FMAN45 Machine Learning, spring 2020
Assignment 3

Salma Kazemi Rashed

May 2020

1 Common Layers and backpropagation.

Exercise 1.1: Derive expressions for $\partial L/\partial x$, $\partial L/\partial \boldsymbol{W}$, and $\partial L/\partial \boldsymbol{b}$ in terms of $\partial L/\partial x'$,$\boldsymbol{W}$ and x. Include a full derivation of your results. The answers should all be given as matrix expressions without any explicit sums.
By using the following equation:

$$\overline{x}'_i = \sum_{j=1}^{m} W_{ij}x_j + b_i \tag{1}$$

For $\frac{\partial L}{\partial x}$, We will have :

$$\frac{\partial L}{\partial x} = \begin{pmatrix} \frac{\partial L}{\partial x_1} \\ \frac{\partial L}{\partial x_2} \\ \vdots \\ \frac{\partial L}{\partial x_m} \end{pmatrix} = \begin{pmatrix} \sum_{l=1}^{n} \frac{\partial L}{\partial \overline{x}'_l}\frac{\partial \overline{x}'_l}{\partial x_1} \\ \sum_{l=1}^{n} \frac{\partial L}{\partial \overline{x}'_l}\frac{\partial \overline{x}'_l}{\partial x_2} \\ \vdots \\ \sum_{l=1}^{n} \frac{\partial L}{\partial \overline{x}'_l}\frac{\partial \overline{x}'_l}{\partial x_m} \end{pmatrix} = \begin{pmatrix} \frac{\partial L}{\partial \overline{x}'_1}W_{11} + \frac{\partial L}{\partial \overline{x}'_2}W_{21} + ... + \frac{\partial L}{\partial \overline{x}'_n}W_{n1} \\ \frac{\partial L}{\partial \overline{x}'_1}W_{12} + \frac{\partial L}{\partial \overline{x}'_2}W_{22} + ... + \frac{\partial L}{\partial \overline{x}'_1}W_{n2} \\ \vdots \\ \frac{\partial L}{\partial \overline{x}'_1}W_{1m} + \frac{\partial L}{\partial \overline{x}'_1}W_{2m} + ... + \frac{\partial L}{\partial \overline{x}'_1}W_{nm} \end{pmatrix} = \boldsymbol{W}^T \frac{\partial L}{\partial \overline{\boldsymbol{x}}'} \tag{2}$$

For $\frac{\partial L}{\partial \boldsymbol{W}}$, We will have :

$$\frac{\partial L}{\partial \boldsymbol{W}} = \begin{pmatrix} \frac{\partial L}{\partial W_{11}} & \frac{\partial L}{\partial W_{12}} & \cdots & \frac{\partial L}{\partial W_{1m}} \\ \frac{\partial L}{\partial W_{21}} & \frac{\partial L}{\partial W_{22}} & \cdots & \frac{\partial L}{\partial W_{2m}} \\ \vdots & & & \\ \frac{\partial L}{\partial W_{n1}} & \frac{\partial L}{\partial W_{n2}} & \cdots & \frac{\partial L}{\partial W_{nm}} \end{pmatrix} = \begin{pmatrix} \frac{\partial L}{\partial \overline{x}'_1}x_1 & \frac{\partial L}{\partial \overline{x}'_1}x_2 & \cdots & \frac{\partial L}{\partial \overline{x}'_1}x_m \\ \frac{\partial L}{\partial \overline{x}'_2}x_1 & \frac{\partial L}{\partial \overline{x}'_2}x_2 & \cdots & \frac{\partial L}{\partial \overline{x}'_2}x_m \\ \vdots & & & \\ \frac{\partial L}{\partial \overline{x}'_n}x_1 & \frac{\partial L}{\partial \overline{x}'_n}x_2 & \cdots & \frac{\partial L}{\partial \overline{x}'_n}x_m \end{pmatrix} = \frac{\partial L}{\partial \overline{\boldsymbol{x}}'}\boldsymbol{x}^T \tag{3}$$

And finally, since $\frac{\partial \overline{x}'_i}{\partial b_i} = 1$ for $\frac{\partial L}{\partial \boldsymbol{b}}$ we will have:

$$\frac{\partial L}{\partial \boldsymbol{b}} = \begin{pmatrix} \frac{\partial L}{\partial \overline{x}'_1} \\ \frac{\partial L}{\partial \overline{x}'_2} \\ \vdots \\ \frac{\partial L}{\partial \overline{x}'_n} \end{pmatrix} = \frac{\partial L}{\partial \overline{\boldsymbol{x}}'} \tag{4}$$

Exercise 1.2: Derive expressions for $\overline{\boldsymbol{X}'}$, $\frac{\partial L}{\partial \boldsymbol{X}}$, $\frac{\partial L}{\partial \boldsymbol{W}}$, and $\frac{\partial L}{\partial \boldsymbol{b}}$, in terms of $\frac{\partial L}{\partial \overline{\boldsymbol{X}'}}$, $\boldsymbol{W}$, and $\boldsymbol{X}$. Include a full derivation of your results. Also add Matlab code that implements

these vectorised expressions that you have just derived. In the code there are two files layers/fully_connected_forward.m and layers/fully_connected_backward.m. Implement these functions and check that your implementation is correct by running tests/test_fully_connected.m. For full credit your code should be vectorized over the batch. Include the relevant code in the report.

