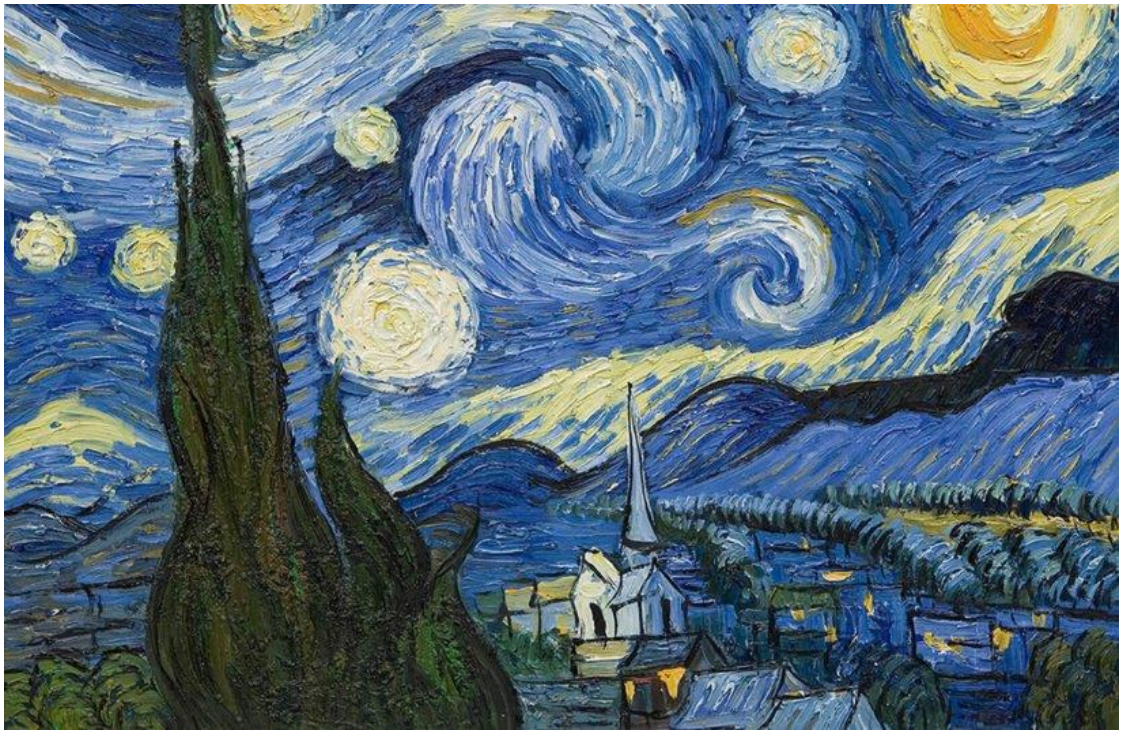
AMRITA VISHWA VIDYAPEETHAM

Creating works of art with GANs

Subject Name: Neural Networks and Deep Learning
Subject Code: 15CSE380



Project By:

Pranav R Nambiar (BL.EN.U4CSE18093)

Sourabh Sooraj (BL.EN.U4CSE18118)

Sreedathan G (BL.EN.U4CSE18119)

S.A.S Sridatta (BL.EN.U4CSE18121)

Project guide: Dr. Suja P.

Abstract

Generative Adversarial Networks, or GANs for short, is an approach to generative modeling using deep learning methods, such as convolutional neural networks. Generative modeling is an unsupervised learning task in machine learning that involves automatically discovering and learning the regularities or patterns in input data in such a way that the model can be used to generate or output new examples that plausibly could have been drawn from the original dataset. GANs are thus set up to utilize the strength of deep discriminative models to bypass the necessity of maximum likelihood estimation and therefore avoid the main failings of traditional generative models. Neural style transfer is an optimization technique used to take three images, a content image, a style reference image (such as an artwork by a famous painter), and the input image you want to style — and blend them such that the input image is transformed to look like the content image, but "painted" in the style of the style image.

