Generative Adversarial Networks : Image generation and text to image generation

Abstract

Generative Adversarial Networks have been developed to generate content with less amount to captioned data to be trained on. It consists of two networks, a Generator network, which is responsible for generating images as realistic as possible and a Discriminator network, whose purpose is to distinguish between the images which are computer generated and not captured from the real world. Both the networks are trained simultaneously and play a competitive game, where the generator aims at fooling the discriminator by generating images as close to the real world as possible, while the discriminator network getting sharper at recognizing fake content from the real content. We can backpropagate the results from the discriminator network to the generator network to improve its efficiency in creating genuine looking images. In this case, the number of parameters are significantly small, as compared to the training data, the models are forced to understand and internalize the data to generate new content.

Introduction

This project aims at implementing text-to-image generation application using the generative adversarial network. This implementation is based on **Generative Adversarial Text to Image Synthesis paper** by Scott Reed, Zeynep Akata, Xinchen Yan. Converting text or high-level language (English sentences) to realistic images, ie depicting the content of the text in a picture, would be a very influential aspect in fields like training artificial intelligence . For example, a statement such as " this girl is wearing black dress and is carrying a white purse" should give us an image of a girl in a black dress with a white purse. Another example, taken from a research paper on similar context, [reference: Generative Adversarial Text-to-Image Synthesis http://arxiv.org/abs/1605.05396.

The following image is generated by training on the CUB dataset of birds.

The following image is generated from the oxford flowers dataset.

In this project, the first part is to generate only images using Generative Models to better understand the algorithm. Later part is the text-to image generation, which is added to future scope for now.

Background and related work:

This section describes the current work and implementations of Generative Adversarial networks.

Yann LeCun, director of Facebook AI, said "Generative Adversarial Networks is the most interesting idea in the last ten years in machine learning." GAN's were originally proposed by Ian Goodfellow, in 2014 and since then have become the next big thing in the deep learning world. The basic structure of GAN is of two neural networks, trained at the same time, participate in a minmax game, to make each other better. The generator network is build to generate fake images to fool the discriminator and the the discriminator network is the investigator, deciding whether an image is real or fake.

Since the first GAN was written, many newer and improved versions have come up nd following are a few of them.

DCGAN : https://arxiv.org/abs/1511.06434

The first stable version and major improvement over vanilla GAN, suitable to be used for images generation in unsupervised learning. The authors focused on improving the architecture of the model and added batch normalization and Relu to the layers.

Improved DCGAN: https://arxiv.org/abs/1606.03498

These were written to improve the DCGAN performance to generate high resolution images. The authors proposed improvements such as feature matching, historical averaging, one-sided label smoothing etc.

Conditional GANS: https://arxiv.org/abs/1411.1784

These GANS are used in the text to image synthesis model. These were written to have control over various aspects of data, like cations along with image are fed into the generator and trained to create caption based images. StackGANs is an implementation.

InfoGANS: https://arxiv.org/abs/1606.03657

GANs that can encode meaningful image features in part of the noise vector Z in an unsupervised manner. For example, encode the rotation of a digit.

These were few of many implementations and enhancements of generative models, used for specific purposes. The article Fantastic GANS and where to find them gives a nice overview.

http://guimperarnau.com/blog/2017/03/Fantastic-GANs-and-where-to-find-them

DATASET:

Moving toward the actual implementation, let us first look at the dataset that will be used.

MNIST dataset : http://yann.lecun.com/exdb/mnist/

The MNIST database of handwritten digits, available from this page, has a training set of 60,000 examples, and a test set of 10,000 examples. It is a subset of a larger set available from NIST. The digits have been size-normalized and centered in a fixed-size image.

Celebrity Faces dataset : http://mmlab.ie.cuhk.edu.hk/projects/CelebA.html

CelebFaces Attributes Dataset (CelebA) is a large-scale face attributes dataset with more than 200K celebrity images, each with 40 attribute annotations. It consists of 10,177 number of identities, 202,599 number of face images, 5 landmark locations, 40 binary attributes annotations per image.

