TDT4265 - A2

Martin Madsen, Sondre A Bergum

February 28, 2019

1 Convolutional Neural Networks

1.1 a

Implemented the neural network specified in the assignment text. This was done by creating a new class LeNet based on the handed out ExampleModel. First we added the specified `'nn.Conv2d(...)'`, `'nn.ReLu()'` and `'nn.MaxPool2d(...)'` to the arguments of `'self.feature_extractor=nn.Sequential(...)'`. Then we added flatten between the feature extractor and the classifier in the forward function and densing in the classifier with `'nn.Linear(...)'`. The code can be found in `'task_1.py'` in the handed in code. The final losses can be found in fig. 1. The lowest loss was obtained in the 6th epoch with a validation loss of 0.8024.

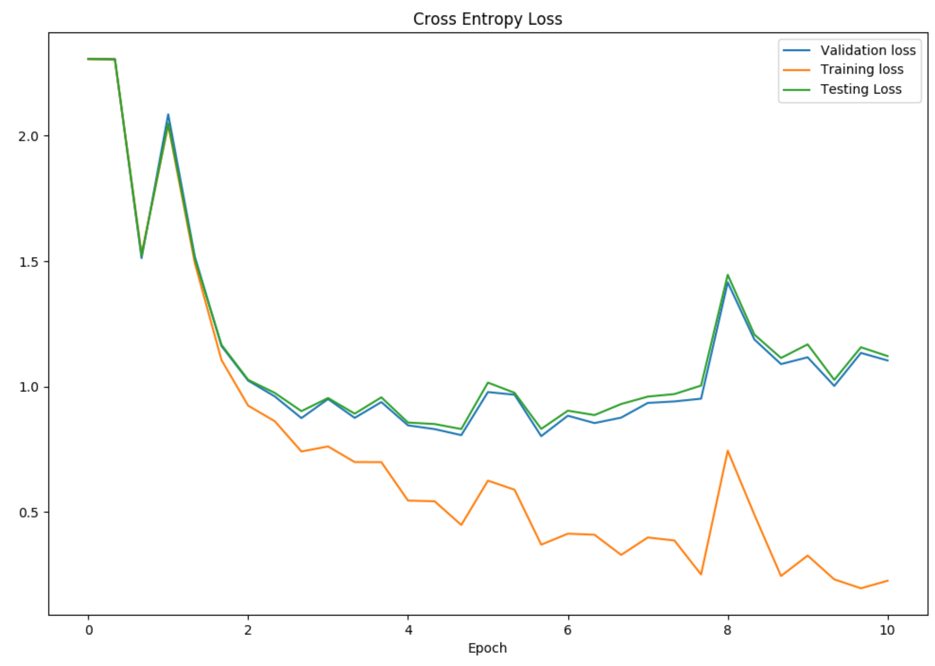

Figure 1: Losses for task 1

1.2 b

Accuracy for this task can be seen in table 1 and fig. 2.

Training accuracy	Validation Accuracy	Test Accuracy
0.9233	0.7092	0.7149

Table 1: Final accuracy achieved after 10 epochs.

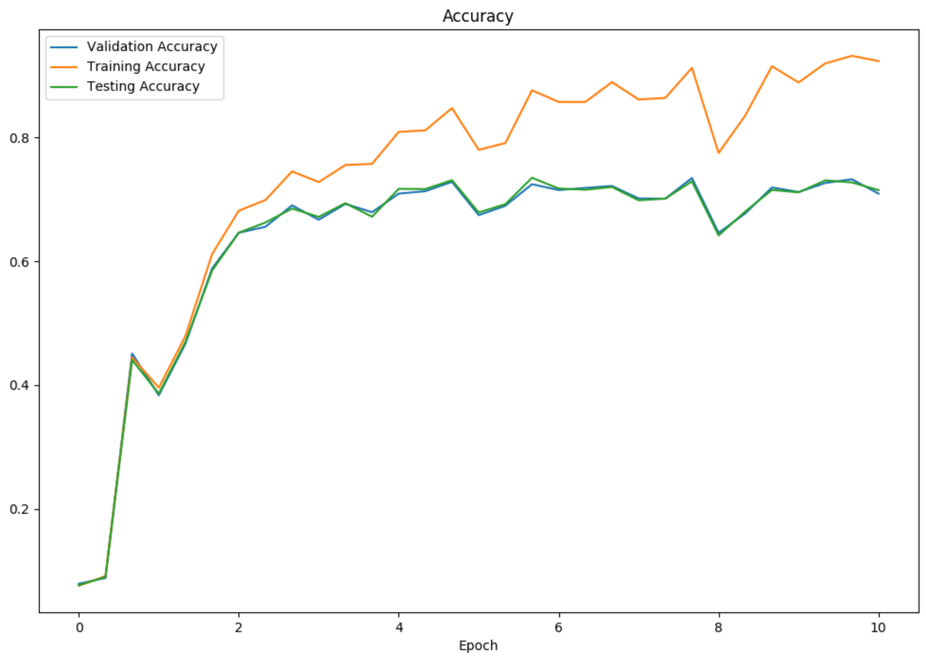

Figure 2: Accuracy for task 1

1.3 c

The number of parameters in our network are summarized below. Each line in the equation is one layer. The final number is 390400.

$$F_{H1} \times F_{W1} \times C_1 \times C_2 + C_2 = 5 \times 5 \times 3 \times 32 + 32 = 2432 \quad (1)$$
$$F_{H3} \times F_{W3} \times C_3 \times C_4 + C_4 = 5 \times 5 \times 32 \times 64 + 64 = 51264 \quad (2)$$
$$F_{H5} \times F_{W5} \times C_5 \times C_6 + C_6 = 5 \times 5 \times 64 \times 128 + 128 = 204928 \quad (3)$$
$$(H_7 \times W_7 + 1) \times C_7 \times D_1 = ((4 \times 4) \times (128) + 1) \times (64) = 131136 \quad (4)$$
$$(D_1) \times (D_2) = (64) \times (10) = 640 \quad (5)$$
$$\sum = 390400 \quad (6)$$

2 Deep Convolutional Network for Image Classication

2.1 a)

2.1.1 First Network

This network has the suggested structure of `(conv-relu)(conv-relu-conv-relu-pool)x2 (affine)x2 softmax`, with some added tricks. At the start we use dropout with 10% probability. Then we use Batch Norm before each pooling. The network architecture is summarized in table 2. The Adam optimizer is used for calculating loss with an initial learning rate of 0.001. All weights in the network are initialized from a Xavier distribution.