In the modern-day, artworks have long become an afterthought. The youth just doesn't seem to get the same joys when visiting a museum and viewing a fine piece of art by great Renaissance painters like Leonardo Da Vinci and Picasso. Instead, they are replaced by filters as seen in Selfie apps. This undermines the effort that these noble and honorable men have put in to bring these beautiful pieces of work to life. So today with modern Neural Networks we seek to reverse that trend and get people talking about these great masterpieces by recreating our own.

Implementation of CNN in our Project

GANs are a clever way of training a generative model by framing the problem as a supervised learning problem with two sub-models: the generator model that we train to generate new examples, and the discriminator model that tries to classify examples as either real (from the domain) or fake (generated). The two models are trained together in a zero-sum game, adversarial, until the discriminator model is fooled about half the time, meaning the generator model is generating plausible examples.

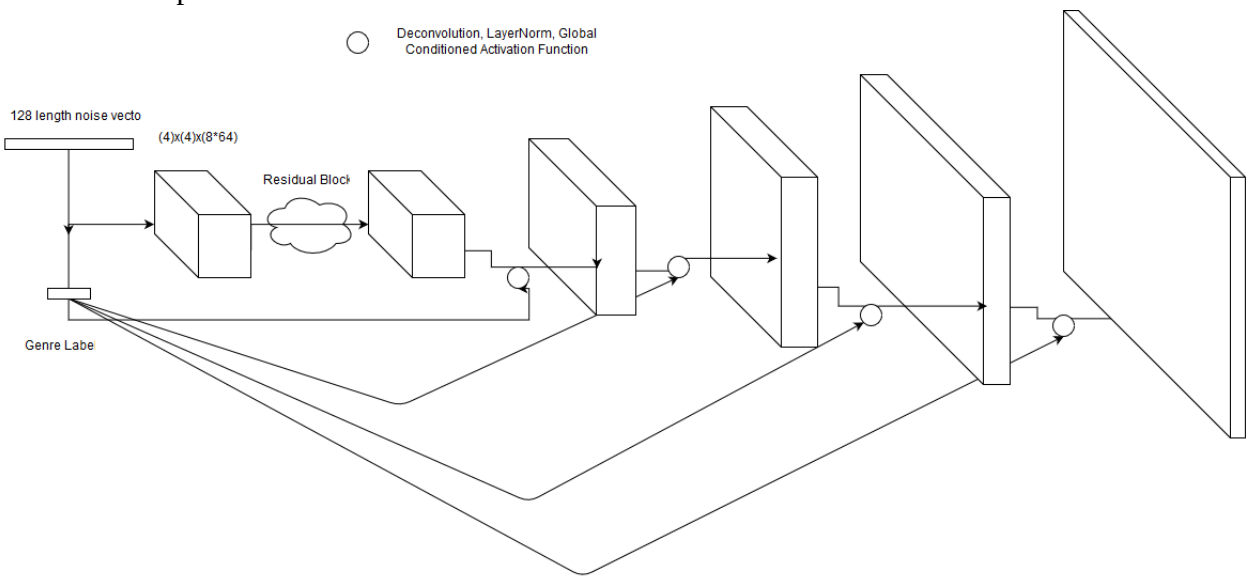


Fig 1. Generator

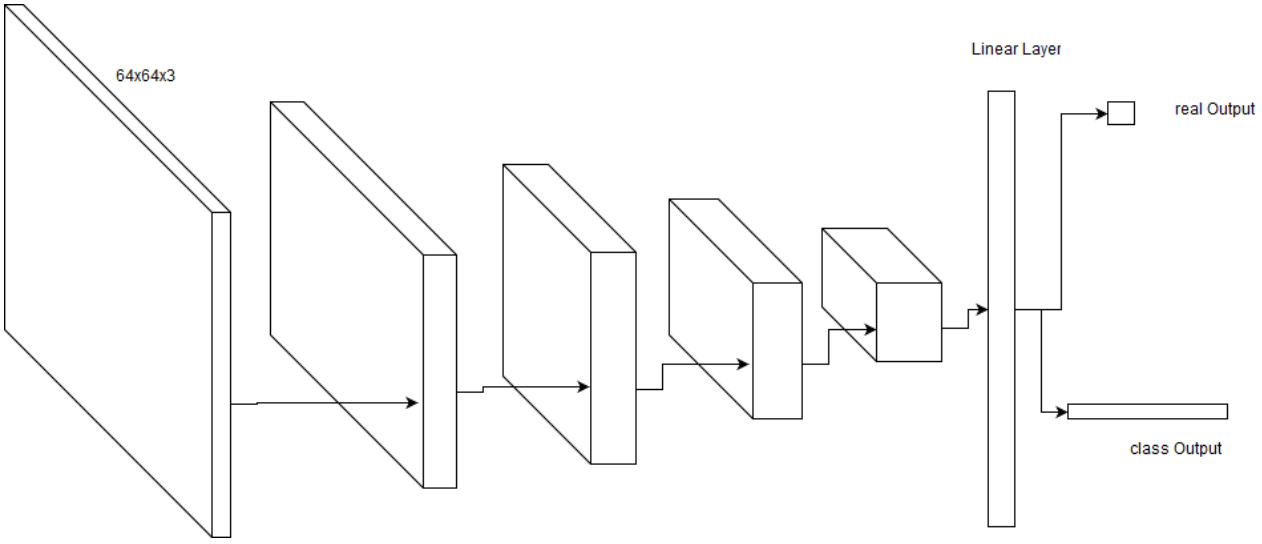


Fig 2. Discriminator

PyTorch

```
