Assignment 3 in Machine Learning FMAN45

Eliot Montesino Petrén, el6183mo-s, 990618-9130, LU Faculty of Engineering F19 , eliot.mp99@gmail.com

I. Exercises

Exercise 1

The task is to derive exact gradient expressions of the loss, including $\frac{\partial L}{\partial \mathbf{x}}, \frac{\partial L}{\partial \mathbf{W}}, \frac{\partial L}{\partial \mathbf{b}}$ only using directly observable parameters $\frac{\partial L}{\partial \mathbf{y}}, \mathbf{W}$ and $\mathbf{x}$. Furthermore, the final expressions will be in matrix form. First I will make the remark that y_l can be rewritten as.

$$y_l = \sum_{j=1}^{m} W_{lj} x_j + b_j \tag{1}$$

Starting with $\frac{\partial L}{\partial \mathbf{x}}$ and utilizing the chain rule gives.

$$\frac{\partial L}{\partial x_i} = \sum_{l=1}^{n} \frac{\partial L}{\partial y_l} \frac{\partial y_l}{\partial x_i} = \sum_{l=1}^{n} \frac{\partial L}{\partial y_l} \frac{\partial}{\partial x_i} \left(\sum_{j=1}^{m} W_{lj} x_j + b_j \right) \tag{2}$$

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

Extrapolate the result in eq 3 to the entire input vector x giving a matrix expression.

$$\frac{\partial L}{\partial \mathbf{x}} = \begin{bmatrix} W_{1,1} & W_{2,1} & \cdots & W_{m,1} \\ W_{1,2} & W_{2,2} & \cdots & W_{m,2} \\ \vdots & \vdots & \ddots & \vdots \\ W_{1,n} & W_{2,n} & \cdots & W_{m,n} \end{bmatrix} \cdot \begin{bmatrix} \frac{\partial L}{\partial y_1} \\ \frac{\partial L}{\partial y_2} \\ \\ \frac{\partial L}{\partial y_m} \end{bmatrix} = \frac{\partial L}{\partial \mathbf{y}} \mathbf{W}^T \tag{4}$$

Moving on to derive $\frac{\partial L}{\partial \mathbf{W}}$ I will repeat the similar steps.

$$\frac{\partial L}{\partial W_{ij}} = \sum_{l=1}^{n} \frac{\partial L}{\partial y_l} \frac{\partial}{\partial W_{ij}} \left(\sum_{k=1}^{m} W_{lk} x_k + b_l \right) \tag{5}$$

$$= \sum_{l=1}^{n} \frac{\partial L}{\partial y_l} (\sum_{k=1}^{m} x_k \frac{\partial W_{lk}}{\partial W_{ij}} + b_l) = \frac{\partial L}{\partial W_{ij}} = \frac{\partial L}{\partial y_i} x_j \tag{6}$$

Where in the last step $\frac{\partial W_{lk}}{\partial W_{ij}} = 1$ as one sets $l = i$, $j = k$ and otherwise 0. Moving on to formulate above in matrix formulation.

$$\frac{\partial L}{\partial \mathbf{W}} = \begin{bmatrix} \frac{\partial L}{\partial y_1} \\ \frac{\partial L}{\partial y_2} \\ \vdots \\ \frac{\partial L}{\partial y_m} \end{bmatrix} \cdot [x_1 \ x_2 \ \cdots \ x_m] = \frac{\partial L}{\partial \mathbf{y}} \mathbf{x}^T \tag{7}$$

Lastly deriving $\frac{\partial L}{\partial \mathbf{b}}$, apply the chain rule and perform similar steps as above.

$$\frac{\partial L}{\partial b_i} = \sum_{l=1}^{n} \frac{\partial L}{\partial y_l} \frac{\partial y_l}{\partial b_i} \tag{8}$$

$$= \sum_{l=1}^{n} \frac{\partial L}{\partial y_l} \frac{\partial}{\partial b_i} \left(\sum_{j=1}^{m} W_{lj} x_j + b_j \right) = \frac{\partial L}{\partial y_i} \tag{9}$$

Above is obtained when $\frac{\partial b_j}{\partial b_i} = 1$ as $i = j$ and otherwise 0. This simply translates in matrix notation to eq 10.

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

Exercise 2

$$\frac{\partial L}{\partial \mathbf{X}} = \left[\mathbf{W}^T \frac{\partial L}{\partial \mathbf{y}^{(1)}} \mathbf{W}^T \frac{\partial L}{\partial \mathbf{y}^{(2)}}, \ldots, \mathbf{W}^T \frac{\partial L}{\partial \mathbf{y}^{(N)}} \right] \tag{11}$$

$$= \mathbf{W}^T \frac{\partial L}{\partial \mathbf{Y}} \tag{12}$$

First, we derive the expression for $\mathbf{Y}$:

$$\mathbf{Y} = \left[\mathbf{W}\mathbf{x}^{(1)} + \mathbf{b}, \mathbf{W}\mathbf{x}^{(2)} + \mathbf{b}, \ldots, \mathbf{W}\mathbf{x}^{(N)} + \mathbf{b} \right] = \mathbf{W}\mathbf{X} + \mathbf{B} \tag{13}$$

where $\mathbf{X} = \left[\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(N)} \right]$ and $\mathbf{B}$ is a matrix with $\mathbf{b}$ repeated N times as columns.

