Vector Quantization VAE

(DL ASSIGNMENT - 5)

Under the guidance of,
Prof. Deepak Mishra

Submitted by,

Abhishek Rai (M23CSE002)
Swati Shrivastava (M23CSE027)
Amresh Kumar(M23CSA004)

Department of Computer Science and Engineering
IIT Jodhpur

May 1, 2024

Introduction:

Vector Quantized Variational Autoencoder (VQ-VAE) is a generative model that combines the variational autoencoder (VAE) framework with vector quantization techniques. It uses a discrete latent space and a learnable codebook to represent input data, enabling more efficient and structured representation learning.
PixelRCNN (Autoregressive Model) is a type of autoregressive generative model used for image generation. It models the conditional probability distribution of each pixel given the values of its predecessors in the image. PixelRCNNs use convolutional neural networks with masked convolutions to ensure causality, allowing the model to generate images pixel by pixel. This approach produces sharp and realistic samples, making PixelRCNNs suitable for tasks like image generation and inpainting.

In this assignment, our objective is to develop and apply advanced generative modeling techniques.

We have implemented and evaluated two primary tasks:

Training a Vector-Quantized Variational Autoencoder (VQ-VAE) to efficiently encode and decode high-dimensional skin lesion images while capturing meaningful latent representations.
Training an Auto Regressive Model to generate new, realistic skin lesion images based on the learned latent space representations.
We undertake a series of experiments to accomplish these objectives. Initially, the input data undergo preprocessing, including normalization and various data transformations, to enhance model performance. Subsequently, we design and train a VQ-VAE network using convolutional neural network (CNN) encoder and decoder modules to learn a quantized latent space using a codebook. Following this, an autoregressive model, PixelRCNN, is designed and trained to generate diverse, realistic skin lesion images using the codebook and decoder trained during the first phase.

Dataset:

The dataset utilized for this assignment is the ISIC dataset, comprising training and test images of skin lesions. The dataset has dermatology images, specifically, the dataset contains images of various skin conditions such as melanoma (MEL), nevus (NV), basal cell carcinoma (BCC), actinic keratosis (AKIEC), benign keratosis-like lesions (BKL), dermatofibroma (DF), and vascular lesions (VASC). Several operations are performed on this dataset to prepare it for model training and evaluation.

Preprocessing the dataset:

Four transformations have been added other than normalization and converting to tensor and the reasons to do so are mentioned below:

1. **Resize: transforms.Resize((256, 256))**
   U-Net models typically require fixed-size input images for efficient processing. Resizing the images to (256, 256) ensures compatibility with the U-Net architecture.

2. **RandomRotation: transforms.RandomRotation(degrees=30)**
   The choice of 30 degrees for the rotation angle in the transformations is somewhat arbitrary and depends on the specific characteristics of the dataset.
   It provides Balance between Variation and Information Preservation.
   A rotation angle of around 30 degrees is commonly used as a default or starting point
   Random rotation introduces variability in the dataset, enabling the model to learn features from different perspectives. This augmentation helps enhance the model's ability to generalize across various orientations of lesions or anatomical structures.

3. **RandomHorizontalFlip: transforms.RandomHorizontalFlip(p=0.5)**
   The probability of horizontal flipping being applied is set to 50%, meaning that each image has an equal chance of being horizontally flipped or not.
   By flipping only half of the images, we ensure that the original orientation of the lesions is preserved in a significant portion of the dataset. Using a probability of 50% for horizontal flipping is a common practice in image augmentation pipelines.
   Horizontal flipping provides additional augmentation by presenting mirror images of lesions. This transformation helps the model learn features invariant to left-right orientation, thereby improving its robustness.

4. **RandomVerticalFlip: transforms.RandomVerticalFlip(p=0.5)**
   This randomly flips the images vertically with a probability of 0.5.
   Same reason listed above as horizontal flipping. Vertical flipping introduces further variability into the dataset, allowing the model to learn features from different orientations, including vertically oriented structures. This augmentation enhances the model's ability to generalize across various lesion orientations and anatomical structures, improving its performance in diverse scenarios.