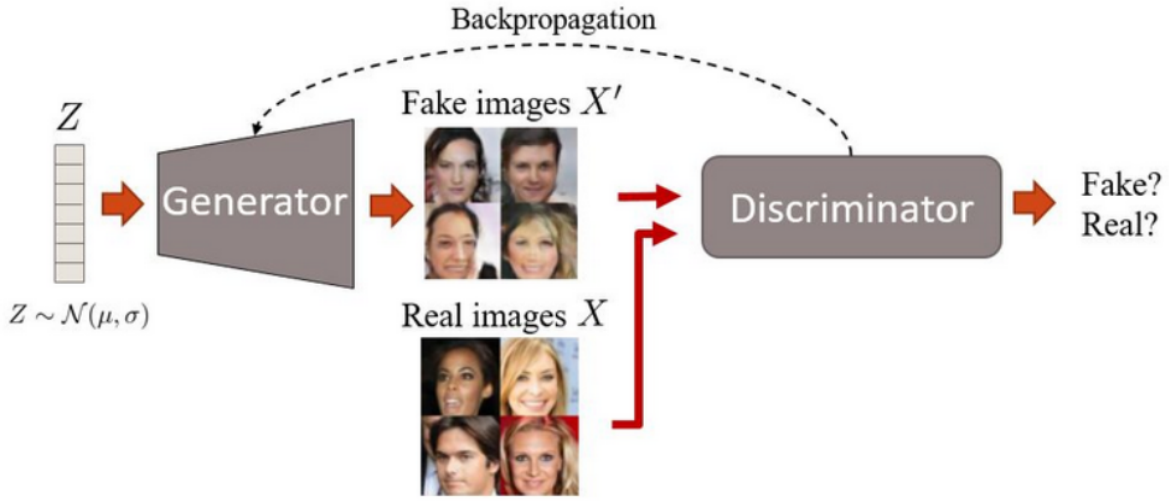
Models

The primary deep learning model used for this project is DCGAN or Deep Convolutional Generative Adversarial Network, which is the first improvement over the initial vanilla GAN. The basic architecture of a generative model or GAN comprises of two neural networks built to defeat each other. In a simple example, it is similar to playing a new game with a fairly experienced opponent. As the game progresses, the new player learns from his mistakes and starts getting better at it. The generative adversarial network is built on the same ideology. It comprises of two neural networks,

Generator network : A deep network that generates realistic images. (fake images)

Discriminator network: A deep network distinguishes real images from computer generated fake images.

Gan Training:



GAN training overview.

Generator Network, takes a random sample as input and tries to generate a sample of data(X'). In the above image, we can see the generator takes a input Z, where Z is a sample from probability distribution. It is then fed to the Discriminator along with real world images(X). The goal of the discriminator is that of a binary classifier, ie, it has to classify or detect whether the image is real or fake, as gives output as labels, 0 for fake and 1 for real images. Mathematically GAN can be described as, if,

- Pdata(x) -> the distribution of real data
- X -> sample from pdata(x)
- P(z) -> distribution of generator
- Z -> sample from p(z)
- G(z) -> Generator Network
- D(x) -> Discriminator Network, then

