Rico Mossinkoff - 12805157 *Hannah Lim - 10588973*
Ewoud Vermeij - 11348860 *Yke Rusticus - 11306386*

Convolutional Neural Networks for Image Classification

Introduction

In this second part of the final project we will use convolutional neural networks (CNN) for image classification. We will use a pre-trained CNN as a starting point and will use transfer learning to fine tune this CNN to fit the provided dataset STL-10. In this assignment we will get familiar with the network structure of the pre-trained CNN and will adjust this structure a little bit so it will fit our data. In addition, we have to structure our data such that it conforms the network structure. To fine-tune the CNN we will experiment with different hyperparameters to construct the CNN with the lowest error for the dataset.

Network structure

The pre-trained convolutional neural network has the structure shown in Table 1. This CNN uses in the layer 1-3: Convolutional layer → Max Pooling layer → ReLU layer. This is not a typical architecture for a CNN. A typical architecture for a CNN is used in the layers 4-9: Convolutional layer → ReLU layer → Average Pooling layer. In the final layer the CNN has the order: Convolutional layer → ReLU layer → Convolutional layer → Softmax layer.

layer	0	1	2	3	4	5	6	7	8	9	10	11	12	13
type	input	conv	mpool	relu	conv	relu	apool	conv	relu	apool	conv	relu	conv	softmxl
name	n/a	layer1	layer2	layer3	layer4	layer5	layer6	layer7	layer8	layer9	layer10	layer11	layer12	layer13
support	n/a	5	3	1	5	1	3	5	1	3	4	1	1	1
filt dim	n/a	3	n/a	n/a	32	n/a	n/a	32	n/a	n/a	64	n/a	64	n/a
filt dilat	n/a	1	n/a	n/a	1	n/a	n/a	1	n/a	n/a	1	n/a	1	n/a
num filts	n/a	32	n/a	n/a	32	n/a	n/a	64	n/a	n/a	64	n/a	10	n/a
stride	n/a	1	2	1	1	1	2	1	1	2	1	1	1	1
pad	n/a	2	0x1x0x1	0	2	0	0x1x0x1	2	0	0x1x0x1	0	0	0	0
rf size	n/a	5	7	7	15	15	19	35	35	43	67	67	67	67
rf offset	n/a	1	2	2	2	2	4	4	4	8	20	20	20	20
rf stride	n/a	1	2	2	2	2	4	4	4	8	8	8	8	8
data size	32	32	16	16	16	16	8	8	8	4	1	1	1	1
data depth	3	32	32	32	32	32	32	64	64	64	64	64	10	1
data num	1	1	1	1	1	1	1	1	1	1	1	1	1	1
data mem	12KB	128KB	32KB	32KB	32KB	32KB	8KB	16KB	16KB	4KB	256B	256B	40B	4B
param mem	n/a	10KB	0B	0B	100KB	0B	0B	200KB	0B	0B	256KB	0B	3KB	0B

parameter memory	569KB (1.5e+05 parameters)
data memory	313KB (for batch size 1)

Table 1: Network architecture of the pre-trained CNN

The CNN filters the image with the convolutional layers, reduces the dimensionality with the Maximum - or Average Pooling layers and uses the ReLU as the nonlinear activation function (ReLU(x) = max{0,x}).

Layer 10 has the most parameters and uses 256KB of memory for parameters. Layer 10 has 4x4x64x64 parameters + 64 bias parameters = 65600 parameters in total.

Rico Mossinkoff - 12805157 *Hannah Lim - 10588973*
Ewoud Vermeij - 11348860 *Yke Rusticus - 11306386*

Preprocessing: data structure

We used the STL-10 given dataset, which contains in total 5000 labeled training and 8000 test images of 10 different classes. In this work, we only selected images with the labels 'airplane', 'bird', 'ship', 'horse' or 'car', resulting in 2500 training and 4000 test images. We set up an imdb struct, holding the data, label, and set (training or test) for each of the used images. Moreover, we resized the images from dimensions 96*96*3 to 32*32*3 and converted the pixel values to singles.

