The Variational Autoencoder

A Deep Dive into Latent Variable Modeling and Manifold Continuity on the CelebAMask-HQ Dataset

Ibrahim Habib

2026-02-10

Table of contents

Informally, a generative model is one that is able to generate new data points where the data could be anything: text, images, audio, etc. This is done through building a model that learns the underlying structure of the data and then using that model to generate new data points.

The challenge is that most interesting data is high-dimensional. Modeling every possible pixel combination in an image is impossible because the number of possibilities is astronomically large. There are multiple techniques to handle this problem. One way is a latent variable model that simplifies the generation by focusing on a set of high-level features that are easier to model.

In this post, my goal is to demonstrate the inner workings of the Variational Autoencoder (VAE), which is a popular type of latent variable model. Instead of seeing it as just some other neural network architecture, we will view it from a probabilistic perspective and see where neural networks come into play. We will also implement a VAE from scratch using PyTorch and train it on the CelebAMask-HQ Dataset to generate new human faces and morph between existing ones. In the end we will see how the VAE is able to learn a smooth latent space and will rely on this property to create a morphing animation between two faces.

You can access the code for this project in this GitHub repository: https://github.com/ibrahimhabibeg/vae-faces.

1 Mathematical Perspective on Generative Models

Informally, a generative model is one that is able to generate new data points similar to the ones in the training set. Usually, the data we are talking about is high-dimensional, such as text, images, audio, etc.

Formally stated, a generative model is a probabilistic model $p_\theta(\mathbf{x})$ that attempts to approximate the true data distribution $p_{data}(\mathbf{x})$ from a training set $\mathcal{D} \sim p_{data}(\mathbf{x})$[1]. Such a model is useful for multiple objectives. Two of the most common ones are:

1. **Data Generation (Sampling)**: Extracting new data points by sampling $\mathbf{x}_{\text{new}} \sim p_\theta(\mathbf{x})$.
2. **Density Estimation**: Evaluating the likelihood of a data point under the model, i.e., computing $p_\theta(\mathbf{x})$ for a given $\mathbf{x}$. This is useful for anomaly detection.

Although they aren't the only option, usually generative models are parametric. For this case, our goal is to find the optimal parameters θ such that $p_\theta(\mathbf{x})$ is as close as possible to $p_{data}(\mathbf{x})$.

1.1 The Difficulty of Training Generative Models

Since the data is usually high-dimensional, representing $p(\mathbf{x})$ is a difficult task. This is due to the redicolsously large number of values that $\mathbf{x}$ can take.

To demonstrate this, assume that $\mathbf{x}$ is a 1024 x 1024 RGB image. Assuming 8-bit color depth, the number of possible images is $256^{1024\times1024\times3} \approx 10^{7.5\times10^6}$, or more generally, $256^{3\times\text{ num\_pixels}}$. For scale, the number of atoms in the observable universe is estimated to be around 10^{80}.

This is an astronomically large number, making it impossible to directly model $p(\mathbf{x})$[2]. To handle this, we need to find a technique that allows us to estimate $p(\mathbf{x})$.

[1]Usually $\mathbf{x}$ is a vector and that's why boldface is used.

[2]Directly modeling $p(\mathbf{x})$ here means explicitly assigning a probability to each possible $\mathbf{x}$. You can think of this as one large lookup table that maps each possible $\mathbf{x}$ to a probability. This is clearly impossible for high-dimensional data.

Every technique (or model) introduces its own assumptions and bias regarding the structure of $\mathbf{x}$.

2 Latent Variable Models and the VAE

Latent variable models are one class of generative models. They introduce a set of latent variables $\mathbf{z}$. They are also called hidden factors or features because of our inability to directly observe them. In this case, the joint distribution of the data and the latent variables is given by:

$$p(\mathbf{x}, \mathbf{z}) = p(\mathbf{x} \mid \mathbf{z}) p(\mathbf{z}) \tag{1}$$

where $p(\mathbf{z})$ is the prior distribution of the latent variables, and $p(\mathbf{x} \mid \mathbf{z})$ is the likelihood of the data given the latent variables. This factorization allows us to see how data is generated from the latent variables: first sample $\mathbf{z} \sim p(\mathbf{z})$, then sample $\mathbf{x} \sim p(\mathbf{x} \mid \mathbf{z})$.

We can calculate both the prior, $p(\mathbf{z})$, and the likelihood, $p(\mathbf{x} \mid \mathbf{z})$. From them we can drive the marginal distribution $p(x)$ to be

$$p(\mathbf{x}) = \int_{\mathbf{z}} p(\mathbf{x}, \mathbf{z}) d\mathbf{z} = \int_{\mathbf{z}} p(\mathbf{x} \mid \mathbf{z}) p(\mathbf{z}) d\mathbf{z} \tag{2}$$

Although sometimes an analytical solution for this integral exists, in the general case it is intractable. We can't sum up over the infinite number of possible $\mathbf{z}$ values.

To train the model we need to maximize the likelihood of the data we've observed in training, $p(\mathbf{x}_1, ..., \mathbf{x}_N)$, which is given by:[3]

$$p(\mathbf{x}_1, ..., \mathbf{x}_N) = \prod_{i=1}^{N} p(\mathbf{x}_i) = \prod_{i=1}^{N} \int_{\mathbf{z}} p(\mathbf{x}_i \mid \mathbf{z}) p(\mathbf{z}) d\mathbf{z} \tag{3}$$

or equivalently, the log-likelihood:

$$\log p(\mathbf{x}_1, ..., \mathbf{x}_N) = \sum_{i=1}^{N} \log p(\mathbf{x}_i) = \sum_{i=1}^{N} \log \int_{\mathbf{z}} p(\mathbf{x}_i \mid \mathbf{z}) p(\mathbf{z}) d\mathbf{z} \tag{4}$$

where N is the number of data points in the training set and $\mathbf{x}_i$ is the i-th data point.

As we've mentioned earlier, we need a way to estimate the integral in the general case, and this is where wwe introduce the **Variational Autoencoder (VAE)**.

2.1 The Variational Autoencoder (VAE)

Let $q(\mathbf{z})$ be an some distribution over $\mathbf{z}$. We will use it to derive a lower bound on the log-likelihood of the data.

