# Advance Visual Recognition Project Report


## Project Title: Face Generation Using Variational AutoEncoders

**Team Members:**

| Name | Roll Number |
|---|---|
| Rittik Panda | MT2022090 |
| Prasad Magdum | MT2022078 |
| Prithviraj Purushottam Naik | MT2022083 |

**Table of Contents:**

## 1. Simple Feed Forward Autoencoder

An autoencoder is a type of artificial neural network used for unsupervised learning of efficient codings. The aim of an autoencoder is to learn a representation (encoding) for a set of data, typically for dimensionality reduction, by training the network to ignore signal "noise."

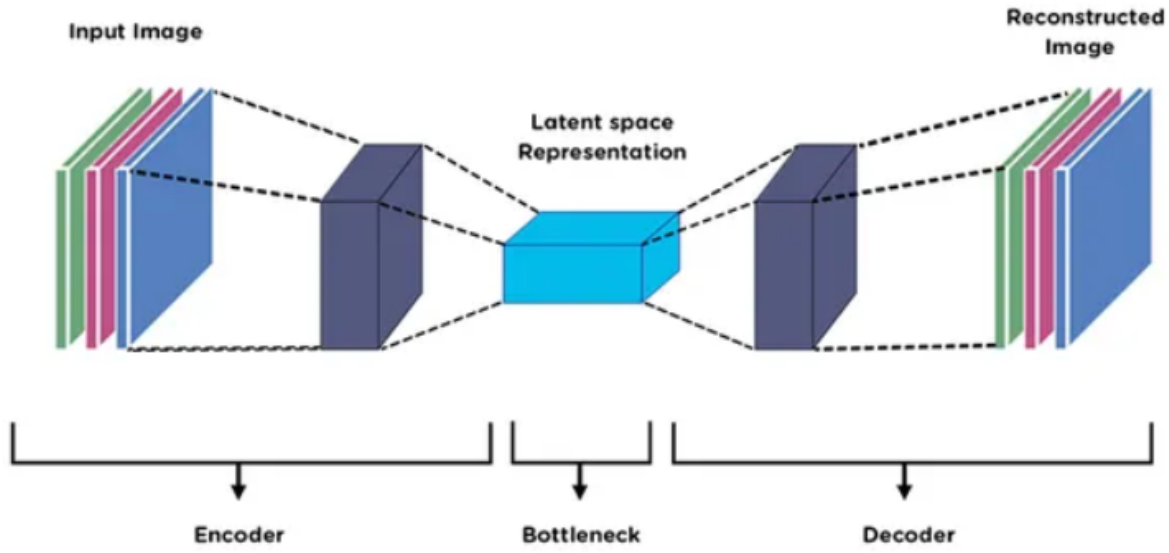



**Figure 1:** Simple Feed Forward Autoencoder Architecture

A simple feedforward autoencoder consists of:

- **Encoder :** This part of the network encodes or compresses the input data into a latent-space representation. The compressed data typically looks garbled, nothing like the original data.

- **Decoder :** This part of the network decodes or reconstructs the encoded data(latent space representation) back to original dimension. The decoded data is a lossy reconstruction of the original data.

The network is trained to minimize reconstruction error, i.e. the difference between the input and output values over the entire training dataset. This forces the network to prioritize which aspects of the data should be

preserved in the reduced encoding to best be able to reconstruct the original input.
This compact representation of the most important features can then be used for other machine learning tasks. The decoding portion can also be reused to reconstruct original inputs from the encoded values.
So in summary, the simple feedforward autoencoder compresses input data into an encoded representation, and is trained to decompress that representation back into a version that closely matches the original input.

# 2. Deep Convolutional Autoencoder

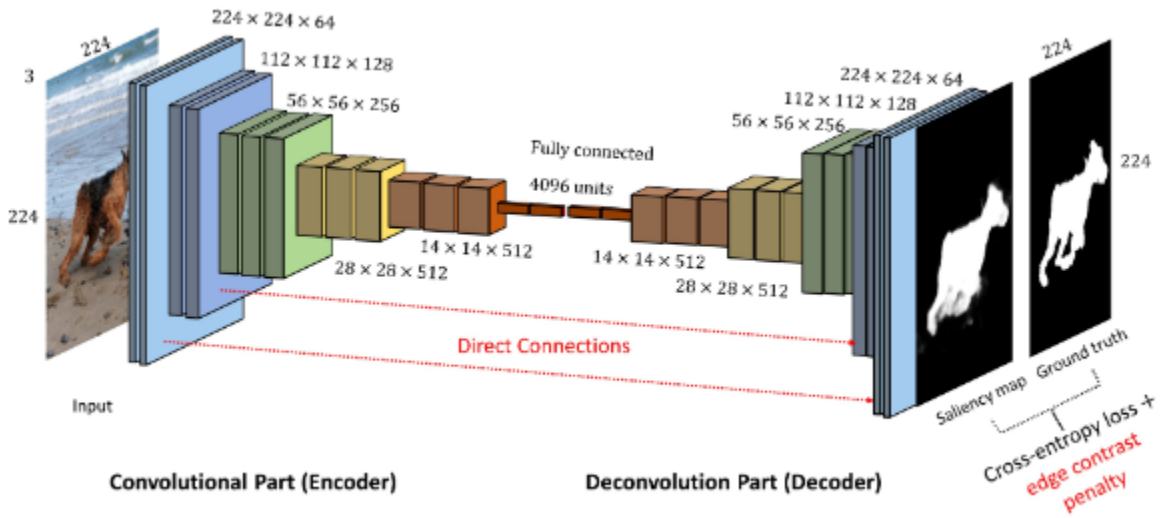



**Figure 2:** Deep Convolutional Autoencoder Architecture

- **Encoder**
  The Encoder part of the network is to compress the input data and passes it to the Bottleneck layer. The compression creates a knowledge representation much smaller than the original input but has most of its features.
  This part of the network comprises blocks of convolutions followed by pooling layers that, in turn, further help to create a compressed data representation. The output of an ideal Encoder should be the same as the input but with a smaller size. The Encoder should be sensitive to the inputs to recreate it and not over-sensitive. Being over-sensitive would make the model memorize the inputs perfectly and then overfit the data.

