Develop a GAN to Generate Fruit Images

Abstract

In this project, new fruit images were generated from random noise data with GAN. Discriminator model with four convolutional layers for downsample and one dense layer for classification was used to determine fake and real images. Also, generator model with first dense layer used; then, reshaped random noise data into something image-like data and three Conv2DTranspose layers used to increase number of pixels to pixel of target images and last Convolutional layer used to convert number of image color channels to number of target color channels. Final model after each ten epochs were saved and images displayed (up to 200 epochs). We could distinguish fruit generated images after 50 epochs; but, images after 200 epochs were very similar to real images and we could distinguish types of fruit.

Introduction

Generative Adversarial Networks, GANs, are a kind of architectures to train generative models; they usually contain deep convolutional neural networks to generate images. Developing a GAN to generate images needs two neural network models: Discriminator, to classify given images (real images) and generated images, as Generator, uses inverse convolutional neural network layers to transform an input to a full two dimensional image of pixel values.
Understanding how GANs work and how deep convolutional neural network models are trained in GANs are two challenges. In this project, we are going to use a simple GAN architecture on a simple fruits image data set (32x32x3); simple model will be trained and developed quickly and allows us to focus more on structure of model architecture and image generation process itself to better understand this process. So, we will try to know more about:

- How to train a discriminator model to learn difference between real and fake images.
- How to train a generator model and a composite of generator and discriminator model.
- How to evaluate performance of GAN.

Fruit 360 Dataset

This data collection contains 82197 images train and test. Training folder contains 61476 image which we just used this folder. This data set contains 120 classes; fruits classes are included:

Apples (different varieties: Crimson Snow, Golden, Golden-Red, Granny Smith, Pink Lady, Red, Red Delicious), Apricot, Avocado, Avocado ripe, Banana (Yellow, Red, Lady Finger), Beetroot Red, Blueberry, Cactus fruit, Cantaloupe (2 varieties), Carambula, Cauliflower, Cherry (different varieties, Rainier), Cherry Wax (Yellow, Red, Black), Chestnut, Clementine, Cocos, Dates, Eggplant, Ginger Root, Granadilla, Grape (Blue, Pink, White (different varieties)), Grapefruit (Pink, White), Guava, Hazelnut, Huckleberry, Kiwi, Kaki, Kohlrabi, Kumsquats, Lemon (normal, Meyer), Lime, Lychee, Mandarine, Mango (Green, Red), Mangostan, Maracuja, Melon Piel de Sapo, Mulberry, Nectarine (Regular, Flat), Nut (Forest, Pecan), Onion (Red, White), Orange, Papaya, Passion fruit, Peach (different varieties), Pepino, Pear (different varieties, Abate, Forelle, Kaiser, Monster, Red, Williams), Pepper (Red, Green, Yellow), Physalis (normal, with Husk), Pineapple (normal, Mini), Pitahaya Red, Plum (different varieties), Pomegranate, Pomelo Sweetie, Potato (Red, Sweet, White), Quince, Rambutan, Raspberry, Redcurrant, Salak, Strawberry (normal, Wedge), Tamarillo, Tangelo, Tomato (different varieties, Maroon, Cherry Red, Yellow), Walnut.

To take pictures, fruits were planted in shaft motor with low speed (3 rpm) and Logitech C920 camera was used to make a short movie of 20 second for train data set (for test images Nexus 5X phone was used). White sheet of paper was used for background. But background of pictures were not uniform because of variations in lighting conditions; so, background of images was extracted with running an algorithm and background of all images were converted to white uniform color. Format of all images are .jpg which is a common format for images. Other specs of Fruit 360 data set were presented in Table 1.

Table 1 Specs of Fruit 360 dataset

	Number of Images
Total number of images	82197
Training set size	61476
Test set size	20618
Multi-fruits set size	103
Number of classes	120
Image size	100x100

