CS4240 Reproducibility Project: (Trying to) reproduce "Variational Autoencoders: A Harmonic Perspective"

Austin Ramana
Mustafa Güverte
Richard Eveleens
Dimme de Groot

April 14, 2022

1 Introduction

The variational autoencoder is a specific case of a deep neural network, it is associated with other autoencoders as in it has an encoder which decreases dimensionality of an input (by keeping the distribution regularised) and a decoder which attempts to reconstruct said input. However, unlike conventional autoencoders, where one maps the input to a vector, for a variational autoencoder (VAE), the input is mapped on a Gaussian distribution which then resolved by the decoder. This report attempts to replicate the results of [1] which showed remarkable results when influencing VAE's with Gaussian noise. They were able to control the model's Lipschitz constant and thus increase adversarial robustness, this report attempts to reproduce the same findings for the Lipschitz constant as well as replicate the network's robustness to the adverserial attack.

A virtual machine was set up and trained on the Sinc, Ciphar10 and CelebA (of which only a portion of the dataset was used due to computing issues). The Lipschitz constant for each respective dataset was estimated, and then the robustness of the modules against adversarial attacks was analysed and compared against an model that did not experience a maximum-damage attack. These findings where then compared with [1] in order to determine if the studies conclusions could be replicated.

2 Determining the Lipschitz constant

In this section, we discuss the Lipschitz constants of various networks. As explained before, the networks are variational autoencoders. They are trained for two different cases. Firstly, for a fixed encoder standard deviation $\sigma_{enc} \in \{0.1, 0.5, 1.0\}$ and without noise injections on the input data. Secondly, for a fixed encoder standard deviation $\sigma_{enc} = 0.5$ and zero-mean Gaussian noise injections on the input data. The noise injections have standard deviation $\sigma_{\mathcal{N}} \in \{0.0, 0.5, 1.0\}$. These values directly correspond to the values used in [1].

For the first case, the Lipschitz constant of the decoder network is calculated. For the second case, the Lipschitz constant of the encoder network is calculated.

In the following, first the network architecture is explained. Then, in Section 2.2, the used datasets are briefly discussed. Lastly, in Section 2.4, the results are presented.

2.1 The network

The networks for both cases are equal. The encoder first vectorizes the input data (which are images). This is then passed through the input layer, which gives a 256-dimensional output. This output is passed through some hidden layers with sigmoid activation functions. The last layer yields the mean value of our latent variable $\mathbf{z} \sim \mathcal{N}\left(\mu(\mathbf{x}), \sigma_{enc}^2 \mathbf{I}\right)$. For two of the three datasets (CelebA and CIFAR10), $\mathbf{z} \in \mathbb{R}^{64}$. For the last dataset (sinc), $\mathbf{z}$ is one dimensional. The network is visually shown in Figure 1.

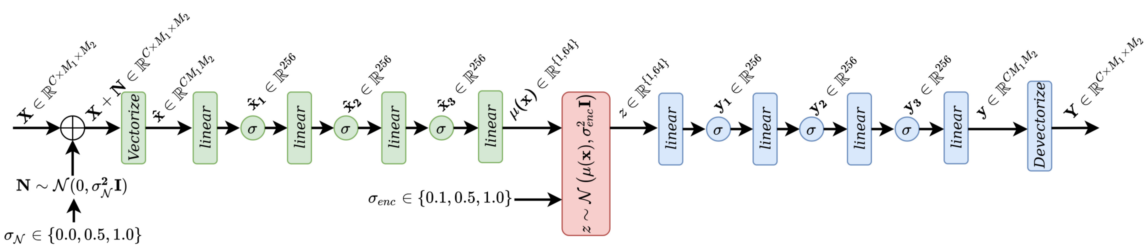


Figure 1: A visual description of the used network. Green corresponds to the encoder network, red corresponds to the latent space and blue corresponds to the decoder network. By setting $\sigma_{\mathcal{N}}$ to zero, the noise injections will effectively be removed.

2.2 The datasets

The network are trained on the following three datasets.

1. **Sinc:** The sinc training dataset consists of 4096 realisations of the function $\text{sinc}(5t)$. Each realisation of the function consists of an ordered time-axis $\mathbf{t}_{\text{ordered}} \in \mathbb{R}^{172\times 1}$. The unordered axis $\mathbf{t}$ is drawn from a uniform distribution with values ranging between $[-1, 1)^{172\times 1}$ and subsequently ordered by increasing value. For the test data, a single sinc with a uniformly spaced time-axis between $[-1, 1]^{172\times 1}$ is used. All data is normalised between $[-1, 1]$. Note that this is done over the full training set, so not every realisation actually reaches 1 and -1. By stacking the ordered time-axis and the corresponding sinc data on top of each other, a single realisation $\mathbf{x} \in \mathbb{R}^{172\times 2}$ is obtained.

2. **CIFAR10**: the CIFAR10 dataset consists of RGB figures size 32×32. The dataset is intended for classification. As was the case for the sinc dataset, the CIFAR10 dataset is normalised between -1 and 1.

3. **CelebA**: the CelebA dataset consists of RGB figures size 64×64. As was the case previously, the CelebA dataset is normalised between -1 and 1. It should be noted that, due to computer limitations, a subset of the CelebA dataset was used. Namely, the first 51200 training images. The complete validation set was used.

Before going to the next section, note that it is expected that the networks trained sinc and CelebA are expected to reproduce the input data relatively well (as in, the difference between the output of the VAE and the input will be relatively small). On the other hand, this is not expected for the networks trained on CIFAR10. The reason is that the CIFAR10 dataset is intended for classification, hence, the training images will not have the same coherence as the sinc and CelebA training data has (e.g., always a face, a mouth, eyes, etc., all approximately at the same scale, location, orientation, etc.).

2.3 Training

The networks were trained on a laptop with 8 GB RAM and an Intel i5-8250U CPU clocked at 1.60 GHz. No external graphics card was available, so the network is trained directly on the CPU. The used loss function is the mean-squared error with Kullback-Leibler divergence as regularisation term. Furthermore, the ADAM optimiser was used with PyTorch its default settings (see [2]). Lastly, a learning rate of 10^{-3} and a batch size of 256 is used for all three datasets.

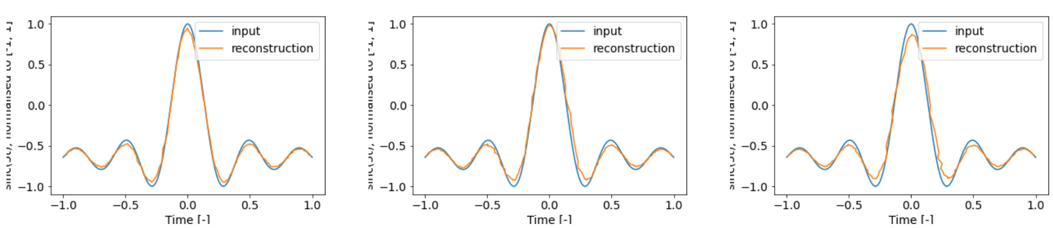
All experiments were repeated three times. The following torch seeds were used (obtained using `torch.seed()`): 14609714069366804553, 10209623728859046282, 13978253786968215756. To avoid repeating these numbers, they will be referred to as seed 1, seed 2 and seed 3 respectively.

2.4 Results

Before presenting the calculated upper bounds for the Lipschitz constants, we first take a look at some sample images from the validation data. In all cases, figures obtained from networks trained using seed 1 are shown.

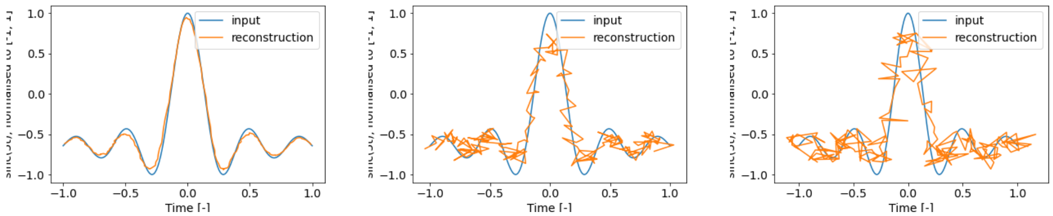
Validating the trained networks

Firstly, the sincs reconstructed by networks trained without noise injections can be found in Figure 2. The results for the networks trained with noise-injections can be found in Figure 3.



(a) $\sigma_{enc} = 0.1$ (b) $\sigma_{enc} = 0.5$ (c) $\sigma_{enc} = 1.0$

Figure 2: The obtained reconstruction of the sinc data from the validation set. The networks are trained without noise injections on the input. The encoder networks have a standard deviation $\sigma_{enc} \in \{0.1, 0.5, 1.0\}$.



(a) $\sigma_{\mathcal{N}} = 0.0$ (b) $\sigma_{\mathcal{N}} = 0.5$ (c) $\sigma_{\mathcal{N}} = 1.0$

Figure 3: The obtained reconstruction of sinc data from the validation set. The networks are trained with zero-mean Gaussian noise injections on the input. The injections have a standard deviation $\sigma_{\mathcal{N}} \in \{0.0, 0.5, 1.0\}$. The encoder networks have a fixed standard deviation $\sigma_{enc} = 0.5$.

From the first set of figures, two things can be noted. Namely (1) the reproduced sincs do follow the input image quite closely. However, the reproductions have trouble reaching the peaks of the input. This is especially pronounced for the network with $\sigma_{enc} = 0.5$. Interestingly, the only network which is able to reach the top of the main lobe is the network with $\sigma_{enc} = 0.5$. This one does, however, have more trouble with reaching the sidelobes than the network with $\sigma_{enc} = 0.1$ has. This might only be due to statistical differences, as the network from Figure 3a has the same standard deviations, but does not reach the peak of the main sidelobe.

From the second set of figures, it can be seen that the noise injections have a large impact on the ability to reproduce the sinc properly, which seems to be directly proportional to the amount of noise added.

