Home Assignment 3

Arvid Gramer

FMAN45 Machine Learning
Lunds Tekniska Högskola, Lunds Universitet
May 2023

1 Common Layers and Backpropagation

Exercise 1: *Derive expressions for* $\partial L/\partial \boldsymbol{x}, \partial L/\partial \boldsymbol{W}, \partial L/\partial \boldsymbol{b}$ *in terms of* $\partial L/\partial \boldsymbol{y}$, $\boldsymbol{W}, \boldsymbol{x}$.

Before we begin we map out the matrix multiplication that we are investigating, namely $\mathbf{y} = \mathbf{W}\mathbf{x} + \mathbf{b}$:

$$\begin{bmatrix} y_1 \\ \vdots \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} W_{1,1} & W_{1,2} & \dots & W_{1,m} \\ W_{2,1} & \ddots & & W_{2,m} \\ \vdots & & \ddots & \vdots \\ W_{n,1} & W_{n,2} & \dots & W_{n,m} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix} + \begin{bmatrix} b_1 \\ \vdots \\ \vdots \\ b_n \end{bmatrix} \tag{1}$$

This more explicit rewrite is to help keep track of dimensions and matrix multiplications. The elongated $\mathbf{y}$ and $\mathbf{b}$ vectors are to help distinguish the two sizes n and m. $\frac{\partial L}{\partial \mathbf{y}}$ is, in the same manner a column vector with each row i having value $\frac{\partial L}{\partial y_i}$

We begin with $\frac{\partial L}{\partial \mathbf{x}}$, and using the chain rule as hinted in the description we have:

$$\frac{\partial L}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial L}{\partial x_1} \\ \vdots \\ \frac{\partial L}{\partial x_m} \end{bmatrix} = \begin{bmatrix} \sum_{l=1}^{n} \frac{\partial L}{\partial y_l} \frac{\partial y_l}{\partial x_1} \\ \sum_{l=1}^{n} \frac{\partial L}{\partial y_l} \frac{\partial y_l}{\partial x_2} \\ \vdots \\ \sum_{l=1}^{n} \frac{\partial L}{\partial y_l} \frac{\partial y_l}{\partial x_m} \end{bmatrix} \tag{2}$$

Investigating the derivative of y_l with respect to x_i we find that, since $y_i = \sum_{j=1}^{m} W_{i,j} x_j + b_i$, $\frac{\partial y_l}{\partial x_i} = W_{l,i}$. This now yields:

$$\begin{bmatrix} \sum_{l=1}^{n} \frac{\partial L}{\partial y_l} W_{l,1} \\ \sum_{l=1}^{n} \frac{\partial L}{\partial y_l} W_{l,2} \\ \vdots \\ \sum_{l=1}^{n} \frac{\partial L}{\partial y_l} W_{l,m} \end{bmatrix}. \tag{3}$$

With matrix notation this can be re-expressed as

$$\frac{\partial L}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial L}{\partial y_1} & \frac{\partial L}{\partial y_2} & \dots & \frac{\partial L}{\partial y_n} \end{bmatrix} \begin{bmatrix} W_{1,1} & W_{1,2} & \dots & W_{1,m} \\ W_{2,1} & \ddots & & W_{2,m} \\ \vdots & & \ddots & \vdots \\ W_{n,1} & W_{n,2} & \dots & W_{n,m} \end{bmatrix} = \left(\frac{\partial L}{\partial \mathbf{y}} \right)^{\top} \mathbf{W}. \tag{4}$$

We continue with $\frac{\partial L}{\partial \mathbf{W}}$. Expanding the full matrix expression we get:

$$\frac{\partial L}{\partial \mathbf{W}} = \begin{bmatrix} \frac{\partial L}{\partial W_{1,1}} & \dots & \frac{\partial L}{\partial W_{1,m}} \\ \vdots & \ddots & \vdots \\ \frac{\partial L}{\partial W_{n,1}} & \dots & \frac{\partial L}{\partial W_{n,m}} \end{bmatrix} = \begin{bmatrix} \sum_{l=1}^{n} \frac{\partial L}{\partial y_l} \frac{\partial y_l}{\partial W_{1,1}} & \dots & \sum_{l=1}^{n} \frac{\partial L}{\partial y_l} \frac{\partial y_l}{\partial W_{1,m}} \\ \vdots & \ddots & \vdots \\ \sum_{l=1}^{n} \frac{\partial L}{\partial y_l} \frac{\partial y_l}{\partial W_{n,1}} & \dots & \sum_{l=1}^{n} \frac{\partial L}{\partial y_l} \frac{\partial y_l}{\partial W_{n,m}} \end{bmatrix} \tag{5}$$