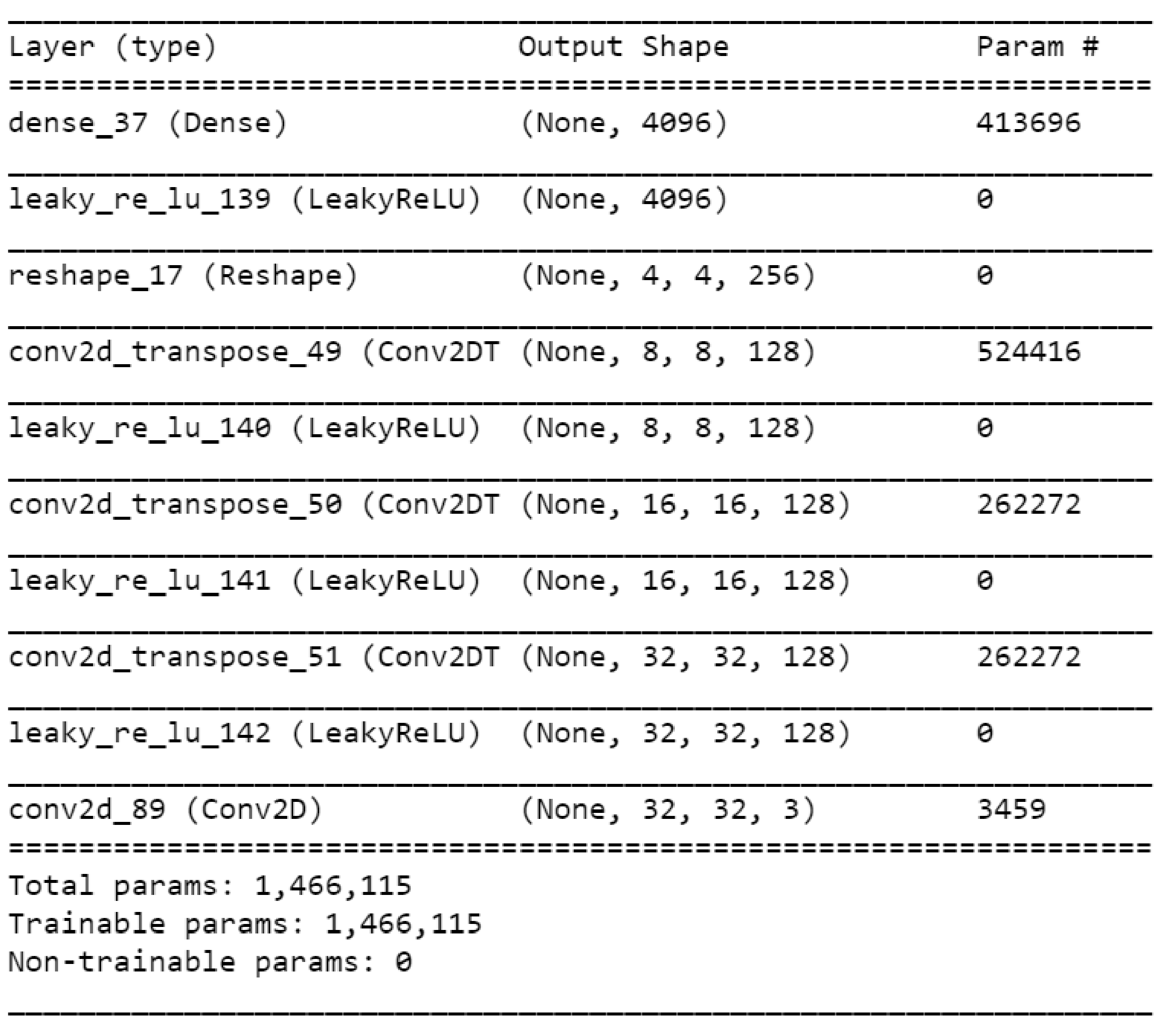
How to train a discriminator model

Discriminator model must take sample images from data set as input and classify them as real or fake images, binary classification.

- **Inputs:** 32x32 pixels size images with three color channel.
- **Outputs:** Binary classification (real or face).

For this project, discriminator model with a normal convolutional layer followed by three convolutional layers to downsample input images was used. The model doesn't have pooling layers and has a sigmoid activation function in output layer to predict a binary result. The model was trained to minimize binary cross entropy loss function for good binary classification result. Discriminator model and parametrizes for input images were defined by *define_discriminator()* function in the program.

Figure 1 shows summarize model architecture contains input and output of each layer. There are aggressive 2x2 stride acts to down-sample input image, first from 32x32 to 16x16, then to 8x8 and more before the model makes an output prediction (without pooling). Also, Dropout, LeakyReLU (instead of ReLU), and Adam version of stochastic gradient descent with momentum of 0.5 were used in this model with learning rate of 0.0002.