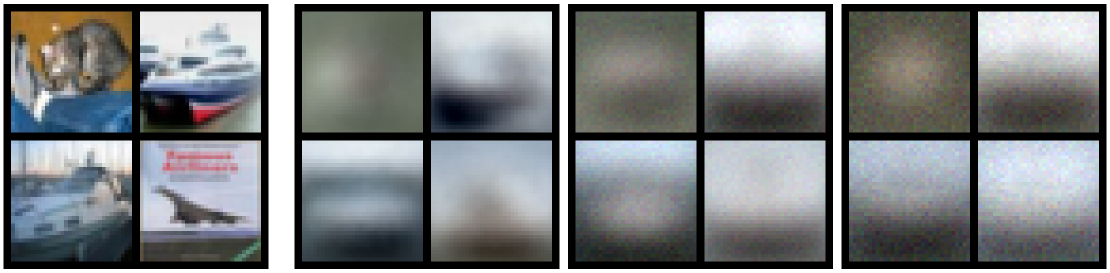
Secondly, the CIFAR10 images reconstructed by networks trained without noise injections can be found in Figure 4. The results for the networks trained with noise-injections can be found in Figure 5.

(a) Input images (b) $\sigma_{enc} = 0.1$ (c) $\sigma_{enc} = 0.5$ (d) $\sigma_{enc} = 1.0$

Figure 4: The obtained reconstruction of four CIFAR10 images from the validation set. The networks are trained without noise injections on the input. The encoder networks have a standard deviation $\sigma_{enc} \in \{0.1, 0.5, 1.0\}$.

From the figures, it can easily be seen that the networks have a hard (softly expressed) time reproducing the CIFAR10 images. The only parts which get reproduced somewhat properly are the background color and a very rough shape. This is expected, as the CIFAR10 dataset is a dataset intended for classification.

From the networks trained without noise-injections on the input, we would say (judged by

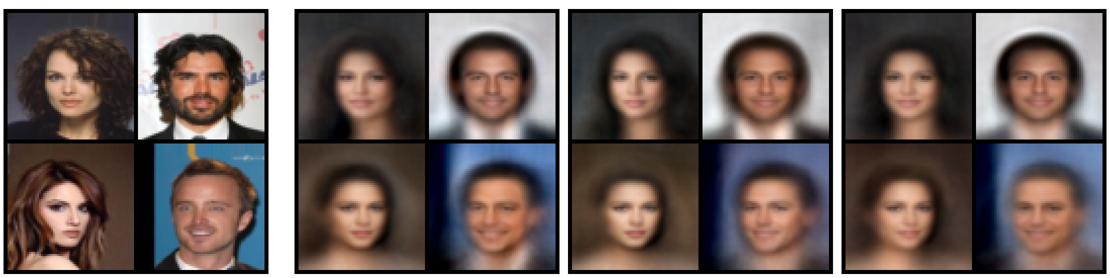


(a) Input images (b) $\sigma_{\mathcal{N}} = 0.0$ (c) $\sigma_{\mathcal{N}} = 0.5$ (d) $\sigma_{\mathcal{N}} = 1.0$

Figure 5: The obtained reconstruction of four CIFAR10 images from the validation set. The networks are trained with zero-mean Gaussian noise injections on the input. The injections have a standard deviation $\sigma_{\mathcal{N}} \in \{0.0, 0.5, 1.0\}$. The encoder networks have a fixed standard deviation $\sigma_{enc} = 0.5$.

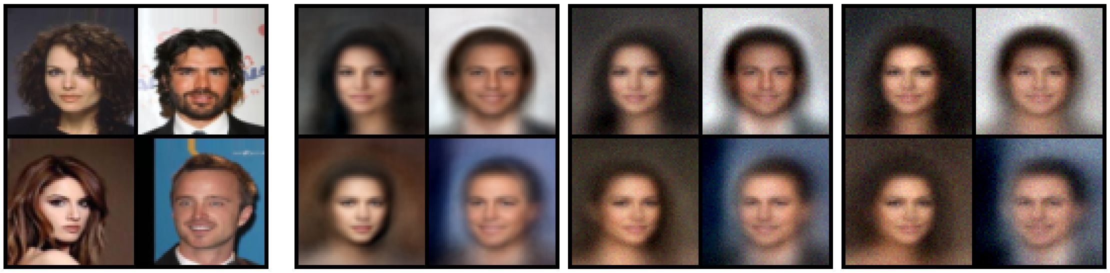
eye) that the networks with a higher σ_{enc} are able to follow the figures better. From the networks trained with noise injections on the input, the outputs also are noisy.

Lastly, the CelebA figures reconstructed by networks trained without noise injections can be found in Figure 6. The results for the networks trained with noise-injections can be found in Figure 7.



(a) Input images (b) $\sigma_{enc} = 0.1$ (c) $\sigma_{enc} = 0.5$ (d) $\sigma_{enc} = 1.0$

Figure 6: The obtained reconstruction of four CelebA images from the validation set. The networks are trained without noise injections on the input. The encoder networks have a standard deviation $\sigma_{enc} \in \{0.1, 0.5, 1.0\}$.



(a) Input images (b) $\sigma_{\mathcal{N}} = 0.0$ (c) $\sigma_{\mathcal{N}} = 0.5$ (d) $\sigma_{\mathcal{N}} = 1.0$

Figure 7: The obtained reconstruction of four CelebA images from the validation set. The networks are trained with zero-mean Gaussian noise injections on the input. The injections have a standard deviation $\sigma_{\mathcal{N}} \in \{0.0, 0.5, 1.0\}$. The encoder networks have a fixed standard deviation $\sigma_{enc} = 0.5$.

From the figures, it can be seen that the background color and the tilt of the head is reproduced properly. For the network with $\sigma_{enc} = 0.1$, even some of the beard remains. As was the case previously, the networks which are trained with noise injections on the inputs also give noisy outputs.

Lipschitz constants

One of the main goals of this reproducibility project is to estimate upper bounds of the Lipschitz constants of the encoder and decoder networks. These estimations are given in Table 1, Table 2 and Table 3. The tables respectively correspond to the models trained on $\text{sinc}(5t)$, trained on CIFAR10 and trained on CelebA. The upper bounds are estimated using layer by layer LipSDP [3, 4].

For all tables, the Lipschitz constants of the decoder networks are shown in the second to fourth column. These networks were trained without noise injections on the input.

In the fifth to last column, the Lipschitz constants of the encoder networks are shown. These networks were trained with noise injections on the input. The noise injections are zero-mean and have standard deviation $\sigma_{\mathcal{N}}$. Furthermore, the encoder has a fixed standard deviation of $\sigma_{enc} = 0.5$.

For both the networks and encoders, we give the mean and standard deviation of the found values in the one-before-last row. In the last row the reference values, directly taken from [1], are shown.

In Table 1, it can be seen that the upper bound of the Lipschitz constant of the decoder networks follow the same trend as would be expected from [1]. The values do, however, not correspond at all. For the upper bound on the Lipschitz constant of the encoder network, the trend predicted by [1] is not seen at all.

In Table 2, it can be seen that the upper bound of the Lipschitz constant of the decoder networks does not follow the trend that is expected from [1]. For the upper bound on the Lipschitz constant of the encoder network, the trend predicted by [1] is not seen at all. For both the encoder and the decoder network, the values are far larger than what would be expected from the reference paper.

Table 1: The estimated upper bounds for the Lipschitz constants of the networks trained on the sinc data.

Seed	$\sigma_{enc} = 1.0$	$\sigma_{enc} = 0.5$	$\sigma_{enc} = 0.1$	$\sigma_{\mathcal{N}} = 1.0$	$\sigma_{\mathcal{N}} = 0.5$	$\sigma_{\mathcal{N}} = 0.0$
1	1071.063	1854.309	4823.285	390.458	615.659	539.196
2	323.527	1799.821	4401.172	441.545	1107	491.179
3	320.910	1843.453	4588.153	794.016	576.915	453.370
Mean	571.8±432.3	1832.5±28.8	4604.2±211.5	542.0±219.7	766.5±295.5	494.6±43.0
Ref. [1]	2.2±0.2	5.2±0.3	17.9±3.2	13.9±2.7	24.6±1.7	29.8±2.2

Table 2: The estimated upper bounds for the Lipschitz constants of the networks trained on the CIFAR10 data.

Seed	$\sigma_{enc} = 1.0$	$\sigma_{enc} = 0.5$	$\sigma_{enc} = 0.1$	$\sigma_{\mathcal{N}} = 1.0$	$\sigma_{\mathcal{N}} = 0.5$	$\sigma_{\mathcal{N}} = 0.0$
1	241546.648	60271.665	720674.945	253960.934	146460.857	21507.984
2	297814.324	94801.987	590816.822	1152937.703	293395.569	29396.455
3	400212.123	99996.113	411597.148	1548858.732	266242.007	36214.927
Mean	313191±80443	85023±21592	574363±155194	985252±663535	235366±78182	29040±7360
Ref. [1]	17.9±1.2	19.1±1.2	27.3±0.3	4.7±0.2	5.6±0.6	8.5±0.8

In Table 3, the result for the networks trained on CelebA are given. As was the case for the CIFAR10 networks and - to a lesser extent - for the sinc networks, it can be seen that the upper bound of the Lipschitz constant of the decoder networks does not follow the trend that is expected from [1]. For the upper bound on the Lipschitz constant of the encoder network, the trend predicted by [1] is not seen at all. For both the encoder and the decoder network, the values are far larger than what would be expected from the reference paper.

Table 3: The estimated upper bounds for the Lipschitz constants of the networks trained on the CelebA data.

