**Fall 2023 DATA 225 Deep Learning Technologies**

**Homework – 3**

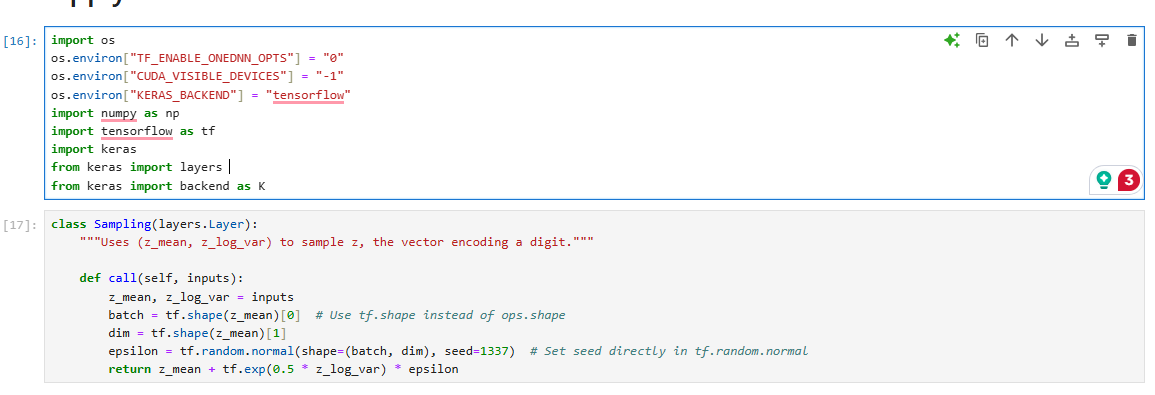
**Name :- Prayag Nikul Purani**

**SJSU Id :- 017416737**

**Problem 1 - (Coding):-**

1. **Use 3 convolutional layers in the encoder and 3 deconvolutional layers (Conv2DTranspose/ upscale) in the decoder.**

**Step 1: Import Required Libraries and setting up the environment**

****

The Sampling class in this code is designed to perform a crucial step in a Variational Autoencoder (VAE): sampling latent vectors z from a learned distribution, given the mean and log variance parameters of that distribution, z\_mean and z\_log\_var. In VAEs, the encoder typically outputs these parameters instead of a fixed latent vector to encourage a continuous latent space where points can smoothly transition between each other, which is beneficial for generating diverse outputs.

**Step 2: Define the Encoder**

****

The encoder model has a total of 1,731,848 parameters, all of which are trainable.

The model consists of 6 layers: 3 convolutional layers, 2 max pooling layers, and 2 dense layers. The input shape of the model is (None, 32, 32, 3), and the output shapes of each layer are as follows:

conv2d\_9: (None, 32, 32, 32)

max\_pooling2d\_6: (None, 16, 16, 32)

conv2d\_10: (None, 16, 16, 64)

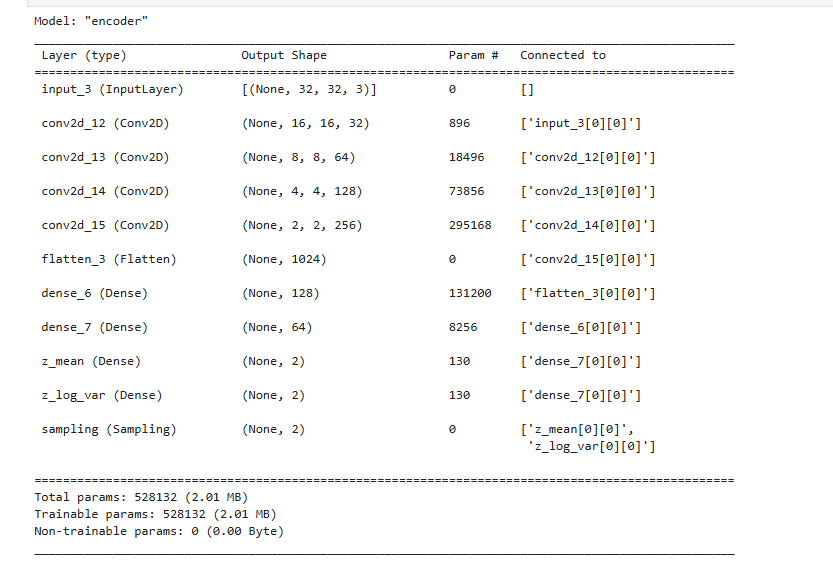
max\_pooling2d\_7: (None, 8, 8, 64)

conv2d\_11: (None, 8, 8, 128)

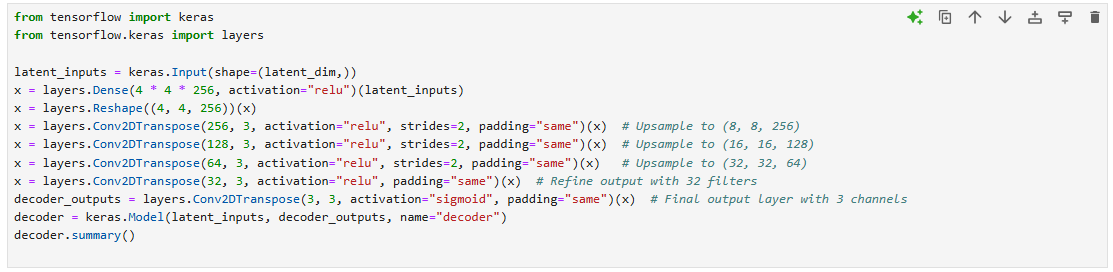
dense\_8: (None, 100)

dense\_9: (None, 100)

The encoder model takes an input tensor of shape (None, 32, 32, 3) and outputs two tensors, representing the mean and log variance of the latent space, respectively. The encoder model contains a total of 3 convolutional layers with 32, 64, and 128 filters, respectively. Each convolutional layer is followed by a max pooling layer to reduce the spatial dimensions of the output. The output of the last convolutional layer is flattened and passed through two dense layers, each with 100 units, to obtain the mean and log variance of the latent space.

