In the Name of Deep Learning

Final project report for Computer Vision 18-19

Wouter Durnez

June 17, 2019

Contents

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 scientific literature, on the other hand, is not discussed in the report, as the focus of the assignment was not the theoretical or mathematical foundations of neural networks. The structure of the report by and large follows the structure of the original assignment.

The project code can be found here: `https://github.com/WouterDurnez/003_FinalProject`. Its structure is shown in figure 1.

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 automatically 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 dataset. 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. All data-related functions are bundled in `scripts/itnodl_data.py`.

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.

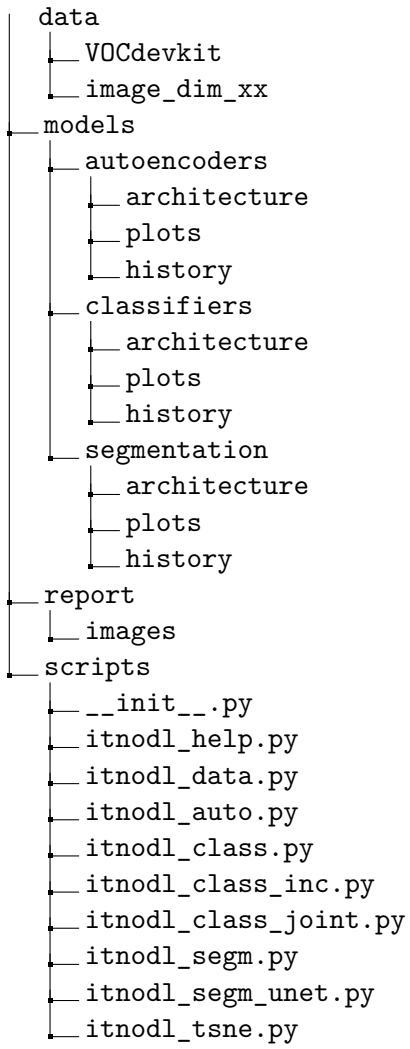

Figure 1: Project code structure.

General parameters Most of the neural networks (and their architectures) described in this report are largely derived from the deep convolutional autoencoder (DCA) described in section 2. I experimented a lot with different network and training 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 yielded output as expected, 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.* Finally, all model training used so-called *model checkpoints*, where the best model—i.e. the model yielding the

least amount of validation loss–was saved intermediately when possible.

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 each subsequent PC covers a maximal portion of the remaining variance while remaining orthogonal with the preceding PCs.

The principal component decomposition of a data matrix $\mathbf{X}$ (with n rows and p columns) 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:

$$\mathbf{T_L} = \mathbf{XW_L} \tag{2}$$

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

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

This statistical method is closely related to a specific type of neural network—the **linear autoencoder (LA)** (see figure 2). This neural 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.

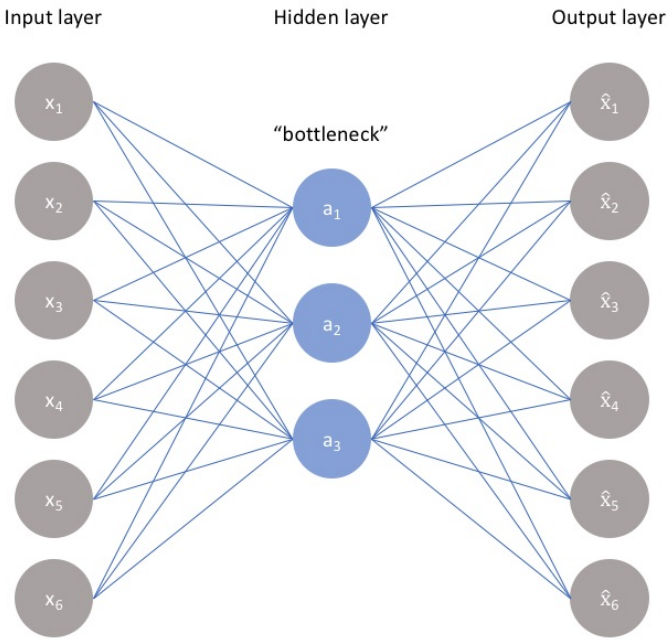

Figure 2: Linear autoencoder architecture with input dimension 6, encoding dimension 3, and compression factor 2 (taken from https://www.jeremyjordan.me/autoencoders/).

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. PCA, on the other hand, can never yield more PCs than the dimensionality of the input space.

2.2 Nonlinear and convolutional

In this section, two distinct autoencoders are described[1]. The 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 3 and 4.

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

[1] The code for this section can be found in `scripts/itnodl_auto.py`.

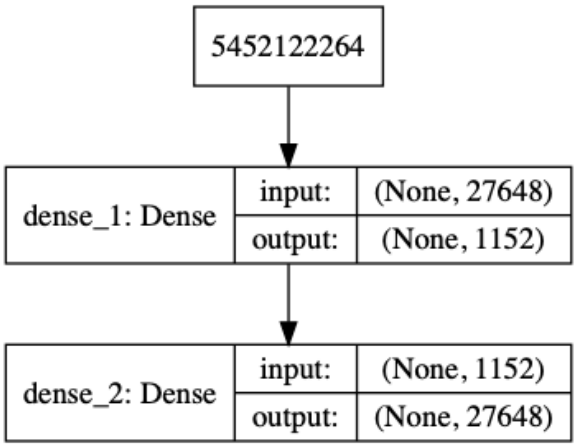

Figure 3: Linear architecture for autoencoder 1. Input and output dimensions $96 \times 96 \times 3 = 27648$, compression factor 24, encoding dimension 1152.

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:

- *ReLU* is a mathematically simple, and thus computationally cheap transfer function. This implies less strain on the CPU during training (a sorely needed benefit).
- *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

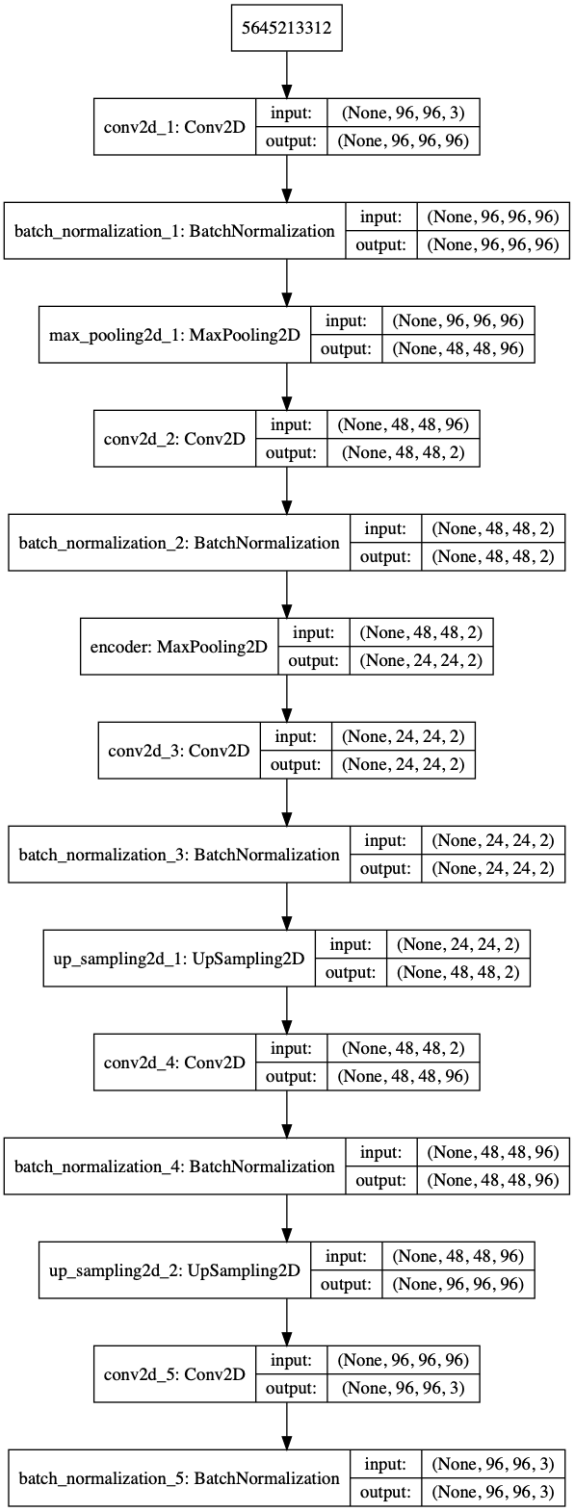

Figure 4: Deep convolutional architecture for autoencoder 2. Same input, output, and encoding dimensions as autoencoder 1.