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_37 (Dense)             (None, 4096)              413696    
_________________________________________________________________
leaky_re_lu_139 (LeakyReLU)  (None, 4096)              0         
_________________________________________________________________
reshape_17 (Reshape)         (None, 4, 4, 256)         0         
_________________________________________________________________
conv2d_transpose_49 (Conv2DT (None, 8, 8, 128)         524416    
_________________________________________________________________
leaky_re_lu_140 (LeakyReLU)  (None, 8, 8, 128)         0         
_________________________________________________________________
conv2d_transpose_50 (Conv2DT (None, 16, 16, 128)       262272    
_________________________________________________________________
leaky_re_lu_141 (LeakyReLU)  (None, 16, 16, 128)       0         
_________________________________________________________________
conv2d_transpose_51 (Conv2DT (None, 32, 32, 128)       262272    
_________________________________________________________________
leaky_re_lu_142 (LeakyReLU)  (None, 32, 32, 128)       0         
_________________________________________________________________
conv2d_89 (Conv2D)           (None, 32, 32, 3)         3459      
=================================================================
Total params: 1,466,115
Trainable params: 1,466,115
Non-trainable params: 0
_________________________________________________________________
```

Figure 1 Summary of Discriminator Model

This discriminator model classified real example as label one and randomly generate samples as label zero. Input image data set for this model was scaled the pixel values from range of [0, 255] to normalized range of [-1, 1]; off course, generator model was generated images with pixel values in range [-1, 1] by using tanh activation function. The model was updated in batches with a collection of real images and a collection of generated images. *generate_real_samples()* function in the program took all images and selected a random sample images for batches with class labels (class label of 1, to indicate real images).

For first step, we checked the discriminator model without using generator model; so, we generated images comprised of random pixel values, specifically random pixel values in range [0, 1], then scaled it to [-1, 1], the same as real images (*generate_fake_samples()* function implemented this behavior in our program). Then, we trained the discriminator model. This process involved repeatedly retrieving samples of real images and samples of generated images and updating for a fixed number of iterations (we ignored epochs in this step and we just fit mode for a fixed number of batches). The *train_discriminator()* function implemented by using a batch size of 128 image (64 real and 64 fake) for each iteration. So, this simple process showed us how discriminator model is performing over time in small scales.

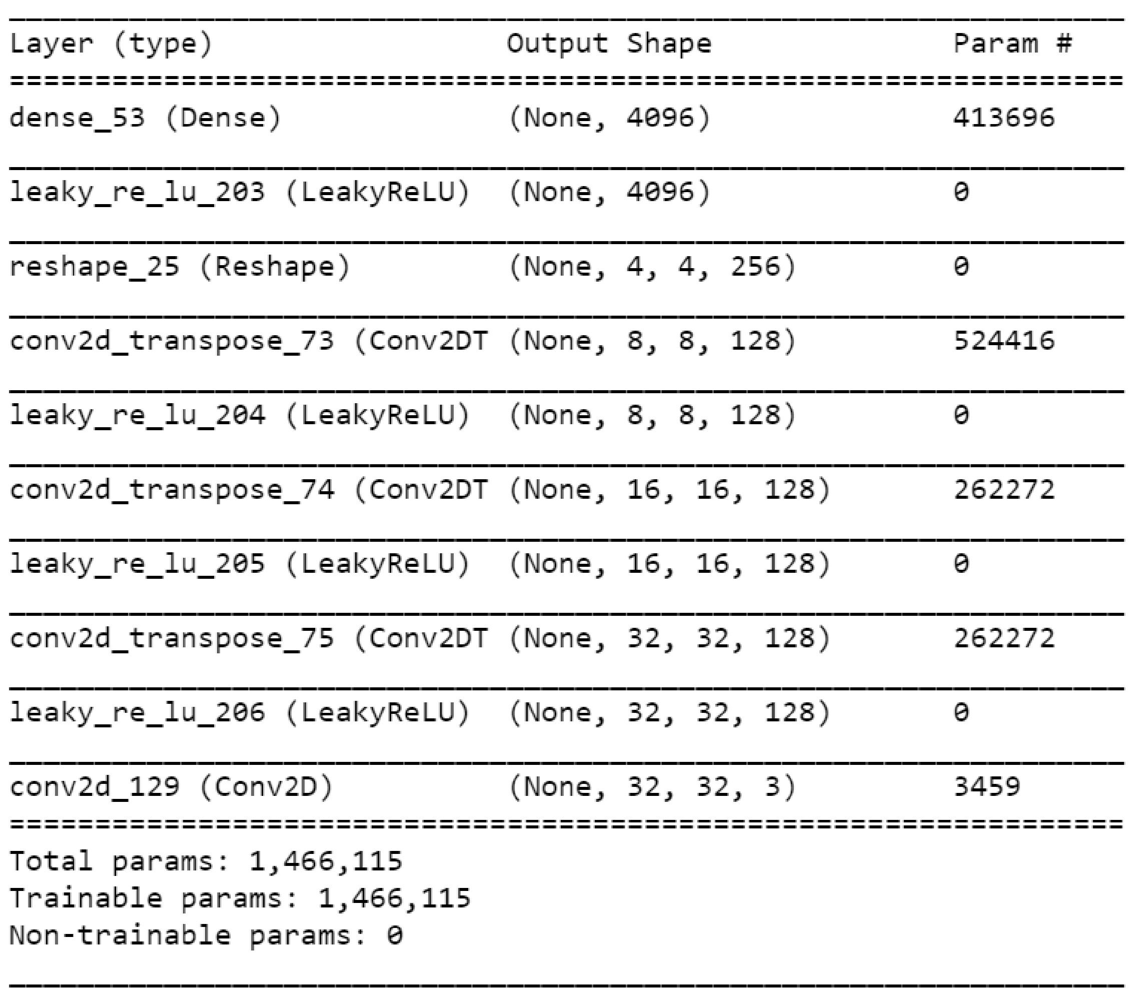
How to Use the Generator Model

The generator model generated fake images of objects by taking a point from latent space as input. The latent space is an arbitrary vector space of Gaussian Distributed Values; for instance, 100 dimensions. It have special meaning; but, by drawing random points from this space and providing them for generator model during training, the generator model used them to represent a compressed representation of the output space.

- **Inputs:** point in latent space, for example, 100 element vector of Gaussian Distributed Values.
- **Outputs:** two-dimensional square color image (3 channels) of 32x32 pixels (between[-1, 1])

Therefore, we had to transform a vector from the latent space with 100 dimensions to a 2D array with 32x32x3 values by using the generator model. There are a number of ways to achieve this gold; but, we used Dense layer as first hidden layer contains 256*4*4 nodes because (first) we wanted to use many parallel versions to interpret of input, (second) input images should be low-resolution version of output images, and (third) Dense layer need enough nodes, such as 256, to multiple versions of our output image. Then, we used reshaping to convert the Dense layer output into something image-like to pass next layer (256*4*4 nodes reshaped to 4x4x256).

A major innovation for generator model was unsampling the low-resolution image to a high-resolution version images, which there are two common ways to do this unsampling: using *UpSampling2D* layer (like reverse pooling layer) followed by a normal *Conv2D* layer. Other way is *Conv2DTranspose* which is combination of those two operation into a single layer (we used second way for this project). We put two more *Conv2DTranspose* layers to achieve 32x32 output image size with *LeakyReLU* with a default slope of 0.2, reported as a best practice when training GAN models in other research articles. Finally, output layer was *Conv2D* with three channels and “same” padding to preserve output dimensions at 32x32x3 pixels and *tanh* activation was best because we need to have output in [-1,1] range. The *define_generator()* function was used to create a generator model for our research (Figure 2 summarizes the layers of generateor model).



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_53 (Dense)             (None, 4096)              413696
_________________________________________________________________
leaky_re_lu_203 (LeakyReLU)  (None, 4096)              0
_________________________________________________________________
reshape_25 (Reshape)         (None, 4, 4, 256)         0
_________________________________________________________________
conv2d_transpose_73 (Conv2DT (None, 8, 8, 128)         524416
_________________________________________________________________
leaky_re_lu_204 (LeakyReLU)  (None, 8, 8, 128)         0
_________________________________________________________________
conv2d_transpose_74 (Conv2DT (None, 16, 16, 128)       262272
_________________________________________________________________
leaky_re_lu_205 (LeakyReLU)  (None, 16, 16, 128)       0
_________________________________________________________________
conv2d_transpose_75 (Conv2DT (None, 32, 32, 128)       262272
_________________________________________________________________
leaky_re_lu_206 (LeakyReLU)  (None, 32, 32, 128)       0
_________________________________________________________________
conv2d_129 (Conv2D)          (None, 32, 32, 3)         3459
=================================================================
Total params: 1,466,115
Trainable params: 1,466,115
Non-trainable params: 0
_________________________________________________________________
```