2.1.2 Second Network

Here we use the same structure as LeNet, but include some tricks. We use dropout to disable 10% of the first convolutional layer, Spatial Batch Norm after each convolutional layer but before it activates with ReLU, then we use have a max-pooling, this is repeated 3 times. The network architecture is summarized in table 2. We use the Adam optimizer built into pytorch, with an initial learning rate of 0.001, batch size of 64 and Xavier initialization for the weights of the network.

Model 1		Model 2	
Layer Type	Number of hidden units / filters	Layer Type	Number of hidden units / filters
Conv2D	32	Conv2D	32
ReLU	-	Dropout2D	-
Dropout2D	-	BatchNorm2d	-
BatchNorm2d	-	ReLU	-
MaxPool2D	-	MaxPool2D	-
Conv2D	64	Conv2D	64
ReLU	-	BatchNorm2d	-
Conv2d	128	ReLU	-
ReLU	-	MaxPool2D	-
BatchNorm2D	-	Conv2D	128
MaxPool2D	-	BatchNorm2d	-
Conv2D	256	ReLU	-
ReLU	-	MaxPool2D	-
Conv2D	512	Flatten	-
ReLU	-	Dense	64
BatchNorm2D	-	BatchNorm1D	-
MaxPool2D	-	RelU	-
Flatten	-	Dense	10
Dense	2048	-	-
ReLU	-	-	-
Dense	64	-	-
ReLU	-	-	-
BatchNorm1D	-	-	-
Dense	10	-	-

Table 2: The layer architecture of our two networks.

2.2 b)

Below are the final losses and accuracies for our two networks summarized for easy reading. Note that we disabled early stopping and ran for the full 10 epochs in order to show convergence of accuracies so better values are obtained earlier in the training process than those we state.

The best validation accuracy for model 1 was 79.04% and obtained in the 9th epoch, while the best validation loss was 0.6595 obtained in the 5th epoch. For model 2 the best validation accuracy was 75.82% in epoch 7, and the lowest loss was 0.7511 also in the 7th epoch.

	Model 1	Model 2
Final training loss	0.0532	0.1196
Final training accuracy	0.9833	0.9638
Final validation loss	0.8744	0.8879
Final validation accuracy	0.7792	0.7386
Final test loss	0.9055	0.8628
Final test accuracy	0.7792	0.7528

Table 3: Caption

2.3 c)

Here is the accuracy of Model 1 during training, which proved to be the best performing network.

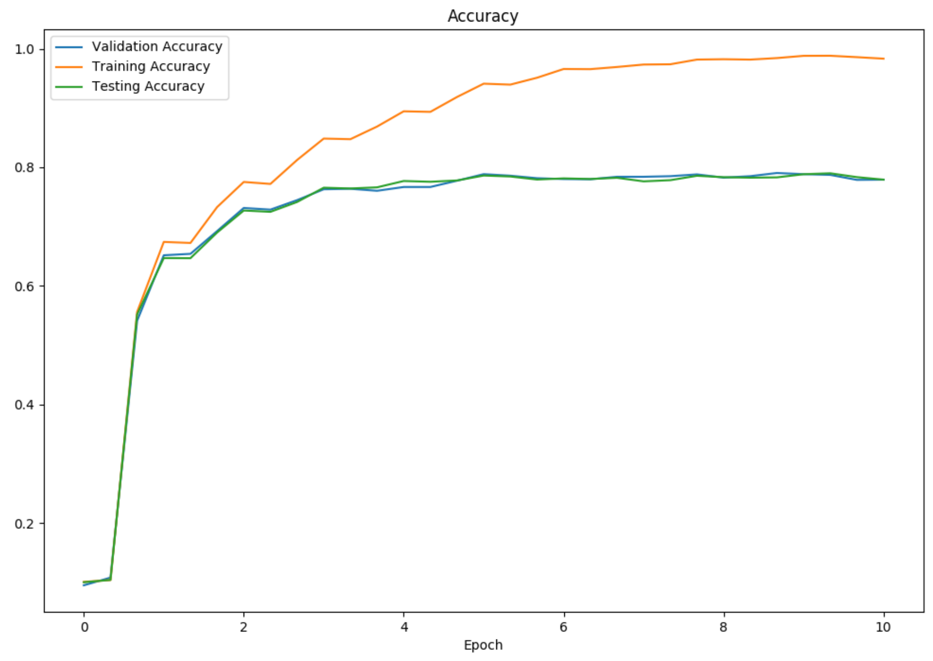

Figure 3: The accuracy of Model 1 which is our best network plotted for each epoch.

2.4 d)

Here are the losses for Model 1 during training.

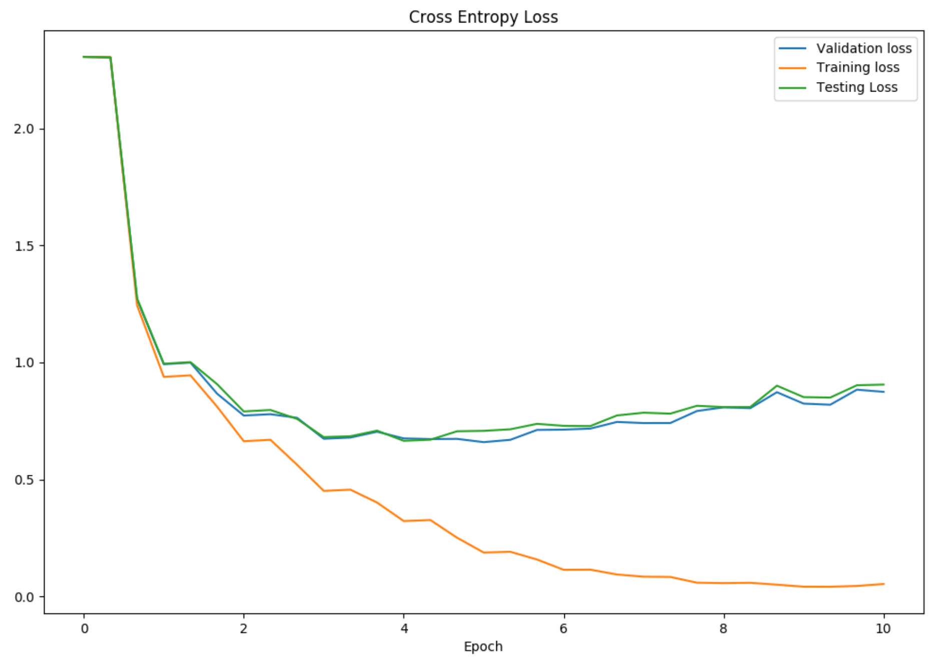

Figure 4: The losses of Model 1 which is our best network plotted for each epoch.

