E0 250 - Deep Learning

Project 4 - Generative Adversarial Networks

Deepesh Virendra Hada
M.Tech, Department of CSA
SR: 17196

15 May, 2020

1 Data

In this project, we use the **CIFAR-10** dataset. The CIFAR-10 dataset consists of 60000 32x32 colour images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images.

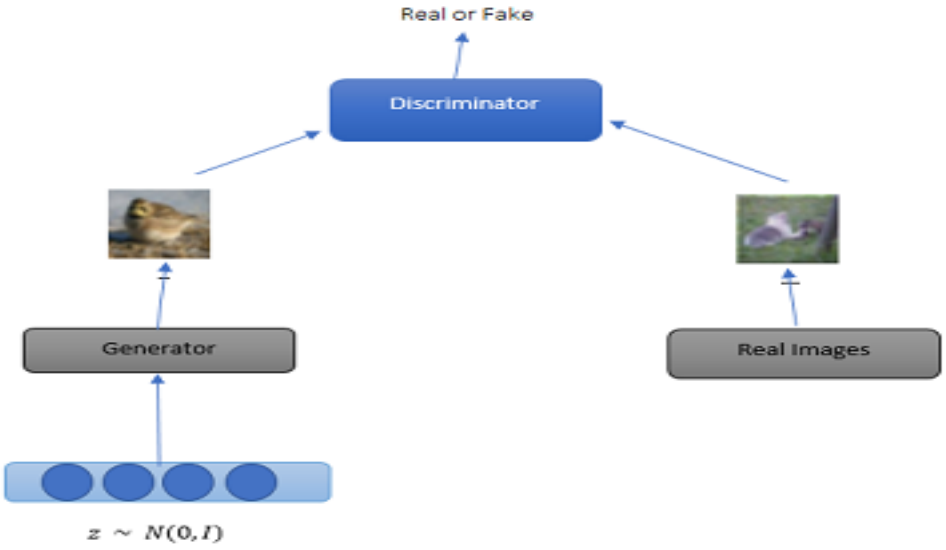
The dataset is divided into five training batches and one test batch, each with 10000 images. The test batch contains exactly 1000 randomly-selected images from each class. The training batches contain the remaining images in random order, but some training batches may contain more images from one class than another. Between them, the training batches contain exactly 5000 images from each class.

There are 10 classes in the dataset: *airplane*, *automobile*, *bird*, *cat*, *deer*, *dog*, *frog*, *horse*, *ship*, and *truck*

Also, the classes are completely mutually exclusive. There is no overlap between *automobiles* and *trucks*.

2 Task

The task is to generate new images which resemble the images from the CIFAR-10 dataset. Following is the Generative Adversarial Network approach for doing this:



From the training set of CIFAR-10, we implement two neural networks: the Generator and the Discriminator.

GANs are a framework for teaching a DL model to capture the training data's distribution so we can generate new data from that same distribution. GANs were invented by Ian Goodfellow in 2014 and first described in the paper Generative Adversarial Nets. They are made of two distinct models, a generator and a discriminator.

The job of the generator is to spawn 'fake' images that look like the training images, in our case, images from the CIFAR-10 dataset. The job of the discriminator is to look at an image and output whether or not it is a real training image or a fake image from the generator.

During training, the generator is constantly trying to outsmart the discriminator by generating better and better fakes, while the discriminator is working to become a better detective and correctly classify the real and fake images. The equilibrium of this game is when the generator is generating perfect fakes that look as if they came directly from the training data, and the discriminator is left to always guess at 50

The two actually play a minmax game until they reach an equilibrium (the Nash equilibrium).

GANs are however, infamous for their notoreity in training. The discriminator usually learns much faster than the generator, and the generator lags behind. A stable training for GANs was suggested in the Deep Convolutional GAN (DC-GAN) paper.

We first look at the implementation of the DC-GAN, its limitations, and see how they are overcome by SA-GANs and Wasserstein distance.

3 Deep Convolutional GAN (DC-GAN)

In this section, we discuss the architecture of the generator and the discriminator as specified in the DC-GAN paper.

A DCGAN is a direct extension of the GAN described above, except that it explicitly uses convolutional and convolutional-transpose layers in the discriminator and generator, respectively. It was first described by Radford et. al. in the paper *Unsupervised Representation Learning With Deep Convolutional Generative Adversarial Networks*.

