Computer Vision: Deep Learning Project

Advanced Master in Artificial Intelligence, KU Leuven

Anthoula Mountzouri

Academic Year: 2018-2019

1. Data:

The PASCAL VOC-2009 dataset was used. This dataset consists of color images of various scenes with different objects classes. 5 classes out of 20 are used this project:

- aeroplane
- car
- chair
- dog
- bird

The original dataset contains 1489 training and 1470 validation images. For the classification purposes the number of images retained the same. The size of the images was selected equal to 212x212 initially, but for some cases the size of 208x208 was also used. For the case of segmentation, 749 training images and 750 validation images are provided. For this reason, is applied data augmentation so that to double the training images. The images were also resized in this case to be 144x144.

2. PCA vs Linear Auto-encoder

Principal Components Analysis (PCA) is a technique used for dimensionality reduction. PCA reduces the data frame by orthogonally transforming the data into a set of principal components. The first principal component explains the most amount of the variation in the data in a single component, the second component explains the second most amount of the variation and so on. By choosing the top k principal components that explain say 90-95% of the variation, the other components can be dropped since they do not significantly benefit the model. If the data contains strongly correlated variables, few principle components are needed in order to explain the data. When there is almost no correlation in the data, we will need many components in order to explain the data and PCA therefore becomes trivial. These components are calculated via the eigenvectors of the correlation matrix of the data.

Autoencoders are a branch of neural network which attempt to compress the information of the input variables into a reduced dimensional space and then recreate the input data set. Typically, the autoencoder is trained over number of iterations using gradient descent, minimizing the mean squared error. The key component is the “bottleneck” hidden layer. This is where the information from the input has been compressed. However, autoencoders are data-specific meaning that can only be used on data similar to what they were trained on, and making them more general requires lots of training data.

Someone can mimic a linear PCA with an autoencoder structure which consists of an input layer, a single fully connected hidden layer and an output layer, a linear activation function and a squared error cost function. Taking into consideration a PCA that projects the data onto its three principle components, then the bottleneck in the autoencoder should contain three hidden units, so that mimic PCA with three principal components.

Differences:

- Autoencoders have all the information from the original data compressed into the reduced layer, retaining data structure that PCA is unable to do.
- PCA transformation is faster than autoencoders.
- Autoencoders can have a higher dimensional hidden layer. So, they can increase data dimension instead of reducing it. As a result, unlike PCA, autoencoders can transform data from one feature space to another.
- Autoencoders are data-specific meaning that can only be used on data similar to what they were trained on. PCA is a more general transformation.
- PCA is a statistical procedure that uses an orthogonal transformation, whereas weights of linear autoencoders are not necessarily orthogonal.
- PCA is restricted to a linear map, while auto encoders can have nonlinear encoders/decoders. Like that an autoencoder can learn more complex data.

3. Convolutional Autoencoders:

Experimenting with different architectures:

In this section, many different autoencoder architectures are tested, containing either different number of coding variables, different number of convolutional layers and downsamples, or optimizers, loss and activation functions. Also, either very large models were constructed or light-weight architectures that are presented subsequently, recording their performance too. Initially, the different parameters that. Some different scenarios used in order to test the baseline models are listed below.

Activation Functions that are tested: Neural Networks are considered **Universal Function Approximators**. It means that they can compute and learn any function. An appropriate activation function makes the network more powerful and adds ability to it to learn something complex and represent non-linear complex arbitrary functional mappings between inputs and outputs. Hence, a non-linear activation can help in generating non-linear mappings from inputs to outputs. Also, another important feature of an activation function is that it should be differentiable. If so, it can perform backpropagation optimization strategy while propagating backwards in the network to compute gradients of Error(loss) with respect to Weights and then accordingly, optimize weights using Gradient descend or any other Optimization technique to reduce Error.

- **ReLu:** Everything but the last layer, in all the architectures, uses either Relu or Leaky Relu. Its form is R(x) = max (0, x) i.e. if x < 0, R(x) = 0 and if x >= 0, R(x) = x. It is a powerful activation function for grabbing non-linear features, avoiding the **Vanishing Gradient problem**. But its limitation is that it should only be used within Hidden layers of a Neural Network Model. That's why for the last(output) layer of all the networks presented, the activation function that is used is the sigmoid function. Another problem with ReLu, is that some gradients can be fragile during training and can die. It can cause a weight update which will make it never activate on any data point again. In other words, ReLu could result in dead neurons.
- **Leaky ReLu**: The Leaky Relu activation function is a modification of the ReLu function in order to fix the problem of dying neurons. It introduces a small slope to keep the updates alive.
- **Sigmoid**: It is an activation function of form f(x) = 1 / 1 + exp(-x). Its range is between 0 and 1. It is a S-shaped curve. The main reason which have made it to fall out of popularity is the Vanishing gradient problem. Also, it has slow convergence. Sigmoid was selected for the output layer of the architectures constructed.

Loss functions that are tested:

- **Mean Squared Error (MSE):** MSE measures the average of squared differences between prediction and actual observation. The higher this value, the worse the model is. It is never negative, since it is squaring the individual prediction-wise errors before summing them, but it would be zero for a perfect model. Its main disadvantage is that in case of making a single very bad prediction, the squaring will make the error even worse and it may skew the metric towards overestimating the model's badness. This could be a very problematic behavior in case of having noisy data, as like that even a good model may have a high MSE in that situation, so it becomes hard to judge how well the model is performing. On the other hand, if all the errors are small, or rather, smaller than 1, then the opposite effect is felt: the model's badness may be underestimated.
- **Mean Absolute Error (MAE):** MAE measures the average of the absolute differences between prediction and actual observation where all individual differences have equal weight. What is important about this metric is that it penalizes huge errors that not as that badly as MSE does. Thus, it's not that sensitive to outliers as mean square error.

Approaches for Downsampling:

- **Maxpooling:** It calculates the maximum value for each patch of a feature map. The pooling layer operates upon each feature map separately to create a new set of the same number of pooled feature maps. This means that the pooling layer will always reduce the size of each feature map by a factor of 2, e.g. each dimension is halved, reducing the number of pixels or values in each feature map to one quarter the size.
- **Strides:** An alternative option for size reduction could be the use of stride [2,2]. The stride of the filter on the input image can be used to down-sample the size of the output feature map. Hence, in order to test this option too, some relevant architectures were constructed where the pooling layers were replaced by stride [2,2].

Optimizers:

- **RMSprop():** RMSprop is one of the most popular optimization algorithms used in deep learning. It is good, fast while its popularity is only surpassed by Adam. Rather than having just a global, scalar learning rate, in this case we have a vector of learning rates for each trainable parameter. It is iteratively updated with a running average of magnitudes of squares of previous gradients.
- **Stochastic gradient descent (SGD):** Stochastic gradient descent (SGD) performs a parameter update for each training example. It is usually fast and can also be used to learn online.
- **Adadelta:** Adadelta adapts learning rates based on a moving window of gradient updates, instead of accumulating all past gradients. This way, Adadelta continues learning even when many updates have been done.

Other considerations in order to improve the models:

- **Batch Normalization:** In order to further speed up the learning the addition of batch normalization layers was also tested in some models. The batch normalization layers were added after the convolution layers (or in the case of using as activation function the Leaky ReLU, after the activation) such that the results are normalized for the next layers. Batch normalization is also said to aid stability or remove noise.

Note: Also, same architectures were tested using different batch size and epochs trying to figure out how these parameters affect the networks performance. The different architectures tested, are presented below.

1st Architecture: The first autoencoder architecture tested, is also illustrated in the next Figure. The encoder part is comprised of three convolutional layers (16->8->8) followed by pooling layers that downsample the representation from 212×212×3 to 27×27×8. The decoder part (8->8->16) performs upsampling operations and brings back the representation to 212×212×3. The activation used was the ReLu function and the number of coding variables

occurred is 5832. A series of different experiments are represented for this model. The training and validation loss are recorded for each experiment as well as a reconstructed image.

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	(None, 212, 212, 3)	0
conv2d_8 (Conv2D)	(None, 212, 212, 16)	448
max_pooling2d_4 (MaxPooling2	(None, 106, 106, 16)	0
conv2d_9 (Conv2D)	(None, 106, 106, 8)	1160
max_pooling2d_5 (MaxPooling2	(None, 53, 53, 8)	0
conv2d_10 (Conv2D)	(None, 53, 53, 8)	584
max_pooling2d_6 (MaxPooling2	(None, 27, 27, 8)	0
conv2d_11 (Conv2D)	(None, 27, 27, 8)	584
up_sampling2d_4 (UpSampling2	(None, 54, 54, 8)	0
conv2d_12 (Conv2D)	(None, 54, 54, 8)	584
up_sampling2d_5 (UpSampling2	(None, 108, 108, 8)	0
conv2d_13 (Conv2D)	(None, 106, 106, 16)	1168
up_sampling2d_6 (UpSampling2	(None, 212, 212, 16)	0
conv2d_14 (Conv2D)	(None, 212, 212, 3)	435

Total params: 4,963
Trainable params: 4,963
Non-trainable params: 0

Figure: Model architecture

1st experiment:

- **Loss Function:** Mean Squared Error
- **Optimizer:** RMSprop

Epochs: 50 and batch size: 128

Figure: Training and validation loss diagram (left), Original Image and Reconstructed one (right)

Epochs: 200 and batch size: 64

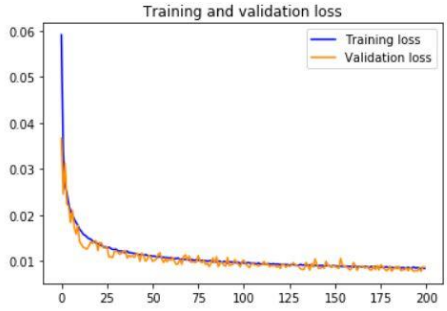


Figure: Training and validation loss diagram (left), Original Image and Reconstructed one (right)

Epochs: 50 and batch size:256

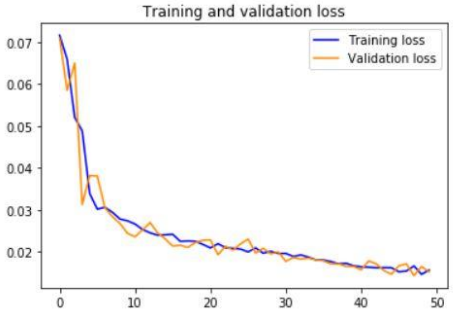


Figure: Training and validation loss diagram (left), Original Image and Reconstructed one (right)

2nd experiment:

- **Loss Function:** Mean Squared Error
- **Optimizer:** stochastic gradient descent (SGD)
- Epochs**:**200 and batch size:64

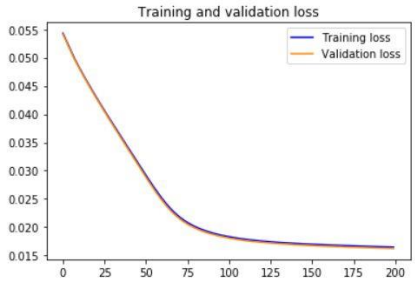


Figure: Training and validation loss diagram (left), Original Image and Reconstructed one (right)

3rd experiment:

- **Loss Function:** Mean Squared Error
- **Optimizer:** Adadelta
- Epochs:200 and batch size:64