2.5 e)

The biggest performance increase came from changing optimizer from stochastic gradient decent to Pytorch's Adam optimizer. The Adam optimizer smoothed out our back-propagation as it uses both an adaptive learning rate and the momentum principle for weight-updates. The result was faster convergence as well as less oscillations in the losses during training.

Using Batch normalization after each convolution and after each fully connected net greatly improved our training speed as we were able to converge much faster. Batch normalization adds noise to our activation and forces the layers to be more robust with regards to variations in their input. We also chose to include dropout, although with a very low rate of 10%, this might not be necessary as this is another way of adding variation into our network. However, it did push our validation accuracy up about 1%. We note that an aggressive and early dropout rate of say 30% actually worsened our performance.

Changing network architecture to the one of model 1 increased our final validation accuracy of about 3%. This network has two more convolutional layer and a final number of filters of 512 instead of 128 compared to model 1. The increase in accuracy can most likely be attributed to the greater number of filters from more convolutions. Model 2 is however more computationally heavy to train because of the increased network size.

We also experimented with striding convolutions instead of pooling, but the results were comparable so we chose to stick with pooling as it made the math surrounding layer sizes easier.

3 Transfer Learning with ResNet

3.1 a)

We implemented transfer learning with Resnet18 in pytorch. We froze the first layers and only back-propagated through the last five. The hyper-parameters were as suggested: Adam optimizer, batch size of 64, learning rate of $5e-4$ and no other data augmentation.

3.2 b)

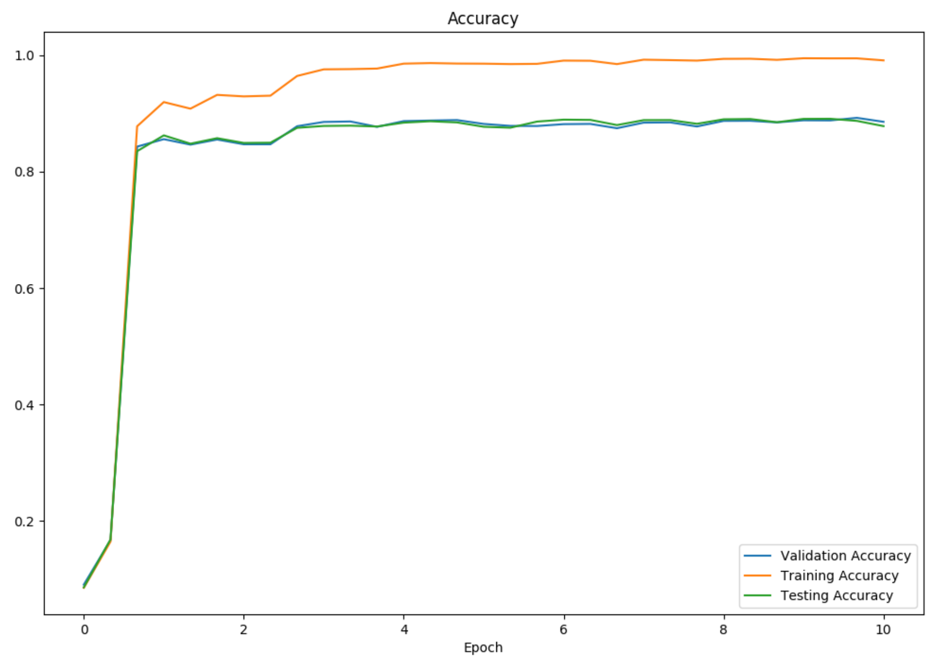

Figure 5: Accuracy for Resnet

3.3 c)

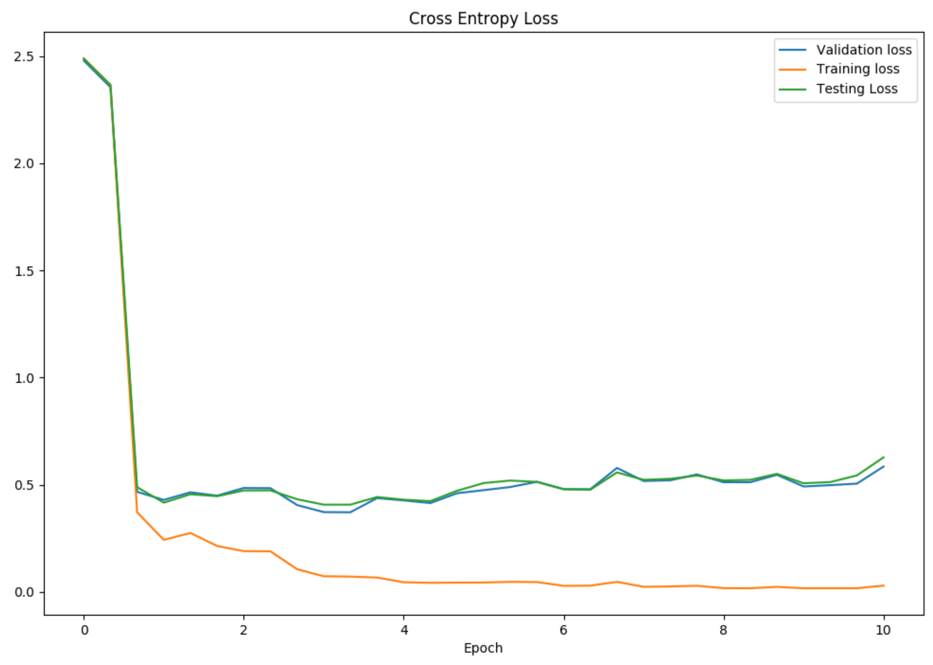

Figure 6: Losses for ResNet

3.4 d)

When comparing the training losses between the two networks we see that ResNet has a much greater learning speed as it converges faster than the other. ResNet also converges to a lower validation loss than the other network. Generally it performs much better.