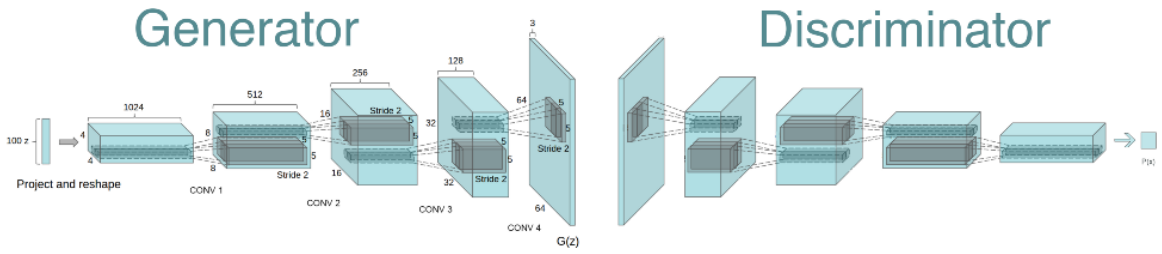
$$\min_G \max_D V(D,G)$$

$$V(D,G) = \mathbb{E}_{x \sim p_{data}(x)}[\log D(x)] + \mathbb{E}_{z \sim p_z(z)}[\log(1 - D(G(z))]$$

Where, in the function V(D,G), the first term is entropy that the data from real distribution (pdata(x)) passes through the discriminator, which the discriminator tries to maximize to 1. The second term is entropy that the data from random input (p(z)) passes through the generator, which then generates a fake sample which is then passed through the discriminator to identify the fakeness.(discriminator tries to maximize it to 0). **Concusivly,the discriminator is trying to maximize function V**.

The task of generator is exactly opposite, i.e. it tries to minimize the function V, so that the differentiation between real and fake data is minimum.

Deep Convolutional Generative Adversarial Networks (DCGAN)



An example architecture for generator and discriminator networks. Both utilize convolutional layers to process visual information

DCGAN are a part of Convolutional Neural Network, and are the first modifications made to the vanilla GANS, making them stable and usable for unsupervised learning. As per Alec Radford & Luke Metz, in their paper, the major alterations to the basic GAN to turn it into a DCGAN are,

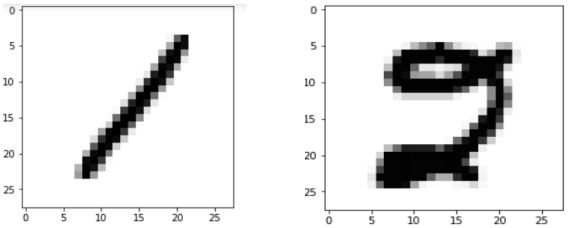
- Replace any pooling layers with strided convolutions (discriminator) and fractional-strided convolutions (generator).
- Use batch normalization in both the generator and the discriminator
- Remove fully connected hidden layers for deeper architectures
- Use ReLU activation in generator for all layers except for the output, which uses Tanh.
- Use LeakyReLU activation in the discriminator for all layers

Implementation and Explanation

Implemented by Author : Aymeric Damien.

Project : https://github.com/aymericdamien/TensorFlow-Examples/

The first implementation of DCGAN, preformed is for the MNIST dataset. As mentioned above, the algorithm is implemented with the changes. The data contains images and labels, we take the images are of size 28x28. A sample image from the dataset looks like,



Discriminator network

```
# Discriminator Network
# Input: Image, Output: Prediction Real/Fake Image
def discriminator(x, reuse=False):
    with tf.variable_scope('Discriminator', reuse=reuse):
        # Typical convolutional neural network to classify images.
        x = tf.layers.conv2d(x, 64, 5, strides=2, padding='same')
        x = tf.layers.batch_normalization(x, training=is_training)
        x = leakyrelu(x)
        x = tf.layers.conv2d(x, 128, 5, strides=2, padding='same')
        x = tf.layers.batch_normalization(x, training=is_training)
        x = leakyrelu(x)
        # Flatten
        x = tf.reshape(x, shape=[-1, 7*7*128])
        x = tf.layers.dense(x, 1024)
        x = tf.layers.batch_normalization(x, training=is_training)
        x = leakyrelu(x)
        # Output 2 classes: Real and Fake images
        x = tf.layers.dense(x, 2)
    return x
```

The discriminator network is a simple convolutional neural network. It works as binary classifier as we understood in the theory. As per DCGAN requirements, batch normalization is applied and the activation function is leaky Rectified Linear Unit applied to all layers of the discriminator. There are no pooling layers, as they are replaced by strided convolutions.

