Course Project Report

Group 1

January 23, 2023

Deep Learning KU

Nr.	Name	Matrikelnummer
1	Julian Gruber	11777009
2	Massimiliano Viola	12213195
3	Haris Mujala	01609338
4	David Mihola	12211951

Contents

1 Introduction

In this project, we trained a convolutional autoencoder capable of correcting and regenerating clean images from their distorted versions. Using all the knowledge we acquired in assignment 3, we built an efficient encoder and focused our attention elsewhere. Specifically, we mainly experimented with different decoder architectures and various latent bottleneck dimensions to assess their relationship with performance on the task of reconstructing images. Then, in a later stage, we added controlled distortions using the Albumentations [1] library and evaluated the performance loss and visual result in comparison to reconstructing an undistorted input.

For the project, we used a subsample of the Animal Faces-HQ data set that we first found on Kaggle[1] and that was initially released with the StarGAN v2 [2] paper. The original data set has 512x512 resolution images and contains three classes: cats, dogs and wildlife. For our experiments, we pre-processed the data set by reducing the image size to 64x64 for memory constraints and fast experimentation while also removing the wildlife class due to high within-class variability.

(a) (b) (c)

Figure 1: Samples of 64x64 resized images.

Since dedicated validation samples were missing in the original release, we split the data into a validation set of 1000 images (500 cats and 500 dogs) and a training set of 8743 images, while leaving roughly 1000 images of cats and dogs from the original test set as our custom test set. The sampled and processed images are made available as part of our submission.

[1] https://www.kaggle.com/datasets/andrewmvd/animal-faces

2 Methods

2.1 Autoencoder Architecture

As mentioned in the previous section, we built the encoder using what we learned from assignment 3. In particular, we went with three convolution-convolution-pooling (CCP) blocks, where convolutional layers have a 3x3 kernel with `ReLU` activation and padding of 1 to keep the same size, while the max pooling layer shrinks the image size by a factor of 2. The number of filters in each CCP block doubles each time, going from 32 to 128 while the image resolution is gradually reduced from 64x64 to 8x8. Additionally, we added batch normalization after each convolutional layer for training stability. Normalized unit-range pixel values are passed to the network, so no normalization layer is used.

```
#encoder
tfl.Input(shape=(64, 64, 3)),
tfl.Conv2D(32, (3, 3), activation="relu", padding="same"),
tfl.BatchNormalization(),
tfl.Conv2D(32, (3, 3), activation="relu", padding="same"),
tfl.BatchNormalization(),
tfl.MaxPool2D(),
tfl.Conv2D(64, (3, 3), activation="relu", padding="same"),
tfl.BatchNormalization(),
tfl.Conv2D(64, (3, 3), activation="relu", padding="same"),
tfl.BatchNormalization(),
tfl.MaxPool2D(),
tfl.Conv2D(128, (3, 3), activation="relu", padding="same"),
tfl.BatchNormalization(),
tfl.Conv2D(128, (3, 3), activation="relu", padding="same"),
tfl.BatchNormalization(),
tfl.MaxPool2D(),
```

As for the latent bottleneck, we flattened the 8x8x128 representation into a 8192 vector and compressed it into a lower dimensional embedding, whose size was tested in a separate set of experiments, described in section 3.1.1.

```
# bottleneck
tfl.Flatten(),
tfl.Dense(bottleneck_size), # bottleneck size
tfl.BatchNormalization(),
```

For the decoder, we brought the image size back up to 64x64 using three architectures with roughly the same number of trainable parameters. The decoders consist of 3 varying layers/blocks of transposed convolution (see the list below). The number of filters in each block was reduced by a factor of 2 while going up towards the output. The batch normalization was kept also after each layer of the decoder and the transposed convolution had the same characteristics as the convolutions in the encoder, i.e. kernel size of 3x3 with `ReLU` activation and padding of 1.

All three decoders are followed by a final convolutional layer with `Sigmoid` activation to convert the features into unit-range pixel values in the 3 color channels (RGB).

The following descriptions and code snippets capture the architectures of the three types of decoders we tested. The study on which architecture works better is performed in section 3.1.