By using the results of previous exercises, we extend the results to N elements in a batch. Therefore, we will have:

$$\overline{\boldsymbol{X}'} = \begin{pmatrix} W_{11} & W_{12} & \dots & W_{1m} \\ W_{21} & W_{22} & \dots & W_{2m} \\ \vdots & & & \\ W_{n1} & W_{n2} & \dots & W_{nm} \end{pmatrix} \begin{pmatrix} x_1^{(1)} & x_1^{(2)} & \dots & x_1^{(N)} \\ x_2^{(1)} & x_2^{(2)} & \dots & x_2^{(N)} \\ \vdots & & & \\ x_m^{(1)} & x_m^{(2)} & \dots & x_m^{(N)} \end{pmatrix} + \boldsymbol{b}\mathbf{1}_{1\times N} = \boldsymbol{W}\boldsymbol{X} + \boldsymbol{b}\mathbf{1}_{1\times N} \quad (5)$$

$$\frac{\partial L}{\partial \boldsymbol{X}} = \begin{pmatrix} \frac{\partial L}{\partial \boldsymbol{x}^{(1)}} & \frac{\partial L}{\partial \boldsymbol{x}^{(2)}} & \dots & \frac{\partial L}{\partial \boldsymbol{x}^{(N)}} \end{pmatrix} = \begin{pmatrix} \boldsymbol{W}^T \frac{\partial L}{\partial \overline{\boldsymbol{x}}'^{(1)}} & \boldsymbol{W}^T \frac{\partial L}{\partial \overline{\boldsymbol{x}}'^{(2)}} & \dots & \boldsymbol{W}^T \frac{\partial L}{\partial \overline{\boldsymbol{x}}'^{(N)}} \end{pmatrix} = \boldsymbol{W}^T \frac{\partial L}{\partial \overline{\boldsymbol{X}'}} \quad (6)$$

$$\frac{\partial L}{\partial \boldsymbol{W}} = \sum_{l=1}^{N} \frac{\partial L}{\partial \overline{\boldsymbol{x}}'^{l}} \boldsymbol{x}^{(l)T} = \begin{pmatrix} \frac{\partial L}{\partial \overline{\boldsymbol{x}}'^{(1)}} & \frac{\partial L}{\partial \overline{\boldsymbol{x}}'^{(2)}} & \dots & \frac{\partial L}{\partial \overline{\boldsymbol{x}}'^{(N)}} \end{pmatrix} \begin{pmatrix} \boldsymbol{x}^{(1)T} \\ \boldsymbol{x}^{(2)T} \\ \vdots \\ \boldsymbol{x}^{(N)T} \end{pmatrix} = \frac{\partial L}{\partial \overline{\boldsymbol{X}'}} \boldsymbol{X}^T \quad (7)$$

$$\frac{\partial L}{\partial \boldsymbol{b}} = \sum_{l=1}^{N} \frac{\partial L}{\partial \overline{\boldsymbol{x}}'^{(l)}} = \frac{\partial L}{\partial \overline{\boldsymbol{X}'}} (\mathbf{1}_{1\times N})^T \quad (8)$$

```
% Suitable for matrix multiplication
X = reshape(X, [features, batch]);
Y = A*X+repmat(b,1,batch);

assert(size(A, 2) == features, ...
    sprintf('Expected %d columns in the weights matrix, got %d', features, size(A,2)));
assert(size(A, 1) == numel(b), 'Expected as many rows in A as elements in b');

%error('Implement this!');

end
```

Figure 1: Forward implementation of $\overline{\boldsymbol{X}}'$.

```
% Implement it here.
% note that dldX should have the same size as X, so use reshape
% as suggested.
dldX=A' * dldY;
dldA=dldY*X';
dldb=dldY*ones(batch, 1);
%error('Implement this!');
```

Figure 2: Implementation of backward function and the $\frac{\partial L}{\partial \boldsymbol{X}}$, $\frac{\partial L}{\partial \boldsymbol{W}}$, and $\frac{\partial L}{\partial \boldsymbol{b}}$ expressions.

```
It  5   3:  -4.35640e+00  -4.35640e+00   4.2186e-12
Everything passed! Your implementation seems correct.
```

Figure 3: The message of fully connected test function.

