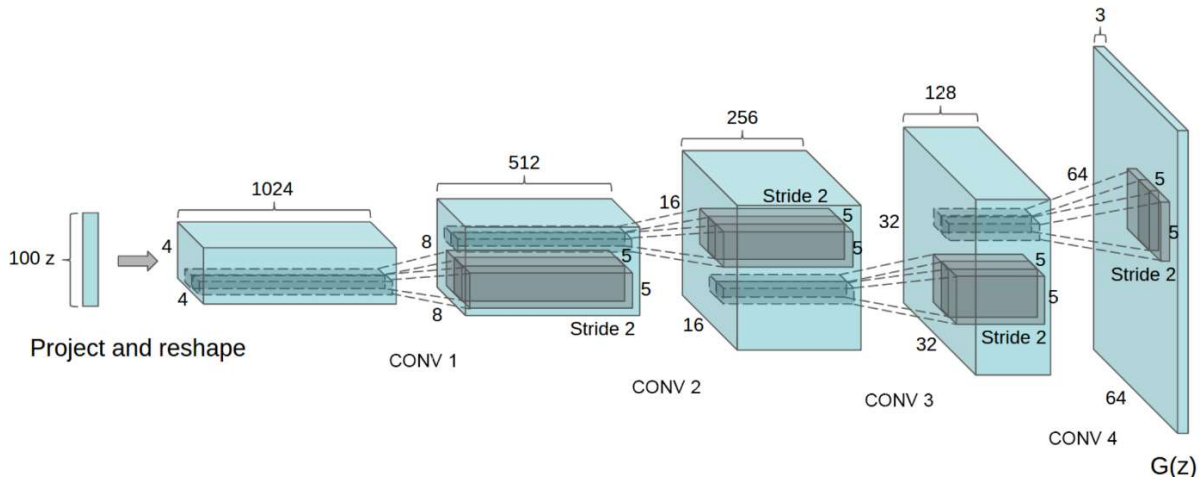
DEEP CONVOLUTIONAL GENERATIVE ADVERSARIAL NETWORKS (DCGANs)

DCGAN is one of the popular and successful network design for GAN. It mainly composes of convolution layers without max pooling or fully connected layers. It uses convolutional stride and transposed convolution for the downsampling and the upsampling. The figure below is the network design for the generator.



DCGAN model structure

THE CIFAR10 DATASET

The CIFAR-10 dataset (Canadian Institute For Advanced Research) is a collection of images that are commonly used to train machine learning and computer vision algorithms. It is one of the most widely used datasets for machine learning research.

The CIFAR-10 dataset contains 60,000 32x32 color images in 10 different classes. The 10 different classes represent airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks. There are 6,000 images of each class.

Examples of CIFAR10 images

Importing required libraries, functions and dataset

From keras, we import the required layers (Dense, Conv2D, Conv2Dtranspose) and also import Sequential from keras.models for building sequential models.

We also import numpy functions for mathematical operations and pyplot from matplotlib for visualization.

```
[ ]  # libraries and functions to be used

     from numpy import expand_dims
     from numpy import zeros
     from numpy import ones
     from numpy import vstack
     from numpy.random import randn
     from numpy.random import randint
     from keras.datasets.cifar10 import load_data
     from keras.optimizers import Adam
     from keras.models import Sequential
     from keras.layers import Dense
     from keras.layers import Reshape
     from keras.layers import Flatten
     from keras.layers import Conv2D
     from keras.layers import Conv2DTranspose
     from keras.layers import LeakyReLU
     from keras.layers import Dropout
     from matplotlib import pyplot
     from keras.utils.vis_utils import plot_model
```

Activation Functions:

All the Conv2D layers use the LeakyReLU activation with alpha set to 0.2. This is defined as best practice for DCGANs and is the recommended activation function.

Using the normal ReLU activation may lead to the model falling into a state termed the 'dying state' where it outputs nothing but zeros for any input. Hence, LeakyReLU serves as a better choice.

The output Dense layer uses a sigmoid activation function to output probability of an image being real (1) or fake (0).

Optimizer:

The optimizer used is Adam, recommended for DCGANs. The Adam optimizer offers better performance than other optimizers such as rmsprop, which led to slower training time and inferior image quality when used with this model.

Further, we also use a learning rate of 0.0002 and a momentum of 0.5 as per the recommended parameters.

Loss Function:

As the discriminator is a binary classifier which classifies images into two categories (real or fake), we use Binary Cross-Entropy (BCE) loss.