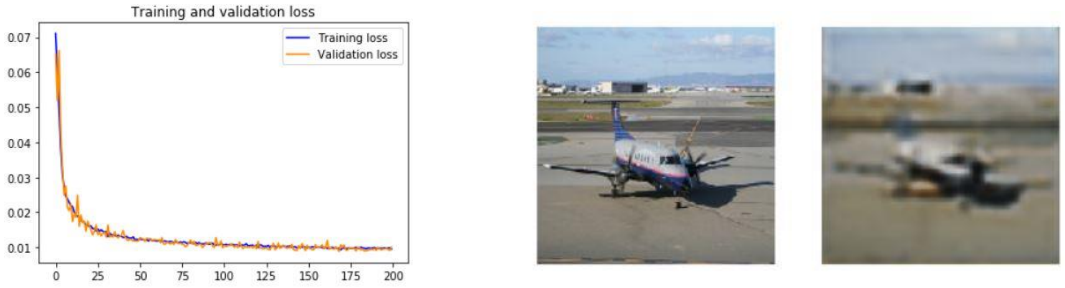


Figure: Training and validation loss diagram (left), Original Image and Reconstructed one (right)

4th experiment:

- **Loss Function:** Mean Absolute Error
- **Optimizer:** RMSprop
- Epochs:200 and batch size:64

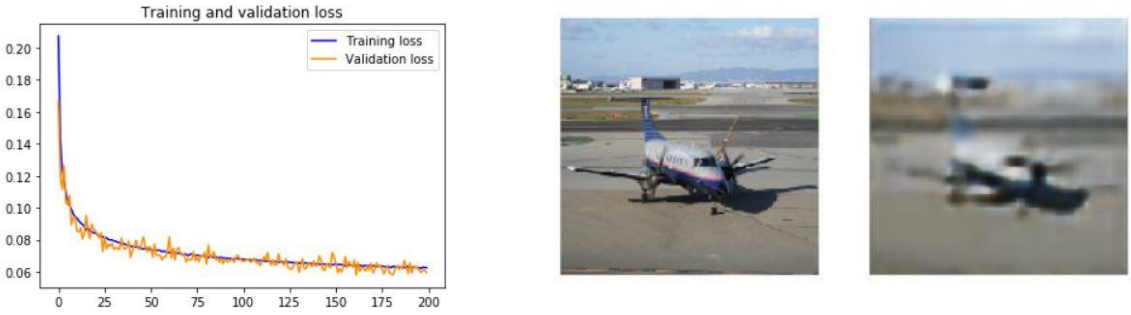


Figure: Training and validation loss diagram (left), Original Image and Reconstructed one (right)

5th experiment:

- **Loss Function:** Mean Absolute Error
- **Optimizer:** Stochastic Gradient Descent
- Epochs:200 and batch size:64

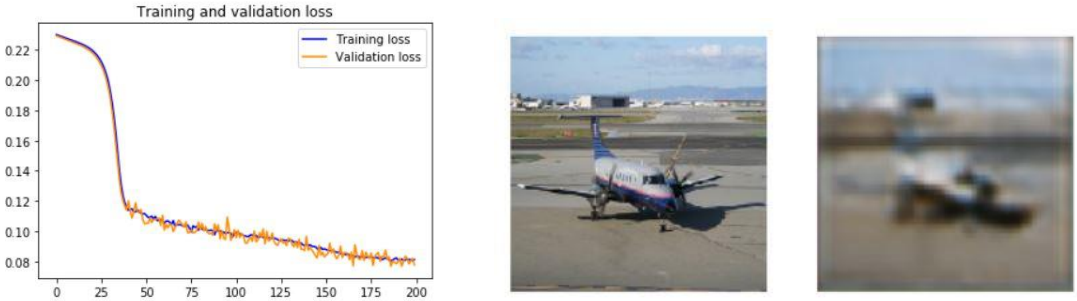


Figure: Training and validation loss diagram (left), Original Image and Reconstructed one (right)

6th experiment:

- **Loss Function:** Mean Absolute Error
- **Optimizer:** Adadelta
- Epochs:200 and batch size:64

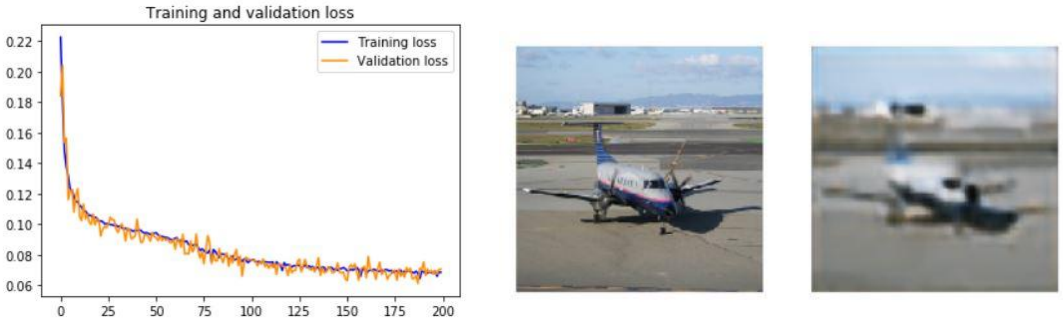


Figure: Training and validation loss diagram (left), Original Image and Reconstructed one (right)

As a conclusion someone could say that either using mae or mse, the stochastic gradient descent has a really bad performance as it converges really fast and proves unable to learn finally. In general, the results of reconstruction are slightly better when using mae, but it is worth mentioning here that the model proved to be a bit slower in this case. The performance of Adadelta optimizer compared to that of RMSprop seems to be the same; slightly better that of Adadelta, though the colors seem a bit brighter in some points.

Figure - MSE, RMSprop(left), MSE adadelta (right)

Figure - MAE, RMSprop(left), MAE adadelta (right)

Figure: Original Image(left), Reconstructed ones (right)

2nd Architecture: The encoder part is comprised of five convolutional layers (64->32->16->8->8) followed by pooling layers that downsample the representation from 212×212×3 to 7×7×8. The decoder part (8->8->16->32->64 feature maps) performs upsampling operations and brings back the representation to 212×212×3. The activation used was the ReLu function and the number of coding variables occurred is 392. A series of different experiments are represented for this model, too. The training and validation loss are recorded for each experiment as well as a reconstructed image.

1st experiment:

- **Loss Function:** Mean Squared Error
- **Optimizer:** RMSprop
-

Epochs: 200 and batch size: 64

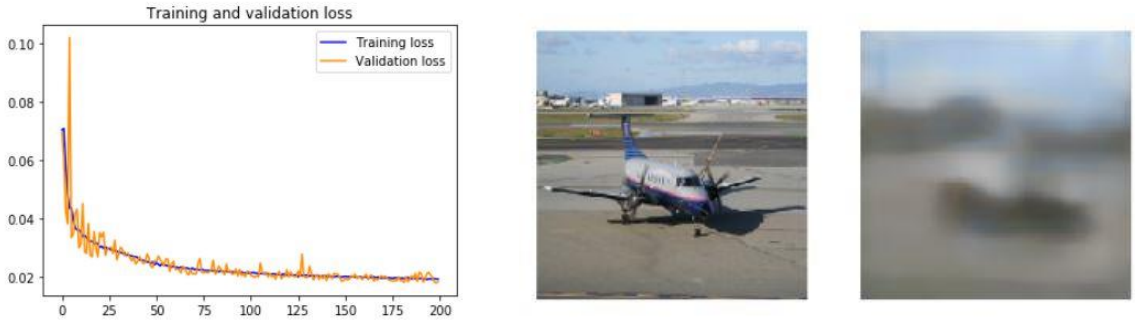


Figure: Training and validation loss diagram (left), Original Image and Reconstructed one (right)

Epochs: 50 and batch size: 64

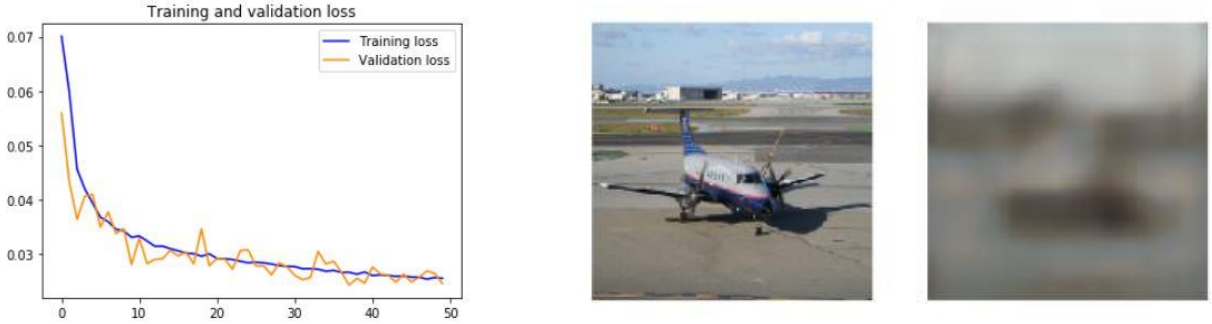


Figure: Training and validation loss diagram (left), Original Image and Reconstructed one (right)

2nd experiment:

- **Loss Function:** Mean Absolute Error
- **Optimizer:** RMSprop
- **Epochs:** 50
- **Batch size:** 128

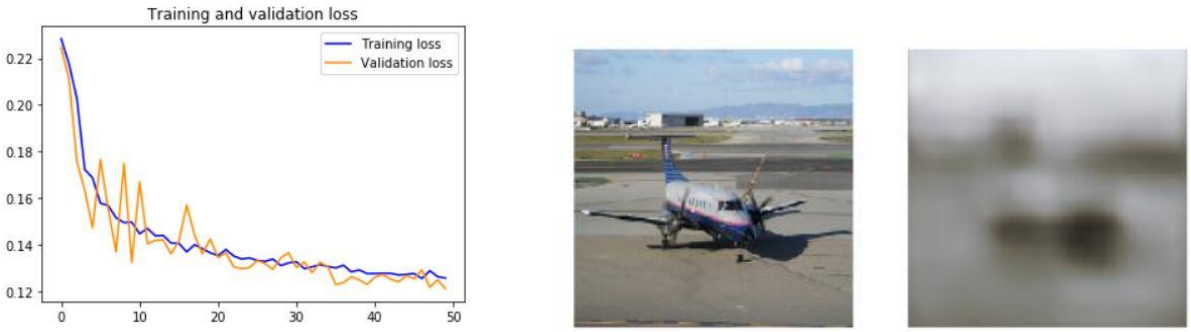


Figure: Training and validation loss diagram (left), Original Image and Reconstructed one (right)

As a conclusion for this architecture, the results in all these cases proved to be very bad which is logical actually, since the number of coding variables used is extremely small.

3rd Architecture: The encoder part is comprised of four convolutional layers (32->16->8->8) followed by pooling layers that downsample the representation from 212×212×3 to 14×14×8. The decoder part (8->8->16->32 feature maps) performs upsampling operations and brings back the representation to 212×212×3. The activation used was the ReLu function and the number of coding variables occurred is 1568. A series of different experiments are represented for this model, too. The training and validation loss are recorded for each experiment as well as a reconstructed image.

