Report for exercise 4 from group A

Tasks addressed: 4
Authors: ALEX PASQUALI (03754113)
NICOLA GUGOLE (03753996)

Last compiled: 2021–12–23
Source code: `https://github.com/AlexPasqua/MLCMS-exercises`

The work on tasks was divided in the following way:

ALEX PASQUALI (03754113)	Task 1	50%
	Task 2	50%
	Task 3	50%
	Task 4	50%
NICOLA GUGOLE (03753996)	Task 1	50%
	Task 2	50%
	Task 3	50%
	Task 4	50%

Report on task 1, Principal component analysis

We implemented a "wrapper" function to compute the SVD decomposition that calls the library function `numpy.linalg.svd` [1]. In addition to that, it makes possible to center the data by simply passing a boolean argument and it returns the 3 matrices U, S and V (instead of U, list of the singular values and V transpose, as numpy's implementation does). The code of this function is depicted in Figure 1. To check the accuracy of the data reconstruction, we used mostly the percentage of energy captured by the principal components that were not set to zero (Eq. 1) and the norm of the difference between the original data and the reconstructed one (Eq. 2). When reconstructing the data completely, i.e. without setting any singular value to zero, the method was very precise, since the value of the metric of Eq. 2 was in the order of 10^{-15}.

$$\frac{1}{trace(S^2)} \sum_{i=1}^{L} \sigma_i^2 \tag{1}$$

$$\|X - USV^T\| \tag{2}$$

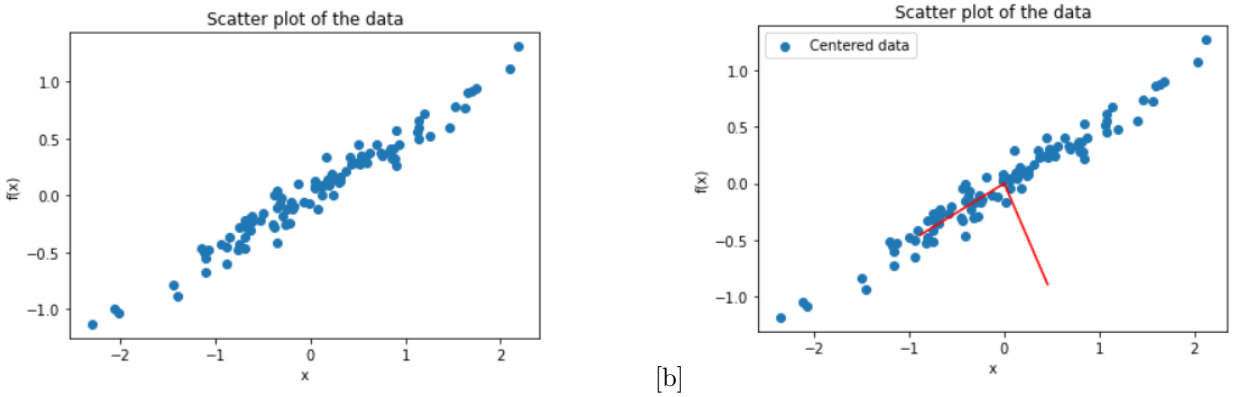
Overall, the method was fairly simple to implement and it only took few days to build and test it.

```python
def svd(data: Union[np.ndarray, pd.DataFrame], center=False):
    """
    Compute the Singular Value Decomposition (SVD) of the "data"
    :param data: data to compute the SVD of
    :param center: if True, center the data before performing SVD
    :returns: the 3 matrices forming the SVD decomposition of "data"
    """
    # make the data a numpy ndarray (if it isn't already)
    if isinstance(data, pd.DataFrame):
        data = data.to_numpy()

    # center the data by removing the mean
    if center:
        data = center_data(data)

    # decompose the data through SVD decomposition
    U, singular_values, Vt = np.linalg.svd(data)  # note that V is already transpose
    # starting from a vector containing the singular values, create the S matrix
    S = np.vstack((
        np.diag(singular_values),
        np.zeros(shape=(data.shape[0] - len(singular_values), len(singular_values)))
    ))
    return U, S, Vt.T
```

Figure 1: Code of the function implementing the SVD decomposition.

Part 1 The dataset `pca_dataset.txt` contains 2-dimensional data whose scatter plot is shown in Figure 2(a). When computing PCA, in the first principal component it is contained 92.33% of the energy, while the second one contains only 7.67% of the total energy. The directions of the 2 principal components are shown in red in Figure 2(b), which is in the original coordinate system. In order to plot the directions of the principal components with this coordinates system, we plotted 2 lines connecting 2 points: the mean point of the data (which is common to both lines) and the point identified by the first and second column of the matrix V (product of the SVD) [1]. The code of the function used for this purpose is shown in Figure 3.

[1] When computing the SVD transformation of the data contained in `pca_dataset.txt`, the results are the usual 3 matrices U, S and V. In this case the matrix V has dimensionality 2x2, therefore each of its columns can represent a point in a 2D plane.



[a] [b]

Figure 2: Scatter plot of the data contained in `pca_dataset.txt`. (b) shows the plot of the centered data with the directions of the 2 principal components.

```python
def plot_principal_directions(data: np.ndarray, V: np.ndarray):
    """
    Displays a 2D scatter plot of the data and adds the directions of the 2 principal components
    :param data: data to be plotted
    :param V: V matrix coming from a SVD decomposition of data, its columns contain the directions of the principal components
    """
    plt.scatter(data[:, 0], data[:, 1], label='Centered data')
    origin = np.mean(data, axis=0)
    x_values = [origin[0], V[:, 0][0]]
    y_values = [origin[1], V[:, 0][1]]
    plt.plot(x_values, y_values, color='r')
    x_values = [origin[0], V[:, 1][0]]
    y_values = [origin[1], V[:, 1][1]]
    plt.plot(x_values, y_values, color='r')
    plt.xlabel("x")
    plt.ylabel("f(x)")
    plt.title("Scatter plot of the data")
    plt.legend()
    plt.show()
```

Figure 3: Function used to plot the directions of the 2 principal components of the data contained in `pca_dataset.txt`.

Part 2 In this part it was required to apply PCA to an image of a racoon [2]. The image has been generated using `scipy.misc.face` [2], converted to gray-scale and reshaped to have size (249 x 185). The columns of the images are considered data points and we applied the PCA to it. Specifically, we tried to reconstruct the image using all, 120, 50 and 10 principal components and the results are shown in Figure 4.
With only 50 principal components it is already possible to see some slight differences between the original and reconstructed images, but only with 10 it becomes very evident. In fact, 120 principal components are enough to capture 99.2% of the energy (loss is less than 1%), 50 principal components capture 93.10% and, finally, 10 principal components only retain 75.76% of the total energy.
Recall that to calculate the percentage of retained energy by a certain number of principal components we used the formula in Eq. 1.

[2] Available at `https://pixnio.com/fauna-animals/raccoons/raccoon-procyon-lotor`