Seed	$\sigma_{enc} = 1.0$	$\sigma_{enc} = 0.5$	$\sigma_{enc} = 0.1$	$\sigma_{\mathcal{N}} = 1.0$	$\sigma_{\mathcal{N}} = 0.5$	$\sigma_{\mathcal{N}} = 0.0$
1	268246.715	89771.343	467391.831	430487.324	114471.238	45554.333
2	338731.299	82746.197	497943.796	408205.796	102431.839	41917.514
3	308692.390	98547.209	450681.648	332710.490	97116.120	41056.642
Mean	305223±35370	90355±7917	472006±23967	390468±51245	104673±8892	42843±2387
Ref. [1]	7.5±1.1	12.0±0.5	13.7±1.2	1.4±0.1	1.6±0.1	1.8±0.1

Generally, it can be noted that our result deviate largely from the results predicted and shown by [1]. This might be due to a difference in network architecture. However, we do not expect that to be the case as the theorems developed in [1] are expected to hold for our simple network. Another reason for the difference could be that the results from [1] are incorrect, which we, again, do not expect. The reason for this is that their results seem far more reasonable than ours. This might indicate a mistake in our code, however, the networks were validated to give sensible results, so the places where mistakes can occur seem to be limited. Another option is that there is some numerical instability. When estimating the Lipschitz constants, the code from [4] was used. The values from our network might have caused the problem to become ill-posed.

Too conclude, a larger investigation is needed and it is advisable to verify that our implementation is indeed corresponding to the implementation proposed in [1]. Furthermore different algorithms for estimating the Lipschitz constant can be considered to verify if the results are similar, or if numerical instability may be a cause for the large differences.

The code used for the above implementations is given in Appendix B.1.

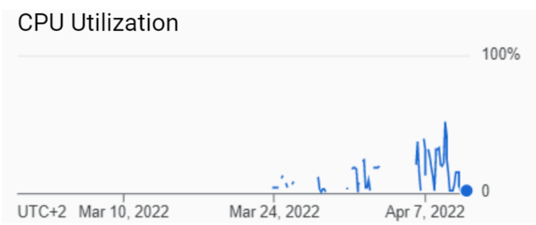
3 Maximum-damage attack

In this section we take a look at the adversarial robustness of the variatonal autoencoder. We obtain insight in this robustness by analysing an attacked or damaged image being fed to the trained network. Comparing the newly obtained decoded image with the original decoded image shows us how an attack alters the results. First we present the setup after which we will show the influence of attacks using examples. Lastly we numerically analyse the damage done and compare the results to the ones presented in the paper. The code used to obtain the results presented in this section can be found in Appendix B.2.

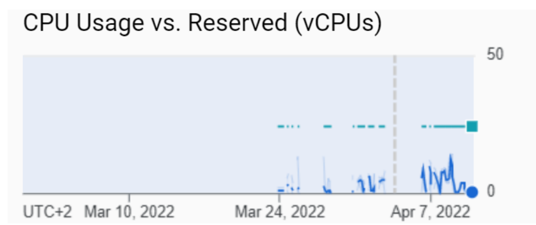
3.1 The network models

hardware specifications

To train the models and compute the maximum damage attack, computing power was acquired through google cloud. The created instance has 24 cores and a 96 GB memory. An overview of the CPU usage can be observed in Figure 8. This configuration proved to be sufficient for all tasks except training a variational autoencoder on CelebA which surprisingly gave memory errors. All tasks in total, about 130 computing hours were utilised (which includes installing software and performing test runs). All sigmoid function computations were done on regular local hardware with relatively negligible computation time.



(a) CPU Utilisation



(b) CPU Usage vs. Reserved

Figure 8: Virtual Machine metrics

The CIFAR-10 model training was performed by training four right and four left column models (note that right column is decoder, left column encoder) in sequence with sigmas (the mean) for 0.1, 0.2, 0.5 and 1.0 with 200 epochs per model and one epoch taking roughly one minute to complete. At peaks, 60% of the CPU was being used for this task.

The maximum damage attack was done on MATLAB and only for sigma 1.0 due to time constraints. It takes roughly 14 hours for a damage attack model to converge. This was partially caused by the fact that only 15% was being used at most for this task and we were unable to increase this value.

3.2 results

The CIFAR10 dataset (and the corresponding models trained in previous sections) will be used to analyse the influence of the attack. The attack used is the so called maximum-damage attack [5] and is defined in Equation (1). (Note that we abuse notation here since we denote the encoder and decoder as deterministic even though they have stochastical behaviour, this is done for sake of simplicity).

$$\delta^* = \arg \max_{\|\delta\|_2 \leq C} \|g(p(x+\delta)) - g(p(x))\|_2 \tag{1}$$

In this equation, x is the original image, δ is the damage done to the original image, $p(.)$ is the encoder operation, $g(.)$ denotes the decoder operation and C is the maximum norm of the attack. The maximum-damage attack thus alters the input image such that the decoded image is maximally damaged, where the maximum is the maximum in terms of the 2-norm.

Since the optimisation problem is not convex, a non convex optimisation technique is used to find a solution to this problem. Because of this, the optimisation will be computationally complex and it is very likely we will not find a global minimum. Since the optimisation includes calculating the output of the network given a lot of different (altered) input images, the optimisation will be even more computational complex.

Even though computing the maximum-damage attack is thus far from ideal, we will still directly optimise the problem posed in Equation (1). The problem is solved using the "interior-point" algorithm supported by the fmincon build-in function in MATLAB. The evaluation is done on the deterministic part of the network, the stochastic part of the network is ignored (as is also done in the paper). In practise this means that the latent space variables are simply equal to the mean values derived during training.

Two example results can be found in Figures 9 for a maximum norm of $C = 10$ and $C = 20$ respectively. In Figures 9a the original image is shown and the corresponding output of the networks fed with these images is shown in Figures 9d. Applying the maximum-damage attack to the images gives us the decoded damaged image shown in Figures 9e and 9f. For sake of completeness, the damaged input images are given in Figures 9b and 9c.

It is clear that the trained network is not able to reproduce the input image. Even though this will limit the relevance and trustworthiness regarding our findings on the maximum-damage attack, we will still analyse the impact and performance of the maximum-damage attack given these results.

As can be seen from the damaged input images (Figures 9b and 9c), the maximum-damage attack severely changes the input image even though it is limited by maximum norm C. From the decoded original and decoded damaged images we can see that the attack drastically alters the decoded images, because of this we conclude that the maximum damage attack works as intended. We can not guarantee that we found a global optimum but it is clear the algorithm finds a solution with the intended outcome.

Visual inspection suggests that the attack inverted the decoded image, the black parts are now coloured and vice versa. We can not prove that this is a general property but it is a noteworthy observation for this particular example.

Comparing the results for a different maximum norm C, clearly shows the expected influence of the norm. Figure 9b corresponding to $C = 10$ is less damaged compared to Figure 9c corresponding to $C = 20$. What is surprising however is how little this big difference in damage at the input images translates to the difference between the damaged output images. A difference between Figures 9e and 9f is noticeable but relatively insignificant. This suggests that the network is robust against attack size. Once again, we can not prove this but it is a noteworthy observation.

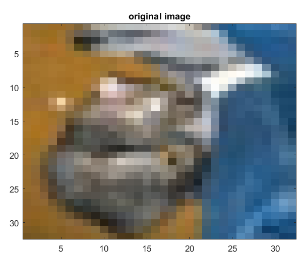
Finally, we will take a look at the likelihood degradation on the original decoded and damaged decoded images, corresponding to Figure 5 of the paper. The results are shown in Table 4 (We use a table instead of a figure since we have very little results). As can be seen from this figure, our results correspond with the observation above, the damage difference between the two different norms is minimal. Opposed to our findings, the paper shows that there is a substantial difference between the results corresponding to the two different norms. Apart from this, the size of the likelihood degradation is similar.

	C = 10	C = 20
Paper log likelihood degradation	-0.6667	-2.5882
Found log likelihood degradation	-2.3019	-2.4052

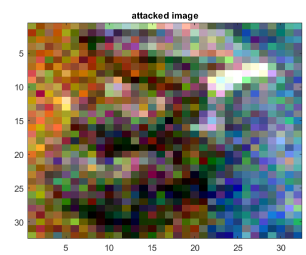
Table 4: log likelihood degradation results.

In conclusion, our findings complement the findings presented in the paper up to a certain extend. The magnitude of the likelihood degradation is similar but the trend found for different norms C differs from the paper's findings. Since our findings are based on too little and unreliable data, no hard conclusions on this can be drawn.

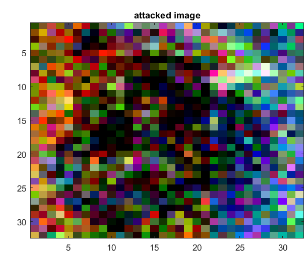
To improve on the found results two major aspects should be considered. First the optimisation procedure should be thoroughly evaluated. The optimisation problem could potentially be rewritten to improve performance, on top of that, the used method can also be analysed and changed to improve results. Secondly, more results should be generated to be able to draw solid conclusions. Currently, our analyses suggests some properties but we are far from able to draw concise conclusions. Apart from these major aspects, it would also be worthwhile to analyse the influence of the quality of the network.



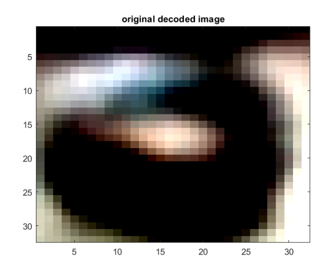
(a) Original input image



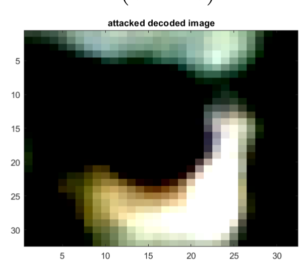
(b) Damaged input image ($C = 10$)



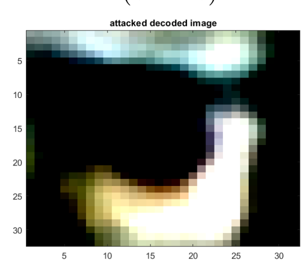
(c) Damaged input image ($C = 20$)



(d) Original image fed to the network



(e) Damaged image fed to the network ($C = 10$)



(f) Damaged image fed to the network ($C = 20$)

Figure 9: Maximum damage attack applied to the network generated with $\sigma = 1.0$, maximum norm of the attack is $C = 10$ or $C = 20$.