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 chose *mean squared error* to compute reconstruction loss.

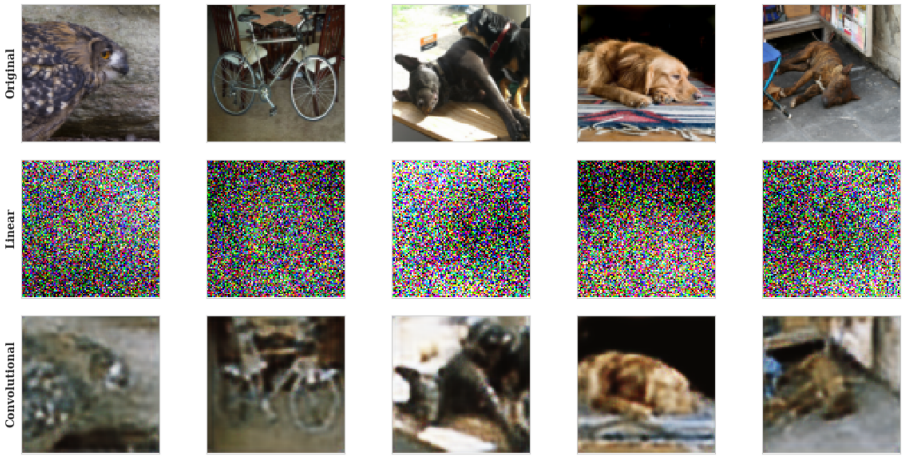

Figure 5: Image reconstructions yielded by both autoencoders for a random selection of 5 'fresh' test images ($96 \times 96 \times 3$ pixels). The top row shows the original images. The middle and bottom row show the reconstructions made by the linear autoencoder and deep convolutional autoencoder, respectively.

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 5), or reconstructing anything that resembles the input image for that matter. Figure 6 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.

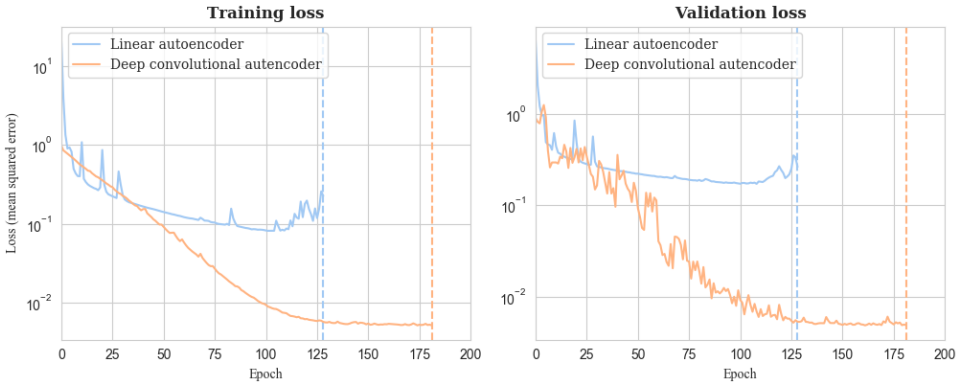

Figure 6: Training history for both autoencoders. The left graph shows the evolution of the loss (mean squared error) on the training data, whereas the right graph shows the evolution of the validation loss. The vertical dotted line indicates when early stopping occurred, due to a lack of improvement in the latter value. Loss is expressed on a logarithmic scale.

- 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 and fine-tune 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 5).
- 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 rank the importance of the encoded features (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 networks

In this section[2], 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 previously learnt encoded representation is as a basis for classification, and 2) relatedly, to see whether the pretrained 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. As such, predictions are considered as a combination of binary labels, rather than a single categorical label.

In addition to these three models, two extra models were built. One was ***based on the convolutional base layers of the Inception architecture, and pretrained on the ImageNet dataset***. These base layers were frozen—a similar approach as described with the first model discussed above—and additional fully connected layers were added to facilitate classification. In other words, this model benefits from a more complex architecture, and is fully pretrained on a larger dataset. (The odds are stacked heavily in its favor.)

A second extra model was a simple ***combination between the deep convolutional autoencoder and a classification network***. More specifically, this model reused the DCA architecture, but appended classification-purposed layers to the encoded layer. The model was then jointly trained on the two different task: image reconstruction and image classification.

[2]This section's supporting code can be found in `scripts/itnodl_class.py` and `scripts/itnodl_class_inc.py`.

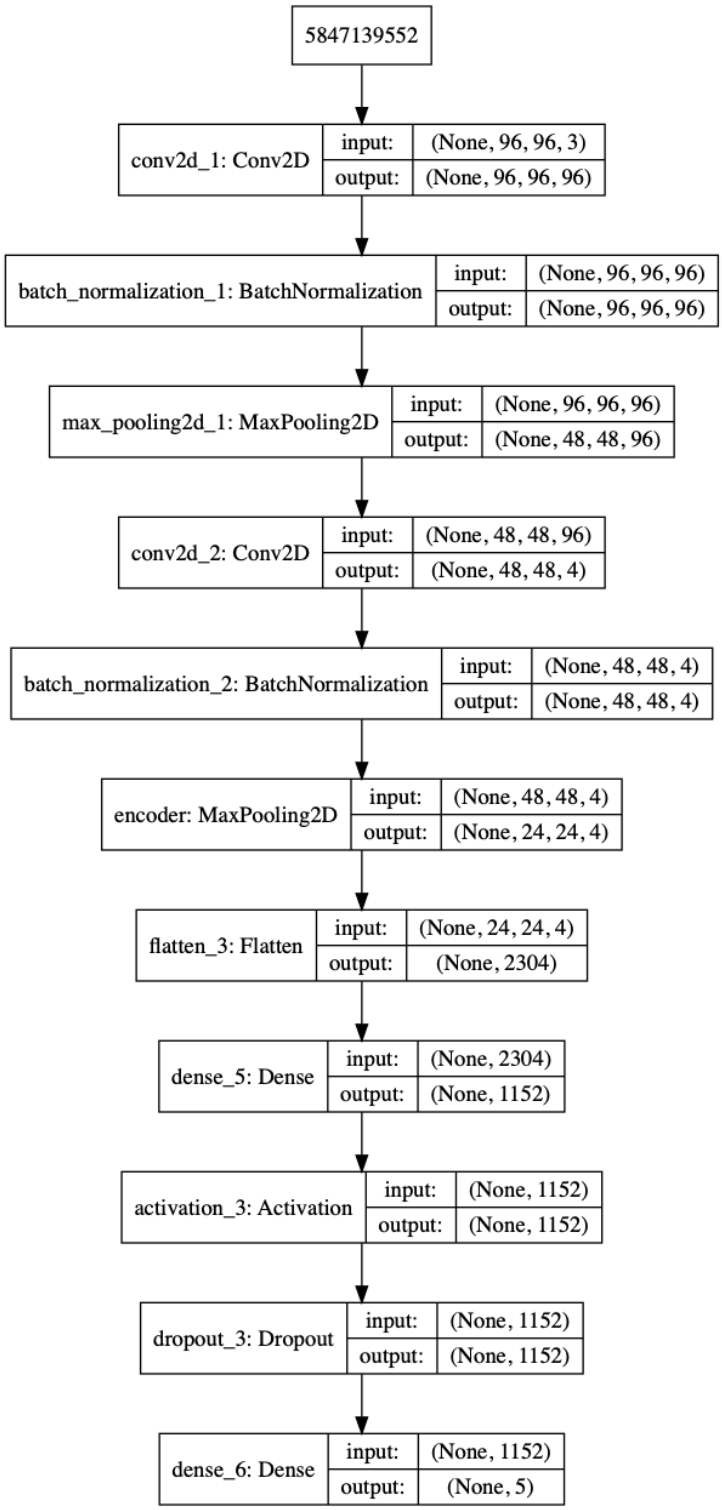

Figure 7: Architecture for the home-cooked classification network. Input dimensions are $96 \times 96 \times 3$, and the output vector is 5×1.

