Report for exercise 4 from group G

Tasks addressed:	4
Authors:	Angelos Kafounis
	Caner Karaoglu
	Joaquin Gomez Sanchez
Last compiled:	2022–08–31
Source code:	`https://github.com/joaquimgomez/ml-crowds-group-g`

The work on tasks was divided in the following way:

Angelos Kafounis	Task 1	30%
	Task 2	30%
	Task 3	40%
	Task 4	33%
Caner Karaoglu	Task 1	30%
	Task 2	40%
	Task 3	30%
	Task 4	33%
Joaquin Gomez Sanchez	Task 1	40%
	Task 2	30%
	Task 3	30%
	Task 4	34%

Report on task 1, Principal component analysis

This first exercise task consists in the application of the well-known Principal Component Analysis (PCA) to three different scenarios, i.e., to a "simple" 2D data, to an image and to pedestrian trajectories data.

First, we are going to implement our own PCA using Singular Value Decomposition (SVD). With this aim, we have coded the class `PCA` available in the file `pca.py`. It has the following methods:

- `__init__`: Initialization methods that does nothing.
- `pca(data)`: It receives the data as a NumPy array, computes the mean in order to center the data and performs the SVD using the SciPy method `linalg.svd`. It saves the output of SVD as class attributes.
- `getReconstructedData(nComponents = None)`: This method reconstructs the data using the given number of required components. If it receives any number of components, it simply gets the entire U matrix, converts the S vector (the method `linalg.svd` returns the sigma diagonal matrix as a vector) to a diagonal matrix and returns the reconstruction performing $U \cdot S \cdot V$, together with the total energy of the reconstruction using the method `getTotalEnergy`. If we specify a concrete number of components, then the method "cuts" the U matrix for the given number of components (`U = self.U[:, 0:nComponents]`).
- `getTotalEnergy(SMatrixNComponents)`: In order to compute the total energy, this method works with a given sigma matrix with the eigenvalues of the selected components. It computes the sum of the squared eigenvalues of the matrix for a specific number of components and divides this sum by the trace of the "original" sigma matrix.
- `getEnergyPerComponent()`: This method computes the energy per component as $Energy_i = \frac{\sigma_i}{\sum_i \sigma_i}$ for $i = 1, \ldots, \#components$.
- `getComponents(n = None)`: It returns the components of PCA, i.e., the V matrix of the SVD. It can return all or only the first n components.

As previously mentioned, firstly, we are going to analyze a given 2-dimensional dataset. We obtain that we have two components with energies 0.99314266 and 0.00685734. We can clearly see that the first component has 99,31% of the energy and we could use it to represent perfectly the entire dataset, because it explains nearly all the variance. We also have that both energies sum up 1, so we can say that our implementation is performing well, or at least without inconsistencies.

In Figure 1 we can see the plotted centered data together with the directions of the components. We have two perpendicular (as required due to the "optimization" of PCA, i.e., $v_k = \arg\max_{\substack{\|v\|_2=1 \\ v \perp v_1,\ldots,v_{k-1}}} \|M_X v\|_2$, $k > 1$) directions. We can appreciate that the larger direction is the one explaining most of the variance, i.e. the first component, and the small one is that explaining a really small part of the variance, i.e. the second component.

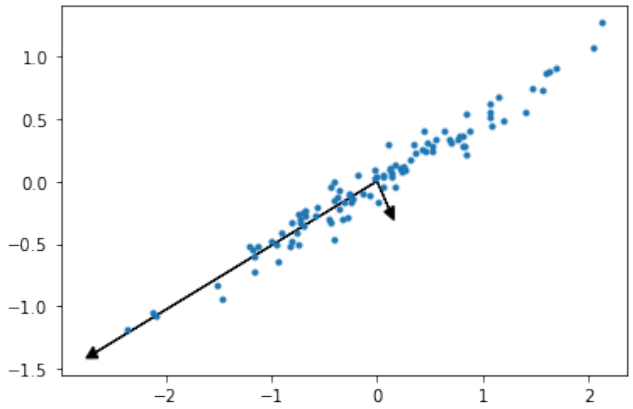
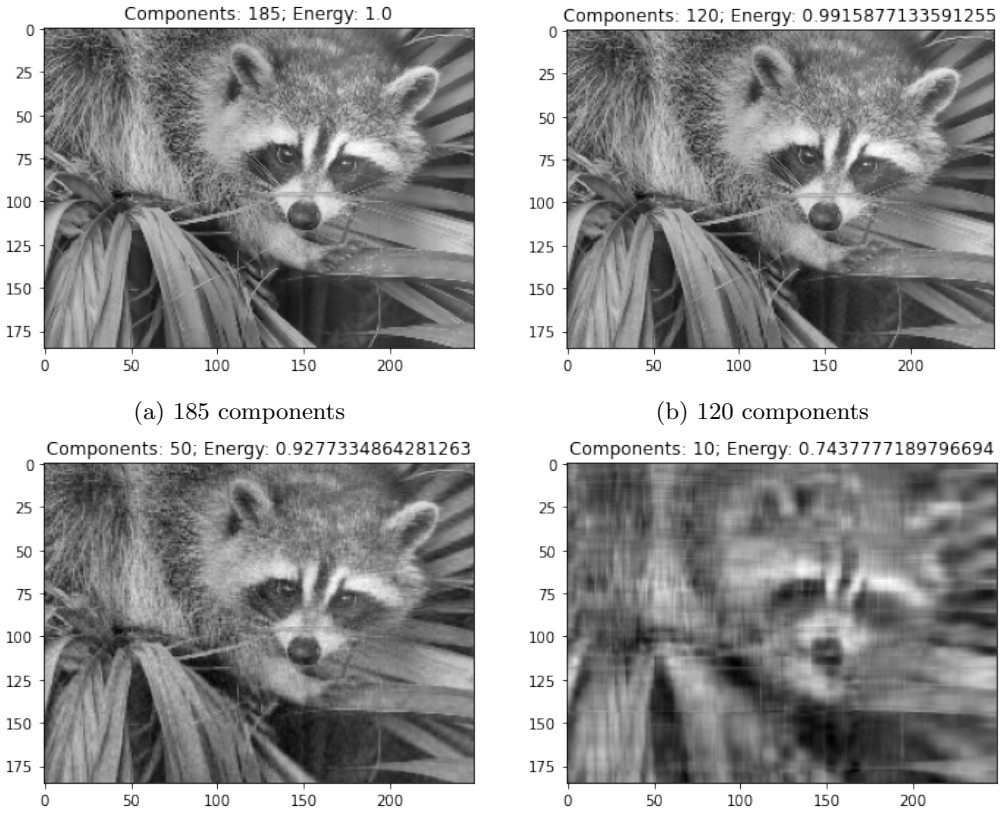


Figure 1: Centered data together with the directions of the PCA components

For the second analysis of this task we are going to work with a raccoon image in gray-scale, available through `scipy.misc.face()`. We have rescaled the image to have size (249×185) using the method `resize`

from `cv2`, the library OpenCV. We have choosen this method since it is a well-established standard for resizing images in Python. For this rescaled image we perform PCA and then we recover the image for five different number of components: 185 (the total number of components), 120, 50 and 10. In Figure 2 are the different reconstructions together with their associated energies.

