Report for exercise 3 from group G

Tasks addressed:	4
Authors:	Zhaozhong Wang (03778350)
	Anastasiya Damaratskaya (03724932)
	Bassel Sharaf (03794576)
	Thanh Huan Hoang (03783022)
	Celil Burak Bakkal (03712329)
Last compiled:	2024–11–28

The work on tasks was divided in the following way:

Zhaozhong Wang (03778350)	Task 1	20%
	Task 2	20%
	Task 3	20%
	Task 4	20%
Anastasiya Damaratskaya (03724932)	Task 1	20%
	Task 2	20%
	Task 3	20%
	Task 4	20%
Bassel Sharaf (03794576)	Task 1	20%
	Task 2	20%
	Task 3	20%
	Task 4	20%
Thanh Huan Hoang (03783022)	Task 1	20%
	Task 2	20%
	Task 3	20%
	Task 4	20%
Celil Burak Bakkal (03712329)	Task 1	20%
	Task 2	20%
	Task 3	20%
	Task 4	20%

Report on task 1: Principal Component Analysis

Part 1: Implementation of Principal Component Analysis

The implementation of the utility functions in the file `utils.py` for this first part is straightforward, as it follows the steps of the Principal Component Analysis (PCA) algorithm stated in the exercise sheet. However, there is one thing we would like to mention, which is the computation of Singular Value Decomposition (SVD). Rather than calculating and implementing the entire procedure from scratch, we utilize the built-in function `numpy.linalg.svd`[1] from the NumPy library. This approach significantly reduces the computational effort, as it provides a convenient and efficient way to perform the decomposition. By setting the `full_matrices` parameter to `False`, the SVD function returns a reduced form of U and V^T, where the shapes of these matrices are adjusted to reflect the number of non-zero singular values (i.e., the rank of the matrix), rather than the full dimensions. Consequently, the matrices U and V^T are not guaranteed to be unitary. Additionally, the function returns S as a vector containing the singular values, which are sorted in descending order. In our case, the resulting shapes for U, S, and V^T are (100, 100), (2,), and (2, 2), respectively.

For visualization purposes, three scatter plots have been created (see Figure 1). All of the plots are saved in the folder `part_1` upon executing the Python script `1_two_dim_data.py`. Figure 1a shows the original dataset, while Figure 1b displays both the original and centered datasets on the same plot. We can observe that there is only a small variance between these two datasets (i.e., there is a small shift in the data after centering). Figure 1c shows the centered dataset along with the two principal component vectors, with their origins starting at the center of the centered dataset, which is at (0, 0). The endpoints of the first and second principal component vectors are, respectively, the coordinates of the first row $(-0.88938337, -0.45716213)$ and second row $(0.45716213, -0.88938337)$ of V^T. It is also noticeable from the plot that these two principal component vectors are perpendicular to each other, as principal component vectors are expected to be.

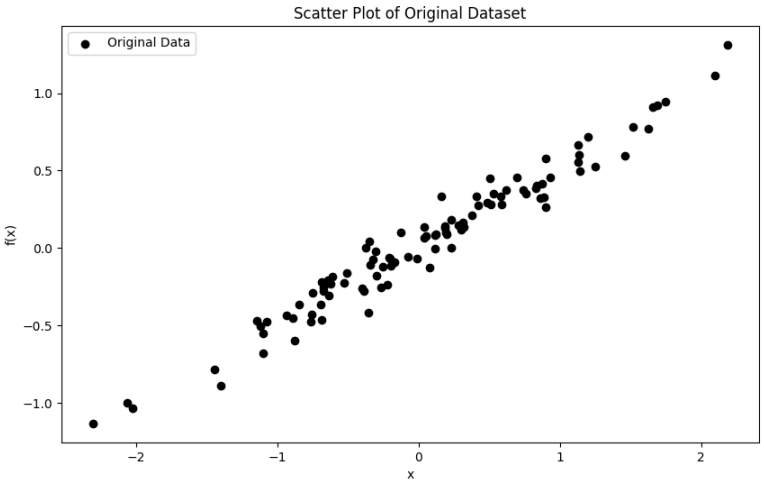
The energy captured by the first and second principal components is 99.3143% (singular value: 9.94340494) and 0.6857% (singular value: 0.82624201), respectively. Since the first principal component captures most of the energy, if we were to span the entire dataset in a one-dimensional linear subspace of the two-dimensional dataset that is optimal in terms of variance reduction, that one-dimensional subspace should be the one spanned by the first principal component vector, $(-0.88938337, -0.45716213)^T$.

Part 2: Apply Principal Component Analysis to an Image

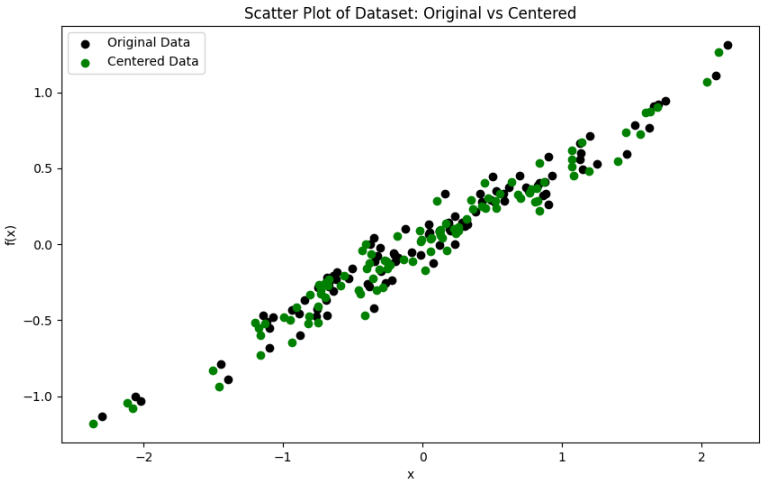
The original image is first loaded in grayscale by specifying the parameter `gray` as `True` when calling the function `scipy.misc.face()`[2], and then resized to the specified dimensions (249, 185) (width, height) using the `resize` function from `skimage.transform`[3]. The parameter `anti_aliasing` is set to `True` to ensure that the resized image appears smoother with less visible pixelation, making it look more natural, especially when the image is resized to smaller dimensions.

We then mean-center the image by calling `utils.center_data()`, and apply SVD on this mean-centered image using `utils.compute_svd()`. The mean here refers to the average pixel intensity value computed across the entire grayscaled image. Using the resulting U, S, and V^T, we reconstruct the mean-centered image with the specified number of principal components by calling `utils.reconstruct_data_using_truncated_svd()`. The function `utils.reconstruct_images()` has been slightly modified to print the cumulative energy for each of the specified number of principal components. A parameter `data_mean` has also been added so that if a value is provided, the function reconstructs the image with the mean added back, effectively recreating the original image instead of the mean-centered image. The original and mean-centered images, along with the reconstructed ones using different numbers of principal components, are shown in Figure 2 and Figure 3, respectively. The figures can be found in the folder `part_2` under the name `original_vs_reconstructed_image.png` and `original_vs_reconstructed_image_with_mean.png`. Compared to the original image, the mean-centered ones appear grayer, with weaker contrast between black and white pixels.

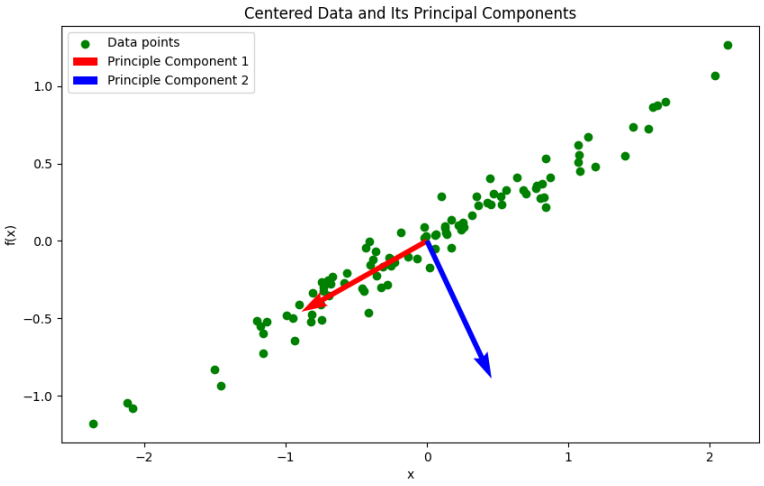
[1] `https://numpy.org/doc/stable/reference/generated/numpy.linalg.svd.html`
[2] `https://docs.scipy.org/doc/scipy/reference/generated/scipy.misc.face.html`
[3] `https://scikit-image.org/docs/stable/api/skimage.transform.html`



(a) Scatter plot of original data



(b) Scatter plot of original data and centered data



(c) Scatter plot of centered data with its principle components

Figure 1: Visualization of the dataset

The cumulative energy for all (185), 120, 50, and 10 principal components are 100.00%, 99.80%, 97.46%, and 82.13%, respectively (all numbers are rounded to two decimal places). Looking at Figure 3, the information loss when using all components and 120 components is hardly visible (energy loss for both cases is less than 1%), whereas with 50 components, we start to notice a slight lack of sharpness in the image. The information loss becomes most noticeable when using only 10 principal components, making the raccoon almost unrecognizable. The energy lost through truncation is smaller than 1% when at least 78 principal components are used. This is computed using the function `utils.compute_num_components_capturing_threshold_energy()`.