Figure 2 Summary of Generator Model

To check this model works well, new points in latent space were generated with *randn()* function of NumPy and they reshaped into n rows with 100 elements per row (The *generate_latent_points()* function used for this purpose in the program). Also, the *generate_fake_samples()* function was used to return generated sample which was associated class labels. Then, we could plot generated samples as we did it with Fruit images example in the previous section (Figure 3 contains 49 images). As the model was not trained, the generated image in Figure 3 were completely random pixel values which rescaled to [0, 1]. They are look like a mess of gray images.

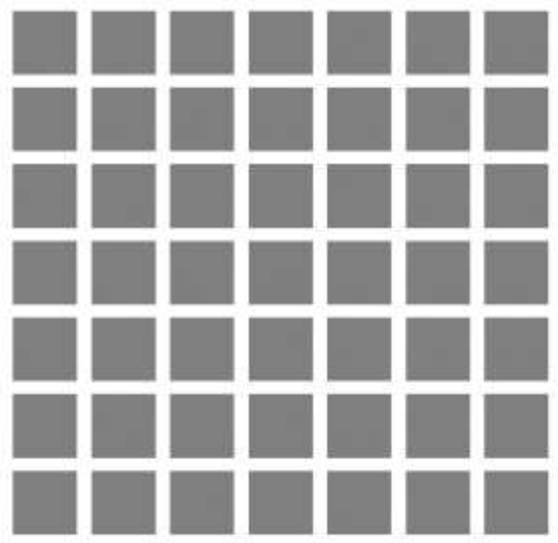


Figure 3 contains 49 images which generated before any training (just to check generator and discriminator models work well)

Results

Overall, there is not an objective way to evaluate performance of a GAN model because there isn't a method to calculate objective error to generate images. But, image must subjectively evaluated for quality by operator, it means there isn't an exact stop point for training and it's completely depending on operator. Therefore, we saved the model and displayed some sample of generated images after each 10 epochs which shows how much generated image is "good enough" for our purpose. Figure 4-8 show a progress of improving to generate images based on Fruit images data set. We can distinguish kinds of fruit in images after 200 epochs which took about 10 hours to compute with CPU core i7-8550U and 8G RAM. This process can continue to train the GAN model over many epochs, hundreds or thousands of epochs, to get better and better images; but, it's not purpose of this research.