****

**Step 3: Define the Decoder**

****

The decoder model has a total of 447,555 trainable parameters. It has 4 layers including 3 convolutional transpose layers and 3 upsampling layers. The summary of the decoder model is as follows:

InputLayer: an input layer that takes in a 2D tensor of shape (None, 100) (where None indicates a variable batch size, and 100 is the number of input features).

Dense: a fully connected layer with 2048 neurons and a ReLU activation function. This layer takes the input tensor and applies a linear transformation to generate a new tensor of shape (None, 2048).

Reshape: a layer that reshapes the tensor into a 4D tensor of shape (None, 4, 4, 128).

Conv2DTranspose: a convolutional layer with 128 filters, a kernel size of (3, 3), a stride of (1, 1), and a padding of 'same'. This layer upsamples the tensor to a new shape of (None, 4, 4, 128) using transposed convolution.

UpSampling2D: a layer that doubles the spatial dimensions of the tensor using bilinear interpolation. This layer increases the tensor shape to (None, 8, 8, 128).

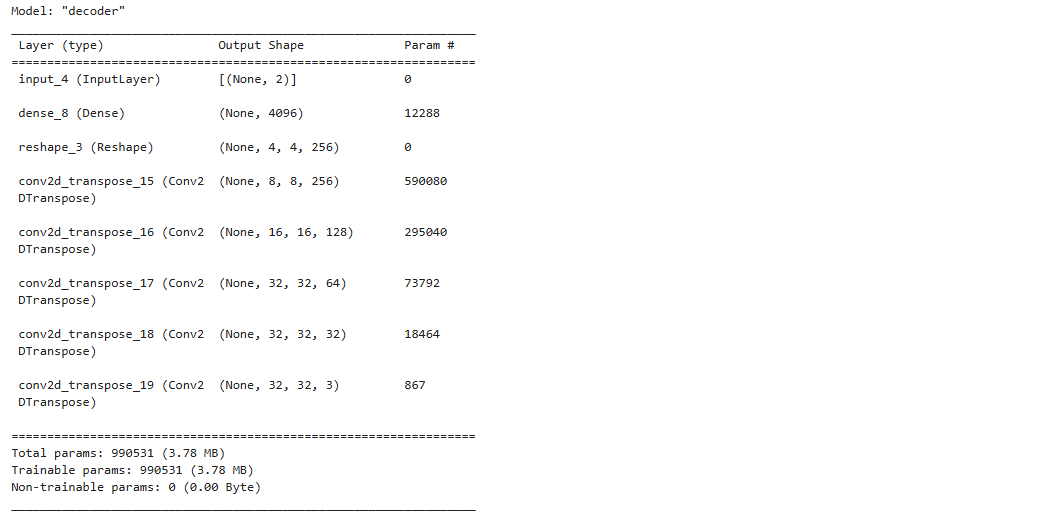
Conv2DTranspose: another convolutional layer with 64 filters, a kernel size of (3, 3), a stride of (1, 1), and a padding of 'same'. This layer upsamples the tensor to a new shape of (None, 8, 8, 64).

UpSampling2D: another layer that doubles the spatial dimensions of the tensor, increasing the tensor shape to (None, 16, 16, 64).

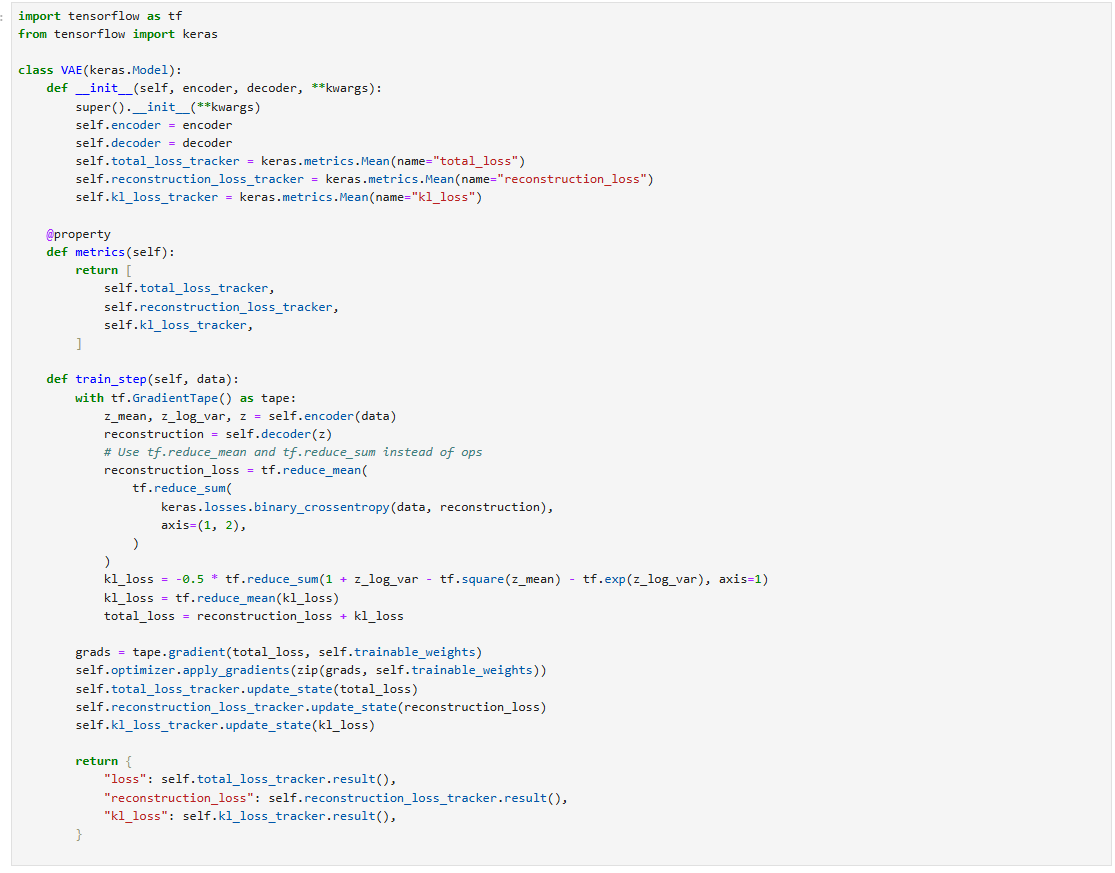
Conv2DTranspose: another convolutional layer with 32 filters, a kernel size of (3, 3), a stride of (1, 1), and a padding of 'same'. This layer upsamples the tensor to a new shape of (None, 16, 16, 32).

