# Machine Learning, LTH 2021

*Assignment 3: Feed-Forward Neural Network for Classification*

Name: Isabelle Frodé
Date: May 14 2021

## Assignment Overview

The assignment was given in the course Machine Learning FMAN45 at LTH spring 2021. The goal with the assignment was to study, implement and train a feed-forward neural network for classification. Expressions for backpropagation was derived and implemented for a dense layer using a linear function and then using the ReLU activation function. The softmax loss function was then implemented in the final layer of the neural network. For the training of the neural network a gradient descent algorithm with momentum was implemented. A neural network using the implemented functions was then used to classify handwritten digits from the MNIST dataset. The network reached 98 % accuracy. A neural network using the implemented functions was then trained on the CIFAR10 dataset to classify tiny images. The baseline network had an accuracy of 48 %. After elaborating on the network it reached an accuracy of 54 %.

# Common Layers and Backpropagation

An artificial neural network (ANN) is a network of nodes arranged in computational layers. A feed-forward neural network is an ANN where the nodes do not form a cycle. The theory section of this report aim to derive backpropagation expressions for common layers in neural networks.

The loss in neural networks $L : \mathcal{X} \times \mathcal{Y} \times \mathcal{W} \to \mathbb{R}$ is a function of the input $\mathbf{x}$, the true value $\mathbf{y}$ and the networks parameters, i.e. the weights $\mathbf{w}$. Neural networks can be trained by looking at the loss obtained for each iteration to fine tune the weights. *Backpropagation* is an algorithm for this, evaluating the gradient $\frac{\partial L}{\partial \mathbf{w}}$.

## Dense Layer

A fixed layer in the network $f = \sigma \circ \ell : \mathbb{R}^m \to \mathbb{R}^n$ was given in the assignment description [1]. The function consists of the linear part $\ell : \mathbb{R}^m \to \mathbb{R}^n$ and the activation function $\sigma : \mathbb{R}^m \to \mathbb{R}^n$. The linear part consists of the weight matrix $\mathbf{W} \in \mathbb{R}^{n \times m}$ and a bias vector $\mathbf{b} \in \mathbb{R}^n$, both trainable parameters. When a data point $\mathbf{x} \in \mathbb{R}^m$ enters the given layer the linear function outputs the values $\mathbf{z} = \ell(\mathbf{x}) = \mathbf{W}\mathbf{x} + \mathbf{b} \in \mathbb{R}^n$. For all $i$ it holds that

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

### E1. Derivation of Backpropagation Expressions for $\frac{\partial L}{\partial \mathbf{x}}$, $\frac{\partial L}{\partial \mathbf{W}}$ and $\frac{\partial L}{\partial \mathbf{b}}$

The first step in deriving the equations for propagation was to express $\frac{\partial L}{\partial \mathbf{x}}$ in terms of $\frac{\partial L}{\partial \mathbf{z}}$ and $\mathbf{W}$. Starting from equation 1 and differentiating w.r.t. $x_j$ gives for all $i$ and $j$:

$$\frac{\partial z_i}{\partial x_j} = \frac{\partial}{\partial x_j}\left(\sum_{j=1}^{m} W_{ij} x_j + b_i\right) = W_{ij} \implies \frac{\partial z_l}{\partial x_i} = W_{li} \tag{2}$$

By using the chain rule and inserting equation 2 the partial derivative $\frac{\partial L}{\partial x_i}$ could be formulated:

$$\frac{\partial L}{\partial x_i} = \sum_{l=1}^{n} \frac{\partial L}{\partial z_l}\frac{\partial z_l}{\partial x_i} = \sum_{l=1}^{n} \frac{\partial L}{\partial z_l} W_{li} = \left(\frac{\partial L}{\partial \mathbf{z}}\right)^T \mathbf{W}^{(i)} \tag{3}$$

where $\mathbf{W}^{(i)} \in \mathbb{R}^{n \times 1}$ is the i:th column of the weight matrix and $\left(\frac{\partial L}{\partial \mathbf{z}}\right)^T = [\frac{\partial L}{\partial z_1}, ..., \frac{\partial L}{\partial z_n}] \in \mathbb{R}^{1 \times n}$. The expression for $\frac{\partial L}{\partial \mathbf{x}} \in \mathbb{R}^{m \times 1}$ could then be expressed in matrix form:

$$\frac{\partial L}{\partial \mathbf{x}} = \mathbf{W}^T \frac{\partial L}{\partial \mathbf{z}} \tag{4}$$

