In the Name of Deep Learning

Final project report for Computer Vision 18-19

Wouter Durnez

June 12, 2019

Contents

1 Introduction **1**

2 Autoencoders **3**
2.1 PCA vs autoencoder 3
2.2 Nonlinear and convolutional 5

3 Classification **10**
3.1 Object classification network 10
3.2 Reflection 13

4 Segmentation **13**
4.1 Binary segmentation network 13
4.2 Reflection 17

5 Discussion **17**

1 Introduction

This report is submitted as part of the examination for the course *Computer Vision*. It describes various types of 'home-made' neural networks, and how their respective network types can be used in the computer vision domain. In line with the spirit of the domain, and under the adage *'a picture says more than a thousand words'*, the report is accompanied by a substantial number of visualizations and graphs. The structure of the report follows the structure of the original assignment.

The project code can be found here: https://github.com/WouterDurnez/003_FinalProject. It is structured as follows:

```
data
├── VOCdevkit
└── image_dim_xx
models
├── autoencoders
│   ├── architecture
│   ├── plots
│   └── history
├── classifiers
│   ├── architecture
│   ├── plots
│   └── history
└── segmentation
    ├── architecture
    ├── plots
    └── history
report
└── images
scripts
├── __init__.py
├── itnodl_help.py
├── itnodl_data.py
├── itnodl_auto.py
├── itnodl_class.py
├── itnodl_segm.py
└── itnodl_tsne.py
```

The code expects the *data*, *models* and *scripts* folders to be present. Furthermore, the scripts should be downloaded in the corresponding folder, as well as the VOCdevkit data. All other folders, models and data are generated automat-

ically when executing the code. Datasets for image dimension *xx* (e.g. 32) are stored in corresponding folders.

Datasets The datasets used in this project were extracted from the PASCAL VOC 2009 Challenge. To minimize the training load, a subset of the full dataset was taken for each task. Filtering was achieved using the following class list: ['aeroplane', 'car', 'chair', 'dog', 'bird']. Only images that were labeled as belonging to one (or more) of the filter classes were withheld. The datasets are described in Table 1 and 2.

The organisation of the dataset in training and validation images was respected. Since it is possible to impart a bias onto a model by having its training process guided too strongly by a validation set, it was import to retain a set of 'fresh' images to use in a final check of each model's quality metrics. Therefore, half of the original 'validation images' were randomly sampled to be test images (denoted in table 1 as val^*), whereas the remaining images in the validation set were kept as validation data (denoted in the table by val). The latter category of images was used to monitor model metrics during training. Test images, on the other hand, were never seen or used during model training. Importantly, a seed was set for the random sampling to always yield the same results.

The dataset for the final task—**segmentation**—is slightly different from the others. Again, the same five classes were used to filter the full dataset to a smaller one. This time, however, only those images that were accompanied by a segmented counterpart could be used. This significantly reduced the number of available images in each of the (sub)sets. I attempted to remedy this problem in part by using data augmentation.

Section		**Autoencoders**	**Classification**	**Segmentation**
Training	Files	$train$	$train$	$train$
	N	1489	1489	286
Validation	Files	val^*	val^*	val^*
	N	735	735	67
Test	Files	val	val	val
	N	735	735	67

Table 1: Dataset definition per section. Selected from main data corpus using class filter ['aeroplane', 'car', 'chair', 'dog', 'bird'].

Class	**Aeroplane**	**Bird**	**Car**	**Chair**	**Dog**
Training	210	257	381	347	294
Validation	118	128	171	150	168
Test	120	129	195	149	152

Table 2: Number of images per class, for each of the classification datasets.

Image dimensions and compression factors Most of the neural networks (and their architectures) described in this report are largely derived from the deep convolutional autoencoder (DCA), outlined in section 2. I varied multiple parameters to observe the impact on the result, its accuracy, and the speed with which it was obtained. To save time, I heavily downsampled the original images to feed to the neural nets (down to $16 \times 16 \times 3$ pixels). Once the algorithms worked, I experimented with higher resolution images (up to $128 \times 128 \times 3$ pixels). However, for the sake of consistency, *all models discussed in this report take input images with dimensionality* $96 \times 96 \times 3$*, except when explicitly mentioned otherwise.*

Platform All models were created using Keras, a deep learning library developed for the Python programming language. For model training, I restricted myself to the loss functions and optimizers that are implemented in this package.

2 Autoencoders

2.1 PCA vs autoencoder

Principal Component Analysis (or PCA) is a statistical technique that is used to transform a set of (potentially) correlated data vectors into a (smaller) number of linearly uncorrelated variables—the Principal Components (PCs). They are sorted in such a way that the first PC accounts for the largest amount of variability, and the subsequent PCs cover a maximal portion of the remaining variance while remaining orthogonal with the preceding PCs.