From the visualization of the reconstructions we can state that for 50 components the loss of information begins to be evident, but slightly; while for 10 components it is completely visible. For 10 components we can have an intuition of the animal, but it looks too much blurred or with a lot of noise. All this becomes substantiated if we talk in terms of energy. For 50 components we have 92.77% of the energy, which is $> 90\%$ of the variation, i.e., we have sufficient information to reconstruct the data. However, for 10 components we have 74.37% of the energy, which is $< 90\%$, i.e., we do not have enough information to properly reconstruct the image.



(a) 185 components

(b) 120 components

(c) 50 components

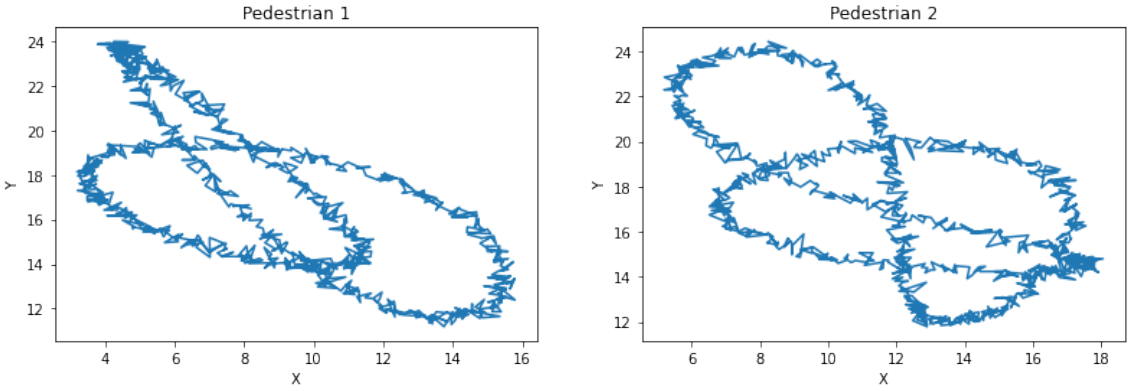
(d) 10 components

Figure 2: Reconstruction of the raccoon image for different number of components

The number of components at what the energy lost through truncation is smaller than 1% is 116 because with 116 components we capture 99.02% of the energy.

For the third and last part of this task we have to work with the trajectory data of 15 pedestrians over 1000 time steps.

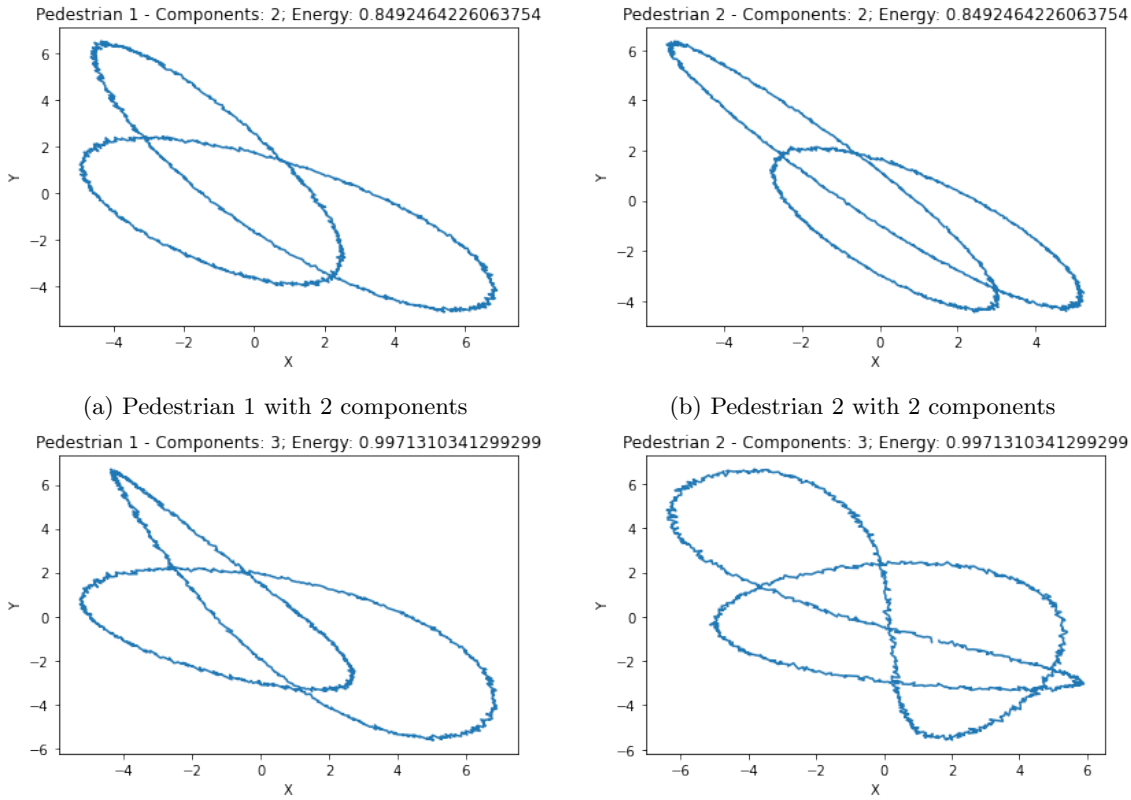
First, let us see how the trajectories look in the space. For this purpose, we plot the the trajectories of the first two pedestrians (see Figure 3). Both pedestrians seem to follow two connected circles. The trajectories look weird and erratic because of the oscillations, probably caused by the simulation model used. We expect that, using PCA, we should be able to maintain the trajectories, while simplifying them, i.e., with less oscillations due to the "lost" of information.



(a) Pedestrian 1 (b) Pedestrian 2

Figure 3: Trajectories of the two first pedestrians for 1000 time steps

We have reconstructed the images, after applying PCA, for 2 and 3 components (see Figure 4). For both number of components we can see that the trajectories seem to be simplified, having 84.92% of energy for 2 components and 99.71% for 3 components. The problem is that for 2 components we have $< 90\%$ of the energy and this clearly affects the trajectories, because the form remains more or less, but plainly deformed. It can be seen, for example, for the second pedestrian, where "the radious" of the above circle is smaller for the reconstructed trajectory. With this, we can state that 2 components are not enough to capture most of the energy/information. However, for 3 components we capture $> 90\%$ of the energy and the trajectories look identical to the original ones.



