Exercise 3: Group 38

Dominik Eitler, Gwendolyn Rippberger, Marko Spegel-Grünberger

July 31, 2020

1 Introduction

For this exercise we tried to recreate a model inversion attack on different models that are used for facial recognition. Our experiments are based on the paper of Fredrikson et. al [1]. We worked with the assumption that the client is able to download the model description, a so called white-box setting. The models attacked are taken from the paper Fredrikson et al., the first one being a softmax regression which is a neural network without a hidden layer. The output layer uses softmax as an activation function and consists of 40 neurons (1 for each class).

The second model tested is a multilayer perceptron (MLP) with one hidden layer of 3000 sigmoid neurons and a softmax output layer. This model performs a non-linear transformation on the feature vector and uses the softmax output layer to classify.

The last model is a stacked denoising autoencoder (DAE). For this model, we split the process into two steps. Firstly, training two separate denoising autoencoder (1000 and 300 tanh units). Secondly, we took the output of the second encoder as input for the softmax regression for classification. In addition to the models that were used in the paper, we decided to try the attack on a convolutional neural network (CNN) which is the most complex of our models. It consists of three convoluational layers and two linear layers again using softmax as activation function and output layer. We divided the experiment into two phases. We first tried to (re-)create the models, training them with our training set. Then using the trained models we attack them using the method mentioned in the paper Fredrikson et al.[1] as reconstruction attack. Summarized, the attack tries to reconstruct images iterative without any knowledge of the training data. Generally, we tried to stick as much as possible to the information and settings explained in the paper.

2 Data

The paper used the AT&T Laboratories Cambridge database of faces. As the link for the dataset from the paper expired and is not working (accessed July 31, 2020), we looked for another source. We found a github repository[1] which offered a framework including the dataset that was initially used.

The dataset consists of 40 different people for which each of them has 10 different images (in total 400 images) taken different angles, etc. Based on the paper, we used a train-test split of 7-3, 7 images for training and 3 for testing. The images are grayscale (values are from 0 to 255) and have a size of 112x92. In the paper it was mentioned that they want to recreate a full vector of pixel intensities that correspond to a floating-point between [0,1]. Therefor we normalized all of our pixel values to a range between [0,1] before using them for training the models.

3 Models

The models were implemented using the `pytorch` library [2]. If not stated otherwise, we used default settings. As in the paper [1] we trained our models using `pytorch`'s stochastic gradient descent (SDG) algorithm. If the model did not show any improvement after 100 iterations we stop otherwise we continue training. Depending on the model we tried a variety of different learning rates. We used batches of 10 images and measured the accuracy on the test dataset which contains 280 images. We know that the values seem overly too good. We randomly checked predictions of the models and can confirm that all of the classes we checked were rightly predicted.

[1] `https://github.com/roshanshrestha01/face-recgonition-cnn`, accessed July 31, 2020
[2] `https://pytorch.org/`, accessed July 31, 2020

Model	Accuracy
Softmax	1.00
MLP	1.00
DAE	0.93
CNN	1.00

Table 1: Table showing the results of the models in terms of accuracy using the test dataset

3.1 Softmax regression

The softmax is our simplest model. Converting the images into a 1-D feature vector with 10304 values (111x92 pixels), we use this as input. The output layer consists of 40 nodes which use softmax as an activation function. The output is a probability distribution for the different classes: each element is non-negative and the sum over all components is 1. We used a batch size of 10 images for training. As mentioned in the paper [1], we trained the model until the model's performance on the training set failed to improve after 100 iterations. As there was no further explanation on which criterion was used for the (SDG) algorithm, we used `CrossEntropyLoss`. As performance measure we take the accuracy of the test set. The softmax model usually converged after 5-10 iterations (without the 100 iterations checking for improvement) with an accuracy of 1 (using a learning rate of 0.1).

