Generative Adversarial networks and Variational Auto-encoder implementation on MNIST and ANIME Cartoon Dataset

Gayathri Madala
UFID: 80357003

Abstract

Generative adversarial network and variational autoencoders models are widely used to approximate probability distributions of datasets and generate new images based on the trained data such that the generated images looks very realistic. [3] Both VAE and GAN use parametrized distributions for underlying data distribution approximation but their behaviours are mostly different. Considering the wide range of applications of Generative models, GAN and VAE were implemented on two different datasets. In this report, the implementation of GAN and VAE on MNIST and ANIME Cartoon datasets is shown along with loss functions, optimizers used, probabilities of models are presented and test images generated are shown. Different types of GAN were discussed and reason to select DCGAN over others was mentioned. A comparison of GAN and VAE usage on two different datasets is analysed and results are presented. In this report a comparison of a VAE and DCGAN is conducted with respect to their capability in synthetic image generation. The models were trained on two different datasets, the MNIST digits dataset and ANIME Cartoon dataset and their performance was measured using different metrics. The results were in line with previous work presented in other papers regarding both the models[3][5]. The generated images for both models are presented in this report.

1. Introduction

Machine learning has been constantly evolving over years and drawing more importance now-a-days. Few major reasons being the vast amounts of data available to the organizations, the growing demand for the necessity of automation and efficiency, and the longing to extract distributions and details from complex data. Machine Learning can help the businesses to build predictive models for task automation reducing the manual effort needed and human errors, for data forecast and for decision making. In recent years, machine learning algorithms led to a significant progress in varied areas and is also surpassing human performance in tasks such as pattern recognition, quantum chemistry, and games like poker and Go[3].

Although these algorithms excel in the tasks for which they are designed, they still lack the creative capacity to comprehend the data and produce the solutions for various problems. To overcome this challenge, the development of generative models is being pursued such that they can understand the underlying distribution of data and in turn produce new samples which looks like the real ones, that were not given during the original training. These have shown great promising results in the fields like image and speech recognition, natural language processing.[2]

Particularly, Generative models are very powerful machine learning subset that is gaining popularity in the recent times. These models are capable of new data samples generation which will be similar to the trained data which was used during training phase, allowing for the synthetic data creation. These models will be very useful in situations where there is only limited data available, expensive to get more data, or when data privacy issues exist.

Generative Adversarial Network (GAN) and Variational Auto-encoder (VAE) are such well-known generative models becoming increasingly popular in the machine learning field. Both GANs and VAEs have a huge range of applications including image and text generation, drug discovery, fraud detection etc. They are very powerful tools for synthetic data creation and to explore new underlying data distributions.

1.1 Generative Adversarial Network (GAN)

Generative Adversarial Network is one of the deep learning model which has two neural networks called as generator and discriminator. The generative model will receive a dataset as input and it processes the dataset through multiple layers of computation involving techniques like optimization and regularization. After this process, this generative model becomes capable of generating new data samples based on training data, which looks realistic and the discriminator will learn to distinguish between the fake or generated samples and the real data which acts as a factor for the amount of training that the model needs to undergo before it generates very realistic data[3].

These two networks will be trained together where the generator will keep generating samples and give them to the discriminator, a binary classifier. The discriminator will try to estimate the probability of sample being from training data and accurately identify the real samples and the model generated ones. A good model will maximize the discriminator's estimate probability. Both Generator and Discriminator are will their performance in every iteration by one proving the other wrong. This back-and-forth training process results in a generator that can create highly realistic synthetic data by taking continuous feedback from discriminator. These two neural networks compete with each other to create an adversarial mechanism which results in great quality data samples.

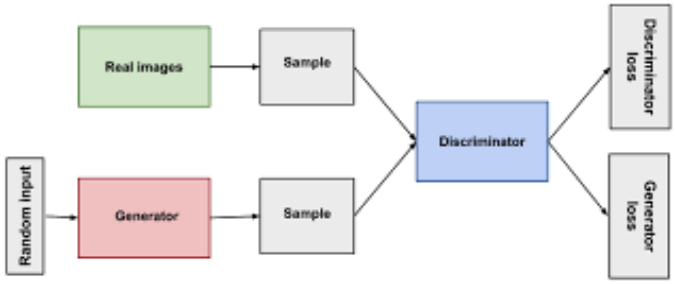


Figure Source:
GAN description by google

1.1.1 Types of Generative Adversarial Network:

There are a different type of GANs that can be implemented as below:

Vanilla GAN:

It is the simplest GAN type which has a generator and a discriminator. The generator generates fake data samples and the discriminator identifies the real and fake samples generated

Conditional GAN:

It is a type of GAN used when generator needs to generate samples conditioned on some specific input data. For an instance, if an image of a cat is given, then the generator can generate distinct images of cats.

