CIS 6800 Project: VAE and Diffusion Transformer

Instructions:

- In this HW you will implement a patch-based VAE and train a Diffusion model using the pre-trained VAE checkpoint.
- We provided you with a zipfile that contains a subset of 10,000 images from the CelebA dataset.
- The Diffusion training session lasts a couple hours, so you should start part (b) as early as you can!
- **Please submit your ipynb notebook as well as a pdf version of it. Include all visualizations and answers to the questions.**
- Part A is due Wed 10/28, and Part B is due Wed 11/4.

Diffusion Transformer Paper: https://arxiv.org/pdf/2212.09748

```
from google.colab import drive
drive.mount('/content/gdrive')
%cd /content/gdrive/MyDrive/CIS6800HW4
```

```
Mounted at /content/gdrive
/content/gdrive/MyDrive/CIS6800HW4
```

```
!pip install -q torch torchvision numpy tqdm datasets torch-ema pytorch-lightning timm
```

```
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 480.6/480.6 kB 33.9 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 815.2/815.2 kB 51.6 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 116.3/116.3 kB 11.9 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 179.3/179.3 kB 18.8 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 134.8/134.8 kB 12.3 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 926.4/926.4 kB 59.8 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 194.1/194.1 kB 20.2 MB/s eta 0:00:00
ERROR: pip's dependency resolver does not currently take into account all the packages that are installed. This behaviour is the source of the following dependency conflicts.
gcsfs 2024.10.0 requires fsspec==2024.10.0, but you have fsspec 2024.9.0 which is incompatible.
```

```
import torch
from torch import nn
import torch.nn.functional as F
from torch.utils.data import Dataset, DataLoader, Subset
from torchvision import transforms as tf
from torchvision.datasets import FashionMNIST
from torchvision.utils import make_grid, save_image
from torch_ema import ExponentialMovingAverage as EMA
import pytorch_lightning as pl

import math
import random
import numpy as np
from tqdm import tqdm
from einops import rearrange, repeat  # Learn about einops at https://einops.rocks
from itertools import pairwise
from accelerate import Accelerator
from types import SimpleNamespace
from typing import Optional
import matplotlib.pyplot as plt
import torch.optim as optim

import zipfile
import io
from PIL import Image
from torchvision import transforms
from torchvision.utils import save_image, make_grid



# Set the seed for reproducibility

def set_seed(seed: int):
    torch.manual_seed(seed)
    if torch.cuda.is_available():
        torch.cuda.manual_seed_all(seed)
    np.random.seed(seed)
    random.seed(seed)

set_seed(42)
```

Part A

Variational Autoencoder (VAE)

In this part of the project you will train a Patch-based VAE to capture key facial features on on a subset of the CelebA dataset, consisting of 10000 face images. You can use the code below to load images from the given zipfile. We recommend resizing the images to (32, 32) for the following tasks but you're welcome to experiment with different sizes.

```
class CelebADataset(Dataset):
    def __init__(self, zip_file, transform=None):
        self.zip_file = zip_file
        self.transform = transform

        # Open the zip file and get the list of images
        self.zip = zipfile.ZipFile(self.zip_file, 'r')
        self.image_list = [file for file in self.zip.namelist() if file.endswith(('.jpg', '.png'))]

    def __len__(self):
        return len(self.image_list)

    def __getitem__(self, idx):
        # Get image name from the list
        img_name = self.image_list[idx]
        try:
            # Read image data from the zip file
            with self.zip.open(img_name) as img_file:
                img_data = img_file.read()
```

```
                img = Image.open(io.BytesIO(img_data)).convert('RGB')

            if self.transform:
                img = self.transform(img)

        except zipfile.BadZipFile:
            # print(f"BadZipFile error encountered with image {img_name}. Skipping this file.")
            return None

        return img

def collate_fn(batch):
    # Filter out None values (e.g., images that couldn't be loaded)
    batch = [b for b in batch if b is not None]
    # If the batch is empty after filtering, return None (can be skipped by DataLoader)
    if len(batch) == 0:
        return None

    return torch.utils.data.dataloader.default_collate(batch)

# Usage of the dataset and dataloader
def get_celeba_dataloader(zip_path, batch_size=32, image_size=(32, 32)):
    transform = transforms.Compose([
        transforms.Resize(image_size),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5])
    ])
    dataset = CelebADataset(zip_path, transform=transform)
    dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True, num_workers=4, collate_fn=collate_fn)

    return dataloader

# Function to display original and reconstructed images (4 images only)
def show_original_reconstructed(orig_images, recon_images, epoch):
    # Move the images back to CPU and denormalize
    orig_images = orig_images.cpu().numpy()
    recon_images = recon_images.cpu().numpy()

    # Clip the values to the valid range [0, 1] for display
    orig_images = np.clip(orig_images * 0.5 + 0.5, 0, 1)  # Denormalize and clip
    recon_images = np.clip(recon_images * 0.5 + 0.5, 0, 1)  # Denormalize and clip

    # Plot images side by side (4 images)
    fig, axes = plt.subplots(nrows=2, ncols=4, figsize=(10, 4))
    for i in range(4):
        # Original image
        axes[0, i].imshow(orig_images[i].transpose(1, 2, 0))  # Correct shape for imshow
        axes[0, i].axis('off')
        axes[0, i].set_title("Original")

        # Reconstructed image
        axes[1, i].imshow(recon_images[i].transpose(1, 2, 0))  # Correct shape for imshow
        axes[1, i].axis('off')
        axes[1, i].set_title("Reconstructed")

    plt.suptitle(f'Epoch {epoch}: Original vs Reconstructed')
    plt.show()


#Initialize your dataloaders here
zip_file_path = "celeba_10000_images.zip"
dataloader = get_celeba_dataloader(zip_file_path, batch_size=128, image_size=(32, 32))
```

Patch-based Variational Autoencoder

PatchVAE divides an input image into smaller patches and operates on these patches rather than the entire image at once. By processing images in patches, the PatchVAE learns to capture fine-grained (local) and high-level (global) patterns, depending on the patch size. In this section you should:

1. Implement the PatchEmbed Class. This class is designed to create overlapping patches from an image, embed those patches into a latent space, and provide a method to reconstruct the image from the latent representations.
2. Implement the PatchVAE model.
    - The `encode` method convert patch embeddings into latent variables (mu and logvar) through a convolutional encoder. Rearrange patches to treat them as channels before passing them through the encoder.
    - The `reparameterize` method applied the reparameterization trick `z = mu + eps * std` where `eps` is sampled from a standard normal distribution.
    - The `decode` method convert the latent variable z back into patch embeddings, then reconstruct the original image from these patches
    - The `forward` method first patchifies the image, encode, reparameterize, decode, and finally reconstruct the image
    - The `compute_loss` calculates the reconstruction loss (Mean Squared Error) and the KL divergence loss to encourage the latent space to follow a normal distribution. Combine these losses to get the total VAE loss.
    - The `sample` method generates new random images from the learned latent space. Random latent vector `z` are sampled, decoded into patches, and reconstructed into full images.
3. Train the model on the faces dataset. **Experiment with different patch sizes and report the patch size with the best tradeoff between low-level details and high-level texture. Plot the training and validation loss and visualize a couple reconstructed images during training.**
4. After training, generate sample images from the latent space. Specifically, you need to sample latent vectors from a normal distribution, decode the latent varaibles, and reconstruct the images. **Visualize 4 generated examples. Do the generated images resemble realistic human faces? Explain in a couple sentences.**

```
class PatchEmbed(nn.Module):
    def __init__(self, img_size=128, patch_size=32, stride=8, channels=3, embed_dim=128, bias=True):
        super().__init__()
        """
        Use Conv2D to create image patches of size (patch_size, patch_size) with overlapping regions.

        Each patch should have embedding size embed_dim.
        """
        self.patch_size = patch_size
```

```
            self.patch_size = patch_size
            self.stride = stride
            self.embed_dim = embed_dim
            self.img_size = img_size

            # Conv2d to generate overlapping patches (from image to latent space)
            self.proj = nn.Conv2d(channels, embed_dim, kernel_size=patch_size, stride=stride, bias=bias)

            # Transposed Conv2d to reconstruct patches from latent space to RGB (from latent to image space)
            self.deconv = nn.ConvTranspose2d(embed_dim, channels, kernel_size=patch_size, stride=stride, bias=bias)

            H_out = (img_size - self.patch_size) // self.stride + 1
            W_out = (img_size - self.patch_size) // self.stride + 1
            self.num_patches = H_out * W_out

        def forward(self, x):
            """
            Input x is an image of size [B, C, img_size, img_size]

            Return patches of size [B, num_patches, embed_dim]
            """

            patches = self.proj(x)
            patches = patches.flatten(2)
            patches = patches.transpose(-1, -2)

            return patches

        def reconstruct(self, patches, img_size):
            """
            Reconstruct the image from the patches by averaging overlapping regions.
            Input patches: [B, num_patches, embed_dim]
            img_size: (img_size, img_size)  # original size of the input image

            Output images: [B, img_size, img_size]
            """

            patches = patches.transpose(-1, -2)
            patches = patches.reshape(patches.shape[0], patches.shape[1], int(np.sqrt(self.num_patches)), int(np.sqrt(self.num_patches)))
            reconstructed_image = self.deconv(patches)


            return reconstructed_image


    class PatchVAE(nn.Module):
        def __init__(self, patch_size, img_channels, img_size,
                     embed_dim=1024, latent_dim=512, stride=8):
            super(PatchVAE, self).__init__()
            self.patch_size = patch_size
            self.img_size = img_size
            self.embed_dim = embed_dim
            self.latent_dim = latent_dim
            self.device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

            # Patch embedding layer (Patchify the image)
            self.patch_embed = PatchEmbed(img_size=img_size, patch_size=patch_size, stride=stride, channels=img_channels, embed_dim=embed_dim)
            self.num_patches = self.patch_embed.num_patches

            # Encoder
            self.encoder = nn.Sequential(
                nn.Conv2d(embed_dim, 256, kernel_size=3, stride=1, padding=1),
                nn.BatchNorm2d(256),
                nn.LeakyReLU(0.2),
                nn.Conv2d(256, 128, kernel_size=3, stride=1, padding=1),
                nn.BatchNorm2d(128),
                nn.LeakyReLU(0.2),
            )
            self.conv_mu = nn.Conv2d(128, latent_dim, kernel_size=3, stride=1, padding=1)
            self.conv_logvar = nn.Conv2d(128, latent_dim, kernel_size=3, stride=1, padding=1)

            # Decoder
            self.decoder_input = nn.Conv2d(latent_dim, 128, kernel_size=3, stride=1, padding=1)
            self.decoder = nn.Sequential(
                nn.BatchNorm2d(128),
                nn.LeakyReLU(0.2),
                nn.ConvTranspose2d(128, 256, kernel_size=3, stride=1, padding=1),
                nn.BatchNorm2d(256),
                nn.LeakyReLU(0.2),
                nn.ConvTranspose2d(256, embed_dim, kernel_size=3, stride=1, padding=1),
            )

        def encode(self, patches):
            """
            Encode the patch embeddings into latent space (mu and logvar).
            Args:
                patches: Patch embeddings of shape [B, num_patches, embed_dim].
            """

            patches = patches.transpose(-1, -2).unsqueeze(2)
            encoded_patches = self.encoder(patches)
            mu = self.conv_mu(encoded_patches)
            logvar = self.conv_logvar(encoded_patches)
            return mu, logvar

        def reparameterize(self, mu, logvar):
            """
            Reparameterization trick to sample from N(mu, var) using N(0,1).
            Args:
                mu: Mean of the latent distribution.
                logvar: Log variance of the latent distribution.
            """

            eps = torch.normal(mean=torch.zeros_like(mu), std=torch.ones_like(mu))
            std = torch.exp(0.5 * logvar) # Putting 0.5 in the exponent is a more efficient alternative to taking the square root outside the log

            return mu + eps * std

        def decode(self, z):
            """
            Decode the latent variable z back to patch embeddings.
```

```
            Args:
                z: Latent variable of shape [B, latent_dim, 1, num_patches].
            """

            patch_recon = self.decoder_input(z)
            patch_recon = self.decoder(patch_recon)

            return rearrange(patch_recon, 'b c 1 p -> b p c')  # Back to (B, num_patches, embed_dim)

        def forward(self, x):
            """
            Forward pass through the VAE. Patchify the input, encode into latent space, reparameterize, and decode.
            Args:
                x: Input image of shape [B, C, img_size, img_size].
            """

            patches = self.patch_embed.forward(x)
            mu, logvar = self.encode(patches)
            z = self.reparameterize(mu, logvar)
            recon_patches = self.decode(z)
            recon_image = self.patch_embed.reconstruct(recon_patches, self.img_size)


            return recon_image, mu, logvar

        def compute_loss(self, recon_image, original_image, mu, logvar):
            """
            Compute the VAE loss, which consists of the reconstruction loss and KL divergence.
            Args:
                recon_image: Reconstructed image. (ERIK NOTE: Should be (batch_size, 3, img_size, img_size))
                original_image: Original input image. (ERIK NOTE: Should be (batch_size, 3, img_size, img_size))
                mu: Mean of the latent distribution. (ERIK NOTE: Should be (batch_size, latent_dim))
                logvar: Log variance of the latent distribution. (ERIK NOTE: Should be (batch_size, latent_dim))
            Returns:
                loss (Tensor): Total loss (reconstruction loss + KL divergence). (ERIK NOTE: Should be (batch_size))
            """

            var = torch.exp(logvar)
            recon_loss = F.mse_loss(recon_image, original_image)
            kl_loss = -0.5 * torch.sum(1 + torch.log(var**2) - mu**2 - var**2)

            return recon_loss, kl_loss

        def sample(self, num_samples):
            """
            Generate random samples from the learned distribution.
            Args:
                num_samples (int): Number of samples to generate.
            Returns:
                samples (Tensor): Generated (ERIK NOTE: Should be (num_samples, 3, img_size, img_size))
            """

            z = torch.normal(mean=torch.zeros((num_samples, self.latent_dim, 1, self.num_patches)), std=torch.ones((num_samples, self.latent_dim, 1, self.num_patches))).to(device)
            sample_patches = self.decode(z)
            sample_images = self.patch_embed.reconstruct(sample_patches, self.img_size)
            return sample_images


    def train_patchvae(model, dataloader, optimizer, device, epochs=10, print_interval=100, checkpoint_path='best_model.pth'):
        """
        Training loop for the PatchVAE model with visualization of reconstructed images and saving the best model checkpoint.

        Args:
            model: The PatchVAE model to be trained.
            dataloader: Dataloader for the training data.
            optimizer: Optimizer for updating the model parameters.
            device: Device (CPU or GPU) on which the training will run.
            epochs: Number of training epochs.
            print_interval: Interval at which the loss will be printed during training.
            checkpoint_path: Path to save the best model checkpoint.

        Returns:
            losses: List of training losses for each epoch.
        """
        model.train()
        model.to(device)
        losses = []
        recon_losses = []
        kl_losses = []
        best_loss = float('inf')  # Initialize with a large value
        initial_beta = 5e-8 # Start with a small KL weight
        final_beta = 5e-6 # Gradually increase to full KL weight

        for epoch in range(epochs):

            running_loss = 0.0
            running_recon_loss = 0.0
            running_kl_loss = 0.0
            if epoch < 50:
                beta = np.exp(np.log(initial_beta) + (np.log(final_beta) - np.log(initial_beta)) * (epoch / (50-1)))
            else:
                beta = final_beta
            for batch_idx, original_image in enumerate(dataloader):
                original_image = original_image.to(device)
                recon_image, mu, logvar = model.forward(original_image)

                recon_loss, kl_loss = model.compute_loss(recon_image, original_image, mu, logvar)
                loss = recon_loss.mean() + beta * kl_loss.mean()

                optimizer.zero_grad()
                loss.backward()
                optimizer.step()
                running_loss += loss.item()
                running_recon_loss += recon_loss.item()
                running_kl_loss += kl_loss.item()

                if batch_idx == 0:
                    if epoch == 0:
                        print(f'Beginning of Training, Loss: {loss.item()}, Reconstruction Loss: {recon_loss.item()}, KL Loss: {kl_loss.item()}')
                        losses.append(loss.item())
```

```
                    recon_losses.append(recon_loss.item())
                    kl_losses.append(kl_loss.item())

                visualize_reconstruction(original_image[:4].detach(), recon_image[:4].detach(), epoch)

        avg_loss = running_loss / len(dataloader)
        avg_recon_loss = running_recon_loss / len(dataloader)
        avg_kl_loss = running_kl_loss / len(dataloader)
        losses.append(avg_loss)
        recon_losses.append(avg_recon_loss)
        kl_losses.append(avg_kl_loss)
        print(f'Epoch: {epoch+1}/{epochs}, Avg Loss: {avg_loss}, Avg Reconstruction Loss: {running_recon_loss/len(dataloader)}, Avg KL Loss: {running_kl_loss/len(dataloader)}, Beta:
        if loss < best_loss:
            best_loss = loss
            torch.save(model.state_dict(), '/content/' + checkpoint_path)

    return losses, recon_losses, kl_losses

def visualize_reconstruction(original_images, recon_images, epoch):
    """
    Visualize original and reconstructed images side by side.

    Args:
        original_images: Batch of original images (values between 0 and 1).
        recon_images: Batch of reconstructed images (values between 0 and 1).
        epoch: Current epoch number.
    """
    num_images = min(4, original_images.size(0))  # Visualize at most 4 images
    fig, axes = plt.subplots(2, num_images, figsize=(12, 4))

    for i in range(num_images):
        # Original images (ensure values are between 0 and 1)
        orig_img = original_images[i].permute(1, 2, 0).cpu().numpy()  # Convert to (H, W, C)
        orig_img = (orig_img + 1) / 2
        orig_img = np.clip(orig_img, 0, 1)  # Clip values to [0, 1] range

        # Reconstructed images (ensure values are between 0 and 1)
        recon_img = recon_images[i].permute(1, 2, 0).cpu().numpy()  # Convert to (H, W, C)
        recon_img = (recon_img + 1) / 2
        recon_img = np.clip(recon_img, 0, 1)  # Clip values to [0, 1] range

        # Display original images
        axes[0, i].imshow(orig_img)
        axes[0, i].set_title("Original")
        axes[0, i].axis('off')

        # Display reconstructed images
        axes[1, i].imshow(recon_img)
        axes[1, i].set_title("Reconstructed")
        axes[1, i].axis('off')

    plt.suptitle(f"Reconstruction at Epoch {epoch+1}")
    plt.show()
```

The code for training the PatchVAE model, alongside example reconstructions from during training and the training loss curves, are shown below for both patch sizes of 32 and 8. As explained below, in our experimentation with different patch sizes, we found that a patch size of 32 provided both the best high-level structure and low-level detail, so our results with a patch size of 32 should be viewed as our true results, while our results with a patch size of 8 should merely be regarded as a reference point for a previous failed attempt which motivated the change to a patch size of 32.

ASIDE: HYPERPARAMETER DISCUSSION

In training the PatchVAE model, we made several hyperparameter decisions in addition to experimenting with the patch size. First, we set the embedding dimension to 1024 and the latent dimension to 512 (the default values from the definition of the PatchVAE class), instead of the values of embed-dim=64 and latent-dim=128 which the provided code passed in when instantiating the vae object for training. We did this because after experimenting with training using embed-dim=64 and latent-dim=128, we found that the model with this size could not simultaneously minimize both the reconstruction loss and the KL loss. We could set the beta parameter to heavily favor minimizing one loss or the other, and that one loss could be minimized while failing to minimize the other, but no set of beta parameters allowed this smaller model to minimize both. When minimizing the reconstruction loss, the model could reconstruct faces moderately well during training, but the lack of regularization would cause the model to generate meaningless noise when sampling the latent space at test time, and when minimizing the KL loss, the model could not even reconstruct faces during training, also resulting in garbage outputs when sampling the latent space at test time.

Seeing that the PatchVAE model at its current size was underfitting to the data, as it was unable to learn a function which simultaneously reduced both losses to an acceptable degree, we increased the model size by setting the embedding dimension to 1024 and the latent dimension to 512, their default values in the definition of the PatchVAE class. This enabled the model to sufficiently minimize both losses and produce results which exhibit both effective reconstruction (due to the minimization of the reconstruction loss) and effective generations when sampling from the latent space (since the latent space had been sufficiently regularized by minimizing the KL loss). In order to train this larger model, we trained for 50 epochs, instead of the 15 in the provided code.

We also decreased the beta values, setting an initial beta of 5e-8 and a final beta of 5e-6. This is because the KL loss is much larger than the reconstruction loss early in training, and should be weighted much less accordingly. Even later in training, beta can increase in order to encourage the model to minimize the KL loss, but this weight cannot approach 1, as this causes the model to prioritize minimizing the KL loss too heavily, which we saw experimentally resulted in substantial decay in the reconstruction performance. The result of this beta schedule is a slightly odd set of loss curves in which the reconstruction loss quickly goes down as the model learns to effectively reconstruct its inputs at the expense of the KL loss, which initially goes up. However, once the model has nearly fully optimized its reconstruction performance, and beta increases for the already large KL loss, it begins optimizing the KL loss, which steadily decreases for the rest of training. Experimentally, we found this method delivered the best reconstruction and generation results. While our total loss curve does increase towards the end of training, this simply occurs because the KL loss is being minimized at the slight expense of the reconstruction loss, and since the weight (beta) on the KL loss is so low, the impact on the total loss of the slight increase of the reconstruction loss actually exceeds the impact of the great decreases in the KL loss. Ultimately, setting the final value of beta to be so low is a necessary engineering decision to prevent the model from overregularizing and losing its reconstruction performance, but does have this effect of making the total loss curve "increase" even as the model is overall improving as it trades off major decreases in KL loss for slight increases in reconstruction loss.

PATCH SIZE EXPERIMENTATION:

To experiment with patch size, we started with a patch size of 8, which caused the image to be patched into a 4x4 grid. Notably, this causes the problem that the patches are passed through the network independently, and thus each patch does not take into account the other patches, so

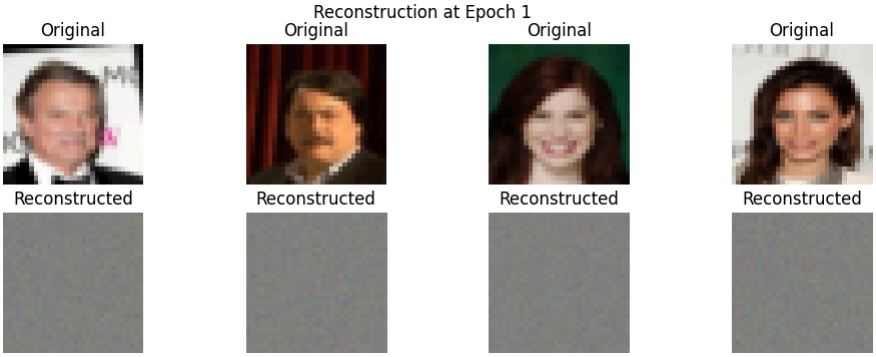
the patches do not form a coherent image. At training time for the model with patch size 8, we see that the reconstructed patches correspond moderately well to the original image, but are misaligned with each other. At test time, the model with patch size 8 produces meaningless noise and mostly darkness, as the model does not know which patches should be foreground and which should be background, and cannot generate anything meaningful.