The other that were mentioned are :

5. **Normalize: transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])**
   Reason: Normalization helps in stabilizing and accelerating the training process by scaling the pixel values to a standardized range. The chosen mean and standard deviation values are commonly used in pre-trained models trained on ImageNet data and have been found effective for various computer vision tasks.

6. **ToTensor: transforms.ToTensor()**
   Reason: Conversion to tensors is essential as deep learning models, including U-Net, operate on tensor inputs. This transformation enables seamless integration of the images with PyTorch.

Methodology:

The **VQ-VAE** model architecture consists of an Encoder, a Decoder, and a Vector Quantization (VQ) layer.

- The Encoder comprises convolutional layers to extract features from input images.
- The Decoder consists of transposed convolutional layers to reconstruct images from the quantized latent space.
- The VQ layer quantizes the latent space and maintains a codebook for encoding and decoding.

The VQ-VAE model is trained using the custom training loop. Adam optimizer is employed in it for parameter optimization, and mean squared error (MSE) loss is utilized as the optimization criterion with a learning rate of 0.001 and the batch size is set to 16 for data loader. During training, reconstructed images are periodically visualized to monitor the model's progress. After training model weights are saved for future usage.

PixelRCNN model is defined for autoregressive image generation, utilizing residual masked convolutional layers.
The PixelRCNN model is trained using the encoded dataset. Here also Adam optimizer is employed for optimization, MSE loss is utilized as the training criterion with a learning rate of 0.001, and the batch size is set to 16 for the data loader.

Process the entire training dataset via the encoder
Function encode_dataset(encoder, data_loader): This function encodes the entire training dataset using the provided encoder. It iterates through the data_loader, which contains batches of training data.

- For each batch, the data is moved to the GPU. The encoder is applied to the data to obtain the encoded representations (encoded_images).
- The encoded representations are then passed through the VQ layer to quantize them (q) and obtain the corresponding indices (indices).
- The quantized representations (q) are appended to the encoded_data list.

Creating a new dataset with code indices:
Class EncodedDataset(torch.utils.data.Dataset): This class defines a custom dataset containing the encoded representations (encoded_data) obtained previously.

- It implements the __len__ method to return the total number of samples in the dataset.
- It implements the __getitem__ method to retrieve a sample at the specified index.

After encoding the entire dataset and obtaining the quantized representations (encoded_data), an instance of EncodedDataset is created using this data.
The encoded_loader is then created using DataLoader, which allows iterating over batches of samples from the encoded dataset. Each batch size is set to 16, and the data is shuffled.
These steps collectively encode the entire training dataset, quantize the encoded representations, create a custom dataset containing the quantized representations, and set up a data loader to iterate over batches of encoded data during training.

Finally, the training process is carried out for 100 epochs, with periodic updates and loss monitoring.

Implementation:

Model Architecture:

1) **Encoder:**
   The encoder processes input images to generate a set of feature maps, which are subsequently used by the decoder.
   It consists of 5 convolutional layers that progressively reduce the spatial dimensions of the input image while increasing the number of channels.

   Specifications:
   Conv1: Kernel size: 4x4, Stride: 2, Padding: 1, In channels: 3 (RGB), Out channels: 32
   Conv2: Kernel size: 4x4, Stride: 2, Padding: 1, In channels: 32, Out channels: 64
   Conv3: Kernel size: 4x4, Stride: 2, Padding: 1, In channels: 64, Out channels: 128
   Conv4: Kernel size: 4x4, Stride: 2, Padding: 1, In channels: 128, Out channels: 256
   Conv5: Kernel size: 4x4, Stride: 2, Padding: 1, In channels: 256, Out channels: 256

Batch normalization layers stabilize and accelerate the training process by normalizing the activations of each layer. Batch normalization is applied after the 1st four convolutional layers to improve the robustness and efficiency of training.

Activation Functions:
ReLU (Rectified Linear Unit) activation function (F.relu) introduces non-linearity into the network by setting negative values to zero. ReLU activation is applied after each batch normalization and 5th conv layer to enable the encoder to learn complex features and representations from input images.