Exercise 1.3: Derive the backpropagation expression for $\frac{\partial L}{\partial x_i}$. Give a full solution. Implement the layer in layers/relu_forward.m and layers/relu_backward.m. Test it with tests/test_relu.m. For full credit you must use built in indexing and not use any for-loops. Include the relevant code in the report.
by using $y_i = \max(x_i, 0)$ we will have:

$$\frac{\partial L}{\partial x_i} = \frac{\partial L}{\partial y_i} \text{step}(x_i) \tag{9}$$

where step function is defined as follows:

$$\text{step}(x) = \begin{cases} 1, & x > 0 \\ 0, & x < 0 \end{cases} \tag{10}$$

```
function y = relu_forward(x)
    y = max(x,0);
end
```

Figure 4: Relu forward function.

```
function dldx = relu_backward(x, dldy)
    dldx = dldy.* (x>0)
end
```

Figure 5: Relu backward function.

```
dldx =

    1


dldx =

    0

Everything passed! Your implementation seems correct.
```

Figure 6: Relu test function result.

Exercise 1.4: Compute the expression for $\frac{\partial L}{\partial x_i}$. Write a full solution. Note that this is the final layer, so there are no gradients to backpropagate. Implement the layer in layers/ softmaxloss_forward.m and layers/softmaxloss_backward.m. Test it with tests/test_softmaxloss.m. Make sure that tests/ test_gradient_whole_net.m runs correctly when you have implemented all layers. For full credit you should use built in functions for summation and indexing and not use any for-loops. Include the relevant code in the report. .
we use $\frac{d(\log(x))}{dx} = \frac{1}{x}$ and $L(x, c) = -x_c + \log(\sum_{j=1}^{m} e^{x_j})$. We know that for all non-correct $(j \neq c)$ elements the derivative with respect to x_i would be:

$$\frac{\partial L}{\partial x_i} = \frac{e^{x_i}}{\sum_{j=1}^{m} e^{x_j}} \tag{11}$$

And for all correct points the derivative would be:

$$\frac{\partial L}{\partial x_c} = \frac{e^{x_c}}{\sum_{j=1}^{m} e^{x_j}} - 1 \tag{12}$$

For this part, the forward softmax and backward softmax functions and the correctly passed test messages for softmax layer and the whole net are shown in Figs. (7–10)

```
xc = x(labels' + (0:batch-1)*features);  % this line only gives correct x's calculated from labels
L =sum(log(sum(exp(x)))-xc)/batch; % average over batches
```

Figure 7: Implementation of forward softmax loss.

```
%% This is for minus xc for correct points and zero for the rest of points
one_on_xc = zeros(sz);
one_on_xc(labels' + (0:batch-1)*features)=1; % elements are one only on correct x's (xc)
dldx = (exp(x)./(repmat(sum(exp(x)),features,1))-one_on_xc)/batch; %averaging over batches
```

Figure 8: Backward implementation of softmax loss derivatives with respect to inputs.

```
It  7  1:  2.14790e-02  2.14787e-02   6.3084e-06
It  8 52:  4.71887e-03  4.71880e-03   7.8674e-06
It  9 35:  6.83425e-03  6.83415e-03   7.6623e-06
It 10 21:  6.07654e-03  6.07644e-03   7.7355e-06
It 11 55:  8.05760e-03  8.05748e-03   7.5449e-06
It 12 27:  2.86907e-02  2.86903e-02   5.6838e-06
It 13  9: -2.43278e-01 -2.43278e-01   2.1202e-07
It 14 45:  5.30491e-03  5.30483e-03   7.8103e-06
It 15 14:  4.75862e-03  4.75855e-03   7.8635e-06
It 16 75:  1.62468e-02  1.62466e-02   6.7790e-06
It 17 56:  6.18862e-03  6.18853e-03   7.7247e-06
It 18 72: -2.42758e-01 -2.42758e-01   2.2742e-07
It 19 65:  1.84667e-03  1.84664e-03   8.1495e-06
It 20 68:  2.22173e-02  2.22170e-02   6.2432e-06
It 21 54:  2.98913e-02  2.98909e-02   5.5825e-06
It 22 60:  3.00516e-03  3.00512e-03   8.0352e-06
It 23 58:  5.39085e-03  5.39077e-03   7.8020e-06
It 24 27:  2.86907e-02  2.86903e-02   5.6838e-06
It 25 45:  5.30491e-03  5.30483e-03   7.8103e-06
It 26 74:  4.35747e-03  4.35740e-03   7.9026e-06
It 27  5:  8.51566e-03  8.51553e-03   7.5011e-06
It 28 43:  1.35656e-02  1.35654e-02   7.0258e-06
It 29 37:  5.05949e-03  5.05941e-03   7.8342e-06
It 30 76:  1.27818e-02  1.27816e-02   7.0987e-06
Everything passed! Your implementation seems correct.
>>
```

Figure 9: Softmax loss layer test result.

```
It 30  12:  -6.47838e-01  -6.47838e-01    1.2748e-11
Everything passed!
>>
```

Figure 10: he whole network test function result.