The principal component decomposition of a data matrix $\mathbf{X}$ is described by

$$\mathbf{T} = \mathbf{XW} \tag{1}$$

where $\mathbf{W}$ is a $p \times p$ matrix of weights whose columns are the eigenvectors of $\mathbf{X}^\top\mathbf{X}$.

Not all PCs need to be retained for PCA to be a useful tool. More specifically, PCA is often used as a means of dimensionality reduction, by projecting the data

vectors onto a subset of all PCs—more specifically, the K first PCs. Equation 1 becomes:

$$\mathbf{T_L} = \mathbf{X}\mathbf{W_L} \tag{2}$$

The goal of PCA can then be described as finding a projection so that the best linear reconstruction of the data is as close as possible to the original data. Keeping in mind that the inverse of an orthogonal matrix equals its transpose, this translates to

$$\mathbf{L_{PCA}} = \|\mathbf{X} - \mathbf{T_L}\mathbf{W_L}^{\mathsf{T}}\|_2^2 \tag{3}$$

This statistical method is closely related to a specific type of neural network—the **linear autoencoder (LA)** (1). This network takes a data vector as input, passes it through a single hidden layer (typically smaller in dimensionality than the input), and attempts to reconstruct the data in the output layer using linear activation functions. As such, the network is forced to learn an optimal strategy in order to minimize the reconstruction loss.

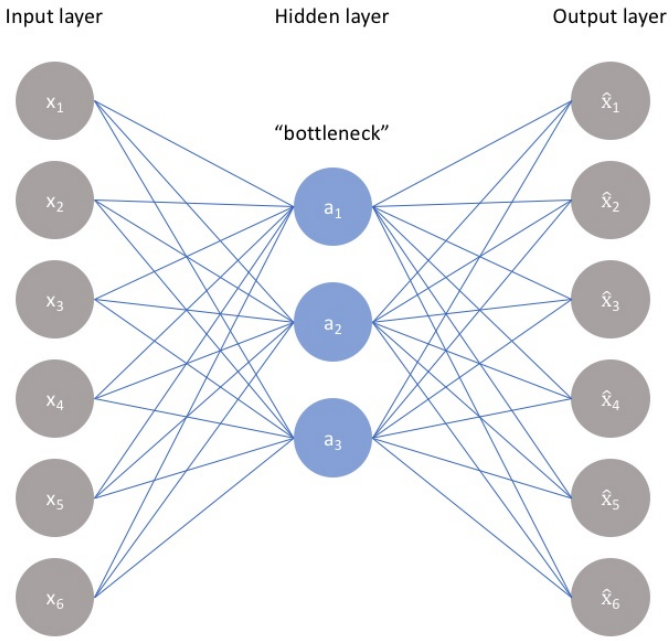

Figure 1: Linear autoencoder architecture with input dimension 6, encoding dimension 3, and compression factor 2 (taken from https://www.jeremyjordan.me/autoencoders/).

From intuition, we can assume that this optimal strategy involves two things: 1) each node in the encoded (hidden) layer must cover a maximal amount of

variability, and 2) the contribution of the nodes to the output must overlap as little as possible. It is already apparent that this mission is heavily related to the workings of PCA.

Suppose we built a linear autoencoder for the same data matrix $\mathbf{X}$. This data is passed on to the hidden layer, when then passes the transferred data on to the output layer. Defining two successive activation functions f and g, this process can be described by the following equations.

$$\mathbf{Z} = f(\mathbf{W_1 X}) \tag{4}$$

$$\hat{\mathbf{X}} = g(\mathbf{W_2 Z}) \tag{5}$$

By definition, training this network means attempting to minimize the reconstruction loss. Given the linearity of the activation functions f and g, this is described by:

$$\mathbf{L_{LA}} = \|\mathbf{X} - \mathbf{W_2 W_1 X}\|_2^2 \tag{6}$$

We can see that optimizing the weight matrix for this linear autoencoder for all intents and purposes equates to the solution for PCA (3).

A difference between both approaches can be found in the boundaries of the dimensionality. In linear autoencoders, the hidden layer can be made up of more nodes than the input and output layers, meaning that the data is transformed to a feature space greater than the original input space.

2.2 Nonlinear and convolutional

In this section, two distinct autoencoders are described. A first autoencoder has a linear architecture—it thus consists of an input layer, output layer, and a single hidden layer. A second autoencoder is created by training a deep convolutional neural network. The architecture for both autoencoders is shown in figures 2 and 3.