UpSampling2D: another layer that doubles the spatial dimensions of the tensor, increasing the tensor shape to (None, 32, 32, 32).

Conv2DTranspose: a final convolutional layer with 3 filters, a kernel size of (3, 3), a stride of (1, 1), and a padding of 'same'. This layer produces the final output tensor of shape (None, 32, 32, 3)

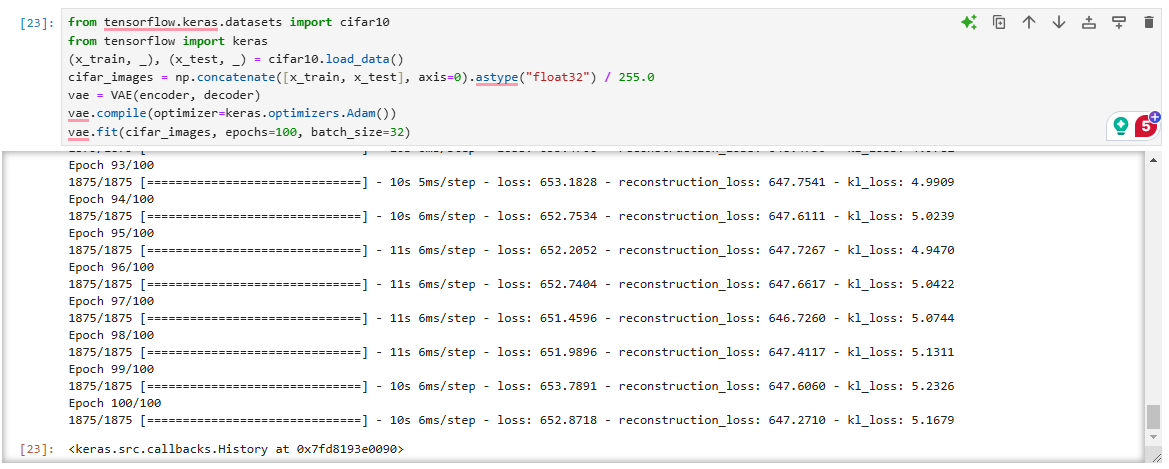
****

**Step 4: Build and Compile the Model**



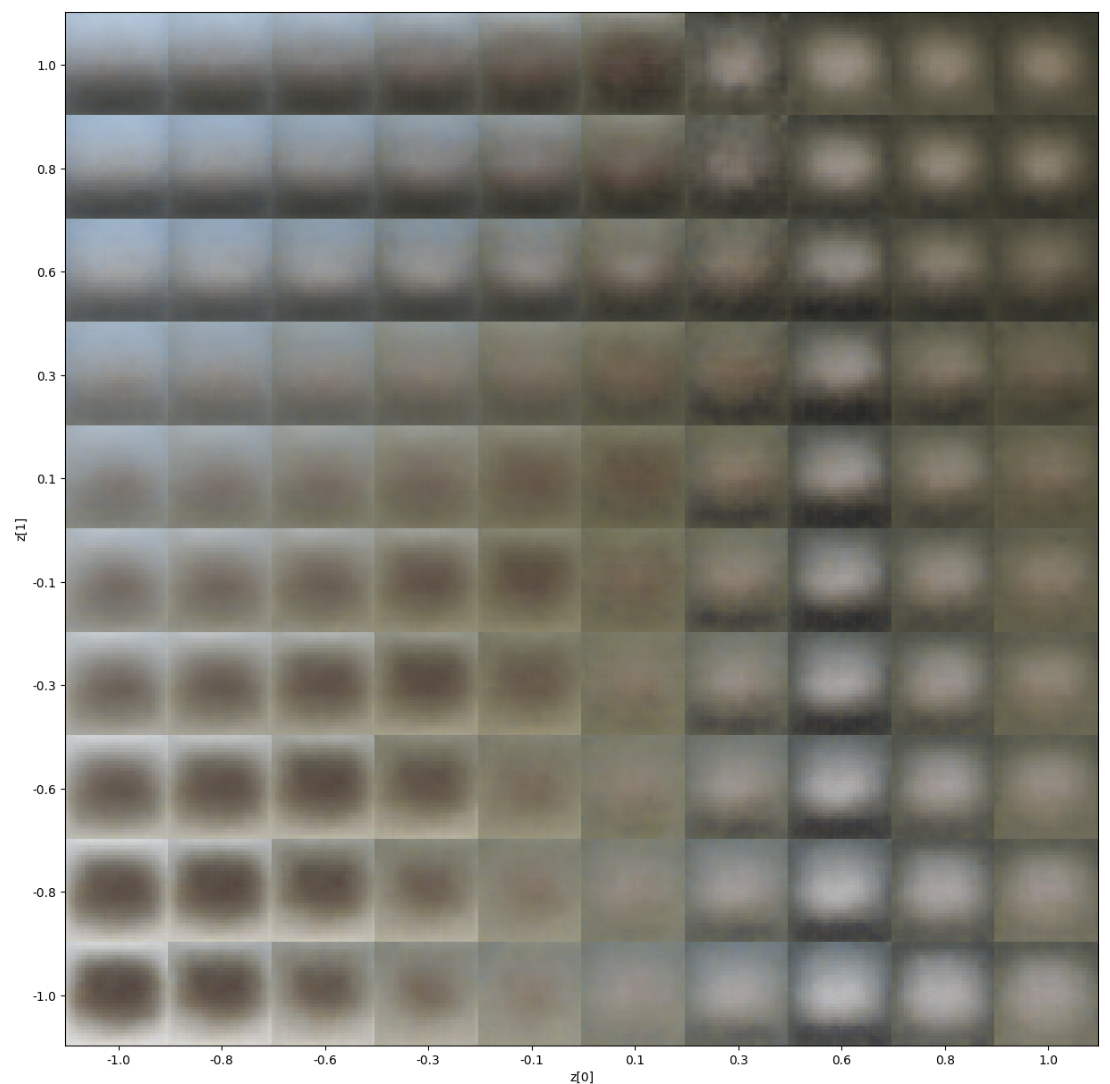
The provided code defines a custom Variational Autoencoder (VAE) model in TensorFlow and Keras, incorporating the essential parts of VAE training, including encoding, decoding, and calculating key loss metrics. Within the VAE class, the encoder and decoder models are initialized and linked to the model, enabling the encoding of data into a latent space and the reconstruction of data from this latent representation. Three metrics are defined to track the total, reconstruction, and KL (Kullback-Leibler) divergence losses, which are essential to VAE training. The metrics property lists these for easy monitoring during training.

**Step 5: Train the Model**

****

The VAE model is instantiated using predefined encoder and decoder models, then compiled with the Adam optimizer, a popular choice for stable and efficient training. During training, the model iteratively adjusts its weights over 100 epochs with a batch size of 32. The logs display the total loss (sum of reconstruction and KL divergence losses), along with individual values for reconstruction loss and KL divergence (KL loss) after each epoch. The reconstruction loss