Report for exercise 4 from group A

Tasks addressed:	4
Authors:	MELIH MERT AKSOY (03716847) ATAMERT RAHMA (03711801) ARDA YAZGAN (03710782)
Last compiled:	2022–06–22
Source code:	https://github.com/mlcms-SS22-Group-A/mlcms-SS22-Group-A/tree/exercise-4

The work on tasks was divided in the following way:

MELIH MERT AKSOY (03716847)	Task 1	33%
	Task 2	33%
	Task 3	33%
	Task 4	33%
ATAMERT RAHMA (03711801)	Task 1	33%
	Task 2	33%
	Task 3	33%
	Task 4	33%
ARDA YAZGAN (03710782)	Task 1	33%
	Task 2	33%
	Task 3	33%
	Task 4	33%

numpy	1.20.3
matplotlib	3.4.3
tensorflow	2.9.0
tensorflow_probability	0.17.0
scipy	1.7.1
skimage	0.18.3
pandas	1.3.4

Table 1: Software versions.

Software that is used in this report with corresponding versions can be seen on table 1.

Report on task Task 1/4: Principal component analysis

1. It took us approximately 3 days to implement and test the methods for Principal Component Analysis.
2. For accuracy measurement we have used the energy equation given in the exercise sheet.
3. PCA method is a dimensionality reduction method where only the specified number of principal components are kept that captures the most of the variance in the data. For example in PCA dataset the second principal component marked with red only captures a small part of the energy therefore truncated. About the racoon image we can see that still 50 components the image is good identifiable and therefore most of the dimensions of the image can be eliminated for example for compression reasons. The pedestrians trajectories cannot be represented well with only 2 components, however the loss with 3 principal components is very low.

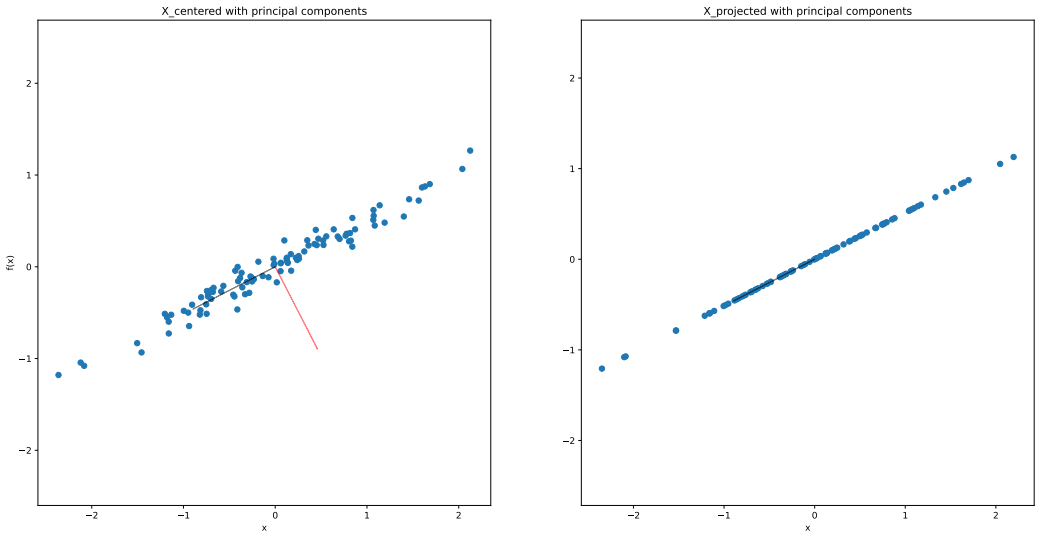


Figure 1: PCA dataset with its princpial components and the reduced dataset.

In the first part of this task we were supposed to implement principal component analysis, where we have used **numpy** library for singular value decomposition (SVD). We have than approximated the one-dimensional, linear subspace of the two dimensional data set **pca_dataset.txt** on the Moodle website that is optimal in the sense of variance reduction. In Figure 1 we have depicted the data set as shown in the exercise sheet on the left sub-figure, where we have also included direction of the two principal components starting from the center of the data set (one colored with red and other with black). On the right sub-figure one can see the projected data-set points using the principal components calculated before.

We have computed the energy kept in each component with get_contained_energy_in_each_component() method. This method uses the following equation given in the exercise sheet:

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

The first principal component marked with black is the one that has the most of the energy of the dataset.The computed energy of the first principal component is approximately 99.314 while the energy of the second principal component, marked with red, has 0.686 namely the rest of the energy.
In the second part of the first task we had to apply principal component analysis to the given racoon image. To import the image we used to scipy library. Thanks to this library we get the racoon image in shape (768, 1024). As described in the exercise sheet, we have set the gray variable true to get the gray scaled image to be able to start the analysis. The gray scale racoon image looks as follows: To resize the image to have the

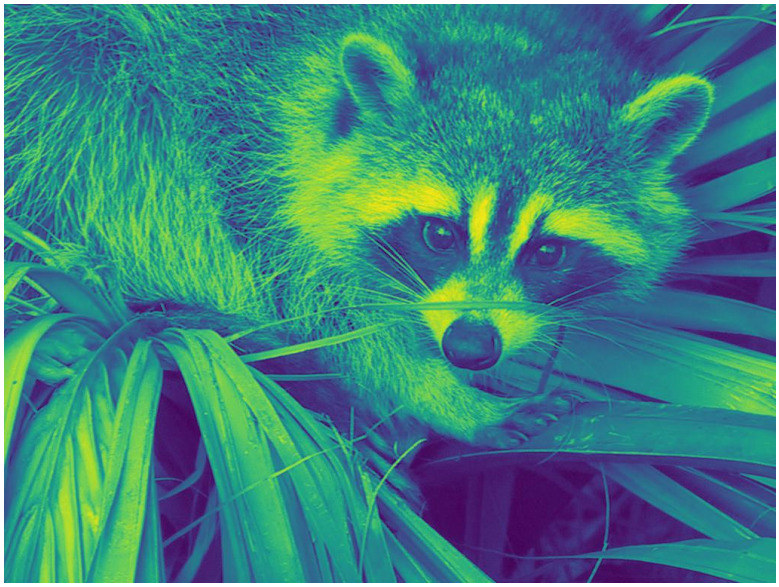


Figure 2: Gray-scale racoon image

shape (249 185) we used the skimages resize function. After we resized the racoon image, we started with the analysis.
First we have centered the data points and extracted the size of the feature dimension, in this case the width of the image, to reconstruct the image with all of its components.
To reconstruct the image we have used the given following equation:

$$\hat{X} = U\hat{S}V^T$$

These components, namely U S and V are obtained via the SVD library of numpy. After that to truncate the number of principal components the values on the diagonal in S matrix are set to zero. To be more precise, the highest L singular values are kept to respect the given number of L components. This results in $\hat{S}$.
We have reconstructed racoon image with all, 120, 50, 10 principal components. The results are as follows :

To determine the number of maximum principal components at which the energy loss is smaller than 1%, we have measured the energy loss starting at the 50 principal components. When we reach the 77 principal

Figure 3: Racoon image reconstructed with all components.

Figure 4: Racoon image reconstructed with 120 components.

Figure 5: Racoon image reconstructed with 50 components.

components, the loss goes under 1%.
All visualised images with 120 and 50 components are still good identifiable, but with 10 components the information loss is very obvious. The percentage of the loss there is also relatively high, namely 17.61%.
The third part deals with a data set, in which the trajectories of pedestrians are given. When we visualize the trajectories of first two pedestrians we can see that, pedestrians move in a circular way making zigzags as shown in the Figure 8. Before we begin with analysis, we first center the data. Then we started to analyse data by

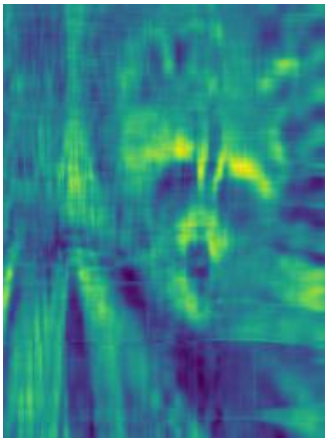


Figure 6: Racoon image reconstructed with 10 components.