Experiments:

- **Loss Function:** Mean Squared Error
- **Optimizer:** RMSprop

Epochs: 50 and batch size: 128

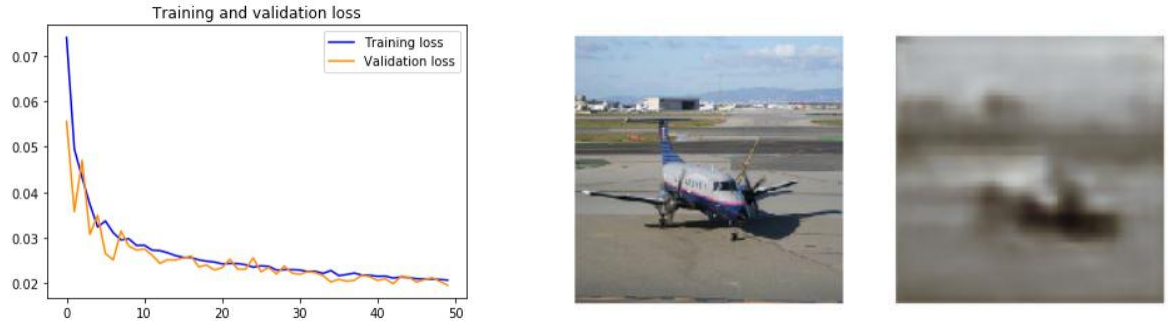


Figure: Training and validation loss diagram (left), Original Image and Reconstructed one (right)

Epochs: 50 and batch size: 256

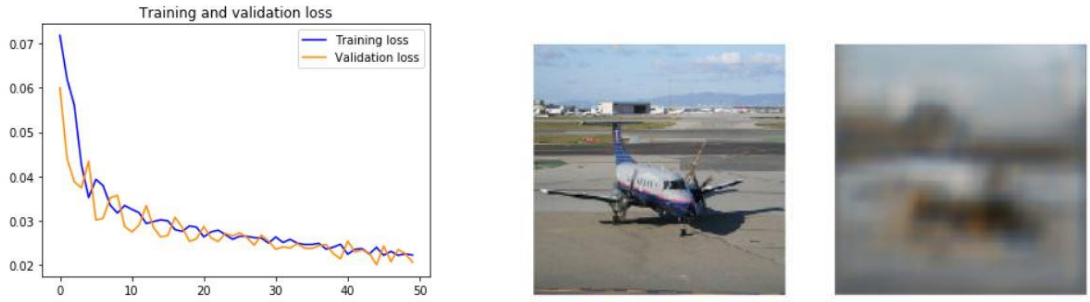


Figure: Training and validation loss diagram (left), Original Image and Reconstructed one (right)

Epochs: 200 and batch size: 128

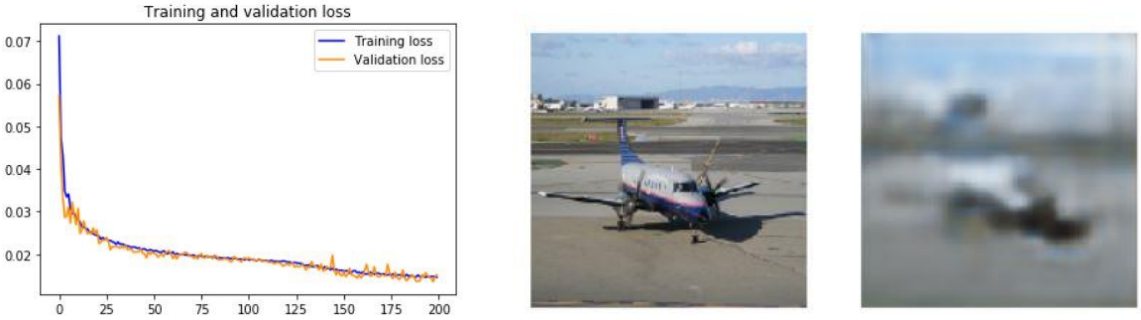


Figure: Training and validation loss diagram (left), Original Image and Reconstructed one (right)

4th Architecture: The encoder part is comprised of four convolutional layers (32->16->8->8) followed by pooling layers that downsample the representation from 212×212×3 to 27×27×8 (the encoder is convolutional layer in this case, compared to the previous architecture). The decoder part (8->8->16) performs upsampling operations and brings back the representation to 212×212×3. The activation used was the ReLU function and the number of coding variables occurred is 5832. The training and validation loss are recorded for each experiment as well as a reconstructed image.

Experiments:

- **Loss Function:** Mean Squared Error
- **Optimizer:** RMSprop
- **Epochs:** 200
- **Batch size:** 128

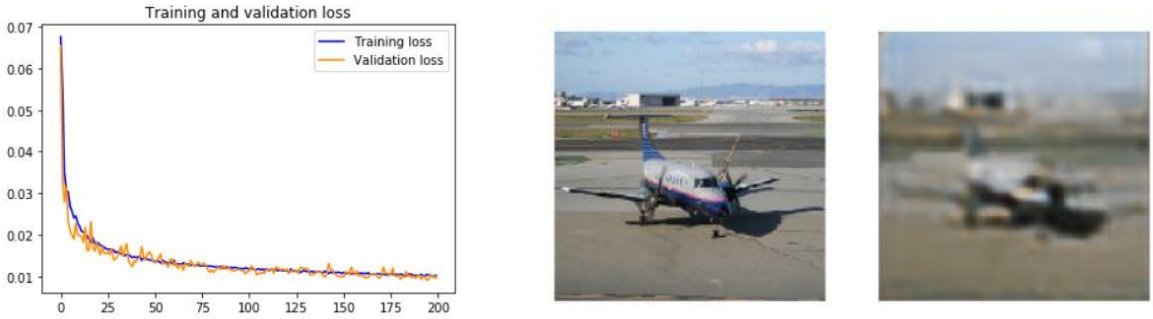


Figure: Training and validation loss diagram (left), Original Image and Reconstructed one (right)

Comparing this architecture with architecture number 4, although they are similar, since the coding variables are more, someone can observe a better reconstruction.

5th Architecture: The encoder part is comprised of five convolutional layers (64->32->16->8->8) followed by pooling layers that downsample the representation from 212×212×3 to 14×14×8. The decoder part (8->8->16->32 feature maps) performs upsampling operations and brings back the representation to 212×212×3. The activation used was the ReLu function and the number of coding variables occurred is 1568. A series of different experiments are represented for this model, too. The training and validation loss are recorded for each experiment as well as a reconstructed image.

1st Experiment:

- **Loss Function:** Mean Squared Error
- **Optimizer:** RMSprop
- **Epochs:** 200
- **Batch size:** 128

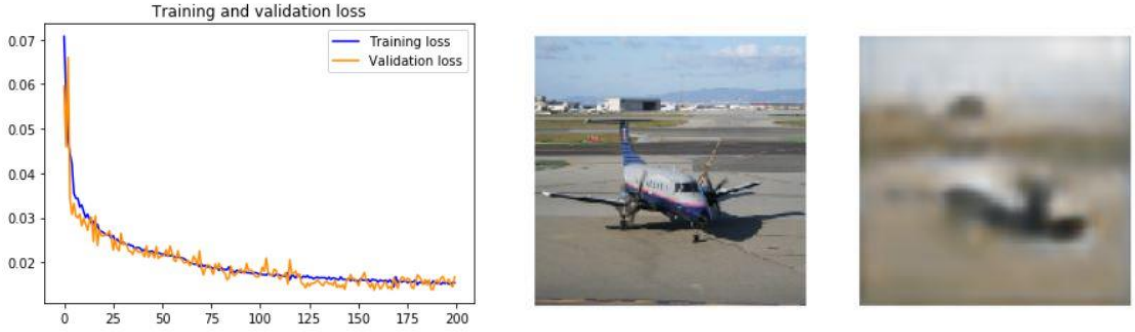


Figure: Training and validation loss diagram (left), Original Image and Reconstructed one (right)

2nd Experiment:

- **Loss Function:** Mean Absolute Error
- **Optimizer:** RMSprop
- **Epochs:** 200
- **Batch size:** 128

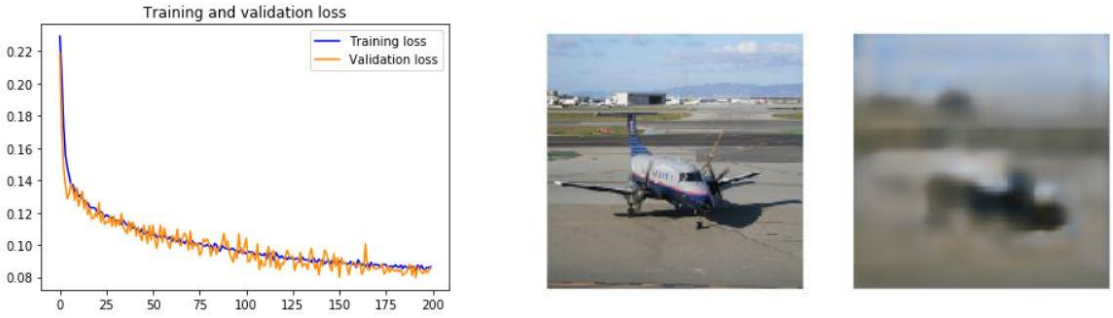


Figure: Training and validation loss diagram (left), Original Image and Reconstructed one (right)

6th Architecture: The encoder part is comprised of six convolutional layers (256->128->64->32->16->8) followed by pooling layers that downsample the representation from 208×208×3 to 13×13×16. The decoder part (8->16->32->64->128) performs upsampling operations and brings back the representation to 208×208×3. The activation used was the ReLu function and the number of coding variables occurred is 2704. A series of different experiments are represented for this model, too. The training and validation loss are recorded for each experiment as well as a reconstructed image.

Experiments:

- **Loss Function:** Mean Squared Error
- **Optimizer:** RMSprop

Epochs: 50 and batch size: 64

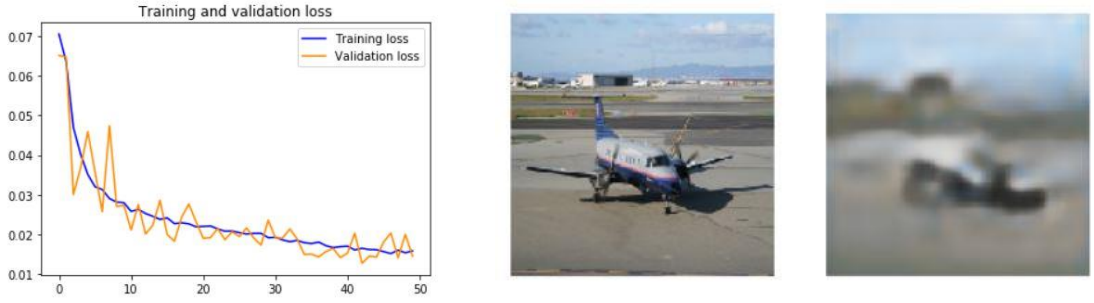


Figure: Training and validation loss diagram (left), Original Image and Reconstructed one (right)

Epochs: 200 and batch size: 64

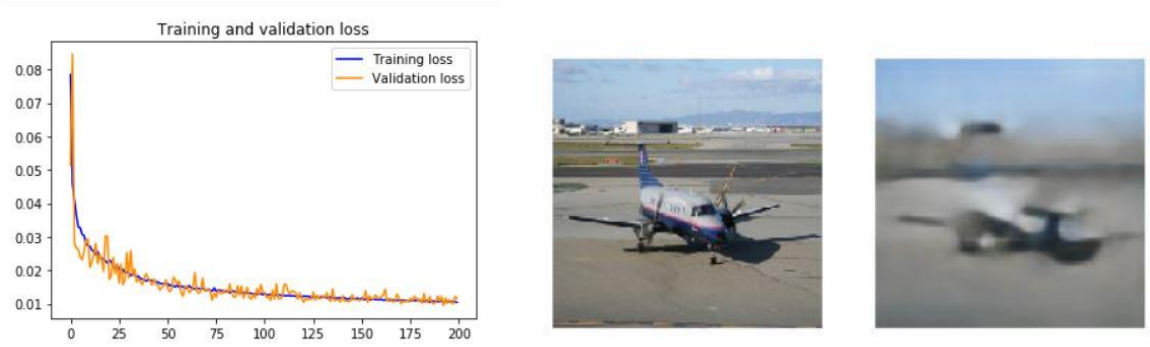


Figure: Training and validation loss diagram (left), Original Image and Reconstructed one (right)

7th Architecture: The encoder part is comprised of six convolutional layers (16->16->pooling-8->8>pooling->8->8->pooling) that downsample the representation from 208×208×3 to 26×26×8. The decoder part (8->8->pooling->8->8->pooling->16->16) performs upsampling operations and brings back the representation to 208×208×3. The activation used was the ReLu function and the number of coding variables occurred is 5408. The training and validation loss are recorded for each experiment as well as a reconstructed image.