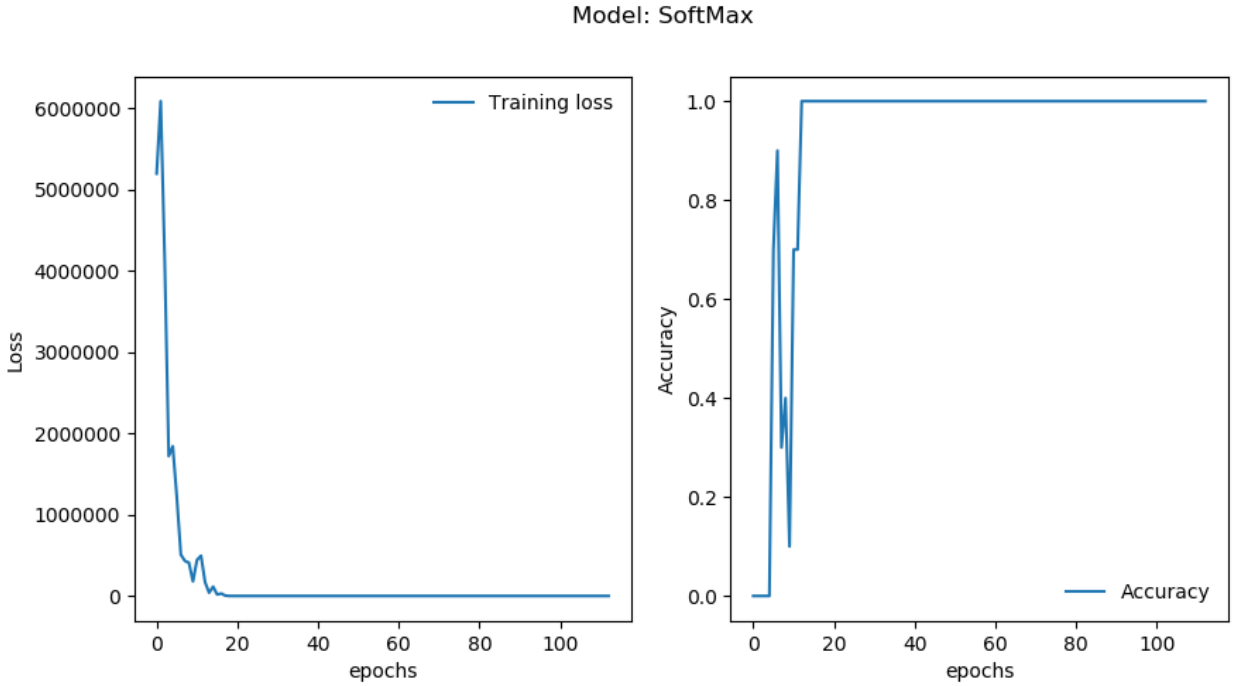

Figure 1: The plots show the training loss and the accuracy over the epochs for the Softmax model.

3.2 Multilayer perceptron

The multilayer perceptron described in the paper [1] consists of one hidden layer of 3000 sigmoid and a softmax outpuf layer. Starting with a learning rate of 0.1 and the initial experimental setup, we only achieved an accuracy of 0.3. Therefor we decided to try tweaking with different learning rates and initial weights. Nevertheless, training the model with waiting 100 epochs for improvement takes a relatively long

time. We tried different weights offer by the pytorch library e.g. `normal` taking values drawn from a normal distribution, `ones` with a tensor full of 1s, `xavier_normal` with values according to the method described in *Understanding the difficulty of training deep feedforward neural networks - Glorot, X. & Bengio, Y. (2010)*, using a normal distribution. Usually the accuracy never improved more than 0.3 but using the normal distributed values as initial weights improved our model to having an accuracy of 1. We used `CrossEntropyLoss` as optimizer and a learning rate of 0.1. The model usually improved in the first 10 epochs as can be seen in Figure 2.

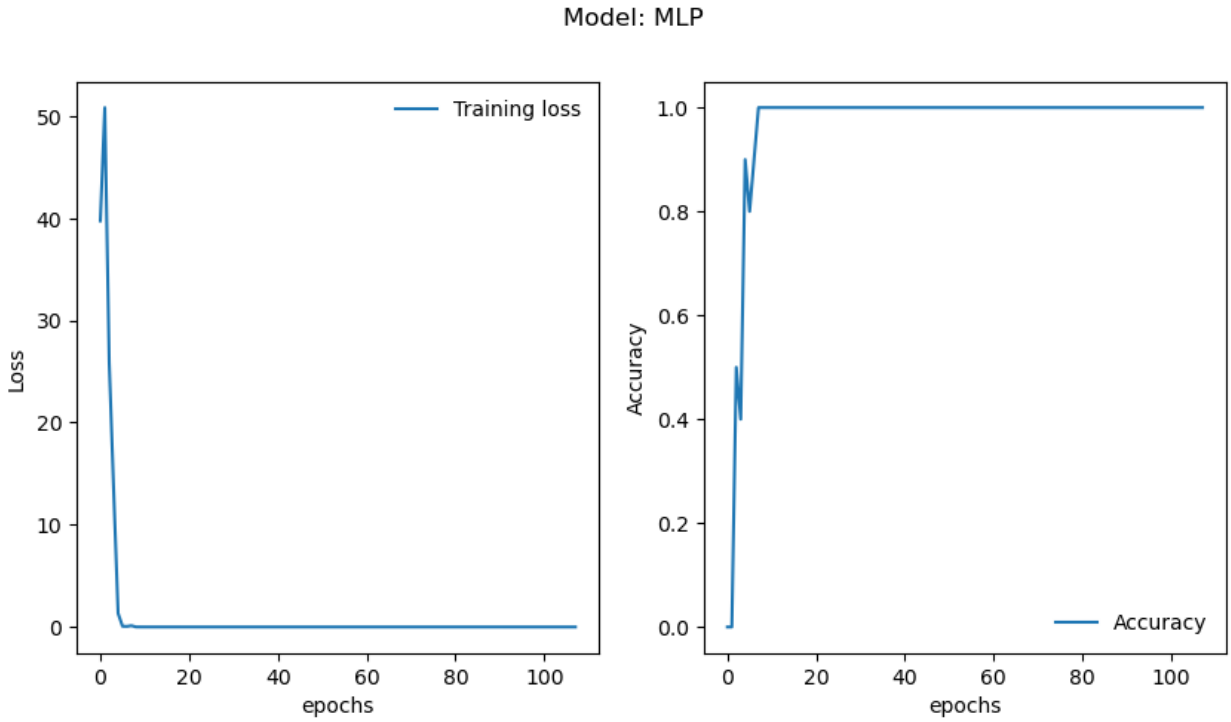

Figure 2: The plots show the training loss and the accuracy over the epochs for the MLP model. We can see that the model improved quite quickly with our improvements made.

