to create our model.



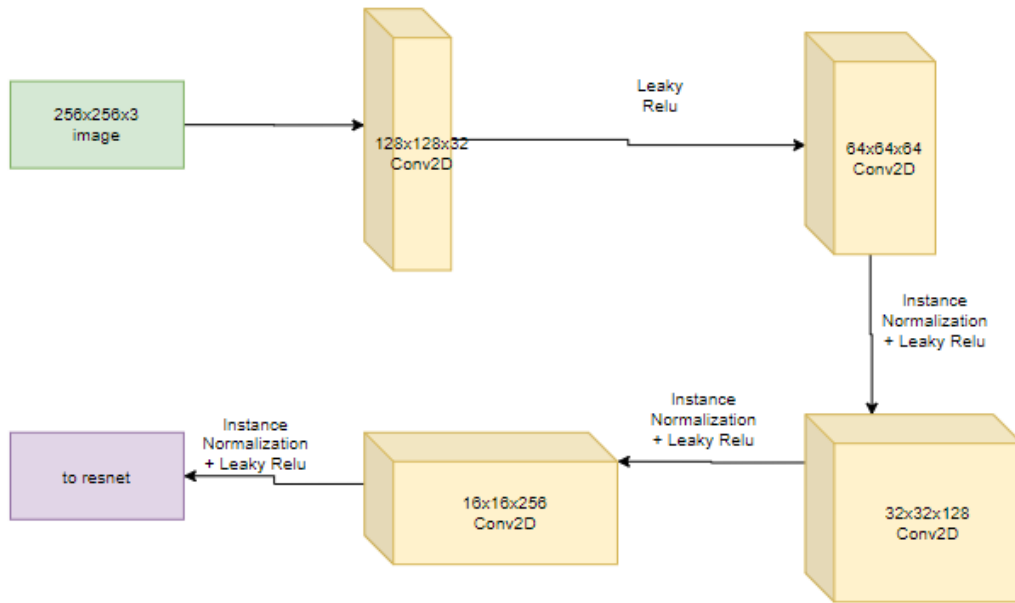
Model design

Cycle GAN: Generator (5 ResNet version)

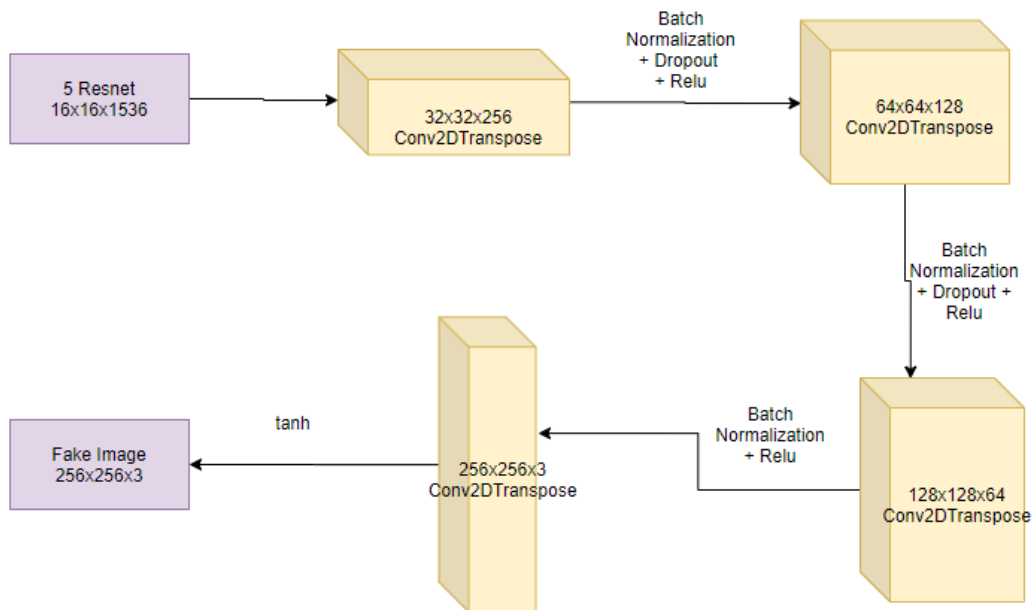
The first part of the generator is the encoder, which will extract the features from the image. Then it will go to the transformation part, which is the ResNet to combining the feature of 2 different font/handwriting. The last part is the decoder, which reconstructs the image by the feature and the transformed feature of the image as output.