Convo2d() : constructs a 2D convolution. Takes a 4D input and filter tensors.

tf.layers.convo2d: Functional interface for the 2D convolution layer. This layer creates a convolution kernel that is cross-correlated with the layer input to produce a tensor of outputs.

relu() : used to add some amount of non-linearity to the output of convolutional layers, to aid the network in learning complex functions.

Generator network

```
def generator(x, reuse=False):
    with tf.variable_scope('Generator', reuse=reuse):
        # TensorFlow Layers automatically create variables and calculate their
        # shape, based on the input.
        x = tf.layers.dense(x, units=7 * 7 * 128)
        x = tf.layers.batch_normalization(x, training=is_training)
        x = tf.nn.relu(x)
        # Reshape to a 4-D array of images: (batch, height, width, channels)
        # New shape: (batch, 7, 7, 128)
        x = tf.reshape(x, shape=[-1, 7, 7, 128])
        # Deconvolution, image shape: (batch, 14, 14, 64)
        x = tf.layers.conv2d_transpose(x, 64, 5, strides=2, padding='same')
        x = tf.layers.batch_normalization(x, training=is_training)
        x = tf.nn.relu(x)
        # Deconvolution, image shape: (batch, 28, 28, 1)
        x = tf.layers.conv2d_transpose(x, 1, 5, strides=2, padding='same')
        # Apply tanh for better stability - clip values to [-1, 1].
        x = tf.nn.tanh(x)
        return x
```

The generator function takes noise and tries to learn how to transform this noise into digits through multiple transpose convolutions. The output layer uses TanH as activation function, along with batch normalization applied on all layers of the network and Rectified Linear Unit activation for rest of the layers.

The generator is a reverse discriminator, CNN working backwards. Thus we start with linear transformation or flatten the values before feeding them forward in the network. This step was performed in the end for the discriminator.

The first hidden layer is reshaped to a small image array, which is sent through the network to become the upscaled image at the end. Batch normalization is applied at all layers. In each layer, we perform inverse convolution and apply non-linearity.

conv2d_transpose() : returns an upscaled image from the inputs which are a vector of noise, the output shape of the image, strides and height and width of the kernels.

TanH activation function is used on the output layer, as described in the DCGAN architecture.

Cost functions:

```
# Build Loss (Labels for real images: 1, for fake images: 0)
# Discriminator Loss for real and fake samples
disc_loss_real = tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_with_logits(
    logits=disc_real, labels=tf.ones([batch_size], dtype=tf.int32)))
disc_loss_fake = tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_with_logits(
    logits=disc_fake, labels=tf.zeros([batch_size], dtype=tf.int32)))
# Sum both loss
disc_loss = disc_loss_real + disc_loss_fake
# Generator Loss (The generator tries to fool the discriminator, thus labels are 1)
gen_loss = tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_with_logits(
    logits=stacked_gan, labels=tf.ones([batch_size], dtype=tf.int32)))
```

Three loss functions, loss of the generator, the loss of the discriminator when using real images and the loss of the discriminator when using fake images. The sum of the fake image and real image loss is the overall discriminator loss.

Loss functions are the most tricky for a GAN. As we have two neural network, each with its own implementation, two separate loss functions are defined, each to benefit that neural network.

Batch normalization: Batch normalization is used incase the images provided to the network are different than the rest, thus creating a bias and the leading to improper learning of the network. It is described by Sergey Ioffe, Christian Szegedy in their paper. The intensity of the image is distributed around a common mean with a set covariance.