We found that this problem could be resolved by setting the patch size to 32, so as to encode the entire image into a singular patch, eliminating problems with trying to reconstruct and generate images from multiple independent patches. The model with patch size 32 performed the best both in terms of high-quality structure and low-level detail, as its reconstructions during training time and its generations at test time not only consisted of broadly-coherent images of realistic human faces, but also included all of the lower-level details of a human face, such as eyes, nose, mouth, and hair. However, the performance was still not perfect, as the model demonstrates a tendency towards female-looking faces both in generation and in reconstruction during training (possibly because female faces are overrepresented in the dataset), and the faces generated at inference time display minimal variation, only providing slight variations in head angle, background color, and skin tone.

```
# Train PatchVAE with Patch Size 32
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
resume_training = False
checkpoint_path = 'best_vae_model.pth'
num_epochs = 50


vae = PatchVAE(patch_size=32, img_channels=3, img_size=32, embed_dim=1024, latent_dim=512).to(device)
optimizer = optim.Adam(vae.parameters(), lr=1e-4)

if resume_training:
    vae.load_state_dict(torch.load(checkpoint_path))

losses, recon_losses, kl_losses = train_patchvae(vae, dataloader, optimizer, device, epochs=num_epochs, checkpoint_path='best_vae_model.pth')
```

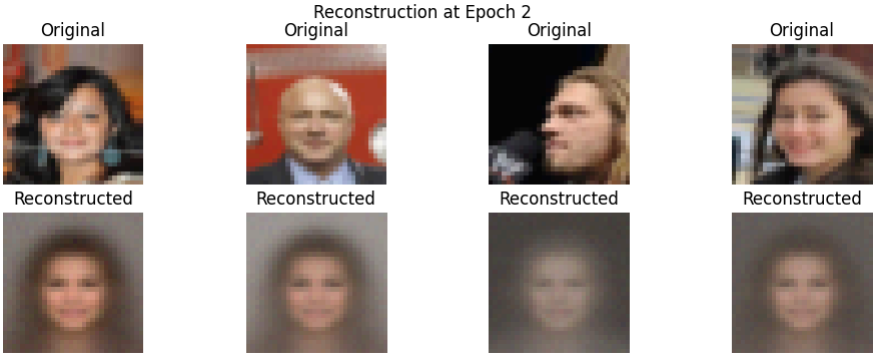
Beginning of Training, Loss: 0.36298665404319763, Reconstruction Loss: 0.3628949877357483, KL Loss: 1763.123046875

Reconstruction at Epoch 1



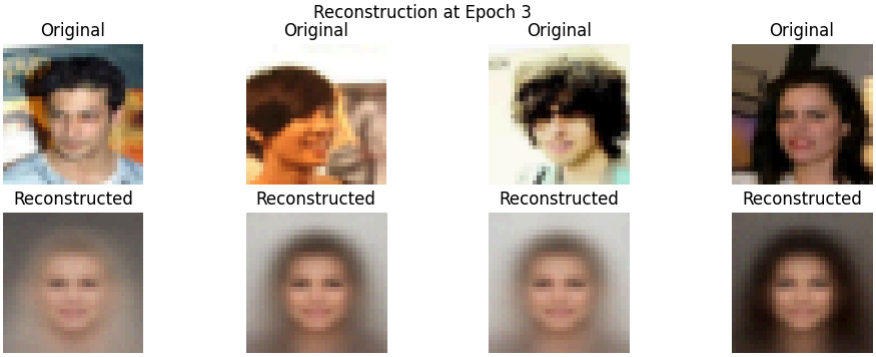
Epoch: 1/50, Avg Loss: 0.2274101712281191, Avg Reconstruction Loss: 0.2272447059048882, Avg KL Loss: 3309.3080251186707, Beta: 5.000000000000006e-08

Reconstruction at Epoch 2



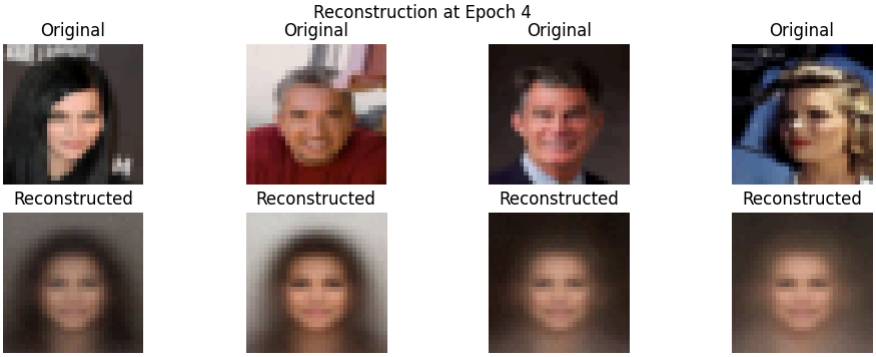
Epoch: 2/50, Avg Loss: 0.17114986007726646, Avg Reconstruction Loss: 0.1708155487534366, Avg KL Loss: 6086.458442011966, Beta: 5.492705709937796e-08

Reconstruction at Epoch 3



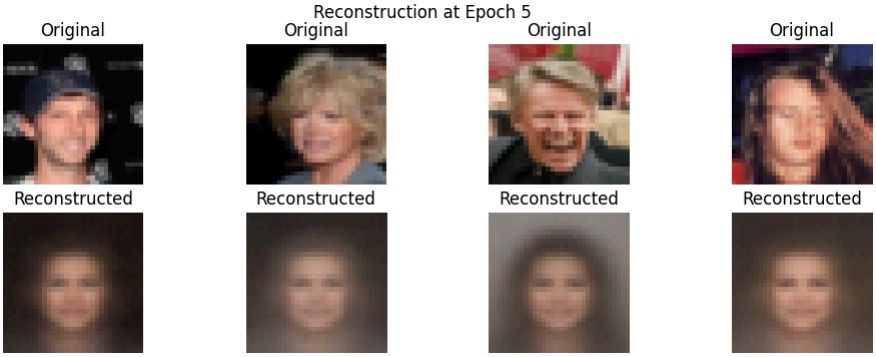
Epoch: 3/50, Avg Loss: 0.1664667203079296, Avg Reconstruction Loss: 0.16595699628697166, Avg KL Loss: 8447.583857669106, Beta: 6.033963203196648e-08

Reconstruction at Epoch 4



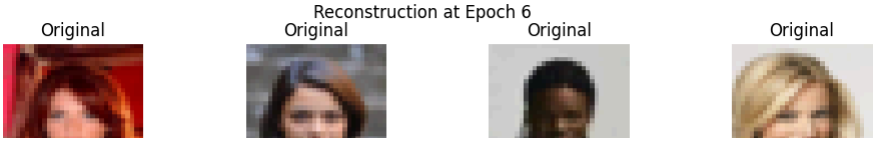
Epoch: 4/50, Avg Loss: 0.16541313578056382, Avg Reconstruction Loss: 0.16476528346538544, Avg KL Loss: 9773.652224770076, Beta: 6.628556827950549e-08

Reconstruction at Epoch 5



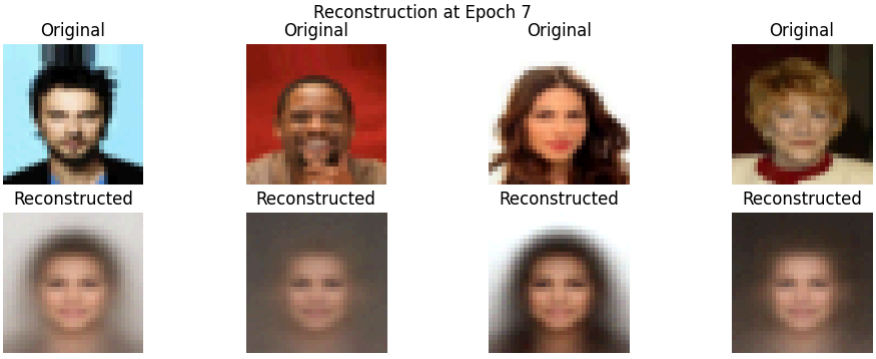
Epoch: 5/50, Avg Loss: 0.15933078489726102, Avg Reconstruction Loss: 0.1586394889068, Avg KL Loss: 9493.555290125594, Beta: 7.281742387506222e-08

Reconstruction at Epoch 6



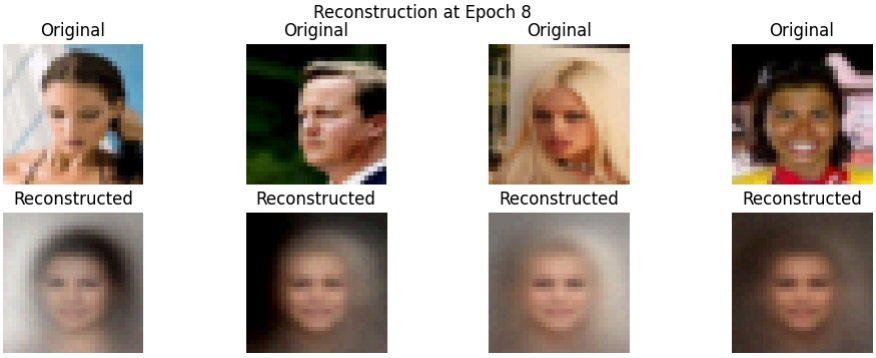
Epoch: 6/50, Avg Loss: 0.15567960976799833, Avg Reconstruction Loss: 0.15475394473045687, Avg KL Loss: 11571.831427079213, Beta: 7.999293598030292e-08

Reconstruction at Epoch 7



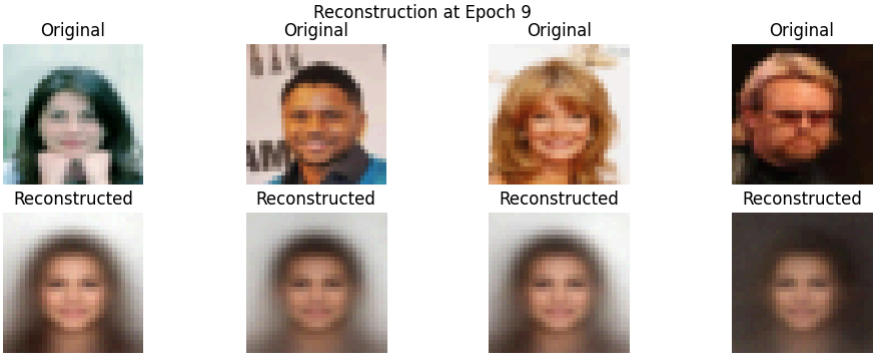
Epoch: 7/50, Avg Loss: 0.15036395795737642, Avg Reconstruction Loss: 0.14928591722929024, Avg KL Loss: 12267.813300410404, Beta: 8.787553124273961e-08

Reconstruction at Epoch 8



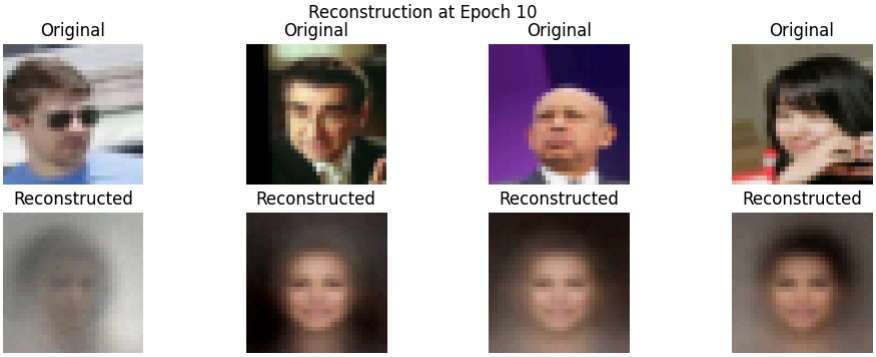
Epoch: 8/50, Avg Loss: 0.1372876277641405, Avg Reconstruction Loss: 0.13612539641842059, Avg KL Loss: 12039.49907597409, Beta: 9.65348864441625e-08