Deep Convolutional GAN (DCGAN):

It is a GAN which uses convolutional neural networks for both the generator and discriminator.

Wasserstein GAN (WGAN):

It is a type of GAN that uses the Wasserstein distance to measure distance between the real and fake distributions.

Cycle GAN:

It is a type of GAN that involves four networks like two generators and two discriminators and is used for converting images from one type to another type

Progressive GAN:

It is a type of GAN which generates images whose quality increases progressively as it starts with low-resolution images and gradually increase the resolution.

The above mentioned are some types of GANs used generally but which type of GAN to select totally depends on the type of task one wants to perform, dataset characteristics and the type of data that needs to be generated. It is very important to try implementation with different GAN types to determine which type works better to perform a task on a dataset.

1.1.2. Criteria for DCGAN Selection:

Generally, different data types will require different GAN architectures. For example, image datasets will usually need a DCGAN or a Style GAN. DCGAN is used in the GAN implementation over Style GAN for generating images from the MNIST and Anime Cartoon datasets because it is simpler, easier and faster to train, and is much resource-efficient for these dataset. Although, Style GAN is a powerful architecture, for the specific task of generating images from the two datasets, DCGAN will be a practical and efficient choice. For datasets like MNIST and Anime Cartoon, using a simpler architecture like DCGAN may be a more resource efficient use and also these may not need additional complexity provided by Style GAN.

1.2. Variational Auto-encoder:

Variational auto-encoder is a neural network model which regularizes the encoding parameters for new data generation. Similar to GAN, VAE also has two components called as a Encoder and a Decoder and the model tries to find the best possible encoder decoder pair. The Encoder compresses the input images to lower dimensions, known as latent space. Encoder filter the number of features by reducing the number of dimensions. The encoder will output a mean and standard deviation describing the latent variables distribution. The Decoder samples from these images and attempts to re-construct so that it looks similar to the original ones and model tries to find pair which reduces the error in estimation. VAE is generally not restricted by any linear subspace. It allows non-linear subspace or layers supporting dimensionality reduction. It is variational because the encoding parameters and the each layer input is regularized which results in a better latent space[3].

The encoder-decoder of this architecture is coupled by a bottleneck layer which optimizes using the backpropagation technique through a normal distribution. This coupling layer will compress the given data into a low-dimensional representation

and the decoder uses it for new sample data generation. Encoder outputs the mean and standard deviation of bottleneck layer activations and are used for sampling from the normal distribution, adding noise to this layer's activations.

A bias term ϵ is derived from the standard deviation and will be added to maintain the gradient's stochasticity. This VAE will learn a probability distribution based on the input data and use it to generate new samples.

$$z=\mu+\sigma\odot\epsilon$$

z is latent space and it will be sent as an input to the decoder.

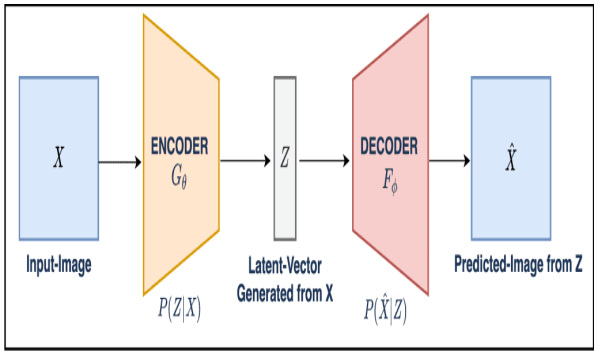


Figure Source: GAN description by google

1.3. Datasets

a) MNIST Dataset:
(Source: MNIST dataset)

The MNIST dataset is a popularly used dataset in the field of machine learning and consists of 70,000 handwritten digits where each of it is represented as a 28x28 grayscale image. The dataset fed to the model in two parts, a training set consisting of 60,000 samples and a test set consisting of 10,000 samples. Interestingly, there is enumerous research been performed on MNIST dataset[5].