- **decoder 1**: 3 transposed convolutions with stride 2 and 64, 32, and 16 filters respectively. Transposed convolutions with stride 2 combine the ability to process information with the up-scaling.

```
# decoder type 1
tfl.Dense(4096),
tfl.BatchNormalization(),
tfl.Reshape((8, 8, 64)),
tfl.Conv2DTranspose(64, (3, 3), strides=2, activation="relu",
                    padding="same"),
tfl.BatchNormalization(),
tfl.Conv2DTranspose(32, (3, 3), strides=2, activation="relu",
                    padding="same"),
tfl.BatchNormalization(),
tfl.Conv2DTranspose(16, (3, 3), strides=2, activation="relu",
                    padding="same"),
tfl.BatchNormalization(),
tfl.Conv2D(3, (3, 3), activation="sigmoid", padding="same")
```

- **decoder 2**: 3 transposed convolutions with stride 1, but each convolution is followed by an up-sampling layer mirroring the encoder style with pooling. Transposed convolutions again have 64, 32, and 16 filters respectively.

```
# decoder type 2
tfl.Dense(4096),
tfl.BatchNormalization(),
tfl.Reshape((8, 8, 64)),
tfl.Conv2DTranspose(64, (3, 3), activation="relu",
                    padding="same"),
tfl.BatchNormalization(),
tfl.UpSampling2D(),
tfl.Conv2DTranspose(32, (3, 3), activation="relu",
                    padding="same"),
tfl.BatchNormalization(),
tfl.UpSampling2D(),
tfl.Conv2DTranspose(16, (3, 3), activation="relu",
                    padding="same"),
tfl.BatchNormalization(),
tfl.UpSampling2D(),
tfl.Conv2D(3, (3, 3), activation="sigmoid", padding="same")
```

- **decoder 3**: 3 blocks of 2-layer transposed convolution, where the first layer has stride 2 and the second layer has stride 1. Both layers in each block have 32,

16, and 8 filters respectively (to balance the number of trainable parameters, the number of filters in the blocks is halved).

```
# decoder type 3
tfl.Dense(4096),
tfl.BatchNormalization(),
tfl.Reshape((8, 8, 64)),
tfl.Conv2DTranspose(32, (3, 3), strides=2, activation="relu",
                    padding="same"),
tfl.BatchNormalization(),
tfl.Conv2DTranspose(32, (3, 3), strides=1, activation="relu",
                    padding="same"),
tfl.BatchNormalization(),
tfl.Conv2DTranspose(16, (3, 3), strides=2, activation="relu",
                    padding="same"),
tfl.BatchNormalization(),
tfl.Conv2DTranspose(16, (3, 3), strides=1, activation="relu",
                    padding="same"),
tfl.BatchNormalization(),
tfl.Conv2DTranspose(8, (3, 3), strides=2, activation="relu",
                    padding="same"),
tfl.BatchNormalization(),
tfl.Conv2DTranspose(8, (3, 3), strides=1, activation="relu",
                    padding="same"),
tfl.BatchNormalization(),
tfl.Conv2D(3, (3, 3), activation="sigmoid", padding="same")
```

2.2 Training Hyperparameters

We used the `Adam` optimizer with default parameters, i.e. 0.001 learning rate, as the optimization algorithm. We chose to minimize the Mean Squared Error (MSE) as a loss function. Considering the images as 3D targets, with a batch of N images being $\mathbf{Y} = \{\mathbf{y}_1, \mathbf{y}_2, ..., \mathbf{y}_N\}$ and corresponding predictions $\hat{\mathbf{Y}} = \{\hat{\mathbf{y}}_1, \hat{\mathbf{y}}_2, ..., \hat{\mathbf{y}}_N\}$, the loss is therefore simply:

$$J_{MSE} = \frac{1}{N} \sum_{i=1}^{N} (\hat{\mathbf{y}}_i - \mathbf{y}_i)^2 \tag{1}$$

The MSE basically represents how close the predicted pixel values are to the target pixel values, which is what we want to know ultimately.

Since the output pixels are normalized to the unit range, binary cross-entropy is also often used in practice[2], but we did not find this choice particularly suited for the nature of the problem and also preferred having a symmetric loss around the true pixel value.