[3]We assume the data points are sampled independently. This allows us to break the joint distribution into a product of individual distributions.

$$
\begin{aligned}
\log p(\mathbf{x}) &= \int_{\mathbf{z}} q(\mathbf{z}) \log p(\mathbf{x}) d\mathbf{z} \\
&= \int_{\mathbf{z}} q(\mathbf{z}) \log \left[\frac{p(\mathbf{x}, \mathbf{z})}{p(\mathbf{z} \mid \mathbf{x})} \right] d\mathbf{z} \\
&= \int_{\mathbf{z}} q(\mathbf{z}) \log \left[\frac{p(\mathbf{x} \mid \mathbf{z}) p(\mathbf{z})}{p(\mathbf{z} \mid \mathbf{x})} \right] d\mathbf{z} \\
&= \int_{\mathbf{z}} q(\mathbf{z}) \log \left[\frac{p(\mathbf{x} \mid \mathbf{z}) p(\mathbf{z})}{p(\mathbf{z} \mid \mathbf{x})} \frac{q(\mathbf{z})}{q(\mathbf{z})} \right] d\mathbf{z} \\
&= \int_{\mathbf{z}} q(\mathbf{z}) \left[\log p(\mathbf{x} \mid \mathbf{z}) - \log \frac{q(\mathbf{z})}{p(\mathbf{z})} + \log \frac{q(\mathbf{z})}{p(\mathbf{z} \mid \mathbf{x})} \right] d\mathbf{z} \\
&= \mathbb{E}_{q(\mathbf{z})}[\log p(\mathbf{x} \mid \mathbf{z})] - \mathbb{E}_{q(\mathbf{z})}\left[\log \frac{q(\mathbf{z})}{p(\mathbf{z})} \right] + \mathbb{E}_{q(\mathbf{z})}\left[\log \frac{q(\mathbf{z})}{p(\mathbf{z} \mid \mathbf{x})} \right] \\
&= \mathbb{E}_{q(\mathbf{z})}[\log p(\mathbf{x} \mid \mathbf{z})] - D_{KL}(q(\mathbf{z}) \parallel p(\mathbf{z})) + D_{KL}(q(\mathbf{z}) \parallel p(\mathbf{z} \mid \mathbf{x}))
\end{aligned}
\tag{5}
$$

where D_{KL} is the Kullback-Leibler (KL) divergence[4]. Notice that the third term depends on $p(\mathbf{z} \mid \mathbf{x})$ which in turn depends on $p(\mathbf{x})$. We want to get rid of $p(\mathbf{x})$.

A useful property of the KL divergence is that it is always non-negative. Hence we know that:

$$
\log p(\mathbf{x}) \geq \mathbb{E}_{q(\mathbf{z})}[\log p(\mathbf{x} \mid \mathbf{z})] - D_{KL}(q(\mathbf{z}) \parallel p(\mathbf{z})) \tag{6}
$$

The right hand side of the inequality is called the **Evidence Lower Bound (ELBO)**. We can indirectly maximize $\log p(\mathbf{x})$ by maximizing the ELBO. Because the ELBO is the sum of two expectations, we can estimate it using Monte Carlo sampling.

Keep in mind that there is a deviation between the ELBO and $\log p(\mathbf{x})$ given by $D_{KL}(q(\mathbf{z}) \parallel p(\mathbf{z} \mid \mathbf{x}))$. This means that maximizing the ELBO is not exactly equivalent to maximizing $\log p(\mathbf{x})$, but it is a good approximation. It will be an exact approximation if $q(\mathbf{z})$ is equal to $p(\mathbf{z} \mid \mathbf{x})$.

$p(\mathbf{x} \mid \mathbf{z})$ is usually parameterized by a neural network with parameters θ.

In the beginning, we said $q(\mathbf{z})$ is some distribution over $\mathbf{z}$. If we consider an amortized variational posterior, that is $q(\mathbf{z}) = q_\phi(\mathbf{z} \mid \mathbf{x})$, we get[5].

$$
\mathrm{ELBO} = \mathbb{E}_{q_\phi(\mathbf{z} \mid \mathbf{x})}[\log p_\theta(\mathbf{x} \mid \mathbf{z})] - \mathbb{E}_{q_\phi(\mathbf{z} \mid \mathbf{x})}\left[\log q_\phi(\mathbf{z} \mid \mathbf{x})\right] + \mathbb{E}_{q_\phi(\mathbf{z} \mid \mathbf{x})}[\log p(\mathbf{z})] \tag{7}
$$

Now, we've arrived at the Variational Autoencoder (VAE) objective function. As mentioned earlier, we can estimate the ELBO using Monte Carlo sampling:

[4] $D_{KL}(p \parallel q) = \mathbb{E}_{p(x)} \log \frac{p(x)}{q(x)}$

[5] The amortized variational posterior is a distribution that depends on the data point $\mathbf{x}$. It is parameterized by a neural network with parameters ϕ.

$$\text{ELBO} \approx \frac{1}{L}\sum_{l=1}^{L}\left[\log p_\theta\left(\mathbf{x} \mid \mathbf{z}^{(l)}\right) - \log q_\phi\left(\mathbf{z}^{(l)} \mid \mathbf{x}\right) + \log p\left(\mathbf{z}^{(l)}\right)\right] \tag{8}$$

where we sample $\mathbf{z}^{(l)} \sim q_\phi(\mathbf{z} \mid \mathbf{x})$ for $l = 1, ..., L$.

Usually, both $p_\theta(\mathbf{x} \mid \mathbf{z})$ and $q_\phi(\mathbf{z} \mid \mathbf{x})$ are parameterized by neural networks. The parameters θ and ϕ are learned by maximizing the ELBO using stochastic gradient descent.

2.2 Another Perspective on the ELBO

The ELBO can also be derived seen from another perspective.

$$\text{ELBO} = \underbrace{\mathbb{E}_{q_\phi(\mathbf{z} \mid \mathbf{x})}[\log p_\theta(\mathbf{x} \mid \mathbf{z})]}_{\text{(negative) Reconstruction Error}} - \underbrace{D_{KL}(q_\phi(\mathbf{z}|\mathbf{x})| \mid p(\mathbf{z}))}_{\text{Regularization}} \tag{9}$$

The first term encourages the model to reconstruct the data well. The second term encourages the variational posterior, $q_\phi(\mathbf{z} \mid \mathbf{x})$, to be close to the prior, $p(\mathbf{z})$ (He (2024)).

2.3 Relation to Autoencoders

We can think of the VAE as two parts:

- An encoder, $q(\mathbf{z} \mid \mathbf{x})$, that maps the data to the latent space.
- A decoder, $p(\mathbf{x} \mid \mathbf{z})$, that maps the latent variables back to the data space.

which makes it similar to an autoencoder.

There is an important distinction here to be made. In a VAE, the encoder is stochastic. The encoder outputs the parameters of a distribution over the latent variables from which $\mathbf{z}$ is sampled. In a regular autoencoder, the encoder is deterministic and directly outputs the latent variables.

The same argument applies to the decoder. In a VAE, the decoder is stochastic and outputs the parameters of a distribution over the data from which $\mathbf{x}$ is sampled. In a regular autoencoder, the decoder is deterministic and directly outputs the reconstructed data.

What difference does that make?

Well, there are two main differences. First, the VAE is a generative model, while a regular autoencoder is a discriminative model. In the VAE, we can sample latent vectors from the prior and pass them through the decoder to generate new data points. In a regular autoencoder, we can't sample from the latent space because as far as we are concerned it is some random vector space that doesn't have any structure. The decoder learns mapping certain areas of the region of which we have no control over.

Second, the latent space of a VAE is more structured than that of a regular autoencoder. The KL divergence term in the ELBO encourages the variational posterior to be close to the prior, which we have knowledge of. This means that the latent space of a VAE is more organized and has a more meaningful structure than that of a regular autoencoder.

3 What We Are Building

What I cannot create, I do not understand - Richard Feynman

and I couldn't agree more. We will use PyTorch to implement a VAE that understands human faces. We will train it on masked human faces from the CelebAMask-HQ Dataset (Lee et al., 2020).