3.3 Stacked Denoising Autoencoder

An autoencoder is a type of multi-layer perceptron which is mainly used for dimensionality reduction and contains two parts. In the encoding part a high-dimensional input such an image is mapped to a lower-dimensional output, while in the decoding part, the network is trained to map this generated output back to its original input as good as possible. For a denoising autoencoder, the original input is intenionally corrupted with some kind of noise to improve the encoding and decoding performance on corrupted data and increase the overall stability of the network. In the paper [1] a stacked denoising autoencoder is described, which works as the following: The original input is an image of size $112 \cdot 92 = 10304$, which dimensionality is first decreased to 1000 with a pretrained autoencoder layer, then again decreased to 300 with another pretrained layer. The last layer of this architecture is described as a linear layer with a softmax activation function similar to 3.1.

The pretrained DAE layers were trained with *mean squared reconstruction error*, where *reconstruction* means that the error is computed between output and input and not output and ground truth. In the paper [1], *stochastic gradient descent* is used as optimizer. This, however, did not work for us, so instead we used

the Adam optimizer[3], which we found being used in a lot of related work. The two autoencoder layers were trained separately, the first one with the original input with added binomial corruption and the second one with the (also noised) outputs of the first layer's encoder as inputs. Both of the encoding functions have a linear layer with *tanh* activation function, while the decoding functions used a linear activation (no separate activation function). For the softmax layer we used cross-entropy loss and stochastic gradient descent on the twice encoded images as data inputs.

The autoencoder layers were both trained with a learning rate of $1 \cdot 10^{-4}$ and 5000 epochs each, which, after some experiments, we found to be the best performing. The en- and decoded images can be seen in Figure 3. For the softmax layer a learning rate of $1 \cdot 10^{-2}$ for 2000 epochs was chosen. With this, the classification of the images ended up with a loss of less than 0.45 and an accuracy of 93.3% (Figure 4).

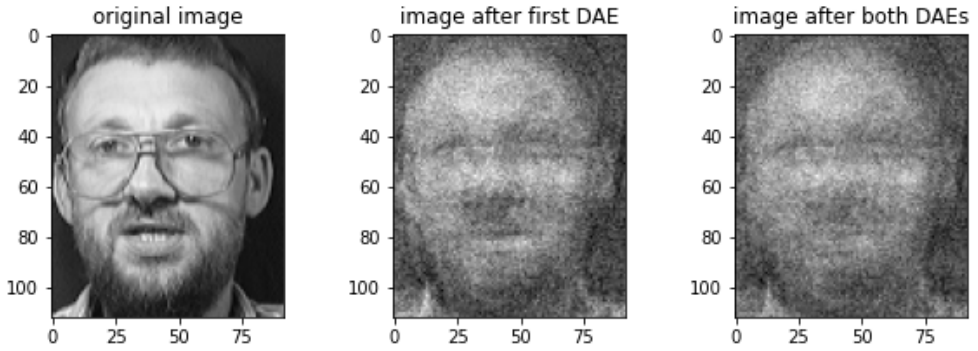

Figure 3: The image in the middle displays the once encoded (to 1000 features) and decoded image. The image on the right is encoded to 1000 and then to 300 feature and then decoded. On the left, the original image can be seen.

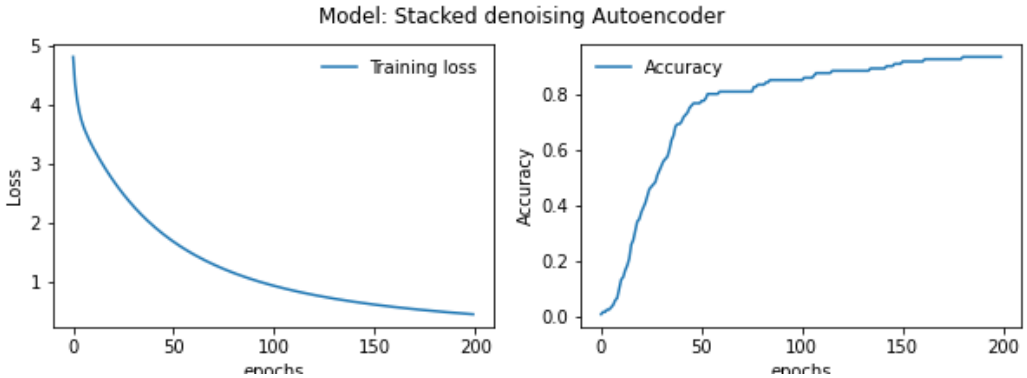

Figure 4: The plots show the training loss and the accuracy over the epochs for the stacked denoising autoencoder model.