Architecture The common architecture of the three approaches, apart from the trainability and weight initialization of the parameters, is shown in figure 7. Several layers were added at the end of the encoder part to allow for classification training. Specifically, the encoded representation was flattened, and passed 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 was 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.

Data augmentation The get the most out of the images at hand, which are in relative low number for the purpose of classification, data augmentation was applied during training. Data augmentation is a technique that simulates additional variability in a dataset by applying minor manipulations, such as rotations or cropping. In this case, the following parameters were applied:

1. A 15 degree range was set for random rotations.
2. A 0.1 range was set for shifts in width.
3. A 0.1 range was set for shifts in height.
4. Horizontal flipping was allowed.

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 four 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[3]. All model training was set for *300 epochs*. Similar to the approach in section 2, training *stopped early* if no improvement was detected in the validation loss for a number epochs. In this case, that number was set to 50.

Evaluation All model training stopped (very) early, despite the forgiving stopping criterion (see figure 8). This suggests that extended training had little to no impact on the validation loss or accuracy of any of the models. Furthermore, the evolution of the training loss suggests that the difference between the approaches boils down to simple rules: 1) the first approach (frozen encoded layers) rendered the model less flexible, thus inhibiting loss decrease compared to the other models, and 2) it appeared better to let the model learn its own weights

[3]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.

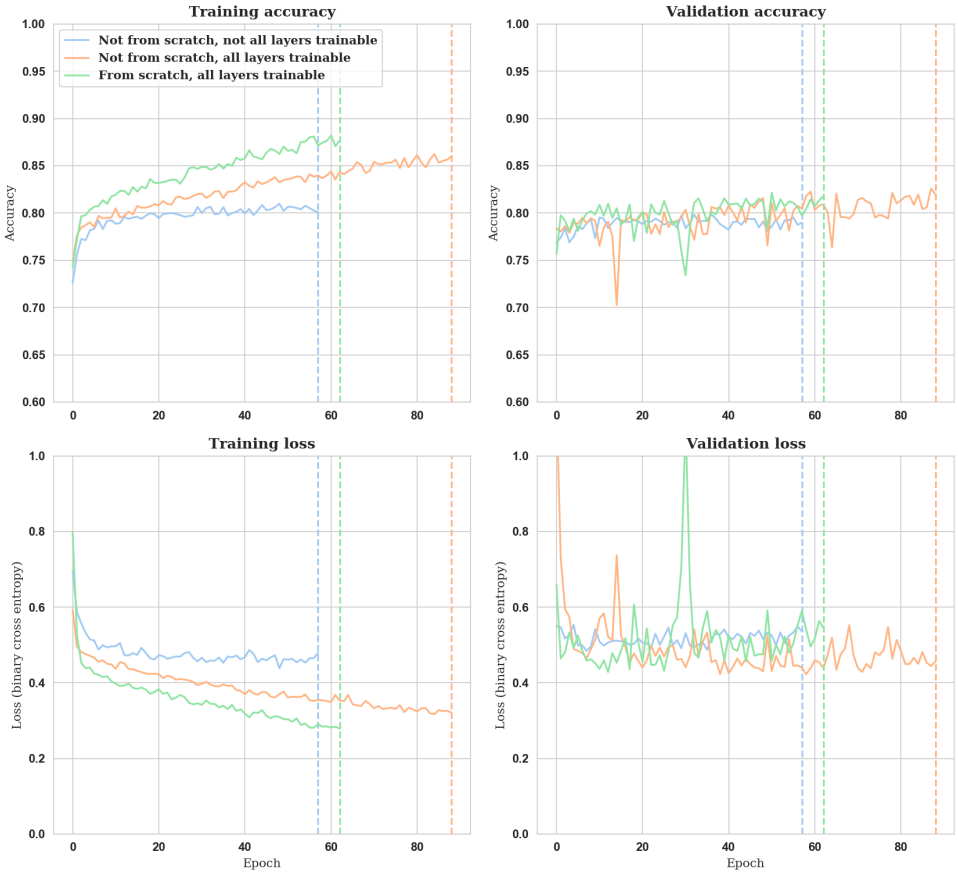

Figure 8: Training history for all classifier models. In the top row, the left graph shows the evolution of the accuracy on the training data, whereas the right graph shows the evolution of the validation accuracy. In the bottom row, the graphs represent the evolution of the training and validation loss (*binary crossentropy*). The vertical dotted line indicates when early stopping occurred, due to a lack of improvement in the latter value.

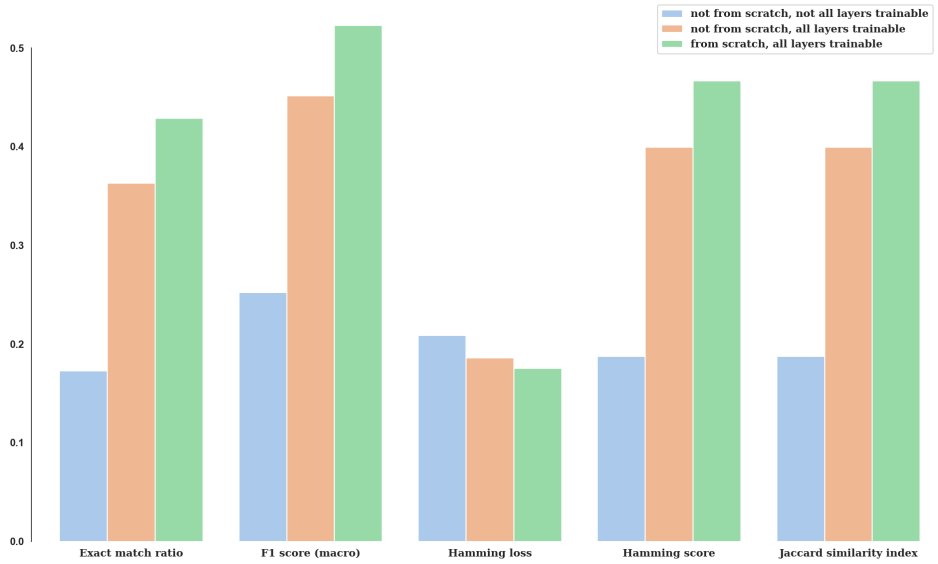

Figure 9: Evaluation metrics applied to test image classification, for each of the five different modeling approaches.

from scratch, rather than initializing them to the values that were learnt for the autoencoder problem, and 3) the Inception-based network simply performed better, likely due to the combination of the more advanced architecture and the (costly) pretraining. Interestingly, the strategy learnt during the training phase of the deep convolutional autoencoder (Approach 1: not from scratch, not all layers trainable) did not simply translate to a better (or more rapidly acquired) strategy for the classification problem. On the contrary, it appeared to be more of a hindrance, causing training performance to lag behind a randomized sibling network.

The problem we are dealing with here is of the **multi-class, multi-label** classification variety. Our architecture deals with this by treating every label independently. In some sense, we are treating the problem as multiple binary classification problems rather than a single categorical classification task. This causes the accuracy calculated by Keras, which is also shown in the figures, to be inflated. Since most images have one, maximum two positive labels, the vast majority of target values is 0. This means that a model that consistently predicts 0 outcomes is likely to be considered fairly 'accurate' by the default metric, even though no strategies are learnt. We thus require better evaluation metrics that account for this inflation problem.