Reconstruction at Epoch 9



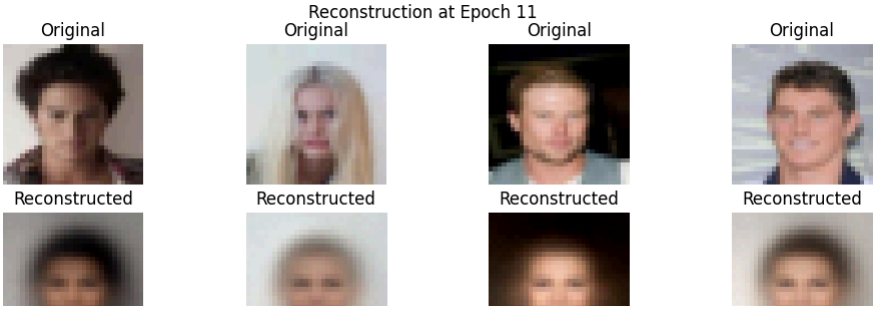
Epoch: 9/50, Avg Loss: 0.13281295731475082, Avg Reconstruction Loss: 0.13139059064508993, Avg KL Loss: 13412.53611272498, Beta: 1.060475443960095e-07

Reconstruction at Epoch 10



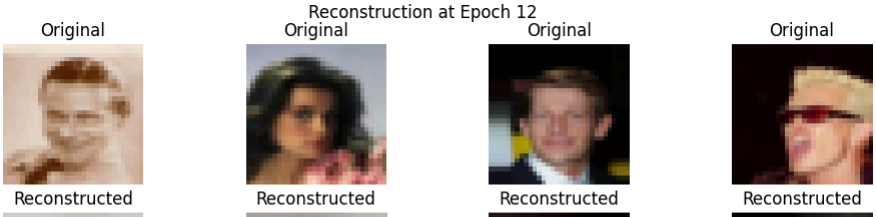
Epoch: 10/50, Avg Loss: 0.12696997755313222, Avg Reconstruction Loss: 0.1252834340439567, Avg KL Loss: 14477.066616396361, Beta: 1.1649759052576877e-07

Reconstruction at Epoch 11



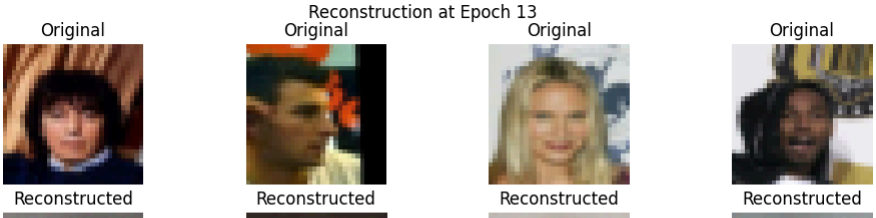
Epoch: 11/50, Avg Loss: 0.12429884245878534, Avg Reconstruction Loss: 0.12231665180076527, Avg KL Loss: 15488.597557357594, Beta: 1.279773961349769e-07

Reconstruction at Epoch 12



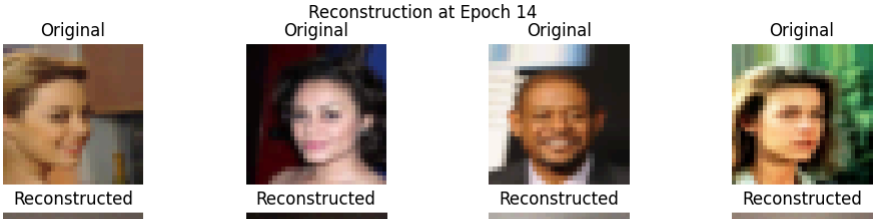
Epoch: 12/50, Avg Loss: 0.12116845716026765, Avg Reconstruction Loss: 0.11899699846023246, Avg KL Loss: 15445.501217612737, Beta: 1.4058843489871165e-07

Reconstruction at Epoch 13



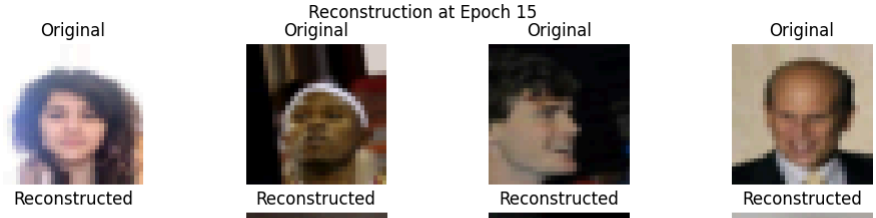
Epoch: 13/50, Avg Loss: 0.11353860597444486, Avg Reconstruction Loss: 0.11127636468485941, Avg KL Loss: 14647.818819842762, Beta: 1.5444217982387415e-07

Reconstruction at Epoch 14



Epoch: 14/50, Avg Loss: 0.11034558430502686, Avg Reconstruction Loss: 0.10759437376562553, Avg KL Loss: 16215.918333415744, Beta: 1.6966108859476655e-07

Reconstruction at Epoch 15

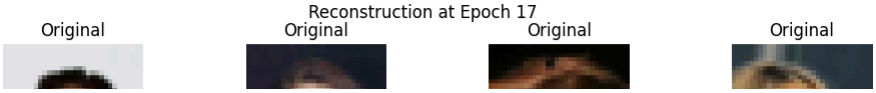


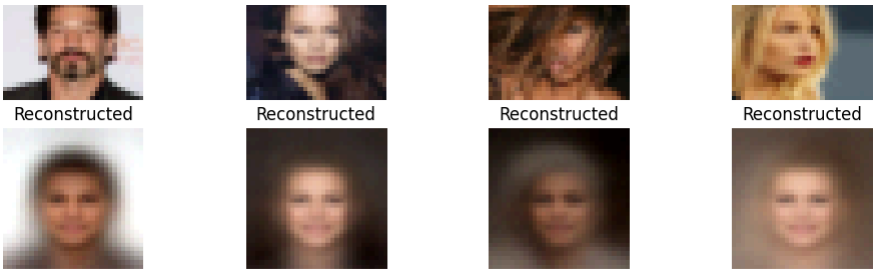
Epoch: 15/50, Avg Loss: 0.10950254403714892, Avg Reconstruction Loss: 0.106385406153866, Avg KL Loss: 16724.667860586433, Beta: 1.863796860157471e-07

Reconstruction at Epoch 16

Epoch: 16/50, Avg Loss: 0.10820597994931136, Avg Reconstruction Loss: 0.10490831883647773, Avg KL Loss: 16106.127064378956, Beta: 2.0474575311902131e-07

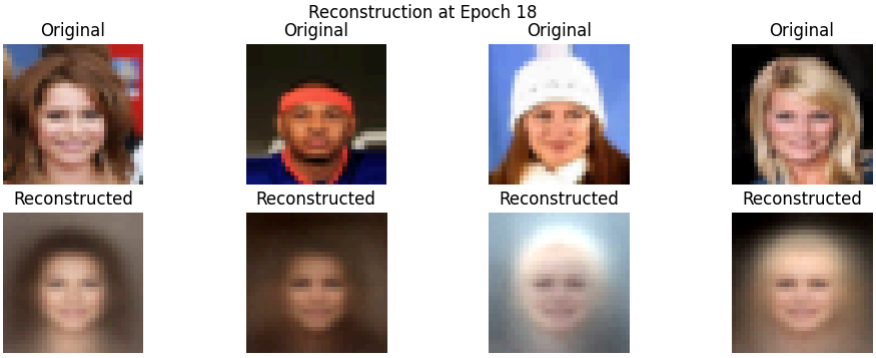
Reconstruction at Epoch 17





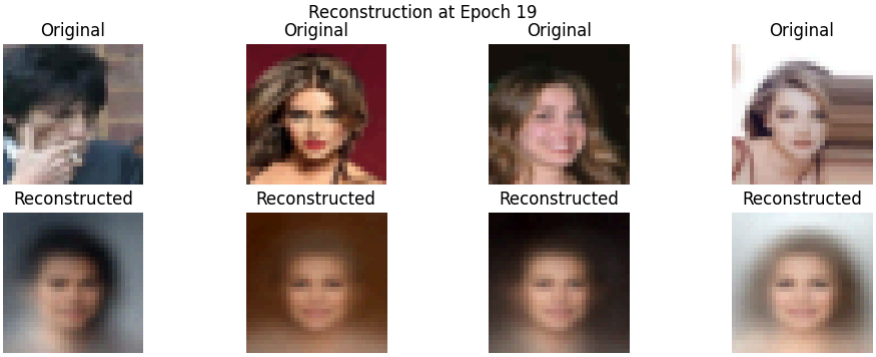
Epoch: 17/50, Avg Loss: 0.10570779772876185, Avg Reconstruction Loss: 0.10224043191233768, Avg KL Loss: 15415.88493868671, Beta: 2.249216334484723e-07

Reconstruction at Epoch 18



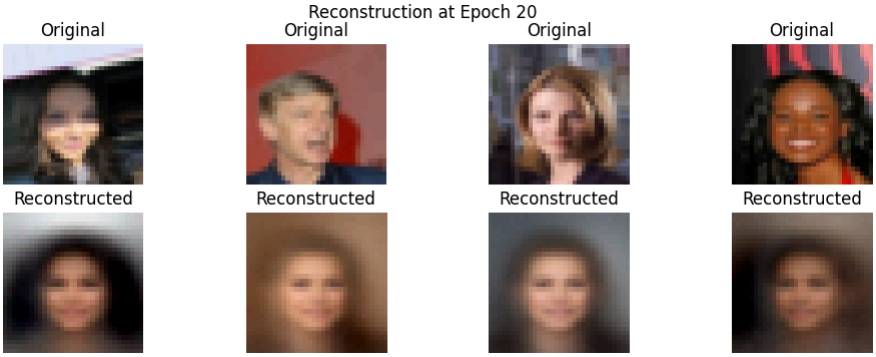
Epoch: 18/50, Avg Loss: 0.10445608585318432, Avg Reconstruction Loss: 0.1005357467109644, Avg KL Loss: 15866.315670737737, Beta: 2.470856680661917e-07

Reconstruction at Epoch 19



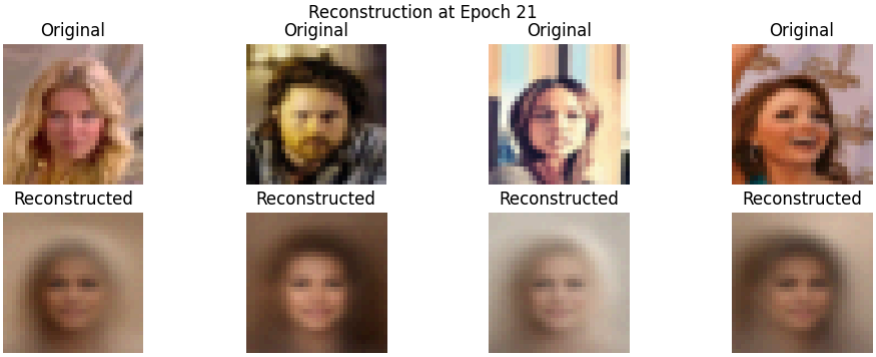
Epoch: 19/50, Avg Loss: 0.1040457023095481, Avg Reconstruction Loss: 0.09976014391153673, Avg KL Loss: 15788.595622774921, Beta: 2.7143377196619345e-07