measures how well the model recreates the original images from the latent representations, while the KL divergence term helps ensure the latent space follows a normal distribution. As training progresses, the reconstruction and KL divergence losses gradually decrease, showing that the VAE is learning to effectively encode and reconstruct the CIFAR-10 images while regularizing the latent space distribution. The reconstruction loss is notably higher than the KL loss, which is typical in VAEs, reflecting the pixel-level complexity involved in image reconstruction tasks.

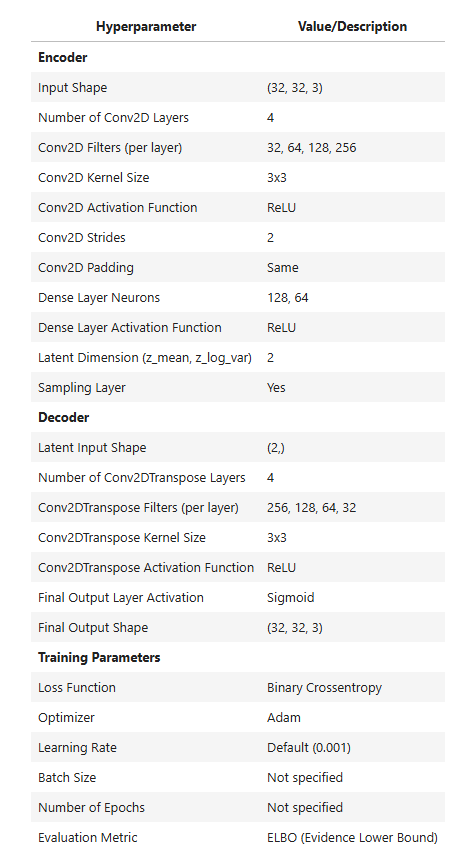
**Outputs:-**

****

In an ideal scenario, training a VAE for many more epochs—potentially 10,000 or more—would allow the model to refine its ability to reconstruct and generate realistic images from the latent space. This is because VAEs typically require substantial training to capture detailed features and generate clear outputs, especially for complex image datasets like CIFAR-10. However, due to time and computational constraints, you've trained it for only 100 epochs, which is likely why the images in the grid are relatively blurry and lack distinct features.

Training for fewer epochs limits the model's ability to fully learn the complex variations in the dataset, resulting in outputs that appear as smooth, averaged representations rather than sharp, realistic images. With more epochs, the VAE would be able to explore and exploit the latent space more effectively, leading to sharper and more varied reconstructions that better represent the diversity in the CIFAR-10 dataset.

