Report for exercise 3 from group A

Tasks addressed: 4
Authors: Hirmay Sandesara (03807348)
Yikun Hao (03768321)
Yusuf Alptigin Gün (03796825)
Subodh Pokhrel (03796731)
Last compiled: 2025–06–10

The work on tasks was divided in the following way:

Hirmay Sandesara (03807348)	Task 1	25%
	Task 2	25%
	Task 3	25%
	Task 4	25%
Yikun Hao (03768321)	Task 1	25%
	Task 2	25%
	Task 3	25%
	Task 4	25%
Yusuf Alptigin Gün (03796825)	Task 1	25%
	Task 2	25%
	Task 3	25%
	Task 4	25%
Subodh Pokhrel (03796731)	Task 1	25%
	Task 2	25%
	Task 3	25%
	Task 4	25%

1 Task 1: Principal Component Analysis (PCA)

For each of the parts required in 1, the report is divided into two parts called **Code** and **Analysis**. Most of the code for the `utils` functions can be run using only a few lines with the help of `numpy` libraries, and the main parts of the code mostly contain using those `utils` functions in a main file. The general outline of this part of the report is thus to talk about each function and file very briefly, followed by the analysis of the results.

1.1 Utility Functions

Here, we discuss the utility functions to be later used in each main file for different parts of the tasks. An outline of these functions is given below:

- `center_data`: Taking advantage of numpy's broadcasting mechanism, as we'll do in most functions, subtracting the `np.mean` of data on each column (axis 0) from the data is sufficient to calculate the centered data.
- `compute_svd`: `np.linalg.The svd` function calculates the SVD of a matrix and returns the necessary output as we want. We just make sure to specify `full_matrices=False` for truncated outputs.
- `compute_energy`: Dividing the squared energy of the specific singular value and dividing by the total squared energy using `np.sum` calculates the energy for a singular component. Also, we multiply by 100 to get the percentage of energy the component has. The formula can be given as:

$$\left(\frac{S_{c-1}^2}{\sum_{i=1}^{n} S_i^2}\right) \times 100 \tag{1}$$

- `compute_cumulative_energy`: While the division part stays the same as in `compute_energy`, we know calculate the total squared energy up to the given singular value for the numerator, this can be done with the slicing operation `S[:c]` in numpy. The formula can be given as:

$$\left(\frac{\sum_{i=1}^{c} S_i^2}{\sum_{i=1}^{n} S_i^2}\right) \times 100 \tag{2}$$

- `load_resize_image`: The function `scipy.misc.face(gray=True)` provided to us enables us to create the image as necessary, all we do afterwards is use the `resize` function to scale the image to its required size of 185x249.
- `reconstruct_data_using_truncated_svd`: Firstly, we create a so-called truncated matrix with the same dimensions as the singular value matrix S using `np.zeros_like`. Then, we slice the matrix up to the nth component and assign the necessary singular values with `S[:n_components]`, turning this matrix into a diagonal one using `np.diag`. With the new singular value matrix created, we can reconstruct the data using the Python infix operator @ for `np.matmul`, U and V transpose matrices, with the formula `U @ S_diagonal @ V_t`.
- `compute_num_components_capturing_threshold_energy`: In this function, we make use of the `np.searchsorted numpy` operation. By calculating the squared total and cumulative energy, this function allows us to send the ratio of them with the energy threshold as inputs to return the index at which the threshold is surpassed. By adding 1 to the result, we get how many singular values are needed to capture the threshold energy.

1.2 Linear Subspace

In this part of the task, a basic PCA application is implemented on a linear subspace. Then, the one-dimensional approximation is computed with energy calculations and the required plots.

1.2.1 Code:

Firstly, we load the dataset from the `txt` file using `np.loadtxt` by providing the necessary path. Then, the mean of the data is calculated using `np.mean` to be later used in plotting. The data is centered, and the SVD is computed consecutively using `utils.center_data` and `utils.compute_svd`. To plot the necessary figures, `matplotlib` is used. The configurations for the plot are defined, and the data is placed onto the plot as scatter data. Then, the direction of the principal components is defined through the `V_t` matrix computed with the SVD function and some scaling factor. These components (vector field) are visualized using the `quiver` function in the plot with the mean and direction. Finally, the energy of the first and second principal components is found using the `utils.compute_energy` function. The plot is shown as a full graph, and the energy calculations are printed.

1.2.2 Analysis:

In Figure 1, the plot of the dataset with the directions for each principal component can be seen. Note that the length of the arrows doesn't correspond to how much variance is captured in each direction, but is shown like this for better visualization. We can see that in the direction of the first principal component, the data is much more spread out, while in the second component direction, there isn't much change. Thus, it can be said that most of the variance is captured in the first principal component. This is also backed up by the energy calculations as the `utils.compute_energy` gives us the following percentages for how much energy each principal component contains:

- **Principal Component 1:** 99.31%
- **Principal Component 2:** 0.69%

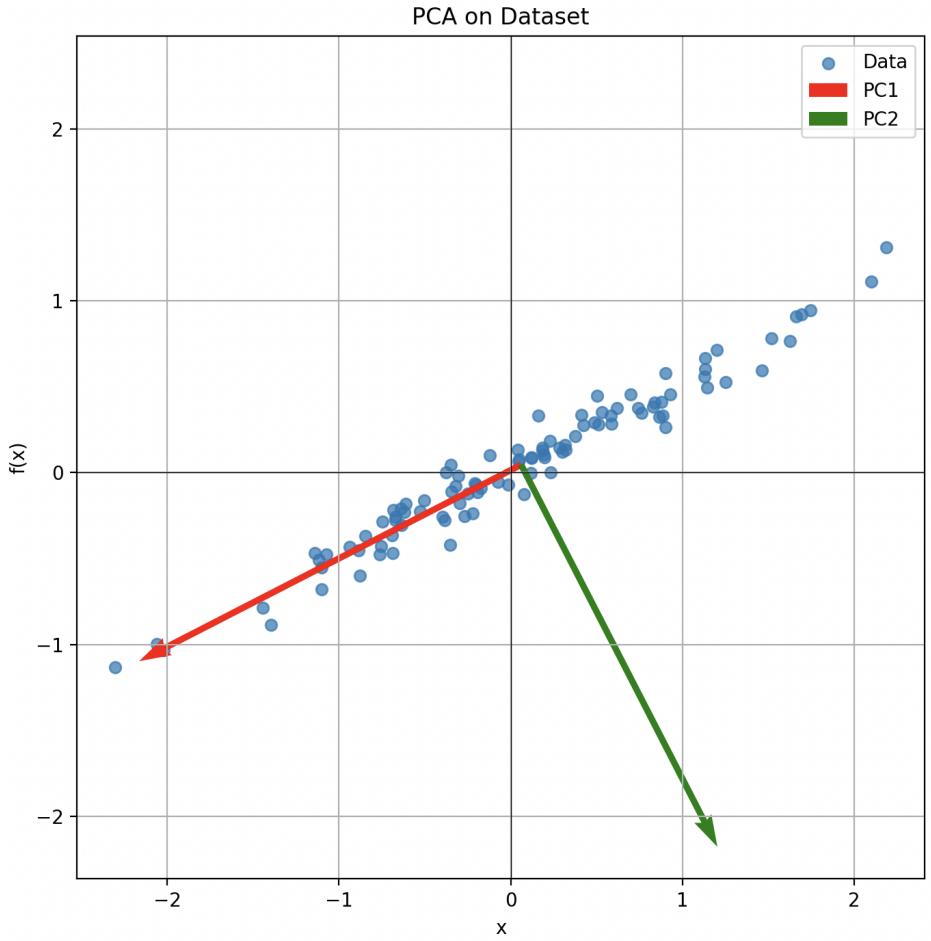


Figure 1: Plot of the data with the direction of the two principal components added

1.3 Image Reconstruction

In this part of the task, a raccoon image is generated to apply different visual reconstructions using PCA. The goal is to find how much information is lost through reconstruction with different amounts of principal components and compute how many are needed for a close to lossless construction.

1.3.1 Code:

Using the `utils` functions for this part of the task is sufficient. The raccoon image is loaded using `utils.load_resize_image` and the SVD of it is computed with `compute_svd`. Then, the image is reconstructed with different amounts of principal components, all, 120, 50, 10, using the `reconstruct_images`; this function plots the images as well. Finally, the number of principal components needed for a reconstruction of no more than a %1 loss is computed using `utils.compute_num_components_capturing_threshold_energy` and the results are printed.

1.3.2 Analysis:

In Figure 2, we can see the reconstructed raccoon image with different numbers of principal components. It's clear that with 10 components, the image has quite a bit of visible loss. But for other images, the reconstruction captures most of the original image. This is also supported by finding out how many components are needed to achieve less than %1 loss through truncation, which is found as **26**.

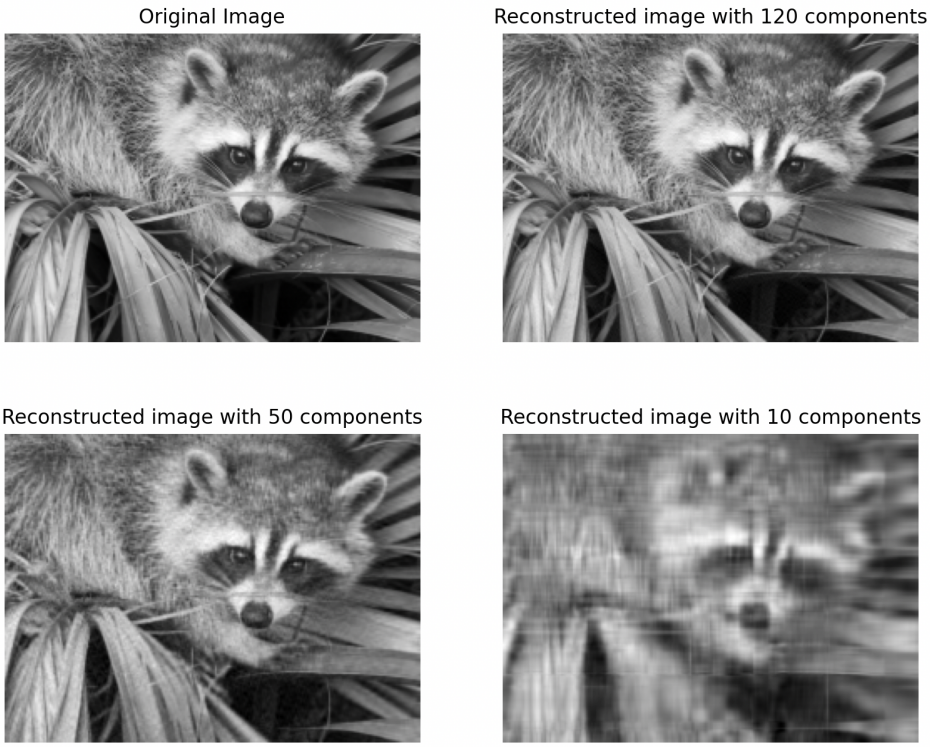


Figure 2: Plot of the data with the direction of the two principal components added

1.4 Pedestrian Paths

In this part of the task, the paths of two pedestrians are visualized with different numbers of principal components to compare and discuss how many components are needed to capture most of the energy.