Training the network:

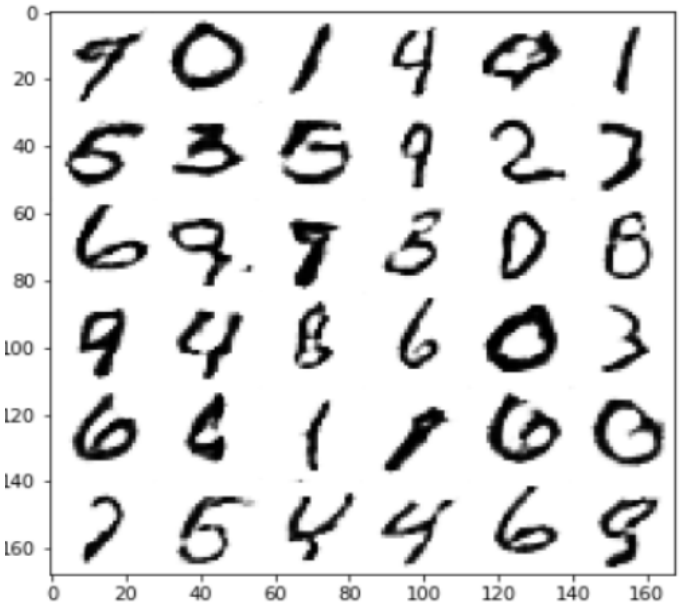
Both the networks are trained together to avoid making one network significantly stronger than the other.

Evaluation:

The generator is fed a noise sample from a uniform distribution, with 100 initial data points. The number of iterations are set to 10,000 and a batch size of 128. The learning rate is set to 0.002.

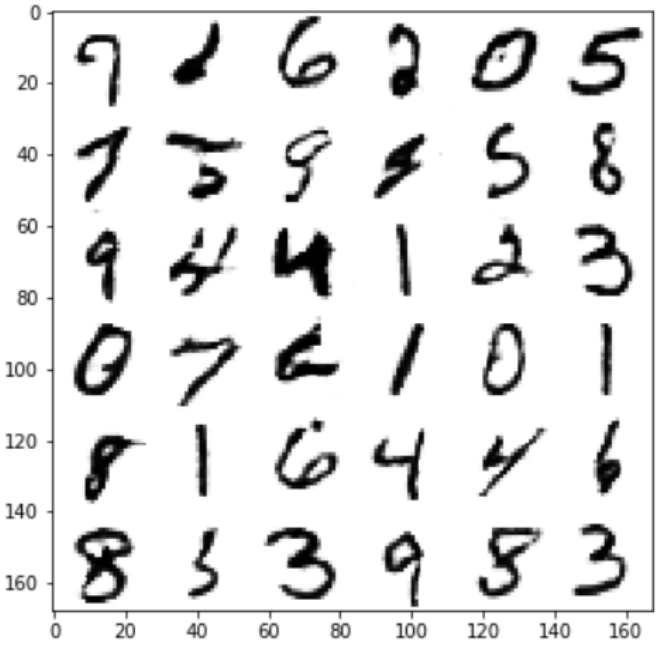
After 5000 steps,

```
Step 5000: Generator Loss: 4.319510, Discriminator Loss:
0.151860
```



After 10000 steps,

```
Step 10000: Generator Loss: 6.213067, Discriminator Loss:
0.086564
```



DCGAN implementation on celebA dataset

(Implemented at *https://github.com/dmonn/dcgan-oreilly/blob/master/DCGANs%20with%20Tensorflow.ipynb****) by O'reilly***

Step 1. Reshape images to 28 x 28. Generate batches to feed to the network.

Step 2. Create 4-Layer Discriminator network with batch normalization and lrelu as activation(similar to MNIST implementation)

Step 3. Create a fully connected Generator network, start by reshape the first layer and run inverse convolution on each layer (similar to MNIST implementation)

Step 4. Define loss function. Three loss functions, loss of the generator, the loss of the discriminator when using real images and the loss of the discriminator when using fake images. The sum of the fake image and real image loss is the overall discriminator loss.

