Assignment 3
FMAN45
Machine Learning

Common Layers and Backpropagation, Training a Neural Network, Classifying Handwritten Digits - MNIST, and Classifying Images - CIFAR

Author

Kajsa Hansson Willis

Lund University

Spring 2025

1 Exercise 1

The aim of the first exercise is to derive expressions for $\frac{\partial L}{\partial \mathbf{x}}$, $\frac{\partial L}{\partial \mathbf{W}}$, and $\frac{\partial L}{\partial \mathbf{b}}$, which are the backpropagation gradients, in terms of $\frac{\partial L}{\partial \mathbf{y}}$, $\mathbf{W}$ and $\mathbf{x}$. $\mathbf{W}$ is the weight matrix and $\mathbf{x}$ is the bias vector. y_i is described by Equation 1.

With the chain rule, the expressions in Equations 2, 3 and 4 are obtained.

$$y_i = \sum_{j=1}^{m} W_{ij} x_j + b_i \tag{1}$$

$$\frac{\partial L}{\partial x_i} = \sum_{l=1}^{n} \frac{\partial L}{\partial y_l} \frac{\partial y_l}{\partial x_i} \tag{2}$$

$$\frac{\partial L}{\partial W_{ij}} = \sum_{l=1}^{n} \frac{\partial L}{\partial y_l} \frac{\partial y_l}{\partial W_{ij}} \tag{3}$$

$$\frac{\partial L}{\partial b_i} = \sum_{l=1}^{n} \frac{\partial L}{\partial y_l} \frac{\partial y_l}{\partial b_i} \tag{4}$$

Substituting the expression for y_l from Equation 1 into Equation 2 yields Equation 5.

$$\frac{\partial L}{\partial x_i} = \sum_{l=1}^{n} \frac{\partial L}{\partial y_l} \frac{\partial (\sum_{j=1}^{m} W_{lj} x_j + b_l)}{\partial x_i} = \begin{cases} \sum_{l=1}^{n} \frac{\partial L}{\partial y_l} W_{li}, & \text{if } 1 \le i \le m \\ 0, & \text{if } i > m \end{cases} \tag{5}$$

This can then be applied to the whole vector as the dimensions will be the same, so the zero-values are irrelevant, which is described by Equation 6 for the case $1 \le i \le m$, which gives the desired expression.

$$\frac{\partial L}{\partial \mathbf{x}} = \begin{bmatrix} W_{1,1} & W_{2,1} & \dots & W_{m,1} \\ W_{1,2} & W_{2,2} & \dots & W_{m,1} \\ \dots & \dots & \dots & \dots \\ W_{1,n} & W_{2,n} & \dots & W_{m,n} \end{bmatrix} * \begin{bmatrix} \frac{\partial L}{\partial y_1} \\ \frac{\partial L}{\partial y_2} \\ \dots \\ \frac{\partial L}{\partial y_m} \end{bmatrix} = \mathbf{W}^T \frac{\partial L}{\partial \mathbf{y}} \tag{6}$$

In order to obtain the expression for $\frac{\partial L}{\partial \mathbf{W}_{i,j}}$, the chain rule in Equation 3 is used. This further yields Equation 7, where it is leveraged that the derivative is zero unless $l = i$ and $s = j$.

$$\frac{\partial L}{\partial W_{i,j}} = \sum_{l=1}^{n} \frac{\partial L}{\partial y_l} \frac{\partial (\sum_{s=1}^{m} W_{ls} x_s + b_l)}{\partial W_{i,j}} = \\ = \begin{cases} \sum_{l=1}^{n} \frac{\partial L}{\partial y_l} \frac{\partial}{\partial W_{i,j}} (\sum_{s=1}^{m} W_{l,s} x_s), & \text{if } 1 \le i \le m \\ 0, & \text{if } i > m \end{cases} = \begin{cases} \frac{\partial L}{\partial y_i} x_j, & \text{if } 1 \le i \le m \\ 0, & \text{if } i > m \end{cases} \tag{7}$$

Generalizing this in the same way as previously yields Equation 8, and the final expression.

$$\frac{\partial L}{\partial \mathbf{W}} = \begin{bmatrix} \frac{\partial L}{\partial y_1} \\ \frac{\partial L}{\partial y_2} \\ ... \\ \frac{\partial L}{\partial y_n} \end{bmatrix} * \begin{bmatrix} x_1 & x_2 & ... & x_n \end{bmatrix} = \frac{\partial L}{\partial \mathbf{y}} * \mathbf{x}^T \tag{8}$$