Updating the Network Architecture

The pre-trained CNN has an output layer of 10 different classes. Since our data set STL-10 uses only 5 different classes we will update the output size of the CNN to fit the STL-10 data by changing the output size of the last convolutional layer (layer 12) from 10 to 5. The input size remains the same because the input size of layer 12 will still be 64, because this is the output size of layer 11. The network architecture of the new CNN can be seen in Table 2.

layer	0	1	2	3	4	5	6	7	8	9	10	11	12	13
type	input	conv	mpool	relu	conv	relu	apool	conv	relu	apool	conv	relu	conv	softmxl
name	n/a	layer1	layer2	layer3	layer4	layer5	layer6	layer7	layer8	layer9	layer10	layer11	layer12	layer13
support	n/a	5	3	1	5	1	3	5	1	3	4	1	1	1
filt dim	n/a	3	n/a	n/a	32	n/a	n/a	32	n/a	n/a	64	n/a	64	n/a
filt dilat	n/a	1	n/a	n/a	1	n/a	n/a	1	n/a	n/a	1	n/a	1	n/a
num filts	n/a	32	n/a	n/a	32	n/a	n/a	64	n/a	n/a	64	n/a	5	n/a
stride	n/a	1	2	1	1	1	2	1	1	2	1	1	1	1
pad	n/a	2	0x1x0x1	0	2	0	0x1x0x1	2	0	0x1x0x1	0	0	0	0
rf size	n/a	5	7	7	15	15	19	35	35	43	67	67	67	67
rf offset	n/a	1	2	2	2	2	4	4	4	8	20	20	20	20
rf stride	n/a	1	2	2	2	2	4	4	4	8	8	8	8	8
data size	32	32	16	16	16	16	8	8	8	4	1	1	1	1
data depth	3	32	32	32	32	32	32	64	64	64	64	64	5	1
data num	1	1	1	1	1	1	1	1	1	1	1	1	1	1
data mem	12KB	128KB	32KB	32KB	32KB	32KB	8KB	16KB	16KB	4KB	256B	256B	20B	4B
param mem	n/a	10KB	0B	0B	100KB	0B	0B	200KB	0B	0B	256KB	0B	1KB	0B

parameter memory	567KB (1.5e+05 parameters)
data memory	313KB (for batch size 1)

Table 2: Network architecture of the updated CNN

Setting up the Hyperparameters

For optimizing the hyperparameters we experimented with different parameters. In Table 1 the experiments are shown. We used the test set as validation set. We have changed the hyperparameters for a different learning rate for the previous layers, for the updated layers and the weight decay. For every experiment in hyperparameter settings we experimented with different batch sizes (50 or 100) and different numbers of epoch (40, 80 or 120).

Because we perform transfer learning we want a low learning rate for the previous layers and a higher learning rate for the updated layers.

Since we noticed that the training error often tends to converge towards zero, but the validation error maintained the same, we adjusted the weight decay such that both the training and the validation errors decrease more gradually. This reduced the chance of overfitting. The optimal weight decay was 0.005. A higher weight decay resulted in underfitting.

In the further part of this project we used the hyperparameters (in the table in bold) with the lowest (or best) errors for the fine-tuned network.

Learning rate previous layers	**Learning rate for updated layers**	**Weight decay**	**Batch size**	**Number of epochs**	**Validation objective (last epoch)**	**Validation error (1err) (last epoch)**
[0.005,0.05]	[0.05,0.2]	0.0001	100	120	1.240	0.494
[0.005,0.05]	[0.05,0.2]	0.0001	100	80	1.246	0.497
[0.005,0.05]	[0.05,0.2]	0.0001	100	40	1.251	0.500
[0.05,.5]	[.25,1]	0.0001	100	120	0.612	0.232
[0.05,.5]	[.25,1]	0.0001	100	80	0.616	0.227
[0.05,.5]	[.25,1]	0.0001	100	40	0.629	0.236
[0.2, 2]	[1, 4]	0.0001	100	120	0.459	0.146
[0.2, 2]	[1, 4]	0.0001	100	80	0.463	0.147
[0.2, 2]	[1, 4]	0.0001	100	40	0.454	0.146
[0.1, 1]	[0.5, 2]	0.005	50	120	0.428	0.141
[0.1, 1]	**[0.5, 2]**	**0.005**	**50**	**80**	**0.425**	**0.144**
[0.1, 1]	[0.5, 2]	0.005	50	40	0.429	0.146
[0.1, 1]	[0.5, 2]	0.005	100	120	0.489	0.169
[0.1, 1]	[0.5, 2]	0.005	100	80	0.494	0.175
[0.1, 1]	[0.5, 2]	0.005	100	40	0.504	0.178
[0.1, 1]	[0.5, 2]	0.001	50	120	0.461	0.150
[0.1, 1]	[0.5, 2]	0.001	50	80	0.462	0.156
[0.1, 1]	[0.5, 2]	0.001	50	40	0.447	0.153