3.4 Convolutional Neural Network

For the convolution neural network, which is our most complex model, we used two convolutional layers, each using a kernel size of 3 and padding of 1. We then use max-pooling for down sampling on both layers using a kernel size of 2. After the down sampling we use a drop out rate of 0.2 and then apply one linear transformations, from the image space, 10304 to 512, then from 512 to our 40 nodes output layer using the

[3] `https://pytorch.org/docs/stable/optim.html#torch.optim.Adam`, accessed July 31, 2020

softmax function to be consistent with the other models. The model was trained with the `CrossEntropyLoss` as optimizer (with SDG) using a learning rate of 0.001. Again, we achieve an accuracy of 1 on the test set which can be seen in Figure 5. The model improves quickly in the first few epochs as well as the Softmax and MLP.

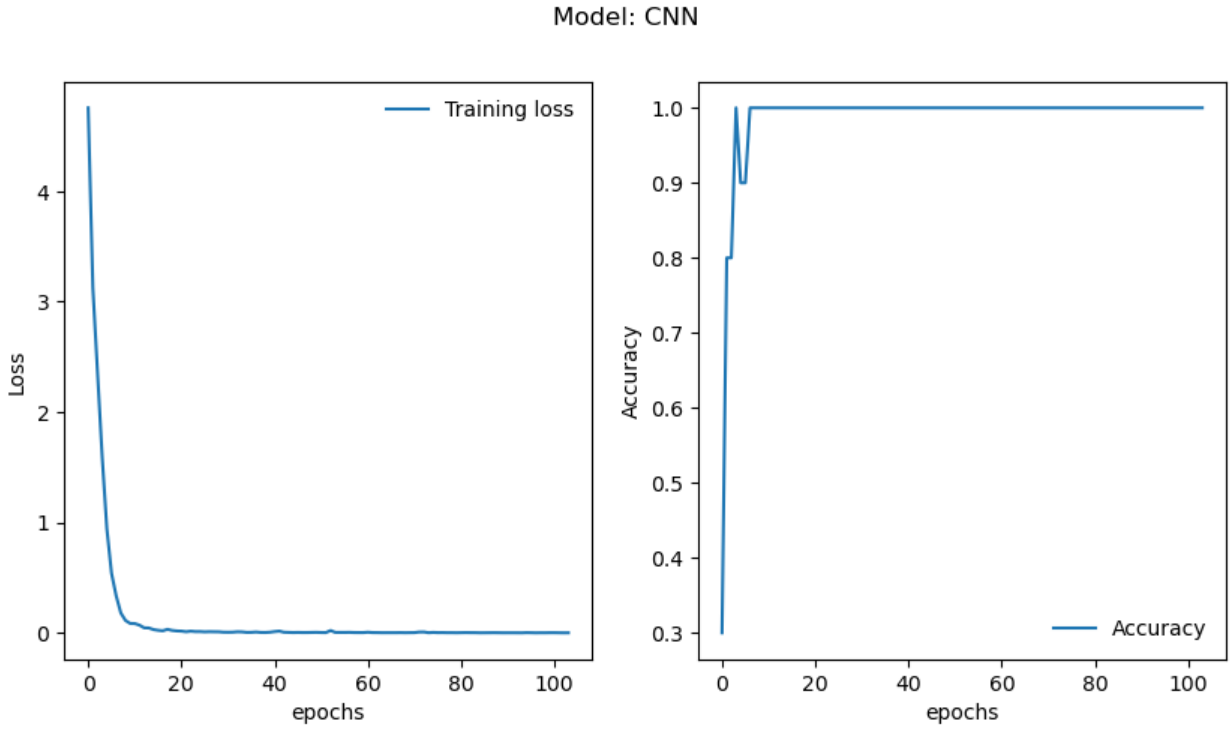

Figure 5: The plots show the training loss and the accuracy over the epochs for the CNN model.

4 Reconstruction Attack

The MI attack presented in Fredrikson et al. (see Algorithm 1), has for our case of reconstructing an image, an $AUXTERM(x) \stackrel{def}{=} 0$ for all x as we assume that the attacker has no auxiliary information aside the target label. Therefor the cost function is described as $c(x) \stackrel{def}{=} 1 - \tilde{f}_{label}(x) + 0$. The algorithm applies gradient descent up α iterations, using gradient steps of size λ. In our implementation we worked with the stochastic gradient descent offered by the `pytorch` library. There are only two cases in which the algorithm ends earlier: if the cost of the candidate fails to improve after β iterations or if the cost is as great as γ.