Architectures The *linear autoencoder* consisted of an input layer, output layer, and a hidden layer. Its transfer functions were, of course, both linear. The *deep convolutional autoencoder*, in turn, was made up of 5 convolutional layers, each of them using a 3×3-sized kernel. The kernel values were initialized in a random uniform manner, and bias was initialized to zero. After each convolutional layer, batch normalization was applied to accelerate and stabilize the training process. The data was downsampled after each convolutional layer preceding the encoded layer (using max pooling), and then upsampled back to attain the original image dimensions. All transfers were modeled by *ReLU* (Rectified Linear Unit), except for the final activation—there, a sigmoid transfer function was used. The main advantages of using *ReLU* in the intermediate layers are:

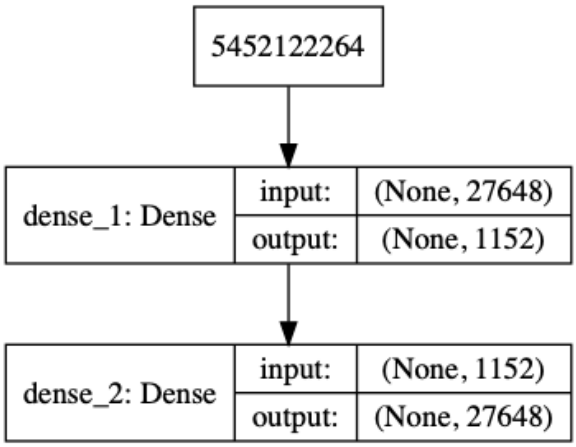

Figure 2: Linear architecture for autoencoder 1. Input and output dimensions $96 \times 96 \times 3 = 27648$, compression factor 24, encoding dimension 1152.

- *ReLU* is a mathematically simple, and thus computationally cheap transfer function. This implies less strain on the CPU during training.
- *ReLU* will likely converge faster during training, since the function slope doesn't 'plateau', as can be the case when using *sigmoid* or *tanh* transfer functions.
- Since *ReLU* is defined as zero for negative input values, it is a sparsely activated transfer function—it is likely that not all input causes activation, which translates to a lower computational demand.

The final layer uses a sigmoid activation function, since we require output in the $[0,1]$ range.

Network parameters A number of parameters were equal in both autoencoder networks. The models were each designed to accept $96 \times 96 \times 3$-sized images as input, and were trained to reconstruct the images in those same dimensions. The 'bottleneck' layer in the middle of the networks—capturing the encoded representation—consisted of 1152 nodes, corresponding with a compression factor of 24. The encoding dimension was chosen after some experimentation: I ran the model with different values, and observed the evolution of the loss function as well as the eventual reconstructed images. Both models were set to train for *200 epochs*. If no improvement was detected in the validation loss for 20 epochs, training *stopped early*. This was implemented to avoid overfitting. In training, I used an *Adam* optimizer algorithm (default parameters), and used *mean squared error* to compute loss.