Table 3: The errors using different hyperparameters. In bold the experiment is shown with the hyperparameters for the lowest objective and (one of the) lowest validation error (top1err, see Figure 1) in the fine-tuned model.

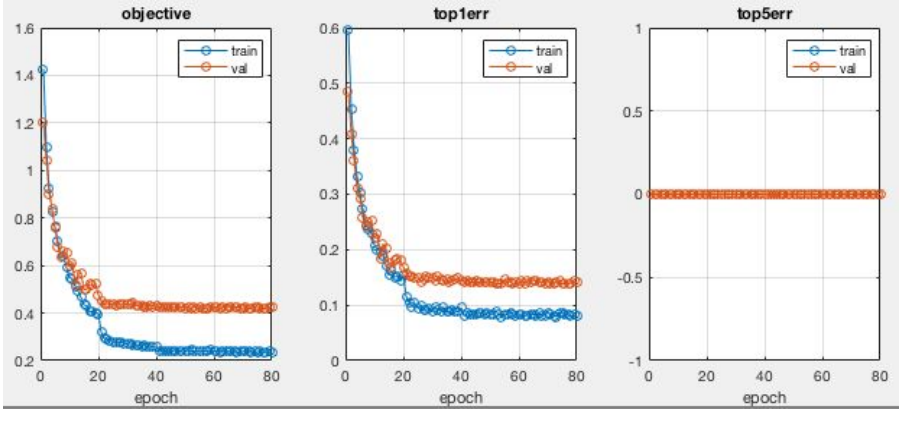

Figure 1: The learning process of the fine-tuned model for the best parameters (see Table 3), where the objective, the top 1 classification and top 5 classification errors are shown per epoch. The top 5 errors are all 0, because we only have 5 classes.

Rico Mossinkoff - 12805157
Ewoud Vermeij - 11348860
Hannah Lim - 10588973
Yke Rusticus - 11306386