Experiment:

- **Loss Function:** Mean Squared Error
- **Optimizer:** RMSprop
- **Epochs: 50**
- **Batch size: 64**

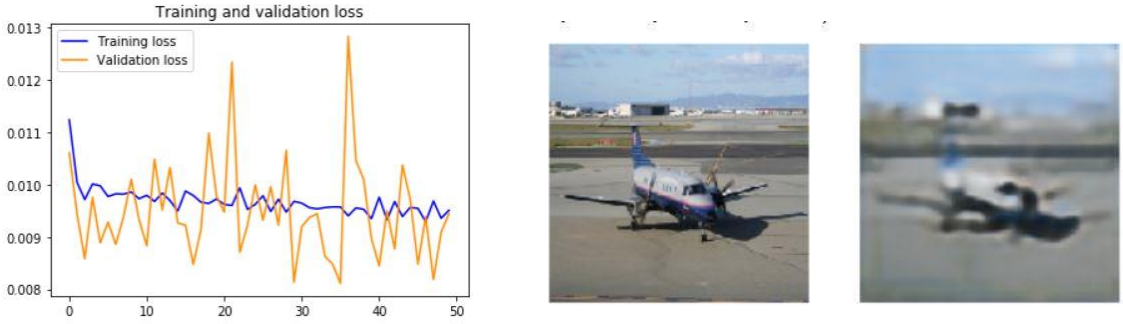


Figure: Training and validation loss diagram (left), Original Image and Reconstructed one (right)

8th Architecture: This architecture is illustrated in the next Figure. The encoder part is comprised of four convolutional layers (256->128->64->32) followed by pooling layers that downsample the representation from 208×208×3 to 26×26×32. The decoder part (32->64->128) performs upsampling operations and brings back the representation to 208×208×3. The activation used was the ReLu function and the number of coding variables occurred is 21632. As loss function was used the mse and the RMSprop optimizer. The training and validation loss are recorded for each experiment as well as a reconstructed image. The batch size was 64 and the number of epochs equal to 50. It is worth mentioning here that this architecture was used later for the segmentation part (and some tests in classification).

Layer (type)	Output Shape	Param #
input_75 (InputLayer)	(None, 208, 208, 3)	0
conv2d_837 (Conv2D)	(None, 208, 208, 256)	7168
max_pooling2d_347 (MaxPoolin	(None, 104, 104, 256)	0
conv2d_838 (Conv2D)	(None, 104, 104, 128)	295040
max_pooling2d_348 (MaxPoolin	(None, 52, 52, 128)	0
conv2d_839 (Conv2D)	(None, 52, 52, 64)	73792
max_pooling2d_349 (MaxPoolin	(None, 26, 26, 64)	0
conv2d_840 (Conv2D)	(None, 26, 26, 32)	18464
conv2d_841 (Conv2D)	(None, 26, 26, 32)	9248
up_sampling2d_343 (UpSamplin	(None, 52, 52, 32)	0
conv2d_842 (Conv2D)	(None, 52, 52, 64)	18496
up_sampling2d_344 (UpSamplin	(None, 104, 104, 64)	0
conv2d_843 (Conv2D)	(None, 104, 104, 128)	73856
up_sampling2d_345 (UpSamplin	(None, 208, 208, 128)	0
conv2d_844 (Conv2D)	(None, 208, 208, 3)	3459

Total params: 499,523
Trainable params: 499,523
Non-trainable params: 0

Figure: Model Architecture

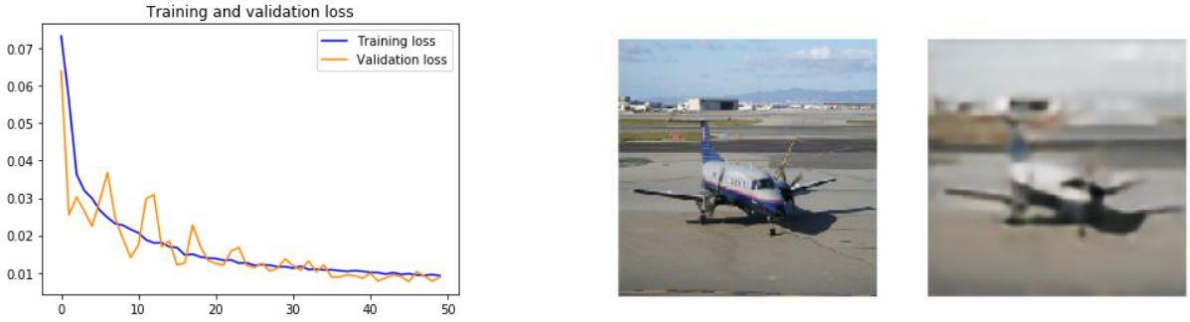


Figure: Training and validation loss diagram (left), Original Image and Reconstructed one (right)

A different approach was used later on, in the next architectures, in order to see how they react when the latent space has a big number of coding variables.

9th Architecture: The encoder part is comprised of 8 convolutional layers (32->32->64->64->128->128->256->256) and the decoder part (128->128->64->64->32->32) performs upsampling operations and brings back the representation to 208×208×3. The activation used was the ReLu function and the number of coding variables occurred is 692224 (small and thick). As loss function was used the mse and the RMSprop optimizer. In this architecture, after each convolution layer was used also a **Batch Normalization layer**. The training and validation loss are recorded for each experiment as well as a reconstructed image. The batch size was 64 and the number of epochs equal to 50.

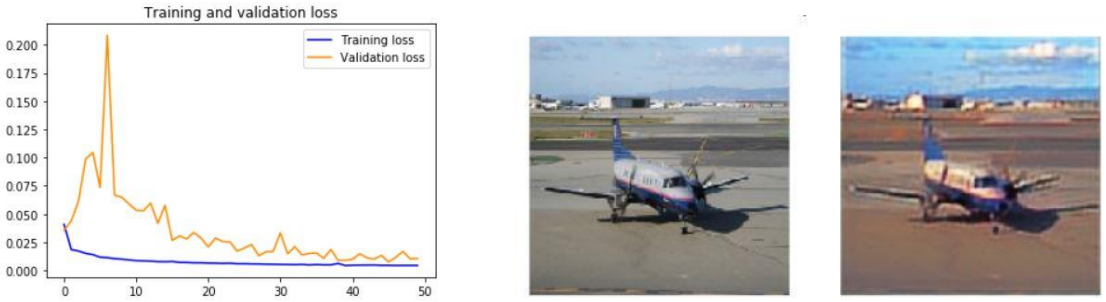


Figure: Training and validation loss diagram (left), Original Image and Reconstructed one (right)

10th Architecture: This architecture is the same with the 9th one, apart from the fact that the Batch Normalization layers were removed (in order to understand somehow how it affects the models).

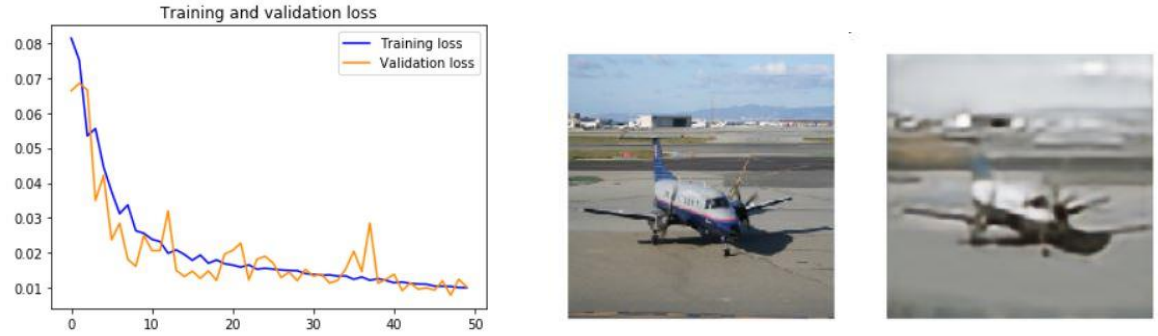


Figure: Training and validation loss diagram (left), Original Image and Reconstructed one (right)

As someone can observe from the previous architectures, the reconstruction is relatively good, but when a Batch normalization layer is used becomes really very good. However, the color totally changes.

11th Architecture: Now, a smaller architecture is tested, using again **Batch Normalization layer** after each convolution layer. The encoder part is comprised of 3 convolutional layers (16->8->8) and the decoder part (8->8->16) followed by pooling layers. The activation used was the ReLu function and the number of coding variables occurred is 5408. As loss function was used the mse and the RMSprop optimizer. The batch size selected was 64 and the number of epochs equal to 50. The training and validation loss are recorded for each experiment as well as a reconstructed image. As someone can observe, the existence of Batch Normalization layers, helps in the reconstruction process either in small or bigger networks. But in any case, the colors change.

Figure: Training and validation loss diagram (left), Original Image and Reconstructed one (right)

Subsequently, some architectures without MaxPooling were tested, which reduce the image size using **strides [2,2]** in some convolutional layers.

12th Architecture: The encoder part is comprised of 8 convolutional layers (32->32->64->64->128->128->256->256) and the decoder part (128->128->64->64->32->32) performs upsampling operations. It was not used Maxpooling here, instead the filter stride was selected equal to 2, in some cases. The activation used was the ReLu function and the number of coding variables occurred is 173056. As loss function was used the mse and the RMSprop optimizer. The training and validation loss are recorded for each experiment as well as a reconstructed image. The batch size was 64 and the number of epochs equal to 50.

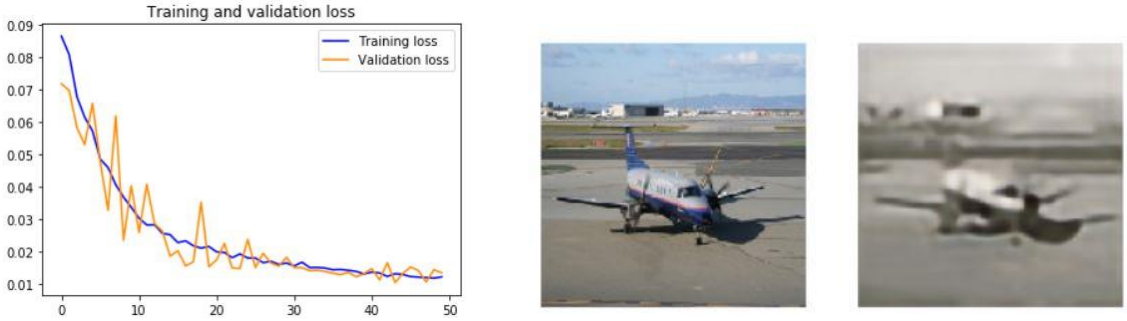


Figure: Training and validation loss diagram (left), Original Image and Reconstructed one (right)

Comparing this architecture with the 10th one, that is the same having though Maxpooling layers and not filter stride=2, this one proves to be less optimal concerning the reconstruction result at least.

13th Architecture: This architecture is the same with the previous one, apart from the fact that the Batch Normalization layers are added (again no MaxPooling is used).

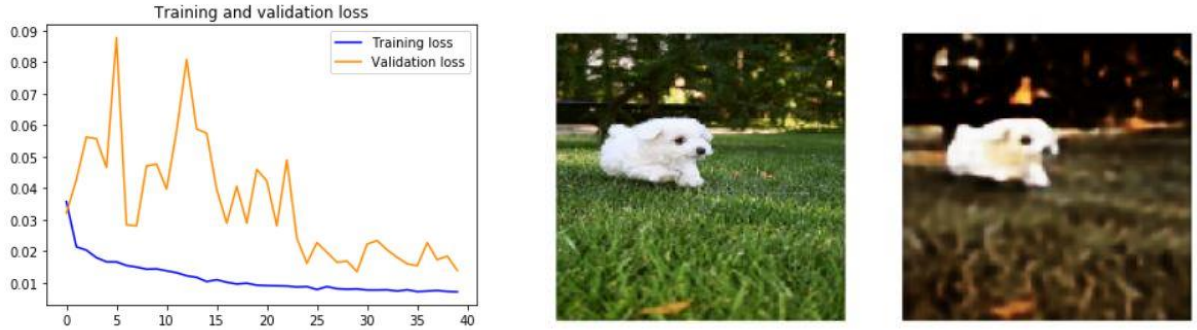


Figure: Training and validation loss diagram (left), Original Image and Reconstructed one (right)

Again, someone can see that Batch Normalization improves the reconstruction. Also, it is worth mentioning that in all cases the existence of Batch Normalization causes big fluctuations in the validation loss function.