Here, several multi-label classification metrics were used to evaluate the different models. They were calculated based on the previously unseen test images. Figure 9 shows the scores and rankings of each of the models. Where

there was a choice, the *macro-average* of the metric was used, rather than the *micro-average*. The former approach computes the metric independently for each class, and returns the average. The micro-average combines the contributions per class to compute the average metric. In a multi-class classification setup, micro-average is preferable if you suspect there might be class imbalance (i.e you may have many more examples of one class than of other classes. This was not the case.

- **Exact match ratio** represents the proportion of target vectors that got predicted exactly, meaning each of the prediction node values matched their respected target values. *Higher values are better.*
- F_1**-score** represents the harmonic average of them model's precision and recall. Calculated globally using the total amount of true positives, false negatives and false positives. *Higher values are better.*
- **Hamming loss** is defined as the fraction of labels that are incorrectly predicted. *Lower values are better.*
- **Hamming score** is calculated as the number of correct labels divided by the union of predicted and true labels. *Higher values are better.*
- **Jaccard similarity score** compares members for two sets —the predicted labels and true labels—to see which members are shared and which are distinct. *Higher values are better.*

Inspection of the metrics (figure 9 suggests that—among these first three models—the third approach was most effective at creating an accurate model, followed closely by the second approach (all layers trainable and starting weights randomized). This is primarily evident in the higher F_1-score and the slightly lower Hamming loss. The first approach (frozen encoder layers) underperformed in comparison, suggesting that the feature representation learned by autoencoders is not directly transferable to the classification problem setting.

Expansion on homemade models Because little to no improvement was found in validation accuracy and loss (and since this came across as suspicious) the early stopping criterion was removed from the home-made model training. This decision was made to verify the longer term effects of training. However, even with extended training duration (see figure 10), no evidence was found of a(n eventual) positive impact on validation accuracy and loss. On the contrary, it seemed as though any improvement in training loss came at the cost of an increase in validation loss, suggesting overtraining. The models thus became increasingly better at handling the training data—especially in the third approach, with all trainable layers and randomized starting weights—but no generally applicable strategy was learnt. In an attempt to better these unsatisfactory results,

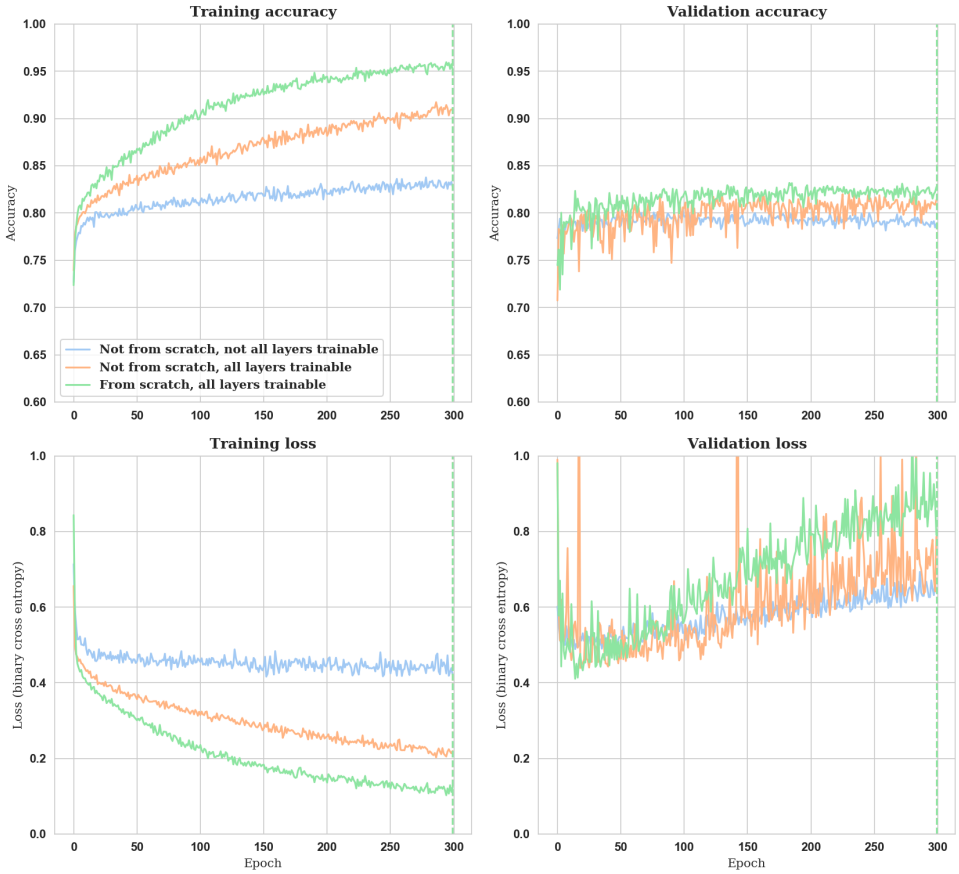

Figure 10: Training history for all classifier models. In the top row, the left graph shows the evolution of the accuracy on the training data, whereas the right graph shows the evolution of the validation accuracy. In the bottom row, the graphs represent the evolution of the training and validation loss (*binary crossentropy*). The vertical dotted line indicates when early stopping occurred, due to a lack of improvement in the latter value.

new modeling approaches were undertaken. They are described in the next sections.

3.2 Inception-based classification

Inception is an image recognition model designed by researchers at Google. It was specifically developed to solve problems in the field of Computer Vision. The architecture is notable for its leveraging of various ideas, old and new, into a performant new breed of neural network.

Architectures The architecture of the Inception classifier is visualized in figure 11. The convolutional base was taken from the InceptionResNetV2 architecture (as it is implemented in Keras). The underlying architecture for these base layers is described here[4]. The model base was pre-trained on the ImageNet database. Similar to the architectures described above, two fully connected layers were added to the model to allow for classification training.

Network parameters The model was trained using the same parameter settings as described in previous section: it trained for *300 epochs* with *early stopping* set at 50, using the *Adam* optimizer function and *binary crossentropy* as loss function. Data augmentation was used during training, similar to what was described in 3.1.

Evaluation Figure 15 depicts the training progression of the Inception-based model. It appears as though the model learnt all it could fairly quickly, after which loss and accuracy stabilized. In that sense, it is similar our the first model (frozen encoder layers): the basis for feature extraction was set (and could not be changed), and the classification layers optimized in short term. In addition, it is clear that validation accuracy is distinctly higher for the Inception-based network. This is likely the result of the pre-training advantage the network was endowed with, illustrating the value of large amounts of training data in deep learning. Evaluation of the final model is shown in 16. Again, it is clear that the Inception-based model is our most performant model thus far. This can be seen in the lesser Hamming loss, the greater F_1-score, and the model's ability to generate a fraction more exact matches.