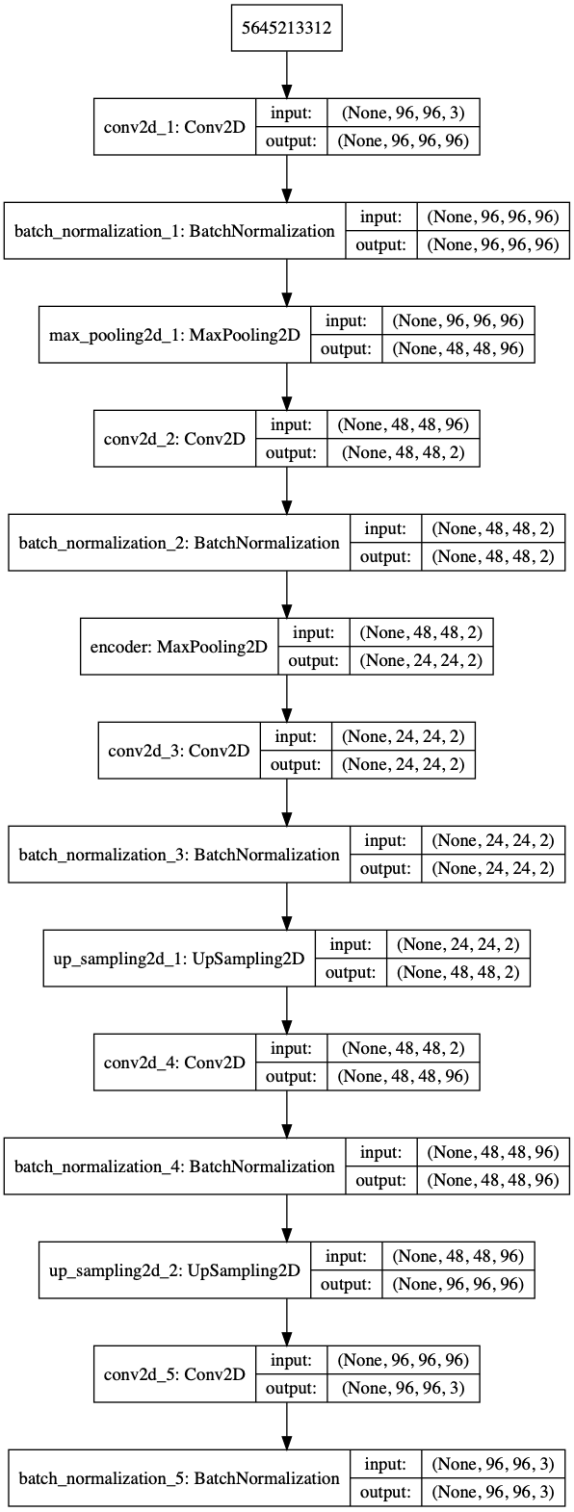

Figure 3: Deep convolutional architecture for autoencoder 2. Same input, output, and encoding dimensions as autoencoder 1.

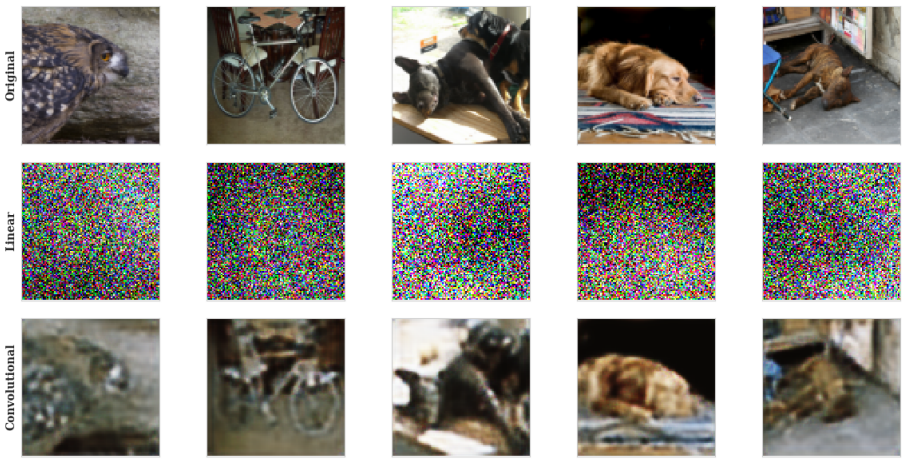

Figure 4: Image reconstructions yielded by both autoencoders for a random selection of 5 'fresh' test images ($96 \times 96 \times 3$ pixels). The top row shows the original images. The middle and bottom row show the reconstructions made by the linear autoencoder and deep convolutional autoencoder, respectively.

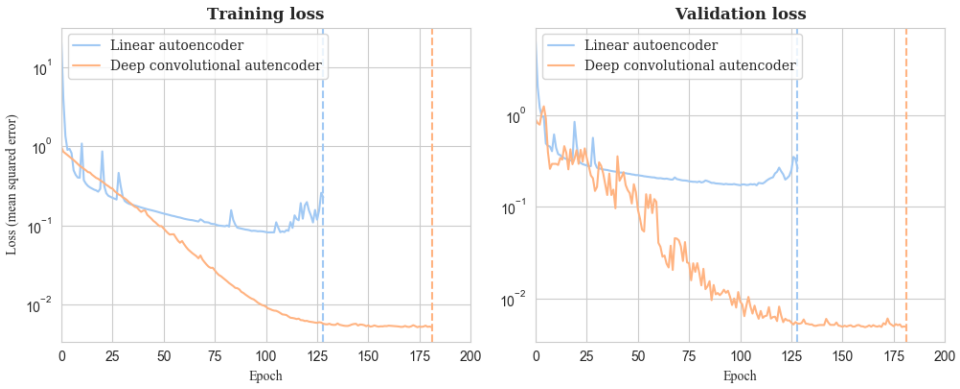

Figure 5: Training history for both autoencoders. The left graph shows the evolution of the loss (mean squared error) on the training data, whereas the right graph shows the evolution of the validation loss. The vertical dotted line indicates when early stopping occurred, due to a lack of improvement in the latter value. Loss is expressed on a logarithmic scale.

