Image Generation Using Generative Adversarial Network (GANS)

By

ABHISHEK VERMA 16BEC1075
PIYUSH SINGH 16BEC1120
KRISHNANUNNI NAIR 16BEC1080

A project report submitted to

Dr. Menaka R.

SCHOOL OF ELECTRONICS ENGINEERING

in partial fulfilment of the requirements for the course of

Neural Networks & Fuzzy Control (ECE3009)

in

B.Tech. ELECTRONICS AND COMMUNICATION ENGINEERING

VIT, CHENNAI CAMPUS

Vandalur – Kelambakkam Road

Chennai – 600127

November 2019

BONAFIDE CERTIFICATE

Certified that this project report entitled "**IMAGE GENERATION USING GENERATIVE ADVERSARIAL NETWORKS (GANs) "**is a bonafide work of Abhishek Verma (16BEC1075), Piyush Singh (16BEC1120), Krishnanunni Nair (16BEC1080) who carried out the Project work under my supervision and guidance.

Dr. Menaka R.

Associate Professor

School of Electronics Engineering (SENSE),

VIT, Chennai Campus

Chennai – 600 127.

ABSTRACT

In this project we have used Generative Adversarial Networks (GANs) to produce new images from a variety of training dataset images. It consists of a Generator, which at first produces static noise images and as it learns from the Discriminator, the generator starts improving the image quality and stars producing images closer to the images present in the dataset.

The Generator takes random noise as an input and generates samples as an output. It's goal is to generate such samples that will fool the Discriminator to think that it is seeing real images while actually seeing fakes. We can think of the Generator as a counterfeit. Discriminator takes both real images from the input dataset and fake images from the Generator and outputs a verdict whether a given image is legit or not. We can think of the Discriminator as a policeman trying to catch the bad guys while letting the good guys free.

ACKNOWLEDGEMENT

We wish to express our sincere thanks and deep sense of gratitude to our project guide, **Dr. Menaka R.,** Associate Professor, School of Electronics Engineering, for his consistent encouragement and valuable guidance offered to us in a pleasant manner throughout the course of the project work.

We are extremely grateful to **Dr. Sivasubramanian A.,** Dean of the School of Electronics Engineering, VIT Chennai, for extending the facilities of the School towards our project and for her unstinting support.

We express our thanks to our Programme Chair, **Dr. Vetrivelan P.** for the support throughout the course of this project.

We also take this opportunity to thank all the faculty of the School for their support and their wisdom imparted to us throughout the course.

We thank our parents, family, and friends for bearing with us throughout the course of our project and for the opportunity they provided us in undergoing this course in such a prestigious institution.

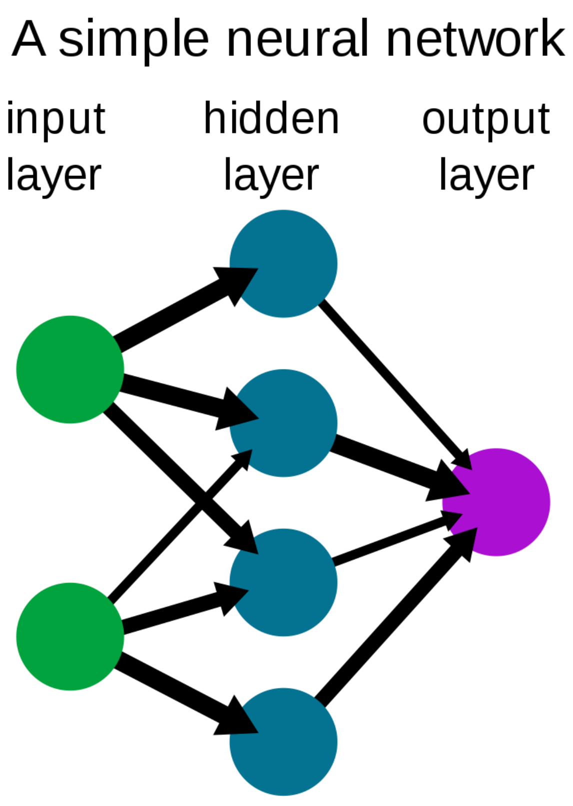
TABLE OF CONTENTS

1. NEURAL NETWORKS

1.1 INTRODUCTION

A **neural network** is a network or circuit of neurons, or in a modern sense, an artificial neural network, composed of artificial neurons or nodes. Thus a neural network is either a biological neural network, made up of real biological neurons, or an artificial neural network, for solving artificial intelligence (AI) problems. The connections of the biological neuron are modeled as weights. A positive weight reflects an excitatory connection, while negative values mean inhibitory connections. All inputs are modified by a weight and summed. This activity is referred as a linear combination. Finally, an activation function controls the amplitude of the output. For example, an acceptable range of output is usually between 0 and 1, or it could be −1 and 1.

These artificial networks may be used for predictive modeling, adaptive control and applications where they can be trained via a dataset. Self-learning resulting from experience can occur within networks, which can derive conclusions from a complex and seemingly unrelated set of information.



A *neural network* (NN), in the case of artificial neurons called *artificial neural network* (ANN) or *simulated neural network* (SNN), is an interconnected group of natural or artificial neurons that uses a mathematical or computational model for information processing based on a connectionistic approach to computation. In most cases an ANN is an adaptive system that changes its structure based on external or internal information that flows through the network.

In more practical terms neural networks are non-linear statistical data modeling or decision making tools. They can be used to model complex relationships between inputs and outputs or to find patterns in data.

An artificial neural network involves a network of simple processing elements (artificial neurons) which can exhibit complex global behavior, determined by the connections between the processing elements and element parameters. Artificial neurons were first proposed in 1943 by Warren McCulloch, a neurophysiologist, and Walter Pitts, a logician, who first collaborated at the University of Chicago.

One classical type of artificial neural network is the recurrent Hopfield network.

The concept of a neural network appears to have first been proposed by Alan Turing in his 1948 paper *Intelligent Machinery* in which called them "B-type unorganised machines".

The utility of artificial neural network models lies in the fact that they can be used to infer a function from observations and also to use it. Unsupervised neural networks can also be used to learn representations of the input that capture the salient characteristics of the input distribution, e.g., see the Boltzmann machine (1983), and more recently, deep learning algorithms, which can implicitly learn the distribution function of the observed data. Learning in neural networks is particularly useful in applications where the complexity of the data or task makes the design of such functions by hand impractical.

Neural networks, in the world of finance, assist in the development of such process as time-series forecasting, algorithmic trading, securities classification, credit risk modeling and constructing proprietary indicators and price derivatives.

A neural network works similarly to the human brain’s neural network. A “neuron” in a neural network is a mathematical function that collects and classifies information according to a specific architecture. The network bears a strong resemblance to statistical methods such as curve fitting and regression analysis.

A neural network contains layers of interconnected nodes. Each node is a perceptron and is similar to a multiple linear regression. The perceptron feeds the signal produced by a multiple linear regression into an activation function that may be nonlinear.

In a multi-layered perceptron (MLP), perceptrons are arranged in interconnected layers. The input layer collects input patterns. The output layer has classifications or output signals to which input patterns may map. For instance, the patterns may comprise a list of quantities for technical indicators about a security; potential outputs could be “buy,” “hold” or “sell.”

Hidden layers fine-tune the input weightings until the neural network’s margin of error is minimal. It is hypothesized that hidden layers extrapolate salient features in the input data that have predictive power regarding the outputs. This describes feature extraction, which accomplishes a utility similar to statistical techniques such as principal component analysis.

1.2 TYPES OF NEURAL NETWORKS

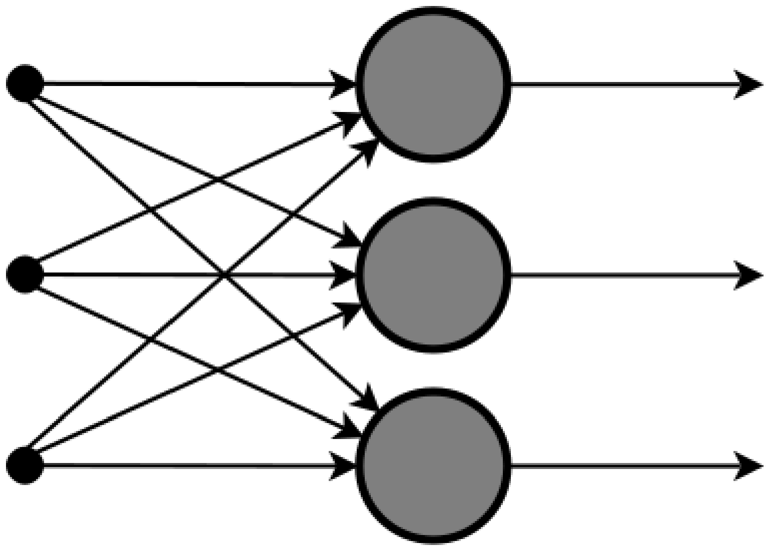
- Feed-Forward Neural Network

This is one of the simplest types of artificial neural networks. In a feedforward neural network, the data passes through the different input nodes till it reaches the output node.

In other words, data moves in only one direction from the first tier onwards until it reaches the output node. This is also known as a front propagated wave which is usually achieved by using a classifying activation function.

Unlike in more complex types of neural networks, there is no backpropagation and data moves in one direction only. A feedforward neural network may have a single layer or it may have hidden layers.

In a feedforward neural network, the sum of the products of the inputs and their weights are calculated. This is then fed to the output. Here is an example of a single layer feedforward neural network.



Feedforward neural networks are used in technologies like face recognition and computer vision. This is because the target classes in these applications are hard to classify.

A simple feedforward neural network is equipped to deal with data which contains a lot of noise. Feedforward neural networks are also relatively simple to maintain.