Again, with $y_i = \sum_{j=1}^{m} W_{i,j} x_j + b_i$:

$$\frac{\partial y_l}{\partial W_{p,q}} = \begin{cases} x_q & \text{if } l = p \\ 0 & \text{else} \end{cases} \tag{6}$$

This gives

$$\frac{\partial L}{\partial \mathbf{W}} = \begin{bmatrix} \frac{\partial L}{\partial y_1} x_1 & \frac{\partial L}{\partial y_1} x_2 & \dots & \frac{\partial L}{\partial y_1} x_m \\ \frac{\partial L}{\partial y_2} x_1 & \ddots & & \frac{\partial L}{\partial y_2} x_m \\ \vdots & & \ddots & \vdots \\ \frac{\partial L}{\partial y_n} x_1 & \frac{\partial L}{\partial y_n} x_2 & \dots & \frac{\partial L}{\partial y_n} x_m \end{bmatrix} = \frac{\partial L}{\partial \mathbf{y}} \mathbf{x}^\top \tag{7}$$

Deriving $\frac{\partial L}{\partial \mathbf{b}}$ is less complicated. Since $y_i = \sum_{j=1}^{m} W_{i,j} x_j + b_i$, the inner part of the chain becomes:

$$\frac{\partial y_l}{\partial b_i} = \begin{cases} 1 & \text{if } l = i \\ 0 & \text{else} \end{cases} \tag{8}$$

This gives:

$$\frac{\partial L}{\partial \mathbf{b}} = \begin{bmatrix} \frac{\partial L}{\partial b_i} \\ \vdots \\ \frac{\partial L}{\partial b_n} \end{bmatrix} = \begin{bmatrix} \frac{\partial L}{\partial y_i} \\ \vdots \\ \frac{\partial L}{\partial y_n} \end{bmatrix} = \frac{\partial L}{\partial \mathbf{y}} \tag{9}$$

Exercise 2: *Derive expressions for* $\boldsymbol{Y}$, $\frac{\partial L}{\partial \boldsymbol{X}}, \frac{\partial L}{\partial \boldsymbol{W}}, \frac{\partial L}{\partial \boldsymbol{b}}$ *in terms of* $\frac{\partial L}{\partial \boldsymbol{Y}}, \boldsymbol{W}, \boldsymbol{X}, \boldsymbol{b}$ *Write Matlab code that implements these vectorised expressions. Complete the functions* `fully_connected_backward.m` *and* `fully_connected_forward.m`.

With

$$\begin{cases} \mathbf{X} = \begin{bmatrix} \mathbf{x}^{(1)} & \dots & \mathbf{x}^{(N)} \end{bmatrix} \\ \mathbf{Y} = \begin{bmatrix} \mathbf{y}^{(1)} & \dots & \mathbf{y}^{(N)} \end{bmatrix} = \begin{bmatrix} \mathbf{W}\mathbf{x}^{(1)} + \mathbf{b} & \dots & \mathbf{W}\mathbf{x}^{(N)} + \mathbf{b} \end{bmatrix} \end{cases} \tag{10}$$

it is straight forward that $\mathbf{Y} = \mathbf{W}\mathbf{X} + \mathbf{b}$. Since

$$\frac{\partial L}{\partial \mathbf{X}} = \begin{bmatrix} \frac{\partial L}{\partial \mathbf{x}^{(1)}} & \dots & \frac{\partial L}{\partial \mathbf{x}^{(N)}} \end{bmatrix} = \begin{bmatrix} \left(\frac{\partial L}{\partial \mathbf{y}^{(1)}} \right)^\top \mathbf{W} & \dots & \left(\frac{\partial L}{\partial \mathbf{y}^{(N)}} \right)^\top \mathbf{W} \end{bmatrix} \tag{11}$$