Evaluation Both autoencoders only differed in terms of model architecture—all training parameters, and the size of the encoded layer, were equal. The results, however, are quite different. Visual inspection of each model's predictions shows that the deep convolutional neural network (DCNN) is distinctly better at preserving the characterizing features of the original image (see figure 4), or reconstructing anything that resembles the input image for that matter. Figure 5 displays the training history of both models. Several observations can be made:

Model	**Train**	**Validation**	**Test**
Linear autoencoder	1.77×10^{-1}	2.62×10^{-1}	2.73×10^{-1}
Deep convolutional autencoder	5.12×10^{-3}	4.92×10^{-3}	5.14×10^{-3}

Table 3: Evaluation of models based on reconstruction loss: *mean squared error* values for training, validation and test datasets.

- The deep convolutional autoencoder took a longer time to converge to an optimal solution (under the early stopping criteria) compared to the linear autoencoder. This is unsurprising, given the different number of training parameters—there are a lot more options for the DCA to improve its reconstruction strategy.
- The final DCA model was characterized by a significant lesser amount of loss, compared to the LA model (see table 3). This quality difference is also apparent from the visual reconstruction of the test images (figure 4).
- The evaluation table (table 3) shows that, for the LA, the reconstruction loss is higher for the evaluation and test images, compared to the training image reconstruction loss. The DCA, on the other hand, shows a fairly constant loss over all datasets, seen and unseen. This implies that the DCA was better at capturing the underlying, example-agnostic features, whereas the LA's learnt strategy was more limited to the images it was presented with during training.

Latent space visualization There are several options to visualize the space of the encoded representations. One option is to take the decoding part of the network—that is, the encoded layer serves as input layer, and constructs an image based on the latent representation it receives. Presenting this decoder subnetwork with a 'one-hot encoded' input vector would then generate an 'eigenface'. This is feasible when the latent space is limited in dimensionality, otherwise we need a way to identify the most important nodes (if that is at all possible). In case of LA, this translates to finding the eigenvalues of the weight matrix. In case of DCA, such an approach is not so straightforward.

3 Classification

3.1 Object classification network

In this section, three different classification networks were built and trained, based on the deep convolutional autoencoder model learnt in section 2.2. Specifically, I retained the encoding part of the DCA (from input to encoded layer), and appended layers to allow for classification training. Three approaches were taken with regard to the model parameters:

1. Encoder layers are ***frozen***.
2. Encoder layers are ***trainable***, and ***encoder weights are kept from DCA***.
3. Encoder layers are ***trainable***, and all parameters ***train from scratch***.

The goal in comparing these approaches is 1) to explore what the value of the encoded representation is as a basis for classification, and 2) relatedly, to see whether the previously learnt encoder weights serve as a better or worse starting point for classification training.

Observation of the class labels shows that the source data represents a **multi-label** problem—that is, multiple classes can be present in a single image.

Architectures The common architecture of the three approaches, apart from the trainability and weight initialization of the parameters, is shown in figure 6. Several layers are added at the end of the encoder part to allow for classification training. Specifically, the encoded representation is flattened, and passes through two more fully connected layers (Dense layers in Keras). The first of these layers used ReLU, whereas the final layer—containing the classification results—has a sigmoid activation function. Between both layers, dropout is applied. In other words, some of the input to the final layer is dropped out randomly, in an attempt to make the network more robust. The output layer consisted of 5 nodes, to accommodate the K-hot encoded label vector.

Network parameters Similar to section 2, all networks were trained using the *Adam* optimizer algorithm. Several loss functions were experimented with. Results varied, but the comparison between the three presented networks remained fairly consistent. In the remainder of this section, I will discuss results for networks trained using the *binary crossentropy* loss function, as this is the go-to loss function for multi-label classification[1]. All model training was set for *300 epochs*. Similar to the approach in section 2, training stopped if no improvement was detected in the validation loss for a number epochs. After some experimentation, this number was set to 50.

[1] If the problem were multi-class, but not multi-label, the preferred loss function would have been *categorical crossentropy*, combined with a *softmax* activation function in the final layer.