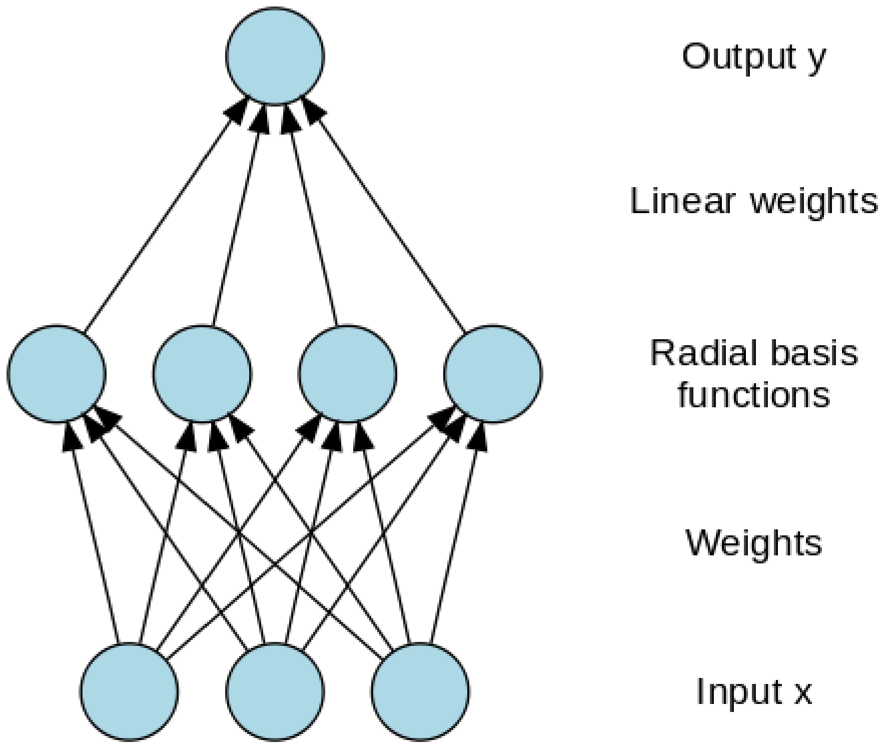
- Radial Basis Function Neural Network

  A radial basis function considers the distance of any point relative to the centre. Such neural networks have two layers. In the inner layer, the features are combined with the radial basis function.

  Then the output of these features is taken into account when calculating the same output in the next time-step. Here is a diagram which represents a radial basis function neural network.

The radial basis function neural network is applied extensively in power restoration systems. In recent decades, power systems have become bigger and more complex.

This increases the risk of a blackout. This neural network is used in the power restoration systems in order to restore power in the shortest possible time.

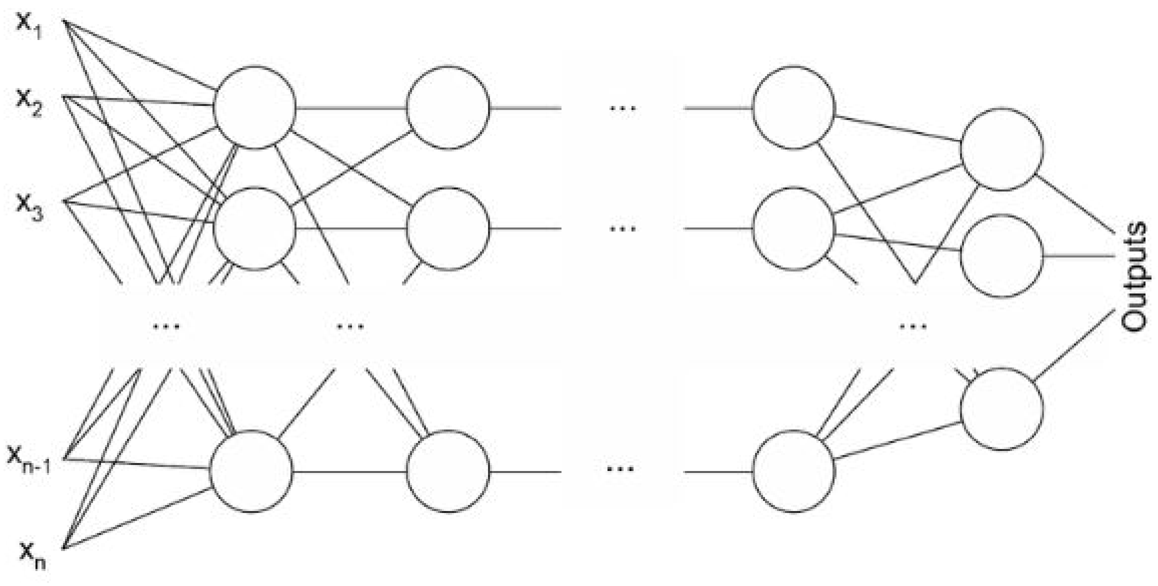


- Multilayer Perceptron Neural Network

A multilayer perceptron has three or more layers. It is used to classify data that cannot be separated linearly. It is a type of artificial neural network that is fully connected. This is because every single node in a layer is connected to each node in the following layer.

A multilayer perceptron uses a nonlinear activation function (mainly hyperbolic tangent or logistic function). Here's what a multilayer perceptron looks like.

This type of neural network is applied extensively in speech recognition and machine translation technologies.



- Convolutional Neural Network

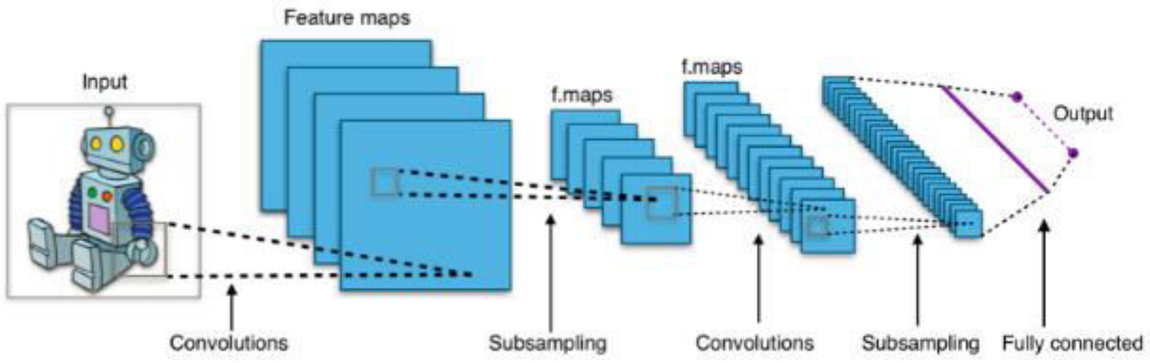
A convolutional neural network(CNN) uses a variation of the multilayer perceptrons. A CNN contains one or more than one convolutional layers. These layers can either be completely interconnected or pooled.

Before passing the result to the next layer, the convolutional layer uses a convolutional operation on the input. Due to this convolutional operation, the network can be much deeper but with much fewer parameters.

Due to this ability, convolutional neural networks show very effective results in image and video recognition, natural language processing, and recommender systems.

Convolutional neural networks also show great results in semantic parsing and paraphrase detection. They are also applied in signal processing and image classification.

CNNs are also being used in image analysis and recognition in agriculture where weather features are extracted from satellites like LSAT to predict the growth and yield of a piece of land. Here's an image of what a Convolutional Neural Network looks like.



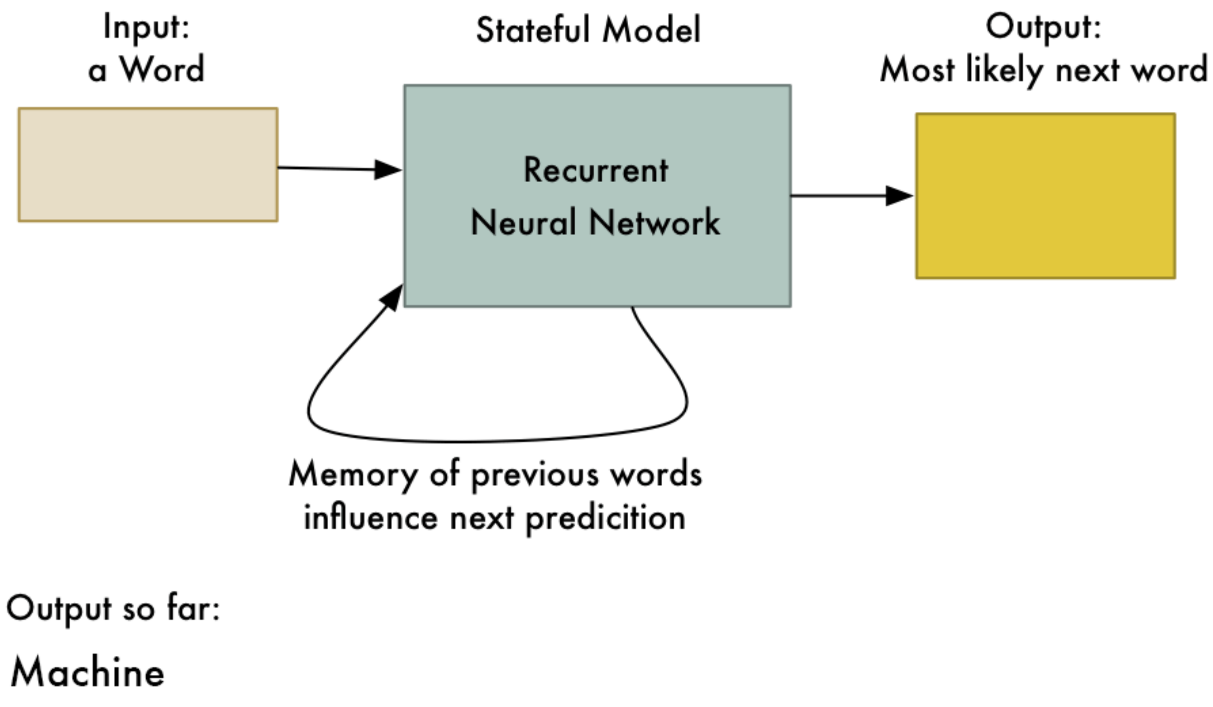
- Recurrent Neural Network (RNN)

A Recurrent Neural Network is a type of artificial neural network in which the output of a particular layer is saved and fed back to the input. This helps predict the outcome of the layer.

The first layer is formed in the same way as it is in the feedforward network. That is, with the product of the sum of the weights and features. However, in subsequent layers, the recurrent neural network process begins.

From each time-step to the next, each node will remember some information that it had in the previous time-step. In other words, each node acts as a memory cell while computing and carrying out operations. The neural network begins with the front propagation as usual but remembers the information it may need to use later.