2 Training a Neural Network

Exercise 5: Implement gradient descent with momentum in training.m. Remember to include weight decay. Include the relevant code in the report.

The implemented code is shown in Fig. (11).

```
%% Gradient descent with momentum
momentum{i}.(s) =  opts.moving_average * momentum{i}.(s) + (1-opts.moving_average)*(grads{i}.(s) + ...
                                                        opts.weight_decay * net.layers{i}.params.(s));
net.layers{i}.params.(s) = net.layers{i}.params.(s) - opts.learning_rate * (momentum{i}.(s) )
%error('Implement this!');
```

Figure 11: Gradient descent with momentum.

3 Classifying Handwritten Digits

Exercise 6: Plot the filters the first convolutional layer learns. Plot a few images that are misclassified. Plot the confusion matrix for the predictions on the test set and compute the precision and the recall for all digits. Write down the number of parameters for all layers in the network. Write comments about all plots and figures.
There are 16 channels of filters in the first convolutional layer that is plotted in Figs. (12–15). The higher values of the filter bold the pixels of those areas more.

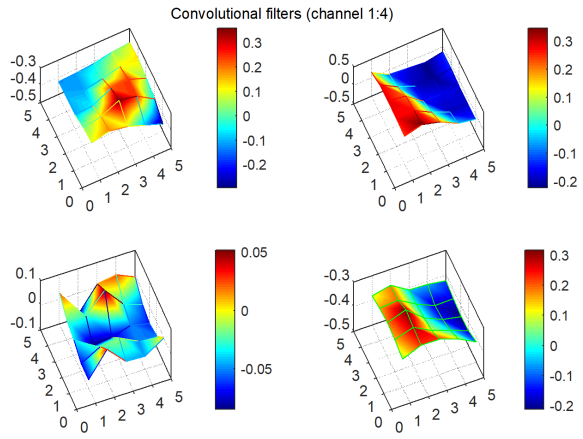


Figure 12: convolutional filters channel 1:4.

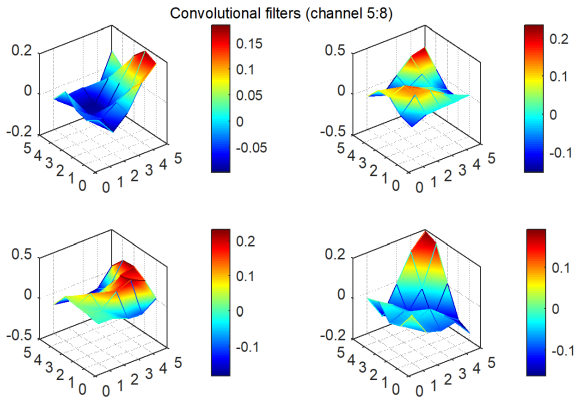


Figure 13: convolutional filters channel 5:8.

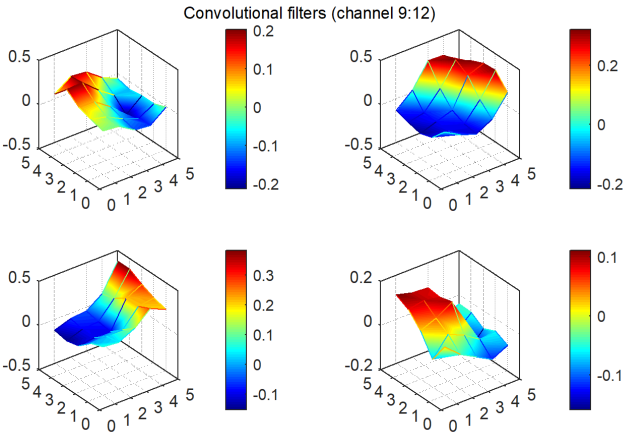


Figure 14: convolutional filters channel 9:12.

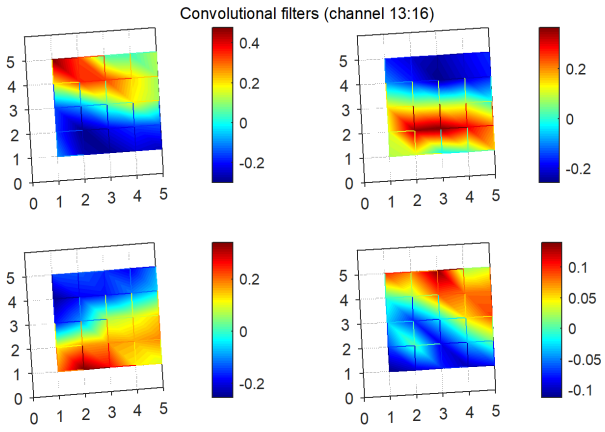


Figure 15: convolutional filters channel 13:16.

Some of the misclassified images ar depicted in Fig. (16).

