ISyE 6740 - Fall 2022
Project Report

Supervisor: **Prof. Yao Xie**

Team Members: **Oojas Salunke (GTID - Osalunke3) and Yash Bhole (GTID - ybhole3)**

Project Title: **Using Generative Adversarial Networks (GANs) to Generate New Images from Existing Datasets** *(Applied to Fashion MNIST and Abstract Art Data Sets)*

Date: **12 Dec 2022**

Contents

1 Introduction

Machine Learning algorithms can interpret images in a manner similar to the human brain. This process of Image Processing is used to improve your image's quality or to help extract useful information from it. There are various applications of these techniques, such as helping medical professionals detect anomalies efficiently from medical images, aiding security, and military agencies in surveillance and authentication, assisting self-driving cars in detecting objects, etc. Some general applications include improving the quality of images, adding filters, recognizing images for faster retrieval from large data sets, etc.

One such application is pattern recognition, which involves classifying objects in images and extracting useful information in analyzable formats. We are exploring a related application in this course project. We are applying Machine Learning techniques to aid the process of designing new images given a repository of designs that are already created. An essential requirement of this image generation process is retaining patterns of the existing data set while creating entirely new visuals.

We are deploying the technique of using Generative Adversarial Networks (GANs) for this purpose. GANs are an approach to conducting generative modeling using two convolutional neural networks. This is an unsupervised learning task in machine learning. It involves learning the regularities or patterns in input data so that the model can generate new data drawn from the input data set. Intuitively, one of the two neural networks is considered a forger and the other a detective. The forger generates its image from random data distribution and tries to fool the detective into interpreting this generated image as a real image. The detective has the record of all the real and generated images to make this classification. The feedback from the detective's output is fed back to the forger, who tries to generate a relatively better image and passes it again to the detective to detect. This process continues until the detective can no longer differentiate clearly between the real and the fake images, as these images are now statistically similar even though they are visually different.

GANs have been the state-of-the-art algorithm for image generation. Invented in 2014 by Ian Goodfellow and his colleagues, it is one of the recently developed frameworks in machine learning. It was first introduced at the 28th International Conference on Neural Information Processing Systems (NIPS 2014), which can be found here.

2 Motivation

We came across this technique of using GANs while searching for a suitable data set to work on for the course project. We then realized the usefulness of this technique and were curiously intrigued by it. This provided a way to apply the methodologies learned in class - namely, understanding a new modeling technique, exploring the intuition and math behind it, and then applying it to a data set of considerable size and testing the functioning of the model - to learn and apply new concepts entirely on our own, using help from research articles and online resources. As machine learning is constantly modifying, it was essential to use the foundation we have gained in this class to learn adaptability.

3 Sourcing Data

We have used two data sets for this course project: abstract art and fashion MNIST datasets.

1. Abstract Art database was sourced from Kaggle, which was originally scraped from WikiArt. A sample image from this data set is shown in Figure 1 (a). This consists of 2,782 high-resolution images of abstract art paintings. The resolutions are not constant across all images, ranging from 500x500 to 1920x1920 with differing aspect ratios.

2. The Fashion-MNIST is a dataset of Zalando's article images, available as part of TensorFlow's data set at this link. This has 70,000 28x28 grayscale images, which includes 7,000 images each of the following ten categories - Ankle boot, Bag, Coat, Dress, Pullover, Sandal, Shirt, Sneaker, T-shirt/top and Trouser. A sample image from this data set is shown in Figure 1 (b)

(a) Abstract Art (b) Fashion-MNIST

Figure 1: Samples of the Two Data Sets Used

4 Methodology

4.1 GANs

The main idea behind GANs can be interpreted as a game between two competing neural networks - the generator and the discriminator:

1. **Generator** — Takes a random vector as input (a random point in the latent vector space), and decodes it into a synthetic image. It aims to generate samples that follow the same data distribution as the training data.

2. **Discriminator** (or adversary) — Takes a real or synthetic image as input, and predicts whether the image is real or fake (that is whether it came from the training set or was created by the generator network). It tries to create a hard classifier between the samples generated by the generator (fake data), and the actual data from the training set (real data).

The generator's goal is to fool the discriminator by approximating the training data distribution. The generator network is trained to pass the discriminator network and evolve iteratively toward generating increasingly realistic images in the training process. The training is continued until it is impossible for the discriminator network to distinguish between the real and synthetic images. Meanwhile, the discriminator is constantly adapting to the gradually improving capabilities of the

generator, setting a high bar of realism for the generated images. Once training is over using a suitable set of hyperparameters, the generator can turn any point in its input space into a statistically similar image. As introduced before, a common analogy of this is that of an art forger trying to fool people into believing forged images as real. The police have the task of identifying forgeries and separating them from the real paintings. The forger is like the generator, trying to counterfeit money or paintings, making them look as legitimate as possible to fool the police. The police are like the discriminator whose goal is to be able to identify the counterfeited items.