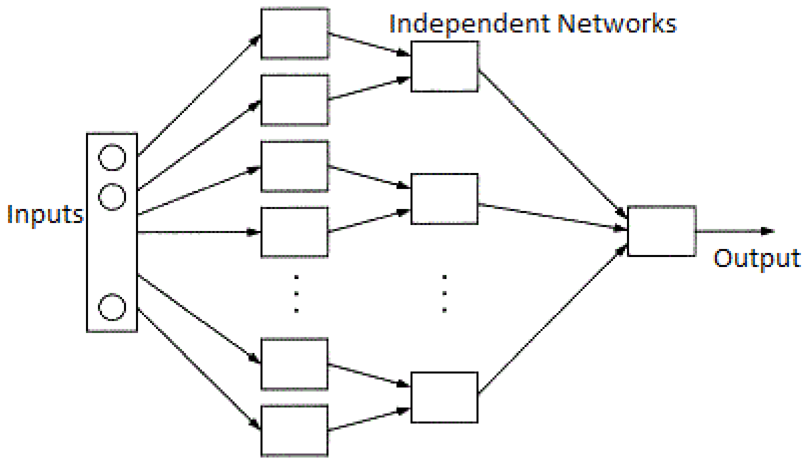
If the prediction is wrong, the system self-learns and works towards making the right prediction during the backpropagation. This type of neural network is very effective in text-to-speech conversion technology. Here’s what a recurrent neural network looks like.



- Modular Neural Network

A modular neural network has a number of different networks that function independently and perform sub-tasks. The different networks do not really interact with or signal each other during the computation process. They work independently towards achieving the output.

As a result, a large and complex computational process can be done significantly faster by breaking it down into independent components. The computation speed increases because the networks are not interacting with or even connected to each other. Here’s a visual representation of a Modular Neural Network.



1.3 APPLIATIONS

Followings are some of the areas, where ANN is being used. It suggests that ANN has an interdisciplinary approach in its development and applications.

Speech Recognition

Speech occupies a prominent role in human-human interaction. Therefore, it is natural for people to expect speech interfaces with computers. In the present era, for communication with machines, humans still need sophisticated languages which are difficult to learn and use. To ease this communication barrier, a simple solution could be, communication in a spoken language that is possible for the machine to understand.

Great progress has been made in this field, however, still such kinds of systems are facing the problem of limited vocabulary or grammar along with the issue of retraining of the system for different speakers in different conditions. ANN is playing a major role in this area. Following ANNs have been used for speech recognition −

- Multilayer networks
- Multilayer networks with recurrent connections
- Kohonen self-organizing feature map

The most useful network for this is Kohonen Self-Organizing feature map, which has its input as short segments of the speech waveform. It will map the same kind of phonemes as the output array, called feature extraction technique. After extracting the features, with the help of some acoustic models as back-end processing, it will recognize the utterance.

Character Recognition

It is an interesting problem which falls under the general area of Pattern Recognition. Many neural networks have been developed for automatic recognition of handwritten characters, either letters or digits. Following are some ANNs which have been used for character recognition −

- Multilayer neural networks such as Backpropagation neural networks.
- Neocognitron

Though back-propagation neural networks have several hidden layers, the pattern of connection from one layer to the next is localized. Similarly, neocognitron also has several hidden layers and its training is done layer by layer for such kind of applications.

Signature Verification Application

Signatures are one of the most useful ways to authorize and authenticate a person in legal transactions. Signature verification technique is a non-vision based technique.

For this application, the first approach is to extract the feature or rather the geometrical feature set representing the signature. With these feature sets, we have to train the neural networks using an efficient neural network algorithm. This trained neural network will classify the signature as being genuine or forged under the verification stage.

Human Face Recognition

It is one of the biometric methods to identify the given face. It is a typical task because of the characterization of “non-face” images. However, if a neural network is well trained, then it can be divided into two classes namely images having faces and images that do not have faces.

First, all the input images must be preprocessed. Then, the dimensionality of that image must be reduced. And, at last it must be classified using neural network training algorithm. Following neural networks are used for training purposes with preprocessed image −

- Fully-connected multilayer feed-forward neural network trained with the help of back-propagation algorithm.
- For dimensionality reduction, Principal Component Analysis

2 GENERATIVE ADVERSARIAL NETWORKS (GANs)

2.1 GANs INTRODUCTION

Generative Adversarial Networks, or GANs for short, are 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 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.

GANs are an exciting and rapidly changing field, delivering on the promise of generative models in their ability to generate realistic examples across a range of problem domains, most notably in image-to-image translation tasks such as translating photos of summer to winter or day to night, and in generating photorealistic photos of objects, scenes, and people that even humans cannot tell are fake.

2.2 HOW DO GANs WORK

One neural network, called the *generator*, generates new data instances, while the other, the *discriminator*, evaluates them for authenticity; i.e. the discriminator decides whether each instance of data that it reviews belongs to the actual training dataset or not.

Let’s say we’re trying to do something more banal than mimic the Mona Lisa. We’re going to generate hand-written numerals like those found in the MNIST dataset, which is taken from the real world. The goal of the discriminator, when shown an instance from the true MNIST dataset, is to recognize those that are authentic.

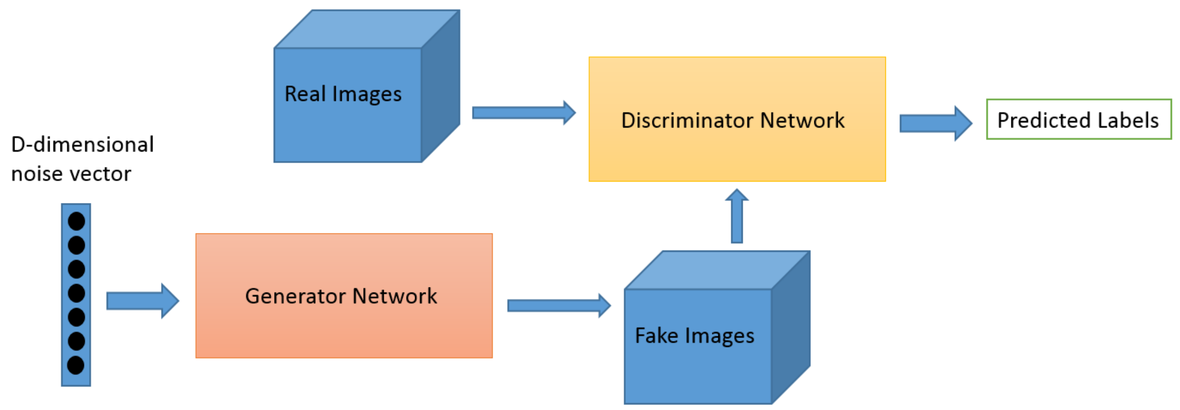
Meanwhile, the generator is creating new, synthetic images that it passes to the discriminator. It does so in the hopes that they, too, will be deemed authentic, even though they are fake. The goal of the generator is to generate passable hand-written digits: to lie without being caught. The goal of the discriminator is to identify images coming from the generator as fake.

Here are the steps a GAN takes:

- The generator takes in random numbers and returns an image.
- This generated image is fed into the discriminator alongside a stream of images taken from the actual, ground-truth dataset.
- The discriminator takes in both real and fake images and returns probabilities, a number between 0 and 1, with 1 representing a prediction of authenticity and 0 representing fake.

So you have a double feedback loop:

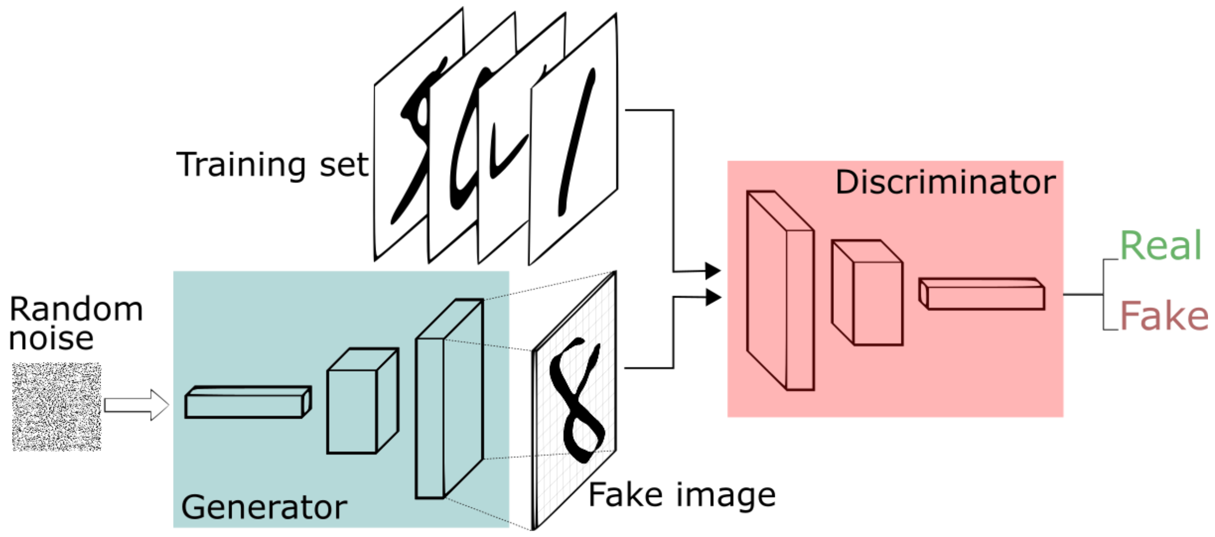
- The discriminator is in a feedback loop with the ground truth of the images, which we know.
- The generator is in a feedback loop with the discriminator.



You can think of a GAN as the opposition of a counterfeiter and a cop in a game of cat and mouse, where the counterfeiter is learning to pass false notes, and the cop is learning to detect them. Both are dynamic; i.e. the cop is in training, too (to extend the analogy, maybe the central bank is flagging bills that slipped through), and each side comes to learn the other’s methods in a constant escalation.

For MNIST, the discriminator network is a standard convolutional network that can categorize the images fed to it, a binomial classifier labeling images as real or fake. The generator is an inverse convolutional network, in a sense: While a standard convolutional classifier takes an image and downsamples it to produce a probability, the generator takes a vector of random noise and upsamples it to an image. The first throws away data through downsampling techniques like maxpooling, and the second generates new data.

Both nets are trying to optimize a different and opposing objective function, or loss function, in a zero-zum game. This is essentially an actor-critic model. As the discriminator changes its behavior, so does the generator, and vice versa. Their losses push against each other.



When you train the discriminator, hold the generator values constant; and when you train the generator, hold the discriminator constant. Each should train against a static adversary. For example, this gives the generator a better read on the gradient it must learn by.

By the same token, pretraining the discriminator against MNIST before you start training the generator will establish a clearer gradient.