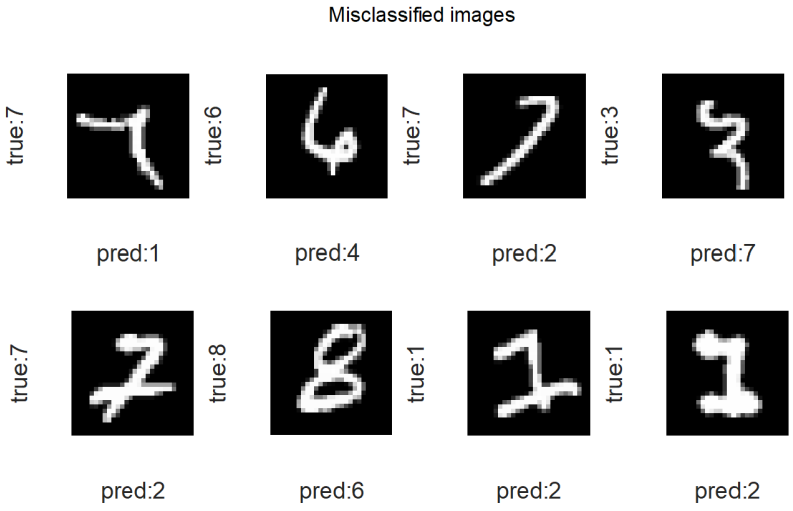
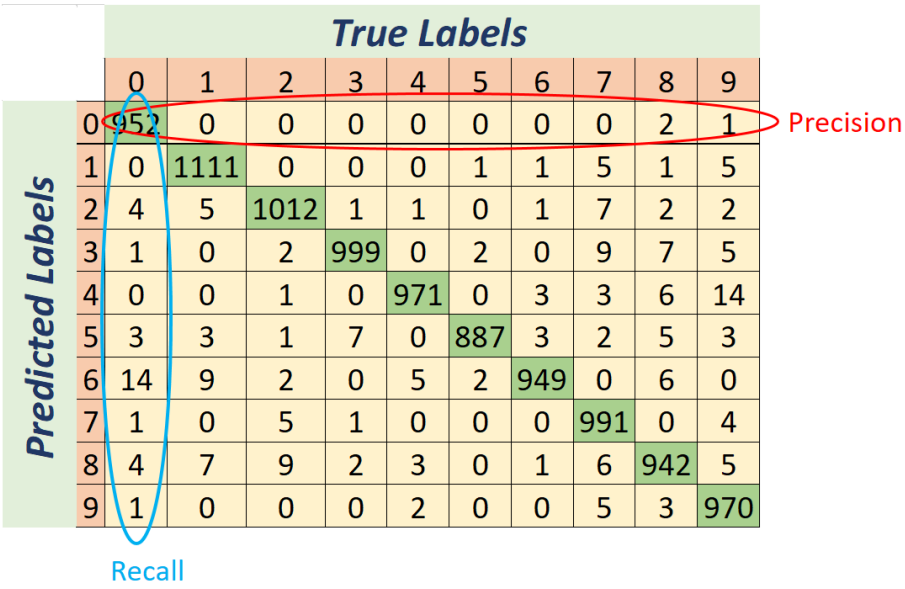


Figure 16: Some of the misclassified images.

Confusion matrix is calculated for all classes and depicted in Fig. (17). According to confusion matrix, precision and recall is calculated for all digits and depicted in Fig. (18). Number 0 has the highest precision and number 6 the lowest. Number 5 has the highest recall.



True Labels										
Predicted Labels	0	1	2	3	4	5	6	7	8	9
0	952	0	0	0	0	0	0	0	2	1
1	0	1111	0	0	0	1	1	5	1	5
2	4	5	1012	1	1	0	1	7	2	2
3	1	0	2	999	0	2	0	9	7	5
4	0	0	1	0	971	0	3	3	6	14
5	3	3	1	7	0	887	3	2	5	3
6	14	9	2	0	5	2	949	0	6	0
7	1	0	5	1	0	0	0	991	0	4
8	4	7	9	2	3	0	1	6	942	5
9	1	0	0	0	2	0	0	5	3	970

Figure 17: Confusion matrix.

Precision = TP/(TP+FP)									
0	1	2	3	4	5	6	7	8	9
0.996859	0.988434	0.977778	0.974634	0.972946	0.97046	0.961499	0.989022	0.962206	0.988787
Recall=TP/(TP+FN)									
0	1	2	3	4	5	6	7	8	9
0.971429	0.978855	0.98062	0.989109	0.988798	0.994395	0.990605	0.964008	0.967146	0.961348

Figure 18: Precision and recall for all digits.

All the parameters in each layer is calculated and depicted in Table (1).

Layer	Type	size	#params
1	input	(28 28 1 8)	0
2	convolution	(28 28 16 8)	400 + 16
3	relu	(28 28 16 8)	0
4	maxpooling	(14 14 16 8)	0
5	convolution	(14 14 16 8)	400 +16
6	relu	(14 14 16 8)	0
7	maxpooling	(7 7 16 8)	0
8	fully-connected	(10 8)	7840 + 10
9	softmaxloss	(1 1)	0

Table 1: Type, size and the number of parameters in each layer. Parameters contain weights and bias values that the network should learn.