we see that $\frac{\partial L}{\partial \mathbf{X}} = \frac{\partial L}{\partial \mathbf{Y}^\top} \mathbf{W}$. Regarding $\frac{\partial L}{\partial \mathbf{W}}$, if we manipulate equation (5) from exercise 1 to fit the new batch-wise input we get:

$$\frac{\partial L}{\partial \mathbf{W}} = \begin{bmatrix} \sum_{l=1}^{N} \frac{\partial L}{\partial y_1^{(l)}} x_1^{(l)} & \sum_{l=1}^{N} \frac{\partial L}{\partial y_1^{(l)}} x_2^{(l)} & \dots & \sum_{l=1}^{N} \frac{\partial L}{\partial y_1^{(l)}} x_m^{(l)} \\ \sum_{l=1}^{N} \frac{\partial L}{\partial y_2^{(l)}} x_1^{(l)} & \ddots & & \sum_{l=1}^{N} \frac{\partial L}{\partial y_2^{(l)}} x_m^{(l)} \\ \vdots & & \ddots & \vdots \\ \sum_{l=1}^{N} \frac{\partial L}{\partial y_n^{(l)}} x_1^{(l)} & \sum_{l=1}^{N} \frac{\partial L}{\partial y_n^{(l)}} x_2^{(l)} & \dots & \sum_{l=1}^{N} \frac{\partial L}{\partial y_n^{(l)}} x_m^{(l)} \end{bmatrix} = \frac{\partial L}{\partial \mathbf{Y}} \mathbf{X}^\top. \tag{12}$$

For $\frac{\partial L}{\partial \mathbf{b}}$, we note that just as in exercise 1 we can use equation (8) and get:

$$\frac{\partial L}{\partial \mathbf{b}} = \begin{bmatrix} \frac{\partial L}{\partial b_1} \\ \vdots \\ \frac{\partial L}{\partial b_n} \end{bmatrix} = \begin{bmatrix} \sum_{l=1}^{N} \frac{\partial L}{\partial y_i^{(l)}} \\ \vdots \\ \sum_{l=1}^{N} \frac{\partial L}{\partial y_n^{(l)}} \end{bmatrix} = \sum_{l=1}^{N} \frac{\partial L}{\partial \mathbf{y}^{(l)}}.?? \tag{13}$$

We then proceed to implement this in the Matlab-scripts `fully_connected_backward.m` and `fully_connected_forward.m`. The lines added to `fully_connected_backward.m` are:

```
% Implement it here.
% note that dldX should have the same size as X, so use reshape
% as suggested.
dldX = (dldY'*W)';
dldX = reshape(dldX, sz);
dldW = dldY*X';
dldb = sum(dldY,2);
```

As for `fully_connected_forward.m`, the only line needed is essentially to implement $\mathbf{Y} = \mathbf{WX} + \mathbf{b}$:

```
Y = W*X+b;
```

Exercise 3: *Derive and implement the back-propagation expression for* $\frac{\partial L}{\partial x_i}$ *in terms of* $\frac{\partial L}{\partial y_i}$*. Implement the methods* `relu_forward` *and* `relu_backward`.

With the function ReLU: $y_i = \max(0, x_i)$, the derivative of y_i w.r.t. x_j simply becomes:

$$\frac{\partial y_i}{\partial x_j} = \begin{cases} 1 & \text{if } x_i > 0 \text{ and } i = j \\ 0 & \text{if } x_i \leq 0 \text{ or } i \neq j. \end{cases} \tag{14}$$

Inserting this into $\frac{\partial L}{\partial x_i} = \sum_{l=1}^{n} \frac{\partial L}{\partial y_l} \frac{\partial y_l}{\partial x_1}$ we arrive at the final equation for the sought derivative

$$\frac{\partial L}{\partial x_i} = \begin{cases} \frac{\partial L}{\partial y_i} & \text{if } x_i > 0 \\ 0 & \text{else} \end{cases} \tag{15}$$

The code needed to implement this is quite simple. For the backward propagation we need a convenient and efficient way of implementing the two cases $x_i > 0$ and $x_i < 0$, preferably in a vectorised manner. I choose a simple boolean vector, and use an element-wise product between it and X:

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

A forward pass just means that we assign $\max(0, x_i) \leftarrow y_i$ which gives us the function

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

Exercise 4: *Derive an expression for* $\frac{\partial L}{\partial x_i}$ *in terms of* y_i*. Implement and test the functions* `softmaxloss_forward` *and* `softmaxloss_backward`.