Subsequently, some more complex networks were tested in order to measure their performance, too.

14th Architecture: The encoder part is comprised of 10 convolutional layers (32->32->64->64->128->128->256->256) and the decoder part (128->128->64->64->32->32->16->16) performs upsampling operations. Maxpooling was used here as well Batch Normalization. The activation function was the ReLu and the number of coding variables occurred is 173056. As loss function was used the mse and the RMSprop optimizer. The training and validation loss are recorded for each experiment as well as a reconstructed image. The batch size was 64 and the number of epochs equal to 40.

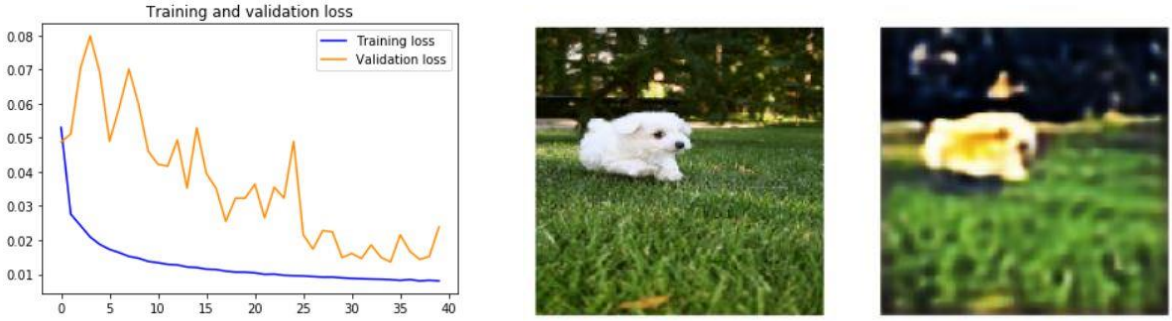


Figure: Training and validation loss diagram (left), Original Image and Reconstructed one (right)

15th Architecture: The encoder part is comprised of 9 convolutional layers (32->32->32->64->64->128->128->256->256) and the decoder part (128->128->64->64->32->32-) performs upsampling operations. Maxpooling was used here as well Batch Normalization. The activation function was the ReLu and the number of coding variables occurred is 692224. As loss function was used the mse and the RMSprop optimizer. The training and validation loss are recorded for each experiment as well as a reconstructed image. The batch size was 64 and the number of epochs equal to 40.

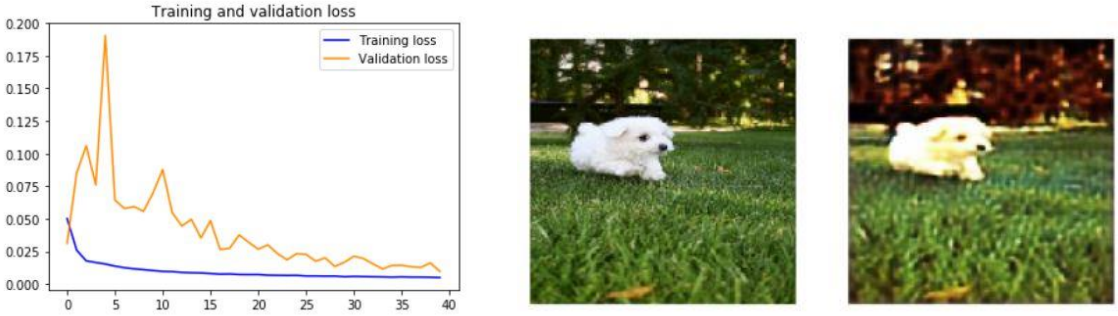


Figure: Training and validation loss diagram (left), Original Image and Reconstructed one (right)

16th Architecture: The next architecture proved to be the best concerning the reconstruction level, however in order to achieve this a vast amount of coding variables was used. Also, in this case the Leaky Relu activation function was used. In more details, the encoder part is comprised of 8 convolutional layers (32->32->32->64->64->128->128->256->256) and the decoder part (128->128->64->64->32->32->) performs upsampling operations. Maxpooling was used here as well Batch Normalization after each activation function. The activation function that was in this case is the Leaky ReLu with parameter alpha=0.01. The number of coding variables occurred is 692224. As loss function was used the mse and the RMSprop optimizer. The training and validation loss are recorded for each experiment as well

as a reconstructed image. The batch size was 64 and the number of epochs equal to 50. This architecture was selected for the classification task, too.

Figure: Training and validation loss diagram (left), Original Images and Reconstructed ones (right)

17th Architecture: The same architecture was tested in this scenario with the difference that in this case the alpha parameter in the Leaky Relu activation function was selected equal to α=0.2.

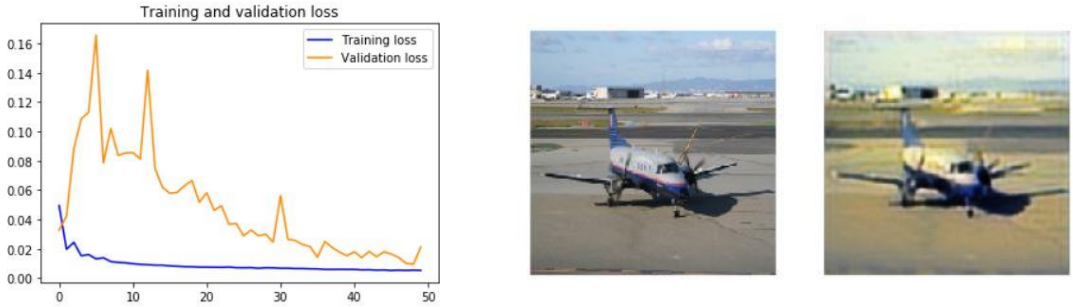


Figure: Training and validation loss diagram (left), Original Image and Reconstructed one (right)

Comparing the results with the previous ones, seem to be worst but still nice.

18th Architecture: The same architecture was tested in this scenario with the difference that in this case the alpha parameter in the Leaky Relu activation function was selected equal to α=0.3.

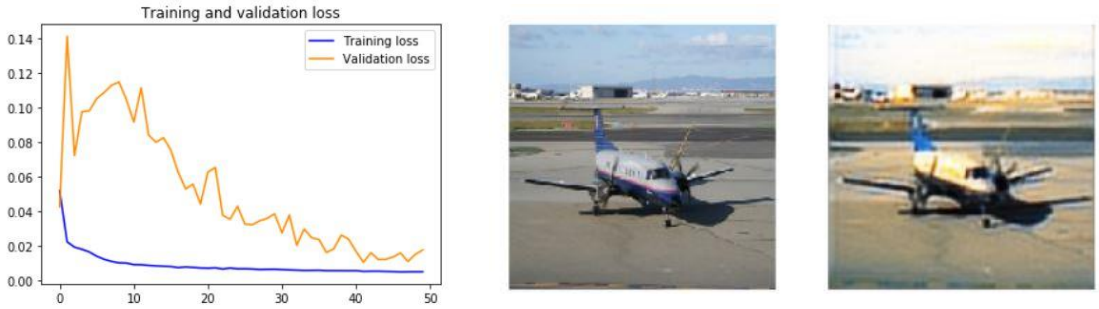


Figure: Training and validation loss diagram (left), Original Image and Reconstructed one (right)

It ends up that the bigger the parameter α, the more outdated the colors.

19th Architecture: Again, the same architecture used (a=0.01) with much more MaxPooling Layers so that the coding variables to be reduced. Like that the coding variables became 43264. Even like that someone can notice that although the reconstruction is not perfect, the characteristics presented in the images are visible in a sense.

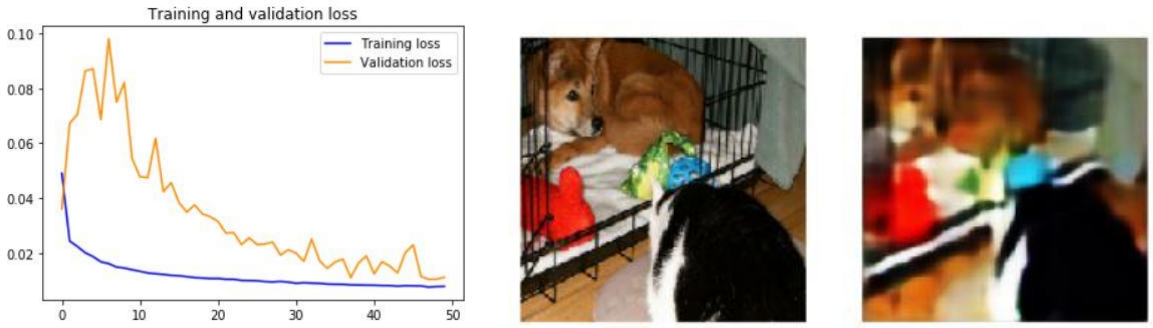


Figure: Training and validation loss diagram (left), Original Image and Reconstructed one (right)

20th Architecture: The same architecture with the 16th one was also tested in this scenario with the difference that the Batch Normalization layer was removed. In this case the alpha parameter in the Leaky Relu activation function is equal to α=0.1. In this case the reconstruction seems to be a bit more blurred.

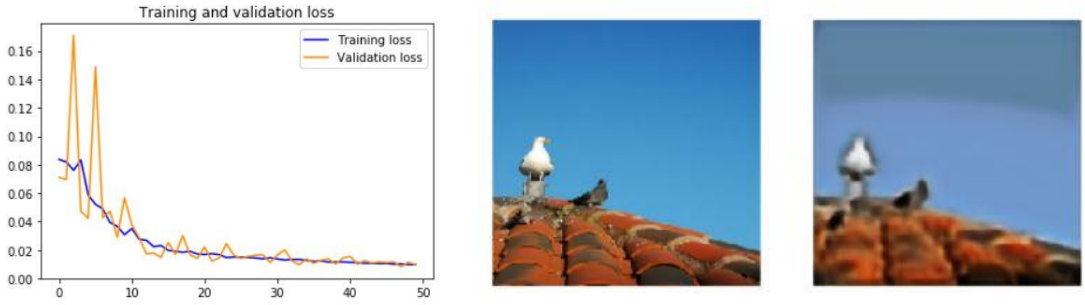


Figure: Training and validation loss diagram (left), Original Image and Reconstructed one (right)

21st Architecture: Finally, a much simpler architecture was tested for the case where Leaky Relu (α=0.01) is used. The encoder part is comprised of 6 convolutional layers (16->16->16->8->8->8) followed by max pooling layers. The decoder part (8->8->8->16->16) performs upsampling operations and brings back the representation to 208×208×3. The number of coding variables occurred is 5408. The Batch size was selected equal to 64 and the epochs equal to 50. There was no use of Batch Normalization at all. The network in this scenario seemed to be probably over simplistic in order to learn the features.