(a) Pedestrian 1 with 2 components (b) Pedestrian 2 with 2 components

(c) Pedestrian 1 with 3 components (d) Pedestrian 2 with 3 components

Figure 4: Reconstructed trajectories with 2 and 3 PCA components of the two first pedestrians for 1000 time steps

All the code implemented in order to obtain the results explained here are in the Jupyter Notebook `Task1.ipynb` and it uses our also explained implementation of PCA in `pca.py`.

The implementation and test of the methods used in this task took approximately 3 hours. PCA allowed us to represent the data using less components than the ones in a dataset, i.e., we can chose the accuracy of the representation of the data. In order to measure the accuracy of the recovered data for a given number of components we have used the energy, which indicates to us the variability explained by the number of selected components. With this task we have learned the implementation of PCA using SVD, because we already known the implementation using a covariance matrix. From the datasets we have learned that PCA can be used to simplify trajectory data in order to obtain path with less oscillations, but loosing the original coordinates.

Report on task 2, Diffusion Maps

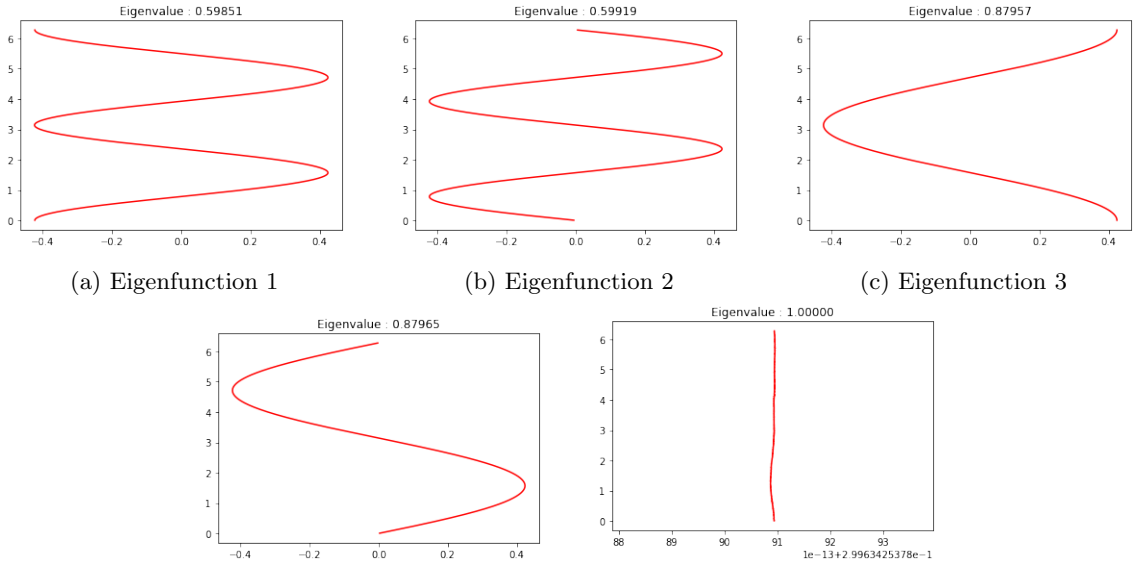
After implementing and using PCA, for this second task we are going to implement and work with Diffusion Maps, which are related to Principal Component Analysis.

Our implementation, in `diffusion_maps.py`, only consist of a class `DiffusionMaps`, with a initialization method that does nothing, and the method `diffusion_algorithm(data, L)`, which is "literal" translation to code of the described Diffusion Map algorithm in the exercise sheet.

For the first part of the task let us demonstrate the similarity of Diffusion Maps and Fourier analysis. For this purpose, we use a periodic dataset with 1000 data points given by

$$X = \{x_k \in \mathbb{R}^2\}_{k=1}^{N}, x_k = (\cos(t_k), \sin(t_k)), t_k = \frac{2\pi k}{N+1}, \tag{1}$$

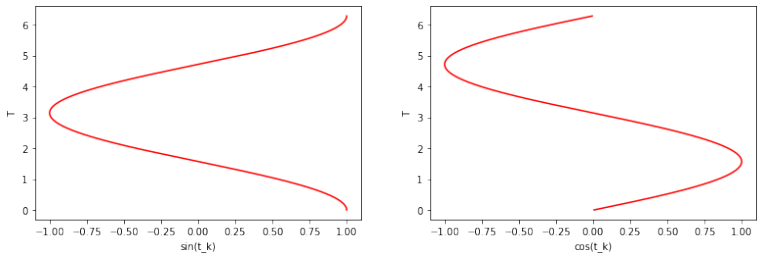
we are going to compute five eigenfunctions ϕ_l associated to the largest eigenvalues λ_l with Diffusion Maps. In Figure 5 we can see the plots of 5 eigenfunctions and in Figure 6 the Fourier analysis.



(a) Eigenfunction 1 (b) Eigenfunction 2 (c) Eigenfunction 3

(d) Eigenfunction 4 (e) Eigenfunction 5

Figure 5: 5 eigenfunctions $\phi_l(x_k)$ against t_k, associated to the largest eigenvalues λ_l and computed using Diffusion Maps



(a) sin values w.r.t to t_k (b) cos values w.r.t to t_k

Figure 6: Fourier analysis

From the Figure 5 we can depict that, as the eigenvalues of the Diffusion operators decrease, one can easily see that the oscillations occur frequently. That means, the eigenvalues are reversely measuring the frequency of each eigenvector. Also eigenvalue values around 0.87, the curves overlap both in classical Fourier analysis outputs and our Diffusion Map model outputs.

For the second part of the task we are going to work with the "Swiss roll" manifold, defined