Reconstruction at Epoch 20



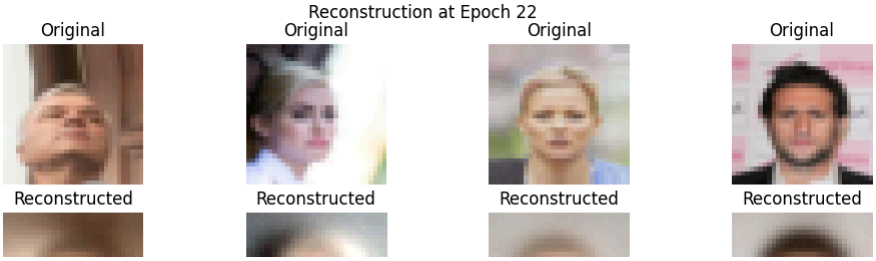
Epoch: 20/50, Avg Loss: 0.1018781650669967, Avg Reconstruction Loss: 0.0975106982867929, Avg KL Loss: 14647.024940973595, Beta: 2.9818116582973254e-07

Reconstruction at Epoch 21



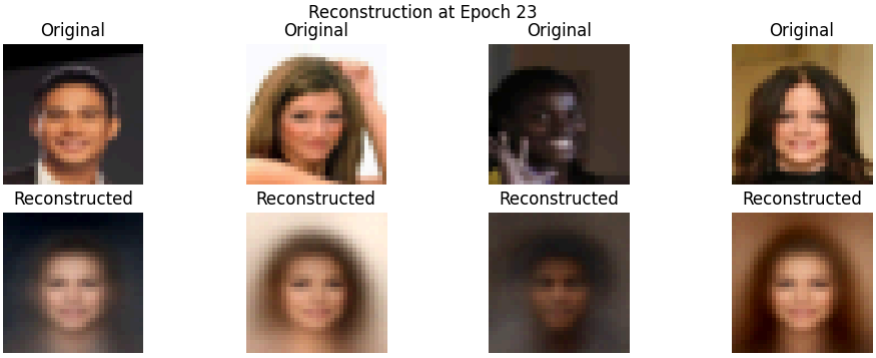
Epoch: 21/50, Avg Loss: 0.10113095944818062, Avg Reconstruction Loss: 0.09627153309462946, Avg KL Loss: 14835.032334726067, Beta: 3.275642784297758e-07

Reconstruction at Epoch 22



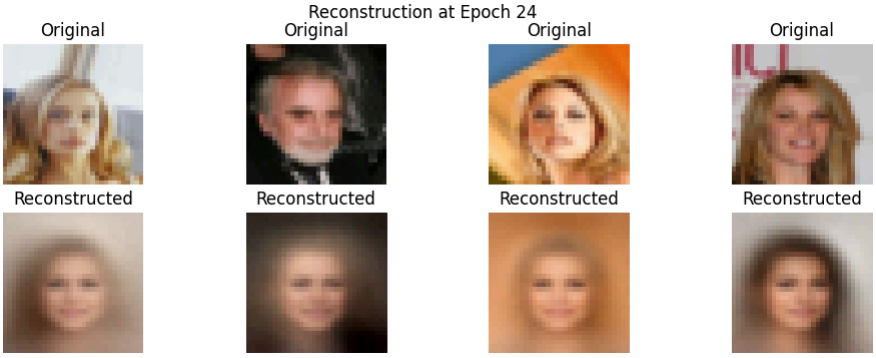
Epoch: 22/50, Avg Loss: 0.10157434548003764, Avg Reconstruction Loss: 0.09632367951960503, Avg KL Loss: 14591.55389636076, Beta: 3.598428365005764e-07

Reconstruction at Epoch 23



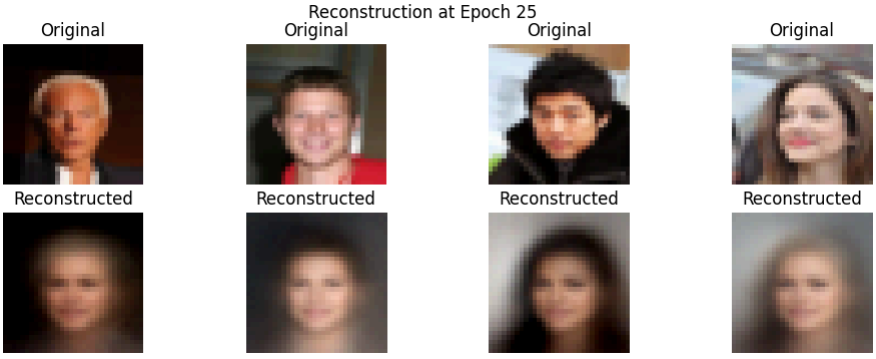
Epoch: 23/50, Avg Loss: 0.10156390778248824, Avg Reconstruction Loss: 0.09594341216585305, Avg KL Loss: 14218.225663197192, Beta: 3.953021605453853e-07

Reconstruction at Epoch 24



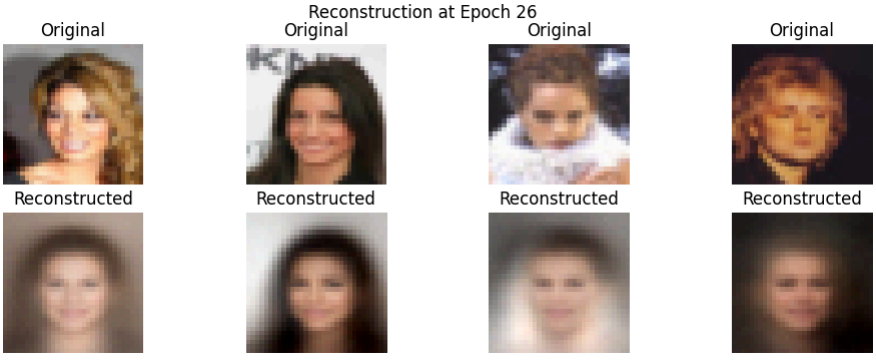
Epoch: 24/50, Avg Loss: 0.10150902114714248, Avg Reconstruction Loss: 0.09561550551200215, Avg KL Loss: 13571.533175311512, Beta: 4.342556868756766e-07

Reconstruction at Epoch 25



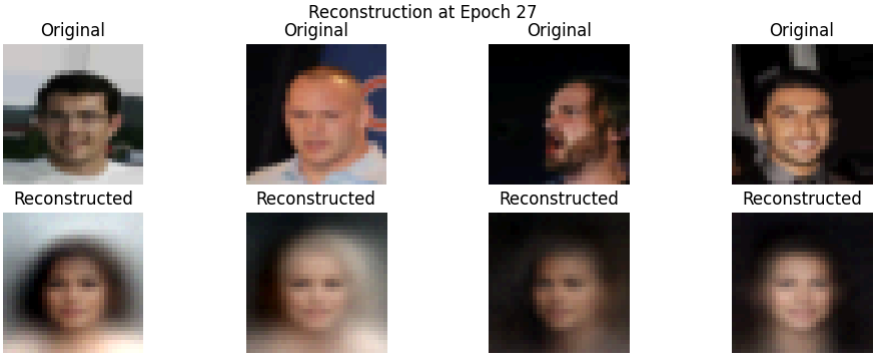
Epoch: 25/50, Avg Loss: 0.10182241557896891, Avg Reconstruction Loss: 0.09564996859695338, Avg KL Loss: 12938.844720381725, Beta: 4.770477381749972e-07

Reconstruction at Epoch 26



Epoch: 26/50, Avg Loss: 0.10114572781927977, Avg Reconstruction Loss: 0.0947123148207423, Avg KL Loss: 12276.180432283425, Beta: 5.240565670773431e-07

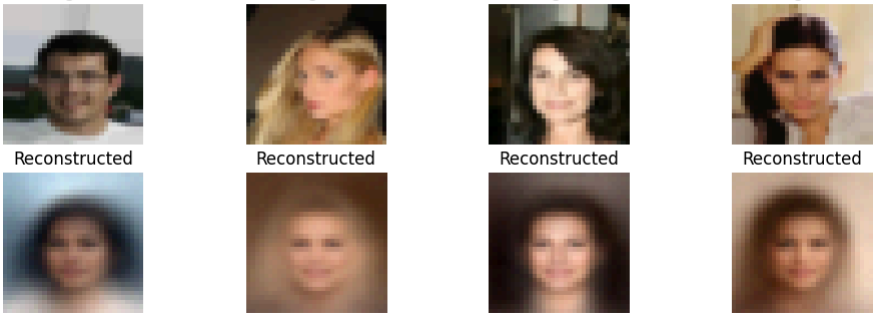
Reconstruction at Epoch 27



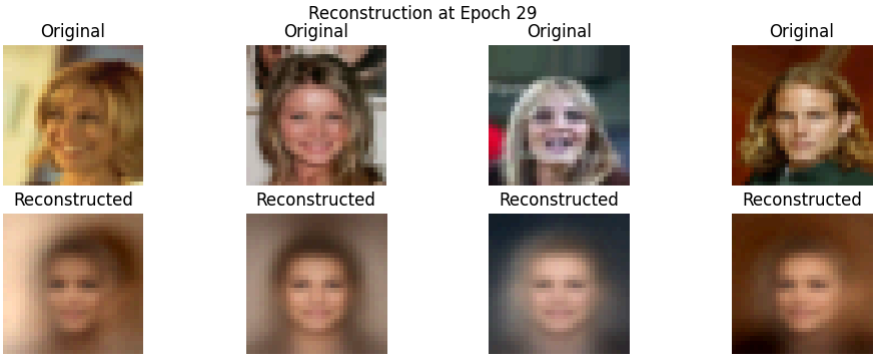
Epoch: 27/50, Avg Loss: 0.1001996603570407, Avg Reconstruction Loss: 0.09328174789117862, Avg KL Loss: 12016.571723261966, Beta: 5.756976996632237e-07

Reconstruction at Epoch 28

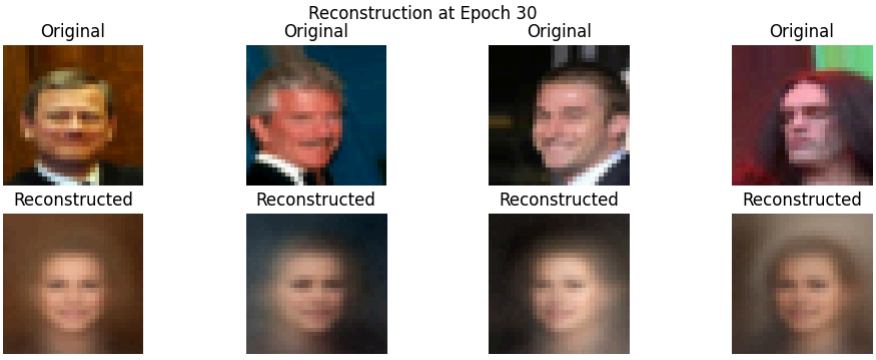
Original Original Original Original



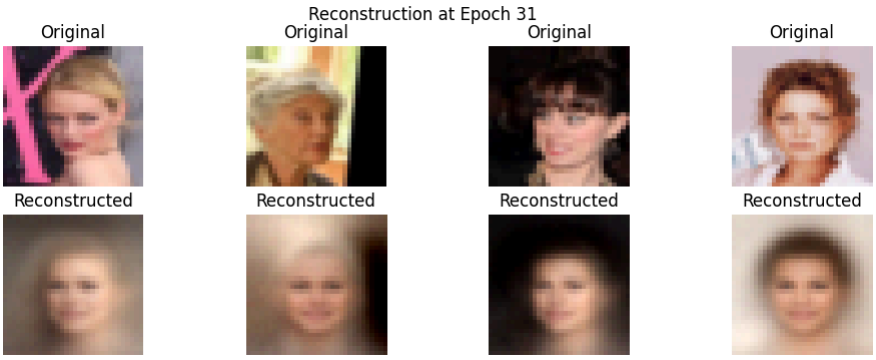
Epoch: 28/50, Avg Loss: 0.100263018208214, Avg Reconstruction Loss: 0.09299421791411654, Avg KL Loss: 11493.490157115308, Beta: 6.324276084276 48e-07