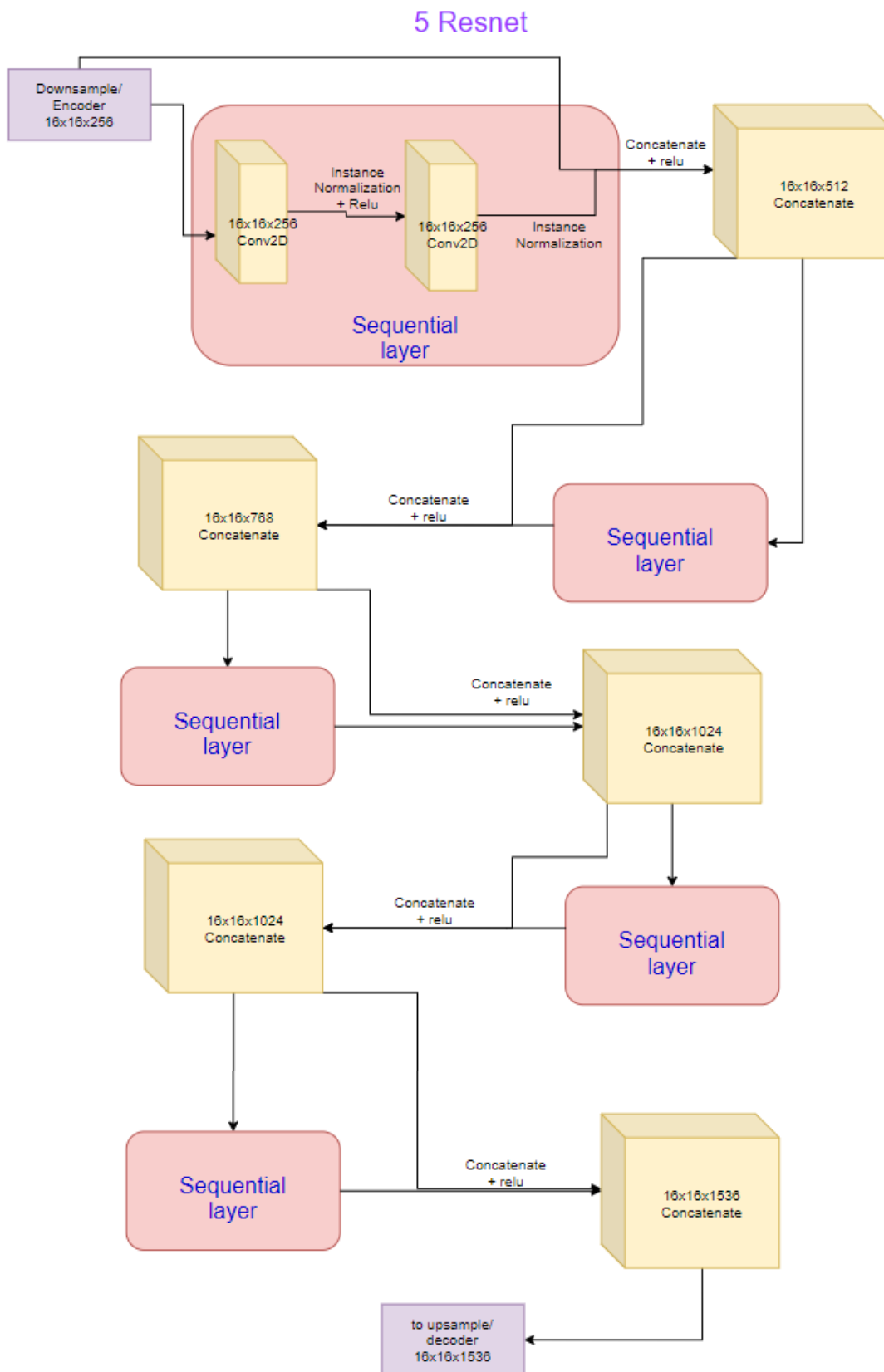


Downsample/Encoder(ResNet ver)



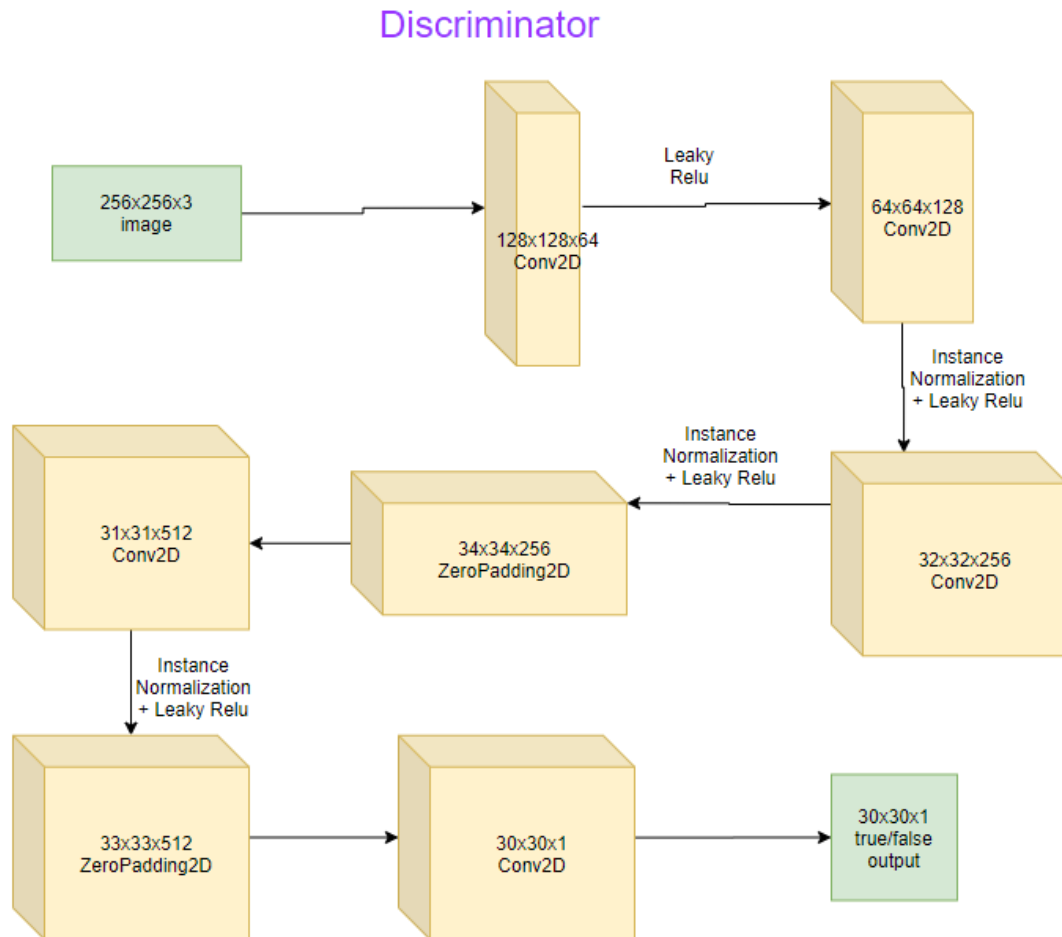
Upsample/Decoder(ResNet ver)