To derive the expression for $\frac{\partial L}{\partial \mathbf{b}}$, the chain rule in Equation 4 is used and the expression from Equation 1 is inserted into the expression for a single element, which is shown in Equation 9.

$$\frac{\partial L}{\partial b_i} = \sum_{l=1}^{n} \frac{\partial L}{\partial y_l} \frac{\partial (\sum_{j=1}^{m} W_{lj} x_j + b_l)}{\partial b_i} = \begin{cases} \frac{\partial L}{\partial y_i}, \text{ if } 1 \leq i \leq m \\ 0, \text{ if } i > m \end{cases} \tag{9}$$

In order to generalize it to the vectors, it is apparent that all elements, i, will be within the m x n limits of the matrix, and thus non-zero (unless the other values are). Further, the expression $\frac{\partial L}{\partial y_i}$ can be generalized to $\frac{\partial L}{\partial \mathbf{y}}$, so thus we obtain Equation 10.

$$\frac{\partial L}{\partial \mathbf{b}} = \frac{\partial L}{\partial \mathbf{y}} \tag{10}$$

2 Exercise 2

The purpose of the second exercise was to derive the expressions for $\mathbf{Y}$, $\frac{\partial L}{\partial \mathbf{X}}$, $\frac{\partial L}{\partial \mathbf{W}}$, and $\frac{\partial L}{\partial \mathbf{b}}$ in terms of $\frac{\partial L}{\partial \mathbf{Y}}$, $\mathbf{W}$, $\mathbf{X}$, and $\mathbf{b}$. After this step, the purpose was to implement the vectorized derived expressions in Matlab functions *fully_ connected_forward* and *fully_ connected_ backward*.

The first step was similar to the first exercise, with the added feature that the matrix is used instead.

The following are known:

$$\mathbf{X} = (\mathbf{x}^{(1)}\ \mathbf{x}^{(2)}\ ...\ \mathbf{x}^{(N)})$$

$$\mathbf{Y} = (\mathbf{W}\mathbf{x}^{(1)} + \mathbf{b} \quad \mathbf{W}\mathbf{x}^{(2)} + \mathbf{b} \quad ... \ \mathbf{W}\mathbf{x}^{(N)} + \mathbf{b} \quad)$$

$$\frac{\partial L}{\partial \mathbf{X}} = (\frac{\partial L}{\partial x^{(1)}} \quad \frac{\partial L}{\partial x^{(2)}} \quad ... \quad \frac{\partial L}{\partial x^{(N)}} \ )$$

$$\frac{\partial L}{\partial \mathbf{Y}} = (\frac{\partial L}{\partial y^{(1)}} \quad \frac{\partial L}{\partial y^{(2)}} \quad ... \quad \frac{\partial L}{\partial y^{(N)}} \ )$$

Equation 11 and 12 are known as well.

$$\frac{\partial L}{\partial W_{ij}} = \sum_{l=1}^{N} \sum_{k=1}^{n} \frac{\partial L}{\partial y_k^{(l)}} \frac{\partial y_k^{(l)}}{\partial W_{ij}} \tag{11}$$

$$\frac{\partial L}{\partial b_i} = \sum_{l=1}^{N} \sum_{j=1}^{n} \frac{\partial L}{\partial y_j^{(l)}} \frac{\partial y_j^{(l)}}{\partial b_i} \tag{12}$$

For $\mathbf{Y}$, it can be rewritten as $\mathbf{Y} = \mathbf{WX} + \mathbf{b}$ from the known expressions for $\mathbf{X}$ and $\mathbf{Y}$.

