

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 adversarial attack.

A virtual machine was set up and trained on the Sinc, Cifar10 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.

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 $\sigma_N \in \{0.0, 0.5, 1.0\}$. 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 $\mathbf{z} \sim \mathcal{N}(\mu(\mathbf{x}), \sigma_{enc}^2 \mathbf{I})$. 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.

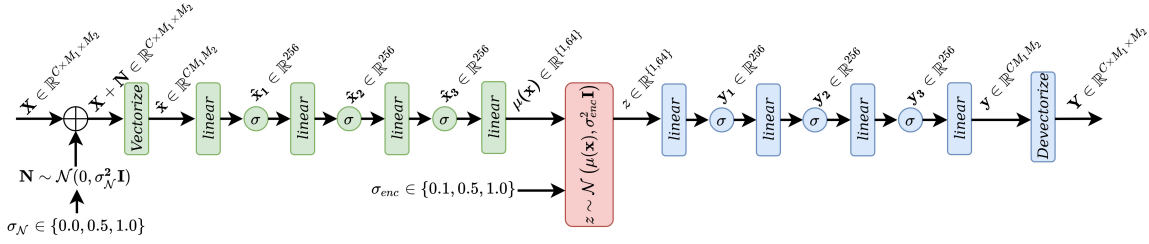


Figure 1: A visual description of the used network. Green corresponds to the encoder network, red corresponds to the latent space and blue corresponds to the decoder network. By setting σ_N to zero, the noise injections will effectively be removed.

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.

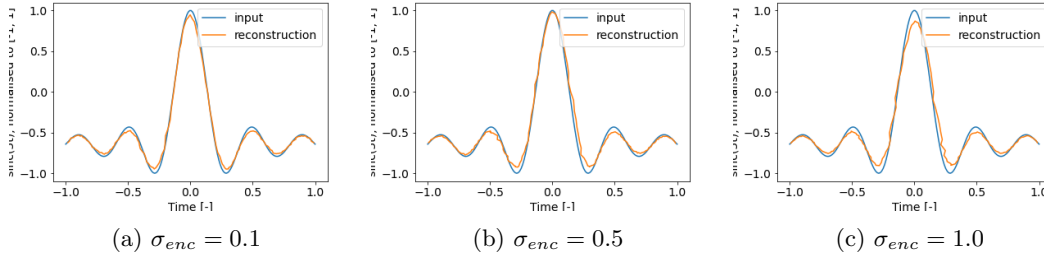


Figure 2: The obtained reconstruction of the sinc data from the validation set. The networks are trained without noise injections on the input. The encoder networks have a standard deviation $\sigma_{enc} \in \{0.1, 0.5, 1.0\}$.

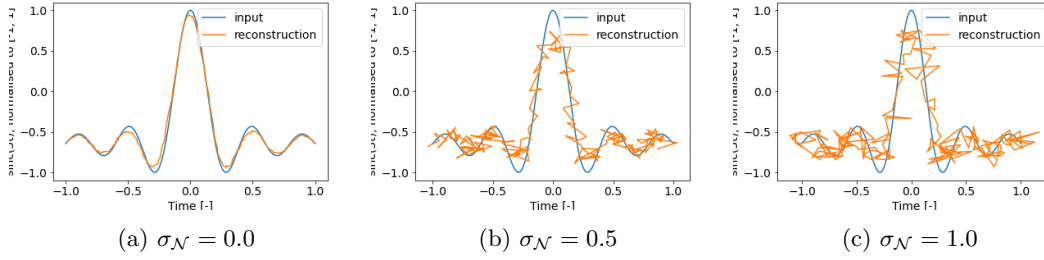


Figure 3: The obtained reconstruction of sinc data from the validation set. The networks are trained with zero-mean Gaussian noise injections on the input. The injections have a standard deviation $\sigma_N \in \{0.0, 0.5, 1.0\}$. The encoder networks have a fixed standard deviation $\sigma_{enc} = 0.5$.

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 $\sigma_{enc} = 0.1$. 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.

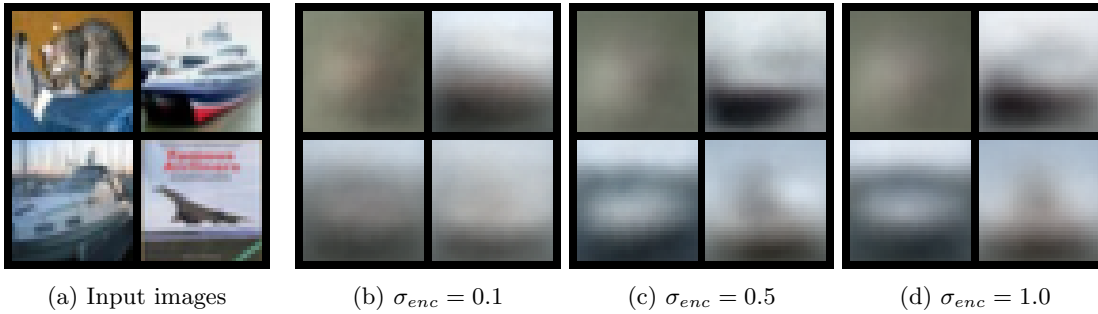


Figure 4: The obtained reconstruction of four CIFAR10 images from the validation set. The networks are trained without noise injections on the input. The encoder networks have a standard deviation $\sigma_{enc} \in \{0.1, 0.5, 1.0\}$.

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

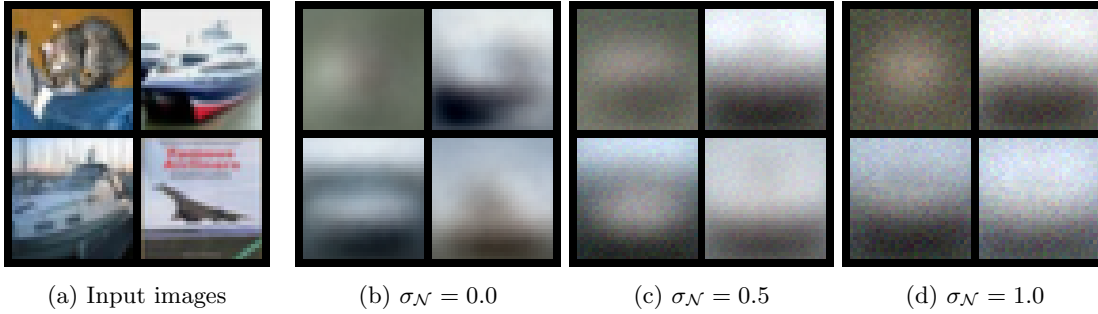


Figure 5: The obtained reconstruction of four CIFAR10 images from the validation set. The networks are trained with zero-mean Gaussian noise injections on the input. The injections have a standard deviation $\sigma_{\mathcal{N}} \in \{0.0, 0.5, 1.0\}$. The encoder networks have a fixed standard deviation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

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.

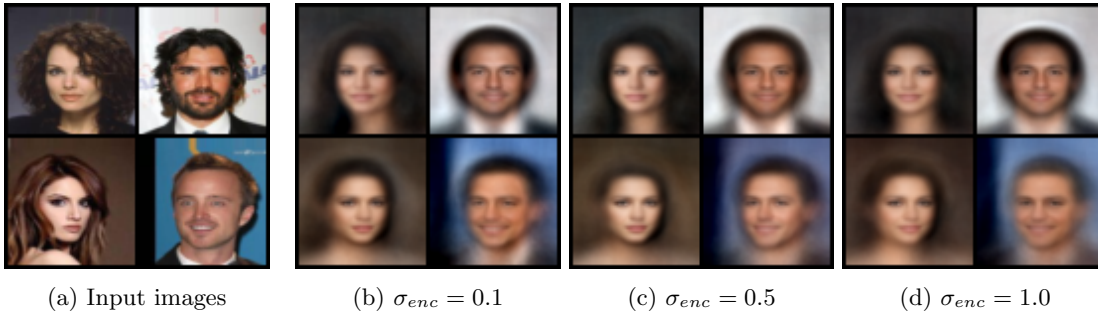


Figure 6: The obtained reconstruction of four CelebA images from the validation set. The networks are trained without noise injections on the input. The encoder networks have a standard deviation $\sigma_{enc} \in \{0.1, 0.5, 1.0\}$.

