Text to Image Synthesis with Generative Adversarial Networks (GAN)

Li Cai
Jingyi Zhang
Jiahao Zhao

Summary	In this assignment, you'll build a Progressive Web App, which loads quickly, even on flaky networks, has an icon on the homescreen, and loads as a top-level, full screen experience.
URL	your-first-pwapp
Category	Web
Environment	web, kiosk, io2016, pwa-dev-summit, pwa-roadshow, chrome-dev-summit-2016, io2017, typtwd17, gdd17, cds17, io2018, tag-web, jsconfeu, devfest18
Status	Draft
Feedback Link	https://github.com/googlecodelabs/your-first-pwapp/issues
Author	petele

Project goals

Pipeline Design

Implementation details
- Input texts & images for training:
- Images: http://www.robots.ox.ac.uk/~vgg/data/flowers/102/
- Preprocessing: Different word embeddings.

Analysis of models
- DC-GAN
  - Structure
  - Implementation
  - Training
  - Testing
- TAC-GAN
  - Structure:
  - Implementation:
  - Training
  - Testing

Details on how to run the model

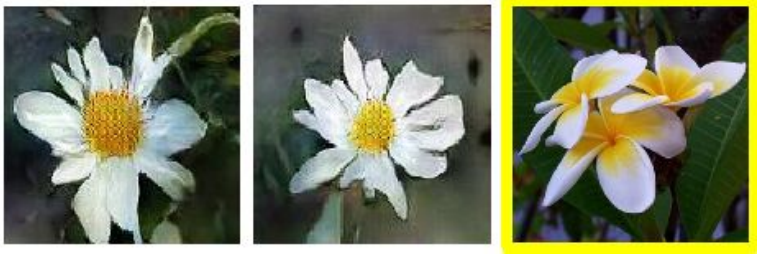
1. Project goals

Automatic synthesis of images from text is an interesting and useful topic, however, it is also one of the most challenging problem in the world of Computer Vision. Our project focuses on flower images, by training with 8000+ realistic images and descriptions of all kinds of flowers , it is going to generate a mimic flower image and try to be as accurate as possible from a paragraph of description.

Here is an example:

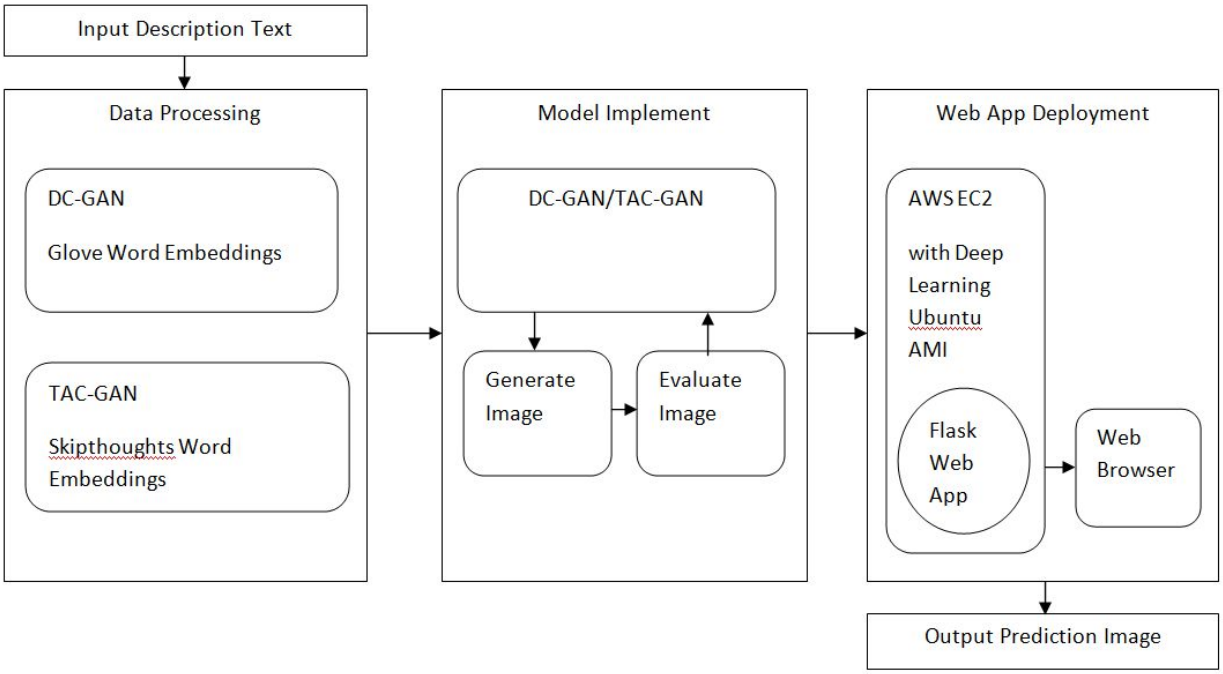
The petals of the flower are purple with a yellow center and have thin filaments coming from the petals.

This flower is white and yellow in color, with petals that are oval shaped



2. Pipeline Design

WorkFlow Diagram:



3. Implementation details

1) Input texts & images for training:

Images: http://www.robots.ox.ac.uk/~vgg/data/flowers/102/

This dataset contains images of flowers belonging to 102 different categories. The images were acquired by searching the web and taking pictures. There are a minimum of 40 images for each category.