This most basic sort of GANs is known as Vanilla GANs, consisting only of the generator and discriminator. Internally, the generator and discriminator use multi-layer perceptrons to generate and classify images. The discriminator seeks to determine the likelihood that the input belongs to a particular class while the generator captures the distribution of the data. After computing the loss function, feedback is sent to both the generator and discriminator, where the effort to minimize the loss is made.

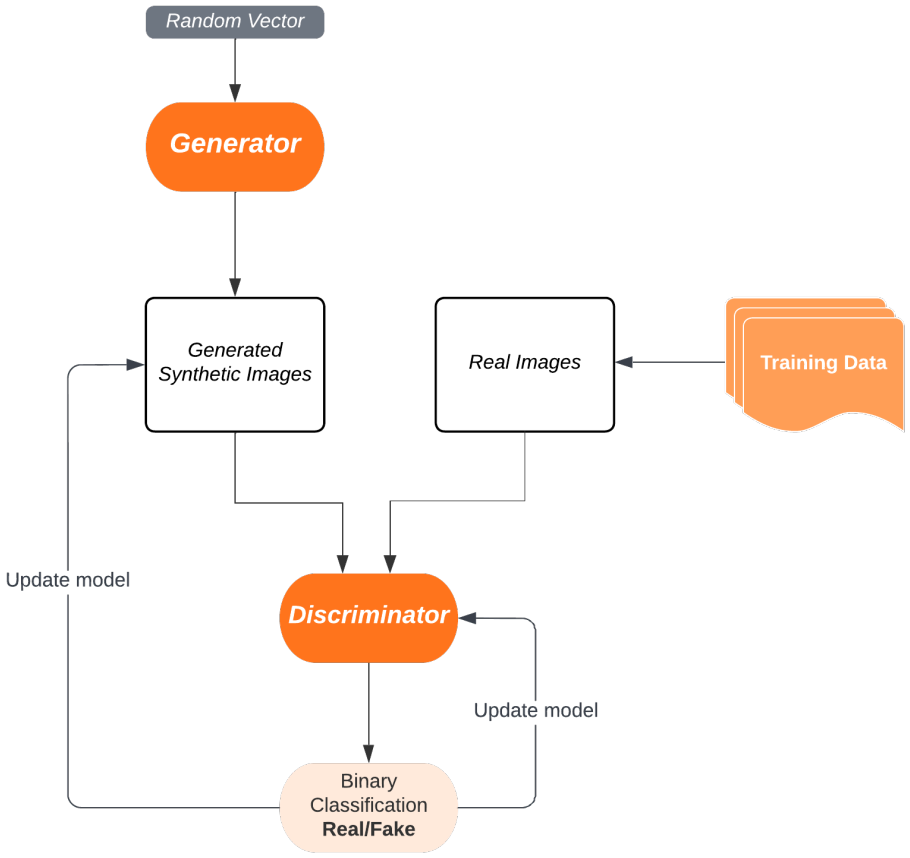


Figure 2: Generative Adversarial Network Model Architecture

4.2 Focus - DCGANs

DCGANs use convolutional layers and Fractionally-strided convolutional layers in the generator and discriminator, respectively. This was introduced by Radford et. al. in 2016, in this paper.

Fractionally-strided convolutions (FSCs) transpose images, typically from a minimized format

to a larger one. Every FSC layer increases the input resolution by a factor of 2, ensuring that the output resolution is the same as the input. FSCs are also known as deconvolutions or transposed convolutions. Generative models in GAN architecture are required to upsample input data to generate an output image. The transposed convolutional layer is an upsampling layer with no weights that will double the input dimensions.

The discriminator consists of convolution layers, batch normalization layers, and LeakyRelu as the activation function. It takes a 64x64x3 input image. The generator consists of convolutional-transpose layers, batch normalization layers, and ReLU activations. The output will be a 3x64x64 RGB image. The generator is comprised of convolutional-transpose layers, batch normalization layers, and ReLU activations. The input to the generator is a latent vector drawn from a standard normal distribution, and the output is a 64x64x3 RGB image. The strided transpose layers allow the latent vector to be transformed into a volume with the same shape as an image. As shown in Figure 3, a 100D uniform distributed vector is projected to spatial convolutions through a series of 4 FSCs to output the image with the required dimensions. There are no fully connected or pooling layers used, which is a significant difference from the original Vanilla GANs