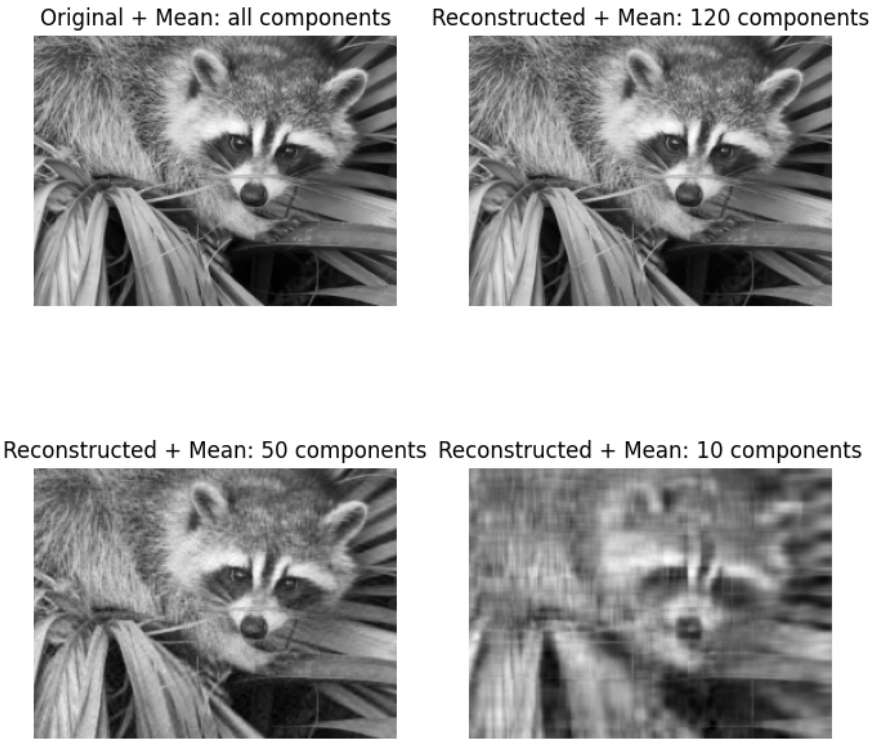


Figure 2: Original and reconstructed images using all (185), 120, 50 and 10 principal components

Part 3: Apply Principal Component Analysis to Trajectory Data

The path of the first two pedestrians in the two-dimensional space is shown in Figure 4a. We modified the `utils.visualize_traj_two_pedestrians()` function by adding a `path` parameter to specify where the visualization is saved. Additionally, we removed the `plt.subplot(1, 2, 1)` statement to allow the plot to occupy the entire figure space. The plot can be found in the `part_3` folder under the name `trajectories.png`. The path of each pedestrian appears fuzzy and jittery, which likely indicates the presence of noise in the data.

The trajectories of pedestrians 1 and 2 when reconstructing the dataset using only two principal components can be seen in Figure 4b. Compared to the original paths, the reconstructed trajectories are less jittery, but the shapes, particularly for pedestrian 2, are less accurate than the originals. The cumulative energy for the first two principal components is 84.72%, which fails to capture most of the dataset's energy. At least three

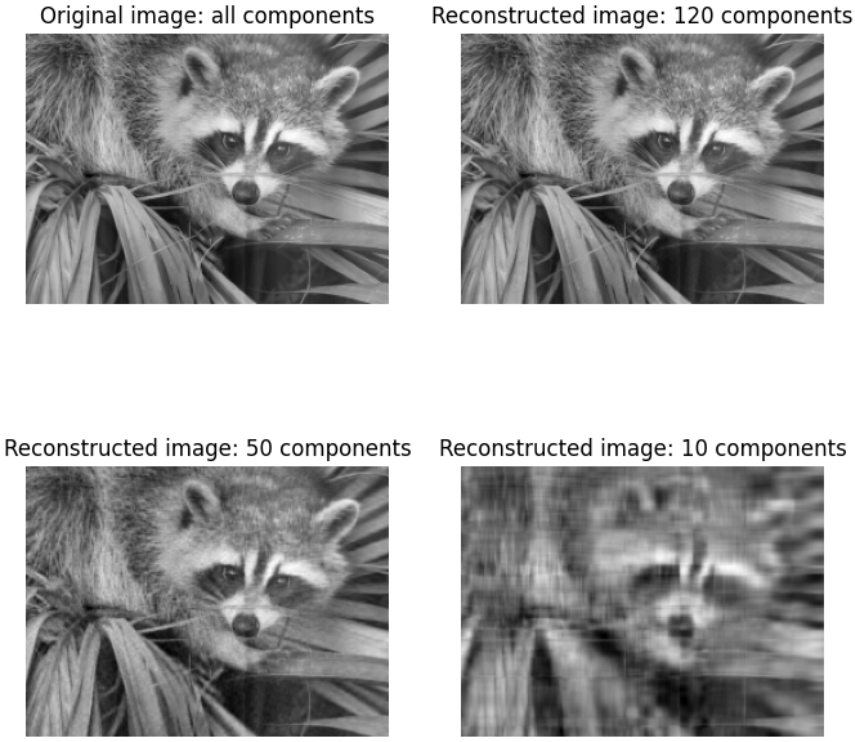


Figure 3: Original mean-centered and reconstructed images using all (185), 120, 50 and 10 principal components

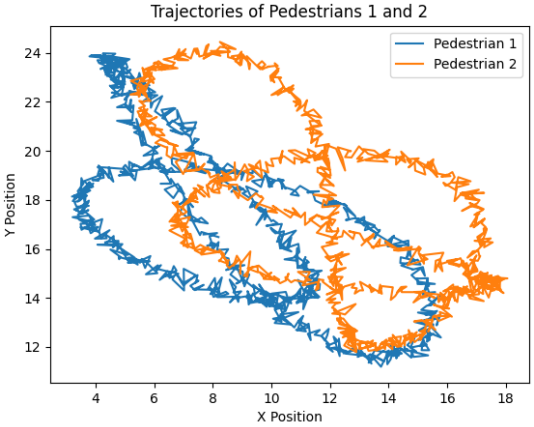
principal components are required to capture most of the energy, with a cumulative energy of 99.71% closely approximating the original data. Figure 4c shows the reconstructed paths of the first two pedestrians using three principal components, where the shapes closely resemble the original trajectories. The plots are located in the `part_3` folder under the names `trajectories_reconstructed_2_pc` and `trajectories_reconstructed_3_pc` for two and three principal components, respectively.

Both reconstructed paths using two and three principal components are less fuzzy than the originals. This is because PCA identifies directions (principal components) in the data where variance is maximized. Noise primarily affects lower-variance components, which represent smaller-scale variations. By projecting onto the first two or three principal components, these noise-dominated dimensions are effectively ignored, resulting in cleaner trajectories.

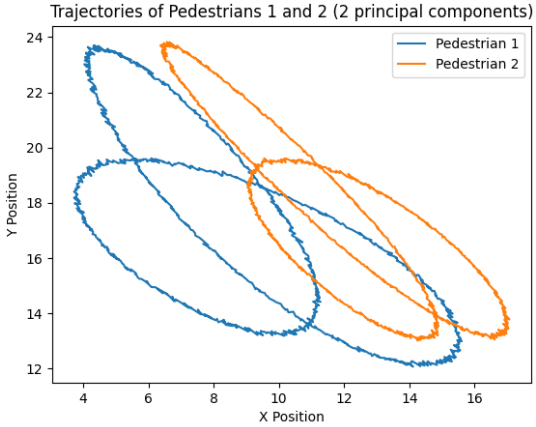
Supplement: The Questions From the First Page of the Exercise Sheet

Question (a): Implementation and Testing Time

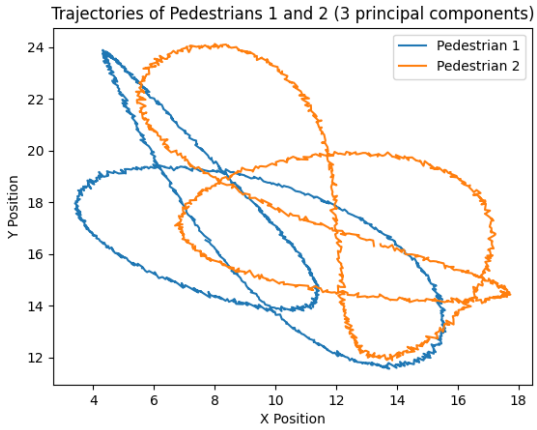
- **utils.py:** 30 minutes
- **Part 1:** 30 minutes
- **Part 2:** 2.5 hours



(a) Original paths



(b) Reconstructed paths using 2 principal components



(c) Reconstructed paths using 3 principal components

Figure 4: Comparison of original and reconstructed trajectories of the first two pedestrians in two-dimensional space, using varying numbers of principal components

- **Part 3:** 2 hour

Question (b): Accuracy of Data Representation

To evaluate how accurately the data can be represented, we use cumulative energy. In our case, achieving over 99% cumulative energy is sufficient to approximate the original data. This is illustrated most clearly in Part 2 and Part 3, where using 120 principal components for the raccoon image (with 99.80% cumulative energy) and 3 principal components for the pedestrians' trajectories (with 99.71% cumulative energy), the difference between the reconstructed data and the original data is small.

Optionally, in addition to cumulative energy, other methods can be used to assess the fit of PCA. For example, the Frobenius norm can be applied to the difference matrix between the reconstructed and original datasets. Since the Frobenius norm measures the similarity between the two matrices, a smaller value indicates a closer match to the original data. However, we do not use this method in this exercise.

Question (c): Insights from Dataset and Method

After the first task, we realized that PCA can be applied to various data types, ranging from tabular to image data. Being able to determine how many principal components are needed to exceed a certain cumulative energy threshold is valuable information that can help people make informed decisions when reducing the dimension-

ality of data. By focusing on maximizing variance, the PCA method revealed that lower-variance components represent smaller-scale variations, which can be ignored to remove noise and achieve a cleaner representation of the data.

Report on task 2: Diffusion Maps

The implementation of plotting functions is based on the answers from ChatGPT.

Part 0: Utils Implementation

In `utils.py`, we implement the functions needed for computing diffusion map: `create_distance_matrix()`, `set_epsilon()`, `create_kernel_matrix()`, and `diffusion_map()`. Additionally, some functions for plotting and accuracy calculation are also placed in `utils.py` to keep the other files concise and readable.
For `create_distance_matrix()`, our first implementation was using KDTree to find the neighbors of each point and register the distances in a sparse matrix (`scipy.sparse.lil_matrix`). Then, we use `toarray()` to convert the sparse matrix to the form we want. A problem here is that this implementation is essentially a nested loop over a 2D array, which is inefficient in Python. Later, in part 2, if we have 5000 points, it can take up to several minutes to obtain the result. So we commented out the original implementation and added a new implementation using `scipy.spatial.distance_matrix` below. As a comparison, the new algorithm takes only around 15 seconds for the same job.