4 Conclusion

To summarise the report, a virtual machine was set up to be train the variational autoencoder networks. The lipschitz constants for two cases was considered, first for an encoder without noise injections, and secondly for an encoder with zero-mean Gaussian noise injections. For the case without the noise injection, the Lipschitz constant of the decoder was calculated, whereas for the case with Gaussian noise, the Lipschitz constant of the encoder was calculated. It was found for the Sinc function, the reproduced results closely resembled their inputs, however had smaller amplitudes compared to its input. CIFAR10 struggled much more in reproducing its inputs, which was expected as the dataset was not intended for this use and is instead directed to classification. Regarding the Lipschitz constant, it was found that the values produced did not reflect [1]. It is not understood what the source for the deviance could be and is a recommendation for further study.

Then, the adversarial robustness of the variational autoencoder was considered by comparing outputs of attacked images with the original ones (using CIFAR10). The insufficiency in data causes unreliability in our results. We observe that there is not much impact on the log likelihood degradation by changing norms, this goes against the findings of [1], where the log likelihood was impacted greatly by changing norms. A suggestion for future research would be increasing the data we had available as the lack in data proved to be the greatest source of unreliability for our analysis.

To conclude, this report was not able to completely replicate the findings of [1], however it was able to reproduce some trends, as well as provide suggestions to further attempt to replicate the results presented.

A. Task division

A.0.1 Austin

Technical: Managed the virtual machine, trained CIFAR-10 models and did data processing of the maximum damage attack.

Non-Technical: wrote report chapter 4.1.

A.0.2 Dimme

Technical: Wrote code for training models for estimating Lipschitz constant/verifying them/storing the weights. Wrote code for generating sinc data. Trained the models needed for Lipschitz estimation. Estimated upper bounds on the Lipschitz constants using LipSDP.

Modified previously mentioned code for training maximum damage attack networks.

Non-Technical: Wrote report chapter 3. Figured out how to install required software (CVX, MOSEK). Figured out how to use CelebA dataset.

A.0.3 Mustafa:

Technical: In the beginning attempted to make a network from scratch, however after we where given a sample code I worked on settings up the virtual machine as well implementation the training models.

Non-technical: Wrote Chapter 1 and 4.

A.0.4 Richard:

Technical: In the first weeks, worked on the code to train a well performing network. After that, worked out the theory regarding the maximum-damage attack. Wrote the MATLAB code and the subfunctions used to perform and present the maximum-damage attack given a trained network (both for sinc and cifar data).

Non-technical: Draw conclusions on the maximum-damage attack results and write chapter 4.2. Find sources on the implementation of variational autoencoders.

B. Code used to obtain results presented in the report

B.1 Lipschitz constant

In this section, the code used for estimating the Lipschitz constants is given. The code consists of three steps. The first step is to train the networks, for which the code is given in Section B.1.1. The second step is to verify the results, for which the code is given in Section B.1.2. The last step is to store the weights, for which the code is given in Section B.1.3. After the weights are stored, LipSDP (refer to [3, 4]) can be used to estimate upper bounds on the Lipschitz constants of the network.

B.1.1 Step 1: training the networks

The code used for training the networks is given below. It is based on [1], [6], [7], [8] and [9].

```
"""
Date:        21-03-2022
Last edit:   24-03-2022
Author:      Austin, Richard, Mustafa, Dimme

Based on:    [1], [2], [3], [4], [5]
Trains:      3x6 models on celeba, cifar, sinc
Descr:       Part of Code for Table 1 of the paper " Variational Autoencoders: A Harmonic Perspective"
             decoder is deterministic (given a latent space)

             Where the first three models have no noise injections on the input, and fixed encoder standard deviation (!) of
                 [1.0, 0.5, 0.1]
             Where the last three models have fixed encoder standard deviation (!) 0.5 and noise injections of
                 a standard normal distribution with zero mean and standard deviation of [1.0, 0.5, 0.0]

Some notes:
    (a) inside the folder where you put this file, make the following two folders:
            (1) Model (used for storing model state)
            (2) Optim (used for storing optimiser state)

Sources:
    [1] examples from https://github.com/didriknielsen/survae_flow
    [2] https://avandekleut.github.io/vae/
    [3] The paper: Variational Autoencoders: A Harmonic Perspective
    [4] Lab 8 of CS4240-Deep Learning course
    [5] https://medium.com/dataseries/variational-autoencoder-with-pytorch-2d359cbf027b
"""

import torch
import torchvision.datasets as datasets