Used sigmoid_cross_entropy_with_logits() : Measures the probability error in discrete classification tasks in which each class is independent and not mutually exclusive(from tensorflow documentations)

Step 5. Define AdamOptimizer with a learning rate of 0.0002 and beta1 = 0.5

Step 6. Function to visualize the generated images after every 400 steps and display generator and discriminator loss.

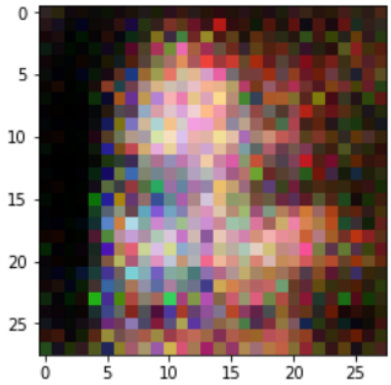
Step 7. Train the network. Use a random uniform distribution to generate the noise vector.

Step 8. Function to generate a graph for the loss values of generator and discriminator.

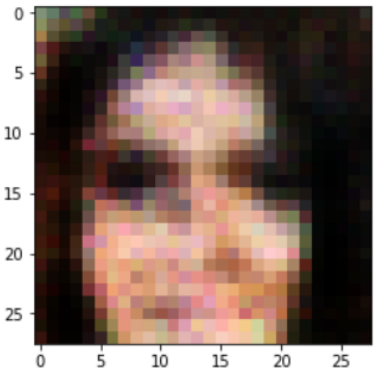
Evaluation:

The following images were generated

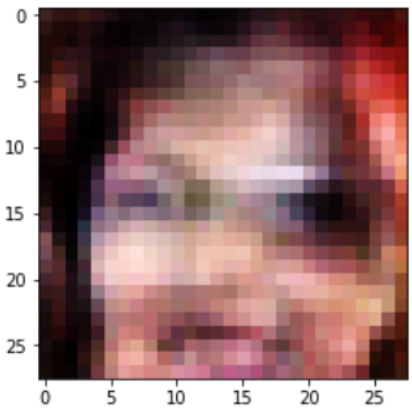
```
Epoch 1/2... Discriminator Loss: 1.2476... Generator Loss: 0.8163
```



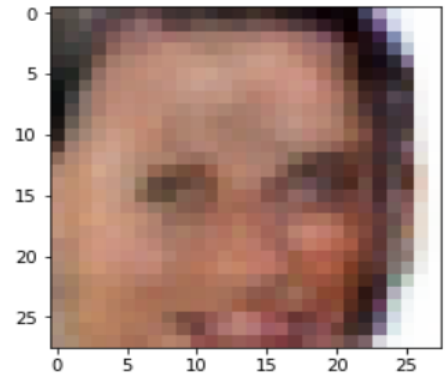

```
Epoch 1/2... Discriminator Loss: 1.3136... Generator Loss: 0.8065
```



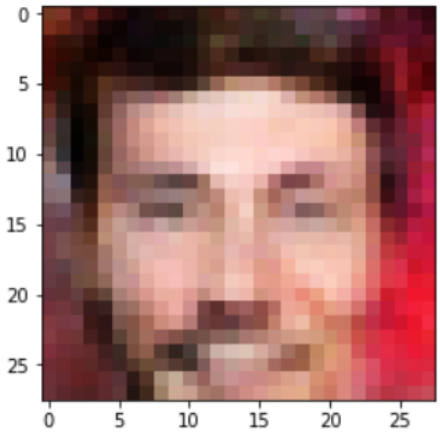

```
Epoch 1/2... Discriminator Loss: 1.2419... Generator Loss: 0.8225
```