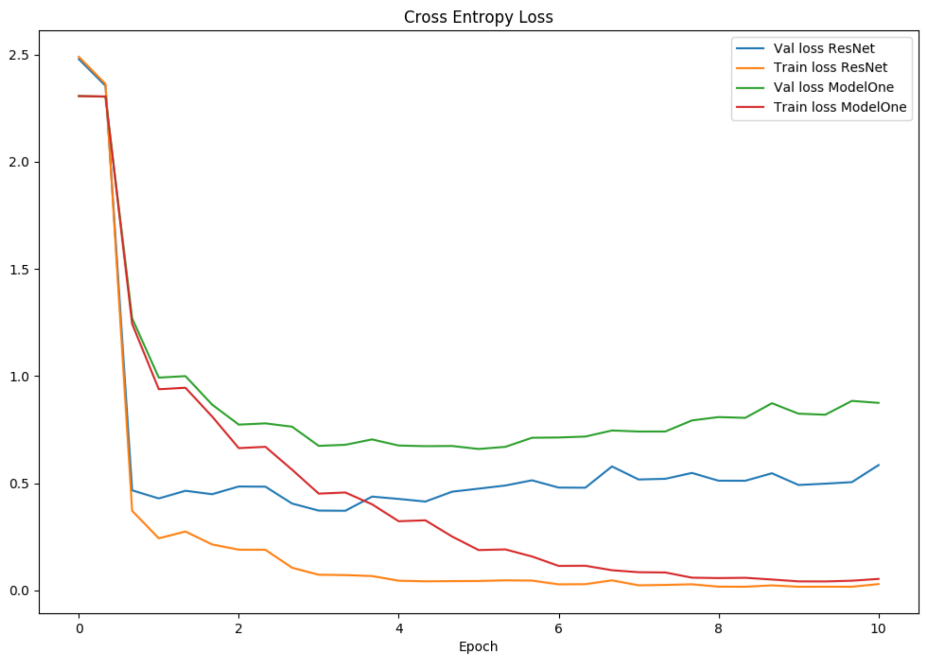

Figure 7: Losses for ResNet and our best model (ModelOne)

3.5 e)

The filter activation from the first layer is shown in figure 8. We input a picture of a zebra shown in 14 and forward pass this through the first layer. By looking at each filter we see that they highlight certain distinct features in our image. Looking at the filters one can see that filter number 11 activates on horizontal lines while filter number 12 activates on vertical lines. Some other filters, like the first one, have no easy graphical interpretation.

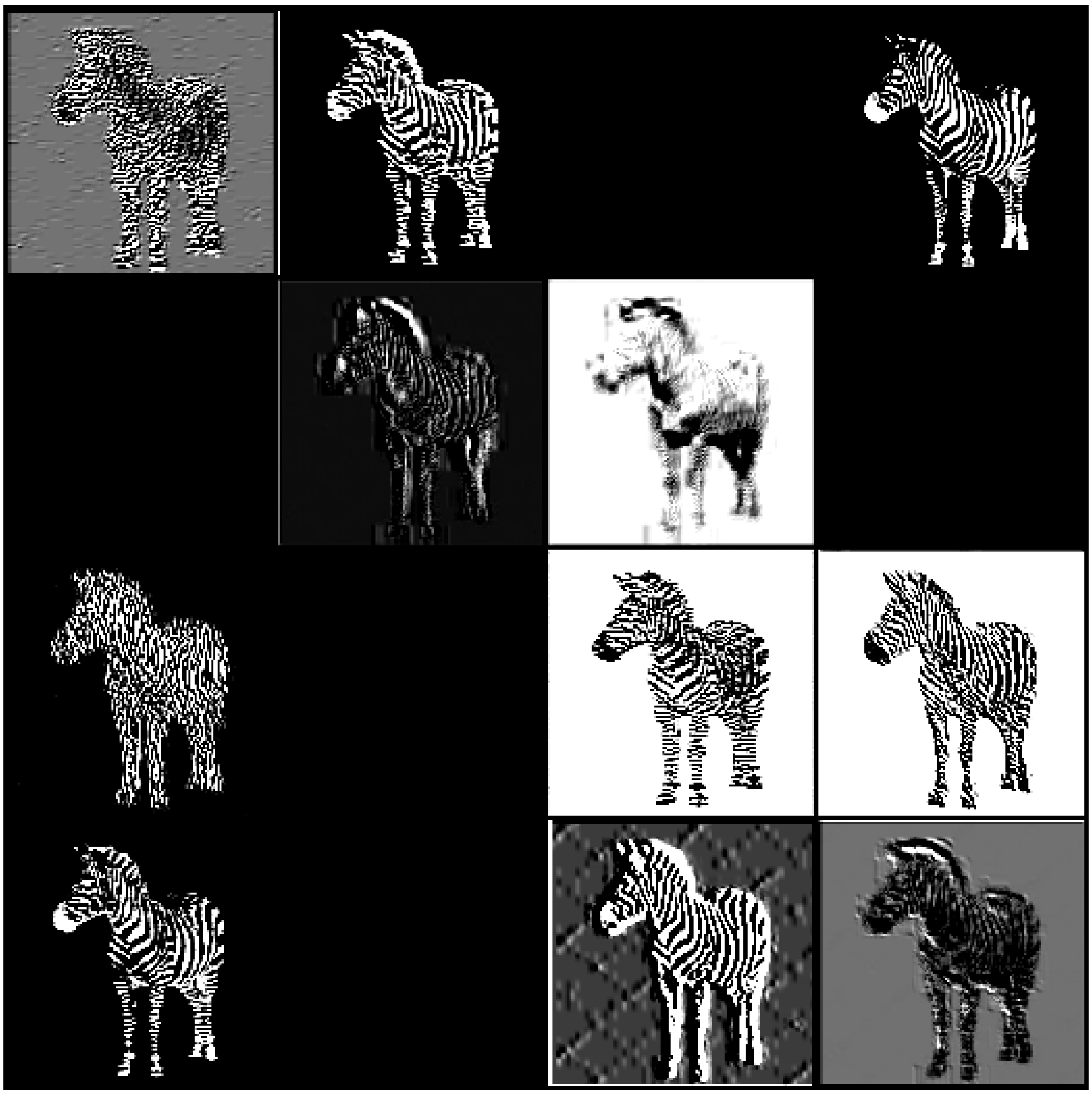
Figure 8: Visualization of the filters from the first convolutional layer. 8×8 filters shown.