1.4.1 Code:

Firstly, we load the dataset from the `txt` file using `np.loadtxt` by providing the necessary path. Then, the data is centered using `utils.center_data`. To get the data for pedestrians 1 and 2, it's sliced for the first and the second two columns with `data[:, 0:2]` and `data[:, 2:4]`. This data is then

visualized with `utils.visualize_traj_two_pedestrians`. To reconstruct the path data with the first 2 principal components, we compute the SVD of the data with `utils.compute_svd` and reconstruct it back with `utils.reconstruct_data_using_truncated_svd` by specifying 2 principal components in the function input. The projected data is sliced and visualized as discussed before. For the final analysis, the energy of the first two principal components, the cumulative sum of them, and the number of principal components needed to capture %90 of the energy are computed with `utils.compute_energy`, `utils.compute_cumulative_energy`, and `utils.compute_num_components_capturing_threshold_energy`. They're also printed for visualization.

1.4.2 Analysis:

In Figure 3, the paths of the first two pedestrians are visualized in 2D space. As a general direction, the pedestrians move in some sort of circular and "8" shaped directions; but for each small section of the movement, we can see that there are random deviations in each direction that show more intricate movement in smaller areas. In Figure 4, the same walks for the pedestrians are visualized by projecting the data into 2-dimensions (the first two principal components). In these paths, even though the general direction of the pedestrians is still clear, the intricate details of smaller sections are close to non-existent. This gives us only a certain portion of the details of the walks, and doesn't capture most of the energy from the data. This is shown by the energy calculations using the `utils` functions. These calculations are as follows:

- **Energy in the Principal Component 1:** 47.33%
- **Energy in the Principal Component 2:** 37.59%
- **Cumulative Energy in the First 2 Principal Components:** 84.92%
- **Cumulative Energy in the First 3 Principal Components:** 99.71%
- **Number of components needed for 90% Energy:** 3

Two principal components only capture 84.92% of the energy, which is less than 90%, but 3 components capture 99.71%, which is more than enough to capture most of the data. Figure 5 shows the paths of the first two pedestrians captured with the first 3 principal components, and it's clear that the intricate details of the paths are much clearer here.

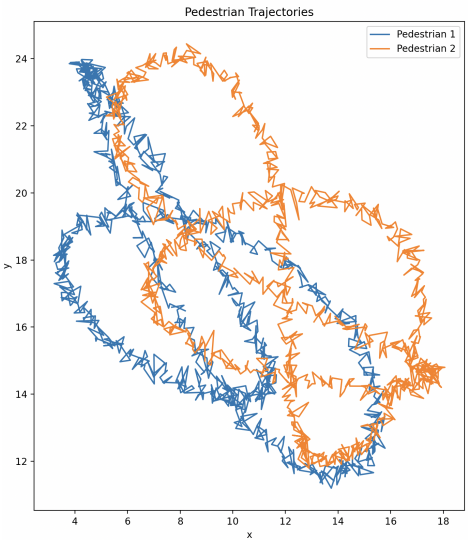


Figure 3: Paths of the two pedestrians with 30-dimensional data

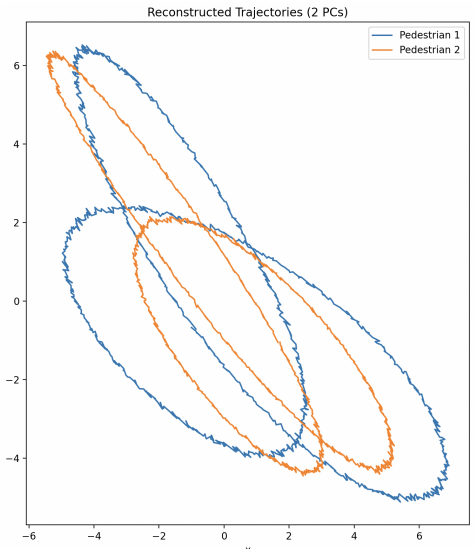


Figure 4: Paths of the two pedestrians with only the first 2 principal components

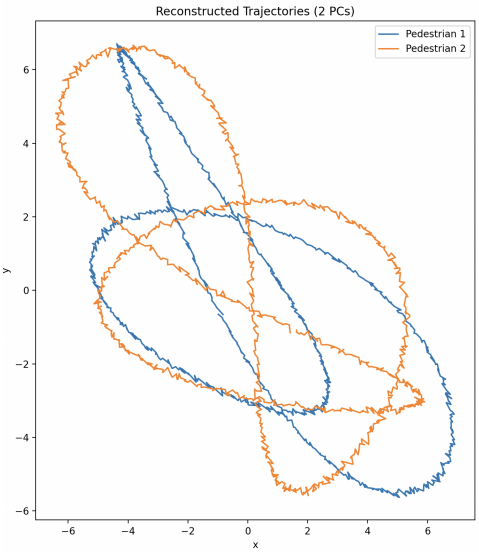


Figure 5: Paths of the two pedestrians with only the first 3 principal components

1.5 Answers to Questions from Page 1

The answers to questions posed on the first page of the exercise sheet are provided below:

(a) Time Estimate: PCA didn't take a long time to implement, 2 hours approximately, as the main ideas used in its implementation, like SVD, centering, etc., are covered in existing Python functions and made even easier with the `numpy` library and its features like broadcasting. Testing took even less time, 1 hour approximately, as PCA runs quite fast and outputs are basically instant.

(b) Accuracy of Representation: Depending on the principal components used for the implementation, we can represent the data in many different sets of accuracies, even going to 100%, if we use all the principal components. The accuracy was measured using the explained variance, referred to as energy. In general, the tasks required a number of principal components to represent the data, and the accuracy depended on whether these components had the necessary energy to capture the intricacies of the data. In the linear subspace and pedestrian path cases, most of the data is captured with only 1 and 3 principal components, while the raccoon image required a lot more components for a good representation.

(c) What was Learned: From the dataset perspective, pedestrian trajectories showed that many dimensions of the data contain redundant information, which is the reason only a few principal components were sufficient to capture most of the energy. This suggests that although the raw pedestrian data is a high-dimensional space, the actual behavior of the pedestrians has a much simpler structure. Also, PCA isn't the best in capturing non-linear variations of the data, such as curved paths and small changes in the general motion. This showed that it's important to take into account the underlying dataset and not take variance as the only measure, since while a linear space is a reasonable first approximation for the data structure, it doesn't capture all the complexity. From a methodological perspective, PCA has been shown to be a very fast and simple technique that can be used for many different tasks, such as dimensionality reduction, noise filtering, different visualizations, etc. This makes PCA a powerful and easy-to-use technique. The complexity and speed make it more attractive compared to more nuanced machine learning techniques. However, it doesn't seem to be taking into account many interpretations of the data, such as temporal or semantic meaning, by simply finding the axes with the maximum energy. This made interpretation and its adaptation to different data harder. While it can be a very efficient and powerful tool, understanding this limitation is crucial when dealing with it.

2 Diffusion Maps

The task is to implement the Diffusion Maps algorithm and apply the algorithm on a periodic dataset as well as a "Swiss Roll" manifold, then analyze the trajectory data of pedestrians.

2.1 The Diffusion Maps Algorithm

2.1.1 Overview

For the task, the Diffusion Maps algorithm is implemented following the 10-step procedure in the exercise sheet. The input dataset is $N \times 2$ matrix. The first step is to return a distance matrix D, with each element represents the Euclidean distance between two data points. As the next step, we set $\epsilon = 5\%$ of the diameter of the dataset, which is the maximum entry in the distance matrix D. This parameter defines the scale at which the data's geometry is analyzed. Then we transfer the distance matrix into an affinity matrix W by $W_{ij} = \exp(-D_{ij}^2/\epsilon)$, where a smaller distance results W_{ij} closer to 1, and a larger distance results in W_{ij} closer to 0. Then, a diagonal matrix P is constructed with $P_{ii} = \sum_{j=1}^{N} W_{ij}$, and this value represents the local density of each data point within the graph structure. The next step is to normalize W to form the kernel matrix with $K = P^{-1}WP^{-1}$. This normalization step is equivalent to an element-wise operation $K_{ij} = \frac{W_{ij}}{P_{ii}P_{jj}}$, which aims to remove the effect of non-uniform sampling density from the data. This step is the case $\alpha = 1$ in [7], and is an approximation of the heat kernel. Then from a diagonal normalization matrix with $Q_{ii} = \sum_{j=1}^{N} K_{ij}$. Here we want to analyze the operator $T = Q^{-1}K$, however, T is asymmetric. As alternative, we create $\hat{T} = Q^{-1/2}KQ^{-1/2}$, which is symmetric and has the same spectrum with T. After eigendecomposition of $\hat{T}$, we can compute eigenvalues and eigenvectors of T by $\lambda_l^2 = a_l^{1/\epsilon}$ and $\phi_l = Q^{-1/2}v_l$. We can then use λ_l^2 and ϕ_l to create a low-dimensional embedding and calculate diffusion distances.

2.1.2 Code

The implementation in Python follows the exact above steps, and the algorithm is implemented as a series of functions within a `utils.py` module. The implementation relies on the `NumPy` library for numerical array operations and `SciPy` for KDTree-based distance calculation and eigendecomposition.
Moreover, it is important to note the usage of `np.flip()` function. As the eigenvalues and eigenvectors are calculated in ascending order, we have to apply `np.flip()` to the eigenvalues to convert the result in descending order, and also apply to the eigenvector matrix to match with the eigenvalues.

2.2 Diffusion Maps and Fourier Analysis

2.2.1 Data Visualization

To demonstrate the similarity of Diffusion Maps and Fourier analysis, a periodic dataset with $N = 1000$ data points is generated and visualized. The dataset is sampled uniformly from a unit circle in a Euclidean space. It represents a one-dimensional manifold embedded in $\mathbb{R}^2$, as it takes one parameter t_k to decide the position of one data point in the dataset. This dataset is generated firstly by set $t_k = \frac{2\pi k}{N+1}$, then map the angular value with $x = sin(t_k)$ and $y = cos(t_k)$. The result is a $N \times 2$ matrix where each row represents a data point. A visualization of the dataset is shown in Figure 6.