With $y_i = e^{x_i} / \sum_{j=1}^{m} e^{x_j}$ and $L(\mathbf{x}, c) = -x_c + \log\left(\sum_{j=1}^{m} e^{x_j}\right)$ the derivation of $\frac{\partial L}{\partial x_i}$ is straight forward:

$$\frac{\partial L}{\partial x_i} = \frac{\partial}{\partial x_i}\left(-x_c + \log\left(\sum_{j=1}^{m} e^{x_j}\right)\right) = -\frac{\partial x_c}{\partial x_i} + \left(\frac{\partial}{\partial x_i}\sum_{j=i}^{m} e^{x_j}\right)\frac{1}{\sum_{j=i}^{m} e^{x_j}} \quad (16)$$

If we tidy up this expression (using $\frac{\partial}{\partial x_i}\sum_{j=i}^{m} e^{x_j} = e^{x_i}$) and insert the definition of y_i we get:

$$\frac{\partial L}{\partial x_i} = \frac{e^x i}{\sum_{j=i}^{m} e^{x_j}} - \frac{\partial x_c}{\partial x_i} = y_i - \begin{cases} 1 & \text{if } i = c \\ 0 & \text{if } i \neq c \end{cases} \quad (17)$$

The code needed to implement this is as follows.

`softmaxloss_backward.m:`

```
    idx = sub2ind(sz,labels',1:batch);
    y = exp(x)./(sum(exp(x)));

    dldx = y;
    dldx(idx) = dldx(idx) - 1;
    dldx = dldx/batch;
```

`softmaxloss_forward.m:`

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

2 Training a neural network

Exercise 5: *Implement gradient descent with momentum. Remember to include weight decay.*

Using the update schema as defined in equation (17) and (18) in the assignment description, we update the function `training.m` as follows:

```
...
    my = opts.momentum;
    momentum{i}.(s) = my*momentum{i}.(s) + ...
                    (1-my)*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) ...
        );
...
```

The `object.{i}` indicates that the object is a Matlab-`Cell Array`, which are data structures for holding different kinds of objects. In this case, every entry in the `momentum`-cell array is an array representing a layer. Each position in that array holds the value of the momentum of the gradient of that parameter, for example all the weights for all the nodes in that layer. `momentum{i}.(s)` therefore accesses the momentum of the parameter at position `s` in layer `i`.

3 Classifying Handwritten Digits

Exercise 6: *Plot the kernels of the first convolutional layer. Plot a few images that are missclassified. Plot the confusion matrix. Compute precision and recall for each digit. Write down the number of parameters for all layers in the network. Write comments about all plots and figures.*

We run an altered version of `mnist_starter`, namely `mnist_with_assignment`, that also outputs the predictions it makes. From the saved model we extract the 16 kernels of the first convolutional layer. These are illustrated as gray-scale images in figure 1. The kernels show how the convolution is made. Brighter means that in the weight of that pixel is larger when all pixels are convoluted to the next layer. This means that areas in the kernel that are bright have higher impact in the next layer. The diagonal lines mean that the first layer is focused on detecting these in the image. We then show a few of the images that were missclassified. They are found in figure 2. Some of the images are easy for the human eye to classify and therefore illustrate drawbacks of our model, while some are difficult even for us.

The confusion matrix is an illustration of what classes are mixed up together by the model. At position (i, j) in the matrix we find the number of samples

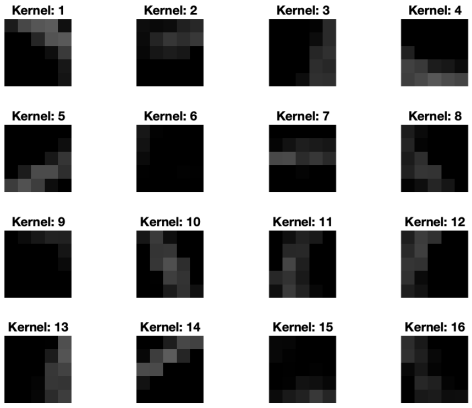


Figure 1 An illustration of the 16 kernels in the first convolutional layer. There are many kernels detecting diagonal lines, which seems natural since hand written digits and letters are often slightly tilted.

