Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks

E4040.2021Spring.UGAN.report
Shambhavi Roy sr3767, Simran Tiwari st3400, Saravanan Govindarajan sg3896
Columbia University

Abstract

The source paper presents a novel architecture of the Deep Convolutional Generative Adversarial Network (DCGAN) and illustrates its use for learning feature representations from images to be used for unsupervised tasks on images. Further, this paper presents an analysis of the learned filters and generators. Our project is adapted from this paper. We have tackled the task of training the proposed DCGAN on multiple image datasets to be able to generate samples resembling the training data. Though this paper mentions the subsequent use of the learned feature representations for image classification tasks and further analysis, our project focuses on training this architecture on multiple datasets to obtain the learned features and visualizing them.

1. Introduction

Generative Adversarial Networks (GANs) [9] are a type of generative models which attempts to learn the probability distribution of the input training data and aims to represent an estimate of this distribution. Thus, GANs can generate new samples resembling the training data.

This paper [1] explores the problem of learning image features from large datasets using Generative Adversarial Networks (GANs) instead of deep Convolutional Neural Networks (CNNs). These learned image features can then be used for image classification tasks.

The authors also mention the problem of unstable training of GANs and obtaining nonsensical outputs. To mitigate this, they mention certain architectural constraints for GANs for successful training on large image datasets. This proposed architecture is called Deep Convolutional Generative Adversarial Networks (DCGANs).

Further, this paper shows the use of the trained discriminator model of the GAN for image classification and compares the results with other common image classification algorithms.

Finally, the concept of the learned generator model having vector arithmetic properties that can be used to understand the quality and properties of the generated image samples is presented in the paper. In addition to that it states that the discriminator of the DCGAN learns meaningful patterns instead of just memorizing the training data.

2. Summary of the Original Paper

2.1 Methodology of the Original Paper

This paper presents the topology of their Convolutional GAN. The authors suggest some architectural constraints for stable training of the Convolutional GAN and they call it the DCGAN. Their suggestions include replacing max-pooling with strided convolutions, removing fully connected layers, using batch normalization in both generator and discriminator parts, and using tanh non-linearity in the last layer, and using LeakyReLU in the discriminator model. This proposed architecture is shown in Figure 1 below.

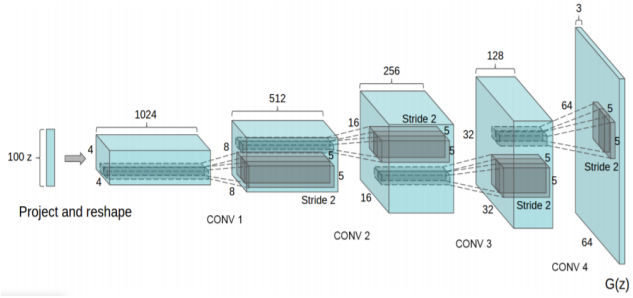

Fig. 1. DCGAN Architecture as in [1]

The authors posit that their architecture can be used to learn feature representations from object parts to scenes, and these learned features can be used for supervised learning tasks. They suggest that their proposed architecture can be used for stable training across a range of large image datasets.

The authors mention the training of the DCGAN on large image datasets like Imagenet 1-K, LSUN (Large-scale Scene Understanding), and a web-scrapped Faces dataset. They preprocess the images by scaling to [-1, 1]. They use mini-batch Stochastic Gradient Descent (SGD) with a batch size of 128. The weights are initialized randomly from a normal distribution with zero mean and a very small standard deviation of 0.02. They have used LeakyReLU activation in the discriminator with a leak value α of 0.2. Finally, they have used Adam

optimizer with a learning rate of 0.0002 and set the momentum term β_1 to 0.5.

The paper further presents details on the datasets used for training the DCGAN. They have used the LSUN bedroom dataset having more than 3 million training examples and performed deduplication to prevent the model from memorizing training examples. They have also trained the DCGAN on a web-scrapped Faces dataset having 350,000 face boxes without data augmentation. Finally, they have trained it on natural images from the Imagenet-1K resized to 32x32 and without data augmentation.

The final section of this paper focuses on analyzing the trained discriminator and generator. More specifically, they focus on understanding the features learned by these models. They present brief subsections with experiments that the discriminator and generator learn good feature representations of images and they can be used for other unsupervised learning and modeling tasks.

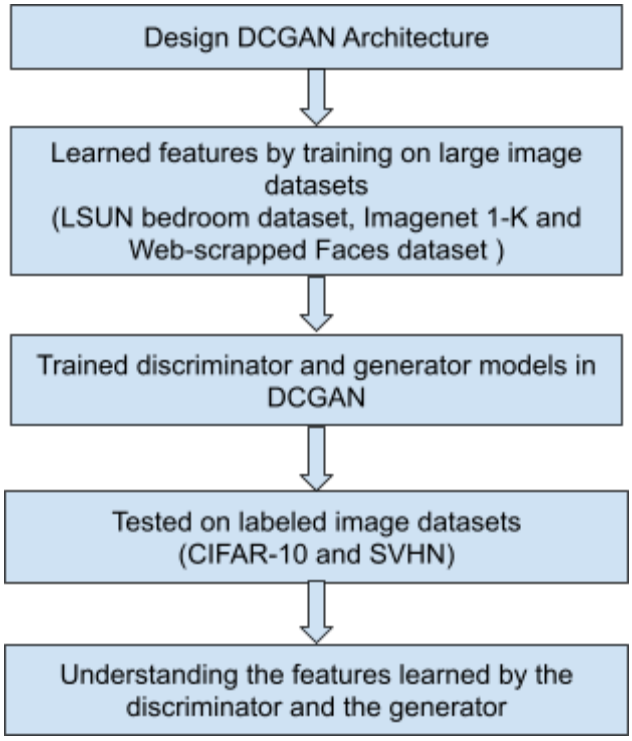

Fig. 2. Main process flow in the paper [1]