After training the model, we will sample new faces from the model. Moreover, we will create a morphing animation between two faces by linearly interpolating between their latent representations. This will demonstrate the smoothness of the latent space. All the code for this project is available in this GitHub repository: https://github.com/ibrahimhabibeg/vae-faces.

A word of caution before we proceed

Since I am writing this article in 2026, you may be expecting stellar results and ultra-realistic faces. This is not my goal here. First of all, the VAE is not the best model for generating high-quality images. As a matter of fact, the VAE is notorious for generating blurry images. If you want to generate high-quality images, you should look into models like GANs and Diffusion Models.

Second, my goal here is to demonstrate the inner workings of the VAE and how to implement it from scratch. I am not attempting to achieve SOTA results and there is a lot of room for improvement in the model we will build.

Now that we are on the same page, let's get started.

4 Distributions Choice

To build the VAE, we need to decide on 3 distributions:

1. The prior distribution, $p(\mathbf{z})$.
2. The likelihood, $p_\theta(\mathbf{x} \mid \mathbf{z})$.
3. The variational posterior, $q_\phi(\mathbf{z} \mid \mathbf{x})$.

For $p(\mathbf{z})$, I will be using a standard multivariate Gaussian distribution with zero mean and identity covariance matrix. This is a common choice for the prior in VAEs because of its simplicity and due to another reason that will become clear later.

For $q_\phi(\mathbf{z} \mid \mathbf{x})$, I will use a multivariate Gaussian distribution with a diagonal covariance matrix. Here I use a diagonal covariance matrix to simplify the model and reduce the number of parameters from $O(d^2)$ to $O(d)$ where d is the dimensionality of the latent space. The mean and the variance values will be produced by the encoder neural network.

$p_\theta(\mathbf{x} \mid \mathbf{z})$ is a bit more tricky. Personally, I would have preferred to use a Categorical distribution since the data is discrete (pixel values). This however will hugely increase the number of parameters of the distribution. To put some numbers on this, if we have a 64 x 64 RGB image, then the number of distribution parameters will be $64 \times 64 \times 3 \times 256 = 3,145,728$. This is such a huge number, and that's only the values outputted by the decoder network. The parameters of the decoder network itself will be larger.

To avoid this, I will be using a Bernoulli distribution for $p_\theta(\mathbf{x} \mid \mathbf{z})$. This means that the decoder will output values between 0 and 1 which will be interpreted as probabilities of the pixel being 255 (full color; red, green, or blue depending on the channel).

When it comes to sampling from $p_\theta(\mathbf{x} \mid \mathbf{z})$, we will *cheat* a little bit. After all, real world images are not binary. Instead of sampling from the Bernoulli distribution, we will use the output of the decoder directly as pixel values. For example, if the decoder outputs 0.5 for the red channel of a pixel, we will set the red channel value to $0.5 \times 255 = 127.5 \approx 128$.

Note that all of these are just design choices and not rules to follow for building a VAE. For example, for $p_\theta(\mathbf{x} \mid \mathbf{z})$, it is quite common to use a Gaussian distribution with a fixed variance (which will correspond to just changing the loss function in the code).

To sum it all up, we have:

Table 1: The choice of distributions for the VAE

Distribution	Choice	Neural Network Name
$p(\mathbf{z})$	$\mathcal{N}(\mathbf{0}, \mathbf{I})$	No neural network, fixed distribution
$q_\phi(\mathbf{z} \mid \mathbf{x})$	$\mathcal{N}\left(\boldsymbol{\mu}_\phi(\mathbf{x}), \operatorname{diag}\left(\boldsymbol{\sigma}_\phi^2(\mathbf{x})\right)\right)$	The Encoder
$p_\theta(\mathbf{x} \mid \mathbf{z})$	Bernoulli $(\theta(\mathbf{z}))$	The Decoder

4.1 Analytical Solution to the KL Divergence

We saw in Equation 8 that we can estimate the ELBO using Monte Carlo sampling. However, due to our distribution choices, we can calculate an analytical solution to the sum of the last two terms in Equation 7, which is equivalent to $-D_{KL}(q_\phi(\mathbf{z}|\mathbf{x})| \mid p(\mathbf{z}))$. The KL divergence is:

$$D_{KL}(q_\phi(\mathbf{z}|\mathbf{x})| \mid p(\mathbf{z})) = -\frac{1}{2}\sum_{j=1}^{d}\left(1 + \log\sigma_j^2 - \mu_j^2 - \sigma_j^2\right) \qquad (10)$$

where d is the dimensionality of the latent space, and μ_j and σ_j^2 are the mean and variance of the j-th dimension of the variational posterior.

Proof (Equation 10). Because the covariance matrix of both distributions is diagonal, we can break down the KL divergence into a sum of KL divergences between univariate Gaussians:

$$D_{KL}(q_\phi(\mathbf{z}|\mathbf{x})| \mid p(\mathbf{z})) = \sum_{j=1}^{d} D_{KL}\left(\mathcal{N}\left(\mu_j, \sigma_j^2\right) \parallel \mathcal{N}(0,1)\right)$$

By definition of the KL divergence, we have

$$D_{KL}\left(\mathcal{N}\left(\mu_j, \sigma_j^2\right) \parallel \mathcal{N}(0,1)\right) = \mathbb{E}_{x\sim\mathcal{N}\left(\mu_j,\sigma_j^2\right)}\left[\log\mathcal{N}\left(x \mid \mu_j, \sigma_j^2\right)\right] - \mathbb{E}_{x\sim\mathcal{N}\left(\mu_j,\sigma_j^2\right)}\left[\log\mathcal{N}(x \mid 0,1)\right]$$

First, I am going to simplify the first term:

$$\begin{aligned}
\log \mathcal{N}(x \mid \mu_j, \sigma_j^2) &= \log\left[\frac{1}{\sqrt{2\pi\sigma_j^2}} \exp\left(-\frac{1}{2}\left(\frac{x-\mu_j}{\sigma_j}\right)^2\right)\right] \\
&= -\frac{1}{2}\left(\frac{x-\mu_j}{\sigma_j}\right)^2 - \frac{1}{2}\log(2\pi) - \frac{1}{2}\log(\sigma_j^2) \\
\implies \mathbb{E}_{x\sim\mathcal{N}(\mu_j,\sigma_j^2)}\left[\log \mathcal{N}(x \mid \mu_j, \sigma_j^2)\right] &= -\frac{1}{2}\log(2\pi) - \frac{1}{2}\log(\sigma_j^2) - \frac{1}{2\sigma_j^2}\mathbb{E}_{x\sim\mathcal{N}(\mu_j,\sigma_j^2)}\left[(x-\mu_j)^2\right] \\
&= -\frac{1}{2}\log(2\pi) - \frac{1}{2}\log(\sigma_j^2) - \frac{1}{2}
\end{aligned}$$

where the last step is derived from the definition of the variance of a random variable, $\mathrm{Var}\ (X) = \mathbb{E}[(X - \mathbb{E}[X])^2]$. Now, I am going to simplify the second term:

$$\begin{aligned}
\log \mathcal{N}(x \mid 0, 1) &= \log\left[\frac{1}{\sqrt{2\pi}} \exp\left(-\frac{1}{2}x^2\right)\right] \\
&= -\frac{1}{2}x^2 - \frac{1}{2}\log(2\pi) \\
\implies \mathbb{E}_{x\sim\mathcal{N}(\mu_j,\sigma_j^2)}[\log \mathcal{N}(x \mid 0, 1)] &= -\frac{1}{2}\log(2\pi) - \frac{1}{2}\mathbb{E}_{x\sim\mathcal{N}(\mu_j,\sigma_j^2)}[x^2]
\end{aligned}$$