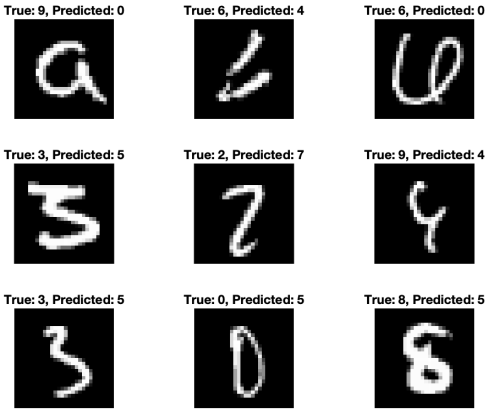


Figure 2 The images that our model has missclassified. Some images are easy to understand how the model has messed up, while others would be difficult even for the human eye. Personally I would also guess that the ”9” at row 2, column 3 is a ”4” and the bird poop at row 1, column 2 is impossible to even see that it is a number.

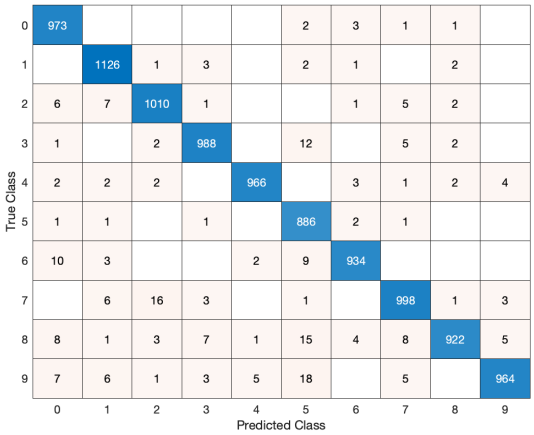


Figure 3 Caption

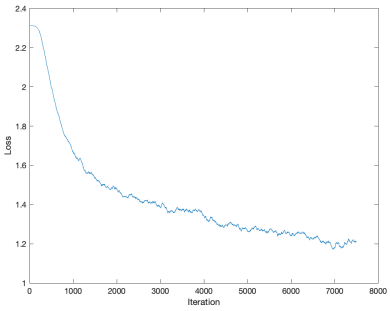
that has ground truth i but is classified as j. Ideally there are only numbers on the diagonal, which of course almost never is the case. The confusion matrix of our model is found in figure 3. In the matrix we can for example see that a lot of digits are being classified as "5", when they acutally are something else, for example "9" or "8". This is found if we examine the column corresponding to a predicted "5". If we instead inspect the row corresponding to a true "5", we see that not many of them are mistaken for another class. That is, almost all the samples end up in the diagonal. Both of the aforementioned observations about the digit "5" are confirmed if we study the precision and recall in table 1. We see that "5" has a precision of 0.9376 and a recall of 0.9933. This means that out of all predicted "5", 93% were correct, while out of all available "5", we managed to retrieve 99% of them. There is often a precision/recall-trade off: we can tune the model to retrieve more of a certain class, with the cost of also falsely classifying more samples as that class. We can also demand that the model is more certain of a certain classification, leading to retrieving less of that class but also misclassifying less of other classes as this class. Precision and recall are important in different use cases. We iterate trough the neural net to count all the parameters that we have to train. The count of these are found in table 2. There are quite a few, 14682 to be exact!

