Title: Generating Images using conditional Variational Autoencoders

Abstract: This project aims to utilize conditional Variational Autoencoders (cVAEs) to generate new images based on input labels. The project uses an unsupervised learning approach to generate images by encoding the images into a latent space and then decoding them to produce new images. The primary objective of this project is to evaluate the effectiveness of VAEs for generating images and explore their potential applications.

Introduction:

In this project, we generated images of various datasets based on input class labels. Such generative tasks can be accomplished with a multitude of generative models, such as VAEs, GANs, AutoRegressive models, and PixelRNN/PixelCNNs, each with its own advantages and disadvantages. Through this project, we plan to use VAEs, mainly because we hope to explore and evaluate the performance of VAEs for such generative tasks, but also because VAEs are great at generating diverse, high-quality images with relatively low computational requirements.

VAEs are neural network architectures used for unsupervised learning tasks, including dimensionality reduction, data compression, and data generation. They are a useful tool for generating images, the goal of this project, and are part of a growing body of research on generative models in machine learning.

In an autoencoder, there exist 2 components: an encoder, which maps the input data into a lower dimensional latent space, and a decoder, which maps the latent space produced by the encoder back to the original space.

VAEs extend this basic autoencoder by adding a probabilistic component to the latent space. Rather than mapping input data directly to the latent space, the encoder maps it to a probability distribution over the latent space. The decoder then maps samples from this distribution back to the original space, creating a reconstruction of the input data. Its primary objective is to learn the parameters of the encoder and decoder so that the generated samples match the distribution of the input data.

What this means is that VAEs enable the generation of new data by sampling from the learned latent space. Through sampling from the probability distribution of the latent space, VAEs can generate new samples that are similar to training data, but not identical to any specific training example.

For our task, we wish to generate images based on input class labels. Thus, we need conditional VAEs. This is a quite simple variation of a VAE where we include the class labels during training and prediction. When provided with the target class labels to generate, the model generates an image conditioned on those class labels, with some random variation due to the probability distributions.

Through creating conditional VAEs for conditional image generation, we will evaluate the performance of VAEs for such tasks.

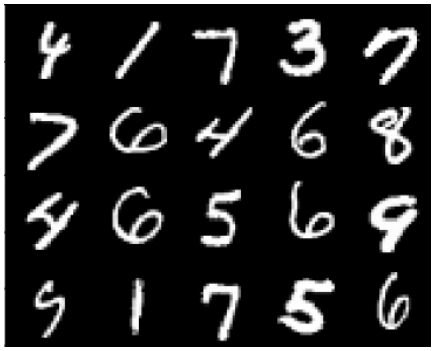
Datasets:

We used two datasets to train our models: the MNIST handwritten digits dataset and the CelebA dataset.

MNIST:

The MNIST dataset consists of 70,000 grayscale images of handwritten digits. Out of these images, 60,000 are used for training purposes, and the remaining 10,000 are used for testing. Each image in the dataset is 28 x 28 pixels in size. The images contain handwritten digits ranging from 0 to 9.

Here are some example images:



CelebA:

The CelebA dataset contains 202,599 color images of celebrities' faces. The images are divided into three sets for training, validation, and testing purposes. The training set consists of 162,770 images, while the validation set and testing set contain 19,867 and 19,962 images, respectively. Each image in the dataset is cropped and aligned to have a size of 178 x 218 pixels. The images are of various celebrities' faces, and they can be used for various facial recognition and analysis tasks.

Here are some example images:

Part 1: Autoencoder:

We first implemented a simple Autoencoder for reconstructing the images.

Architecture**:**

Our implementation uses 3 convolutional layers for the encoder and 3 transpose convolutional layers for the decoder. We use the relu activation function between layers of both the encoder and decoder and the sigmoid activation function at the end of the decoder.