3.6 f)

In figure 9 we show the activations of the filters in the last convolutional layer in our network. The filters are now much more abstract and as the input to the convolutional layer has much smaller dimensions than the first layer it is harder for us to interpret what the filter prioritizes graphically.

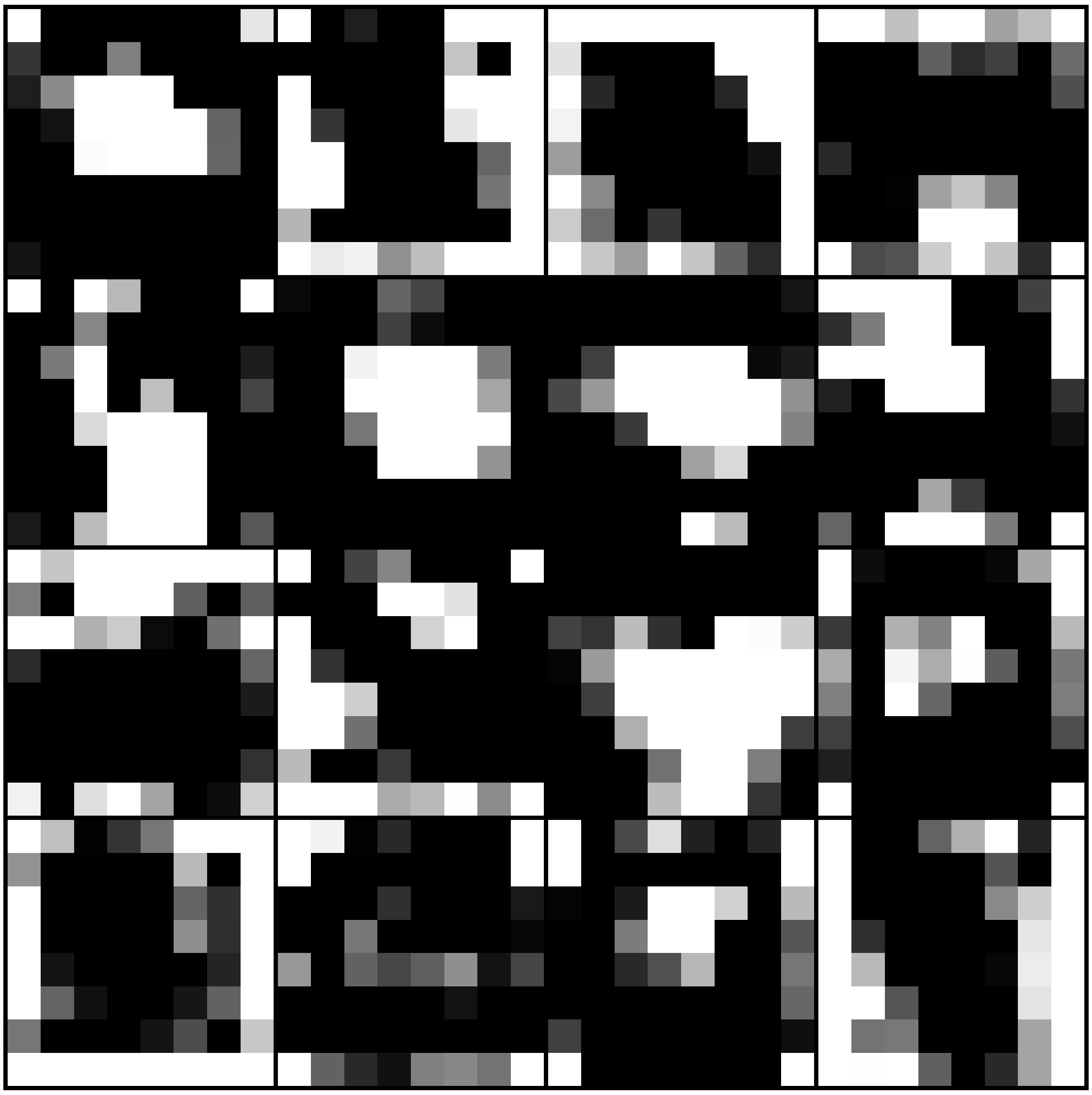

Figure 9: Visualization of the filters from the last convolutional layer. 8 × 8 filters shown.

3.7 g)

A visualization of the weights from the first convolutional layer can be found in fig. (10-13). These represent what features this specific filter is ”looking” for. Each pixel represents the weight for that specific element in the convolutional filter kernel. A higher weight is represented by a brighter color. A graphical interpretation of a kernel can be that it gives a higher activation for those weights that are high, such that for example the first kernel in figure 10 would be responsible for detecting green horizontal lines in the input picture.