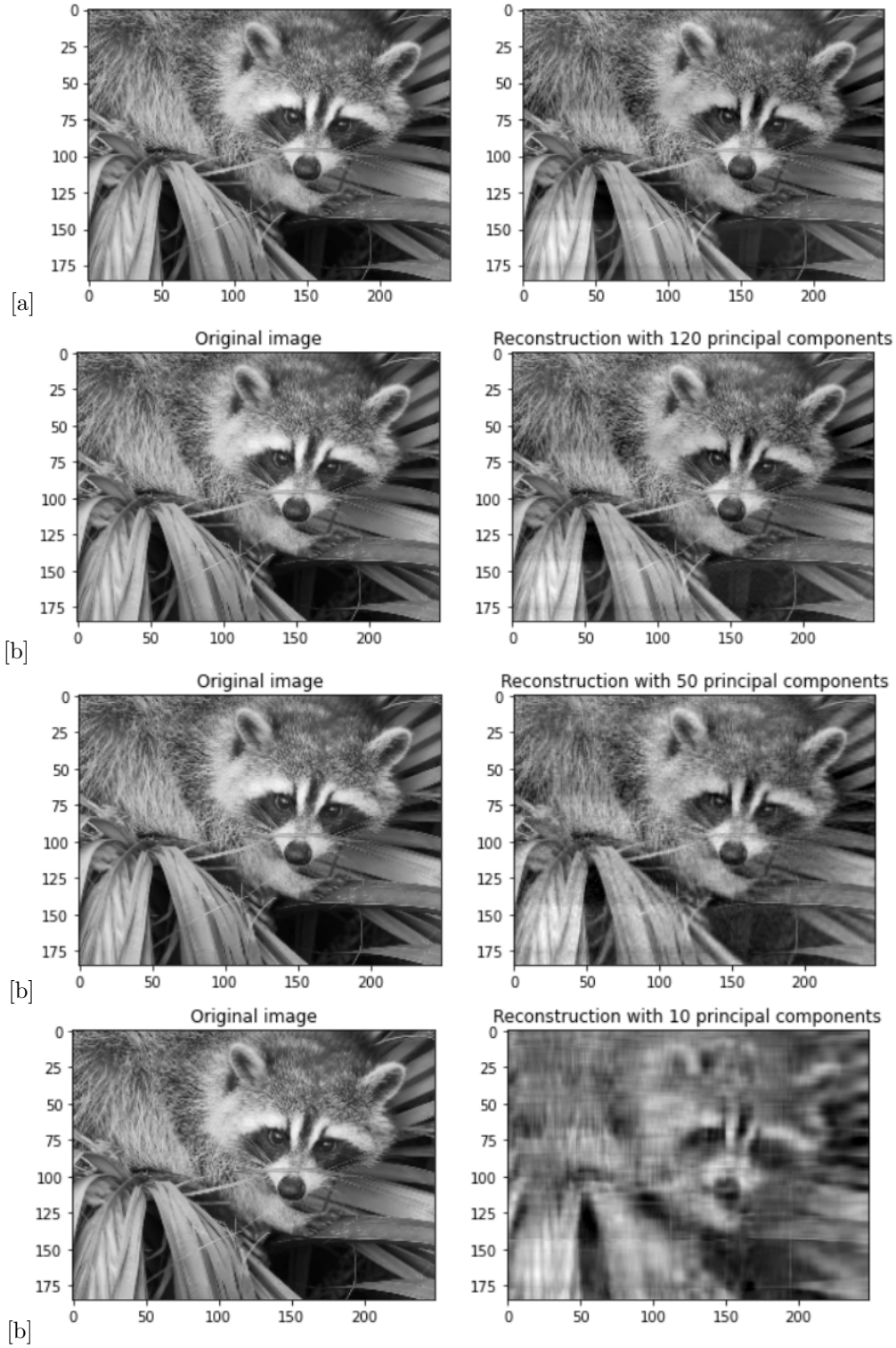


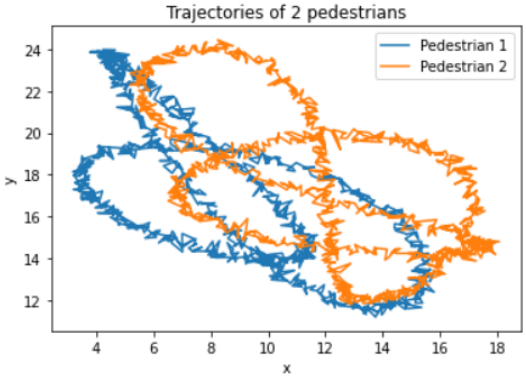
Figure 4: Different reconstructions of the same image using respectively (a) all, (b) 120, (c) 50, and (d) 10 principal components.

Part 3 This part was about applying PCA to data containing trajectories of various pedestrians in the 2D space. These can be observed in Figure 5 and it is possible to notice that they are very "fuzzy", probably indicating some amount of noise in the data.
Later, we projected the data into the first 2 principal components by computing the SVD decomposition, setting

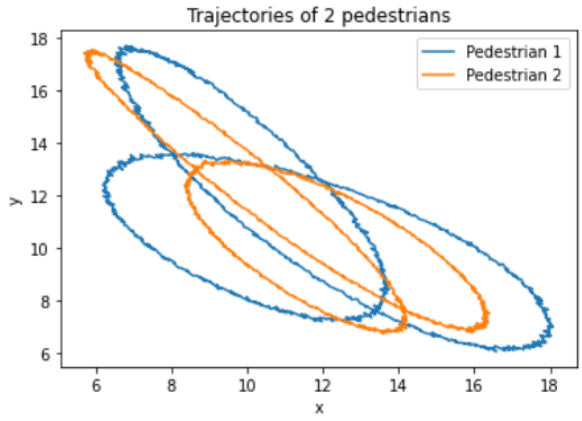
all the singular values after the second one (σ_2) to zero and reconstructing the projected data through Eq. 3

$$\tilde{X} = US_2V^T \tag{3}$$

where S_2 is the matrix S coming out of the SVD ($X = USV^T$) where only the first 2 singular values are not set to zero.
The first 2 principal components capture 84.92% of the energy, while to capture most of it ($\geq 90\%$) 3 principal components are necessary. With 3 principal components it is possible to retain 99.71% of the energy, which not only suffices to capture most of the dataset's energy (defined as $\geq 90\%$), but it produces an improvement of 14.79% compared to using 2 principal components. It is interesting to notice how the behaviour of the first pedestrian (blue) in **Figure 5(a)** is well represented by the first PCA (also blue) in **Figure 5(b)**.



[a]



[b]

Figure 5: (a) Trajectories of 2 pedestrians in the 2D space. (b) 2 principal components of the 30 dim space.

Report on task 2, Diffusion Maps

We implemented most utilities for plotting and computation for this task in a class (`DiffusionMap`), especially the `execute_algorithm` function, corresponding to the algorithm requested by the exercise sheet. The implementation was slightly more difficult to implement with respect to the PCA, with the implementation and testing window being three days due to subtle erroneous mathematical computations due to wrong use of `numpy` methods. For example we used `np.linalg.eig` instead of `np.linalg.eigh` to calculate eigenvalues and eigenvectors, even though at that point the matrix on which actuate the eigendecomposition is Hermitian.
To assure the accuracy of measurements, once the procedure proved to be correct, was ensured by the numerical stability of using `numpy`. The correctness of results was ensured by comparing results with a standard library as `DataFold` as well as using its measuring utility for backing up our assertions (e.g. using *LocalRegressionSelection* to enforce the correct selection of eigenfunctions for part three of the task.

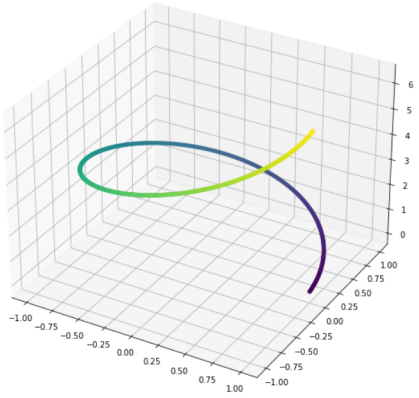


Figure 6: dataset of 1000 point, through $\texttt{get}_p art_o ne_d ataset$

Part 1 The dataset (**Figure 6**) composed of 1000 points is created through a simple implemented method, `get_part_one_dataset`, and the five (six for a better figure) eigenfunctions ϕ_l associated to the largest eigenvalues λ_l are shown in **Figure 7** against the t_k variable, as requested.

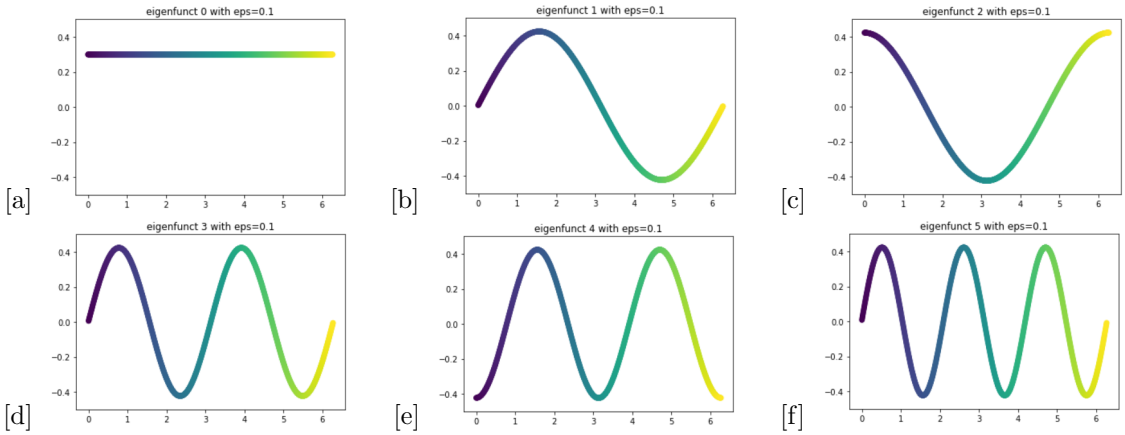


Figure 7: eigenfunctions relative to largest eigenvalues plotted against t_k

From the figure can be noticed that the most relevant eigenfunction, (a), is constant. This fact is present also in further figures and is remarked also in the exercise sheet. Another interesting fact is that all eigenfunctions have a sinusoidal behaviour in time, with a higher (f) or lower frequency (b), hinting at the correlation that exists between *Diffusion Maps* and *Fourier Analysis*. Both the methods do in this case in fact portray the original dataset through a combination of sinusoidal waves, with different coefficients and frequency, represented differently in Fourier and in the DM.