Other parts related to diffusion map computation are copied and pasted from the exercise sheet PDF, so we omit the details of their implementations here. One thing to note is that for matrix multiplications like $K = P^{-1}WP^{-1}$, since P is diagonal, we do not explicitly create a matrix P and do the matrix multiplication because for large matrices it's inefficient. Instead, we utilize the fact that a diagonal matrix D multiplies another matrix A is equivalent to resizing A row-wise, and A multiplies D is like resizing A column-wise. In terms of accuracy calculation, we use the K-Nearest Neighbors (KNN) method. The basic idea is that the points that are close to each other in the original space should also be close to each other in the embedded space. For example, if in the original space point, A's four nearest neighbors are points B, C, D and E, and in embedded space, they are points C, D, E, and F, then the accuracy is 3/4=75%, because here they have three nearest neighbors in common (C, D, and E) and there are totally four nearest neighbors taken into account. One thing to note is that in `sklearn.neighbors.NearestNeighbors`, the nearest neighbors also include the point itself, which will increase the accuracy because a point will always be its own nearest neighbors no matter which manifold learning method we use. However, since we primarily handle a relatively extensive data set, the accuracy calculation will not be affected much.

Part 1: Periodic Data Set

Figure 5 shows the 2-dimensional periodic data set that can be visualized by plotting the data points in a 2-D plane. The eigenfunctions plotted against t are shown in Figure 6. In Figure 6, we can see that the eigenfunctions corresponding to the larger eigenvalues generally have bigger amplitudes than those corresponding to the smaller eigenvalues. Also, the former eigenfunctions have a lower frequency than the latter eigenfunctions. It is similar to the Fourier transform to some extent, where the high-frequency components typically represent the noise and have smaller amplitudes. In Fourier analysis, we usually filter out the high-frequency components and keep only the low-frequency ones. Here, in terms of the periodic data set, we should also use the first few eigenfunctions with lower frequencies to represent the data. If we use five eigenfunctions for embedding and ten nearest neighbors for accuracy calculation, the result is 95.2%, which is pretty good. But here, it's just a toy example, and using a diffusion map to represent a 2-D dataset (which is already visualizable) with 5-dimensional data is not sensible.

Part 2: Swiss Roll Data Set

In this part we use `sklearn.datasets.make_swiss_roll` to generate the swiss roll data set. The visualization can be seen in Figure 7 (with 5000 points). Here, we compute the first ten meaningful eigenfunctions and plot all the other eigenfunctions in the first one. The result is shown in Figure Figure 8.

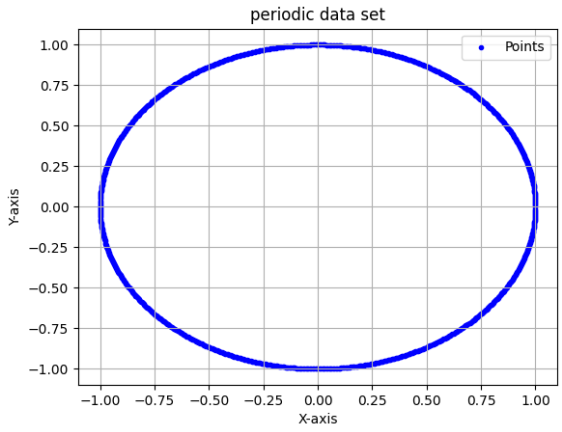


Figure 5: Visualization of the periodic dataset.

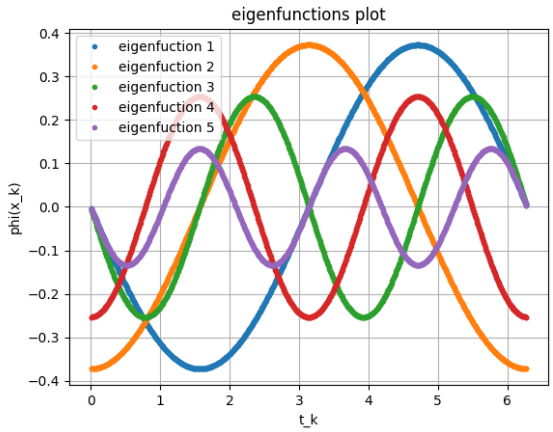


Figure 6: Eigenfunctions Ψ_1 to Ψ_5 plotted against t_k.

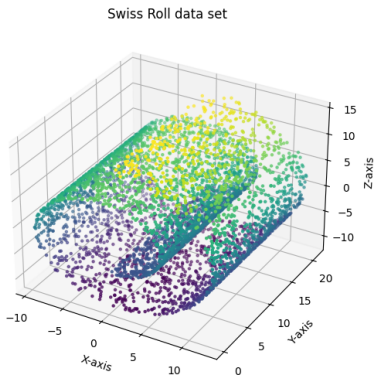


Figure 7: Visualization of the Swiss roll dataset with 5000 points.

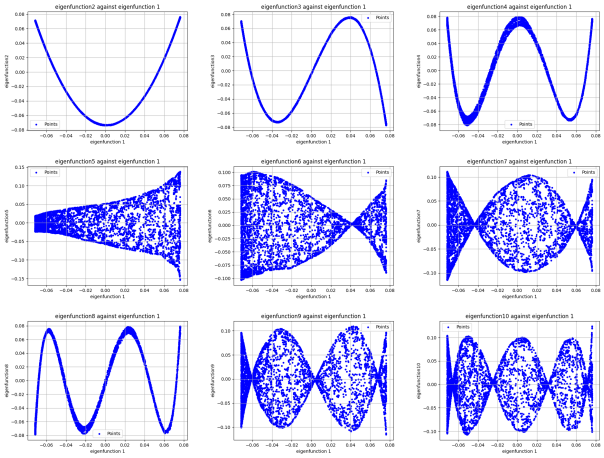


Figure 8: Eigenfunctions Ψ_2 to Ψ_{10} plotted against Ψ_1 (5000 points).

In Figure 8, we can see that starting from the fifth eigenfunction, the eigenfunctions are no longer strongly dependent on the first eigenfunction (except the eighth one). Pairings like the first and second eigenfunctions are good examples of a "functional dependence" between eigenvectors. The embeddings are poor for pairings with a functional dependence because the second eigenvector does not go along a new and independent direction compared to the first eigenvector. The best pair here should be the fifth and the first eigenfunctions because their relevance is the weakest among all pairs[4]. However, a more concise embedding doesn't tend to yield higher accuracy because we still leave out more information, although that information omitted is less significant. In our test, using ten eigenfunctions and ten nearest neighbors will give us an accuracy of around 80%, while using the best pair of eigenfunctions and ten nearest neighbors only gives us around 65% accuracy. Nevertheless, when we use manifold learning, especially for visualizing high-dimensional data, we would prefer to reduce the embedded space to only 2- or 3-dimensional. In the case of the Swiss roll, the original data set is 3-dimensional, so it is pointless to use 10-dimensional points in embedding space to represent the data. The wisest way to represent the data is to use the best eigenfunction-pair, namely the first and the fifth.

Since the Swiss roll manifold is strongly non-linear and doesn't have a low variance in every direction, so we can't use only two principal components obtained from PCA to represent the data. By calculating the explained variance of each component using `sklearn`, we see that the explained variance of each component is [50.44129405 41.02224099 37.14977531], so it's obvious that no component can be reduced.

If we use only 1000 points, the corresponding visualization and eigenfunction-pairs can be seen in Figure 9 and Figure 10. The glaring result in Figure 10 is that the relevance between all the eigenfunction-pairs seems to be weaker than in the case of 5000 points. It is the natural result of using fewer data points because it means that the Euclidean distance between the data points on the same layer of Swiss roll will be larger, and thus the similarity between them is more minor. When using 5000 data points, for each point, the nearest points on the same layer have a significantly more substantial similarity than those who lie on the other layers, so it forms a dense manifold, and the relevance of the eigenfunctions can be seen clearly. When using 1000 points, for each point, although the nearest points still tend to be those on the same layer, the similarity between them is relatively weaker, so points on the other layers will seem relatively "closer". As a result, the algorithm is less sure about the manifold structure, and in the plot, the points look more scattered, indicating a weaker relevance. Since our goal is to find the pair of eigenfunctions that are the least relevant to each other, it essentially makes the selection harder. In the case of using 1000 points, the accuracy of using all the eigenfunctions and only the best eigenfunction-pair is around 80% and 58%, respectively, with ten nearest neighbors taken into account. This also corresponds to our expectation because using only two eigenfunctions to represent a less perfect manifold structure will lower the accuracy.

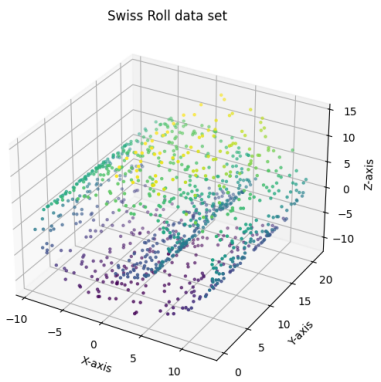


Figure 9: Visualization of the Swiss roll dataset with 1000 points.

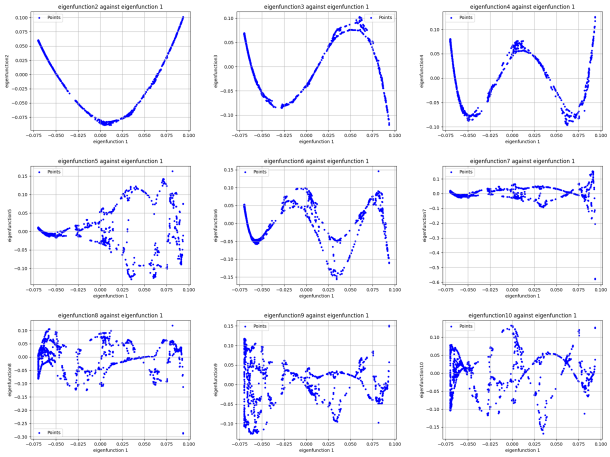


Figure 10: Eigenfunctions Ψ_2 to Ψ_{10} plotted against Ψ_1 (1000 points).

When using 1000 points, in terms of PCA, the explained variance of each component is [50.28850325 39.86954894

[4] `https://datafold-dev.gitlab.io/datafold/tutorial_03_dmap_scurve.html`

37.21693323]. There's no significant change compared to the case of 5000 points because the non-linearity of the manifold is still very strong with 1000 points.

Part 3: Embedding Pedestrians Trajectories

Part 3 handles a data set of shape 1000x30, which represents 15 pedestrians' x-y positions in 1000 timesteps. The trajectories of the first two pedestrians are displayed in Figure 11. From Figure 11, we can vaguely see the pedestrians always end their trajectories around where they start because the trajectories are smooth and curvy at most parts while being pointy at only one part. We assume that the pedestrians do not make a sharp turn in the middle of their trajectories, and then we get the conclusion stated before. We use a diffusion map to keep only the first two meaningful eigenfunctions and plot them against each other, and the result is shown in Figure 12. It is then evident in Figure 12 that all the pedestrians end their trajectories around where they start because each point in this 2-D plane represents the configuration in a timestep, and we see that the configuration changes gradually as time elapses, and the start configuration is basically the same as the end configuration because the points in embedding space form a closed ellipse.