- **Bottleneck Layer**
  The Bottleneck is the most important layer of an Autoencoder. This module stores the compressed knowledge that is passed to the Decoder. The Bottleneck restricts information flow by only allowing important parts of the compressed representation to pass through to the Decoder. Doing so ensures that the input data has the maximum

possible information extracted from it and the most useful correlations found. This part of the architecture is also a measure against overfitting as it prevents the network from memorizing the input data directly.

- **Decoder**
  This part of the network is a "Decompressor" that attempts to recreate an image given its latent attributes. The Decoder gets the compressed information from the Bottleneck layer and then uses upsampling and convolutions to reconstruct it. The output generated by the Decoder is compared with the ground truth to quantify the network's performance.

- **Latent Space Structure**
  The latent space of a network is the compressed representation it creates from an input dataset. This latent space usually has hundreds of dimensions and is hard to visualize directly. More complex neural networks have latent spaces so hard to visualize that they are generally called black boxes. In a convolutional autoencoder, the better the representation of the data, the richer the latent space. The space structure here is a large matrix of tensors that encode the weights of network layers.

## 3. Denoising Autoencoder

Denoising Autoencoders are neural network models that remove noise from corrupted or noisy data by learning to reconstruct the initial data from its noisy counterpart. We train the model to minimize the disparity between the original and reconstructed data. We can stack these autoencoders together to form deep networks, increasing their performance.

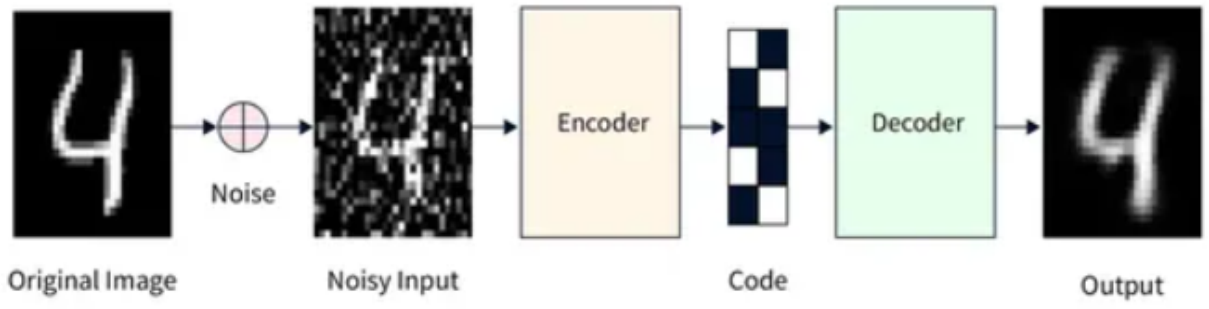



**Figure 3:** Denoising Autoencoder Architecture

- **Encoder**
  The encoder creates a neural network equipped with one or more hidden layers. Its purpose is to receive noisy input data and generate an encoding, which represents a low-dimensional representation of the data. Understand an encoder as a compression function because the encoding has fewer parameters than the input data.

- **Decoder**
  Decoder acts as an expansion function, which is responsible for reconstructing the original data from the compressed encoding.
  It takes as input the encoding generated by the encoder and reconstructs the original data. Like encoders, decoders are implemented as neural networks featuring one or more hidden layers.

During the training phase, present the denoising autoencoder (DAE) with a collection of clean input examples along with their respective noisy counterparts. The objective is to acquire a function that maps a noisy input

to a relatively clean output using an encoder-decoder architecture. To achieve this, a reconstruction loss function is typically employed to evaluate the disparity between the clean input and the reconstructed output. A DAE is trained by minimizing this loss through the use of backpropagation, which involves updating the weights of both encoder and decoder components.

## 4. Variational Autoencoder

- ***Working of Variational Autoencoder in detail:***

So far we have seen how autoencoders compress input and later reconstruct that same input which is learned and stored in the form of encoded representation in the bottleneck layer. But till now we were learning directly from data which means transforming input as encoded representation first and later we feed those representations to the decoder directly to reconstruct and see what the input looks like. That's the simple intuition behind auto encoders .

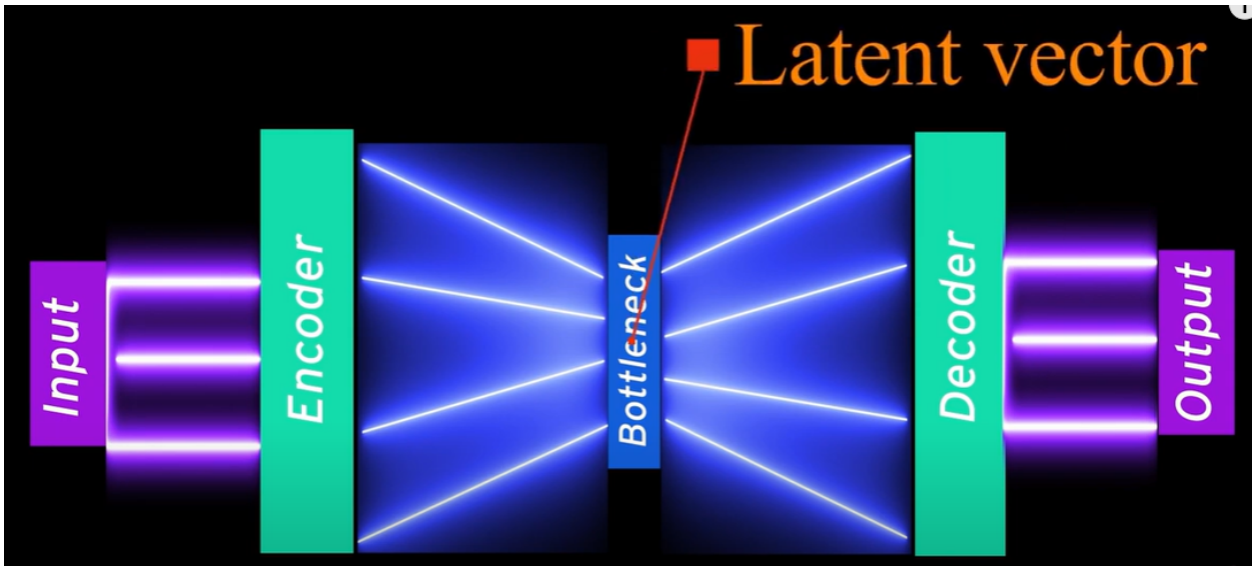



**Figure 4:** Autoencoders

We have done some pretty amazing job like removing noise from images using denoising autoencoder where we prevented an autoencoder from learning the identity function. But still we were reconstructing inputs and we don't have any way to generate new images. Now it's time to break that limitation to make autoencoders generate new data instead of reconstructing that same boring inputs again and again. This is where we need a variational autoencoder to do that. Instead of learning from input data a variational autoencoder learns from input distribution it means the bottleneck layer will be replaced by two separate layers one is the mean of the distribution and the other is a standard deviation