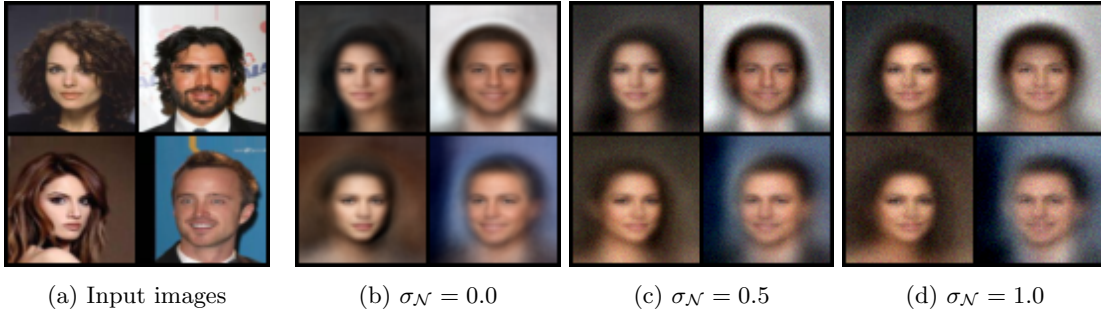


Figure 7: The obtained reconstruction of four CelebA images from the validation set. The networks are trained with zero-mean Gaussian noise injections on the input. The injections have a standard deviation $\sigma_{\mathcal{N}} \in \{0.0, 0.5, 1.0\}$. The encoder networks have a fixed standard deviation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

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	$\sigma_{enc} = 1.0$	$\sigma_{enc} = 0.5$	$\sigma_{enc} = 0.1$	$\sigma_N = 1.0$	$\sigma_N = 0.5$	$\sigma_N = 0.0$
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 \pm 432.3	1832.5 \pm 28.8	4604.2 \pm 211.5	542.0 \pm 219.7	766.5 \pm 295.5	494.6 \pm 43.0
Ref. [1]	2.2 \pm 0.2	5.2 \pm 0.3	17.9 \pm 3.2	13.9 \pm 2.7	24.6 \pm 1.7	29.8 \pm 2.2

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

Seed	$\sigma_{enc} = 1.0$	$\sigma_{enc} = 0.5$	$\sigma_{enc} = 0.1$	$\sigma_N = 1.0$	$\sigma_N = 0.5$	$\sigma_N = 0.0$
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 \pm 80443	85023 \pm 21592	574363 \pm 155194	985252 \pm 663535	235366 \pm 78182	29040 \pm 7360
Ref. [1]	17.9 \pm 1.2	19.1 \pm 1.2	27.3 \pm 0.3	4.7 \pm 0.2	5.6 \pm 0.6	8.5 \pm 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	$\sigma_{enc} = 1.0$	$\sigma_{enc} = 0.5$	$\sigma_{enc} = 0.1$	$\sigma_N = 1.0$	$\sigma_N = 0.5$	$\sigma_N = 0.0$
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 \pm 35370	90355 \pm 7917	472006 \pm 23967	390468 \pm 51245	104673 \pm 8892	42843 \pm 2387
Ref. [1]	7.5 \pm 1.1	12.0 \pm 0.5	13.7 \pm 1.2	1.4 \pm 0.1	1.6 \pm 0.1	1.8 \pm 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 variational 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.

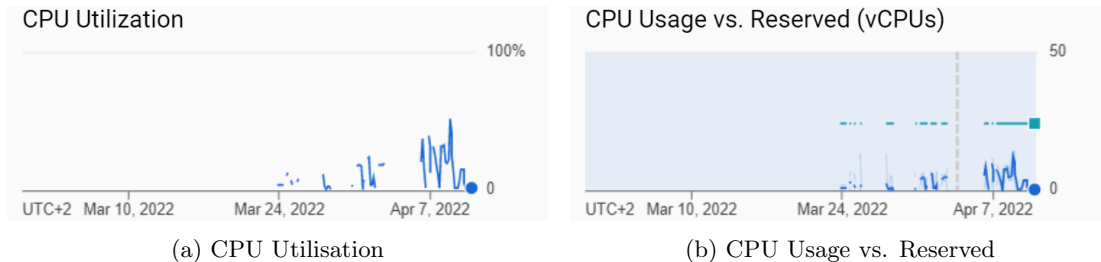


Figure 8: Virtual Machine metrics

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).

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

In this equation, x is the original image, δ is the damage done to the original image, $p(\cdot)$ is the encoder operation, $g(\cdot)$ 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.

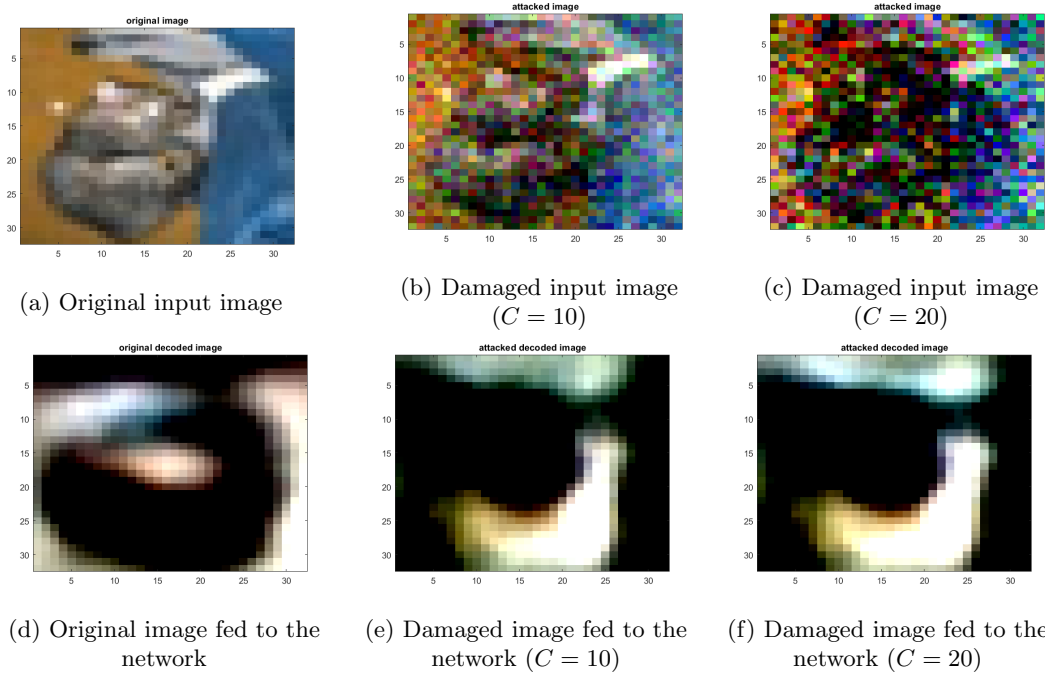


Figure 9: Maximum damage attack applied to the network generated with $\sigma = 1.0$, maximum norm of the attack is $C = 10$ or $C = 20$.

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 of the trends found in the paper, as well as provide suggestions for further research on how to more successfully reproduce the paper.