Here is our implementation in Pytorch:

```python
class Encoder(nn.Module):
    def __init__(self, encoded_space_dim):
        super().__init__()
        self.encoder_cnn = nn.Sequential(
            nn.Conv2d(1, 8, 3, stride=2, padding=1),
            nn.ReLU(True),
            nn.Conv2d(8, 16, 3, stride=2, padding=1),
            nn.BatchNorm2d(16),
            nn.ReLU(True),
            nn.Conv2d(16, 32, 3, stride=2, padding=0),
            nn.ReLU(True)
        )
        self.flatten = nn.Flatten(start_dim=1)
        self.encoder_lin = nn.Sequential(
            nn.Linear(3 * 3 * 32, 128),
            nn.ReLU(True),
            nn.Linear(128, encoded_space_dim)
        )

    def forward(self, x):
        x = self.encoder_cnn(x)
        x = self.flatten(x)
        x = self.encoder_lin(x)
        return x
```

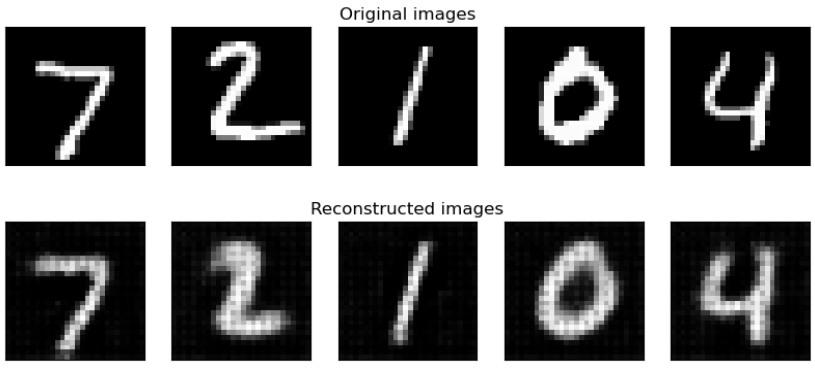
```python
class Decoder(nn.Module):
    def __init__(self, encoded_space_dim):
        super().__init__()
        self.decoder_lin = nn.Sequential(
            nn.Linear(encoded_space_dim, 128),
            nn.ReLU(True),
            nn.Linear(128, 3 * 3 * 32),
            nn.ReLU(True)
        )
        self.unflatten = nn.Unflatten(dim=1, unflattened_size=(32, 3, 3))
        self.decoder_conv = nn.Sequential(
            nn.ConvTranspose2d(32, 16, 3, stride=2, output_padding=0),
            nn.BatchNorm2d(16),
            nn.ReLU(True),
            nn.ConvTranspose2d(16, 8, 3, stride=2, padding=1, output_padding=1),
            nn.BatchNorm2d(8),
            nn.ReLU(True),
            nn.ConvTranspose2d(8, 1, 3, stride=2, padding=1, output_padding=1)
        )

    def forward(self, x):
        x = self.decoder_lin(x)
        x = self.unflatten(x)
        x = self.decoder_conv(x)
        x = torch.sigmoid(x)
        return x
```

Learning and Hyperparameters:

After some moderate testing, we found that the model produced the best results with a batch size of 512 and a latent space dimension of 4. We used the Adam optimizer with a learning rate of .001 and weight decay rate of .00001 to train the model. We trained the model for 15 epochs. We used reconstruction error as our loss in order to optimize how closely the model reconstructs the images from the latent space.

Results:

We ran the autoencoder on the MNIST digit dataset. Here are the results:



Part 2: Variational Autoencoder:

Once we had a working autoencoder, we needed to make it variational in order to achieve a generative model.