Forward Pass:

- The input images are sequentially passed through the encoder layers:
- Convolution and Down-sampling: Each convolutional layer reduces the spatial dimensions of the input feature maps while increasing the number of channels.
- Batch Normalization and Activation: Batch normalization and ReLU activation are applied after each convolutional layer to stabilize activations and introduce non-linearity.
- Feature Extraction: The final convolutional layer extracts high-level features from the input images, producing the final set of feature maps.

2) Decoder:

The decoder receives a set of feature maps as input, typically generated by an encoder.

Transpose convolutional layers, also known as deconvolutional layers, upsample the input feature maps to reconstruct the original spatial dimensions of the input data. Each Transpose conv 2d layer learns to map spatially larger representations to smaller ones.

Specifications:
Deconv1: Kernel size: 4x4, Stride: 2, Padding: 1, In channels: 256, Out channels: 256
Deconv2: Kernel size: 4x4, Stride: 2, Padding: 1, In channels: 256, Out channels: 128
Deconv3: Kernel size: 4x4, Stride: 2, Padding: 1, In channels: 128, Out channels: 64
Deconv4: Kernel size: 4x4, Stride: 2, Padding: 1, In channels: 64, Out channels: 32

Batch normalization layers stabilize and accelerate the training process by normalizing the activations of each layer. Batch normalization is applied after the 1st four transpose convolutional layers to improve the robustness and efficiency of training.

Activation Functions:

ReLU (Rectified Linear Unit) activation function introduces non-linearity into the network by setting negative values to zero.
ReLU activation is applied after each batch normalization layer to enable the decoder to learn complex mappings from input features to output pixel values.

Output Layer:
The final layer (deconv5) applies a transpose convolution operation to produce the output image.
A sigmoid activation function is applied to ensure that the pixel values of the output image are within the range [0, 1].

Forward Pass:

- The input features (encoded representations) are sequentially passed through the decoder layers:
- Upsampling: Each transpose convolutional layer up samples the input features, gradually increasing the spatial dimensions.
- Batch Normalization and Activation: Batch normalization and ReLU activation are applied after each transpose convolutional layer to stabilize activations and introduce non-linearity.
- Output Reconstruction: The final transpose convolutional layer produces the reconstructed output image.
- Sigmoid Activation: A sigmoid activation function is applied to ensure that pixel values are bounded within the valid range [0, 1], producing a valid image.

3) Vector Quantization:

VQ Layer: The Vector Quantization (VQ) layer quantizes the continuous latent space into discrete codes by mapping input vectors to the nearest embedding vectors.
It consists of an embedding matrix initialized with random values and learns to update these embeddings during training to minimize the quantization error.
Specifications:

Embedding Layer :
The embedding layer initializes the embedding matrix with random values.
The size of the embedding matrix is determined by the number of embeddings (num_embeddings=512) and the embedding dimension (embedding_dim=64).
The embedding matrix is updated during training to minimize the quantization error and improve codebook accuracy.

Forward Pass:
The forward pass of the VQ layer involves following steps:

- Flatten Input: The input tensors are flattened to create 2D representations for computing distances.
- Compute Distances: Distances between input vectors and embedding vectors are calculated using mean squared error (MSE).
- Find Closest Embeddings: The index of the nearest embedding vector for each input vector is determined.
- Quantize and Unflatten: The input vectors are quantized by replacing them with their corresponding embedding vectors. The quantized vectors are then unflattened to restore their original shape.
- Loss Calculation: The quantization loss (difference between quantized and original vectors) and the commitment loss (distance between original and quantized vectors) are computed.
- Total Loss: The total loss is calculated as the sum of quantization loss and commitment loss.

4) **VQ-VAE Model:**
The VQ-VAE model combines an encoder, a decoder, and the VQ layer to perform image compression and reconstruction.

VQ Layer:
The VQ layer quantizes the latent space representation generated by the encoder.
It replaces continuous latent vectors with discrete codes from the codebook, reducing the dimensionality of the latent space.
The VQ layer also computes the quantization and commitment losses, which are used to train the model.

Training Parameters:
num_embeddings: Number of embeddings in the codebook.
embedding_dim: Dimensionality of each embedding vector.
commitment_cost: Weight parameter for the commitment loss, controlling the trade-off between reconstruction fidelity and codebook accuracy.