[2]https://keras.io/examples/vision/autoencoder/

3 Results

3.1 Model Selection

As a first step, we ran a set of experiments to evaluate the reconstruction ability of the three networks with different decoders (see the list in 2.1) when dealing with images with no perturbation added.

The encoder architecture was fixed to the one described in 2.1 and the latent dimension size was set to 4096 to avoid the bottleneck effect for the moment. The networks were trained for a maximum of 50 epochs, but in all cases terminated due to early stopping, as we set the patience only to 3 epochs. The development of the loss on train and validation sets during training can be seen in figure 2.

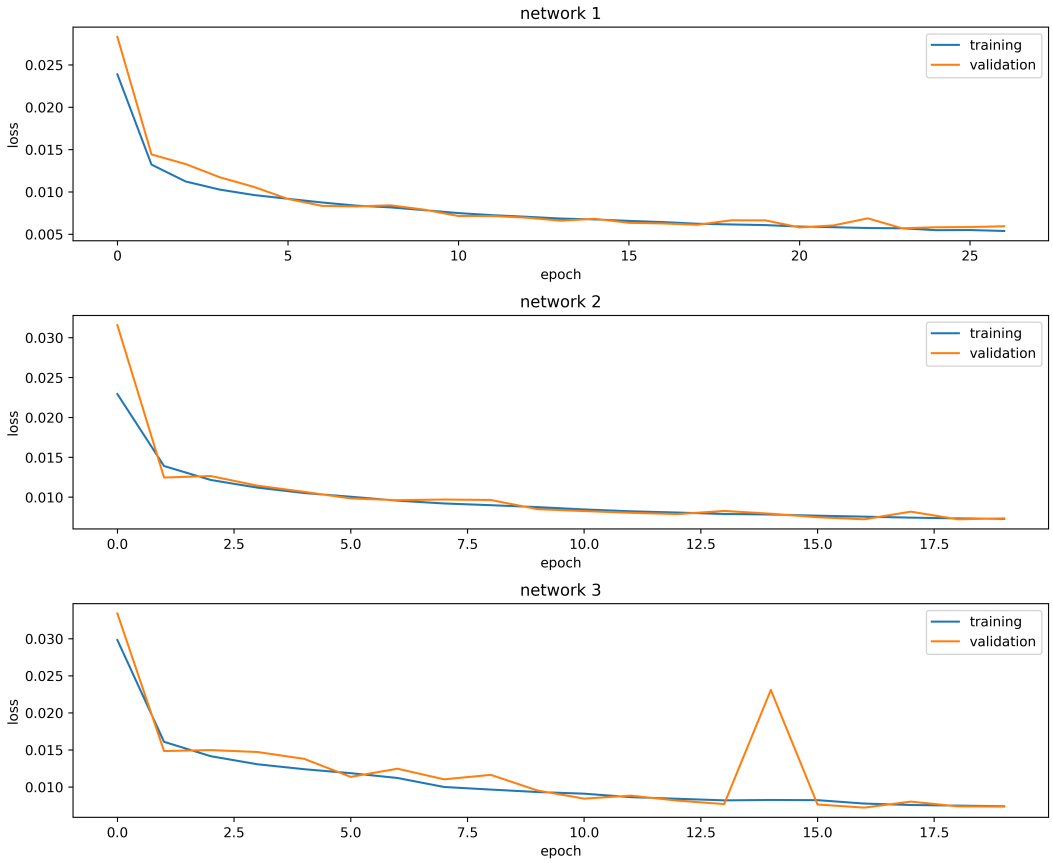

Figure 2: Training and validation loss of networks with different decoders

Table 1 shows a summary of the results for networks 1-3. Based on these results, we chose the architecture of network 1 for further experiments as it seemed significantly better than the others.