The partial derivative $\frac{\partial L}{\partial \mathbf{W}}$ and $\frac{\partial L}{\partial \mathbf{b}}$ could then be derived in a similar way. Starting from equation 1 and differentiating w.r.t. $W_{ij}$ and $b_i$ respectively gives:

$$\frac{\partial z_i}{\partial W_{ij}} = \frac{\partial}{\partial W_{ij}} \left( \sum_{j=1}^{m} W_{ij} x_j + b_i \right) = x_j \implies \frac{\partial z_l}{\partial W_{ij}} = x_j \delta_{il} \tag{5}$$

$$\frac{\partial z_i}{\partial b_i} = \frac{\partial}{\partial b_i} \left( \sum_{j=1}^{m} W_{ij} x_j + b_i \right) = 1 \implies \frac{\partial z_l}{\partial b_i} = \delta_{il} \tag{6}$$

where $\delta_{il}$ is the Kronecker delta function which means that $\delta_{il}$ is 1 if $i = l$ and 0 otherwise. Again, by using the chain rule and inserting equation 5 and 6, the partial derivative $\frac{\partial L}{\partial W_{ij}}$ and $\frac{\partial L}{\partial b_i}$ could be formulated:

$$\frac{\partial L}{\partial W_{ij}} = \sum_{l=1}^{n} \frac{\partial L}{\partial z_l} \frac{\partial z_l}{\partial W_{ij}} = \sum_{l=1}^{n} \frac{\partial L}{\partial z_l} x_j \delta_{il} = \sum_{i=1}^{n} \frac{\partial L}{\partial z_i} x_j \tag{7}$$

$$\frac{\partial L}{\partial b_i} = \sum_{l=1}^{n} \frac{\partial L}{\partial z_l} \frac{\partial z_l}{\partial b_i} = \sum_{l=1}^{n} \frac{\partial L}{\partial z_l} \delta_{il} = \sum_{i=1}^{n} \frac{\partial L}{\partial z_i} \tag{8}$$

The expression for $\frac{\partial L}{\partial \mathbf{W}} \in \mathbb{R}^{n \times m}$ and $\frac{\partial L}{\partial \mathbf{b}} \in \mathbb{R}^{n}$ could then be expressed in matrix form:

$$\frac{\partial L}{\partial \mathbf{W}} = \frac{\partial L}{\partial \mathbf{z}} \mathbf{x}^T \tag{9}$$

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

### E2. Derivation of propagation Expressions for Z, $\frac{\partial L}{\partial \mathbf{X}}$, $\frac{\partial L}{\partial \mathbf{W}}$ and $\frac{\partial L}{\partial \mathbf{b}}$

A neural network training method is to compute the gradients w.r.t. $N$ elements in a batch instead of just one. For the dense layer given in the assignment description [1], there are $N$ inputs $\mathbf{X} = (\mathbf{x}^{(1)}\ \mathbf{x}^{(2)}\ ...\ \mathbf{x}^{(N)})$. The output we aim to compute is denoted $\mathbf{Z}$ and it is given by:

$$\mathbf{Z} = (\mathbf{z}^{(1)}\ \mathbf{z}^{(2)}\ ...\ \mathbf{z}^{(N)}) = (\mathbf{W}\mathbf{x}^{(1)} + \mathbf{b} \quad \mathbf{W}\mathbf{x}^{(2)} + \mathbf{b} \quad ... \quad \mathbf{W}\mathbf{x}^{(N)} + \mathbf{b}) \tag{11}$$

A matrix expression of $\mathbf{Z}$ could be formulated from equation 11:

$$\mathbf{Z} = \mathbf{W} \cdot [\mathbf{x}^{(1)}\ \mathbf{x}^{(2)}\ \mathbf{x}^{(N)}] + \mathbf{b} \cdot \mathbf{1}^T = \mathbf{W}\mathbf{X} + \mathbf{b}\mathbf{1}^T \tag{12}$$

where $\mathbf{1}^T = [1\ 1\ ...\ 1] \in \mathbb{R}^{1 \times N}$ adds the bias to all elements.

The next step in the derivation of the equations for propagation was to express $\mathbf{Z}$, $\frac{\partial L}{\partial \mathbf{X}}$, $\frac{\partial L}{\partial \mathbf{W}}$ and $\frac{\partial L}{\partial \mathbf{b}}$ in terms of $\frac{\partial L}{\partial \mathbf{Z}}$, $\mathbf{W}$ and $\mathbf{X}$.

When propagating to $\mathbf{X}$ we compute:

$$\frac{\partial L}{\partial \mathbf{X}} = \begin{pmatrix} \frac{\partial L}{\partial \mathbf{x}^{(1)}} & \frac{\partial L}{\partial \mathbf{x}^{(2)}} & \dots & \frac{\partial L}{\partial \mathbf{x}^{(N)}} \end{pmatrix} \tag{13}$$

Insertion of the obtained expression for each element $\frac{\partial L}{\partial \mathbf{x}}$ from equation 4 in equation 11 gives:

$$\frac{\partial L}{\partial \mathbf{X}} = \begin{pmatrix} \mathbf{W}^T \frac{\partial L}{\partial \mathbf{z}^{(1)}} & \mathbf{W}^T \frac{\partial L}{\partial \mathbf{z}^{(2)}} & \dots & \mathbf{W}^T \frac{\partial L}{\partial \mathbf{z}^{(N)}} \end{pmatrix} = \mathbf{W}^T \frac{\partial L}{\partial \mathbf{Z}} \tag{14}$$