Part 2 This part of the task focused on using the algorithm to obtain the first ten eigenfunctions of the *Laplace Beltrami* operator on the *swiss roll* manifold. In particular, using `SkLearn`, the dataset was created with ease, composed of 5000 non-noisy samples as shown in **Figure 9**.

The computation of the algorithm retrieves the ten desired eigenfunctions, which are shown (plotted against the first non constant eigenfunction ϕ_1) in **Figure 8**, on which a brief discussion can be applied. First of all it can once again be noticed how ϕ_0 is constant, as when plotted against ϕ_1 it delivers a line. Secondly, many other eigenfunctions look to be related to ϕ_1 as they are in fact functions of it. Examples are ϕ_2, ϕ_3 and ϕ_4. On the contrary the first ϕ to deliver a non-function behaviour when plotted agains ϕ_1 is ϕ_5. The pairing of ϕ_1 and ϕ_5 is a good example (and the first $l > 1$ to deliver that!) for a *functional independence*. Finding functional independence in such a *manifold learning model* is fundamental to have non poor embeddings. In fact *functional dependence* means that the first eigenvector ϕ_x does not go along a new and independent direction compared to eigenvector ϕ_y, not giving as much information as could be possibly given in the same dimensional subspace but with a different pair. We further ensured the possibility for (ϕ_1,ϕ_5) being a good unfolding is confirmed visually by the figure and mathematically through the use of *LocalRegressionSelection*, a technique available for the selection of most relevant couples, triples, etc (you decide the dimensionality!) of eigenfunctions in data (**Figure 14**).

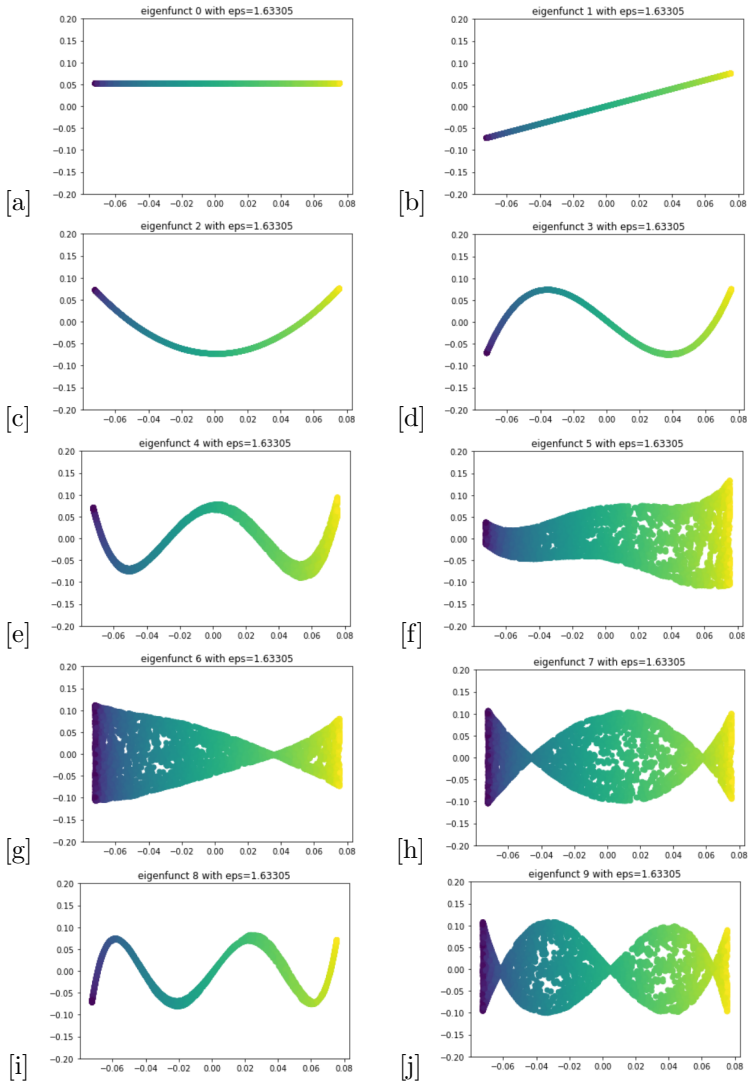


Figure 8: plots where the first non costant eigenfunction (ϕ_1) is plotted against the rest (5000 samples)

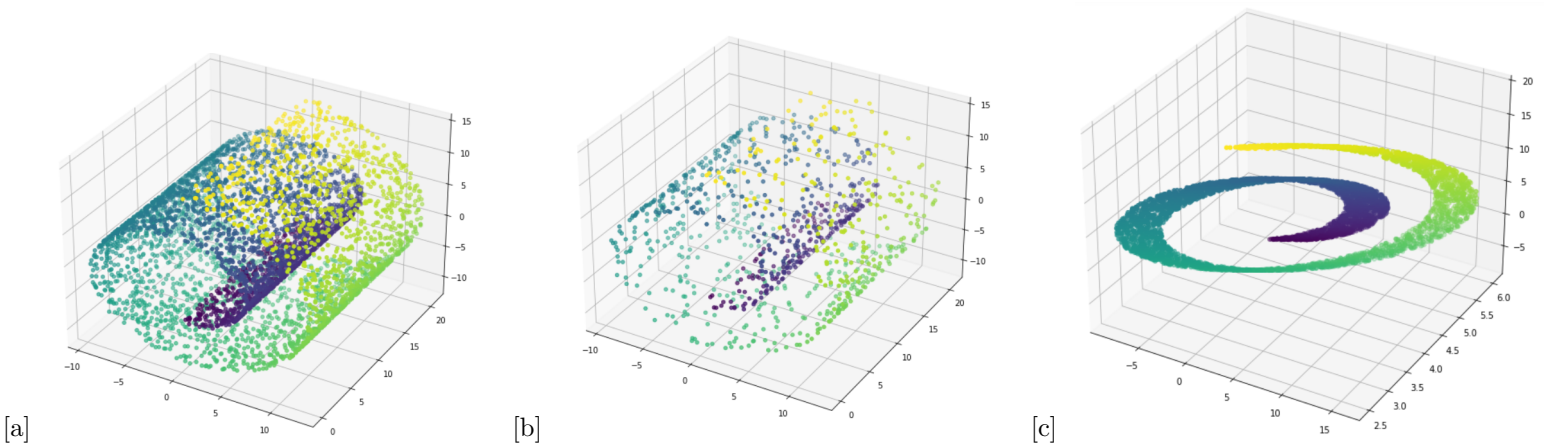
The task proceeds asking to compute the three principal components of the swiss-roll dataset, which energy result to be, taking away one PC at a time from left to right:

(1.0, 0.7104704462758858, 0.3933604936493569)

This already hints at answering the question: *Why is it impossible to only use two principal components to represent the data?* As one can appreciate both by the *energy value* of using only the first two components as well as looking at the reconstruction proposed by the first two components (**Figure 9(c)**), the loss in energy (and therefore data variance) due to not using the third principal component is unbearable for the original data, which looses too much information.

Another interesting proposed experiment involves the number of samples, reducing it from 5000 to 1000 leads to the dataset proposed in **Figure 9(b)**. This new dataset is for sure less representative of the manifold with respect to the richer dataset proposed before. The effect of this sparseness travels to the produced *Diffusion Maps*, as can be appreciated in **Figure 10**, where the eigenfunctions become much more noisier with respect to the more populated case.