The MNIST dataset is pre-processed and fed to GAN or VAE models for training to generate new samples of handwritten digits images. GAN and VAE were implemented for this dataset and the next section of this paper will provide more insights of results.

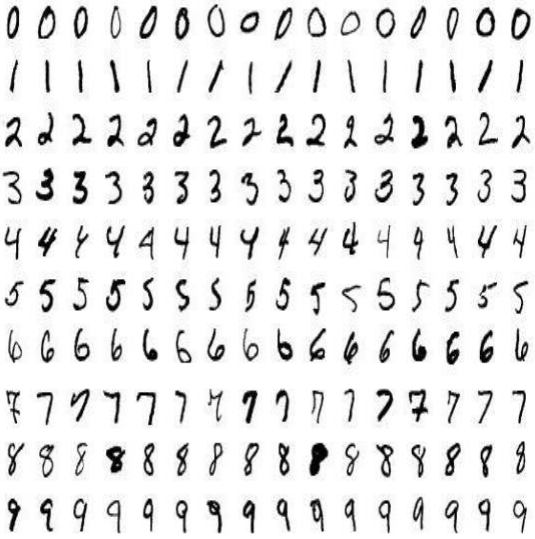


Figure Source: MNIST Dataset

b) Anime Cartoon Dataset:
(Source: Anime Dataset)

This is the dataset which has 63,632 "high-quality" anime faces in it. This dataset is taken from Kaggle. This is an open-source dataset which is used to train the VAE model.

Anime Cartoon dataset is pre-processed and fed to GAN or VAE models for training to generate new samples of anime cartoon images. GAN and VAE were implemented for this dataset and the next section of this paper will provide more insights of results.

Figure Source: ANIME Cartoon Dataset

2. Implementation

All the GAN and VAE implementations used in this project were based on pre-existing implementations available in TensorFlow and Keras libraries. In addition to these, PyTorch was also used for some of the implementations. Standard Python libraries were used for computation and plotting.

2.1 DCGAN

2.1.1. MNIST Dataset:

In this implementation, DCGAN is used for image generation based on MNIST dataset. DCGAN is implemented using TensorFlow with the addition of layers where each of it handles the input from previous layer or step, applies activation function like leaky ReLU etc and batch normalization etc.

The dataset fed to the model in two parts, a training set consisting of 60,000 samples and images are normalized to [-1,1]. The below code depicts this [6] :

```
(train_images, train_labels), (_, _) = tf.keras.datasets.mnist.load_data()
train_images = train_images.reshape(train_images.shape[0], 28, 28, 1).astype('float32')
train_images = (train_images - 127.5) / 127.5  # Normalize the images to [-1, 1]
BUFFER_SIZE = 60000
BATCH_SIZE = 256

# Batch and shuffle the data
train_dataset = tf.data.Dataset.from_tensor_slices(train_images).shuffle(BUFFER_SIZE).batch(BATCH_SIZE)
```

Generator:

The goal of this implementation is to train a generator model such that it generates new and realistic images of handwritten digits similar to the samples in the MNIST dataset. This generator is trained along with a discriminator model which distinguishes between real images from the MNIST dataset and fake images generated by the generator.

Generatoruses tf.keras.layers.Conv2DTranspose layers to produce images from random noise(seed). The below code shows how it starts with a Dense layer which takes this seed as input and up-sample multiple times till desired 28x28x1 image size is achieved. Each layer uses LeakyReLU activation function for each layer and the layer which will output uses tanh function.

Output of this generator for this dataset is 28*28*1 image because it is grey scale as it has just two values 0 and 1.

```
def make_generator_model():
    model = tf.keras.Sequential()
    model.add(layers.Dense(7*7*256, use_bias=False, input_shape=(100,)))
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Reshape((7, 7, 256)))
    assert model.output_shape == (None, 7, 7, 256)

    model.add(layers.Conv2DTranspose(128, (5, 5), strides=(1, 1), padding='same', use_bias=False))
    assert model.output_shape == (None, 7, 7, 128)
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Conv2DTranspose(64, (5, 5), strides=(2, 2), padding='same', use_bias=False))
    assert model.output_shape == (None, 14, 14, 64)
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Conv2DTranspose(1, (5, 5), strides=(2, 2), padding='same', use_bias=False, activation='tanh'))
    assert model.output_shape == (None, 28, 28, 1)

    return model
```

This generates image samples which will be sent to the discriminator.