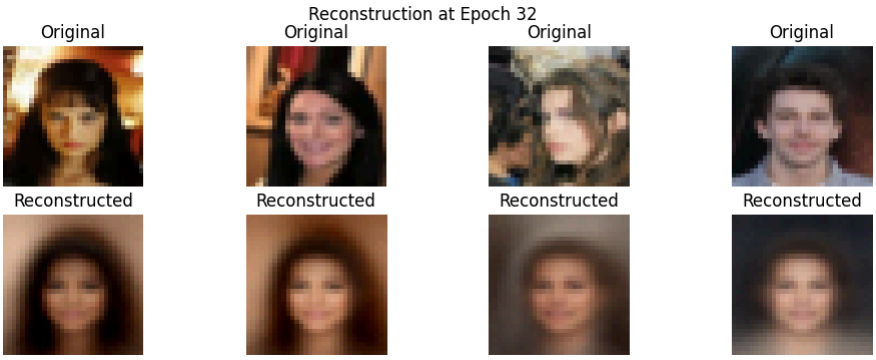
Epoch: 29/50, Avg Loss: 0.10114385857235027, Avg Reconstruction Loss: 0.09341597217547742, Avg KL Loss: 11123.297792845135, Beta: 6.947477471865687e-07



Epoch: 30/50, Avg Loss: 0.10067030212169961, Avg Reconstruction Loss: 0.09257968552882158, Avg KL Loss: 10600.78806584998, Beta: 7.632089835876164e-07



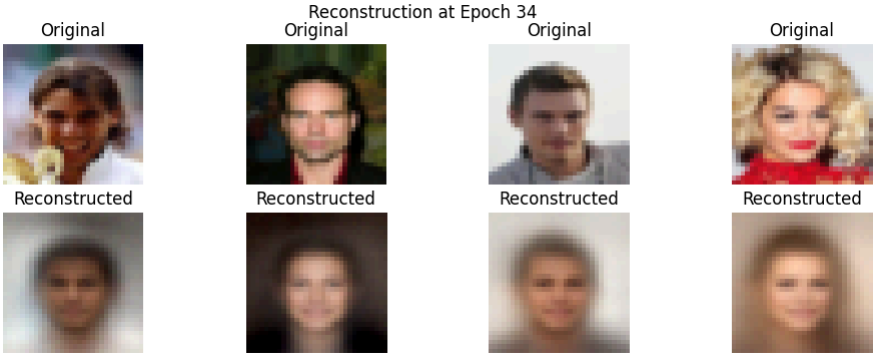
Epoch: 31/50, Avg Loss: 0.10145693998548048, Avg Reconstruction Loss: 0.09295064154305036, Avg KL Loss: 10145.672354010087, Beta: 8.384164684055036e-07



Epoch: 32/50, Avg Loss: 0.10173045542043975, Avg Reconstruction Loss: 0.09293954545938515, Avg KL Loss: 9544.59939150267, Beta: 9.210349846633574e-07

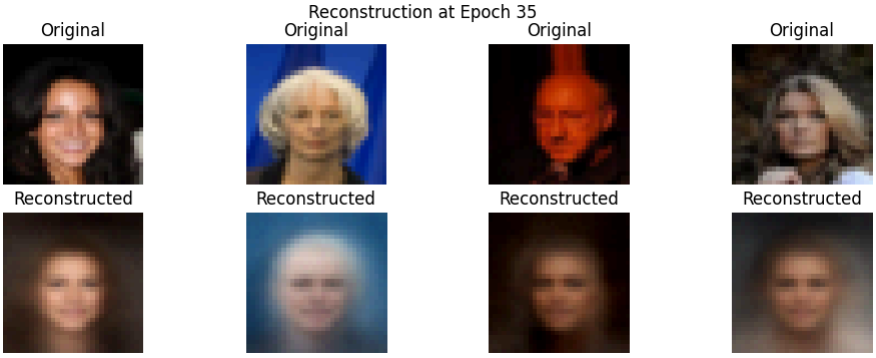
Epoch: 33/50, Avg Loss: 0.10134292186438283, Avg Reconstruction Loss: 0.09209374517579622, Avg KL Loss: 9141.356113095826, Beta: 1.0117948238625774e-06

Reconstruction at Epoch 34



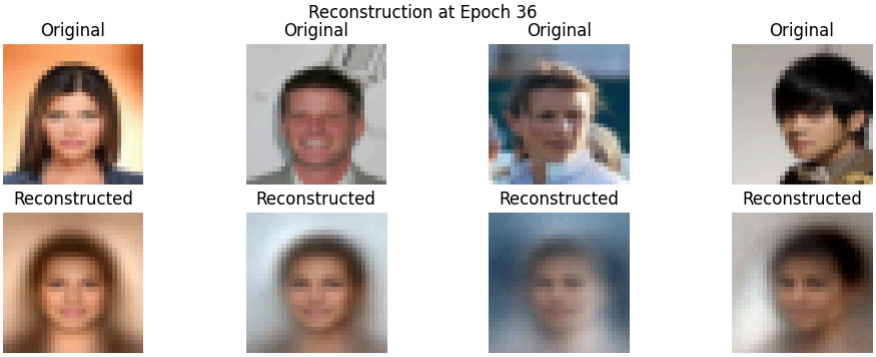
Epoch: 34/50, Avg Loss: 0.10074475388738173, Avg Reconstruction Loss: 0.09104541868348665, Avg KL Loss: 8726.361294130736, Beta: 1.1114982412630962e-06

Reconstruction at Epoch 35



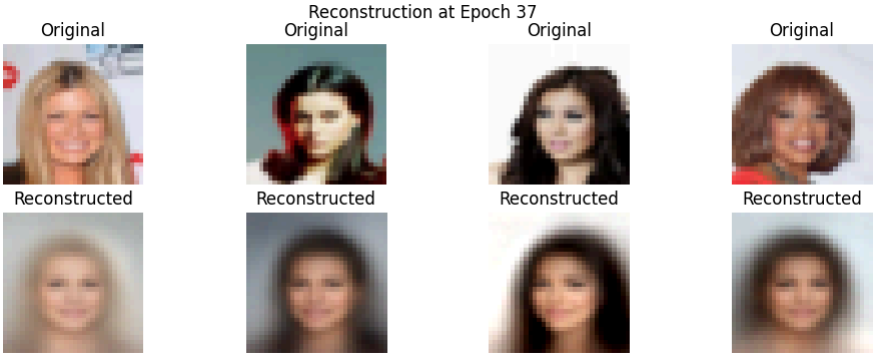
Epoch: 35/50, Avg Loss: 0.10126044723806502, Avg Reconstruction Loss: 0.09091418604307537, Avg KL Loss: 8473.412319521361, Beta: 1.2210265472743261e-06

Reconstruction at Epoch 36



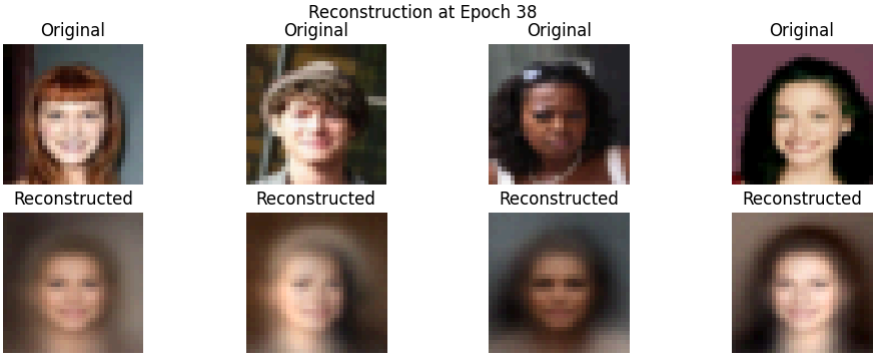
Epoch: 36/50, Avg Loss: 0.10173017597651118, Avg Reconstruction Loss: 0.09102917764383027, Avg KL Loss: 7977.7944614072385, Beta: 1.3413478976398633e-06

Reconstruction at Epoch 37

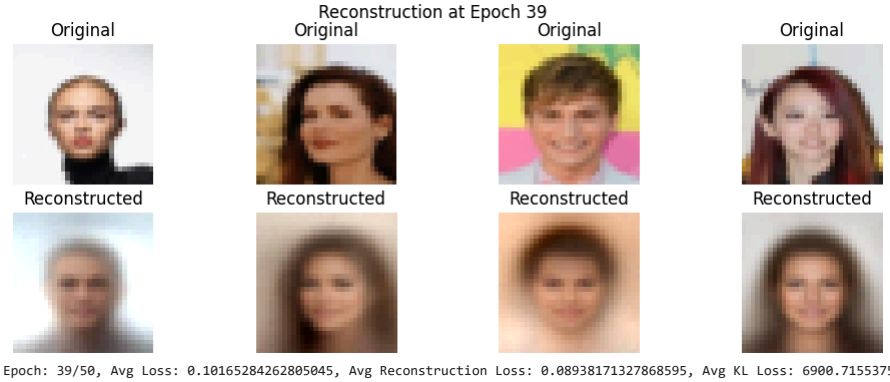


Epoch: 37/50, Avg Loss: 0.10144803201473213, Avg Reconstruction Loss: 0.0900460064788408, Avg KL Loss: 7737.920748553699, Beta: 1.4735258512759055e-06

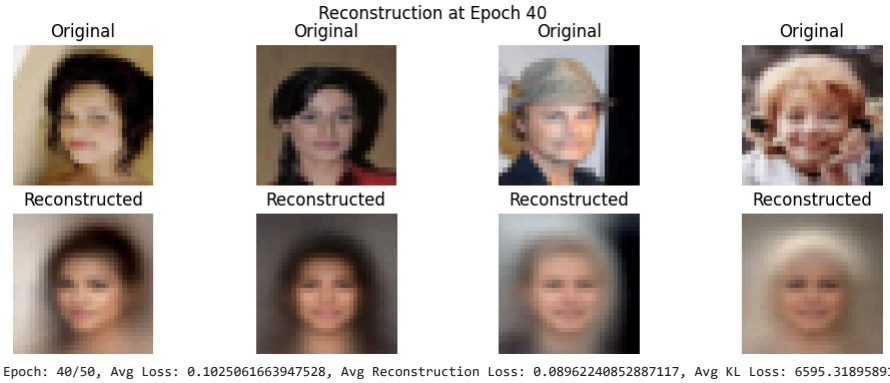
Reconstruction at Epoch 38



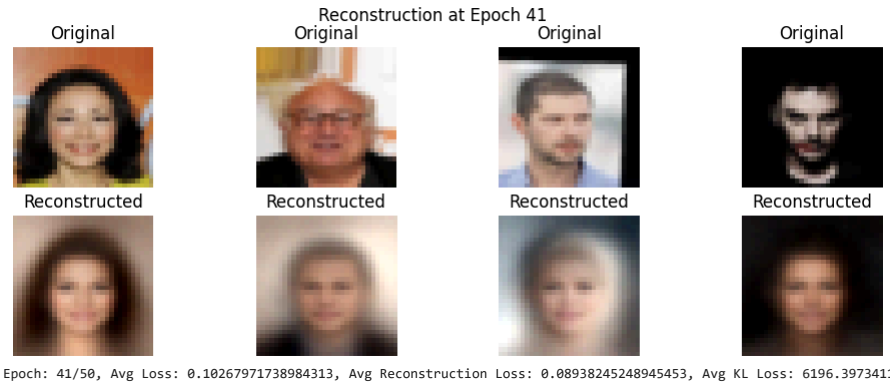
Epoch: 38/50, Avg Loss: 0.10154075524475001, Avg Reconstruction Loss: 0.08965122435666338, Avg KL Loss: 7344.980442481705, Beta: 1.6187287714088219e-06