Table 1: Validation results for networks 1-3

	trainable parameters	epoch	training loss	validation loss
network 1	50,704,803	24	0.0057	0.0057
network 2	50,704,803	17	0.0076	0.0072
network 3	50,680,971	17	0.0078	0.0072

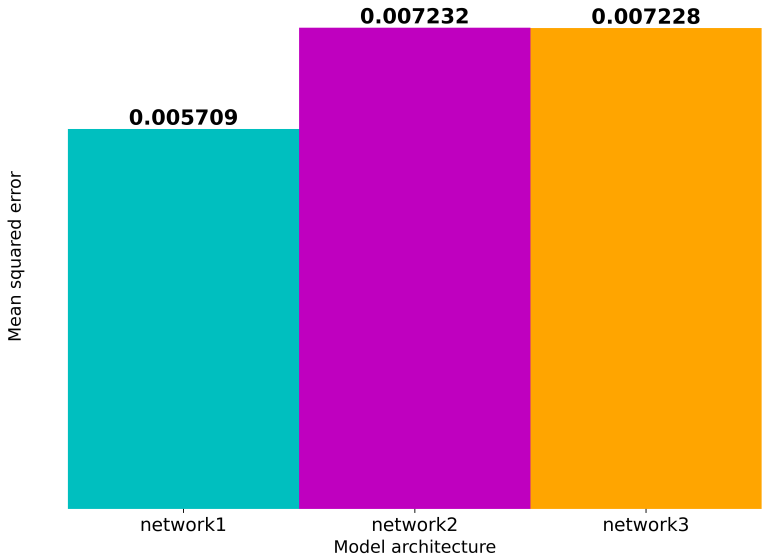

Figure 3: Validation reconstruction errors by networks with different decoders

3.1.1 Latent Bottleneck Test

As a second set of experiments, we wanted to identify a point of diminishing returns in terms of reconstruction performance when modifying the latent dimension size. Not only this would be the baseline to compare against when adding image perturbations, but also it is a way of understanding the amount of information, that can be lost before degrading the image too much. Therefore, we kept the autoencoder structure of network 1, modified just the number of units in the bottleneck dense layer and reconstructed the original images without perturbations. We got the trend captured in figure 4.

From the figure, we can see that the gain in terms of MSE is limited after we pass the 1000 neurons mark. Therefore, we picked this size with a validation MSE of 0.0058 as a reference for the final model. It is worth mentioning that this is a very good compression rate for the 64x64 RGB image, with 12288 pixel values, now down to only 1000.