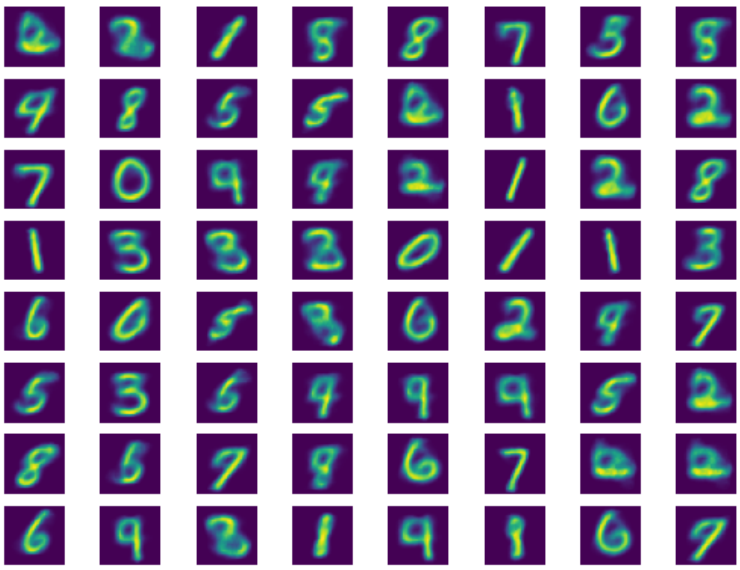
Architecture**:**

Our implementation uses linear layers for simplicity, as this wasn't the main focus of our project. We used 4 linear layers for the encoder (3 sequentially, but we had a unique layer for mu and log_var), and 3 linear layers for decoder. We use the relu activation function between layers of both the encoder and decoder and the sigmoid activation function at the end of the decoder. Here is our implementation in Pytorch:

```
class VAE(nn.Module):
    def __init__(self, x_dim, h_dim1, h_dim2, z_dim):
        super(VAE, self).__init__()

        # encoder part
        self.fc1 = nn.Linear(x_dim, h_dim1)
        self.fc2 = nn.Linear(h_dim1, h_dim2)
        self.fc31 = nn.Linear(h_dim2, z_dim)
        self.fc32 = nn.Linear(h_dim2, z_dim)
        # decoder part
        self.fc4 = nn.Linear(z_dim, h_dim2)
        self.fc5 = nn.Linear(h_dim2, h_dim1)
        self.fc6 = nn.Linear(h_dim1, x_dim)

    def encoder(self, x):
        h = F.relu(self.fc1(x))
        h = F.relu(self.fc2(h))
        return self.fc31(h), self.fc32(h) # mu, log_var

    def sampling(self, mu, log_var):
        std = torch.exp(0.5*log_var)
        eps = torch.randn_like(std)
        return eps.mul(std).add_(mu) # return z sample

    def decoder(self, z):
        h = F.relu(self.fc4(z))
        h = F.relu(self.fc5(h))
        return F.sigmoid(self.fc6(h))
```

Learning and Hyperparameters:
After moderate testing, we found that the model produced the best results with a batch size of 512 and a latent space size of 20. We used a hidden dimension size of 32. We used the Adam optimizer with a learning rate of .001 to train the model. We trained the model for 10 epochs on the MNIST dataset. We used a combination of reconstruction error and KL divergence as our loss in order to optimize how closely the model reconstructs the images from the latent space as well as how closely the variance in our model's generated images match our target distribution (specifically, a Guassian distribution).

Results:
The following figure shows the results of generating 64 random images from the VAE.