Experiments - Visualisation

Pre-trained model: features per layer

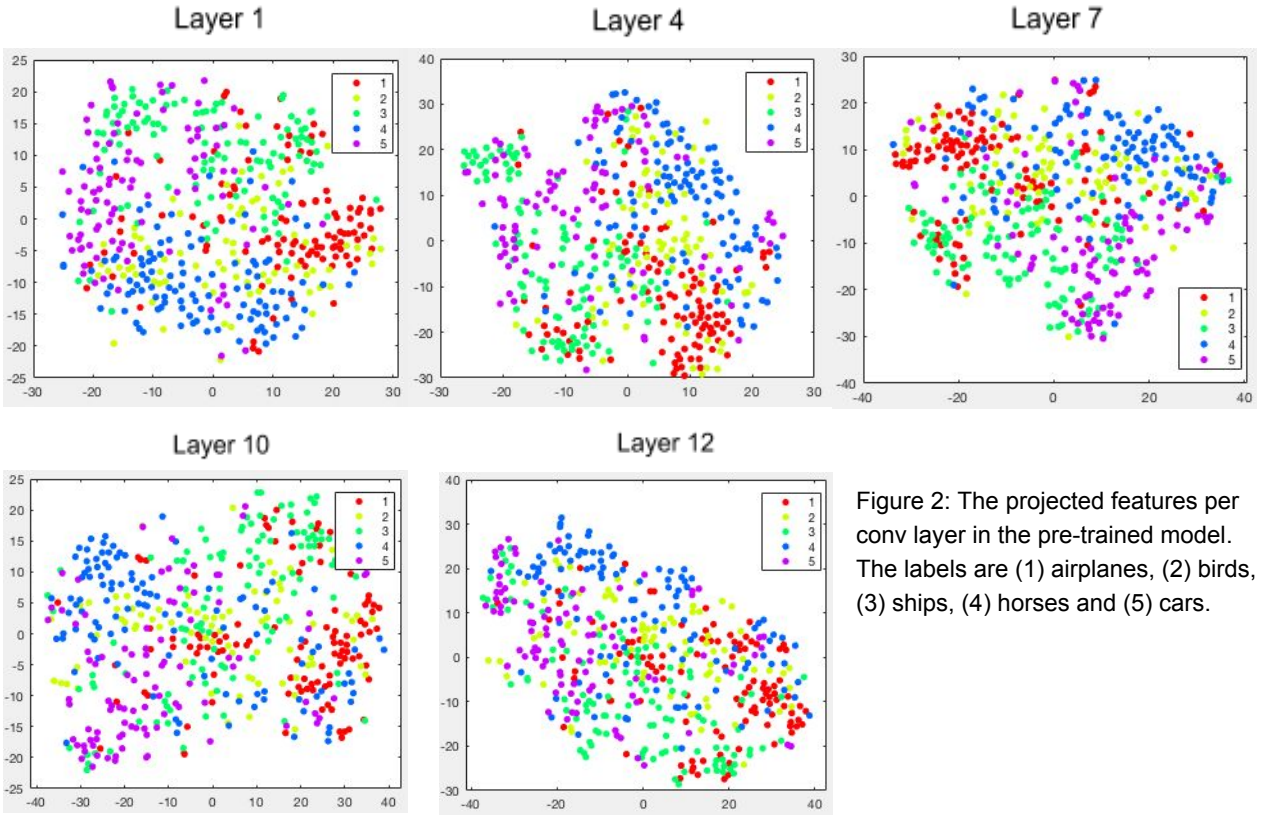

Figure 2: The projected features per conv layer in the pre-trained model. The labels are (1) airplanes, (2) birds, (3) ships, (4) horses and (5) cars.