Part 3 The final part of this task goes back to a dataset already utilized in **Task 1**, regarding walking pedestrians (**Figure 5**). With *Diffusion Maps* it is not possible to use the same concept of energy as previously used with PCA, therefore it helps to first go through a visual route hoping to find a good combination of



[a] [b] [c]

Figure 9: swiss roll dataset sampled through `sklearn` (5000 samples - 1000 samples - only 2 PCA)

eigenfunctions. But what is a good combination of eigenfunctions? It is a set of eigenfunctions leading to a subspace where the data is unfolded, meaning there are **no intersections**. We started taking a look at 3D subspaces with three consecutive eigenfunctions, more for fun than for science rigorosity.

The result shown in **Figure 11** shows that in the 3D space it is possible to fully unfold the data (for example look at the subspace defined by ϕ_1, ϕ_2, ϕ_3) but that no all combinations are unfolding the data (for example ϕ_2, ϕ_3, ϕ_4). An interesting triple is the first one, where the usual ϕ_0 being constant shows how there can possibly be a 2D subspace already showing a good combination of eigenfunctions.

Eventually the test was moved to searching in 2D subspaces, as shown in **Figure 12**. The hint given by the 3D plots was confirmed, showing how the subspace produced by ϕ_1 and ϕ_2 has no intersections and is therefore a *proper embedding*. Knowing that the 2D space can have a good embedding led us to confirm our findings by utilizing again *LocalRegressionSelection*, which as can be seen in **Figure 13**, retrieves the same results.

```
selection = LocalRegressionSelection(
    intrinsic_dim=2, n_subsample=500, strategy="dim"
).fit(dmap_2.eigenvectors_)
print(f"Found parsimonious eigenvectors (indices): {selection.evec_indices_}")
selection.residuals_
```

```
Found parsimonious eigenvectors (indices): [1 2]

array([       nan, 1.        , 0.98243244, 0.2117499 , 0.31226566,
       0.23934658, 0.31486124, 0.2431183 , 0.2823897 ])
```

Figure 13: code showing a possible selection modality

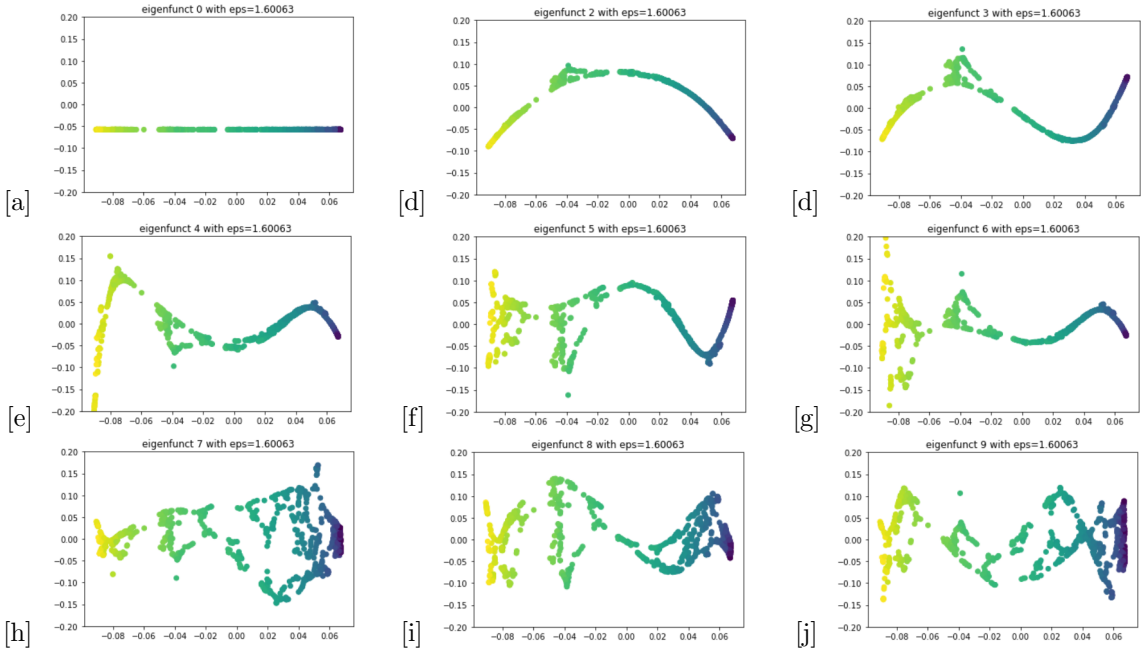


Figure 10: plots where the first non costant eigenvalue (ϕ_1) is plotted against the rest (1000 samples)

Bonus After having downloaded *DataFold* library on Windows, we followed the tutorial already present for the *s-curve manifold* and changed the necessary bits to apply the same snippet on the *swiss-roll manifold*. The code proved to be very easy and intuitive as well as being optimized and therefore reasonably fast. Much compact and we would for sure recommend it for this case scenario. Plotting the *swiss-roll* eigenfunctions brings the same results produced by our implementation, as can be appreciated in **Figure 15**. The incredibly useful part of having delved into this library is finding how actually is possible to understand which are the most relevant eigenfunctions for a good reduction and therefore a good embedding. In fact, having a 2D, 3D or 4D space makes it still reasonable to find good eigenfunction combination with our bare eyes, but in higher space the situation is much more complicated. A possible solution to that is given by the library, utilizing (mentioning here for the third time!) *LocalRegressionSelection*, which gives the relevance of each eigenfunction given a precise dimensionality reduction (**Figure 14**).

```
selection = LocalRegressionSelection(
    intrinsic_dim=2, n_subsample=500, strategy="dim"
).fit(dmap.eigenvectors_)
print(f"Found parsimonious eigenvectors (indices): {selection.evec_indices_}")
```

```
Found parsimonious eigenvectors (indices): [1 5]
```

Figure 14: code showing an automatic selection of the best choice pair ϕ_l (with a given dimension)

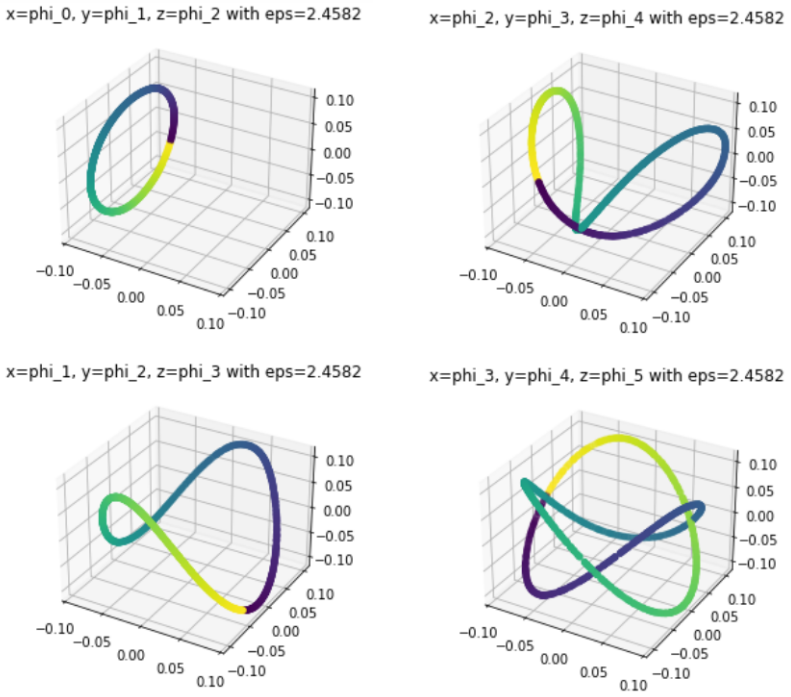


Figure 11: example of 3 consecutive eigenfunctions (showing possibility of no intersection)