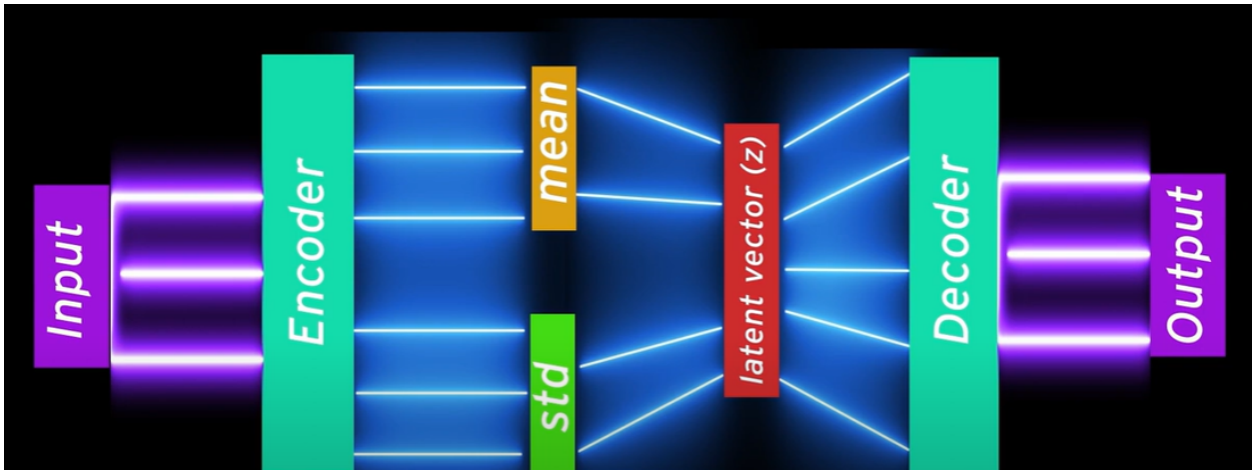



**Figure 5:** Variational Autoencoders

But what about the decoder now how will it reconstruct? Well, to do that we take a sample from distribution then feed it into the decoder let's call it **z.** At the output end we'll have a reconstruction loss function which will measure how far the output is from the original result and once we have the loss we can back propagate by pushing gradients.

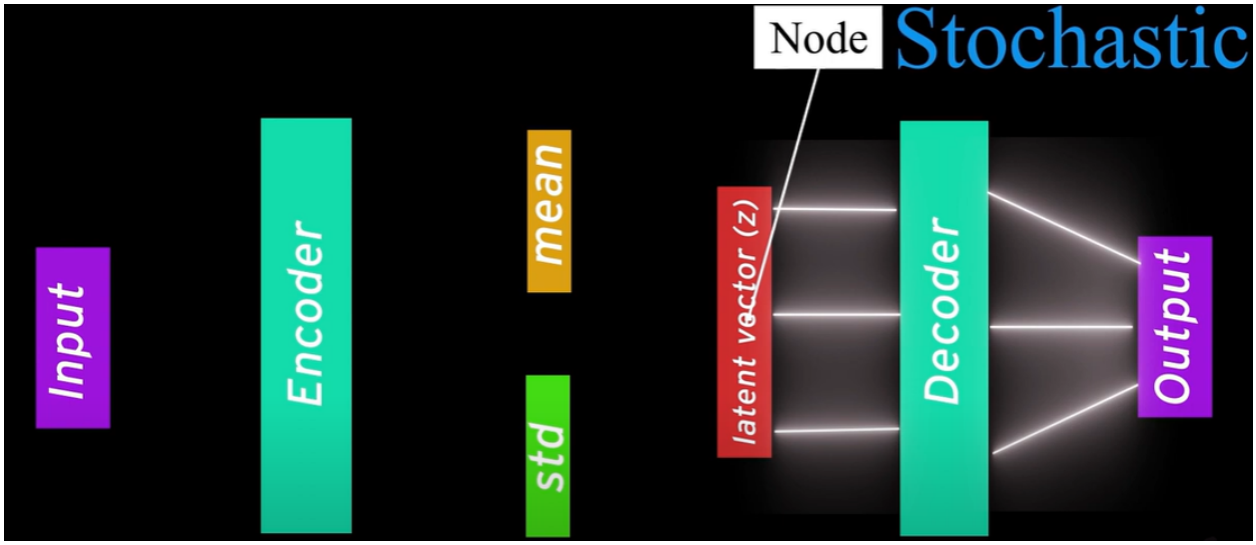



**Figure 6:** Variational Autoencoders

but we cannot apply back propagation on a sampling layer normally because this sampling node collects samples from the distribution hence it is stochastic in nature. Moreover, we can't push gradients through a stochastic node. Instead of taking samples directly from distribution we can

use an expression to generate latent vector **z** which is mu plus sigma times epsilon where mu is mean and the sigma is a standard deviation which are already learnable parameters and epsilon is a standard normal distribution having mean of 0 and standard deviation of 1.

