Report for exercise 3 from group I

Tasks addressed:	4
Authors:	Kejia Gao (03779844)
	Jingyi Zhang (03785924)
	Maximilian Mayr (03738789)
	Yizhi Liu (03779947)
	Felipe Antonio Diaz Laverde (03766289)
Last compiled:	2024–05–27

The work on tasks was divided in the following way:

Kejia Gao (03779844)	Task 1	20%
	Task 2	20%
	Task 3	20%
	Task 4	20%
Jingyi Zhang (03785924)	Task 1	20%
	Task 2	20%
	Task 3	20%
	Task 4	20%
Maximilian Mayr (03738789)	Task 1	20%
	Task 2	20%
	Task 3	20%
	Task 4	20%
Yizhi Liu (03779947)	Task 1	20%
	Task 2	20%
	Task 3	20%
	Task 4	20%
Felipe Antonio Diaz Laverde (03766289)	Task 1	20%
	Task 2	20%
	Task 3	20%
	Task 4	20%

1 TASK 1

Report on task TASK 1, Principal component analysis

Principal Component Analysis (PCA) is a powerful statistical technique used for dimensionality reduction, data visualization, and feature extraction. In the realm of data science and machine learning, PCA plays a pivotal role by transforming high-dimensional datasets into lower dimensions while preserving as much variance as possible. This reduction facilitates the simplification of complex datasets, making it easier to analyze patterns and correlations. Additionally, PCA helps in mitigating issues related to overfitting in machine learning models by reducing the number of features, thus enhancing the generalization performance.

1.1 Applying Principal Component Analysis to Two-Dimensional Data

In this section, we describe the process of applying principal component analysis (PCA) to the two-dimensional dataset "pca_dataset.txt". This data set contains 100 data points. The steps involved in this process include loading the data set, calculating the mean of the data, centering the data, and finally performing a singular value decomposition (SVD) to extract the principal components. The functions required to center the data and perform SVD are defined in an external utility script *utils.py.* By decomposing the centered data through SVD, three matrices U, S and V_t are obtained. Their shapes are (100, 2) (2,) (2, 2) respectively. The meaning of U, S and V_t is explained as follows

- U(100, 2): The matrix U contains the left singular vector of the centered data. Each column of U is a left singular vector corresponding to the projection of the original data in the two principal component directions. A row number of 100 means there are 100 samples in the data set. Column number 2 means that we retain 2 principal components.
- S(2,): S contains singular values. The length of S means that there are 2 singular values, corresponding to 2 principal components. Each singular value represents the variance (square root of the eigenvalue) of the original data in the direction of the corresponding principal component.
- V_t(2, 2): This matrix V_t is the transpose matrix of the orthogonal matrix V. Each row of V_t is a right singular vector, corresponding to the principal component direction in the data feature space. The number of rows and columns are both 2, indicating that there are 2 principal component directions.

Based on the results of PCA, we use visualization to generate a scatter plot of two-dimensional data points and visualize the principal component directions on this plot. Principal components are represented as vectors derived from the mean of the data.

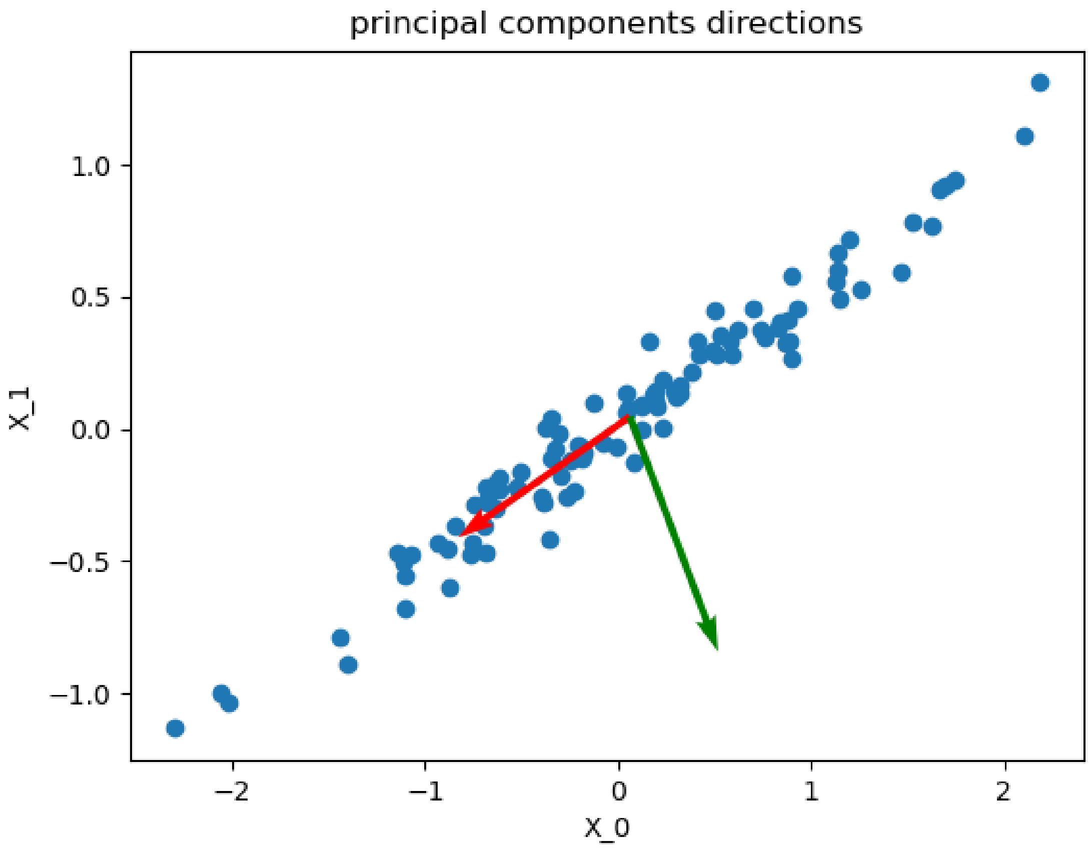


Figure 1: Visualization of 2 principal components

The red vector represents the direction of the first principal component, highlighting the direction of maximum variance in the data. The green vector represents the direction of the second principal component, which is orthogonal to the first component and represents the direction of the second highest variance in the data.
In order to compare how much information two principal components can represent, the *compute_energy* function needs to be used. The function *compute_energy* calculates the percentage of the total energy (explained variance) of a specific principal component based on the singular values of the data matrix. It takes an array S of singular values and an integer c (indicating which principal component to consider, starting from 1). The function checks whether c exceeds the length of S, prints a warning if it does, and then calculates the energy by squaring the singular value at position c, dividing by the sum of the squares of all singular values, and multiplying by 100 to get the percentage. By calling this function, we get that the energy of principal component 1 accounts for 99.3142656% of all energy. The remainder is the energy occupied by principal component 2.

1.2 Applying PCA to an image

In this task, we perform PCA on a picture of a raccoon.We use the function load_resize_image to load a grayscale sample image using scipy.misc.face, resize it to 185x249 dimensions and anti-alias it to preserve image quality, convert the resized image to a NumPy array, and return that array.

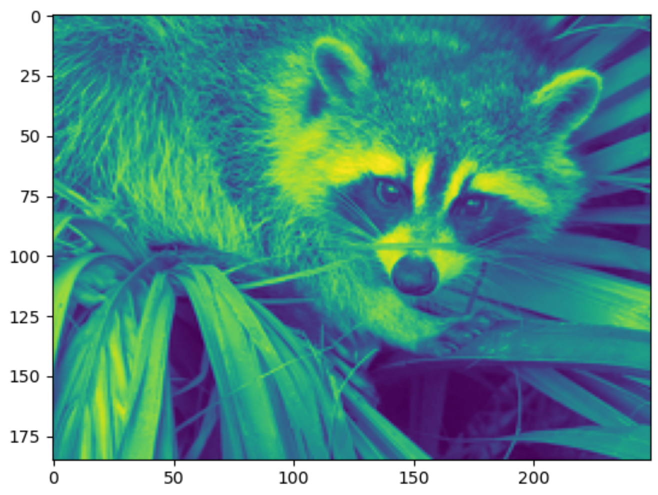


Figure 2: raccoon gray scaled

Then we reconstruct the image. This process uses truncated SVD to reduce dimensionality, filter out noise, compress data, and extract key features. By retaining only the most significant components of the original data, this approach simplifies analysis and improves the efficiency and performance of machine learning models.

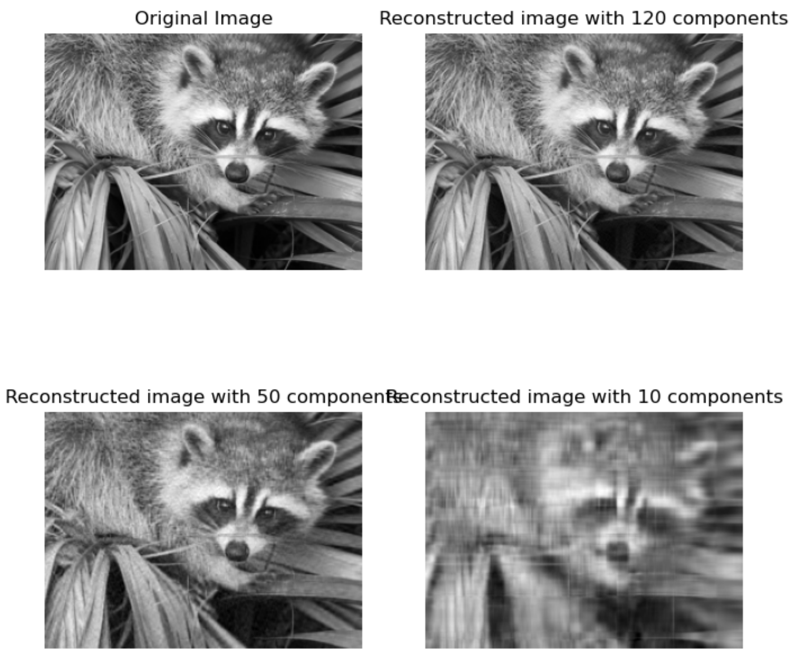

Original Image
Reconstructed image with 120 components
Reconstructed image with 50 components
Reconstructed image with 10 components

Figure 3: reconstructed image of raccoon

Fig.3 illustrates the process of image reconstruction using singular value decomposition (SVD). It compares the original grayscale image of the raccoon to reconstructed versions using 120, 50, and 10 principal components. The 120 components have been rebuilt to closely resemble the originals, retaining most of the details. With 50 components, the image loses some of the finer details, but is still recognizable. The reconstruction with 10 components is significantly blurrier and less detailed. Therefore, at number of 10 components is the information loss visible. By calling the *compute_num_components_capturing_threshold_energy* function, you can get the number of components required under the specified threshold energy. By visualizing the energy versus component curve, we can see that 26 components are needed to achieve 99% of the energy ratio.