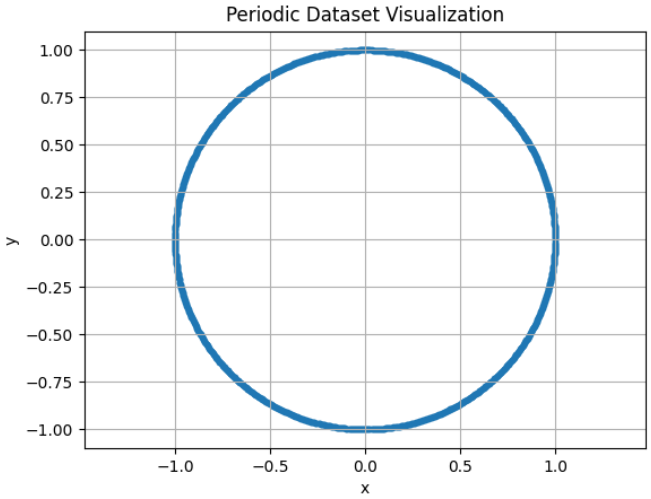


Figure 6: Visualization of periodic dataset

2.2.2 Algorithm Application

Then we implement the diffusion maps algorithm on the dataset. Set `n_eig_vals` $= 5$, as five eigenfunctions ϕ_l associated to the largest eigenvalues λ_l are wanted. The result is shown as figure 7. The figure displays the first five non-trivial eigenfunctions, denoted ϕ_1 through ϕ_5. The horizontal axis represents the angular parameter t_k, which parametrizes the position on the circle, while the vertical axis shows the value of each eigenfunction at the corresponding data points. The plot demonstrates that the eigenfunctions appear in pairs with the same frequency but different phases. For example, the pair ϕ_1 and ϕ_2 corresponds to the fundamental frequency, appearing as a sine and cosine wave, respectively. Similarly, the pair ϕ_3 and ϕ_4 represents the next harmonic with double the frequency. The fifth eigenfunction, ϕ_5, is a sine wave with an even higher frequency, corresponding to the third harmonic mode.

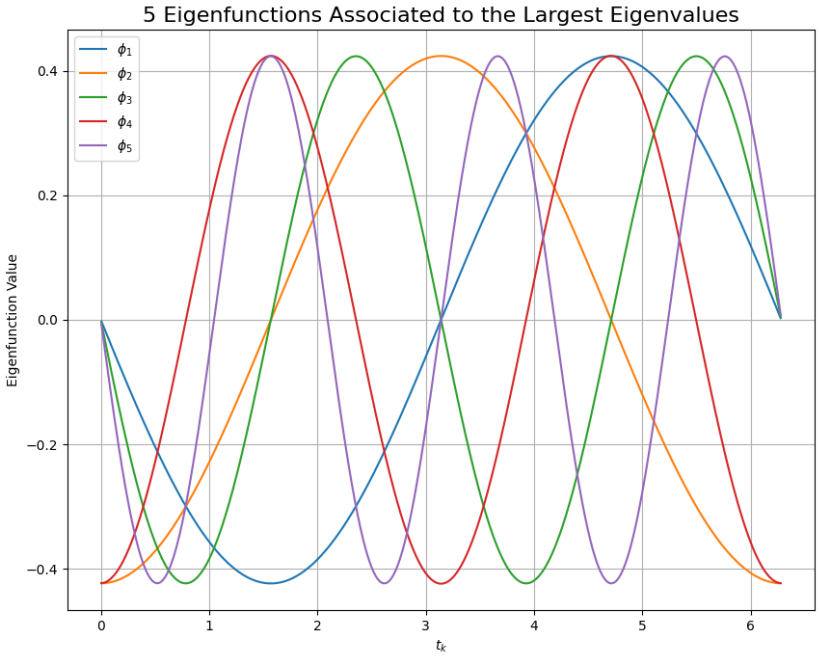


Figure 7: 5 Eigenfunctions associated with 5 largest Eigenvalues on a periodic dataset

2.2.3 Bonus: Relation of the Results to Fourier Analysis

The computed eigenfunctions on the unit circle dataset are sine and cosine waves, which are exactly the basis functions that Fourier analysis uses to decompose a signal. So both diffusion maps and Fourier analysis can find the basis function of data by applying eigen-decomposition to an operator, and therefore reveal the intrinsic structure of data.

2.3 "Swiss Roll" Manifold

2.3.1 Data Visualization

The task aims to apply the Diffusion Maps algorithm to a "Swiss Roll" manifold. The first step is to generate and visualize the "Swiss Roll" dataset. As shown in Figure 8, the "Swiss Roll" dataset is a two-dimensional manifold embedded in $\mathbb{R}^3$ space. The definition is

$$X = \left\{x_k \in \mathbb{R}^3\right\}_{k=1}^{N}, \quad x_k = (u_k \cos(u_k), v_k, u_k \sin(u_k)),$$

where $u_k \in [3/2\pi, 3\pi]$ and $v_k \in [0, 21]$ are chosen uniformly at random. The parameter u_k controls the "unfolding" extent of the data points along the spiral direction. Note that by using `make_swiss_roll`, the value range of u_k can not be modified, and therefore the figure is different from the definition of u_k. The parameter v_k describes simply the y-axis, which represents the "thickness" of the "Swiss Roll" along the y-axis. In the figure, a consistent color map is implemented to describe u_k. A smaller u_k value results in color blue, and a larger u_k value results in color yellow.

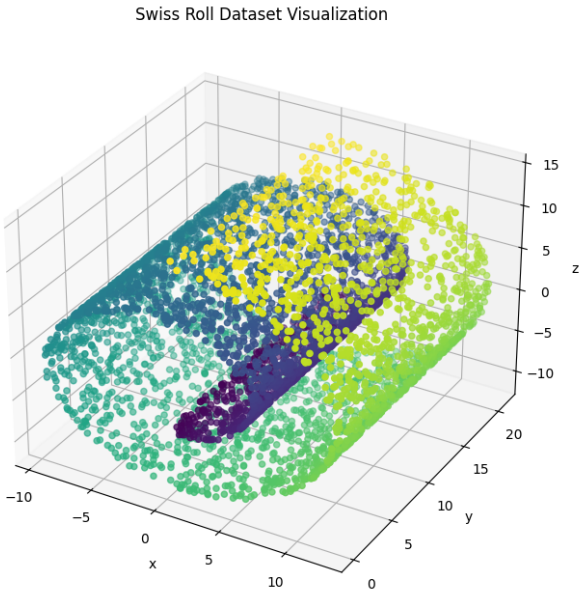


Figure 8: "Swiss Roll" Dataset Visualization

2.3.2 Algorithm Application

To analyze the intrinsic structure of the non-linear Swiss roll manifold, the implemented Diffusion Maps algorithm was used to compute the first ten non-trivial eigenfunctions (ϕ_1 to ϕ_{10}). This is fulfilled by calling `diffusion_map` function in `utils`. The results are shown in 9, as the first non-constant eigenfunction ϕ_1 against the other eigenfunctions(ϕ_2 to ϕ_{10}) in a 3×3 array of plots.

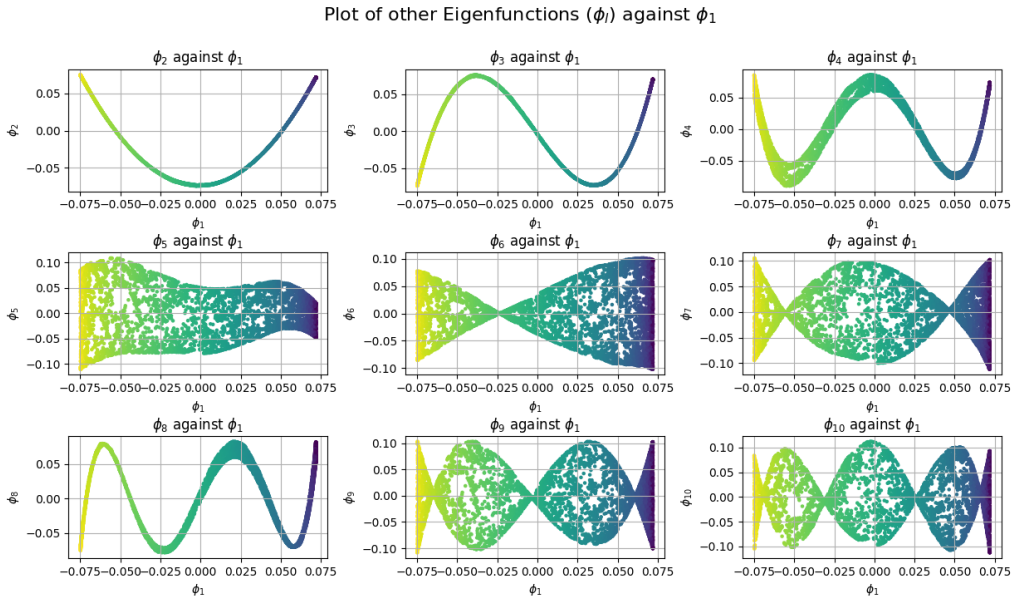


Figure 9: Plot of other eigenfuncitons(ϕ_2 to ϕ_{10}) against ϕ_1 with $N = 5000$ data points

Analysis:

- As demonstrated in the figure, ϕ_l is no longer a simple function of ϕ_1 at $l = 5$. Starting from the subplot of $l = 5$, the figures start to fold themselves, and one ϕ_1 data point corresponds to multiple ϕ_l data points. It shows that ϕ_l starts to capture features that are no longer dependent on ϕ_1. We can observe from the color in the plots (yellow to the left side and blue to the right side) that there exists a monotonic relationship between ϕ_1 and u_k, so from $l = 5$ ϕ_l start to capture features that are possibly related to y-axis.

- Run PCA analysis to get the 3 principal components of the "Swiss Roll" dataset. Here we import `PCA` from `sklearn.decomposition`, then print `pca.components_` and `pca.explained_variance_ratio` to get PCA vectors and energy of each principal component. Results are as follows:
  Principal Components:

$$\begin{bmatrix} 0.5082909 & -0.01762039 & 0.86100516 \\ 0.86013941 & 0.05964981 & -0.50655908 \\ -0.04243303 & 0.99806384 & 0.04547545 \end{bmatrix}$$

  Explained variance ratio: [0.3945 0.3184 0.2871].
  The result indicates that the energy in principle component 1 is 39.45%, the energy in principle component 2 is 31.84%, and the energy in principle component 3 is 28.17%. Therefore, the cumulative energy in the first 2 principal components is 71.29%, and the cumulative energy in the first 3 principal components is 99.46%. Two principal components only capture 71.29% of the energy, which is way less than the required 90%. Therefore, it is impossible to only use two principal components to represent the data: it is to map 3D points to a 2D surface, which breaks the intrinsic topological relationships of data points in different layers.

- As shown is Figure 10, if we use only $N = 1000$ data points instead of $N = 5000$ data points, the resulting embeddings show the same underlying structures. However, with fewer samples ($N = 1000$), the approximation is less accurate and significantly noisier. This illustrates that a denser sampling of the manifold allows for a more accurate approximation.