3.3 Jointly-trained classification network

To explore the advantage of joint training—i.e. building a network for two separate purposes and training it on both at the same time–a final network was built. Here, the network attempts to both reconstruct and classify an image, based on the deep convolutional autoencoder principle (section 2.2). The network's loss

[4] https://arxiv.org/pdf/1512.00567v3.pdf

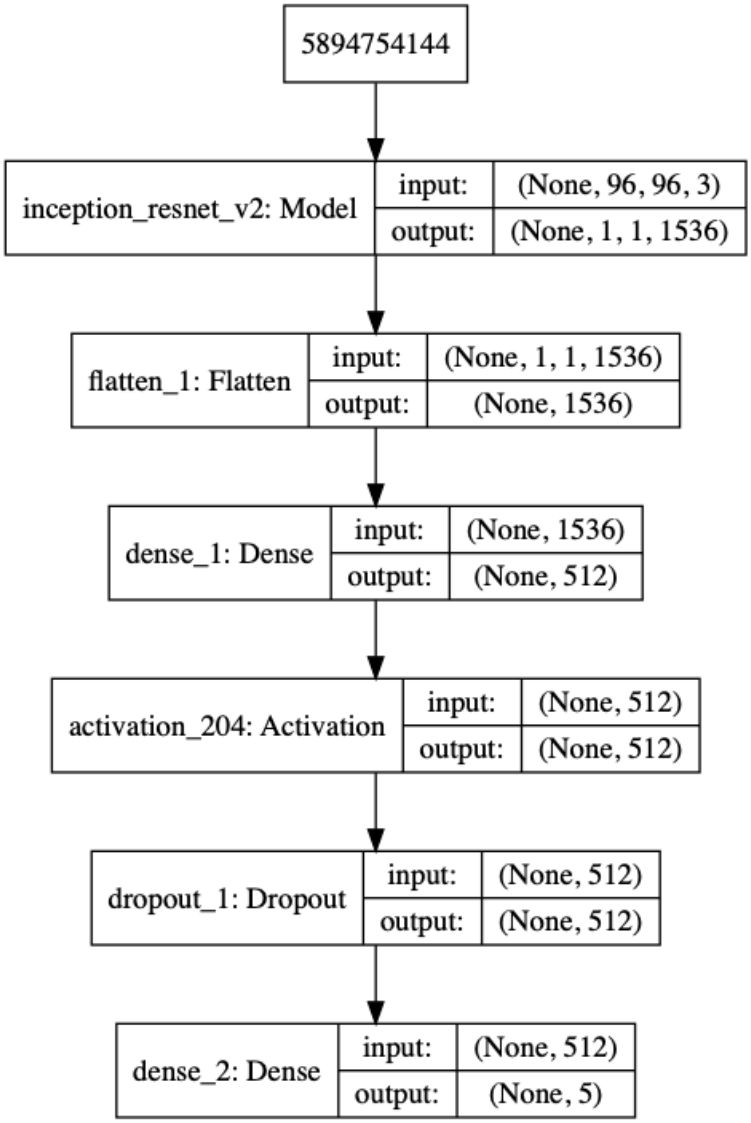

Figure 11: Architecture for the Inception-based classification network. Input dimensions are $96 \times 96 \times 3$, and the output vector is 5×1.

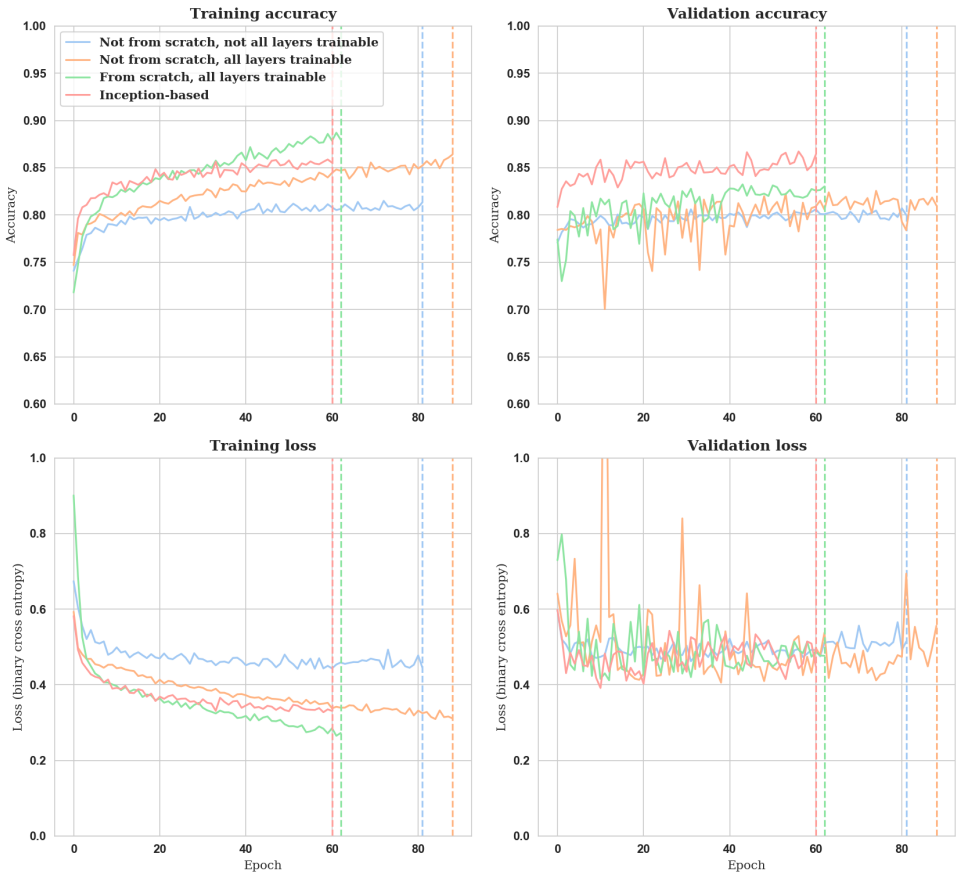

Figure 12: Training history for the Inception-based classification model (red). The previous models are shown as a reference. The vertical dotted line indicates when early stopping occurred.

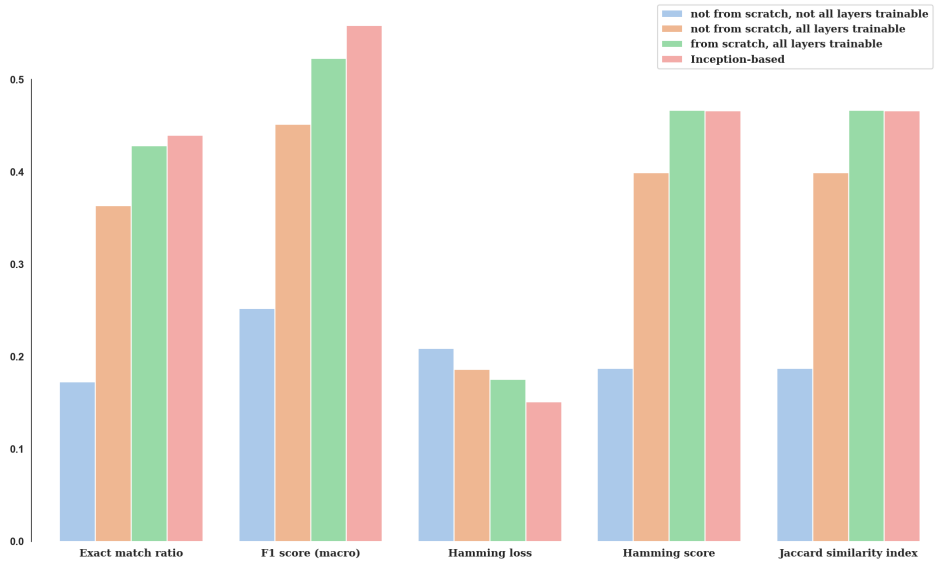

Figure 13: Evaluation metrics for Inception-based classification network. Previous approaches are shown as a reference.

is defined as the sum of the losses calculated on both network outputs. Training attempts to minimize this joint loss, *agnostic to the components it is made out of*.

Architectures This section's architecture is shown in figure 14

Network parameters The model was trained using the same parameter settings as described in previous section: it trained for *300 epochs* with *early stopping* set at 50, using the *Adam* optimizer function and *binary crossentropy* as loss function.

Evaluation Inspection of both the validation loss and accuracy (**??**), as well as the metrics yielded by the final model (**??**), suggests that the classifier that resulted from our joint training approach is of little value. While this approach appeared to boost training greatly, the strategies learnt did not translate to the validation data. On the contrary: while training loss decreased seemingly exponentially, validation loss exploded. Interestingly (though not depicted in the figure), the autoencoder part of the network benefitted from a similar boost. In this case, however, the decrease in loss could also be seen in the validation data. This suggests that it is useful to implement such joint training approach for the purpose of image reconstruction, but not for classification.