Text: https://drive.google.com/file/d/0B0ywwgffWnLLcms2WWJQRFNSWXM/
The texts are from a academic used dataset which is a exact match for this set of images. Each class has a separate .txt file that contains descriptive statements, segmented by '\n'.

2) Preprocessing: Different word embeddings.

We introduced both Glove and skipthoughts embeddings in the preprocessing step for different GAN models. While skipthought is a semi-supervised model works for 'any text', it is also 'high weighted', thus may not be the most efficient embedding for understanding such a scoped texts.

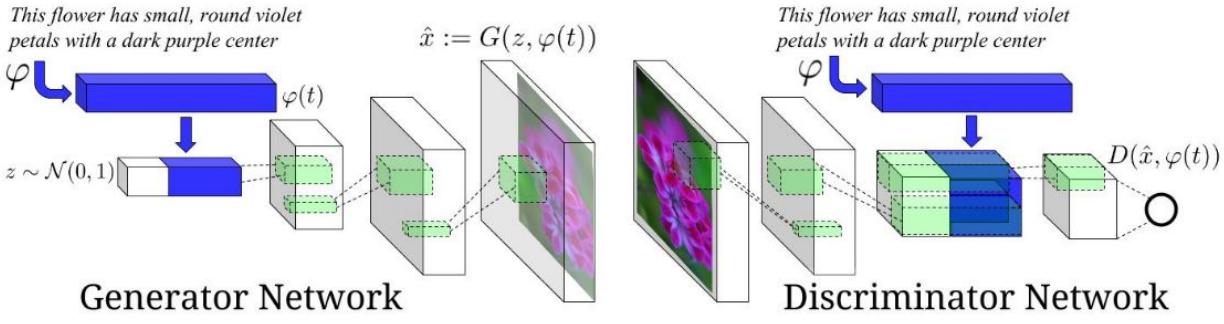
We deployed Glove embeddings before training our DCGAN model, for a relatively lighter weight and a better efficiency.

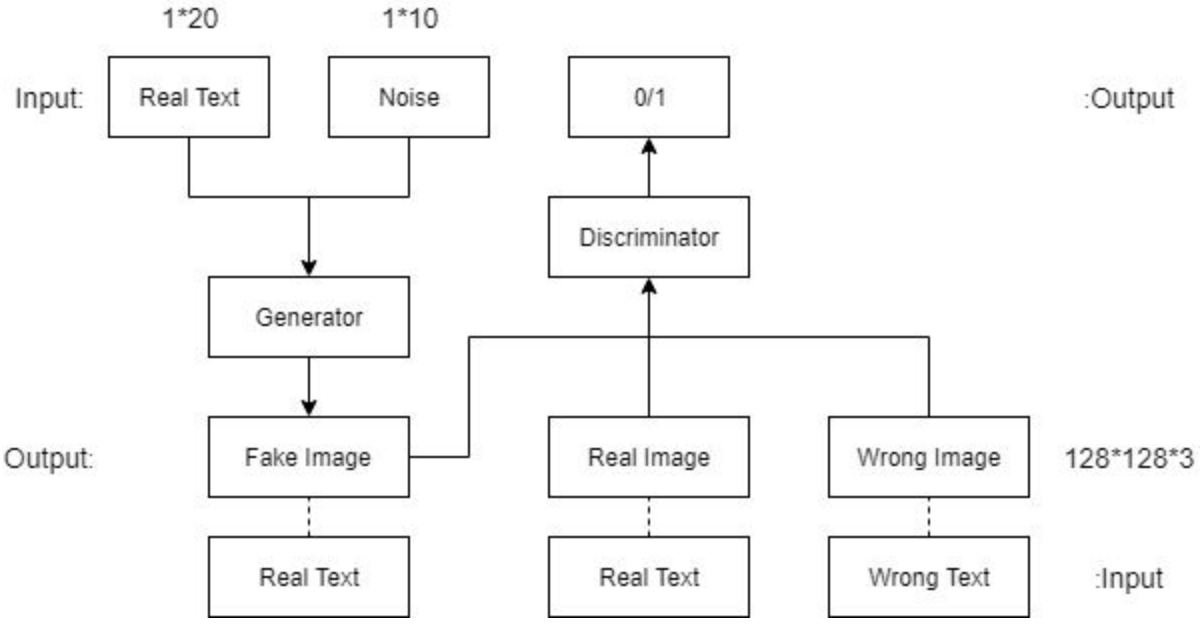
The images' shape are resized into 128*128*3 for training efficiency.