For $\frac{\partial L}{\partial \mathbf{X}}$, it can be rewritten as Equation 13 with the help of the known expressions and Equation 6.

$$\frac{\partial L}{\partial \mathbf{X}} = (W^T \frac{\partial L}{\partial y^{(1)}} \quad W^T \frac{\partial L}{\partial y^{(2)}} \quad ... \quad W^T \frac{\partial L}{\partial y^{(N)}} \ ) \tag{13}$$

This can be further expressed as Equation 14.

$$\frac{\partial L}{\partial \mathbf{X}} = \mathbf{W}^T \frac{\partial L}{\partial \mathbf{Y}} \tag{14}$$

The expression for $\frac{\partial L}{\partial \mathbf{W}}$ can be aided by Equation 11, as the following can be expressed:

$$\frac{\partial L}{\partial W_{ij}} = \sum_{l=1}^{N} \frac{\partial L}{\partial y_i^{(l)}} x_j^{(l)}$$

This can then be generalized to $\frac{\partial L}{\partial \mathbf{W}} = \frac{\partial L}{\partial \mathbf{Y}} * \mathbf{X}^T$ for the matrix.

For $\frac{\partial L}{\partial \mathbf{b}}$, Equation 12 can be leveraged to determine the expression in Equation 15, where $\mathbf{1}$ is a vector of ones of the corresponding dimensions.

$$\frac{\partial L}{\partial \mathbf{b}} = \sum_{l=1}^{N} \frac{\partial L}{\partial \mathbf{y}^{(l)}} = \frac{\partial L}{\partial \mathbf{Y}} * \mathbf{1} \tag{15}$$

The derived expressions were then implemented into the code layers in the functions. In the forward propagation, the function for Y, termed Z in the function, was *Z = W * X + b*. In the backward propagation, the gradients *dldX = W'*dldZ*, *dldW = dldZ*X'*, and *dldb = sum(dldZ,2)* were inserted. Here, dldZ corresponds to $\frac{\partial L}{\partial Y}$, and the rest correspond to the associated gradient. The functions were successfully tested with the *test_fully_ connected*-test.

3 Exercise 3

In the third exercise, the aim was to derive the backpropagation expression for $\frac{\partial L}{\partial x_i}$ in terms of $\frac{\partial L}{\partial y_i}$ and to then implement the layer in code in the Matlab functions *relu_forward* and *relu_ backward.*

To derive the backpropagation expression, the expression can be written as Equation 16 according to the chain rule in and results in Equation 2.

$$\frac{\partial L}{\partial x_i} = \sum_{l=1}^{n} \frac{\partial L}{\partial y_l} \frac{\partial y_l}{\partial x_i} = \sum_{l=1}^{n} \frac{\partial L}{\partial y_l} \frac{\partial}{\partial x_i}(\max(x_i, 0)) = \begin{cases} \frac{\partial L}{\partial y_i}, \text{ if } x_i > 0 \\ 0 \text{ else} \end{cases} \tag{16}$$

This expression is then used in the Matlab functions. The forward propagation, ie the *relu_forward*-function, was implemented with the critical code *Z=max(0,X)*. The backward propagation, ie the *relu_backward*-function, was implemented with the critical code *dldX = dldZ .* (X > 0)*. In the code, Z corresponds to what is here known as Y, so dldX means $\frac{\partial L}{\partial \mathbf{X}}$.

The functions were successful when tested with the *test_relu*-Matlab test.

4 Exercise 4

In the fourth exercise, the purpose is to compute the expression for $\frac{\partial L}{\partial x_i}$ in terms of y_i, and to then implement this layer in the code through the functions *softmaxloss_forward* and *softmaxloss_backward.*

The loss, L, whose gradient is to be computed, can be expressed as Equation 17, where c is the ground truth class. Further, it is known that $y_i = \frac{e^{x_i}}{\sum_{j=1}^{m} e^{x_j}}$.

$$L(\mathbf{x}, c) = -x_c + \log(\sum_{j=1}^{m} e^{x_j}) \tag{17}$$

It is then possible to derive an expression for $\frac{\partial L}{\partial x_i}$, shown in Equation 18.

$$\frac{\partial L}{\partial x_i} = \frac{\partial}{\partial x_i}(-x_c + \log(\sum_{j=1}^{m} e^{x_j})) = \begin{cases} -1 + \frac{e^{x_i}}{\sum_{j=1}^{m} e^{x_j}}, \text{ if } i = c \\ \frac{e^{x_i}}{\sum_{j=1}^{m} e^{x_j}}, \text{ if } c \neq i \ (1 \leq c \leq m) \\ 0, \text{ if } i > m \text{ or } i < 1 \end{cases} \tag{18}$$

Inserting the expression for y_i into Equation 18 yields Equation 19 as the expression for $\frac{\partial L}{\partial x_i}$ in terms of y_i.