Then expressions for $\frac{\partial L}{\partial \mathbf{W}}$ and $\frac{\partial L}{\partial \mathbf{b}}$ was derived. Using the chain rule w.r.t. each element in every $\mathbf{z}^{(i)}$ and summing it up gave:

$$\frac{\partial L}{\partial W_{ij}} = \sum_{l=1}^{N} \sum_{k=1}^{n} \frac{\partial L}{\partial z_k^{(l)}} \frac{\partial z_k^{(l)}}{\partial W_{ij}} = \sum_{l=1}^{N} \sum_{i=1}^{n} \frac{\partial L}{\partial z_i^{(l)}} x_j^{(l)} \implies \frac{\partial L}{\partial \mathbf{W}} = \sum_{l=1}^{N} \frac{\partial L}{\partial \mathbf{z}^{(l)}} \mathbf{x}^{(l)^T} = \frac{\partial L}{\partial \mathbf{Z}} \mathbf{X}^T \tag{15}$$

$$\frac{\partial L}{\partial b_i} = \sum_{l=1}^{N} \sum_{j=1}^{n} \frac{\partial L}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial b_i} = \sum_{l=1}^{N} \sum_{i=1}^{n} \frac{\partial L}{\partial z_i^{(l)}} \implies \frac{\partial L}{\partial \mathbf{b}} = \sum_{l=1}^{N} \frac{\partial L}{\partial \mathbf{z}^{(l)}} \tag{16}$$

Two Matlab function `layers/fully_connected_forward.m` and `layers/fully_connected_backw` was then implemented using the derived vectorised expressions. The most important parts of the functions are presented below:

```
function Z = fully_connected_forward(X, W, b)
...
    b = repmat(b, 1, batch);
    Z = W*X+b;
end
```

```
function [dldX, dldW, dldb] = fully_connected_backward(X, dldZ, W, b)
...
    dldX = transpose(W)*dldZ;
    dldX = reshape(dldX, sz);
    dldW = dldZ*transpose(X);
    dldb = sum(dldZ, 2);
end
```

### ReLU

The recrified linear unit (ReLU) function is an activation function commonly used as a non-linearity for training deep neural networks. It was early (1960s) used for visual feature

extraction. Compared to other activation function such as the sigmoid function, the ReLU function allow faster and cheaper training of deep neural networks and on large and complex datasets.

The ReLU function maps $x_i$ to $z_i$ and it is defined as follows:

$$\mathbb{R} \to \mathbb{R}: \quad x_i \mapsto z_i := \max(x_i, 0) \tag{17}$$

**E3. Derivation of the propagation Expression for $\frac{\partial L}{\partial x_i}$**

Starting from equation 17 and differentiating w.r.t. $x_i$ gives, for all i:

$$\frac{\partial z_i}{\partial x_i} = \begin{cases} 0, & \text{if } x_i < 0 \\ 1, & \text{if } x_i > 0 \end{cases} \tag{18}$$

The propagation expression for $\frac{\partial L}{\partial x_i}$ in terms of $\frac{\partial L}{\partial z_i}$ could then be formulated:

$$\frac{\partial L}{\partial x_i} = \frac{\partial L}{\partial z_i}\frac{\partial z_i}{\partial x_i} = \begin{cases} 0, & \text{if } x_i < 0 \\ \frac{\partial L}{\partial z_i}, & \text{if } x_i > 0 \end{cases} \tag{19}$$

Two Matlab function `layers/relu_forward.m` and `layers/relu_backward.m` was then implemented using the derived expression. The functions are presented below:

```
function Z = relu_forward(X)
    Z = max(X,0);
end
```

```
function dldX = relu_backward(X, dldZ)
    dldX = (X > 0).*dldZ;
end
```

## Softmax Loss

The softmax loss function is a generalization of the logistic function to multiple dimensions. It takes as input a vector $\mathbf{x}$ of $m$ real numbers, and normalizes it into a probability distribution consisting of $m$ probabilities.

It was given in the assignment description [1] that there is vector $\mathbf{x} = [x_1, ..., x_m]^T \in \mathbb{R}^m$ and a goal to classify inputs in $m$ different classes. $x_i$ is a score for the particular class $i$. Larger $x_i$ corresponds to a higher score. The softmax function interprets the scores $x_i$ as probabilitie $z_i$ for the class $i$ as defined below:

$$z_i = \frac{e^{x_i}}{\sum_{j=1}^{m} e^{x_j}} \tag{20}$$

It was supposed that the ground truth class was $c$, hence the loss $L$ could be minimized as the negative log likelihood shown below:

$$L(\mathbf{x}, c) = -\log(z_c) = -\log\left(\frac{e^{x_c}}{\sum_{j=1}^{m} e^{x_j}}\right) = -x_c + \log\left(\sum_{j=1}^{m} e^{x_j}\right) \tag{21}$$

**E4. Computation of $\frac{\partial L}{\partial x_i}$**

The softmax loss function was then used to compute the expression for $\frac{\partial L}{\partial x_i}$ in terms of $z_i$ for the final layer of the neural network.