4. Analysis of models

1) DC-GAN

- Structure





- Implementation

Generator:

Layer (type)	Output Shape	Param #	Connected to
input_2 (InputLayer)	(None, 20)	0	
input_1 (InputLayer)	(None, 10)	0	
embedding_1 (Embedding)	(None, 20, 100)	500000	input_2[0][0]
dense_1 (Dense)	(None, 100)	1100	input_1[0][0]
lstm_1 (LSTM)	(None, 100)	80400	embedding_1[0][0]
concatenate_1 (Concatenate)	(None, 200)	0	dense_1[0][0] lstm_1[0][0]
dense_2 (Dense)	(None, 32768)	6586368	concatenate_1[0][0]
reshape_1 (Reshape)	(None, 16, 16, 128)	0	dense_2[0][0]
up_sampling2d_1 (UpSampling2D)	(None, 32, 32, 128)	0	reshape_1[0][0]
conv2d_1 (Conv2D)	(None, 32, 32, 128)	147584	up_sampling2d_1[0][0]

batch_normalization_1 (BatchNor	(None, 32, 32, 128)	512	conv2d_1[0][0]
activation_1 (Activation)	(None, 32, 32, 128)	0	batch_normalization_1[0][0]
up_sampling2d_2 (UpSampling2D)	(None, 64, 64, 128)	0	activation_1[0][0]
conv2d_2 (Conv2D)	(None, 64, 64, 64)	73792	up_sampling2d_2[0][0]
batch_normalization_2 (BatchNor	(None, 64, 64, 64)	256	conv2d_2[0][0]
activation_2 (Activation)	(None, 64, 64, 64)	0	batch_normalization_2[0][0]
up_sampling2d_3 (UpSampling2D)	(None, 128, 128, 64)	0	activation_2[0][0]
conv2d_3 (Conv2D)	(None, 128, 128, 32)	18464	up_sampling2d_3[0][0]
batch_normalization_3 (BatchNor	(None, 128, 128, 32)	128	conv2d_3[0][0]
activation_3 (Activation)	(None, 128, 128, 32)	0	batch_normalization_3[0][0]
conv2d_4 (Conv2D)	(None, 128, 128, 3)	867	activation_3[0][0]
activation_4 (Activation)	(None, 128, 128, 3)	0	conv2d_4[0][0]

Total params: 7,409,471
Trainable params: 7,409,023
Non-trainable params: 448

Discriminator:

Layer (type)	Output Shape	Param #	Connected to
input_3 (InputLayer)	(None, 128, 128, 3)	0	
conv2d_6 (Conv2D)	(None, 64, 64, 32)	896	input_3[0][0]
zero_padding2d_1 (ZeroPadding2D	(None, 65, 65, 32)	0	conv2d_6[0][0]
batch_normalization_4 (BatchNor	(None, 65, 65, 32)	128	zero_padding2d_1[0][0]
leaky_re_lu_2 (LeakyReLU)	(None, 65, 65, 32)	0	batch_normalization_4[0][0]
dropout_2 (Dropout)	(None, 65, 65, 32)	0	leaky_re_lu_2[0][0]
conv2d_7 (Conv2D)	(None, 33, 33, 64)	18496	dropout_2[0][0]
batch_normalization_5 (BatchNor	(None, 33, 33, 64)	256	conv2d_7[0][0]
leaky_re_lu_3 (LeakyReLU)	(None, 33, 33, 64)	0	batch_normalization_5[0][0]
dropout_3 (Dropout)	(None, 33, 33, 64)	0	leaky_re_lu_3[0][0]
conv2d_8 (Conv2D)	(None, 33, 33, 128)	73856	dropout_3[0][0]

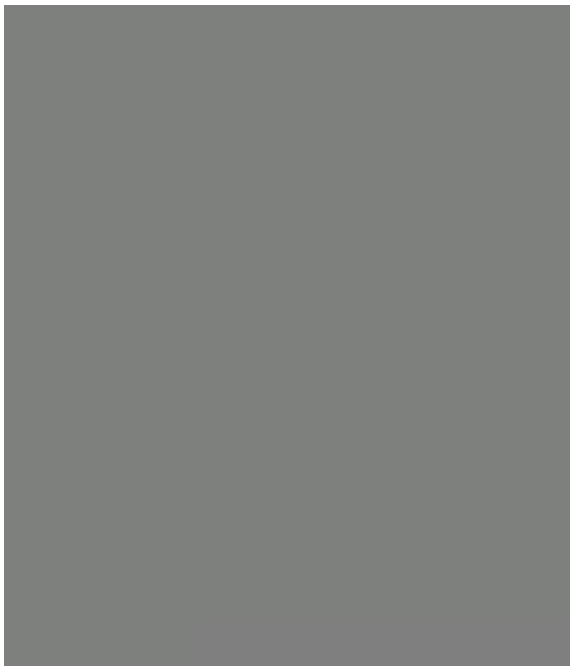
batch_normalization_6 (BatchNor	(None, 33, 33, 128)	512	conv2d_8[0][0]
leaky_re_lu_4 (LeakyReLU)	(None, 33, 33, 128)	0	batch_normalization_6[0][0]
dropout_4 (Dropout)	(None, 33, 33, 128)	0	leaky_re_lu_4[0][0]
input_4 (InputLayer)	(None, 20)	0	
flatten_1 (Flatten)	(None, 139392)	0	dropout_4[0][0]
embedding_2 (Embedding)	(None, 20, 100)	500000	input_4[0][0]
dense_3 (Dense)	(None, 100)	13939300	flatten_1[0][0]
lstm_2 (LSTM)	(None, 100)	80400	embedding_2[0][0]
concatenate_2 (Concatenate)	(None, 200)	0	dense_3[0][0] lstm_2[0][0]
activation_5 (Activation)	(None, 200)	0	concatenate_2[0][0]
dense_4 (Dense)	(None, 1)	201	activation_5[0][0]
activation_6 (Activation)	(None, 1)	0	dense_4[0][0]