2.2 Key Results of the Original Paper

The paper presents two experiments to show that the proposed DCGAN architecture is capable of being used as a feature extractor. The first experiment in the paper presents the use of the features learned by the trained discriminator as an extractor topped by the K-means algorithm for feature learning achieving 80.6% accuracy. Since the DCGAN has been trained on the Imagenet-1K dataset and tested as a feature extractor on the CIFAR-10 dataset, this experiment proves that the DCGAN is a robust feature extractor. A similar experiment has been performed on the SVHN dataset achieving a 22.48% test error.

Finally, the paper examines the trained discriminator and generator in some ways. It explores subsequent generated images and shows that apparently if objects seem to be added or removed, the discriminator model has learned the image feature representations well. These features are interesting as they activate on some parts of the image. It also explores that the generator learns features representing the major parts of the image. These observations are backed by experiments on images by the model.

This paper concludes by stating that the proposed DCGAN architecture is stable in training and is capable of learning good feature representations from images. However, they observed model instability in some cases that can be further explored in research.

3. Methodology (of the Students' Project)

Our project focuses on implementing the proposed DCGAN architecture for training on image datasets. We have understood the proposed architecture and implemented it using TensorFlow 2.0 and Keras.

CIFAR-10[2], CelebA[3], SVHN[4], and 20% of the images of LSUN-Bedroom[10] datasets were used to train the models. This choice of datasets for training the DCGAN is different from the original paper. We chose these datasets because they are smaller and would require lesser computational resources and training time.

Dataset	**Description**	**Number of samples available**	**Shape**
CIFAR-10	Consists of tiny color images in 10 classes.	60000 (50000 training images and 10000 test images)	(32, 32, 3)
CelebA	Large-scale face attributes dataset with 40 annotated attributes. Consists of images with	202599	(32, 32, 3)

	large pose variations and background clutter.		
SVHN	Real-world dataset obtained from house numbers in Google Street View images. It has 10 classes and each digit is labeled.	73257	(32, 32, 3)
LSUN	20% sample of the bedroom category from the Large Scale Scene Understanding dataset	303125	(64, 64,3)

Table. 1. Details of datasets used in this project

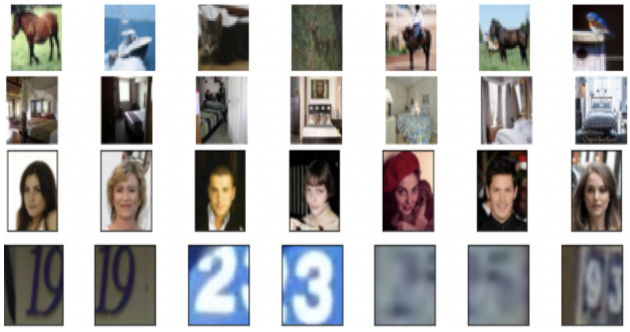

Fig. 3. Sample images of each dataset. First row: CIFAR, second row: LSUN, third row: CelebA, fourth row: SVHN

Our project focuses on implementing the proposed DCGAN architecture for training on image datasets. Further, we have tried to visualize the learned features by performing analysis experiments similar to the source paper. This is done using guided backpropagation.

3.1. Objectives and Technical Challenges

The objective of this project is to implement the proposed DCGAN architecture from scratch and test its capability as a feature extractor by using it for image classification.

However, we ran into a few technical challenges while working on this project. We realized that the datasets used in the paper were too large like the LSUN dataset. Oftentimes our Google Cloud virtual instance to shut down. So instead, we used a publicly available dataset on Kaggle[10] of 20% the size of the LSUN dataset to perform the training task.

We experimented with the architecture and observed that we could get suitable outputs by removing batch normalization from the generator and discriminator models, as opposed to the suggestions in the paper.

3.2.Problem Formulation and Design Description

The problem at hand requires us to implement the correct DCGAN architecture to train on large image datasets. This task required us to define many utility functions for smaller subtasks as described in the subsequent paragraphs.

The first part of the project was to acquire the image datasets from the relevant sources and preprocess them. DCGANs require input images to be scaled to the range [-1, 1] and we performed that using simple NumPy functions. Similarly, we also defined a scaleback function to scale back the image from the range [-1, 1] to [0, 255].

Next, we define the generator and discriminator models constituting the DCGAN. These are the two models that are trained with an adversarial process. The generator learns to generate new images resembling the input dataset and the discriminator learns whether these images are real or fake. This is followed by defining the DCGAN architecture. We have explained the detailed architecture of these models in the next section.