```

```
from torch import nn
from survae.data.loaders.image import CelebA
from torch.utils.data import DataLoader
from torchvision.transforms import ToTensor
from torch.optim import Adam
from pytictoc import TicToc

#use one of three seeds for reproducable results+calculating standard deviation
#seeds obtained using torch.seed() three times.

torch.manual_seed(14609714069366804553)
#torch.manual_seed(10209623728859046282)
#torch.manual_seed(13978253786968215756)

device = 'cuda' if torch.cuda.is_available() else 'cpu'
t = TicToc()

#User settings
select = [2]                                    #select which models, 0: CelebA, 1: Cifar, 2: Sinc; [0, 1, 2] runs all
epochs, lr = [200, 200 , 100], [1e-3, 1e-3, 1e-3]  #number of epochs, learning rate; epochs[0] corresponds to CelebA, etc.
load_old = [0, 0, 1]                            #set to one if you want to continue training. load_old[0] corresponds to CelebA, etc.
norm = [0, 1, 2]                                #set type of normalisation: 0: 8 bit integer to float [-1, 1]; 1: float to float [-1, 1]; 2: no normalisation


#Define model architecture
################################################################################
class VariationalEncoder(nn.Module):
    def __init__(self, latent_dims, nChan, imSizeH, imSizeV):
        super(VariationalEncoder, self).__init__()
        self.input = nn.Flatten(1)
        self.dense1 =  nn.Linear(imSizeH*imSizeV*nChan, 256)
        self.dense2 =  nn.Linear(256, 256)
        self.dense3 =  nn.Linear(256, 256)
        self.outputMu =  nn.Linear(256, latent_dims)
        #self.outputSig = nn.Linear(256, latent_dims)       #this layer is not needed as sigma is fixed

        self.N = torch.distributions.Normal(0, 1)
        self.kl = 0

    def forward(self, x, sigma_en):
        x = self.input(x)
        x = torch.sigmoid(self.dense1(x))
        x = torch.sigmoid(self.dense2(x))
        x = torch.sigmoid(self.dense3(x))
        mu =  self.outputMu(x)
        #sigma = torch.exp(self.outputSig(x))               #this layer is not needed as sigma is fixed
```

```
        z = mu + (sigma_en**2)*self.N.sample(mu.shape)      #combination of encoders output of layer2, layer3
        self.kl = (sigma_en**2 + mu**2 - torch.log(sigma_en) - 1/2).sum()
        return z

class Decoder(nn.Module):
    def __init__(self, latent_dims, nChan, imSizeH, imSizeV):
        super(Decoder, self).__init__()
        self.dense1 = nn.Linear(latent_dims, 256)
        self.dense2 = nn.Linear(256, 256)
        self.dense3 = nn.Linear(256, 256)
        self.output = nn.Linear(256, nChan*imSizeH*imSizeV)
        self.outputShape = nn.Unflatten(dim=1, unflattened_size = (nChan, imSizeH, imSizeV))

    def forward(self, z):
        z = torch.sigmoid(self.dense1(z))
        z = torch.sigmoid(self.dense2(z))
        z = torch.sigmoid(self.dense3(z))
        z = self.output(z)
        z = self.outputShape(z)
        return z

class fnc_get_model(nn.Module):
    def __init__(self, latent_dims, nChan, imSizeH, imSizeV):
        super(fnc_get_model, self).__init__()
        self.encoder = VariationalEncoder(latent_dims, nChan, imSizeH, imSizeV)
        self.decoder = Decoder(latent_dims, nChan, imSizeH, imSizeV)

    def forward(self, x, sigma_en):
        z = self.encoder(x, sigma_en)
        return self.decoder(z)
###############################################################################

##############################################################################3
#function to train model (left column)
def fnc_train(epochs, sigma_en, sigma_in, norm, nChan, imSizeH, imSizeV):
    if norm == 0:
        scale, div, offset = 2, 255, 1
    elif norm == 1:
        scale, div, offset = 2, 1, 1
    else:
        scale, div, offset = 1, 1, 0

    if sigma_in == 0:
        for epoch in range(epochs):
            l = 0.0
            for i, x in enumerate(train_loader):
                if isinstance(x, list):
                    x = x[0]
                x = scale*(x/div)-offset   #normalise between [-1,1]
                x = x.to(device)
```

```
                optimizer.zero_grad()
                x_hat = model(x, sigma_en)
                loss = ((x - x_hat)**2).sum() + model.encoder.kl
                loss.backward()
                optimizer.step()
                l += loss.detach().cpu().item()
            print('Epoch: {}/{}, Loss: {:.3f}'.format(epoch+1, epochs, l/(i+1), end='\r'))
    else:
        for epoch in range(epochs): #this part is based on [3,4]
            l = 0.0
            for i, x in enumerate(train_loader):
                if isinstance(x, list):
                    x = x[0]
                noise = torch.normal(0,sigma_in, (nChan,imSizeH,imSizeV))
                x = scale*(x/div)-offset   #normalise between [-1,1]
                x = x+noise
                x = x.to(device)
                optimizer.zero_grad()
                x_hat = model(x, sigma_en)
                loss = ((x - x_hat)**2).sum() + model.encoder.kl #b
                loss.backward()
                optimizer.step()
                l += loss.detach().cpu().item()
            print('Epoch: {}/{}, Loss: {:.3f}'.format(epoch+1, epochs, l/(i+1), end='\r'))

###############################################################################

########## Functions to get training data #####################################
def fnc_getTrain_loader(selector):
    if selector == 0:
        data = CelebA()
        train_loader = DataLoader(dataset=data.train, batch_size=256, shuffle=True, num_workers=8, drop_last=True)
        t.toc()
    elif selector == 1:
        cifar_trainset = datasets.CIFAR10(root='../DATA', train=True, download=True, transform=ToTensor())
        train_loader = DataLoader(dataset=cifar_trainset, batch_size=256, shuffle=True, num_workers=8, drop_last=True)
        t.toc()
    else:
        w, t_s, t_e, N, M = 5, -1, 1, 172, 8*512            #frequency, start time, end time, data of track, number of tracks
        ft, t_ax = sinc(w, t_s, t_e, N, M)                  #get raw data
        data, offset, scale = preprocess(ft, t_ax)          #preprocess raw data. Max of data in [-1, 1]
        train_loader = DataLoader(dataset=data, batch_size=256, shuffle=True, num_workers=8, drop_last=True)
        t.toc()
    return train_loader

#function to get sinc data
```

```python
def sinc(w, t_s, t_e, N, M):
    t_ax = 2*torch.rand(M, 1, N, 1) - 1
    t_ax = torch.sort(t_ax, dim=2)
    t_ax = t_ax[0]
    f_eval = torch.sinc(w*t_ax)
    return f_eval, t_ax

#function to preprocess sinc data
def preprocess(ft, t):
    offset = torch.min(ft)
    ft = ft - offset                #make minimum zero for both t and f(t)
    scale = torch.max(ft)
    ft = ft/scale                   #make max one for both t and f(t)
    ft = 2*ft - 1                   #from t, f(t) in [0,1] to t, f(t) in [-1, 1]
    out = torch.cat((ft, t),-1)
    return out, offset, scale           #return data, offset and scale. Note that the 2* and -1 are not returned
###############################################################################

###############################################################################
def fnc_getStore_and_Load(selector):
    if selector==0: #CelebA
        #Save/load paths for CelebA, left column
        path_ll = ["Models/modelL1-sigma-1_0", "Models/modelL1-sigma-0_5", "Models/modelL1-sigma-0_1"]   #Load locations
        path_sl = ["Models/modelL1-sigma-1_0", "Models/modelL1-sigma-0_5", "Models/modelL1-sigma-0_1"]   #Store locations
        path_lol = ["Optim/modelL1-sigma-1_0", "Optim/modelL1-sigma-0_5", "Optim/modelL1-sigma-0_1" ]    #Load locations
        path_sol = ["Optim/modelL1-sigma-1_0", "Optim/modelL1-sigma-0_5", "Optim/modelL1-sigma-0_1" ]    #Store locations

        #Save/load paths for CelebA, right column
        path_lr = ["Models/modelR1-sigma-1_0", "Models/modelR1-sigma-0_5", "Models/modelR1-sigma-0_0"]   #Load locations
        path_sr = ["Models/modelR1-sigma-1_0", "Models/modelR1-sigma-0_5", "Models/modelR1-sigma-0_0"]   #Store locations
        path_lor = ["Optim/modelR1-sigma-1_0", "Optim/modelR1-sigma-0_5", "Optim/modelR1-sigma-0_0" ]    #Load locations
        path_sor = ["Optim/modelR1-sigma-1_0", "Optim/modelR1-sigma-0_5", "Optim/modelR1-sigma-0_0" ]    #Store locations

    elif selector == 1: #Cifar
        #Save/load paths for CIFAR, left column
        path_ll = ["Models/modelL2-sigma-1_0", "Models/modelL2-sigma-0_5", "Models/modelL2-sigma-0_1"]   #Load locations
        path_sl = ["Models/modelL2-sigma-1_0", "Models/modelL2-sigma-0_5", "Models/modelL2-sigma-0_1"]   #Store locations
        path_lol = ["Optim/modelL2-sigma-1_0", "Optim/modelL2-sigma-0_5", "Optim/modelL2-sigma-0_1" ]    #Load locations
        path_sol = ["Optim/modelL2-sigma-1_0", "Optim/modelL2-sigma-0_5", "Optim/modelL2-sigma-0_1" ]    #Store locations
```

```python

        #Save/load paths for CIFAR, right column
        path_lr = ["Models/modelR2-sigma-1_0", "Models/modelR2-sigma-0_5", "Models/modelR2-sigma-0_0"]   #Load locations
        path_sr = ["Models/modelR2-sigma-1_0", "Models/modelR2-sigma-0_5", "Models/modelR2-sigma-0_0"]   #Store locations
        path_lor = ["Optim/modelR2-sigma-1_0", "Optim/modelR2-sigma-0_5", "Optim/modelR2-sigma-0_0" ]    #Load locations
        path_sor = ["Optim/modelR2-sigma-1_0", "Optim/modelR2-sigma-0_5", "Optim/modelR2-sigma-0_0" ]    #Store locations

    else: #Sinc
        #Save/load paths for SINC, left column
        path_ll = ["Models/modelL3-sigma-1_0", "Models/modelL3-sigma-0_5", "Models/modelL3-sigma-0_1"]   #Load locations
        path_sl = ["Models/modelL3-sigma-1_0", "Models/modelL3-sigma-0_5", "Models/modelL3-sigma-0_1"]   #Store locations
        path_lol = ["Optim/modelL3-sigma-1_0", "Optim/modelL3-sigma-0_5", "Optim/modelL3-sigma-0_1" ]    #Load locations
        path_sol = ["Optim/modelL3-sigma-1_0", "Optim/modelL3-sigma-0_5", "Optim/modelL3-sigma-0_1" ]    #Store locations

        #Save/load paths for SINC, right column
        path_lr = ["Models/modelR3-sigma-1_0", "Models/modelR3-sigma-0_5", "Models/modelR3-sigma-0_0"]   #Load locations
        path_sr = ["Models/modelR3-sigma-1_0", "Models/modelR3-sigma-0_5", "Models/modelR3-sigma-0_0"]   #Store locations

        path_lor = ["Optim/modelR3-sigma-1_0", "Optim/modelR3-sigma-0_5", "Optim/modelR3-sigma-0_0" ]    #Load locations
        path_sor = ["Optim/modelR3-sigma-1_0", "Optim/modelR3-sigma-0_5", "Optim/modelR3-sigma-0_0" ]    #Store locations
    return path_ll, path_sl, path_lol, path_sol, path_lr, path_sr, path_lor, path_sor

##############################################################################

##############################################################################
#some general model settings
sigma_in1 = torch.tensor(0)                     #input standard deviation, left column
sigma_enc1 = torch.tensor([1.0, 0.5, 0.1])  #encoder standard deviation, left column
sigma_in2 = torch.tensor([1.0, 0.5, 0.0])   #input standard deviation, right column
sigma_enc2 = torch.tensor(0.5)                  #encoder standard deviation, right column

latent_dims = [64, 64, 1]                       #latent dimensions
imSizeH = [64, 32, 172]                         #image size 1 (horizontal)
imSizeV = [64, 32, 2]                           #image size 2 (vertical)
nChan = [3, 3, 1]                               #number of channels

#start timer
t.tic()

for k in select:    #k denotes the data
```

```
    print("the selected model is {}. Note: 0=celeba, 1=cifar, 2=sinc".format(k))
    print("Step 1: Loading data")
    train_loader = fnc_getTrain_loader(k)
    path_ll, path_sl, path_lol, path_sol, path_lr, path_sr, path_lor, path_sor = fnc_getStore_and_Load(k)

    print(" ")
    print("training for left column")
    for i in range(3): #i denotes the settings for sigma_enc
        model = fnc_get_model(latent_dims=latent_dims[k], nChan=nChan[k], imSizeH=imSizeH[k], imSizeV=imSizeV[k])
        optimizer = Adam(model.parameters(), lr=lr[k])
        if load_old[k] == 1:
            model.load_state_dict(torch.load(path_ll[i]))
            optimizer.load_state_dict(torch.load(path_lol[i]))

        model.train()
        fnc_train(epochs=epochs[k], sigma_en=sigma_enc1[i], sigma_in=sigma_in1, norm=norm[k], nChan=nChan[k], imSizeH=imSizeH[k], imSizeV=imSizeV[k])

        torch.save(model.state_dict(), path_sl[i])           #Save current state of model
        torch.save(optimizer.state_dict(), path_sol[i])      #Save current state of optimiser
        print("Saved model for data {}: sigma_enc={}, sigma_in={}".format(k, sigma_enc1[i], sigma_in1))
        print(" ")
    print("Done training for left column")

    print(" ")

    print("Training for right column")
    for i in range(3): #i denotes the settings for sigma_in
        model = fnc_get_model(latent_dims=latent_dims[k], nChan=nChan[k], imSizeH=imSizeH[k], imSizeV=imSizeV[k])
        optimizer = Adam(model.parameters(), lr=lr[k])
        if load_old[k] == 1:
            model.load_state_dict(torch.load(path_lr[i]))
            optimizer.load_state_dict(torch.load(path_lor[i]))

        model.train()
        fnc_train(epochs=epochs[k], sigma_en=sigma_enc2, sigma_in=sigma_in2[i], norm=norm[k], nChan=nChan[k], imSizeH=imSizeH[k], imSizeV=imSizeV[k])

        torch.save(model.state_dict(), path_sr[i])           #Save current state of model
        torch.save(optimizer.state_dict(), path_sor[i])      #Save current state of optimiser

        print("Saved model for data {}: sigma_enc={}, sigma_in={}".format(k, sigma_enc2, sigma_in2[i]))
        print(" ")
    print("Done training for right column")

    print(" ")
```