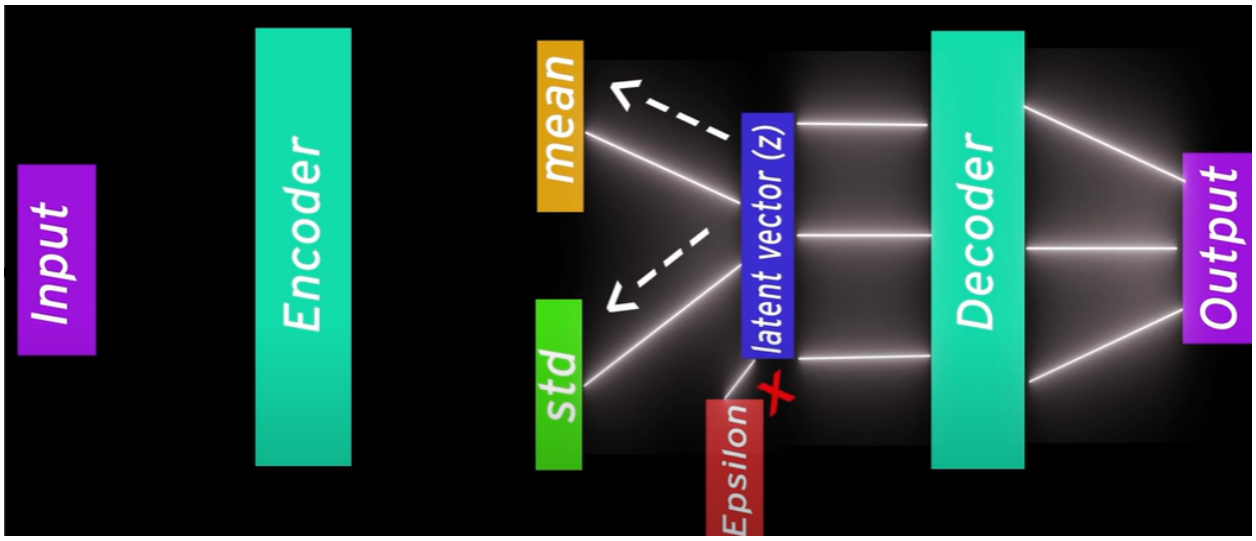



**Figure 7:** Variational Autoencoders

Now we have a modified architecture in which we can push gradients from this way without touching the epsilon; this trick is called the **reparameterization trick.**

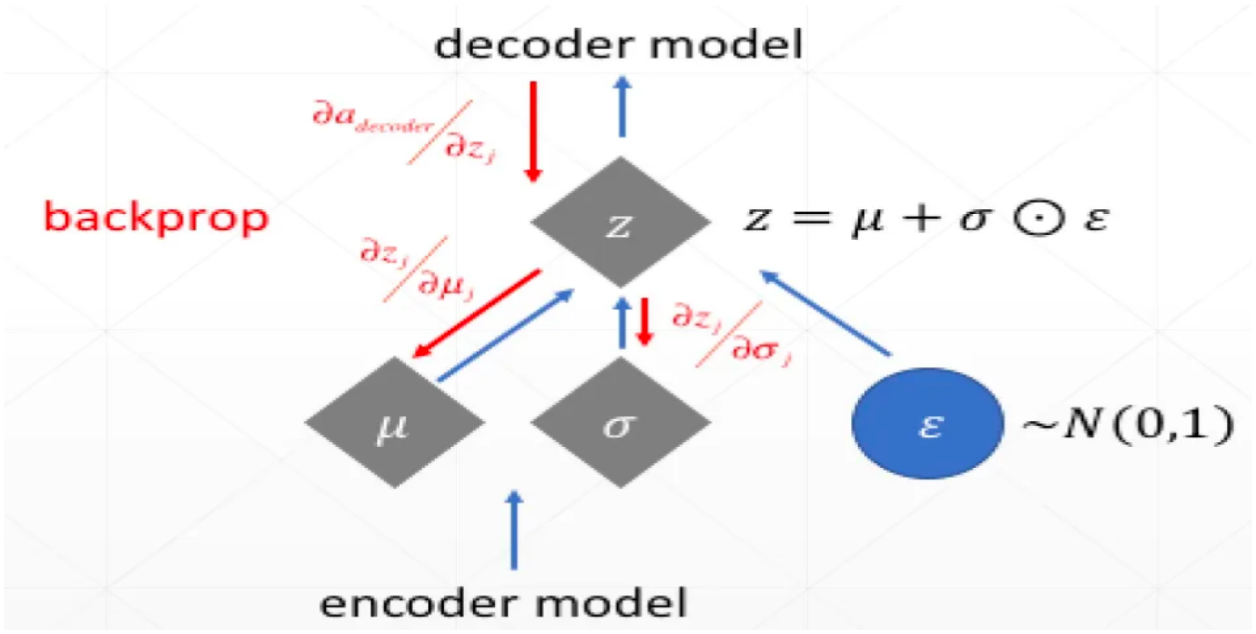



**Figure 8:** Reparameterization trick

Now we can back propagate and update parameters for every layer in the network. But the network is not really ready to train yet.
Little bit improvement is needed near the mean and standard distribution layer as in Fig 6. As we are learning from the distribution we have to make sure that the distribution we are learning is not too far from the standard normal distribution. In short we need a regularization here to force the learn distribution remain close to
standard normal distribution.
Here comes a concept known as Kullback–Leibler divergence better known as the KL divergence. It is the measure of how two distributions are different from each other.

$$KL(P||Q) = \sum P(X)log(P(X)) - P(X)log(Q(X))$$

**Equation 1:** KL Divergence formula

For example the KL divergence of distribution p to p is zero because they belong to the same distribution. Hence there is no difference between them so if we add this into our current loss function it will
act as a regularization function.