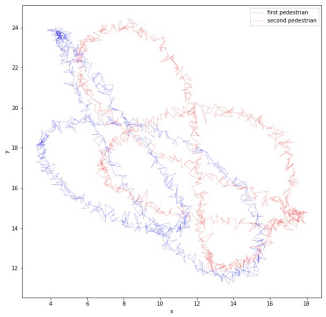


Figure 7: 2d representation of the trajectories of two pedestrians.

reducing the 30 dimension of given dataset, namely the trajectories of 15 pedestrians, to 2 dimensions by using PCA and then approximating trajectories matrix with the given equation above. After the approximation of the matrix, the trajectories of first two pedestrians look as follows:

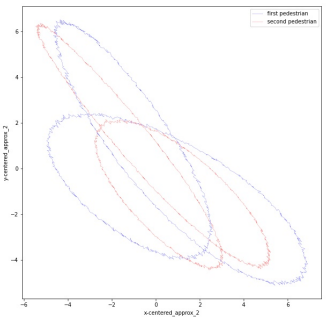


Figure 8: 2d representation of the trajectories of two pedestrians using only 2 components.

After the dimensionality reduction, we have measured the energy loss. The measured energy loss is approximately 15.08% which means we can not capture the most of the energy ($> 90\%$) by using only 2 principal components.
When we again reduce the dimensionality of 30 to 3, then the energy loss drastically decreases from 15.08% to 0.28% which means we can capture the most of the energy using 3 principal components.

Report on task Task 2/4: Diffusion Maps

1. It took as approximately 3 days to implement the Diffusion Map algorithm, to apply it on different data sets, observe the outcomes and make conclusions about the method we have used.
2. As Diffusion Map algorithm is able to reduce dimensionality of nonlinear manifolds as well, we were able to accurately represent the datasets on which we have applied the algorithm on. As a measure of our accuracy, we have used features such as making sure that the curves in the pedestrian data-set does not interact with each other.
3. We have learned that Diffusion Map algeorithm is a dimensionality reduction algorithm, where its popular use case is nonlinear manifolds where PCA does not work very well. The swiss-roll data set is also a non-linear manifold where we have calculated first ten eigenfunctions of the Laplace Betrami operator on this specific dataset. We have also analyzed the pedestrian dataset and realized that we can represent it using 3 different eigenfunctions without obtaining any intersection of the curves in the end.

In this task we were tasked to implement the Diffusion Map algorithm that is explained in steps in the lecture ourselves. We have used Python as the programming language for this task and used **numpy** and **scipy** as basic library support for linear algebra calculations (inverse, power of a matrix etc).

In the first part of the task, we used the diffusion algorithm that we have implemented on a periodic data set which consists $N = 1000$ points. The points in the dataset are given by the following formula:

$$X = \{x_k \in \mathbb{R}^3\}_{k=1}^{N}, x_k = (cos(t_k), sin(t_k)), t_k = (2\pi k)/(N+1)$$

In Figure 9 the eigen-functions that belong to the five largest eigenvalues are depicted. As explained in the exercise sheet as well, ϕ_0 is a constant function, so we are mainly interested into $\phi_{1..4}$. We can actually observe that these are oscillating functions, just like sine and cosine functions. Main difference between different eigen-functions is that they have either a phase shift or a different frequency.

In the second part of our task, we used the Diffusion Map algorithm on the **swiss roll** manifold. We have used the **sklearn** library as suggested to create the required data (**make_swiss_roll()**), where it contains 5000 points without any additional noise (see Figure 10). After creating the data set we computed ten eigen-functions of the Laplace Betrami operator on it and plotted it against the first non-constant eigen-function, ϕ_1.

In Figure 11 one can see these plots, where we have observed that at the value of $l = 4$ the function ϕ_4 is not a function of ϕ_1 anymore. We have arrived at this conclusion since we observe that after this value of l, the eigen-functions have multiple values for a single ϕ_1 value which cannot be analytically expressed by a function of ϕ_1 itself. In the first three plots however, we do not observe such a behaviour, hence one can state that they can still be represented with a function of ϕ_1.

We have also computed the three principal component of the swiss-roll dataset using the implementation in the first task. We have realized that it is impossible to represent the data using only two principal components, since we cannot fit a plane through the whole data set, the manifold loses too much energy and does not represent correctly the original data-set, since the topology of the points in the data-set also gets corrupted.