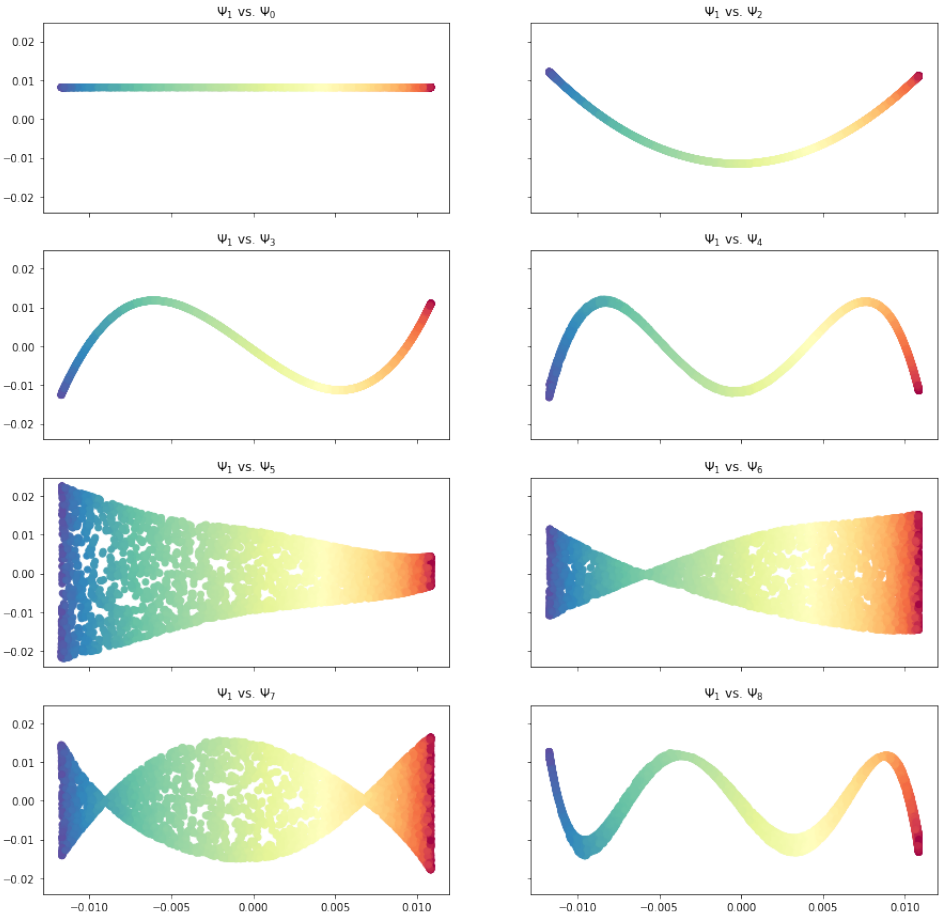


Figure 15: ϕ_1 against the other eigenfunctions using datafold - swiss roll

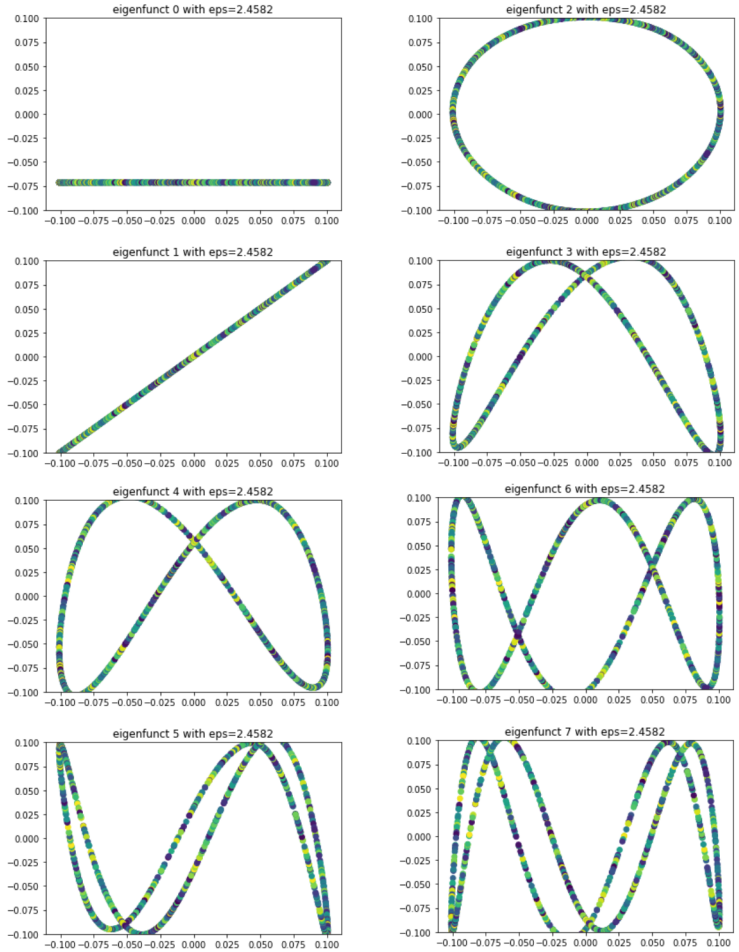


Figure 12: 2D latent space of eigenfunctions, first one is always ϕ_1 (showing possibility of no intersection in 2D)

Report on task 3, Variational Autoencoders

This task proved to be the most time consuming due to library set up issues, with the implementation and testing taking six to seven days. The implementation started by attempting to implement the code proposed by the tutorial mentioned in the exercise sheet. A first problem came up with the `TensorFlow` version required, since our laptops had `TensorFlow 2.7.0` installed. After setting up a virtual environment with the version 1.15 (which also requires a Python version below 3.8, leading to other setting up annoying situations), we started playing with the model, seeing the functioning of it. We were not anyway fully satisfied with this implementation, therefore, being masochists, we moved to `TensorFlow 2`, a useful experience to understand some of the main differences between the two major releases. Eventually we decided to go to `Pytorch` (version 1.9.0 with Python 3.8.6), where we remained to implement our final solution.
After the rollercoaster of libraries we decided to implement the model in a class (`VAE`) subclassing `nn.Module`, containing some fundamental methods and the required globally trainable variable defining the reconstruction *standard deviation*:

- `__init__`: initializer, allows to give the input data dimensionality, as well as the feature space dimensionality (number of neurons in hidden layers) and finally the latent space dimensionality (both for the *mean* and the *scale*). Batch size is always excluded from these dimensions. An additional parameter has been added (`use_BCE_loss`) to allow for a quick change in behaviour of last decoder activation function, switching from *sigmoid* to *linear*. This will be further discussed later.
- `reparameterize`: applies the reparameterization trick to allow backprop to flow to encoder part of the model

- `encode`: applies feedforward to the encoder model, getting as input the data from the dataloader and outputting the *mean* and *log variance* of the latent space distribution. We decided to output the *log variance* for convenience in the *KLDivergence* loss. We anyway reverted it back to the *standard deviation* where needed (e.g. in the reparameterization trick)
- `decode`: applies the feedforward to the decoder model, getting as input data sample batch from the latent space and outputting the *mean* of the output distribution
- `forward`: applies the overall feedforward, calling the encode and decode method internally, outputting both the means and the log variance (only the one coming from the latent space)
- `generate_many`: utility function to call decode multiple times after having sampled data from the latent space the same number of times
- `log_var_rec`: an `nn.Parameter`, implementing the globally trainable *standard deviation* for the reconstruction distribution. Actually, once again, we are treating this variable as the *log variance* for the alternative loss (will be introduced later) convenience. This does not hinder the possibility of retrieving the *standard deviation* where needed by applying some exponential transformation.