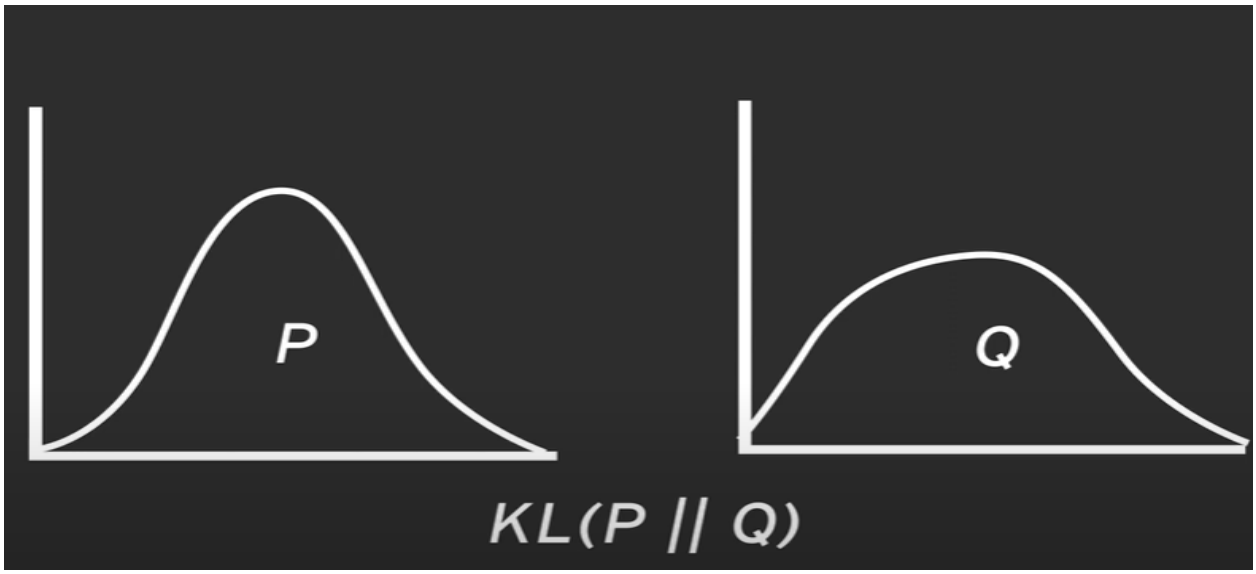



**Figure 9:** KL Divergence

It measure how the currently learned distribution is far from the normal distribution and make it close to the mean of 0 and standard deviation of 1.

$$Loss = mse + KL(q(Z) \,||\, N(0,1))$$

**Equation 2:** Loss function for Variational Autoencoder

Hence this is the original loss function mentioned in the paper and by minimizing this we can recover a perfect variational autoencoder so this is the working of variational autoencoder.

- ***Implementation of Variational Autoencoder:***

1. **Dataset:**

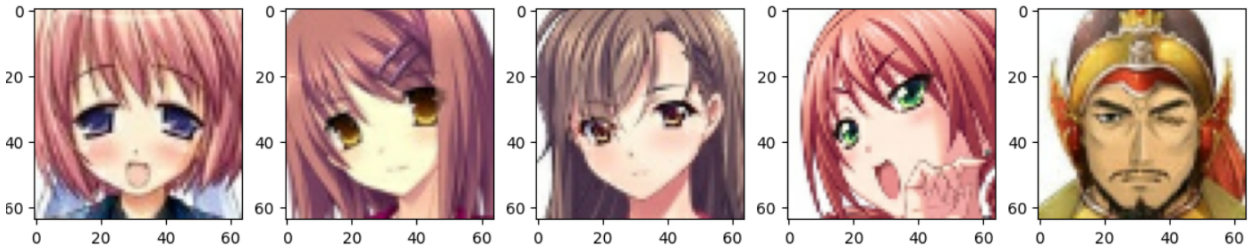
**In this project we worked with 2 datasets.**

**A. <u>Anime Faces</u>**

I. Number of Images: 30750

II. Image dimension size: random(Average width: 104.88874796747967, Average height: 104.88832520325204)

III. Sample of Images:



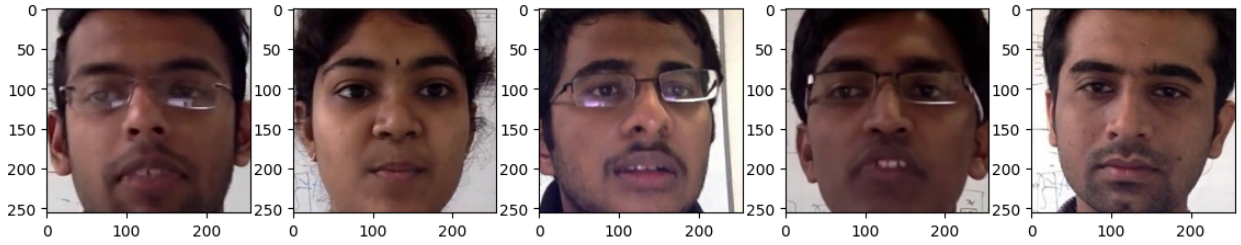
IV. Dataset link: https://drive.google.com/file/d/1o7DgpRE198-hV1i58VThzXYK9_0AAsl4/view?usp=sharing

**B.<u>IIITB-Faces</u>**

I. Images of 49 different students & for each student there are nearly 15-20 images.

II. Total 832 images of random dimensions.(Average width:499.92668269, Average height: 499.30408653846155)

III. Sample of Images:



IV. Dataset link:https://drive.google.com/drive/folders/1-vXd9v3DPDhgfyE-i4jovSxYWKWjwgYA?usp=sharing

2. **Preprocessing Steps:**

**A. <u>Anime Faces</u>**

**I. Read the image file:**

tf.io.read_file(image): This function reads the content of the image file specified by the 'image' parameter.

**II. Decode the JPEG-encoded image:**

tf.io.decode_jpeg(image): This function decodes the JPEG-encoded image content, converting it to a numerical tensor.

**III. Convert the image data type to float32:**

tf.cast(image, tf.float32): This line converts the data type of the image tensor to float32. Deep learning models often work with floating-point values.

**IV. Resize the image:**

tf.image.resize(image, (image_size, image_size)): This step resizes the image to the specified dimensions (height = 64, width = 64). Resizing is a common preprocessing step to ensure all input images have the same size.

**V. Normalize pixel values:**

image / 255.0: This line normalizes the pixel values to the range [0, 1]. It's a standard practice to scale pixel values before feeding them into a neural network, as it helps in better convergence during training.

**VI. Reshape the image:**

tf.reshape(image, shape=(image_size, image_size, 3)): This line reshapes the image tensor to the desired shape, which is (64, 64, 3). The '3' corresponds to the three color channels (red, green, and blue) commonly found in RGB images.

