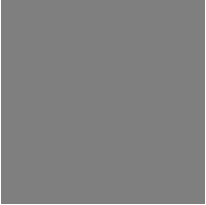
**Methods/Network architecture**

The network we used is an AutoEncoder-like network. Which means it has an encoder, a narrow bottleneck and a decoder part. The difference is in the input and output. In an AE network we usually use the same data as input and output and goal is to compress the data. In our case we give the grayscale image as input and the a and b channels of the colorized pictures as the output. So not just the input and output are different, but moreover their shapes are also dissimilar.

Firstly let’s take a look at the input, which is a 256x256x1 sized matrix. It means that it has 256x256 pixels and only 1 channel.

1

256

256

It goes through the **encoder** first which was built from several convolutional layers:

1. input: 1@256x256

filter: 3x3 @ 64 (3x3 sized filter with 64 depth)

stride: 2x2

padding: 1 column and 1 row 0 padding at the end (in keras padding=“same”)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| x | x | … | x | 0 |
| x | x | … | x | 0 |
| … |  |  |  |  |
| x | x | … | x | 0 |
| 0 | 0 | 0 | 0 | 0 |

output: (256-3+1)/2 +1 x (256-3+1)/2 +1 => 64@128 x 128

1. input: 64@128 x 128

filter: 3x3@128 (3x3 sized filter with 64 depth)

stride: 1x1

padding: (in keras padding = “same”)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 0 | … | 0 | 0 |
| 0 | x | … | x | 0 |
| … |  |  |  |  |
| 0 | x | … | x | 0 |
| 0 | 0 | 0 | 0 | 0 |

output: (128-3+2\*1)/1+1 x (128-3+2\*1)/1+1 => 128@128x128

1. input: 128@128x128

filter: 3x3@128 (3x3 sized filter with 128 depth)

stride: 2x2

padding: same as in the 1) CNN layer: 1 column and 1 row 0 padding at the end (in keras padding=“same”)

output: (128-3+1)/2+1 x (128-3+1)/2+1 => 128@64x64

1. input: 128@64x64

filter: 3x3@256

stride: 1x1

padding: same as in the 2) CNN layer

output: (64-3+2\*1)/1+1 x (64-3+2\*1)/1+1 => 256@64x64

1. input: 256@64x64

filter: 3x3@256

stride: 2x2

padding: same as in the 1) CNN layer: 1 column and 1 row 0 padding at the end (in keras padding=“same”)

output: (64-3+1)/1+1 x (64-3+1)/2+1 => 256@32x32

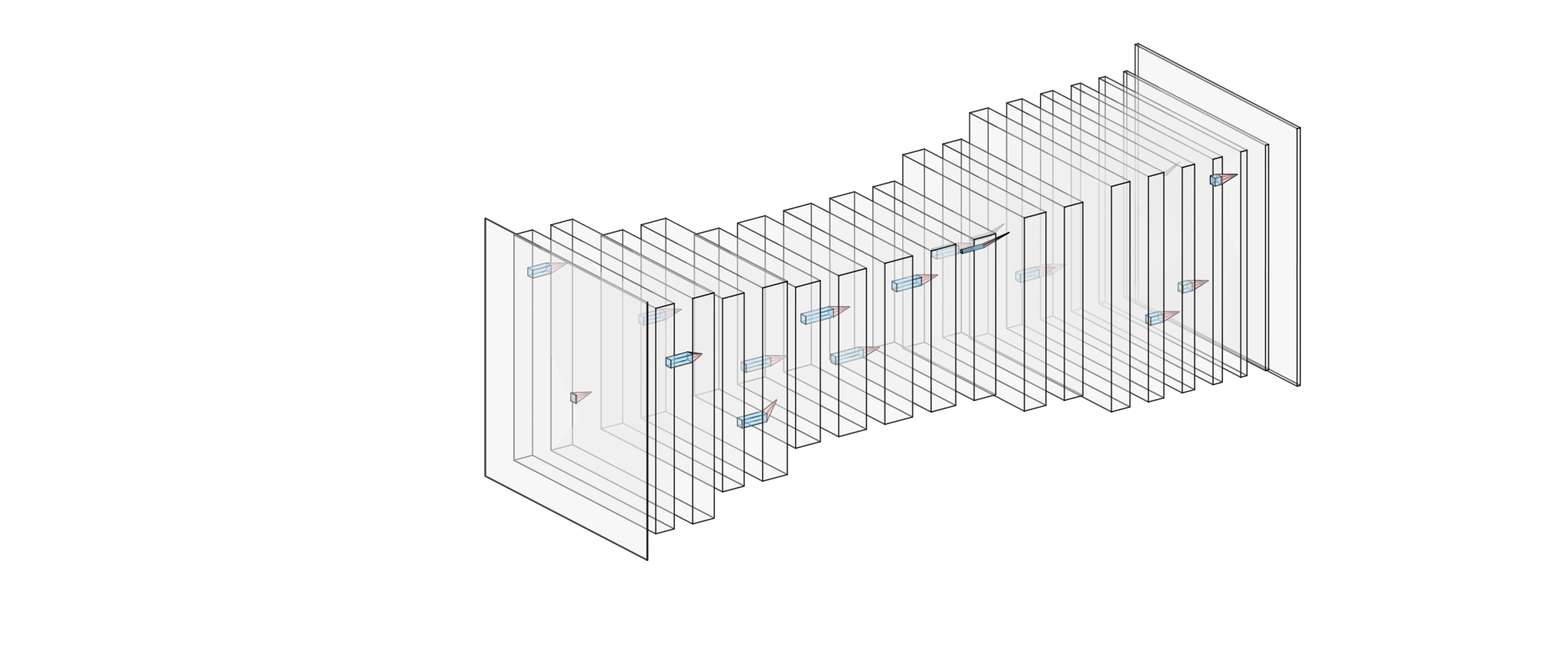
The “**bottleneck**”: 2 same convolutional layers

6)-7) filter = 3x3@512, stride = 1x1, padding: same as in the 2)CNN layer

output of the bottleneck: 512@32x32

The **decoder** network:

The task of the decoder part is to make this 512@32x32 tensor to fit the desired output dimensions. To achieve the height and width we use Keras UpSampling2D layers, which repeats the rows and columns of the data by the given size respectively. (in our case we used size = (2,2) in every UpSampling2D layer) It uses the interpolation what’s given as a parameter, we used the default nearest interpolation method. The depth of the output is 2 (the a and b color channels), to make our data the wished depth we use several convolutional layers. This part is almost symmetrical to the encoder part, since we half the depth with each layer. (opposite to the encoder where we double the depth with each layer)



Activations

At the beginning we used the ReLu function as activation in all the layers except the last convolutional layer in the decoder part, where we used tanh since we wanted to get a result between -1 and 1. (the standardized values for a and b channel)

When we heard about the ‘swish’ activation on lecture we did a small research about it. We found out that it was fairly similar to the ReLu function but it outperformed ReLu most of the time. So we decided to implement and try swish. (in the teaching part where we write about optimization you can find the details)

The swish function: *f(x) = x\*sigmoid(x)*