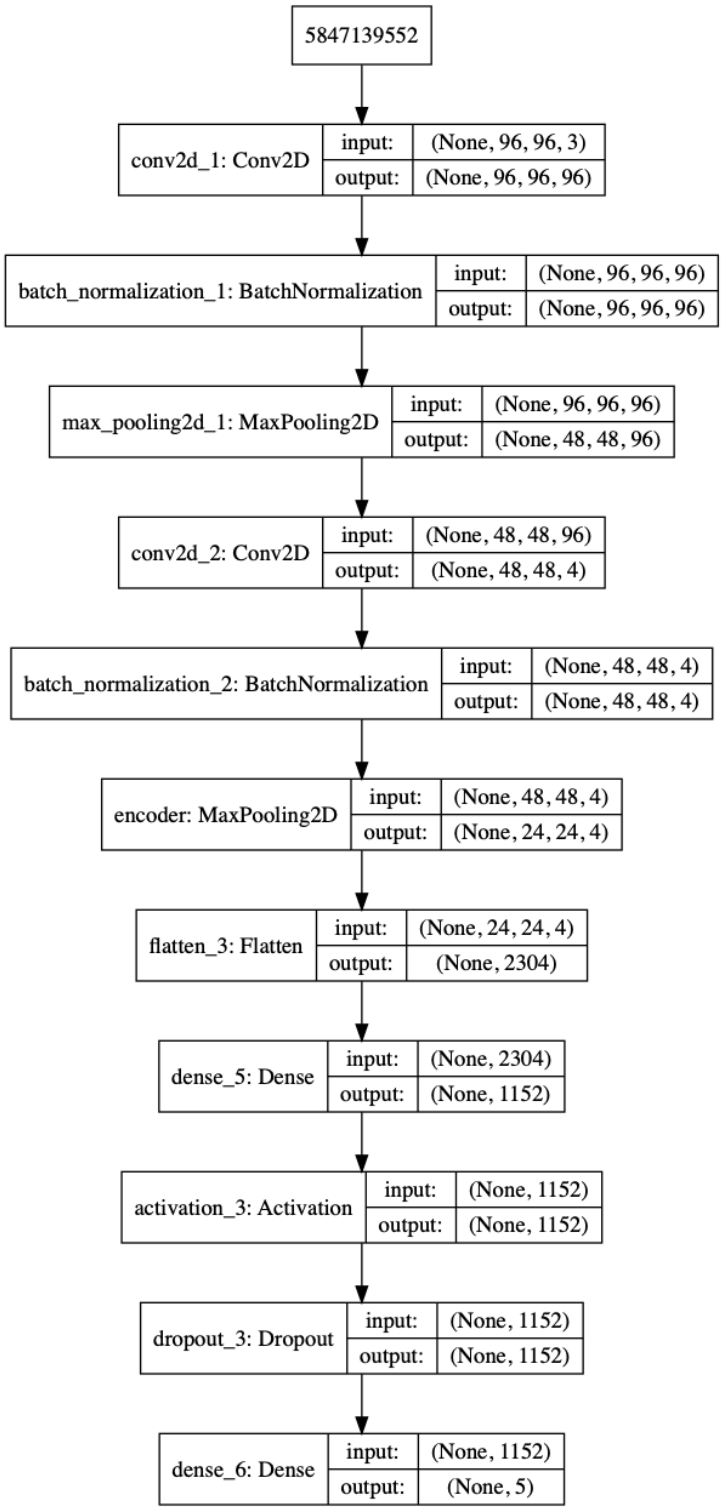

Figure 6: Architecture for classification network. Input and output dimensions $64 \times 64 \times 3 = 12288$, compression factor 24, encoding dimension 512.