For the stacked DAE uses the *Process* function defined in Algorithm 2 post-processing step. For the other models, we use the identity function. The original parameters of the experiment (defined by the authors through experience of running the algorithm on test data) are: $\alpha = 5000, \beta = 100, \gamma = 0.99$ and $\lambda = 0.1$.

After going trough the algorithm, we noticed that the cost function only takes values in a range of [0,1] due to the fact that the probabilities are between 0 and 1 as well. The higher the probability, meaning the model classifies with more certainty a class label, the smaller the cost. If $\gamma = 0.99$, the model basically stops whenever there is no improvement after β iterations or the probability is very low, so the cost is high. Taking this into consideration, we used $\gamma = 0.01$ to conduct our experiment.

Algorithm 1 Inversion attack for facial recognition models, see Fredrikson et a. [1]

function MI-FACE($label, \alpha, \beta, \gamma, \lambda$)
  $c(x) \stackrel{def}{=} 1 - \tilde{f}_{label}(x)$ + AUXTERM(x)
  $x_0 \leftarrow 0$
  for $i \leftarrow 1...\alpha$ **do**
    $x_i \leftarrow$ PROCESS$(x_{i-1} - \lambda \cdot \nabla c(x_{i-1}))$
    if $c(x_i) \geq max(c(x_{i-1}), ..., c(x_{i-\beta}))$ **then**
      break
    if $c(x_i) \leq \gamma$ **then**
      break
  return $[argmin_{xi}(c(x_i)), min_{xi}(c(x_i))]$

Algorithm 2 Processing function for stacked DAE, see Fredrikson et al. [1]

function PROCESS-DAE(x)
  encoder.DECODE(x)
  $x \leftarrow$ NLMEANSDENOISE(x)
  $x \leftarrow$ SHARPEN(x)
  return *encoder*.ENCODE(vecs)

5 Performance Measures

To evaluate the performance of the attacks we used the mean squared error (MSE), normalized root mean squared error (NRMSE) and structural similarity (SSM) to compute the differences between the reconstructed image and the original image. All of them were computed using the methods provided by the scikit-image library [4].

5.1 Mean Squared Error

The MSE measures the average of error squares i.e. the average squared difference between the estimated values and true value. Values close to zero are better because they mean the reconstructed image is closer to the original image.

$$\text{MSE}(x, y) = \frac{1}{N} \sum_{i=1}^{N} (x - y)^2. \tag{1}$$

5.2 Normalized Root Mean Squared Error

Per default the scikit-image library uses the euclidean norm of the original image to normalize the root mean squared error (RSME). The formula is taken from the library documentation[5].

$$\text{NRMSE}(x, y) = \frac{\sqrt{MSE(x, y)} \cdot \sqrt{N}}{||y||} \tag{2}$$

5.3 Structural Similarity Index

The MSE is not highly indicative of perceived similarity. The Structural Similarity Index aims to address this shortcoming by taking texture into account. The implementation offered by skicit-image is based on

[4] `https://scikit-image.org/`, accessed July 31, 2020
[5] `https://scikit-image.org/docs/stable/api/skimage.metrics.html#normalized-root-mse`, accessed July 31, 2020

the paper of Wang et al. [2]. The two variables, c_1 and c_2 are there to stabilize the division with a weak denominator.

$$\text{SSIM}(x, y) = \frac{(2\mu_x\mu_y + c_1) \cdot (2\sigma_{xy} + c_2)}{(\mu_x^2 + \mu_y^2 + c_1) \cdot (\sigma_x^2 + \sigma_y^2 + c_1)} \tag{3}$$

The formula returns a value between -1 and 1 indicating how similar both images are. 0 means no structural similarities where as 1 means the images are identical.

6 Implementation

Library	**Version**
opencv-python	4.3.0.36
matplotlib	3.2.2
torch	1.5.1
torchvision	0.6.1
pandas	1.0.5
sklearn	0.23.1
scikit-image	0.16.2

Table 2: Python libraries and their versions used during this project.

In the Table 2 the libraries and their versions that were used during this project can be found. The submission folder contains a `README.md` with instructions on how to use the framework provided. As the models take some time to build and the reconstruction attack some time to invert, we are providing the results of both in the submission file. The trained models to only try the reconstruction attack (can be found in the `models` folder) or the images of the reconstruction attack to inspect (can be found in `data/results/`). The reconstruction attack `reconstruct.py` provides to possibility to try different parameters:

- model: model being attacked, either Softmax, MLP, DAE, CNN or all
- alpha: number of epochs the SDG runs through (default 5000)
- beta: number of iterations that the algorithm waits for improvement (default 100)
- gamma: constraint value for the cost, the algorithm stops once the cost value is below (default 0.01)
- delta: learing rate of the SDG (0.1)

If not stated otherwise, we use the parameters default values which are the ones from the paper with the adapted gamma. Running the `prepare.py` there is also the possibility to try different train/test splits. The `modules.py` contains the code for training the models and the reconstruction attack. The `networks.py` contains our models specifications.

In order to run either only parts of the project or the whole, we provide different shell scripts.