4 Classifying Tiny Images

Exercise 7: Do whatever you want to improve the accuracy of the baseline network. Write what you have tried in the report, even experiments that did not work out.
First, I checked the accuracy of the current network on CIFAR-10 dataset which was around 46 %. For improving the performance of the network, I googled a little to find out what people have done for getting a better results on CIFAR-10 dataset. According to this link, I tried a deeper fully CNN model where all the layers are Convolutional except the last layer which is a fully connected layer. I used the convolutional layers of [3,20,40,80] followed by a fully connected layer. The implemented model is shown in Table (2).

Layer	Type	size	#params
Layer 1	(input)	(32 32 3 8)	0
Layer 2	(convolution)	(32 32 20 8)	1500 + 20
Layer 3	(relu)	(32 32 20 8)	0
Layer 4	(maxpooling)	(16 16 20 8)	0
Layer 5	(convolution)	(16 16 40 8)	20000 +40
Layer 6	(relu)	(16 16 40 8)	0
Layer 7	(maxpooling)	(8 8 40 8)	0
Layer 8	(convolution)	(8 8 80 8)	80000+8
Layer 9	(relu)	(8 8 80 8)	0
Layer 10	(maxpooling)	(4 4 80 8)	0
Layer 11	(convolution)	(4 4 16 8)	32000+16
Layer 12	(relu)	(4 4 16 8)	0
Layer 13	(fully_connected)	(10 8)	2560+10
Layer 14	(softmaxloss)	(1 1)	0

Table 2: Type and size of new network.

The achieved accuracy was as following:
Iteration 7500:
Classification loss: 1.078770
Weight decay loss: 0.013471
Total loss: 1.092241
Training accuracy: 0.624107
Validation accuracy: 0.584034
Accuracy on the test set: 0.565000
The validation and training accuracy is also plotted in Fig. (19)

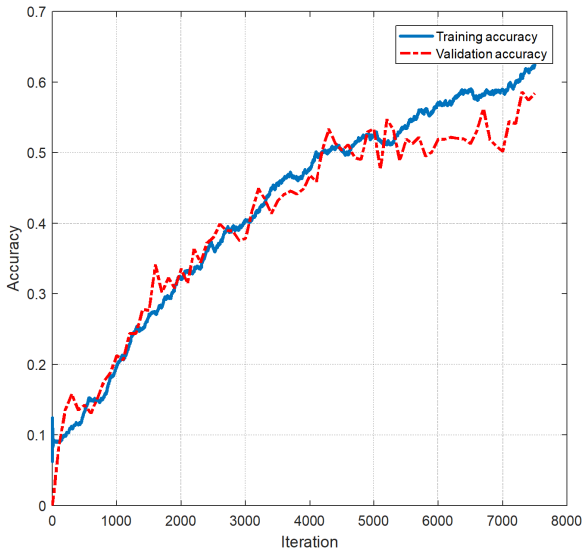


Figure 19: Achieved accuracy for the network shown in Table (2).

Then I tried other techniques to improve the performance of the network even more. I tried to implement batch-normalization tecnique which is "*a recent idea introduced by Ioffe et al, 2015 to ease the training of large neural networks. The idea behind it is that neural networks tend to learn better when their input features are uncorrelated with zero mean and unit variance. As each layer within a neural network see the activations of the previous layer as inputs, the same idea could be apply to each layer. Batch normalization does exactly that by normalizing the activations over the current batch in each hidden layer, generally right before the non-linearity.*" taken from this link. The implemented batchnorm-forward and batchnorm-backward functions are shown in Figs. (20, 21) according to the derivaties taken from this link.
In batch normalization, the input points in each batch first normalized to their mean and variance plus an epsilon and the output would be an scaled and shifted form of the normalized input as following:

$$y = \gamma \hat{x} + \beta \tag{13}$$

where γ and β are the parameters of the layer that could be learned where their size is as large as the number of batches. I added this layer after convolutional layers before Relu. However, I could not get a better result compared to the deeper CNN I tried first. I am still working on batch normalization technique testing it on different models. The other reason for using batch normalization is for increasing the learning rate in deeper networks in order to achieve a faster convergence without the concern of divergence in high learning rates.
I used the following derivatives in batchnorm backward implementation taken from this link.