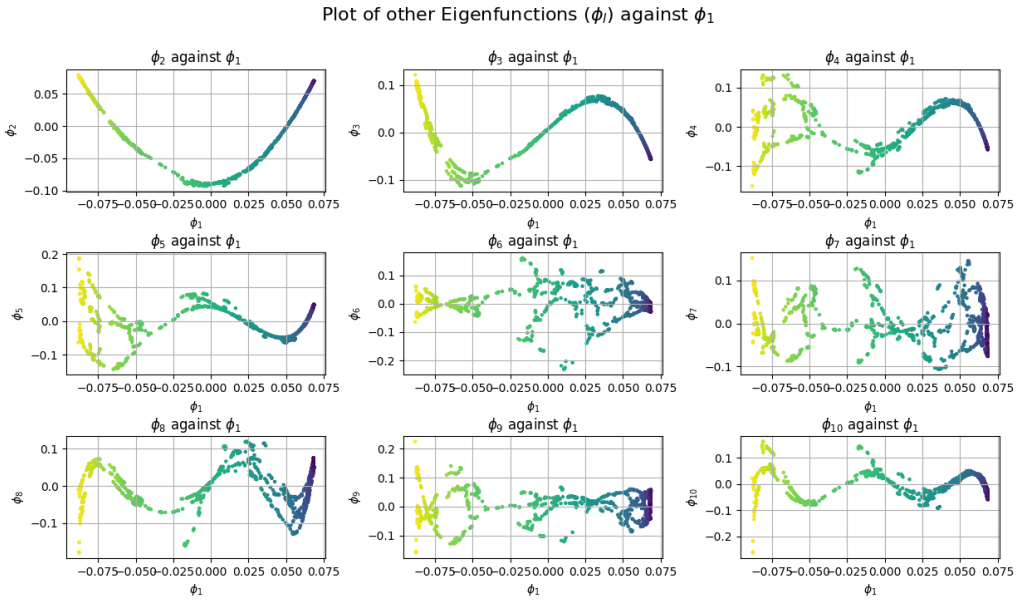


Figure 10: Plot of other eigenfuncitons(ϕ_2 to ϕ_{10}) against ϕ_1 with $N = 1000$ data points

2.4 Trajectory Data Analysis

2.4.1 Diffusion Map Embedding

In this section, the Diffusion Map algorithm is applied to the pedestrian trajectory dataset to analyze its underlying structure. The primary goal is to determine the minimum number of eigenfunctions required to obtain an accurate representation of the dataset, i.e., no intersections of the curves. The generated embedding must not self-intersect, because a self-intersection in the low-dimensional embedding would imply that two distinct original data points are mapped to the exact same point. In this case, the topological structure of the original trajectory dataset is broken. This form of information loss in dimensionality reduction is not acceptable, rather than the kind of information loss we expect, e.g., noise and other non-significant dynamic motions.

2.4.2 Result Analysis

In Figure 11, four different eigenfunctions ϕ_2, ϕ_3, ϕ_4 and ϕ_5 are plotted against ϕ_1 in a 2×2 array of subplots, and the function ϕ_1 is on the horizontal axis. We can observe from the embedding created by the first two non-constant eigenfunctions(ϕ_1 and ϕ_2) that this representation forms a perfect circle. The geometric shape is completely free of self-intersections, and this demonstrates that a minimum of only two eigenfunctions is sufficient to accurately represent the data manifold. This result indicates that the pedestrian motion is periodic, as the figure is a closed loop. In contrast, the other subplots, which show the relationship between higher-order eigenfunctions (ϕ_2, ϕ_3, ϕ_4 and ϕ_5) to ϕ_1, all have intersections. That means for higher-order eigenfunctions, the feature they captured is more complex and can not be projected to a two-dimensional space without loss of critical information. The result highlights the effectiveness of the Diffusion Map algorithm in identifying the intrinsic, low-dimensional geometry of a complex dynamic system.

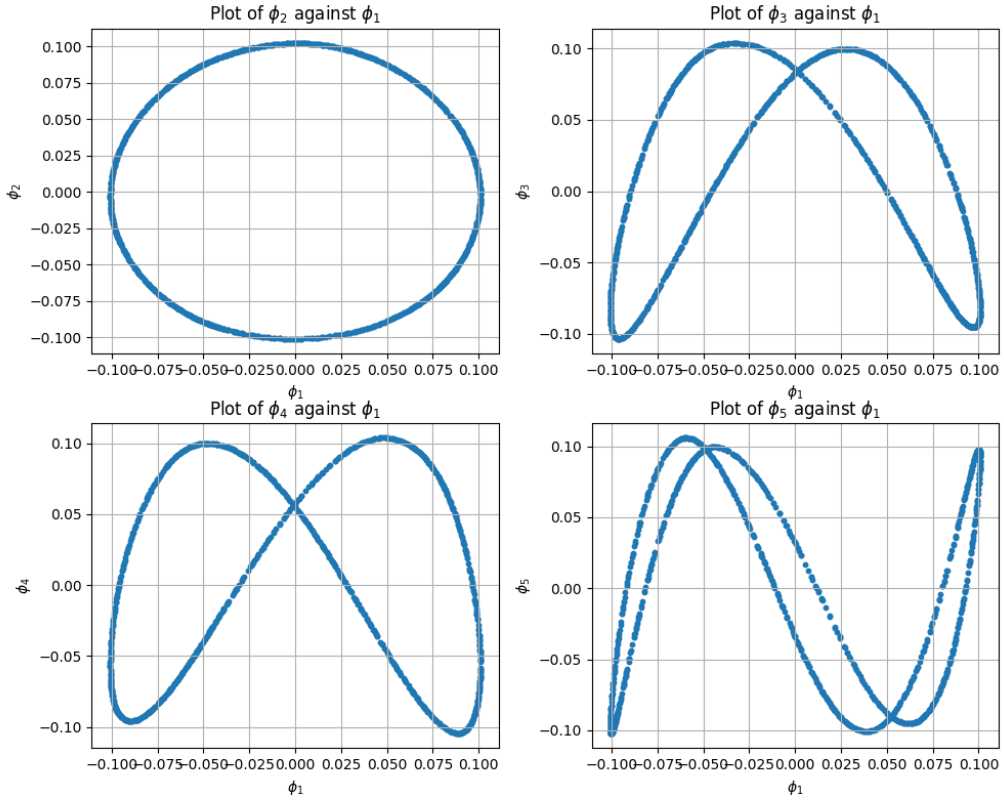


Figure 11: Plot of other eigenfuncitons(ϕ_2 to ϕ_5) against ϕ_1

2.5 Bonus: Visualization with Datafold Library

After having downloaded `Datafold` library, we modify the code from generating s-curve manifold to generating "swiss roll" dataset with `make_swiss_roll`.

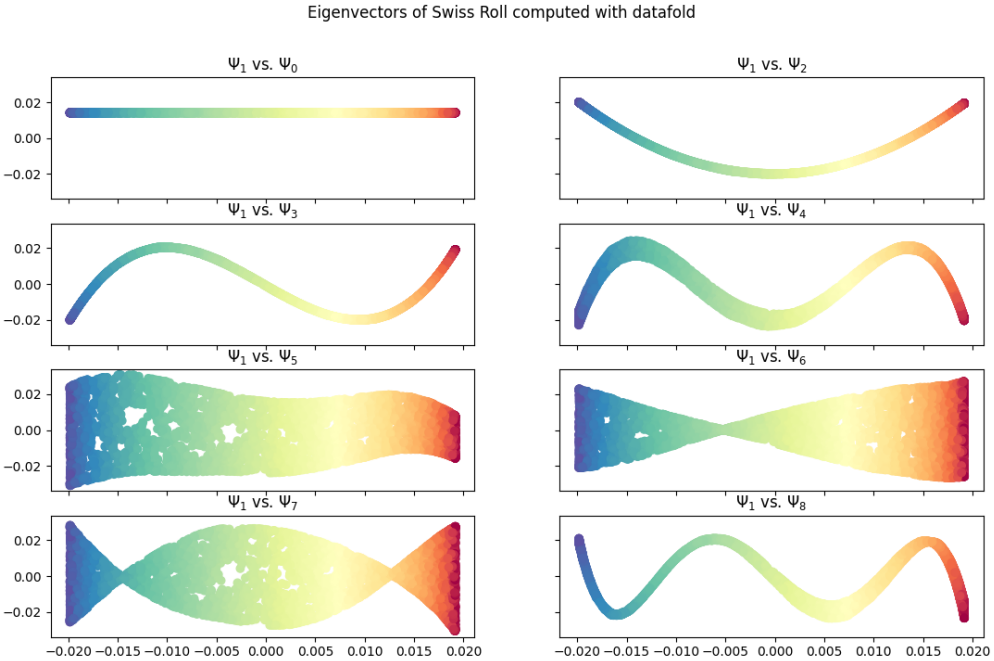


Figure 12: Plot of other eigenfuncitons (ϕ_0, ϕ_2 to ϕ_8) against ϕ_1 with $N = 5000$ data points

From Figure 12, we can observe that the code yields the same result as our implementation in Section 2.3. Therefore, our implementation is verified. However, the code with `datafold` is much simpler, as many modules are already pre-defined. The user only needs to initialize an object `dmap`, then call a method `dmap.fit()`. At the same time, the code runs much faster with `datafold` library. On MacOS, running the code in Section 2.3 takes $\approx 50s$, and with `datafold` it takes only several seconds. Considering the figure is also neater and more visually appealing, `datafold` library is arguably a better tool and can be utilized in professional research.

2.6 Answers to Questions from Page 1

The answers to questions posed on the first page of the exercise sheet are provided below:

(a) Time Estimate: The implementation of the Diffusion Maps algorithm is straightforward, as it follows the instructions of the exercise sheet. Meanwhile, the application of the Diffusion Maps algorithm in three parts and understanding the algorithm is time-consuming. Including coding, debugging, and finishing the report, the exercise took around 20 hours of work.

(b) Accuracy of Representation: For Diffusion Maps, the measure of accuracy is qualitative and focuses on how well the low-dimensional embedding can preserve the geometric property. For example, for part 2, the "Swiss Roll" dataset, the functional relationship between ϕ_1 and other eigenfunction values is visually interpretable. In part 3, the measure of accuracy is whether the embeddings of the pedestrian trajectory are free of self-intersection. The results show a high degree of accuracy, as an embedding with two eigenfunctions produced a nearly perfect circle. For PCA analysis, the measure of accuracy is quantitative. The explained variance ratio results in very high accuracy, almost up to 100%.