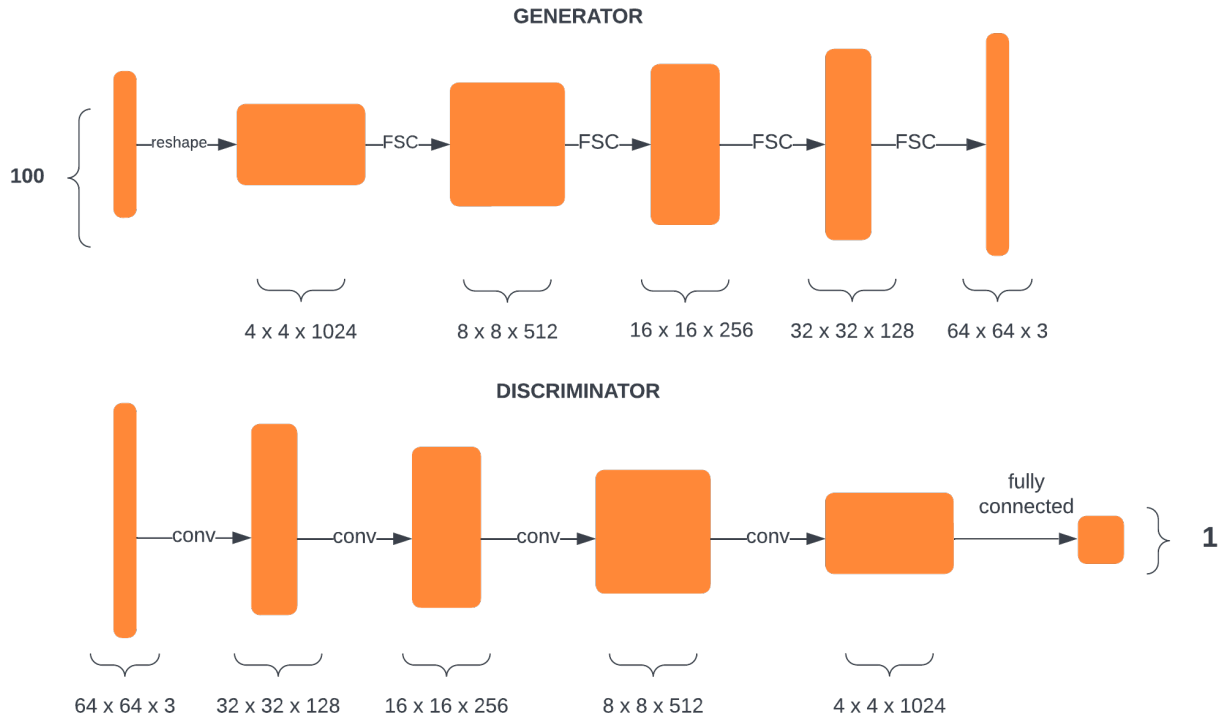


Figure 3: Deep Convolutional GAN Model Architecture

Batch Normalization is used in all layers of both generator and discriminator for faster convergence. This standardizes the data to a constant mean and standard deviation, like Gaussian distribution, to stabilize the training.

In gradient descent optimization problems such as GANs, we may encounter the vanishing gradient problem. During each training epoch, the weights are given an update proportional to the partial derivative of the error function. When the gradient is very small, it prevents the weights from updating. Activation functions are used to address the problem of vanishing gradients. ReLU activation function is used in the generator except for the output layer, which uses the Tanh

function, and Leaky ReLu is used in the discriminator. ReLU restricts values between 0 and 1, whereas Leaky ReLu allows for controlled negative values. The output is positive when a positive input is passed to the ReLU or LeakyReLU. When negative values are passed, ReLU outputs 0, whereas Leaky ReLU outputs a small negative value. GAN researchers observed that using a bounded activation allowed the model to quickly saturate and cover the color space of the training distribution. Tanh activation function is used in the final output layer to ensure gradients are restricted between -1 and 1 and not zero to prevent stopping the learning process. Since the tanh function output is zero centered; hence we can map the output as strongly negative, neutral, or strongly positive.

In summary, Deep convolutional GANs are used to produce high-resolution and high-quality images, as they uphold the spatial aspect of the dataset. In the Discriminator, all layers utilize the convolutional layers, batch normalization, and Leaky-ReLu activation. In contrast, in the Generator, all layers use transposed convolutional layers, batch normalization, and ReLU activation except the final layer (Tanh).

4.3 Other types of GANs

Conditional GAN (CGAN)
In this GAN, the extra information—a class label or any modal data—is given to the generator and discriminator. The additional information, as the name implies, aids the discriminator in determining the conditional probability as opposed to the joint probability.
Loss function of the conditional GAN:

$$\min_G \max_D E_{\mathbf{x}\sim p_r} \log[D(\mathbf{x} \mid \mathbf{y})] + E_{\mathbf{z}\sim p_\mathbf{z}} \log[1 - D(G(\mathbf{z} \mid \mathbf{y}))]$$

CycleGAN
This GAN is designed to map one image to another or image-to-image translations. For instance, if summer and winter are subjected to the Image-Image translation process, we discover a mapping function that could transform summer images into winter images and vice versa by adding or removing features in accordance with the mapping function, such that the predicted output and actual output have the smallest loss.

Generative Adversarial Text to Image Synthesis
In this, the GANs can find an image from the dataset closest to the text description and generate similar images. The generator network is trying to generate based on the description, and the differentiation is done by the discriminator based on the features mentioned in the text description.