Recall that $\mathrm{Var}\ (X) = \mathbb{E}[X^2] - (\mathbb{E}[X])^2$. Hence, we can rewrite the last step as:

$$\mathbb{E}_{x\sim\mathcal{N}(\mu_j,\sigma_j^2)}[\log \mathcal{N}(x \mid 0, 1)] = -\frac{1}{2}\log(2\pi) - \frac{1}{2}\sigma_j^2 - \frac{1}{2}\mu_j^2$$

Substituting the simplified terms back into the KL divergence, we get:

$$\begin{aligned}
D_{KL}(\mathcal{N}(\mu_j, \sigma_j^2) \parallel \mathcal{N}(0,1)) &= -\frac{1}{2}\log(2\pi) - \frac{1}{2}\log(\sigma_j^2) - \frac{1}{2} + \frac{1}{2}\log(2\pi) + \frac{1}{2}\sigma_j^2 + \frac{1}{2}\mu_j^2 \\
&= -\frac{1}{2}(1 + \log\sigma_j^2 - \mu_j^2 - \sigma_j^2)
\end{aligned}$$

Substituting this back into the sum, we get the final expression for the KL divergence:

$$D_{KL}(q_\phi(\mathbf{z}|\mathbf{x})| \mid p(\mathbf{z})) = -\frac{1}{2}\sum_{j=1}^{d}(1 + \log\sigma_j^2 - \mu_j^2 - \sigma_j^2)$$

5 Implementation

Now that we have decided on the distributions we will be using, we can start implementing the model. The full code is available in the GitHub repository https://github.com/ibrahimhabibeg/vae-faces. Here I will be showing the most important parts of the code and explaining them.

i Running the Code

There are two folders in the repository: `scripts` and `src`. The `scripts` folder contains scripts for downloading the datasets, preprocessing the data, and training the model. The `src` folder contains the implementation of the VAE model and the training loop.

By default, all data will be stored in the `data` folder and all trained models will be stored in the `checkpoints` folder. You can can change these paths when running any script through the command line arguments.

The project uses `uv` to manage the dependencies. You can install the dependencies by running `uv sync` in the root directory of the project.

To view the configurations of any script, you can run `uv run <path-to-script> -h`.

5.1 Data Preparation

The codebase is built to support two datasets: the CelebAMask-HQ Dataset (Lee et al., 2020) and the CelebA Dataset (Liu et al., 2015).

In the beginning, I used the CelebA dataset. After running some iterations, I found that the model was able to learn facial features well, but struggled with the background. Moreover, the facial features were fighting the background for the capacity of the latent space. To keep focused on the facial features, I switched to the CelebAMask-HQ dataset which provides masks for multiple facial features and garments. We will focus on the CelebAMask-HQ Dataset, but keep in mind that the codebase supports both datasets and you can easily switch between them.

There are two scripts to download the datasets: `scripts/celeba_download.py` and `scripts/celebamask_hq_download.py`. There is no preprocessing script for the CelebA dataset since the images are already cropped and resized. For the CelebAMask-HQ dataset, there is a preprocessing script `scripts/celebamask_hq_prep.py`.

There are two dataset classes in the codebase: `CelebA` and `CelebAMaskHQ`. They are implemented in `src/face_vae.py`, and you can access their code here: https://github.com/ibrahimhabibeg/vae-faces/blob/main/src/face_vae/data.py.

The data coming to the model will be 64 x 64 RGB images. For the CelebAMask-HQ dataset, the preprocessing script crops the images and set the background to black (the color is configurable). For the CelebA dataset, its dataset class crops the images on the fly and resizes them to 64 x 64.

5.2 The Encoder

The encoder is implemented in the `Encoder` class in `src/face_vae/model.py`.

```python
class Encoder(nn.Module):
    """
    Encoder module for VAE that takes in an image and outputs a latent vector.
    Defines q(z|x) = N(z; mu(x), diag(var(x))) where mu and var are predicted
```

```
by the network.
    """

    def __init__(
        self,
        encoder_net: nn.Module,
        latent_dim: int,
        encoder_output_dim: int = 256 * 4 * 4,
    ):
        """
        Encoder module for VAE.

        Args:
            encoder_net (nn.Module): The convolutional encoder network.
            latent_dim (int): Dimension of the latent space.
            encoder_output_dim (int): Dimension of the encoder output. Default
is 256 * 4 * 4.
        """
        super().__init__()
        self.encoder_net = encoder_net
        self.head_mu = nn.Linear(encoder_output_dim, latent_dim)
        # Predict log var instead of var because var must be positive while
log var can be any real number.
        self.head_logvar = nn.Linear(encoder_output_dim, latent_dim)

    def reparametrize(self, mu, logvar):
        """
        Reparameterization trick to sample from N(mu, var) using N(0, 1).
        Necessary for backpropagation through the sampling step.
        """
        std = torch.exp(0.5 * logvar)
        eps = torch.randn_like(std).to(std.device)  # eps ~ N(0, 1)
        return mu + eps * std  # Return sampled latent vector (z ~ N(mu, var))

    def forward(self, x):
        assert x.shape[1] == 3, "Expected input with 3 channels (RGB)"
        assert x.dim() == 4, "Expected input with shape (batch_size, 3, H, W)"
        enc_out = self.encoder_net(x)
        mu = self.head_mu(enc_out)
        logvar = self.head_logvar(enc_out)
        z = self.reparametrize(mu, logvar)
        return (
            z,
            mu,
            logvar,
        )  # Return the sampled latent vector (z) for reconstruction and mean
and log variance for loss calculation
```

```
    def sample(self, x=None, mu=None, logvar=None):
        """
        Sample a latent vector from the encoder. Can be used for both training
and inference.
        If x is provided, it will encode x to get mu and logvar. Otherwise, it
will use the provided mu and logvar.
        """
        if x is not None:
            z, mu, logvar = self.forward(x)
            return z
        assert mu is not None and logvar is not None, (
            "Must provide either x or both mu and logvar"
        )
        z = self.reparametrize(mu, logvar)
        return z
```

I want the encoder to be flexible. On initialization, it takes in an `encoder_net` which can be any network as long as it accepts a batch of images. The goal of the `encoder_net` is to extract features from the images and output a vector of size `encoder_output_dim`.

Since we are using a Gaussian distribution for the variational posterior, we need to output both the mean and the variance. To do this, we have two linear layers: `head_mu` and `head_logvar`. They take the output of the `encoder_net` and output the mean and log variance of the latent distribution respectively.