B.1.2 Step 2: Verify models

The code used for verifying the models is presented below. It is based on [1], [6], [7], [8] and [9].

```python
"""
Date:   25-03-2022
Author: Austin, Mustafa, Richard, Dimme
Based on: [1], [2], [3], [4], [5]
Descr:  Part of Code for Table 1 of the paper " Variational Autoencoders: A Harmonic Perspective"
        Stores figures for verification purposes.

Sources:
    [1] examples from https://github.com/didriknielsen/survae_flow
    [2] https://avandekleut.github.io/vae/
    [3] The paper: Variational Autoencoders: A Harmonic Perspective
    [4] Lab 8 of CS4240-Deep Learning course
    [5] https://medium.com/dataseries/variational-autoencoder-with-pytorch-2d359cbf027b

Notes:
    (a) you need to have trained the models before using this file, and stored at the proper location.
    (b) you need to have a folder Figures inside the directory where this .py file is stored.
"""

import torch
import torchvision.datasets as datasets
import torchvision.utils as vutils
import matplotlib.pyplot as plt
import numpy as np

from torch import nn
from survae.data.loaders.image import CelebA
from torch.utils.data import DataLoader
from torchvision.transforms import ToTensor
from skimage import color

#User settings
select = [2]                    #select which models, 0: CelebA, 1: Cifar, 2: Sinc


plt.rc('font', size=14) #controls default text size

#Define model architecture
###############################################################################
class VariationalEncoder(nn.Module):
    def __init__(self, latent_dims, nChan, imSizeH, imSizeV):
        super(VariationalEncoder, self).__init__()
        self.input = nn.Flatten(1)
        self.dense1 =  nn.Linear(imSizeH*imSizeV*nChan, 256)
        self.dense2 =  nn.Linear(256, 256)
        self.dense3 =  nn.Linear(256, 256)
        self.outputMu =  nn.Linear(256, latent_dims)
```

```
        #self.outputSig = nn.Linear(256, latent_dims)       #this layer is not needed as sigma is fixed

        self.N = torch.distributions.Normal(0, 1)
        self.kl = 0

    def forward(self, x, sigma_en):
        x = self.input(x)
        x = torch.sigmoid(self.dense1(x))
        x = torch.sigmoid(self.dense2(x))
        x = torch.sigmoid(self.dense3(x))
        mu =  self.outputMu(x)
        #sigma = torch.exp(self.outputSig(x))               #this layer is not needed as sigma is fixed

        z = mu + (sigma_en**2)*self.N.sample(mu.shape)      #combination of encoders output of layer2, layer3
        self.kl = (sigma_en**2 + mu**2 - torch.log(sigma_en) - 1/2).sum()
        return z

class Decoder(nn.Module):
    def __init__(self, latent_dims, nChan, imSizeH, imSizeV):
        super(Decoder, self).__init__()
        self.dense1 = nn.Linear(latent_dims, 256)
        self.dense2 = nn.Linear(256, 256)
        self.dense3 = nn.Linear(256, 256)
        self.output = nn.Linear(256, nChan*imSizeH*imSizeV)
        self.outputShape = nn.Unflatten(dim=1, unflattened_size = (nChan, imSizeH, imSizeV))

    def forward(self, z):
        z = torch.sigmoid(self.dense1(z))
        z = torch.sigmoid(self.dense2(z))
        z = torch.sigmoid(self.dense3(z))
        z = self.output(z)
        z = self.outputShape(z)
        return z

class fnc_get_model(nn.Module):
    def __init__(self, latent_dims, nChan, imSizeH, imSizeV):
        super(fnc_get_model, self).__init__()
        self.encoder = VariationalEncoder(latent_dims, nChan, imSizeH, imSizeV)
        self.decoder = Decoder(latent_dims, nChan, imSizeH, imSizeV)

    def forward(self, x, sigma_en):
        z = self.encoder(x, sigma_en)
        return self.decoder(z)
##############################################################################

##############################################################################
def fnc_getStore_and_Load(selector):
    if selector==0: #CelebA
        #Save/load paths for CelebA, left column
```

```
        path_ll = ["Models/modelL1-sigma-1_0", "Models/modelL1-sigma-0_5", "Models/modelL1-sigma-0_1"]                            #Load locations
        path_sfl = ["Figures/FigL1-sigma-1_0-ev.png", "Figures/FigL1-sigma-0_5-ev.png", "Figures/FigL1-sigma-0_1-ev.png"]       #Store locations evaluation
        path_sdl = ["Figures/FigL1-sigma-1_0-data.png", "Figures/FigL1-sigma-0_5-data.png", "Figures/FigL1-sigma-0_1-data.png"] #Store locations data

        #Save/load paths for CelebA, right column
        path_lr = ["Models/modelR1-sigma-1_0", "Models/modelR1-sigma-0_5", "Models/modelR1-sigma-0_0"]               #Load locations
        path_sfr = ["Figures/FigR1-sigma-1_0-ev.png", "Figures/FigR1-sigma-0_5-ev.png", "Figures/FigR1-sigma-0_0-ev.png"]       #Store locations evaluation
        path_sdr = ["Figures/FigR1-sigma-1_0-data.png", "Figures/FigR1-sigma-0_5-data.png", "Figures/FigR1-sigma-0_0-data.png"] #Store locations data

    elif selector == 1: #Cifar
        #Save/load paths for CIFAR, left column
        path_ll = ["Models/modelL2-sigma-1_0", "Models/modelL2-sigma-0_5", "Models/modelL2-sigma-0_1"]                            #Load locations
        path_sfl = ["Figures/FigL2-sigma-1_0-ev.png", "Figures/FigL2-sigma-0_5-ev.png", "Figures/FigL2-sigma-0_1-ev.png"]       #Store locations evaluation
        path_sdl = ["Figures/FigL2-sigma-1_0-data.png", "Figures/FigL2-sigma-0_5-data.png", "Figures/FigL2-sigma-0_1-data.png"] #Store locations data

        #Save/load paths for CIFAR, right column
        path_lr = ["Models/modelR2-sigma-1_0", "Models/modelR2-sigma-0_5", "Models/modelR2-sigma-0_0"]                            #Load locations
        path_sfr = ["Figures/FigR2-sigma-1_0-ev.png", "Figures/FigR2-sigma-0_5-ev.png", "Figures/FigR2-sigma-0_0-ev.png"]       #Store locations evaluation
        path_sdr = ["Figures/FigR2-sigma-1_0-data.png", "Figures/FigR2-sigma-0_5-data.png", "Figures/FigR2-sigma-0_0-data.png"] #Store locations data

    else: #Sinc
        #Save/load paths for SINC, left column
        path_ll = ["Models/modelL3-sigma-1_0", "Models/modelL3-sigma-0_5", "Models/modelL3-sigma-0_1"]                            #Load locations
        path_sfl = ["Figures/FigL3-sigma-1_0-ev.png", "Figures/FigL3-sigma-0_5-ev.png", "Figures/FigL3-sigma-0_1-ev.png"]       #Store locations evaluation
        path_sdl = ["not used", "not used", "not used"] #Store locations data

        #Save/load paths for SINC, right column
        path_lr = ["Models/modelR3-sigma-1_0", "Models/modelR3-sigma-0_5", "Models/modelR3-sigma-0_0"]                            #Load locations
        path_sfr = ["Figures/FigR3-sigma-1_0-ev.png", "Figures/FigR3-sigma-0_5-ev.png", "Figures/FigR3-sigma-0_0-ev.png"]       #Store locations evaluation
        path_sdr = ["not used", "not used", "not used"] #Store locations data

    return path_ll, path_lr, path_sfl, path_sfr, path_sdl, path_sdr
###############################################################################

###############################################################################
def fnc_getValid_loader(selector):
```

```
    if selector == 0:
        data = CelebA()
        valid_loader = DataLoader(dataset=data.valid, batch_size=256, shuffle=False, num_workers=8, drop_last=True)
    elif selector == 1:
        cifar_trainset = datasets.CIFAR10(root='../DATA', train=False, download=True, transform=ToTensor())
        valid_loader = DataLoader(dataset=cifar_trainset, batch_size=256, shuffle=False, num_workers=8, drop_last=True)
    else:
        w, t_s, t_e, N, M = 5, -1, 1, 172, 2        #frequency, start time, end time, number of samples
        data_p = sinc(w, t_s, t_e, N, M)            #get raw data
        data, offset, scale = preprocess(data_p)    #preprocess raw data. Using scale and offset og data can be returned
        valid_loader = data
    return valid_loader

#function to get sinc data
def sinc(w, t_s, t_e, N, M):
    t_ax = torch.linspace(t_s, t_e, N)
    t_ax = t_ax.unsqueeze(1)
    t_ax = t_ax.unsqueeze(0)
    t_ax = t_ax.unsqueeze(0)
    f_eval = torch.sinc(w*t_ax)
    out = torch.cat((f_eval, t_ax),-1)
    return out

#function to preprocess sinc data
def preprocess(data):
    offset = torch.min(data, dim=2)
    data = data - offset.values         #make minimum zero for both t and f(t)
    scale = torch.max(data, dim=2)
    data = data/scale.values            #make max one for both t and f(t)
    data = 2*data - 1                   #from t, f(t) in [0,1] to t, f(t) in [-1, 1]
    return data, offset, scale          #return data, offset and scale. Note that the 2* and -1 are not returned
###############################################################################

###############################################################################
def fnc_plot(model, data_name, result_name, valid_loader, selector):
    N = 4
    fft = 0
    if selector == 0: #CELEBA
        img = next(iter(valid_loader))[:N]
        img = (2*img/255.0)-1 #Normalise between [-1, 1]
        samples = torch.zeros(N,3,64,64)

        #some extra for when FFT+converting to gray scale is needed
        samples2 = np.zeros((N, 64, 64, 3))
        img2 = np.zeros((N,64, 64, 3))
        samplesGray = np.zeros((N, 64, 64))   #store gray scale
```

```
        imgGray = np.zeros((N, 64, 64))       #store gray scale
        SAMPLES = np.zeros((N, 64, 64))       #store freq domain
        IMG = np.zeros((N,64,64))             #store freq domain

        for i in range(N):
            samples[i] = model(img[i].unsqueeze(dim=0), torch.tensor(0))

        img = (img+1)/2          #Normalise between [0,1]
        samples = (samples+1)/2 #Normalise between [0,1]

        if fft==1:
            samples = samples.detach().numpy()
            img = img.detach().numpy()

            for i in range(N): #permute submatrices
                samples2[i] = np.transpose(samples[i], (1, 2, 0))
                img2[i] = np.transpose(img[i], (1, 2, 0))

            for i in range(N): #convert to gray scale
                samplesGray[i] = color.rgb2gray(samples2[i])
                imgGray[i] = color.rgb2gray(img2[i])
                SAMPLES[i] = 20*np.log10(abs(np.fft.fft2(samplesGray[i])))
                IMG[i] = 20*np.log10(abs(np.fft.fft2(imgGray[i])))

            plt.subplot(2, 2, 1)
            plt.imshow(SAMPLES[0], vmin=-65, vmax=65, cmap='jet', aspect='auto')
            plt.colorbar()
            plt.subplot(2, 2, 2)
            plt.imshow(SAMPLES[1], vmin=-65, vmax=65, cmap='jet', aspect='auto')
            plt.colorbar()
            plt.subplot(2, 2, 3)
            plt.imshow(SAMPLES[2], vmin=-65, vmax=65, cmap='jet', aspect='auto')
            plt.colorbar()
            plt.subplot(2, 2, 4)
            plt.imshow(SAMPLES[3], vmin=-65, vmax=65, cmap='jet', aspect='auto')
            plt.colorbar()
            plt.show()

        else:
            vutils.save_image(img.cpu().float(), fp=data_name, nrow=2)
            vutils.save_image(samples.cpu().float(), fp=result_name, nrow=2)
        return img, samples

    elif selector == 1: #CIFAR
        img = next(iter(valid_loader))[:N]
        img = img[0]
        img = (2*img)-1 #Normalise between [-1, 1]
        samples = torch.zeros(N,3,32, 32)
        img2 = torch.zeros(N,3,32, 32)

        for i in range(N):
            samples[i] = model(img[i].unsqueeze(dim=0), torch.tensor(0))
```

```
            img2[i] = img[i]

        img2 = (img2+1)/2         #Normalise between [0,1]
        samples = (samples+1)/2 #Normalise between [0,1]
        vutils.save_image(img2.cpu().float(), fp=data_name, nrow=2)
        vutils.save_image(samples.cpu().float(), fp=result_name, nrow=2)
        return img2, samples

    else: #sinc
        x = next(iter(valid_loader))[:172]
        sample = torch.zeros(1, 1, 172,2)
        sample = model(x, torch.tensor(0))
        x = torch.squeeze(x)
        sample = torch.squeeze(sample)
        x = x.detach().numpy()
        sample = sample.detach().numpy()
        fig = plt.figure( )
        plt.plot(x[:,1], x[:, 0], label='input')
        plt.plot(sample[:, 1], sample[:, 0], label='reconstruction')
        plt.xlabel("Time [-]")
        plt.ylabel("sinc(5t), normalised to [-1, 1]")
        plt.legend(loc='upper right')
        fig.savefig(result_name)
        return x, sample
###############################################################################

latent_dims = [64, 64, 1]       #latent dimensions
imSizeH = [64, 32, 172]         #image size 1 (horizontal)
imSizeV = [64, 32, 2]           #image size 2 (vertical)
nChan = [3, 3, 1]               #number of channels

for k in select:
    print("the selected model is {}. Note: 0=celeba, 1=cifar, 2=sinc".format(k))
    print(" ")
    print("Step 1: Loading data")
    valid_loader = fnc_getValid_loader(k)
    path_ll, path_lr, path_sfl, path_sfr, path_sdl, path_sdr = fnc_getStore_and_Load(k)     #get location to load models, store figures

    print("Saving figures for left column")
    for j in range(3):
        model = fnc_get_model(latent_dims=latent_dims[k], nChan=nChan[k], imSizeH=imSizeH[k], imSizeV=imSizeV[k])
        model.load_state_dict(torch.load(path_ll[j]))
        model.eval()
        data, samples = fnc_plot(model, path_sdl[j], path_sfl[j], valid_loader, k)

    print("Saving figures for right column")

    for j in range(3):
        model = fnc_get_model(latent_dims=latent_dims[k], nChan=nChan[k], imSizeH=imSizeH[k], imSizeV=imSizeV[k])
```

```python
        model.load_state_dict(torch.load(path_lr[j]))
        model.eval()

        data, samples = fnc_plot(model, path_sdr[j], path_sfr[j], valid_loader, k)
    print(" ")
```