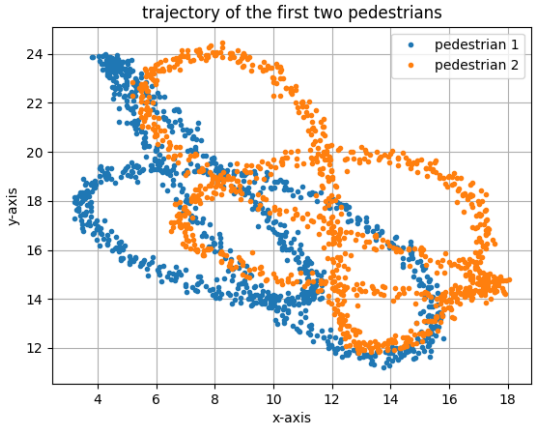


Figure 11: Visualization of the trajectories of the first two pedestrians.

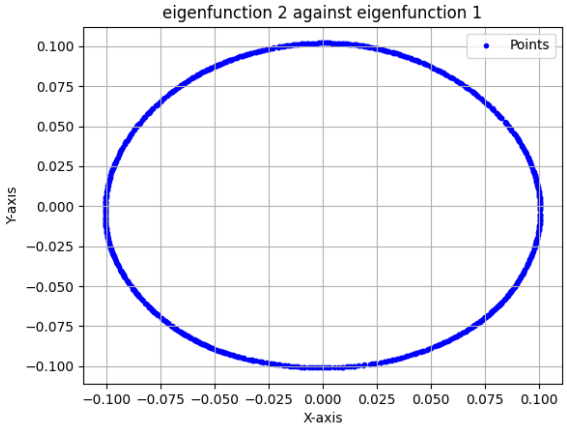


Figure 12: Eigenfunction Ψ_2 plotted against Ψ_1 in pedestrians' trajectories dataset.

By using only the first two eigenfunctions to form the embedding space, we already get an accuracy of around 92% with ten nearest neighbors taken into account. This verifies that the configuration in the original space, despite being 30-dimensional, can be effectively represented by a 2-D point in embedding space.

Bonus: Using "datafold" Package

The datafold package can help us get the eigenfunctions in a faster and more concise way. Moreover, it can help us automatically select the best eigenfunctions that are to be used for embedding the data points. For the Swiss roll, we know we can just unfold the manifold to a 2-D structure, so we use the "fixed dimension" strategy and set `intrinsic_dim`=2 when creating the instance of class `LocalRegressionSelection`. The eigenfunction-pairs' plot and the final embeddings can be seen in Figure 13 and Figure 14, respectively (here, we use 5000 points). We can see that after ignoring the subgraph "Ψ_1 vs Ψ_0" (since Ψ_0 is not meaningful anyway), the result of Figure 13 looks pretty much like the result of our own implementation in part 2 Figure 8. Since we do not specify the parameter `random_state` in the function `make_swiss_roll`, the set of points generated in the datafold implementation differs from the one in the part 2 implementation. As a result, the plots of eigenfunction-pairs also look slightly different in both cases, but it's still evident that their patterns match with each other.

Supplement: The Questions From the First Page of the Exercise Sheet

Question (b) about accuracy is already integrated into the contents above, so in the following, only questions (a) and (c) are answered.

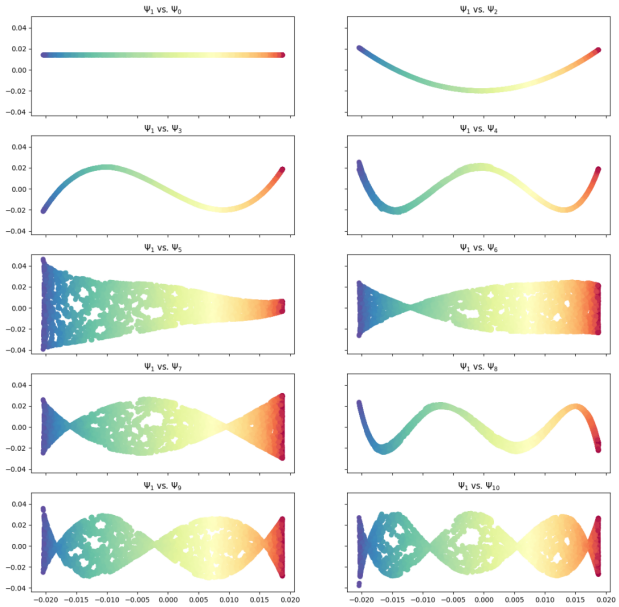


Figure 13: Eigenfunctions Ψ_0, and Ψ_2 to Ψ_{10} plotted against Ψ_1 using datafold package (5000 points).

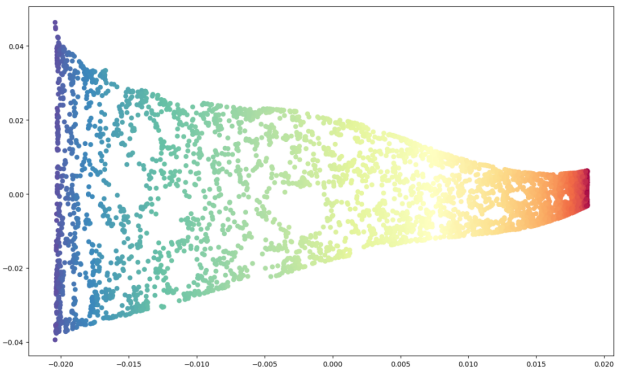


Figure 14: The best eigenfunction-pair plotted against each other (Ψ_5 against Ψ_1).

Question (a): Duration of implementing and testing each part

- **Utils.py:** implementation 1h, test 1h
- **Part 1:** implementation 30min, test 10min
- **Part 2:** implementation 1h30min, test 1h
- **Part 3:** implementation 40min, test 20min
- **Bonus datafold:** implementation + test 15min

Question (c): What I learned about the dataset and the method

About the dataset: In real life, the dataset can be pretty high-dimensional, but typically, we do not care about how they exactly look in their original space. Most of the time, we want to see the evolvement of the configuration over time, like if the configurations are converging or intersecting with each other, etc. Even if the data points in the original space form a very complex manifold, we can already gain great insight into what's really happening in the data by analyzing the manifold learning results in 2-D or 3-D embedding space.

In the Swiss roll data set, we can see that if we only want to see the similarity relationship between the points, we can actually unroll the manifold to a 2-D plane. The similarity relationship is still well preserved, i.e., the points close to each other in the original Swiss roll are still close to each other in the unrolled Swiss roll and vice versa. In the pedestrian x-y position dataset, we can see that the overall configuration will go back to where it starts. There is no intersection in the embedding space, which is natural because it is rare that two configurations in the middle of the simulation can be the same, indicating that the pedestrians all return to where they were some minutes ago simultaneously.

It is also worth noting that we always need to check accuracy to make sure we are getting the embeddings that really represent the original data well enough.

About the method diffusion map: The diffusion map method can handle strong non-linearity with the help of the kernel trick. PCA only works well if the data points approximately form a lower-dimensional manifold, i.e., their variance along one or several directions is relatively low. Sometimes, the manifold formed by the data points is like a "deformed plane" in a lower dimension. Figuratively, it's like a 2-D paper being bent into a 3-D

cylinder. The Swiss roll dataset is just a good example; it can be seen as a 2-D stripe being bent to a 3-D Swiss roll, and different layers have a high Euclidean distance between each other so they can be safely unrolled back to a 2-D plane. In the pedestrian x-y position dataset, we can see that with over 90% accuracy, the original 30-dimensional data can be represented by the 2-D embedding points. Since those 2-D points form an ellipse in the 2-D plane, we can say that the original 30-dimensional points also form something like a deformed ellipse in the original 30-dimensional space, although we never know what it looks like exactly.

Report on task 3: Training a VAE on MNIST

Activation Function Choices

For our VAE implementation, the activation functions were chosen as follows:

Encoder Output (Approximate Posterior):

- Mean (μ): No activation function (linear)
- Standard deviation (σ): Exponential function for logvar

Reasoning: The mean should be unconstrained as it can take any real value, while the standard deviation must be positive. Using logvar with exponential transformation ensures positive standard deviations while maintaining numerical stability during training.

Decoder Output (Likelihood):

- Sigmoid activation for the final layer

Reasoning: Since MNIST pixel values are normalized to [0,1], sigmoid activation ensures outputs stay within this range.

Analysis of Reconstruction vs Generation Quality

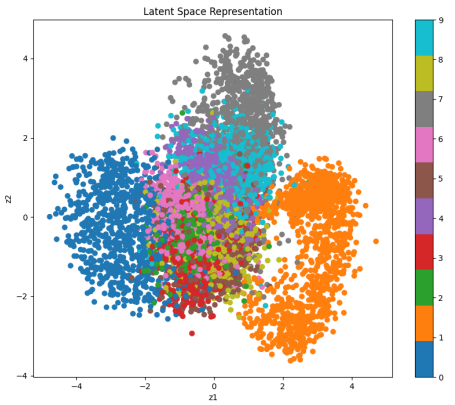
When we observe good reconstructions but poor generated samples, several factors might be responsible:

1. **Posterior Collapse:** The model might learn to ignore the latent space, leading to:
   - Good reconstructions through direct mapping
   - Poor generations due to uninformative latent space
2. **KL Divergence Imbalance:** If the KL term in the ELBO loss is too small:
   - Reconstruction term dominates
   - Latent space doesn't follow prior distribution well
3. **Overfitting:** The model might learn specific data patterns without generalizing:
   - Good at copying seen examples
   - Struggles with interpolation and generation

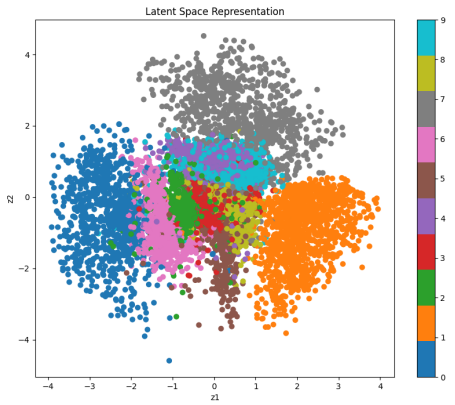
Latent Space Evolution

The evolution of our VAE's 2D latent space representation, shown in Figure 15, provides fascinating insights into how the model learns to organize and understand the MNIST digits throughout training. This progression unfolds across four key stages of training, each revealing different aspects of the learning process.