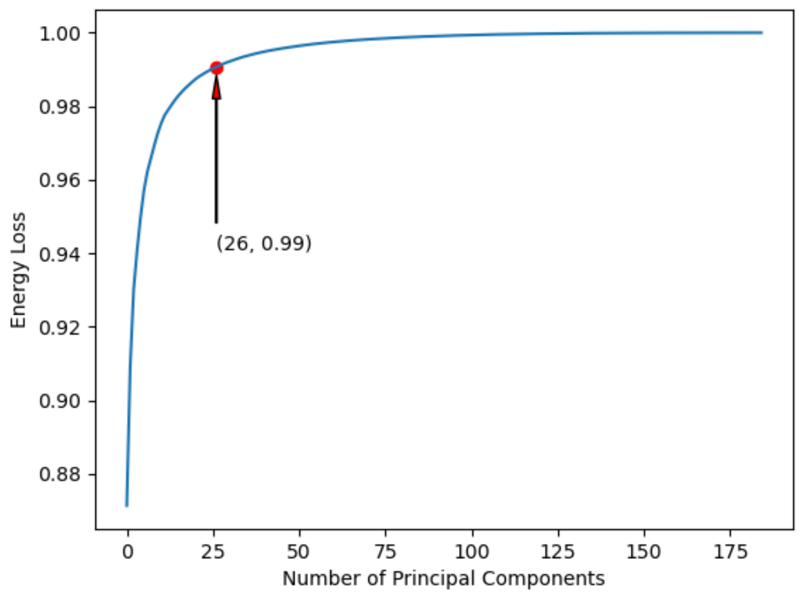
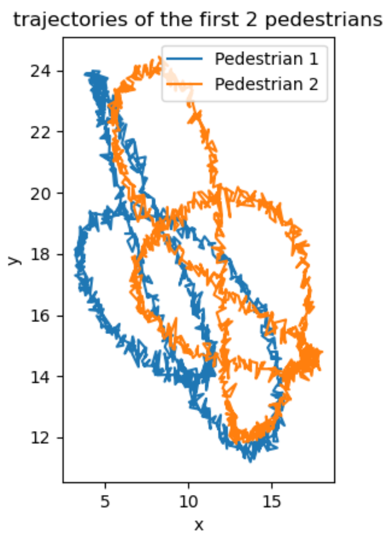


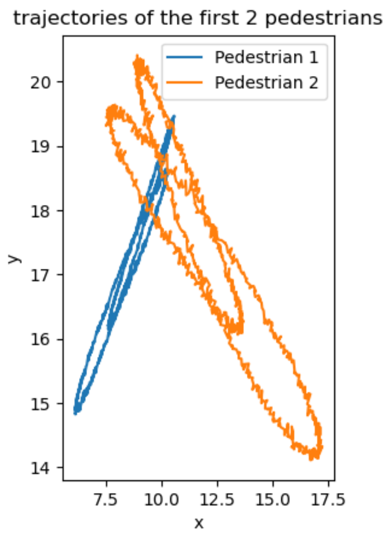
Figure 4: energy curve

1.3 Applying PCA to the trajectory of pedestrians

In this section, we will use PCA to analyze the trajectories of 15 pedestrians at 1000 time steps. First, we visualize the trajectories of the first two pedestrians. By directly calling the function *Visualize_traj_two_pedestrians* , the trajectories of two pedestrians can be drawn on a single graph. This feature is useful for visually comparing the motion patterns of two pedestrians within the same coordinate system.



(a) Original trajectory



(b) Reconstructed trajectory

Figure 5: Trajectory

Then by calling the *reconstruct_data_using_truncated_svd* function, only the first two components are used to reconstruct the data matrix. We use the reconstructed matrix to visualize the trajectories of the first two pedestrians again.

Through comparison, it can be seen that the reconstructed matrix still retains the general trajectory of the pedestrian, but because only the first two components are considered, too much information is omitted, so the correct trajectory is not obtained. And it is smoother than Fig.5 (a), which is also due to the lack of details. So keeping only two components is not enough to capture enough energy. This conclusion can also be drawn through the *compute_cumulative_energy* function. The energy obtained using the first two components is 84.925%, which is not enough to express sufficient information.

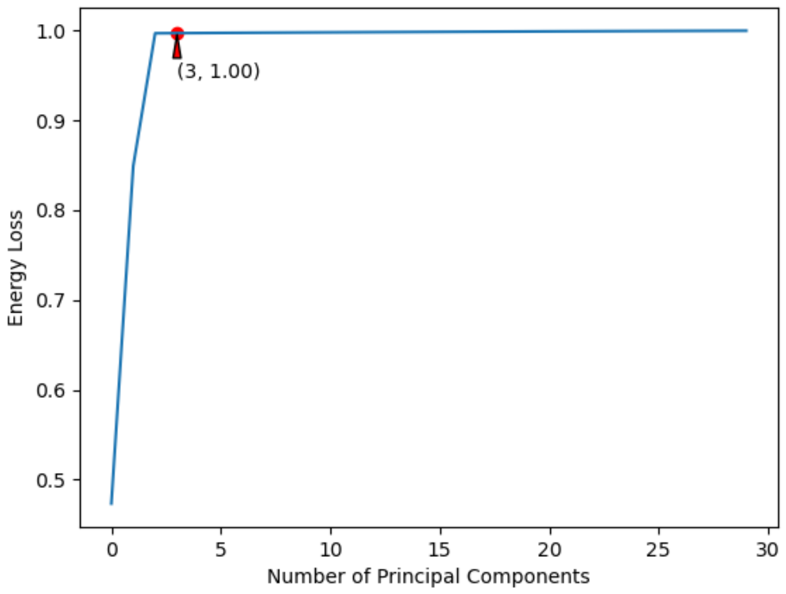


Figure 6: energy curve

By generating the energy curve, we can know that at least three components are needed to represent enough information, that is, the energy reaches 100%.

1.4 Answering three questions on exercise sheet

- I spent nearly two days compiling and testing the methods in the project.
- When using Principal Component Analysis (PCA) for dimensionality reduction, the accuracy of the data representation is measured by the proportion of the total variance in the data that is captured by the retained principal components. In this task, the accuracy is expressed by calculating the energy. The specific energy value has been stated in the previous task.
- From the PCA task, we learned that PCA is an effective technique for reducing dimensionality while maintaining significant variance, which facilitates data visualization and analysis. The process of reducing the dimensionality is also accompanied by the loss of data. High-energy components represent more information, so we should ensure that we also pay attention to changes in energy during the process of reducing the dimensionality. For example, when the energy is greater than 90%, it means that a large amount of information can be obtained. partial information.

2 TASK 2

Report on task TASK 2, Diffusion Maps

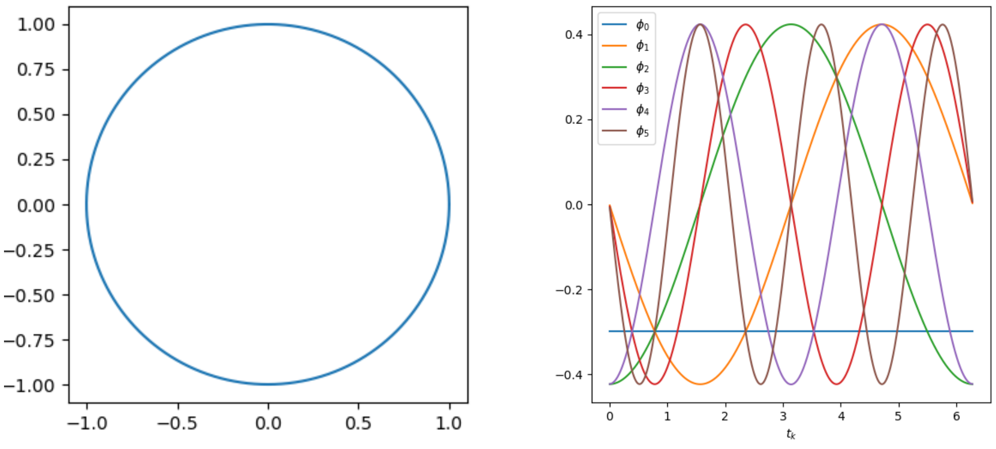
2.1 Implementation

The implementation was split in separate files. In *utils.py*, the computation of the Diffusion Map is implemented, returning the eigenvalues and eigenvectors of the Laplace-Beltrami operator for a given data set. In the remaining files (i.e. *1_fourier_analysis.py*, *2_swiss_roll.py*, and *3_trajectory_data.py*), different data sets were instantiated and passed to the utility functions to perform the diffusion map algorithm. Both, the data set and the eigenfunctions were plotted using the Python library `matplotlib`.

The implementation of the Diffusion Map algorithm follows the 10 steps listed on the exercise sheet. The calculations of the distance matrix, the value ϵ, and the kernel matrix (i.e. steps 1-3) happen in separate functions. These are called from the function `diffusion_map`, which uses these values to perform the remainder of the algorithm. In the end, the largest $n+1$ eigenvalues λ_0-λ_n (where n is an input parameter) and its corresponding eigenvectors are returned.

2.2 Comparison of Diffusion Maps and Fourier analysis on a periodic data set

In this part, a periodic data set is used. It contains $N = 1000$ data points which lie on the unit circle in equal distances to each other. The data set is portrayed in Figure 7a. Figure 7b shows the eigenfunction $\phi_0(t_k)$ corresponding to the largest eigenvalue and the largest 5 non-constant eigenfunctions $\phi_1(t_k)$-$\phi_5(t_k)$, where t_k is the angle in rad of the k-th point on the unit circle.



(a) Periodic data set (b) Eigenfunctions

Figure 7: The periodic data set and its eigenfunctions. The 6 eigenfunctions included (5 of which are non-constant) are associated to the 6 largest eigenvalues.

As can be seen, the eigenfunctions are scaled versions of sine and cosine functions. Thus, the result resembles a Fourier series since the periodic input signal is split into its trigonometric basis functions. The formula of a Fourier series is displayed below.

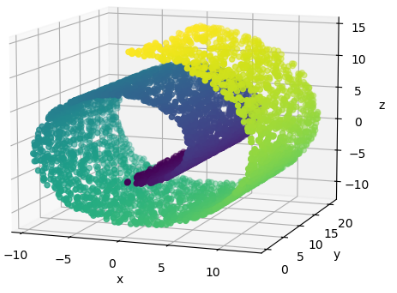
$$s(x) = a_0 + \sum_{k=1}^{\infty} a_k \cos(\frac{2\pi}{T}kx) + \sum_{k=1}^{\infty} b_k \sin(\frac{2\pi}{T}kx)$$

Using the eigenfunctions as basis functions, the Fourier series can be rewritten as the equation below.

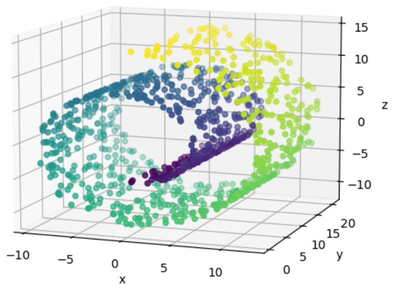
$$s(x) = a'_0\phi_0(x) + \sum_{k=1}^{\infty} a'_k\phi_{2k}(x) + \sum_{k=1}^{\infty} b'_k\phi_{2k-1}(x)$$

2.3 Applying Diffusion Maps to the swiss-roll data set

In this section the swiss-roll data set will be covered. The name origins from the pastries named the same because their shapes resemble. We generated the data by using the Python library `sklearn`. Three-dimensional images of the data set can be found in Figure 8. The version with 1000 samples will be referred to as *sparse swiss-roll*.



(a) 5000 samples



(b) 1000 samples

Figure 8: Swiss-roll data sets with 5000 and 1000 samples

Figure 9 depicts the eigenfunctions ϕ_0-ϕ_{10} plotted against ϕ_1. The plot of ϕ_1 against ϕ_1 is omitted since it would obviously be a linear function. The plots show that naturally ϕ_0 can be expressed as a function of ϕ_1 as it is constant. Furthermore, ϕ_2 and ϕ_3 are also functions of ϕ_1. As a result, no new information can be gained from either of these that cannot be gained from ϕ_1. Thus, 4 non-constant eigenfunctions are needed that ϕ_4 is no longer a function of ϕ_1 and provides new information.