B.1.3 Step 3: store mat

The code used for storing the network weights is given below. It is based on [1], [6], [7], [8] and [9] and [4].

```python
"""
Date:   25-03-2022
Author: Austin, Mustafa, Richard, Dimme
Based on: [1], [2], [3], [4], [5], [6]
Descr:      Part of Code for Table 1 of the paper " Variational Autoencoders: A Harmonic Perspective"

            Store weights from (1) decoders and (2) encoders from models made using Step1-model_training.py
                as a .mat file. This .mat file can be used to estimate Lipschitz constant L using code from [4]

            note: does not store biases. For the left column of table 1, decoders are stored (L). For the right column, encoders are stored (R)

Sources:
    [1] examples from https://github.com/didriknielsen/survae_flow
    [2] https://avandekleut.github.io/vae/
    [3] The paper: Variational Autoencoders: A Harmonic Perspective
    [4] Lab 8 of CS4240-Deep Learning course
    [5] https://medium.com/dataseries/variational-autoencoder-with-pytorch-2d359cbf027b
    [6] LipSDP: https://github.com/arobey1/LipSDP
Notes:
    (a) you need to have trained the models before using this file, and stored at the proper location.
    (b) you need to have a folder Weights inside the directory where this .py file is stored.
"""

import torch
from torch import nn
from scipy.io import savemat
import numpy as np

#User settings
select = [2]   #select which models, 0: CelebA, 1: Cifar, 2: Sinc

#Define model architecture
###############################################################################
class VariationalEncoder(nn.Module):
```

```python
    def __init__(self, latent_dims, nChan, imSizeH, imSizeV):
        super(VariationalEncoder, self).__init__()
        self.input = nn.Flatten(1)
        self.dense1 =  nn.Linear(imSizeH*imSizeV*nChan, 256)
        self.dense2 =  nn.Linear(256, 256)
        self.dense3 =  nn.Linear(256, 256)
        self.outputMu =  nn.Linear(256, latent_dims)
        #self.outputSig = nn.Linear(256, latent_dims)       #this layer is not needed as sigma is
        ↪  fixed

        self.N = torch.distributions.Normal(0, 1)
        self.kl = 0

    def forward(self, x, sigma_en):
        x = self.input(x)
        x = torch.sigmoid(self.dense1(x))
        x = torch.sigmoid(self.dense2(x))
        x = torch.sigmoid(self.dense3(x))
        mu =  self.outputMu(x)
        #sigma = torch.exp(self.outputSig(x))               #this layer is not needed as sigma is
        ↪  fixed

        z = mu + (sigma_en**2)*self.N.sample(mu.shape)      #combination of encoders output of
        ↪  layer2, layer3
        self.kl = (sigma_en**2 + mu**2 - torch.log(sigma_en) - 1/2).sum()
        return z

class Decoder(nn.Module):
    def __init__(self, latent_dims, nChan, imSizeH, imSizeV):
        super(Decoder, self).__init__()
        self.dense1 = nn.Linear(latent_dims, 256)
        self.dense2 = nn.Linear(256, 256)
        self.dense3 = nn.Linear(256, 256)
        self.output = nn.Linear(256, nChan*imSizeH*imSizeV)
        self.outputShape = nn.Unflatten(dim=1, unflattened_size = (nChan, imSizeH, imSizeV))

    def forward(self, z):
        z = torch.sigmoid(self.dense1(z))
        z = torch.sigmoid(self.dense2(z))
        z = torch.sigmoid(self.dense3(z))
        z = self.output(z)
        z = self.outputShape(z)
        return z

class fnc_get_model(nn.Module):
    def __init__(self, latent_dims, nChan, imSizeH, imSizeV):
        super(fnc_get_model, self).__init__()
        self.encoder = VariationalEncoder(latent_dims, nChan, imSizeH, imSizeV)
        self.decoder = Decoder(latent_dims, nChan, imSizeH, imSizeV)

    def forward(self, x, sigma_en):
        z = self.encoder(x, sigma_en)
```

```
        return self.decoder(z)
################################################################################

################################################################################
#Edited version of code presented in [4]
def extract_weights(net):
    weights = []
    bias = []
    for param_tensor in net.state_dict():
        tensor = net.state_dict()[param_tensor].detach().numpy().astype(np.float64)
        if 'weight' in param_tensor:
            weights.append(tensor)
        if 'bias' in param_tensor:
            bias.append(tensor)
    return weights, bias
################################################################################

################################################################################
def fnc_getStore_and_Load(selector):
    if selector==0: #CelebA
        #Save/load paths for CelebA, left column
        path_ll = ["Models/modelL1-sigma-1_0", "Models/modelL1-sigma-0_5", "Models/modelL1-sigma-0_1"]                                  #Load locations
        path_swl = ["Weights/WeightsL1-sigma-1_0.mat", "Weights/WeightsL1-sigma-0_5.mat", "Weights/WeightsL1-sigma-0_1.mat"]    #Store locations

        #Save/load paths for CelebA, right column
        path_lr = ["Models/modelR1-sigma-1_0", "Models/modelR1-sigma-0_5", "Models/modelR1-sigma-0_0"]                                  #Load locations
        path_swr = ["Weights/WeightsR1-sigma-1_0.mat", "Weights/WeightsR1-sigma-0_5.mat", "Weights/WeightsR1-sigma-0_0.mat"]    #Store locations

    elif selector == 1: #Cifar
        #Save/load paths for CIFAR, left column
        path_ll = ["Models/modelL2-sigma-1_0", "Models/modelL2-sigma-0_5", "Models/modelL2-sigma-0_1"]                                  #Load locations
        path_swl = ["Weights/WeightsL2-sigma-1_0.mat", "Weights/WeightsL2-sigma-0_5.mat", "Weights/WeightsL2-sigma-0_1.mat"]    #Store locations

        #Save/load paths for CIFAR, right column
        path_lr = ["Models/modelR2-sigma-1_0", "Models/modelR2-sigma-0_5", "Models/modelR2-sigma-0_0"]                                  #Load locations
        path_swr = ["Weights/WeightsR2-sigma-1_0.mat", "Weights/WeightsR2-sigma-0_5.mat", "Weights/WeightsR2-sigma-0_0.mat"]    #Store locations

    else: #Sinc
        #Save/load paths for SINC, left column
        path_ll = ["Models/modelL3-sigma-1_0", "Models/modelL3-sigma-0_5", "Models/modelL3-sigma-0_1"]                                  #Load locations
        path_swl = ["Weights/WeightsL3-sigma-1_0.mat", "Weights/WeightsL3-sigma-0_5.mat", "Weights/WeightsL3-sigma-0_1.mat"]    #Store locations