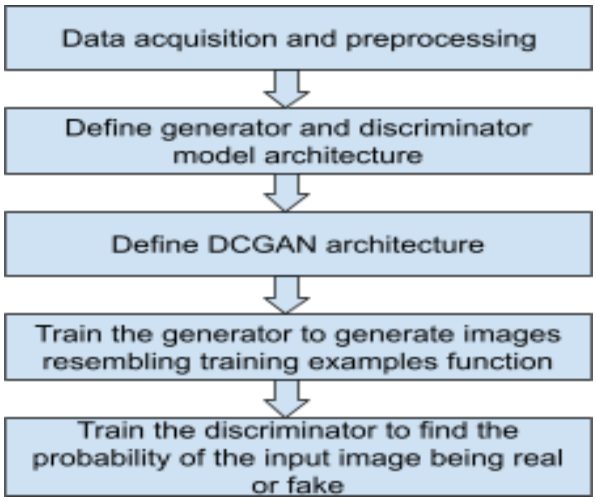

Fig. 4. Basic problem formulation diagram

The problem at hand requires us to implement the correct DCGAN architecture to train on large image

datasets. This task required us to define many utility functions for smaller subtasks as described in the subsequent paragraphs.

The required utility functions in separate .py files. We defined the true_samples_generator function to select random real training samples with label = 1. On the other hand, the fake_samples_generator function is used to generate fake samples (label = 0) using the generator model. We defined the make_latent_samples to generate random latent samples given as input to the generator.

Other utility functions are defined to plot the generator and discriminator losses and plot them and display the generated images.

An important function defined is the performance_summarizer function. It is used to calculate the accuracy of the generated sample being real or fake. We use this function to save the generated image for future reference and finally, we save the generator and discriminator model.

4. Implementation

We have implemented this project by dividing it into smaller tasks, each coded in Python on Jupyter notebooks and multiple .py files. We have utilized TensorFlow 2.0 and Keras for the deep learning tasks. All the training and inference tasks have been performed on a Google Cloud virtual instance powered by Nvidia Tesla K80 GPU.

The datasets we have used for this project are publicly available.

4.1. Deep Learning Network

The first part of our project implementation is coding the DCGAN architecture. We followed the guidelines suggested in the paper, however, we needed to perform minor adjustments for stable training and proper outputs.

We begin by defining the generator architecture. It is a Sequential model. It accepts the input of a latent sample (100 randomly generated numbers) and produces an image that resembles a training example. It consists of a dense layer followed by LeakyReLU activation with leak value α as 0.2 as suggested in the paper and the output is reshaped to (4, 4, 256). A highlight of the generator architecture is the use of the transpose convolution (strided convolution) operation which results in increasing the dimensions of the input. This is done because the generator has to generate images of the same size as that of the training examples. The output activation used here is tanh.

```
Generator Summary:
Model: "sequential_7"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_5 (Dense)              (None, 4096)              413696
_________________________________________________________________
leaky_re_lu_20 (LeakyReLU)   (None, 4096)              0
_________________________________________________________________
reshape_2 (Reshape)          (None, 4, 4, 256)         0
_________________________________________________________________
conv2d_transpose_6 (Conv2DTr (None, 8, 8, 128)         524416
_________________________________________________________________
leaky_re_lu_21 (LeakyReLU)   (None, 8, 8, 128)         0
_________________________________________________________________
conv2d_transpose_7 (Conv2DTr (None, 16, 16, 128)       262272
_________________________________________________________________
leaky_re_lu_22 (LeakyReLU)   (None, 16, 16, 128)       0
_________________________________________________________________
conv2d_transpose_8 (Conv2DTr (None, 32, 32, 128)       262272
_________________________________________________________________
leaky_re_lu_23 (LeakyReLU)   (None, 32, 32, 128)       0
_________________________________________________________________
conv2d_14 (Conv2D)           (None, 32, 32, 3)         3459
=================================================================
Total params: 1,466,115
Trainable params: 1,466,115
Non-trainable params: 0
```

Fig. 5. Our implementation of Generator- Model summary

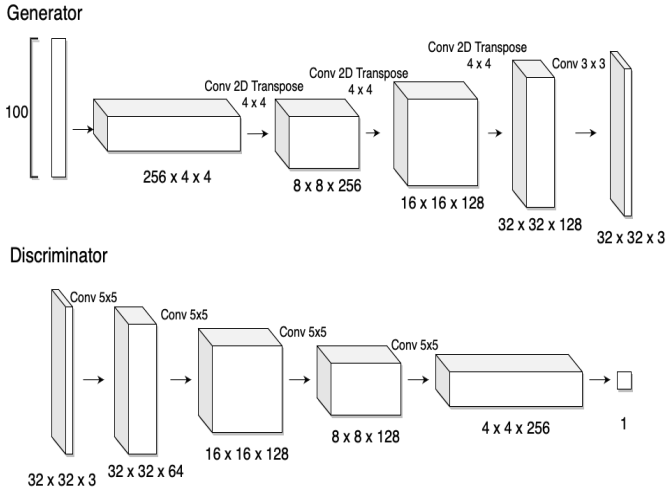

Fig. 6. Our implementation of Generator & Discriminator model architecture for 32 x 32 sized input images