Figure 10 shows the same plots generated by the software `datafold`. The code using the software can be found in the latter half of *2_swiss_roll.py* and is an adaption of the code used in one of datafold's tutorials [1] to perform a similar task on an S-curved manifold. The results are relatively similar. All graphs share the same shape. They only differ in the absolute values on both axes, and some graphs are flipped horizontally. This results from different values being used for the cutoff radius for distance computations and for ϵ resulting in our eigenvectors being scaled by a constant factor in comparison to datafold's eigenvectors. The horizontal flips result from some of our eigenvectors having a different sign than its `datafold` equivalent, which might result from different method used to compute eigenvectors. Apart from that, it can still be observed that ϕ_4 is the first eigenfunctions which is no function of ϕ_1.

Figure 11 shows the same plots for the sparse swiss-roll and Figure 12 those created with `datafold`. The plots show that ϕ_3 can no longer be represented as a function of ϕ_1 completely accurately. However, both figures indicate that ϕ_3 still is a function of ϕ_1 and the deviations are caused by computational inaccuracies. So, ϕ_4 is still the first eigenvector which is no function of ϕ_1. Moreover, the graphs lead to the conclusion that our solution is inferior to `datafold`, especially with lower numbers of samples.

When PCA is used on the swiss-roll data set, one can see that all three principal components are required to avoid suffering a major loss of information. This can be seen on the singular values of the centered swiss-roll data with 5000 data points. They are $\overline{\sigma_1} = 505.5$, $\overline{\sigma_2} = 453.9$, and $\overline{\sigma_3} = 427.7$. (Note that `sklearn` randomly samples data points. Thus, the actual data sets differ for each creation of a swiss-roll. Hence, the singular values are means of multiple calculations with different data sets.) As can be seen, the singular values are very close to each other. This results in a major loss of energy when omitting the third principal component to reduce the dimensionality. For the values above, a reduction to two dimensions would lead to a loss of over 28% of the total energy. Computations on a data set with 1000 samples yield similar results. The mean singular values are $\overline{\sigma_1} = 227.2$, $\overline{\sigma_2} = 202.4$, and $\overline{\sigma_3} = 191.5$. A dimensionality reduction by one dimension keeps 71.6% of the total energy (which is the exact same value rounded to 3 decimal digits as for 5000 samples).

The results show that PCA cannot reduce the dimensionality of the swiss-roll data set while preserving a sufficient information about the data. This is caused by the non-linearity of the swiss-roll data set since PCA aims to discover linear connections between several dimension which can than be subject to dimensionality reduction. However, two eigenfunctions are sufficient to map the data set to a two-dimensional space because the Diffusion Map algorithm correctly identifies the non-linear manifold the data points are sampled from. This manifold is the swiss-roll, which can be deduced from Figure 8. Although this manifold is not linear, "unrolling" the swiss-roll transforms the manifold to a two-dimensional plane.

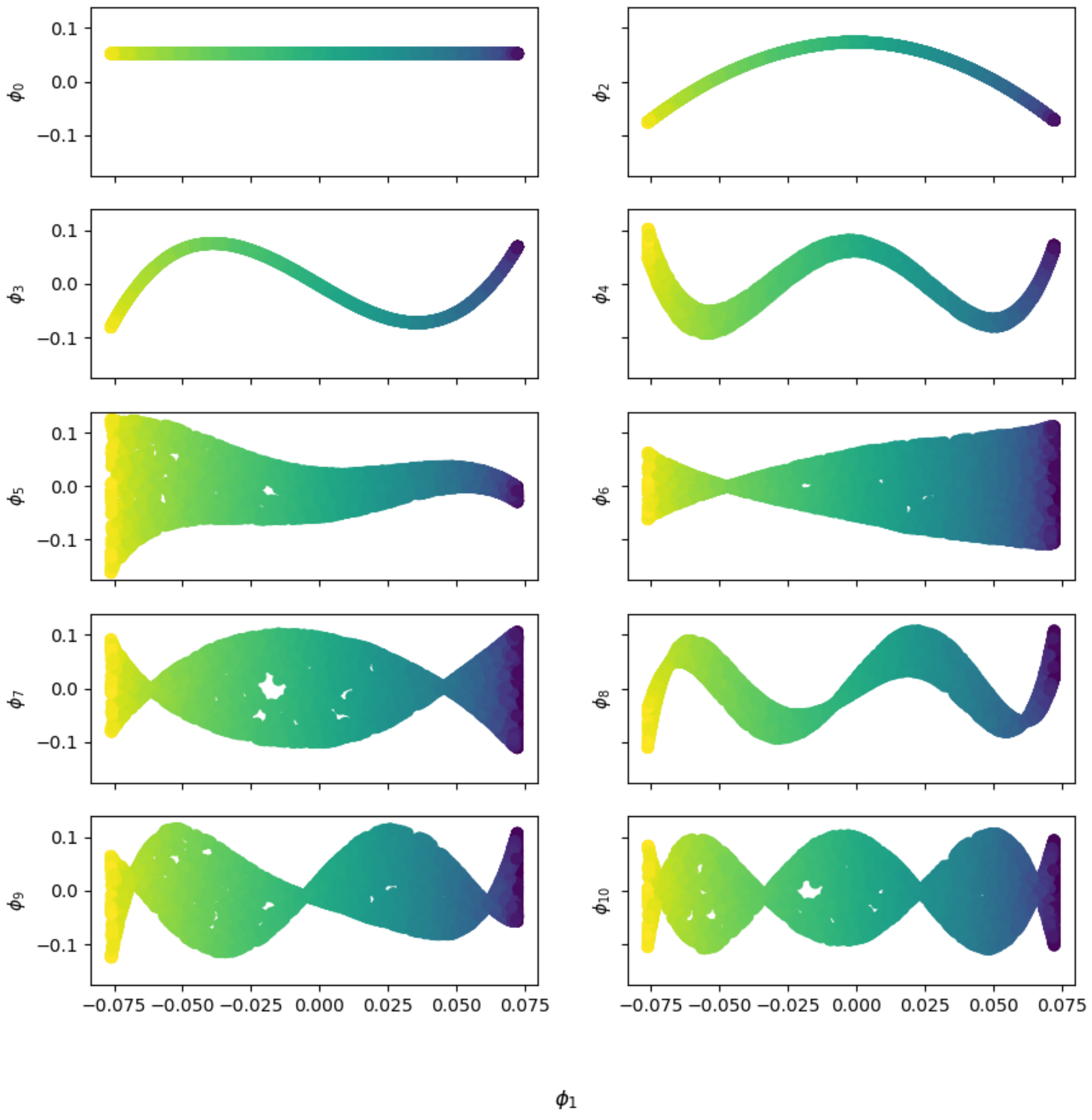


Figure 9: The eigenfunctions associated to the largest eigenvalues of the swiss-roll plotted against the first non-constant eigenfunction

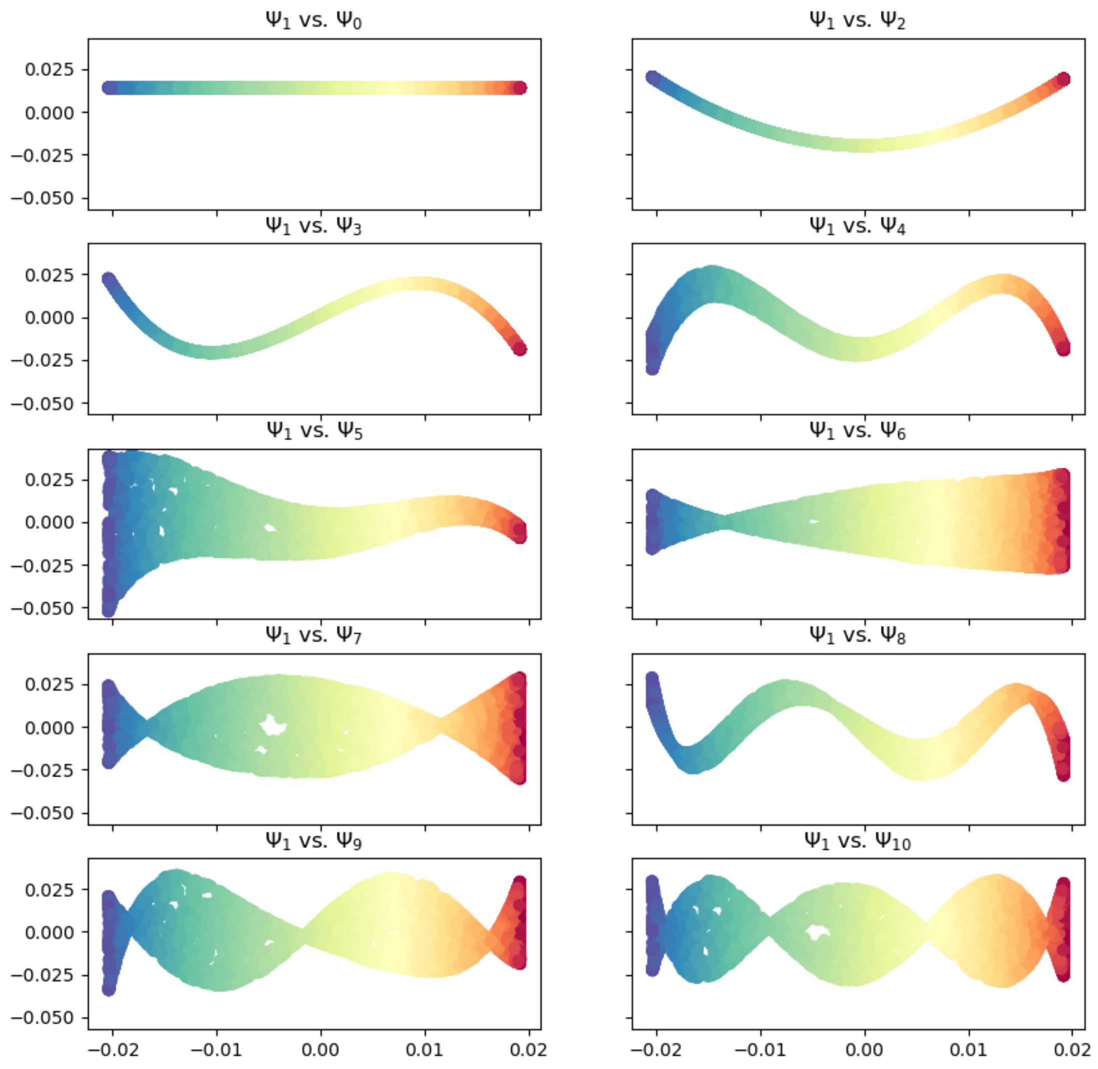


Figure 10: The eigenfunctions computed by datafold associated to the largest eigenvalues of the swiss-roll plotted against the first non-constant eigenfunction