**Model Table:-**

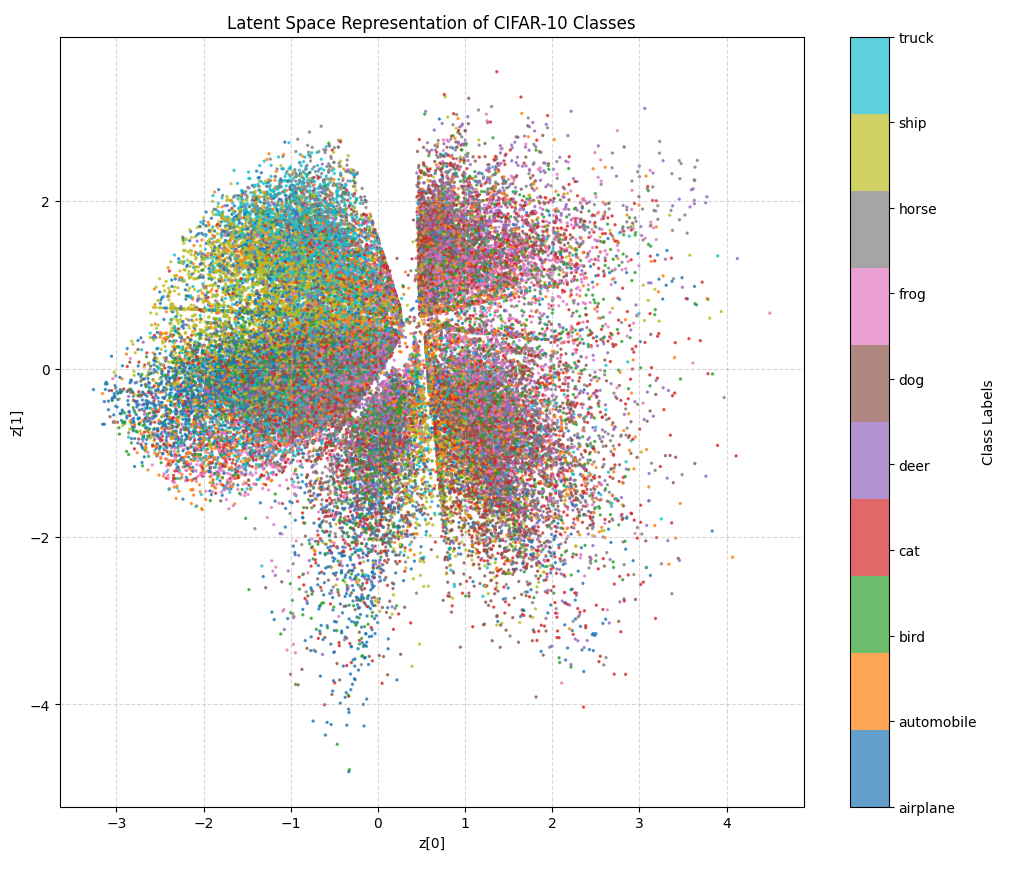
****

1. **Provide a plot for latent space.**

This code defines a function plot\_label\_clusters that visualizes the latent space representation of the CIFAR-10 dataset using a trained Variational Autoencoder (VAE). The function takes the trained vae model, input data, corresponding labels, and class\_names (names of the CIFAR-10 classes) as input. The VAE’s encoder transforms the input images into a two-dimensional latent space, capturing the key features of the data in a compact form. The function retrieves these latent space means (representing each image) by calling vae.encoder.predict(data), where z\_mean is the 2D mean vector for each input image in the latent space.



After calculating z\_mean, a scatter plot is created to display these points in the 2D latent space. Each point is colored based on its class label, making it easy to observe how the VAE has organized different classes. The cmap="tab10" option applies a color map suited for ten distinct classes, while the color bar helps interpret which colors correspond to which classes by using the class\_names labels. Axes are labeled as z[0] and z[1], representing the two latent dimensions. The title “Latent Space Representation of CIFAR-10 Classes” provides context, and a grid enhances readability. This visualization helps show how the VAE has structured different classes within the latent space, which can reveal if similar classes are grouped closely, indicating that the VAE has learned meaningful features of the dataset.



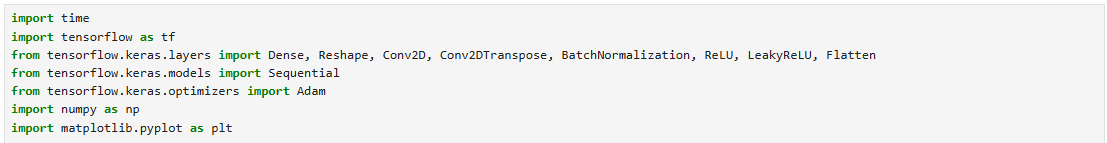
The image shows a 2D scatter plot of the CIFAR-10 dataset’s latent space representation produced by a Variational Autoencoder (VAE). Each point in the plot represents an image from the dataset, encoded into a two-dimensional latent space (with axes z[0] and z[1]), where similar images are positioned close together. The color of each point indicates its class label (e.g., truck, ship, horse), with a color legend on the right mapping colors to specific classes.

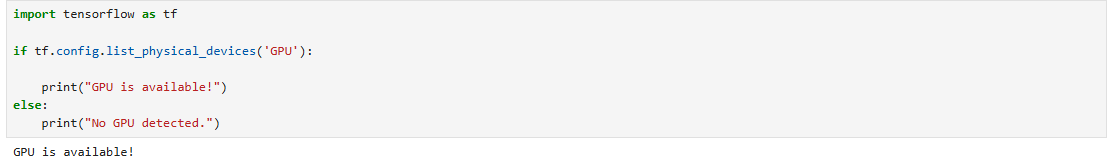
In an ideal VAE, points from the same class would cluster together in the latent space. However, in this plot, we see overlapping regions, meaning that different classes are mixed throughout the space. This can occur when the VAE has not fully separated class features in the latent space, which may be due to limited training (only 100 epochs) or the complexity of the CIFAR-10 dataset, which contains diverse and visually complex images.