Next, we define the discriminator architecture. It is a Sequential model and its input is an image with each pixel in the range [-1, 1]. The model consists of convolutional layers followed by LeakyReLU layers. The leak parameter α is set to 0.2 as suggested in the paper. Finally, the Flatten, Dropout and Dense layers are used. The output of a discriminator is the probability of the image being real or fake.

```
Discriminator Summary:
Model: "sequential_6"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_10 (Conv2D)           (None, 32, 32, 64)        1792
_________________________________________________________________
leaky_re_lu_16 (LeakyReLU)   (None, 32, 32, 64)        0
_________________________________________________________________
conv2d_11 (Conv2D)           (None, 16, 16, 128)       73856
_________________________________________________________________
leaky_re_lu_17 (LeakyReLU)   (None, 16, 16, 128)       0
_________________________________________________________________
conv2d_12 (Conv2D)           (None, 8, 8, 128)         147584
_________________________________________________________________
leaky_re_lu_18 (LeakyReLU)   (None, 8, 8, 128)         0
_________________________________________________________________
conv2d_13 (Conv2D)           (None, 4, 4, 256)         295168
_________________________________________________________________
leaky_re_lu_19 (LeakyReLU)   (None, 4, 4, 256)         0
_________________________________________________________________
flatten_2 (Flatten)          (None, 4096)              0
_________________________________________________________________
dropout_2 (Dropout)          (None, 4096)              0
_________________________________________________________________
dense_4 (Dense)              (None, 1)                 4097
=================================================================
Total params: 522,497
Trainable params: 0
Non-trainable params: 522,497
```

Fig. 7. Our implementation of Discriminator- Model summary

Though the paper also suggests the use of Batch Normalization layers in both the generation and the discriminator, we obtained noisy outputs when we tried this in our implementation. On the contrary, we obtained better results by removing Batch Normalization layers. The reason for this is to be understood.

These two models are used to define the DCGAN. The DCGAN is a Sequential model with the discriminator and generator as its components. It uses the Adam optimizer with a learning rate of 0.0002 and momentum term β_1 as 0.5 as suggested in the paper.

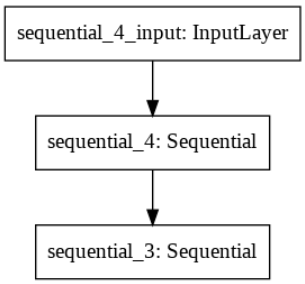

Fig. 8. Our implementation of DCGAN architecture

4.2. Software Design

The software design was done using the available functions in TensorFlow 2.0 and Keras on Jupyter Notebooks.

The flowchart below shows the basic code execution for DCGAN training.

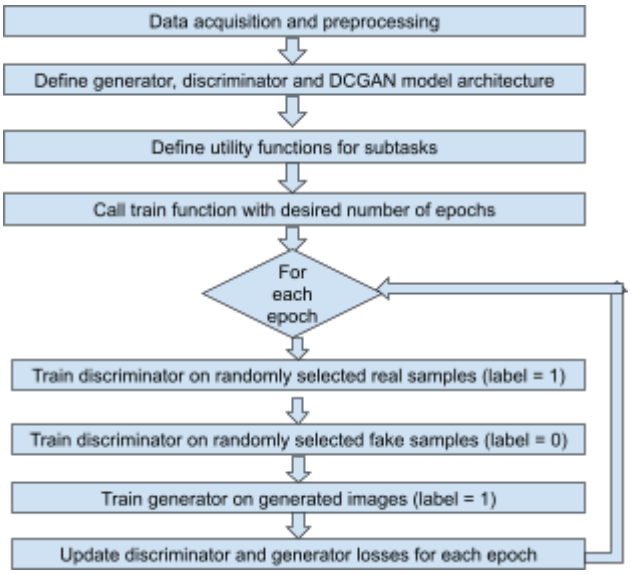

Fig. 9. A basic flowchart for DCGAN training

We begin with data acquisition and preprocessing. This is followed by the training process. The training can be performed for the desired number of epochs and batch size. For instance, we have trained the DCGAN on the SVHN dataset for 10 epochs with a batch size of 64 while implementing batch normalization. Next, we trained the DCGAN on the SVHN dataset for 40 epochs with a batch size of 128 without implementing batch normalization.

The algorithm for the training process is as follows:

Algorithm for train function:
Input: Number of epochs, batch size, training data, generator model, discriminator model, DCGAN model, and size of the latent sample.
Step 1: Calculate the number of batches = (Size of training data)/(batch size)
Step 2: Define lists to store generator_loss and discriminator_loss
Step 3: For each epoch
Create batches over the training set
Randomly select real samples to train discriminator and update discriminator weights
Randomly select fake samples to train discriminator and update discriminator weights

Select latent samples and give as input to the DCGAN to train the generator
Update the losses for generator and discriminator and summarize loss on batch

Table. 2. Pseudocode for the train function

As shown in the pseudocode above, the train function is given the training data, generator model, discriminator model DCGAN, size of the latent sample, to be executed for the desired number of epochs and batch size. Within this function, the number of batches is calculated and we use python lists to keep track of the discriminator loss and generator loss. For each epoch, we create batches over the training set. Then, we randomly select real samples from the training set. This is done by selecting samples and assigning them a label of 1. These samples are used to train the discriminator model. Next, we randomly select samples and assign them to a label of 0 to annotate fake samples and train the discriminator model. This is the process of adversarial learning. Further, we select latent samples as input to the generator. These samples are given input to the DCGAN and the generator is trained to learn the distribution of the samples. Here, all the samples in latent space are given with a label as 1. Finally, we update the losses of the discriminator and generator for the epoch and summarize the losses obtained on this batch. This loop is repeated for all epochs.