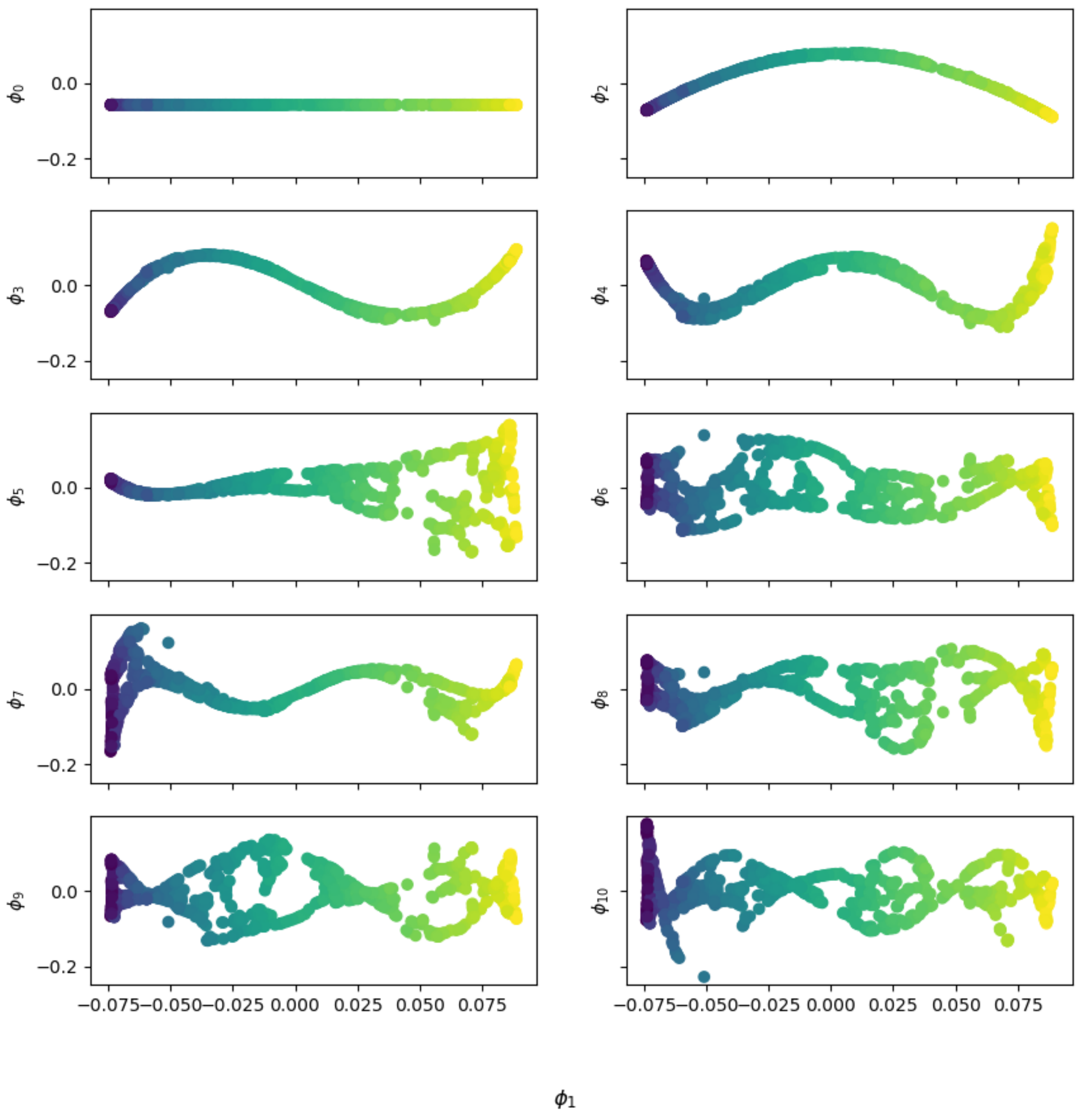


Figure 11: The eigenfunctions associated to the largest eigenvalues of the sparse swiss-roll plotted against the first non-constant eigenfunction

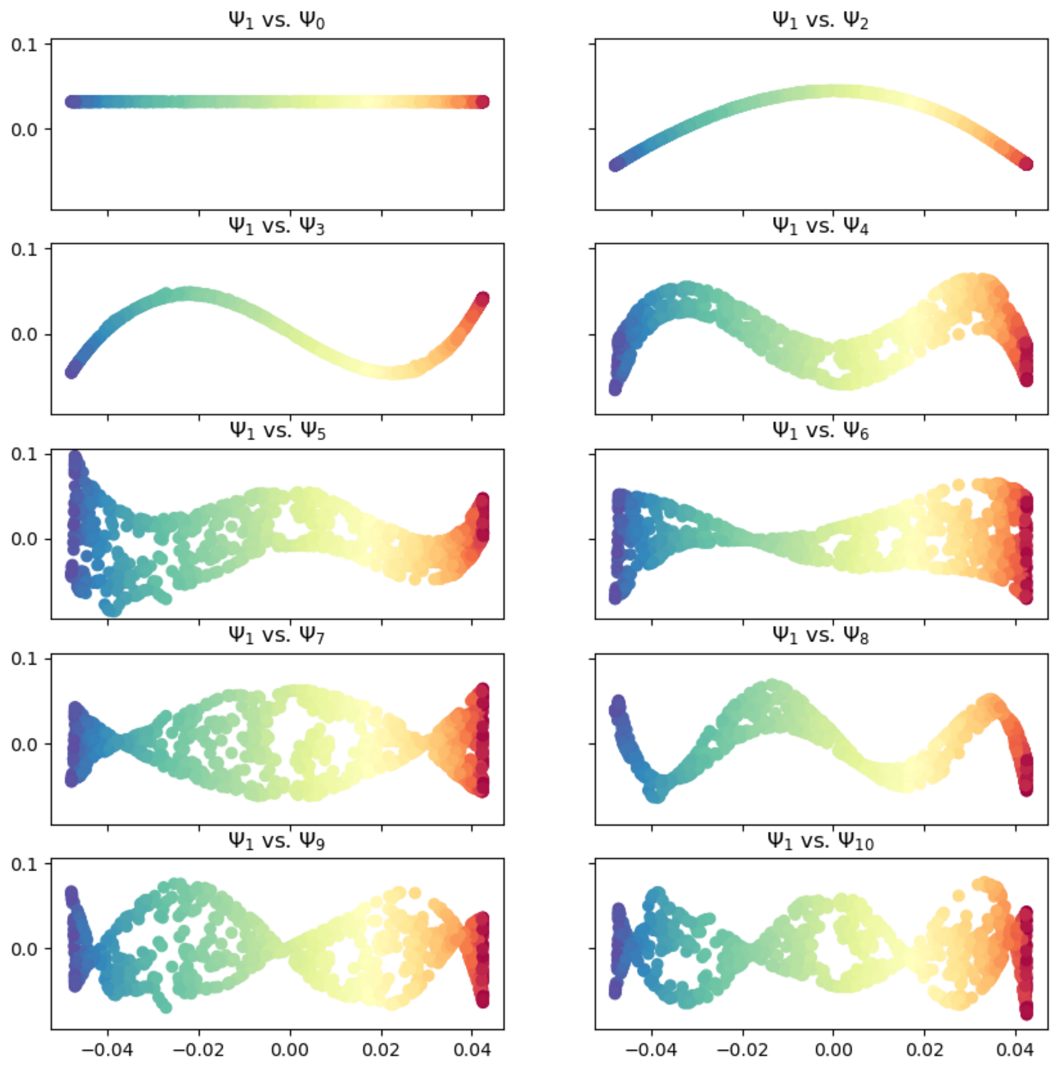


Figure 12: The eigenfunctions computed by datafold associated to the largest eigenvalues of the sparse swiss-roll plotted against the first non-constant eigenfunction

2.4 Applying Diffusion Maps to the trajectory of pedestrians

This section, again, covers the pedestrian data described and depicted in subsection 1.3. On the same way as in the previous section, the diffusion map algorithm was applied to the data and then, the eigenfunctions were plotted against each other. Figure 13 shows eigenfunctions plotted against ϕ_1. The top right graph illustrates that, like before, the two eigenfunctions ϕ_1 and ϕ_2 suffice to accurately represent the data because the graph does not intersect itself. All other eigenvectors represent features of the underlying manifold which do not fit for a dimensionality reduction to two-dimensions with ϕ_1. Within the first ten non-constant eigenfunctions no other fitting pairs could be found, although some were relatively close as Figure 14 suggests.

Like for the swiss-roll data set, here the data can be accurately represented with Diffusion Maps in two dimensions whereas PCA requires three. This leads to the conclusion that the 30-dimensional data set has some underlying non-linear manifold on which the points lie in a two-dimensional manner. On the other hand, the data set has three groups of linearly dependent dimensions which are linearly independent of each other.

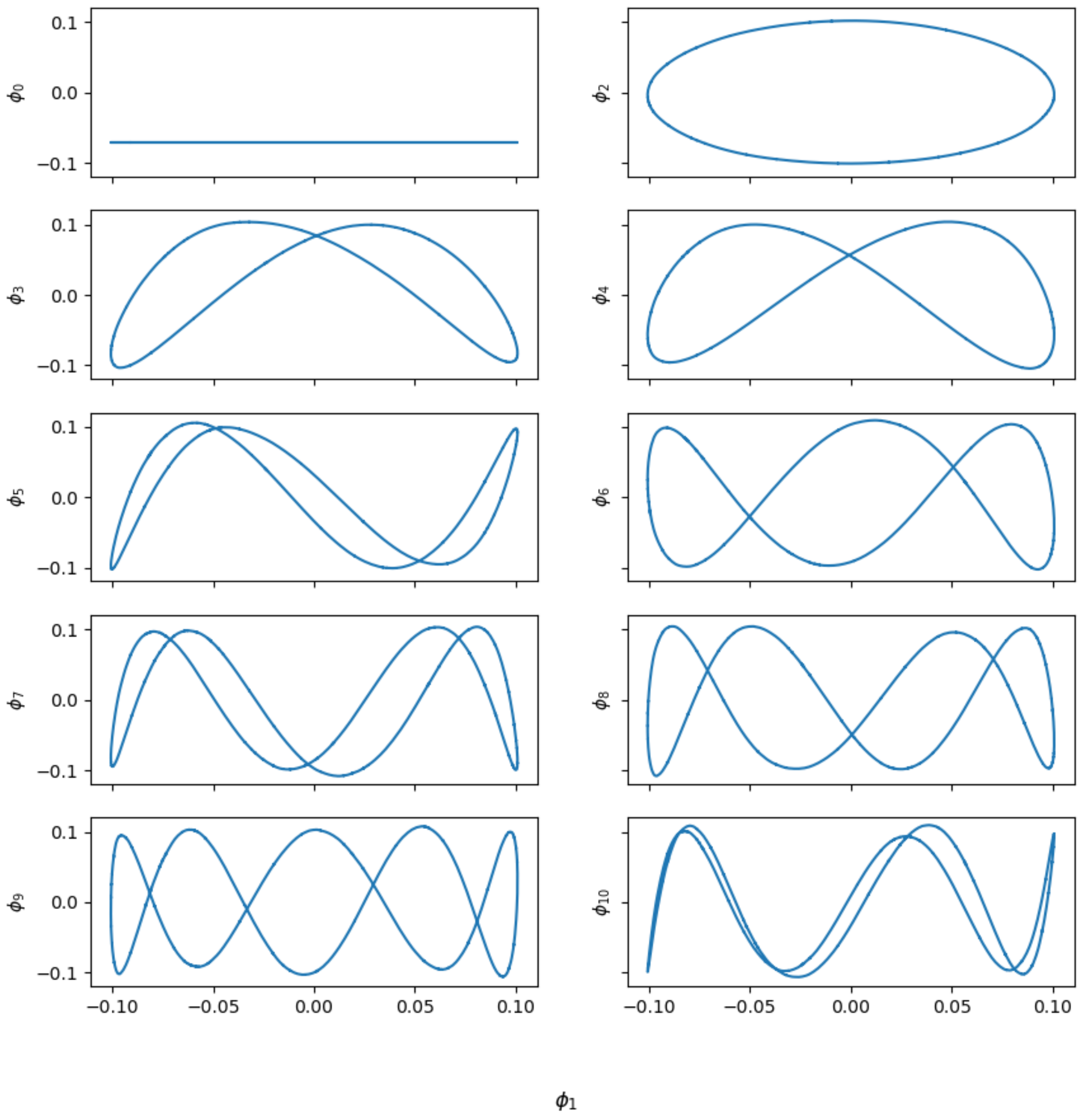


Figure 13: The eigenfunctions associated to the largest eigenvalues of the trajectory data plotted against the first non-constant eigenfunction

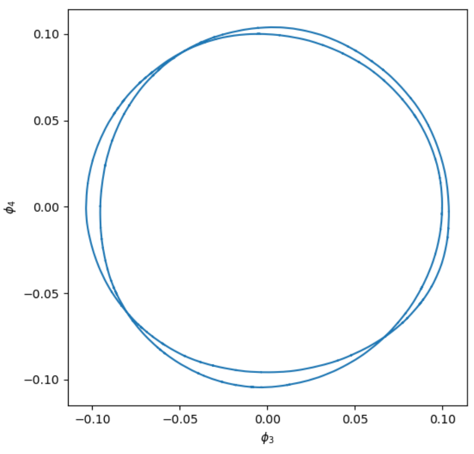


Figure 14: The eigenfunction ϕ_3 plotted against ϕ_4

2.5 Implementation time, accuracy, and learning outcome

Implementation time

The implementation and testing of the Diffusion Map algorithm defined on the exercise sheet in *utils.py* took approximately 3-5 hours. The creation of the remaining 3 scrips took about 10-15 hours, which were mostly spent on creating figures which present the results in the best possible fashion.