Label	Precision	Recall
1	0.9774	0.9921
2	0.9758	0.9787
3	0.9821	0.9782
4	0.9918	0.9837
5	0.9376	0.9933
6	0.9852	0.9749
7	0.9746	0.9708
8	0.9893	0.9466
9	0.9877	0.9554
0	0.9653	0.9929

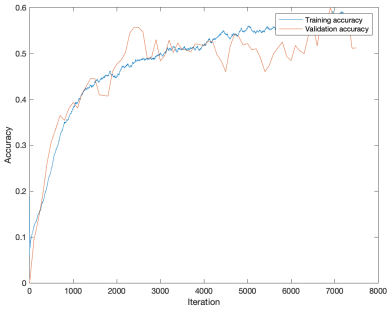
Table 1 We see for example that many digits are being mistaken for the digit "5", since the class has quite low precision when compared to the other classes, but high recall. We therefore retrieve many of the true "5":s but also falsely believe many other digits are "5":s.

Layer	Layer type	Weights	Biases	params
1/9	input	-	-	0
2/9	convolution	5 x 5 x 1 x 16	16	416
3/9	relu	-	-	0
4/9	maxpooling	-	-	0
5/9	convolution	5 x 5 x 16 x 16	16	6416
6/9	relu	-	-	0
7/9	maxpooling	-	-	0
8/9	fully_connected	10 x 784	10	7850
9/9	softmaxloss	-	-	0
Total number of parameters				14682

Table 2 The number of trained parameters in the net. It is understandable why the optimisation task can be difficult, since we need to find a minima in this 14682-dimensional parameter space.



(a) The training loss over the 7500 iterations that we trained the final model. We are approaching convergence but not quite there yet.



(b) The training and validation accuracy of the final model over the 7500 iterations. Just as in 4a, we are approaching convergence but could train for a couple of 1000 more iterations.

Figure 4 Loss and accuracy for the final model plotted against the 7500 iterations we trained it for.

4 Classifying tiny images

Exercise 7: *Improve the baseline model* `cifar10_starter.m`. *Write what experiments you have done and include the same figures and tables as you did in the previous exercise.*

The task is to improve the baseline, which for me received 49.57% accuracy on test set. I begin with removing the last two layers (the fully connected one and the softmax classifier) and instead add one 16-channel 5x5 convolutional layer, followed by ReLU activation and max-pooling for size reduction. I then add another 16-channel 5x5 convolutional layer, followed by ReLU activation and then a fully connected layer and the softmax classification. Since the weights of the first two convolutional layers were already trained, I figured the model would have a *warm start* and therefore not need as many iterations. I therefore reduced the number of iterations to 3500 and trained the model. My last assumption proved to be faulty, and it probably has to do with the fact that the last new layers, reshuffle the optimisation landscape. The model scored 51.04% on test data. I also realised, after doing the training, that I had actually made the model smaller despite trying to make it larger. This comes from the fact that the layer with the largest amount of parameters is the fully connected one, and my additional max pooling reduced both dimensions by half, making the following layers a quarter of the size. I also notice, when studying the accuracy-iteration-plot, that neither my first model nor the baseline seemed to have converged. My conclusions from this was that I could add more convolutional layers and should also increase the number of iterations. I therefore swap the last two (fully connected and softmax) layers for another convolutional, then add ReLU

activation and yet another convolutional layer. The net finishes with a fully connected and a softmax layer. I train this network for an increased number of 7500 iterations. This resulted in my final model and it scored 55.84% on test data. An overview of the layers and their numbers of parameters is found in table 3. The training loss and accuracy on train and validation data for the model is found in figure 4.

Layer	**Layer type**	**Weights**	**Biases**	**params**
1/16	input	-	-	0
2/16	convolution	5 x 5 x 3 x 16	16	1216
3/16	relu	-	-	0
4/16	maxpooling	-	-	0
5/16	convolution	5 x 5 x 16 x 16	16	6416
6/16	relu	-	-	0
7/16	convolution	5 x 5 x 16 x 16	16	6416
8/16	relu	-	-	0
9/16	maxpooling	-	-	0
10/16	convolution	5 x 5 x 16 x 16	16	6416
11/16	relu	-	-	0
12/16	convolution	5 x 5 x 16 x 16	16	6416
13/16	relu	-	-	0
14/16	convolution	5 x 5 x 16 x 16	16	6416
15/16	fully_connected	10 x 1024	10	10250
16/16	softmaxloss	-	-	0
			Total number of parameters	43546

Table 3 The number of trained parameters in the convolutional neural network trained on Cifar10 images. There are quite a few and therefore took a while to train, but there is also a lot of information to digest from colour-images. Unfortunately, the training time is too long to do much more experimenting, but I would like to try to increase the number of channels in some of the convolutional layers.