Once the training process is completed, we perform other complementary tasks like plotting the loss curves for the training process and displaying the generated samples as outputs. We have also tried to compile the samples generated at each epoch into videos, the link for which is provided in the supplementary section.

5. Results

5.1. Impact of Batch Normalization on training

The network mentioned in Fig. 6. was trained on the CIFAR-10 dataset with and without the Batch Normalization (BN) layers, and surprising results were observed which were contradicting the details mentioned in the DCGAN paper. We observed that the model training remains stable when the BN is not used. Whenever the BN layer was included in the network, the loss values of the generator network quickly went to zero and it was not able to recover to continue with the training process, one reason for this could be that the BN layer forces all the values to become very small comparatively, and due to this, the gradient flow for the generator gets vanished in few epochs. Whereas, the model training remained stable otherwise as shown in Fig. 10**.** This same behavior was observed with other datasets as well. For this reason, we decided not to include the BN layers in the network for further analysis. The results of epoch 1 and epoch 20 are compared in Fig. 11, Fig. 12 with and without BN.

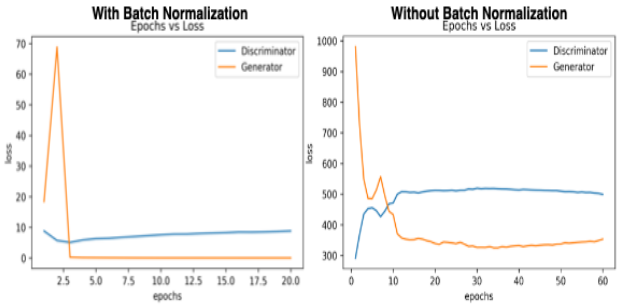

Fig. 10. Training loss graphs with and without Batch Normalization for CelebA dataset

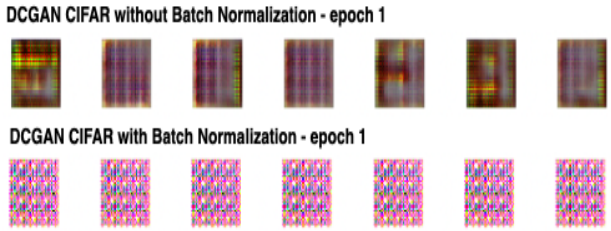

Fig. 11. Results of Epoch 1 with and without BN

Fig. 12. Results of Epoch 20 with and without BN

5.2. Impact of the training data size on image generation

To evaluate the impact of the amount of training data on the image generation, we trained the network twice on LSUN, one with the 10% of LSUN dataset which contained around 300k images (DCGAN-LSUN$_1$), and the other with a subset of the LSUN images which contained around 19k images (DCGAN-LSUN$_2$). Since the image size of each image in the LSUN was (64, 64, 3), the model architecture was modified slightly to incorporate this difference.

DCGAN-LSUN$_2$ took 2.5 hours and DCGAN-LSUN$_1$ took 6 hours to get trained. The generated images were compared against each other epoch-wise, and it appears that the DCGAN-LSUN$_1$ was able to generate good quality images starting from epoch number 1 as compared to the blurry images generated by the network DCGAN-LSUN$_2$ during the first few epochs. However, there was not much noticeable improvement in the DCGAN-LSUN$_1$ generated images as the epochs increased whereas the DCGAN-LSUN$_2$ was still trying to make better quality images with the increase in epochs. This could potentially mean that, when DCGAN is trained on a large dataset, it might stop generating better images after the first few epochs. Images generated by both models at Epoch 1 and 10 are shown in Fig. 13.

Fig. 13. Epochs results with 19k and 300k training samples

5.3. Understanding the training losses

As mentioned in the previous sections, we trained DCGAN on four different datasets that include both small and large image datasets (60k to 300k images). We tried to analyze the losses and their convergence behavior during the training time as shown in Fig. 14. for further analysis. It is evident from Fig. 14. that, for a smaller dataset there appears to be a point at which the $loss_{generator}$ and $loss_{discriminator}$ of the DCGAN intersect and after that point onwards, the $loss_{generator}$ starts to either decrease or tries to remain stable. Whereas when training the DCGAN on the large datasets, there seems to be no point in the early epochs at which the $loss_{generator}$ and $loss_{discriminator}$ of the DCGAN meet. This could possibly explain why the generated images stop improving after a few epochs.