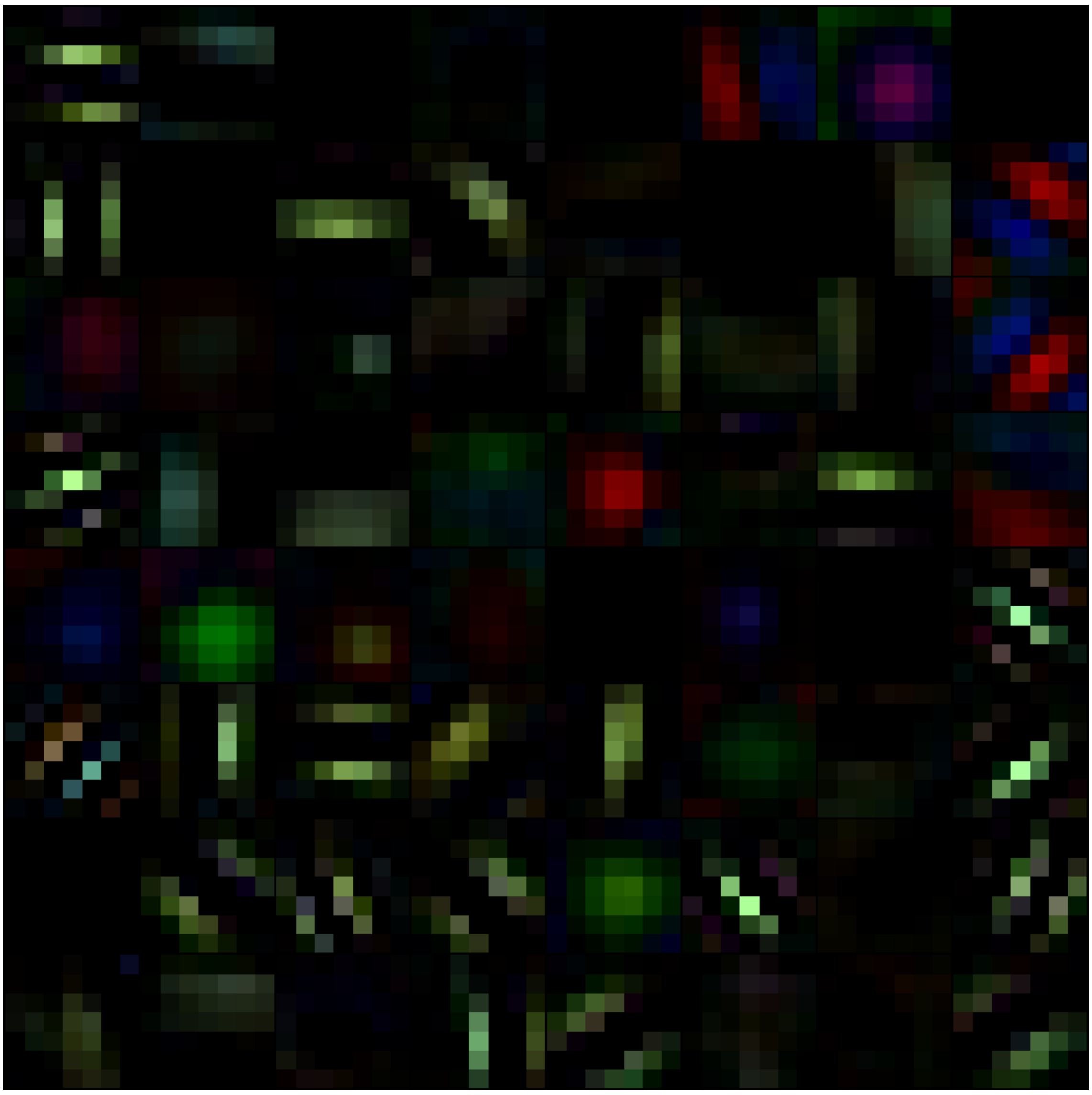
Figure 10: Weights for the first convolutional layer

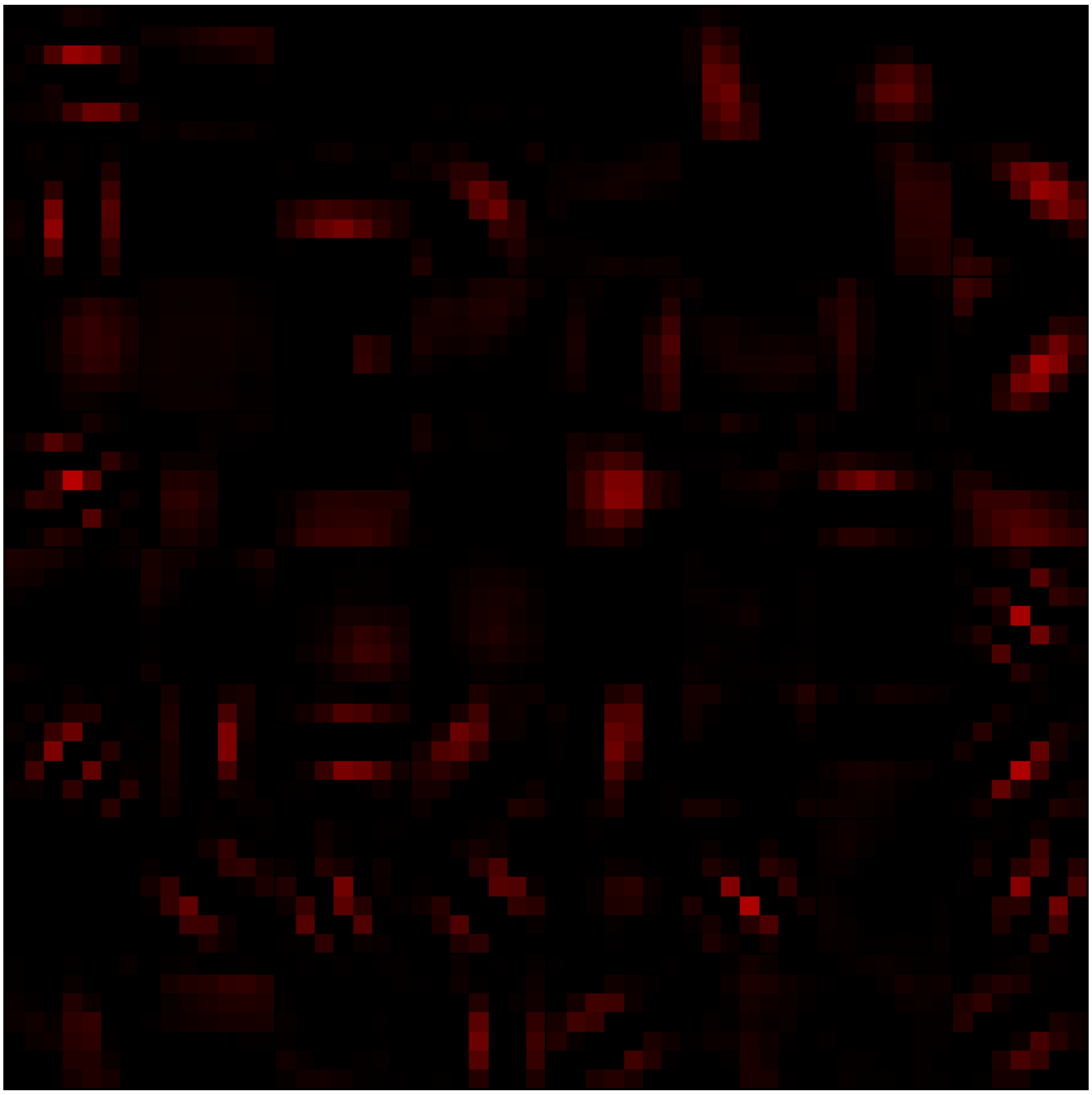
Figure 11: Weights for the first convolutional layer, red only