Style GAN
Other GANs are focused on improving the discriminator, but we improve the generator in this case. This GAN generates by taking a reference picture. The Style GAN uses the AdaIN or the Adaptive Instance Normalization, which is defined as

$$AdaIN\left(\mathbf{x}_i, \mathbf{y}\right) = \mathbf{y}_{s,i} \frac{\mathbf{x}_i - \mu\left(\mathbf{x}_i\right)}{\sigma\left(\mathbf{x}_i\right)} + \mathbf{y}_{b,i}$$

Here, x_i is the feature map, which is tweaked by using the component y.

Super Resolution GAN (SRGAN)

The primary purpose of this type of GAN is to make a low-resolution picture more detailed. This is one of the most researched problems in Computer vision. The Generator takes the input data and finds the average of all possible solutions that pixel-wise, when formulated, looks like:

$$\hat{\theta}_G = \arg\min_{\theta_G} \frac{1}{N} \sum_{n=1}^{N} l^{SR}\left(G_{\theta_G}\left(I_n^{LR}\right), I_n^{HR}\right)$$

here, I^{hr} represents high resolution image , whereas I^{lr} represents low resolution image and l^{sr} represents loss and $_g$ represents the weight and biases at layer L.
Now the perceptual loss is given as

$$l^{SR} = \underbrace{\underbrace{l_{\mathrm{X}}^{SR}}_{contentloss} + \underbrace{10^{-3} l_{Gen}^{SR}}_{adversarialloss}}_{perceptualloss(forVGGbasedcontentlosses)}$$

Finally, the content loss is summed with adversarial loss, and the Mean Squared Error is taken to find the best pixel-wise solution.

5 Math Behind GAN

x : Real data
z : Latent vector
$G(z)$: Fake data
$D(x)$: Discriminator's evaluation of real data
$D(G(z))$: Discriminator's evaluation of fake data
$Error(a, b)$: Error between a and b

The purpose of the discriminator is to correctly label generated images as false and empirical data points as true. It performs hard classification. Therefore, the loss function of the discriminator can be written as:

$$L_D = Error(D(x), 1) + Error(D(G(z)), 0)$$

The goal of the discriminator is to reduce the error between the probability distribution of the real data and the correct label (1), as well as to minimize the error between that of the synthetic data and its correct label (0).

The purpose of the generator is to confuse the discriminator in labeling the generated images as true. The loss function for the generator can be similarly written as:

$$L_G = Error(D(G(z)), 1)$$

We need to minimize the loss function, and in this case, we need to minimize the difference between the label of the true data (1) and the discriminator evaluation of the generated data.

The binary cross entropy function is commonly used as the loss function in binary classification problems.

$$H(p, q) = E_{x \sim p(x)}[-\log q(x)]$$

In classification, we can rewrite the above loss function as a summation, due to the random variable being discrete:

$$H(p, q) = -\sum_{x \in X} p(x) \log q(x)$$

Since there are only two labels: zero and one.

$$H(y, \hat{y}) = -\sum y \log(\hat{y}) + (1-y)\log(1-\hat{y}) - -(1)$$

This is the Error function that we have been loosely using above. Binary cross entropy measures how different two distributions are in the binary classification of true or false data points.

Hence we can write,

$$L_D = -\sum_{x \in X, z \in Z} \log(D(x)) + \log(1 - D(G(z)))$$

and

$$L_G = -\sum_{z \in Z} \log(D(G(z))$$

We need the loss function to be minimized when $D(G(z))$ is almost 1, $\log(1) = 0$. In the end, we want the final generated image to show this behavior.

The overall objective function of GANs can be intuitively written as:

$$\min_G \max_D \{\log(D(x)) + \log(1 - D(G(z)))\}$$

Here the discriminator tends to maximize the given quantity, and the generator tries to do the reverse. This demonstrates the adversarial nature of the competition between the two blocks. Throughout the training process, we expect the discriminator and generator to compete against each other in terms of the loss values. We need an optimal G that minimizes this objective and an optimal D that maximizes it.

The following function of G and D is called the value function:

$$V(G, D) = E_{x \sim p_{data}}[\log(D(x))] + E_{z \sim p_z}[\log(1 - D(G(z)))]$$

Substituting, $y = G(z)$:

$$V(G, D) = E_{x \sim p_{data}}[\log(D(x))] + E_{y \sim p_g}[\log(1 - D(y))] = \int_{x \in \chi} p_{data}(x)\log(D(x)) + p_g(x)\log(1 - D(x))dx$$