Starting from equation 21 the expression for $\frac{\partial L}{\partial x_i}$ in terms of $z_i$ can be formulated:

$$\frac{\partial L}{\partial x_i} = \frac{\partial}{\partial x_i}\left(-x_c + \log\left(\sum_{j=1}^{m} e^{x_j}\right)\right) = -\delta_{ci} + \frac{e^{x_i}}{\sum_{j=1}^{m} e^{x_j}} = z_i - \delta_{ci} \tag{22}$$

The layer was implemented in the functions `layers/softmaxloss_forward.m` and `layers/softmaxloss_backward.m` and the relevant parts of the codes is presented below:

```
function L = softmaxloss_forward(x, labels)
    ...
    idx = sub2ind(size(x), labels', 1:batch);
    L = -x(idx) + log(sum(exp(x)));
    L = sum(L, 'all')/batch;
end
```

```
function dldx = softmaxloss_backward(x, labels)
    ...
    labels = double(labels);
    dci = full(ind2vec(labels', features));
    dldx = (softmax(x)-dci)/batch;
end
```

## Training a Neural Network

A neural network can be trained, with the aim to minimize the loss function:

$$L(\mathbf{w}) = \frac{1}{N}\sum_{i=1}^{N} L(x^{(i)}, y^{(i)}; \mathbf{w}) \tag{23}$$

where $x^{(i)}$ is the input $y^{(i)}$ the ground truth for the class $i$ and $\mathbf{w}$ all parameters of the network. $L(x^{(i)}, y^{(i)}; \mathbf{w})$ is the loss for a single example and $N$ is the number of elements.

**E5. Implementation of Gradient Descent with Momentum**

The training of a neural network can be done much faster and cheaper if the loss function is evaluated over less elements. A method commonly used is *gradient descent with momentum*. Batches of $n << N$ elements are used and the gradient estimations are averaged over time. The method is defined:

$$\mathbf{m}_n = \mu \mathbf{m}_{n-1} + (1-\mu)\frac{\partial L}{\partial \mathbf{w}} \tag{24}$$

$$\mathbf{w}_{n+1} = \mathbf{w}_n - \alpha \mathbf{m}_n \tag{25}$$

$\mathbf{m}_n$ is a moving average of gradient estimations and $\mu \in ]0,1[$ a hyperparameter that controlls the smoothness.

The gradient descent with momentum was implemented in `training.m`. The relevant code implementing equation 24 and 25 respectively is presented below:

```
function net = training(net, x, labels, x_val, labels_val, opts)
    ...
    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));
end
```

## Classifying Handwritten Digits

In this section a neural network that was given in the assignment description [1] was used to classify handwritten digits in the MNIST dataset. The network reaches around 98% accuracy on the test set.

The first task was to plot the filters that the first convolutional layer learns. The results are presented in Figure 1 below.

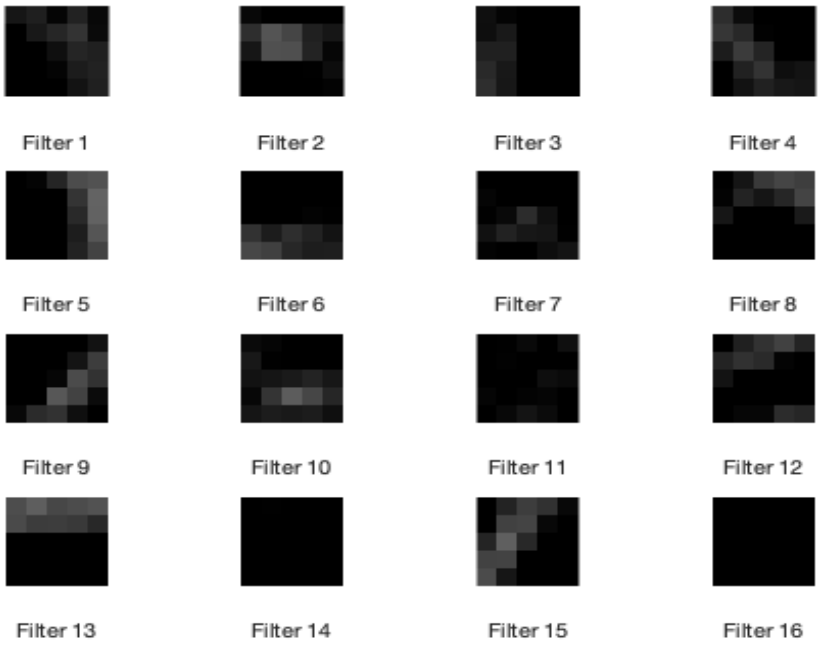


**Figure 1:** Visualization of the filters that the first convolutional layer learns

The filters plotted in Figure 1 indicates that the network is initially trained to detect some lines or dots, even though they are a wage in the first filtering.