Accuracy

We could not find a common accuracy measurement for Diffusion Maps in the literature, unlike for instance PCA's energy. Assessments of the accuracy based on the visual inspection of the results can be found in the prior sections. In this section, we discuss the accuracy of our diffusion map algorithm by comparing the results to datafold's results, which we believe to be relatively accurate. On the periodic data set, the results only differ in the amplitude and signs of the trigonometric functions while the period lengths and order of eigenfunctions is equal. The swiss-roll data set has already been discussed in the respective section. We discovered that our implementation is less accurate than datafold's, especially with a reduced number of samples. We expect datafold's approach of dynamically calculating fitting values for the cutoff distance and ϵ to increase the accuracy over our approach of static hard-coded values. The comparison of our and datafold's results on the pedestrian data yields similar results as for the periodic data. The eigenfunctions only differ in absolute values and sign, while the relative values of the elements of a single eigenfunction compared to the other elements, and of eigenvectors compared to each other remain identical (except for sign changes).

Learning outcome

As mentioned in the previous sections, we got ascertained that every periodic signal can be split up in its basic frequencies. Furthermore, we learned that both, the swiss-roll and the pedestrian data set, have an underlying two-dimensional non-linear manifold. The manifold of the swiss-roll can be illustrated by a plane, which is rolled in. Unfortunately, there is no simple thought experiment to describe the manifold of the pedestrian data set.

Additionally, we learned that Diffusion Maps are able to identify such manifolds. Quite interesting was the experience that Diffusion Maps can be used like a Fourier analysis.

3 TASK 3

Report on task TASK 3, Training a Variational Autoencoder on MNIST

3.1 Answering two questions on exercise sheet

1. What activation functions should be used for the mean and standard deviation of the approximate posterior and the likelihood and why?

Mean and log-variance of the approximate posterior are the output of the encoder.

The mean represents the average of the latent variable distribution and can be any real value. Using a linear activation function (*self.fc_mu*) allows the mean to freely span the entire real number space without limiting the value range.

The log-variance can also take any real value because variance is always positive, and taking the logarithm of a positive number yields a real number $(-\infty, +\infty)$. Usually, standard deviation is a small positive number, which is hard for optimization, where the floating point arithmetic brings numerical instabilities. The log transformation maps the small numbers into wider space to work with. By using a linear activation function (*self.fc_logvar*), the model can output log-var in a wide range to represent all the possibilities of standard deviation.

Likelihood is the output of the decoder. A multivariate diagonal Gaussian distribution is used as likelihood, so it is not possible to get values binarised. In the exercise sheet, it is required to normalise the pixel values between 0 and 1 in the preprocessing. Likelihood should be also between 0 and 1, otherwise the ELBO loss function can't be calculated. Therefore, sigmoid, a continuous function with the range between 0 and 1 is used for likelihood.

2. What might be the reason if we obtain good reconstructed but bad generated digits?

Overfitting might be the reason. According to the description, the problem is that the model cannot generalize with unseen data because it "memorizes" too many features from the training set instead of learning. The complexity of VAE (e.g. the dimension of latent space) may be overly large, which causes the VAE model to encode unimportant details or noises. Moreover, lack of data augmentation and overtraining can also cause the overfitting problem.

3.2 Implementation

This task is based on three files: *model.py*, *utils.py* and *1_mnist.py.*

The file model.py involves a class called *VAE(nn.Module).* The VAE has an encoder, reparameterization and a decoder. Both encoder and decoder have 2 hidden layers with 256 units respectively and ReLU activation functions. At the end of encoder, following the last ReLU, two linear layers are used to compute the mean and log-variance, which is the input of reparameterization. Reparameterization is used to randomly sample latent vectors from a Gaussian distribution. The output of reparameterization will serve as the input of the decoder. With the sigmoid function at the end of the decoder, it is supposed to output reconstructed data to fit the normalized pixel values. These above-mentioned steps are implemented in a function named *forward(self, x: npt.NDArray[np.float64]).*

The file *utils.py* stores the utility functions for the model training. Evidence Lower Bound (ELBO) loss is the objective function, which is the addition of reconstruction loss (binary cross entropy) and Kullback-Leibler (KL) loss. In the function *training_loop()*, the model calls the functions *train_epoch* and *evaluate* to conduct training and testing in each epoch. *train_epoch* has access to training set and contains the general processes of initialization, forward pass, calculation of loss, backward pass and optimization step, with *train_loss* as return. *evaluate* computes the loss of the model on test set and returns *test_loss. train_loss* and *test_loss* in every epoch will be stored into two lists, in order to plot the loss curve after training the model. *latent_representation* is used to extract the mean and log-var output by the encoder and visualize the 2 latent dimensions of the mean with its label. *reconstruct_digits* visualizes the reconstructed data from the output of the forward pass, compared with the original digits. *generate_digits* samples vectors in the latent space randomly and they are input into the decoder to generate digits, which will be plotted.

The file *1_mnist.py* sets the basic parameters for the models, trains the models, and achieves outputs. Two models are set up according to the exercise sheet:

learning rate = 0.001,

batch size = 128,

epochs = 50,

input dimension = $28 \times 28 = 784$ (number of pixels of a digit figure in MNIST dataset),

number of neurons in the hidden layer = 256

The only difference of the 2 models lies in the latent dimension. The latent dimension of the first model is 2 and that of the second model is 32, since it is required to compare the effects of different latent dimensions, which is mainly discussed in the following pages.

3.3 Results of VAE with 2-dimensional latent space

Figure 15, Figure 16, Figure 17 and Figure 18 demonstrate the results of the 1st, 5th, 25th and 50th epoch.

Figure 15a, Figure 16a, Figure 17a and Figure 18a are latent representations of these 4 epochs. As the training goes further, the samples with the same label tend to converge gradually. A few outliers exist without getting close to the zones to which their labels should belong. The samples of label 0 and 6 are close to each other, which is consistent with human senses. In the same way, 2, 3, 8, 4, 9 and 1, 7 are also considered as groups of digits with similar shapes. The samples with visually similar digits tend to adjacent to be adjacent to each other in the latent space.

Figure 15b, Figure 16b, Figure 17b and Figure 18b visualize the comparisons of original and reconstructed digits of the 4 epochs. As the training goes on, the number of reconstructed digits mismatched with original ones decreases from 11 (the 1st epoch) to 5 (the 5th epoch). After that, the number is getting stable, as the number of mismatching digits is 6 and 5 in the 25th and 50th respectively. The reconstructed digits of the 1st epoch is blurry, while those of the 5th, 25th and 50th epoch are clearer. The further training after the 5th epoch contributes little to the improvement of reconstructed data.

Figure 15c, Figure 16c, Figure 17c and Figure 18c show the generated digits based on sampled vectors in latent space. As the training continues, the digits have higher contrast, i.e they look more explicit compared with the background. The number of ambiguous or abnormal digits decreases from 8 (the 1st epoch) over 6 (the 5th epoch) to 4 (the 25th and 50th epoch), where "ambiguous" means a digit can be recognized as 2 or more digits by human due to the uncertainty, while "abnormal" means a digit can't be recognized as a digit by human at all.

Figure 19 illustrates the loss curve of the VAE with latent dimension of 2. Both train loss and test loss decline with the epoch, with slight fluctuations during the training. Train loss is slightly less than test loss, indicating there is no overfitting problem.

3.4 Results of VAE with 32-dimensional latent space

Figure 20, Figure 21, Figure 22 and Figure 23 depict the generated digits of a VAE model with latent dimension of 32 in the 1st, 5th, 25th and 50th epoch. Similar to subsection 3.3, the digits become visually and semantically more explicit as the training proceeds. However, in comparison with the generated digits of the same epoch in subsection 3.3, the digits generated by VAE with 32-dimensional latent space are more tortured. The numbers of ambiguous and abnormal digits in the 1st, 5th, 25th and 50th epoch are 11, 10, 10 and 8 respectively, much higher than those generated when the latent dimension is 2. Some of the digits look like a combination of 2 digits (0 and 8, 1 and 9, 3 and 8, 7 and 9, 6 and 8, 4 and 9, etc.). The increase of latent dimension enhances the model complexity but weakens the ability of generalization, which is called overfitting. The reason is that oversized latent dimension makes the VAE model fail to extract valuable features efficiently and memorize redundant information. Practically, 32-dimensional latent space is overly large for small figures in the MNIST dataset.

Figure 24 illustrates the loss curve of the VAE model with 32-dimensional latent space. The train loss and test loss decrease synchronously and stably, while the test loss is tinily lower than train loss. Both of them are much lower than those in Figure 19. Therefore, this model has a good performance in training and validation, but can't generalize well with the unseen test data.

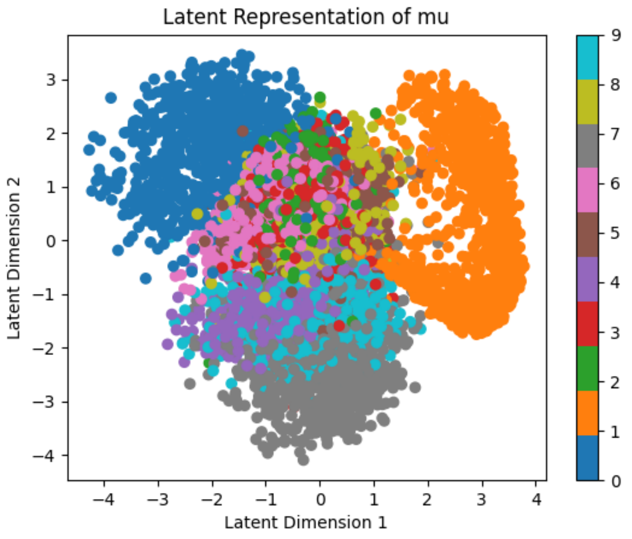
3.5 Answering three questions on Page 1 of exercise sheet

1. An estimate on how long it took you to implement and test the method.

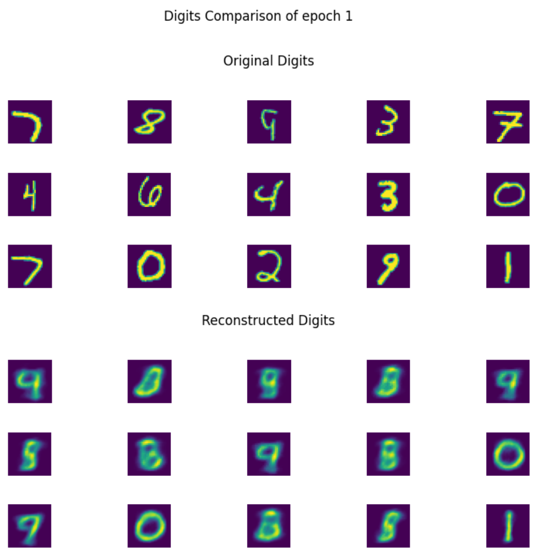
It will take me 9 hours to implement the method and 3 hours for the test. The implementation and test proceed synchronously. For example, I commit the implementation of two functions and inspect the error message from Artemis.