$$\frac{d\mathcal{L}}{d\beta_j} = \sum_k \frac{d\mathcal{L}}{dy_{kj}} \tag{14}$$

$$\frac{d\mathcal{L}}{dx_{ij}} = \frac{1}{N}\gamma_j(\sigma_j^2+\epsilon)^{-1/2}\left(N\frac{d\mathcal{L}}{dy_{ij}} - \sum_k \frac{d\mathcal{L}}{dy_{kj}} - (x_{ij}-\mu_j)(\sigma_j^2+\epsilon)^{-1}\sum_k \frac{d\mathcal{L}}{dy_{kj}}(X_{kj}-\mu_j)\right) \tag{15}$$

$$\frac{d\mathcal{L}}{d\gamma_j} = \sum_k \frac{d\mathcal{L}}{dy_{kj}}(x_{kj} - \mu_j)(\sigma_j^2 + \epsilon)^{-1/2} \tag{16}$$

```
function [Y] = batchnorm_forward(X, gamma, beta)


eps = 1e-3;
sz = size(X);
batch = sz(end);
features = prod(sz(1:end-1));

X_reshape = reshape(X,[features, batch]);
mu = 1./features*sum(X_reshape,1);
var = 1./features*sum((X_reshape-mu).^2, 1);



hath = (X_reshape-mu).*(var+eps).^(-1./2);

 out = hath.*gamma+beta ;
Y = reshape(out,[sz(1:end-1),batch]);

end
```

Figure 20: batch normalization forward function.

```
function [dx, dgamma, dbeta] = batchnorm_backward( X,dy, gamma,beta)

eps = 1e-3;
sz = size(X);
batch = sz(end);
features = prod(sz(1:end-1));

X_reshape = reshape(X,[features, batch]);
mu = 1./features*sum(X_reshape,1);
var = 1./features*sum((X_reshape-mu).^2, 1);

dy_reshape =reshape(dy,[features, batch]);
dbeta = sum(dy_reshape,1);
dgamma = sum((X_reshape-mu) .* (var + eps).^(-1. / 2.) .* dy_reshape, 1);
dh = (1/ features) .* gamma .* (var + eps).^(-1./ 2) .* (features * dy_reshape -sum(dy_reshape,1) -...
      (X_reshape - mu) .* (var + eps).^(-1.0) .* sum(dy_reshape .* (X_reshape - mu), 1));

dx = reshape(dy,[sz(1:end-1),batch] );

end
```

Figure 21: batch normalization backward function.

The other results of my final model (Table (2)) are depicted in Figs. (22–26). I have plotted only a few filters of the first convolutional filter shown in Figs.(22, 23).

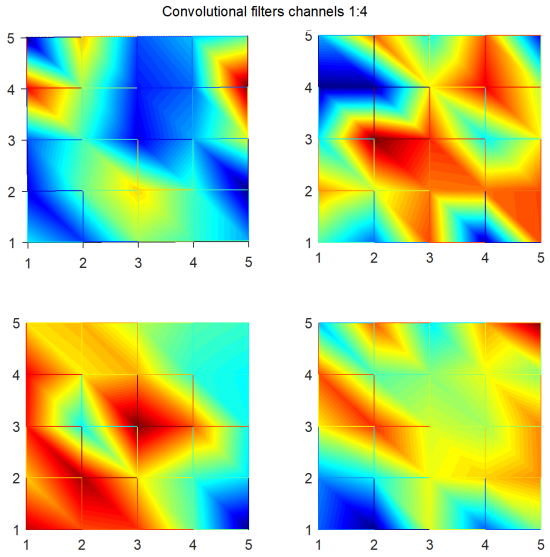


Figure 22: 5×5 filters of the first convolutional layer.

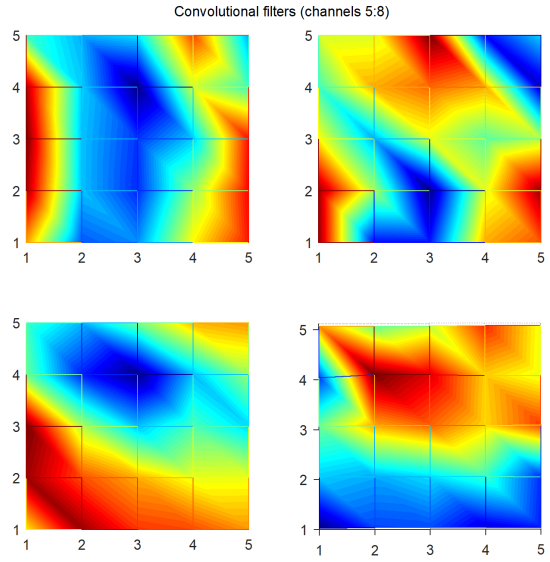


Figure 23: 5×5 filters of the first convolutional layer.

Some of the misclassified images are also shown in Fig. (24).