Activation Functions:

All the Conv2DTranspose layers and the input layer use the LeakyReLU activation with alpha set to 0.2. This is defined as best practice for DCGANs and is the recommended activation function.

Using the normal ReLU activation may lead to the model falling into a state termed the 'dying state' where it outputs nothing but zeros for any input. Hence, LeakyReLU serves as a better choice.

The output Conv2D layer uses a tanh activation function to output values between -1 and 1.

The generator is not trained directly. It is trained within the gan model using feedback from the discriminator. Hence, the generator is not compiled here.

Activation Functions:

The Leaky ReLU activation function, with alpha set to 0.2, is used in this model.

Optimizer:

The optimizer used is Adam, recommended for DCGANs. The Adam optimizer offers better performance than other optimizers such as rmsprop, which led to slower training time and inferior image quality when used with this model. Further, we also use a learning rate of 0.0002 and a momentum of 0.5 as per the recommended parameters.

Loss Function:

The output of the GAN is the output of the discriminator, that is, a binary classification label for real (1) or fake (0) images. Hence, we use Binary Cross-Entropy (BCE) loss here.

Training the GAN

Now, we train the GAN model.

- We run the process for 200 epochs, and set the batch size to 128 images.
- In each iteration, we draw 64 real images from the dataset, and use the generator to produce 64 fake images.
- The discriminator is trained separately using these real and fake images, with the correct labels.
- Then, we generate 128 random latent space points to be used as input to the GAN (and thereby, the generator). We also pass in 128 real labels (1) as we want the model to generate images that the discriminator identifies as real.
- We pass in the points and labels to the GAN and train it.
- We print the loss for the discriminator on real images, the loss for the discriminator on fake images, and the loss for the GAN in each iteration.
- We also save the generated images after every 10 epochs for comparison.

Training Process:

Initially, we train the discriminator two times, once with a set of real images and 1 labels and again with a set of fake generated images and 0 labels.

Once this is complete, we generate a set of latent points, and pass them into the GAN. We also pass in real labels (1s).

Here, the generator receives the latent points as input, and attempts to generate replications of CIFAR10 images. These generated images are then passed into the discriminator.

The discriminator assesses these images and outputs the probability of them being real (1) or fake (0).

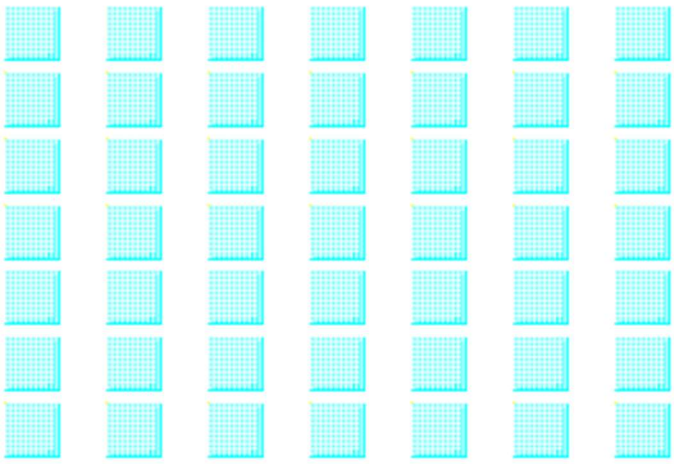
The GAN uses this label to compare against the input labels (all real labels) and thus computes the loss.

As the discriminator cannot be trained from within GAN, it's parameters are unchanged. The generator, meanwhile, uses the computed loss to adjust it's parameters in a direction such that the discriminator becomes more likely to label it's generated images as real (1).

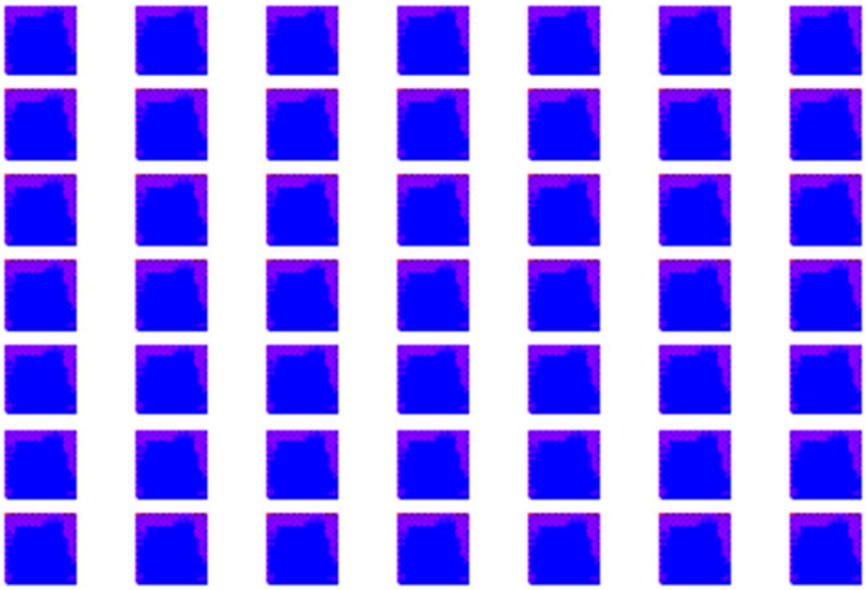
In this manner, the generator learns and improves it's performance across each epoch, using feedback from the discriminator.