Notice here that we are predicting the log variance instead of the variance. The variance must be positive while the output of a linear layer can be any real number. The log variance can be any real number, however, so we can predict it instead and then exponentiate to get the variance when needed.

5.2.a The Reparameterization Trick

There is a problem we didn't consider yet. There is no analytical solution for $p_\theta(\mathbf{x} \mid \mathbf{z})$ term in the ELBO; thus, we need to estimate it using Monte Carlo sampling. This means that we need to sample $\mathbf{z}$ from $q_\phi(\mathbf{z} \mid \mathbf{x})$ during training.

However, sampling isn't a differentiable operation (you can't differentiate randomness). If we directly sample the latent variables from the output of the encoder, we won't be able to backpropagate the gradients through the sampling step; hence, we won't be able to train the encoder.

A great solution to this is called the **reparameterization trick**. Instead of sampling $\mathbf{z}$ directly from $q_\phi(\mathbf{z} \mid \mathbf{x})$, we can sample $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ and then compute $\mathbf{z}$ as follows:

$$\mathbf{z} = \mu_\phi(\mathbf{x}) + \sigma_\phi(\mathbf{x}) \odot \boldsymbol{\epsilon}$$

The expectation and variance of $\mathbf{z}$ will still be $\mu_\phi(\mathbf{x})$ and $\sigma_\phi^2(\mathbf{x})$ respectively. Moreover, since $\boldsymbol{\epsilon}$ is independent of the parameters of the encoder, we can backpropagate through this operation and train the encoder.

We basically sampled from $\mathcal{N}\left(\mu_\phi(\mathbf{x}), \text{diag}\left(\sigma^2_\phi(\mathbf{x})\right)\right)$ indirectly by sampling from $\mathcal{N}(\mathbf{0}, \mathbf{I})$, which we can sample from easily due to independence from the encoder parameters.

The method `reparametrize` in the code implements the reparameterization trick.

5.2.b Sampling from the Encoder

The method `sample` in the code allows us to sample from the encoder.

This method shines the light on the differnce between a VAE's encoder and a regular autoencoder's encoder. In a regular autoencoder, the encoder is deterministic and directly outputs the latent variables. In a VAE, the encoder is stochastic. While the output of a VAE's encoder network is diterministic, the network doesn't predict the latent variables. Instead, it predicts the parameters of a distribution over the latent variables from which we can sample a stochastic latent vector.

5.3 The Decoder

The decoder is implemented in the `Decoder` class in `src/face_vae/model.py`.

```python
class Decoder(nn.Module):
    """
    Decoder module for VAE that takes in a latent vector and outputs a
reconstructed image.
    p(x|z) definition depends on the loss function used.
    For binary cross-entropy loss, p(x|z) = Bernoulli(x; decoder(z)) where
decoder(z) outputs the probability of each pixel being 1.
    For mean squared error loss, p(x|z) = Gaussian(x; decoder(z), I) where
decoder(z) outputs the mean and the mean is bounded to [0, 1] using sigmoid
activation.
    Categorical distribution isn't used due to computational constraints
    """

    def __init__(self, decoder_net: nn.Module):
        super().__init__()
        self.decoder_net = decoder_net

    def forward(self, z):
        assert z.dim() == 2, "Expected input with shape (batch_size,
latent_dim)"
        recon_x = self.decoder_net(z)
        recon_x = torch.sigmoid(recon_x)  # Apply sigmoid to get pixel values
in [0, 1]
        return recon_x  # Return recon_x with shape (batch_size, 3, 64, 64)
and values in [0, 1]
```

Once again, its built in a flexible manner. On initialization, it takes in a `decoder_net` which can be any network as long as it accepts a batch of latent vectors and outputs a tensor of shape `(batch_size, C, H, W)` where $C = 3, H = 64, W = 64$ in our case.

The output of the `decoder_net` is passed through a sigmoid activation to transform them to probabilities.

You will find here that I decided not to create a sampling method. This is once again because I am *cheating*. I am supposed to sample the pixel values from the Bernoulli distribution parameterized by the output of the decoder. However, this will result in very bad looking images (a pixel is either on or off; no middle ground). As I mentioned earlier, instead of sampling from the Bernoulli distribution, we will use the output of the decoder directly as pixel values.

Note that this is just a design choice and not a rule to follow when building a VAE.

5.4 The Prior

I create a seperate class for the prior.

```python
class Prior(nn.Module):
    """
    Prior module for VAE that defines the prior distribution p(z).
    Uses a standard normal distribution N(0, I) as the prior.
    """

    def __init__(self):
        super().__init__()

    def forward(self, batch_size, latent_dim, device):
        return self.sample(batch_size, latent_dim, device)

    def sample(self, batch_size, latent_dim, device):
        return torch.randn(batch_size, latent_dim, device=device)
```

It is a very simple class that just samples from a standard normal distribution. I've added an explicit class for it to demonstrate its significane and to hide the prior's distribution choice when sampling a latent vector.

Some implementations, like in Tomczak (2024), add a function in the enocder, decoder, and prior classes to calculate the log probability of a given input under the corresponding distribution, which are the three components of the Monte Carlo estimate of the ELBO in Equation 8. I really like this design choice because it helps hide knowledge about the distribution choice from the error calculation in the training loop.

However, I didn't follow it because for the regualization term in the ELBO (the KL divergence) I won't be using the Monte Carlo estimator and will instead use the analytical solution we derived in Equation 10. Moreover, for the decoder, I wanted to keep it flexible and not tie it to a specific distribution choice. As a matter of fact, by just changing the loss function in the training loop, we can change the distribution choice for the decoder without changing its code at all (more on that later).

5.5 The VAE Class

Finally, we have the `VAE` class that puts everything together.

```
class VAE(nn.Module):
    """
    Variational Autoencoder (VAE) class that combines the Encoder, Decoder,
and Prior modules.
    """

    def __init__(self, encoder: Encoder, decoder: Decoder, prior: Prior):
        super().__init__()
        self.encoder = encoder
        self.decoder = decoder
        self.prior = prior

    def forward(self, x):
        z, mu, logvar = self.encoder(x)
        recon_x = self.decoder(z)
        return recon_x, mu, logvar

    def sample(self, batch_size, latent_dim, device):
        """
        Sample from the prior and decode to get a new image.
        """
        z = self.prior.sample(batch_size, latent_dim, device)
        recon_x = self.decoder(z)
        return recon_x

    def encode(self, x):
        """
        Encode an image to get its latent representation (mu and logvar).
        """
        if x.dim() == 3:
            x = x.unsqueeze(0)  # Add batch dimension if missing
        assert x.shape[1] == 3, "Expected input with 3 channels (RGB)"
        _, mu, logvar = self.encoder(x)
        return mu.squeeze(0), logvar.squeeze(0)  # Remove batch dimension for
single image input

    def decode(self, z):
        """
        Decode a latent vector to get the reconstructed image.
        """
        if z.dim() == 1:
            z = z.unsqueeze(0)  # Add batch dimension if missing
        assert z.dim() == 2, "Expected input with shape (batch_size,
latent_dim)"
        recon_x = self.decoder(z)
        return recon_x.squeeze(0)  # Remove batch dimension for single image
output
```

The forward `method` here returns the reconstructed image, the mean, and the log variance which are all used for the loss calculation in the training loop.