The next task was to plot a few images that were misclassified. The results are shown in Figure 2.

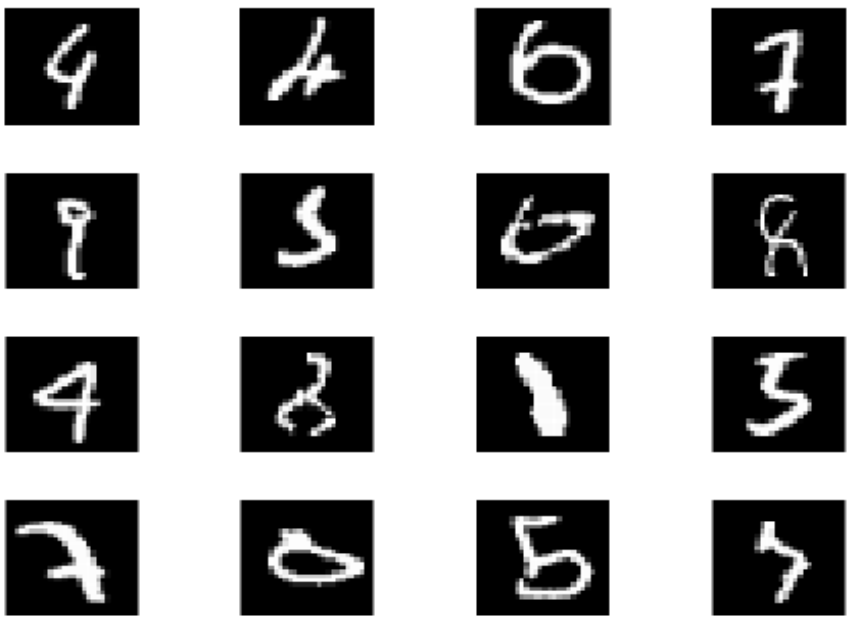

**Figure 2:** 16 examples of misclassified digits

The findings presented in Figure 2 was quite expected. Some of the digits are impossible even for me to interpret. That the neural network could not manage to classify it correctly was therefor not surprising.

Then the confusion matrix was plotted for the predictions on the test set, shown in Figure 3.

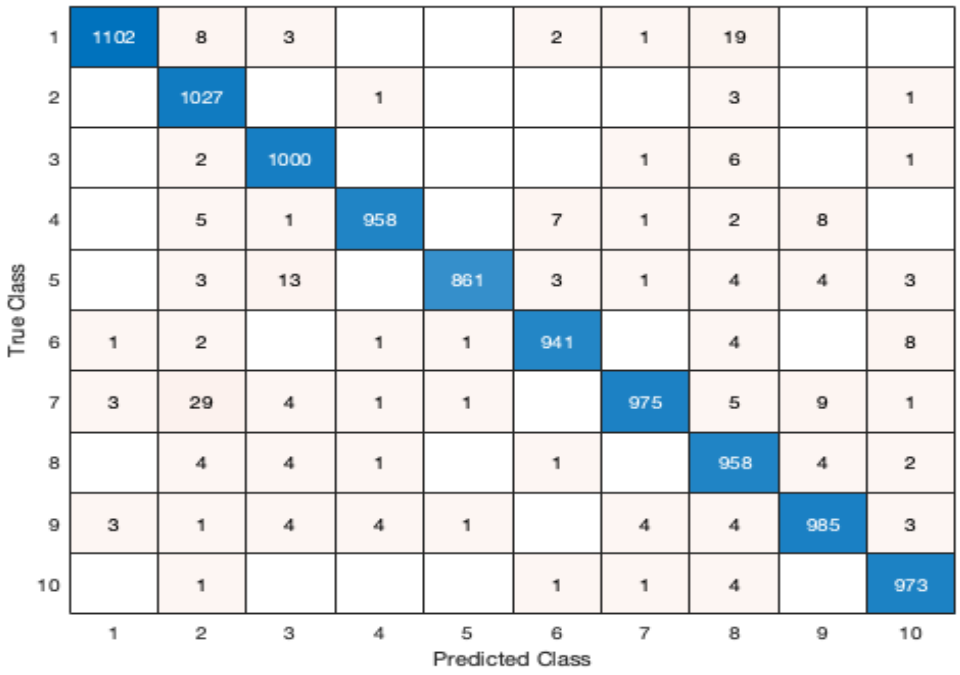


**Figure 3:** Confusion matrix for the predictions on the test set. Note that 10 represents digit 0

The confusion matrix show that some missclassifications were more common than others. The predicted class *2* for the true class *7* was the most frequent missclassification (29 examples). The digits 2 and 7 could look quite similar and the same reasoning holds for most of the other examples with high missclassification rates, which make the results seem reasonable.