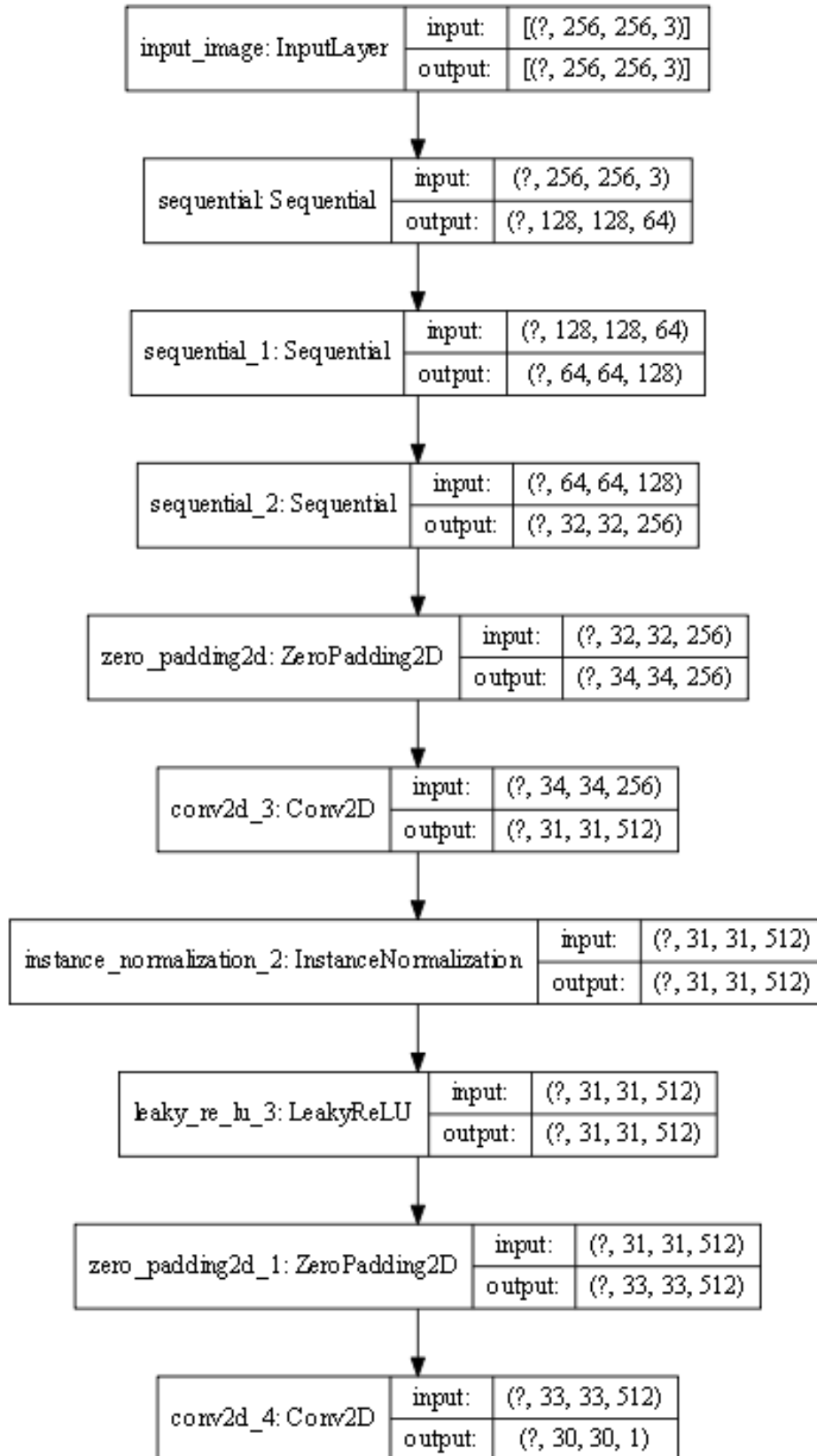




Cycle GAN: Discriminator

The architecture of discriminator is nearly the same as Pix2Pix's discriminator, except we do not have the target image as the second input.



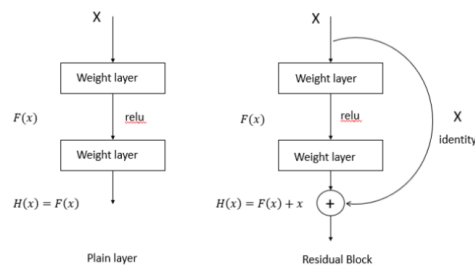


Cycle GAN Methodology

Residual block (ResNet)

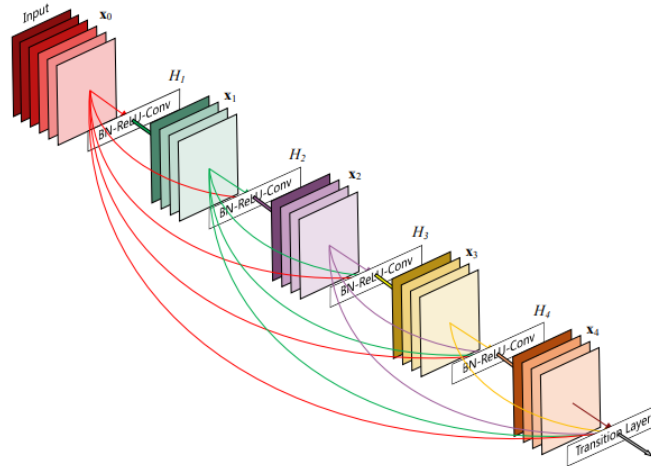
According to the cycle GAN paper, when they build the generator, they use several residual blocks as a transformer between encoder and decoder. According to the Deep Residual Learning paper (He, Zhang, Ren, & Sun, 2016). Residual block is creating a short cut connection on the plain network. One of the functions is to ensure the properties of input of previous layers are available for later layers as well. In handwriting style changing, we need to keep the handwritten Chinese character in its own shape when passing through each layer of the generator, making the output image will not be extremely different from the original input.