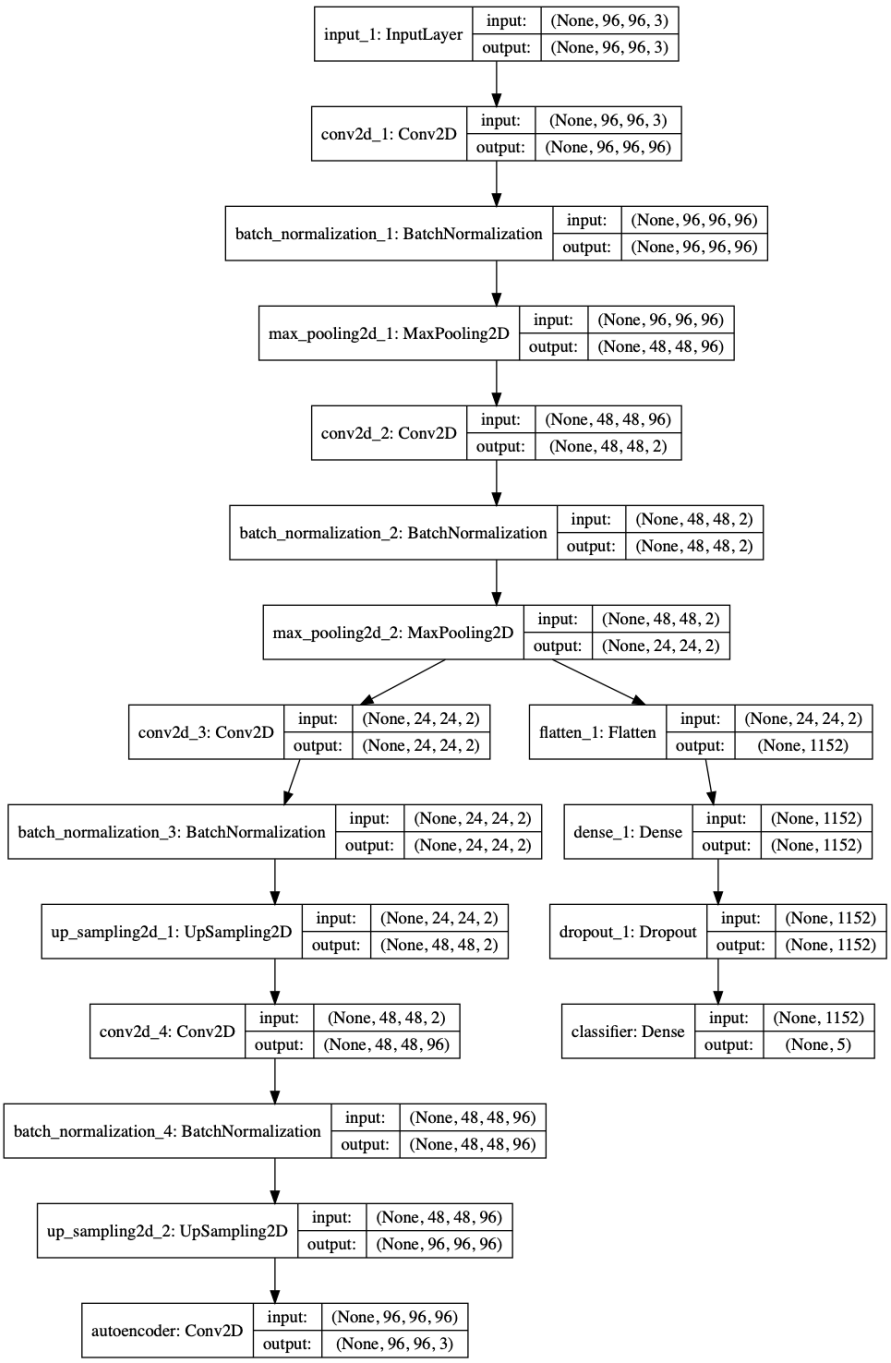

Figure 14: Architecture for the jointly-trained network. Input dimensions are $96 \times 96 \times 3$, and the output consists of both a $96 \times 96 \times 3$ image reconstruction matrix, and a 5×1 classification vector.

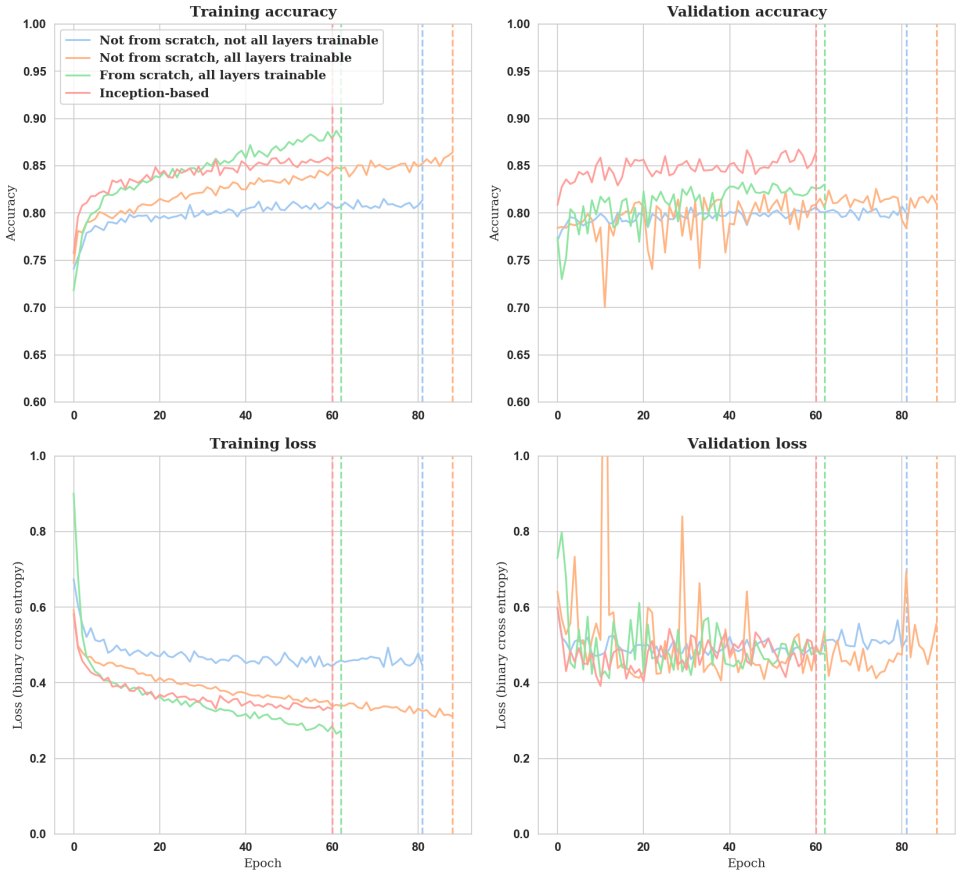

Figure 15: Training history for the Inception-based classification model (red). The previous models are shown as a reference. The vertical dotted line indicates when early stopping occurred.

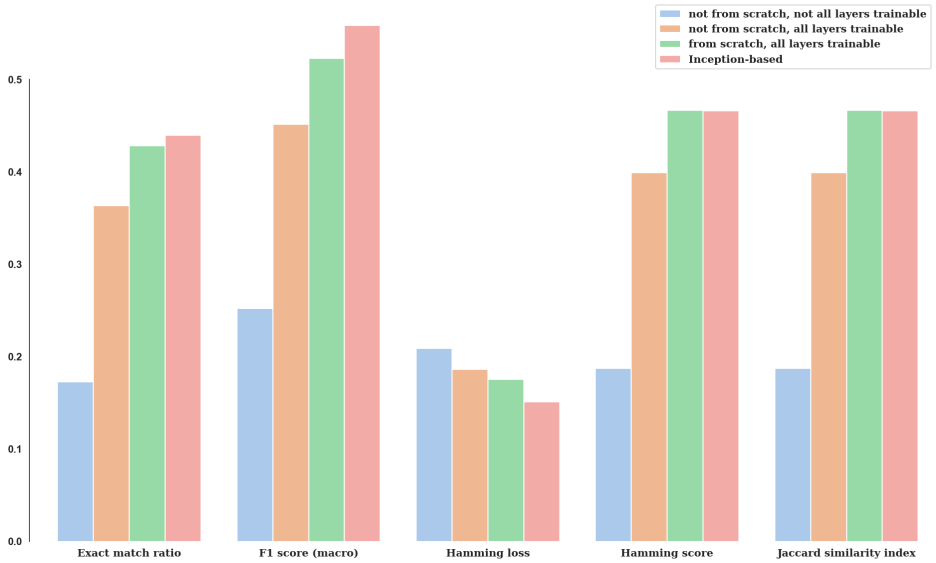

Figure 16: Evaluation metrics for Inception-based classification network. Previous approaches are shown as a reference.