$$\frac{\partial L}{\partial x_i} = \begin{cases} -1 + y_i, \text{ if } i = c \\ y_i, \text{ if } c \neq i \ (1 \leq c \leq m) \\ 0, \text{ if } i > m \text{ or } i < 1 \end{cases} \tag{19}$$

In order to implement this into the code, the forward propagation implemented the Softmax loss through the code below to calculate the average negative log-likelihood over the batch elements.

```
lnsumexp = log(sum(exp(x), 1));
idx = sub2ind(size(x), labels(:)', 1:batch);
 correct = x(idx);
 losses = -correct + lnsumexp;
L = mean(losses);
```

For the backward propagation, the derived expression in Equation 19 was calculated through the code below in order to determine the average of the partial derivative over the batch elements.

```
icidx = sub2ind(size(x), labels(:)', 1:batch);
dldx = exp(x) ./ sum(exp(x));
dldx(icidx)= dldx(icidx) - 1;
dldx = dldx ./batch;
```

The code was then tested with the Matlab test functions *test_ softmaxloss* and *test_ gradient_ whole_ net* and both were successful.

5 Exercise 5

In exercise five, gradient descent with momentum was implemented in the Matlab function *training.*

This was based on Equations 20 and 21 where $\mathbf{m}_n$ is a moving average of the gradient estimations and μ is a hyperparameter between 0 and 1 controlling the smoothness.

$$\mathbf{m}_n = \mu \mathbf{m}_{n-1} + (1-\mu)\frac{\partial L}{\partial \mathbf{w}} \tag{20}$$

$$\mathbf{w}_{n+1} = \mathbf{w}_n - \alpha \mathbf{m}_n \tag{21}$$

Further, weight decay is used, which means that Equation 21 must be adjusted to account for it, which gives Equation 22, where λ is the weight decay factor.

$$\mathbf{w}_{n+1} = \mathbf{w}_n - \alpha(\mathbf{m}_n + \lambda \mathbf{w}_n) \tag{22}$$

These were then implemented into the code in the *training*-function. The inserted code can be seen below:

```
 momentum{i}.(s) = opts.momentum * momentum{i}.(s) + (1 - opts.momentum) * grads{i}.(s);
net.layers{i}.params.(s) = net.layers{i}.params.(s) - opts.learning_rate*
* (momentum{i}.(s)+opts.weight_decay*net.layers{i}.params.(s));
```

In the code *opts.momentum* corresponds to μ, *momentum* corresponds to $\mathbf{m}$, *grads* corresponds to $\frac{\partial L}{\partial \mathbf{w}}$, *net.layers.params* corresponds to $\mathbf{w}$, *opts.learning_ rate* corresponds to α, and *opts.weight_ decay* corresponds to λ.

6 Exercise 6

In the sixth exercise, the purpose was to display plot of the kernels for the first convolutional layer, a few misclassified images, the confusion matrix for the predictions on the test set, as well as compute the precision and recall for all the digits and count the number of parameters for all layers in the network.

This was done with the *mnist_ starter* as a guide. The accuracy for the test set reaches 97,6% which implies that the code is correct. The accuracy over the iterations can be studied in Figure 1. Additionally, the loss per iteration is displayed in Figure 2.

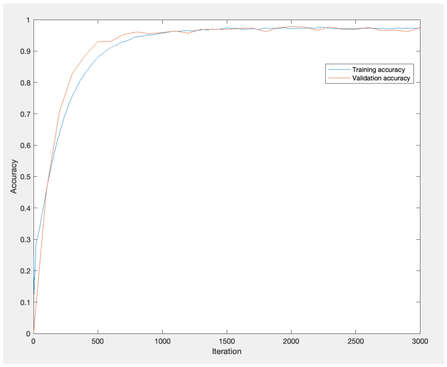


Figure 1: The training and validation accuracy per iteration for the MNIST data set

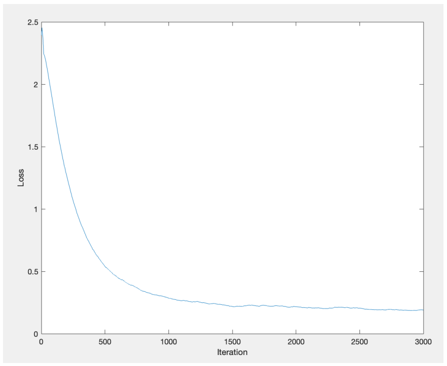


Figure 2: The loss per iteration for the MNIST data set defined by the loss function

The kernels for the first convolutional layer learns are displayed in Figure 3. It is likely that they aim to detect lines or edges, which can be considered the most rough perception of the images, and the images at hand align with this hypothesis.