Discriminator

This discriminator is a binary classifier. It consists of multiple layers of convolutional and dense layers. It contains two convolutional layers with 64 and 128 filters,

respectively, each followed by a LeakyReLU activation function and a dropout layer to prevent overfitting. The output of these layers is flattened and passed through a dense layer with a single output node, representing the discriminator's final decision on whether the input image is real or fake.

This discriminator takes a 28x28 grayscale image as input and produce a single output value indicating the likelihood that the produced input image is real.

```
def make_discriminator_model():
    model = tf.keras.Sequential()
    model.add(layers.Conv2D(64, (5, 5), strides=(2, 2), padding='same',
                            input_shape=[28, 28, 1]))
    model.add(layers.LeakyReLU())
    model.add(layers.Dropout(0.3))

    model.add(layers.Conv2D(128, (5, 5), strides=(2, 2), padding='same'))
    model.add(layers.LeakyReLU())
    model.add(layers.Dropout(0.3))

    model.add(layers.Flatten())
    model.add(layers.Dense(1))

    return model
```

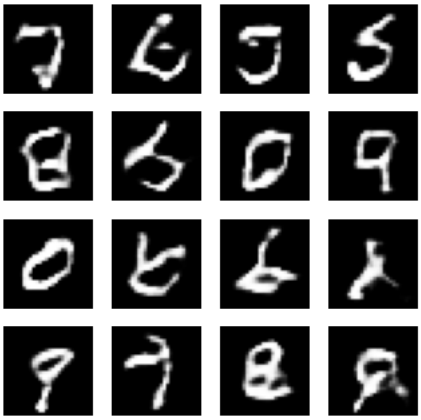
Loss Function and Optimizer

In a GAN, the discriminator and generator have their own loss functions and optimizers. The discriminator's loss function measures its ability to distinguish between real and fake images, while the generator's loss function measures its ability to produce realistic-looking images. Optimizers are used to update the network parameters to minimize the loss functions. The loss functions and optimizers indicate how well each network is performing during training.

```
def discriminator_loss(real_output, fake_output):
    real_loss = cross_entropy(tf.ones_like(real_output), real_output)
    fake_loss = cross_entropy(tf.zeros_like(fake_output), fake_output)
    total_loss = real_loss + fake_loss
    return total_loss

def generator_loss(fake_output):
    return cross_entropy(tf.ones_like(fake_output), fake_output)

generator_optimizer = tf.keras.optimizers.Adam(1e-4)
discriminator_optimizer = tf.keras.optimizers.Adam(1e-4)
```

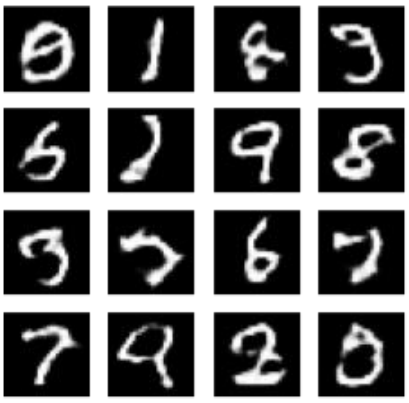
2.1.1(a) Results

After training, the below figure shows an image at epoch 46.



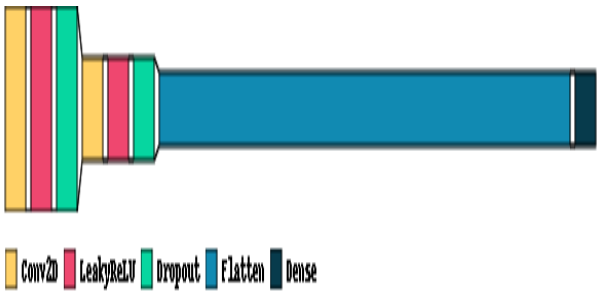
Time for epoch 46 is 11.903103828430176 s

The below figure is the last generated image by the generator at the end of epoch =100