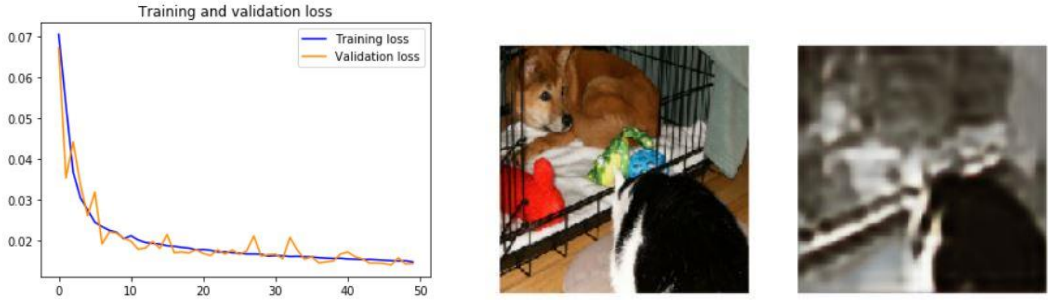


Figure: Training and validation loss diagram (left), Original Image and Reconstructed one (right)

In conclusion, since the number of the available training data is not satisfactory it proves that more large networks quickly lead to overfitting and that light-weight networks are preferable in this case. Also, the existence of Batch Normalization layers is crucial for the reconstruction performance, and of course the more the coding variables the best the reconstruction. Both Relu and Leaky Relu are good activations functions, the selection of parameters α though in the second case affects in a way the results. Moreover, the mean absolute error achieves better results, however, proved to be slower in the architectures presented, that's why the mean squared function was mainly used. Furthermore, when the Batch normalization was introduced in the networks, strong fluctuations were observed in the validation loss graph, especially in the beginning of the training process. Finally, max pooling was preferred instead of a convolutional filter of stride 2 and of course the positions that were selected for the max pooling layers in the networks, also affected the results.

4. Classification

In this section, a simple logistic regression (classification task) is taking place. The output of the encoder of an autoencoder architecture selected from the ones represented before, is used as input to a fully connected layer. As a first task the encoding part of the auto encoder is frozen, and a fully connected layer is added at the end (a technique called fine tuning). As a second task, the same network architecture is used but all parameters are trained from scratch. The results of these different scenarios are presented subsequently. The image size that was selected for this task is 208x208.

Loss Functions used:

- **Categorical Cross Entropy**: Categorical cross entropy is a loss function that is used for single label categorization. This is when only one category is applicable for each data point. In other words, an example can belong to one class only (multi-class classification). It is perfectly combined with the Softmax activation function.
- **Binary Cross Entropy:** Also called Sigmoid Cross-Entropy loss. It is a Sigmoid activation plus a Cross-Entropy loss. Perfectly used for multi-label classification tasks, where many labels can be assigned to each object and it is combined with the Sigmoid activation functions for better results.

Since the provided data for this classification task are multi-labeled data, we expect the use of the binary cross entropy loss function, combined with the sigmoid activation function, to provide better results for the classifier than that of the categorical cross entropy.

Activation Functions used:

- Softmax
- Sigmoid

Optimizer:

- RMSprop
- Adam: Adam is currently recommended as the default algorithm to use, and often works slightly better than RMSProp.

First Architecture: The architecture that was initially tested is the 16th architecture described in section 3 and was tested under different scenarios. These scenarios are presented subsequently as well as the results of the classification.

Freeze the encoder Layers:

1st Experiment:

- **Loss function**: categorical cross entropy
- **Activation**: Softmax
- **Optimizer:** Adam

Figure: Classifier Results

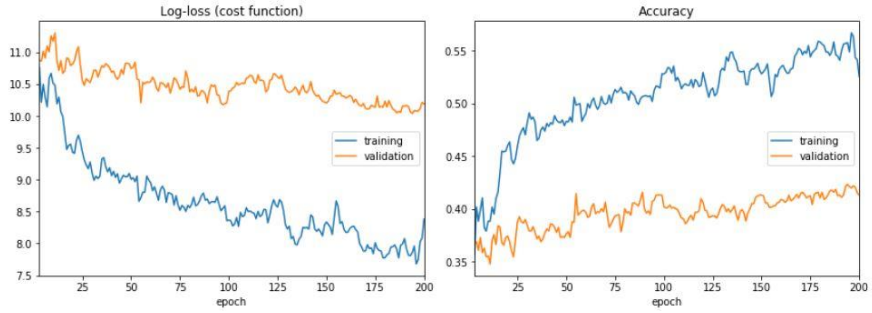


Figure: Cost Function (left), Accuracy (right) of the classifier

2nd Experiment:

- **Loss function:** categorical cross entropy
- **Activation:** Softmax
- **Optimizer:** RMSprop

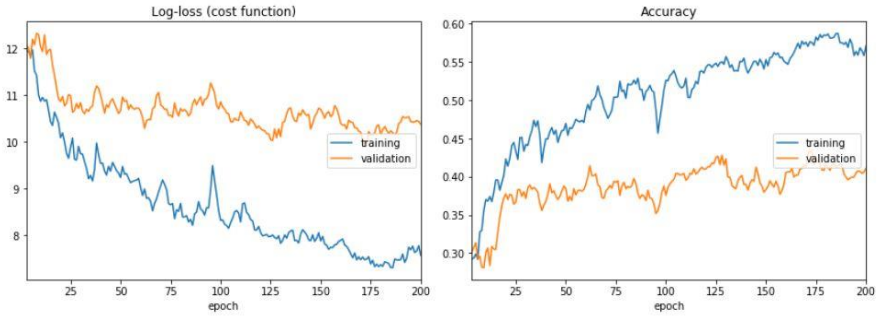


Figure: Cost Function (left), Accuracy (right) of the classifier

Almost the same behavior is observed between the first and second experiment, where categorical cross entropy and Softmax activation function are used in both examples with different optimizers used though in each case. In both examples, the performance of the classifier is bad, with the validation accuracy reaching slightly above the 40%.

3rd Experiment:

- **Loss:** binary cross entropy
- **Activation**: sigmoid
- **Optimizer**: Adam

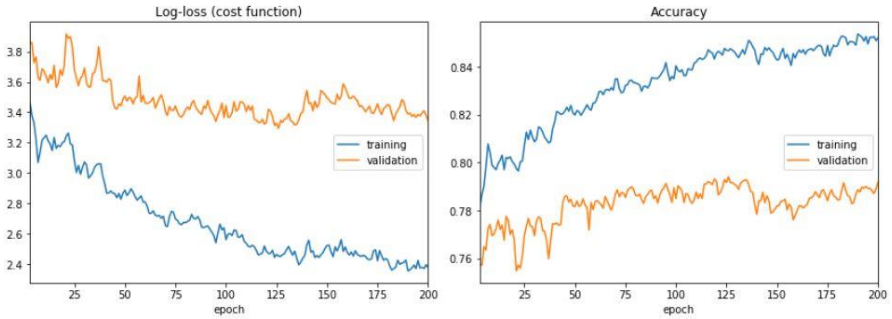


Figure: Cost Function (left), Accuracy (right) of the classifier

4th Experiment:

- **Loss:** binary cross entropy
- **Activation:** sigmoid
- **Optimizer:** RMSprop

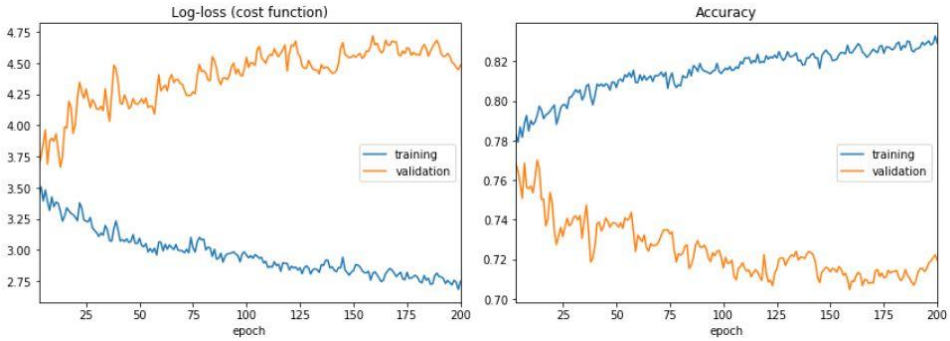


Figure: Cost Function (left), Accuracy (right) of the classifier

In both these two last scenarios, the performance has noted great improvement, with the case of using Adam optimizer to be a better option, since better accuracy is achieved in this case.

Retrain the encoder Layers:

1st Experiment: In this scenario, the trained process early stopped in order to avoid overfitting. Found 307 correct labels and 1163 incorrect.

- Loss function: categorical cross entropy
- Activation: Softmax
- Optimizer: Adam

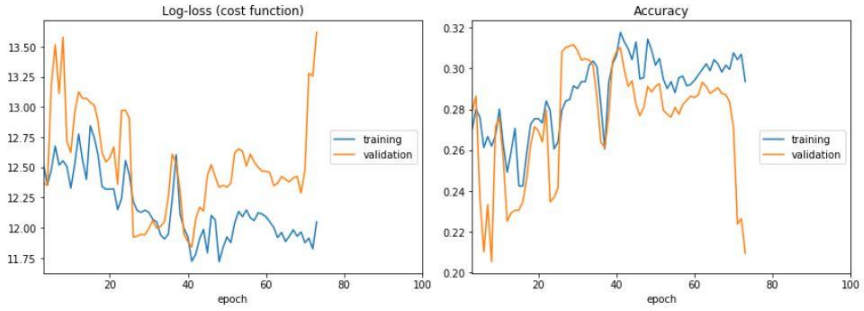


Figure: Cost Function (left), Accuracy (right) of the classifier

2nd Experiment: In this scenario, again the trained process early stopped in order to avoid overfitting.

- Loss function: Binary cross entropy
- Activation: Sigmoid
- Optimizer: Adam

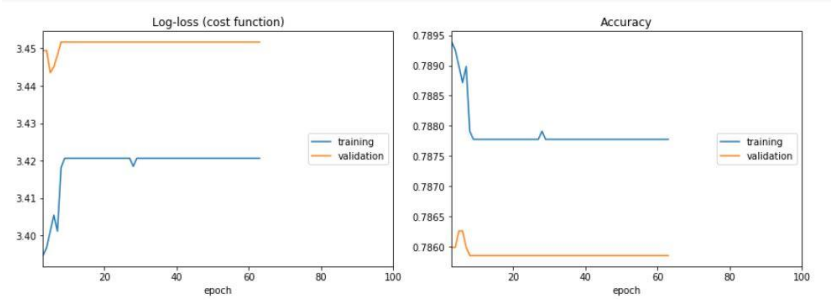


Figure: Cost Function (left), Accuracy (right) of the classifier

Again, comparing the previous two scenarios, binary cross entropy combined with sigmoid activation function, seems to be a better option for the classifier. The loss/accuracy seem to converge really fast in this second case, giving the impression though of overfitting.

Second Architecture: Subsequently, the architecture that was tested is the 8th architecture described in section 3 (with fewer coding variables) and was tested under different scenarios. These scenarios are presented subsequently as well as the results of the classification.

Freeze the encoder Layers:

1st Experiment:

- Loss function: Binary cross entropy
- Activation: Sigmoid
- Optimizer: Adam

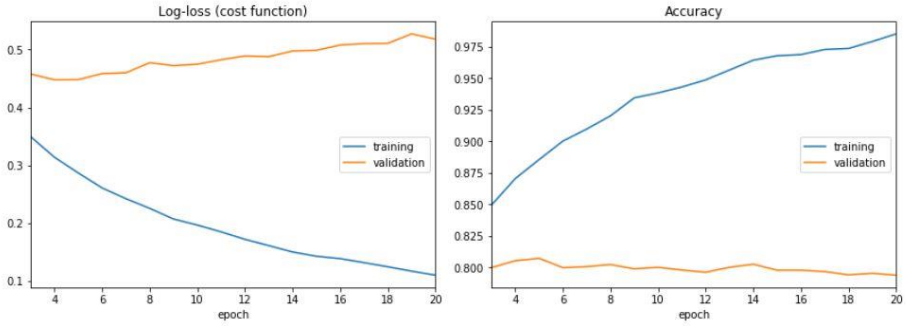
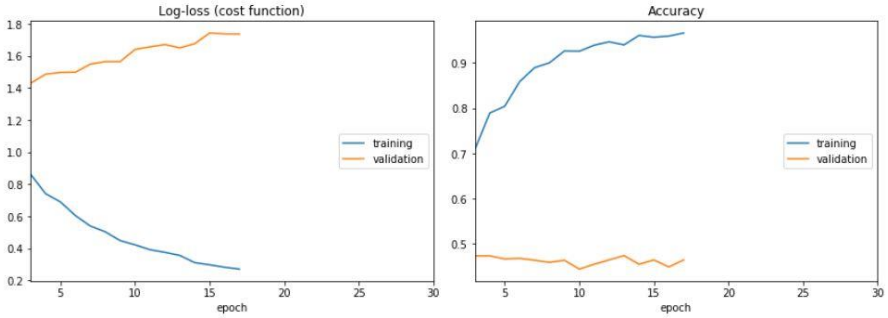


Figure: Cost Function (left), Accuracy (right) of the classifier

2nd Experiment: In this scenario, the training process early stopped in order to avoid overfitting.