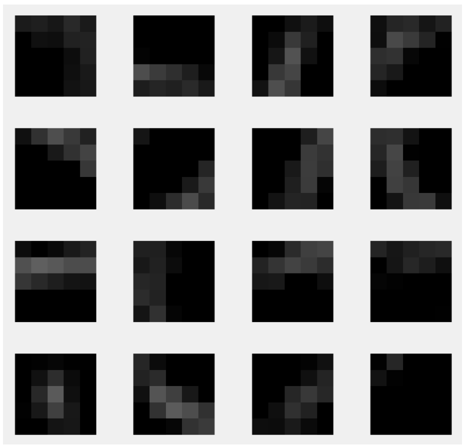


Figure 3: The kernels for the first convolutional layer learns from the MNIST data set

Although the accuracy is very high, it is not perfect and a few misclassified images are displayed in Figure 4. The confusion matrix is displayed in Figure 5 to give a larger picture of the misclassifications. The classes start with zero, so class ten corresponds to the digit nine.

From the images of the misclassified digits, it is apparent that a majority are true eights that have been predicted incorrectly. There are many incorrect predictions of two of these, but a variety of numbers are represented. Additionally, it is not uncommon that the images have been incorrectly predicted as nines, as it holds for a quarter of the displayed misclassified images. I would have a hard time predicting some of the misclassified images, so it is a difficult task. A lot of these numbers have irregularities or unusual dimensions for the digit, so I can understand the decision in a few cases. The confusion matrix shows this as well, as class 9, corresponding to digit eight, has a substantial amount of misclassifications, especially for true values incorrectly predicted as other classes, and zero and two are especially common. Nine is also problematic, especially when it comes to non-nines being predicted as nines. Another value that sticks out a little is two, which was some other digits predicted incorrectly as twos. It is likely that the curved shapes and similar structures of these digits make it harder to predict them correctly. It is also apparent that the mechanism has high precision as nearly all of the digits are correctly predicted.

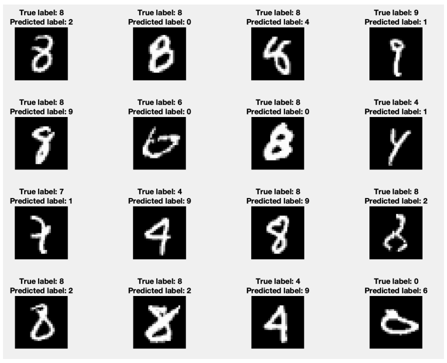
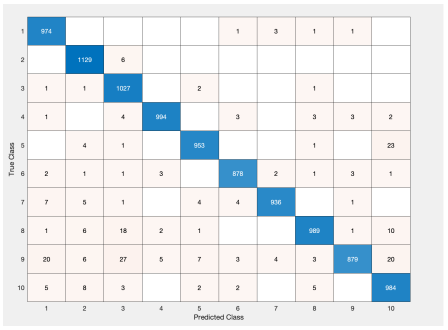


Figure 4: A selection of a few misclassified images with their true labels and predicted labels of the MNIST data set



	1	2	3	4	5	6	7	8	9	10
1	974					1	3	1	1	
2		1129	6							
3	1	1	1027		2			1		
4	1		4	994		3		3	3	2
5		4	1		953			1		23
6	2	1	1	3		878	2	1	3	1
7	7	5	1		4	4	936		1	
8	1	6	18	2	1			989	1	10
9	20	6	27	5	7	3	4	3	879	20
10	5	8	3		2	2		5		984

Figure 5: The confusion matrix from the MNIST data set starting with digit zero at index one and ending with digit nine at index 10

The precision and recall for all the digits are displayed in Table 1. It is important to note that it starts at zero, so the tenth class corresponds to nine. This is clearly in line with the results from the confusion matrix, as the recall for eight is the lowest, followed by the precision for two and nine. These are the only situations where the probability is below 95% and in many cases the likelihood is very close to 100%. This means that the neural network is very successful. It is interesting to note that there is no clear pattern between the precision and the recall, which likely means that the neural network does not have problems distinguishing between two digits for example, but rather there are more general difficulties when the predictions go awry.