 # Create optimizers
    opt_d = torch.optim.Adam(discriminator.parameters(), lr=lr, betas=(0.5, 0.999)) #Converges faster and produces lower loss for the generator
    opt_g = torch.optim.Adam(generator.parameters(), lr=lr, betas=(0.5, 0.999))

# LeakyRelu is used in discriminator for all layers
    nn.LeakyReLU(0.2, inplace=True),
#LeakyReLU helps prevent a lot of dead values as compared to ReLU, hence more accurate results as -ve values pass through all layers
    # out: 64 x 32 x 32

    nn.Conv2d(64, 128, kernel_size=4, stride=2, padding=1, bias=False),
    nn.BatchNorm2d(128),
    nn.LeakyReLU(0.2, inplace=True),
    # out: 128 x 16 x 16

    nn.Conv2d(128, 256, kernel_size=4, stride=2, padding=1, bias=False),
    nn.BatchNorm2d(256),
    nn.LeakyReLU(0.2, inplace=True),
    # out: 256 x 8 x 8

    nn.Conv2d(256, 512, kernel_size=4, stride=2, padding=1, bias=False),
    nn.BatchNorm2d(512),
```

```
    nn.LeakyReLU(0.2, inplace=True),
    # out: 512 x 4 x 4

    nn.Conv2d(512, 1, kernel_size=4, stride=1, padding=0, bias=False),
    # out: 1 x 1 x 1

    nn.Flatten(),
    nn.Sigmoid()) #We use a logistic sigmoid activation function because it needs to put out probabilistic values (0 OR 1, FAKE OR REAL)
```