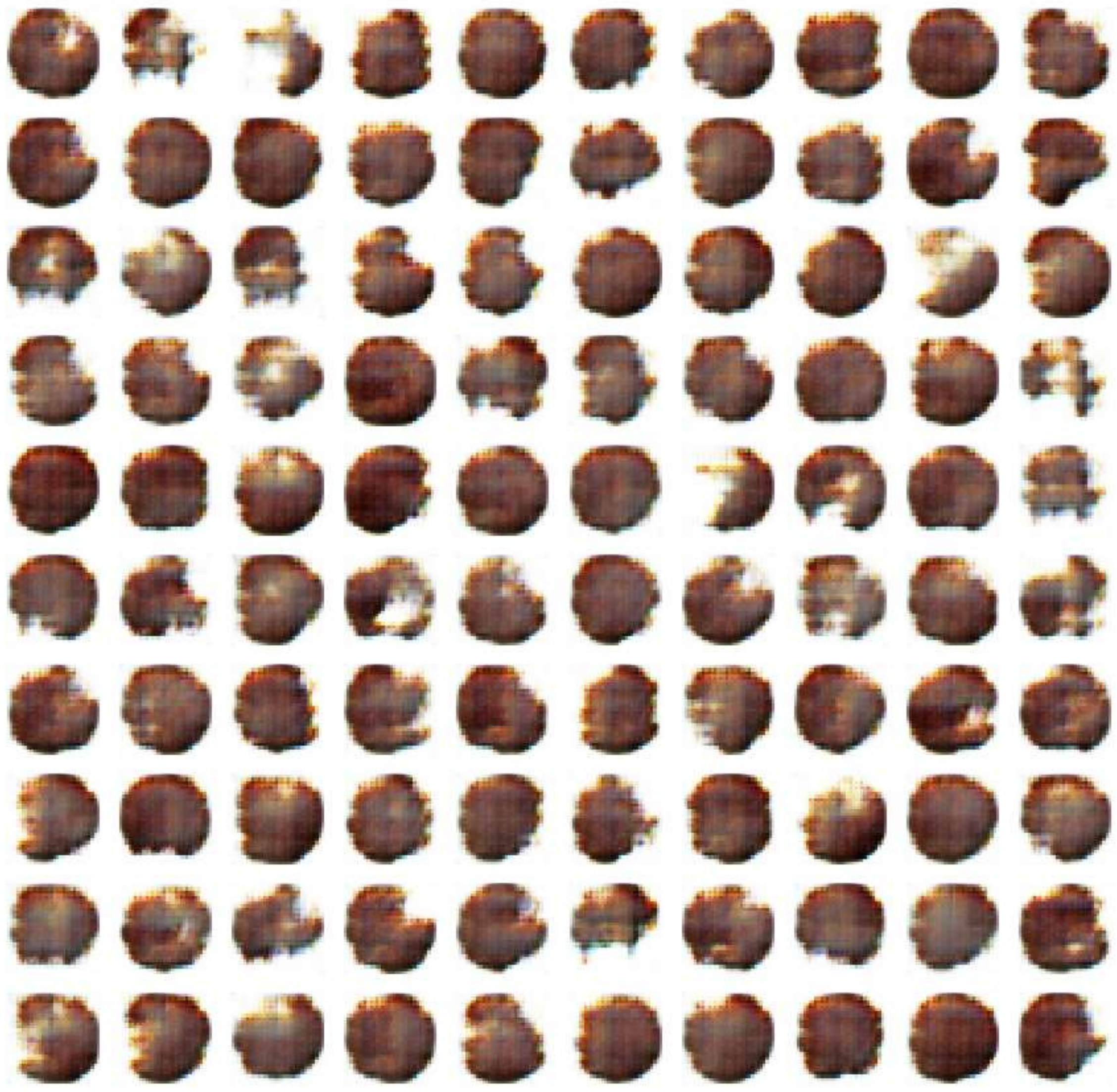


Figure 4 Generated images after 10 epochs

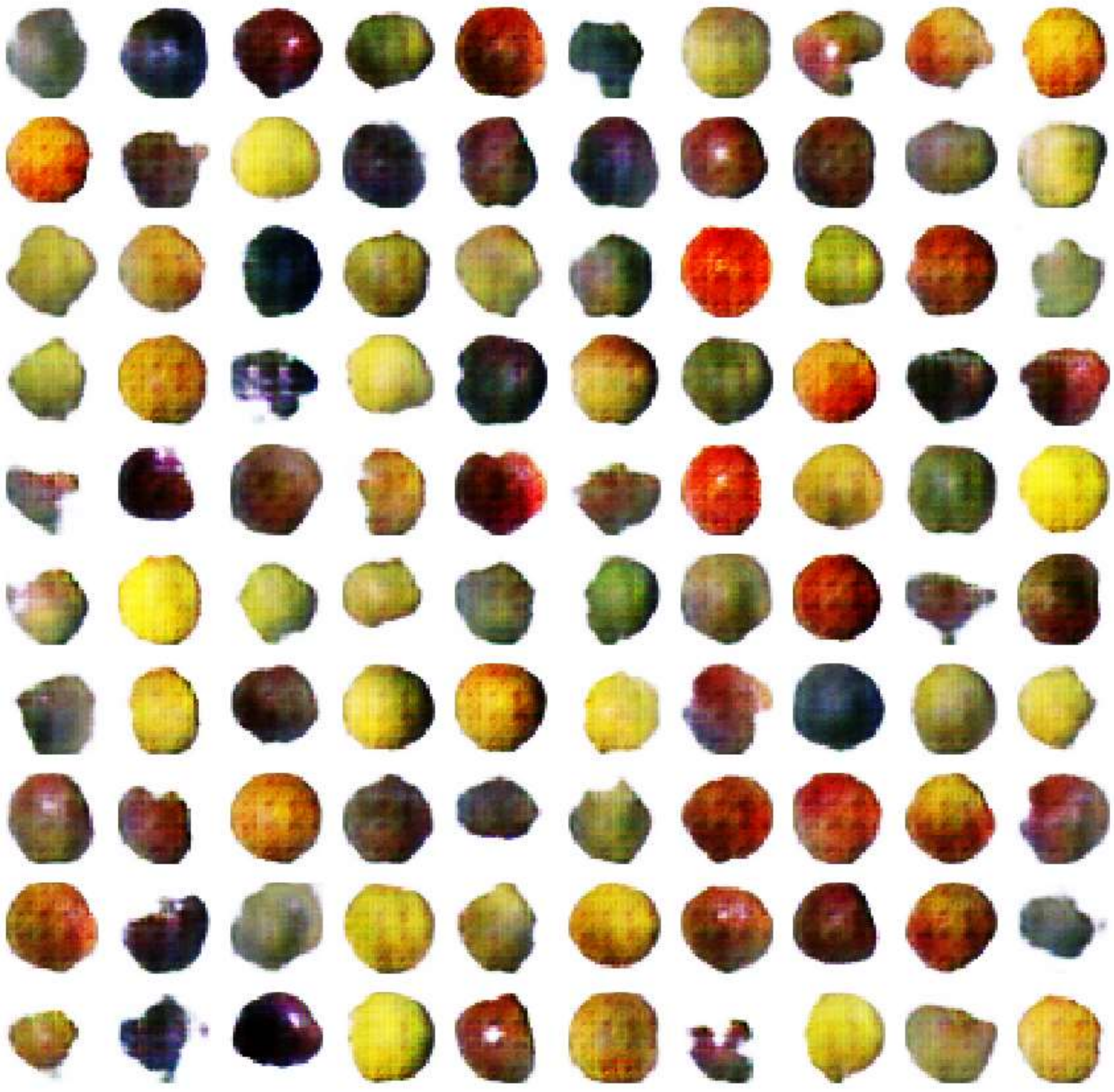


Figure 5 Generated images after 50 epochs