2. How accurate you could represent the data and what measure of accuracy you used.

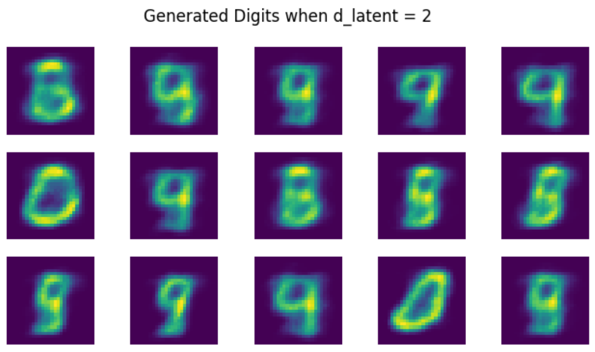
ELBO loss is used to describe the precision of model fitting. It can be used in training but the second model with lower ELBO loss has a worse generalization. I personally assess the model performance by counting the number of ambiguous and abnormal digits among the 15 generated digits. The reason why I focus on generated



(a) Latent Representation

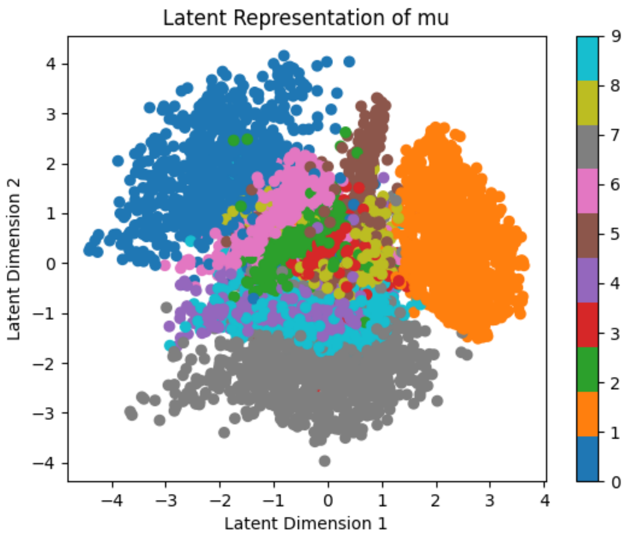


(b) Original and reconstructed digits

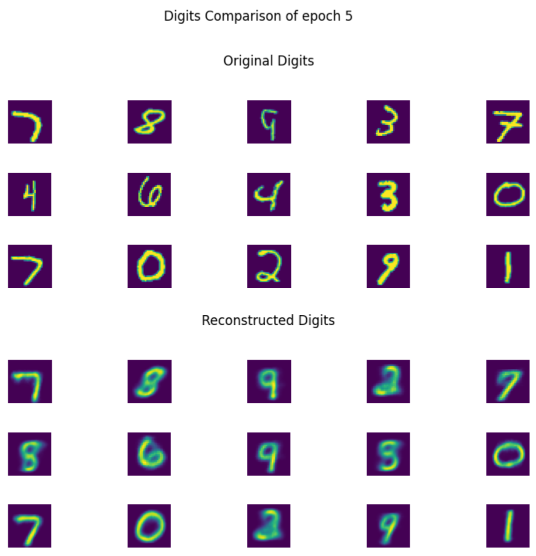


(c) Generated Digits

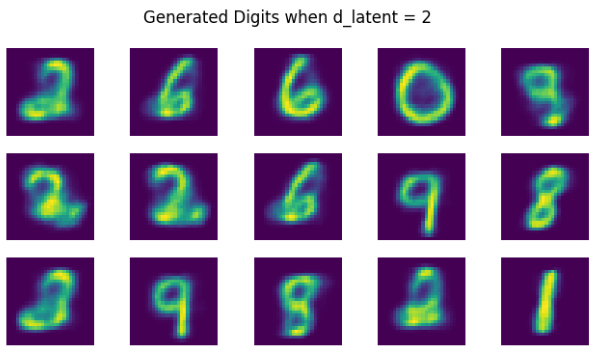
Figure 15: Plots of digits for the 1st epoch, when the latent space's dimension of VAE is 2



(a) Latent Representation

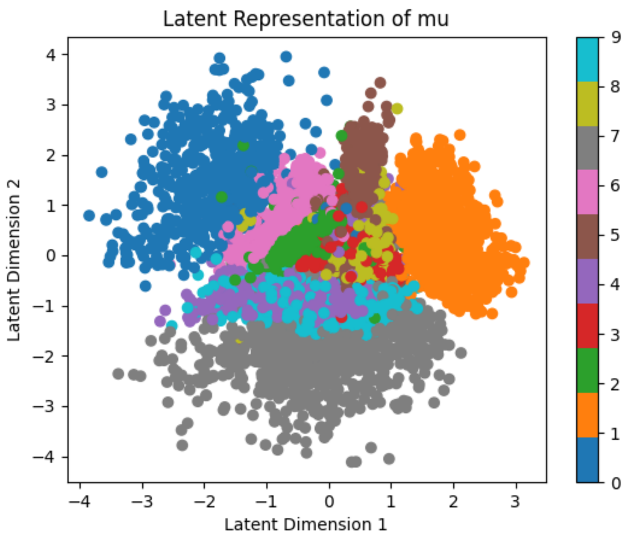


(b) Original and reconstructed digits

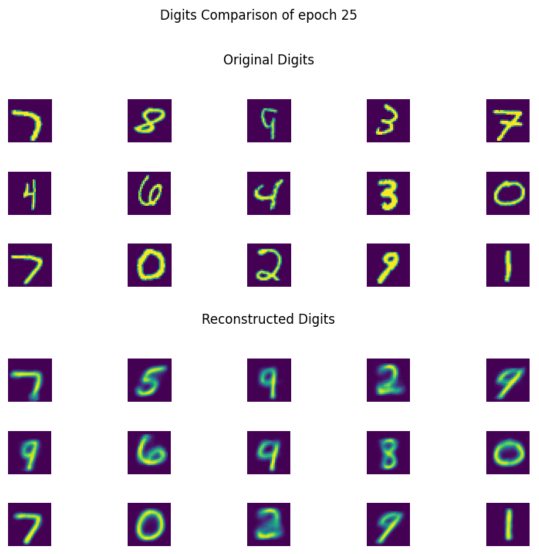


(c) Generated Digits

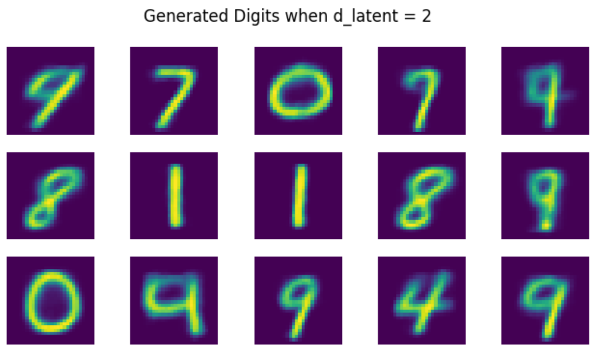
Figure 16: Plots of digits for the 5th epoch, when the latent space's dimension of VAE is 2



(a) Latent Representation

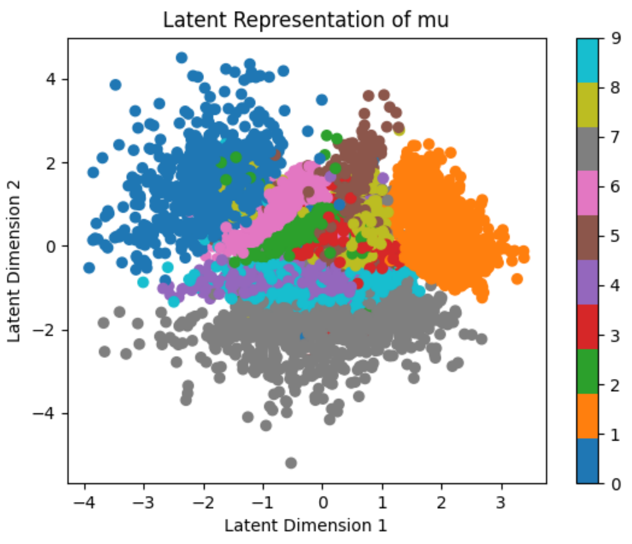


(b) Original and reconstructed digits

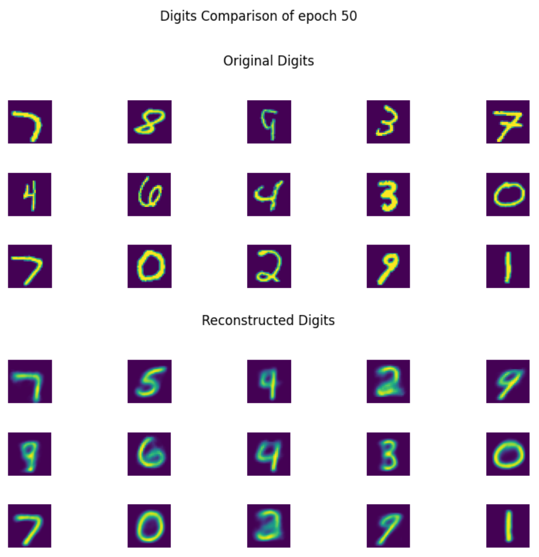


(c) Generated Digits

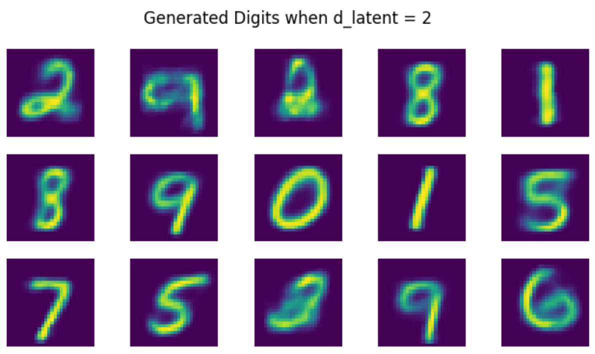
Figure 17: Plots of digits for the 25th epoch, when the latent space's dimension of VAE is 2



(a) Latent Representation



(b) Original and reconstructed digits



(c) Generated Digits

Figure 18: Plots of digits for the 50th epoch, when the latent space's dimension of VAE is 2