Number of Parameters = 3545155

Forward Pass:
- The forward pass of the VQ-VAE model involves encoding input images, quantizing the latent space representation, reconstructing images from the quantized representation, and computing the total loss.
- The reconstruction loss measures the difference between the reconstructed images and the original input images.

- The total loss is the sum of the reconstruction loss and the quantization loss.
- Initialization: The VQ-VAE model is initialized with the 512 embeddings, with embedding dimension=64, and commitment cost=0.25.

Training loop:

- The function iterates over the specified number of epochs (num_epochs).
- Inside each epoch loop, the model is set to training mode using model.train().
- The total_loss variable is initialized to track the cumulative loss for each epoch.
- The training data is iterated over in batches using the train_loader.
- Each batch of data is moved to the GPU for computation if available.
- The optimizer gradients are zeroed using optimizer.zero_grad() to prevent accumulation from previous iterations.
- The model is called with the batch of data to obtain the reconstructed batch and the loss.
- The loss is backpropagated through the model's parameters using loss.backward().
- The optimizer takes a step based on the gradients to update the model's parameters (optimizer.step()).
- The batch loss is accumulated in the total_loss variable.

5) MaskedConvolution and Residual MaskedConvolution :

MaskedConv2d is a convolutional layer that extends nn.Conv2d to incorporate masks. It inherits from nn.Conv2d, inheriting its functionality and attributes.

- The mask is registered as a buffer using register_buffer. This mask is multiplied element-wise with the weights of the convolutional layer.
- The mask is designed to enforce certain constraints on the convolutional operation, enabling the network to learn specific patterns or structures in the data.
- The forward method applies the mask to the weights before performing the convolution operation, ensuring that only certain weights are used for computation.

ResidualMaskedConv2d is a module that implements residual connections between MaskedConv2d layers. It inherits from nn.Module.

- It defines a sequence of MaskedConv2d layers each followed by a ReLU activation function.
- The purpose of using residual connections is to facilitate the training of deeper networks by mitigating the vanishing gradient problem.
- The forward method executes the forward pass through the defined sequence of layers and adds the input to the output using a residual connection.

This architecture allows the model to learn both the original features and the residual features simultaneously, improving the flow of gradients during training. ResidualMaskedConv2d utilizes MaskedConv2d layers within its sequential network architecture. The MaskedConv2d layers within ResidualMaskedConv2d are responsible for learning specific patterns or structures in the data while enforcing constraints through the applied masks. By using residual connections between MaskedConv2d layers, ResidualMaskedConv2d enables the model to learn complex representations effectively and facilitates the training of deeper networks. Together, these components form a building block that can be incorporated into larger neural network architecture, PixelRNNs, to capture dependencies between image pixels effectively while ensuring efficient training.

6) **PixelCNN:**

Initialization (__init__) Method:
It takes three arguments: in_channels, out_channels, and conv_filters. These parameters define the number of input channels, output channels, and convolutional filters used in the network, respectively.
Inside the initialization method:

The layers of the network are defined sequentially:
The first layer is a masked convolutional layer (MaskedConv2d) with mask type='A', kernel size=7, and padding= 3. This layer is followed by batch normalization and ReLU activation.
Then, there are 8residual masked convolutional blocks (ResidualMaskedConv2d). Each block consists of a masked convolutional layer, batch normalization, and ReLU activation.
Finally, there's another masked convolutional layer with mask type='B' and a kernel size=1.

Forward Method:
This method defines the forward pass of the network.
It takes input x and passes it through the layers defined in self.net. The output of the last layer is returned.

Model Initialization: After defining the PixelRCNN class, an instance of this class is created, specifying the number of input channels (no_channels=256), the number of output channels (out_channels=256), and the number of convolution filters (convolution_filters=120). This instance is then moved to the GPU for computation.

This PixelRCNN class defines a deep neural network architecture with masked convolutional layers and residual connections, suitable for generating images pixel by pixel.

Training loop:
Training Function (train_pixelrcnn):
n_epochs = 100: Number of training epochs.
lr = 0.001: Learning rate for the Adam optimizer.