The discriminator is made up of strided convolution layers, batch norm layers, and LeakyReLU activations. The input to the discriminator is a 3x64x64 input image and the output is a scalar probability that the input is from the real data distribution.

The generator is comprised of convolutional-transpose layers, batch norm layers, and ReLU activations. The input is a latent vector, z, that is drawn from a standard normal distribution and the output is a 3x64x64 RGB image. The strided conv-transpose layers allow the latent vector to be transformed into a volume with the same shape as an image.

In the paper, the authors also give some tips about how to setup the optimizers, how to calculate the loss functions, and how to initialize the model weights.

The deep learning library, *PyTorch*, has been used to implement the architecture.

3.1 Data and Transforms

We load the CIFAR-10 dataset through PyTorch's *torchvision* package. To train the generator and the discriminator, we'll be only using the training set of the dataset, which contains $50K$ labelled training examples. However, these labels are of little importance to us because of the task at hand. The task here is to generate new images, and not classify an image into one of the 10 classes.

The original images are of the dimension 32×32, and since they are coloured, there are 3 channels associated with each image. Since this resolution is very small to visualize, and for the generated images, visualization would be even harder, we resize the image to 64×64 dimension. Also, we'll generate images which have a 64×64 dimension. Note that the number of channels would remain the same even after this tranformation.

While loading the images, we convert them to Tensors, which would be a requirement for our Generator model.

3.2 Weight Initialization

From the DCGAN paper, the authors specify that all model weights shall be randomly initialized from a Normal distribution with $mean = 0$, $stdev = 0.02$. We've defined a *weights_init* function that takes an initialized model as input and reinitializes all the convolutional, convolutional-transpose, and batch normalization layers to meet this criteria. This function is applied to the models immediately after their respective initializations.

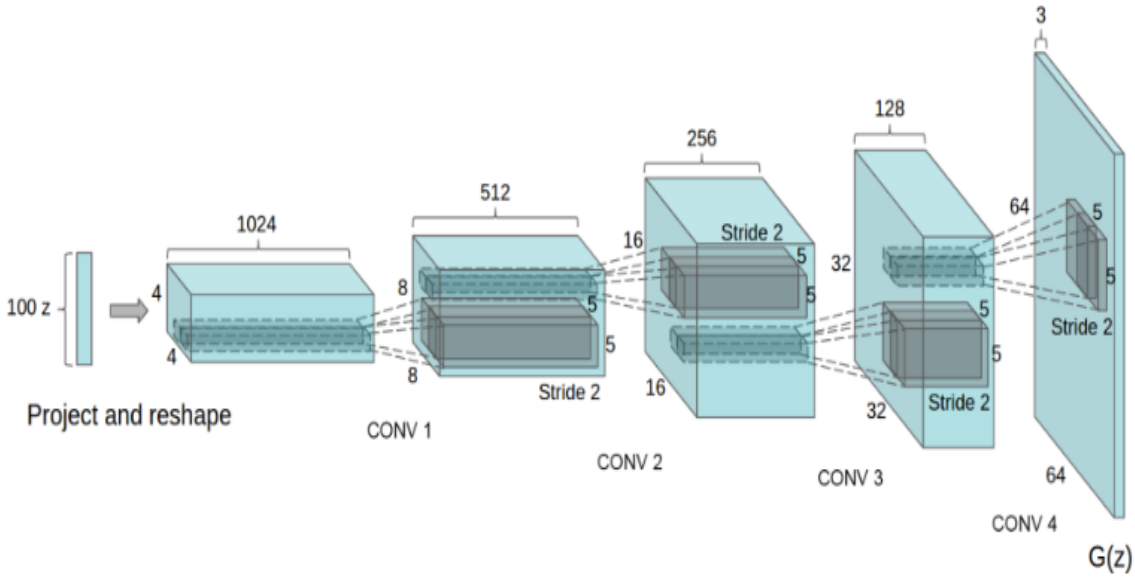
3.3 Network Architecture - Generator

We first discuss the neural network architecture of the Generator in the implemented model.

The generator, G, is designed to map the input latent space vector, z to data-space. Since our data are images, converting z to data-space means ultimately creating an RGB image with the same size as the training images, i.e., the resized images, 3x64x64. In practice, this is accomplished through a series of strided two dimensional convolutional transpose layers, each paired with a 2D batch normalization layer and a ReLU activation. The output of the generator is fed through a *tanh* function to return it to the input data range between -1 and 1.