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 were 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].

```
1  """
2  Date:      21-03-2022
3  Last edit: 24-03-2022
4  Author:    Austin, Richard, Mustafa, Dimme
5
6  Based on:  [1], [2], [3], [4], [5]
7  Trains:    3x6 models on celeba, cifar, sinc
8  Descr:     Part of Code for Table 1 of the paper " Variational Autoencoders: A Harmonic
   ↪ Perspective"
9             decoder is deterministic (given a latent space)
10
11            Where the first three models have no noise injections on the input, and fixed encoder
   ↪ standard deviation (!) of
12            [1.0, 0.5, 0.1]
13            Where the last three models have fixed encoder standard deviation (!) 0.5 and noise
   ↪ injections of
14            a standard normal distribution with zero mean and standard deviation of [1.0, 0.5,
   ↪ 0.0]
15
16  Some notes:
17      (a) inside the folder where you put this file, make the following two folders:
18          (1) Model (used for storing model state)
19          (2) Optim (used for storing optimiser state)
20
21  Sources:
22      [1] examples from https://github.com/didriknielsen/survae_flow
23      [2] https://avandekleut.github.io/vae/
24      [3] The paper: Variational Autoencoders: A Harmonic Perspective
25      [4] Lab 8 of CS4240-Deep Learning course
26      [5] https://medium.com/dataserie/variational-autoencoder-with-pytorch-2d359cbf027b
27  """
28
29  import torch
30  import torchvision.datasets as datasets
31
```

```

32 from torch import nn
33 from survae.data.loaders.image import CelebA
34 from torch.utils.data import DataLoader
35 from torchvision.transforms import ToTensor
36 from torch.optim import Adam
37 from pytorch import TicToc
38
39 #use one of three seeds for reproducible results+calculating standard deviation
40 #seeds obtained using torch.seed() three times.
41
42 torch.manual_seed(14609714069366804553)
43 #torch.manual_seed(10209623728859046282)
44 #torch.manual_seed(13978253786968215756)
45
46 device = 'cuda' if torch.cuda.is_available() else 'cpu'
47 t = TicToc()
48
49 #User settings
50 select = [2] #select which models, 0: CelebA, 1: Cifar, 2: Sinc;
51 ↪ [0, 1, 2] runs all
52 epochs, lr = [200, 200, 100], [1e-3, 1e-3, 1e-3] #number of epochs, learning rate; epochs[0]
53 ↪ corresponds to CelebA, etc.
54 load_old = [0, 0, 1] #set to one if you want to continue training.
55 ↪ load_old[0] corresponds to CelebA, etc.
56 norm = [0, 1, 2] #set type of normalisation: 0: 8 bit integer to float
57 ↪ [-1, 1]; 1: float to float [-1, 1]; 2: no normalisation
58
59 #Define model architecture
60 #####
61 class VariationalEncoder(nn.Module):
62     def __init__(self, latent_dims, nChan, imSizeH, imSizeV):
63         super(VariationalEncoder, self).__init__()
64         self.input = nn.Flatten(1)
65         self.dense1 = nn.Linear(imSizeH*imSizeV*nChan, 256)
66         self.dense2 = nn.Linear(256, 256)
67         self.dense3 = nn.Linear(256, 256)
68         self.outputMu = nn.Linear(256, latent_dims)
69         #self.outputSig = nn.Linear(256, latent_dims) #this layer is not needed as sigma is
70         ↪ fixed
71
72         self.N = torch.distributions.Normal(0, 1)
73         self.kl = 0
74
75     def forward(self, x, sigma_en):
76         x = self.input(x)
77         x = torch.sigmoid(self.dense1(x))
78         x = torch.sigmoid(self.dense2(x))
79         x = torch.sigmoid(self.dense3(x))
80         mu = self.outputMu(x)
81         #sigma = torch.exp(self.outputSig(x)) #this layer is not needed as sigma is
82         ↪ fixed

```



```

78
79     z = mu + (sigma_en**2)*self.N.sample(mu.shape)      #combination of encoders output of
    ↪ layer2, layer3
80     self.kl = (sigma_en**2 + mu**2 - torch.log(sigma_en) - 1/2).sum()
81     return z
82
83 class Decoder(nn.Module):
84     def __init__(self, latent_dims, nChan, imSizeH, imSizeV):
85         super(Decoder, self).__init__()
86         self.dense1 = nn.Linear(latent_dims, 256)
87         self.dense2 = nn.Linear(256, 256)
88         self.dense3 = nn.Linear(256, 256)
89         self.output = nn.Linear(256, nChan*imSizeH*imSizeV)
90         self.outputShape = nn.Unflatten(dim=1, unflattened_size = (nChan, imSizeH, imSizeV))
91
92     def forward(self, z):
93         z = torch.sigmoid(self.dense1(z))
94         z = torch.sigmoid(self.dense2(z))
95         z = torch.sigmoid(self.dense3(z))
96         z = self.output(z)
97         z = self.outputShape(z)
98         return z
99
100 class fnc_get_model(nn.Module):
101     def __init__(self, latent_dims, nChan, imSizeH, imSizeV):
102         super(fnc_get_model, self).__init__()
103         self.encoder = VariationalEncoder(latent_dims, nChan, imSizeH, imSizeV)
104         self.decoder = Decoder(latent_dims, nChan, imSizeH, imSizeV)
105
106     def forward(self, x, sigma_en):
107         z = self.encoder(x, sigma_en)
108         return self.decoder(z)
109     #####
110
111     #####3
112     #function to train model (left column)
113     def fnc_train(epochs, sigma_en, sigma_in, norm, nChan, imSizeH, imSizeV):
114         if norm == 0:
115             scale, div, offset = 2, 255, 1
116         elif norm == 1:
117             scale, div, offset = 2, 1, 1
118         else:
119             scale, div, offset = 1, 1, 0
120
121         if sigma_in == 0:
122             for epoch in range(epochs):
123                 l = 0.0
124                 for i, x in enumerate(train_loader):
125                     if isinstance(x, list):
126                         x = x[0]
127                         x = scale*(x/div)-offset    #normalise between [-1,1]
128                         x = x.to(device)

```

```

129         optimizer.zero_grad()
130         x_hat = model(x, sigma_en)
131         loss = ((x - x_hat)**2).sum() + model.encoder.kl
132         loss.backward()
133         optimizer.step()
134         l += loss.detach().cpu().item()
135         print('Epoch: {}/{}, Loss: {:.3f}'.format(epoch+1, epochs, l/(i+1), end='\r'))
136     else:
137         for epoch in range(epochs): #this part is based on [3,4]
138             l = 0.0
139             for i, x in enumerate(train_loader):
140                 if isinstance(x, list):
141                     x = x[0]
142                     noise = torch.normal(0, sigma_in, (nChan, imSizeH, imSizeV))
143                     x = scale*(x/div)-offset #normalise between [-1,1]
144                     x = x+noise
145                     x = x.to(device)
146                     optimizer.zero_grad()
147                     x_hat = model(x, sigma_en)
148                     loss = ((x - x_hat)**2).sum() + model.encoder.kl #b
149                     loss.backward()
150                     optimizer.step()
151                     l += loss.detach().cpu().item()
152                 print('Epoch: {}/{}, Loss: {:.3f}'.format(epoch+1, epochs, l/(i+1), end='\r'))
153
154     #####
155
156     ##### Functions to get training data #####
157     def fnc_getTrain_loader(selector):
158         if selector == 0:
159             data = CelebA()
160             train_loader = DataLoader(dataset=data.train, batch_size=256, shuffle=True,
161                                     ↪ num_workers=8, drop_last=True)
162             t.toc()
163         elif selector == 1:
164             cifar_trainset = datasets.CIFAR10(root='../DATA', train=True, download=True,
165                                     ↪ transform=ToTensor())
166             train_loader = DataLoader(dataset=cifar_trainset, batch_size=256, shuffle=True,
167                                     ↪ num_workers=8, drop_last=True)
168             t.toc()
169         else:
170             w, t_s, t_e, N, M = 5, -1, 1, 172, 8*512 #frequency, start time, end time, data
171             ↪ of track, number of tracks
172             ft, t_ax = sinc(w, t_s, t_e, N, M) #get raw data
173             data, offset, scale = preprocess(ft, t_ax) #preprocess raw data. Max of data in
174             ↪ [-1, 1]
175             train_loader = DataLoader(dataset=data, batch_size=256, shuffle=True, num_workers=8,
176                                     ↪ drop_last=True)
177             t.toc()
178         return train_loader
179
180     #function to get sinc data