**B.<u>IIITB-Faces</u>**

1.**Face Detection & Cropping:**One of the first steps in face verification is to isolate the actual face from the background of the image. This step chosen for various reasons:

I. It effectively eliminated a substantial amount of background and extraneous elements from the input data.
II. It enhanced the extraction of facial features as face-detected images tend to have consistent positioning of facial landmarks like eyes, nose, and mouth.
III. Face detection algorithms also are able to deal with bad and inconsistent lighting and various facial positions such as tilted or rotated faces. Here for face detection we have used OpenCV's haar cascade classifier.

Original Image

After Face Detection & Cropping

**2.**After that, same preprocessing steps as the '**Anime Faces'** dataset were done with image width = 256 and image height = 256.

## 3. Model Architecture:

## I. Encoder:

**Anime Faces:**

| Layer (type) | Output Shape | Param # | Connected to |
|---|---|---|---|
| input_1 (InputLayer) | [(None, 64, 64, 3)] | 0 | [] |
| conv2d (Conv2D) | (None, 64, 64, 32) | 2432 | ['input_1[0][0]'] |
| batch_normalization (BatchNormalization) | (None, 64, 64, 32) | 128 | ['conv2d[0][0]'] |
| conv2d_1 (Conv2D) | (None, 32, 32, 64) | 51264 | ['batch_normalization[0][0]'] |
| batch_normalization_1 (BatchNormalization) | (None, 32, 32, 64) | 256 | ['conv2d_1[0][0]'] |
| conv2d_2 (Conv2D) | (None, 16, 16, 128) | 204928 | ['batch_normalization_1[0][0]'] |
| batch_normalization_2 (BatchNormalization) | (None, 16, 16, 128) | 512 | ['conv2d_2[0][0]'] |
| conv2d_3 (Conv2D) | (None, 8, 8, 256) | 819456 | ['batch_normalization_2[0][0]'] |
| batch_normalization_3 (BatchNormalization) | (None, 8, 8, 256) | 1024 | ['conv2d_3[0][0]'] |
| conv2d_4 (Conv2D) | (None, 4, 4, 512) | 3277312 | ['batch_normalization_3[0][0]'] |
| batch_normalization_4 (BatchNormalization) | (None, 4, 4, 512) | 2048 | ['conv2d_4[0][0]'] |
| flatten (Flatten) | (None, 8192) | 0 | ['batch_normalization_4[0][0]'] |
| dense (Dense) | (None, 1024) | 8389632 | ['flatten[0][0]'] |
| batch_normalization_5 (BatchNormalization) | (None, 1024) | 4096 | ['dense[0][0]'] |
| dense_1 (Dense) | (None, 512) | 524800 | ['batch_normalization_5[0][0]'] |
| dense_2 (Dense) | (None, 512) | 524800 | ['batch_normalization_5[0][0]'] |
| tf.compat.v1.shape (TFOpLambda) | (2,) | 0 | ['dense_1[0][0]'] |
| tf.compat.v1.shape_1 (TFOpLambda) | (2,) | 0 | ['dense_1[0][0]'] |
| tf.math.multiply (TFOpLambda) | (None, 512) | 0 | ['dense_2[0][0]'] |
| tf.__operators__.getitem (SlicingOpLambda) | () | 0 | ['tf.compat.v1.shape[0][0]'] |
| tf.__operators__.getitem_1 (SlicingOpLambda) | () | 0 | ['tf.compat.v1.shape_1[0][0]'] |
| tf.math.exp (TFOpLambda) | (None, 512) | 0 | ['tf.math.multiply[0][0]'] |
| tf.random.normal (TFOpLambda) | (None, 512) | 0 | ['tf.__operators__.getitem[0][0]', 'tf.__operators__.getitem_1[0][0]'] |
| multiply (Multiply) | (None, 512) | 0 | ['tf.math.exp[0][0]', 'tf.random.normal[0][0]'] |
| add (Add) | (None, 512) | 0 | ['dense_1[0][0]', 'multiply[0][0]'] |

Total params: 13802688 (52.65 MB)
Trainable params: 13798656 (52.64 MB)
Non-trainable params: 4032 (15.75 KB)

**IIITB-Faces:** Encoder part for this dataset is same as the above only the input dimension is 256*256

## II. Decoder:

**Anime Faces:**

| Layer (type) | Output Shape | Param # | Connected to |
|---|---|---|---|
| input_1 (InputLayer) | [(None, 64, 64, 3)] | 0 | [] |
| encoder (Functional) | [(None, 512), (None, 512), (None, 512)] | 13802688 | ['input_1[0][0]'] |
| sequential (Sequential) | (None, 64, 64, 3) | 13283023 | ['encoder[0][2]'] |
| tf.__operators__.add (TFOpLambda) | (None, 512) | 0 | ['encoder[0][1]'] |
| tf.math.square (TFOpLambda) | (None, 512) | 0 | ['encoder[0][0]'] |
| tf.math.subtract (TFOpLambda) | (None, 512) | 0 | ['tf.__operators__.add[0][0]', 'tf.math.square[0][0]'] |
| tf.math.exp_1 (TFOpLambda) | (None, 512) | 0 | ['encoder[0][1]'] |
| tf.math.subtract_1 (TFOpLambda) | (None, 512) | 0 | ['tf.math.subtract[0][0]', 'tf.math.exp_1[0][0]'] |
| tf.math.reduce_mean (TFOpLambda) | () | 0 | ['tf.math.subtract_1[0][0]'] |
| tf.math.multiply_1 (TFOpLambda) | () | 0 | ['tf.math.reduce_mean[0][0]'] |
| add_loss (AddLoss) | () | 0 | ['tf.math.multiply_1[0][0]'] |

Total params: 27085711 (103.32 MB)
Trainable params: 27078665 (103.30 MB)
Non-trainable params: 7046 (27.52 KB)