It is worth noting the existence of the batch norm functions after the conv-transpose layers, as this is a critical contribution of the DCGAN paper. These layers help with the flow of gradients during training.

An image of the generator from the DCGAN paper is shown below:



The input to the generator is a random noise vector, z, sampled from a standard normal distribution. The paper suggests to use a 100-dimensional noise vector, z.

The *Project and Reshape* operation can be performed by the convolution-transpose operation itself. For this, instead of treating z as a 100-dimensional vector, we treat it as a $100 \times 1 \times 1$ tensor.

Notice, how the inputs we set in the input section influence the generator architecture in code. The output of the generator is $3 \times 64 \times 64$ image, where there are 3 color channels.

After the generator is built, we can instantiate it by applying the *weights_init* function.

3.4 Network Architecture - Discriminator

The discriminator is made up of strided convolution layers, batch norm layers, and LeakyReLU activations.

As mentioned earlier, the discriminator, D, is a binary classification network that takes an image as input and outputs a scalar probability (sigmoid!) that the input image is real (as opposed to fake). Here, D takes a 3x64x64 input image, processes it through a series of Conv2d, BatchNorm2d, and LeakyReLU layers, and outputs the final probability through a Sigmoid activation function.

This architecture can be extended with more layers if necessary for the problem, but there is significance to the use of the strided convolution, BatchNorm, and LeakyReLUs.

The DCGAN paper mentions it is a good practice to use strided convolution rather than pooling to downsample because it lets the network learn its own pooling function. Also, batch norm and leaky relu functions promote healthy gradient flow which is critical for the learning process of both G and D.

Now, as with the generator, we can create the discriminator and apply the weights_init function.

3.5 Loss Functions and Optimizers

With the discriminator and the generator set up, we now specify how they learn through the loss functions and optimizers. The Binary Cross Entropy loss (BCELoss) function has been used which is defined in PyTorch as:

$$\ell(x, y) = L = \{l_1, \dots, l_N\}^\top, \quad l_n = -\left[y_n \cdot \log x_n + (1 - y_n) \cdot \log(1 - x_n)\right] \tag{1}$$

This function provides the calculation of both the log components in the objective function, i.e., log(D(x)) and log(1 - D(G(z))). We can specify what part of the BCE equation to use with the y input label. This is accomplished in the training loop which will be explained soon. It is important how we choose which component of the loss function we wish to calculate in the above equation, just by changing the label values, y.

We define our *real label* as 1 and the *fake label* as 0. These labels will be used when calculating the losses of D and G, and this is also the convention used in the original GAN paper.

Finally, two separate *optimizers* are set up, one for D and another one for G. As specified in the *DCGAN* paper, both are *Adam* optimizers with learning rate 0.0002 and beta1 0.5.

For keeping track of the generator's learning progression, we have generated a fixed batch of noise vectors that are drawn from a Gaussian distribution (i.e. fixed_noise). In the training loop, we have periodically input this *fixed_noise* into G, and over the iterations analyzed the images formed out of this fixed_noise.

3.6 Training

Finally, having defined all of the parts of the GAN framework, we now look at its training procedure. Training is split up into two main parts which are discussed below. Part 1 updates the Discriminator and Part 2 updates the Generator.

1. **Training Discriminator:** Recall that the goal of training the discriminator is to maximize the probability of correctly classifying a given input as real or fake. As was written in the original GAN paper, we wish to *update the discriminator by ascending its stochastic gradient.* Practically, we want to maximize log(D(x)) + log(1 - D(G(z))).

   First, we will construct a batch of real samples from the training set, forward pass through D, calculate the loss (log(D(x))), then calculate the gradients in a backward pass. Secondly, we will construct a batch of fake samples with the current generator, forward pass this batch through D, calculate the loss, log(1 - D(G(z)), and accumulate the gradients with a backward pass. Now, with the gradients accumulated from both the all-real and all-fake batches, we call it a step of the Discriminator's optimizer.

2. **Training Generator:** As stated in the original paper, we want to train the Generator by minimizing log(1 - D(G(z))) in an effort to generate better fakes. As mentioned, this was shown

by Goodfellow to not provide sufficient gradients, especially early in the learning process. As a fix, we instead wish to maximize log(D(G(z))).