```

```

175 def sinc(w, t_s, t_e, N, M):
176     t_ax = 2*torch.rand(M, 1, N, 1) - 1
177     t_ax = torch.sort(t_ax, dim=2)
178     t_ax = t_ax[0]
179     f_eval = torch.sinc(w*t_ax)
180     return f_eval, t_ax
181
182 #function to preprocess sinc data
183 def preprocess(ft, t):
184     offset = torch.min(ft)
185     ft = ft - offset #make minimum zero for both t and f(t)
186     scale = torch.max(ft)
187     ft = ft/scale #make max one for both t and f(t)
188     ft = 2*ft - 1 #from t, f(t) in [0,1] to t, f(t) in [-1, 1]
189     out = torch.cat((ft, t), -1)
190     return out, offset, scale #return data, offset and scale. Note that the 2* and -1 are
191     ↪ not returned
192
193 #####
194 def fnc_getStore_and_Load(selector):
195     if selector==0: #CelebA
196         #Save/load paths for CelebA, left column
197         path_l1 = ["Models/modelL1-sigma-1_0", "Models/modelL1-sigma-0_5",
198             ↪ "Models/modelL1-sigma-0_1" #Load locations
199         path_s1 = ["Models/modelL1-sigma-1_0", "Models/modelL1-sigma-0_5",
200             ↪ "Models/modelL1-sigma-0_1" #Store locations
201         path_l0l = ["Optim/modelL1-sigma-1_0", "Optim/modelL1-sigma-0_5",
202             ↪ "Optim/modelL1-sigma-0_1" #Load locations
203         path_s0l = ["Optim/modelL1-sigma-1_0", "Optim/modelL1-sigma-0_5",
204             ↪ "Optim/modelL1-sigma-0_1" #Store locations
205
206         #Save/load paths for CelebA, right column
207         path_lr = ["Models/modelR1-sigma-1_0", "Models/modelR1-sigma-0_5",
208             ↪ "Models/modelR1-sigma-0_0" #Load locations
209         path_sr = ["Models/modelR1-sigma-1_0", "Models/modelR1-sigma-0_5",
210             ↪ "Models/modelR1-sigma-0_0" #Store locations
211         path_lor = ["Optim/modelR1-sigma-1_0", "Optim/modelR1-sigma-0_5",
212             ↪ "Optim/modelR1-sigma-0_0" #Load locations
213         path_sor = ["Optim/modelR1-sigma-1_0", "Optim/modelR1-sigma-0_5",
214             ↪ "Optim/modelR1-sigma-0_0" #Store locations
215
216     elif selector == 1: #Cifar
217         #Save/load paths for CIFAR, left column
218         path_l1 = ["Models/modelL2-sigma-1_0", "Models/modelL2-sigma-0_5",
219             ↪ "Models/modelL2-sigma-0_1" #Load locations
220         path_s1 = ["Models/modelL2-sigma-1_0", "Models/modelL2-sigma-0_5",
221             ↪ "Models/modelL2-sigma-0_1" #Store locations
222         path_l0l = ["Optim/modelL2-sigma-1_0", "Optim/modelL2-sigma-0_5",
223             ↪ "Optim/modelL2-sigma-0_1" #Load locations
224         path_s0l = ["Optim/modelL2-sigma-1_0", "Optim/modelL2-sigma-0_5",
225             ↪ "Optim/modelL2-sigma-0_1" #Store locations

```

```

214
215     #Save/load paths for CIFAR, right column
216     path_lr = ["Models/modelR2-sigma-1_0", "Models/modelR2-sigma-0_5",
217               ↪ "Models/modelR2-sigma-0_0"] #Load locations
218     path_sr = ["Models/modelR2-sigma-1_0", "Models/modelR2-sigma-0_5",
219               ↪ "Models/modelR2-sigma-0_0"] #Store locations
220     path_lor = ["Optim/modelR2-sigma-1_0", "Optim/modelR2-sigma-0_5",
221               ↪ "Optim/modelR2-sigma-0_0"] #Load locations
222     path_sor = ["Optim/modelR2-sigma-1_0", "Optim/modelR2-sigma-0_5",
223               ↪ "Optim/modelR2-sigma-0_0"] #Store locations
224
225 else: #Sinc
226     #Save/load paths for SINC, left column
227     path_ll = ["Models/modelL3-sigma-1_0", "Models/modelL3-sigma-0_5",
228               ↪ "Models/modelL3-sigma-0_1"] #Load locations
229     path_sl = ["Models/modelL3-sigma-1_0", "Models/modelL3-sigma-0_5",
230               ↪ "Models/modelL3-sigma-0_1"] #Store locations
231     path_lol = ["Optim/modelL3-sigma-1_0", "Optim/modelL3-sigma-0_5",
232               ↪ "Optim/modelL3-sigma-0_1"] #Load locations
233     path_sol = ["Optim/modelL3-sigma-1_0", "Optim/modelL3-sigma-0_5",
234               ↪ "Optim/modelL3-sigma-0_1"] #Store locations
235
236     #Save/load paths for SINC, right column
237     path_lr = ["Models/modelR3-sigma-1_0", "Models/modelR3-sigma-0_5",
238               ↪ "Models/modelR3-sigma-0_0"] #Load locations
239     path_sr = ["Models/modelR3-sigma-1_0", "Models/modelR3-sigma-0_5",
240               ↪ "Models/modelR3-sigma-0_0"] #Store locations
241
242     path_lor = ["Optim/modelR3-sigma-1_0", "Optim/modelR3-sigma-0_5",
243               ↪ "Optim/modelR3-sigma-0_0"] #Load locations
244     path_sor = ["Optim/modelR3-sigma-1_0", "Optim/modelR3-sigma-0_5",
245               ↪ "Optim/modelR3-sigma-0_0"] #Store locations
246
247     return path_ll, path_sl, path_lol, path_sol, path_lr, path_sr, path_lor, path_sor
248
249 #####
250 #####
251 #some general model settings
252 sigma_in1 = torch.tensor(0) #input standard deviation, left column
253 sigma_enc1 = torch.tensor([1.0, 0.5, 0.1]) #encoder standard deviation, left column
254 sigma_in2 = torch.tensor([1.0, 0.5, 0.0]) #input standard deviation, right column
255 sigma_enc2 = torch.tensor(0.5) #encoder standard deviation, right column
256
257 latent_dims = [64, 64, 1] #latent dimensions
258 imSizeH = [64, 32, 172] #image size 1 (horizontal)
259 imSizeV = [64, 32, 2] #image size 2 (vertical)
260 nChan = [3, 3, 1] #number of channels
261
262 #start timer
263 t.tic()
264
265 for k in select: #k denotes the data