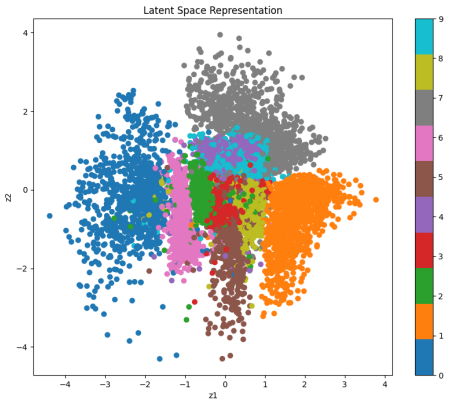
At Epoch 1 in Figure 15(a), we observe the initial attempts at organization. Even at this early stage, the model begins to show signs of understanding digit similarities, though the organization is still rough and unclear. Most digits remain heavily mixed in the central region, though interestingly, some digits (particularly the ones shown in orange) have already begun to form a distinct cluster. This early separation suggests that certain digits have features that are more easily distinguishable even in the initial stages of learning.



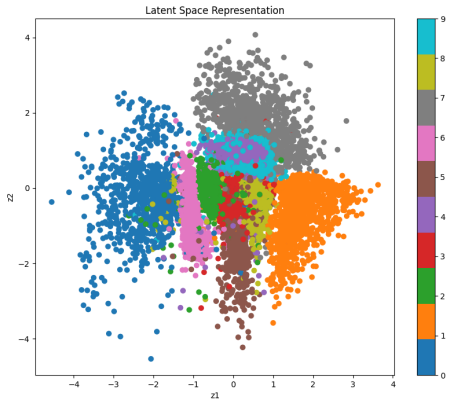
(a) Epoch 1



(b) Epoch 5



(c) Epoch 25



(d) Epoch 50

Figure 15: Evolution of 2D latent space representation across training epochs. Colors indicate different digit classes. Note how the clustering becomes more defined over time.

By Epoch 5 in Figure 15(b), the latent space organization shows marked improvement. The previously muddled central region begins to show a clearer structure, with different digit classes starting to migrate toward their own regions of the space. The cluster of ones becomes more pronounced, and zeros (shown in blue) begin to form their own distinct group. This stage represents the model's first real success at meaningful digit organization, though considerable overlap between similar digits remains.

The transformation by Epoch 25 Figure 15(c) is remarkable. The latent space now exhibits well-defined clusters for each digit class, with logical positioning that reflects real similarities between digits. Particularly noteworthy is how the model places visually similar digits, such as 4s and 9s, in proximity to each other. This organization isn't random but reflects genuine visual relationships between different digit classes. The boundaries between clusters become clearer, though maintaining smooth transitions where appropriate.

The final organization at Epoch 50 Figure 15(d) shows the culmination of the learning process. The latent space now displays a mature, stable organization with clear clustering while preserving meaningful relationships between digit classes. The model has achieved an impressive balance between separation and continuity, creating distinct regions for each digit while maintaining a logical overall structure. The consistency of certain clusters, particularly for digits 0 and 1, demonstrates the model's confidence in identifying these more distinctive digit shapes.

This visualization serves as compelling evidence that our VAE has successfully learned to compress the high-dimensional MNIST data into a meaningful low-dimensional representation, preserving both the individual characteristics of each digit class and their relationships to one another. The clear and logical organization of the latent space helps explain why the model performs well in both reconstruction and generation tasks, as it has developed a rich understanding of the underlying structure of handwritten digits.

Reconstruction Quality Progress

The reconstruction quality of our VAE shows clear progression through training, as illustrated in Figure 16. At Epoch 1 Figure 16(a), the model produces basic digit shapes but lacks detail, with most reconstructions appearing blurry and imprecise. By Epoch 5 Figure 16(b), while still blurry, the reconstructions show improved structural definition and better resemblance to the original digits.

A significant improvement is visible at Epoch 25Figure 16(c), where the model begins to capture finer details of each digit, though some softness in the reconstructions remains. The final state at Epoch 50 Figure 16(d) demonstrates the model's full capability, with sharp, well-defined reconstructions that closely match the original digits while maintaining smooth, natural-looking shapes.

The progression clearly shows how the VAE learns to balance between accurate reconstruction and meaningful latent space representation, ultimately achieving high-quality digit reproduction while maintaining the ability to generate new samples.

Generation Quality Progress

The generation capabilities of our VAE demonstrate notable evolution across training epochs, as shown in Figure 17. Initially at Epoch 1 Figure 17(a), the generated digits appear as blurry, indistinct shapes with minimal digit-like characteristics. Epoch 5 Figure 17(b) shows improvement with more recognizable digit forms, though still lacking sharp definition.

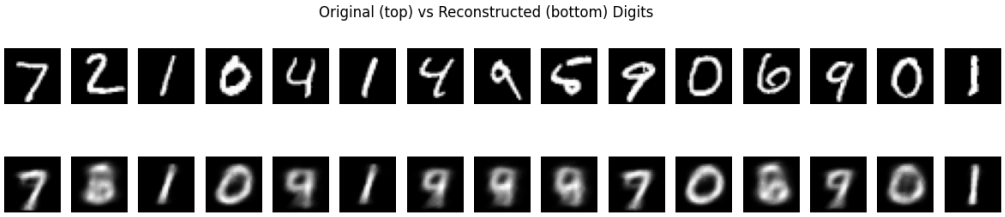
By Epoch 25 Figure 17(c), the model begins producing clearer, more distinct digits with better structural definition. The final state at Epoch 50 Figure 17(d) showcases the model's ability to generate sharp, realistic digits with natural variations in style. This progression demonstrates how the VAE learns to not just reconstruct existing digits but also generate new, plausible examples by effectively sampling from the learned latent space distribution.

Loss Curve

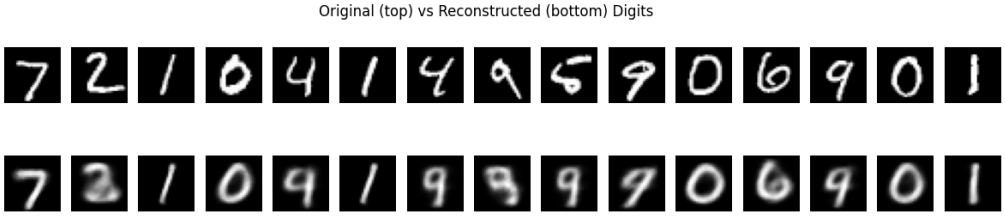
The loss curve in Figure 18 demonstrates the stable training process of our VAE model. The rapid initial descent in both training and test loss during the first 10 epochs indicates quick learning of basic digit features. The subsequent gradual improvement shows a refinement of the learned representations. The close alignment between training and test curves suggests good generalization without overfitting, while the final convergence at around 1.1 indicates successful training.

Digit Generation Comparison

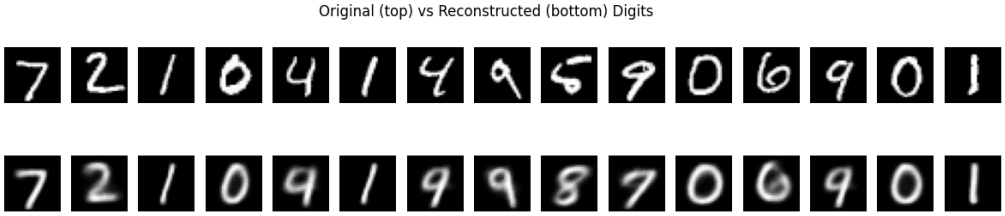
The generated digits from the 32-dimensional latent space in Figure 19 showcase distinct characteristics when compared to the 2D model's output in Figure 17(d). The 32D model produces sharper and more defined digits with notably clearer stroke patterns, as evident in digits like "5", "3", and "9". The increased latent dimensions allow for better capture of fine-grained details such as the curves in "2" and "3", and the intersections in "4" and



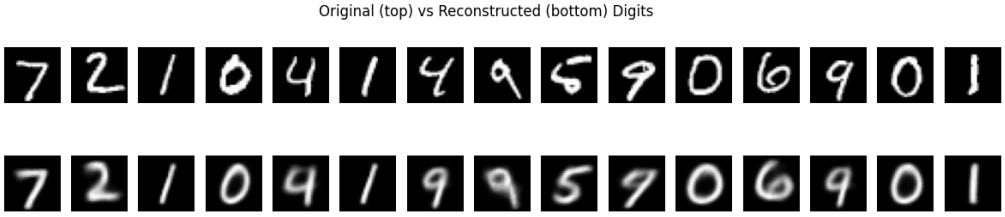
(a) Epoch 1: Initial attempts at reconstruction show basic shapes



(b) Epoch 5: Improved definition but still blurry



(c) Epoch 25: Clear digit structures emerging



(d) Epoch 50: Sharp reconstructions with good detail preservation

Figure 16: Progression of reconstruction quality. The top row in each subplot shows original digits, bottom row shows reconstructions.

"8". While both models generate recognizable digits, the 32D model demonstrates superior clarity, particularly in complex characters that require more detailed feature representation. This improvement can be attributed to the expanded capacity of the latent space to encode subtle variations in digit formation, stroke width, and orientation.

Loss Curve Comparison

Both models demonstrate stable training characteristics but with notably different performance levels as seen in Figure 20. The 32D model shows significantly faster initial convergence and achieves a lower final loss (0.8) compared to the 2D model (1.1). This improvement in ELBO loss suggests that the additional latent dimensions allow for better data compression and reconstruction. Importantly, both models show minimal gap between training and test curves, indicating robust generalization regardless of latent space dimension.

(a) Epoch 1

(b) Epoch 5

(c) Epoch 25

(d) Epoch 50

Figure 17: Evolution of generation capabilities. Each subplot shows 15 randomly generated digits sampled from the latent space.

