Generating Images using a DCGAN

Anonymous author

Abstract

This paper uses a Deep Convolutional Adversarial Network DCGAN to generate new images at a resolution of 64x64. The training datasets being used will be CIFAR-10 and STL-10, these contain low quality square images of animals and vehicles (STL10 has higher quality images). The DCGAN used is a unsupervised learning method and yields realistic output images.

1 Methodology

1.1 DCGAN Explanation

A basic generative adversarial network (GAN) is a framework used to model a training data sets distribution so that we can generate data with the same distribution. They are built out of two separate models, a generator and a discriminator.
The Generator is used to generate fake images that are meant to mimic the training images. Its job is to repeatedly create fake images until they have the same distribution as the training data and the discriminator can not tell the difference. G(Z) is the generator function which maps z (the latent vector sampled from a standard normal distribution) to the data-space.
The Discriminator takes an image and output if it is a real image from the training set or a fake image from the generator. The discriminator works to get better at telling images apart and the generator tries to improve at fooling the detector. Using this minimax game iteratively improves both the generator and the discriminator until we have realistic fake images and a very accurate discriminator by slowly changing eaches weights using back propagation. D(x) is the discriminator network, x is an image of size 3x64x64 and the output is the probability of an that image being real. Equation 1 is the basic GAN methodology

$$GminDmaxV(D,G) = Ex \sim pdata(x)[logD(x)] + Ez \sim pz(z)[log(1 - D(G(z)))] \quad (1)$$

A DCGAN builds upon the standard GAN by uing convolutional and convolutional-transpose layers in the generator and discriminator networks as proposed by Radford [**paper1**]. These layers help with the flow of gradients during training The discriminator network consists of strided convolution layers, batch normalization layers, and LeakyRelu as the activation function. The generator network is built up of convolutional-transpose layers, batch normalization layers, and ReLU activations.

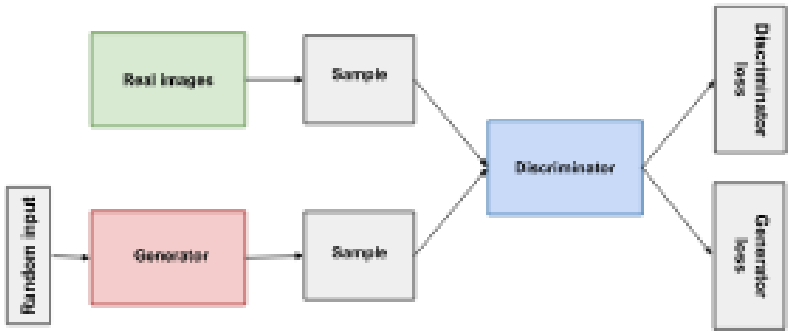


Figure 1. Diagram of how the Generator and the Discriminator interact with each other

1.2 Training Process

To train the model we do the following process, for a given number of epochs we go through a number of batches of training data (in ours batch size is 128 images). The generator and discriminator can be trained separately or at different intervals, but in this implementation both are trained simultaneously using the most recent generator output to train the discriminator. We start the training by initialising the weights for the model, these are randomly selected from a normal disribution using a mean of 0.00 and standard deviation of 0.02. The Generator and the Discriminator models are also initialised.

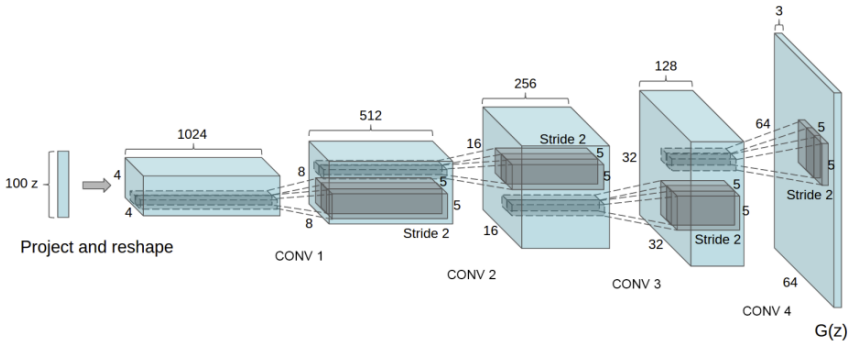


Figure 2. Image of the Generator network G(z) layers from the original DCGAN paper

Next we setup our loss function used to calculate the loss during training. For the DCGAN it is recommended to use the Binary Cross Entropy Loss function (BCEloss) which is defined as

$$\ell(x, y) = L = [l1, \ldots, lN]^T, ln = -[yn \cdot logxn + (1 - yn) \cdot log(1 - xn)] \quad (2)$$

Note that we can choose which part of the function to calculate by changing y between 0 and 1. Adam optimizers are used for both the discriminator and the generator with a learning rate = 0.0002, beta = 0.05 and gamma = 0.999 as specified in the paper. Now that the entire DCGAN framework is defined, the main deep learning training loop begins:
For a given number of epochs (e.g. 1000):

1. Collect a batch of real image samples from the training set and forward pass these through the Discriminator network. Calculate the loss of this using log(D(x)) and calculate the gradients using back propagation.
2. Create a batch of fake image samples using the generator and forward pass these through the Discriminator network. Calculate the loss of this using log(1-D(G(z))) and calculate the gradients using back propagation.
3. Using the accumulated gradients let the discriminator optimizer take a step, this will update the discriminators parameters and network layers.
4. Classify the generators fake images using the discriminator and calculate the loss of this using log(1-D(G(z))).
5. Also calculate the loss of the generator when passing forward on real images using log(D(x)). This compares the real distribution with the current distribution of the generator network.
6. Use back propagation to calculate G's gradients and update the generators network parameters using an optimizer step.