```

```

254     print("the selected model is {}".format(k))
255     print("Step 1: Loading data")
256     train_loader = fnc_getTrain_loader(k)
257     path_ll, path_sl, path_lol, path_sol, path_lr, path_sr, path_lor, path_sor =
        ↪ fnc_getStore_and_Load(k)
258
259     print(" ")
260     print("training for left column")
261     for i in range(3): #i denotes the settings for sigma_enc
262         model = fnc_get_model(latent_dims=latent_dims[k], nChan=nChan[k], imSizeH=imSizeH[k],
        ↪ imSizeV=imSizeV[k])
263         optimizer = Adam(model.parameters(), lr=lr[k])
264         if load_old[k] == 1:
265             model.load_state_dict(torch.load(path_ll[i]))
266             optimizer.load_state_dict(torch.load(path_lol[i]))
267
268         model.train()
269         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])
270
271         torch.save(model.state_dict(), path_sl[i]) #Save current state of model
272         torch.save(optimizer.state_dict(), path_sol[i]) #Save current state of optimiser
273         print("Saved model for data {}: sigma_enc={}, sigma_in={}".format(k, sigma_enc1[i],
        ↪ sigma_in1))
274         print(" ")
275     print("Done training for left column")
276
277     print(" ")
278
279     print("Training for right column")
280     for i in range(3): #i denotes the settings for sigma_in
281         model = fnc_get_model(latent_dims=latent_dims[k], nChan=nChan[k], imSizeH=imSizeH[k],
        ↪ imSizeV=imSizeV[k])
282         optimizer = Adam(model.parameters(), lr=lr[k])
283         if load_old[k] == 1:
284             model.load_state_dict(torch.load(path_lr[i]))
285             optimizer.load_state_dict(torch.load(path_lor[i]))
286
287         model.train()
288         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])
289
290         torch.save(model.state_dict(), path_sr[i]) #Save current state of model
291         torch.save(optimizer.state_dict(), path_sor[i]) #Save current state of optimiser
292
293         print("Saved model for data {}: sigma_enc={}, sigma_in={}".format(k, sigma_enc2,
        ↪ sigma_in2[i]))
294         print(" ")
295     print("Done training for right column")
296
297     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].

```
1  """
2  Date: 25-03-2022
3  Author: Austin, Mustafa, Richard, Dimme
4  Based on: [1], [2], [3], [4], [5]
5  Descr: Part of Code for Table 1 of the paper " Variational Autoencoders: A Harmonic Perspective"
6          Stores figures for verification purposes.
7
8  Sources:
9      [1] examples from https://github.com/didriknielsen/survae_flow
10     [2] https://avandekleut.github.io/vae/
11     [3] The paper: Variational Autoencoders: A Harmonic Perspective
12     [4] Lab 8 of CS4240-Deep Learning course
13     [5] https://medium.com/dataserie/variational-autoencoder-with-pytorch-2d359cbf027b
14
15  Notes:
16     (a) you need to have trained the models before using this file, and stored at the proper
↪ location.
17     (b) you need to have a folder Figures inside the directory where this .py file is stored.
18  """
19
20  import torch
21  import torchvision.datasets as datasets
22  import torchvision.utils as vutils
23  import matplotlib.pyplot as plt
24  import numpy as np
25
26  from torch import nn
27  from survae.data.loaders.image import CelebA
28  from torch.utils.data import DataLoader
29  from torchvision.transforms import ToTensor
30  from skimage import color
31
32  #User settings
33  select = [2] #select which models, 0: CelebA, 1: Cifar, 2: Sinc
34
35
36  plt.rc('font', size=14) #controls default text size
37
38  #Define model architecture
39  #####
40  class VariationalEncoder(nn.Module):
41      def __init__(self, latent_dims, nChan, imSizeH, imSizeV):
42          super(VariationalEncoder, self).__init__()
43          self.input = nn.Flatten(1)
44          self.dense1 = nn.Linear(imSizeH*imSizeV*nChan, 256)
45          self.dense2 = nn.Linear(256, 256)
46          self.dense3 = nn.Linear(256, 256)
47          self.outputMu = nn.Linear(256, latent_dims)
```

```

48     #self.outputSig = nn.Linear(256, latent_dims)      #this layer is not needed as sigma is
    ↪ fixed
49
50     self.N = torch.distributions.Normal(0, 1)
51     self.kl = 0
52
53     def forward(self, x, sigma_en):
54         x = self.input(x)
55         x = torch.sigmoid(self.dense1(x))
56         x = torch.sigmoid(self.dense2(x))
57         x = torch.sigmoid(self.dense3(x))
58         mu = self.outputMu(x)
59         #sigma = torch.exp(self.outputSig(x))          #this layer is not needed as sigma is
    ↪ fixed
60
61         z = mu + (sigma_en**2)*self.N.sample(mu.shape)  #combination of encoders output of
    ↪ layer2, layer3
62         self.kl = (sigma_en**2 + mu**2 - torch.log(sigma_en) - 1/2).sum()
63         return z
64
65     class Decoder(nn.Module):
66         def __init__(self, latent_dims, nChan, imSizeH, imSizeV):
67             super(Decoder, self).__init__()
68             self.dense1 = nn.Linear(latent_dims, 256)
69             self.dense2 = nn.Linear(256, 256)
70             self.dense3 = nn.Linear(256, 256)
71             self.output = nn.Linear(256, nChan*imSizeH*imSizeV)
72             self.outputShape = nn.Unflatten(dim=1, unflattened_size = (nChan, imSizeH, imSizeV))
73
74         def forward(self, z):
75             z = torch.sigmoid(self.dense1(z))
76             z = torch.sigmoid(self.dense2(z))
77             z = torch.sigmoid(self.dense3(z))
78             z = self.output(z)
79             z = self.outputShape(z)
80             return z
81
82     class fnc_get_model(nn.Module):
83         def __init__(self, latent_dims, nChan, imSizeH, imSizeV):
84             super(fnc_get_model, self).__init__()
85             self.encoder = VariationalEncoder(latent_dims, nChan, imSizeH, imSizeV)
86             self.decoder = Decoder(latent_dims, nChan, imSizeH, imSizeV)
87
88         def forward(self, x, sigma_en):
89             z = self.encoder(x, sigma_en)
90             return self.decoder(z)
91     #####
92
93     #####
94     def fnc_getStore_and_Load(selector):
95         if selector==0: #CelebA
96             #Save/load paths for CelebA, left column

```

```

97     path_ll = ["Models/modelL1-sigma-1_0", "Models/modelL1-sigma-0_5",
98         ↪ "Models/modelL1-sigma-0_1"] #Load locations
99     path_sf1 = ["Figures/FigL1-sigma-1_0-ev.png", "Figures/FigL1-sigma-0_5-ev.png",
100         ↪ "Figures/FigL1-sigma-0_1-ev.png"] #Store locations evaluation
101     path_sdl = ["Figures/FigL1-sigma-1_0-data.png", "Figures/FigL1-sigma-0_5-data.png",
102         ↪ "Figures/FigL1-sigma-0_1-data.png"] #Store locations data
103
104     #Save/load paths for CelebA, right column
105     path_lr = ["Models/modelR1-sigma-1_0", "Models/modelR1-sigma-0_5",
106         ↪ "Models/modelR1-sigma-0_0"] #Load locations
107     path_sfr = ["Figures/FigR1-sigma-1_0-ev.png", "Figures/FigR1-sigma-0_5-ev.png",
108         ↪ "Figures/FigR1-sigma-0_0-ev.png"] #Store locations evaluation
109     path_sdr = ["Figures/FigR1-sigma-1_0-data.png", "Figures/FigR1-sigma-0_5-data.png",
110         ↪ "Figures/FigR1-sigma-0_0-data.png"] #Store locations data
111
112     elif selector == 1: #Cifar
113         #Save/load paths for CIFAR, left column
114         path_ll = ["Models/modelL2-sigma-1_0", "Models/modelL2-sigma-0_5",
115             ↪ "Models/modelL2-sigma-0_1"] #Load locations
116         path_sf1 = ["Figures/FigL2-sigma-1_0-ev.png", "Figures/FigL2-sigma-0_5-ev.png",
117             ↪ "Figures/FigL2-sigma-0_1-ev.png"] #Store locations evaluation
118         path_sdl = ["Figures/FigL2-sigma-1_0-data.png", "Figures/FigL2-sigma-0_5-data.png",
119             ↪ "Figures/FigL2-sigma-0_1-data.png"] #Store locations data
120
121         #Save/load paths for CIFAR, right column
122         path_lr = ["Models/modelR2-sigma-1_0", "Models/modelR2-sigma-0_5",
123             ↪ "Models/modelR2-sigma-0_0"] #Load locations
124         path_sfr = ["Figures/FigR2-sigma-1_0-ev.png", "Figures/FigR2-sigma-0_5-ev.png",
125             ↪ "Figures/FigR2-sigma-0_0-ev.png"] #Store locations evaluation
126         path_sdr = ["Figures/FigR2-sigma-1_0-data.png", "Figures/FigR2-sigma-0_5-data.png",
127             ↪ "Figures/FigR2-sigma-0_0-data.png"] #Store locations data
128
129     else: #Sinc
130         #Save/load paths for SINC, left column
131         path_ll = ["Models/modelL3-sigma-1_0", "Models/modelL3-sigma-0_5",
132             ↪ "Models/modelL3-sigma-0_1"] #Load locations
133         path_sf1 = ["Figures/FigL3-sigma-1_0-ev.png", "Figures/FigL3-sigma-0_5-ev.png",
134             ↪ "Figures/FigL3-sigma-0_1-ev.png"] #Store locations evaluation
135         path_sdl = ["not used", "not used", "not used"] #Store locations data
136
137         #Save/load paths for SINC, right column
138         path_lr = ["Models/modelR3-sigma-1_0", "Models/modelR3-sigma-0_5",
139             ↪ "Models/modelR3-sigma-0_0"] #Load locations
140         path_sfr = ["Figures/FigR3-sigma-1_0-ev.png", "Figures/FigR3-sigma-0_5-ev.png",
141             ↪ "Figures/FigR3-sigma-0_0-ev.png"] #Store locations evaluation
142         path_sdr = ["not used", "not used", "not used"] #Store locations data
143
144     return path_ll, path_lr, path_sf1, path_sfr, path_sdl, path_sdr
145
146     #####
147
148     #####
149
150     def fnc_getValid_loader(selector):

```