The number of trainable parameters are 14 682 and further displayed in Table 2. From the table, it is apparent that the convolution layers and the fully

	0	1	2	3	4	5	6	7	8	9
Precision:	0.9634	0.9733	0.9439	0.9900	0.98349	0.9854	0.9905	0.9851	0.9899	0.9462
Recall:	0.9939	0.9947	0.9952	0.9842	0.9705	0.9843	0.9770	0.9621	0.9025	0.9752

Table 1: The precision and recall values for the digits from the MNIST data set

connected, ie dense, layer provide all of the trainable parameters. Many of the layers do not have any parameters.

Layer:	***Weight Parameters:***	***Bias Parameters:***	***Total Parameters:***
Input	0	0	0
Convolution	400	16	416
ReLU	0	0	0
Maxpooling	0	0	0
Convolution	6400	16	6416
ReLU	0	0	0
Maxpooling	0	0	0
Fully connected	7840	10	7850
Softmaxloss	0	0	0
Total:	**14 640**	**42**	**14 682**

Table 2: The amount of parameters for all layers of the network for the MNIST data set in regards to both weights and biases, as well as the total amount of parameters

7 Exercise 7

In the last exercise, the purpose is to improve the accuracy of the baseline network. This was done by changing factors such as the network structure, the training time, and other hyperparameters.

The initial accuracy on the test set was 48.57% for 5 000 iterations. The accuracy across the iterations can be studied in Figure 6. Similarly, the loss over the iterations can be studied in Figure 7.

In order to improve the accuracy, the neural network was first examined. Its purpose is to predict which of ten categories an image belongs to. The network consists of eight layers, described by Table 3. The structure is the same as for the *mnist_ starter* except that the second Maxpooling layer is removed and the convolutional and fully connected layers have many more weight parameters. This leads to a very large increase in the amount of trainable parameters from the earlier neural network.

It was determined that it would be easier to try to fine-tune the parameters before looking at adding extra layers in the network. Thus, that is where the project to increase the accuracy began by adjusting the hyperparameters

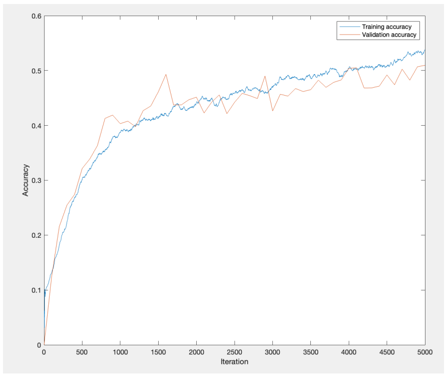


Figure 6: The initial training and validation accuracy per iteration for the CIFAR10 data set

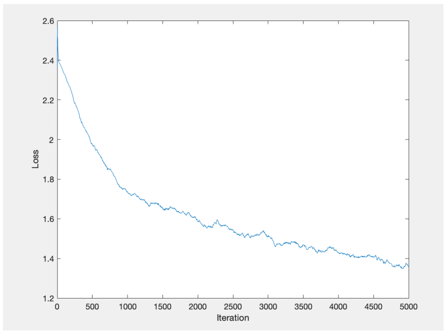


Figure 7: The initial loss per iteration for the CIFAR10 data set, defined by the loss function

Layer:	***Weight Parameters:***	***Bias Parameters:***	***Total Parameters:***
Input	0	0	0
Convolution	1 200	16	1 216
ReLU	0	0	0
Maxpooling	0	0	0
Convolution	6 400	16	6416
ReLU	0	0	0
Fully connected	40 960	10	40 970
Softmaxloss	0	0	0
Total:	**48 560**	**42**	**48 602**

Table 3: The amount of parameters for all layers of the network in regards to both weights and biases, as well as the total amount of parameters for the initial CIFAR10 model

slightly. In case this was not sufficient, the network structure would be reconsidered by adding layers to it.

The first change that was made was to increase the amount of training data to see if that would make a difference. The hypothesis was that with more training data, the model might be able to train itself better. The amount of images uploaded was doubled to 40000. This led to an accuracy on the test set of 49.21%. Thus, the accuracy was increased, but more changes are required to reach the target of 55%.