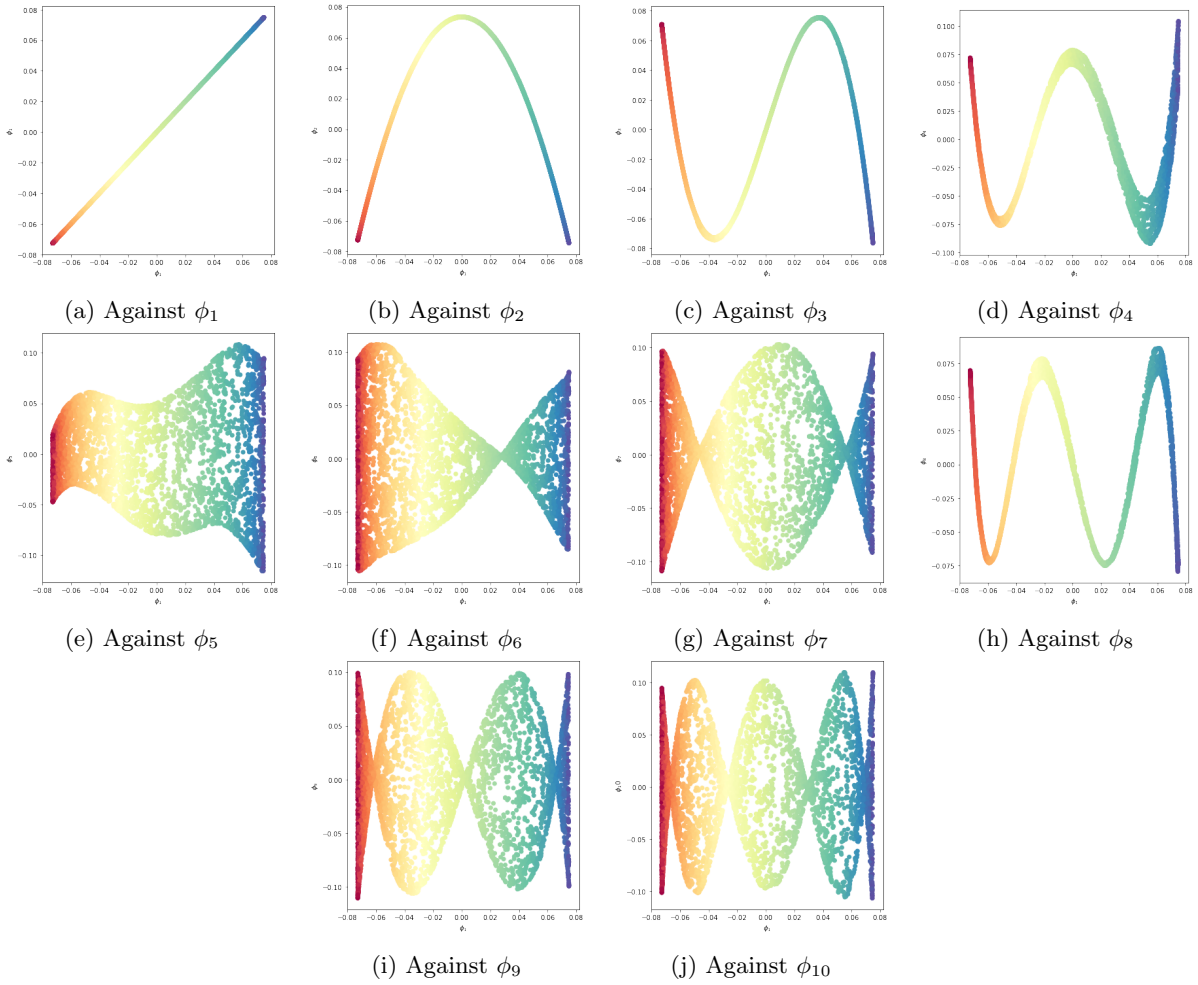
$$X = \{x_k \in \mathbb{R}^3\}_{k=1}^N, x_k = (u\cos(u), v, u\sin(u)), \tag{2}$$

where $(u, v) \in [0, 10]^2$ are chosen uniformly. To obtain 5000 data points of this "Swiss roll" we have used the suggested `sklearn` method[1].

Using the Diffusion Map algorithm we have obtained approximations of the eigenfunctions. In Figure 7 we can see the first non-constant eigenfunction ϕ_1 against the other eigenfunctions.

By looking at the plots (Figure 7), one can see that, after the $l = 4$ of ϕ_l, ϕ_l becomes no longer of a function of ϕ_1. After that particular point, we cannot clearly visualize the whole behavior of the system by only looking the 2D plot. That means, further values of ϕ_l are not only dependent on the ϕ_1.

[1] Available at: `https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_swiss_roll.html`.



(a) Against ϕ_1 (b) Against ϕ_2 (c) Against ϕ_3 (d) Against ϕ_4

(e) Against ϕ_5 (f) Against ϕ_6 (g) Against ϕ_7 (h) Against ϕ_8

(i) Against ϕ_9 (j) Against ϕ_{10}

Figure 7: First non-constant eigenfunction ϕ_1 against the other eigenfunctions

If we compute the three principal components of the data using the PCA implementation described in the previous task we obtain that the energy per component of them are: 0.389, 0.333 and 0.278. If we decide to use only two principal components this is a bad decision, since with two components we only capture $\sim$72% of the energy, i.e., less than 90%. Using two components we have also that the distribution of the data is not correctly preserved, as we can see in Figure 8.

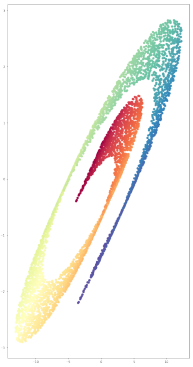
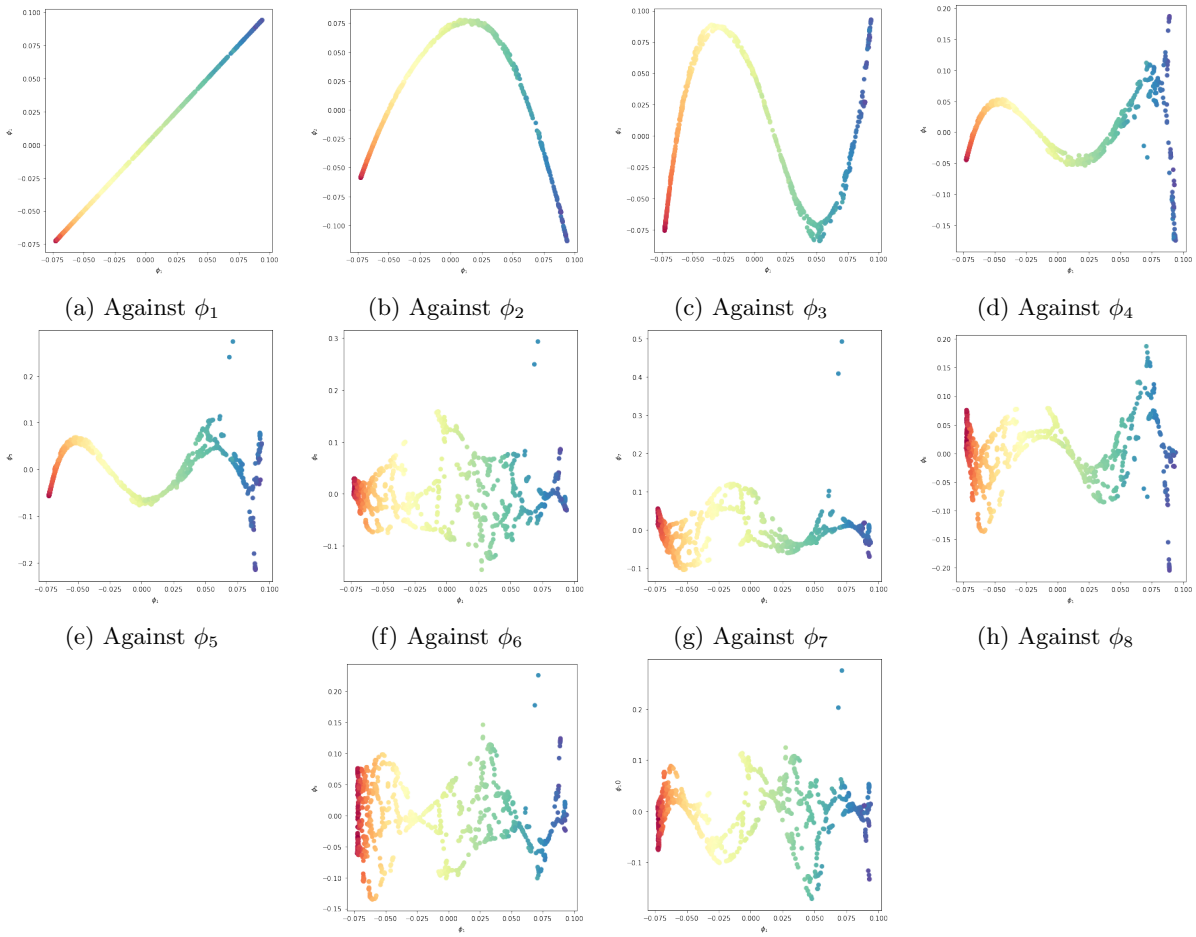