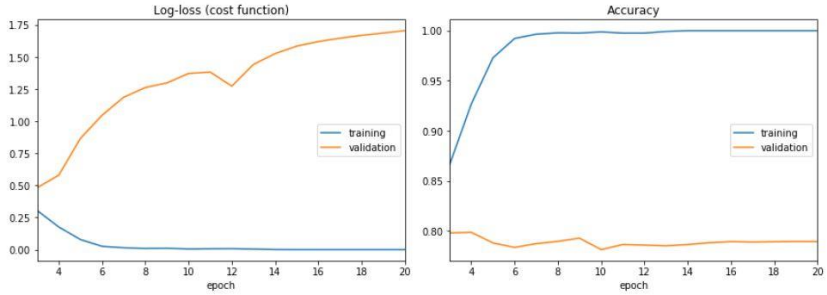
- Loss function: Categorical cross entropy
- Activation: Softmax
- Optimizer: Adam



Retrain the encoder Layers:

1st Experiment:

- Loss function: Binary cross entropy
- Activation: Sigmoid
- Optimizer: Adam



In these scenarios, the results do not seem to be the expected ones. Comparing the case of retraining the encoder layers with that of freezing the layers, perfect results are recorded for the training data as expected. However, in all the scenarios the models seem to get overfitted, already from the first epochs, as it is figured out in the validation plots where the loss function increases instead of decreasing. An explanation probably, is that the encoder was already trained more than needed, and a result got easily overfitted.

5. Segmentation

In this section, a segmentation CNN is constructed in order to extract the foreground from an image. This segmentation CNN can be regarded as a binary classification CNN, assigning every pixel in the image to a label. Each pixel belonging to the foreground is classified to label 1 and every pixel belonging to the background is classified to label 0 (semantic segmentation). For this task the 16th autoencoder, presented in section 3, having as output one channel image though. Moreover, different loss functions are used in this case and apart from the accuracy a different range of metrics is calculated in this segmentation task, too. Also, since the numbers of data was not sufficient data augmentation was applied using rotation techniques, so that increase the training images. However, it was noticed that when all of them were used, the network was overfitting even from the second run, so usually a part of them was used (either 1100 or 1200 training images). Finally, the image size was reduced to 144x144 because the training was slower otherwise.

Optimizer: **Adam** is currently recommended as the default algorithm to use, and often works slightly better than RMSProp. Adam can be looked at as a combination of RMSprop and Stochastic Gradient Descent with momentum. It uses the squared gradients to scale the learning rate like RMSprop and it takes advantage of momentum by using moving average of the gradient instead of gradient itself like SGD with momentum.

Loss Functions:

- **Binary Cross Entropy:** Since the segmentation task is treated as a binary classification task, binary cross entropy is good choice.

- **Dice Loss:** Another popular loss function for image segmentation tasks is based on the Dice coefficient, which is essentially a measure of overlap between two samples. This measure ranges from 0 to 1 where a Dice coefficient of 1 denotes perfect and complete overlap. The Dice coefficient was originally developed for binary data, and can be calculated as:

$$Dice = \frac{2\,|A \cap B|}{|A| + |B|}$$

  where |A∩B| represents the common elements between sets A and B, and |A| represents the number of elements in set A (and likewise for set B).

- **Weighted Cross Entropy:** It is a variant of cross entropy where all positive examples get weighted by some coefficient. It is used in the case of class imbalance. For example, when you have an image with 10% black pixels and 90% white pixels, regular cross entropy won't work very well. In order to decrease the number of false negatives, β should be > 1. To decrease the number of false positives, β < 1.

Metrics Calculated:

- **Intersection over Union (IoU):** Intersection over union also referred to as the Jaccard index, is essentially a method to quantify the percent overlap between the target mask and the prediction output. This metric is closely related to the Dice coefficient. Quite simply, the IoU metric measures the number of pixels common between the target and prediction masks divided by the total number of pixels present across both masks. The IoU score is calculated for each class separately and then averaged over all classes to provide a global, mean IoU score of our semantic segmentation prediction.
- **Dice Score:** Dice score also referred to as F- score is an important image overlap metric, especially in (medical) image segmentation. Also, it is worth mentioning here that IoU and F-score are always within a factor of 2 of each other: $F/2 \leq IoU \leq F$, and also that they meet at the extremes of one and zero under the conditions that someone would expect (perfect match and completely disjoint).
- **Receiver Operating Characteristic (ROC):** AUC - ROC curve is a performance measurement for classification problem at various thresholds settings. ROC is a probability curve and AUC represents the degree or measure of separability. It tells how much model is capable of distinguishing between classes. Was selected, since the task is treated as a binary classification problem.

Regularization: In some scenarios, dropout layers are added with a 0.5 probability, that can be regarded as a regularization step.

The network that was selected for this task is the same as in the case of the classification task and is illustrated in the next figure. The encoder part is comprised of four convolutional layers (256->128->64->32) followed by pooling layers that downsample the representation from 208×208×3 to 26×26×32. The decoder part (32->64->128) performs upsampling operations and transforms the representation to 208×208×1, one channel in this case. A series of experiments took place and the results are presented subsequently. In all the experiments the Adam optimizer is used.

Layer (type)	Output Shape	Param #
input_4 (InputLayer)	(None, 144, 144, 3)	0
conv2d_41 (Conv2D)	(None, 144, 144, 256)	7168
max_pooling2d_16 (MaxPooling	(None, 72, 72, 256)	0
conv2d_42 (Conv2D)	(None, 72, 72, 128)	295040
max_pooling2d_17 (MaxPooling	(None, 36, 36, 128)	0
conv2d_43 (Conv2D)	(None, 36, 36, 64)	73792
max_pooling2d_18 (MaxPooling	(None, 18, 18, 64)	0
conv2d_44 (Conv2D)	(None, 18, 18, 32)	18464
conv2d_45 (Conv2D)	(None, 18, 18, 32)	9248
up_sampling2d_16 (UpSampling	(None, 36, 36, 32)	0
conv2d_46 (Conv2D)	(None, 36, 36, 64)	18496
up_sampling2d_17 (UpSampling	(None, 72, 72, 64)	0
conv2d_47 (Conv2D)	(None, 72, 72, 128)	73856
up_sampling2d_18 (UpSampling	(None, 144, 144, 128)	0
conv2d_48 (Conv2D)	(None, 144, 144, 1)	1153

Total params: 497,217
Trainable params: 497,217
Non-trainable params: 0

Figure: Model architecture for segmentation task

1st Experiment:

- **Without Data Augmentation**
- **Loss:** Binary Cross Entropy
- **Epochs:** 40
- **Batch size:** 32

In the following figures, someone can see the masks occurred after training and testing the neural network. In the left, the initial reconstruction is visible (slightly gray) and in the right the one acquired after applying a threshold.

Figure: Segmentation results before and after threshold

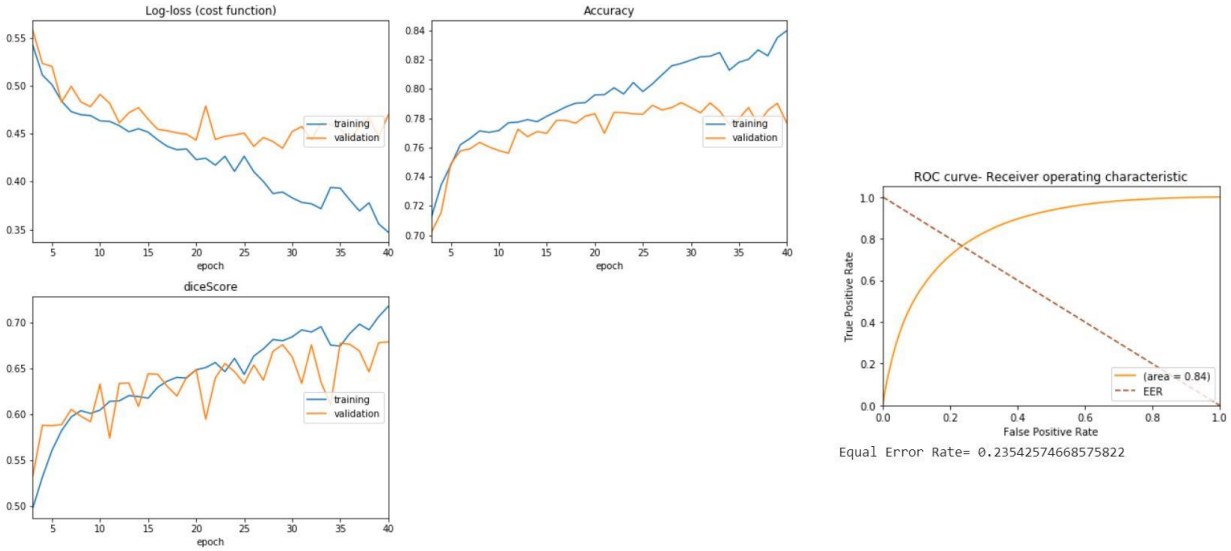


Figure: Results of various metrics

The results that occurred are good enough, achieving an area under curve of 84% for the validation dataset.

2nd Experiment: The same logic with the previous one, but data augmentation is included in this case.

- **Data Augmentation (1000 train samples)** (Overfitting was observed when more were tested)
- **Loss:** Binary Cross Entropy
- **Epochs:** 40
- **Batch size:** 32

The results of this experiment are illustrated subsequently.

Figure: Expected mask (left), mask occurred after the segmentation (right)

Figure: Expected mask (left), mask that occurred after the segmentation, represented with after a threshold (right)

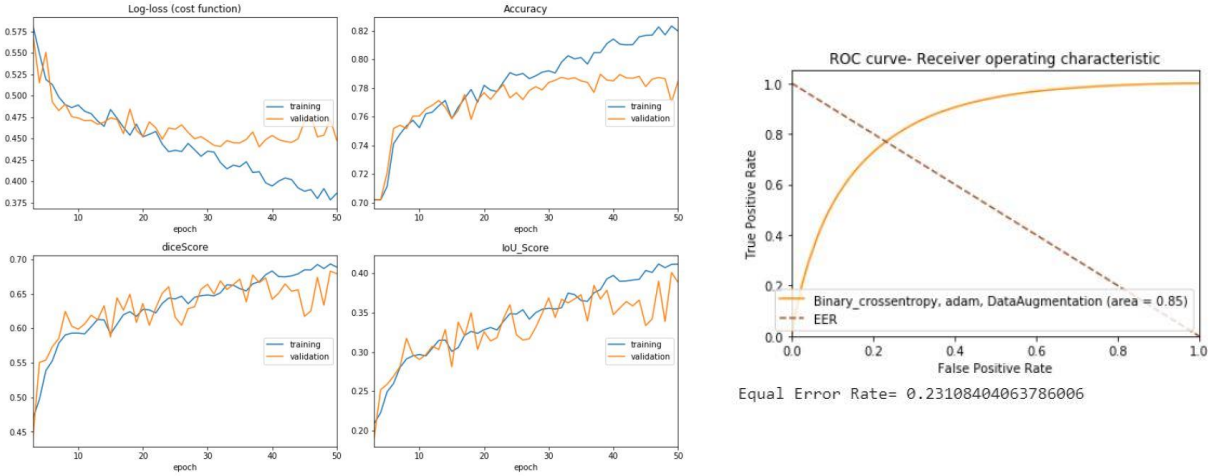


Figure: Results of various metrics

The results that occurred are good enough, achieving an area under curve of 85% on the validation dataset.