The other utilities needed for training and plotting have been relegated to `vae_utils.py`, which contains in particular utilities for:

- fitting the model
- testing the model
- calculating the ELBO loss
- plotting/saving the **15** reconstructed digits with respect to the input data
- plotting/saving the produced latent space over the test dataset
- plotting/saving the produced latent space over the test dataset, but using digits and not scatter (see Figureasd)
- plotting/saving the **15** generated digits sampled from the latent space

A peculiarity that can be found is that we decided to propose two alternatives in the training, varying the *reconstructed loss*. This was also the reason for several time spent over implementation and **research**, as having the mean as output is not taken in consideration by a large part of the implementations out there. Going more in detail:

- the *standard* training/testing method: we implemented the reconstruction loss utilizing `BCELoss`, a Cross Entropy solution which has as general objective finding the difference between two probability distributions, in our case the output distribution and the input distribution. The reconstruction loss is therefore computed by the library and is added to the KLD.
- the *alternative* training/testing method: we decided to also implement a loss which could take advantage of the globally trained log variance as well as being more faithful to the ELBO definition. We therefore mathematically hardcoded the definition of *log likelihood* of a sample (input data in our case) given the output distribution (which is supposed to be a normal distribution).

```python
def elbo_loss_alternative(mu_rec, logvar_rec, mu_latent, logvar_latent, orig_x, kl_weight=1):
    """
    An alternative which tries to achieve the same underlying goal, in a more mathematical and hands-one manner
    This function will add the reconstruction loss (log likelihood of output distribution for the input data) and the
    KL-Divergence.
    KL-Divergence = 0.5 * sum(1 + log(sigma^2) - mu^2 - sigma^2)
    Reconstruction Loss =  - loglikelihood over normal distribution given a sample (orig_x)
        = sum(0.5 * log(variance) + (sample - mu)^2 / (2 * variance^2) + log(2pi))
    :param mu_rec: decoder output, mean of reconstruction
    :param logvar_rec: globally trainable parameter, log_variance of reconstruction
    :param mu_latent: the mean from the latent vector
    :param logvar_latent: log variance from the latent vector
    :param orig_x: input data
    :param kl_weight: possible variant, weight for the KLD so to give an arbitrary weight to that part of loss
    """
    logvar_rec = logvar_rec * torch.ones_like(mu_rec)  # replicate scalar logvar_rec to create diagonal shaped as mu_rec
    REC_LOSS = torch.sum(0.5 * (logvar_rec + (orig_x - mu_rec) ** 2 / (torch.exp(logvar_rec)) + math.log(2 * np.pi)))
    KLD = kl_weight * (-0.5 * torch.sum(1 + logvar_latent - mu_latent.pow(2) - logvar_latent.exp()))
    return KLD, REC_LOSS, REC_LOSS + KLD
```

Figure 16: elbo loss utilized for the *alternative* implementation

Changing from *standard* to *alternative* can be simply achieved by swapping between using methods *fit/fit_alternative* and *test/test_alternative*. After this brief discussion the usefulness of the aforementioned `use_BCE_loss` parameter might become clearer. The *standard* implementation needs a sigmoid as last decoding activation function since the output is expected to be between 0 and 1, while the *alternative* does not. That parameter easily allows for the presence or absence of said activation function.

Part 1 The first part of the task asks for two reasoning questions:

- *What activation functions should be used for the mean and standard deviation of the approximate posterior and the likelihood—and why?*: since the probability distributions we want to follow are multivariate gaussian distribution we should avoid *mean* and *standard deviation* values which are not valid for modelling such a distribution. A multivariate gaussian distribution is in particular parametrized with a *real mean* and a *positive standard deviation* in matrix form (positive semi-definite matrix in particular). Therefore there should be no constraint on the *mean* as it can take whatever real value. Regarding the *standard deviation* instead, since it has to be formed of positive values, there should be some care in utilizing a non-negative (but with no upper limit) activation function, such as ReLU or Softplus. As a little parenthesis, in our case this problem is not present, as utilizing the *log variance* leads to the possibility of having negative values (so no need for activation functions) while the constraint of a positive *standard deviation* is maintained per se.

- *What might be the reason if we obtain good reconstructed but bad generated digits?*: having good generated digits all depend on having a good latent space. If the latent space discriminates well between classes and is well distributed (while not being too sparse) then the results will be fantastic, with coherence in near points. Problems and therefore bad generations can appear for a number of reasons in VAE. First of all it might be the case that the net is *not powerful enough*, but this would also mean that not only the generated but also the reconstructed images are not the best. Secondly, there could be an *imbalance* between the two addends of the KL Divergence, leading to other approaches such as β-VAE where the two addends get weighted. If KL Divergence is irrelevant then the VAE is pretty much a basic AE, with more effort in learning a mapping than an actual space reduction. Lastly, the problem could be related to having a *too high dimensional latent space*. That is also the case in our scenario when going from 2 to 32 dimensions in the latent space. In fact if your latent space is too big, it can lead to generating out-of-distribution samples because the latent space gets incredibly sparse. This can lead to seeing sharp blobs of white and black, or maybe seeing the same digit over and over. If we were to take this concept to the extreme we could actually reach an *overfitted latent space*, where the dimensionality of the latent space is equal to the number of training data, with each training data being mapped in the latent space to a different and orthogonal dimension.

Part 2 This second part is centered on showing the results of training a VAE model with some given hyper-parameters on the MNIST dataset. The dataset is normalized between 0 and 1, dividing each cell by 255 (the image is a single channel grayscale). We are going to first show the results on the *standard* implementation to then move to the *alternative*.

Figure 17 reports the model implementation of the most important methods:

```python
def forward(self, x):
    """
        feedforward
        :param x: input data
        :return: mean of reconstructed data,
        mean of latent space,
        log_var of latent space
    """
    mu_latent, log_var_latent = self.encode(x)
    # get the latent vector through reparameterization
    z = self.reparameterize(mu_latent, log_var_latent)
    # decoding
    mu_rec = self.decode(z)
    return mu_rec, mu_latent, log_var_latent
```

```python
def encode(self, x):
    """
    encoding feedforward
    :param x: input data
    :return: mean of latent space, log_var of latent space
    """
    x = F.relu(self.enc1(x))
    x = F.relu(self.enc2(x))
    x = self.enc_out(x).view(-1, 2, self.latent_dim)
    # get `mu` and `log_var`
    mu_latent = x[:, 0, :]  # the first feature values as mean
    log_var_latent = x[:, 1, :]  # the other feature values as variance
    return mu_latent, log_var_latent
```

```python
def decode(self, z):
    """
    decoding feedforward
    :param z: sampled value from latent space distribution
    :return: mean of reconstructed data
    """
    # decoding feedforward
    x = F.relu(self.dec1(z))
    x = F.relu(self.dec2(x))
    if self.use_BCE_loss:
        mu_rec = F.sigmoid(self.dec_out(x))  # apply sigmoid if using BCELoss otherwise not needed
    else:
        mu_rec = self.dec_out(x)
    return mu_rec
```

Figure 17: main model methods - from top to bottom *forward*, *encode*, *decode*

Standard results for loss and the required plots about latent space, reconstruction and generation are shown in **Figure 18** (losses), **Figure 3** (results 2D latent space) and **Figure 1** (results 32D latent space). Analyzing the images some interesting discussion can be brought up. In the loss figure it is possible to see that both in the case of 2D and 32D latent space, as the reconstruction loss gets lower and lower (the model is learning well how to reconstruct the input images), the KL Divergence rises to act as a *regularizer*. This gives the desired effect, with the model being slightly less perfect in reconstructing if put in comparison with a normal *AutoEncoder* to fulfill the objective of creating a meaningful latent space.

Continuing the analysis on the 2D case, **Figure 3** shows the results divided by epoch. In particular it is possible to appreciate the evolution of the latent space, which in the end is a 2D distribution that maintains coherence between nearby points, effectively dividing the classification in space regions. Taking instead a look at the reconstruction it is possible to notice how the data (first row of each plot) gets reconstructed better and better as epochs go on, as is also deductible by the decreasing *loss*. The VAE model gets better at reconstructing while also constructing a relevant latent space, from which generated samples become more and more trustworthy, as can be noticed in the same figure. The generated images look a little blurry but for sure readable.