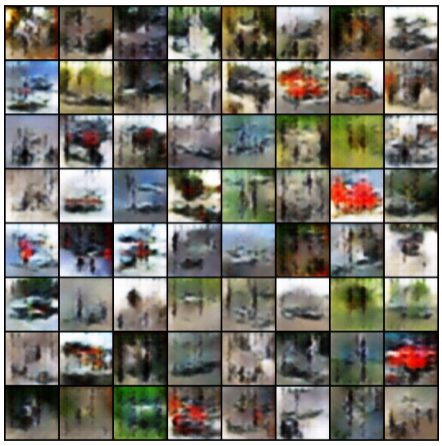
In the code we accomplish this by: classifying the Generator output with the help of Discriminator, computing G's loss using real labels (and not fake labels!), computing G's gradients in a backward pass, and finally updating G's parameters with an optimizer step. It may seem counter-intuitive to use the real labels as Generator training labels for the loss function, but this actually allows us to use the log(x) part of the BCELoss (rather than the log(1 - x) part) which is exactly what we want.

3.7 Visualizing Generator's progress across the epochs

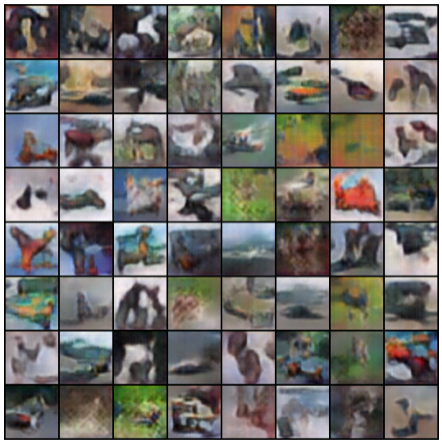
Recall that we had saved a fixed_noise vector to track generator' progress after every epoch of training. The progression of G's training is shown below:

1. Epoch 0.5**:**

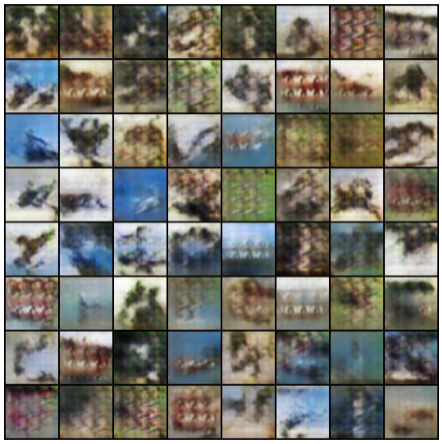
2. Epoch 4**:**



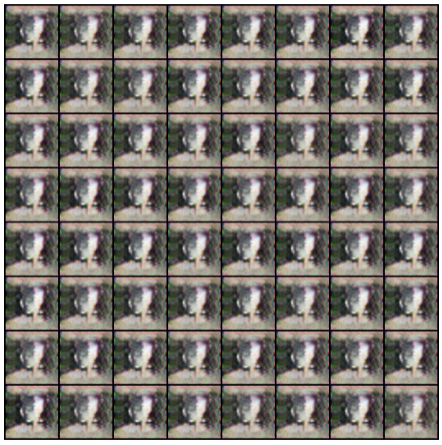
3. **Epoch** 10**:**



4. **Epoch** 18**:**



5. Epoch 25**:**



We see that the Generator learns producing meaningful images till certain epochs, but the quality of the images starts deteriorating after that. As can be seen in the final epoch, **mode collapse** occurs, i.e., no matter the input fed to the generator, the output is always the same. This is one of the limitations of DC-GANs and GANs in general, which we discuss in the next subsection.

3.8 Limitations

DCGANs exhibit limitations while modelling long term dependencies for image generation tasks. For example, dogs are often drawn with realistic fur texture but without clearly defined separate feet. The problem with DCGANs exists because model relies heavily on convolution to model the dependencies across different image regions. Since convolution operator has a local receptive field, long ranged dependencies can only be processed after passing through several convolutional layers. This could prevent learning about long-term dependencies for a variety of reasons:

1. A small model may not be able to represent them.
2. Increasing the size of the convolution kernels can increase the representational capacity of the network but doing so also loses the computational and statistical efficiency obtained by using local convolutional structure.
3. **Mode collapse:** Mode collapse occurs when the generator generates a limited diversity of samples, or even the same sample, regardless of the input noise vector.

To alleviate the first two problems, a *self-attention* (SA) module is introduced in DCGANs, whereas to mitigate the third one, GANs are trained with Wasserstein Loss.

We now discuss the SA-GAN with Wasserstein loss.