Generated Images

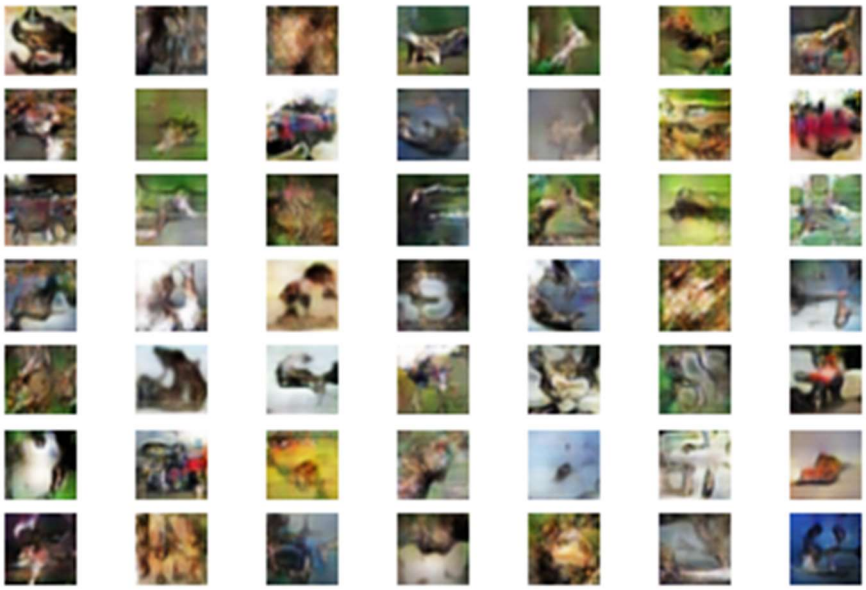
After 10 epochs



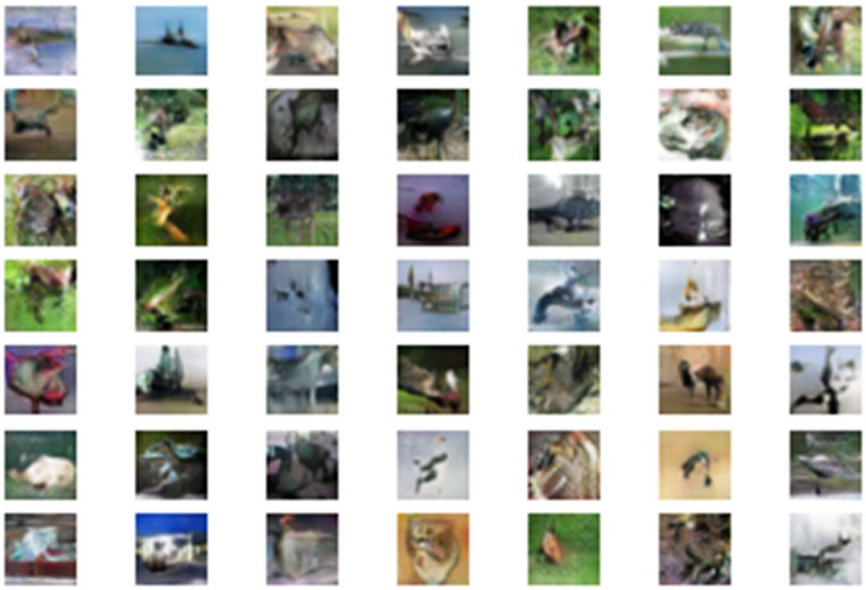
After 20 epochs



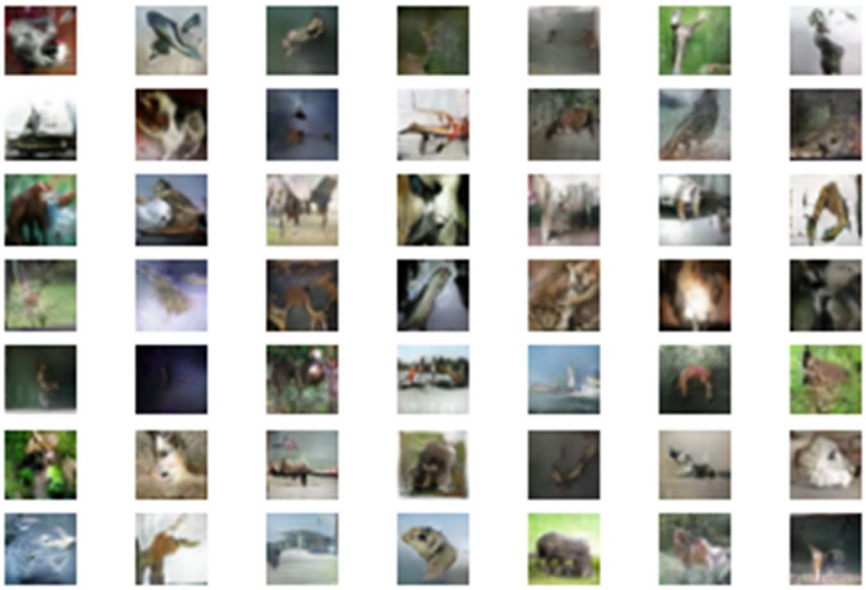
After 50 epochs



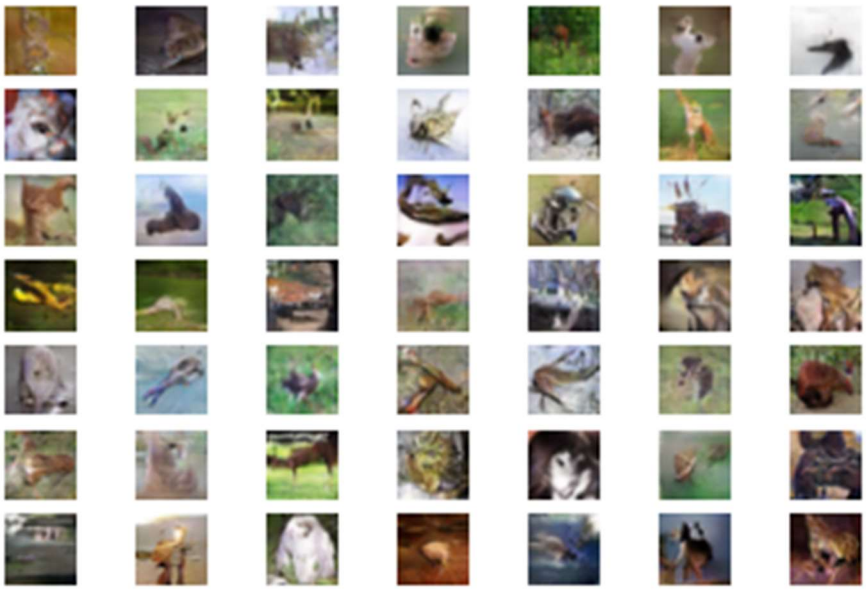
After 100 epochs



After 150 epochs



After 200 epochs



- We observe that after a fair number of epochs, the generator starts generating images that are very similar to the images of the dataset.
- After 200 epochs, the images generated are of very high quality.
- This trained generator can then be used on its own to generate Cifar10 image replications by simply passing in a latent space vector.
- This GAN architecture can be further improved by adopting various distance scheme based loss functions such as Wasserstein loss.

Errors encountered and other observations:

The complete model takes around 3 hours to run on GPU (we used Google Colab for execution), and as such was susceptible to network failures and runtime limits.

With regards to the model architecture, we experimented with smaller hidden layers in both the discriminator and generator (by using less number of nodes) but this sometimes led to a marked decrease in image quality after generation. We also used different activation functions and optimizers to analyse their effects on performance. Using a ReLU activation led to a slight but not significant loss of image quality as compared to the LeakyReLU used here. On the other hand, using RMSProp as the optimizer led to a great decrease in the quality of generated images, with the results after 100 epochs of GAN training still lacking distinct features resembling the Cifar10 images.

We also observed that the model performance is not affected by swapping the tanh activation in the output of the generator with a sigmoid function, when accompanied with appropriate changes to the data scaling used.