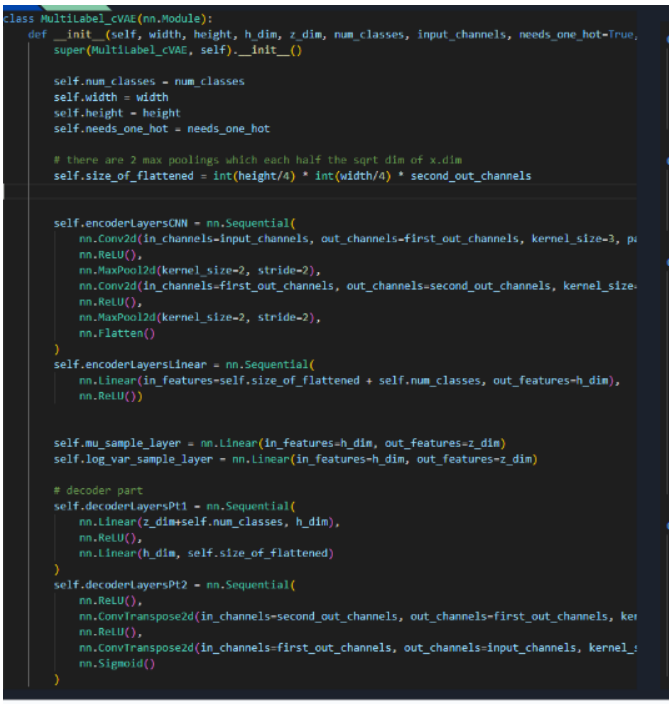


Part 3: Conditional Variational Autoencoder:

Once we had a generative model, we could now make the model conditional on the class labels in order to produce images with specific labels on command. This is what our project was mainly focused on, so we put extra effort into the architecture of the cVAE and tuning the hyperparameters.

Architecture**:**

Our implementation uses 2 convolutional layers for the encoder and 2 transpose convolutional layers for the decoder. We use the relu activation function between layers of both the encoder and decoder and the sigmoid activation function at the end of the decoder. Here is our implementation in Pytorch:



```
class MultiLabel_cVAE(nn.Module):
    def __init__(self, width, height, h_dim, z_dim, num_classes, input_channels, needs_one_hot=True,
        super(MultiLabel_cVAE, self).__init__()

        self.num_classes = num_classes
        self.width = width
        self.height = height
        self.needs_one_hot = needs_one_hot

        # there are 2 max poolings which each half the sqrt dim of x.dim
        self.size_of_flattened = int(height/4) * int(width/4) * second_out_channels


        self.encoderLayersCNN = nn.Sequential(
            nn.Conv2d(in_channels=input_channels, out_channels=first_out_channels, kernel_size=3, pa
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Conv2d(in_channels=first_out_channels, out_channels=second_out_channels, kernel_size=
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Flatten()
        )
        self.encoderLayersLinear = nn.Sequential(
            nn.Linear(in_features=self.size_of_flattened + self.num_classes, out_features=h_dim),
            nn.ReLU())


        self.mu_sample_layer = nn.Linear(in_features=h_dim, out_features=z_dim)
        self.log_var_sample_layer = nn.Linear(in_features=h_dim, out_features=z_dim)

        # decoder part
        self.decoderLayersPt1 = nn.Sequential(
            nn.Linear(z_dim+self.num_classes, h_dim),
            nn.ReLU(),
            nn.Linear(h_dim, self.size_of_flattened)
        )
        self.decoderLayersPt2 = nn.Sequential(
            nn.ReLU(),
            nn.ConvTranspose2d(in_channels=second_out_channels, out_channels=first_out_channels, ker
            nn.ReLU(),
            nn.ConvTranspose2d(in_channels=first_out_channels, out_channels=input_channels, kernel_s
            nn.Sigmoid()
        )
```

```
def encoder(self, x, c_one_hot):
    h = self.encoderLayersCNN(x)
    concat = torch.cat([h, c_one_hot], dim=1)
    h = self.encoderLayersLinear(concat)
    return self.mu_sample_layer(h), self.log_var_sample_layer(h) # mu, log_var

def sampling(self, mu, log_var):
    std = torch.exp(0.5*log_var)
    eps = torch.randn_like(std)
    return eps.mul(std).add_(mu) # return z sample

def decoder(self, z, c_one_hot):
    zc = torch.cat([z, c_one_hot], dim=1)
    x = self.decoderLayersPt1(zc)
    pixelsComponent = x.shape[1] / second_out_channels
    width_component = int(self.width/4)
    height_component = int(self.height/4)

    #width_component = int(pixelsComponent * self.width_percentage)
    #height_component = int(pixelsComponent-width_component)
    x = x.view(-1, second_out_channels, width_component, height_component)
    x = self.decoderLayersPt2(x)
    return x

def forward(self, x, c):
    c_onehot = c
    if self.needs_one_hot:
        c_onehot = torch.nn.functional.one_hot(c, num_classes=self.num_classes)
    # flatten
    mu, log_var = self.encoder(x, c_onehot)
    z = self.sampling(mu, log_var)
    return self.decoder(z, c_onehot), mu, log_var
```