(c) What was Learned: About the Diffusion Maps method, what was learned is that it is a powerful dimensional reduction technique. The method outperforms PCA in terms of non-linear datasets, such as in the "Swiss Roll" dataset. It allows uncovering the underlying structure of the dynamic system and reveals the fundamental mode of behavior. That is, how can an arbitrary function defined on a dataset be decomposed into a combination of basis functions through Diffusion Maps? About the dataset, for example, for the pedestrian trajectory data, what was learned is that this seemingly complex system is governed by a simple and periodic dynamic, as the embedding is a perfect circle in a low dimension.

3 Task 3: Variational Auto-Encoder (VAE)

Variational Autoencoder (VAE) is a type of generative machine learning model that learns to encode data into a compressed representation while also allowing for the generation of new, similar data[6]. The model takes input data (in our case, images), learns (encoder) to transform the high dimensionality of input data to a lower-dimensional latent vector. The decoder of the model learns to revert from the latent vector to re-generate the original input at a higher dimension. The autoencoder, a coming together of encoder, latent space, and decoder, is trained to minimize the difference between the original and reconstructed data.[8]
In this part of the task, the input is MNIST (a database of handwritten numbers). The final model should be able to recreate numbers from the latent space and also generate them.

3.1 Activation functions

In VAE, the encoder outputs the parameters of the approximate posterior distribution $q(z|x)$, which is usually modeled as a Gaussian. The outputs are $\mu(x)$ and $\sigma(x)$ of the latent variable distribution. The encoding of x as a distribution (in contrast to fixed latent vectors in autoencoders) lets the model capture uncertainties and makes the latent space continuous, which helps make the generation of new samples better.
For the calculation of the mean, no activation function is needed as it can be both positive and negative, and therefore no restriction is required. The standard deviation, which can only be positive, needs to have an exponentiation (the square of a negative number is positive). The insurance that standard deviation is positive is done in the function `reparameterize()`.
For likelihood, which is the decoder, a sigmoid activation is needed to constrain pixel values between 0 and 1. This is because MNIST images are greyscale images with pixel activate between 0 and 1. In order to recreate the same images, their output should also be constrained between 0 and 1.

3.2 Good reconstruction but bad generated digits

A good reconstruction of digits, but a bad generation usually means that the latent space isn't well-structured. One reason for bad generation could be when the model learns to ignore the latent variables[5]. This is known as posterior collapse and occurs when the model is powerful enough to 'memorize' the input to be able to recreate the images without considering the encoded information.

Another cause for a bad reconstruction could be an imbalance between Reconstruction Loss and Kullback-Leibler (KL) divergence[1]. A high reconstruction loss leads to a model overfit, leading to the decoder ignoring the latent code (posterior collapse). On the other hand, too much KL divergence means that the model could forget the input data and output blurry or generic samples. A solution for this is to multiply the KL Divergence by a scaling factor β. This would mean:

$$\mathcal{L}_{\text{ELBO}}(\theta, \phi) = \mathbb{E}_{p_{\mathcal{D}}(\mathbf{x})} \left[\mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})} \left[\log p_\theta(\mathbf{x}|\mathbf{z}) \right] - \beta \cdot \text{KL} \left(q_\phi(\mathbf{z}|\mathbf{x}) \,\|\, p(\mathbf{z}) \right) \right] \tag{3}$$

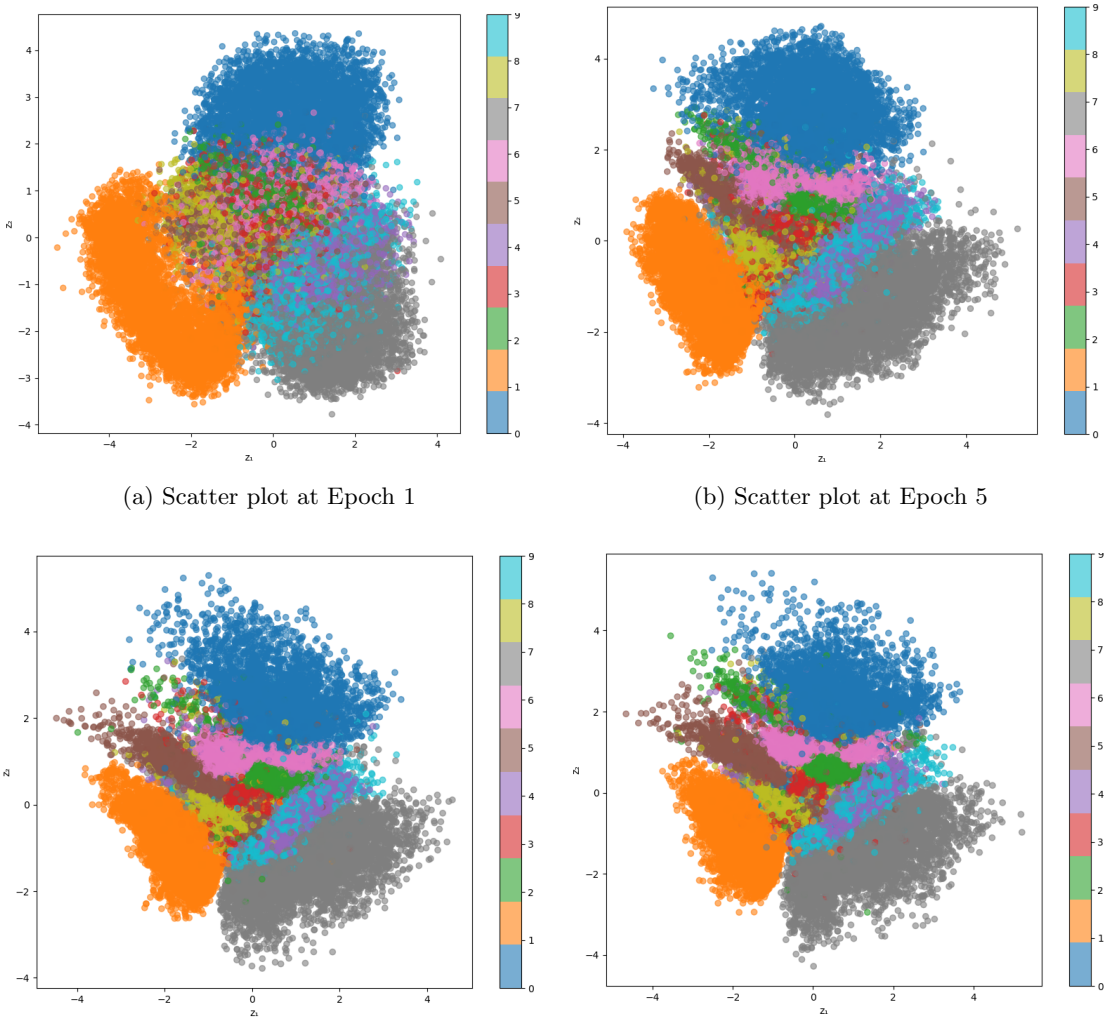
- $\beta = 1$: standard VAE
- $\beta > 1$: encourages more structured latent space, risk of worse reconstructions
- $\beta < 1$: encourages better reconstructions, risk of overfitting/latent collapse

3.3 VAE training

After filling in the TODOs from the source course, it was possible to train the model and get Figures 13, 14, and 15. These images were generated for the learning rate of 0.001 at a batch size of 128 for training and 2 hidden layers with 256 units for the encoder and decoder.

Figure 13 shows how the latent space representation of the VAE evolves during training on the MNIST dataset from epoch 1 to epoch 50. With each full cycle of the training through the dataset, it can be seen that the clustering of the points gets better with an increase in the epoch number. In 13 (a), the latent space is disorganized, with data points with different labels overlapping and no concrete clustering. This slowly gets better through Figures 13 (b) and (c). In Figure 13, clusters for numbers are clearly visible and distinct. This progression shows that the model is learning to encode digit identity in the latent space.

3.3.1 Scatter plot



(a) Scatter plot at Epoch 1

(b) Scatter plot at Epoch 5

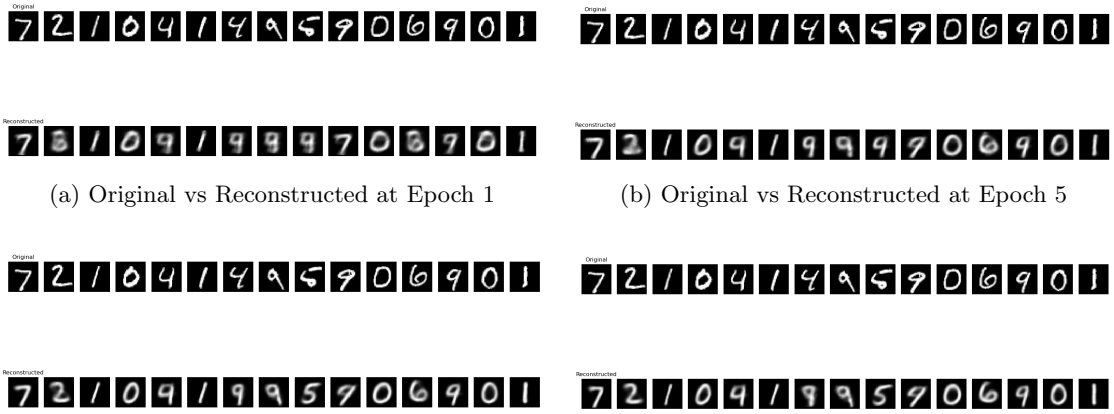
(c) Scatter plot at Epoch 25

(d) Scatter plot at Epoch 50

Figure 13: Scatter plots at different Epoch values for 2-dimensional latent representation

3.3.2 Original vs Reconstructed

Figure 14 shows the progression of the original image vs its reconstruction through epoch values from 0 to 50. As visualized with Figure 13, there increase in epoch number certainly makes a difference in the reconstruction as there is a direct correlation between epoch number and reconstruction quality. In Figure 13 (a), some reconstructions are blurry and reconstructed in a way that they could be mistaken for a different number (for example, 2 in position 2 can be confused with 8 or 3). This is further emphasized by the fact that the scatter plot in Figure 13 (a) shows points for numbers: 2, 3, and 8 overlapping each other at the same region in the latent representation. The reconstruction of 2, as well as other numbers, does get better with increasing epoch, similarly to how the clusters are separable in Figure 13. There are still considerable inaccuracies at the epoch value of 50, but the reconstructed numbers are visually more distinguishable.