- `run.sh`: trains all models and attacks using the values specified in this report, this file also includes the set up installing the required libraries and splitting the dataset into train and test set
- `train_models.sh`: only trains models
- `invert.sh`: only attacks models, parameters can be set

7 Results

In this section we present the results of the different reconstruction attacks taking the performance measures on average over all images. We also present the average runtime per class. A summary can be found in Table 3.

Model	Runtime	Epochs	MSE	NRMSE	SSM
Softmax	0.525	1.325	14551.602	264.935	0.002
MLP	15.2	78.55	12595.553	258.870	0.0002
DAE	0.25	101.05	546644.354	3.690	-7.977e-05
CNN	5.85	134.625	14910.904	0.538	6.658e-05

Table 3: Table showing the average results of the inversion attack.

7.1 Softmax

Although the Softmax Regression model is rather simple the attack worked quite well. Although the average performance might indicate otherwise as e.g. the similarity index is close to zero and indicates almost no structural similarity. To compare the results we show a case where the reconstruction worked well in Figure 6 and the worst result of the dataset in Figure 7. We see quite a drastic difference as the worst result does not show any outlines or similar. This shows that the initial image used which is a all black image with the original size is not improved at all although the algorithm stopped earlier due to the low cost value.

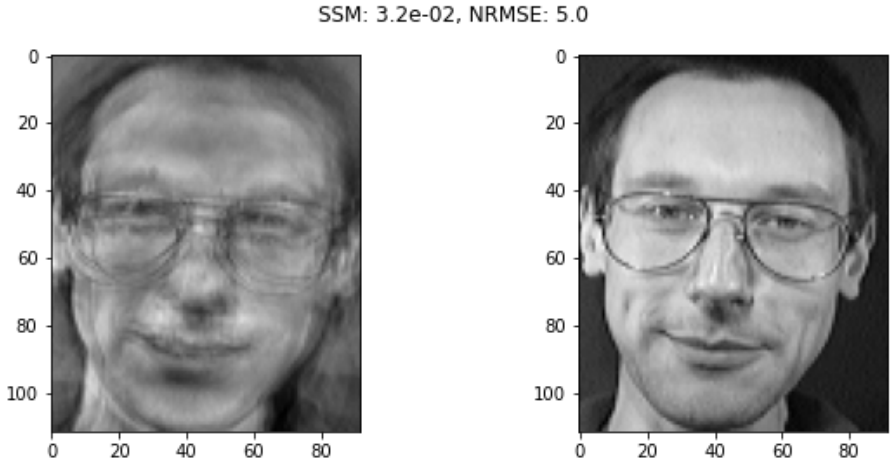

Figure 6: An example of a reconstruction attack on the Softmax model with a good result which is also represented in the SSM and NRMSE on top of the plot.

7.2 MLP

The MLP results seemed to generally be just noise but some showed outlines of faces. As there is only one image in the original paper [1] Figure 10 presenting the result of the reconstruction attack on the MLP model, we were not sure what to expect. Considering the attack always terminated because there was no improvement we thought a higher beta would improve the results. We tried using a beta higher betas up to a value of 1000 but the results did not improve as can be seen in Figure 10 and the algorithm still terminated

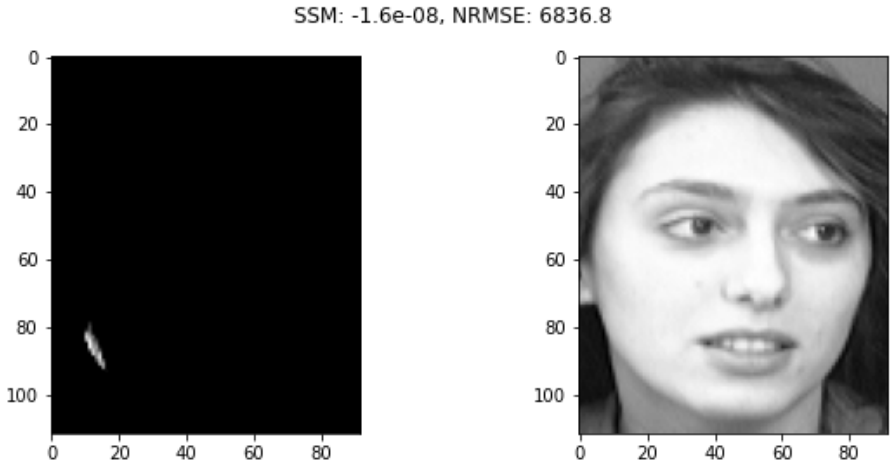

Figure 7: An example of a reconstruction attack on the Softmax model with a bad result which is also represented in the SSM and NRMSE on top of the plot. The initial black training image is basically not changed.

with high cost values although the model has an accuracy of 1 on the set set. The best result seen in Figure 8 is purely subjective as we see some outlines although the performance values say otherwise. The bad result seen in Figure 9 can be interpreted as just noise but is still better than the worst softmax result (Figure 7).