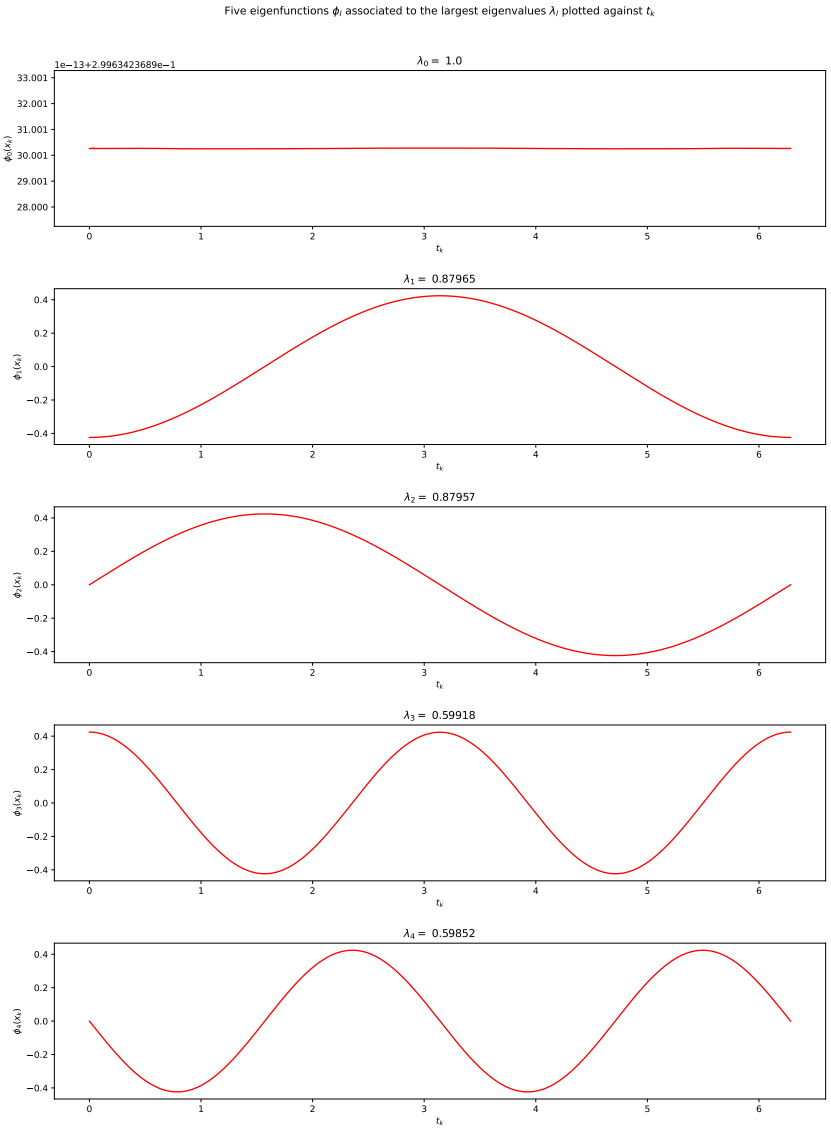


Figure 9: Five eigen-functions ϕ_l associated to the largest eigenvalues λ_l

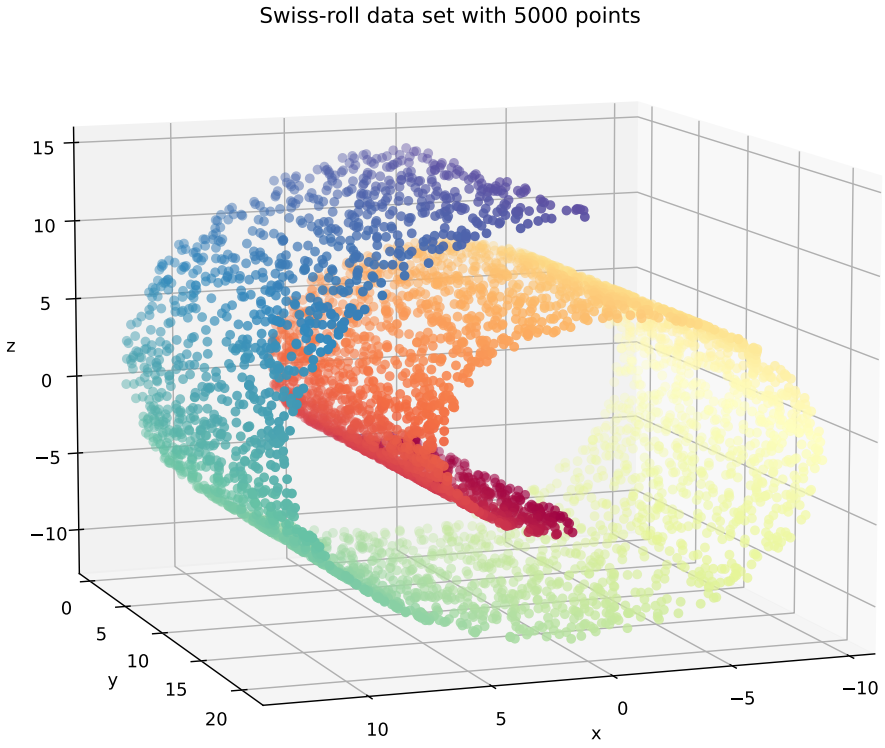


Figure 10: Swiss-roll manifold with 5000 points

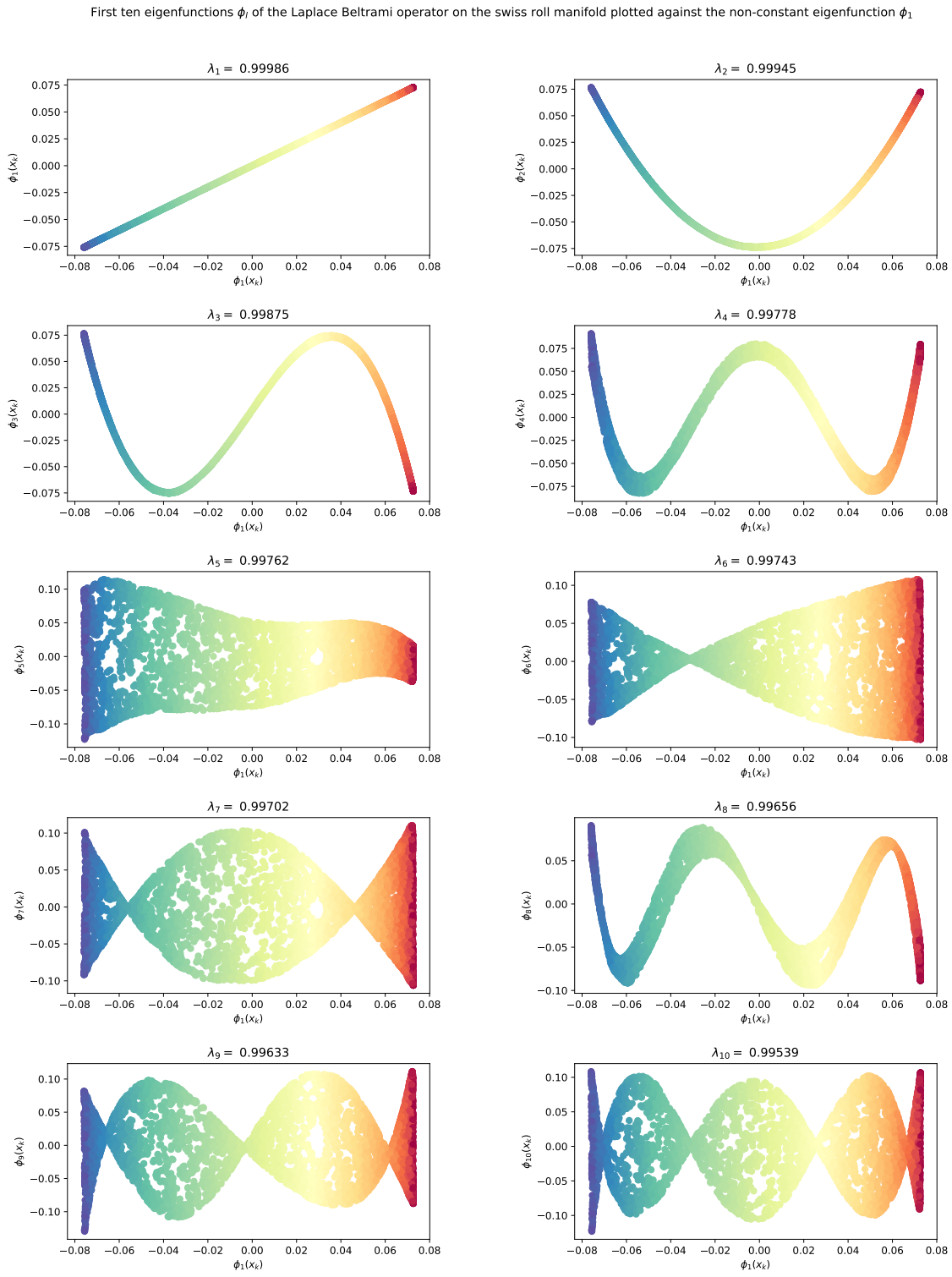


Figure 11: Ten eigen-functions of the Laplace Betrami operator on the swiss-roll data set

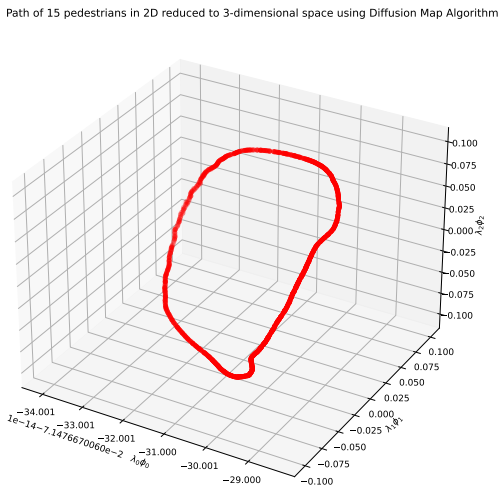


Figure 12: The paths of the pedestrians represented by 3 principal components

In the third part of the task our goal was to do the same analysis that we have done about the pedestrians, this time using the Diffusion Map algorithm. We have again loaded the pedestrian data from the given **.txt** file on Moodle using loadtxt() method of numpy library. Then we have applied the Diffusion Map algorithm with different number of principal components. We have tried different values of principal components to obtain a representation of the data that does not contain any intersection of the curves, as described in the exercise sheet and concluded with a principal component number of 3. After we have reduced our data to be represented by only 3 principal components, we have also created a plot of it, see Figure 12. It clearly shows a data distribution without any intersection of the curves as requested.

Report on task Task 3/4: Training a Variational Autoencoder on MNIST

In this section we build a machine learning model (Variational Autoencoder) on the MNIST dataset. For this we use `tensorflow` and `Sequential` defined in `tensorflow.keras` to group a stack of layers into a `keras` Model [1]. In the following sections `Dataset` and `Model` we explain how we preprocess the dataset and feed it into our model. Lastly we discuss the evaluation and the trained model.

Dataset

```
(x_train, _), (x_test, _) = tensorflow.keras.datasets.mnist.load_data()
```

Figure 13: Loading and splitting the MNIST dataset using tensorflow.

Using `tensorflow` we have downloaded the MNIST dataset as seen in figure 13. This splits the dataset into training (60000 samples) and test (10000 samples) sets. Each sample has one color channel, and have 28 by 28 dimension. In figure 14 we have visualized a sample image from the training set in order to get a feeling what the data looks like.

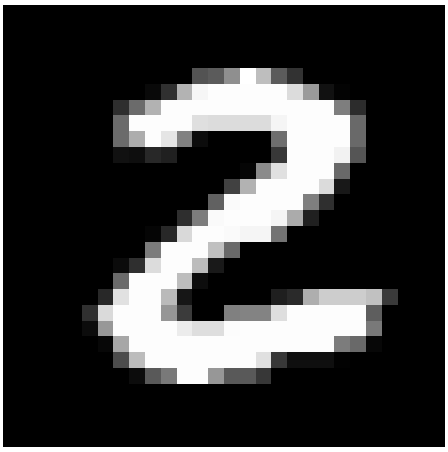


Figure 14: Visualization of the 25-th image of the MNIST training set.

The original images have the type `uint8`, so in order to normalize them we need to convert them to `float32` type and divide by 255. After this preprocessing step we are ready to batch our data.

```
train_dataset = (tensorflow.data.Dataset.from_tensor_slices(train_images)
                 .shuffle(train_size).batch(batch_size))
```

Figure 15: Shuffles and split the data into batches using tensorflow.