Total params: 14,614,045
Trainable params: 14,613,597
Non-trainable params: 448

- Training

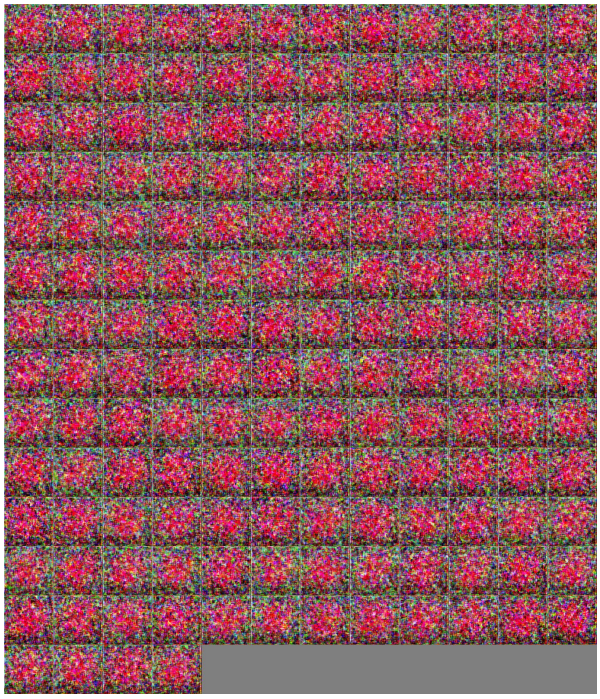
(1) Without input noise

If the only input of generator is text, the output will be always a single color image:



(2) With input noise

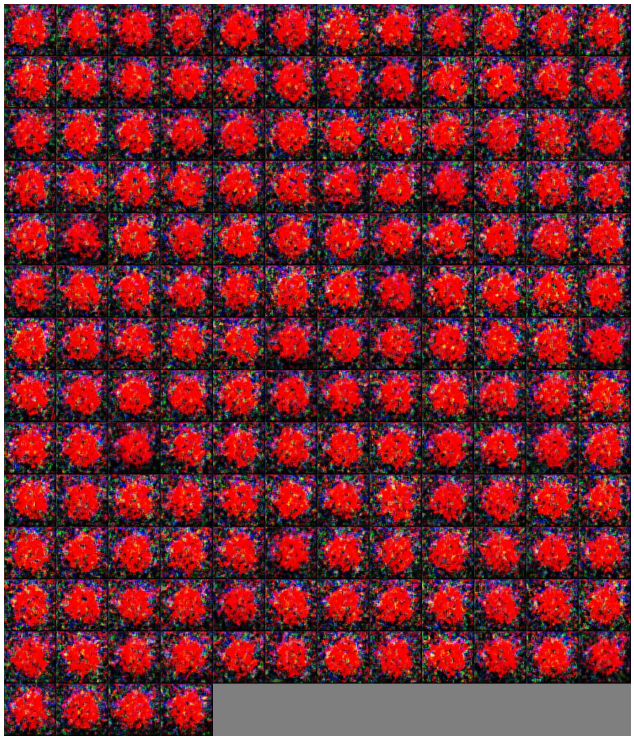
The input of generator is a combination of text and random noise between (-1,1) and get an output like random noise:



Epoch 1

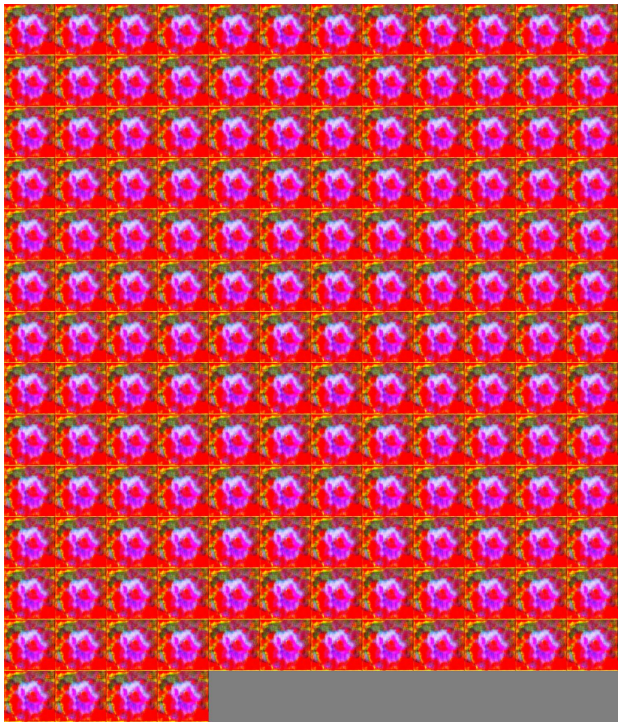
(3) With input noise for generator and mismatched text and images pairs for discriminator:

Except (real text, real image) and (real text, generated image), add a group of mismatched pairs (wrong text, wrong image) for the discriminator:

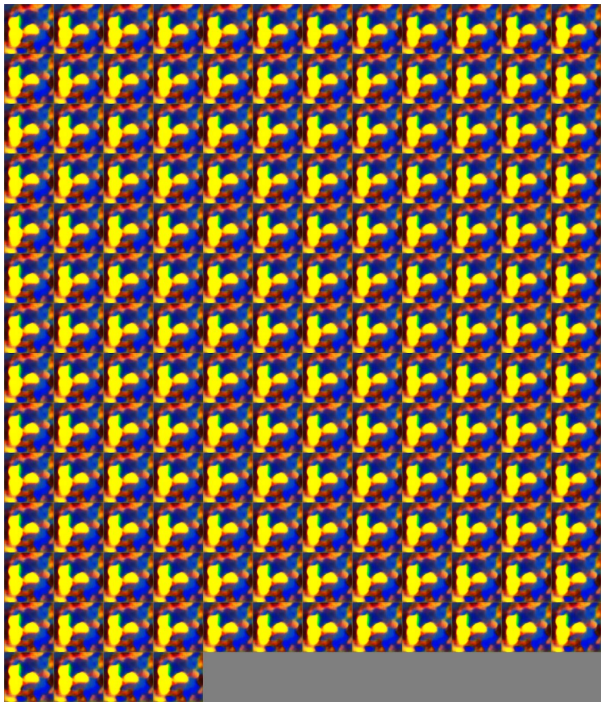


Epoch 1

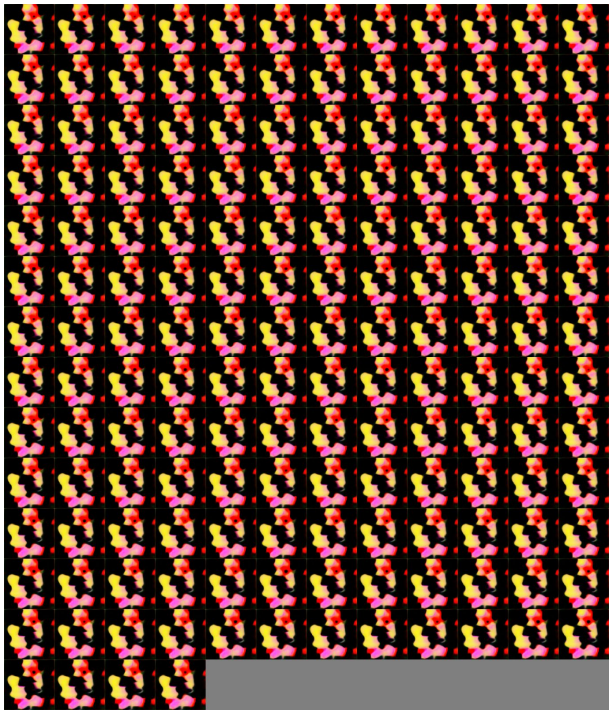
It’s better than the former, but the optimiser is not good enough. Replace the SGD to Adam and the result is totally different:



Epoch 1



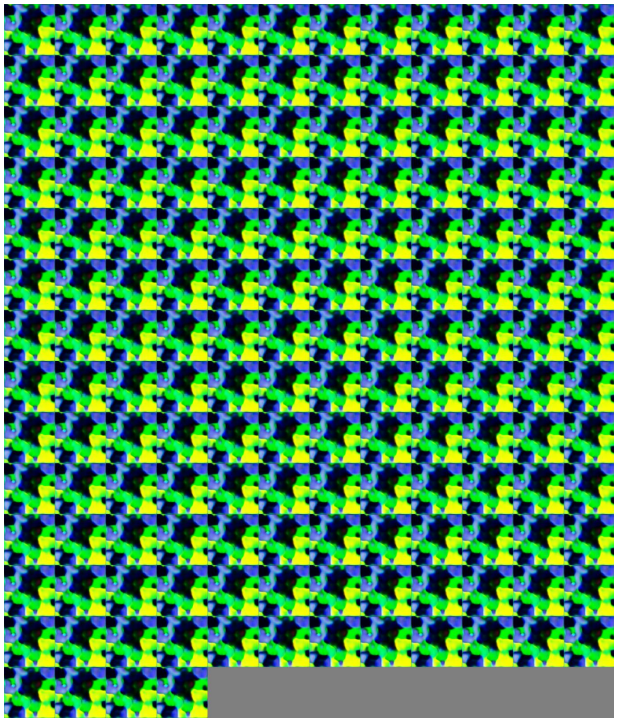
Epoch 1



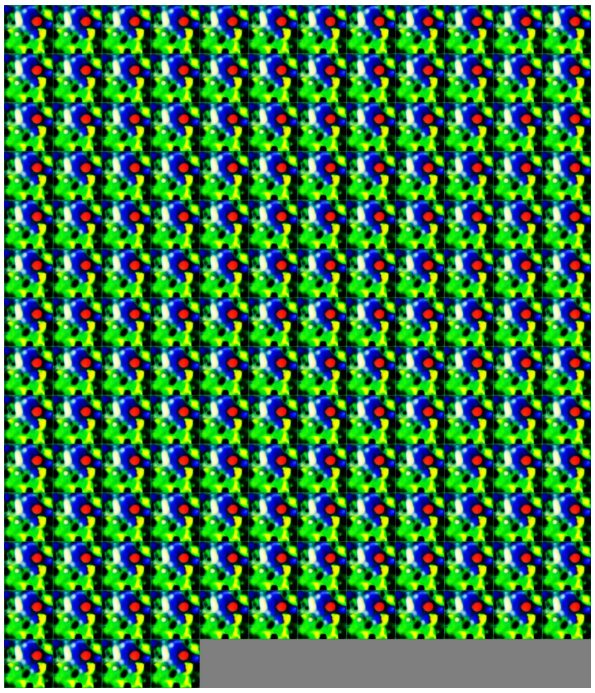
Epoch 4

The final model is based on it with adjustment of noise length from 20 to 10

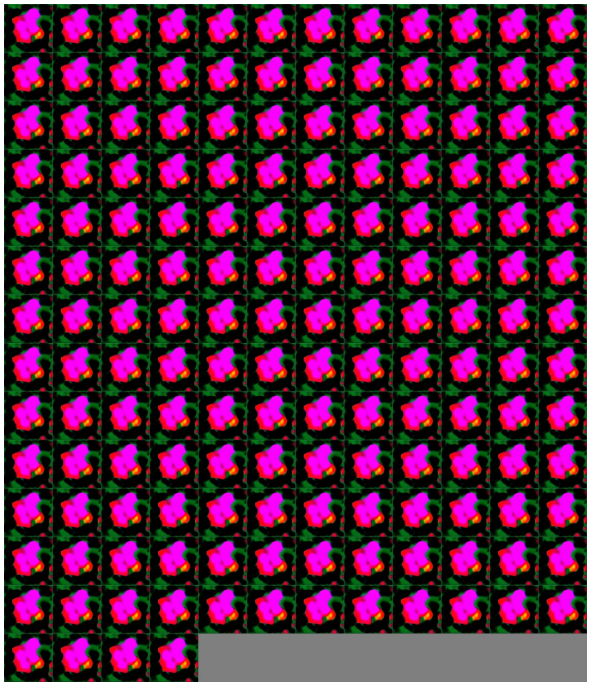
Epoch 1



Epoch 3



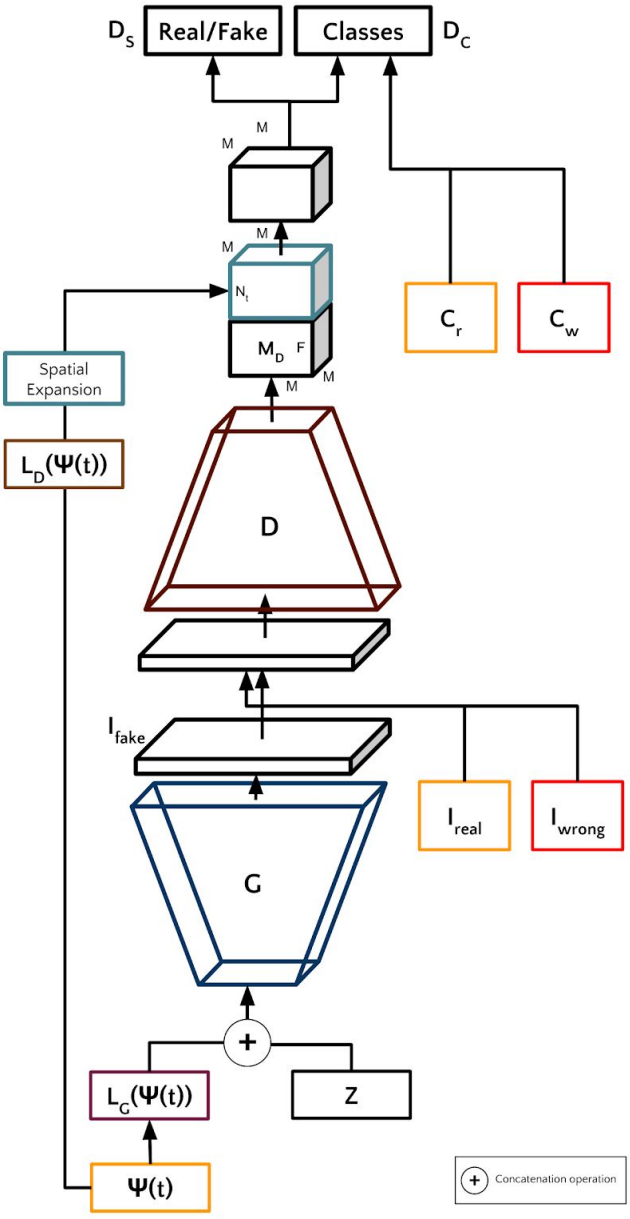
Epoch 6



Epoch 9

2) TAC-GAN

- Structure:



- Implementation:

  For this step, we referenced a built application from:
  https://github.com/dashayushman/TAC-GAN

  And thanks to the many help offered by its author during reproducing it, an updated version for our case with several minor fixes, i.e. reusing variables in the generator graph, creating a pipeline for deployment and so on, is now available on:
  https://github.com/Allen001822480/TAC-GAN
  In the following steps, all use cases can be reached via command line interfaces.