3.4 Reflection

For the most part, the models presented in this section clearly have limited use beyond the training data. The Inception-based model appears to have most predictive value. There are a several possible explanations for this failure.

- With respect to the autoencoder-based models, 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, this 'relevancy' may have been uniquely suited for the reconstruction problem, which is different from the classification problem.

- 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 could be simplified into a binary classification task, to reduce its complexity. Alternatively, additional images could be sought and added to the training data corpus (we could steal some from the validation set).

- Perhaps *binary crossentropy* is not the most efficient loss function for this sort of problem. In the case of multi-label classification, the loss function appears to suffer from inflation, as it is based on the individual output

Figure 17: Classification scores for each of the class labels, generated by each model for a set of test images. Values over 0.5 (dotted line) would be considered a 'detection'.

node values. Alternatively, it may be more effective (but computationally not more efficient) to instantiate a separate model for each class. The multi-label classification problem then becomes set of binary classification problems. This way, the weights of each model can adjust themselves separately, specifically learning strategies to detect a single class.

4 Segmentation

4.1 Binary segmentation network

This section[5] 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 section 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 18. It is very closely related to the DCA architecture (see figure 4, 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 model gives fair results, however, hence this architecture was kept. The kernel values were initialized in a random uniform manner throughout the network, 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, the model was trained 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.

Evaluation The training process is visualized in figure 19. The graph suggests that some improvement was still possible, but overtraining would become a real

[5] The code for this section is found at `scripts/itnodl_segm.py`.

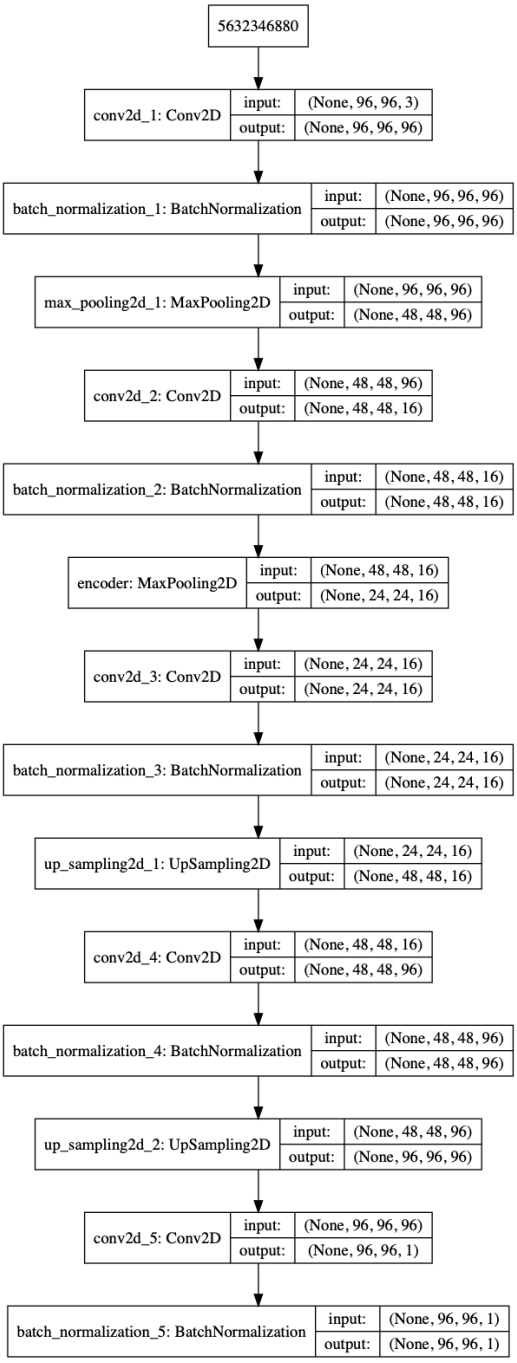

Figure 18: Deep convolutional architecture for the segmentation network. Input dimensions are $96 \times 96 \times 3$. The model predicts a score per pixel position, ignoring color channels—hence the output dimensions are $96 \times 96 \times 1$.

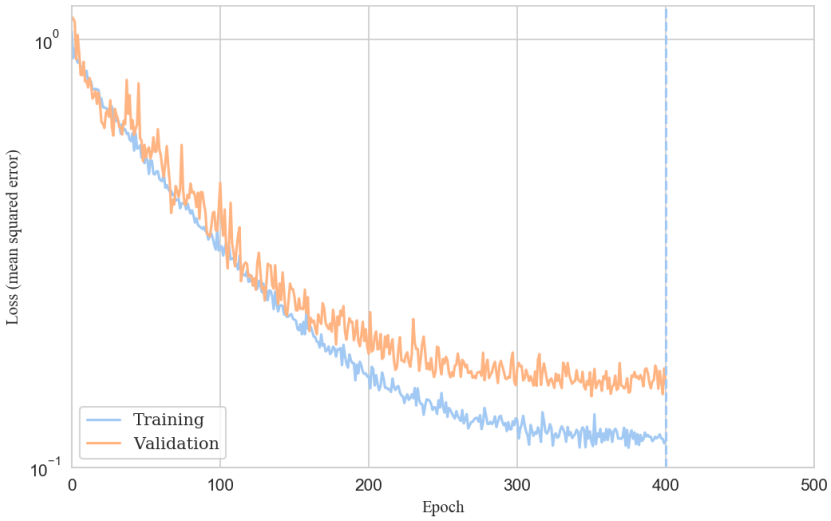

Figure 19: Training history for the segmentation network. The y-axis represents the loss value (mean squared error). Both training and validation loss are plotted. The vertical dotted line indicates when early stopping occurred, due to a lack of improvement in the validation losse. Loss is expressed on a logarithmic scale.

risk. Figure 20 shows how the network segmented a random selection of (previously unseen) test images. The model appears to yield acceptable results, especially given the very limited amount of training images it could work with. The average *mean squared error* loss for each of the datasets is shown in table 4, along with the average *dice loss*. The table indicates that the model is better accustomed to the training data, without being overly biased towards the validation images.

Figure 20: 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 base	Loss	**Train**	**Validation**	**Test**
Autoencoder	*MSE*	9.32×10^{-2}	1.50×10^{-1}	1.43×10^{-1}
	Dice loss			
U-Net	*MSE*	1.34×10^{-1}	1.39×10^{-1}	1.50×10^{-1}
	Dice loss	5.80×10^{-1}	6.00×10^{-1}	5.94×10^{-1}

Table 4: Evaluation of segmentation models using *mean squared error (MSE)* and *dice loss* values for training, validation and test datasets.

4.2 U-Net-based segmentation network

In an effort to improve segmentation results, a second segmentation network was built. This time, the architecture was derived from U-Net—the convolutional network proposed for the purpose of biomedical image segmentation.

Architecture The second segmentation model's architecture is shown in figure 21. It is slightly more convoluted compared to the previous model, but still relatively simple. The so-called *skip connections* mentioned in the assignment text can be seen, departing from every other convolutional layer to a 'partner' convolutional layer, thus helping the model to fine-tune its results in an early stage.