The other requested plots and tables are found below. Since the kernels also take negative values, the illustration of them in terms of colour becomes slightly odd. I have chosen to re-scale each kernel between 1 and 0 before showing it. However, it is still hard to draw any conclusions from the colours.

The confusion matrix, in figure 6, show that there are quite a few mix-ups, but most of them are between classes that are understandable to mix up. As we see in both the confusion matrix and in table 4, the label "dog" has terrible recall. It is mainly mixed up with cat and horse, which is not very surprising since all three are four-legged mammals covered in fur. Other common mix-ups, such as truck and automobile, are also forgivable.

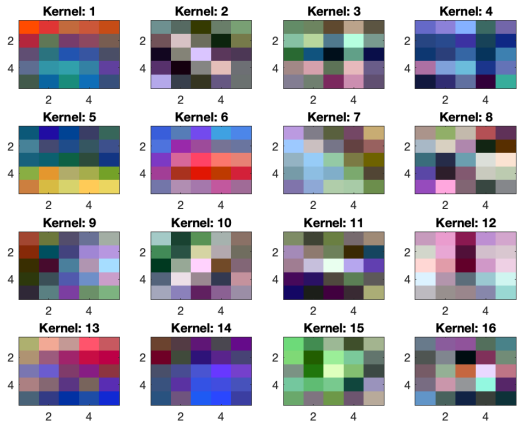


Figure 5 The kernels of the first convolutional layer for the Cifar10-model. It is quite difficult to draw any conclusions from it.

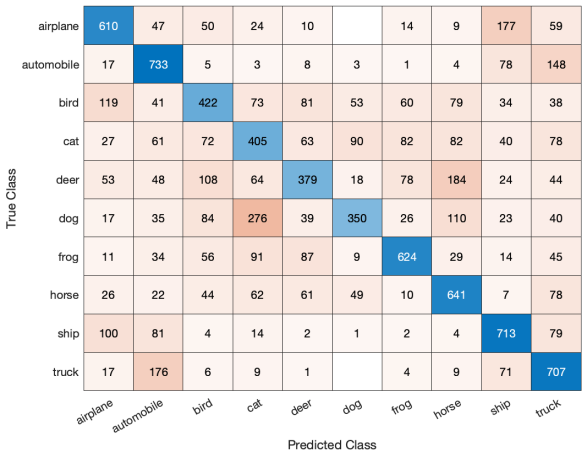


Figure 6 Confusion matrix of the Cifar10 model. It seems to mainly have trouble between animals such as deer-horse and cat-dog, which is quite understandable.

Label	Precision	Recall
airplane	0.6118	0.6100
automobile	0.5736	0.7330
bird	0.4959	0.4220
cat	0.3967	0.4050
deer	0.5185	0.3790
dog	0.6108	0.3500
frog	0.6926	0.6240
horse	0.5569	0.6410
ship	0.6037	0.7130
truck	0.5372	0.7070

Table 4 Just as in the confusion matrix, the fur covered animals seem hardest to classify, probably due to their similar shapes and colors.

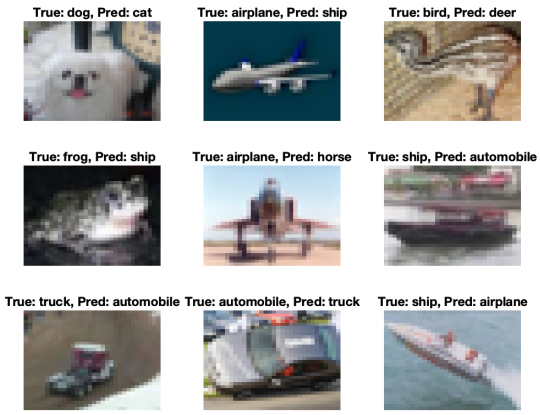


Figure 7 The missclassified images from Cifar10. Here we see two examples of the common mix-ups cat/dog and truck/automobile. To be fair to the model: the boat in the lower right corner is actually airborne :).