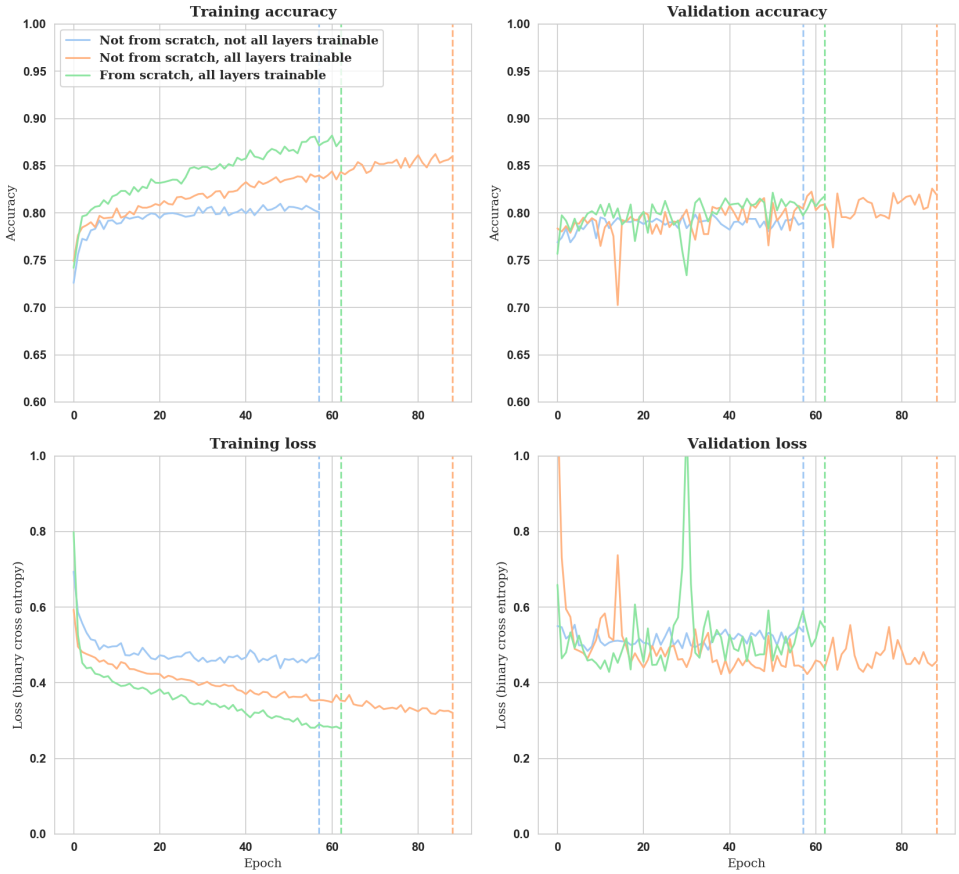

Figure 7: Training history for all classifier models. In the top row, the left graph shows the evolution of the accuracy on the training data, whereas the right graph shows the evolution of the validation accuracy. In the bottom row, the graphs represent the evolution of the training and validation loss (binary crossentropy). The vertical dotted line indicates when early stopping occurred, due to a lack of improvement in the latter value.

Evaluation All models stopped early re. The bottom left graph in figure 7 shows how training loss decreases steadily for all networks. Unsurprisingly, loss values drop more rapidly in the models with all trainable layers, as these models are more flexible compared to the partially frozen model. The graphs also make clear that training loss would likely have kept decreasing for a significant number of additional epochs. However, judging from the bottom right graph—the validation loss evolution—no real useful information is learnt that applies beyond the training data.

3.2 Reflection

The models presented in this section clearly have limited to no use beyond the training data. There are a several possible explanations for this failure.

- First, it is entirely possible that the model architecture was simply not adequate to capture the relevant information needed to classify the images correctly. If it in fact did succeed in capturing relevant information, then the data compression may have caused the loss of too much necessary information.
- Given the limited number of training images per class (see table 2), it is possible that the model did not see enough examples to be able to extract recurring semantic features. The problem should therefore probably best be simplified into a binary classification task, to reduce its complexity. Alternatively, additional images could be sought and used as training data (or stolen from the validation set).

4 Segmentation

4.1 Binary segmentation network

This section discusses a deep convolutional binary segmentation network, based largely on the autoencoder described in section 2.2. To train the model, a subset of the full dataset was used (see 1 for details). In accordance with the assignment, training labels were binarized. This was accomplished by converting the segmented images to grayscale and applying a threshold of .02 to the pixel values. This made sure that only the black pixels remained 0, and all the others were set to 1. Throughout this section, I'll refer to the 1-values as 'foreground', and the 0-values as 'background'.

Architecture The architecture of the segmentation network is shown in 8. It is very closely related to the DCA architecture (see figure 3, except for a few minor changes. More explicitly, the number of filters in the convolutional layers was increased in an effort to capture more information. This decision was made after some experimentation—the impact is likely to be on the smaller end. The