```

133     if selector == 0:
134         data = CelebA()
135         valid_loader = DataLoader(dataset=data.valid, batch_size=256, shuffle=False,
136             ↪ num_workers=8, drop_last=True)
137     elif selector == 1:
138         cifar_trainset = datasets.CIFAR10(root='../DATA', train=False, download=True,
139             ↪ transform=ToTensor())
140         valid_loader = DataLoader(dataset=cifar_trainset, batch_size=256, shuffle=False,
141             ↪ num_workers=8, drop_last=True)
142     else:
143         w, t_s, t_e, N, M = 5, -1, 1, 172, 2           #frequency, start time, end time, number of
144             ↪ samples
145         data_p = sinc(w, t_s, t_e, N, M)               #get raw data
146         data, offset, scale = preprocess(data_p)       #preprocess raw data. Using scale and offset
147             ↪ og data can be returned
148         valid_loader = data
149     return valid_loader
150
151 #function to get sinc data
152 def sinc(w, t_s, t_e, N, M):
153     t_ax = torch.linspace(t_s, t_e, N)
154     t_ax = t_ax.unsqueeze(1)
155     t_ax = t_ax.unsqueeze(0)
156     t_ax = t_ax.unsqueeze(0)
157     f_eval = torch.sinc(w*t_ax)
158     out = torch.cat((f_eval, t_ax),-1)
159     return out
160
161 #function to preprocess sinc data
162 def preprocess(data):
163     offset = torch.min(data, dim=2)
164     data = data - offset.values           #make minimum zero for both t and f(t)
165     scale = torch.max(data, dim=2)
166     data = data/scale.values             #make max one for both t and f(t)
167     data = 2*data - 1                   #from t, f(t) in [0,1] to t, f(t) in [-1, 1]
168     return data, offset, scale           #return data, offset and scale. Note that the 2* and -1
169     ↪ are not returned
170
171 #####
172 #####
173 def fnc_plot(model, data_name, result_name, valid_loader, selector):
174     N = 4
175     fft = 0
176     if selector == 0: #CELEBA
177         img = next(iter(valid_loader))[:N]
178         img = (2*img/255.0)-1 #Normalise between [-1, 1]
179         samples = torch.zeros(N,3,64,64)
180
181         #some extra for when FFT+converting to gray scale is needed
182         samples2 = np.zeros((N, 64, 64, 3))
183         img2 = np.zeros((N,64, 64, 3))
184         samplesGray = np.zeros((N, 64, 64)) #store gray scale

```

```

179     imgGray = np.zeros((N, 64, 64))           #store gray scale
180     SAMPLES = np.zeros((N, 64, 64))           #store freq domain
181     IMG = np.zeros((N,64,64))                 #store freq domain
182
183     for i in range(N):
184         samples[i] = model(img[i].unsqueeze(dim=0), torch.tensor(0))
185
186     img = (img+1)/2           #Normalise between [0,1]
187     samples = (samples+1)/2 #Normalise between [0,1]
188
189     if fft==1:
190         samples = samples.detach().numpy()
191         img = img.detach().numpy()
192
193         for i in range(N): #permute submatrices
194             samples2[i] = np.transpose(samples[i], (1, 2, 0))
195             img2[i] = np.transpose(img[i], (1, 2, 0))
196
197         for i in range(N): #convert to gray scale
198             samplesGray[i] = color.rgb2gray(samples2[i])
199             imgGray[i] = color.rgb2gray(img2[i])
200             SAMPLES[i] = 20*np.log10(abs(np.fft.fft2(samplesGray[i])))
201             IMG[i] = 20*np.log10(abs(np.fft.fft2(imgGray[i])))
202
203         plt.subplot(2, 2, 1)
204         plt.imshow(SAMPLES[0], vmin=-65, vmax=65, cmap='jet', aspect='auto')
205         plt.colorbar()
206         plt.subplot(2, 2, 2)
207         plt.imshow(SAMPLES[1], vmin=-65, vmax=65, cmap='jet', aspect='auto')
208         plt.colorbar()
209         plt.subplot(2, 2, 3)
210         plt.imshow(SAMPLES[2], vmin=-65, vmax=65, cmap='jet', aspect='auto')
211         plt.colorbar()
212         plt.subplot(2, 2, 4)
213         plt.imshow(SAMPLES[3], vmin=-65, vmax=65, cmap='jet', aspect='auto')
214         plt.colorbar()
215         plt.show()
216
217     else:
218         vutils.save_image(img.cpu().float(), fp=data_name, nrow=2)
219         vutils.save_image(samples.cpu().float(), fp=result_name, nrow=2)
220     return img, samples
221
222 elif selector == 1: #CIFAR
223     img = next(iter(valid_loader))[:N]
224     img = img[0]
225     img = (2*img)-1 #Normalise between [-1, 1]
226     samples = torch.zeros(N,3,32, 32)
227     img2 = torch.zeros(N,3,32, 32)
228
229     for i in range(N):
230         samples[i] = model(img[i].unsqueeze(dim=0), torch.tensor(0))

```