Next, the training parameters were observed and the next change made was to increase the amount of iterations to 7000 from 5000 and decrease the momentum parameter to 0.9 from 0.95 in the training of the model. The idea was that more iterations could lead to better results, as the accuracy is a bit shaky over iterations which can be observed in Figure **??** and to try to optimize the momentum. After this change, the accuracy increased to 52.85%, so the adjustments were beneficial.

The third change was to adjust the weight decay parameter to 0.002 from 0.001 to try to see if optimization of the weight decay parameter is possble. This led to an accuracy of 50.62%, which was lower than the previous value. Further, a shift to 0.0005 was considered that gave an accuracy of 53.81%, an improvement from the previous change. Thus, this change was implemented.

The fourth change considered was to increase the learning rate from 0.001 to 0.002. This change led to a significantly decreased accuracy of 49.95%. Thus, a decrease of the learning rate from 0.0005 was considered as well. This led to an accuracy of 53.53%, which is just a little bit lower than it was previous. Thus, a change in the learning rate was not implemented.

The hyperparameters have thus been adapted to increase the performance of the network to 53.81%, but to further improve the accuracy structural changes should be considered as well. Therefore, it is interesting to explore adding layers.

A convolutional layer was added after the second ReLU-layer, ie as the seventh layer. This was done to add more parameters to the network as they might be able to capture valuable traits. The network did not seem to be overfitted which strengthens the hypothesis that this may work. This convolution was selected to be 3x3 in order to allow for more detailed refinement. Further, padding was added to maintain the size. The accuracy with the convolution layer was 57.22% as the added convolutional layer introduces additional feature extraction capacity that allows the network to better distinguish classes with only subtle differences. This was above the target of 55% so the network was considered satisfactory. The accuracy throughout the iterations can be seen in Figure 8. The losses per iteration can be seen in Figure 9.

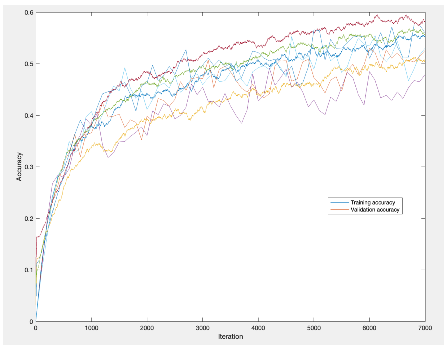


Figure 8: The training and validation accuracy per iteration for the CIFAR10 data set of the improved network

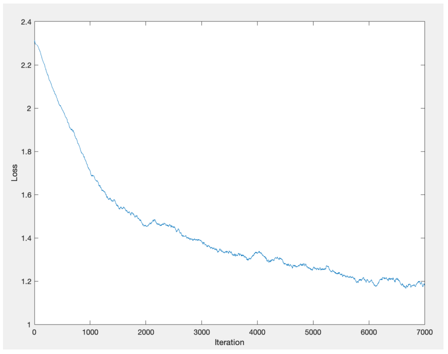


Figure 9: The loss per iteration for the CIFAR10 data set of the improved network, defined by the loss function

The kernels for the first convolutional layer learns can be seen in Figure 10. From the images, it appears that the filters try to sort based on backgrounds and colors in the first step, which appears reasonable with the added color scheme for this data set.

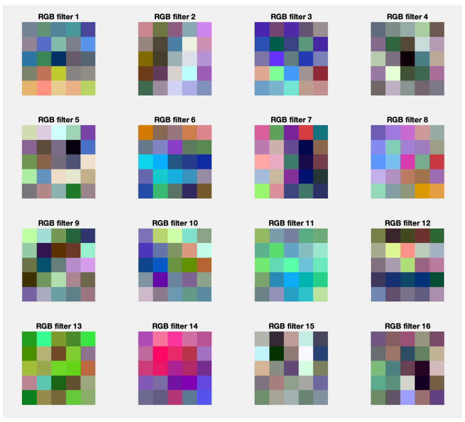


Figure 10: The kernels for the first convolutional layer learns from the improved network for the CIFAR10 data set

Some misclassified images are displayed in Figure 11. The idea that the first layer filters by colors and background is strengthened by these images, as the most green images are incorrectly predicted to be frogs. It appears that images of animals may be overrepresented among the errors and deer seem to be the hardest to predict. Cats appear to be predicted incorrectly frequently as well.