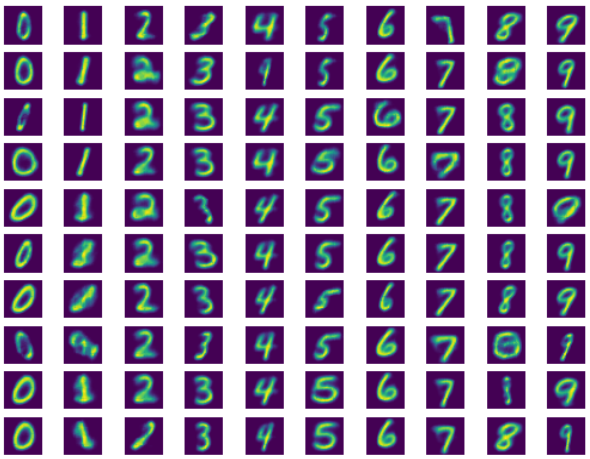
Learning and Hyperparameters:

After moderate testing, we found that the model produced the best results with a batch size of 100 and a latent space size of 2. We used a hidden dimension size of 512 on both datasets. We used the Adam optimizer with a learning rate of .001 to train the model. We trained the model for 15 and 2 epochs on the MNIST and CelebA datasets respectively. Just as before with our VAE, we used a combination of reconstruction error and KL divergence as our loss.

Results:

MNIST dataset:

The following figure displays the images that the model generated when told to generate digits 0-9 10 times. The subsequent figure shows the code used to generate the FID (Frechet Inception Distance) score on the MNIST dataset and the resulting score (182.47). We used the pytorch_fid package to calculate the FID score.



```
#Calculate FID for MNIST dataset
mnist_train_dataset = datasets.MNIST(root=dataset, train=True,
transform=transforms.ToTensor(), download=True)

# Store 10 train images into a folder
for i in range(10):
    train_image = (torchvision.utils.make_grid
    (mnist_train_dataset[i][0], nrow=1, padding=0))
    torchvision.utils.save_image(train_image,
    'mnist_train_images/train_image_{}.png'.format(i))

# Generate 10 images from MNIST vae
for (i, img) in enumerate(generated_images):
    torchvision.utils.save_image(img, 'mnist_generated_images/
    generated_image_{}.png'.format(i))
```

✓ 0.0s Python

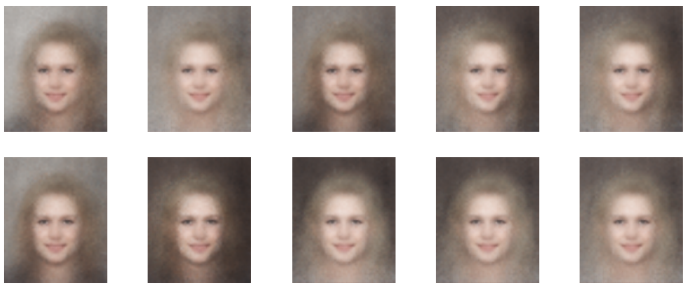
```
fid_value = fid_score.calculate_fid_given_paths
(['mnist_train_images', 'mnist_generated_images'],
batch_size=10, dims=2048, device = device)
print(fid_value)
```

✓ 24.6s Python

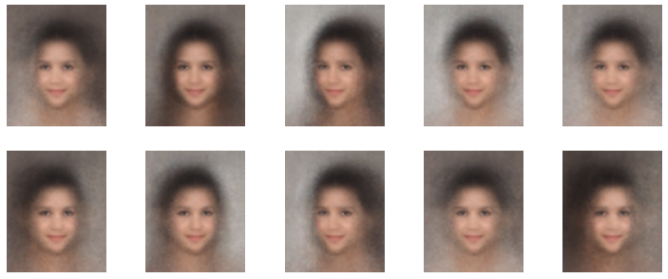
```
100%|██████████| 1/1 [00:06<00:00,  6.60s/it]
100%|██████████| 10/10 [00:11<00:00,  1.10s/it]
182.4744605316847
```