The discriminator tries to maximize the value function, and we find an optimal discriminator $D^*(x)$ occur after a partial derivative of $V(G, D)$ with respect to $D(x)$

$$\frac{p_{data}(x)}{D(x)} - \frac{p_g(x)}{1 - D(x)} = 0$$

$$D^*(x) = \frac{p_{data}(x)}{p_{data}(x) + p_g(x)}$$

Training the generator, and keeping the discriminator fixed, we analyze the value function, plugging in $D^*(x)$ we get

$$V(G, D^*) = E_{x \sim p_{data}}\left[\log\left(D^*(x)\right)\right] + E_{x \sim p_g}\left[\log\left(1 - D^*(x)\right)\right]$$

$$= E_{x \sim p_{data}}\left[\log \frac{p_{data}(x)}{p_{data}(x) + p_g(x)}\right] + E_{x \sim p_g}\left[\log \frac{p_g(x)}{p_{data}(x) + p_g(x)}\right]$$

The equation for the similarity between two probability distributions given by the Jensen–Shannon divergence method is as below

$$JS\left(p_1 \| p_2\right) = \frac{1}{2} E_{x \sim p_1} \ln\left(\frac{p_1}{\frac{p_1+p_2}{2}}\right) + \frac{1}{2} E_{x \sim p_2}\left(\frac{p_2}{\frac{p_1+p_2}{2}}\right)$$

$V\left(G, D^*\right)$ can be accordingly written as

$$V\left(G, D^*\right) = E_{x \sim p_{data}}\left[\log \frac{2p_{data}(x)}{p_{data}(x) + p_g(x)}\right] + E_{x \sim p_g}\left[\log \frac{2p_g(x)}{p_{data}(x) + p_g(x)}\right] - 2log2$$

$$V\left(G, D^*\right) = -\log 4 + 2 \cdot JS\left(p_{data} \| p_g\right)$$

Minimizing the value function made the JS divergence between the data distribution and generated data as small as possible. This is the same as saying that we want the generator to learn the data distribution from the training data. In other words, p_g and p_{data} should be as close to each other as possible. This is the optimal G that makes the two distributions very close.

6 Algorithms

6.1 Generator and Discriminator Code Blocks

Generator - Following steps are followed in order to build the generator block -

1. Initialize Sequential container to add chains of NN modules, using Sequential() function of the nn module in PyTorch package.
2. Create neural net layers to execute in order - add Transposed Convolutional 2D layers to upsample input data using ConvTranspose2D, apply Batch Normalization using BatchNorm2D, and apply rectified linear unit function (ReLU) element-wise (activation) using ReLU. All these functions are a part of the PyTorch package.
3. Repeat Step 2 for the number of layers required to achieve the desired upsampling (6 times in this case)
4. Apply the tanh activation function to the final output layer using tanh in PyTorch.

Discriminator - Following steps are followed in order to build the discriminator block -

1. Initialize Sequential container to add chains of NN modules, using Sequential() function of the nn module in PyTorch package.
2. Create neural net layers to execute in order - add Convolutional 2D layers to downsample input image data using Conv2D, apply Batch Normalization using BatchNorm2D, and apply Leaky rectified linear unit function (LeakyReLU) element-wise (activation). All these functions are a part of the PyTorch package.
3. Repeat Step 2 for the number of layers required to achieve the desired upsampling (6 times in this case)
4. Apply sigmoid activation to the final layer out function to restrict value between 0 and 1 (range of valid probabilities)

6.2 Training

The original paper by Ian Goodfellow on GANs was studied and the following algorithm was used by the researchers to train GANs

Minibatch stochastic gradient descent training of generative adversarial nets. The number of steps to apply to the discriminator, k, is a hyperparameter.
for max_iter (number of training iterations) do
for k steps (start inner loop)
do

- Sample minibatch of m noise samples $\left\{z^{(1)}, \ldots, z^{(m)}\right\}$ from noise prior $p_g(z)$
- Sample minibatch of m examples $\left\{x^{(1)}, \ldots, x^{(m)}\right\}$ from data generating distribution $p_{data}(x)$
- Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(x^{(i)}\right) + \log\left(1 - D\left(G\left(z^{(i)}\right)\right)\right)\right]$$

end for (end inner loop)

- Sample minibatch of m noise samples $\left\{z^{(1)}, \ldots, z^{(m)}\right\}$ from noise prior $p_g(z)$
- Update the generator by descending its stochastic gradient:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(z^{(i)}\right)\right)\right)$$

end for

Based on this, the following algorithms were implemented to train the generator and discriminator blocks. Each of the steps listed under Discriminator and Generator was repeated for all epochs, which is one of the tuning hyperparameters.