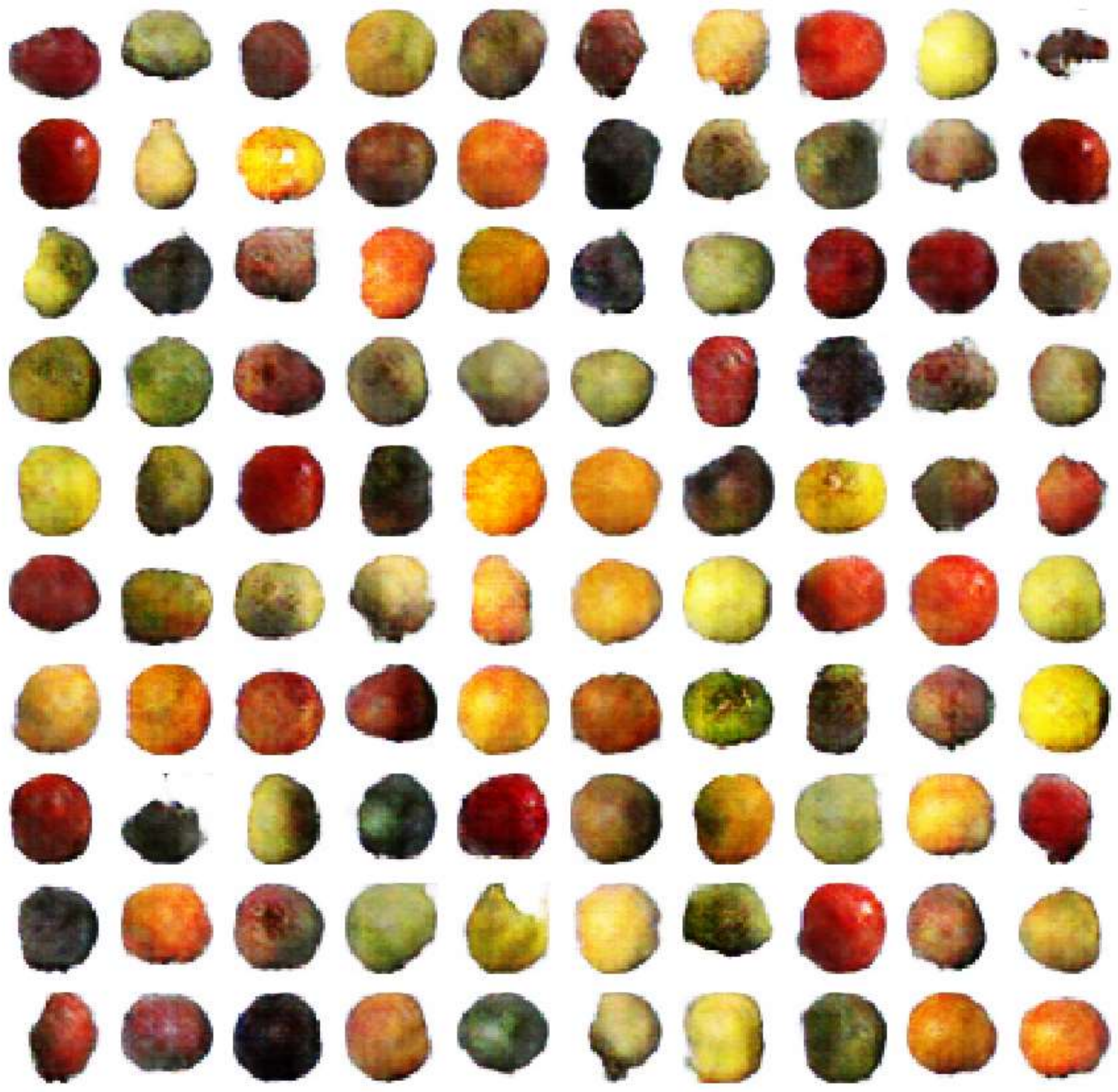


Figure 6 Generated images after 100 epochs

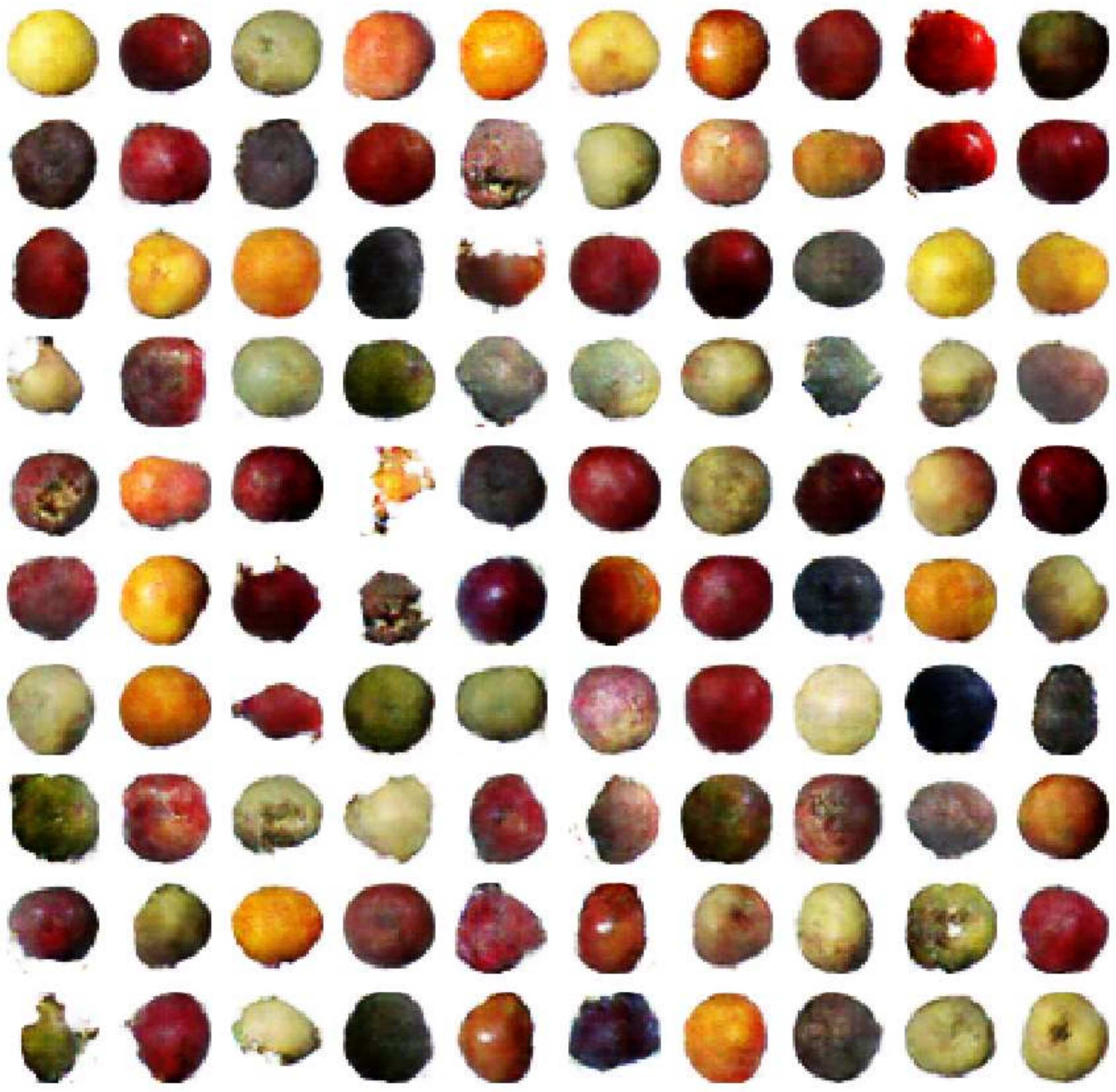