The `sample` method allows us to sample new images from the model by sampling from the prior and passing the sampled latent vector through the decoder.

The images returned by the `forward` method are attempts of the model to reconstruct the input images. That's why they will look similar to the input images. The images returned by the `sample` method are generated from the model by sampling from the prior. These images are totally new, and not based on any specific data point. This is the generative aspect of the VAE.

5.6 The Loss Function

`src/face_vae/loss.py` contains the implementation of the loss function.

```
def bce_reconstruction_loss(recon_x, x):
    """
    Binary Cross Entropy reconstruction loss for VAE.
    Args:
        recon_x: Reconstructed image (output of decoder).
        x: Original image (input to encoder).
    Returns:
        BCE loss value.
    """
    assert recon_x.shape == x.shape, (
        "Reconstructed and original images must have the same shape"
    )
    bce_loss = F.binary_cross_entropy(recon_x, x, reduction="sum")
    return bce_loss


def mse_reconstruction_loss(recon_x, x):
    """
    Mean Squared Error reconstruction loss for VAE.
    Args:
        recon_x: Reconstructed image (output of decoder).
        x: Original image (input to encoder).
    Returns:
        MSE loss value.
    """
    assert recon_x.shape == x.shape, (
        "Reconstructed and original images must have the same shape"
    )
    mse_loss = F.mse_loss(recon_x, x, reduction="sum")
    return mse_loss
```

```
def kl_divergence_loss(mu, logvar):
    """
    KL Divergence loss for VAE.
    Assumes the prior p(z) is a standard normal distribution N(0, I) and the
approximate posterior q(z|x) is a diagonal Gaussian with mean mu and log
variance logvar.
    This is based on the closed-form solution for KL divergence between two
Gaussians: KL(q(z|x) || p(z)) = -0.5 * sum(1 + logvar - mu^2 - variance)
    Args:
        mu: Mean of the latent distribution (output of encoder).
        logvar: Log variance of the latent distribution (output of encoder).
    Returns:
        KL divergence loss value.
    """
    var = torch.exp(logvar)
    kl_loss = -0.5 * torch.sum(1 + logvar - mu.pow(2) - var)
    return kl_loss

def vae_loss(recon_x, x, mu, logvar, recon_loss_fn=bce_reconstruction_loss,
beta=1.0):
    """
    Total VAE loss combining reconstruction loss and regularization (KL
divergence) loss.
    The KL divergence is weighted by a factor.
    The used regularization assumes p(z) is a standard normal distribution
N(0, I) and q(z|x) is a diagonal Gaussian with mean mu and log variance
logvar.
    Increase the beta factor will make images generated from random latent
vectors sampled from the prior look more like real images, but may reduce the
quality of reconstructions.
    Decrease it will improve reconstructions but may make generated images
look worse as th latent space might have gaps and q(z|x) might not match p(z)
well.
        recon_x: Reconstructed image (output of decoder).
        x: Original image (input to encoder).
        mu: Mean of the latent distribution (output of encoder).
        logvar: Log variance of the latent distribution (output of encoder).
        recon_loss_fn: Reconstruction loss function to use (default is binary
cross-entropy).
        beta: Weighting factor for the KL divergence loss (default is 1.0).
    Returns:
        Total VAE loss value.
    """
    recon_loss = recon_loss_fn(recon_x, x)
    kl_loss = kl_divergence_loss(mu, logvar)
    total_loss = recon_loss + beta * kl_loss
    return total_loss
```

This is the main function that calculates the total VAE loss. Note that PyTorch minimizes the loss, so it returns the negative of the ELBO.

As we have seen in Equation 9, the ELBOcan be seen as a combination of a reconstruction error and a regularization term. The `vae_loss` function makes no assumption about the distribution choice for the decoder; thus, it takes in a `recon_loss_fn` argument which is the reconstruction loss function to use. This function should return $-\log p_\theta(\mathbf{x} \mid \mathbf{z})$ given the reconstructed image and the original image.

By setting `recon_loss_fn` to `bce_reconstruction_loss`, we are assuming that $p_\theta(\mathbf{x} \mid \mathbf{z})$ is a Bernoulli distribution, and thus we are using the binary cross-entropy loss as the reconstruction loss. I've left another function `mse_reconstruction_loss` in the codebase which can be used for the reconstruction loss. If we use the MSE insttead of the BCE, we are bsically transforming $p_\theta(\mathbf{x} \mid \mathbf{z})$ from a Bernoulli distribution to a Gaussian distribution with the identity covariance matrix and the mean between 0 and 1 (due to the sigmoid activation in the decoder) that is outputted by the decoder.

The `vae_loss` assumes that the prior is a standard normal distribution and the variational posterior is a diagonal Gaussian distribution, which is why it uses the `kl_divergence_loss` function we derived in Equation 10.

6 Sample Results

We can finally train the model and see some results. The training script is `scripts/train.py`. You can run it using the command `uv run train.py` to train on the CelebAMask-HQ dataset.