Fine-tuned model: features per layer

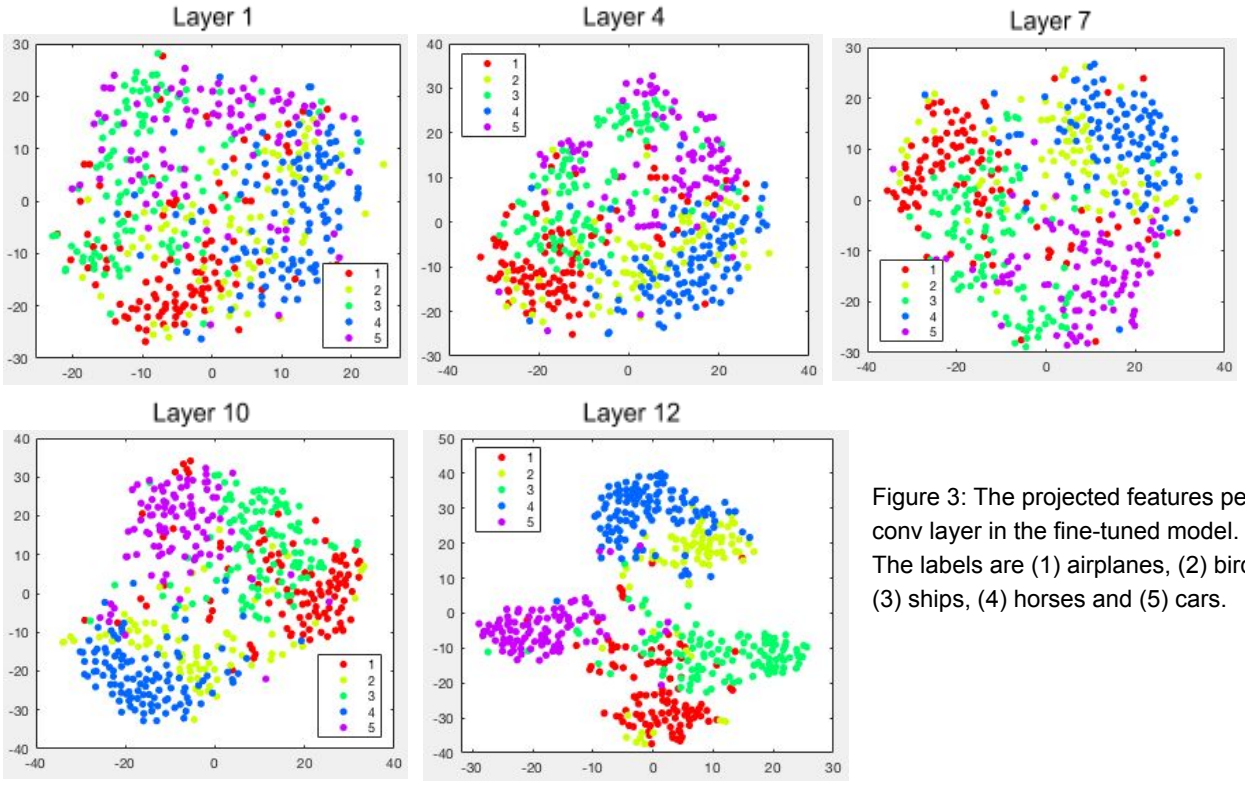

Figure 3: The projected features per conv layer in the fine-tuned model. The labels are (1) airplanes, (2) birds, (3) ships, (4) horses and (5) cars.

In the figures above we can clearly see that the features in the fine-tuned network are much better separated per label than in the pre-trained network.

In the figures of the layers for the pre-trained network layer 1 and 4, we notice that features seem more separated than in layer 10 and 12. A possible reason for this is, that the first layers of a CNN filter on more general features of an image. In the later layers of the network, they will filter more on specific parts of an image.

In the figures of the layers for fine-tuned network we can notice that the features in layer 12 perform the final and best classification for the images.

Experiments - Accuracy

As expected, the accuracy of the SVM using fine-tuned features is higher compared to the SVM using pre-trained features. This makes sense, because the pre-trained CNN is trained on a different dataset. It is remarkable that the pre-trained CNN still gives an accuracy of 62.15% which is not bad.

We can notice that there is almost no difference between the fine-tuned CNN and the SVM using the fine-tuned features.

Configuration	Accuracy (%)
SVM using pre-trained features	62.15
SVM using fine-tuned features	85.82
Fine-tuned CNN	86.00

Table 4: the accuracy of the different models

The highest accuracy (91,78%) of the first part of the project was gained when performing a SVM using dense SIFT operator. The accuracy of the SVM using keypoint SIFT operator has a much lower accuracy (78,62%). The accuracy of the SVM using the dense SIFT operator is considerably higher than the accuracy gained with the Convolutional Neural Network (86.00%) or the SVM using the feature from the CNN (85.82%). The fine-tuned CNN does perform better than the Bag of Words SVM using keypoint SIFT operator.

Conclusion

As expected the fine-tuned network performed better (accuracy: 85.82%) on the dataset then the pre-trained network. Although the pre-trained network (accuracy: 62.15%) also classifies pretty good. This is possibly because the pre-trained network is trained on a dataset containing similar images. As we can see in the *experiments - visualization* part of the report the first layers of the pre-trained network perform better on the data used in this work than the last layers of the network. The first layers of a CNN consist of more general features, like for instance edge and corner detection. These layers are probably quite similar for the pre-trained network as for the fine-tuned network.

Because the pre-trained network contained an output layer of 10 classes instead of 5 classes in our case, we had to update the structure for the network. Besides this small change in the network structure we remained the structure of the pre-trained network. Because the size of the images of our dataset was different than the expected input size of the pre-trained network we adjusted our dataset to fit the input size of the network.

We fine-tuned the network to fit our own dataset by experimenting with the hyperparameters. We found that a lower learning rate for the previous layers than for the updated layers results in the best CNN. Also we regularized the network to prevent overfitting.

Finally, we can conclude that transfer learning on a pre-trained network using optimal hyperparameters can optimize the network. In addition, we have to conclude that doing image classification using Bag of Words was more accurate.