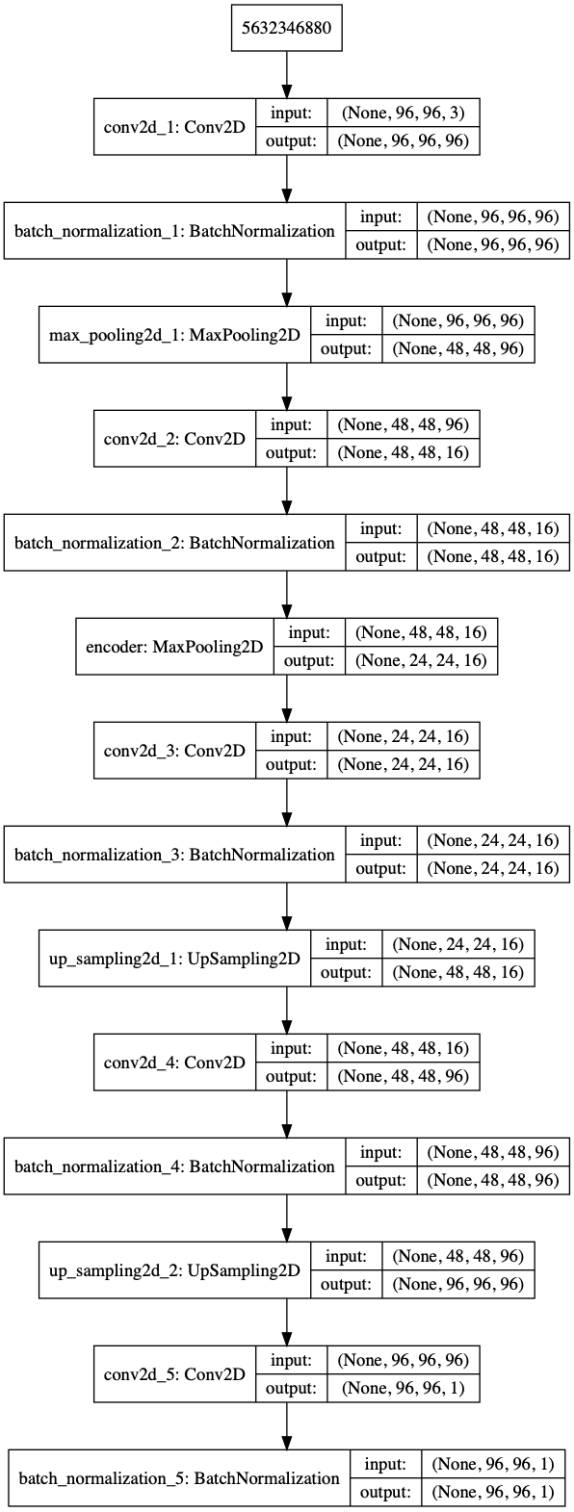

Figure 8: Deep convolutional architecture for the segmentation network. Input dimensions are $96 \times 96 \times 3$. The model predicts a score per pixel position, ignoring color channels—hence the output dimensions are $96 \times 96 \times 1$.

model gives fair results, however, hence this architecture was kept. The kernel values were initialized in a random uniform manner, and bias was initialized to zero. Again, all activation functions were *ReLU*, save the final activation, where a *sigmoid* was used to obtain values in the $[0, 1]$ range.

Network parameters As was the case in sections 2 and 3, training was done using the *Adam* optimizer algorithm. The loss function was *mean squared error*, as was the case with the autoencoder models. Due to the larger amount of trainable parameters, the number of training epochs was set to 500—a fairly high number. The early stopping criterion was set to 50 epochs without validation loss improvement. This criterion was purposely set to a very lenient value, after observing several training attempts and noticing a significant variability in the validation loss progression.

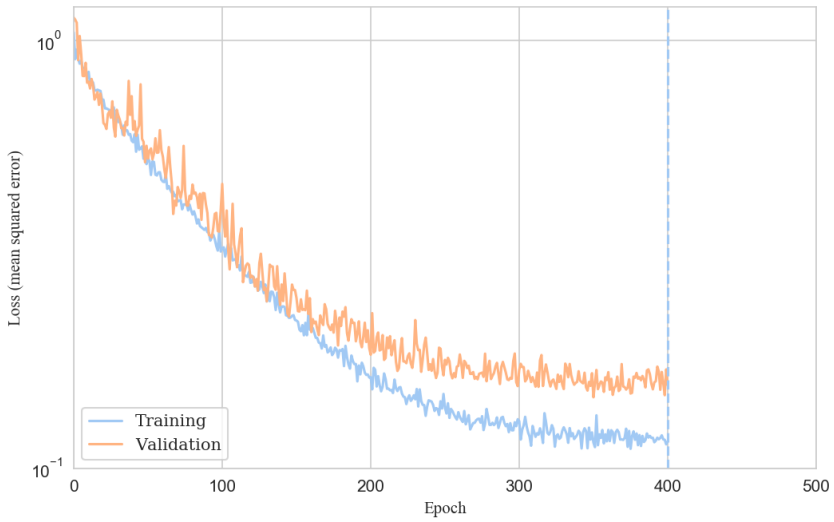

Figure 9: Training history for the segmentation network. The y-axis represents the loss value (mean squared error). The vertical dotted line indicates when early stopping occurred, due to a lack of improvement in the latter value. Loss is expressed on a logarithmic scale.

Evaluation The training process is visualized in figure 9. The graph suggests that some improvement was still possible, but overtraining would become a real risk. Figure 10 shows how the network segmented a random selection of (previously unseen) test images. The average loss for each of the datasets is shown in table 4. It shows how the model is better accustomed to the training data, but not overly biased towards the validation images.

Figure 10: Segmentation given by network for a random selection of 5 'fresh' test images ($96 \times 96 \times 3$ pixels). The top row shows the original images. The second row shows the binarized segmentation labels. The middle row shows the model output for the source images. The second to last row shows binarized predictions (threshold = 0.5). The bottom row, finally, uses the binarized prediction as a mask over the source image.

Model	**Train**	**Validation**	**Test**
Segmentation network	9.32×10^{-2}	1.50×10^{-1}	1.43×10^{-1}

Table 4: Evaluation of segmentation model based on reconstruction loss: *mean squared error* values for training, validation and test datasets.

4.2 Reflection

5 Discussion