CelebA dataset:

The following figures display outputs of our model when told to generate 10 images with the corresponding class labels specified below the results (None means false, 1 means true). The subsequent figure shows the code used to generate the FID score on the CelebA dataset and the resulting score (379.94). We used the pytorch_fid package to calculate the FID score.



```
{'5_o_Clock_Shadow': None, 'Arched_Eyebrows': None, 'Attractive': 1,
 'Bags_Under_Eyes': None, 'Bald': None, 'Bangs': None, 'Big_Lips': None,
 'Big_Nose': None, 'Black_Hair': None, 'Blond_Hair': 1, 'Blurry': None,
 'Brown_Hair': None, 'Bushy_Eyebrows': None, 'Chubby': None, 'Double_Chin': None,
 'Eyeglasses': None, 'Goatee': 1, 'Gray_Hair': None, 'Heavy_Makeup': None,
 'High_Cheekbones': None, 'Male': None, 'Mouth_Slightly_Open': None, 'Mustache': None,
 'Narrow_Eyes': None, 'No_Beard': 1, 'Oval_Face': None, 'Pale_Skin': 1, 'Pointy_Nose': None,
 'Receding_Hairline': None, 'Rosy_Cheeks': 1, 'Sideburns': None, 'Smiling': 1,
 'Straight_Hair': 1, 'Wavy_Hair': None, 'Wearing_Earrings': 1, 'Wearing_Hat': None,
 'Wearing_Lipstick': 1, 'Wearing_Necklace': None, 'Wearing_Necktie': None, 'Young': 1}
```



```
{'5_o_Clock_Shadow': None, 'Arched_Eyebrows': None, 'Attractive': 1,
 'Bags_Under_Eyes': None, 'Bald': None, 'Bangs': None, 'Big_Lips': None,
 'Big_Nose': None, 'Black_Hair': 1, 'Blond_Hair': None, 'Blurry': None,
 'Brown_Hair': None, 'Bushy_Eyebrows': None, 'Chubby': None, 'Double_Chin': None,
 'Eyeglasses': None, 'Goatee': 1, 'Gray_Hair': None, 'Heavy_Makeup': None,
 'High_Cheekbones': None, 'Male': None, 'Mouth_Slightly_Open': None, 'Mustache': None,
 'Narrow_Eyes': None, 'No_Beard': 1, 'Oval_Face': None, 'Pale_Skin': None, 'Pointy_Nose': None,
 'Receding_Hairline': None, 'Rosy_Cheeks': 1, 'Sideburns': None, 'Smiling': 1,
 'Straight_Hair': 1, 'Wavy_Hair': None, 'Wearing_Earrings': None, 'Wearing_Hat': None,
 'Wearing_Lipstick': 1, 'Wearing_Necklace': None, 'Wearing_Necktie': None, 'Young': 1}
```

```
{'5_o_Clock_Shadow': None, 'Arched_Eyebrows': None, 'Attractive': None,
 'Bags_Under_Eyes': None, 'Bald': None, 'Bangs': None, 'Big_Lips': None,
 'Big_Nose': None, 'Black_Hair': 1, 'Blond_Hair': None, 'Blurry': None,
 'Brown_Hair': None, 'Bushy_Eyebrows': None, 'Chubby': None, 'Double_Chin': None,
 'Eyeglasses': None, 'Goatee': 1, 'Gray_Hair': None, 'Heavy_Makeup': None,
 'High_Cheekbones': None, 'Male': 1, 'Mouth_Slightly_Open': None, 'Mustache': 1,
 'Narrow_Eyes': None, 'No_Beard': 1, 'Oval_Face': None, 'Pale_Skin': None, 'Pointy_Nose': None,
 'Receding_Hairline': None, 'Rosy_Cheeks': 1, 'Sideburns': None, 'Smiling': 1,
 'Straight_Hair': 1, 'Wavy_Hair': None, 'Wearing_Earrings': None, 'Wearing_Hat': None,
 'Wearing_Lipstick': None, 'Wearing_Necklace': None, 'Wearing_Necktie': None, 'Young': 1}
```

```
#Evaluate Results using Fréchet Inception Distance (FID)
from pytorch_fid import fid_score
import torchvision
import torchvision.transforms as transforms

# Store 10 train images into a folder
for i in range(10):
    train_image = (torchvision.utils.make_grid
    (celeb_train_dataset[i][0], nrow=1, padding=0))
    torchvision.utils.save_image(train_image, 'train_images/
    train_image_{}.png'.format(i))

# Save the generated images to files for fid calculation
for i in range(num_images):
    torchvision.utils.save_image(images[i], 'generated_images/
    generated_image_{}.png'.format(i))

# Calculate FID
fid_value = fid_score.calculate_fid_given_paths
(['train_images', 'generated_images'], batch_size=10,
dims=2048, device = device)
print(fid_value)
```