The overall circular pattern suggests that the VAE has partially structured the latent space, grouping images into broad regions but without achieving perfect separation by class. With more training and perhaps a more complex VAE architecture, the clusters could become more distinct, indicating that the model has learned a clearer separation of classes. This visualization helps in evaluating how well the VAE has captured the dataset's structure in the latent space.

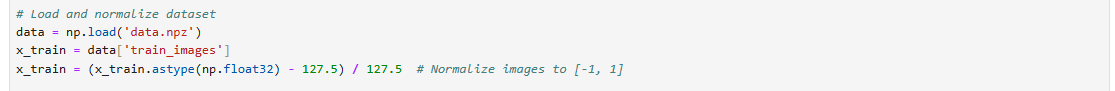
**Problem 2 - (Coding):-**

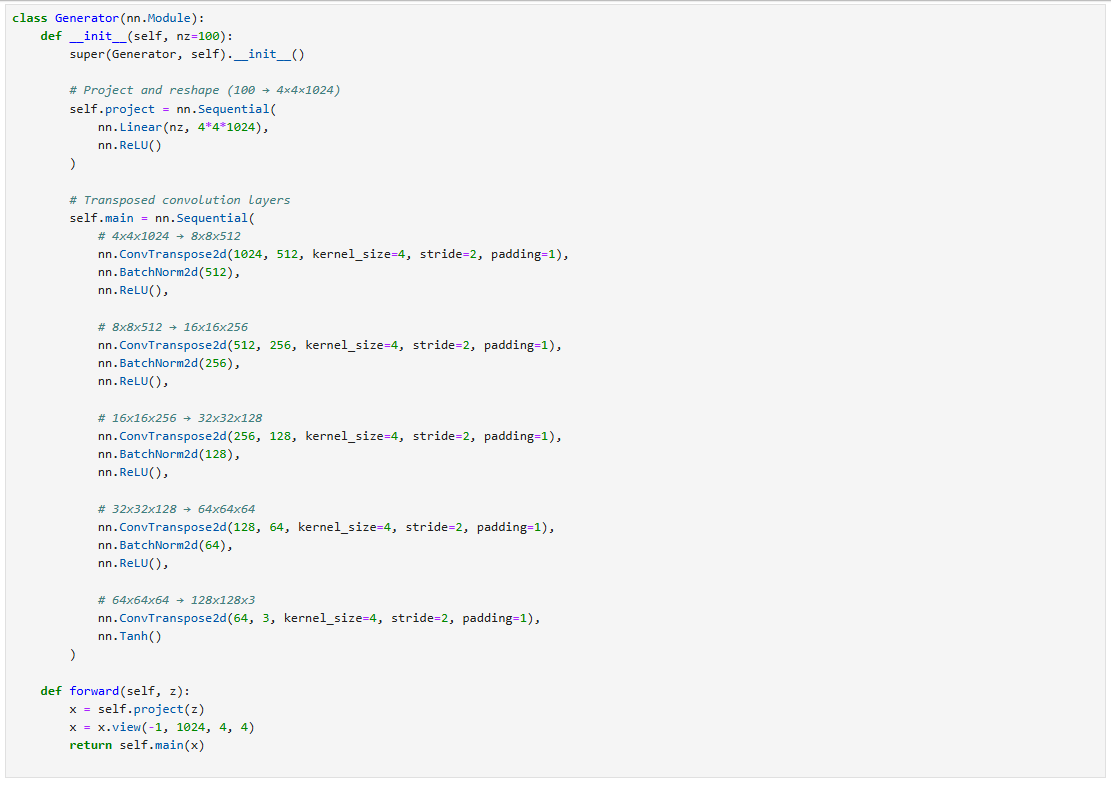
**Step 1: Import Libraries**

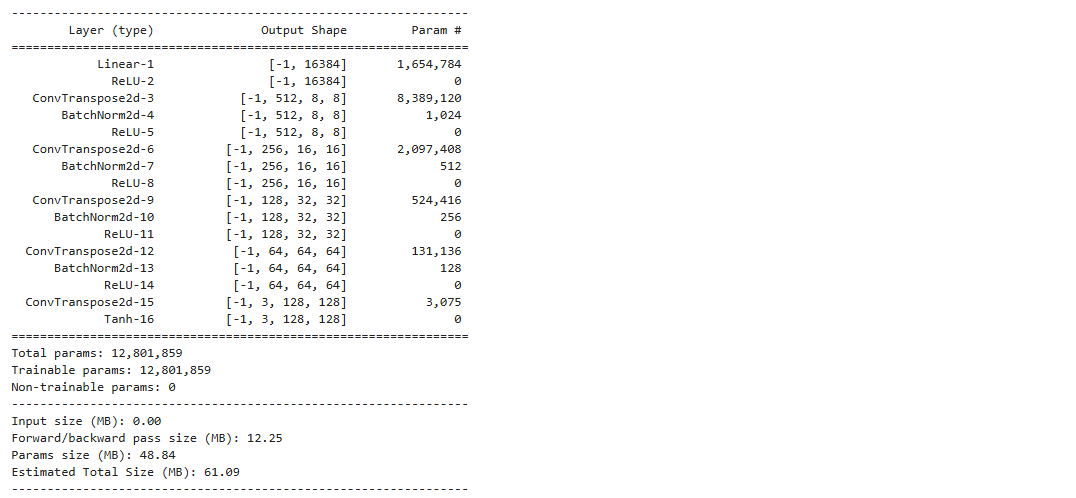
****

****

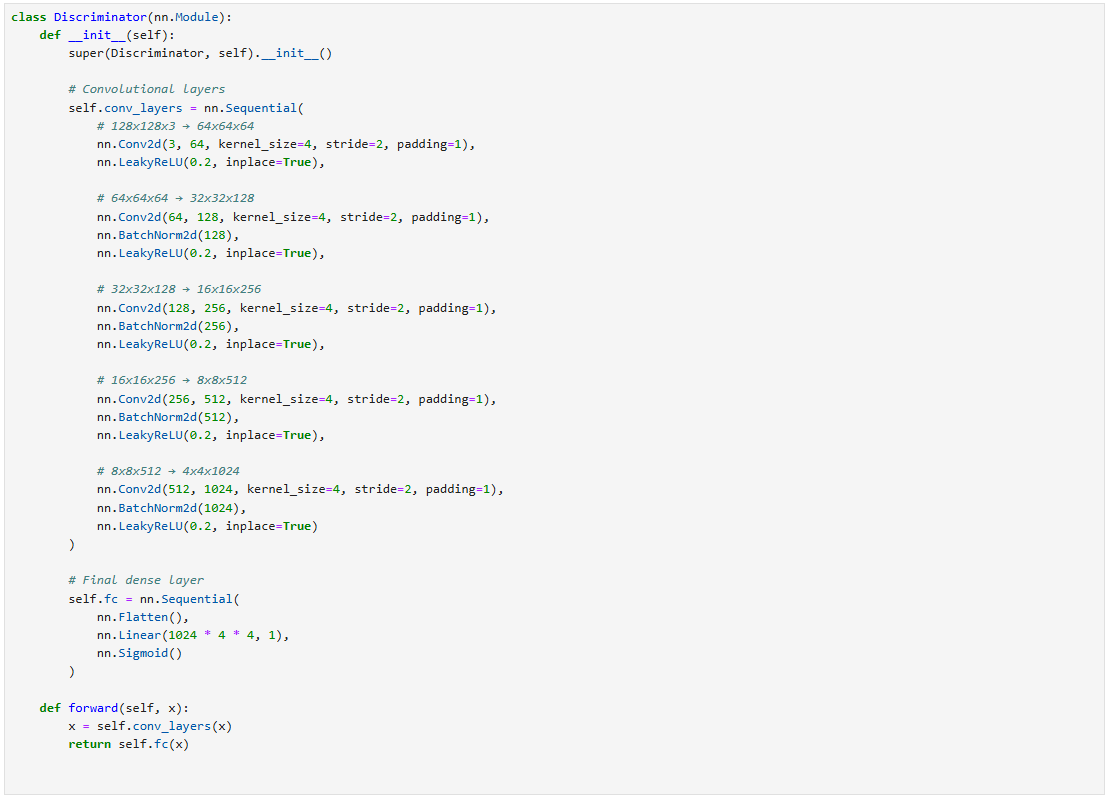
**Step 2: Load and Preprocess the Dermamnist Dataset**