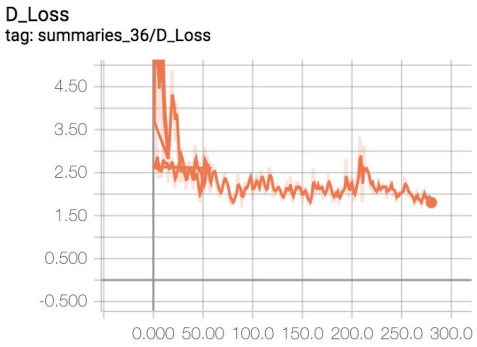
- Training
  - Data: Before the training process we split the dataset into train and test on a portion of 9 to 1.

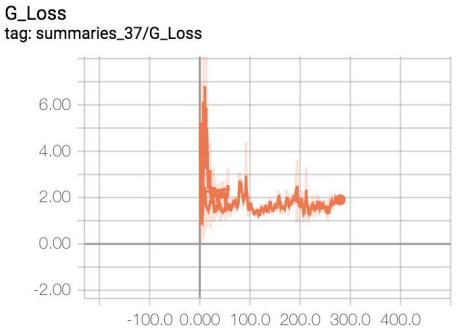
  - Optimizer: The Optimizer defined(Adam) is to minimize the loss for both discriminator and generator.

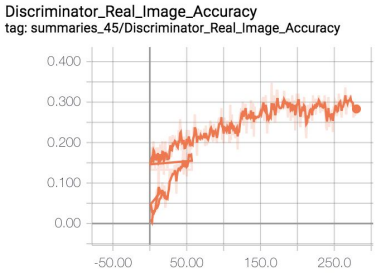
  - Hyperparams: We downsampled the images into a size of 100^2 and increased the batch_size until 256, The time cost per epoch is reduced to approximately 510s from over 700s.

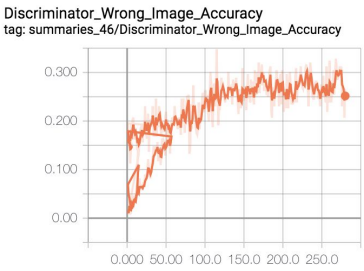
  - Callbacks:
    - Checkpoints are saved after every epoch.
    - All cardinal command line calls are documented in a jupyter notebook.
    - During the training process we printed the following loss values and exported the procedure to tensorboard.

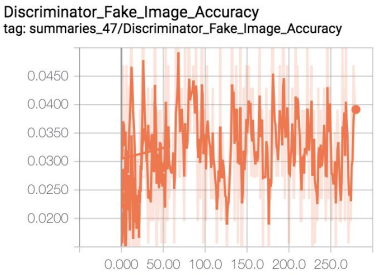
```
LOSSES
Discriminator Loss: 1.9538739919662476
Generator Loss: 1.0611664056777954
Batch Number: 22
Epoch: 11,
Total Batches per epoch: 28


G loss-1 [Real/Fake loss for fake images] : 1.0082467794418335
G loss-2 [Aux Classifier loss for fake images]: 0.0529196597635746

D total loss: 1.865512490272522
D loss-1 [Real/Fake loss for real images] : 1.0728847980499268
D loss-2 [Real/Fake loss for wrong images]: 0.11369965970516205
D loss-3 [Real/Fake loss for fake images]: 0.555875301361084
D loss-4 [Aux Classifier loss for real images]: 0.03353866934776306
D loss-5 [Aux Classifier loss for wrong images]: 0.03394211828708649
```









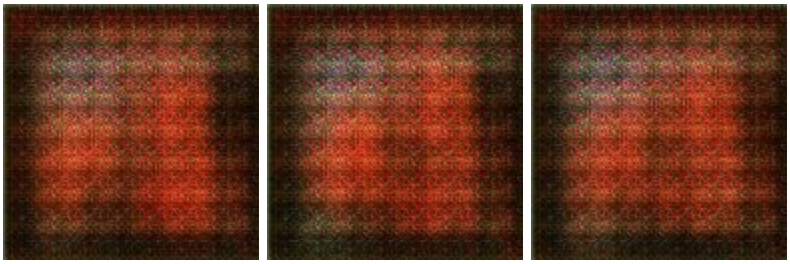


- Testing

  With a setting mentioned above, here we present some images generated from training set. The following images are all generated from text:

  ‘a flower with red petals which are pointed‘

- Epoch 1:



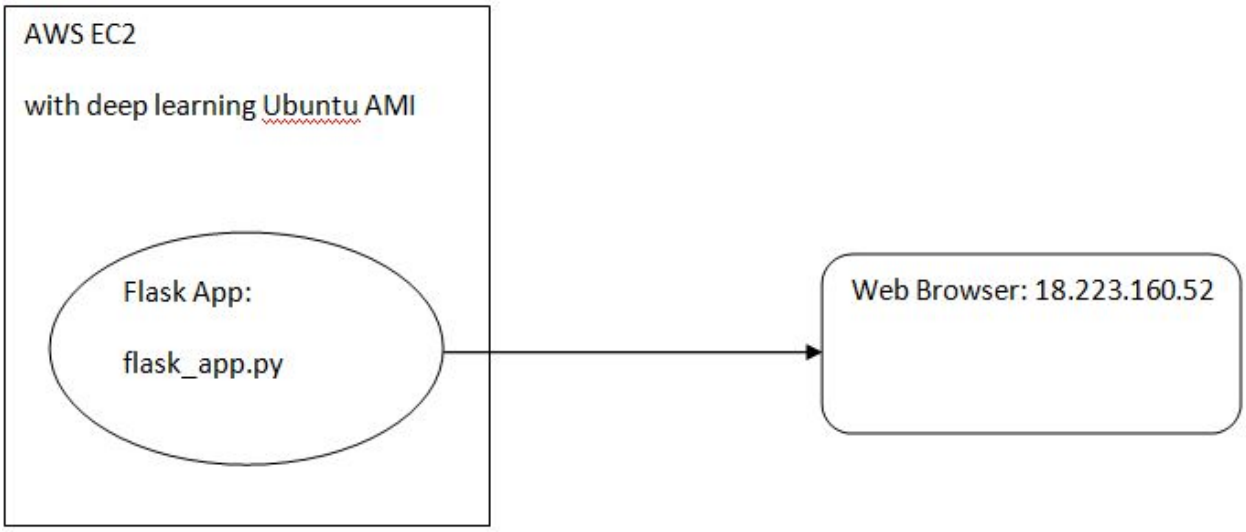
- Epoch 10:

Images started to vary from each other.

- Epoch 30:

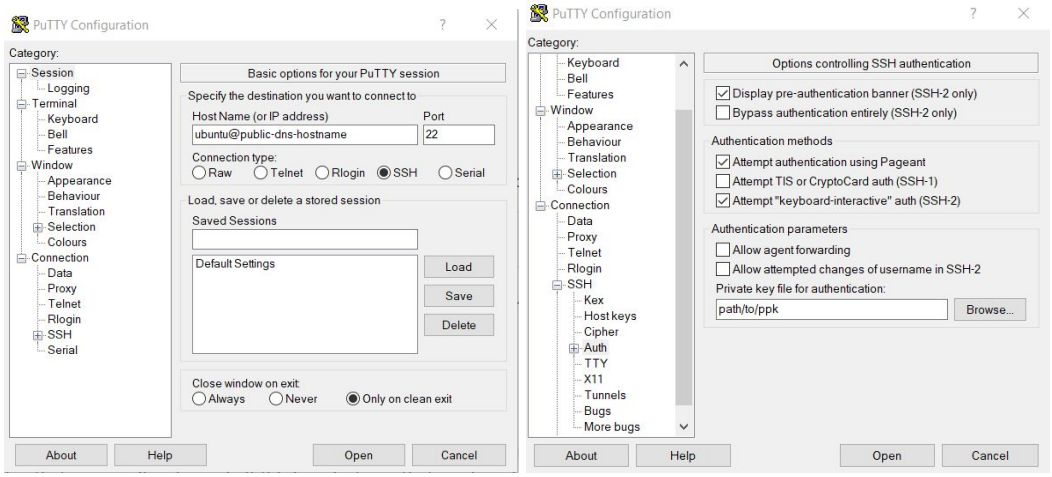
The correlation between image and text started to fade.

5. Details on how to run the model



1. Login to AWS ec2 instance
SSH:
ssh -i /path/my-key-pair.pem ubuntu@public-dns-hostname (in your local machine terminal)
https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/AccessingInstancesLinux.html

Putty:



https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/putty.html

More Information of trouble shooting:

https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/TroubleshootingInstancesConnecting.html#TroubleshootingInstancesConnectingMindTerm

2. Run flask.py

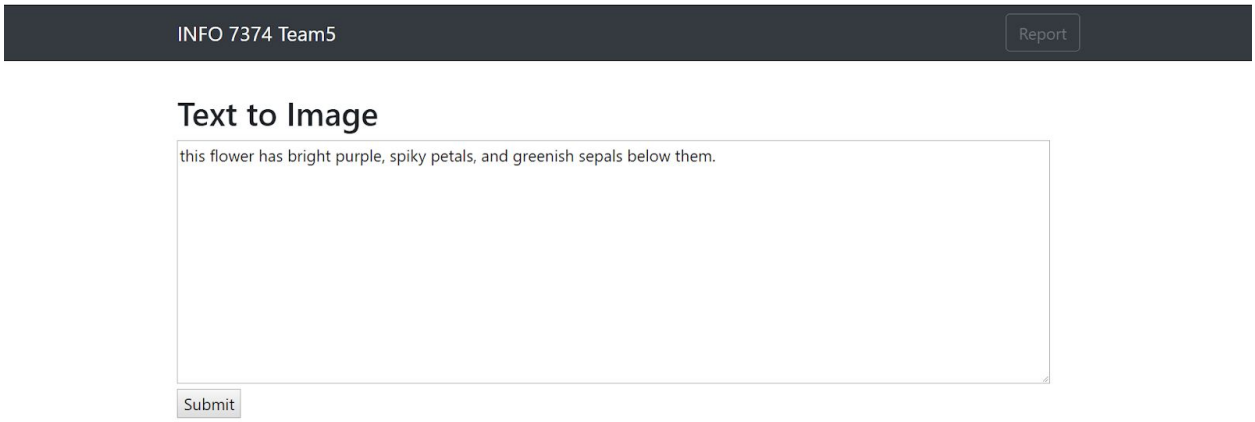
After connecting your local machine with EC2, and in EC2's terminal:

cd flask

sudo python3 flask_app.py

Now you can access flask app on your browser by entering: 18.223.160.52

3. Enter the description of the flower you want



4. Click submit