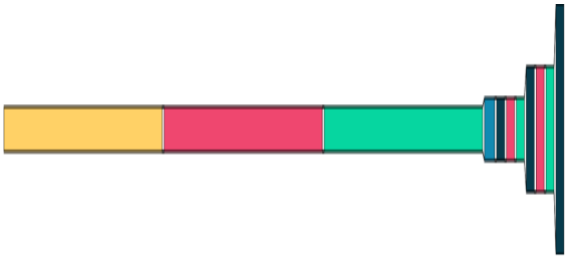


The figure below depicts the layers included in the implementation: [1]

Discriminator Visualisation:



Generator Visualisation:



2.1.2 ANIME Cartoon Dataset

In this implementation, DCGAN is used for image generation based on ANIME CARTOON dataset. DC GAN is implemented using PyTorch with the addition of layers where each of it handles the input from previous layer or step, applies activation function like ReLU etc and batch normalization etc. [4]

Generator:

This Generator also has same functionality as quoted for MNIST above but below is the code for generator used with multiple layers.

```
generator = nn.Sequential(
    # in: latent_size x 1 x 1

    nn.ConvTranspose2d(latent_size, 512, kernel_size=4, stride=1, padding=0, bias=False),
    nn.BatchNorm2d(512),
    nn.ReLU(True),
    # out: 512 x 4 x 4

    nn.ConvTranspose2d(512, 256, kernel_size=4, stride=2, padding=1, bias=False),
    nn.BatchNorm2d(256),
    nn.ReLU(True),
    # out: 256 x 8 x 8

    nn.ConvTranspose2d(256, 128, kernel_size=4, stride=2, padding=1, bias=False),
    nn.BatchNorm2d(128),
    nn.ReLU(True),
    #out: 128 x 16 x 16

    nn.ConvTranspose2d(128, 64, kernel_size=4, stride=2, padding=1, bias=False),
    nn.BatchNorm2d(64),
    nn.ReLU(True),
    #out: 64 x 32 x 32

    nn.ConvTranspose2d(64, 3, kernel_size=4, stride=2, padding=1, bias=False),
    nn.Tanh()
    #out: 3 x 64 x 64
)
```

Discriminator:

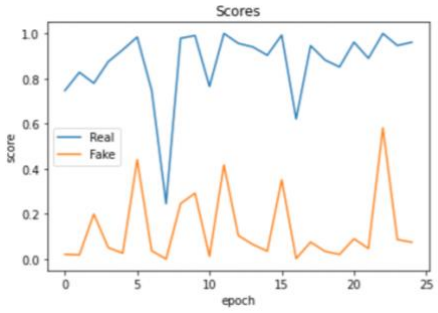
Output of above generator for this dataset is 64*64*3 image because it contains 64*64 image size and 3 because these images are colourful using RGB code.

```
discriminator = nn.Sequential(
    # in: 3 x 64 x 64

    nn.Conv2d(3, 64, kernel_size=4, stride=2, padding=1, bias=False),
    nn.BatchNorm2d(64),
    nn.LeakyReLU(0.2, inplace=True),
    # 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())
```

2.1.2(a) Results

The below figure depicts one of the randomly generated Anime image after some epochs

The below graph shows the scores as the epoch progress



2.2. VAE

2.2.1 MNIST

In VAEs, the goal is to learn a latent representation of the MNIST images that can be used to generate new, similar images. The VAE is trained to minimize the difference between the input images and the reconstructed images generated by the decoder, while also encouraging the encoder to map similar images to nearby points in the latent space. Through this training process, the VAE learns to generate new images by sampling points from the latent space and decoding them with the decoder.[8]

Encoder and Decoder:

Encoder contains 2 visible layers of input and output and 3 hidden layers to perform convolution and Decoder is simply a mirror image and thus contains equal number of deconvolution layers.

```
def __init__(self, latent_dim):
  super(CVAE, self).__init__()
  self.latent_dim = latent_dim
  self.encoder = tf.keras.Sequential(
      [
          tf.keras.layers.InputLayer(input_shape=(28, 28, 1)),
          tf.keras.layers.Conv2D(
              filters=32, kernel_size=3, strides=(2, 2), activation='relu'),
          tf.keras.layers.Conv2D(
              filters=64, kernel_size=3, strides=(2, 2), activation='relu'),
          tf.keras.layers.Flatten(),
          # No activation
          tf.keras.layers.Dense(latent_dim + latent_dim),
      ]
  )

  self.decoder = tf.keras.Sequential(
      [
          tf.keras.layers.InputLayer(input_shape=(latent_dim,)),
          tf.keras.layers.Dense(units=7*7*32, activation=tf.nn.relu),
          tf.keras.layers.Reshape(target_shape=(7, 7, 32)),
          tf.keras.layers.Conv2DTranspose(
              filters=64, kernel_size=3, strides=2, padding='same',
              activation='relu'),
          tf.keras.layers.Conv2DTranspose(
              filters=32, kernel_size=3, strides=2, padding='same',
              activation='relu'),
          # No activation
          tf.keras.layers.Conv2DTranspose(
              filters=1, kernel_size=3, strides=1, padding='same'),
      ]
  )
```

Training

- Iterate over the dataset such that in each iteration, the encoder gets an image and then obtain a set of mean and log-variance parameters of the approximate

  posterior $q(z|x)$

- Apply the reparameterization trick to sample from $q(z|x)$
- Later pass the reparametrized samples to the decoder to obtain the logits of the generative distribution $p(x|z)$
- ELBO on the test set differs slightly than the results in the literature which uses dynamic binarization of Larochelle's MNIST.

Generating images

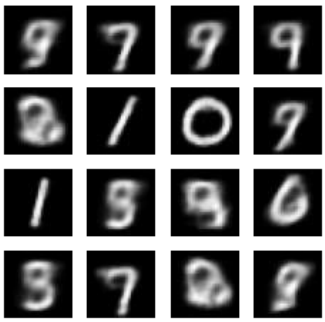
- After training completed, generation of images will start
- Sample a set of latent vectors from the unit Gaussian prior distribution $p(z)$
- The generator then converts the latent sample z to logits of the observation, giving a distribution $p(x|z)$
- Plot the probabilities of Bernoulli distributions

2.2.1(a) Results

We can see model improving over time and as the training set grows and that is because it is able to minimize the reconstruction error by learning parameters for gaussian.

When epochs is set to 8,

Epoch: 1, Test set ELBO: -157.3997039794922, time elapse for current epoch: 9.56974172592163

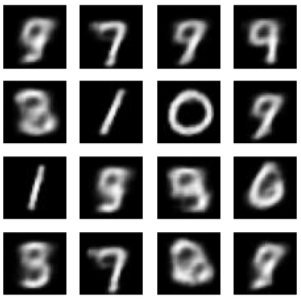


It similarly generated for different epoch values till epoch=8

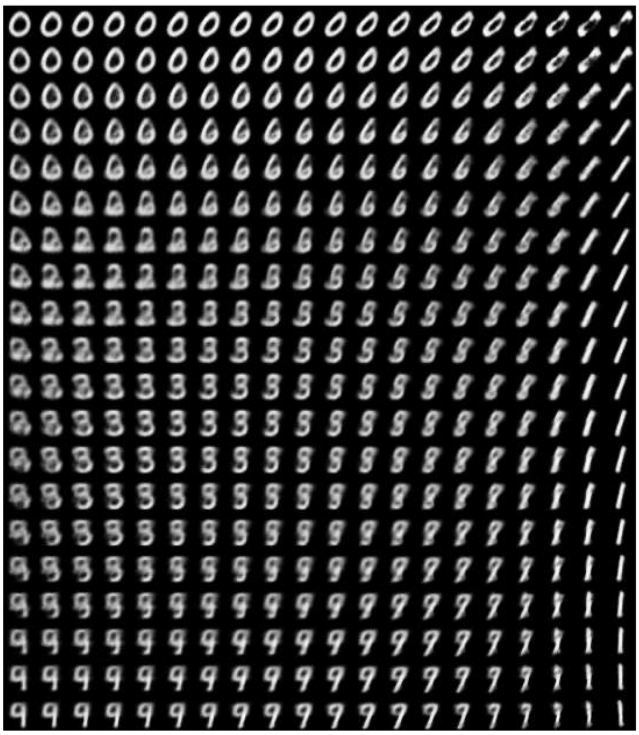
Epoch: 8, Test set ELBO: -157.97857666015625, time elapse for current epoch: 6.2572922706604