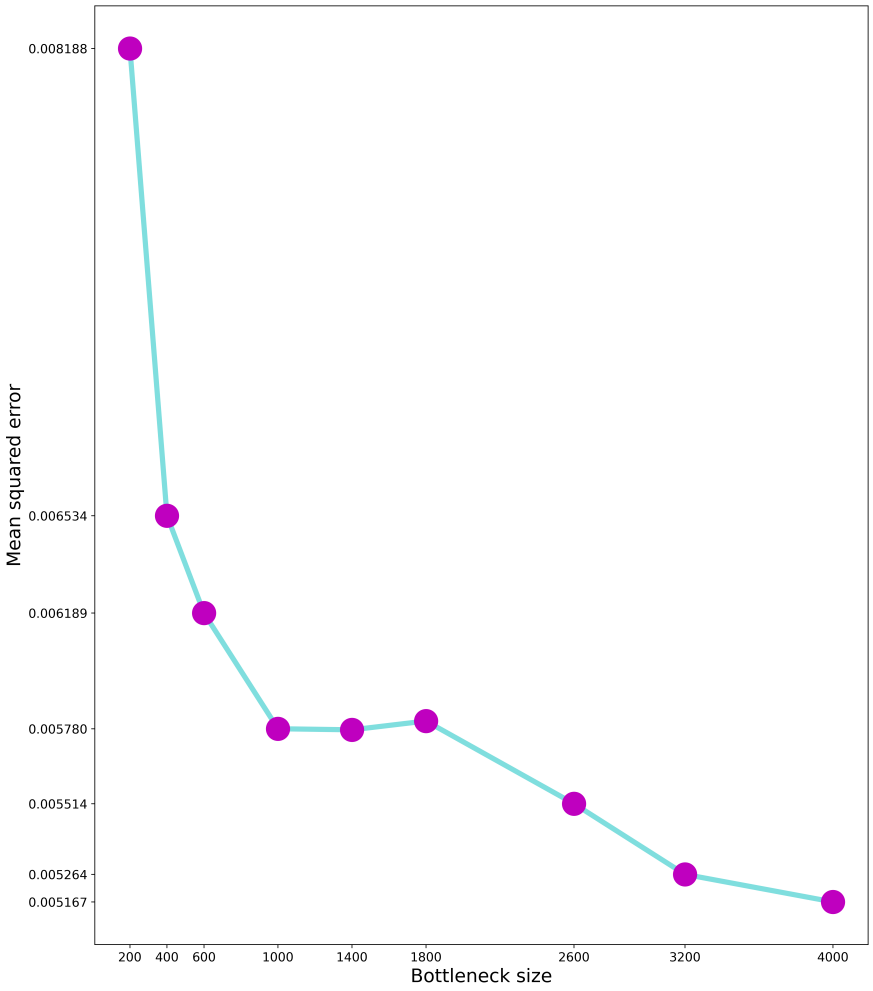

Figure 4: Validation reconstruction error vs latent bottleneck size

3.1.2 Final Model Architecture

Our final model architecture, which we finally fed with perturbed images as described in subsection 3.1.3, consists of a network 1 type model, chosen in experiments 3.1, with a bottleneck latent dimension of 1000, proven to be a point of diminishing returns in 3.1.1. The final model architecture can be analyzed in detail by looking at figure 5. We trained the model using the configuration described in 2.2 for a maximum of 50 epochs with early stopping, setting the patience to 3 epochs.

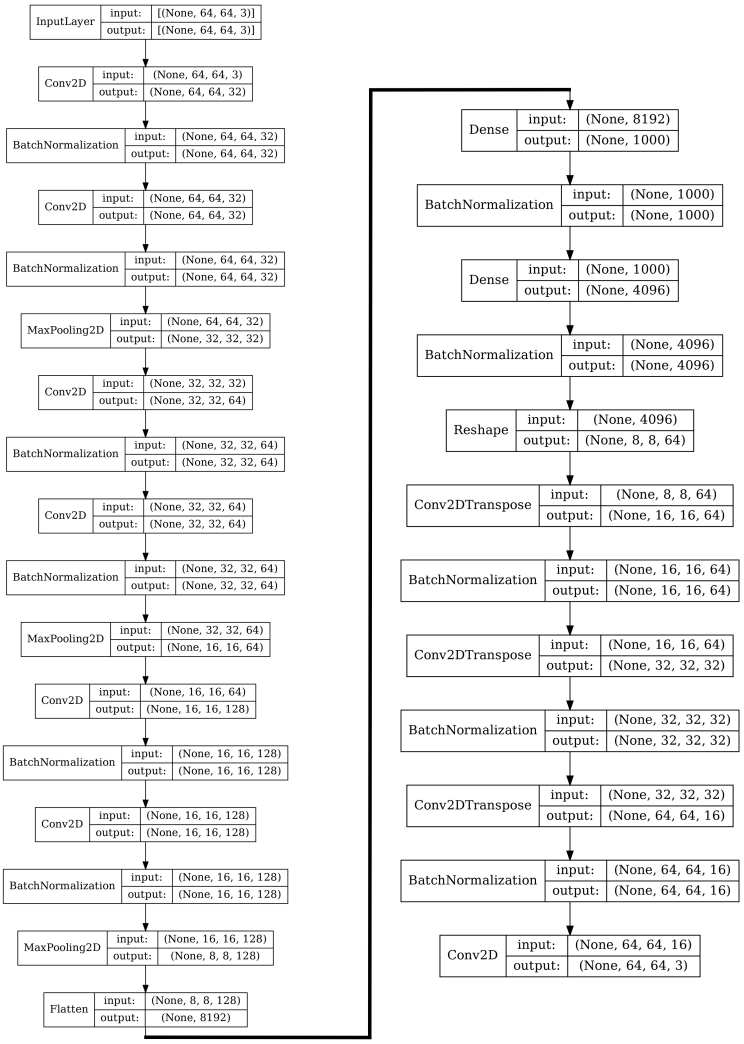

Figure 5: Final autoencoder with latent dimension 1000.

3.1.3 Image Perturbations

We included a selection of distortions to apply to the images from a list consisting of:

- Additive Gaussian noise
- Random cutout/occlusion of the image with three 12x12 black squares
- Channel-wise salt and pepper noise

Multiple distortions at once were also applied with success during the experiments, increasing the generalization of the model in handling various sources of errors. Results can be seen in table 2.