Going forward, the precision and recall could be calculated for all digits using the confusion matrix. The results are presented in Table 1 .

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| Precision | 0.9808 | 0.9937 | 0.9492 | 0.9718 | 0.9917 | 0.9965 | 0.9853 | 0.9909 | 0.9495 | 0.9752 |
| Recall | 0.9929 | 0.9709 | 0.9952 | 0.9901 | 0.9756 | 0.9652 | 0.9823 | 0.9484 | 0.9836 | 0.9762 |

**Table 1:** Precision and recall for all digits 0-10

Precision measures how many of the "guesses" on a certain digit that were correctly guessed. I.e. if there are a total of 100 predictions: digit *2* and there were only 10 true digit *2* in the data set, the precision would be 0.10.

Recall measures the fraction of relevant instances retrieved. It means that if there are 100 true digits *2* and prediction: $n = 20$ numbers of digits *2*, the recall would be 0.2. Both precision and recall should therefore be as high as possible. Table 1 show that the neural network performed well for all classes, but it performed better predicting some digits than

others. The neural network performed very well predicting class *0* and a bit worse for class *8* and *9*.

The number of parameters for all layers in the network is presented in Table 4.

| Layer | Layer type | Inputs | Outputs | Filter | Bias | **Number of parameters** |
|---|---|---|---|---|---|---|
| 1 | Input | | | | | 0 |
| 2 | **Convolutional** | 1 | 16 | [5×5 × 1] | [16×1] | (5×5 × 1 + 1)16 = 416 |
| 3 | ReLU | | | | | 0 |
| 4 | Maxpooling | | | | | 0 |
| 5 | **Convolutional** | 16 | 16 | [5×5 × 16] | [16×1] | (5×5 × 16 + 1)16 = 6416 |
| 6 | ReLU | | | | | 0 |
| 7 | Maxpooling | | | | | 0 |
| 8 | **Fully connected** | 784 | 10 | | [10×1] | (784+1)10=7850 |
| 9 | Softmaxloss | | | | | 0 |

**Table 2:** Number of parameters for all layers in the network. Input/ output channels, filter size and bias size are included for the relevant layers, i.e layers with parameters

## Classifying Tiny Images

In this section a simple network that could be trained on the CIFAR10 dataset was used. The baseline network had an accuracy of 48 % after training on 5000 iterations. The task was to elaborate on the existing neural network to improve the accuracy.

The CIFAR10 dataset consists of 60.000 small images. The classes in the dataset are: `{airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck}` corresponding to numbers `{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}`.

The first thing I tried was to delete some of the hidden layers to see what happened. More layers does not necessarily need to improve the accuracy on the test data. Recent trends in neural network image classification competitions however indicates that performance increase with number of layers. Therefore I first tried to add another convolutional-, maxpooling- and ReLU layer, followed by additionally added layers. Out of curiosity I also tried to train the network more by changing the size of the steps in the step method. I halved the learning rate and doubled the number of iterations. I also tried to double the number of batches. The accuracy obtained for each model is presented in Table 3.

| Model | Network changes | Nbr of layers | Accuracy |
|---|---|---|---|
| 0 | | 8 | 48 % |
| 1 | first ReLU layer deleted | 7 | 48 % |
| 2 | both ReLU layers deleted | 6 | 45 % |
| 3 | + convolutional layer with $[5 \times 5 \times 16]$ filter maxpooling layer and ReLU layer | 11 | 54 % |
| 4 | model 3 + fully connected layer | 12 | 53 % |
| 5 | model 3 + ReLU and fully connected layer | 13 | 54 % |
| 6 | + 2 convolutional layer with $[5 \times 5 \times 16]$ filter 2 maxpooling layer and 2 ReLU layer | 14 | 48 % |
| 6 | halve learning rate & double nbr of iterations | 8 | 56 % |
| 7 | double nbr of batches | 8 | 55 % |

**Table 3:** Changes to the baseline network for each tested model and achieved accuracy on test data

The results presented in Table 4 show that the first ReLU layer did not necessarily contribute in terms of accuracy on test data. However did the accuracy decrease when deleting both of the hidden ReLU layers. The results further show that it was beneficial to deepen the network by adding one additional convolutional layer, as all of those models showed improvements in accuracy. When there were 2 convolutional layers with $[5 \times 5 \times 16]$ filters, the accuracy however went down again. The difference between the training and validation accuracy was smaller for model 3 than model 4, which indicates that model 3 show less tendency to be overfitted to the test data. It can however not be ruled out that model 3 showed performed better than model 4 only by coincidence as the tests was not run several times and the models performed similar.