Another function is that when the training progress is nearly saturated, it will cause the identity mapping problem, which means that the layer's parameter will not change scientifically, making the accuracy of the model decrease. As the property of backpropagation is started tuning the layer near the output layer, the higher layer of the model cannot be well trained. ResNet can train the model better because the skip connection makes the backpropagation process can train the higher layers even through, the lower layers are well trained.



A plain layer compares to the residual block (Zhang, Ren, Sun, & Jian. (2015))

DenseNet



5-layer DenseNet Block (Huang, Liu, Maaten, Weinberger. 2018)

The paper *Densely Connected Convolutional Networks* (Huang, Liu, Maaten & Weinberger, 2018) suggest a new network architecture that can improve information flow between layers with different connectivity pattern. It is similar to ResNet but more skip connection. The concept is that the layer will have a direct connection to all subsequent layers to implement feature reuse and increase the efficiency to train the network.

Loss function

For discriminator, we use the same loss function of Pix2Pix. For generator, include the similar adversarial loss from Pix2Pix, we add one more loss function that is called cycle consistency loss.

Adversarial loss

$$E_y[\log(D_Y(y))] + E_x[\log(1 - D_Y(G_{x \rightarrow y}(x)))] \leftarrow$$

$$E_x[\log(D_X(x))] + E_y[\log(1 - D_X(G_{y \rightarrow x}(y)))] \leftarrow$$

the first formula is generator x to y and its discriminator D_Y

the second formula is generator y to x and its discriminator D_X

x is the real input image (style1)

y is the target image (style2)

$D_X(x)$ is the probability of the discriminator's estimate the real image x is real

$D_Y(y)$ is the probability of the discriminator's estimate the real image y is real

$G_{x \rightarrow y}(x)$ is the generator x to y 's output when given image x

$G_{y \rightarrow x}(y)$ is the generator y to x 's output when given image y

$D_X(G_{y \rightarrow x}(y))$ is the probability of the discriminator's estimate the fake image

$G_{y \rightarrow x}(y)$ is real

$D_Y(G_{x \rightarrow y}(x))$ is the probability of the discriminator's estimate the fake image

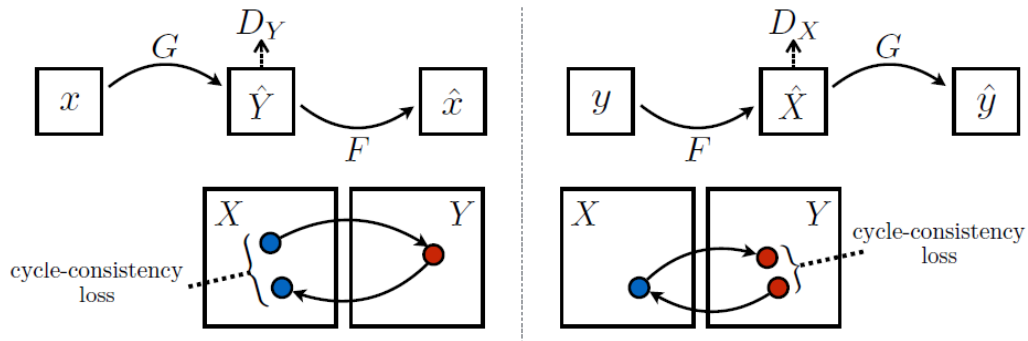
$G_{x \rightarrow y}(x)$ is real

E_X is the expected value of real images x or fake images x

E_Y is the expected value of real images y or fake images y

Both generators aim to minimize the function while both discriminators aim to maximize the function.

Cycle Consistency Loss



This model does not need to pair the image data to do the style transformation. Cycle consistency loss is the supervising signal for the model training. This concept explains that when a source image A (x) transform to image B ($G(x)$) by the generator A to B (G) then use the generate image B ($G(x)$) transform back to image A ($F(G(x))$) by the generator B to A (F). In theory, the source image A (x) and the cycled image A ($F(G(x))$) should be looking the same. Therefore, when training the model, the difference between the source image A and the cycled image A and the difference between the target image B and the cycled image B will be the loss of both generators. In math formula, it will look like this:

$$L_{cyc-x} = |x - F(G(x))|$$

$$L_{cyc-y} = |y - G(F(y))|$$

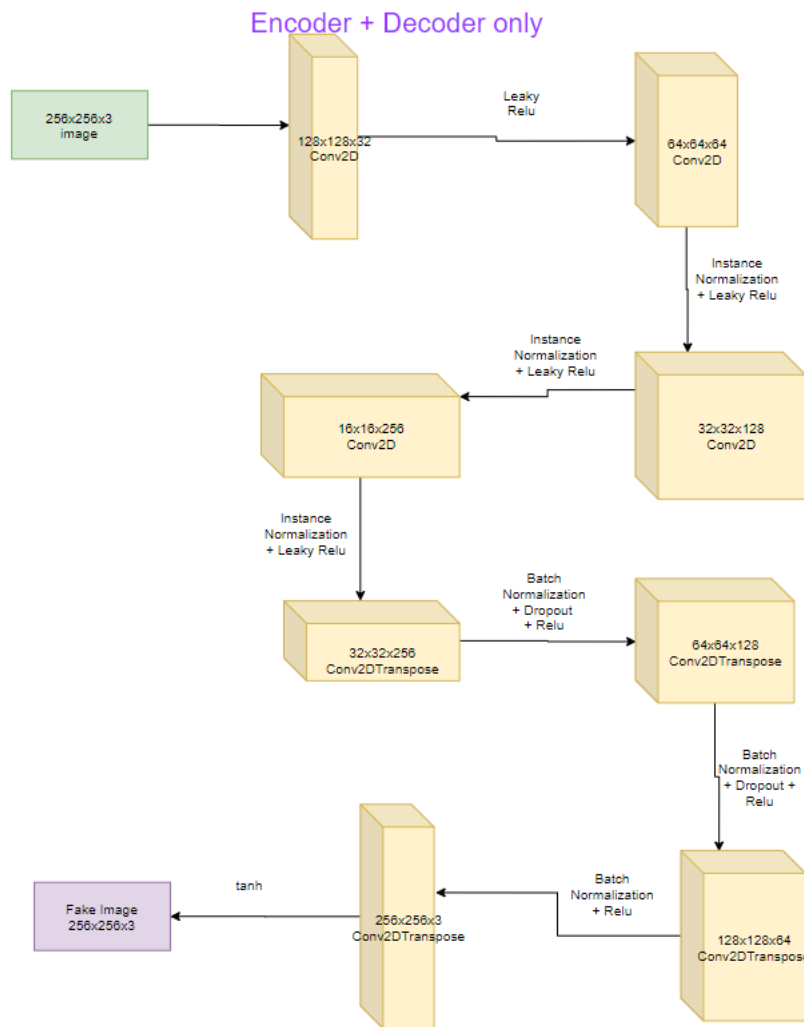
$$L_{cyc-total} = L_{cyc-x} + L_{cyc-y}$$

Therefore, the total loss of the generator = generator's adversarial loss + Cycle consistency loss.

For all the results below. Unless further specify some parameters, we are using 15 epochs for training, batch size = 1, learning rate = 0.0002 with Adam optimizers, and using the same discriminator. Furthermore, those images show below are all unseen images when training the network.

Cycle GAN Encoder + Decoder Only Version Result

To test our Cycle GAN training progress is working. We implement the basic encoder-decoder only generator. Moreover, as the baseline for further improving our generator comparison.



存	存	春	春
張	張	絲	絲
白	白	亮	亮
農	農	湖	湖
遠	遠	概	概

We can observe that by just using the encoder-decoder architecture, the generated image can learn some of the styles from hzwong (marked as red circle).

Cycle GAN U-net Version Result

Next, we are trying to modify the generator of Cycle GAN to the U-net structure, just like Pix2Pix's generator. As mention above, the U-net structure also using the similar architecture of encoder-decoder except for the encoder, and the decoder will down/unsampled until the bottleneck layer reached, and it added the skip connection to directly pass the feature from the input image between encoder and decoder.

As a result, we move the generator of Pix2Pix and port into Cycle GAN's generator and use our handwritten and hzwong font with the same setting to train the network.

The result as follow:

(left side is the handwritten input image; the right side is the generated image)



農 農 湖 湖
遠 遠 概 概

By just observation, we can see not much different compare with the encoder-decoder only network result.

In addition, we tested on the characters from Mr. Chen Chung Chien and our other handwritten style on a network trained with 16 batch size and 100 epochs.

Input	Predicted	Input	Predicted
句	句	句	句
醫	醫	厭	厭
之	之	士	士

From the above result, we can observe that the overall structure of the characters is generated. Nevertheless, in some complicated words (second row), the lines are

tangled. Still, we can see some characteristics in Chinese calligraphy. In 橫畫 and 豎畫, the network utilizes the concept of 起筆 and 收筆¹³.

Cycle GAN Resnet Version Result

This time, we use ResNet block to the generator mention on the network architecture above, but we use three blocks of ResNet first. We use DFKai-SB font and hzwong font to test our network is working. The input image below is all unseen images.

(left side is the DFKai-SB font input image; the right side is the generated image)



¹³ <http://163.20.160.14/~ntc/mod/page/view.php?id=22>

聲 聲 至 至

As we can observe clearly, the input image with DFKai-SB font and the generated image with hzwong font is nearly identical. Cycle GAN does not need to pair the image data between 2 fonts. There is no ground truth as the target image to move the shape of the Chinese character, so the stroke placement in the image are the same. Moreover, if we observe carefully, we can observe that the style has changed a little that looks like hzwong font. As a result, it is hard to tell that our network is working or not.

Therefore, in our second approach, we use our handwritten and hzwong font to train the network with three blocks of ResNet. The result is as follow:

(The left side is the handwritten input image; the right side is the generated image)

存 存 春 春
張 張 絲 絲
白 白 亮 亮



This time we can clearly see that the style changing of the handwritten Chinese character. We can observe that some styles of hzwong font can be learned (mark as a red circle on the image). However, we notice some problems that if the strokes of the input image are really close to each other, the generated image's strokes will overlap with each other (mark as a green circle on the image). But the overall result is still acceptable.

So we try to add more blocks of ResNet to check if it has any improvement. Below is the 5 ResNet block of the generator with unseen image:

(The left side is the handwritten input image; the right side is the generated image)





We can notice some improvements in the generated Chinese characters. The shape of the image is clearer. For example, the dot of the character 「絲」, the bottom right part of the character 「張」. They look sharper and show the style of hzwong font.

Cycle GAN ResNet + U-net Version Result

While the generator of the ResNet version can produce more similar images to the ground truth, and the U-net can also produce similar results. However, the ResNet version generator has one problem. Some features of the input image may disappear while passing through the encoder. Therefore, we want to combine those two types of methodologies to create a new generator. The idea is simple; we based on the ResNet version generator and added the skip connection between encoder and decoder so the feature of the input image will not lose during the encoder part of the generator. As we think that the skip connection can help to do some of the transformation processes and improve the generated image result.