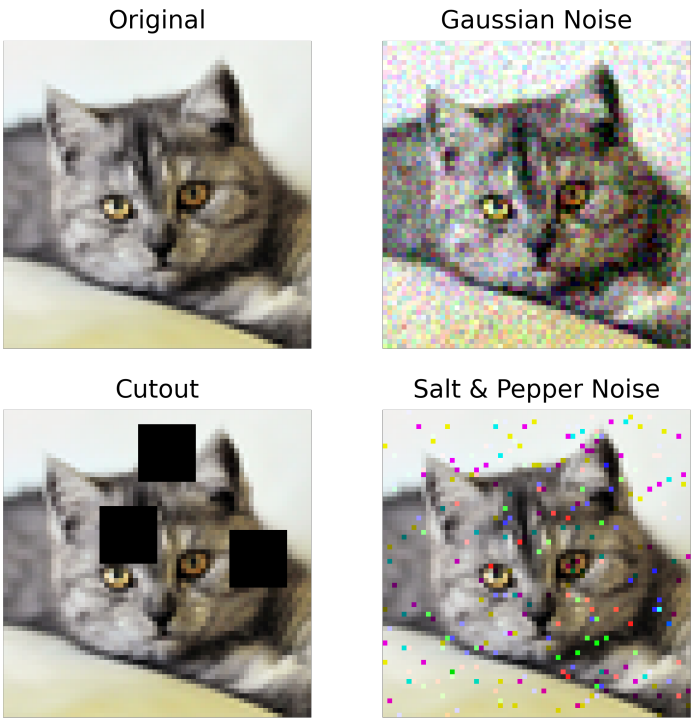

Figure 6: Perturbations applied over an example image.

We did not consider rotations, brightness shifts and horizontal flips of the input image as the resulting images would be totally realistic and therefore it would be very difficult for the model to correctly handle such a non-consistent perturbation if not applied to the whole data set but mixed with others.

3.2 Model Performance

We decided to apply the aforementioned distortions in our final network first one by one on the data set. Then we combined two out of these three distortions each and subsequently applied them to the images. In the end, we finally applied all the three distortions combined.

Table 2 represents the resulting test errors of the data sets with different distortions and their combinations and figure 7 depicts the reconstruction errors visually for the multiple distortion case.

Distortion	**Test error**
Coarse	0.00765
Gaussian	0.00673
Salt and Pepper	0.00676
Coarse + Gaussian	0.00738
Coarse + Salt and Pepper	0.00729
Gaussian + Salt and Pepper	0.00626
Coarse + Gaussian + Salt and Pepper	0.00668

Table 2: Test errors on differently perturbed data sets.

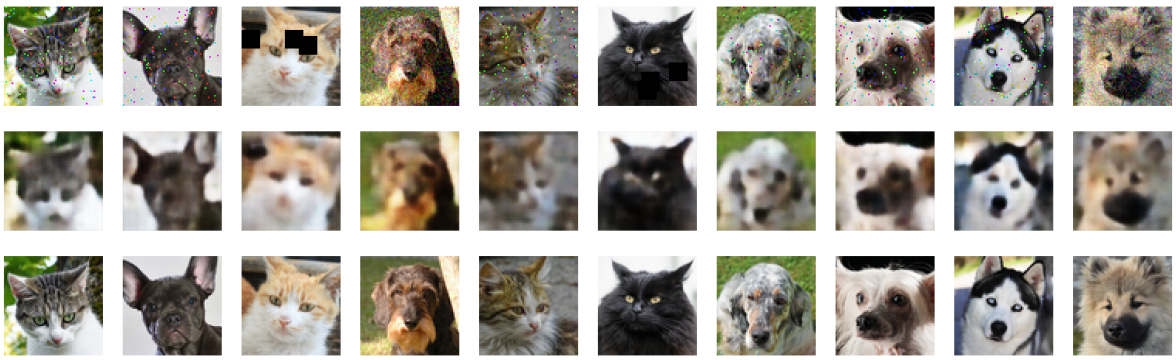

Figure 7: Reconstruction of the network 1 on test images (distorted, reconstructed, original)

As can be seen from the images, the network is able to identify the noise and act to remove it, however, the results are very blurry.