```

231         img2[i] = img[i]
232
233         img2 = (img2+1)/2           #Normalise between [0,1]
234         samples = (samples+1)/2 #Normalise between [0,1]
235         vutils.save_image(img2.cpu().float(), fp=data_name, nrow=2)
236         vutils.save_image(samples.cpu().float(), fp=result_name, nrow=2)
237         return img2, samples
238
239     else: #sinc
240         x = next(iter(valid_loader))[:172]
241         sample = torch.zeros(1, 1, 172, 2)
242         sample = model(x, torch.tensor(0))
243         x = torch.squeeze(x)
244         sample = torch.squeeze(sample)
245         x = x.detach().numpy()
246         sample = sample.detach().numpy()
247         fig = plt.figure( )
248         plt.plot(x[:,1], x[:, 0], label='input')
249         plt.plot(sample[:, 1], sample[:, 0], label='reconstruction')
250         plt.xlabel("Time [-]")
251         plt.ylabel("sinc(5t), normalised to [-1, 1]")
252         plt.legend(loc='upper right')
253         fig.savefig(result_name)
254         return x, sample
255     #####
256
257     latent_dims = [64, 64, 1]           #latent dimensions
258     imSizeH = [64, 32, 172]            #image size 1 (horizontal)
259     imSizeV = [64, 32, 2]              #image size 2 (vertical)
260     nChan = [3, 3, 1]                  #number of channels
261
262     for k in select:
263         print("the selected model is {}".format(k))
264         print(" ")
265         print("Step 1: Loading data")
266         valid_loader = fnc_getValid_loader(k)
267         path_ll, path_lr, path_sfl, path_sfr, path_sdl, path_sdr = fnc_getStore_and_Load(k) #get
268         ↪ location to load models, store figures
269
270         print("Saving figures for left column")
271         for j in range(3):
272             model = fnc_get_model(latent_dims=latent_dims[k], nChan=nChan[k], imSizeH=imSizeH[k],
273             ↪ imSizeV=imSizeV[k])
274             model.load_state_dict(torch.load(path_ll[j]))
275             model.eval()
276             data, samples = fnc_plot(model, path_sdl[j], path_sfl[j], valid_loader, k)
277
278         print("Saving figures for right column")
279
280         for j in range(3):
281             model = fnc_get_model(latent_dims=latent_dims[k], nChan=nChan[k], imSizeH=imSizeH[k],
282             ↪ imSizeV=imSizeV[k])

```

```

280         model.load_state_dict(torch.load(path_lr[j]))
281         model.eval()
282
283         data, samples = fnc_plot(model, path_sdr[j], path_sfr[j], valid_loader, k)
284     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].

```

1  """
2  Date: 25-03-2022
3  Author: Austin, Mustafa, Richard, Dimme
4  Based on: [1], [2], [3], [4], [5], [6]
5  Descr: Part of Code for Table 1 of the paper " Variational Autoencoders: A Harmonic
        ↳ Perspective"
6
7          Store weights from (1) decoders and (2) encoders from models made using
        ↳ Step1-model_training.py
8          as a .mat file. This .mat file can be used to estimate Lipschitz constant L using
        ↳ code from [4]
9
10         note: does not store biases. For the left column of table 1, decoders are stored (L).
        ↳ For the right column, encoders are stored (R)
11
12 Sources:
13     [1] examples from https://github.com/didriknielsen/survae_flow
14     [2] https://avandekleut.github.io/vae/
15     [3] The paper: Variational Autoencoders: A Harmonic Perspective
16     [4] Lab 8 of CS4240-Deep Learning course
17     [5] https://medium.com/dataserries/variational-autoencoder-with-pytorch-2d359cbf027b
18     [6] LipSDP: https://github.com/arobey1/LipSDP
19 Notes:
20     (a) you need to have trained the models before using this file, and stored at the proper
        ↳ location.
21     (b) you need to have a folder Weights inside the directory where this .py file is stored.
22 """
23
24 import torch
25 from torch import nn
26 from scipy.io import savemat
27 import numpy as np
28
29 #User settings
30 select = [2] #select which models, 0: CelebA, 1: Cifar, 2: Sinc
31
32 #Define model architecture
33 #####
34 class VariationalEncoder(nn.Module):

```

```

35     def __init__(self, latent_dims, nChan, imSizeH, imSizeV):
36         super(VariationalEncoder, self).__init__()
37         self.input = nn.Flatten(1)
38         self.dense1 = nn.Linear(imSizeH*imSizeV*nChan, 256)
39         self.dense2 = nn.Linear(256, 256)
40         self.dense3 = nn.Linear(256, 256)
41         self.outputMu = nn.Linear(256, latent_dims)
42         #self.outputSig = nn.Linear(256, latent_dims)      #this layer is not needed as sigma is
43         ↪ fixed
44
45         self.N = torch.distributions.Normal(0, 1)
46         self.kl = 0
47
48     def forward(self, x, sigma_en):
49         x = self.input(x)
50         x = torch.sigmoid(self.dense1(x))
51         x = torch.sigmoid(self.dense2(x))
52         x = torch.sigmoid(self.dense3(x))
53         mu = self.outputMu(x)
54         #sigma = torch.exp(self.outputSig(x))      #this layer is not needed as sigma is
55         ↪ fixed
56
57         z = mu + (sigma_en**2)*self.N.sample(mu.shape)      #combination of encoders output of
58         ↪ layer2, layer3
59         self.kl = (sigma_en**2 + mu**2 - torch.log(sigma_en) - 1/2).sum()
60         return z
61
62 class Decoder(nn.Module):
63     def __init__(self, latent_dims, nChan, imSizeH, imSizeV):
64         super(Decoder, self).__init__()
65         self.dense1 = nn.Linear(latent_dims, 256)
66         self.dense2 = nn.Linear(256, 256)
67         self.dense3 = nn.Linear(256, 256)
68         self.output = nn.Linear(256, nChan*imSizeH*imSizeV)
69         self.outputShape = nn.Unflatten(dim=1, unflattened_size = (nChan, imSizeH, imSizeV))
70
71     def forward(self, z):
72         z = torch.sigmoid(self.dense1(z))
73         z = torch.sigmoid(self.dense2(z))
74         z = torch.sigmoid(self.dense3(z))
75         z = self.output(z)
76         z = self.outputShape(z)
77         return z
78
79 class fnc_get_model(nn.Module):
80     def __init__(self, latent_dims, nChan, imSizeH, imSizeV):
81         super(fnc_get_model, self).__init__()
82         self.encoder = VariationalEncoder(latent_dims, nChan, imSizeH, imSizeV)
83         self.decoder = Decoder(latent_dims, nChan, imSizeH, imSizeV)
84
85     def forward(self, x, sigma_en):
86         z = self.encoder(x, sigma_en)

```

```

84         return self.decoder(z)
85 #####
86
87 #####
88 #Edited version of code presented in [4]
89 def extract_weights(net):
90     weights = []
91     bias = []
92     for param_tensor in net.state_dict():
93         tensor = net.state_dict()[param_tensor].detach().numpy().astype(np.float64)
94         if 'weight' in param_tensor:
95             weights.append(tensor)
96         if 'bias' in param_tensor:
97             bias.append(tensor)
98     return weights, bias
99 #####
100
101 #####
102 def fnc_getStore_and_Load(selector):
103     if selector==0: #CelebA
104         #Save/load paths for CelebA, left column
105         path_ll = ["Models/modelL1-sigma-1_0", "Models/modelL1-sigma-0_5",
106                  ↪ "Models/modelL1-sigma-0_1"] #Load locations
107         path_sw1 = ["Weights/WeightsL1-sigma-1_0.mat", "Weights/WeightsL1-sigma-0_5.mat",
108                  ↪ "Weights/WeightsL1-sigma-0_1.mat"] #Store locations
109
110         #Save/load paths for CelebA, right column
111         path_lr = ["Models/modelR1-sigma-1_0", "Models/modelR1-sigma-0_5",
112                  ↪ "Models/modelR1-sigma-0_0"] #Load locations
113         path_swr = ["Weights/WeightsR1-sigma-1_0.mat", "Weights/WeightsR1-sigma-0_5.mat",
114                  ↪ "Weights/WeightsR1-sigma-0_0.mat"] #Store locations
115
116     elif selector == 1: #Cifar
117         #Save/load paths for CIFAR, left column
118         path_ll = ["Models/modelL2-sigma-1_0", "Models/modelL2-sigma-0_5",
119                  ↪ "Models/modelL2-sigma-0_1"] #Load locations
120         path_sw1 = ["Weights/WeightsL2-sigma-1_0.mat", "Weights/WeightsL2-sigma-0_5.mat",
121                  ↪ "Weights/WeightsL2-sigma-0_1.mat"] #Store locations
122
123         #Save/load paths for CIFAR, right column
124         path_lr = ["Models/modelR2-sigma-1_0", "Models/modelR2-sigma-0_5",
125                  ↪ "Models/modelR2-sigma-0_0"] #Load locations
126         path_swr = ["Weights/WeightsR2-sigma-1_0.mat", "Weights/WeightsR2-sigma-0_5.mat",
127                  ↪ "Weights/WeightsR2-sigma-0_0.mat"] #Store locations
128
129     else: #Sinc
130         #Save/load paths for SINC, left column
131         path_ll = ["Models/modelL3-sigma-1_0", "Models/modelL3-sigma-0_5",
132                  ↪ "Models/modelL3-sigma-0_1"] #Load locations
133         path_sw1 = ["Weights/WeightsL3-sigma-1_0.mat", "Weights/WeightsL3-sigma-0_5.mat",
134                  ↪ "Weights/WeightsL3-sigma-0_1.mat"] #Store locations