Discriminator (D) –

1. Pass the transformed batch of real images to D (Section 7.1 for more details) and get the discriminator score predictions for the real images
2. Calculate binary cross entropy (BCE) which is the loss function, of real predictions compared with target predictions which is 1 for real data (binary_cross_entropy in nn. Functional module in PyTorch)
3. Calculate mean discriminator scores for the real predictions (torch.mean)
4. Pass a random vector in latent dimensional space to the generator and get the synthetic image as output
5. Pass the generated batch of fake images to D and get the discriminator score predictions for the fake images

6. Calculate BCE for these generated predictions comparing with target predictions which are 0 for fake data for the discriminator
7. Calculate mean discriminator scores for the fake predictions
8. Add the two BCEs for total discriminator loss
9. Use backward and step functions to update the gradients of the calculated loss

Generator (G) –

1. Pass a random vector in latent dimensional space to the generator and get the synthetic image as output
2. Pass the generated batch of fake images to D and get the discriminator score predictions for the fake images
3. Calculate BCE for these generated predictions comparing with target predictions which are 1 for fake data for the generator
4. Use backward and step functions to update the gradients of the calculated loss

7 Implementation

7.1 Training Data Processing

Using the image transformations function Compose in the transforms module of the torchvision package, the following steps were undertaken to process the training data -

1. Resize images to same size 128x128 (Resize)
2. Crop the images at the center (CenterCrop)
3. Randomly flip images horizontally and vertically each with a probability of 0.5 to ensure data augmentation. (RandomHorizontalFlip and RandomVerticalFlip)
4. Convert to Tensor object (ToTensor)
5. Normalize using difference with mean 0.5, divided by standard deviation 0.5 in all dimensions (Normalize)

We created shuffled batch sizes of 128 images using the DataLoader function in utils.data module in the PyTorch package.

7.2 Python in Google Colab

We majorly used PyTorch and Torchvision with some uses of Tensorflow, tqdm, matplotlib as well. We began with utilizing the free GPU runtime accelerator on Google Colab. The dataset was uploaded to Google Drive and mounted to Colab. To quicken the progress, we purchased 100 computing units to be able to run the abstract art dataset multiple times while tuning the hyperparameters.

7.3 Challenges and Limitations

There are cost and hardware challenges while building and running the GAN model. In a few of the examples of GANs that we have seen online, a large number of iterations were required before convergence, implying the need for high computing power. The usage of GANs involves complex calculations, due to which GPU-enabled machines will make the process easier. We used Google Colab with GPU acceleration to decrease the training time. Google Colab offers additional features such as collaborative development, cloud-based hosting, and GPU and TPU accelerated training with the free version of Google Colab.

The Performance of GANs can improve with more epochs and larger data sets. Due to system hardware limitations and restricted Colab usage, we could not train more than 200 epochs as we would run into shutdown messages (runtime disconnected, maximum usage reached, etc.).

Running GANs comes with challenges in the quality of output as well. We encountered the problem of Mode Collapse when we tried implementing the entire program using the Keras library. In this, the generator can generate only a limited variety of data and cannot update further. We tried various versions of hyperparameter tuning, altered the number of epochs, and increased the data set but could only get good-quality outputs after switching the entire implementation to PyTorch. Following is an example of Mode collapse that we encountered -

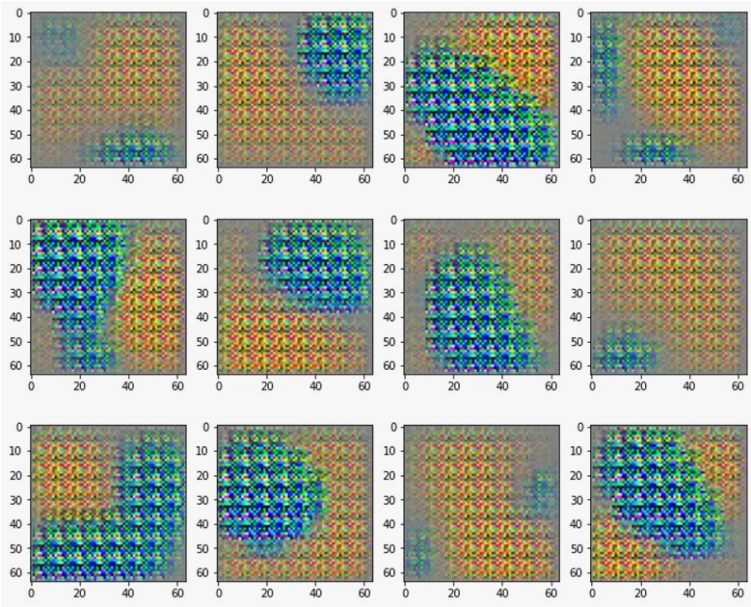


Figure 4: Example of Mode Collapse

8 Evaluation and Final Results

8.1 Fashion Dataset

To test the GAN model that was developed, the fashion MNIST dataset was fed into the model, where the images were transformed, and the discriminator and the generator functions were applied to all the training epochs. The dataset that was used to train the model Fig:5 included images of footwear, handbags, and other clothing items.

Figure 5: Training Dataset for Fashion MNIST

The model was run for 50 epochs, and, at each epoch, generated images were saved for analysis fig:6. Initially, there was only noise, but the model quickly adapted. It was able to start extracting key features from the training dataset and began to generate images that were statistically similar to the training set. Around step 42 is when the clearest images were noticed. Footwear, handbags, and other clothing items were clearly discernible. After step 42, a diminishing return was noticed as the steps/epochs were increased the images became slightly noisier.