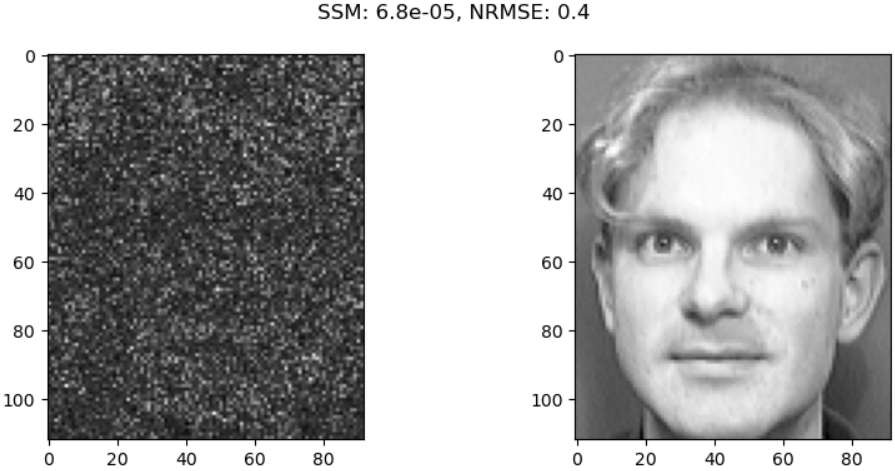

Figure 8: An example of a reconstruction attack on the MLP model with a relatively good result as we see outlines of a face although the performance measures suspect otherwise.

7.3 Stacked DAE

Since the stacked DAE could be seen as a Softmax Regressor with the DAE layers as preprocessing, we were expecting similar and even better results, than our Softmax network returned. Also the performance of the stacked DAE was the best-performing in the paper [1]. However, for us, this model was performing very poorly. Even after experiments and parameter tuning, we were only able to get the same generated image

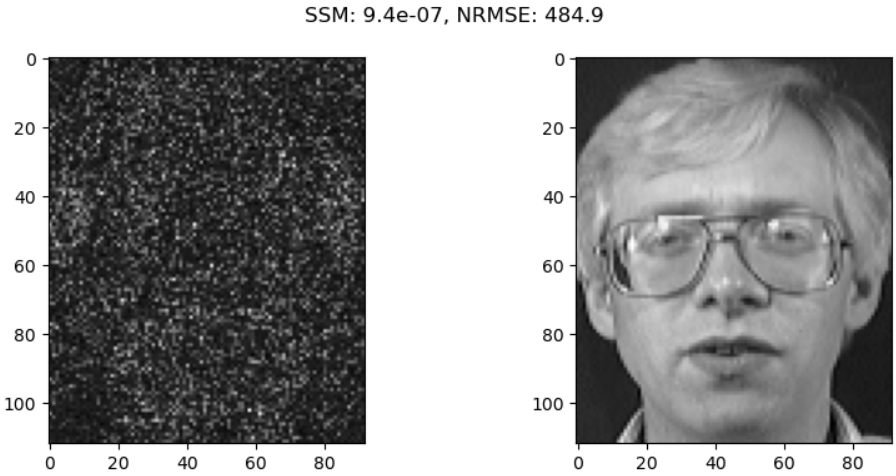

Figure 9: An example of a reconstruction attack on the MLP model with a bad result which is also represented in the SSM and NRMSE on top of the plot.

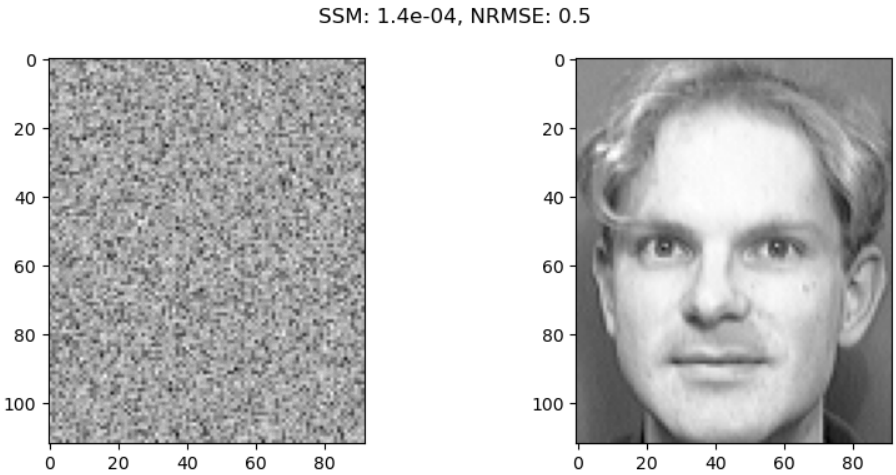

Figure 10: Best result after increasing beta to 1000.

for all classes 11.

The `process` function (described in Algorithm 2), improved the quality metrics slightly, but the output did not resemble the original images any closer, however it increased the run time by a factor of > 100.