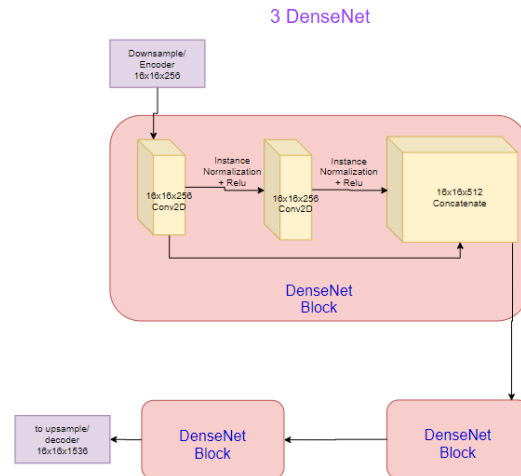
(The left side is the handwritten input image; the right side is the generated image)

存	存	春	春
張	張	絲	絲
白	白	亮	亮
農	農	湖	湖
遠	遠	概	概

We can observe a significant result that is the Chinese characters 「農」. We can see every stroke clearly, and the style changed comparing to other architecture above. Moreover, we can also observe that the dot of 「絲」 is much brighter than the other two.

Cycle GAN DenseNet Version Result

Lastly, we implement the three blocks of the DenseNet version of the generator to test the result.



(The left side is the handwritten input image; the right side is the generated image)

存	存	春	春
張	張	絲	絲
白	白	亮	亮
農	農	湖	湖
遠	遠	概	概

The result shows that it is not much different compare to the ResNet version. We think that maybe our DenseNet layer is not enough to train through the characteristics of the font. Therefore, we add more DenseNet blocks to the generator to train. However, because of the dense shortcut connection. The channel of the layer becomes larger and larger; it needs to read the memory more frequently. Causing the cuDNN always crashes due to memory reason, so we cannot see the result.

Non-Chinese character Result

As Cycle GAN is mainly learning the style between 2 font but it does not learn what is the word exactly. Therefore, we try to input some non-Chinese character into the trained network to show that the style is also working on other language even though the network never seen those shape of the character.

(The left side is Korean character, the right side is Japanese character)

Using 5 Resnet version generator result:

가	가	は	は
금	금	ん	ん
임	임	せ	せ

근 근 の の
장 장 あ あ

Using ResNet and U-net version result:

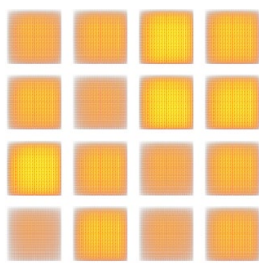
장 장 ん ん
가 가 せ せ
임 임 あ あ
근 근 は は
금 금 の の

We can observe that those non-Chinese characters can generate the image with the hzwong font style. It shows that our network is general enough to learn the font and apply it to other language's characters.

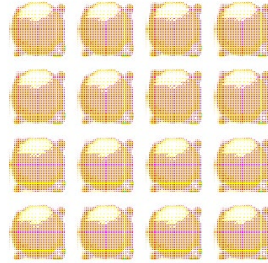
5. Difficulties and Solution

The transparency layer of the image

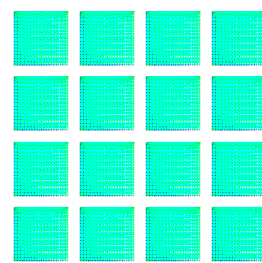
All the images we collected have four channels – RGBA, when we input those four layers as input and train for a long period of the epoch, but the GAN still cannot find the pattern of those images. It keeps change the background color for a long epoch until around epoch 3000. We can finally observe some shape of the emoji. It is wasting time and resources to train.