Figure 8: Representation of the Swiss-roll with 5000 points and 2 components

Now, let us perform the same analysis, but with 1000 data points instead of 5000. We can see in Figure 9 how, after aplying Diffusion Map, the plots of the eigenfunction ϕ_1 against others are not correct and homogeneous

as the ones with 5000 data points. This is due to the lack of information or the sparsity of the data points.

If we apply PCA to the 1000 data points, we obtain that the energy per component is: 0.407, 0.321 and 0.272 respectively. We have that the distribution of energy is almost the same. If we use two components to represent the data we capture ∼73%, the same as for 5000 data points, but in this case the distribution of the data is different, as we can see in Figure 10. The distribution seems closer to the original, but as we are not capturing 90% of the energy, we cannot state that we are correctly representing the data with 2 components.



(a) Against ϕ_1 (b) Against ϕ_2 (c) Against ϕ_3 (d) Against ϕ_4

(e) Against ϕ_5 (f) Against ϕ_6 (g) Against ϕ_7 (h) Against ϕ_8

(i) Against ϕ_9 (j) Against ϕ_{10}

Figure 9: First non-constant eigenfunction ϕ_1 against the other eigenfunctions with 1000 data points

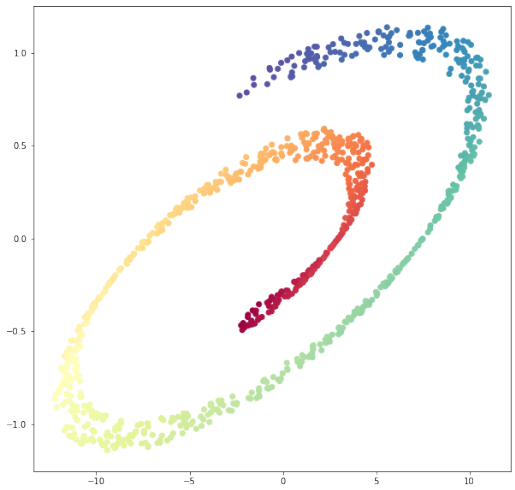


Figure 10: Representation of the Swiss-roll with 1000 points and 2 components

For the last part of the second task we are going to work with the given trajectory data. In this case, if we want to perform the same analysis as in PCA, we cannot use the concept of energy in the same way. Instead, if we want to know how many eigenfunctions we need to accurately represent the data we have to check at what number of eigenfunctions the intersections of the curves appear. We can state that the curves appear for $L \geq 3$, i.e., with four eigenfunctions, as we can see in Figure 11.

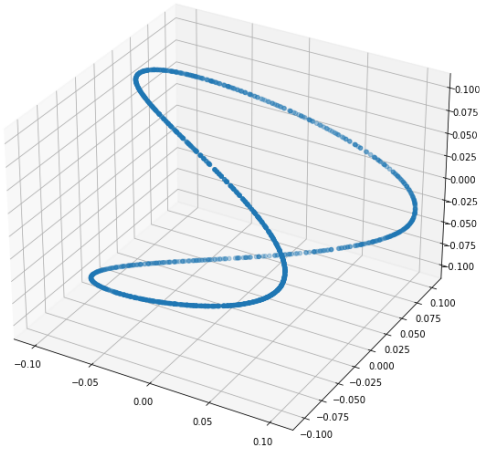


Figure 11: Four eigenfunctions for the trajectory data

In terms of time, implementation and testing of the method took approximately 3-4 hours. The data that we have used for the method is obtained directly from the Fourier transform equation, which stated in the exercise sheet. Also, using this data we have calculated the Euclidian distance matrix to obtain pairwise distance metric for accuracy measuring. Due to implementation problems, we could not have to manage to integrate the KDTree sparse matrix evaluation part to the algorithm. As the data consists of oscillations i.e outputs of the sin() and cos() functions, our model basically removes the influence of the initial sampling density of the data points. That means that for the approximation of the optimal basis functions using Laplace-Beltrami operators, we discard the sampling density of the original dataset by computing the eigenfunctions of the Laplace-Beltrami operator.

Report on task 3, Training a Variational Autoencoder on MNIST

In this third task we are going to implement a Variational Autoencoder (VAE) and train it using the well-known MNIST dataset consisting of digit images. For the implementation of VAE we have followed a tutorial

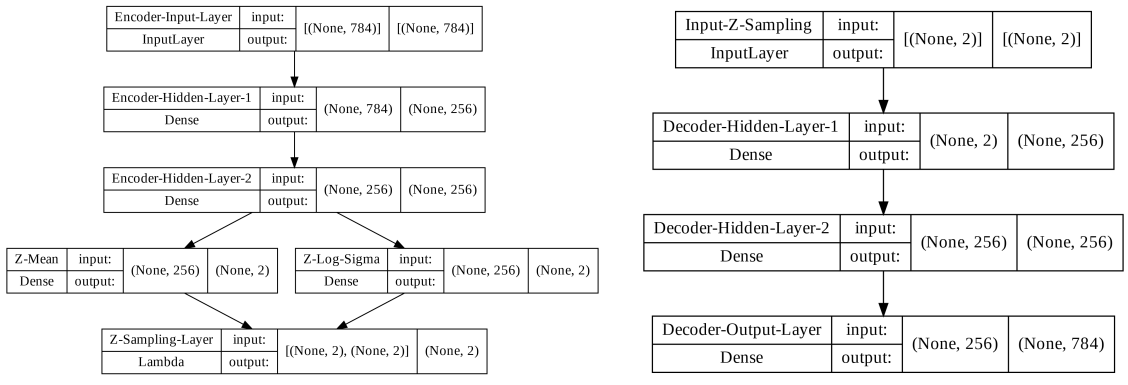
published in Towards Data Science[2].

We have obtained the MNIST dataset using the function of TensorFlow `keras.datasets.mnist.load_data()`. After that we have normalized the images diving by 255 and reshaped them.