7.4 CNN

For the attack on the CNN we can summarize that trying different parameters of the construction attack we were not successful reconstructing the training images. Most of the attacks stopped after not being able to improve the cost function. Still some classes were able to terminate with low cost values but the reconstructed image still looked as in Figure 12. Therefor we conclude that including convolutions into the model or making it in general more complex by including more layers made it safer against the reconstruction attacks.

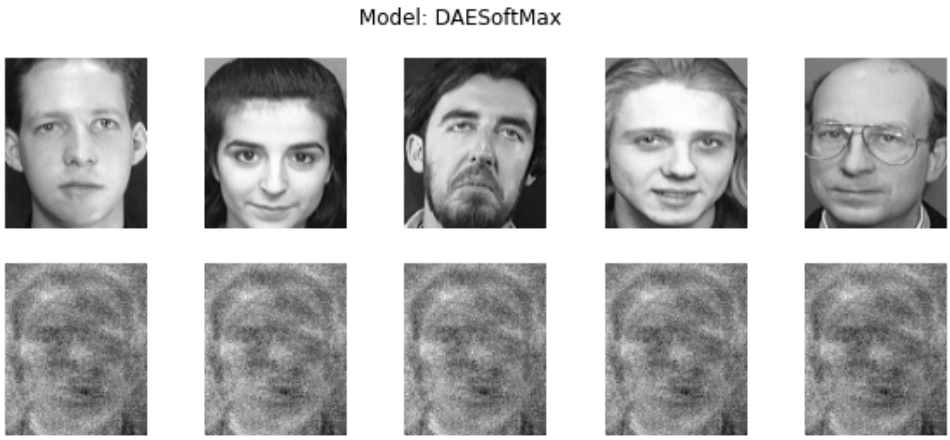

Figure 11: The inversion on the stacked DAE returns only one image.

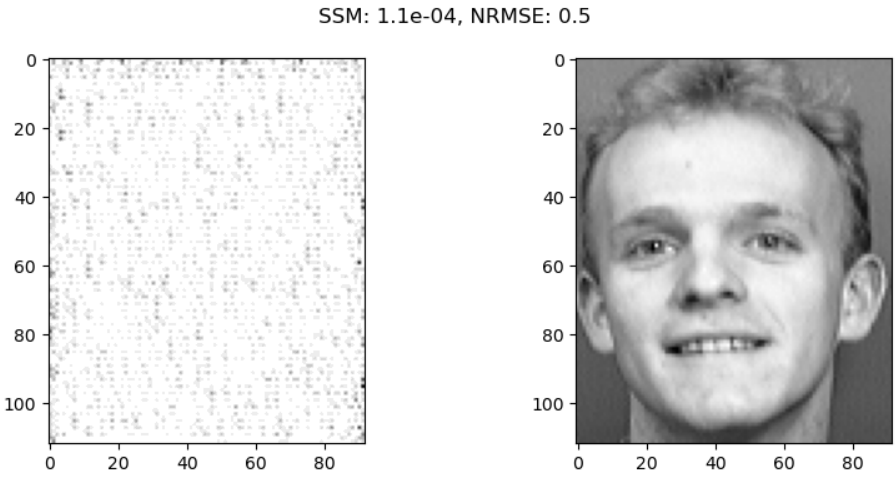

Figure 12: One of the results attacking the CNN model. The result is still unrecognizable although the attack terminated with a cost value of 0.05.

8 Analysis

Firstly, we realized that it was really difficult to reproduce the paper only based on its descriptions of the models and algorithms. A lot of details regarding the implementation e.g. learning rates, weights, exact train and test split (which images were used for training) etc. are not mentioned and have an influence on the quality of the results. Still, we were able to reproduce some of the attack results.

While inspecting the attack results of the different models, we noticed that always the same classes would be reconstructed well e.g. class label s18. Taking a look a the training and test set we noticed that it shows similar images of the person where as other labels had a higher variance in their training set images. If there was no recognizable image we took the cost value from the attack as reference.

If we take a look at the runtime of the models the DAE and the Softmax are by far the fastest. We think this is due to the model simplicity of the Softmax and the size of the returned vector for the DAE.

As mentioned before, the stacked denoising autoencoder, whose dae layers performed a good job at

reconstructing the original image from the reduced space and whose output layer also had a fairly good performance for classifying the images, did, against our expectations, not work well for the inversion attack. Even after a lot of experimentation and code revisions, we were not able, to generate an acceptable result with the model inversion algorithm (Algorithm 1). We cannot give a satisfying answer, why this is the case and how to fix this issue, but we are fairly certain, that there must be a bug in our implementation, that we were not able to comprehend.

Although all of our models have a high accuracy (higher than all the results describe in the paper), we were only able to successfully reconstruct images for the softmax model.

References

[1] Matt Fredrikson, Somesh Jha, and Thomas Ristenpart. Model inversion attacks that exploit confidence information and basic countermeasures. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page 1322–1333, New York, NY, USA, 2015. Association for Computing Machinery.

[2] Zhou Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing*, 13(4):600–612, 2004.