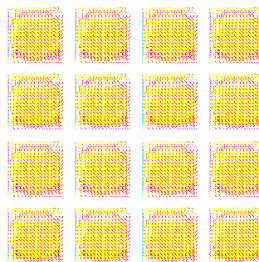
Epoch 30



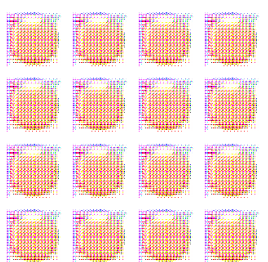
Epoch 465



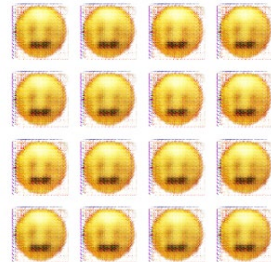
Epoch 960



Epoch 2055



Epoch 2550



Epoch 3235

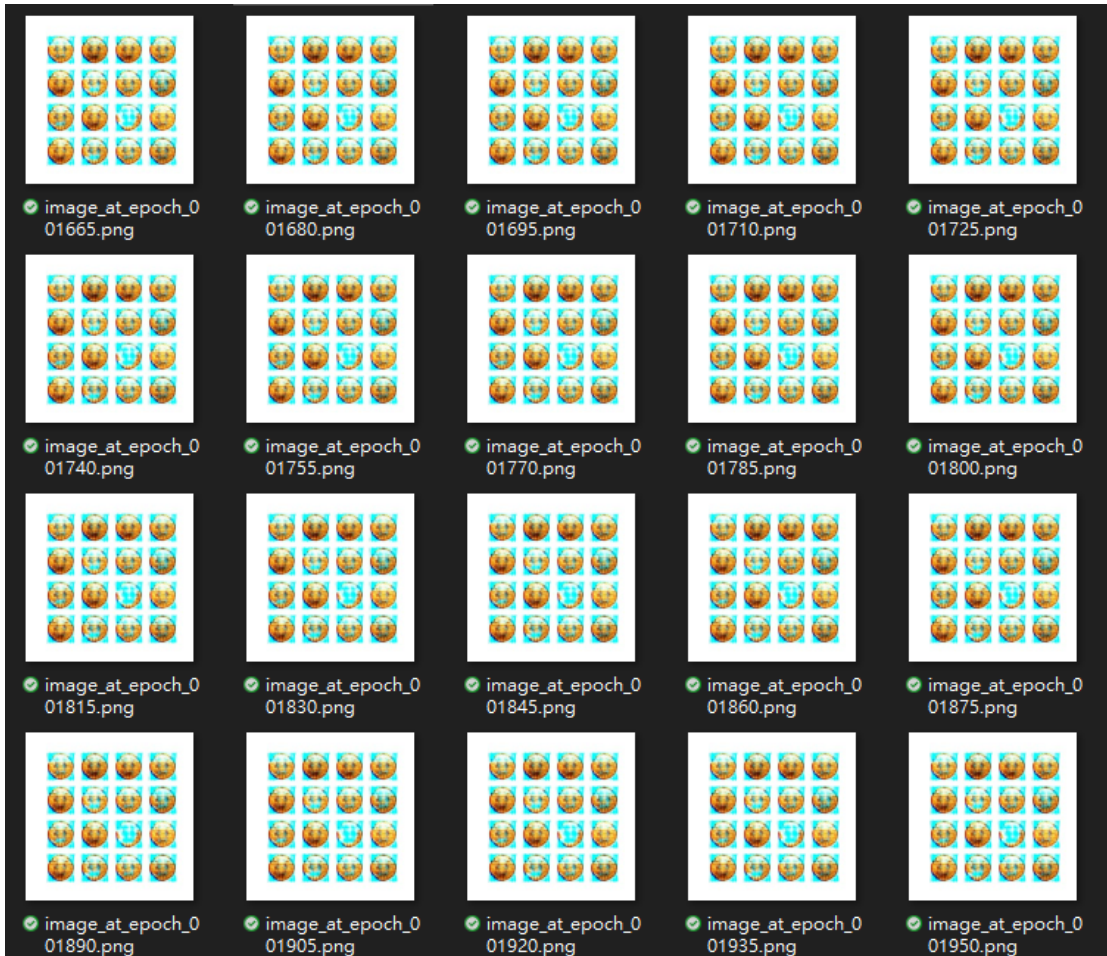
As a result, we decide to remove the transparency layer for training. Although the results may not as beautiful as the image contain transparency, the training time is much faster.

Either generator or discriminator is too good

GAN has two separate training networks; we need to handle those two networks not to defeat the other one entirely.

If the generator is too good, the discriminator needs to guess the result randomly. It makes the feedback meaningless, causing the training becomes random, making the quality of the image becomes worse over time.

If the discriminator is too good, the loss value provides to the generator become nearer and nearer to 0, making the generator stop training as the gradient is 0.



The generator has no progress after a long period, and the loss value of both generator and discriminator does not change.

Therefore, we need to balance the training speed of 2 networks by modifying the learning rate, add more noise in one of the networks, etc. Also, Wasserstein loss is designed to prevent discriminator is too good as the maximum output value of the discriminator is not 1. It can be infinity, so it can still provide some information to the generator for training.

Lack of Data

Because of the lack of data, we were investigating the amount of data required to train a promising network. We tried to use different amounts of data from Fashion MNIST to train a GAN in 100 epochs. With training data size 10000, we start to retrieve a more realistic photo. However, it is worth noting that the images are only 28 x 28 dimensions and in grayscale. It means that if the dimension goes up, more training data than 10000 may be needed. Below are the results.



Training Size 2000



Training Size 5000



Training Size 10000



Training Size 20000

Future Work

In this section, we provided some of the possible research that can be done.

Learning the overall structure of Chinese character and font style without supervised learning

Pix2Pix network can learn both the font and the position of the Chinese character stroke, but it needs both input and ground truth to be similar, and the dataset needs to be precise and rigid to produce a quality image. Cycle GAN can learn the font style and apply it to the input image, but it is unsupervised learning. Therefore, Cycle GAN can just learn the characters of the font, but it cannot learn the structure of the Chinese character. We hope that is a new method of network layer or model and learn both font and the label about Chinese characters without supervised learning.

Learn multiple font styles in one network

In our case of CycleGAN and Pix2pix, we can just build the one-to-one pair font style changing. If we want to change it into other fonts, we need to train a new model to fit the condition. We can try to use the category embedding method to create an embedding space to fit many styles into the embedding space.

Solve real-life problem

As mentioned in the introduction, past calligraphers have their work related to ancient Chinese characters. We can utilize GAN to extract their style and generate their style in Chinese characters that is more common in the modern world. Besides, ancient Chinese used Traditional Chinese. We can even translate to style to see

what it looks in Simplified Chinese. Furthermore, there are Asian countries that have a history of calligraphy too, such as Japan and Korea. It will be interesting for us to use these countries' calligraphy style and generate in Chinese characters or vice versa.