6.1 Image Reconstruction

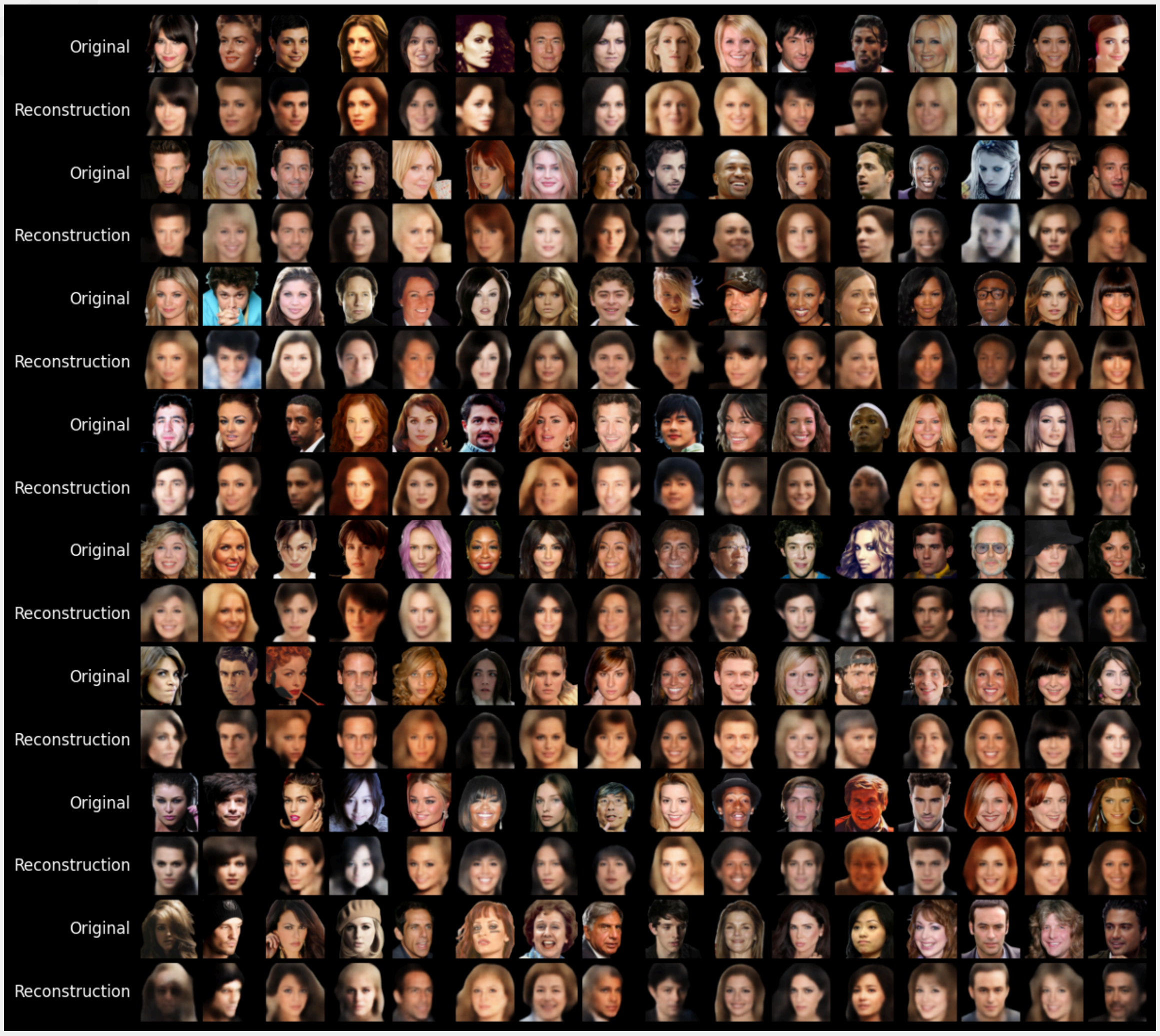

Figure 1: Reconstructed Images Grid. Every image is the reconstruction of the image directly above it

Figure 1 shows a grid of reconstructed images. The rows alternate between original images and their reconstructions.

We say that the model is able to capture high level features of the faces. The model is able to capture the hair color, skin color, geneder, and pose. We see that the reconstructions are blurry, which is a common problem with VAEs. However, the reconstructions are still recognizable and share the same high level features as the original images.

i Note on Viewing the Images

This website uses lightbox to display. You can click on the images to zoom in and see the details better.

6.2 Image Generation

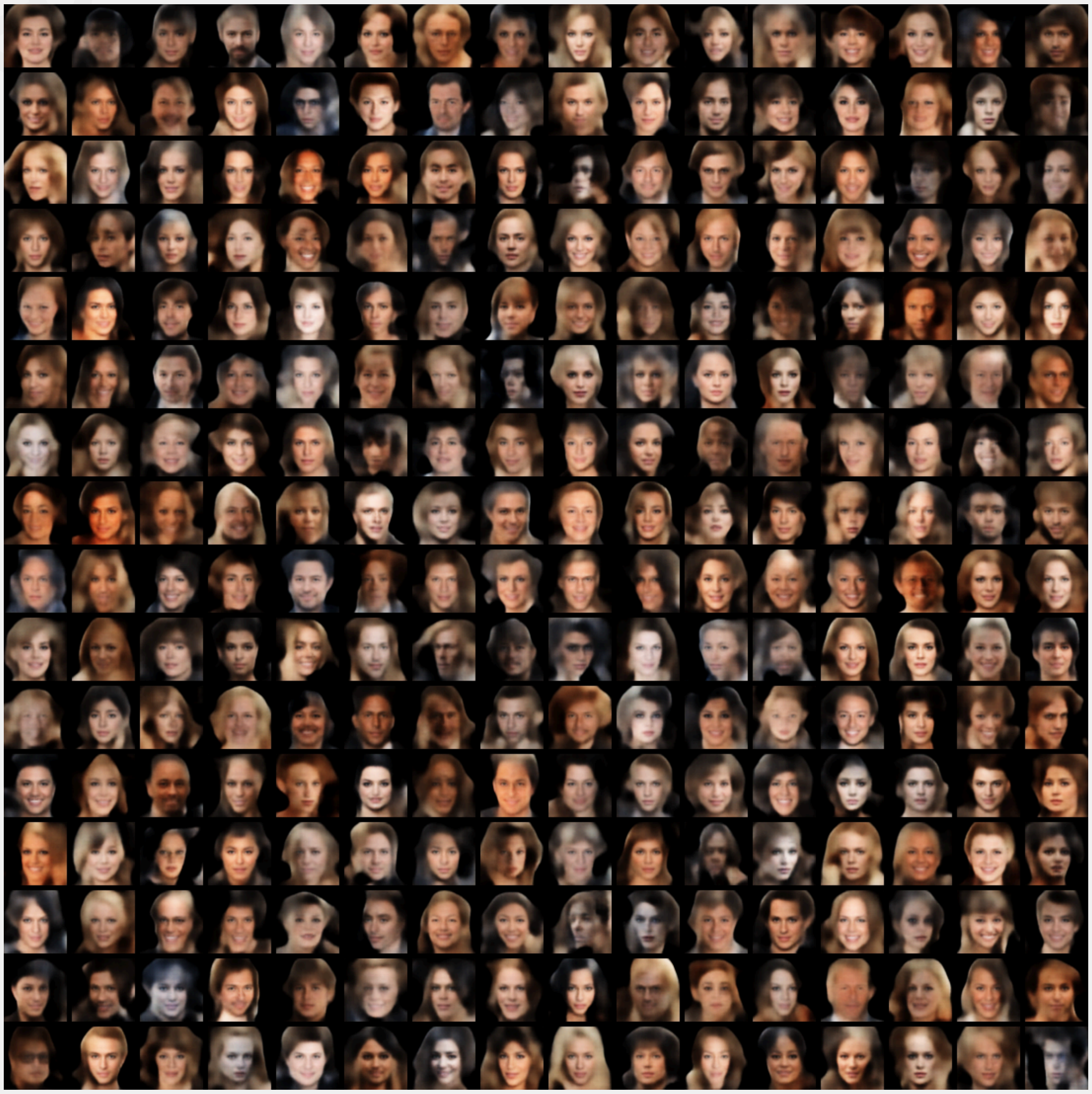

Figure 2: Generated Images Grid

Figure 2 shows a grid of images generated by the model by sampling a latent vector from the prior and passing it through the decoder.

The images are a bit blurry, but they do look like human faces. The model is able to generate faces with different hair style, hair colors, skin colors, and poses. This shows that the model is able to learn a good representation of the data and generate new samples from it.

You may notice that the black backgound isn't well generates and sometimes obscurs the faces. The reason is that in the current setup we are teaching the model to reconstruct both the images and the masks. A better approach would be to have two decoders: one for the images and one for the masks (a Bernoulli distribution). This way, the model can learn to generate better masks from a given latent vector. In such setup, the true masks should be applied on the generated images during training so that they don't pollute the loss on the image.

Since we are focusing on the properties of the VAE here and not attempting to create any contest winning model, I decided to keep the current setup for simplicity. However, if you want to improve the quality of the generated images, you can try the two decoder approach I just mentioned.