Each side of the GAN can overpower the other. If the discriminator is too good, it will return values so close to 0 or 1 that the generator will struggle to read the gradient. If the generator is too good, it will persistently exploit weaknesses in the discriminator that lead to false negatives. This may be mitigated by the nets' respective learning rates. The two neural networks must have a similar "skill level." [1]

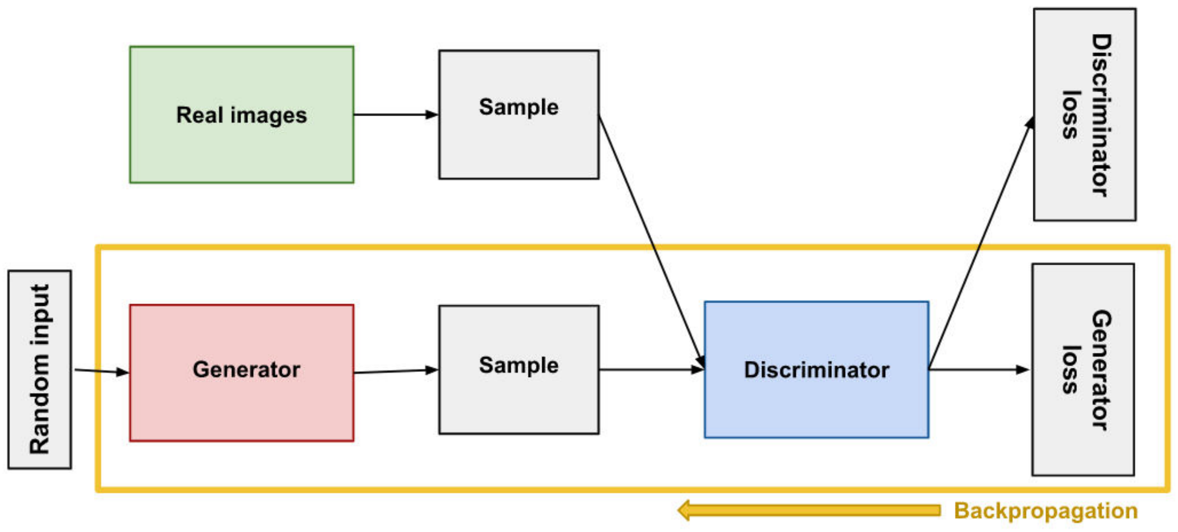
GANs take a long time to train. On a single GPU a GAN might take hours, and on a single CPU more than a day. While difficult to tune and therefore to use, GANs have stimulated a lot of interesting research and writing.

2.3 GENERATOR AND DISCRIMINATOR IN GANs

The generator part of a GAN learns to create fake data by incorporating feedback from the discriminator. It learns to make the discriminator classify its output as real.

Generator training requires tighter integration between the generator and the discriminator than discriminator training requires. The portion of the GAN that trains the generator includes:

- random input
- generator network, which transforms the random input into a data instance
- discriminator network, which classifies the generated data
- discriminator output
- generator loss, which penalizes the generator for failing to fool the discriminator



Neural networks need some form of input. Normally we input data that we want to do something with, like an instance that we want to classify or make a prediction about. But what do we use as input for a network that outputs entirely new data instances?

In its most basic form, a GAN takes random noise as its input. The generator then transforms this noise into a meaningful output. By introducing noise, we can get the GAN to produce a wide variety of data, sampling from different places in the target distribution.

Experiments suggest that the distribution of the noise doesn't matter much, so we can choose something that's easy to sample from, like a uniform distribution. For convenience the space from which the noise is sampled is usually of smaller dimension than the dimensionality of the output space.

Using the Discriminator to Train the Generator

To train a neural net, we alter the net's weights to reduce the error or loss of its output. In our GAN, however, the generator is not directly connected to the loss that we're trying to affect. The generator feeds into the discriminator net, and the *discriminator* produces the output we're trying to affect. The generator loss penalizes the generator for producing a sample that the discriminator network classifies as fake.

This extra chunk of network must be included in backpropagation. Backpropagation adjusts each weight in the right direction by calculating the weight's impact on the output — how the output would change if you changed the weight. But the impact of a generator weight depends on the impact of the discriminator weights it feeds into. So backpropagation starts at the output and flows back through the discriminator into the generator.

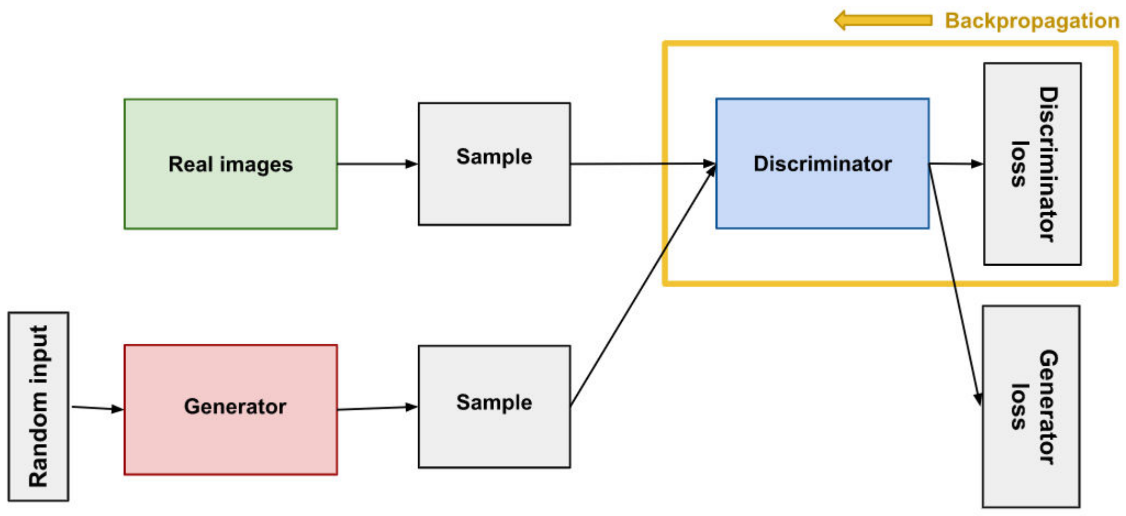
At the same time, we don't want the discriminator to change during generator training. Trying to hit a moving target would make a hard problem even harder for the generator.

So we train the generator with the following procedure:

1. Sample random noise.
2. Produce generator output from sampled random noise.
3. Get discriminator "Real" or "Fake" classification for generator output.
4. Calculate loss from discriminator classification.
5. Backpropagate through both the discriminator and generator to obtain gradients.
6. Use gradients to change only the generator weights.

This is one iteration of generator training. In the next section we'll see how to juggle the training of both the generator and the discriminator.

The discriminator in a GAN is simply a classifier. It tries to distinguish real data from the data created by the generator. It could use any network architecture appropriate to the type of data it's classifying.



The discriminator's training data comes from two sources:

- **Real data** instances, such as real pictures of people. The discriminator uses these instances as positive examples during training.
- **Fake data** instances created by the generator. The discriminator uses these instances as negative examples during training.

In Figure 1, the two "Sample" boxes represent these two data sources feeding into the discriminator. During discriminator training the generator does not train. Its weights remain constant while it produces examples for the discriminator to train on.

Training the Discriminator

The discriminator connects to two loss functions. During discriminator training, the discriminator ignores the generator loss and just uses the discriminator loss. We use the generator loss during generator training, as described in the next section.

During discriminator training:

1. The discriminator classifies both real data and fake data from the generator.
2. The discriminator loss penalizes the discriminator for misclassifying a real instance as fake or a fake instance as real.

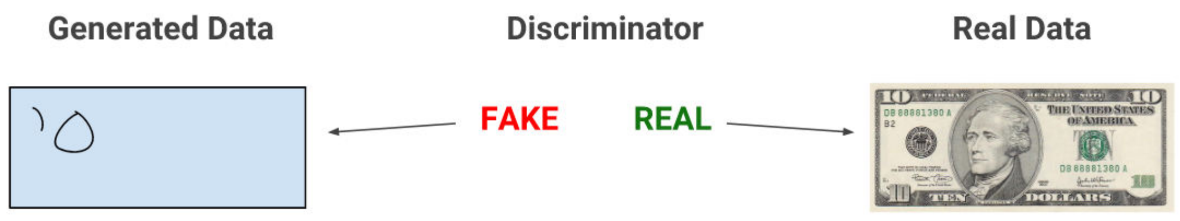
3. The discriminator updates its weights through backpropagation from the discriminator loss through the discriminator network.

GANs try to replicate a probability distribution. They should therefore use loss functions that reflect the distance between the distribution of the data generated by the GAN and the distribution of the real data.

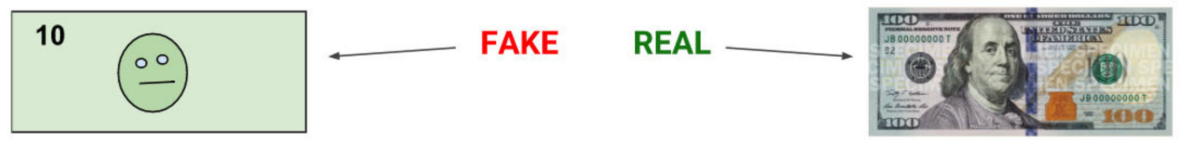
How do you capture the difference between two distributions in GAN loss functions? This question is an area of active research, and many approaches have been proposed. We'll address two common GAN loss functions here, both of which are implemented in TF-GAN:

- **minimax loss**: The loss function used in the paper that introduced GANs.
- **Wasserstein loss**: The default loss function for TF-GAN Estimators. First described in a 2017 paper.

When training begins, the generator produces obviously fake data, and the discriminator quickly learns to tell that it's fake:

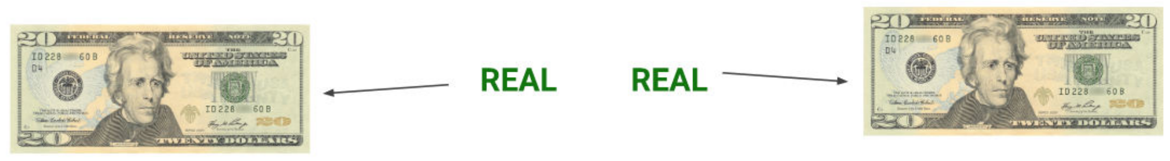


As training progresses, the generator gets closer to producing output that can fool the discriminator:



Finally, if generator training goes well, the discriminator gets worse at telling the difference between real and fake. It starts to classify fake data as real, and its accuracy decreases.