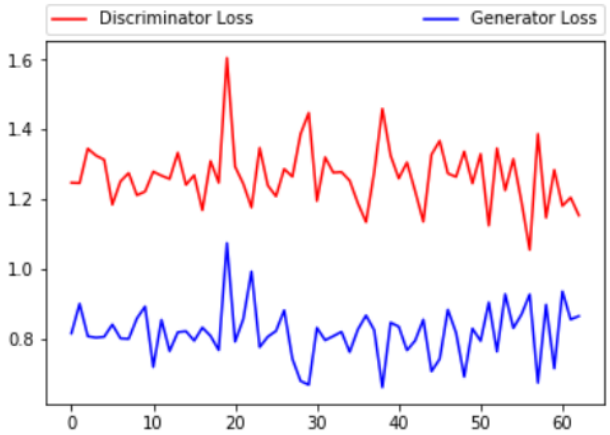
```
Epoch 2/2... Discriminator Loss: 1.3283... Generator Loss: 0.7074
```




```
Epoch 2/2... Discriminator Loss: 1.1250... Generator Loss: 0.9049
```

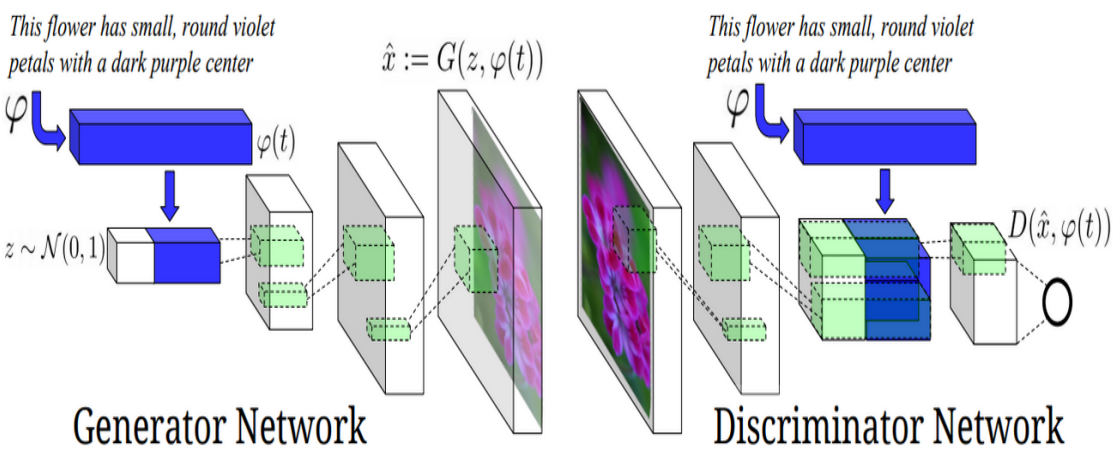


The model dosen't seem to converge just yet, but maybe training to 100 epochs might give better results.



Future Scope: Text-to-image Generation.

- Train and improve on the generative model
- Learn word2vec and lstm for text encoding
- Implement the text to image model mentioned in the introduction.
-



- Steps of the algorithm Input : minibatch images x, matching text t, mismatching,t^, number of training batch steps S
- 1.encode matching text descriptions
- 2.Encode mis-matching text description
- 3.Draw sample of random noise
- 4.Forward through generator, generate image
- 5.calculate score for real image and right text (image and its coresponding text)
- 6.calculate score for real image wrong text
- 7.calculate score for fake image, right text
- 8.Determine the scores of discriminator and generators
- 9.Update the discriminator/generator by determinig the gradient of D and G 's objective with respect to its parameters.
- Batch normalization is used in both the networks.

References

https://arxiv.org/pdf/1511.06434.pdf

https://medium.com/@awjuliani/generative-adversarial-networks-explained-with-a-classic-spongebob-squarepants-episode-54deab2fce39

https://arxiv.org/abs/1502.03167

https://github.com/aymericdamien/TensorFlow-Examples/

http://guimperarnau.com/blog/2017/03/Fantastic-GANs-and-where-to-find-them

https://www.analyticsvidhya.com/blog/2017/06/introductory-generative-adversarial-networks-gans/

https://medium.com/@awjuliani/generative-adversarial-networks-explained-with-a-classic-spongebob-squarepants-episode-54deab2fce39