- The training loop iterates over each epoch and each batch in the training data loader:
- Computes the output of the PixelRCNN model.
- Calculates the MSE loss between the output and the input batch. Adam optimizer is used.
- Performs backpropagation and optimization to update the model parameters.
- Appends the loss to the train_losses list.
- Prints the progress after each epoch.

Generating Samples:

1. Sampling Pixel Values: The code iterates over each pixel in the 8x8 grid and samples pixel values for each pixel independently:
2. Computes the logits (unnormalized probabilities) for each pixel using the PixelRCNN model.
3. Computes the probabilities from the logits for the pixel at position (i, j).
4. Samples a new index (pixel value) from the probabilities using multinomial sampling.
5. The sampled index is placed in the correct location in the sampled tensor.

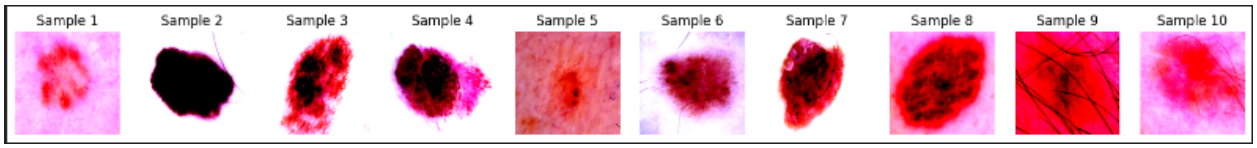
Quantization:
After sampling pixel values for each pixel independently, the code quantizes the samples using the VQ-VAE model:

1. Rounds the sampled pixel values and converts them to integers.
2. Reshapes the sample tensor into a 1D tensor of encoding indices having dim=4096x1
3. Initializes a tensor to hold one-hot encodings of the indices. Computes the one-hot encodings based on the encoding indices.

4. Reconstructs the quantized samples by multiplying the one-hot encodings with the embedding matrix of the VQ-VAE model and reshaping them to the original shape(16,256,8,8).
5. Decoding: Decodes the quantized samples using the decoder of the VQ-VAE model to generate images.

Results and Analysis:

Visualization of samples from the dataset:



Every 100 batches, the average loss for the last 100 batches is printed to monitor training progress.
The total_loss is reset to zero after printing.

Reconstruction Visualization:
After each epoch, reconstructed images are visualized to monitor the quality of reconstruction.
The model is switched to evaluation mode using model.eval() to disable dropout and batch normalization layers.
Five batches of data are randomly selected from the training loader using next(iter(train_loader)) for visualization.

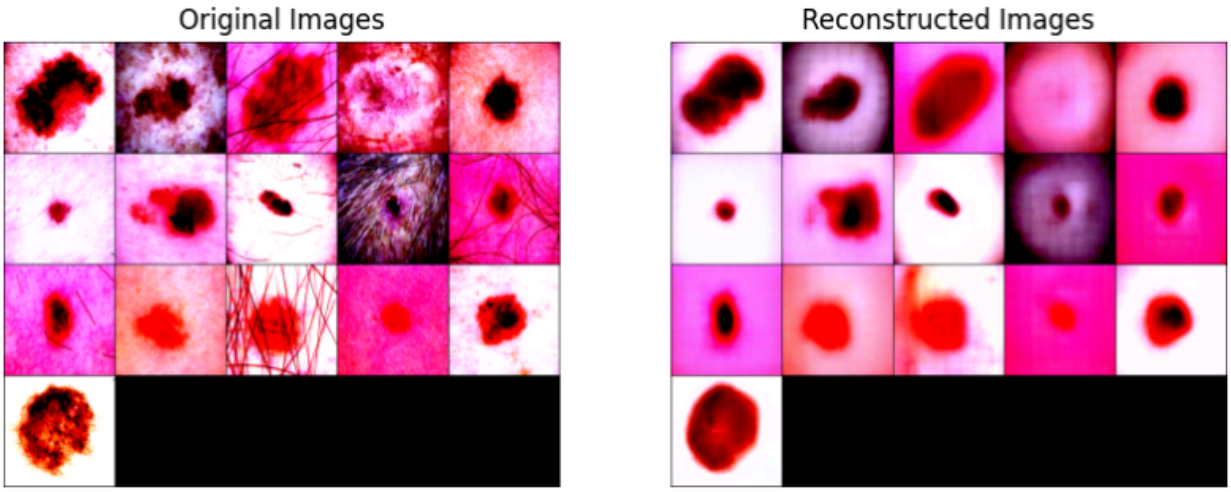
The model is called with the batch of data to obtain the reconstructed batch.
The training loop performs forward and backward passes through the model, updates the model parameters based on the computed gradients, and provides visual feedback on the reconstruction quality during training.