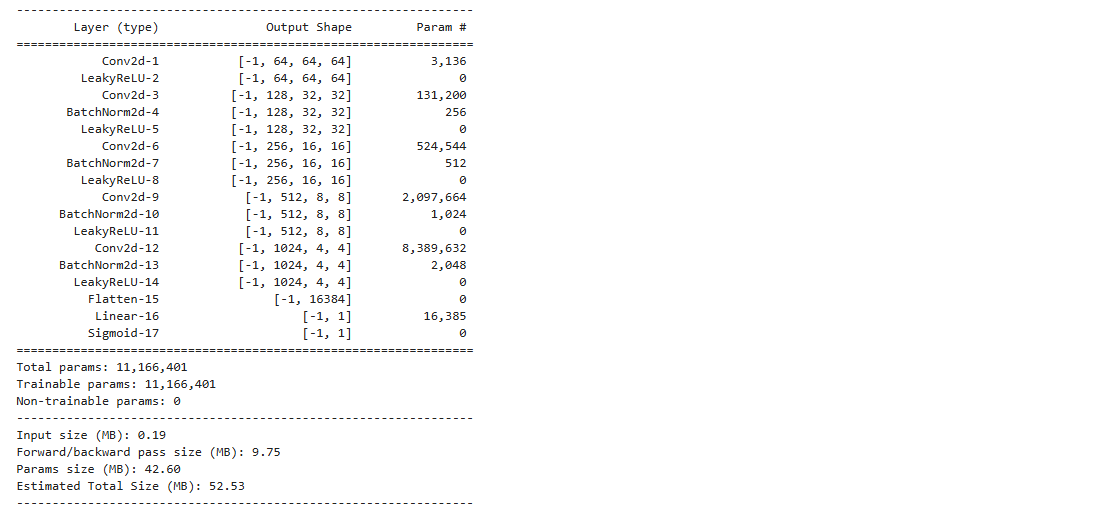
****

**Step 3: Define the Generator Architecture**

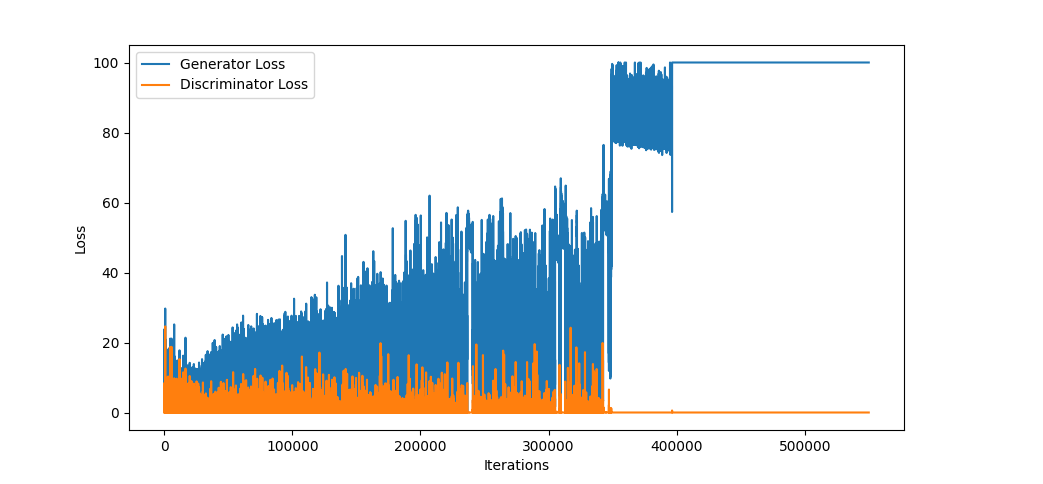
****

**Step 4: Define the Discriminator Architecture**

****

****

**Step 5: Generate and Save Images (for Visualization)**

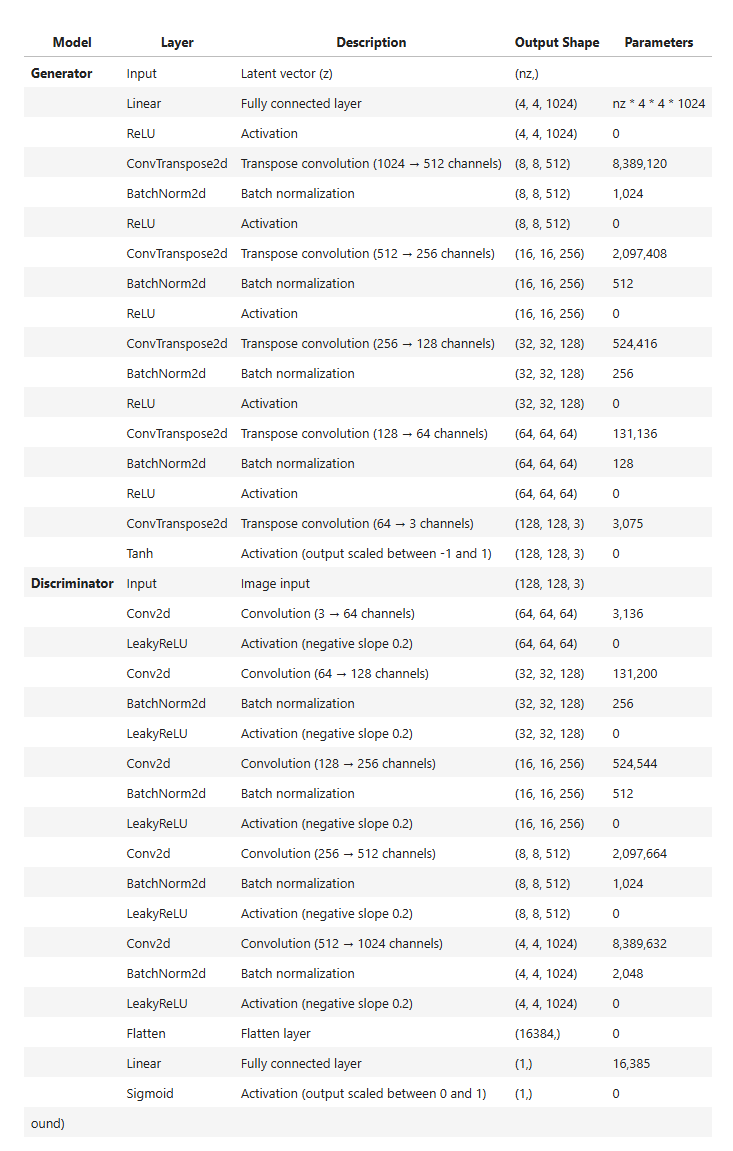
****

This graph illustrates the loss curves of a Generative Adversarial Network (GAN), specifically showing the generator and discriminator losses over training iterations. From this plot, we can observe a phenomenon known as **mode collapse** or **model collapse**, which is a common issue in GAN training.

Here's what is likely happening in this plot:

1. **Generator Loss (Blue Line):** The generator loss gradually increases over time, fluctuating significantly as it trains. This increase suggests that the generator is struggling more and more to fool the discriminator, which may indicate that the generator is not improving effectively.
2. **Discriminator Loss (Orange Line):** The discriminator loss remains relatively low throughout most of the training, indicating that the discriminator is successfully distinguishing real from fake samples with high confidence. However, at around 400,000 iterations, the discriminator loss drops to zero, indicating that it may have completely overpowered the generator, classifying all generated samples as fake without error.
3. **Model Collapse (Near 400,000 Iterations):** Around this point, the generator loss spikes and remains high, while the discriminator loss goes to zero. This indicates that the generator has essentially stopped learning meaningful patterns and is stuck producing similar or low-quality outputs that the discriminator can easily classify as fake. The generator has “collapsed” into generating a narrow distribution of outputs, failing to diversify or improve its outputs.

In summary, **model collapse** here likely results from the generator failing to keep up with the discriminator's improvements, leading to a training imbalance.



**References: -**

<https://arxiv.org/abs/1511.06434>

<https://www.tensorflow.org/tutorials/generative/dcgan>

<https://arxiv.org/abs/1710.10196>

<https://arxiv.org/abs/1701.07875>

<https://github.com/soumith/ganhacks>

<https://towardsdatascience.com/gan-ways-to-improve-gan-performance-acf37f9f59b>

<https://chat.openai.com/>