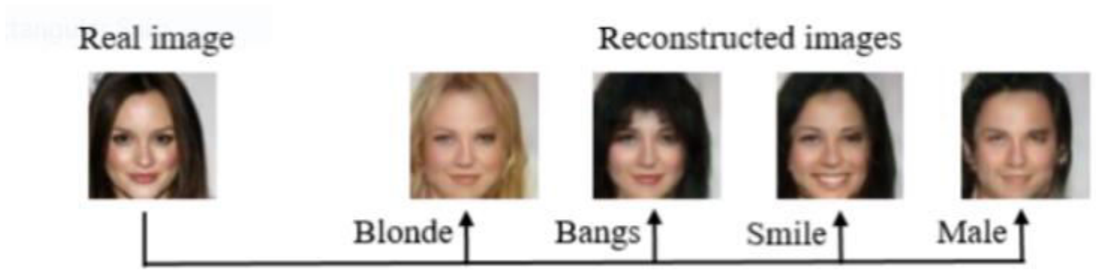
Finally, if generator training goes well, the discriminator gets worse at telling the difference between real and fake. It starts to classify fake data as real, and its accuracy decreases.



2.4 APPLICAIONS OF GANs

GANs for Image Editing

Most image editing software these days don’t give us much flexibility to make creative changes in pictures. For example, let’s say you want to change the appearance of a 90-year-old person by changing his/her hairstyle. This can’t be done by the current image editing tools out there. But guess what? Using GANs, we can reconstruct images and attempt to change the appearance drastically.



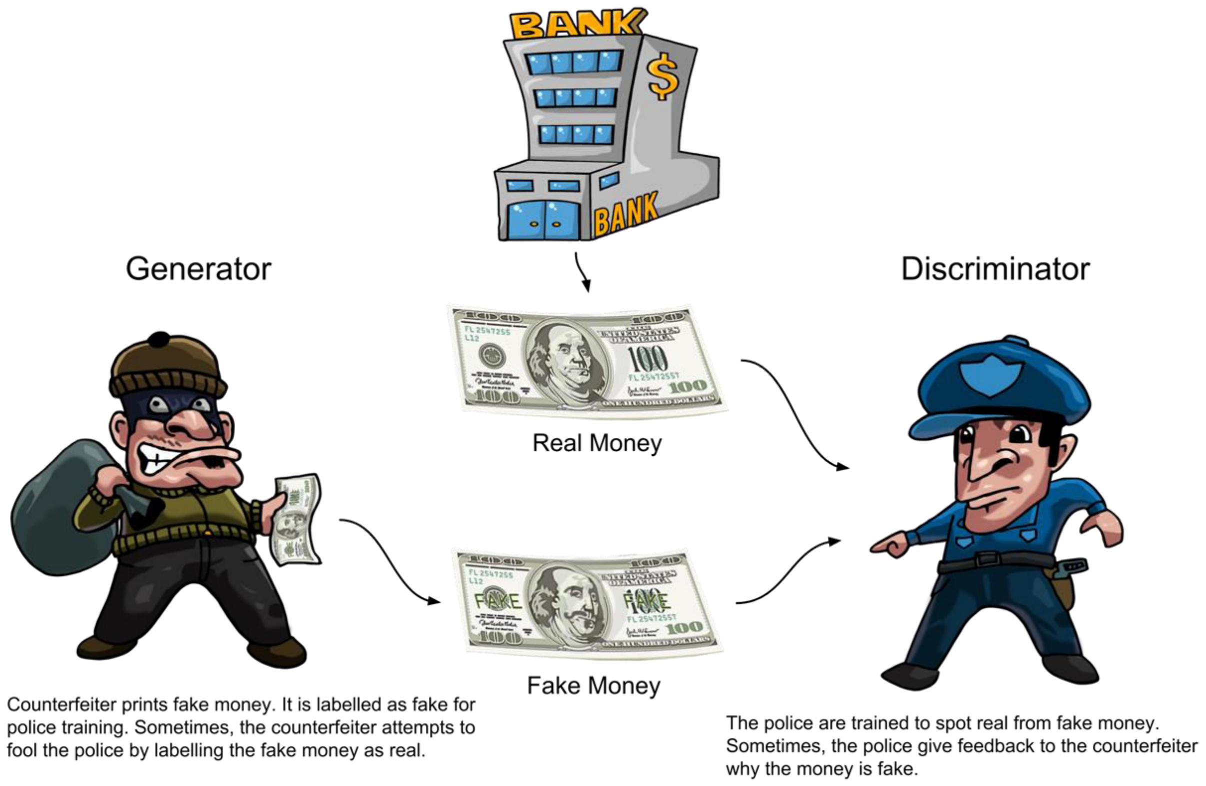
Using GANs for Security

The rise of artificial intelligence has been wonderful for most industries. But there’s a real concern that has shadowed the entire AI revolution – cyber threats. Even deep neural networks are susceptible to being hacked.

A constant concern of industrial applications is that they should be robust to cyber attacks. There’s a lot of confidential information on the line! GANs are proving to be of immense help here, directly addressing the concern of “adversarial attacks”.

These adversarial attacks use a variety of techniques to fool deep learning architectures. GANs are used to make existing deep learning models more robust to these techniques. How? By creating more such fake examples and training the model to identify them. Pretty clever stuff.

A technique called SSGAN is used to do steganalysis of images and detect harmful encodings which shouldn't have been there.


BANK
Generator
Discriminator
Real Money
Fake Money
FAKE
Counterfeiter prints fake money. It is labelled as fake for police training. Sometimes, the counterfeiter attempts to fool the police by labelling the fake money as real.
The police are trained to spot real from fake money. Sometimes, the police give feedback to the counterfeiter why the money is fake.

<u>Generating Data with GANs</u>

Who among us wouldn't love to collect more data for building our deep learning model? The availability of data in certain domains is a necessity, especially in domains where training data is needed to model supervideepeeop learning algorithms. The healthcare industry comes to mind here.

GANs shine again as they can be used to generate synthetic data for supervision. That's right! You know where to go next time you need more data.

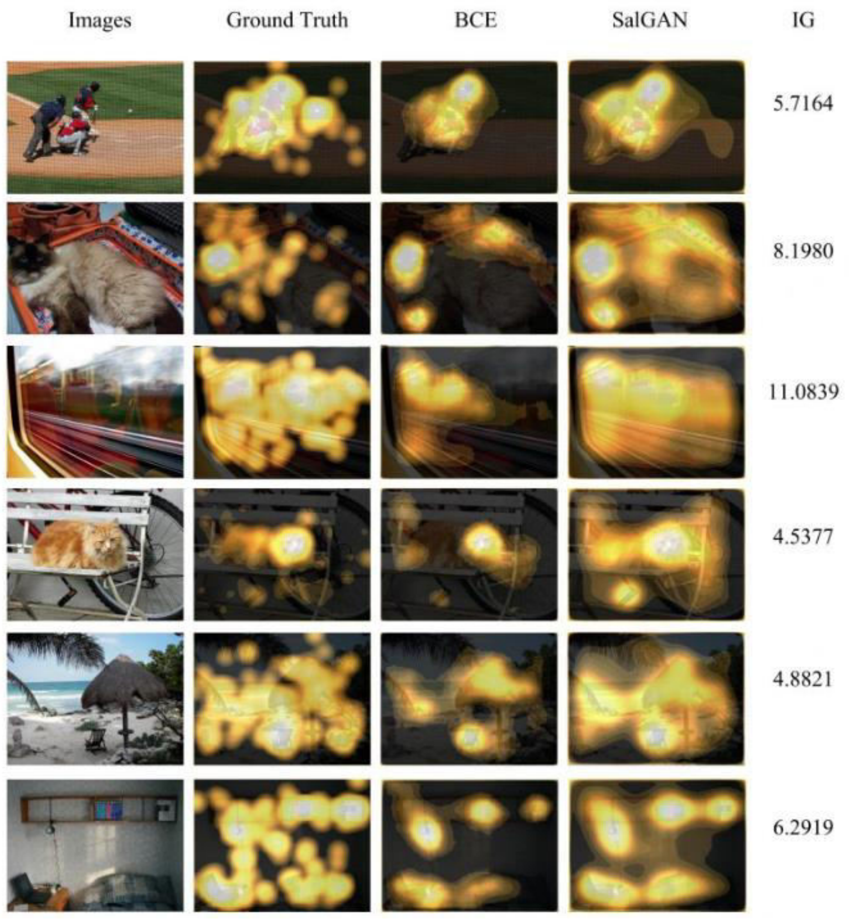
For instance, GANs can be used to explore the creation of synthetic data with the help of GANs for training deep learning algorithms by creating realistic eye images.

GANs for Attention Prediction

When we see an image, we tend to focus on a particular part (rather than the entire image as a whole). This is called attention and is an important human trait. Knowing where a person would look beforehand would certainly be a useful feature for businesses, as they can optimize and position their products better.

For example, game designers can focus on a particular portion of the game to enhance the features and make it more engrossing.

This enthralling idea is explored in GANs, where the authors try to identify the most appealing parts of given images using GANs.



GANs for 3D Object Generation

It won't surprise you to know GANs are quite popular in the gaming industry.

Game designers work countless hours recreating 3D avatars and backgrounds to give them a realistic feel. And let me assure you, it certainly takes a lot of effort to create 3D models by imagination. Does this seem unrealistic? Then I suggest you watch this video. You might believe the incredible power of GANs, wherein they can be used to automate the entire process!