We again use `tensorflow` to shuffle and batch our data as seen in figure 15. We have used a batch size of 128 as given in the exercise sheet. We have also did the same for the test set. At this point our data is ready for training/testing, so we can define our machine learning model.

Model

Our Variational Autoencoder inherits `tensorflow.keras.Model`. It has `encoder` and `decoder` variables as `Sequential` Models. Additionally it defines three functions: `encode`, `decode`, `reparameterize`. We first look at the layers of the encoder and then talk about the reparametrisation. Finally we look at our decoder, how we have compute the loss and how we implement the training step.

For our Variational Autoencoder we define the following layers for the encoder $[0,1]^{28\times 28} \to (\mathbb{R}^L, \mathbb{R}^L)$; x image is mapped to the latent space distribution variables $(\mu_\phi, \log(\sigma_\phi))$ with the composition of the following functions in order:

- Input Layer: $[0,1]^{28\times 28}$
- Flatten Layer: $[0,1]^{28\times 28} \to [0,1]^{784}$
- Hidden Dense Layer 1 (fully connected with 256 units) with `ReLU` activation function: $[0,1]^{784} \to \mathbb{R}^{256}_{0,+}$
- Hidden Dense Layer 2 (fully connected with 256 units) with `ReLU` activation function: $\mathbb{R}^{256}_{0,+} \to \mathbb{R}^{256}_{0,+}$
- Output Layer with linear activation (no activation function): $\mathbb{R}^{256}_{0,+} \to (\mathbb{R}^L, \mathbb{R}^L)$

where `L` is the dimension of the latent space. We assume multivariate diagonal Gaussian as approximate posterior $q_\phi(z|x)$: $\mathcal{N}(\mu_\phi, \texttt{diag}(\sigma_\phi))$ where `diag` produces a diagonal matrix of shape (L, L) from the vector σ_ϕ. So we have $\phi = (\mu_\phi, \log(\sigma_\phi))$. We get the logarithm of the variance as the output of the encoder neural network for numerical stability.

Next we use the reparametrisation trick as defined in equation 1.

$$z \sim \mathcal{N}(\mu_\phi, \texttt{diag}(\sigma_\phi)) \quad \to \quad z = \mu_\phi + L\epsilon,\ \epsilon \sim \mathcal{N}(0, I),\ \texttt{diag}(\sigma_\phi) = LL^T \tag{1}$$

which immediately follows as $z = \mu_\phi + \sqrt{\sigma_\phi}\epsilon$ (element-wise square root) since $\texttt{diag}(\sigma_\phi)$ is a diagonal matrix.

```
def reparameterize(self, mean, logvar):
    """
    Computes z using the reparametrisation trick with the multivariate diagonal Gaussian parameters.
    :param mean: Mean of the approximate posterior (multivariate diagonal Gaussian is assumed)
    :param logvar: Logarithm of the variance of the approximate posterior (multivariate diagonal Gaussian is assumed)
    :returns: z = mean + sqrt(var) * epsilon : epsilon is sampled from N(0,I)
    """
    eps = tensorflow.random.normal(shape=mean.shape)
    return eps * tensorflow.exp(logvar * .5) + mean
```

Figure 16: Reparametrisation implementation in python.

In code 16 we see the reparametrisation trick implemented in `python`. We feed the outputs of the `encoder` into this function to sample a z in the latent space such that in the computation graphs the gradient calculations can be done.

```
def encode(self, x):
    """
    Encode the input tensor x using the Encoder defined in the VAE.
    :param x: Input tensor of shape (28,28)
    :returns: (mean, log(variance)) Parameters of the approximate posterior distribution.
    """
    mean, logvar = tensorflow.split(self.encoder(x), num_or_size_splits=2, axis=1)
    return mean, logvar
```

Figure 17: Encode function implementation of our model.

`Encode` function is implemented as depicted in code 17. It utilized the `encoder` network and splits the $(L + L)$ shaped output into two (L) shaped outputs which are the mean and the log-variance terms.

Our decoder network $\mathbb{R}^L \to \mathbb{R}^{28\times 28}$; latent sample (z) with dimension L is mapped to a sample $\hat{x}$ in the input space with composition of the following functions in order:

- Input Layer: $\mathbb{R}^L$
- Hidden Dense Layer 1 (fully connected with 256 units) with `ReLU` activation function: $\mathbb{R}^L \to \mathbb{R}^{256}_{0,+}$
- Hidden Dense Layer 2 (fully connected with 256 units) with `ReLU` activation function: $\mathbb{R}^{256}_{0,+} \to \mathbb{R}^{256}_{0,+}$

- Output Layer with linear activation (no activation function): $\mathbb{R}^{256}_{0,+} \rightarrow \mathbb{R}^{784}$
- Reshape Layer: $\mathbb{R}^{784} \rightarrow \mathbb{R}^{28\times 28}$

where the output is not normalized between 0 and 1 anymore as one may notice. However this should not be an issue and we expect the trained model to produce predictions near the normalized space $[0, 1]$. We assume multivariate diagonal Gaussian with Identity covariance matrix as the likelihood $p_\theta(x|z)$: $\mathcal{N}(\mu_\theta, I)$ So we have $\theta = \mu_\theta$.

```
def decode(self, z):
    """
    Decode the latent sample z using the Decoder defined in the VAE.
    :param z: The sample tensor z in the latent space
    """
    return self.decoder(z)
```

Figure 18: Decode function implementation of our model.

The `decode` function in the model is relatively simple as depicted in figure 18. We feed in the output of the reparametrisation into the `decode` function and get the output logits (no activation is applied in the end). They should approximate the input image in the trained model.

Optimisation

We first define our loss function. We use multivariate diagonal standard Gaussian as prior $p(z) = \mathcal{N}(0, I)$ as suggested in the exercise sheet. Additionally assume multivariate diagonal Gaussians for the approximate posterior and the likelihood as explained above. The loss for a single sample can be defined as $-\mathcal{ELBO}$ as defined in equation 3 with the Kullback–Leibler divergence as the regularization term. The term can be simplified into logarithm of the distributions and one can further simplify the logarithm terms for the prior and the posterior. From the lecture we know that the likelihood term depends on the squared error between the real data x and the model prediction $\hat{x}$ which is the output of the decoder in our case (we take the mean of the likelihood distribution as the sample, so we do not re-sample the output distribution).

$$\mathcal{ELBO} = \mathbb{E}_{z\sim q_\phi(z|x)}[\log p_\theta(x|z) - \mathbb{KL}(q_\phi(z|x)||p(z))] = \mathbb{E}_{z\sim q_\phi(z|x)}[\log p_\theta(x|z) + \log p(z) - \log(q_\phi(z|x)] \tag{2}$$

$$\sim \mathbb{E}_{z\sim q_\phi(z|x)}\left[-\frac{1}{2}||x-\hat{x}||^2 + \log p(z) - \log(q_\phi(z|x)\right] \tag{3}$$

One can easily implement this term using Monte Carlo estimations and with basic equation of the pdf of the logarithm of the standard Gaussians as computed in equation 4 where μ is the mean and σ is the standard deviation of the distribution. We simply use this equation to compute the log terms for the prior and the approximate posterior with Monte Carlo estimates of the mean of the current batch (128 sample) in each iteration. We also do a MC-estimate for the squared error term defined in the $-\mathcal{ELBO}$ loss.

$$\log\left[\frac{1}{\sigma\sqrt{2\pi}}\exp\left(-\frac{1}{2}\frac{(x-\mu)^2}{\sigma^2}\right)\right] = -\frac{1}{2}\left[\frac{(x-\mu)^2}{\sigma} + \log\sigma + \log 2\pi\right] \tag{4}$$

After implementing the loss computation we just needed to implement to the train step where we do the forward and backward pass and update the parameters of our NN (weights) towards the negative of the gradient of the final loss. With `tensorflow` one can do this step using the code in figure 19.

We use `ADAM` optimiser from `keras.optimizers` with a learning rate of 0.001 as given in the exercise sheet.

Training

For the training we have considered the following things to be important indicators of how well the model is training:

- Latent representation of the test set, i.e. we encode and see how the data is spread around the origin of the 2-dimensional latent space. The model should be able to differentiate / cluster different images w.r.t their labels accordingly.

```
def train_step(model, x, optimizer):
    """
    Executes one training step and returns the loss.
    Computes the loss and gradients.
    Updates the model's parameters.
    """
    with tensorflow.GradientTape() as tape:
        loss = compute_loss(model, x)
    gradients = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients(zip(gradients, model.trainable_variables))
```

Figure 19: Single training step implementation using tensorflow.