## IIITB-Faces:

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_3 (Dense)             (None, 1024)              525312

 batch_normalization_6 (Bat  (None, 1024)              4096
 chNormalization)

 dense_4 (Dense)             (None, 8192)              8396800

 reshape (Reshape)           (None, 8, 8, 128)         0

 conv2d_transpose (Conv2DTr  (None, 16, 16, 256)       819456
 anspose)

 batch_normalization_7 (Bat  (None, 16, 16, 256)       1024
 chNormalization)

 conv2d_transpose_1 (Conv2D  (None, 32, 32, 128)       819328
 Transpose)

 batch_normalization_8 (Bat  (None, 32, 32, 128)       512
 chNormalization)

 conv2d_transpose_2 (Conv2D  (None, 64, 64, 64)        204864
 Transpose)

 conv2d_transpose_3 (Conv2D  (None, 256, 256, 3)       4803
 Transpose)

 batch_normalization_9 (Bat  (None, 256, 256, 3)       12
 chNormalization)

=================================================================
Total params: 10776207 (41.11 MB)
Trainable params: 10773385 (41.10 MB)
Non-trainable params: 2822 (11.02 KB)
_________________________________________________________________
```

## III. Complete Model:

**In both the models ,latent dimension is 512.**

**Anime Faces:**

| Layer (type) | Output Shape | Param # |
|---|---|---|
| dense_3 (Dense) | (None, 1024) | 525312 |
| batch_normalization_6 (BatchNormalization) | (None, 1024) | 4096 |
| dense_4 (Dense) | (None, 8192) | 8396800 |
| reshape (Reshape) | (None, 4, 4, 512) | 0 |
| conv2d_transpose (Conv2DTranspose) | (None, 8, 8, 256) | 3277056 |
| batch_normalization_7 (BatchNormalization) | (None, 8, 8, 256) | 1024 |
| conv2d_transpose_1 (Conv2DTranspose) | (None, 16, 16, 128) | 819328 |
| batch_normalization_8 (BatchNormalization) | (None, 16, 16, 128) | 512 |
| conv2d_transpose_2 (Conv2DTranspose) | (None, 32, 32, 64) | 204864 |
| batch_normalization_9 (BatchNormalization) | (None, 32, 32, 64) | 256 |
| conv2d_transpose_3 (Conv2DTranspose) | (None, 64, 64, 32) | 51232 |
| batch_normalization_10 (BatchNormalization) | (None, 64, 64, 32) | 128 |
| conv2d_transpose_4 (Conv2DTranspose) | (None, 64, 64, 3) | 2403 |
| batch_normalization_11 (BatchNormalization) | (None, 64, 64, 3) | 12 |

Total params: 13283023 (50.67 MB)
Trainable params: 13280009 (50.66 MB)
Non-trainable params: 3014 (11.77 KB)

## IIITB Faces:

```
Model: "vae"
_________________________________________________________________________________________________
 Layer (type)                Output Shape              Param #   Connected to
=================================================================================================
 input_1 (InputLayer)        [(None, 256, 256, 3)]     0         []

 encoder (Functional)        [(None, 512),             1396318   ['input_1[0][0]']
                              (None, 512),             08
                              (None, 512)]

 sequential (Sequential)     (None, 256, 256, 3)       1077620   ['encoder[0][2]']
                                                       7

 tf.__operators__.add (TFOp  (None, 512)               0         ['encoder[0][1]']
 Lambda)

 tf.math.square (TFOpLambda  (None, 512)               0         ['encoder[0][0]']
 )

 tf.math.subtract (TFOpLamb  (None, 512)               0         ['tf.__operators__.add[0][0]',
 da)                                                              'tf.math.square[0][0]']

 tf.math.exp_1 (TFOpLambda)  (None, 512)               0         ['encoder[0][1]']

 tf.math.subtract_1 (TFOpLa  (None, 512)               0         ['tf.math.subtract[0][0]',
 mbda)                                                            'tf.math.exp_1[0][0]']

 tf.math.reduce_mean (TFOpL  ()                        0         ['tf.math.subtract_1[0][0]']
 ambda)

 tf.math.multiply_1 (TFOpLa  ()                        0         ['tf.math.reduce_mean[0][0]']
 mbda)

 add_loss (AddLoss)          ()                        0         ['tf.math.multiply_1[0][0]']