Comparing the just discussed results with the results portrayed in **Figure 3**, one can reassert what already expressed in **Part 1** of this task. The reconstructed images do in fact look much better, appearing almost identical to the original data by the end of the training. This happens because the latent space is much bigger than before, allowing for a clearer internal representation of the data, nevertheless hindering the generation capability. The sharpness in reconstructing is in fact transferred to the generation capability, where the generated images are much sharper but less understandable. One can in fact appreciate in the last pictured epochs how some numbers are not at all understandable, probably due once again to the higher dimensional latent space which leads to a sparser data representation. It is interesting to notice how the KL Divergence grows much higher with respect to the 2D case, trying to regularize a bigger latent space, effectively creating a bigger gap

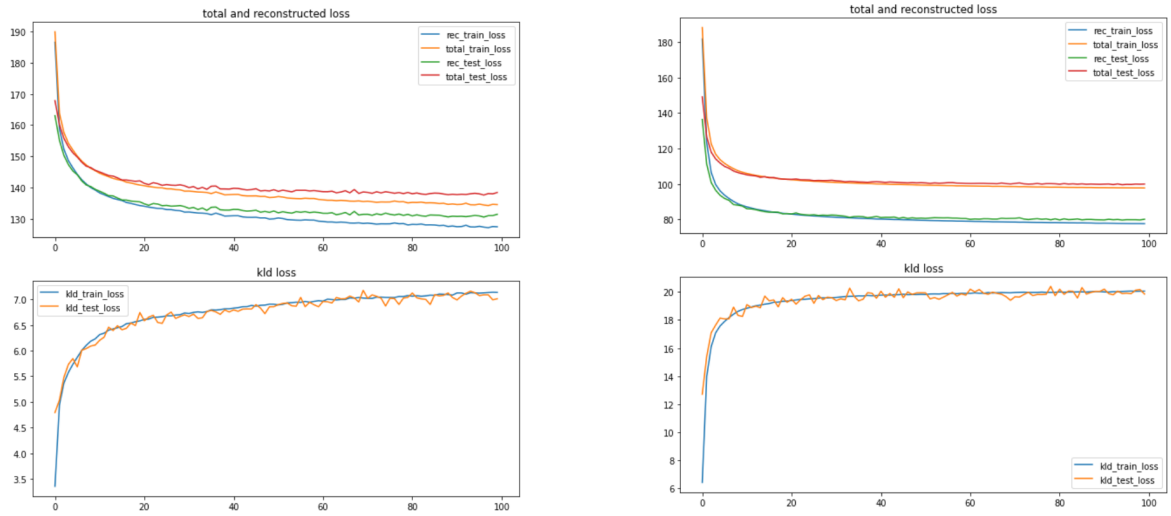


Figure 18: total loss and sublosses (KLD and REC) for both the *standard* 2d (left) and 32d (right) latent space

between the *total loss* and the *reconstruction loss*.

Briefly moving on to the *alternative* implementation, the results for loss and the required plots about latent space, reconstruction and generation are shown in **Figure 19** (losses), **Figure 4** (results 2D latent space) and **Figure 2** (results 32D latent space). The loss behaviour is similar to the models using *Cross Entropy*, with a little slower loss/KL Divergence descent/ascent. One might notice that the loss reaches a negative value, a peculiar behaviour for sure. This can be explained by the reconstruction distribution: we monitored the change in value of the globally trainable *log variance* and noticed how it decreased until reaching a negative value (which is possible), meaning the standard deviation of the reconstructed distribution stands between 0 and 1. This leads to a narrow multivariate distribution, for which the log likelihood can possibly give a negative value. The model proved nevertheless to improve its capabilities and could continue to learn, further lowering the loss.

Another interesting difference between *standard* and *alternative* can be instead noticed in the *latent space* construction. Apart from having a rougher start compared to the *standard* implementation, the *alternative* tends to reach a *zero centered* distribution, distinctively different from the other created *latent space* (the experiment has been repeated multiple times, the behaviour stays the same). Finally, moving to the generated and reconstructed images, there is no noticeable difference in the 2D aspect, while a relevant decrease in generation capability can be appreciated in the 32D case if compared to the analogous with the *standard* implementation.

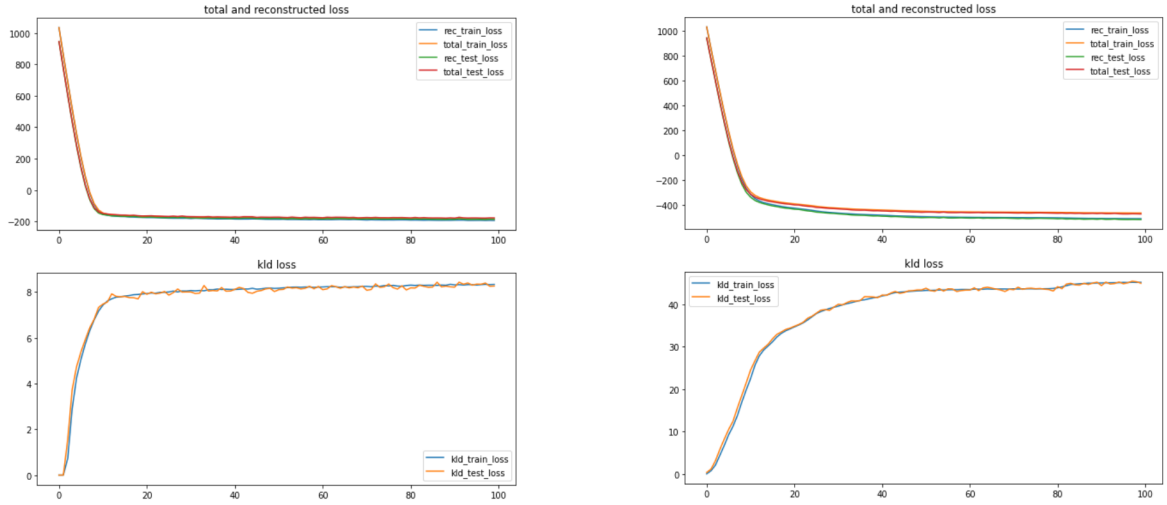


Figure 19: total loss and sublosses (KLD and REC) for both the *alternative* 2d (left) and 32d (right) latent space

---

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

The last task asks for the use of Machine Learning for the analysis of people distribution in the MI Building in the case of need for a fire evacuation. In particular it is interesting to study where people density gets high, so to plan ahead and possibly give different escape routes to different groups. We had little time remaining for this last task, so it took us only a day but at the same time we did not get the results we wanted. Through this task we learned that even little dimensional input datasets can prove to be hard to tackle.

The initial part of the task asks to download the dataset (which is in `.npy` format). The scatter of both the train and test set can be appreciated in **Figure 19**.

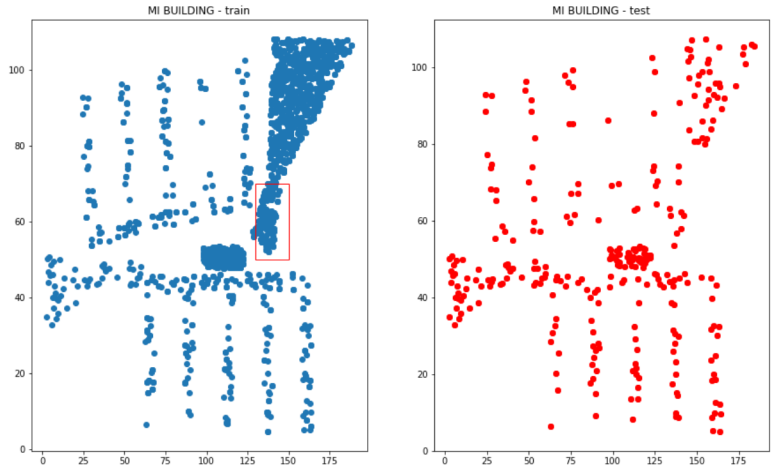


Figure 20: MI Building dataset train (*left*) and test (*right*), relevant region is the *red rectangle*

The training on this data proved that our model is insufficient since it is limited to efficiently learning data distribution that lies between 0 and 1. Unfortunately this dataset proved very sensitive to rescaling, with the suggested $(-1, 1)$ being the only scaling to lead to consistent training. We achieved the scaling with `SkLearn` method `MinMaxScaler`, so we could both transform and transform back (to see the reconstruction in the original space) the data. To train the data we changed the input and ouput neurons to being 2 instead of 748.

After trying various changes in activation function and intermediate dimensions the dataset proved to be difficult to fit. This fact, combined with our model suboptimality in representing output distributions, leads to

results which are not the best. The final model we utilized has the following hyperparameters:

- *learning rate*: 0.001
- *batch size*: 64
- *epochs*: 100
- *latent dimensionality*: 10
- *intermediate dimensionality*: 512
- *number of hidden layers*: 2 for the encoder, 2 for the decoder

The resulting model gave the loss plot proposed in **Figure 21**, highly unstable as one can appreciate, showing probably the need for a lower learning rate (which we tried!) or a more thorough hyperparameter search (which we did not do due to time constraints, we simply sampled some configurations).