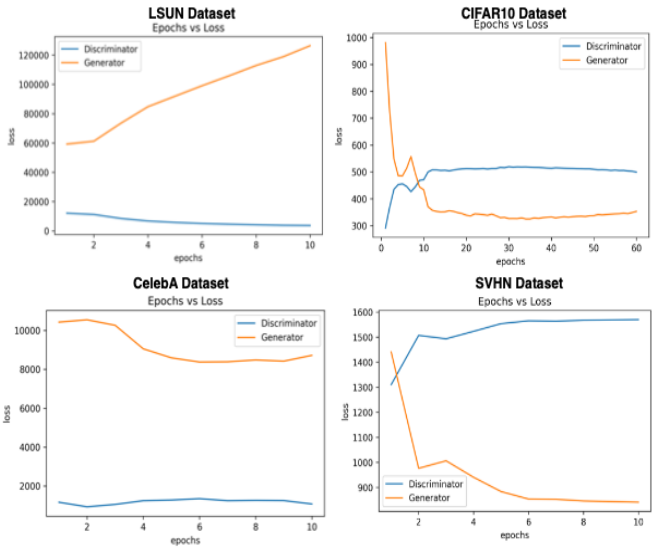

Fig. 14. Loss graphs of the DCGAN with different datasets

5.4. Epoch wise results of the generated images

In this section, the DCGAN generated images are shown in Fig. 15 - 17 per epoch for various datasets. DCGAN trained on CelebA and the LSUN seems to have produced the best results for this network setting.

Fig. 15. Generator outputs for CelebA dataset

Fig. 16. Generator outputs for LSUN dataset

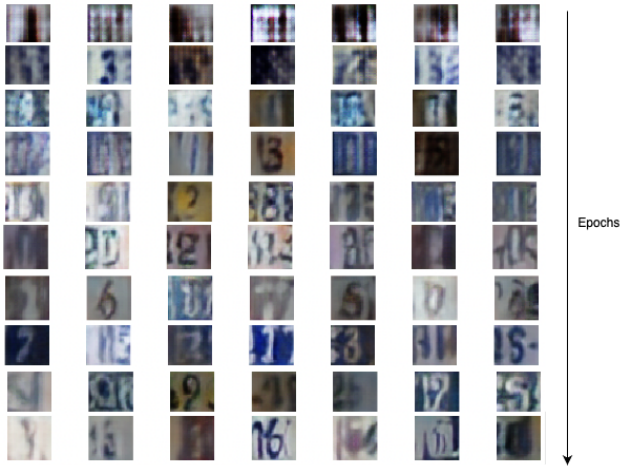

Fig. 17. Generator outputs for the SVHN dataset

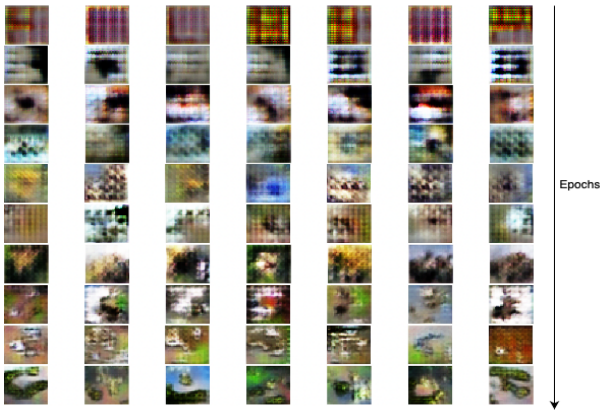

Fig. 18. Generator outputs for the CIFAR-10 dataset

5.5. Visualization of the discriminator through Grad-CAM

Grad-CAM [11] can be seen as a tool for visualizing the activation maps in a CNN, specifically, it generates a heatmap for every input image to indicate the spatial importance. Grad-CAM uses the gradients that flow into the last Conv layer of the network for understanding the relative importance of each unit for making a particular prediction. The flow of Grad-CAM can be seen in Fig. 19.

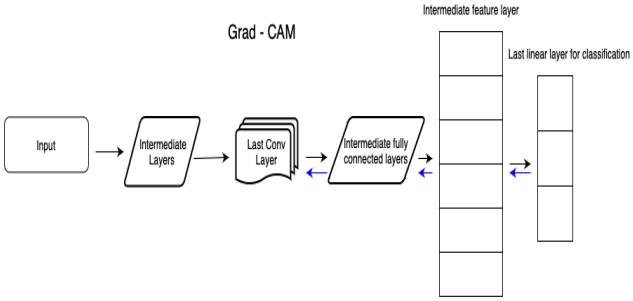

Fig. 19. Grad-CAM flow chart. The black lines represent the forward propagation in the network.

In this case, the Grad-CAM is used to visualize the features of the DCGAN's discriminator network which was trained on the SVHN dataset. Given an image, the discriminator of the DCGAN tries to predict whether that given sample is real or fake (if it was generated by the generator network). Visualization results for three test images can be seen in Fig. 20. The images on the left column represent the input images from the SVHN dataset, and the corresponding image on the right column represents the Grad-CAM result. Blue colored portion of the Grad-CAM images indicates the part which CNN did not use much for making the prediction, and the red and yellowish-colored portion in the images indicates the section which was important in making the prediction (predicting true in these examples). For the first case, it is clear that the section "3" and "5" contributed the most to call out whether it is a fake or real image. Even for the second and the third image shown in Fig. 20, the discriminator learned to look at the numbers present in the image for making the prediction. This indicates that the trained discriminator learned to look at meaningful patterns rather than just memorizing samples from the input training data as indicated by the authors of the DCGAN paper.