In order to implement and train the VAE model we have used TensorFlow and we have coded the following methods:

- `create_encoder(original_dim, latent_dim)`. This method returns an encoder with one input layer with the specified `original_dim` input dimensions, two hidden fully connected layers with 256 units each adn ReLU activation functions. The output of the encoder is the mean and the standard deviation of $q_\phi(\mathbf{z}|\mathbf{x})$, i.e., we have a multivariate diagonal Gaussian distribution as likelihood.
- `create_decoder(latent_dim, original_dim)`. This method returns a decoder, which is a reflection of the encoder, but it outputs only the mean of $p_\theta(\mathbf{x}|\mathbf{z})$.
- `create_vae(encoder, decoder, visible, original_dim)`. Given an encoder and a decoder, this methods compiles a VAE model with both parts. It also defines a reconstruction loss (using the Mean Square Error (MSE)) and the KL divergence loss to compare the encoded latent distribution with the standard normal. The VAE loss is a combination of both losses. As an optimizer we use the Adam optimizer with a learning rate of 0.001.
- `sampling(args)`. A function used to randomly sample for the latent space distribution.
- `train_and_plot_vae(vae, encoder, decoder, X_train, X_test, epochs, batch_size, callbacks)`. Given the VAE model, the different parts, the data, the number of epochs and the batch size, this method trains the VAE model using the data for the given number of epochs, using the given batch size. While training, it plots the model loss by epoch. We have fixed the batch size to 128 for the following experiments and we have used as a callback the `EarlyStopping` from `Keras`, which given a patience, allows us to train until convergence, taking into account the validation loss.
- `visualize_latent_space((encoder, X_test, y_test))`. It plots the latent space of the given encoder for a given dataset. It adds color depending on the given labels.
- `reconstruct_digits(X_test, encoder, decoder)`. Given a specific set of data, it predicts with the entire VAE for each instance, in order to reconstruct the data.
- `generate_digits(decoder)`. Given a decoder, this method generates 15 digits using the model.

The Figure 12 outputted using the method `plot_model()` from `tensorflow.keras.utils` shows an schematic view of the encoder and the decoder, both with the architecture previously explained.



(a) Encoder (b) Decoder

Figure 12: Schemes of the encoder and the decoder

[2]Available at: `https://towardsdatascience.com/vae-variational-autoencoders-how-to-employ-neural-networks-to-generate-new-image`

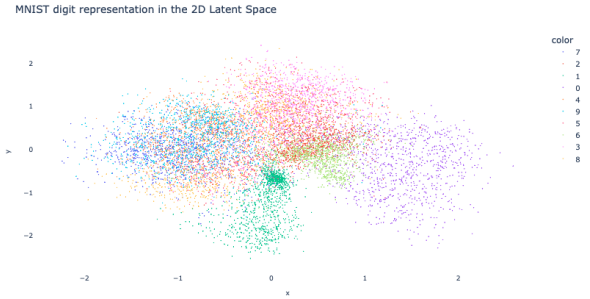
As we are working with the mean and the standard deviation of the approximate posterior, and the likelihood, an important decision is the activation function of them. The best choice here will be the sigmoid (or the softmax) activation function, because we are building on probabilities in a standard Normal distribution and this activation function always return values between 0 and 1. More precisely, its output can be interpreted a a distribution of probability.

A problem that we can have with the VAE model is that we can obtain good reconstructed but bad generated images. The reason for that might be that the variational autoencoder is overfitting on the data. As a result, when we try to reconstruct some digits, our model has learnt quite good how to reconstruct them. On the other hand, we are called to use our model in order to generate new digits, then our model can't function well because it hasn't learn to generalize well.

First, we are going to analyze the training and the results using a 2-dimensional latent space. For this purpose we are going to analyze the results after the 1st epoch, the 5th epoch, the 25th epoch and the 50th epoch. We also include the result training until convergence. In order to decide this convergence we have used early stopping, as mentioned befor, with a patience of 5. Our model converges at epoch 34.

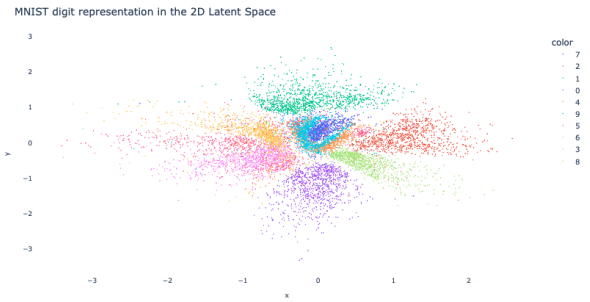
Let us plot firstly the latent representation for the different epochs in Figure 13. We can see how at the beginning the clusters are mixed except for the green (digit 1) and purple (digit 0) cluster. But, the purple cluster has a part mixed with the olive green (digit 6), which makes sense since both have circular form. We can appreciate that the model rapidly separates the clusters because at the 5th epoch the different samples are more or less well separated. The difference between the 5th and the 25 epoch is that for the 25 epochs the clusters look less sparse, and all the clusters seem better separated except for the blue clusters (digits 9 and 7). Between the 25th and the 50th epoch the differences, broadly speaking, are minimal, except for the change on the distribution of the clusters. This absence of improvements could imply, just looking at the latent space, that we have reach the learning capacity of the model.

This last points is clear when we have that the model converges at the 34th epoch. Looking at the latent space of that epoch we can see that we have the better clusters because all them are clearly separated, with small errors. The only complicated clusters are the hard orange (digit 4) and sky blue (digit 9), which makes sense because in many handwriting forms both digits can be really similar.