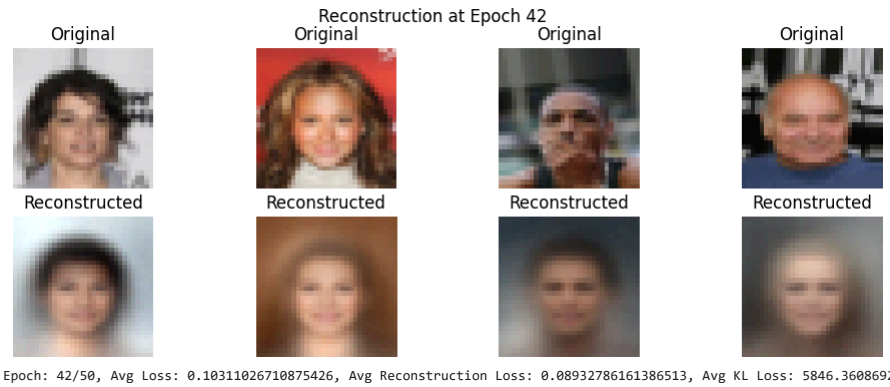
Epoch: 39/50, Avg Loss: 0.10165284262805045, Avg Reconstruction Loss: 0.08938171327868595, Avg KL Loss: 6900.715537542029, Beta: 1.7782401531115642e-06



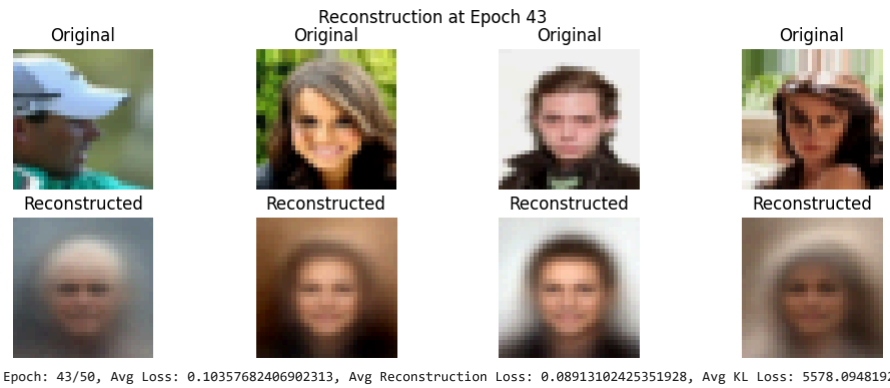
Epoch: 40/50, Avg Loss: 0.1025061663947528, Avg Reconstruction Loss: 0.08962240852887117, Avg KL Loss: 6595.318958910206, Beta: 1.953469968527308e-06



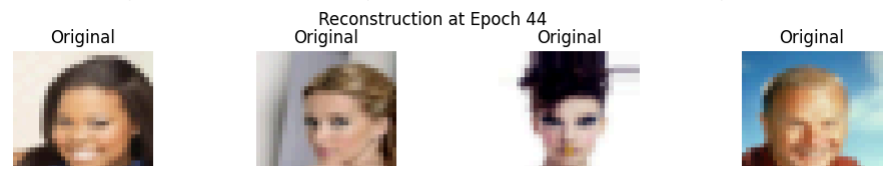
Epoch: 41/50, Avg Loss: 0.10267971738984313, Avg Reconstruction Loss: 0.08938245248945453, Avg KL Loss: 6196.397341184978, Beta: 2.145967130064388e-06

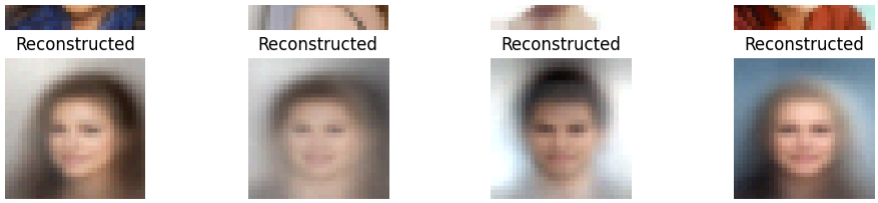


Epoch: 42/50, Avg Loss: 0.10311026710875426, Avg Reconstruction Loss: 0.08932786161386513, Avg KL Loss: 5846.360869202433, Beta: 2.3574331817286995e-06



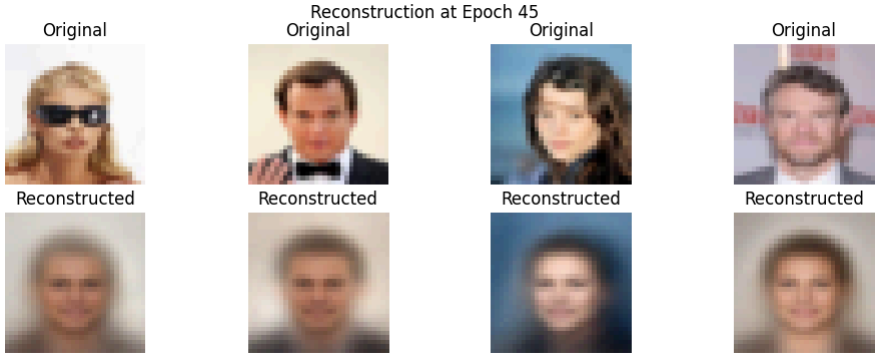
Epoch: 43/50, Avg Loss: 0.10357682406902313, Avg Reconstruction Loss: 0.08913102425351928, Avg KL Loss: 5578.09481927413, Beta: 2.589737339615603e-06





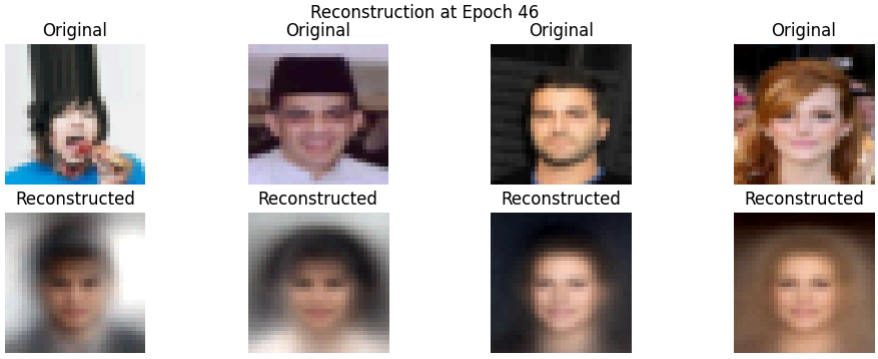
Epoch: 44/50, Avg Loss: 0.10491310512717766, Avg Reconstruction Loss: 0.09000068459706971, Avg KL Loss: 5241.747402529173, Beta: 2.8449330145091504e-06

Reconstruction at Epoch 45



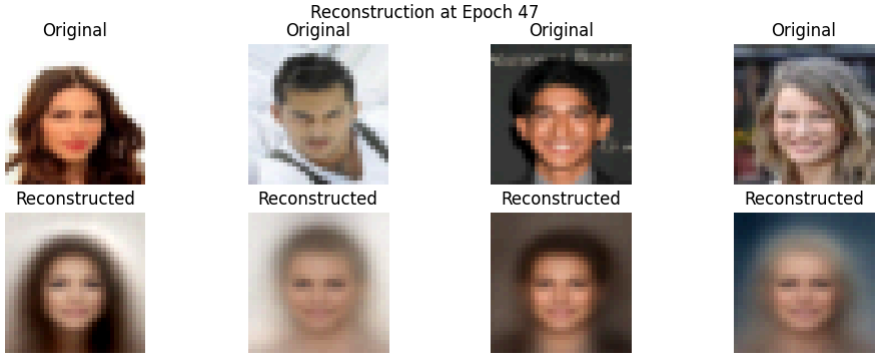
Epoch: 45/50, Avg Loss: 0.10543672844201704, Avg Reconstruction Loss: 0.0898341863781591, Avg KL Loss: 4992.37265130538, Beta: 3.1252759626369885e-06

Reconstruction at Epoch 46



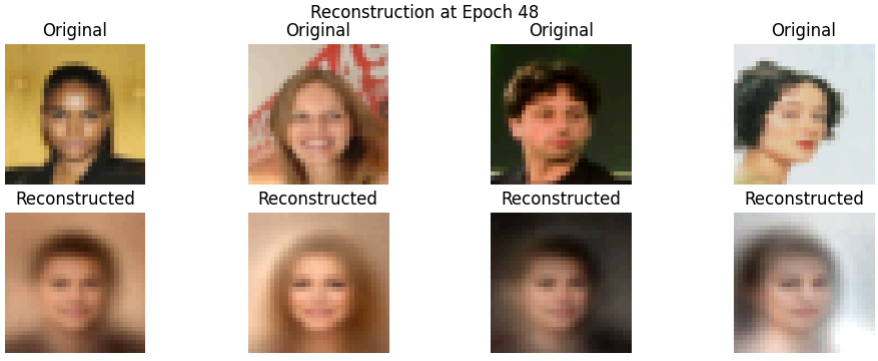
Epoch: 46/50, Avg Loss: 0.10511342549248587, Avg Reconstruction Loss: 0.08923739779599105, Avg KL Loss: 4624.205597310127, Beta: 3.433244225021502e-06

Reconstruction at Epoch 47



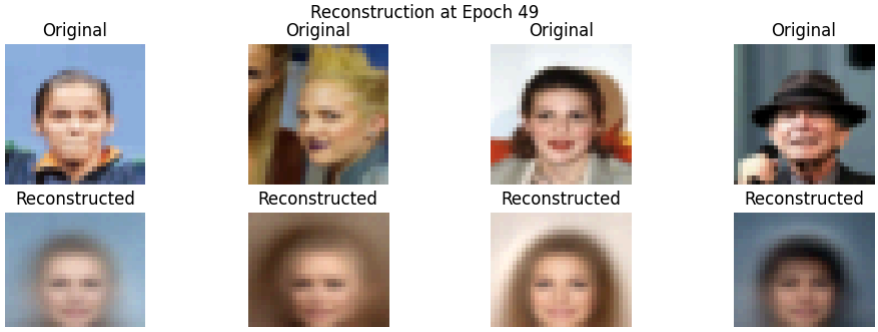
Epoch: 47/50, Avg Loss: 0.1065768726830241, Avg Reconstruction Loss: 0.08981620369455483, Avg KL Loss: 4443.96192410626, Beta: 3.77156003167730 96e-06

Reconstruction at Epoch 48

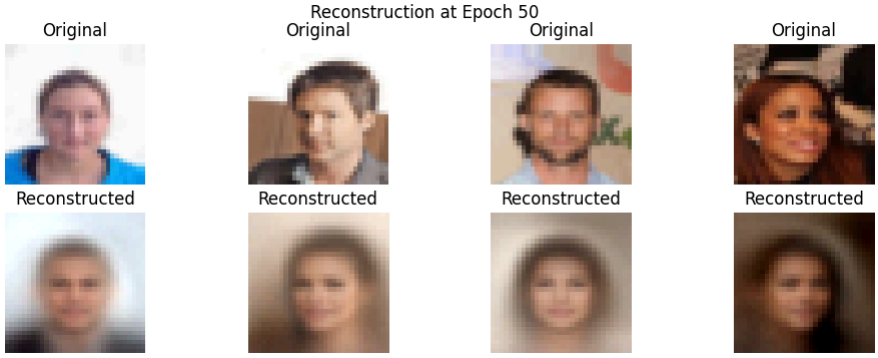


Epoch: 48/50, Avg Loss: 0.10733948185851303, Avg Reconstruction Loss: 0.08982311585281469, Avg KL Loss: 4227.72417363034, Beta: 4.143213864273422e-06