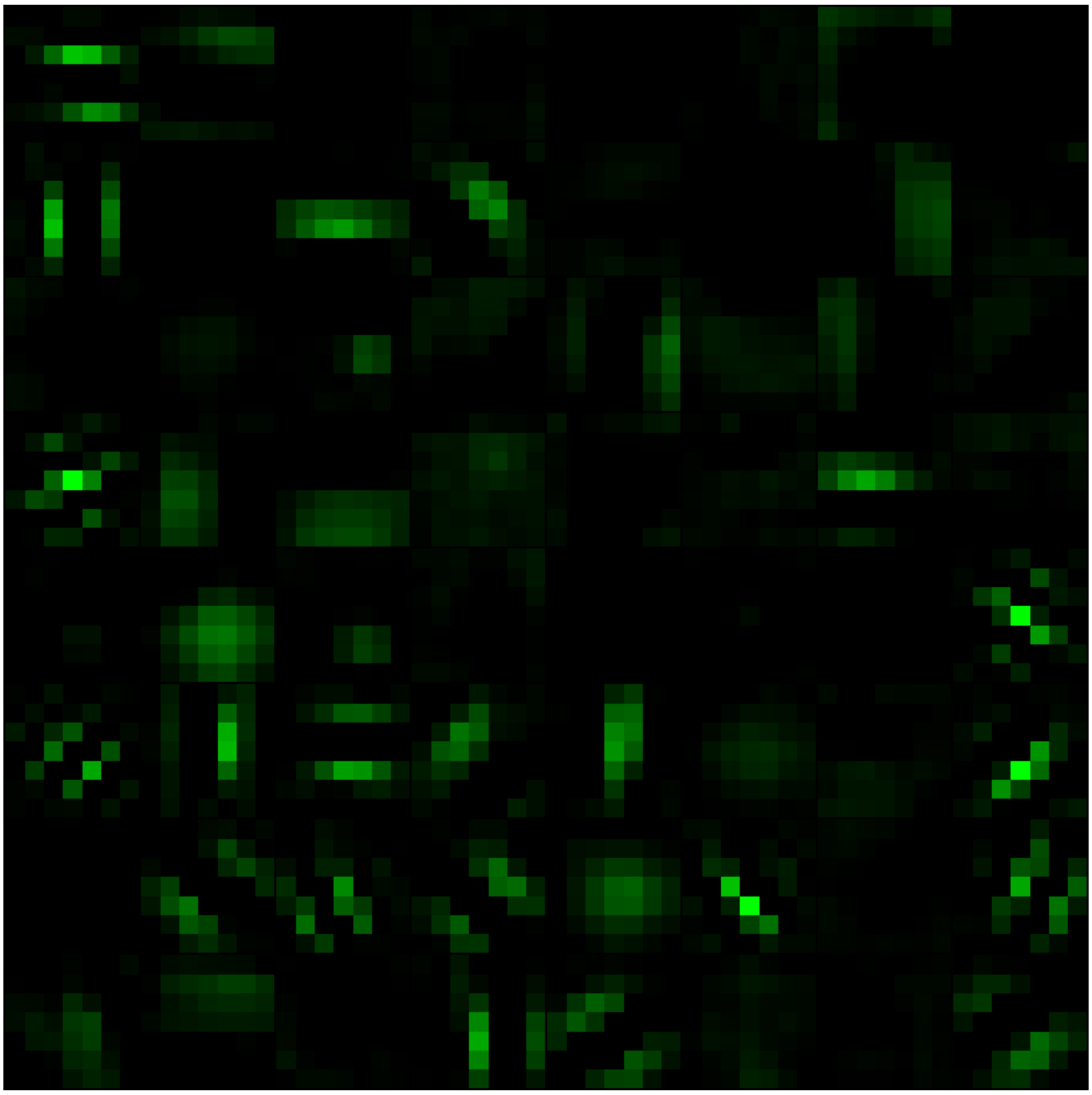

Figure 12: Weights for the first convolutional layer, green only

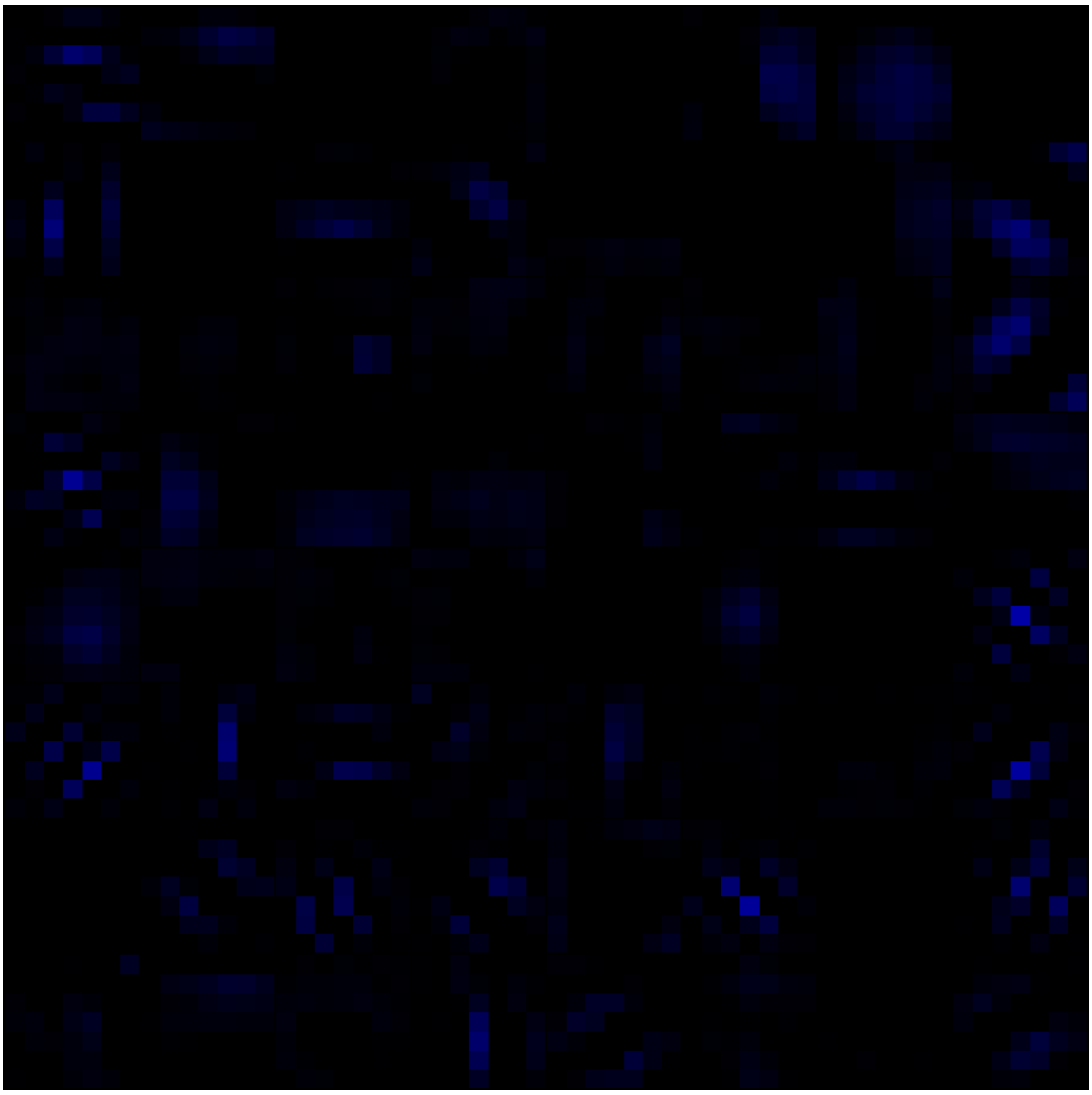
Figure 13: Weights for the first