During the most time-efficient run using the Google Colab GPU accelerator, we could run each epoch in approximately 0.5 seconds, with 50 epochs taking nearly 30 seconds to run entirely. The loss and score outputs were plotted vs. training epochs, using the smoothing function Univariate-Spline using a positive smoothing factor of 10 for the loss plot:7 (a).

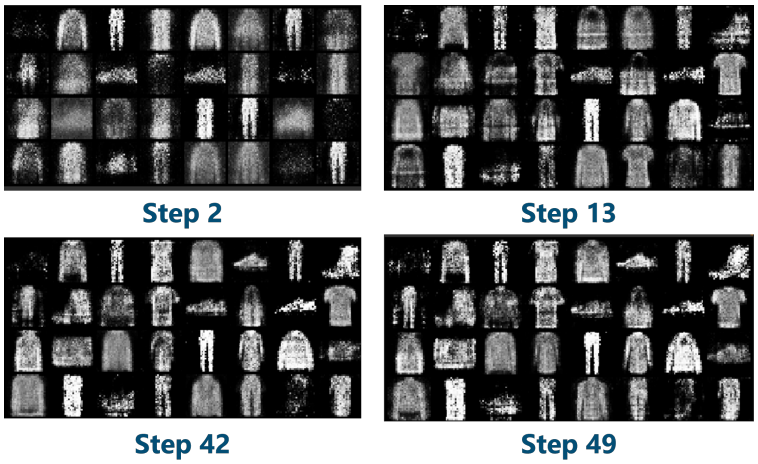
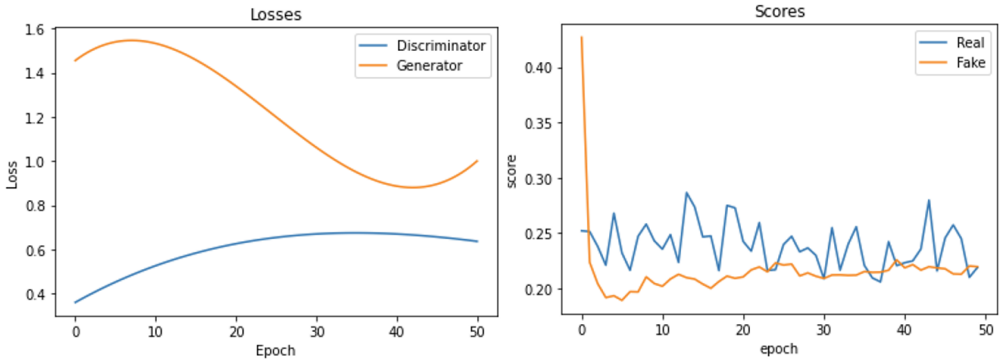


Figure 6: Generated Images from the Fashion MNIST dataset

One key indicator for the performance of a GAN is the Generator and Discriminator loss. As discussed previously, the objective cost function of a GAN is adversarial in nature. That same property can be noticed in the loss plotted for the Fashion MINST dataset fig:7 (a). Initially, as the generator loss decreases, the discriminator loss increases. Around step/epoch 42, there is an inflection point where the generator loss increases and the discriminator loss decreases. This explains why the images become slightly noisier after step/epoch 42.

The generated images' scores can indicate any underlying flaws in the model. As mentioned before, GANs learn to generate statistically similar data. This can explain the score plot generated to analyze the scores of the real and fake images fig:7 (b). Initially, there is a significant difference between the real and fake images, but as we increase the steps/epochs, the scores for the fake images drop rapidly to match the scores of the real images. Thus, the generated images perform as expected.



(a) Loss (b) Score

Figure 7: Performance Plots for the Abstract Art Generation Process

8.2 Abstract Art

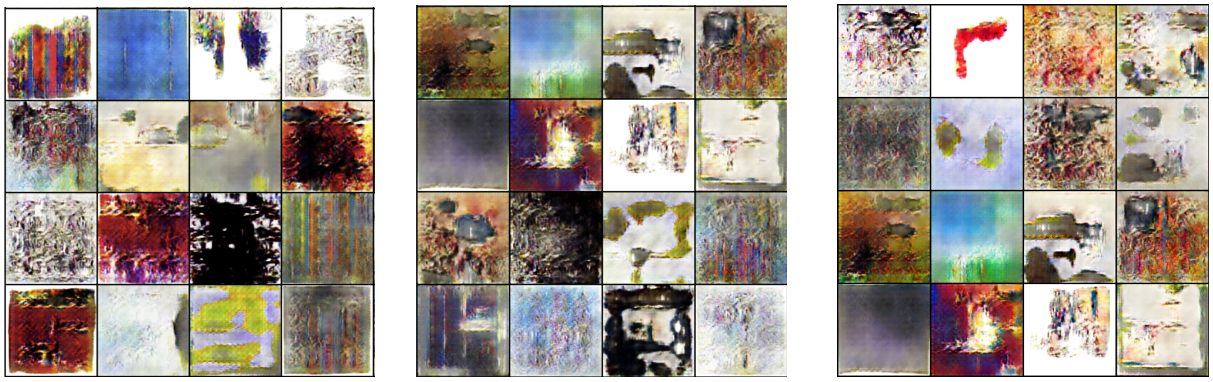
After the performance check in the test run on the fashion MNIST data set, the abstract art data set was fed into the model. The steps in Sections 6 and 7 were run for this data set. This was followed by hyper-parameter tuning for the various operations in the neural network layers. The images that were used to train the model fig:8 included a collection of abstract art from a wide range of artists.