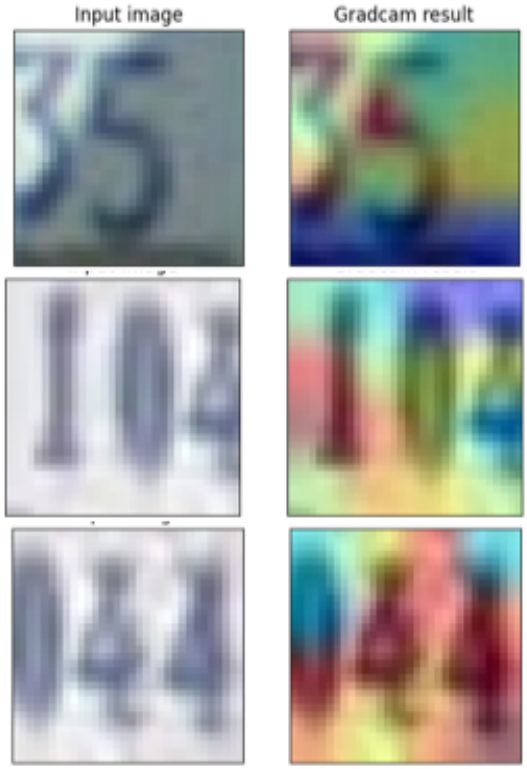

Fig. 20. Grad-CAM results for the SVHN examples

5.6. Visualization of the discriminator through guided backpropagation

To show that unsupervised DCGAN trained on a large image dataset can also learn a hierarchy of interesting features we used guided backpropagation, to visualize the trained filters of the last convolution layer of the discriminator model.

Guided backpropagation sets all negative gradients (which signifies that a pixel is not important) to zero. Values in the filter map greater than zero signify pixel importance which is overlapped with the input image to show which pixel from the input image contributed the most. The visualizations show that a significant amount of features respond to beds - the central object in the LSUN bedrooms dataset. The input images and visualizations of the filter are shown in Fig. 21 and Fig. 22.

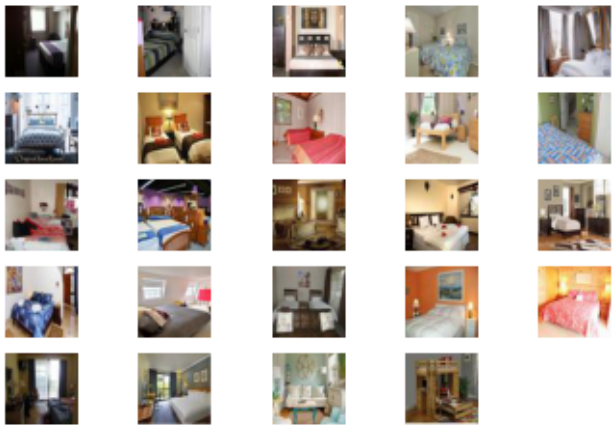

Fig. 21. Input images to the discriminator

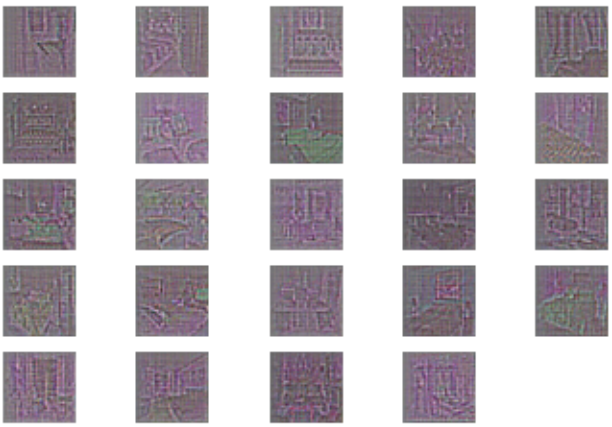

Fig. 22. Visualization of trained filters of last convolution layer of discriminator using guided backpropagation

5.7. Comparison of the Results Between the Original Paper and Students' Project

The paper[1] shows the generated outputs obtained by training the DCGAN on the LSUN dataset after 5 epochs. The authors note here that is visual underfitting as there is repeated noise texture in some outputs parts of the images.

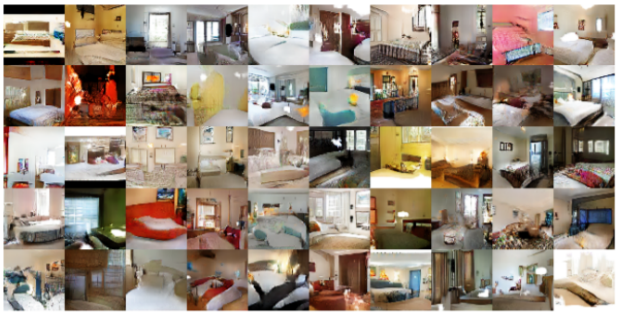

Fig. 23. The paper [1] outputs showing generated images resembling the LSUN dataset.