Reconstruction at Epoch 49

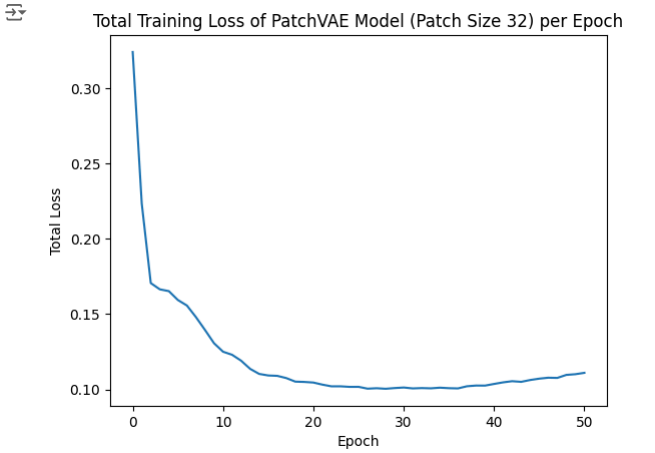


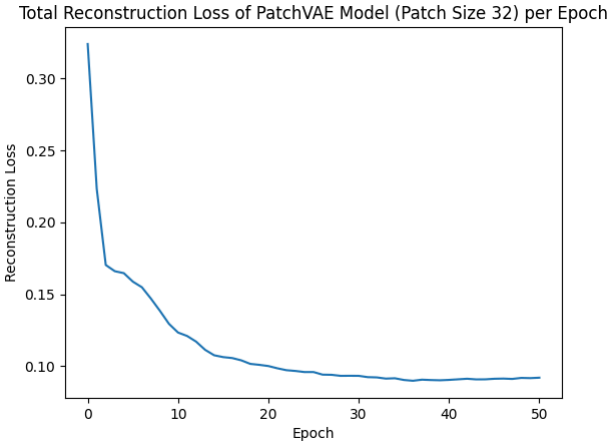
Epoch: 49/50, Avg Loss: 0.10891151560258262, Avg Reconstruction Loss: 0.09074912984159929, Avg KL Loss: 3990.4257572994957, Beta: 4.551490889957609e-06

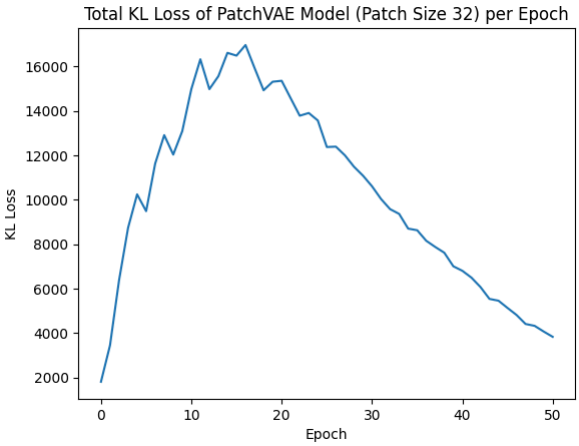


Epoch: 50/50, Avg Loss: 0.11039666595715511, Avg Reconstruction Loss: 0.09147284679774996, Avg KL Loss: 3784.7639789822733, Beta: 4.999999999999999e-06

```
# Plot Loss Curves for Patch Size 32

epoch = np.arange(0, num_epochs+1)
plt.plot(epoch, losses)
plt.xlabel('Epoch')
plt.ylabel('Total Loss')
plt.title('Total Training Loss of PatchVAE Model (Patch Size 32) per Epoch')
plt.savefig('total_loss.png')
plt.show()

plt.plot(epoch, recon_losses)
plt.xlabel('Epoch')
plt.ylabel('Reconstruction Loss')
plt.title('Total Reconstruction Loss of PatchVAE Model (Patch Size 32) per Epoch')
plt.savefig('recon_loss.png')
plt.show()

plt.plot(epoch, kl_losses)
plt.xlabel('Epoch')
plt.ylabel('KL Loss')
plt.title('Total KL Loss of PatchVAE Model (Patch Size 32) per Epoch')
plt.savefig('kl_loss.png')
plt.show()
```





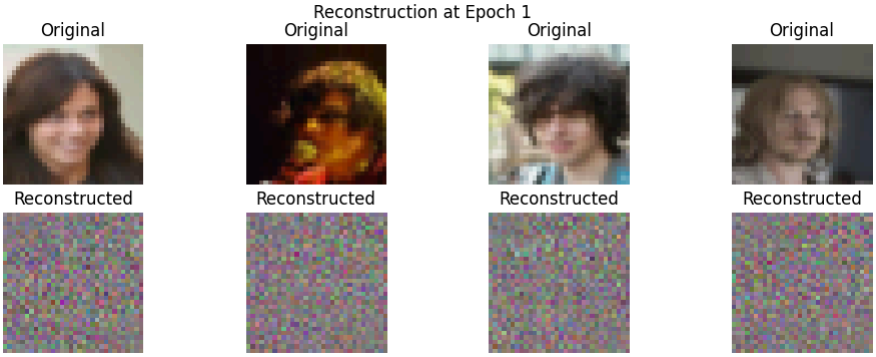



```
# Train PatchVAE for Patch Size 8
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
resume_training = False
checkpoint_path = 'best_vae_model.pth'
num_epochs = 50


vae = PatchVAE(patch_size=8, img_channels=3, img_size=32, embed_dim=1024, latent_dim=512).to(device)
optimizer = optim.Adam(vae.parameters(), lr=1e-4)

if resume_training:
    vae.load_state_dict(torch.load(checkpoint_path))
```

```
losses, recon_losses, kl_losses = train_patchvae(vae, dataloader, optimizer, device, epochs=num_epochs, checkpoint_path='best_vae_model.pth')
```

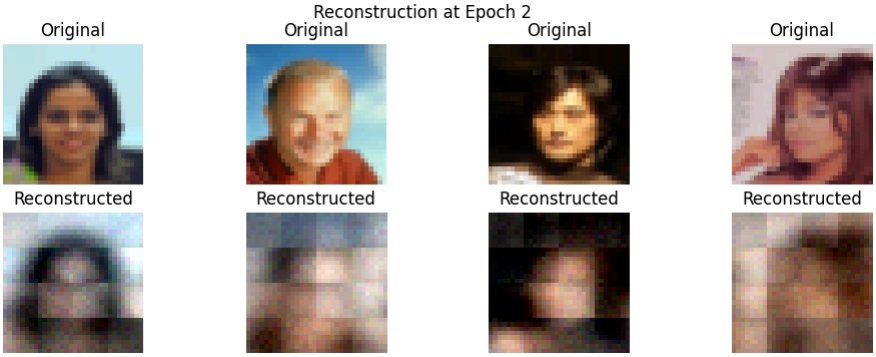
```
Beginning of Training, Loss: 0.37135177850723267, Reconstruction Loss: 0.3668542802333832, KL Loss: 89950.2421875
```

Reconstruction at Epoch 1



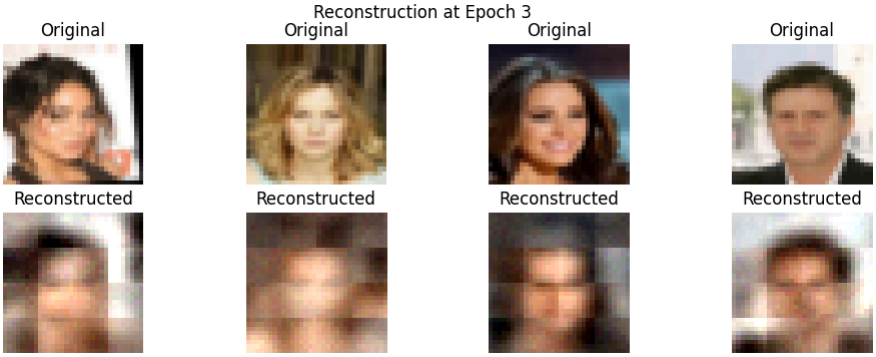

```
Epoch: 1/50, Avg Loss: 0.13431762602133088, Avg Reconstruction Loss: 0.13005228755594808, Avg KL Loss: 85306.77009988132, Beta: 5.000000000000006e-08
```

Reconstruction at Epoch 2



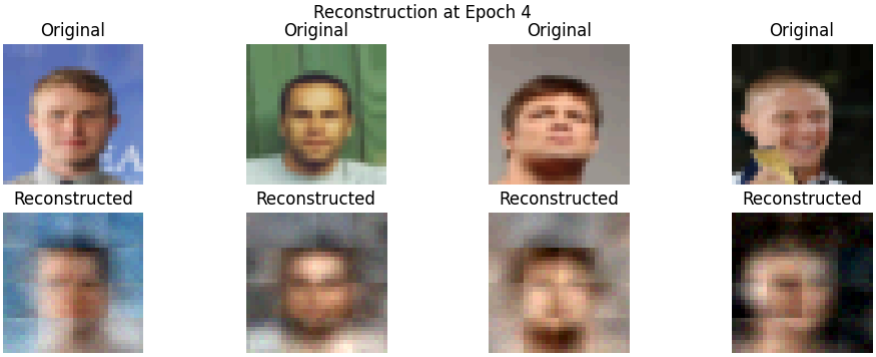

```
Epoch: 2/50, Avg Loss: 0.06810231912362424, Avg Reconstruction Loss: 0.0625806319751317, Avg KL Loss: 100527.63533425632, Beta: 5.492705709937796e-08
```

Reconstruction at Epoch 3



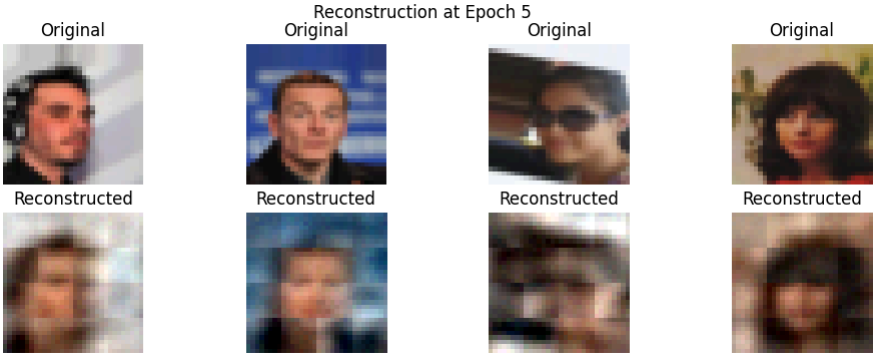

```
Epoch: 3/50, Avg Loss: 0.05862858226593537, Avg Reconstruction Loss: 0.05223259070440184, Avg KL Loss: 105999.84923852848, Beta: 6.033963203196648e-08
```

Reconstruction at Epoch 4




```
Epoch: 4/50, Avg Loss: 0.05258307346626173, Avg Reconstruction Loss: 0.045544453719748725, Avg KL Loss: 106186.31345183939, Beta: 6.628556827950549e-08
```

Reconstruction at Epoch 5




```
Epoch: 5/50, Avg Loss: 0.049597273899030084, Avg Reconstruction Loss: 0.04140519619553904, Avg KL Loss: 112501.61622329905, Beta: 7.281742387506222e-08
```

Reconstruction at Epoch 6