=================================================================================================
Total params: 150408015 (573.76 MB)
Trainable params: 150401161 (573.73 MB)
Non-trainable params: 6854 (26.77 KB)
_________________________________________________________________________________________________
```

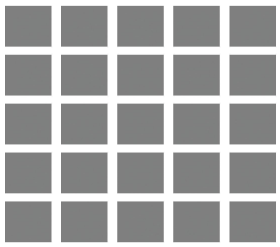
4. **Training:**
    A. **Anime Faces:**
    I. **Batch size:** 128
    II. **Loss:** MSE loss and KL divergence.
    III. **Optimizer:** Adam
    IV. **Number of Epochs:** 150
    B. **IIITB Faces:**
        I. **Batch size:** 128
        II. **Loss:** MSE loss and KL divergence.
        III. **Optimizer:** Adam
        IV. **Number of Epochs:** 1500
        But it only ran till 234 epochs, after that session crashed for all of the ram being used.

For both the training we were using google colab free T4 gpus.

5. **Output:**

While training in the beginning of each epoch and ending of each epoch we were saving random 25 images generated by the model, so that we can visualize how the model is learning.

**Visualization Of Generated Anime Faces:**

| Epoch | Beginning | Ending |
|---|---|---|
| 1 |  |  |

| | | |
|---|---|---|
| **10** |  |  |
| **30** |  |  |
| **60** |  |  |
| **100** |  |  |

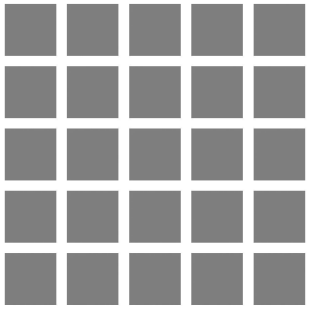
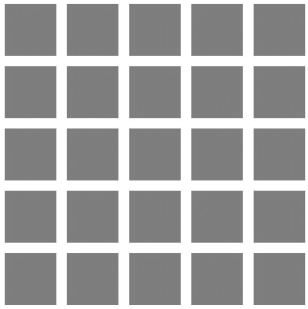
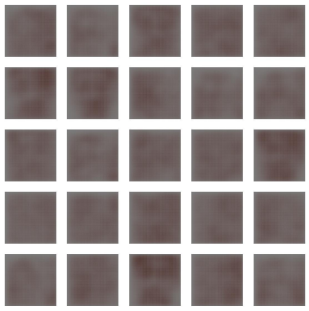
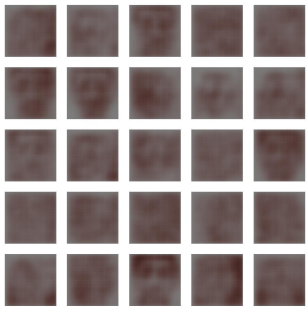
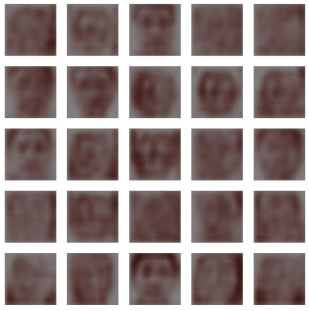
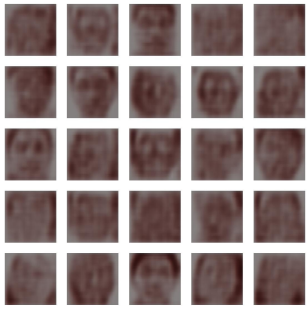
| | | |
|---|---|---|
| **150** |  |  |

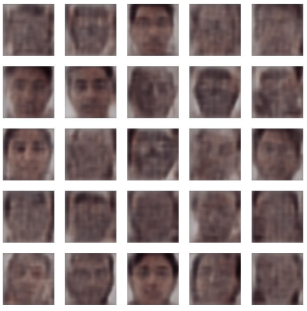
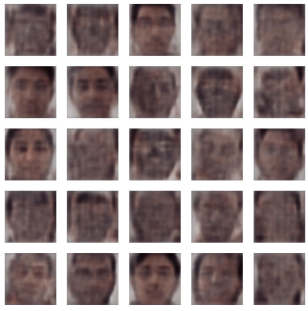
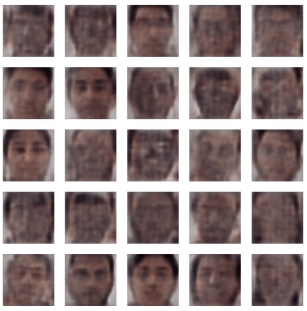
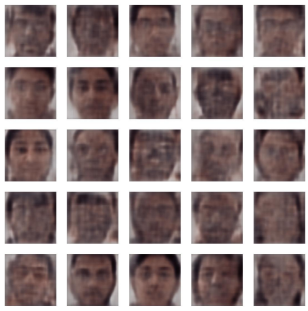
**At the beginning of the 1st epoch MSE loss: 0.09867621 ,KL loss: 0.00084340724**
**After 150 epochs MSE loss: 0.009890133 ,KL loss: 0.334876**

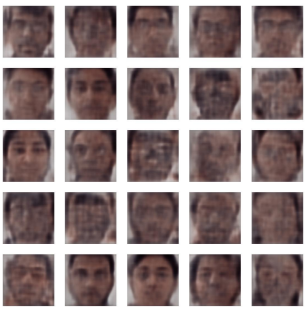
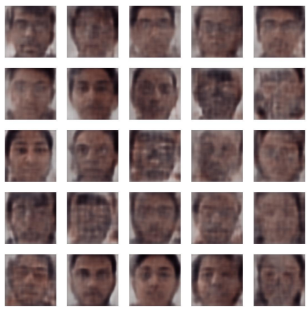
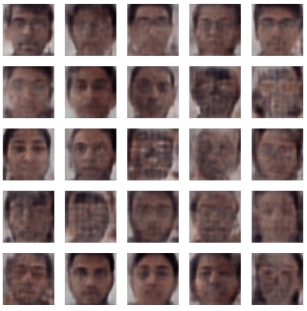
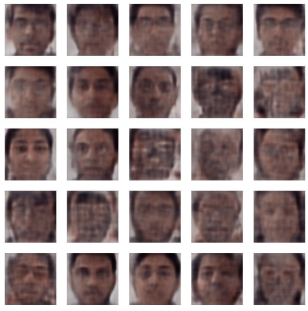
**Learning Of The Model:**

**Preview**

## Visualization Of Generated IIITB Faces:

| Epoch | Beginning | Ending |
|---|---|---|
| 1 |  |  |
| 7 |  |  |
| 20 |  |  |

| | | |
|---|---|---|
| **50** |  |  |
| **100** |  |  |
| **150** |  |  |

| | | |
|---|---|---|
| **200** |  |  |
| **234** |  |  |

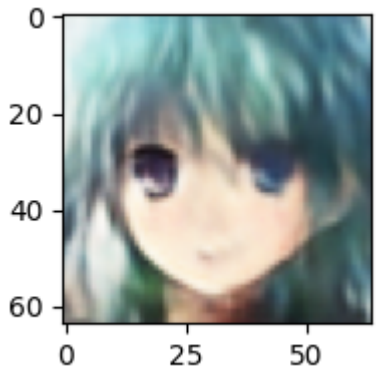
**At the beginning of the 1st epoch MSE loss: 0.08020973 - KL loss: 0.0006859753**
**After 234 epochs MSE loss: 0.0054317173 - KL loss: 0.17909229**

**Learning Of The Model:**

**Preview**

**Conclusion:**

In case of IIITB faces dataset we had less data only 832 images compared to anime dataset's 30750 images, that's why to learn the underlying distribution of the IIITB- datasets we need extensive training , so we tried with 1500 epochs but it ended after 234 epochs as ram limit got exceeded.In case of anime faces dataset only 150 epochs produced good results.After training we passed random samples of 512 dimension vector from standard normal distribution to the decoder and got the below results for anime faces.



## References:


1. https://medium.com/@birla.deepak26/autoencoders-76bb49ae6a8f
2. https://www.gabormelli.com/RKB/Convolutional_Autoencoder
3. https://www.scaler.com/topics/deep-learning/convolutional-autoencoder/
4. https://www.analyticsvidhya.com/blog/2023/07/unveiling-denoising-autoencoders/
5. https://blog.51cto.com/u_13804357/2708714
6. https://www.youtube.com/watch?v=d0W_Ab-aZGo