- We choose 16 images from the test set at the beginning of the training and we try to reconstruct them using our model to see how well our model can manage to reduce the reconstruction error. If the reconstruction error stays too high, then we can notice this by inspecting these reconstructions and see if they improve over the number of epochs.
- We generate 16 images by randomly sampling latent samples z from the prior distribution $p(z)$ and feeding them into our trained `Decoder` and see how well the generation behaves with different latent samples.
- Finally when using 2-dimensional latent space we choose linearly spaced latent samples and generate images from them using our `Decoder`. This way we should see images from all the labels and maybe try to catch the similar images like '4' and '6' or '8' and '9'.

We check the above explained criteria by plotting them at each epoch iteration. While training our model we have noticed that using more than 100 batch iterations results in a convergence already. It is important to note that we have used 2-dimensional latent space for this training as suggested in the exercise sheet. We will however include a comparison with the 32-dimensional latent space later in this task report. We have plotted the loss curve in batch iteration number with training and test losses in figure 20. As one can see training the model furthermore results in oscillations in the loss curves. Therefore we have used a manual early stopping at epoch 100, and we use this trained model in the further analysis.

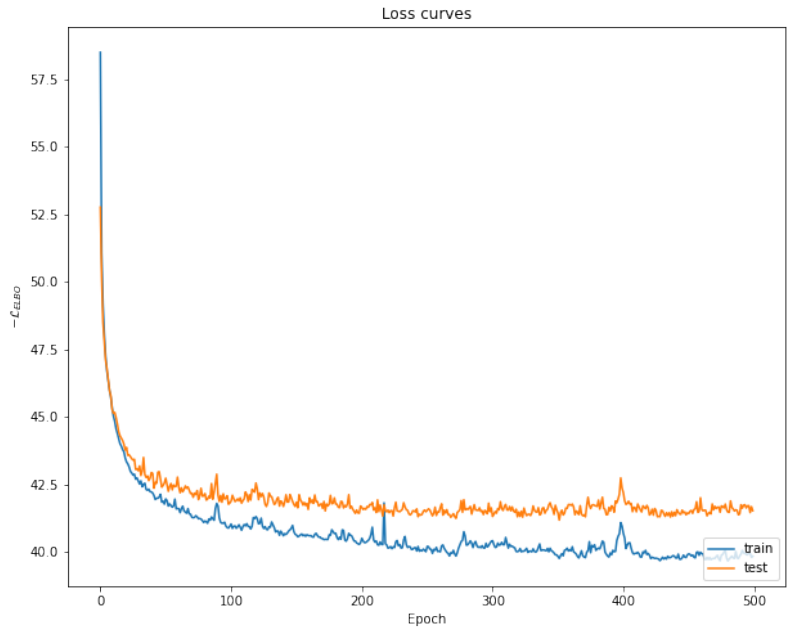
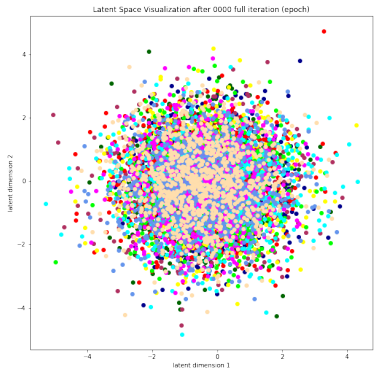
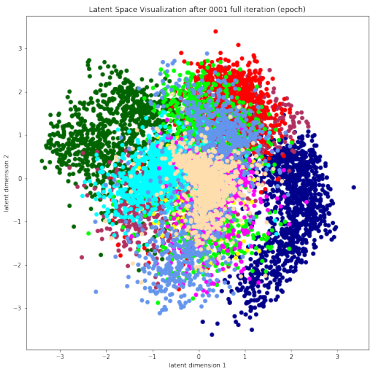


Figure 20: Plot of the loss curves (training + test sets)

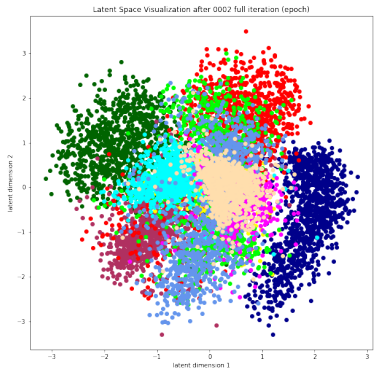
First let's look at the latent space as we train the model up to 100 batch iterations. We see in figure 21 and 22 the 2-dimensional latent space plots in some of the batch iterations. At the beginning we can see that the initial parameters of the model does not have a specific structure that represent different labels of the images. But it is immediate to see that the model manages to cluster different labels into different clusters, but still struggles with some of the labels such as the red and green ones, which corresponds to the digits '3' and '5'. Also pink and brown (sand) which corresponds to the digits '2' and '9' respectively seem to be overlapping. The reason for this might be that they can look similar when they are handwritten. Outliers do however exist and they can be because of the badly written digits, so a '9' may look like a '0' and vice versa.



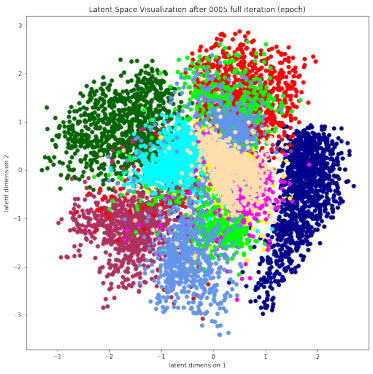
(a) Initial



(b) Batch Iteration: 1

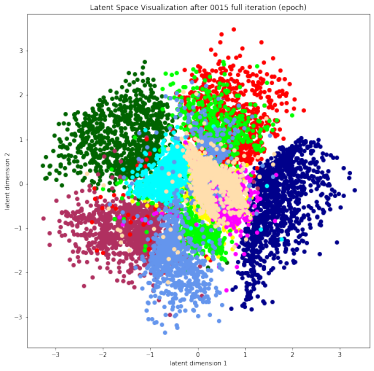


(c) Batch Iteration: 2

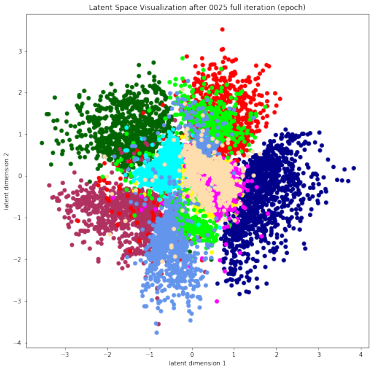


(d) Batch Iteration: 5

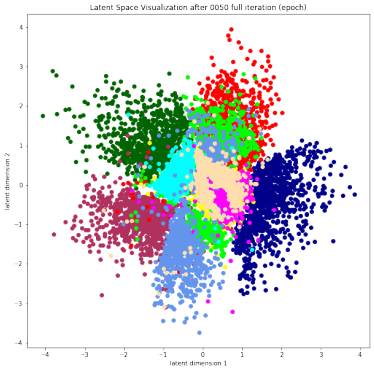
Figure 21: Latent Space Plots after some Batch Iterations