Question (a): Implementation and Testing Time

- Total implementation time: $\sim$ 6 hours
  - VAE architecture (`model.py`): 2 hours
  - Loss functions and training loop (`utils.py`): 2 hours
  - Visualization and evaluation code: 1 hour
  - Testing and debugging: 1 hour

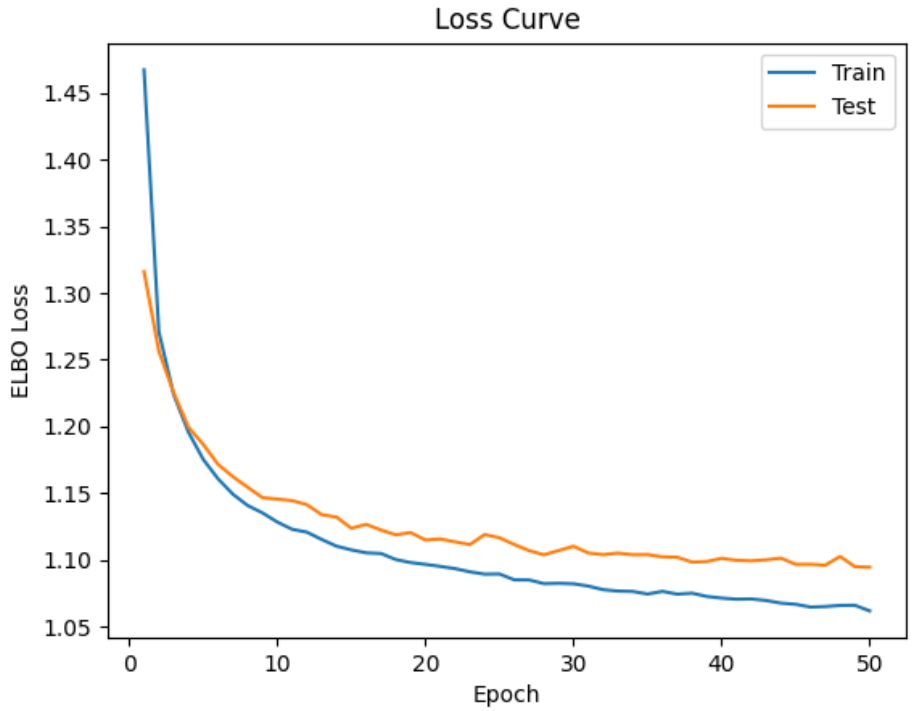


Figure 18: Training and test ELBO loss curves over 50 epochs for 2D latent space. The close tracking between training and test loss indicates good generalization, while the smooth descent suggests stable training. Final convergence is reached with training loss at approximately 1.1.

Figure 19: Generation capabilities in 32D after the last epoch

Question (b): Accuracy of Data Representation

The implementation achieved strong performance through a carefully structured architecture. The encoder network used two hidden layers of 256 units with ReLU activations, followed by separate linear layers for mean and log variance. The decoder mirrored this structure with a final sigmoid activation for normalized pixel values. The training utilized the ELBO loss combining binary cross-entropy reconstruction loss with KL divergence. The 32D model achieved superior compression with an ELBO loss of 0.8 compared to 1.1 for the 2D model, while both maintained close training-test loss alignment indicating good generalization. Reconstruction quality was validated through visualization of both reconstructed and generated samples at various training stages (epochs 1, 5, 25, and 50).

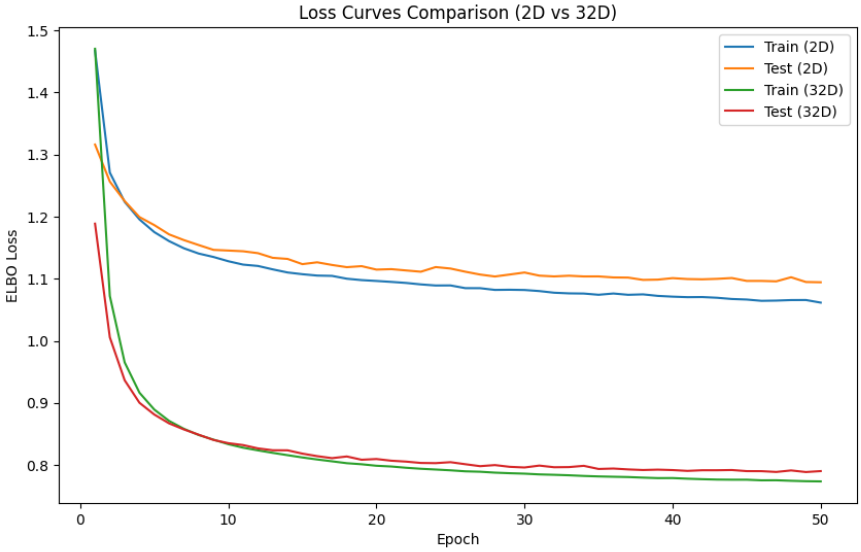


Figure 20: Training and test ELBO loss curves for both 2D and 32D latent spaces over 50 epochs. The 2D model (blue/orange) converges to approximately 1.1, while the 32D model (green/red) achieves a lower loss at around 0.8. Both models show good generalization with close tracking between training and test losses.

Question (c): Insights from Dataset and Method

The implementation revealed several technical insights about VAEs and MNIST. The Adam optimizer, with a learning rate of 0.001 and batch size of 128, provided stable training dynamics. The success of the 2D model suggested MNIST's inherent low-dimensional structure, while the improved performance with 32D highlighted additional capacity benefits. The reparameterization trick implementation proved crucial for stable training, and the choice of sigmoid activation for the decoder output effectively handled the normalized pixel value range. The progressive improvement in latent space organization demonstrated the VAE's ability to learn meaningful representations incrementally, with digit clustering emerging naturally without explicit supervision.

Report on task 4: Fire Evacuation for the MI Building

VAE Implementation and Training on the FireEvac Dataset

To train our VAE model on the FireEvac dataset and create scatter plots visualizing the training results (e.g., plots of loss curves, reconstructed test set and generated samples), the file `src/task4_fire_evac/2_fire_evac.py` can be run. The saved plots files can be found in the folder `src/task4_fire_evac/FireEvacPlots`. After downloading the train and test datasets of FireEvac and creating a plot of them, we set hyper-parameters and dimensions of our VAE model. The detailed structure of our VAE model is explained in the sub-section VAE Implementation. Afterwards, based on the recommendation of task 4 in the exercise sheet, we scale the input data to a range of [-1, 1] before training. For that, we compute the minimum and maximum values of the training set for both x and y coordinates and then apply the formula below to the coordinates of both the train and test sets. Note that for both sets, we use the minimum and maximum values of the train set to avoid data leakage from test set because otherwise we have information from test set (i.e. minimum and maximum of test set):

$$x_{\text{rescaled}} = 2 \cdot \frac{x - x_{\min}}{x_{\max} - x_{\min}} - 1 \tag{1}$$

$$y_{\text{rescaled}} = 2 \cdot \frac{y - y_{\min}}{y_{\max} - y_{\min}} - 1 \tag{2}$$

Subsequently, we instantiate a VAE model with the maximum and minimum values of train set and set up an Adam optimizer. We start a training loop, calculating the train and test loss and visualizing the plots of the reconstructed test set, generated set, and the latent space for pre-defined epochs. Ultimately, we plot the loss curves of train and test sets and use them to identify the most optimal number of epochs. The details of the loss function calculation and its plots are explained and demonstrated in section Question (b): Accuracy Measurement. In the end, we use the method `count_and_plot_coordinates_in_area`, which generates data and counts the number of people in the sensitive area for each number of generated samples (people) and makes a plot of it. With this plot, we estimate the critical number of people for the MI building.

VAE Implementation

The implementation of the VAE model used for FireEvac dataset can be found in the file `task4_fire_evac/model_fire_evac.py`. Here are the essential hyper-parameters and construction details of our VAE model:

- `learning_rate = 0.005`
- `Optimizer = Adam optimizer`
- `batch_size = 128`
- `epochs = 251` (The value of 250 returned sometimes worse train loss)
- `d_in = 2`
- `d_latent = 2`
- `d_hidden_layer = 64`
- `Activation function = Tanh (with all layers)`
- `Loss function: ELBO loss but MSE loss is used as reconstruction loss instead of BCE and KL loss is scaled with factor beta = 1 / (epoch + 1)`

As listed above, we use the same learning rate and optimizer suggested in the exercise sheet. We also use the suggested architecture of $2 - 64 - 64 - 2$ for both the encoder and decoder parts of VAE, i.e. two-dimensional latent space and 64 neurons each in two hidden layers. It is also worth noting that there is an additional output layer in the decoder part. Contrary to the suggestion in the exercise sheet, we trained the model for a longer period of time, i.e., 251 epochs instead of 200, which resulted in a slight improvement in model performance and reduced the loss. Moreover, we double the batch size from 64 to 128. This reduced the total time needed for the training process, enabled faster convergence, and slightly reduced the ELBO loss for both train and test sets. Furthermore, we used Tanh as an activation function with all hidden layers of both the encoder and decoder. We also used it at the output layer to have an output range (-1, 1), which matches the range of rescaled input data. For activation functions with hidden layers, we also tried ReLU or Leaky ReLU with a negative slope of 0.2 but they resulted in slightly worse or nearly the same loss values, so we stayed with Tanh activation function. The reason for using MSE and scaling factor beta in ELBO loss is explained in detail in section Question (b): Accuracy Measurement.

Scatter Plots: Reconstructed Test Set and 1000 Generated Samples

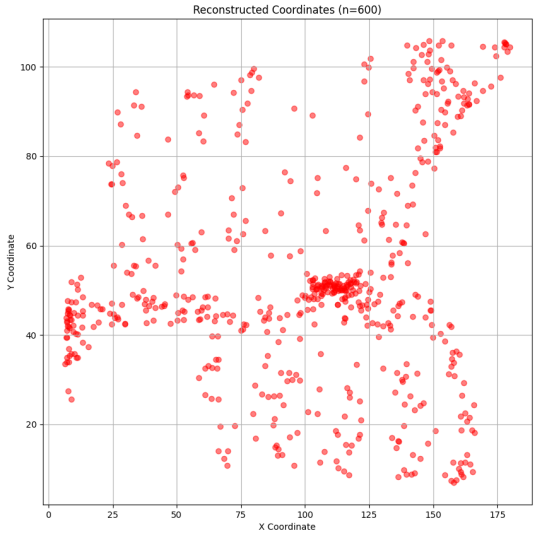


Figure 21: Reconstructed test set.