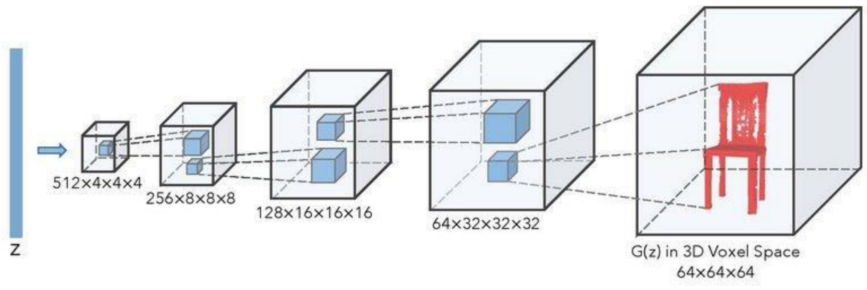


Figure 1: The generator of 3D Generative Adversarial Network (3D-GAN)

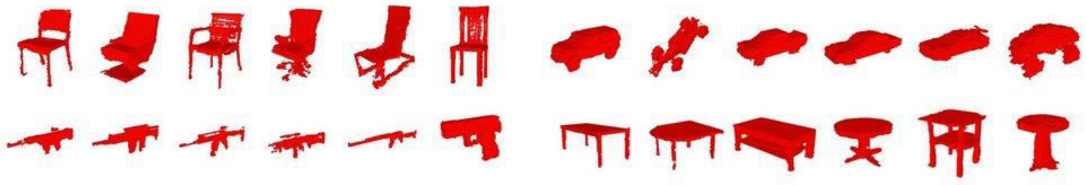


Figure 2: Shapes synthesized by 3D-GAN

3. GENERATIVE ADVERSARIAL NETWORKS IMPLEMENTATION

3.1 CODE

```
# Importing the libraries
from __future__ import print_function
import torch
import torch.nn as nn
import torch.nn.parallel
import torch.optim as optim
import torch.utils.data
import torchvision.datasets as dset
import torchvision.transforms as transforms
import torchvision.utils as vutils
from torch.autograd import Variable

# Setting some hyperparameters
batchSize = 64 # We set the size of the batch.
imageSize = 64 # We set the size of the generated images (64x64).

# Creating the transformations
transform = transforms.Compose([transforms.Resize(imageSize),
transforms.ToTensor(), transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),]) # We
create a list of transformations (scaling, tensor conversion, normalization) to
apply to the input images.

# Loading the dataset
dataset = dset.CIFAR10(root = './data', download = True, transform = transform)
# We download the training set in the ./data folder and we apply the previous
transformations on each image.
dataloader = torch.utils.data.DataLoader(dataset, batch_size = batchSize, shuffle
= True, num_workers = 2) # We use dataLoader to get the images of the
training set batch by batch.

# Defining the weights_init function that takes as input a neural network m
and that will initialize all its weights.
def weights_init(m):
    classname = m.__class__.__name__
    if classname.find('Conv') != -1:
        m.weight.data.normal_(0.0, 0.02)
    elif classname.find('BatchNorm') != -1:
        m.weight.data.normal_(1.0, 0.02)
        m.bias.data.fill_(0)
```

# Defining the generator

```
class G(nn.Module):

    def __init__(self):
        super(G, self).__init__()
        self.main = nn.Sequential(
            nn.ConvTranspose2d(100, 512, 4, 1, 0, bias = False),
            nn.BatchNorm2d(512),
            nn.ReLU(True),
            nn.ConvTranspose2d(512, 256, 4, 2, 1, bias = False),
            nn.BatchNorm2d(256),
            nn.ReLU(True),
            nn.ConvTranspose2d(256, 128, 4, 2, 1, bias = False),
            nn.BatchNorm2d(128),
            nn.ReLU(True),
            nn.ConvTranspose2d(128, 64, 4, 2, 1, bias = False),
            nn.BatchNorm2d(64),
            nn.ReLU(True),
            nn.ConvTranspose2d(64, 3, 4, 2, 1, bias = False),
            nn.Tanh()
        )

    def forward(self, input):
        output = self.main(input)
        return output
```

# Creating the generator

```
netG = G()
netG.apply(weights_init)
```

# Defining the discriminator

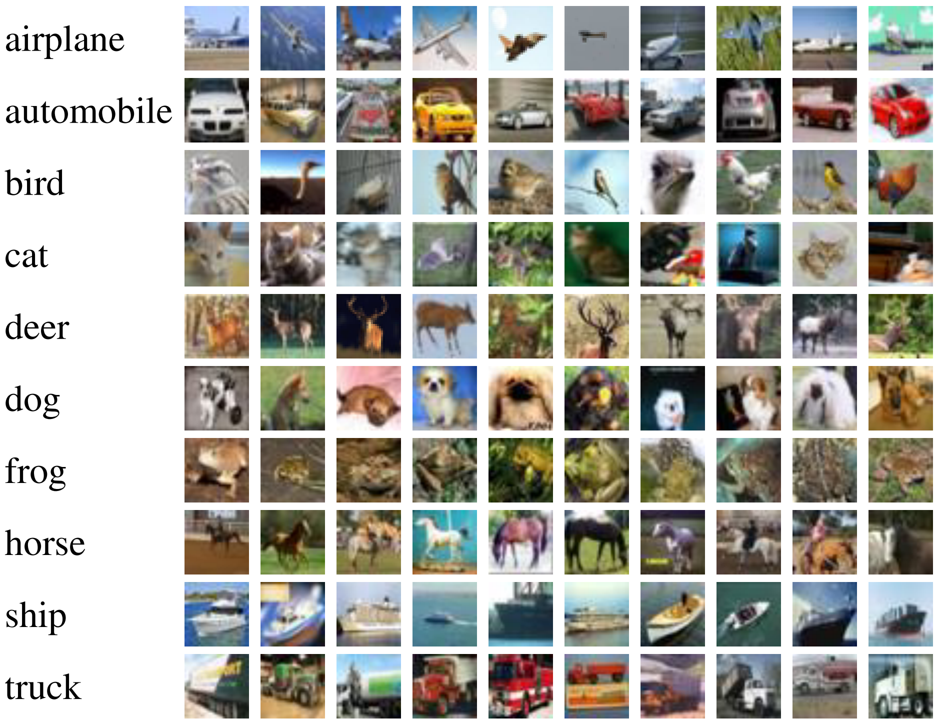
```
class D(nn.Module):

    def __init__(self):
        super(D, self).__init__()
        self.main = nn.Sequential(
            nn.Conv2d(3, 64, 4, 2, 1, bias = False),
            nn.LeakyReLU(0.2, inplace = True),
            nn.Conv2d(64, 128, 4, 2, 1, bias = False),
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.2, inplace = True),
            nn.Conv2d(128, 256, 4, 2, 1, bias = False),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.2, inplace = True),
            nn.Conv2d(256, 512, 4, 2, 1, bias = False),
            nn.BatchNorm2d(512),
            nn.LeakyReLU(0.2, inplace = True),
            nn.Conv2d(512, 1, 4, 1, 0, bias = False),
            nn.Sigmoid()
        )

    def forward(self, input):
        output = self.main(input)
        return output.view(-1)

# Creating the discriminator
netD = D()
netD.apply(weights_init)

# Training the DCGANs

criterion = nn.BCELoss()
optimizerD = optim.Adam(netD.parameters(), lr = 0.0002, betas = (0.5, 0.999))
optimizerG = optim.Adam(netG.parameters(), lr = 0.0002, betas = (0.5, 0.999))

for epoch in range(25):

    for i, data in enumerate(dataloader, 0):

        # 1st Step: Updating the weights of the neural network of the
discriminator

        netD.zero_grad()

        # Training the discriminator with a real image of the dataset
        real, _ = data
        input = Variable(real)
        target = Variable(torch.ones(input.size()[0]))
        output = netD(input)
        errD_real = criterion(output, target)
        # Training the discriminator with a fake image generated by the
generator
        noise = Variable(torch.randn(input.size()[0], 100, 1, 1))
        fake = netG(noise)
        target = Variable(torch.zeros(input.size()[0]))
        output = netD(fake.detach())
        errD_fake = criterion(output, target)

        # Backpropagating the total error
        errD = errD_real + errD_fake
        errD.backward()
        optimizerD.step()

        # 2nd Step: Updating the weights of the neural network of the
generator

        netG.zero_grad()
        target = Variable(torch.ones(input.size()[0]))
        output = netD(fake)
        errG = criterion(output, target)
        errG.backward()
        optimizerG.step()

        # 3rd Step: Printing the losses and saving the real images and the
generated images of the minibatch every 100 steps

        print('[%d/%d][%d/%d] Loss_D: %.4f Loss_G: %.4f' % (epoch, 25, i,
len(dataloader), errD.data, errG.data)) #.data[0] in both
        if i % 100 == 0:
            vutils.save_image(real, '%s/real_samples.png' % "./results", normalize =
True)
            fake = netG(noise)
            vutils.save_image(fake.data, '%s/fake_samples_epoch_%03d.png' %
("./results", epoch), normalize = True)
```

- **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.

Here are the classes in the dataset, as well as 10 random images from each:

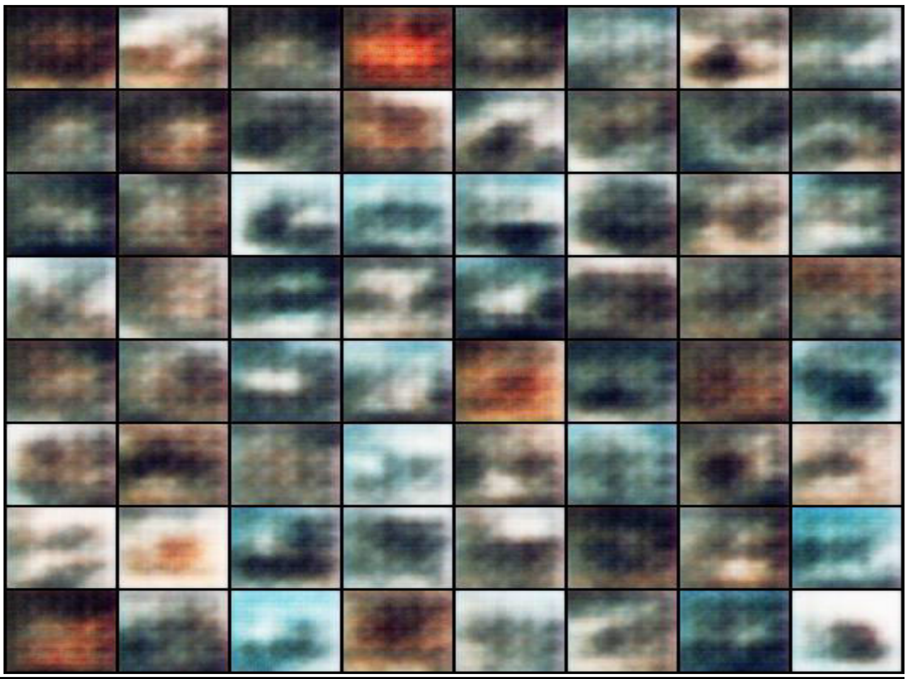


The classes are completely mutually exclusive. There is no overlap between automobiles and trucks. "Automobile" includes sedans, SUVs, things of that sort. "Truck" includes only big trucks. Neither includes pickup trucks.