Network parameters The model was trained using the same strategy as described in section 4.1. Image dimensions were kept the same, which is possible due to the dimension-agnostic nature of the U-Net architecture. The *Adam* optimizer was used in training, in combination with *mean squared error* loss. The total number of *training epochs* was set to 500, and *early stopping* occurred when improvement in the validation loss was absent for 50 epochs.

Evaluation Figure 19 shows the training history. The average *mean squared error* loss for each of the datasets is shown in table 4, along with the average *dice loss*. The table indicates that the model is better accustomed to the training data, without being overly biased towards the validation images. Some segmentation examples (on test images) are shown in figure 22.

4.3 Reflection

Possible enhancement strategies While the segmentation network described here has its merits, it is obviously far from perfect. Depending on the actual purpose of the network, additional operations could be applied to get the most out of the results. Should the problem be reframed as a detection problem (i.e.,

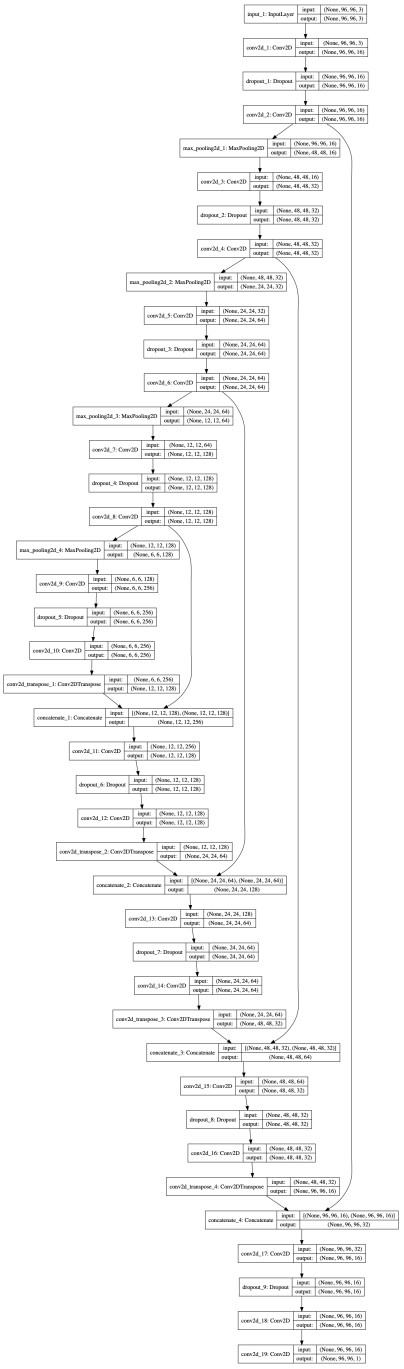

Figure 21: U-Net-based architecture for the second segmentation network. Input dimensions are $96 \times 96 \times 3$. Output dimensions are $96 \times 96 \times 1$.

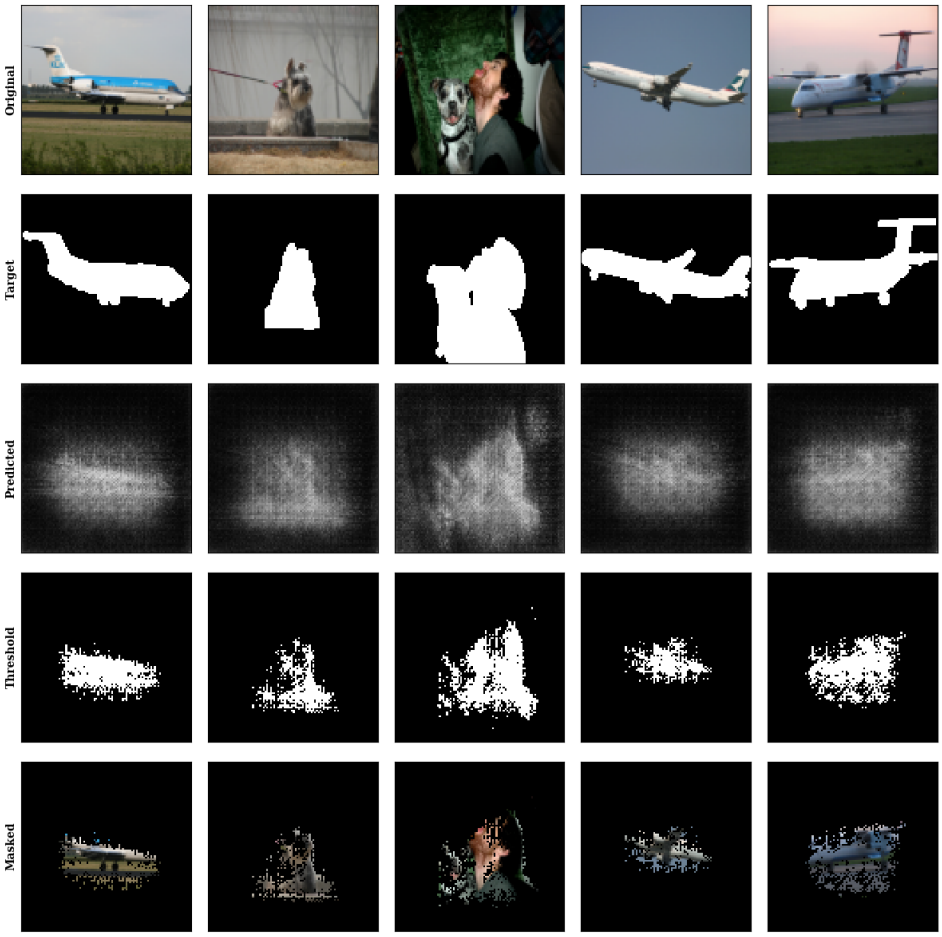

Figure 22: Segmentation given by U-Net-based mdoel for a random selection of 5 'fresh' test images ($96 \times 96 \times 3$ pixels). The top row shows the original images. The second row shows the binarized segmentation labels. The middle row shows the model output for the source images. The second to last row shows binarized predictions (threshold = 0.5). The bottom row, finally, uses the binarized prediction as a mask over the source image.

there is one object in the foreground: find it!), then it could suffice to find the center of the predicted foreground pixels and draw a rectangle around them (of which the dimensions are again based on the output activation).

If the objective is to extract the foreground object (by using the prediction as mask), this can be done more effectively by combining the network results with edge detection algorithms and morphological operations. The former may help create a better, more sensible outline for the detected foreground object(s). The latter may help in, for instance, filling in gaps, or generally smoothing out the result.

Figure 23: This picture shows three purposely selected training images, along with their target, predicted, and thresholded segmentations, to illustrate apparent model strategizing attempts.

Observation While observing the output generated by the network, some interesting observations could be made. Sometimes the model makes mistakes in such a way that it shows its attempts at creating a generalizable strategy. More specifically, we can see how the model searches for distinct features such as edges, color differences, etc.

Consider figure 23. The first image (the bird sitting on the wire) came accompanied by a segmentation image that highlighted the bird, and nothing else. The model, on the other hand, shows tentative activation of the pixels in and around the wire. This 'mistake' is in fact rather intuitive, as the position of the wire is at about the same depth as the bird (the bird is setting the wire, after all). It can therefore be considered 'foreground' without much argument, suggesting that the model attempted to override the somewhat 'erroneous' labeling it was given. Similarly, for the second image, the model highlights the structure on the right on top of the bird. This is again a fairly forgivable mistake, as the structure appears to be quite distinct from the background, making it foreground by virtue of the problem's binary nature. The third image, finally, shows how these strategies can in some cases lead to errors that are not as easily classified as 'better judgment of the model'. Here, we see a discoloration in the background, due to a rock or gravel heap. This element is of course still in the background—something that is rather clear for a human observer. Due to its distinct visual features, however, the model mistakenly tries to add it to the foreground.

Importantly, all the images shown here are training images. The mistakes made here persisted despite numerous chances for the model to adjust for them. They are thus inherently linked with the model's attempt at learning a strategy.

5 Discussion