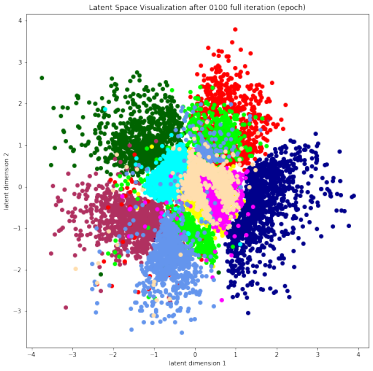
(a) Batch Iteration: 15



(b) Batch Iteration: 25



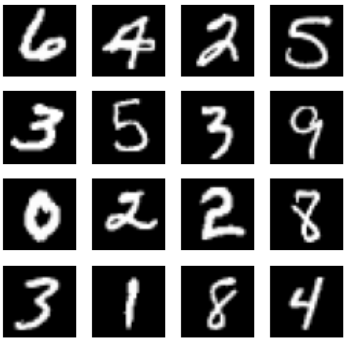
(c) Batch Iteration: 50



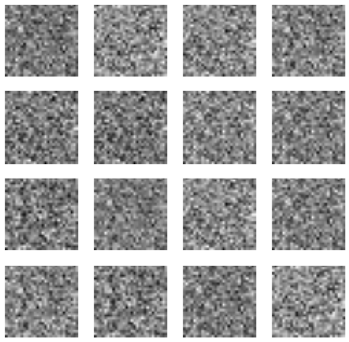
(d) Batch Iteration: 100

Figure 22: Latent Space Plots after some Batch Iterations

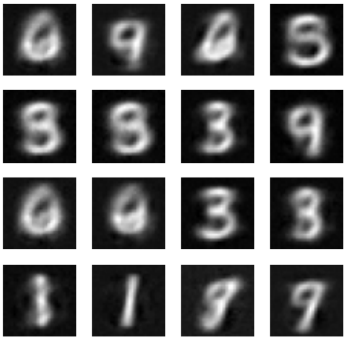
The reconstructions of 16 randomly picked images from the test dataset is plotted in figures 23 and 24. With the initial weights the model can only produce white noise like images as one can see at batch iteration 0. One thing we have noticed is that for reconstruction we require at least 10 batch iterations to get meaningful results. In our model we have clear reconstructions at the final batch iteration 100. But some of the digits are wrong. In the end of the first row the original digit was '5' but our model reconstruct it and produced a '3' on the test set. Similarly on the second row one of the images is actually '5' but our model reconstructed it as '3'. So one might say that our model can confuse the digits '5' and '3'. The model also cannot reproduce the digit '4' in the last row, and computes that it should be a '9'. However if we look at the original image we notice that it is quite similar to a '9', but we can still understand that it is a '4'. The model however does not. Our reconstructions are however not very bad and not very blurry at the end, which may be because of our clustering in the latent space is distinct enough to differentiate (map) different labels to different latent variables.



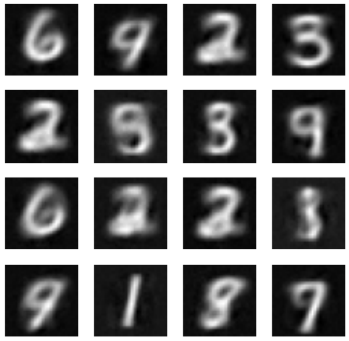
(a) Original 16 images from test set



(b) Reconstructions at batch iteration: 0

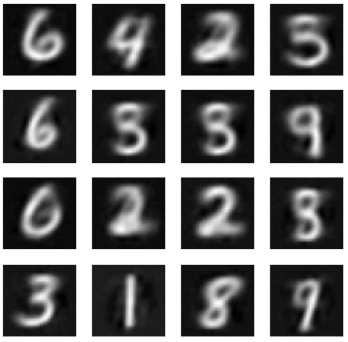


(c) Reconstructions at batch iteration: 1

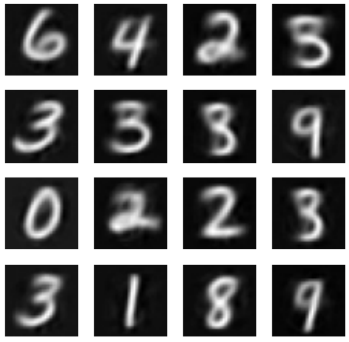


(d) Reconstructions at batch iteration: 2

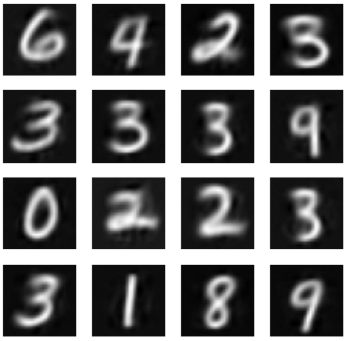
Figure 23: 16 images reconstructed during the training.



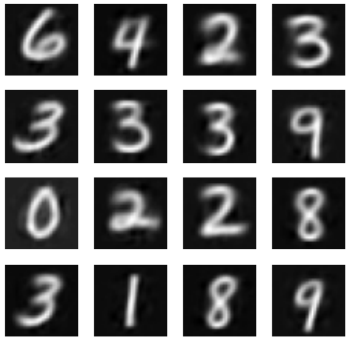
(a) Reconstructions at batch iteration: 5



(b) Reconstructions at batch iteration: 25



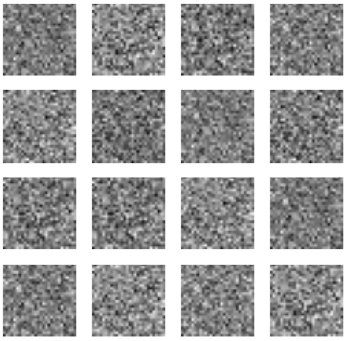
(c) Reconstructions at batch iteration: 50



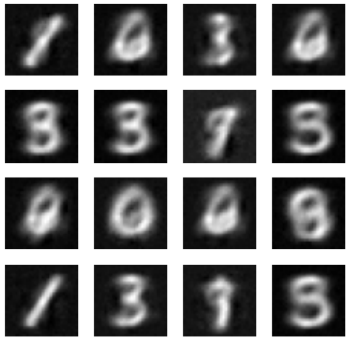
(d) Reconstructions at batch iteration: 100

Figure 24: 16 images reconstructed during the training.

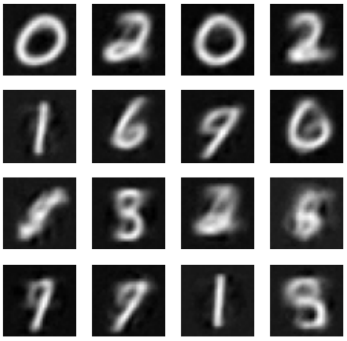
Randomly generated – sampled from the prior distribution $p(z)$ and fed into the `Decoder` during the training. Figures 25 and 26 depict these generations in different batch iterations. Images generated after the 100-th batch iteration can be identified and labelled by the reader.



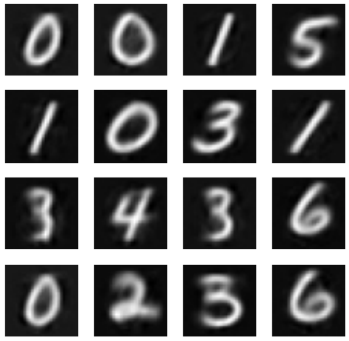
(a) Random Generations with the initial model weights



(b) Random Generations at batch iteration: 1

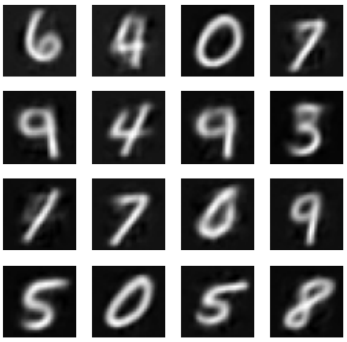


(c) Random Generations at batch iteration: 5

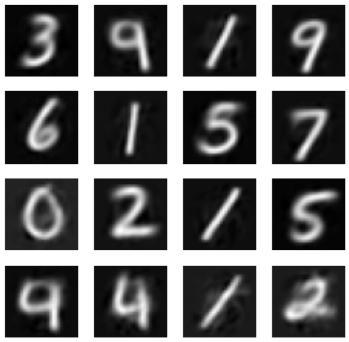


(d) Random Generations at batch iteration: 25

Figure 25: 16 random image generations during the training.



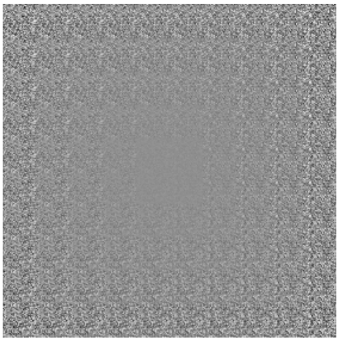
(a) Random Generations at batch iteration: 50



(b) Random Generations at batch iteration: 100

Figure 26: 16 random image generations during the training.

Figures 27 and 28 are the generated images at different batch iterations where we sample the latent variables evenly spaced. Therefore these images should include all the digits from 0 to 9. With initial weights we again get whit-noise. Later we see how different digits converge and placed in the square of 100 images. At the top right of the image of the 100-th batch iteration we see the digits representing '1's. At the bottom left there are '0's and so on. One can notice that some of the transitions are not smooth, e.g. the transition from '2' to '6' or '4' to '5'.