Tensorflow

```
    model = Sequential()

    model.add(Dense(64*64,activation="relu",input_dim=seed_size))     #64x64 units
    model.add(Reshape((4,4,256)))
  # Relu activation layer except the output layer

    model.add(UpSampling2D())
    model.add(Conv2D(256,kernel_size=3,padding="same"))
    model.add(BatchNormalization(momentum=0.8))#To help the gradient descent converge faster and reduces chances of overfitting
    model.add(Activation("relu"))

    model.add(UpSampling2D())
    model.add(Conv2D(256,kernel_size=3,padding="same"))
    model.add(BatchNormalization(momentum=0.8))
    model.add(Activation("relu"))

    model.add(Conv2D(256,kernel_size=3,padding="same"))
    model.add(BatchNormalization(momentum=0.8))
    model.add(Activation("relu"))

    # Optional additional upsampling goes here

    model.add(UpSampling2D(size = (2,2))) #4,4 for 128x128, 2,2 for 64x64
```

```
    model.add(Conv2D(256,kernel_size=3,padding="same"))
    model.add(BatchNormalization(momentum=0.8))
   model.add(Activation("relu"))#We use a ReLU to address the Vanishing Gradient problem

    model.add(UpSampling2D(size=(2,2)))
    model.add(Conv2D(256,kernel_size=3,padding="same"))
    model.add(BatchNormalization(momentum=0.8))
# tanh is used in the output layer
    model.add(Activation("relu"))

    model.add(Conv2D(channels,kernel_size=3,padding="same"))
    model.add(Activation("tanh"))  #We observed that using a bounded activation allowed the model to learn more quickly to saturate and cover the color space of the training distribution. The output will be mapped to (-1,1)


    return model
```

Hyperparams: Batch size, no. of epochs, Learning rate, Activation Function, Beta, batch size x iterations = no. of epochs, no. of layers. For the generator, we use strided convolution, whereas in the case of the discriminator, we go for fractional strided convolution.

Tools used

- PyTorch
- Tensorflow
- Keras

Dataset

The data set is a collection of artworks of different artists(Best art like Da Vinci, Michaelangelo, and other influential artists. Link here
artists.csv: a dataset of information for each artist.
images.zip: a collection of images (full size), divided into folders and sequentially numbered.
resized.zip: same collection but images have been resized and extracted from a folder structure.

Python Libraries Used:

- os
- torchvision
- matplotlib
- pandas

Description of Python modules used:

def discriminator_loss(real_output, fake_output):

The discriminator's loss is based on its ability to distinguish real images from fakes. It compares its predictions on real images to an array of ones (remember 1 being real) and its predictions on fake images to an array of zeros (0 being fake). The goal is to classify all real images as 1 and all fakes as 0. The total loss is then these two losses added together.

def generator_loss(fake_output):

The generator's loss is a measurement of how good it performed at fooling the discriminator.

If the discriminator classifies the fake images as 1, the generator did a good job.

Problems faced:

Problem: The model does not have very high accuracy.

Solution: We trained the model for 5000 epochs to achieve a relatively better output.

Problem: The Streamlit app takes a lot of time to load.

Solution: Reduced the number of training and testing images.

Result

An artwork model was successfully designed using GANs where the generator was successful in generating fake images on its own from scratch and the discriminator after discarding fake images trained the generator to make perfect images thereby making the discriminator believe that the artwork is perfect. The images progressed from pretty bad shaped to having sharp features as the training progressed to an extent that it resembled actual art-works in the data-set model that we used from Kaggle.

Contribution of each team member

Pranav R Nambiar: GANs implementation using PyTorch
Sourabh Sooraj: GANs implementation using PyTorch
Sreedathan G: GANs implementation using TensorFlow
Sriram Ananta Sai Sridatta: GANs implementation using TensorFlow

Scope of future work

Any future work should try to fix the following issues:

- Inconsistent coloring- objects contain spatially inconsistent colors, such as color shifts.
- Few distorted images- can be improved when implementing training with better accuracy.

A substantial limiting factor in our project was our access to computational resources. Working with GPUs that had only 2 GB of RAM, we were forced to stick to producing 64x64 pixel images, as any higher resolution forced significant reductions in both our batch size and dimensionality of the model. We noticed a significant improvement in our model when comparing the generated images from our 64x64 models to our initial 32x32 models, so we hypothesize that with additional computational resources that allow scaling to 128x128 images, the quality of the images would also substantially increase under our current architecture.

Whether you will extend the project:

Yes, we would like to improve the accuracy and the quality of the images produced.

Hopefully, with the inclusion of Auto-Encoders, we will be able to achieve a better result.