It was obvious that smaller step size and more batches increased the accuracy on the test data, it outperformed the baseline network in terms of validation accuracy. The difference between the accuracy on the training and the validation data was quite large which implicates that the model may be a overfit using this parameters. Another trade-off in terms of cost was prominent, it took significantly longer time to train the network using these parameters.

If I were to further increase the accuracy outside the scope of this project, I would test to implement a Leaky ReLU activation function. Due to deadline and time pressure it was unfortunately not possible for me to try it within this project. Other changes that could increase the accuracy would be to change number of parameters, add more layers and change the filter sizes. This is what I would have tried if I had more time to run tests.

Model 3 was chosen as the *final* model since it was a good compromise between accuracy and cost, and it did not seem to be overfitted to the test data. The number of parameters for all layers in the *Final* network is presented in Table 4.

| Layer | Layer type | Inputs | Outputs | Filter | Bias | **Number of parameters** |
|---|---|---|---|---|---|---|
| 1 | Input | | | | | 0 |
| 2 | **Convolutional** | 1 | 16 | [5×5 × 3] | [16×1] | (5×5 × 3 + 1)16 = 1216 |
| 3 | ReLU | | | | | 0 |
| 4 | Maxpooling | | | | | 0 |
| 5 | **Convolutional** | 16 | 16 | [5×5 × 16] | [16×1] | (5×5 × 16 + 1)16 = 6416 |
| 6 | ReLU | | | | | 0 |
| **7** | Maxpooling | | | | | 0 |
| **8** | **Convolutional** | 16 | 16 | [5×5 × 16] | [16×1] | (5×5 × 16 + 1)16 = 6416 |
| **9** | ReLU | | | | | 0 |
| 10 | **Fully connected** | 1024 | 10 | | [10×1] | (1024+1)10=10250 |
| 11 | Softmaxloss | | | | | 0 |

**Table 4:** Number of parameters for all layers in the *Final* model network. Layer 7, 8 and 9 were added to the baseline network and layer 10 changed. Input/ output channels, filter size and bias size are included for the relevant layers, i.e layers with parameters

Note that the inputs to the fully connected layer was adjusted from 4096 to 1024, generating less parameters. Less parameters are good to avoid overfitting but it can also lead to lower accuracy.

The filters that the first convolutional layer in the *Final* model learns were plotted. The results are presented in Figure 4 below.

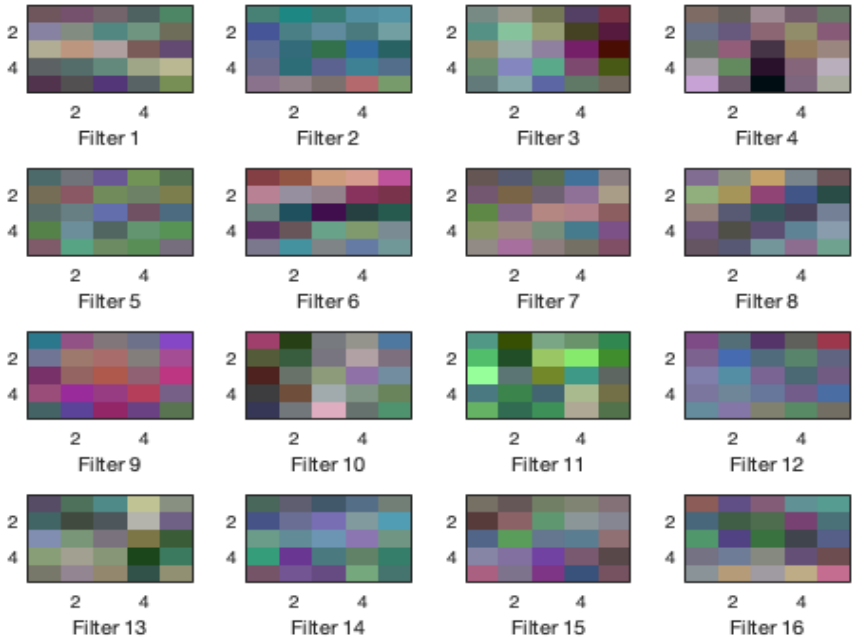


**Figure 4:** Visualization of the filters that the first convolutional layer learns

The filters plotted in Figure 1 indicates that the network is initially trained to find some kind of variations in color and intensity. The filters plotted could be considered wage in this first filtering.

The next task was to plot a few images that were missclassified. The results are shown in

Figure 5.