convolutional layer, blue only

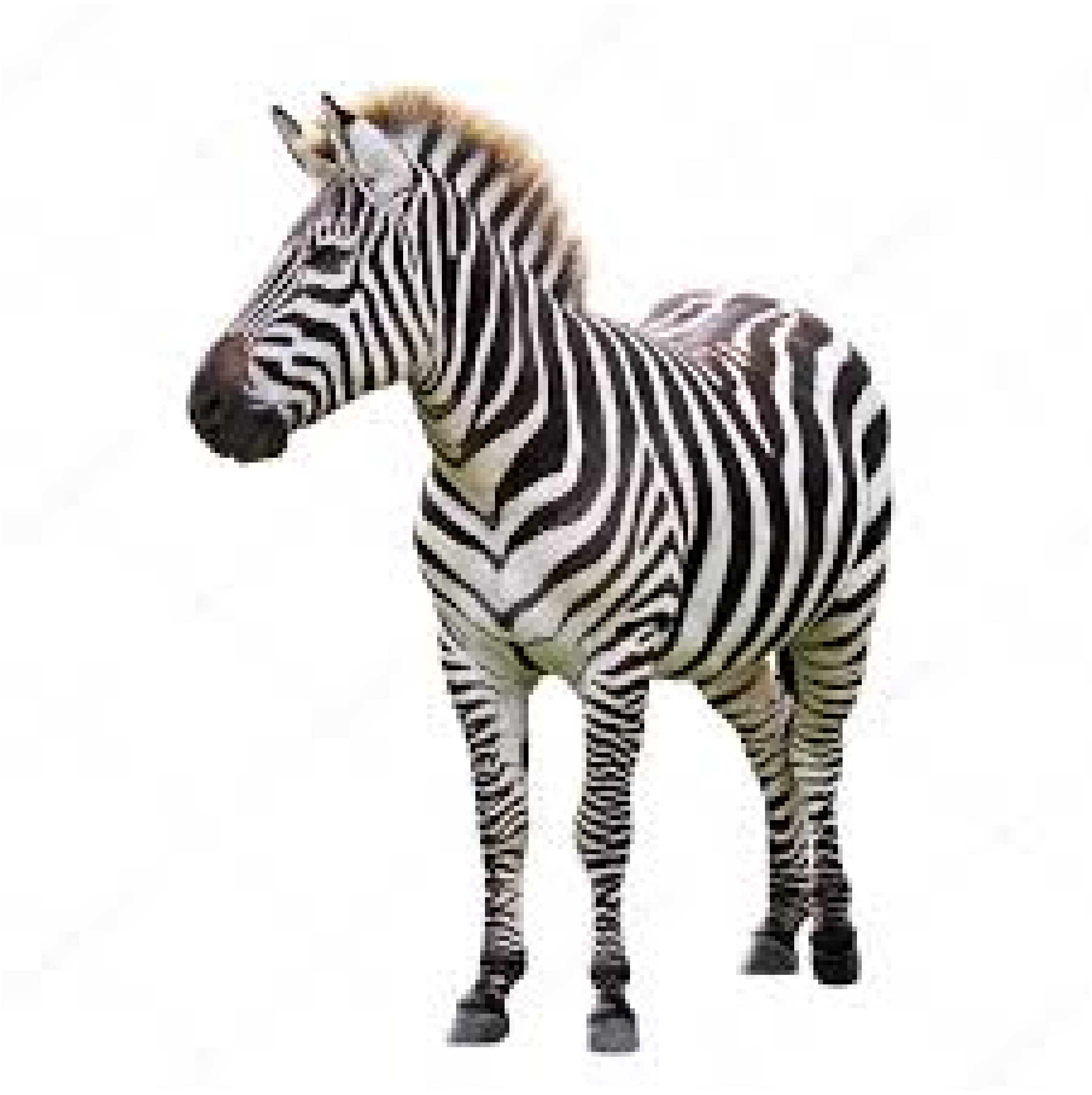

Figure 14: This is a picture of a Zebra. The picture is used to produce the filter activations shown earlier.