Losses and accuracies are also plotted on Wandb for better visualizations:

VQ-VAE:

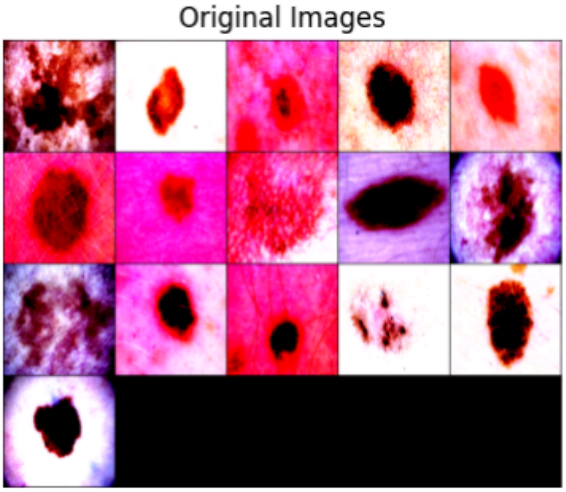
Here original and reconstructed images are plotted side by side for comparison:

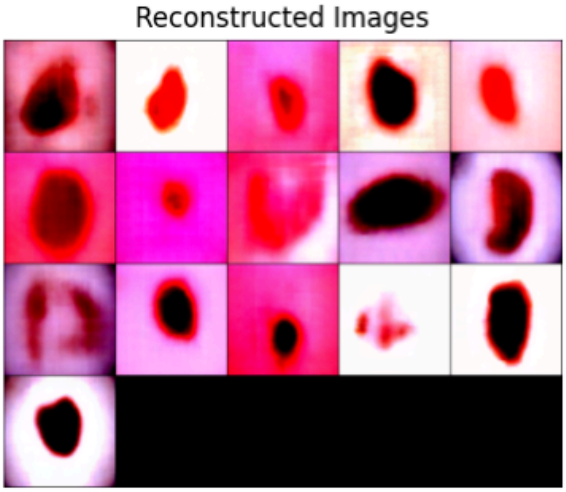
Output at final epochs:
Epoch 18, Batch 100/564, Loss: 0.413299
Epoch 18, Batch 200/564, Loss: 0.418819
Epoch 18, Batch 300/564, Loss: 0.430923
Epoch 18, Batch 400/564, Loss: 0.398370
Epoch 18, Batch 500/564, Loss: 0.420093



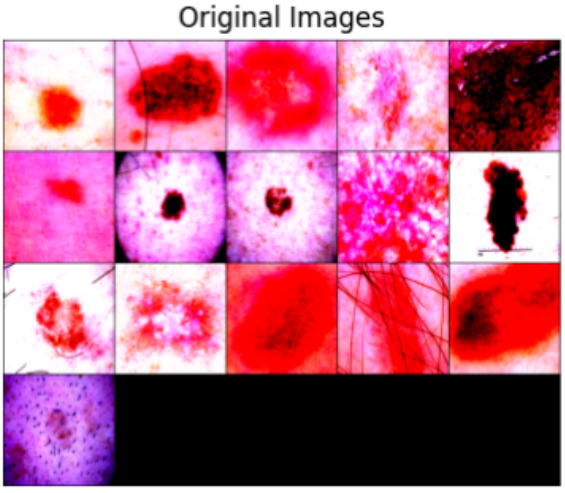
Epoch 19, Batch 100/564, Loss: 0.404565
Epoch 19, Batch 200/564, Loss: 0.447475