6.3 Image Morphing

We know that the latent space of a VAE is continuous and smooth. To visualize this, we can take two images, encode them to get their latent representations, and then linearly interpolate between the two latent vectors and decoding the steps in between. That is we will be moving in the latent space between the two images and seeing how the generated images change as we move.

If the latent space is smooth, we should see a smooth transition between the two images. There should be no sudden jumps or changes. Image features such as face pose or hair color should change gradually as we move in the latent space.

You can find the function used for morphing here: https://github.com/ibrahimhabibeg/vae-faces/blob/main/src/face_vae/morph.py.

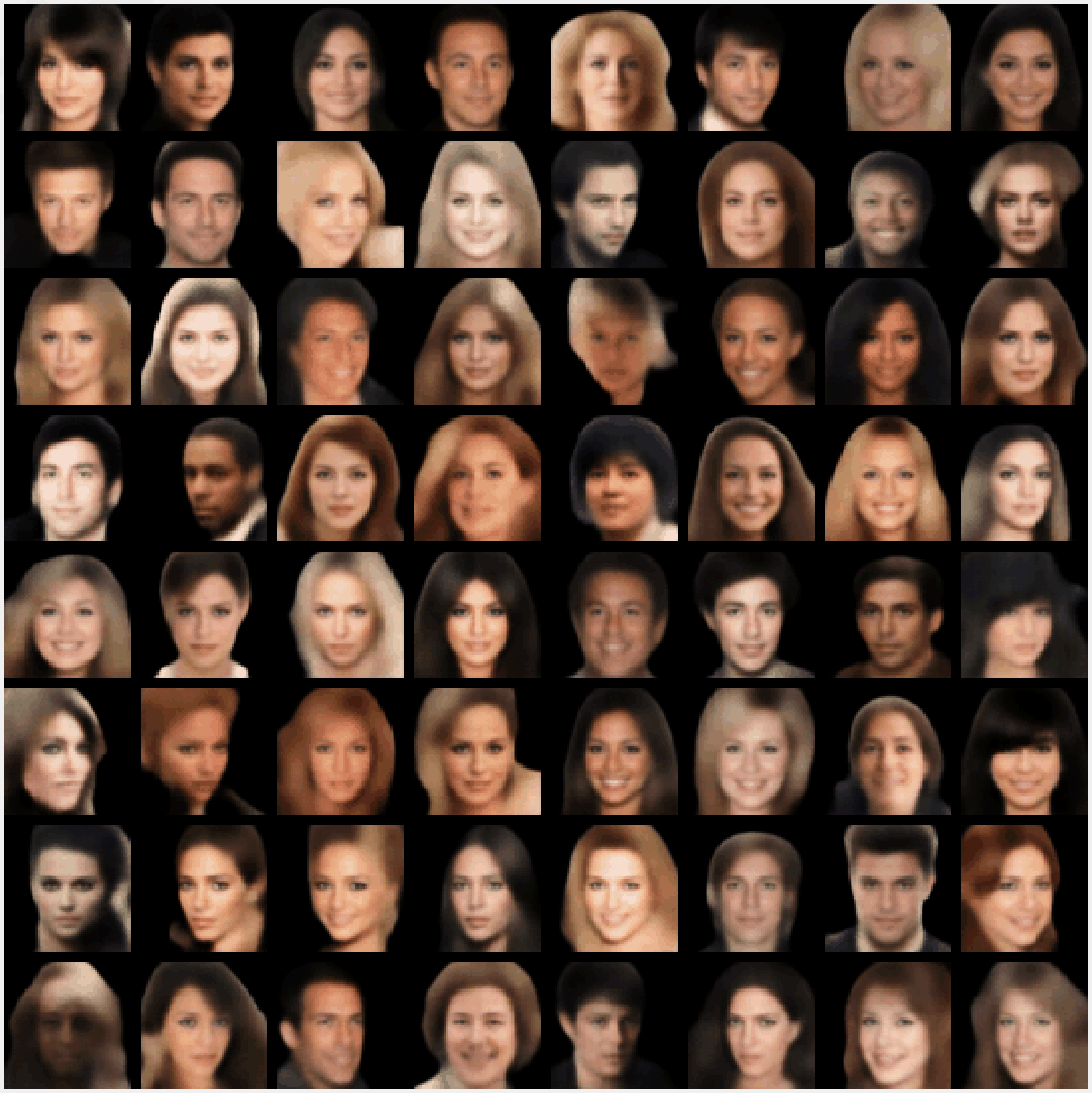

Figure 3: Image Morphing Grid

You can see the morphing results in Figure 3. You can set your focus on a single image in the grid and see how it changes as we move in the latent space.

You can view the GIFs by accessing the web version of this blog post at https://ibrahimhabib.me/blogs/variational-autoencoder/.

For example, focus with me on the second image in the first row. The starting image has short black hair and is looking to the right. The final image has long light brown hair and is looking to the left. We can see in the GIF that face smoothly rotates to the left while maintaining the same expression. As the image transition, the hair outlines slowly appears and smoothly the color transitions from black to light brown.

Note that here I am not merging the two decoded images together. I am only mergeing the two latent vectors together and then decoding the merged latent vector at each step. The image smoothness is a result of the smoothness of the latent space.

If a classic autoencoder was used instead of a VAE, the latent space wouldn't be smooth and we would see sudden jumps in the generated images as we move in the latent space. The classic autoencoder doesn't have the regularization term in the loss function; thus, similar images are indeed clustered together in the latent space, but we have zero knowledge of the structure of the latent space between these clusters. Usually, the latent space of a classic autoencoder is discontinuous and has gaps between the clusters of similar images.

This isn't necessarily a problem in the autoencoder. The autoencoder just isn't built for this task; it's not a generative model.

7 Conclusion

In this blog post, we explored the Variational Autoencoder (VAE) model. We started by understanding from a mathematical lens what a generative model is. We then moved on to latent variable models and VAE. After that we built a VAE from scratch and trained it on the CelebA-Mask-HQ dataset. To build the model, we started with deciding on the distributions to use for the encoder, decoder, and prior. We then derived an analytical solution for the KL divergence term in the ELBO based on the distribution choices. Finally, we implemented the model and trained it to see some results.

I hope you enjoyed this blog post and found it useful. If you have any questions or feedback, don't hesitate to reach out to me on Twitter @ibrahimhabibeg or LinkedIn https://www.linkedin.com/in/ibrahimhabibeg/.

Thanks for reading!

Bibliography

He, K. (2024, September). *Variational Autoencoder.* https://mit-6s978.github.io/assets/pdfs/lec2_vae.pdf

Lee, C.-H., Liu, Z., Wu, L., & Luo, P. (2020,). MaskGAN: Towards Diverse and Interactive Facial Image Manipulation. *IEEE Conference on Computer Vision and Pattern Recognition (CVPR).*

Liu, Z., Luo, P., Wang, X., & Tang, X. (2015, December). Deep Learning Face Attributes in the Wild. *Proceedings of International Conference on Computer Vision (ICCV).*

Tomczak, J. M. (2024). *Deep Generative Modeling* (2nd ed. 2024). Springer International Publishing. https://doi.org/10.1007/978-3-031-64087-2