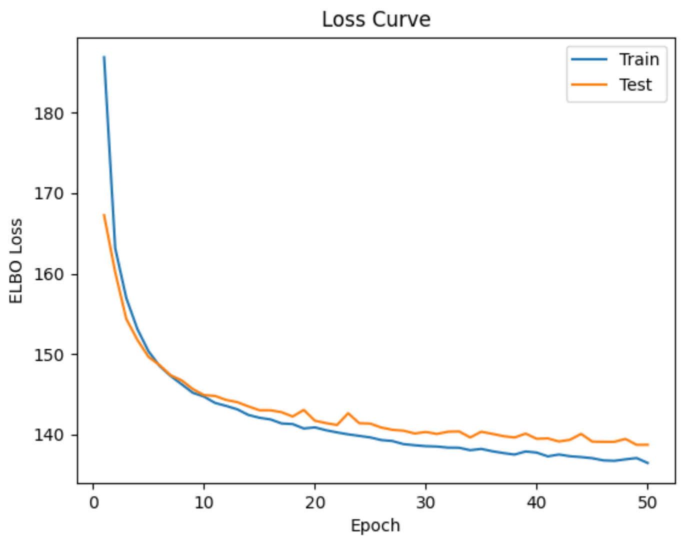


Figure 19: Plot of loss curve for 50 epochs, when the latent space's dimension of VAE is 2

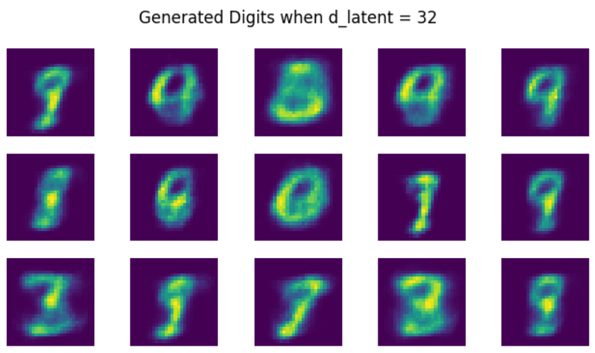


Figure 20: Generated Digits of the 1st epoch

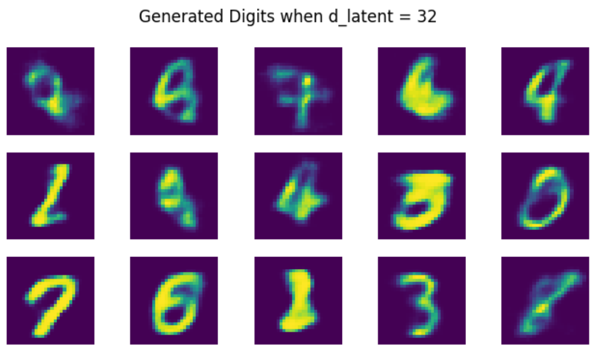


Figure 21: Generated Digits of the 5th epoch

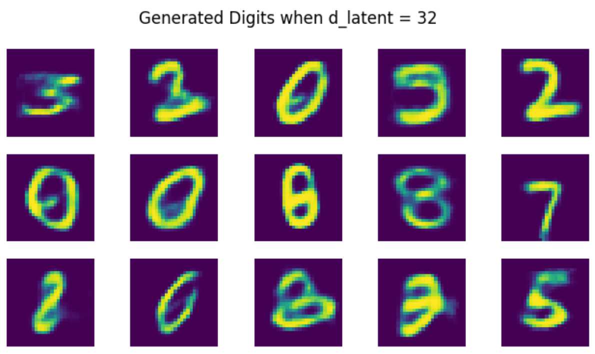


Figure 22: Generated Digits of the 25th epoch

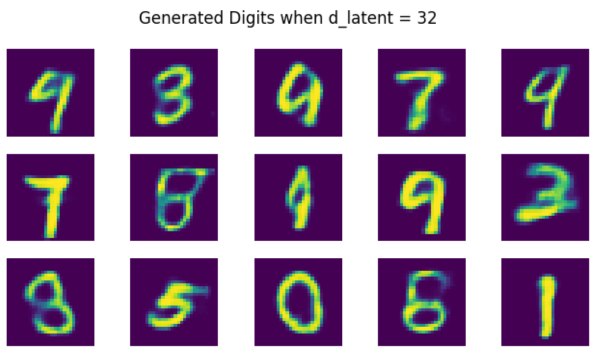


Figure 23: Generated Digits of the 50th epoch

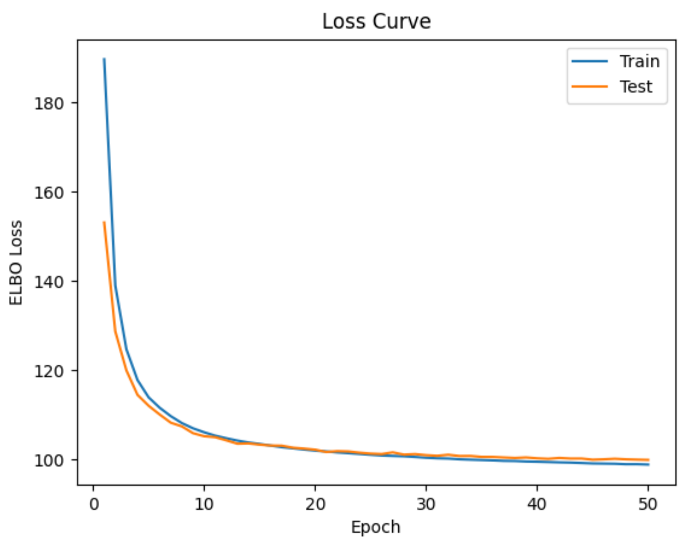


Figure 24: Plot of loss curve for 50 epochs, when the latent space's dimension of VAE is 32

digits is that they are generated from sampled vectors of latent space which is considered as unseen test data and can be used to evaluate the generalization ability of the model. The proportion of ambiguous and abnormal digits reflects the extent to which what the model decodes is different from the human sense. The smaller proportion represents higher consistency between model generating and human understanding.

3. What you learned about both the dataset and the method (which is probably different from what the machine learned).

The MNIST dataset is a relatively small dataset with 60000 figures of handwritten digits (from 0 to 9), each of which has a size of 28×28 pixels. Therefore, a small dimension of latent space is sufficient to encode the data. Data augmentation (rotation, translation, scaling, adding noise, adjusting contrast, etc.) can be used in this case to improve the robustness against the random noises and individual differences.

Variational Autoencoder (VAE) is a type of generative model that combines principles from variational inference and deep learning. It is comprised of two neural networks: an encoder that maps input data to a lower-dimensional latent space and a decoder that reconstructs the data based on this latent representation, which enables VAE to generate new data points by sampling from the learned latent space distribution. During the implementation, it is significant to choose the activation function for the encoder and decoder. The value ranges of specific activation functions should be consistent with those of the represented statistical variables (mean, log-var and likelihood). In addition, if a variable is within a small range (e.g standard deviation), we can use logarithm transformation (log-var) to expand the range to a larger space, which makes the optimization more efficient.

4 TASK 4

Report on task TASK 4, Fire Evacuation Planning for the MI Building

In this task, the additional files `utils2.py` and `model2.py` were included. This is because there are no tests for task 4, which might require having default files `utils.py` and `model.py`. Attempting to create a modular code that works for both implementations (task 3 & 4) proved to make the tests fail, although the code was intended to work for both data types (datasets: images and coordinates). This can be seen in commit `bacf736b974027aec075898aa0b139eff9d5a32c`. Therefore, the easiest solution was to simply split the codes and behavior of both applications into separate files.

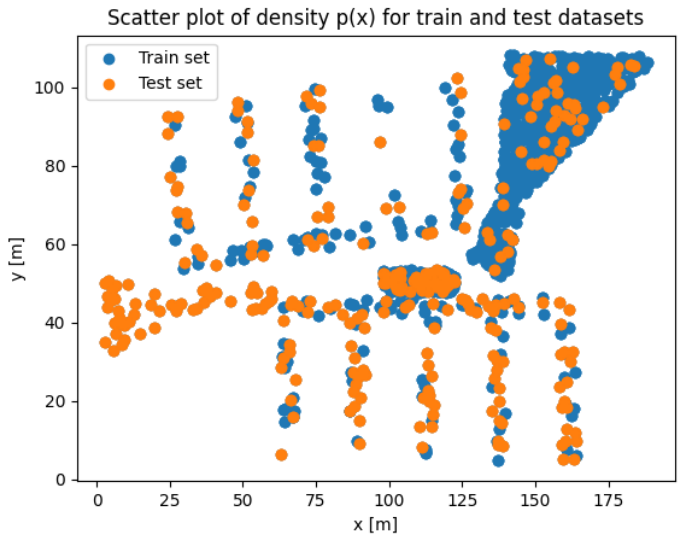


Figure 25: Plot of original training and testing data for fire evacuation task.

For the last task, we want to reconstruct and learn the distribution of how positions are distributed inside the MI building from TUM (see Figure 25). For this, we use again the VAE model, making slight changes to it with respect to its original implementation is Task 3, to adapt it to this particular task. In particular, the necessary changes are: as the input data is rescaled to the range $[-1,1]^2$, as suggested in the Exercise (as rescaling the data, such that all different input features have a zero mean and the same range helps making the optimization landscape have a more suitable shape for convergence), consequently, as inputs and therefore targets have to be in said range, using a `Sigmoid` activation layer for the output layer does not work anymore, as it maps inputs into probabilities (the range $[0,1]^2$). Then the activation layer has to be changed into a `Tanh` activation layer that does have the desired output range.

Furthermore, as we are no longer using probabilities, `BCE` (binary cross entropy) for the loss function cannot be used anymore, as it requires for targets and inputs to be probabilities. In our case then, we opted to use a `MSE` (mean square error) loss function, as it is more suitable for these tasks. Furthermore, we noticed that most likely due to the `MSE` loss being not too big as it increases with the square of distances bounded inside an square of area 4 ($[\text{a.u.}]^2$) making distances to be quite small, the KL loss, which increases with the square of the mean and standard deviation of each component in the latent space, forces the model to map all outputs into a single output, most likely due to the inputs collapsing into a single point in the latent space or the encoder overfitting the data and simply trying to learn the identity function as the latent and input dimensions are the same, removing all generalization power for the decoder to be able to retrieve back the original data. This can be observed in the big generalization gap between the training and the testing error, and both errors plateauing almost right away and simply oscillating due to the stochastic nature of batch gradient descent, which indicates the model already learned the best hypothesis possible, which tends to be an indicative of overfitting when studying a model's learning curves. Said learning curves can be observed in Figure 26.

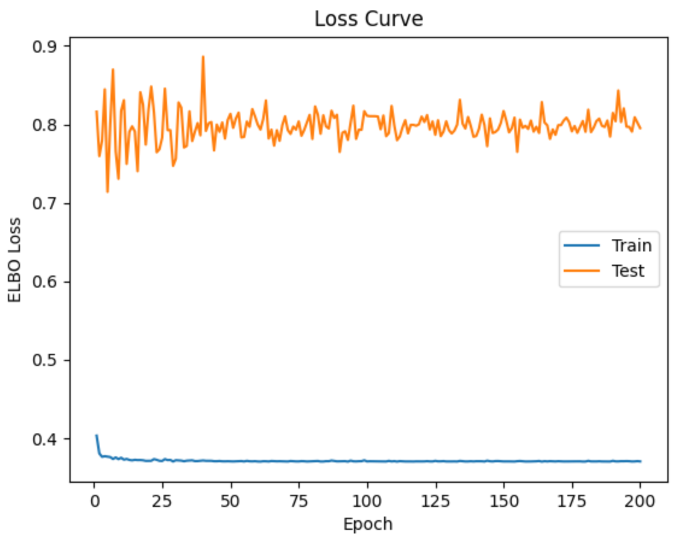


Figure 26: Plot of loss curve for 200 epochs for fire evacuation dataset using "ELBO" (with MSE) loss.