(a) Original vs Reconstructed at Epoch 1

(b) Original vs Reconstructed at Epoch 5

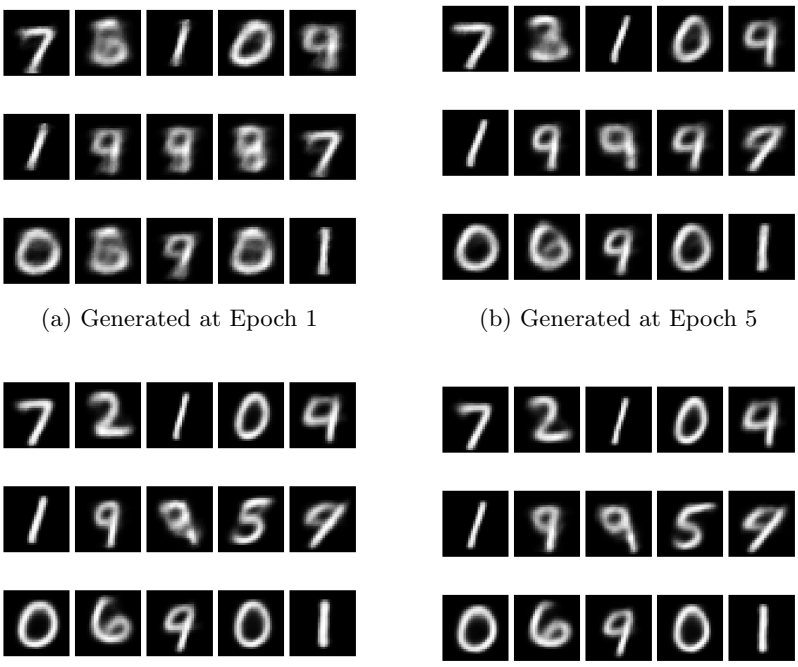
(c) Original vs Reconstructed at Epoch 25

(d) Original vs Reconstructed at Epoch 50

Figure 14: Original vs Reconstructed at different Epoch values for 2-dimensional latent representation

3.3.3 Generated

In Figure 15, the exact numbers from Figure 14 are reconstructed, again at different Epoch values. The generation only uses a single batch for efficiency, speed, and reproducibility. As with the scatter plot and reconstruction, the generation of digits shows the exact same pattern, with numbers being clearer and intelligible at a higher epoch value compared to a lower one. This means that the model learns to encode digits into a structured latent space and decodes from this space to reconstruct or generate meaningful images. Since there is also a stagnation in terms of the increase in quality of reconstructed and generated images (as well as clustering), we can say that the model stabilizes after sufficient training epochs. However, it is still clear that some of the regenerated digits are not what they are supposed to be (this is known since the 'originals' from Figure 14 are supposed to be reconstructed. Even the ones that are accurate are blurry. This is because regeneration from 2D latent space- where a lot of information about each number is already small to represent all characteristics of a number, has overlapping coarse and averaged representations leading to blurry outputs.



(a) Generated at Epoch 1

(b) Generated at Epoch 5

(c) Generated at Epoch 25

(d) Generated at Epoch 50

Figure 15: Generated numbers at different Epoch values for 2-dimensional latent representation

3.4 Loss curve

Figure 16 is the loss curve that shows how the loss value changes over time during training and testing across various epochs. The training loss shows how well the model fits the training data, and the test shows how the model generalizes to test(unseen) data. The decrease in training loss shows that the model is learning from the data, and the decrease in test loss signifies an improvement in the generalization of the test samples. The loss curve helps show issues such as overfitting (decreasing training loss; increasing test loss), underfitting (high test and training loss), or noisy/unstable learning (high fluctuation in the curves). However, this is not the case in our model, as the losses decrease and then stabilize with an increase in epoch, and are in the typical range of loss (80-120 per sample) for MNIST data.

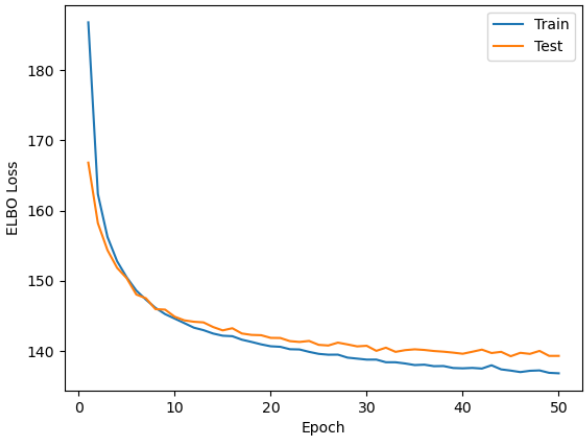


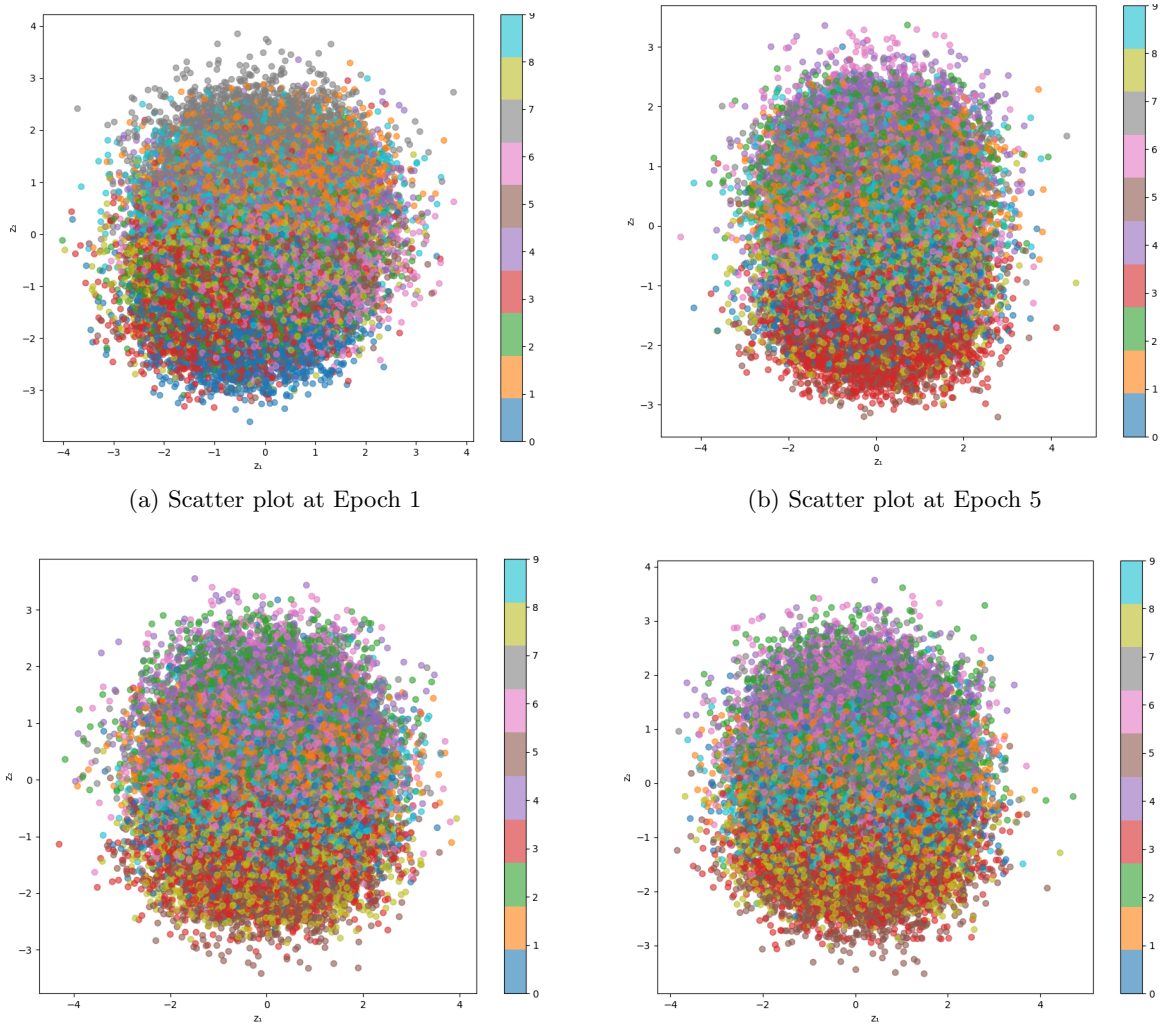
Figure 16: Loss curve (per sample) for 2D latent representation

3.5 VAE training with 32-dimensional

In this part of the task, the dimension of the latent space of the VAE is taken to increased to 32 (from 2 in Section 3.3). This means that each input data point is encoded into a vector with 32 numbers. This change should change the plots, reconstruction, and generation in comparison to a 2-dimensional latent space. This change means a higher capacity to encode features such as digit shapes, thickness, slant, etc, as well as allow for finer distinctions between samples.

3.5.1 Scatter plot

In Figure 17, a scatter plot is again shown for various epoch values from 0 to 50. This time, however, there is little to no progression in terms of distinct clustering between numbers. This is because a 32-dimensional representation of data is shown on a 2-dimensional plane. In reality, the clustering should be getting better with increasing epoch, as was the case in Figure 13.



(a) Scatter plot at Epoch 1

(b) Scatter plot at Epoch 5

(c) Scatter plot at Epoch 25

(d) Scatter plot at Epoch 50

Figure 17: Scatter plots at different Epoch values for 32-dimensional latent representation

3.5.2 Original vs Reconstructed

Since a 32D latent space allows for better storage of features of the original number, the reconstruction is expected to be closer to the original than the model with a 2D space[9] [4]. However, it is still expected that the behavior of the reconstruction getting better for higher epochs still holds.

Comparing Figures 14 (a) and 18 (a), no big difference can be seen in terms of which reconstruction is better. In both cases, the reconstruction is blurry and in some cases, not accurate. The reason behind this is most likely that the clustering of individual numbers in the latent space is not individual but is overlapping. At Epoch 5, however, the 32-D model is already better at recreating the numbers. This is again explainable as individual traits of each number are better captured in the higher-dimensional space. As the epoch value increases, the reconstruction gets better in both cases, but considerably so in the model with higher dimensions, as they are accurate and clearer. In the lower dimension, there are still errors (2, 5, and 6 are reconstructed as an 8 while 4 is reconstructed as a 9 even at epoch = 50), and the reconstructed images are blurry.