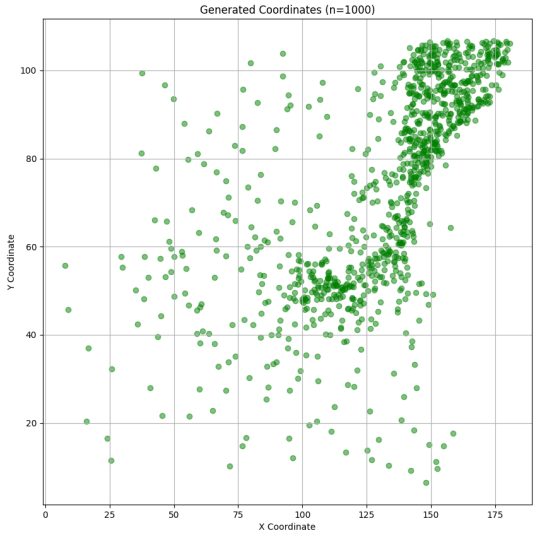


Figure 22: Generated coordinates for 1000 samples.

In Figure 21, the scatter plot of a reconstructed test set of FireEvac dataset is demonstrated and it is located in `src/task4_fire_evac/FireEvacPlots/reconstruct_coordinates.png`. This plot is done with the method `reconstruct_coordinates`, which is located in the file `task4_fire_evac/utils_fire_evac.py`. It can be seen that the model has provided a fairly successful reconstruction of the test set, which is shown with red dots in Figure 26. It has captured the more densely crowded areas, i.e., the middle of the left side and the oval-shaped area in the middle of the figure. It also did not neglect the top right area, which is less crowded in test set than the train set. It also did not neglect the top and bottom inter-spaced lines. Even though the data points do not construct perfect lines and do not have proper distance between each other, they roughly resemble the original input data.

In Figure 22, the scatter plot of 1000 generated samples from the model is displayed, which can be found in `src/task4_fire_evac/FireEvacPlots/generate_coordinates.png`. This plot is done with the method `generate_coordinates`, located in the file `task4_fire_evac/utils_fire_evac.py`. The model generally prioritizes the areas where the training set was more densely crowded, i.e., the oval-shaped area in the middle of the figure and the top right corner. The training set is shown with blue dots in Figure 26. Furthermore, even though it does not entirely neglect and generate a few points for the middle of the left side, which is dense in the test set but absent in the training set, it tends to generally prioritize the training set and ignore this area considerably. Moreover, a reasonable amount of points are generated for the inter-spaced lines on the upper and lower sides, but it still does not capture the inter-spaced linear shape well, especially for the upper side.

Critical Number of People Estimation

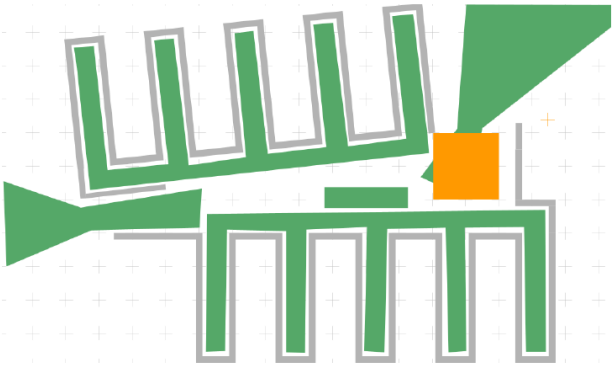


Figure 23: Schematic of the MI building in Garching, including the areas where pedestrians can be located (green) and are measured (orange). Source: Exercise sheet 3, task 4, Fig 6

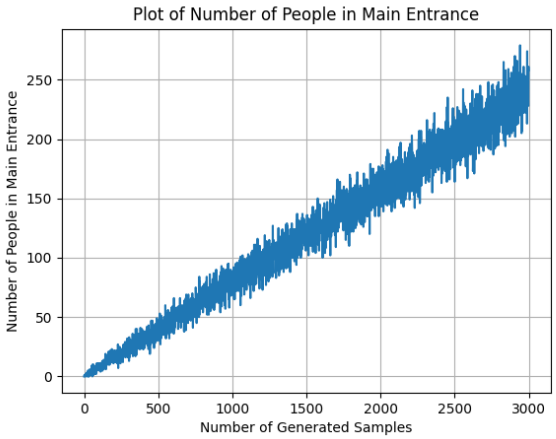


Figure 24: Scatter plot for a number of people in the main entrance of MI building.

In task 4, a sensitive area in front of the main entrance is defined with the x,y coordinates (130, 70) for the top left corner and (150, 50) for the bottom right corner. In Figure 23 it is marked by the orange rectangle. In this orange area number of people should not exceed 100.

In order to determine how many samples (people) are needed to exceed the critical number at the main entrance, we use the method `count_and_plot_coordinates_in_area` in the file `task4_fire_evac/2_fire_evac.py`. In this method, we iterate over $i \in \{1, 2, \ldots, 3000\}$, where `i` represents the number of generated samples. Then for each `i` we count how many of the generated samples lie within the orange marked sensitive area. Note that 3000 is also the size of the training set. After that, we demonstrate our results with the plot shown in Figure 24. Here, the x-axis represents the number of generated samples, and the y-axis represents the number of samples (people) at the main entrance when that number (x) of samples is generated. Based on these results illustrated in Figure 24, we conclude that approximately 1000 samples (people) are needed to exceed the critical number of 100 at the main entrance. This figure is located in `src/task4_fire_evac/FireEvacPlots/count_and_plot_coordinates_in_area.png`.

Supplement: The Questions From the First Page of the Exercise Sheet

Question (a): Implementation and Testing Time

Here is a time estimate for each of the steps implemented in task 4:

- Implementing `2_fire_evac.py`: 30 minutes
- Implementing `model_fire_evac.py`: 30 minutes
- Implementing `utils_fire_evac.py`: 1 hour
- Testing and debugging: 1 hour
- Testing: 20 minutes for hyperparameters tuning
- Bonus `MI.scenario`: approximately 1 hour, the pedestrians' positions were generated with a Python script, but the obstacles had to be placed manually, especially the above ones.

Question (b): Accuracy Measurement

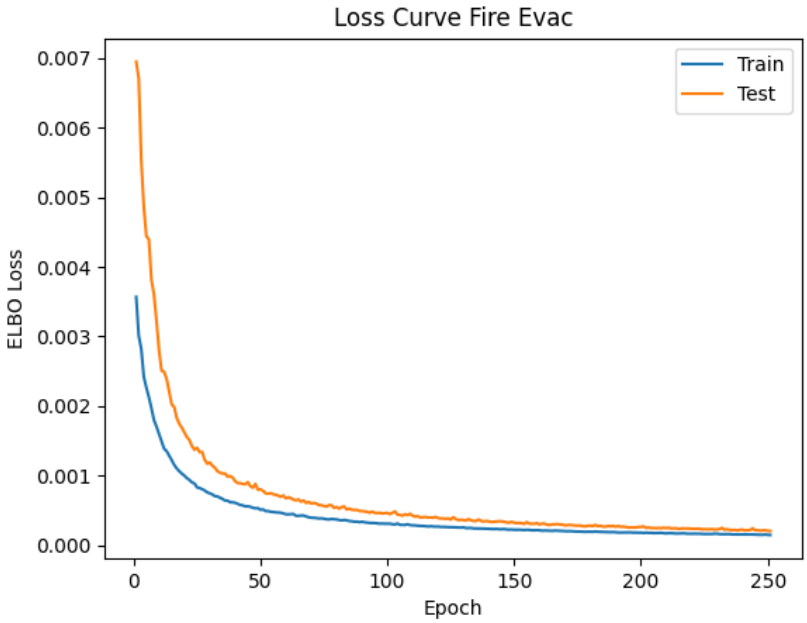


Figure 25: ELBO Loss curves acquired with train (blue) and test (orange) sets of FireEvac dataset

As a measure of accuracy, just like the previous task, Evidence Lower Bound (ELBO) loss is used. The results of our version of ELBO loss can be observed in Figure 25. It is the sum of two loss terms: reconstruction loss and Kullback-Leibler (KL) loss. Reconstruction loss rewards accurate reconstruction of input data. In contrast, KL loss acts as a regulariser (the KL) that prevents collapsing to the deterministic autoencoder by minimizing the divergence between the distribution of our latent space and a Gaussian distribution. They are terms that challenge each other, such that usually, if one gets better (lower), the other gets worse (higher). There are two main differences between the implementation of the previous task (task 3) and the current task (task 4) regarding the ELBO loss.

The first difference is that in the current task, we use Mean Squared Error (MSE) loss instead of Binary Cross Entropy (BCE) loss as the reconstruction loss. The reason for this is that we rescale the input data to a range of [-1, 1], which is also recommended in the exercise sheet, before training the model, and we also use the Tanh activation function at the output layer, which has the output range (-1, 1) and therefore matches the range of rescaled input data. Contrary to MSE loss, BCE loss requires targets given to it to be numbers between 0 and 1, therefore, it is unsuitable for our VAE model and rescaled data.

The second difference is we multiply the KL loss with a dynamic scale factor beta β. It is computed with the following formula with gradually increasing epoch number:

$$\beta = \frac{1}{\text{epoch} + 1}$$

This scale factor β is used for dampening the KL loss and holding it under control. Because the more accurate reconstructions we have (lower reconstruction loss), the worse (higher) KL loss we get. So, in order to avoid the domination of the KL loss with its high values and prevent it from hindering the reconstruction capability of our model, we have to reduce it. Therefore, with the formulation mentioned above of scale factor β, we dampen the gradually increasing KL loss more heavily with each increasing epoch number. Otherwise, the KL loss dominates the reconstruction loss and prevents the model from building accurate reconstructions. Note that the term "+1" is for avoiding division by zero at epoch 0.

From the Figure 25, it can be observed that the data was fairly accurately reconstructed and represented, such that a considerably low loss is achieved both on training and test sets. Here, the blue curve represents the training loss, and the orange curve represents the test loss. In machine learning, train and test loss are critical metrics for evaluating the performance and reliability of a model since they provide insights into how well the model is learning, generalizing, and handling unseen data. The loss curves in this figure show a stable training process, as there are no abnormal oscillations or big jumps. Both curves converge smoothly, indicating that the model learns to reconstruct data steadily. There is no large divergence between train and test loss curves that might indicate over-fitting. Both loss curves converge to a low value smoothly with the close gap between each other, which means our model generalizes well and can handle unseen data (test set) properly. The main improvement takes place approximately in the first 100 epochs. After that, it optimizes further by slowly making small improvements. In the end, we achieved considerably accurate performance results after 251 training epochs, i.e., `Train Loss: 0.0001, Test Loss: 0.0002`.