In this task, and similar to the previous task, and common in data compression tasks which auto-encoders are generally used for, thanks to the reduced dimensionality of the latent space, we can visualize it to try to study and diagnose this. Figure 27 shows the plot obtained from the latent space.

Here we can indeed observe how, the two components of the mean from the z latent space distribution have a high correlation, indicating that indeed we might be experiencing a loss of information during training. We also tried visualizing the different components of the sample data z as a function of the two components of the input training data x, to try to see if there was any correlation between both variables, but neither of the four plots seemed to show any particular behaviour, most of the points in the plot $z_i(x_j)$ for $i,j=1,2$ seemed to show a constant behavior. In this case, both the reconstructed and the generated data showed very poor results.

Therefore, and as explained before, we opted to replace the ELBO loss for a MSE loss, furthermore, and again as suggested, we used the following hyperparameters for training, as some preliminary random searches for better hyperparameters showed worse performing results with respect to the reconstruction data:

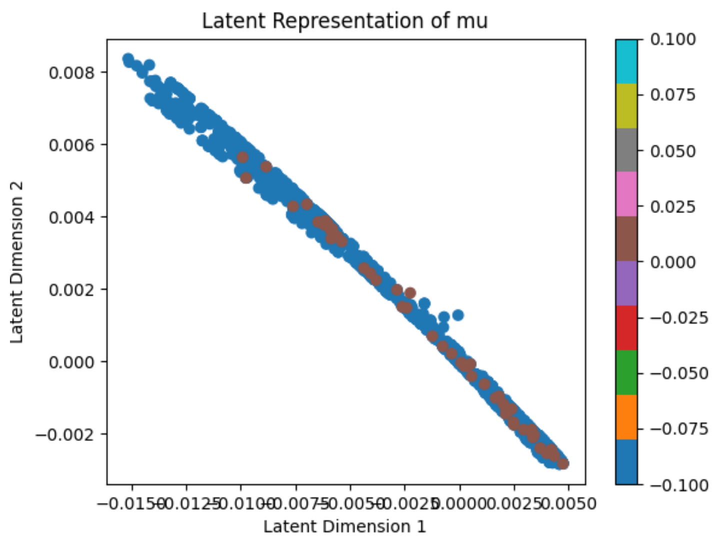


Figure 27: Plot of latent space after 200 epochs for fire evacuation dataset using "ELBO" (with MSE) loss.

$$
\begin{aligned}
&\text{learning rate} = 0.005,\\
&\text{batch size} = 64,\\
&\text{epochs} = 200,\\
&\text{input dimension} = 2,\\
&\text{number of neurons in the hidden layer} = 64,\\
&\text{number of hidden layers} = 2, \text{(for both encoder and decoder).}
\end{aligned}
$$

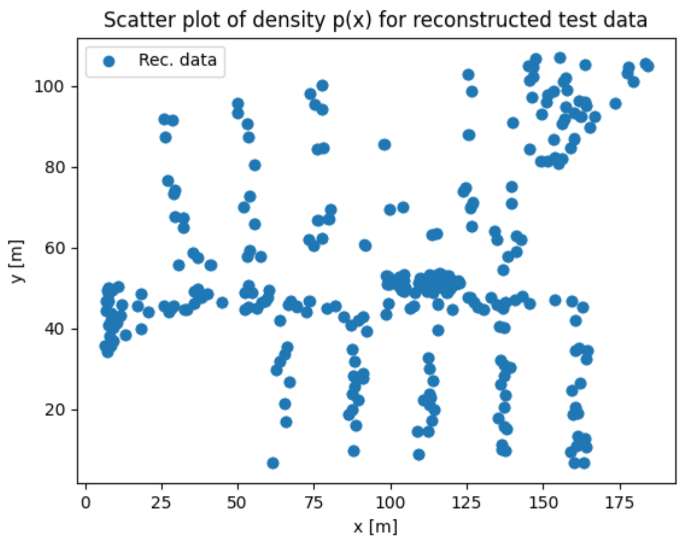


Figure 28: Plot of reconstructed test data after 200 epochs.

Figure 28 shows the reconstructed data for the training set after training for 200 epochs. We can see how the reconstructed datapoints not only show an structure similar as the desired one (from the MI building as can be observed in Figure 25), but upon closer inspection, they are very similar from the original data. At the end of the training loop, the reported (average MSE) loss for the test set is of approximately $\bar{e}_{MSE} \approx 5.085 \times 10^{-4}$, which means that on average, for the test set, the reconstructed data is a radius of $\Delta r \approx 4.17$m away from the original datapoint (this computation is assuming the distance vector between both points is, for simplicity, in the higher range component in the data, where the error is the biggest as a distance then transforms as $\Delta r = \Delta\tilde{r}(x_{max} - x_{min})$, with $\tilde{r}$ the distance in the rescaled space, meaning the error distance is actually upper bounded by this radius Δr), which given the dimensions of the MI building seem to be very close to the original datapoint.

Now we plot the generated data in Figure 29, which shows also the training dataset for scale and to be able to more or less locate the generated data due to the fact that it does not seem to resemble the original data, so it has no structure. We tried to look into the reason for such results in the generated data, but unfortunately it was not possible to debug it and generate real looking data out of randomly sampled noise in the latent space. It is worth nothing that on consecutive runs of the data generation algorithm, the results obtained vary slightly in that the quadrilateral parametrizing the generated data was shifted, but kept its structure and relative size.

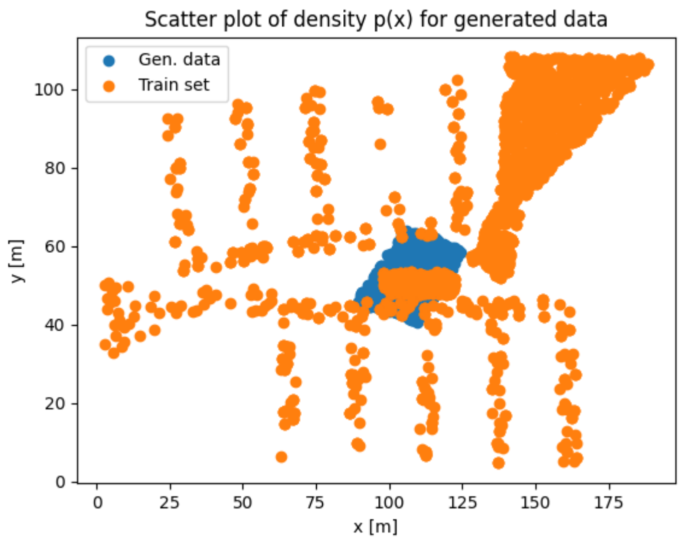


Figure 29: Plot of 1000 generated data points with trained model.

To study the critical amount of people the building can hold before the entrance is full of people, although the generated data is odd looking and does not allow us to do a correct analysis, we incorporated the function to generate a plot of the number of people in the entrance, Ne, as a function of the total number of people in the building, Nb. In the plot showed in Figure 30, the red horizontal dashed line marks the point at which the critical point (Ne = 100) is reached. It is worth noting that, just as explained before, due to the variability on the location of the generated data, and it being highly clustered, in some runs the generated quadrilateral region has an intersection with the entrance region, causing the plot of Ne(Nb) to increase very sharply and show a noisy behaviour due to the stochastic nature of the sampled data in the latent space, naturally showing very inconsistent results.

Finally, we attempted to do the bonus using Vadere, for which we created an scenario file with the indicated topology showed in the exercise sheet. The model was created with the SFM (Social Force Model) as it is the simulation model that seemed to show the most realistic results in the last exercise. Furthermore, due to the generated data being completely meaningless, we saved the training data (instead of the generated data) into a `.txt` file to load these positions and be able to simulate the circulation of pedestrians inside the MI building topology. For this, the same approach studied in the last exercise was used in the `Add_Pedestrians.ipynb` file, to parse the `.JSON` scenario file and write the pedestrians dynamically into the simulation file. Figure 31 shows the final output of the scenario created with the loaded pedestrian positions. As can be seen unfortunately, as the scenario was created by hand, some pedestrians (blue dots) contained in the original dataset actually lay off the source regions (green), a better approach could have been possibly to first load the pedestrians into an empty canvas and then, on top of the contour of the building dictated by the pedestrian positions, to create the different regions of the scenario for them to perfectly fit the pedestrians. Furthermore, when running the simulation, only a very small amount of pedestrians are actually spawned, therefore, we could also not get

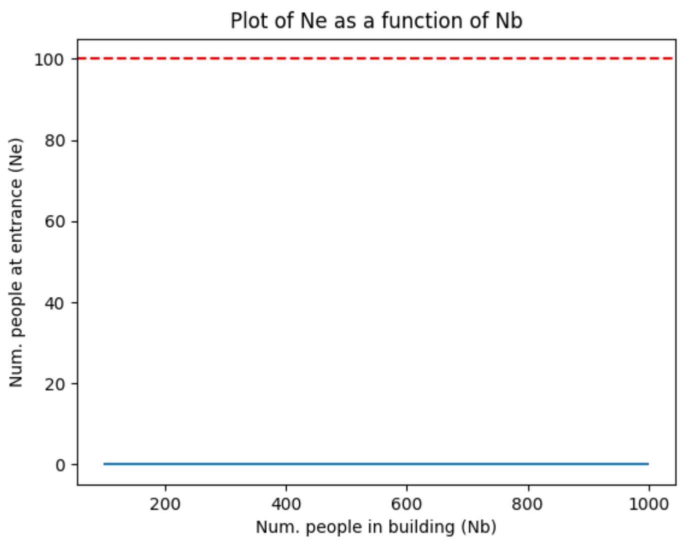


Figure 30: Plot of critical amount of people as a function of people at MI building.

significant results out of this section of the analysis.

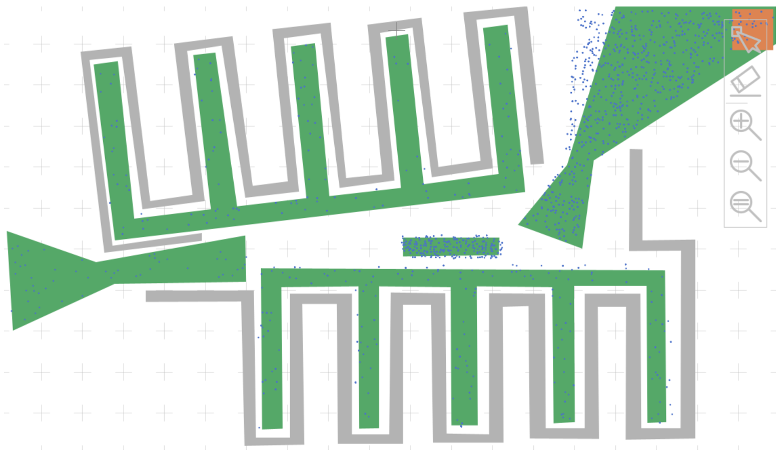


Figure 31: Picture of the MI building scenario created in Vadere to simulate pedestrian flow with task 4 training data.

References

[1] Diffusion maps: Embedding of an s-curved manifold. `https://datafold-dev.gitlab.io/datafold/tutorial_03_dmap_scurve.html`. Accessed: 2024-05-22.