```

```
        #Save/load paths for SINC, right column
        path_lr = ["Models/modelR3-sigma-1_0", "Models/modelR3-sigma-0_5", "Models/modelR3-sigma-0_0"]                          #Load locations
        path_swr = ["Weights/WeightsR3-sigma-1_0.mat", "Weights/WeightsR3-sigma-0_5.mat", "Weights/WeightsR3-sigma-0_0.mat"]    #Store locations
    return path_ll, path_lr, path_swl, path_swr
##############################################################################

latent_dims = [64, 64, 1]                   #latent dimensions
imSizeH = [64, 32, 172]                     #image size 1 (horizontal)
imSizeV = [64, 32, 2]                       #image size 2 (vertical)
nChan = [3, 3, 1]                           #number of channels

#Loop through all six models
for k in select:
    print("the selected model is {}. Note: 0=celeba, 1=cifar, 2=sinc".format(k))
    print(" ")
    path_ll, path_lr, path_swl, path_swr = fnc_getStore_and_Load(k)

    print("storing weights for left column")
    for j in range(3):
        model = fnc_get_model(latent_dims=latent_dims[k], nChan=nChan[k], imSizeH=imSizeH[k], imSizeV=imSizeV[k])
        model.load_state_dict(torch.load(path_ll[j]))
        model.eval()

        weights2, bias = extract_weights(model.decoder)
        data = {'weights': np.array(weights2, dtype=object)}
        savemat(path_swl[j], data)
        print("Stored weights from model {} at location {}".format(path_ll[j], path_swl[j]))
        print(" ")

    print("storing weights for right column")
    for j in range(3):
        model = fnc_get_model(latent_dims=latent_dims[k], nChan=nChan[k], imSizeH=imSizeH[k], imSizeV=imSizeV[k])
        model.load_state_dict(torch.load(path_lr[j]))
        model.eval()

        weights, _ = extract_weights(model.encoder)
        data = {'weights': np.array(weights, dtype=object)}
        savemat(path_swr[j], data)
        print("Stored weights from model {} at location {}".format(path_lr[j], path_swr[j]))
        print(" ")
```

B.2 Maximum-damage attack

In this section the matlab files for the maximum-damage algorithm is presented. First the main file will be presented, after which the used (non-standard) functions are given. The Python code used for generating the networks is the same code as shown before, however, slightly modified to accomodate the maximum damage attack settings (in terms of the standard deviations). Hence, it will not be repeated here.

B.2.1 Main code

Here the main code is presented in which the optimisation is done and the results are plotted.

```
clear all
close all
clc

% source this method is based on:
%https://arxiv.org/pdf/2102.07559.pdf
%% input x
cifar10_data=importdata('cifar10_data.png');
imge = cifar10_data(3:34,3:34,1:3); %rand(1,32*32*3);
imge=reshape(im2double(imge),[1,3072])';
%% optimize
% describe the optimization problem
net_imge = network_fnc_cif(imge);
nonlcon = @non_linear_cont;
options =
↪  optimoptions('fmincon','Display','iter','Algorithm','interior-point','MaxIterations',1000,'MaxFunctionEvaluations',10
fun = @(delta)-(norm(network_fnc_cif(imge+delta)-net_imge));
x0 = ones(1,32*32*3)'.*10^-5;
% can be used to force decoded attacked signal into y \in [-1,1]
%A = [AB_ende; -AB_ende];
%b = [ones(344,1)-AB_ende*x_t_sinc;ones(344,1)+AB_ende*x_t_sinc];
%
delta = fmincon(fun,x0,[],[],[],[],[],[],nonlcon,options);

const_1 = norm(delta);

%% plot

est_or = network_fnc_cif(imge);
est_att = network_fnc_cif(imge+delta);
% plot(imge(2:2:344),imge(1:2:343))
% hold on
% plot(est_or(2:2:344),est_or(1:2:343))
% plot(est_att(2:2:344),est_att(1:2:343))
% legend("og_sinc", "decoded sinc", "attacked decoded sinc")


figure
image(img_size_convert(imge,32,32,3))
title('original image')
figure
image(img_size_convert(imge+delta,32,32,3))
title('attacked image')
figure
```

```
image(img_size_convert(network_fnc_cif(imge),32,32,3))
title('original decoded image')
figure
image(img_size_convert(network_fnc_cif(imge+delta),32,32,3))
title('attacked decoded image')

%% loglikelihood degradation
logdeg = mean(log(abs(est_or-est_att))./log(abs(est_or)));
```

B.2.2 Sigmoid function

Here the element-wise sigmoid function is presented.

```
function y = sig_imp (x)
y = zeros(length(x),1);
for i = 1:length(x)
    y(i,1) = 1/(1+exp(-x(i)));
end

end
```

B.2.3 Network function

Here the function that describes the network. Note that we load in the weights trained for a specific network.

```
function [x_est] = network_fnc_cif(x)
% convert and load all the weights and biases generated by the learner
addpath('Weights2')
load('WeightsR2-sigma-1_0.mat')



encode1 = cell2mat(weights(1));
encode2 = cell2mat(weights(2));
encode3 = cell2mat(weights(3));
encodeu = cell2mat(weights(4));
load('WeightsL2-sigma-1_0.mat')
decode1 = cell2mat(weights(1));
decode2 = cell2mat(weights(2));
decode3 = cell2mat(weights(3));
decode4 = cell2mat(weights(4));



load('biasR2-sigma-1_0.mat')
bias_en1 = cell2mat(bias(1));
bias_en2 = cell2mat(bias(2));
bias_en3 = cell2mat(bias(3));
```

```
bias_enu = cell2mat(bias(4));
load('biasL-sigma-1_0.mat')
bias_de1 = transpose(cell2mat(bias(1)));
bias_de2 = transpose(cell2mat(bias(2)));
bias_de3 = transpose(cell2mat(bias(3)));
bias_de4 = transpose(cell2mat(bias(4)));

z = transpose(encodeu*sig_imp(encode3*sig_imp(encode2*sig_imp(encode1*x+bias_en1)+bias_en2)+bias_en3)+bias_enu);
x_est = transpose(decode4*sig_imp(decode3*sig_imp(decode2*sig_imp(decode1*z+bias_de1)+bias_de2)+bias_de3)+bias_de4);

end
```

B.2.4 Non linear constraints

In here the non-linear constraints are given for the optimisation.

```
function [c, ceq] = non_linear_cont(delta)
c_d = 20;
c(1) = norm(delta)-c_d;
ceq = [];
end
```

B.2.5 Image size conversion

Function used to convert the vectors back to image dimensions in a proper way.

```
function y = img_size_convert(x, a, b, c)
len_x = length(x);
y = zeros(a,b,c);
for i = 0:c-1
    x_tmp = x(i*len_x/c+1:(i+1)*len_x/c);
    len_x_tmp = length(x_tmp);
    for j = 0:b-1
        y(j+1,:,i+1) = x_tmp(j*len_x_tmp/a+1:(j+1)*len_x_tmp/a);
    end
end
end
```

Bibliography

[1] A. Camuto and M. Willetts, "Variational autoencoders: A harmonic perspective," 2021.

[2] Torch, "Adam." https://pytorch.org/docs/stable/generated/torch.optim.Adam.html, 2019. Accessed on 13-04-2022.

[3] M. Fazlyab, A. Robey, H. Hassani, M. Morari, and G. J. Pappas, "Efficient and accurate estimation of lipschitz constants for deep neural networks," 2019.

[4] M. Fazlyab, A. Robey, H. Hassani, M. Morari, and G. J. Pappas, "Lipsdp." https://github.com/arobey1/LipSDP, 2019. Accessed on 13-04-2022.

[5] B. Barrett, A. Camuto, M. Willetts, and T. Rainforth, "Certifiably robust variational autoencoders." https://arxiv.org/pdf/2102.07559.pdf, 2021. Accessed on 29-03-2022.

[6] D. Nielsen, P. Jaini, E. Hoogeboom, O. Winther, and M. Welling, "survae flows." https://github.com/didriknielsen/survae_flows, 2020. Accessed on 13-04-2022.

[7] v. d. K. Alexander, "Variational autoencoders (vae) with pytorch." https://avandekleut.github.io/vae/, 2020. Accessed on 13-04-2022.

[8] J. C. van Gemert and D. Tax, "Assignment 8 - solutions." https://colab.research.google.com/drive/1sdL0w-BfY5UjFGN_vVZb76HZhuujA7T6?usp=sharing, 2022. Accessed on 13-04-2022.

[9] A. Anello, "Variational autoencoder with pytorch." https://medium.com/dataseries/variational-autoencoder-with-pytorch-2d359cbf027b, 2021. Accessed on 13-04-2022.