Figure 8: Training Dataset for Abstract Art

The model was run for 200 epochs on the Google Colab GPU hardware accelerator. Moreover, at each epoch, the generated images were saved for analysis. Initially, there was only noise, but the model quickly adapted. It was able to start extracting key features from the training data set and began to generate images that were statistically similar to the training set. The result from the model was very reminiscent of the training data set and far exceeded expectations. A few images from each of the three successful rounds of testing are shown in Fig:9.

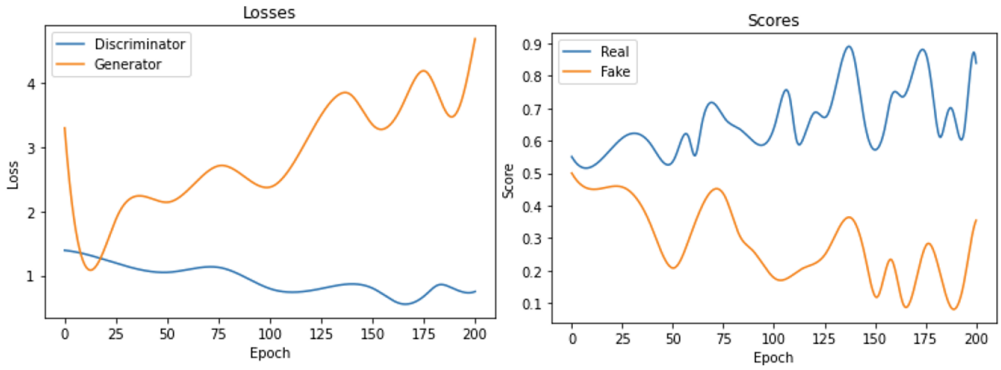
In the most optimal run using the Google Colab GPU accelerator, we could run each epoch in approximately 30 seconds, with 200 epochs taking around 2 hours to run entirely. The loss and score outputs were plotted vs. training epochs, using the smoothing function UnivariateSpline using a positive smoothing factor of 25 for discriminator loss and 200 for generator loss. For scores, it is 5 for both the generator and discriminator.

The loss for the abstract art data set fig:10 (a), was similar in nature to the loss of the fashion MINTS data set fig:7 (a). The adversarial effect of the cost function was noticed here as well, as then the generator loss decrease the discriminator loss increased, and vice versa.



(a) First Round (b) Second Round (c) Third Round

Figure 9: Snippets of Generated Abstract Art Images in Three Successful Testing Rounds



(a) Loss (b) Score

Figure 10: Performance Plots for the Abstract Art Generation Process

The score for the abstract art dataset fig:10 (b), was similar in nature to the score of the fashion MINTS dataset fig:7 (b). The model is trying to generate images that are statistically similar to the training data. This is why, the real images have a relatively constant score, whereas, the fake images are trying to normalize.

This technique can be extended to real applications such as the generation of new cartoon characters, human faces generation, conversion of earth view maps to satellite views and vice versa, adding color to black and white images, converting night images to day and vice versa, etc. The possibilities are endless with the models still evolving. The quality and usability very heavily dependent on the strength and capacity of the hardware used

9 Contribution Table

	Main Lead	Contributor
Coding and Training Models		
Fashion	Oojas	Yash
Abstract Art	Yash	Oojas
Presentation		
Presenter	Oojas	
Design and Structure	Yash	
Content	Yash	Oojas
Results Section	Oojas	
Report		
Structure	Yash	
Introduction	Yash	
Sourcing Data	Oojas	Yash
Methodology	Yash	Oojas
Math	Yash	Oojas
Algorithms	Yash	Oojas
Implementation	Yash	
Evaluation and Final Results	Oojas	Yash

Figure 11: Contribution Table

10 References

1. Deep Learning by Python, book by François Chollet
2. GANs in Action, book by Jakub Langr
3. Opengenus Types of Generative Adversarial Networks (GANs)
4. The Math Behind GANs - JakeTae, Youtube video by Normalized Nerd
5. Data Science Stack Exchange
6. Medium: DCGAN: Deep Convolutional Generative Adversarial Network
7. Google Developers: Common Problems with GANs
8. YouTube videos: Edureka, IBM Tech and Generating Pokemon with a Generative Adversarial Network
9. Python Documentations: Pytorch, Tensorflow and Keras
10. Machine Learning Mastery: GANs Training Tips
11. DCGAN Tutorial by PyTorch