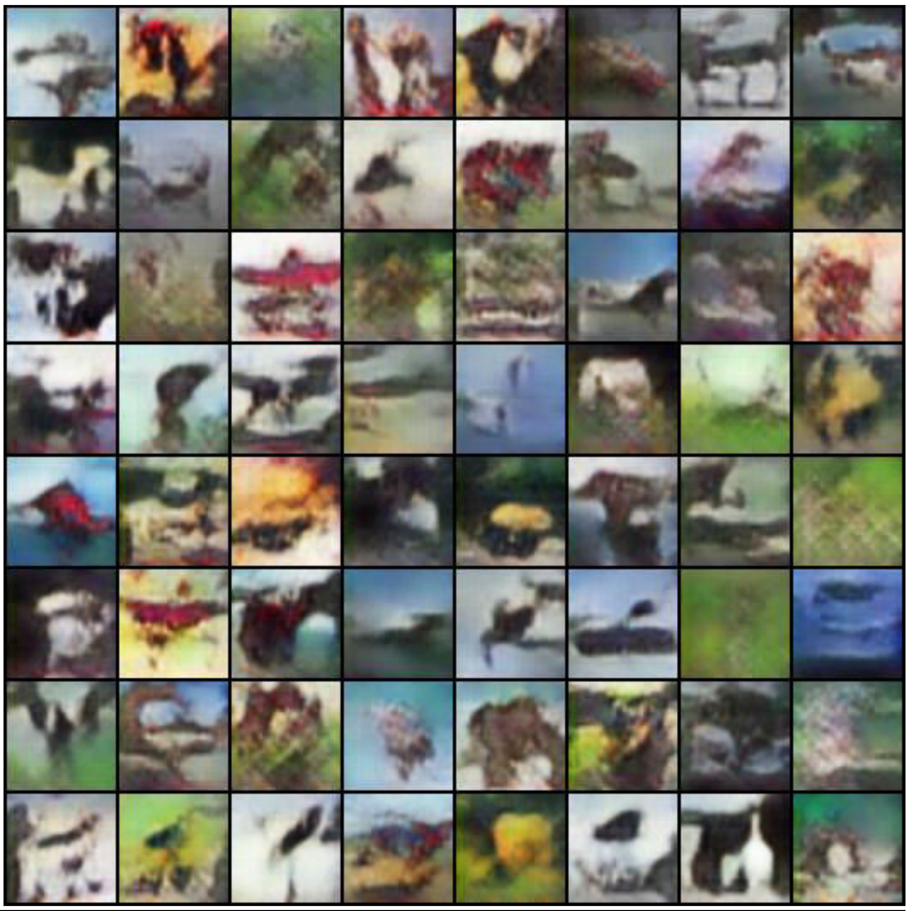
- **RESULTS**

TRAINING DATASET SAMPLE

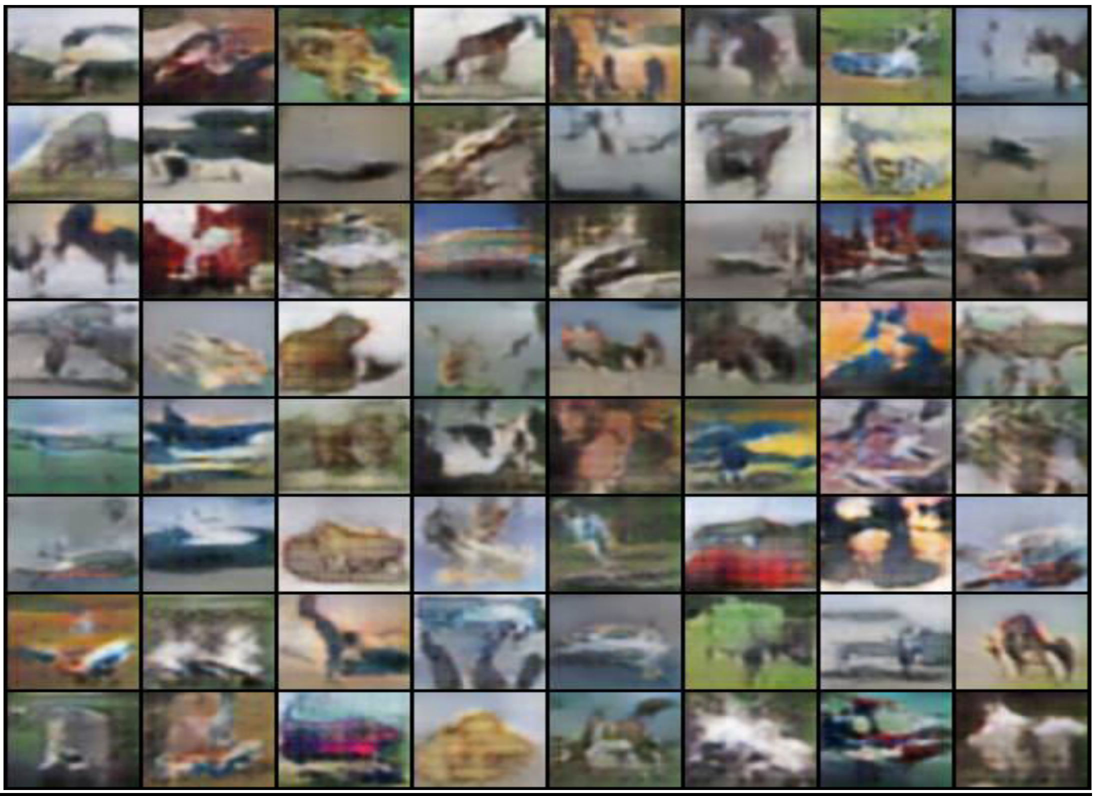
RANDOM NOISE GENERATED



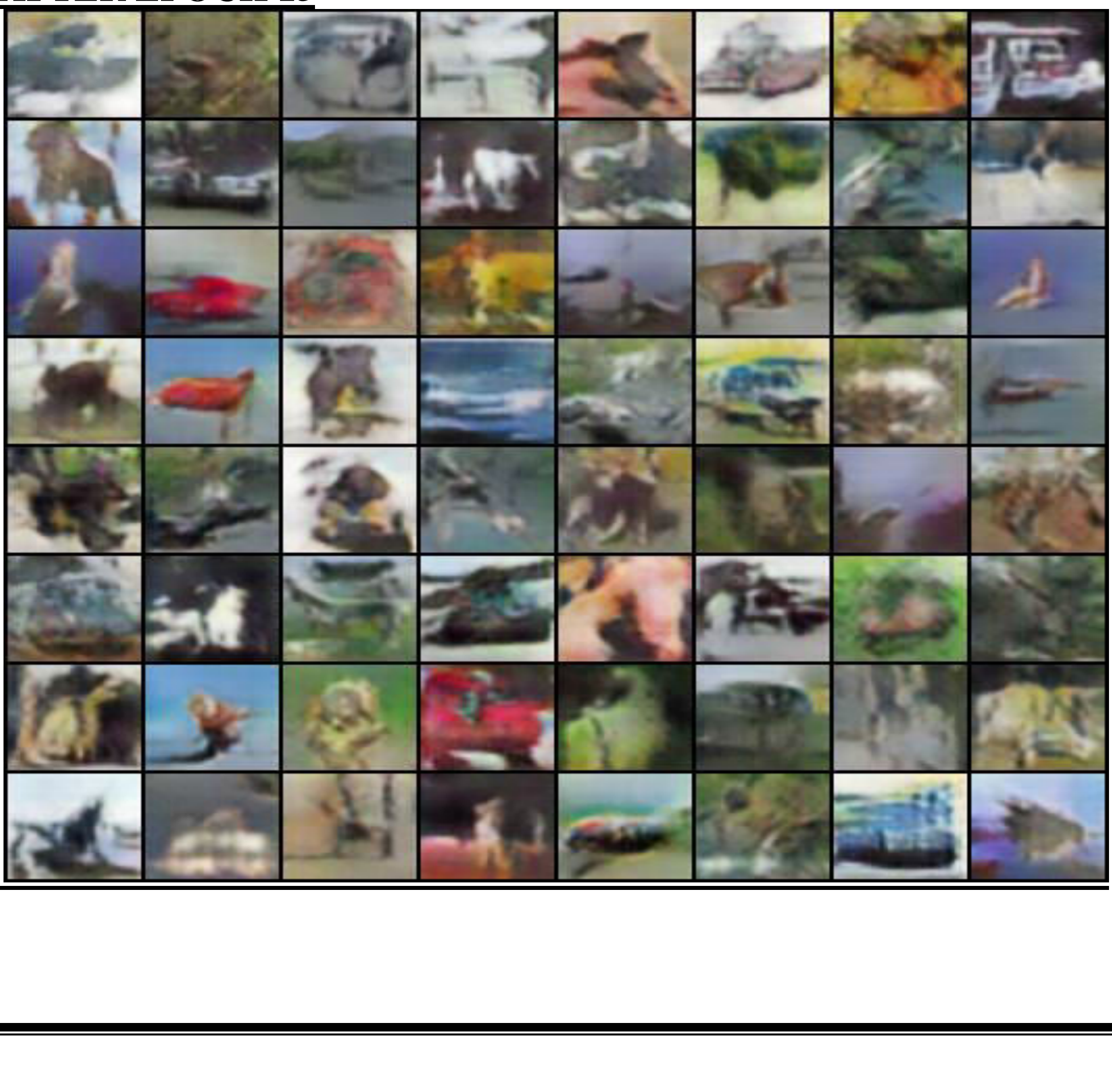
AFTER EPOCH 5



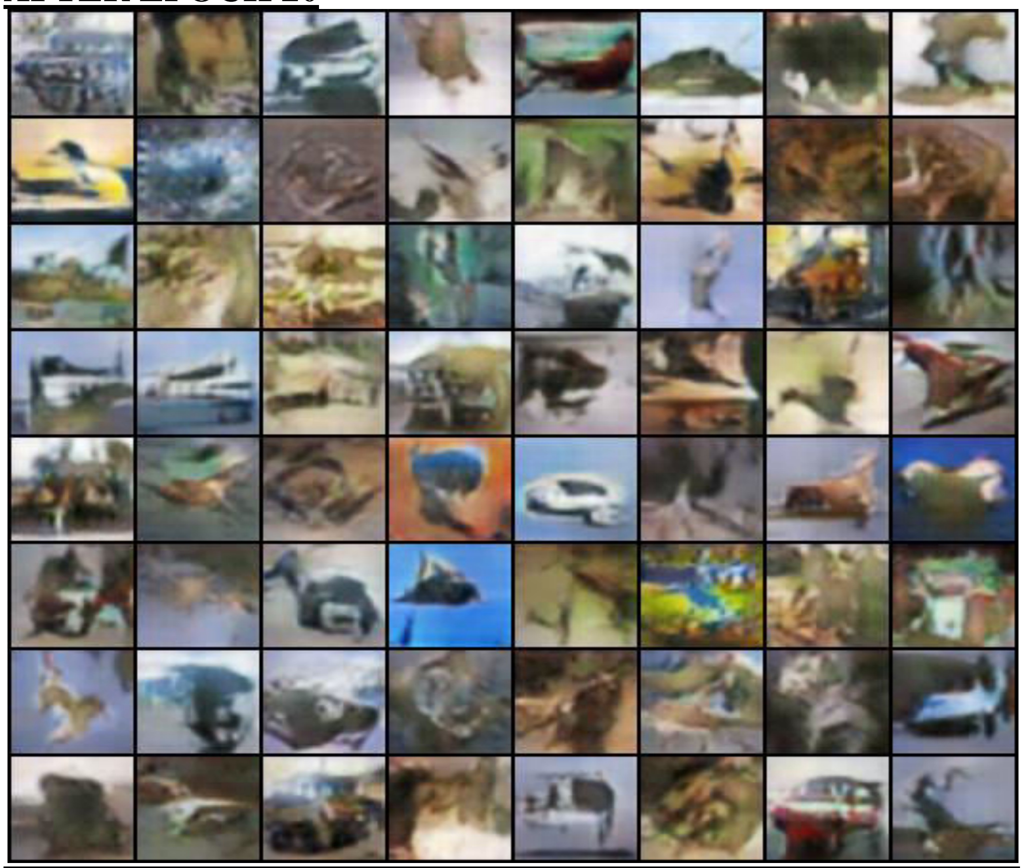
AFTER EPOCH 10



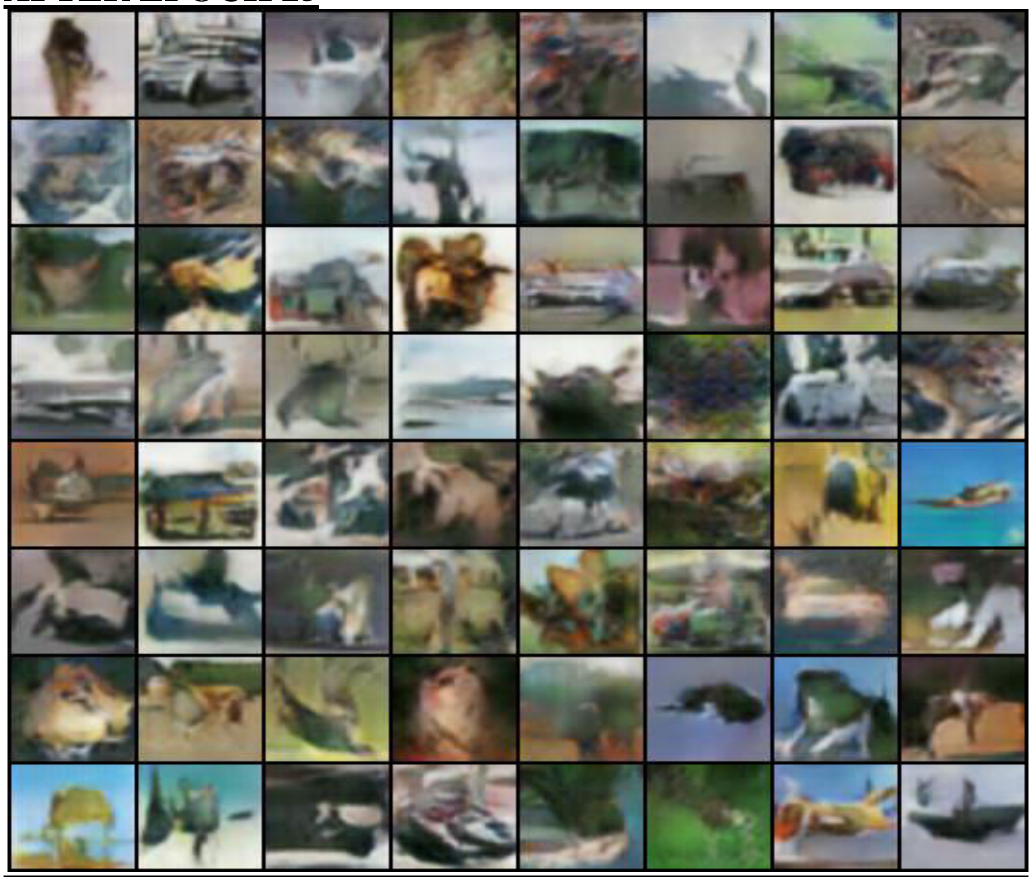
AFTER EPOCH 15



AFTER EPOCH 20



AFTER EPOCH 25



3.2 MACHINE LEARNING FRAMEWORK (PYTORCH)

PyTorch is an open source machine learning library based on the Torch library,used for applications such as computer vision

and natural language processing. It is primarily developed by Facebook's artificial intelligence research group. It is free and open-source software released under the Modified BSD license. Although the Python interface is more polished and the primary focus of development, PyTorch also has a C++ frontend. Furthermore, Uber Pyro probabilistic programming language software uses PyTorch as a backend.

PyTorch provides two high-level features:

- Tensor computing (like NumPy) with strong acceleration via graphics processing units (GPU)
- Deep neural networks built on a tape-based autodiff system

PyTorch is a Python-based scientific computing package that uses the power of graphics processing units. It is also one of the preferred deep learning research platforms built to provide maximum flexibility and speed. It is known for providing two of the most high-level features; namely, tensor computations with strong GPU acceleration support and building deep neural networks on a tape-based autograd systems.

There are many existing Python libraries which have the potential to change how deep learning and artificial intelligence are performed, and this is one such library. One of the key reasons behind PyTorch’s success is it is completely Pythonic and one can build neural network models effortlessly. It is still a young player when compared to its other competitors, however, it is gaining momentum fast.

Why use PyTorch in research?

Anyone who is working in the field of deep learning and artificial intelligence has likely worked with TensorFlow before, Google’s most popular open source library. However, the latest deep learning framework – PyTorch solves major problems in terms of research work. Arguably PyTorch is TensorFlow’s biggest competitor to date, and it is currently a much favored deep learning and artificial intelligence library in the research community.

Dynamic Computational graphs

It avoids static graphs that are used in frameworks such as TensorFlow, thus allowing the developers and researchers to change how the network behaves on the fly. The early adopters are preferring PyTorch because it is more intuitive to learn when compared to TensorFlow.

Different back-end support

PyTorch uses different backends for CPU, GPU and for various functional features rather than using a single back-end. It uses tensor backend TH for CPU and THC for GPU. While neural network backends such as THNN and THCUNN for CPU and GPU respectively. Using separate backends makes it very easy to deploy PyTorch on constrained systems.

Imperative style

PyTorch library is specially designed to be intuitive and easy to use. When you execute a line of code, it gets executed thus allowing you to perform real-time tracking of how your neural network models are built. Because of its excellent imperative architecture and fast and lean approach it has increased overall PyTorch adoption in the community.

Highly extensible

PyTorch is deeply integrated with the C++ code, and it shares some C++ backend with the deep learning framework, Torch. Thus allowing users to program in C/C++ by using an extension API based on cFFI for Python and compiled for CPU for GPU operation. This feature has extended the PyTorch usage for new and experimental use cases thus making them a preferable choice for research use.

Python-Approach