Figure 7 Generated images after 150 epochs

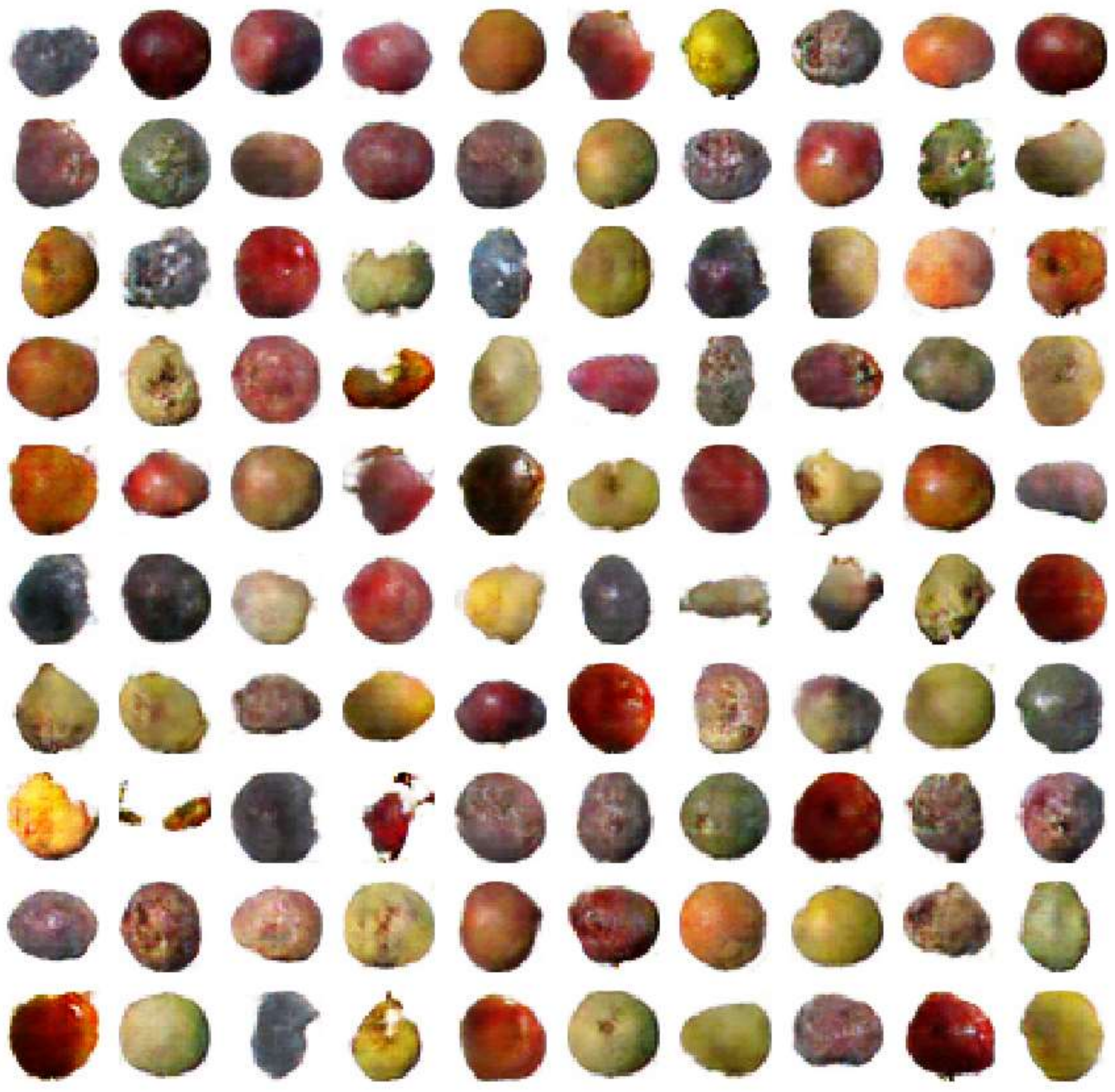


Figure 8 Generated images after 200 epochs

Reference

1. https://machinelearningmastery.com/how-to-develop-a-generative-adversarial-network-for-a-cifar-10-small-object-photographs-from-scratch/
2. https://www.kaggle.com/moltean/fruits