In addition to the interpretation of the loss curves, we compared the scatter plots of the reconstructed set and generated coordinates to the original input data distribution to assess how well we can represent data and also to fine-tune the hyper-parameters. The discussion and interpretation of these plots can be found in the section Scatter Plots: Reconstructed Test Set and 1000 Generated Samples.

Question (c): What we learned about the dataset and the method

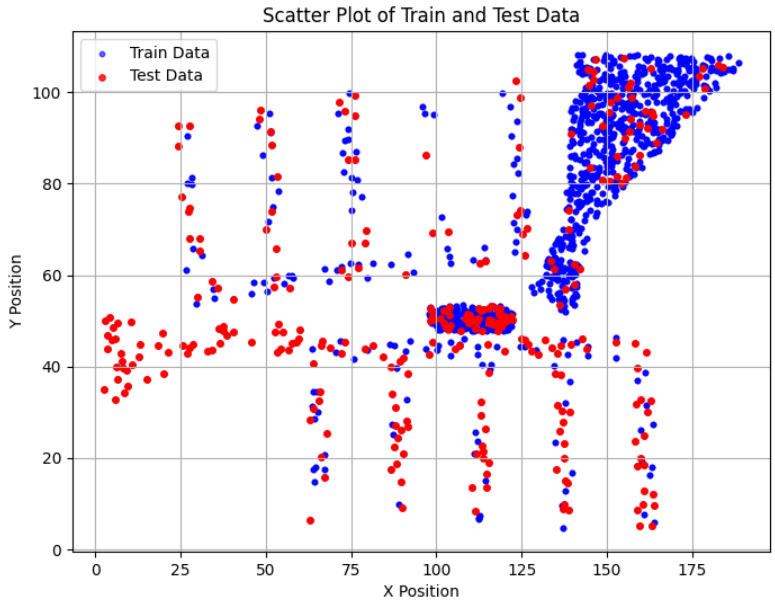


Figure 26: Scatter plot of the FireEvac Data-set.

The FireEvac dataset contains 3000 x-y-positions for a training dataset and 600 x-y-positions for a testing dataset. The positions depict the coordinates of pedestrians located in the MI building. To visualize the exact location, we distinguish between train and test data as demonstrated in Figure 26. The most filled area of train data is the top right corner (approx. between 125-180 along the x-axis and between 50-110 along the y-axis) and an oval-shaped area in the middle (approx. between 100-125 along the x-axis and between 40-60 along the y-axis), while test data is mainly filled in the middle of the left side (approx. between 0-50 along the x-axis and between 30-50 along the y-axis) and further the oval-shaped area in the middle of the figure just like the train set. All other points are distributed similarly.

Although FireEvac is a low-dimensional (2-dimensional) dataset, it is not a simple one. It has various density areas with non-trivial distinct shapes like inter-spaced lines on both upper and lower sides, an oval shape in the

middle, and the areas on the top right and middle left sides.

In our performance measurement method, i.e., ELBO loss, even with minor reconstruction improvements, KL loss increases substantially. So, in such a task requiring fairly good reconstruction performance for a non-simple dataset, KL loss should be dampened and held under control so as not to diverge too much from the input data. However, we cannot discard the KL loss completely, i.e., scale factor beta should never be completely zero, because this would make the model over-fit the training dataset too much and fail with unseen data. For example, the test set of FireEvac has a densely crowded part on the left middle area, which is completely absent in the training set. So we still need a bit of variance enforced by KL loss so that the model still generates samples (people) on areas entirely ignored in the training set.

Bonus

For the bonus task, we created a Vadere scenario to simulate the evacuation from the MI building. Starting with setting the size of the canvas, we analyzed the maximum x and y position of pedestrians possible from the FireEvac dataset. According to that, the maximum x value is 188.49331237472774, and the maximum y value is 108.17577426393086 for the training set, so the size 190*120 would include all of the needed points, as well as some additional space. To place the pedestrians in the scenario, we trained the FireEvac dataset and generated 100 samples as coordinates, as shown in Figure 27, selecting the body's radius = 1. For a more accurate analysis, we placed the obstacles as the walls in the scenario, simulating the structure of the MI building and further locating the target as required at the right top of the scenario (Figure 28). The position of obstacles was mainly obtained by analyzing the FireEvac data-set coordinates and rebuilding the sketch from the exercise sheet. The whole scenario file can be found in `src/task4_fire_evac/bonus_Vadere/MI.scenario`.

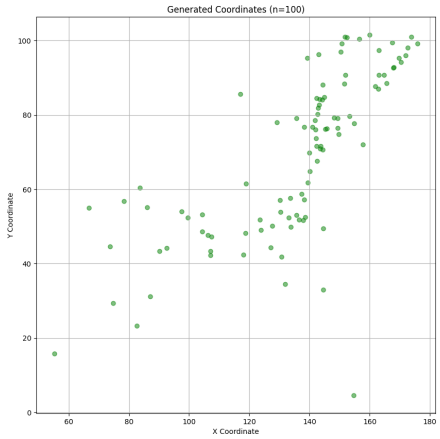


Figure 27: Generated coordinates for 100 samples.

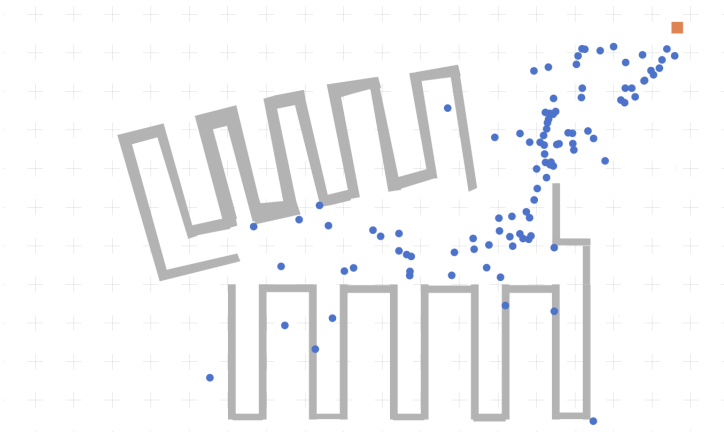


Figure 28: Vadere scenario with pedestrians starting positions based on the generated coordinates.

After executing the scenario using the Optimal Steps Model, we illustrated movement trajectories in Figure 29. Such a model was selected due to the observations from the previous exercise, where the Optimal Steps Model, Gradient Navigation Model, and Social Force Model were compared. As a result, the Optimal Steps Model demonstrated its adaptability and efficiency regarding trajectories that minimize distance and avoid overlaps and collisions. The absorption of the target is not enabled since the trajectories will be lost after reaching the target. As expected, all obstacles were avoided, and the most optimal distance was chosen. No overlapping between pedestrians was identified.

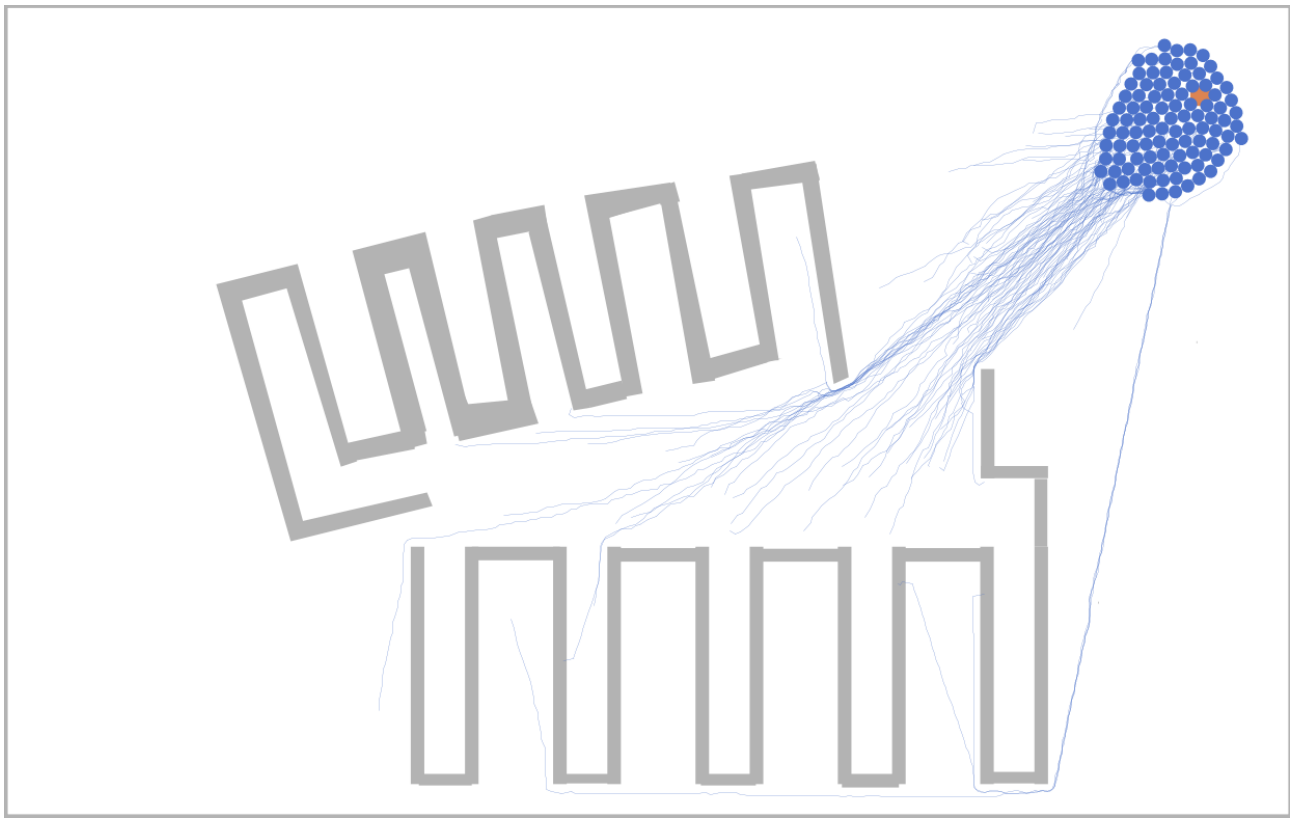


Figure 29: Vadere scenario with the trajectories of movement of pedestrians towards their target.