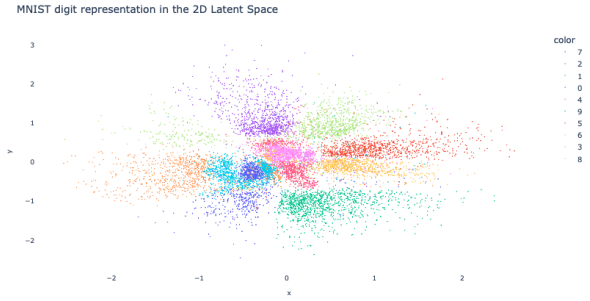


(a) 1st epoch

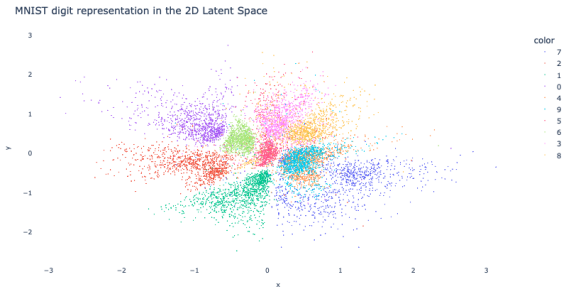
(b) 5th epoch



(c) 25th epoch



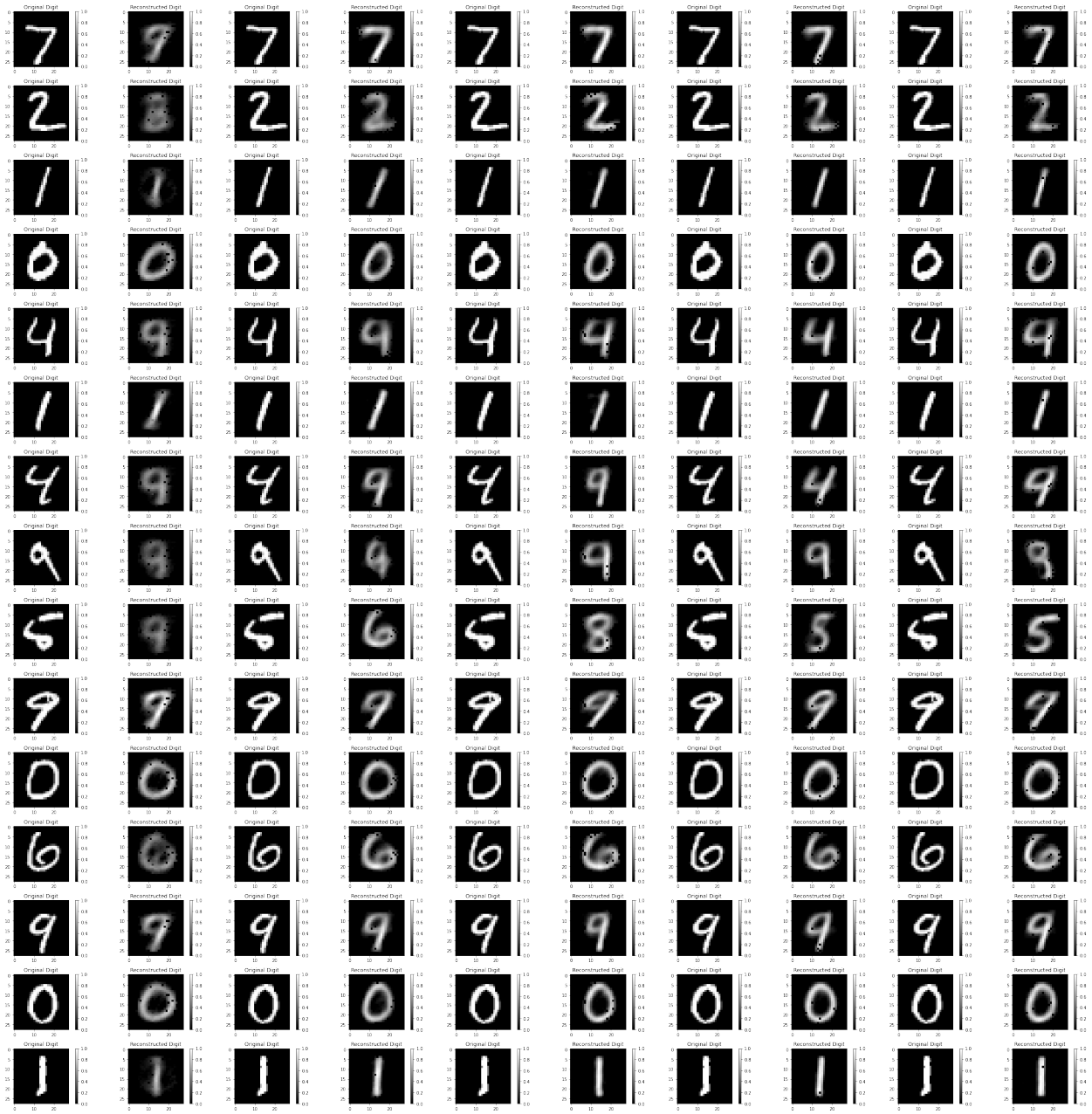
(d) 50th epoch



(e) Convergence (34)

Figure 13: Latent representation of VAE model for different epochs

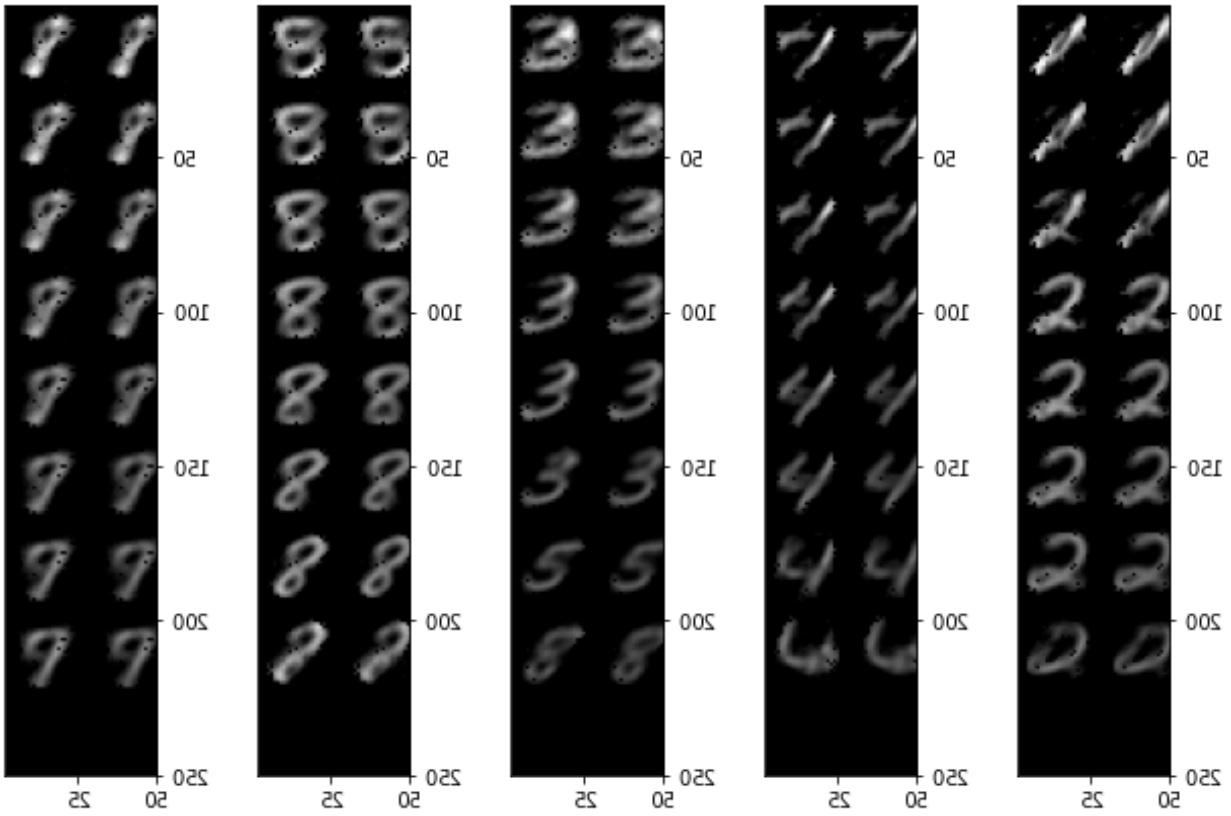
Now let's see how the model has learned to reconstruct images. Looking at the different epochs reconstruction in Figure 14 we can see how at the first epoch the models reconstructs wrongly numbers like 7 (1st row, reconstructed as 9), like 5 (9th row, reconstructed as 9) or like 6 (row 12th, reconstructed as something similar to 0). At the 50th epoch the model clearly reconstruct well all the numbers, we can say that it reconstructs even better than in the converged epoch. In the converged epoch, i.e., the 34th epoch, some numbers like the four of the 7th row is reconstructed as a nine.