(a) Latent decodings (generations) with the initial model weights

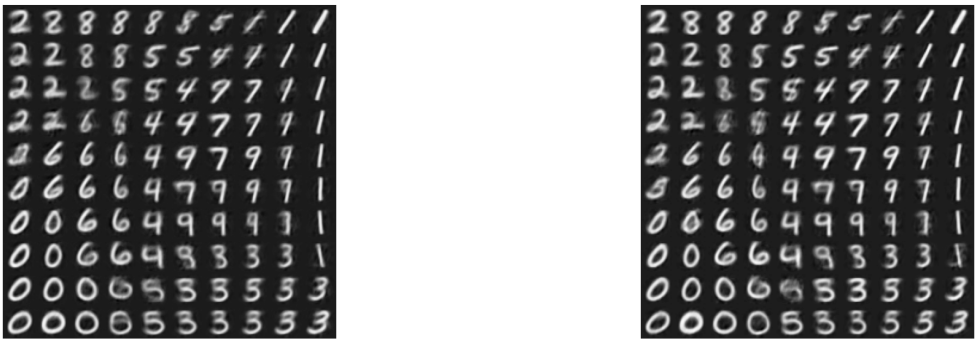
(b) Latent decodings (generations) at batch iteration: 1

(c) Random Generations at batch iteration: 50

(d) Random Generations at batch iteration: 25

Figure 27: Latent image generations (100 images) with evenly spaced sampled latent variables z.

When using 32-dimensional latent space the loss curve is depicted in figure 29. We notice less oscillations than the loss curves of the 2-dimensional latent space model. We again use 100 batch iterations to train this model (manual early stopping) but the randomly generated images were very similar for ¿100 batch iterations. In figure 30 we see the randomly generated images. We did not include the reconstructions but their accuracy look similar to the 2-dimensional latent space model that we have trained. But for the generation our 32-dimensional latent space model generates blurry images as one can see in the figure, compared to our 2-dimensional latent space model.



(a) Random Generations at batch iteration: 50 (b) Random Generations at batch iteration: 100

Figure 28: Latent image generations (100 images) with evenly spaced sampled latent variables z.

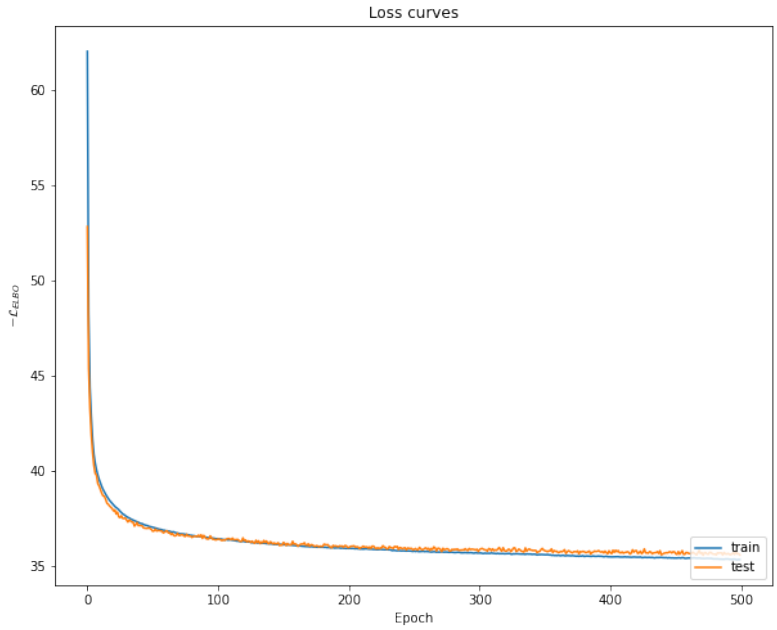
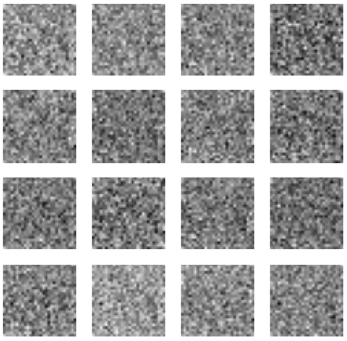


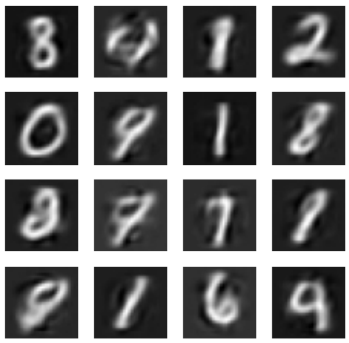
Figure 29: Plot of the loss curves (training + test sets) of the 32-dimensional latent space model.



(a) Random Generations with initial weights

(b) Random Generations at batch iteration: 1

(c) Random Generations at batch iteration: 5



(d) Random Generations at batch iteration: 25

(e) Random Generations at batch iteration: 50

(f) Random Generations at batch iteration: 100

Figure 30: Random 16 image generations with 32-dimensional latent space.

What activation functions should be used for the mean and standard deviation of the approximate posterior and the likelihood—and why? We think, identity (no/linear activation) should be used since the targets are floating point numbers, and the model should keep in the range of the positive real numbers after the training. We have also validated this by looking at the whole image generation and reconstructions and the minimum pixel value we got was about 0 and the maximum was about 1 (since we train on the normalized pixel values this makes sense).

What might be the reason if we obtain good reconstructed but bad generated digits? This may mean no (or

little) regularization is used, so the KL-divergence between the encoder and the prior is not taken into account while training the network. As a result the reconstruction loss may have converged all the distributions into single points. For good generation we need to be able to sample latent variables z from the prior and use them to generate realistic (small loss) images. But for this our approximate posterior should not diverge too far from the prior distribution.

Since we did not analyze the performance / efficiency of our implementation, we leave out the hardware that we have used for the training (even our laptops could handle the training for this relatively small dataset).

General questions:

1. it took us about 10 days to completely implement, test and train the models.
2. for the measure of accuracy we used the loss function (squared error of the pixel values) to see how well we have reconstructed the data. However for the generation we have observed the outputs and tried to make reasonable statements for the samples where the model could not accurately generate.
3. dataset is handrwritten digits. Some of the images are hard to notice even with human eye (e.g. '9' looks like a '0'). VAE however could classify and generate some of the images very accurately. This method can be used for dimensionality reduction or data generation.

Report on task Task 4/4: Fire Evacuation Planning for the MI Building

In this task we first download the given 'npy' files for the train and test datasets using `load` function in the `numpy` module. The training set contains 3000 and the test set contains 600 samples. For this task we do not need to train the model for dimensionality reduction as we only want to generate realistic data. Therefore we set the dimensionality of the latent space to '2' which equals the dimensionality of the data samples.

We first visualize the training data to get a better feeling of what we are handling. In figure 31 we plot the data set.

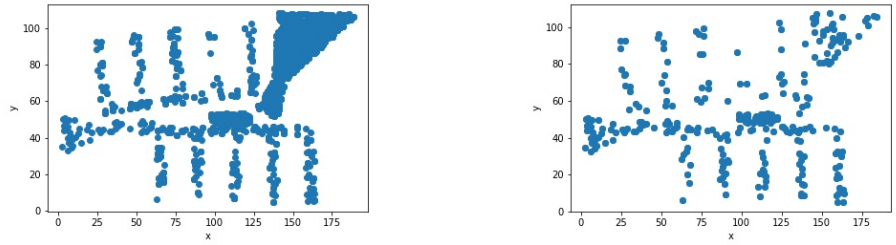


Figure 31: Training/Test data set scatter plot of the fire evacuation plan.

Now we normalize the data set between -1 and 1. The final plot can be seen in figure 32.

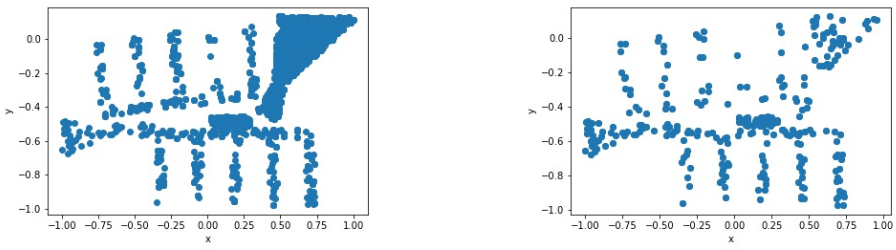


Figure 32: Training data set (normalized between -1 and 1) scatter plot of the fire evacuation plan.