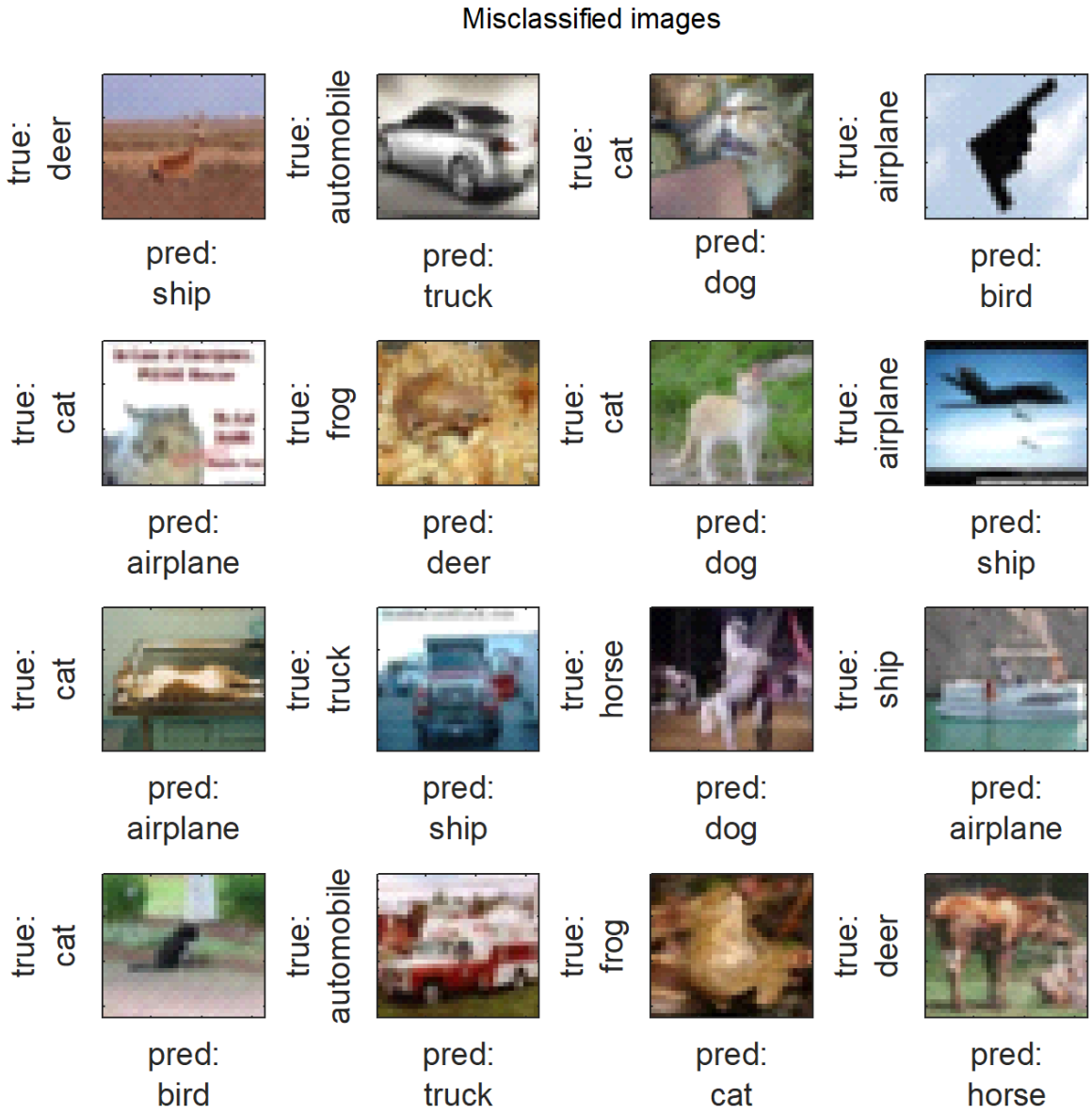


Figure 24: Misclassified images.

Finally, the confusion matrix and the precision and recall are calculated and depicted for each class in Figs. (25, 26). Among objects, 'automobile' has the highest precision while 'ship' has the highest recall.

		True labels									
		airplane	automobile	bird	cat	deer	dog	frog	horse	ship	truck
Predicted Labels	airplane	525	23	76	35	33	18	6	15	54	21
	automobile	19	468	7	6	8	4	3	0	38	56
	bird	46	12	323	69	91	69	65	18	10	9
	cat	12	15	112	378	86	172	131	60	15	25
	deer	30	21	143	75	453	70	130	77	7	14
	dog	11	9	118	192	63	490	28	86	19	11
	frog	9	9	49	36	53	10	519	7	2	15
	horse	25	12	81	97	149	104	34	684	20	46
	ship	235	114	49	30	28	27	16	10	775	100
	truck	88	317	42	82	36	36	68	43	60	703

Precision

Recall

Figure 25: Confusion matrix.

Precision = TP/(TP+FP)									
airplane	automobile	bird	cat	deer	dog	frog	horse	ship	truck
0.651365	0.768473	0.453652	0.375746	0.444118	0.477118	0.732017	0.546326	0.559971	0.47661
Recall=TP/(TP+FN)									
airplane	automobile	bird	cat	deer	dog	frog	horse	ship	truck
0.525	0.468	0.323	0.378	0.453	0.49	0.519	0.684	0.775	0.703

Figure 26: Precision and Recall.