4 Self-Attention GAN (SA-GAN)

We had listed the limitations of GANs in the previous section. As described earlier, a self-attention module can help in mitigating the long-range dependency problems associated with the convolutional layers. Self-attention (SA), exhibits a better balance between the ability to model long-range dependencies and the computational and statistical efficiency.

Moreover, to alleviate the problem of mode collapsing, we introduce Wasserstein loss.

We discuss the SA module and the Wasserstein loss before moving on to the architecture of the two networks.

4.1 Self-Attention Module

The self-attention module is complementary to convolutions and helps with modeling long-range, multi-level dependencies across image regions. Armed with self-attention, the generator can draw images in which fine details at every location are carefully coordinated with fine details in distant portions of the image. Moreover, the discriminator can also more accurately enforce complicated geometric constraints on the global image structure.

Convolution processes the information in a local neighborhood, thus using convolutional layers alone is computationally inefficient for modeling long-range dependencies in images. We have adapted the self-attention module to the GAN framework, enabling both the generator and the discriminator to efficiently model relationships between widely separated spatial regions.

The image features from the previous hidden layer, $x \in^{\mathrm{C\times N}}$ are first transformed into two feature spaces f, g to calculate the attention:

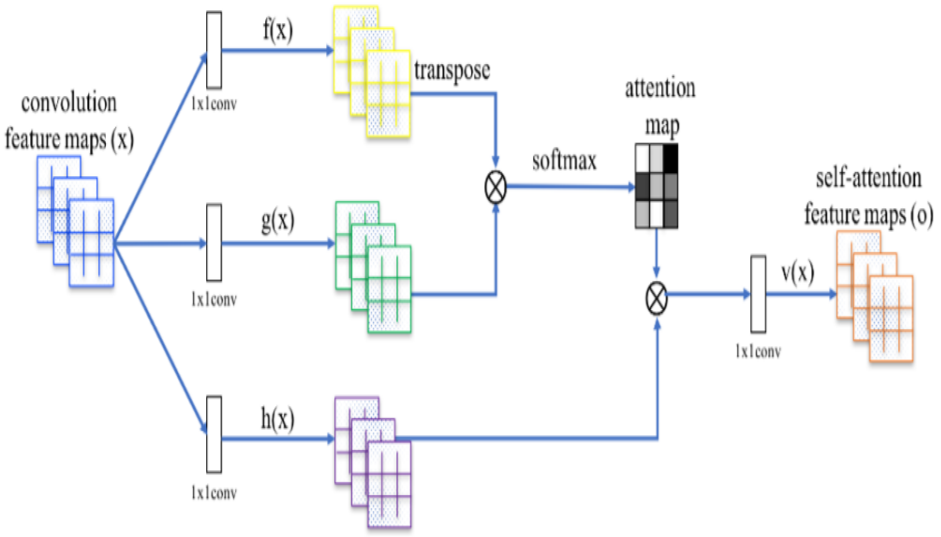
$$\beta_{j,i} = \frac{exp(s_{ij})}{\sum exp(s_{ij})} \tag{2}$$

Here, s_{ij} is the similarity between the transformed feature spaces f and g. These can be thought of as *query* and *key*, respectively. Also, $N = width \times height$ coming from the previous hidden layer. $\beta_{j,i}$ indicates the *importance* or the *attention* of the i^{th} location in synthesizing the j^{th} location in a feature map.

C is the number of channels and N is the number of feature locations of features from the previous hidden layer.

In addition, we further multiply the output of the attention layer by a scale parameter and add back the input feature map. This can be viewed as adding the respective *values* to synthesize the j^{th} location through a weighted average, where the weights are given by the attention weights mentioned in the equation above.

Following is the architecture of the Self-Attention module:



4.2 Spectral Normalization

Miyato et al. (2018) originally proposed stabilizing the training of GANs by applying spectral normalization to the discriminator network. Doing so constrains the Lipschitz constant of the discriminator by restricting the spectral norm of each layer.

The authors of SA-GAN argued that the generator can also benefit from spectral normalization, based on an evidence that the conditioning of the generator is an important causal factor in GANs' performance. They further added that spectral normalization in the generator can prevent the escalation of parameter magnitudes and avoid unusual gradients.

We have used a third party code to implement spectral normalization in the SA-GAN model. Here, is the link: `https://github.com/christiancosgrove/pytorch-spectral-normalization-gan`

4.3 Wasserstein Loss

TODO

With the Wasserstein loss, the Generator is actually termed as an *Actor*, and the Discriminator the *Critic*. We'll, however, stick with the earlier terminologies of calling them *Generator* and *Discriminator*.