Division of labor

This project is divided into two main parts, one is related to the emoji generation, and the other part is focusing on the Chinese character font style changes. In this FYP course, we did the emoji generation in the first semester and did the Chinese character font style changing in the second semester. The following table summarizes the labor of each of us:

Semester	CHOI Ki Fung	Tsang Ka Hung
1 st Semester Emoji Generation	<ul style="list-style-type: none"> • Modify and test GAN, DCGAN • Data gather and preprocessing • Label the emoji • Create and process one-hot encoding of emoji label • Create concept illustration graph • Proofread and formatting 	<ul style="list-style-type: none"> • Build and test the GAN and DCGAN, CGAN • Test different loss function • Analyze the generated emoji by eye observation and loss comparison • Create the network structure graphs
2 nd Semester Font Style Changing	<ul style="list-style-type: none"> • Build and test Pix2Pix • Web scraping • Data gather and preprocessing 	<ul style="list-style-type: none"> • Font preprocessing • Preprocessing Pix2Pix model input preprocessing • Modify and test Pix2Pix

	<ul style="list-style-type: none"> • Create self-written handwritten images • Proofread and formatting 	<ul style="list-style-type: none"> • Build and test different version of Cycle GAN (ResNet, ResNet + U-net, DenseNet) • Create self-written handwritten images • Create the network structure graph
--	--	--

The contribution of the report is summarized as follows:

Title	Choi Ki Fung	Tsang Ka Hung
Introduction	3 pages	
Background	7 pages	
Data Gathering	3 pages	3 pages
Data Preprocess	5 pages	4 pages
Generative Adversarial Network (GAN)		7 pages
Conditional Generative Adversarial Network (CGAN)	6 pages	17 pages
Pix2Pix	3 pages	8 pages
Cycle-Consistent Generative Adversarial Network (Cycle GAN)	2 pages	21 pages

Difficulties and Solution	3 pages	3 pages
Future Work	2 pages	1 page
Total Page Count	34 pages	64 pages

Contribution Detail

My contribution to this project is mainly data gathering and preprocessing.

Data gathering is the process of obtaining raw data from the Internet or the real world. For the first term, I have to search for any existing datasets that are useful for us, which is the ideal situation. Then, if those datasets do not exist, I need to gather those sparse data from the Internet. There are tons of resources on the Web; I need to screen out those unqualified data, such as images with watermark, inconsistent images, licensed images, and so on. Then, after filtering out a useful dataset, I have to scrape the data in an automated process while categorizing them. Regarding manual work, I labeled the description of the emojis for the CGAN and written Chinese characters for the second semester.

Data preprocessing can be divided into Data Cleansing and Data Wrangling. Data cleansing is to eliminate incomplete, inaccurate, incorrect, and irrelevant data. For example, there are duplicated emoji and emoji with modifiers (skin tone modifiers and sexuality modifiers); there are calligraphy images that are only part of a typical Chinese character. On the other hand, Data wrangling is the process of mapping the raw data to another state that is qualified to use. For instance, coupling emoji and its description to the one-hot encoding format and resizing images into a consistent dimension and format. I also leveraged scripting to do the batch processing work.

Besides, I also helped in model design and training. In the first semester, my partner stuck in getting a good result in his DCGAN because of mode collapsing. I designed another architecture of DCGAN and yielded a better result. In the training process, I also trained the networks with different hyperparameters to crosscheck which networks' architecture is producing a more promising result.

Reference

- Anoff. (2017, October 22). anoff/deep-emoji-gan. Retrieved from <https://github.com/anoff/deep-emoji-gan>.
- Arjovsky, Martin, Soumith, & Léon. (2017, December 6). Wasserstein GAN. Retrieved from <https://arxiv.org/abs/1701.07875>.
- Dabbas, E. (2019, October 22). Full Emoji Database.
- Iamcal. (2019, August 8). iamcal/emoji-data. Retrieved from <https://github.com/iamcal/emoji-data>.
- Eriklindernoren. (2019, August 31). eriklindernoren/Keras-GAN. Retrieved from <https://github.com/eriklindernoren/Keras-GAN>.
- Goodfellow, I. J., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., ... Bengio, Y. (2014). Generative Adversarial Networks. Retrieved from <https://arxiv.org/abs/1406.2661>.
- He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep Residual Learning for Image Recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. doi: 10.1109/cvpr.2016.90
- Hinton, G. E. (2006). Reducing the Dimensionality of Data with Neural Networks. *Science*, 313(5786), 504–507. doi: 10.1126/science.1127647
- Radford, A., Metz, L., & Chintala S. (2016). Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks. Retrieved from <https://arxiv.org/abs/1511.06434>

- Ronneberger, O., Fischer, P., & Brox, T. (2015). U-Net: Convolutional Networks for Biomedical Image Segmentation. *Lecture Notes in Computer Science Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015*, 234–241. doi: 10.1007/978-3-319-24574-4_28
- Mirza, M., & Osindero, S. (2014). Conditional Generative Adversarial Nets. Retrieved from <https://arxiv.org/abs/1411.1784>
- Isola, P., Zhu, J.-Y., Zhou, T., & Efros, A. A. (2017). Image-to-Image Translation with Conditional Adversarial Networks. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. doi: 10.1109/cvpr.2017.632
- Quito, A. (2019, October 18). Why we can't stop using the “face with tears of joy” emoji. *Quartz*. Retrieved from <https://qz.com/1726756/the-psychology-behind-the-most-popular-emoji/>
- Unicode. (2019). *Emoji Counts, v12.0* [Chart]. Retrieved from Unicode website: <https://www.unicode.org/emoji/charts-12.0/emoji-counts.html>
- Zhu, J.-Y., Park, T., Isola, P., & Efros, A. A. (2017). Unpaired Image-to-Image Translation Using Cycle-Consistent Adversarial Networks. *2017 IEEE International Conference on Computer Vision (ICCV)*. doi: 10.1109/iccv.2017.244
- Zhang, Ren, Sun, & Jian. (2015). Deep Residual Learning for Image Recognition. Retrieved from <https://arxiv.org/abs/1512.03385>

Huang, Gao, Liu, Maaten, van der, Laurens, Weinberger, & Kilian. (2018).

Densely Connected Convolutional Networks. Retrieved from

<https://arxiv.org/abs/1608.06993>