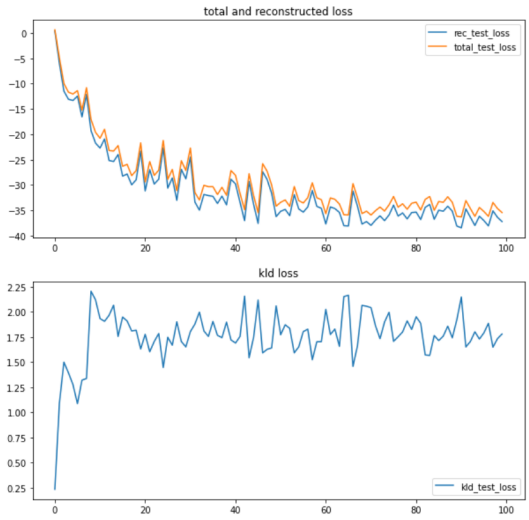


Figure 21: loss for VAE model trained on the MI Building dataset

The resulting reconstruction can be seen in **Figure 22**, showing that the model has only learnt the entrance behaviour, without being able to capture more subtler events such as the corridors in the building and not even a big event such as the bar!

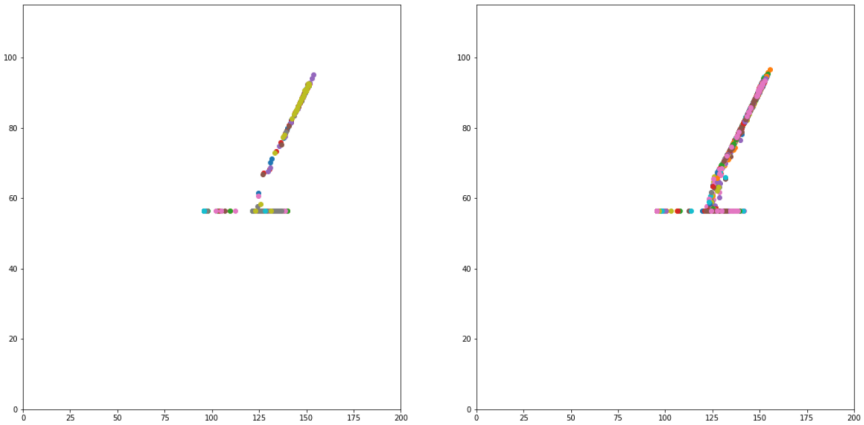


Figure 22: reconstruction after training on MI Building dataset - test (left) train (right)

Lastly, to answer the last question, a continuous generation showed that the critical number for the MI building as predicted for our model is pretty little: **∼1000**. An example of generating 1000 people with the number of people in the red box being underneath the figure is shown in **Figure 23**

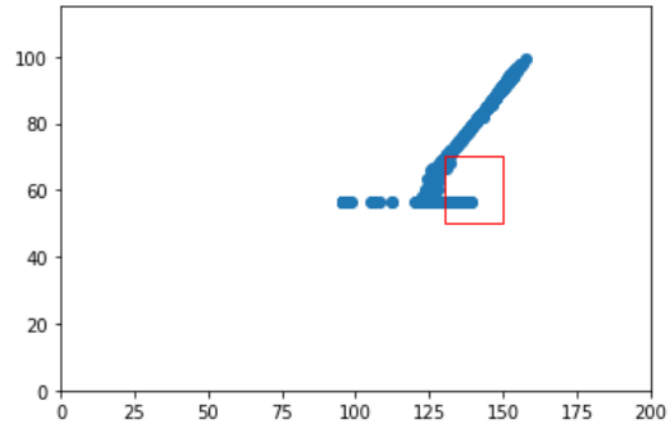


Figure 23: generation of 1000 people

References

[1] Numpy's svd implementation. `https://numpy.org/doc/stable/reference/generated/numpy.linalg.svd.html`.

[2] Scipy's `face` function. `https://docs.scipy.org/doc/scipy/reference/generated/scipy.misc.face.htmll`.

1 Figures

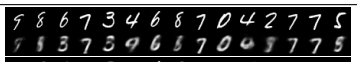
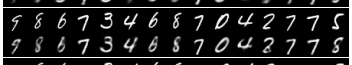
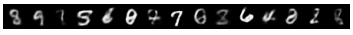
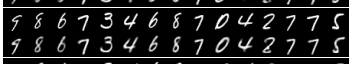
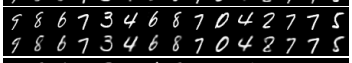
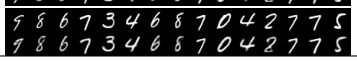
Epoch num	15 reconstructed	15 generated
Epoch 1		
Epoch 5		
Epoch 25		
Epoch 50		
Epoch 100		

Table 1: results for *standard* implementation - **32D latent space**

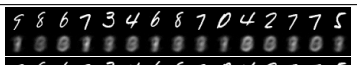
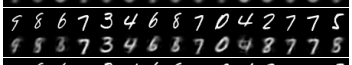
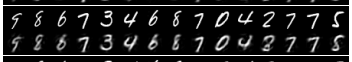
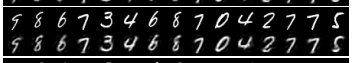
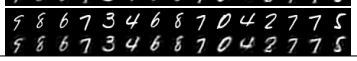
Epoch num	15 reconstructed	15 generated
Epoch 1		
Epoch 5		
Epoch 25		
Epoch 50		
Epoch 100		

Table 2: results for *alternative* implementation - **32D latent space**

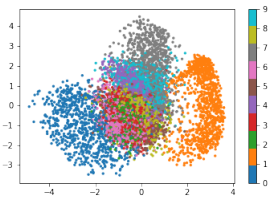
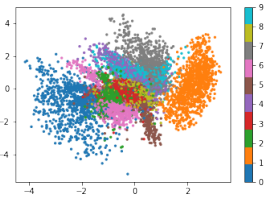
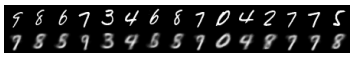
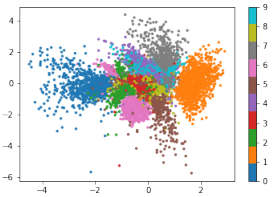
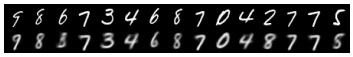
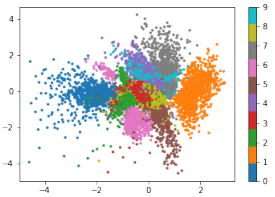
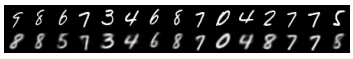
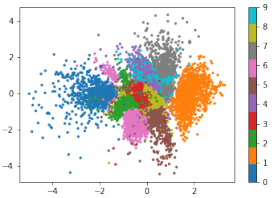
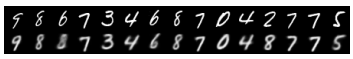
Epoch num	latent space	15 reconstructed	15 generated
Epoch 1			
Epoch 5			
Epoch 25			
Epoch 50			
Epoch 100			

Table 3: results for *standard* implementation - **2D latent space**

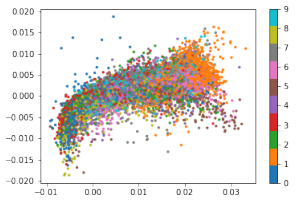
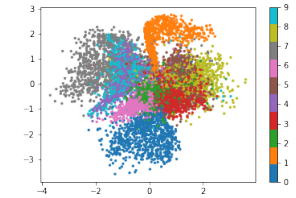
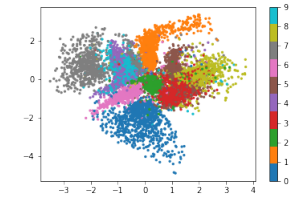
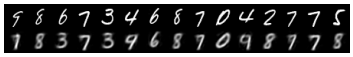
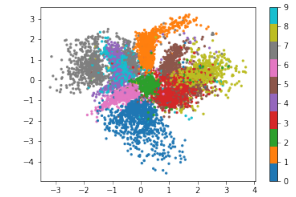
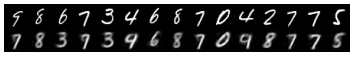
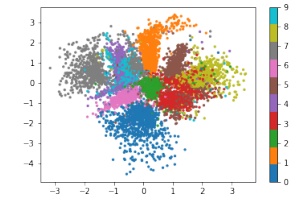
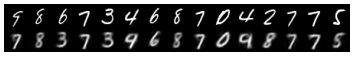
Epoch num	latent space	15 reconstructed	15 generated
Epoch 1			
Epoch 5			
Epoch 25			
Epoch 50			
Epoch 100			

Table 4: results for *alternative* implementation - **2D latent space**