PyTorch is a native Python package by design. Its functionalities are built as Python classes, hence all its code can seamlessly integrate with Python packages and modules. Similar to NumPy, this Python-based library enables GPU-accelerated tensor computations plus provides rich options of APIs for neural network applications. PyTorch provides a complete end-to-end research framework which comes with the most common building blocks for carrying out everyday deep learning research. It allows chaining of high-level neural network modules because it supports Keras-like API in its torch.nn package.

4. CONCLUSION AND FUTURE WORK

4.1 CONCLUSION

- Generative Adversarial Network (GANs) was implemented in Spyder environment using Python 3.5.
- PyTorch, Machine Learning library was used to export function in-order to train GANs
- Generator and Discriminator were trained separately for Image Generation using Generative Adversarial Network (GANs).

4.2 FUTURE WORK

GANs have a number of common failure modes. All of these common problems are areas of active research. While none of these problems have been completely solved, we'll mention some things that people have tried.

Vanishing Gradients

Research has suggested that if your discriminator is too good, then generator training can fail due to vanishing gradients. In effect, an optimal discriminator doesn't provide enough information for the generator to make progress.

Attempts to Remedy

- **Wasserstein loss**: The Wasserstein loss is designed to prevent vanishing gradients even when you train the discriminator to optimality.
- **Modified minimax loss**: The original GAN paper proposed a modification to minimax loss to deal with vanishing gradients.

Mode Collapse

Usually you want your GAN to produce a wide variety of outputs. You want, for example, a different face for every random input to your face generator.

However, if a generator produces an especially plausible output, the generator may learn to produce *only* that output. In fact, the generator is always trying to find the one output that seems most plausible to the discriminator.

If the generator starts producing the same output (or a small set of outputs) over and over again, the discriminator's best strategy is to learn to always reject that output. But if the next generation of discriminator gets stuck in a local minimum and doesn't find the best strategy, then it's too easy for the next generator iteration to find the most plausible output for the current discriminator.

Each iteration of generator over-optimizes for a particular discriminator, and the discriminator never manages to learn its way out of the trap. As a result the generators rotate through a small set of output types. This form of GAN failure is called **mode collapse**.

Attempts to Remedy

The following approaches try to force the generator to broaden its scope by preventing it from optimizing for a single fixed discriminator:

- **Wasserstein loss**: The Wasserstein loss alleviates mode collapse by letting you train the discriminator to optimality without worrying about vanishing gradients. If the discriminator doesn't get stuck in local minima, it learns to reject the outputs that the generator stabilizes on. So the generator has to try something new.
- **Unrolled GANs**: Unrolled GANs use a generator loss function that incorporates not only the current discriminator's classifications, but also the outputs of future discriminator versions. So the generator can't over-optimize for a single discriminator.

Failure to Converge

GANs frequently fail to converge, as discussed in the module on training.

Attempts to Remedy

Researchers have tried to use various forms of regularization to improve GAN convergence, including:

- **Adding noise to discriminator inputs**: See, for example, Toward Principled Methods for Training Generative Adversarial Networks.
- **Penalizing discriminator weights**: See, for example, Stabilizing Training of Generative Adversarial Networks through Regularization.

References

- https://arxiv.org/abs/1406.2661
- https://skymind.ai/wiki/generative-adversarial-network-gan
- https://towardsdatascience.com/understanding-generative-adversarial-networks-gans-cd6e4651a29
- https://developers.google.com/machine-learning/gan
- https://papers.nips.cc/paper/5423-generative-adversarial-nets.pdf
- https://developers.google.com/machine-learning/gan/programming-exercise