4 Discussion

Although our model is able to detect and correct distortions, in fact, we believe the dense layer version is too disruptive for a data set like animal faces, producing very blurry results in the end, probably due to also mixing global information of the whole image at the dense bottleneck level. Considering this strong compression, we feel like there's not much of an improvement we can do with these settings. For this reason, we additionally tried to improve quality with switching to a fully convolutional autoencoder (without any elaborate testing but more so as a proof of concept).

Not only a fully convolutional autoencoder keeps an image-like structure throughout the whole process, facilitating reconstruction, but also does not permit fusing information from completely separate image regions (say top left corner with the bottom right one) by design. Therefore, we designed a network with the following encoder-decoder structure, basically removing the dense and reshaping layers from our previous architecture:

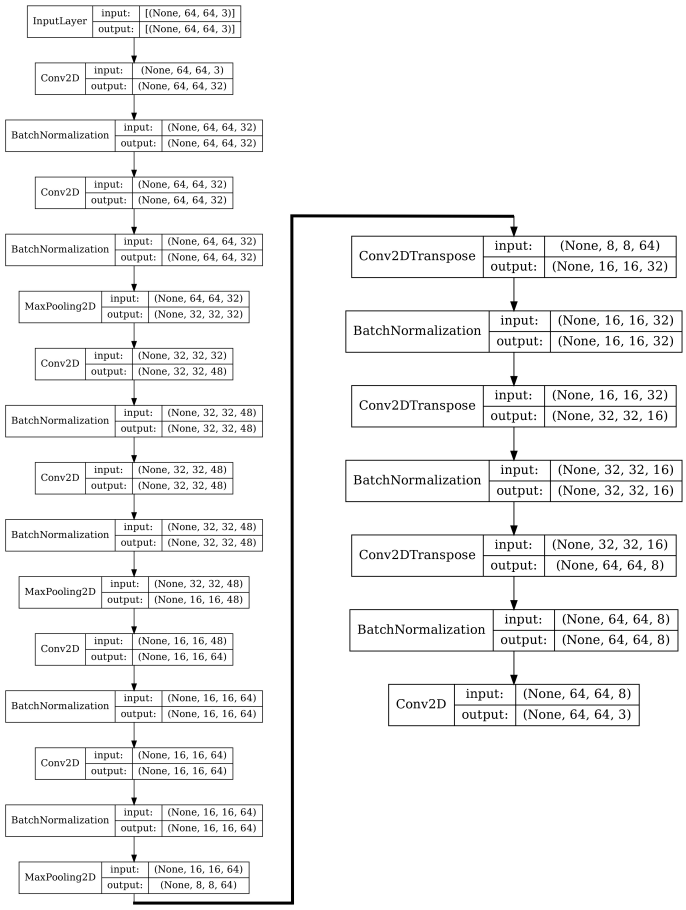

Figure 8: Fully convolutional autoencoder.

Note how the decoder was made lighter with 32, 16, and 8 filters respectively. Also, since having a latent representation of 8x8x128=8192 would not be too much of a compression

from the 64x64x3=12288 input pixels, the number of filters in the encoder was reduced to 32, 48, 64, thus leading to a latent representation of 4096 values.

By also substituting the last convolution in the decoder producing pixel values with a 1x1 convolution from the original 3x3, introducing exponential learning rate scheduling and increasing the patience in the early stopping, this network is able to achieve a validation error of 0.0037 over 50 epochs on images with multiple perturbations, greatly improving from the dense version. Visual results also confirm an overall better quality, already hinted by the lower MSE.

Figure 9: Reconstruction of the fully convolutional autoencoder on test images (distorted, reconstructed, original)

In conclusion, by switching settings, we were able to achieve a better-quality output. For a complex data set like animal faces, these images certainly look really good and we are very satisfied with the results.

References

[1] Alexander Buslaev et al. "Albumentations: Fast and Flexible Image Augmentations". In: *Information* 11.2 (2020). ISSN: 2078-2489. DOI: 10.3390/info11020125. URL: https://www.mdpi.com/2078-2489/11/2/125.

[2] Yunjey Choi et al. "StarGAN v2: Diverse Image Synthesis for Multiple Domains". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2020.