(a) 1st epoch (b) 5th epoch (c) 25th epoch (d) 50th epoch (e) Convergence (34)

Figure 14: Reconstruction of digits for different epochs

The last thing to check in a Variational Autoencoder is the generation. In Figure 15 there are 15 different generations for the aforementioned epochs. At the first epoch the generations are not very intuitive, except for the last generations which seem nines. For the 5th and the 25th epochs the generation improves, but with small errors like the threes with the bottom circle closed. The generations of the 50th epoch are bad, except for some generated fours. We can also easily see that there are a lot of black pixels without value.



(a) 1st epoch (b) 5th epoch (c) 25th epoch (d) 50th epoch (e) Convergence (34)

Figure 15: Generation of digits for different epochs

If we take a look at the loss curve (see Figure 16 we can see how the model learns until the convergence at the 34th epoch. Before that, the learning gap increases from the 20th epoch, so we can see that the model starts overfitting, then the convergence at the 20th epoch makes sense given the patience of 5 for the early stopping.

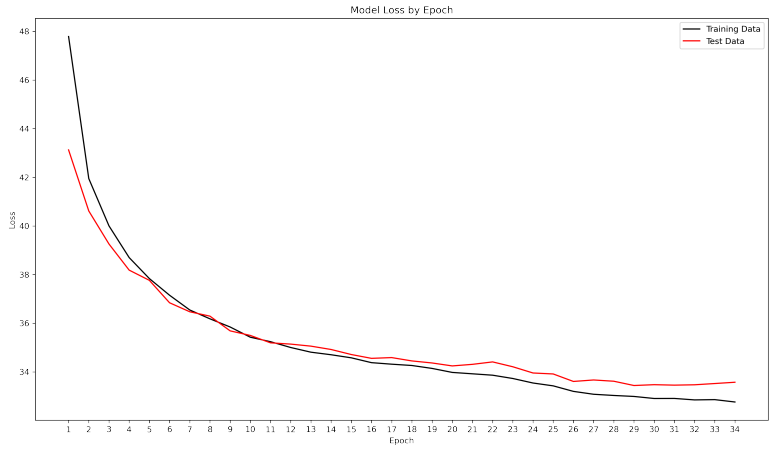


Figure 16: Loss curve

Finally, let us see the impact of changing the latent space to a 32-dimensional latent space. First, let us analyze the generation capacity. If we take a look at Figure 17 we can see that the model is really good generating the number 8 and also the number 1, but it still having imperfections.

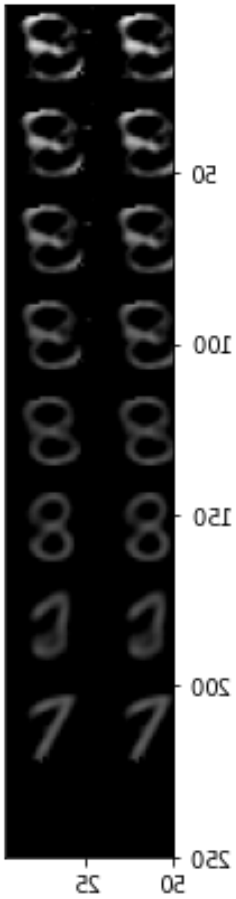


Figure 17: Generation of digits with the 32-dimensional latent space VAE model

Looking at the loss plot (see Figure 18) we can see that the model starts with a loss greater than 40, as the one with 2-dimensional latent space. This model trains for more epochs, 45, but it also achieves a better loss, ∼22.5. We also have that the learning gap is smalls, this means that the model is not overfitted.

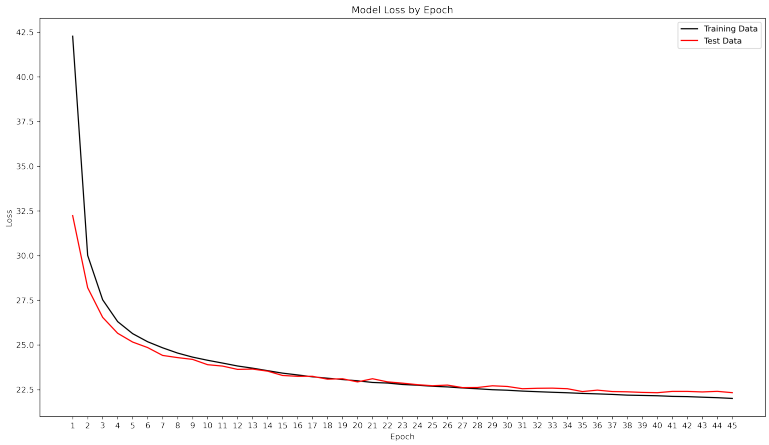


Figure 18: Loss curve for the model with a 32-dimensional latent space

Concluding, here are the answers to the important questions about this task.

1. It took us around 1 day to implement the model, understand and manipulate the data in order to find out how to put them into a functioning model. In Addition to that, the training and testing of the two

models lasted a couple of hours (since we Google colab was a bit slow)

2. The final result of the 32-dimensional latent space variational autoencoder was quite accurate and we managed to get a good result on both reconstructed and newly-generated digits. For the accuracy we used the mean of the reconstruction loss and the KL divergence loss.

3. From the specific dataset and method we learned that going as low as a 2-dimensinal latent space using only the given fully connected layers can result in a lot of information loss. On the other hand, if we have a 32-dimensional latent space, using this specific model architecture of a VAE, even though our model is fairly simple we still get to keep a lot of important information about the digits and in the end we get a quite good reconstruction and generation of digits

---

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

For this last task we are requested to work with the FireEvac dataset, training a VAE model on it and generating data. Unfortunately, we have not been able to carry out this task. For some reason that we do not understand, we have been not able to properly train a model. We have tried different configurations, for example, we have tried to keep the network of the previous task and train with a 32-dimensional latent space for 1000 epochs with and without changing the batch size and/or the learning rate, but no tried configuration sames to work. We always obtain a bad loss plots and a reconstruction with a single horizontal line.