```

```

126     #Save/load paths for SINC, right column
127     path_lr = ["Models/modelR3-sigma-1_0", "Models/modelR3-sigma-0_5",
128     ↪      "Models/modelR3-sigma-0_0"]          #Load locations
129     path_swr = ["Weights/WeightsR3-sigma-1_0.mat", "Weights/WeightsR3-sigma-0_5.mat",
130     ↪      "Weights/WeightsR3-sigma-0_0.mat"]    #Store locations
131
132     return path_ll, path_lr, path_swl, path_swr
133 #####
134
135 latent_dims = [64, 64, 1]          #latent dimensions
136 imSizeH = [64, 32, 172]           #image size 1 (horizontal)
137 imSizeV = [64, 32, 2]             #image size 2 (vertical)
138 nChan = [3, 3, 1]                 #number of channels
139
140 #Loop through all six models
141 for k in select:
142     print("the selected model is {}".format(k))
143     print(" ")
144     path_ll, path_lr, path_swl, path_swr = fnc_getStore_and_Load(k)
145
146     print("storing weights for left column")
147     for j in range(3):
148         model = fnc_get_model(latent_dims=latent_dims[k], nChan=nChan[k], imSizeH=imSizeH[k],
149         ↪      imSizeV=imSizeV[k])
150         model.load_state_dict(torch.load(path_ll[j]))
151         model.eval()
152
153         weights2, bias = extract_weights(model.decoder)
154         data = {'weights': np.array(weights2, dtype=object)}
155         savemat(path_swl[j], data)
156         print("Stored weights from model {} at location {}".format(path_ll[j], path_swl[j]))
157         print(" ")
158
159     print("storing weights for right column")
160     for j in range(3):
161         model = fnc_get_model(latent_dims=latent_dims[k], nChan=nChan[k], imSizeH=imSizeH[k],
162         ↪      imSizeV=imSizeV[k])
163         model.load_state_dict(torch.load(path_lr[j]))
164         model.eval()
165
166         weights, _ = extract_weights(model.encoder)
167         data = {'weights': np.array(weights, dtype=object)}
168         savemat(path_swr[j], data)
169         print("Stored weights from model {} at location {}".format(path_lr[j], path_swr[j]))
170         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.

```

1 clear all
2 close all
3 clc
4
5 % source this method is based on:
6 %https://arxiv.org/pdf/2102.07559.pdf
7 %% input x
8 cifar10_data=importdata('cifar10_data.png');
9 imge = cifar10_data(3:34,3:34,1:3); %rand(1,32*32*3);
10 imge=reshape(im2double(imge),[1,3072]);
11 %% optimize
12 % describe the optimization problem
13 net_imge = network_fnc_cif(imge);
14 nonlcon = @non_linear_cont;
15 options =
16     ↳ optimoptions('fmincon','Display','iter','Algorithm','interior-point','MaxIterations',1000,'MaxFunctionEvaluations',1000);
17 fun = @(delta)-(norm(network_fnc_cif(imge+delta)-net_imge));
18 x0 = ones(1,32*32*3)'.*10^-5;
19 % can be used to force decoded attacked signal into y \in [-1,1]
20 %A = [AB_ende; -AB_ende];
21 %b = [ones(344,1)-AB_ende*x_t_sinc;ones(344,1)+AB_ende*x_t_sinc];
22 %
23 delta = fmincon(fun,x0,[],[],[],[],[],[],nonlcon,options);
24
25 const_1 = norm(delta);
26
27 %% plot
28
29 est_or = network_fnc_cif(imge);
30 est_att = network_fnc_cif(imge+delta);
31 % plot(imge(2:2:344),imge(1:2:343))
32 % hold on
33 % plot(est_or(2:2:344),est_or(1:2:343))
34 % plot(est_att(2:2:344),est_att(1:2:343))
35 % legend("og_sinc", "decoded sinc", "attacked decoded sinc")
36
37 figure
38 image(img_size_convert(imge,32,32,3))
39 title('original image')
40 figure
41 image(img_size_convert(imge+delta,32,32,3))
42 title('attacked image')
43 figure

```



```

44 image(img_size_convert(network_fnc_cif(imge),32,32,3))
45 title('original decoded image')
46 figure
47 image(img_size_convert(network_fnc_cif(imge+delta),32,32,3))
48 title('attacked decoded image')
49
50 %% loglikelihood degradation
51 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.

```

1 function y = sig_imp (x)
2 y = zeros(length(x),1);
3 for i = 1:length(x)
4     y(i,1) = 1/(1+exp(-x(i)));
5 end
6
7 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.

```

1 function [x_est] = network_fnc_cif(x)
2 % convert and load all the weights and biases generated by the learner
3 addpath('Weights2')
4 load('WeightsR2-sigma-1_0.mat')
5
6
7
8 encode1 = cell2mat(weights(1));
9 encode2 = cell2mat(weights(2));
10 encode3 = cell2mat(weights(3));
11 encodeu = cell2mat(weights(4));
12 load('WeightsL2-sigma-1_0.mat')
13 decode1 = cell2mat(weights(1));
14 decode2 = cell2mat(weights(2));
15 decode3 = cell2mat(weights(3));
16 decode4 = cell2mat(weights(4));
17
18
19
20 load('biasR2-sigma-1_0.mat')
21 bias_en1 = cell2mat(bias(1));
22 bias_en2 = cell2mat(bias(2));
23 bias_en3 = cell2mat(bias(3));

```

```

24 bias_enu = cell2mat(bias(4));
25 load('biasL-sigma-1_0.mat')
26 bias_de1 = transpose(cell2mat(bias(1)));
27 bias_de2 = transpose(cell2mat(bias(2)));
28 bias_de3 = transpose(cell2mat(bias(3)));
29 bias_de4 = transpose(cell2mat(bias(4)));
30
31 z =
    ↪ transpose(encodeu*sig_imp(encode3*sig_imp(encode2*sig_imp(encode1*x+bias_en1)+bias_en2)+bias_en3)+bias_enu);
32 x_est =
    ↪ transpose(decode4*sig_imp(decode3*sig_imp(decode2*sig_imp(decode1*z+bias_de1)+bias_de2)+bias_de3)+bias_de4);
33
34 end

```

B.2.4 Non linear constraints

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

```

1 function [c, ceq] = non_linear_cont(delta)
2 c_d = 20;
3 c(1) = norm(delta)-c_d;
4 ceq = [];
5 end

```

B.2.5 Image size conversion

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

```

1 function y = img_size_convert(x, a, b, c)
2 len_x = length(x);
3 y = zeros(a,b,c);
4 for i = 0:c-1
5     x_tmp = x(i*len_x/c+1:(i+1)*len_x/c);
6     len_x_tmp = length(x_tmp);
7     for j = 0:b-1
8         y(j+1,:,i+1) = x_tmp(j*len_x_tmp/a+1:(j+1)*len_x_tmp/a);
9     end
10 end
11 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/dataserries/variational-autoencoder-with-pytorch-2d359cbf027b>, 2021. Accessed on 13-04-2022.