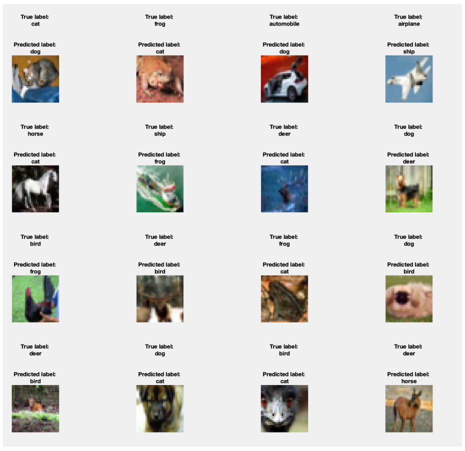
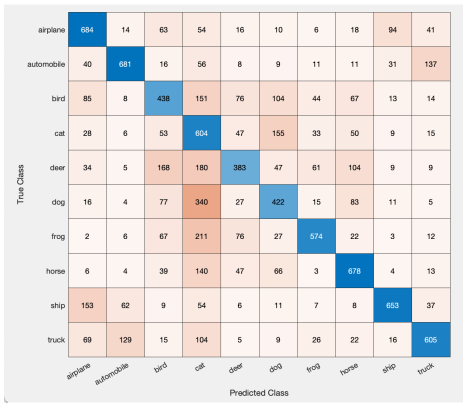


Figure 11: A selection of a few misclassified images with their true labels and predicted labels of the CIFAR10 data set

The confusion matrix is displayed in Figure 12. This gives more general evidence to the theory that cats are very often predicted incorrectly, especially when the image displays a dog. Dogs being mistaken for cats is not unheard of either, according to the confusion matrix. It is clear that there are other cases as well where two things are frequently mistaken for each other. Automobile and truck is one such case that intuitively makes sense, as they can be quite similar. The recall and precision values can be found in Table 4. It is apparent that no prediction reaches the success from the MNIST predictions, but some are better than others. The precision of cats is, for instance, very low. The recall of deer is also quite low. This information is not surprising after seeing some of the misclassified images and the confusion table. Automobiles and ships are involved in the most precise predictions, which is in line with the hypothesis that the inanimate objects were easier to predict.



	airplane	automobile	bird	cat	deer	dog	frog	horse	ship	truck
airplane	684	14	63	54	16	10	6	18	94	41
automobile	40	681	16	56	8	9	11	11	31	137
bird	85	8	438	151	76	104	44	67	13	14
cat	28	6	53	604	47	155	33	50	9	15
deer	34	5	168	180	383	47	61	104	9	9
dog	16	4	77	340	27	422	15	83	11	5
frog	2	6	67	211	76	27	574	22	3	12
horse	6	4	39	140	47	66	3	678	4	13
ship	153	62	9	54	6	11	7	8	653	37
truck	69	129	15	104	5	9	26	22	16	605

Figure 12: The confusion matrix from the improved network for the CIFAR10 data set

	Airplane	Automobile	Bird	Cat	Deer	Dog	Frog	Horse	Ship	Truck
Precision:	0.6124	0.7410	0.4635	0.3189	0.5542	0.4907	0.7359	0.6378	0.7746	0.6813
Recall:	0.6840	0.6810	0.4380	0.6040	0.3830	0.4220	0.5740	0.6780	0.6530	0.6050

Table 4: The precision and recall values for the improved network for the CIFAR10 data set

A table of the amount of parameters in all of the layers in the network can be found in Table 5. The amount of trainable parameters has increased a little bit from Table 3 with the new convolution layer, but it is no considerable change compared to the total amount. However, this change combined with optimizing the hyperparameters, led to an increase in the performance of the network.

Layer:	***Weight Parameters:***	***Bias Parameters:***	***Total Parameters:***
Input	0	0	0
Convolution	1 200	16	1 216
ReLU	0	0	0
Maxpooling	0	0	0
Convolution	6 400	16	6 416
ReLU	0	0	0
Convolution	2 304	16	2 320
Fully connected	40 960	10	40 970
Softmaxloss	0	0	0
Total:	**50 880**	**42**	**50 922**

Table 5: The amount of parameters for all layers of the network in regards to both weights and biases, as well as the total amount of parameters for the final improved neural network