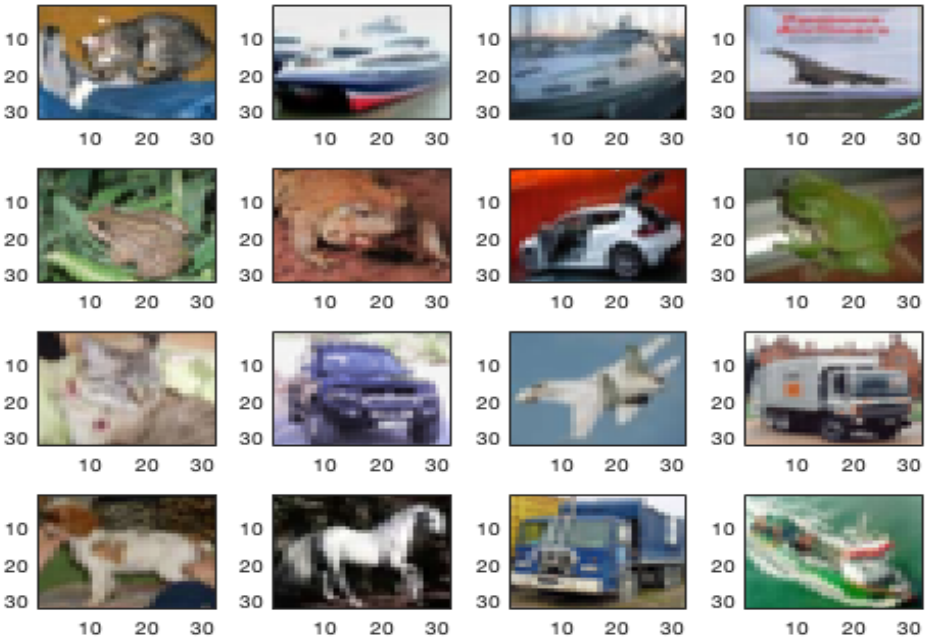

**Figure 5:** 16 examples of misclassified images

The findings presented in Figure 2 was quite expected. Some of the images are hard to interpret and it is obvious that it could be mixed up. That the neural network could not manage to classify it correctly was therefor not surprising.

Then the confusion matrix was plotted for the predictions on the test set, shown in Figure 6.

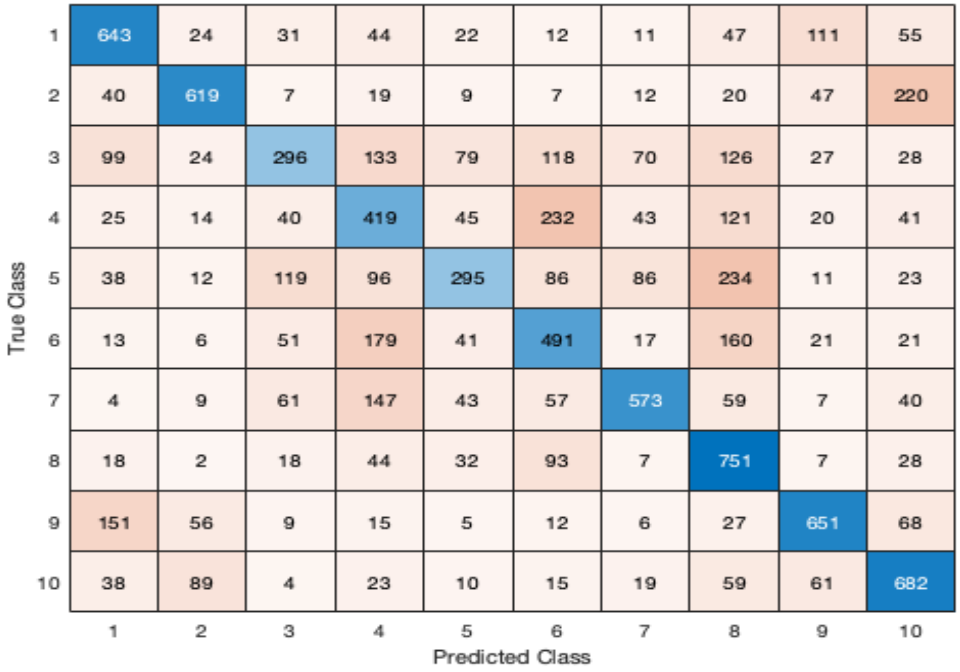


**Figure 6:** Confusion matrix for the predictions on the test set.

The confusion matrix show that some classes were more commonly mixed up than others. The precision and recall could be calculated for all classes using the confusion matrix. The results are presented in Table 1.

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| Precision | 0.6015 | 0.7240 | 0.4654 | 0.3744 | 0.5077 | 0.4372 | 0.6789 | 0.4682 | 0.6760 | 0.5655 |
| Recall | 0.6430 | 0.6190 | 0.2960 | 0.4190 | 0.2950 | 0.4910 | 0.5730 | 0.7510 | 0.6510 | 0.6820 |

**Table 5:** Precision and recall for all classes

Table 5 show that the neural network performed better predicting some classes than others. The neural network classified class 9 the best and class 3 the worst.

## References

[1] FMAN45 Machine Learning *Assignment Instructions: Assignment 3* Spring 2021. LTH. Downloaded: 2021-05-04. https://canvas.education.lu.se/courses/11000/files/1221084?wrap=1