3rd Experiment:

- **Without Data Augmentation**
- **Loss:** Dice Loss
- **Epochs:** 50
- **Batch size:** 64

Figure: Segmentation results before and after threshold

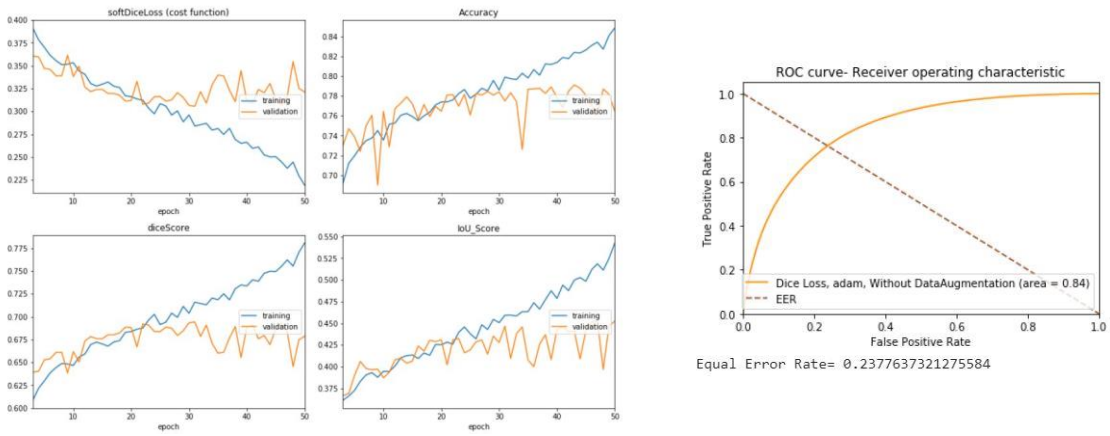


Figure: Results of various metrics

The results that occurred are quite good, achieving an area under curve of 84% for the validation dataset. As a whole, they seem a bit better though than the ones from the previous experiments.

4th Experiment:

- **With Data Augmentation:** 1100 training images used. Overfitting with all the data from augmentation.
- **Loss:** Dice Loss
- **Epochs:** 50
- **Batch size:** 64

Figure: Segmentation results before and after threshold

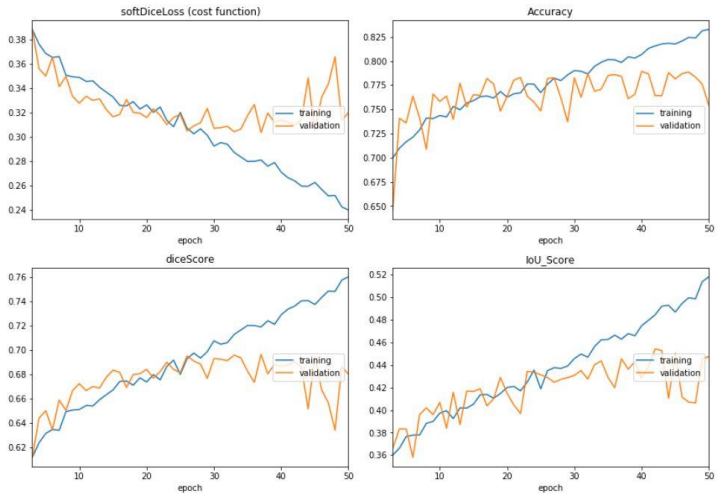


Figure: Results of various metrics

In this scenario slightly better results are achieved compared to the previous ones, where no data augmentation was achieved.

5th Experiment:

- **With Data Augmentation:** 1200 training images used. Overfitting with all the data from augmentation.
- **Loss:** Weighted Cross Entropy (β=3)
- **Epochs:** 50
- **Batch size:** 64

In this scenario, the training was stopped earlier in order to avoid overfitting. As someone observe, the results are moderate with the segmented masks be a bit different from the expected ones. However, after applying thresholding, the segmented images look better.

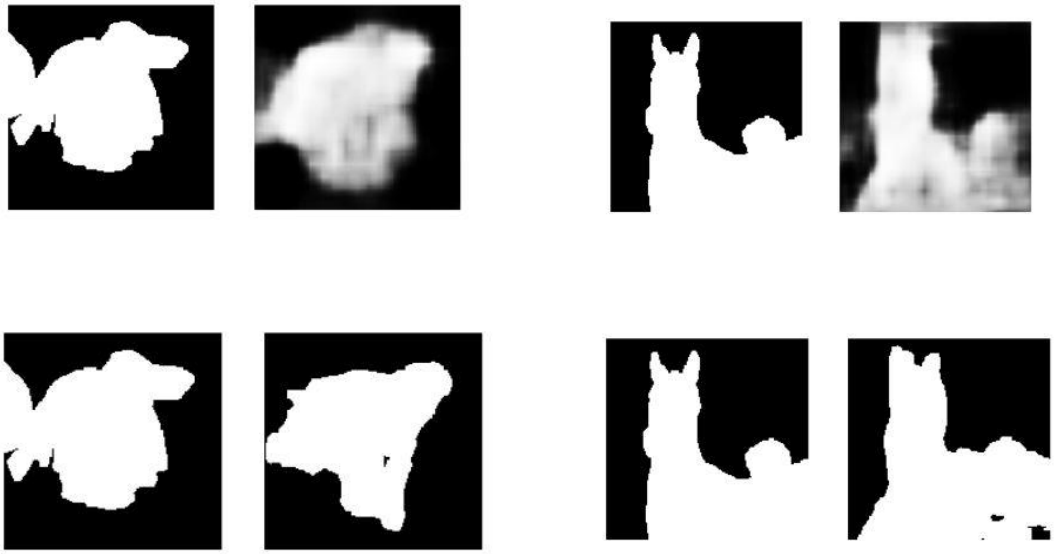


Figure: Various Segmentation results before and after threshold

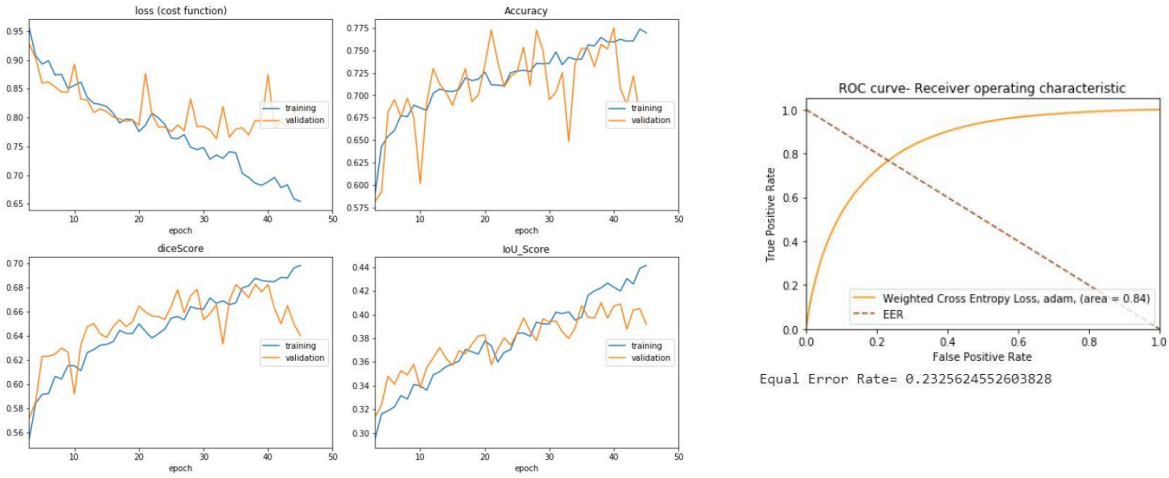


Figure: Results of various metrics

In conclusion, many fluctuations are observed in the metric networks. The network performance is quite good, with some of the segmented images being slightly different, though, from the expected ones. However, after thresholding, the segmented images are improved. Also, we have to mention that the network here could easily get overfitted, even for small iterations and as result there were times that enforced interruption was used.

References:

1. https://towardsdatascience.com/dimensionality-reduction-does-pca-really-improve-classification-outcome-6e9ba21f0a32
2. https://towardsdatascience.com/an-approach-to-choosing-the-number-of-components-in-a-principal-component-analysis-pca-3b9f3d6e73fe
3. https://www.r-bloggers.com/pca-vs-autoencoders-for-dimensionality-reduction/
4. http://alexhwilliams.info/itsneuronalblog/2016/03/27/pca/?fbclid=IwAR0KZgX8lkeHum3WPrC0kRTEyILmO18STZncUZLh8eYplsAooZgEk61FTBI
5. https://adeshpande3.github.io/adeshpande3.github.io/A-Beginner's-Guide-To-Understanding-Convolutional-Neural-Networks/?fbclid=IwAR3KNJmBOeqhnnYYUY541skfcWefeSernhtpB_M5Wz2qwF45KgXYfLZZ0ok
6. https://blog.keras.io/building-autoencoders-in-keras.html
7. https://keras.io/optimizers/
8. https://adeshpande3.github.io/A-Beginner%27s-Guide-To-Understanding-Convolutional-Neural-Networks-Part-2/
9. https://towardsdatascience.com/activation-functions-and-its-types-which-is-better-a9a5310cc8f
10. https://jhui.github.io/2017/03/16/CNN-Convolutional-neural-network/
11. https://medium.com/@iamvarman/how-to-calculate-the-number-of-parameters-in-the-cnn-5bd55364d7ca

12. https://medium.com/human-in-a-machine-world/mae-and-rmse-which-metric-is-better-e60ac3bde13d
13. https://ml-cheatsheet.readthedocs.io/en/latest/loss_functions.html
14. https://gombru.github.io/2018/05/23/cross_entropy_loss/
15. https://lars76.github.io/neural-networks/object-detection/losses-for-segmentation/
16. https://towardsdatascience.com/understanding-binary-cross-entropy-log-loss-a-visual-explanation-a3ac6025181a
17. https://towardsdatascience.com/how-to-select-the-right-evaluation-metric-for-machine-learning-models-part-1-regrression-metrics-3606e25beae0
18. https://towardsdatascience.com/a-look-at-gradient-descent-and-rmsprop-optimizers-f77d483ef08b
19. http://ruder.io/optimizing-gradient-descent/
20. https://towardsdatascience.com/semantic-segmentation-popular-architectures-dff0a75f39d0
21. https://nanonets.com/blog/how-to-do-semantic-segmentation-using-deep-learning/
22. https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/
23. https://www.jeremyjordan.me/semantic-segmentation/#loss
24. https://www.jeremyjordan.me/evaluating-image-segmentation-models/
25. https://www.datacamp.com/community/tutorials/autoencoder-classifier-python?fbclid=IwAR1rdwM1JxsryB6R9Imk-Lu72uJ_TR1W2OZOSRyd_-iZiGGYLWbeDqkOUrc
26. https://keras.io/initializers/
27. https://keras.io/activations/
28. https://www.dlology.com/blog/how-to-choose-last-layer-activation-and-loss-function/
29. https://glassboxmedicine.com/2019/05/26/classification-sigmoid-vs-softmax/