To test our model/generate images, we pass our generator a sample of fixed noise. It will pass this through the network and output a 3x64x64 image. This is used after each epoch and after sufficient training to track progress of the deep learning model and render the models performance results. The output images of DCGAN begin as normally distributed random noise and over many iterations they refine to become realistic looking images which

the discriminator can detect at a rate of 0.5 (this level means that the generator and discriminator are at equilibrium which helps training). Below details the actual sequential networks for both the generator and the discriminator models used in the implementation

2 Results

The results below are 64 non cherry-picked samples from both CIFAR10 and STL10. The images themselves are fairly clear and many objects can be pointed out clearly such as animal bodies and parts of vehicles. The results are positive and took many hours to train on each model, below are some random samples of each. These were generated through the google colab premium GPU and the images were incrementally exported to google drive so the learning process could be monitored.

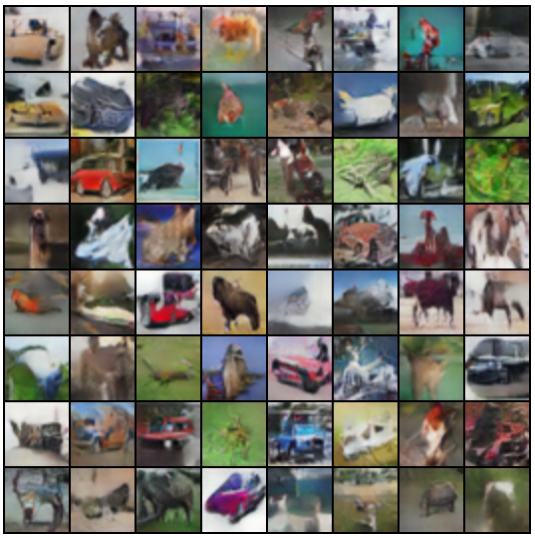


Figure 3. Random samples generated from the CIFAR10 data set at 64x64 resolution

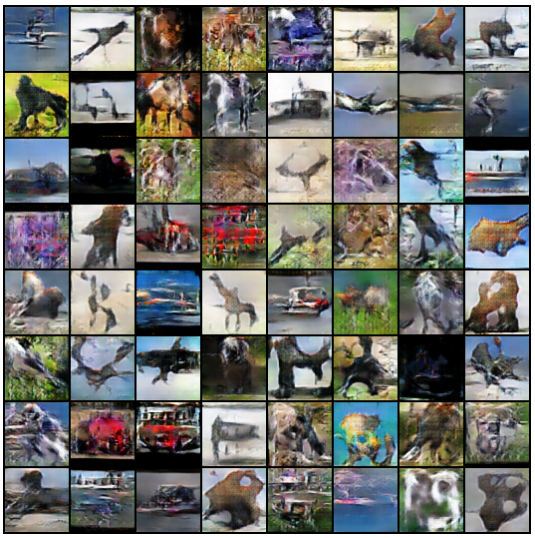


Figure 4. Random image samples generated from the STL10 data set at 64x64 resolution

The images show a lot of variation in the output and despite some being blurry/unrealistic it is usually still clear that the image is a real object. Even longer training could possibly help and more training images may benefit the model, especially for STL10.

The next results show images generated by interpolating between points in the models latent space:

Figure 3. Random interpolations generated from the CIFAR10 data set at 64x64 resolution

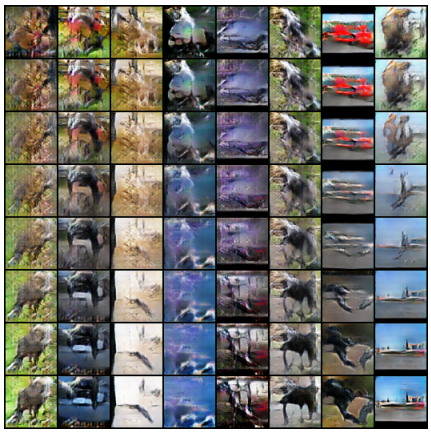
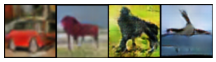


Figure 4. Random interpolations generated from the STL10 data set at 64x64 resolution

The interpolation results are quite impressive and show good exploration of the latent space with little difference between each image in the sequence. The STL10 data set performed particularly well on these and most of the output images were realistic even when the start and end images were not. And here are some cherry-picked samples that show the best outputs the model has generated:



3 Limitations

Many of the results have the shape and features of the training data but fall short of being realistic, some are also too similar to the training data images. A diffusion model would be a better deep learning method for generating realistic images with these data sets. In addition more training time with powerful GPU's would improve the images but they were limited by the available GPU's and time for training.

4 Bonuses

The bonus for this paper is -4. (GAN -8, STL10 at 64x64 +4)

5 REFERENCES

[1] UNSUPERVISED REPRESENTATION LEARNING WITH DEEP CONVOLUTIONAL GENERATIVE ADVERSARIAL NETWORKS, Alec Radford Luke Metz, 2016

[2] https://github.com/Yangyangii/GAN-Tutorial/blob/master/CelebA/BEGAN.ipyn

[3] https://github.com/Ksuryateja/DCGAN-CIFAR10-pytorch/blob/master/gan$_c ifar.py$

[4] https://github.com/pytorch/examples/tree/main/dcga