(a) Original vs Reconstructed at Epoch 1

(b) Original vs Reconstructed at Epoch 5

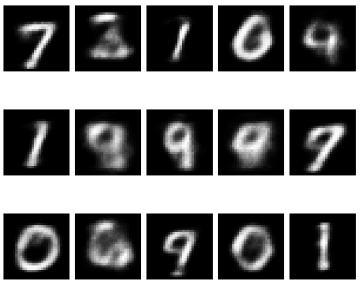
(c) Original vs Reconstructed at Epoch 25

(d) Original vs Reconstructed at Epoch 50

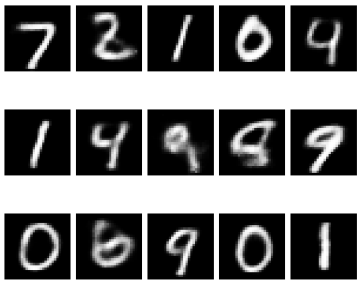
Figure 18: Original vs Reconstructed at different Epoch values for 32-dimensional latent representation

3.5.3 Generated

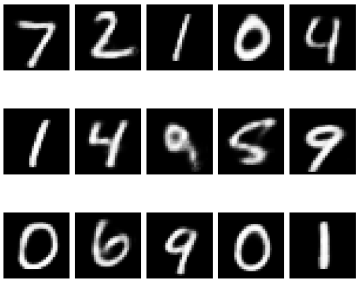
Dai and Wipf (2019) state that increasing the latent dimensionality improves the reconstruction quality but may hurt the generative quality. This could be because in a high-dimensional latent space, the encoder may not use all dimensions meaningfully, with some dimensions carrying no information (i.e., become "ignored") or the KL divergence term becoming harder to balance, and therefore the model only optimizing reconstruction[3]. Also, it is possible that in higher dimensions, the latent space may become sparse, spread out, or non-continuous. Sampling randomly from this may mean landing in between modes or in dead zones, producing meaningless outputs[5].



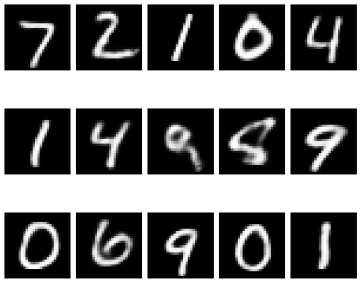
(a) Generated at Epoch 1



(b) Generated at Epoch 5



(c) Generated at Epoch 25



(d) Generated at Epoch 50

Figure 19: Generated numbers at different Epoch values for 2-dimensional latent representation

3.5.4 Loss Curve

Figure 20 shows the training vs test losses over epochs. When comparing it with Figure 16, a few differences are easily noticeable:

- Higher loss for 2D
- Faster stabilization in 32D
- Smaller gap between test and training losses in 32D

A higher-dimensional latent space typically has a lower loss, as a larger latent space gives a higher degree of freedom for the model to learn the underlying structure of the data and better represent complex relationships and patterns. This leads to a more accurate and compressed representation of the data and therefore a lower loss[2]. This also corresponds to the faster stabilization of loss. This would mean that the reconstruction loss is lower and outweighs the KL loss.

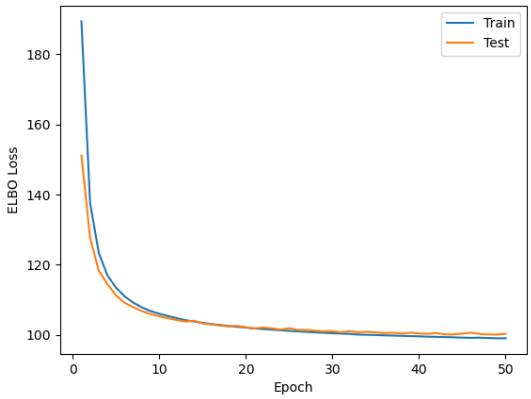


Figure 20: Loss curve (per sample) for 32D latent representation

3.6 Answers to Questions from Page 1

The answers to questions posed on the first page of the exercise sheet are provided below:

(a) Time Estimate: To implement the method was pretty straightforward due to the TODOs in the source code. It took a total of two days of work (not working on the tasks the whole time) to complete it, and a further two to fix bugs, create proper graphs, and more.

(b) Accuracy of Representation: In terms of data representation, two quality measures are feasible. The first one is obviously visual. At different epochs, the reconstructed digits seem to look different. Since the originals are also alongside, it can be determined by just looking at them how well the reconstruction is. Sometimes, when individual digits are confusing (8 looking like a 3), it is deemed to be a bad reconstruction. The quantitative way would be to dive into the losses. However, this is not done.

(c) What was Learned: A simple yet subtle learning from the dataset is the diversity of how humans write digits as the dataset is images of hand-written digits and how, because of this diversity, some of the digits seem to have overlapping characteristics. From the model, the role of the latent space was understood, and how it affects reconstruction and generation. An excerpt on the balancing between having a good reconstruction and bad generation always showed up when looking into details about the model on the internet, but this was not an issue during the implementation.

4 Task 4: Fire Evacuation for the MI Building

The task involves revising the plan for reconsidering the fire evacuation plan for the Mathematics and Informatics (MI) building due to an increasing number of students. The core of the task is to learn the distribution of people within the building, denoted as $p(x)$, using data from tracked students and employees. We re-use a Variational Auto-Encoder (VAE) with different instantiations (recommended in the report), which is a deep generative model. We have already described it in 3, so we'll not go into detail again. We will describe the modification to layers and loss function in Subsection 4.2. A model is needed to learn this distribution from a provided dataset of x-y positions.

The primary goal is to use the learned model for a first experiment: estimating the number of people that is critical for the building. This is simplified by focusing on a sensitive area in front of the main entrance, defined by the rectangle with corners at (130, 70) and (150, 50), where the number of people should not exceed 100.

4.1 Visualisation of Data and Pre-processing

The provided `FireEvac` dataset was loaded from local `npy` files. Before training, we visualized the dataset to understand its structure. As suggested in Exercise Sheet 3, the input data was also rescaled to a range of $[-1, 1]$ to help stabilize the training process of the neural network. The initial distribution of the training data is shown in Figure 21.

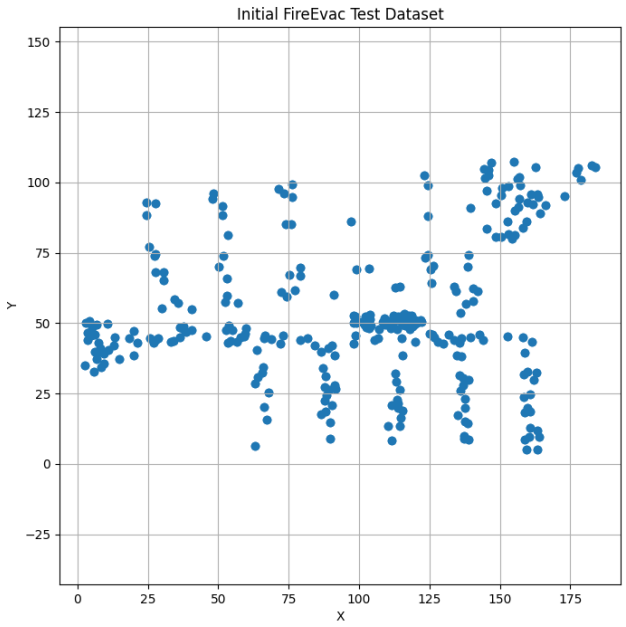


Figure 21: Scatter plot of the FireEvac training dataset; visualizing the spatial distribution of pedestrians.

4.2 VAE Model and Training

We have reused the VAE implementation from the previous MNIST task, described in Section 3.

4.2.1 Model Architecture

- **Encoder:** Model takes a 2D coordinate as an input. Processes it through two 64-unit (as recommended in the exercise sheet) dense layers (having ReLU activation), and outputs the params: mean and log-variance. For a 2-D multivariate Gaussian distribution. We did some experiments and came to the conclusion that the recommended parameters give the optimal results (given the resources and time used).

- **Decoder:** Takes a 2-D vector sampled from the latent space, processes it through two 64-unit dense layers (ReLU activation), and outputs a 2D coordinate. The final activation function of the decoder is the hyperbolic tangent $TanH$, which naturally maps the output to the $[-1, 1]$ range, matching our data pre-processing step.

4.2.2 Loss Function

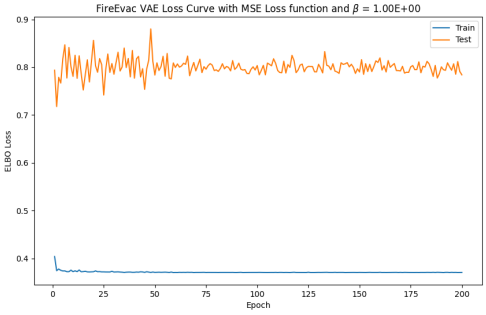
The selection of an appropriate loss function was a critical factor in the successful training of the VAE for this particular task. The standard `ELBO` loss was adapted significantly to suit the continuous, two-dimensional coordinate data.

Initial attempts using a standard Mean Squared Error (MSE) for the reconstruction loss yielded suboptimal results, with reconstructions often appearing blurry or failing to capture the finer details of the data distribution. The training was also less stable.

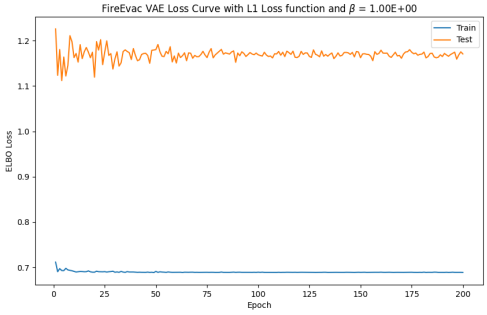
We got a breakthrough with a two-part modification leading to a new loss function. This is implemented in the `elbo_loss_l1` function in `utility.py`:

- **Switching to L1 Loss:** The reconstruction term was changed from our initially chosen Mean Squared Error (MSE) Loss function to L1 Loss (sum of absolute differences). L1 loss for our case is more robust to outliers than MSE. For our 2D coordinate data, it means that a few points are far from their original position. And it does not disproportionately penalize the model. This, in our case, leads to a more stable training process and sharper, more accurate reconstructions.

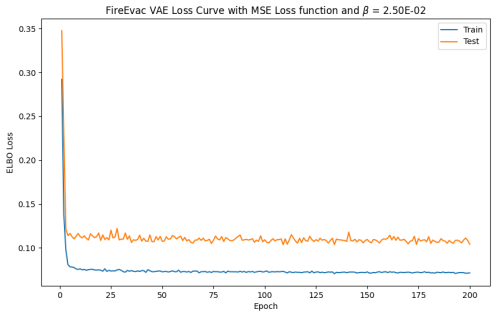
- **Introducing a Beta (β) Factor:** A scaling factor of β was applied to the Kullback-Leibler (KL) divergence term, to the initial formula proposed in the exercise sheet. We have already mentioned Equation 3 and a detailed description in Subsection 3.2. For a recap, this hyperparameter is crucial for balancing the two competing objectives of the VAE: accurate data reconstruction and a regularized latent space suitable for generation. The chosen value provides an excellent trade-off, allowing the model to prioritize minimizing the reconstruction error, which was essential for capturing the complex geometry of the `FireEvac` dataset.



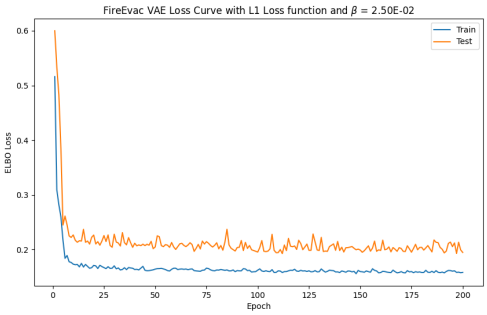
(a) MSE Loss without β scaling ($\beta = 1.0$).