✓ 28.6s Python

```
100%|██████████| 1/1 [00:06<00:00,  6.74s/it]
100%|██████████| 1/1 [00:06<00:00,  6.51s/it]
379.0195467765217
```

Experiment Results:

In our project, we started by building a basic auto encoder using linear layers, which allowed us to compress the input data and then reconstruct it with minimal loss. For this experiment, we used the MNIST dataset of handwritten numbers. Surprisingly, this simple architecture performed well, achieving high accuracy in mostly reconstructing the original input. We experimented with latent space sizes and found that a latent space of 2 was the smallest that would allow a reasonable reconstruction, allowing us to reduce the dimensionality of the input significantly while still retaining the essential information. Overall, our results demonstrated the effectiveness of using autoencoders for dimensionality reduction and reconstruction tasks, even with simple architectures.

Building upon our initial success with the basic auto encoder, we decided to explore the use of a variational autoencoder (VAE) by adding two extra layers for mu and sigma. This enabled us to not only compress the input data but also generate new data samples by sampling from a learned distribution. We trained the VAE on our dataset and manually inspected the images produced by the model. We found that the generated images had reasonable results, with some showing interesting variations and new patterns that were not present in the original dataset. This confirmed the effectiveness of the VAE architecture in generating novel data samples and exploring the latent space of the input data.

To make our Variational Autoencoder (VAE) conditional, we added feature labels as input to the encoder and decoder. These feature labels could be in the form of a one-hot encoding or a feature vector that specifies the characteristics of the image we want to generate. By adding these labels, we were able to control the generation of images by specifying which features should be present in the generated image. For example, when working with the MNIST dataset, we wanted to specify which specific number to be represented. Then, we proceeded to work with a larger dataset with higher resolution images - the CelebA dataset. This dataset contained images 4x the resolution and with 40 different features. Using the cVAE, we were able to generate images of new people with specific features. Overall, this conditional VAE approach allows us to generate more specific and targeted images based on our desired features.

Summary:

In conclusion, our project focused on exploring the capabilities of autoencoders and variational autoencoders (VAEs) for dimensionality reduction, image reconstruction, and generation of new images. We started with a simple autoencoder architecture and achieved excellent results in reconstructing the input data. We then explored the VAE architecture and found that it not only allowed us to compress the data but also generate new data samples by sampling from a learned distribution. We also experimented with making the VAE architecture conditional by adding feature labels as inputs to the encoder and decoder. This allowed us to control the generation of images by specifying which features should be present in the generated image. Our results showed that this approach was effective in generating more specific and targeted images based on desired features. Overall, our experiments demonstrated the power and flexibility of autoencoder and VAE architectures for image-related tasks.

References:

Conditional VAEs: https://ijdykeman.github.io/ml/2016/12/21/cvae.html
CelebA Dataset: https://mmlab.ie.cuhk.edu.hk/projects/CelebA.html
FID: https://en.wikipedia.org/wiki/Fr%C3%A9chet_inception_distance