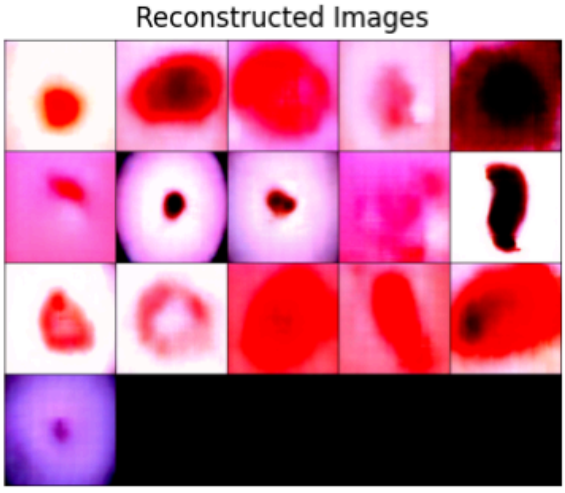
Epoch 19, Batch 300/564, Loss: 0.411083
Epoch 19, Batch 400/564, Loss: 0.403992
Epoch 19, Batch 500/564, Loss: 0.389486





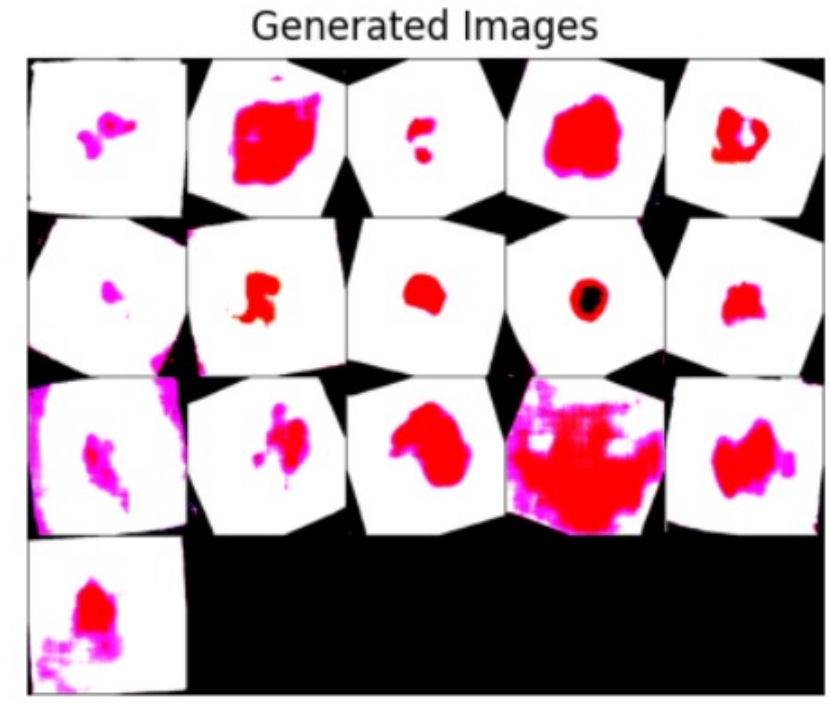
Epoch 20, Batch 100/564, Loss: 0.421070
Epoch 20, Batch 200/564, Loss: 0.432942
Epoch 20, Batch 300/564, Loss: 0.419344
Epoch 20, Batch 400/564, Loss: 0.389900
Epoch 20, Batch 500/564, Loss: 0.391272





PixelRCNN:

Loss of PixelRCNN after 100 epochs:
[100.00%] Epoch 100
Epoch [100/100], Loss: 0.4750



Conclusion:

In this assignment, we explored advanced generative modeling techniques by implementing and evaluating Vector Quantized Variational Autoencoder (VQ-VAE) and PixelRCNN models for skin lesion image generation. These models offer unique approaches to capturing meaningful latent representations and generating realistic images pixel by pixel, respectively.

The VQ-VAE model combines the power of variational autoencoders with vector quantization techniques to efficiently encode and decode high-dimensional input data while learning structured latent representations. By training the model on the ISIC dataset, comprising various skin lesion images, we demonstrated its ability to capture meaningful features and reconstruct images with fidelity.

Additionally, we developed the PixelRCNN model, an autoregressive generative model capable of generating diverse and realistic skin lesion images. By leveraging masked convolutional layers and residual connections, PixelRCNN can model the conditional probability distribution of each pixel given its predecessors.