Based on the epoch 8, the image generated is below with coordinates:

(-0.5, 287.5, 287.5, -0.5)

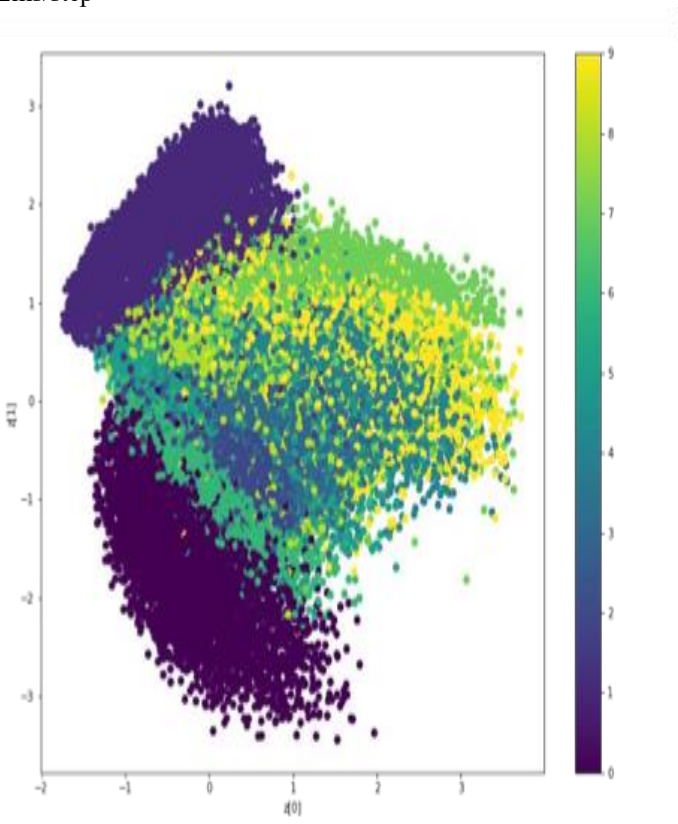


The image below represents the continuous distribution of digits or the images and how one image transforms into another over a 2D latent space. This is also a testament to model's capability to classify digits accurately from MNIST dataset.



When a plot was drawn between two features z[0] and z[1] , below is the output.

1875/1875
[==============================] - 4s 2ms/step



2.2.2 ANIME

Encoder & Decoder:

Encoder contains 2 visible layers of input and output and 3 hidden layers to perform convolution and Decoder is simply a mirror image and thus contains equal number of deconvolution layers.

```
        # Encoder
        for h_dim in hidden_dims:
            modules.append(
                nn.Sequential(
                    nn.Conv2d(in_channels, out_channels=h_dim,kernel_size= 3, stride= 2, p
adding  = 1),
                    nn.BatchNorm2d(h_dim),
                    nn.LeakyReLU()
                )
            )
            in_channels = h_dim

        self.encoder = nn.Sequential(*modules)
        self.fc_mu = nn.Linear(hidden_dims[-1]*4, latent_dim)
        self.fc_var = nn.Linear(hidden_dims[-1]*4, latent_dim)
```

```
        hidden_dims.reverse()

        for i in range(len(hidden_dims) - 1):
            modules.append(
                nn.Sequential(
                    nn.ConvTranspose2d(hidden_dims[i],
                                       hidden_dims[i+1],
                                       kernel_size = 3,
                                       stride = 2,
                                       padding = 1,
                                       output_padding=1),
                    nn.BatchNorm2d(hidden_dims[i+1]),
                    nn.LeakyReLU()
                )
            )
        self.decoder = nn.Sequential(*modules)

        self.final_layer = nn.Sequential(
            nn.ConvTranspose2d(hidden_dims[-1],
                               hidden_dims[-1],
                               kernel_size=3,
                               stride=2,
                               padding=1,
                               output_padding=1),
            nn.BatchNorm2d(hidden_dims[-1]),
            nn.LeakyReLU(),
            nn.Conv2d(hidden_dims[-1], out_channels=3, kernel_size=3, padding=1),
            nn.Sigmoid()
        )
```

Loss function and other functions:

Similar to GAN, the loss function is used for VAE and the below code depicts it. This remains same for both the datasets. This Loss function uses KL divergence, cross entropy and logarithmic.

```
def vae_loss(recon_x, x, mu, logvar) -> float:
    BCE = F.binary_cross_entropy(recon_x.view(-1, image_size*image_size*3),
                                 x.view(-1, image_size*image_size*3), reduction='sum')
    KLD = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp())
    return BCE + KLD
```

```
    def reparameterize(self, mu, logvar):
        std = logvar.mul(0.5).exp_()
        eps = torch.randn(*mu.size()).cuda()
        z = mu + std * eps
        return z

    def bottleneck(self, h):
        mu = self.h2mu(h)
        logvar = self.h2sigma(h)
        z = self.reparameterize(mu, logvar)
        return z, mu, logvar

    def encode(self, x):
        return self.bottleneck(self.encoder(x))[0]

    def decode(self, z):
        return self.decoder(self.z2h(z))

    def forward(self, x):
        h = self.encoder(x)
        z, mu, logvar = self.bottleneck(h)
        z = self.z2h(z)
        return self.decoder(z), mu, logvar
```

3. Conclusion

- GAN and VAE are two widely used generative models and they use parametrized distributions for underlying data distribution approximation but their behaviours are mostly different
- VAE has lesser training time required than GAN based on the results produced for both the datasets
- GAN produced comparatively better and high quality images when executed for same number of epochs
- Probability of discriminator of GAN for both the datasets was close to 0.5 indicating it was a good model

```
import visualkeras
from PIL import ImageFont

discriminator = make_discriminator_model()
decision = discriminator(generated_image)
discriminator_probability = tf.sigmoid(decision)

# Print the probability
print("Discriminator probability:", discriminator_probability.numpy())
print (decision)
```

```
Discriminator probability: [[0.50050724]]
tf.Tensor([[0.0020289]], shape=(1, 1), dtype=float32)
```

4. Future Scope

In this report, implementation of GAN and VAE was performed on two different datasets- MNIST(grey-scale digits images) and ANIME Cartoon (Colour Anime images) depicting how model performs based on different data characteristics. Similarly, this work can be extended to work on varied datasets. Different variations of GAN and VAE can be used these datasets and check results. These generative models can also be extended further and be used to solve classification and regression problems.

In conclusion, these generative models are very powerful tools for businesses, medicine field, researchers to extract important insights from the data and for process automation in this increasingly data-blooming world. The application of Generative models will gradually increase in coming years demanding more sophisticated models[3].

5. Comparison of Results

After GAN and VAE were implemented on two datasets, the results and observations were logged in below table.

6. References

1) https://www.analyticsvidhya.com/blog/2022/03/visualize-deep-learning-models-using- visualkeras/
2) Jingzhao, ZhangLu, MiMacheng Shen1, Massachusetts Institute of Technology,Cambridge,USA(https://www.arxiv-vanity.com/papers/1812.05676/)
3) An empirical comparison of generative capabilities of GAN vs VAE, HANNA PETERS, NORMA CUETO CELIS
4) https://www.kaggle.com/code/muhammad4hmed/anime-gan
5) https://arxiv.org/pdf/1406.2661.pdf
6) https://colab.research.google.com/drive/1MZ7LRe7vT2omxO4Usa5mXRkp8RbF6qHe?usp=sharing
7) https://www.kaggle.com/code/emilkuban/understanding-vae-by-generating-anime-faces
8) https://www.tensorflow.org/tutorials/generative/cvae

7. Results Comparision:

MNIST:

	Probability of Discriminator	Time taken for running on MNIST dataset	Image Quality	Latent Space
GAN	0.5005	For 50 epochs, 1020 s	High	Do not model the distribution of the latent variables explicitly, so hard for manipulation
VAE	-	For 50 epochs, 192.5 s	Good, but GAN has high quality images	models the distribution of the latent variables explicitly , easy for manipulation

ANIME:

	Probability of Discriminator	Time taken for running on ANIME dataset	Image Quality	Latent Space
GAN	0.49052	For 50 epochs, 3476.6 s	High	Do not model the distribution of the latent variables explicitly, so hard for manipulation
VAE		For 50 epochs,386.5 s	Good but little blurry when compared to GAN	models the distribution of the latent variables explicitly, easy for manipulation