4.4 Network Architecture - Generator

The architecture of the Generator in SA-GAN is almost the same as that of the Generator in DCGAN, with very little additions and modifications.

The number of layers, along with the dimensions, remain exactly the same. However, before performing Batch normalization after every layer of the generator, we first apply Spectral Normalization

to the output feature maps generated by the convolutional-transpose layers. Batch norm is done to the output of the Spectral norm in each layer.

The self-attention layers are added after the third and the fourth layers in our implementation. This is becuase the final layers of the generator have the responsibility of producing more realistic features than the previous layers.

4.5 Network Architecture - Discriminator

The architecture of the Discriminator remains almost the same too with minor modifications. We apply Spectral Normalization here too, like we did in the generator of SA-GAN.

The self-attention layers are added after the third and the fourth layers in our implementation of the Discriminator. Similar arguments can be made behind this reasoning. The final layers of the discriminator have a greater responsibility and are closest to generating the score. Hence, providing attention at these layers makes more sense than adding them in the initial layers.

With the Wasserstein loss, as mentioned in the WGAN paper, the Discriminator outputs a real value instead of a probabilistic score.

4.6 Optimizer

For both the Generator and the Discriminator models, we use the *Adam* optimizer with $\beta_1 = 0$ and $\beta_1 = 0.9$ for training. By default, the learning rate for the discriminator is 0.0004 and the learning rate for the generator is 0.0001.

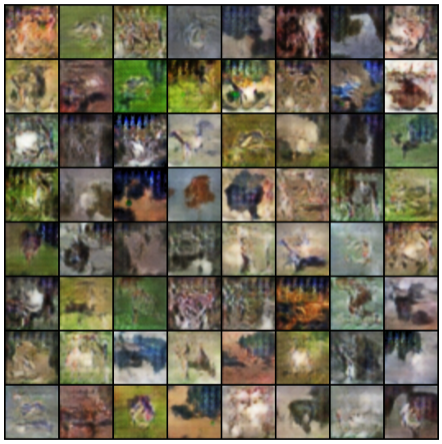
4.7 Results

We've used a third-party implementation to calculate the FID Score. It can be found here: `https://github.com/mseitze`

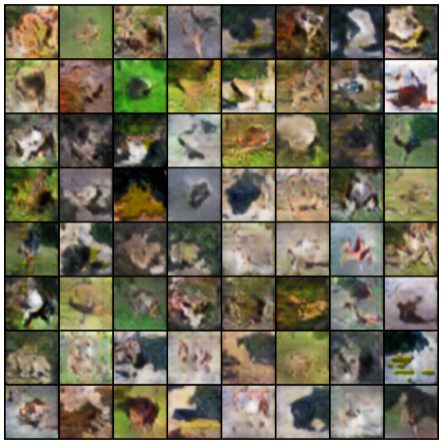
The entire model was trained for 45 epochs, with the generator being trained for every training step of the discriminator.

Here is the progress of the Generator across the epochs:

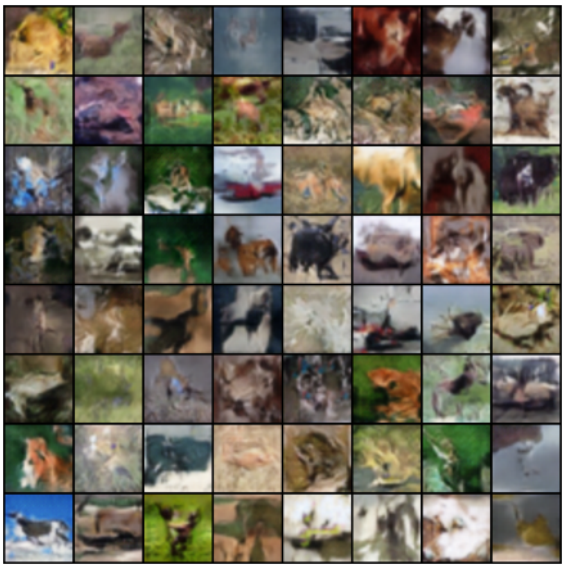
1. **Epoch** 6**:**



2. **Epoch** 12**:**



3. Epoch 40**:**



The FID score obtained after the 45th epoch is 71.27.

Another key thing to notice here is that we have got completely rid of the **mode collapse** problem which was observed in DC-GANs, even though we trained the two models for longer.