(b) L1 Loss without β scaling ($\beta = 1.0$).



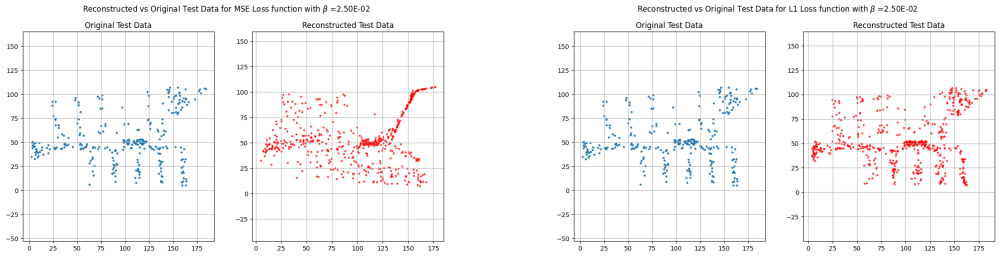
(c) MSE Loss with $\beta = 0.025$.



(d) L1 Loss with $\beta = 0.025$ (Final Model).

Figure 22: Comparison of training and test loss curves for different loss functions (MSE vs. L1) and (β) values. The introduction of a small β value (d) resulted in the model being trained. Without beta (i.e. $\beta = 1$) configurations led to almost no training.

It is evident from Figure 22 that the training does not even happen without introducing β parameter, specifically $\beta < 1$. Furthermore, it seems that L1 and MSE loss perform similarly based on the loss value computation, but from Figure 23 it is evident that L1 is much better.



(a) Reconstruction with MSE Loss ($\beta = 0.025$). (b) Reconstruction with L1 Loss ($\beta = 0.025$).

Figure 23: Visual comparison of VAE reconstructions on the test set. Both models were trained with $\beta = 0.025$. The reconstruction using L1 loss (c) more accurately captures the sharp structure and density of the original data (a) compared to the reconstruction using MSE loss (b), which appears more diffuse.

This combination of L1 loss and a carefully tuned β factor proved highly effective, as seen from Figure 22 and Figure 23, enabling the model to converge to a state with both high-fidelity reconstructions and meaningful generated samples.

4.2.3 Training Process

The model was trained for 200 epochs using the Adam optimizer with a learning rate of 0.005, and the same configuration as described in Subsection 4.2.

4.2.4 Loss Curve

The progression of the previously specified ELBO loss (with our β param) for both the training and test sets can be seen from Figure 22d. Both losses decrease consistently and converge, indicating training success. The considerable marginal gap between the two curves suggests that the model generalizes well to unseen data, and there is enough to avoid significant overfitting.

4.2.5 Data Reconstruction and Generation

Qualitative results from Figure 24 and Figure 25 showcase our model's effectiveness, as requested by the exercise sheet.

- **Reconstruction:** Our VAE can successfully reconstruct the test data, and it captures the intricate shape of MI's accessible areas. Reconstruction is not completely perfect, but the overall structure is well-preserved.
- **Generation:** The thousand samples generated from our model's prior distribution closely follow the original data's layout. It indicates that our VAE has learned a meaningful representation of the underlying pedestrian distribution $p(x)$.

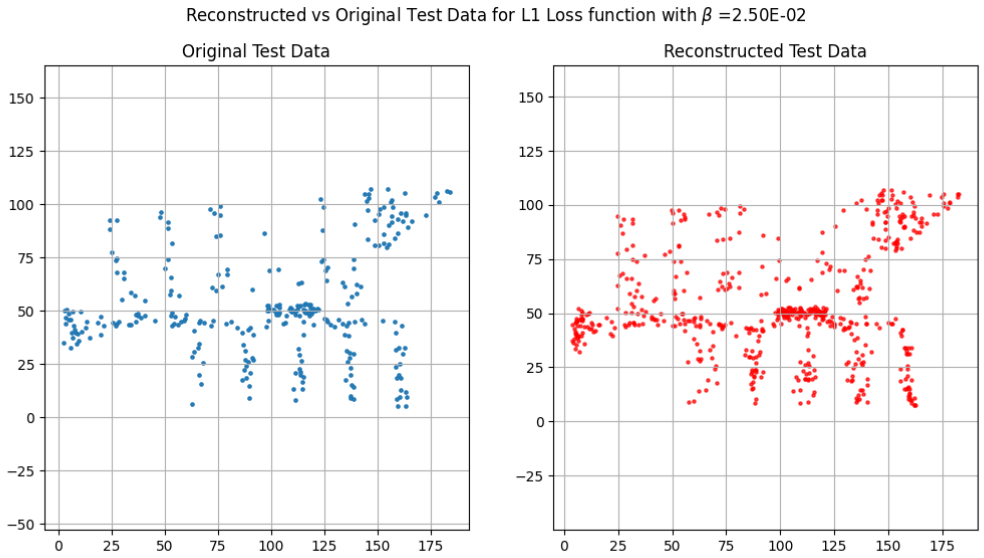
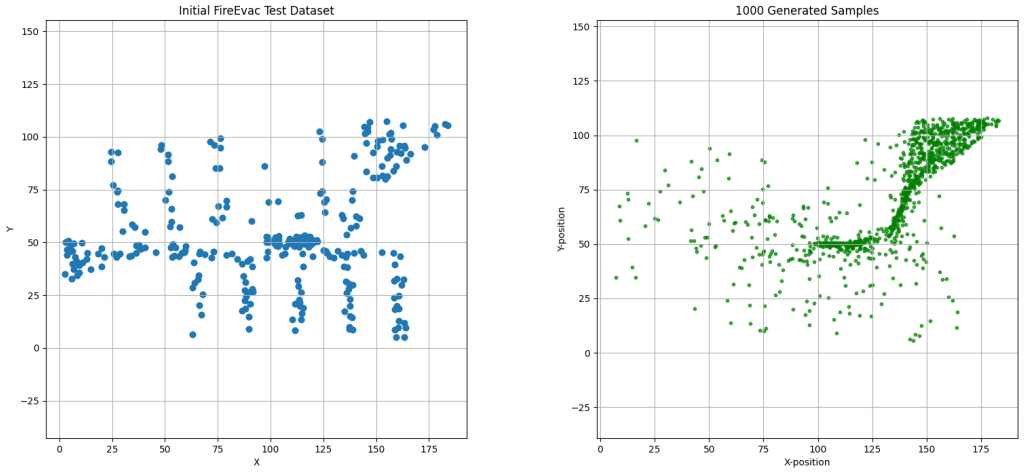


Figure 24: Comparison of original test data vs VAE reconstructed data.



(a) Original data.

(b) Scatter Plot having 1000 samples

Figure 25: Original data vs Scatter plot of 1000 samples.

4.2.6 Critical Number Estimation

We had to estimate the critical number of people. So we used our trained generator to sample from the learned distribution $p(x)$. We counted how many of these generated samples came within the sensitive area. We continued generating samples in batches of 100 until this count exceeded the critical threshold of 100 people.

Based on our model and configuration described before, the number of people in the sensitive area exceeded the threshold after $\approx$ **1200** samples being generated.

4.3 Answers to Questions from Page 1

The answers to questions posed on the first page of the exercise sheet are provided below:

(a) Time Estimate: Implementation of this task with necessary elements like adapting the VAE, training, and generating the plots, took approximately four full work days. An additional approx

two days were spent on fine-tuning hyperparameters (like different learning rate and β), model configs, debugging, and such, and writing the final report.

(b) Accuracy of Representation: The representation accuracy was evaluated as seen from the above plots.

- *Qualitatively*: Graphical similarity between the original, reconstructed, and generated data plots. Figures: 21, 24, 25 display that model has learned accurate representation of $p(x)$.
- *Quantitatively*: Accuracy metric is the converged ELBO loss on the test set, Figure 22d). The low final loss indicates a good model fit. In our case, the L1 loss gives a direct measure of the average geometric error between the true and reconstructed coordinates.

(c) What was Learned:

- **About the dataset:** The dataset in question represents a challenging, non-Gaussian 2D distribution with a complex shape. As it reflects our MI building's floor plan. It showcases the necessity of using flexible, non-parametric models like VAEs.
- **About method/model:** Doing this particular task provided several insights into VAEs. Firstly, it highlighted the importance of choosing a loss function that matches the data; In our case, L1 loss for continuous coordinates is far more appropriate than binary cross-entropy or MSE loss function.

  Second, it showcased the practical usage of generative models for simulation and risk assessment. This can be seen in the critical number estimation in Subsection 4.2.6. Lastly, it demonstrated that a properly configured VAE model can efficiently and effectively learn a useful representation of even a low-dimensional and complex dataset.

References

[1] Andrea Asperti and Matteo Trentin. Balancing reconstruction error and kullback-leibler divergence in variational autoencoders, 2020. URL: https://arxiv.org/abs/2002.07514, arXiv:2002.07514.

[2] Stefano Chiesa and Sergio Taraglio. Traffic request generation through a variational auto encoder approach. *Computers*, 11:71, 04 2022. doi:10.3390/computers11050071.

[3] Bin Dai, Ziyu Wang, and David Wipf. The usual suspects? reassessing blame for vae posterior collapse, 2019. URL: https://arxiv.org/abs/1912.10702, arXiv:1912.10702.

[4] Bin Dai and David Wipf. Diagnosing and enhancing vae models, 2019. URL: https://arxiv.org/abs/1903.05789, arXiv:1903.05789.

[5] Junxian He, Daniel Spokoyny, Graham Neubig, and Taylor Berg-Kirkpatrick. Lagging inference networks and posterior collapse in variational autoencoders, 2019. URL: https://arxiv.org/abs/1901.05534, arXiv:1901.05534.

[6] Diederik P. Kingma and Max Welling. An introduction to variational autoencoders. *Foundations and Trends® in Machine Learning*, 12(4):307–392, 2019. URL: http://dx.doi.org/10.1561/2200000056, doi:10.1561/2200000056.

[7] Boaz Nadler, Stephane Lafon, Ronald R. Coifman, and Ioannis G. Kevrekidis. Diffusion maps, spectral clustering and reaction coordinates of dynamical systems, 2005. URL: https://arxiv.org/abs/math/0503445, arXiv:math/0503445.

[8] Alexander Van de Kleut. Variational autoencoders (vae) with pytorch. https://avandekleut.github.io/vae/, 2020, May 14. Accessed: 2025-06-05.

[9] Jingyi Xu, Hieu Le, and Dimitris Samaras. Assessing sample quality via the latent space of generative models, 2024. URL: https://arxiv.org/abs/2407.15171, arXiv:2407.15171.