We have changed our model a bit by increasing the number of hidden units to 512 in the hidden layers of the decoder and encoder. Additionally we have added one more hidden layer with 512 units and with `tanh` activation function to the encoder and decoder since this was the best configuration we could get when training the network. We again have used a batch size of 512 samples and the `ADAM` optimizer with a learning rate of

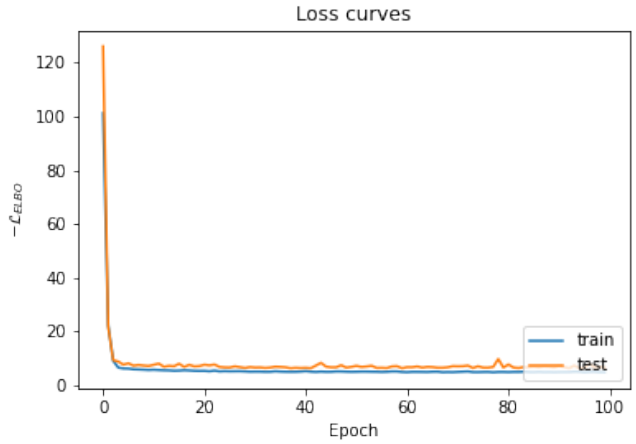


Figure 33: Loss cures (train and test) of the fire evacuation training.

0.001. We let the model train for 100 epochs and in figure 33 we see the loss curve. As one can see, 100 epoch iterations are more than enough to train the network.

We have reconstructed the test set and got the results depicted in figures 34. As one can see, we the model manages to converge to the original plot epoch by epoch. The human eye does not see much difference between the plots at epoch iteration 50 and 100, but in the first epoch iterations the convergence can be noticed.

Our trained model however cannot handle the generation tasks as good as it handles the reconstruction tasks. Figure 35 shows how the generation has converged to its final state at epoch iteration 100. We have tried to put different weights to the KL-divergence term in order to get better generations but this time we could not get a sufficiently low reconstruction error. These results are the best we could get.

We have tried to answer the last question in the exercise sheet about the required number of people to exceed the critical people in the orange area. Since our data generation is not very accurate, we could not exceed the critical number of pedestrians in the orange region even if we sampled 1 million points.

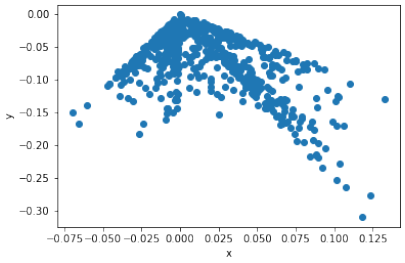
1. it took us approximately 2 days to implement, test and write the report about this task.

2. for reconstruction we tried to compare our reconstruction with the test set and tried to guess if it fits the shape of the test set. For generation we have looked at the whole dataset shape at see if our generations could capture the shape of the dataset again. For our VAE we have used the same loss that we have used in the previous task.

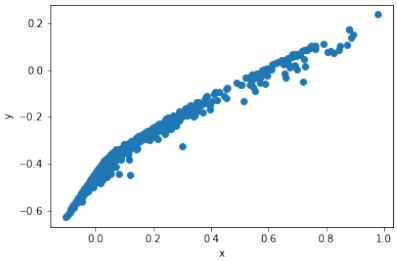
3. the dataset consists of 2-dimensional samples without any labels. Since we are familiar with the building, we could guess why the shape looks like it looks and why people have gathered in the entrance (this would also makes sense with other data regarding the fire evacuation of course). The data is however very complex to model for us with a single probability distribution like a multivariate Gaussian for example. We have thought that our model could capture this complex structure of the model and could generate meaningful data after the training. However the method is 'too weak' to capture the data structure with our prior distribution assumption. This may mean that our model needs more hidden layers or other activation functions. We did not have the time to do a hyperparameter optimization like a grid-search or a random-search, but this may increase the performance of our model.

References

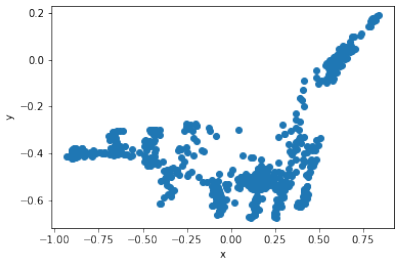
[1] https://www.tensorflow.org. *Tensorflow*. May 2022.



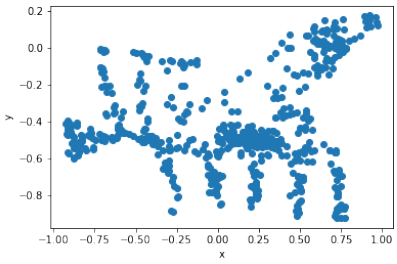
(a) Reconstruction using the initial weights



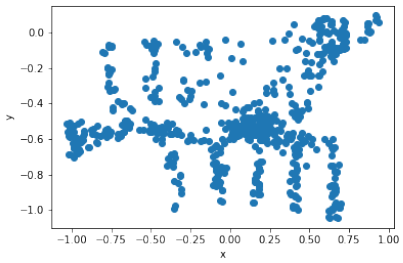
(b) Reconstruction after epoch iteration: 1



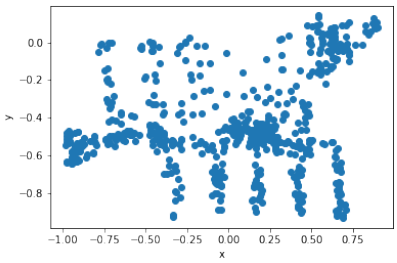
(c) Reconstruction after epoch iteration: 2



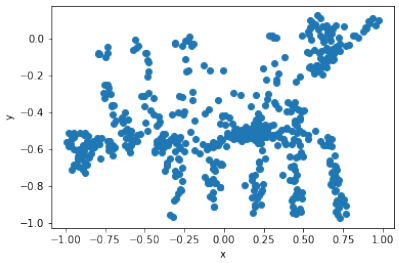
(d) Reconstruction after epoch iteration: 3



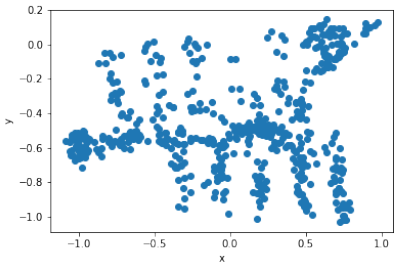
(e) Reconstruction after epoch iteration: 4



(f) Reconstruction after epoch iteration: 5

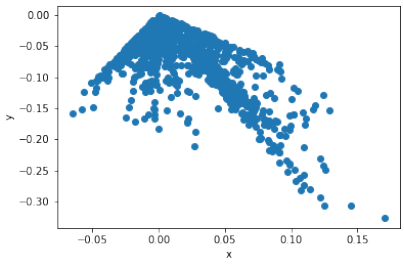


(g) Reconstruction after epoch iteration: 25

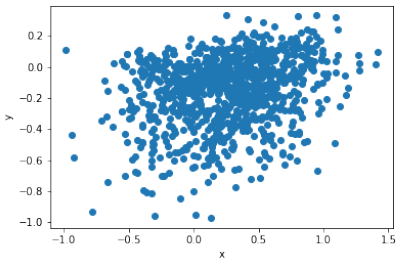


(h) Reconstruction after epoch iteration: 100

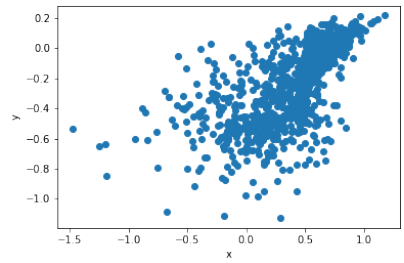
Figure 34: Reconstructions of the test set on the fire evacuation model at different epoch iterations.



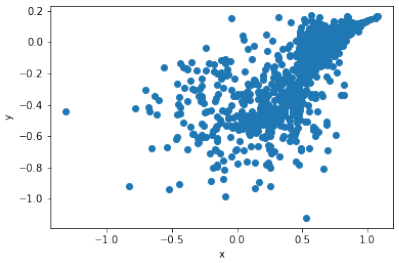
(a) Reconstruction using the initial weights



(b) Reconstruction after epoch iteration: 1



(c) Reconstruction after epoch iteration: 2



(d) Reconstruction after epoch iteration: 3

Figure 35: Random sample generations (1000 samples) of the fire evacuation model at different epoch iterations.