We have observed similar noise textures on our generated outputs on the LSUN dataset. The paper, however, does not show their generated results on other datasets. As mentioned in the paper, we have observed similar noise textures across generated images from all the datasets we used.

The paper further illustrates the use of the learned models as feature extractors. They have presented the use of the discriminator topped by commonly used classifiers like SVM and K-means. They have compared the performance of this architecture with CNNs for image classification. In this project, however, we could not perform this step.

The paper also argues that the discriminator learns meaningful patterns for distinguishing images rather than just memorizing the training examples. We have also reached this conclusion by visualizing the discriminator's features using Grad-CAM and the Guided backpropagation algorithms.

We further did experimentations to observe the stability of the network by changing the size of the training data, adding the batch normalization layer. We observed that we obtained stable outputs without batch normalization.

5.8. Discussion of Insights Gained

We implemented the DCGAN architecture as explained in the paper for training on CIFAR-10 and CelebA datasets. Our implementations could generate samples resembling the images in these datasets. However, we had to tweak the proposed architecture a bit for generating good results. This involved using a dropout of rate 0.4 in the discriminator model.

The paper did not provide any insights on the number of epochs to train the model so we tried different values to observe results.

6. Conclusion

This project focuses on implementing the architecture of a Deep Convolutional Generative Adversarial Network (DCGAN). We have learned that the generator and discriminator models are the two components of the DCGAN architecture. We implemented this architecture following suggestions in the paper. We trained this model on the CelebA, SVHN, CIFAR-10, and LSUN datasets. Our results showed that the generator could output images resembling the training dataset. The impact of the Batch Normalization layer was studied and the discriminator's features were also visualized. However, this project can be improvised in the future. The next step to be implemented in this project is to test the trained discriminator as a feature extractor for image classification tasks.

7. Acknowledgement

We are grateful to Prof. Kostic for giving us the opportunity to work on this project. We are thankful to the TAs and our peers for providing us advice and suggestions in a timely manner both online and offline.

8. References

[1] Radford et al. "Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks", arXiv:1511.06434

[2] CIFAR-10 dataset: Alex Krizhevsky, Learning Multiple Layers of Features from Tiny Images, 2009 [Online] https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf

[3] CelebA dataset: Liu, Ziwei and Luo, Ping and Wang, Xiaogang and Tang, Xiaoou, Deep Learning Face Attributes in the Wild, *Proceedings of International Conference on Computer Vision (ICCV), December 2015.*

[4] SVHN dataset: Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, Andrew Y. Ng Reading Digits in Natural Images with Unsupervised Feature Learning *NIPS Workshop on Deep Learning and Unsupervised Feature Learning 2011.* [Online] http://ufldl.stanford.edu/housenumbers/

[5] Fisher Yu, Ari Seff, Yinda Zhang, Shuran Song, Thomas Funkhouser, and Jianxiong Xiao, LSUN: Construction of a Large-scale Image Dataset using Deep Learning with Humans in the Loop, arXiv:1506.03365 [cs.CV], 10 Jun 2015

[6] Deep Convolutional Generative Adversarial Network [Online] https://www.tensorflow.org/tutorials/generative/dcgan

[7] Having fun with Deep Convolutional GANs[Online] https://naokishibuya.medium.com/having-fun-with-deep-convolutional-gans-f4f8393686ed

[8] Deep Convolutional GAN (DCGAN) with SVHN

[Online] https://github.com/naokishibuya/deep-learning/blob/master/python/dcgan_svhn.ipynb
[9] Goodfellow, NIPS 2016 Tutorial: Generative Adversarial Networks arXiv:1701.00160v4
[10] 20% sample of LSUN dataset https://www.kaggle.com/jhoward/lsun_bedroom
[11] Ramprasaath R Selvaraju, Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. Grad-cam: Visual explanations from Deep Networks via Gradient-based Localization. In *Proceedings of the IEEE International Conference on Computer Vision*, *pages 618–626, 2017*.

9. Appendix

9.1 Individual Student Contributions in Fractions

	sr3767	st3400	sg3896
Last Name	Roy	Tiwari	Govindarajan
Fraction of (useful) total contribution	1/3	1/3	1/3
What I did 1	Trained the DCGAN on the SVHN dataset	Trained the DCGAN on the LSUN dataset	Trained the DCGAN on CelebA, SVHN, CIFAR10 dataset. Wrote bash scripts for downloading the datasets.
What I did 2	Report writing and code maintenance	Implemented Guided Backpropagation on LSUN	Implemented Grad-CAM & experimented with Batchnorm
What I did 3	implement classifier on top of feature extractor but did not succeed	website	the project report

9.2 If/as needed: additional diagrams, source code listing, circuit schematics, relevant datasheets etc.

The souce code for this project is available on: https://github.com/ecbme4040/e4040-2021Spring-Project-UGAN-sr3767-st3400-sg3896

Project website: https://ecbme4040.github.io/e4040-2021Spring-Project-UGAN-sr3767-st3400-sg3896/

Google drive folder: https://drive.google.com/drive/folders/1DQ_by-k33v1k-8cY3zsSSCJC30lUg-8q?usp=sharing