Next, we derive the expression for $\frac{\partial L}{\partial \mathbf{X}}$. Using the chain rule and matrix notation, we have:

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

To derive $\frac{\partial L}{\partial \mathbf{W}}$, we start by considering the scalar form of the loss with respect to a single element W_{ij}:

$$\frac{\partial L}{\partial W_{ij}} = \sum_{k=1}^{N} \sum_{l=1}^{m} \frac{\partial L}{\partial Y_{lk}} \frac{\partial Y_{lk}}{\partial W_{ij}}. \tag{15}$$

Given that $\mathbf{Y} = \mathbf{W}\mathbf{X} + \mathbf{B}$, we have:

$$\frac{\partial Y_{lk}}{\partial W_{ij}} = \frac{\partial (\mathbf{W}\mathbf{X})_{lk}}{\partial W_{ij}} = X_{jk} \delta_{il}. \tag{16}$$

Thus,

$$\frac{\partial L}{\partial W_{ij}} = \sum_{k=1}^{N} \sum_{l=1}^{m} \frac{\partial L}{\partial Y_{lk}} X_{jk} \delta_{il} \tag{17}$$

$$= \sum_{k=1}^{N} \frac{\partial L}{\partial Y_{ik}} X_{jk}. \tag{18}$$

This result can be written in matrix form as:

$$\frac{\partial L}{\partial \mathbf{W}} = \frac{\partial L}{\partial \mathbf{Y}} \mathbf{X}^T. \tag{19}$$

Lastly, for $\frac{\partial L}{\partial \mathbf{b}}$, we sum the gradients over all elements in the batch:

$$\frac{\partial L}{\partial \mathbf{b}} = \sum_{i=1}^{N} \frac{\partial L}{\partial \mathbf{y}^{(i)}} = \left(\frac{\partial L}{\partial \mathbf{Y}}\right) \mathbf{1} \tag{20}$$

where $\mathbf{1}$ is a column vector of ones with appropriate dimensions.

The vectorized expressions for backpropagation are now fully derived. Below is the relevant implementation of the forward and backward passes.

```
# Perform the forward pass
b_repmat = np.tile(b, (1, batch))
Z = np.dot(W, X) + b_repmat
```

Fig. 1. Relevant code in fully_connected_forward

```
# Reshape the input vector so that all features for a single batch
# element are in the columns. X is now as defined in the assignment.
X_reshaped = np.reshape(X, (features, batch), order='F')

assert W.shape[1] == features, f"Expected {features} columns in the weights matrix, got {W.shape[1]}"
assert W.shape[0] == len(b), "Expected as many rows in W as elements in b"

# Implement it here
dldX = np.dot(W.T, dldZ)
dldX = np.reshape(dldX, sz, order='F')  # Reshape to original input size
dldW = np.dot(dldZ, X_reshaped.T)
dldb = np.sum(dldZ, axis=1, keepdims=True)

return dldX, dldW, dldb
```

Fig. 2. Relevant code in fully_connected_backward

Exercise 3

Here is the derivation of the backpropagation expression using what has been written from the instructions and the chain rule.

$$\frac{\partial L}{\partial x_i} = \sum_{l=1}^{n} \frac{\partial L}{\partial y_l} \frac{\partial y_l}{\partial x_i} = \sum_{l=1}^{n} \frac{\partial L}{\partial y_l} \frac{\partial}{\partial x_i} \left(\max(x_l, 0)\right) \tag{21}$$

$$= \begin{cases} \frac{\partial L}{\partial y_i} & \text{if } x_i > 0 \\ 0 & \text{else} \end{cases} \tag{22}$$

The relevant code in the forward and backward pass was implemented as in figures below

```
# Implement here
dldX = (X > 0).astype(float) * dldZ
return dldX
```

Fig. 3. Relevant code in relu_backward

```
# Implement here
Z = np.maximum(X, 0)
return Z
```

Fig. 4. Relevant code in relu_forward

Exercise 4

By utilizing

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

... One can derive and expression for $\frac{\partial L}{\partial x_i}$:

$$\frac{dL}{dx_i} = \frac{d}{dx_i}\left(-x_c + \log\left(\sum_{j=1}^{m} e^{x_j}\right)\right) =$$

$$\begin{cases} \frac{e^{x_i}}{\sum_{j=1}^{m} e^{x_j}} - 1, & i = c \\ \frac{e^{x_i}}{\sum_{j=1}^{m} e^{x_j}}, & i \neq c \end{cases} = \begin{cases} y_i - 1, & i = c \\ y_i, & i \neq c \end{cases}$$

The relevant code in softmaxloss code was implemented as found in figures

```
# Implement here

y = np.exp(x) / np.sum(np.exp(x), axis=0, keepdims=True)
y[labels, np.arange(batch)] -= 1
dldx = y / batch

return dldx
```

Fig. 5. Relevant code in softmaxloss_backward

The backward pass was implemented as

```
# Implement here

y = np.exp(x) / np.sum(np.exp(x), axis=0, keepdims=True)
log_y = -np.log(y)
L = np.mean(log_y[labels, np.arange(batch)])
return L
```

Fig. 6. Relevant code in softmaxloss_forward

Exercise 5

This exercise was cut out

Exercise 6

See figure 7 for the kernels of the first layer's filters of the CNN. These kernels are supposed to extract the most rough defining features of the written digits and sift out some classes for a given picture. Subsequent layers are supposed to extract more fine details to eventually classify the input with higher certainty that it is correct. I find it tricky to judge anything exact by just observing the filters but generally they seem to resemble lines in different angles, i.e edge detectors mostly. This guess is further justified by the nature of the input data we are handling (white digits against black background) and it seems like a reasonable first step if I would have been a classification network for MNIST data.

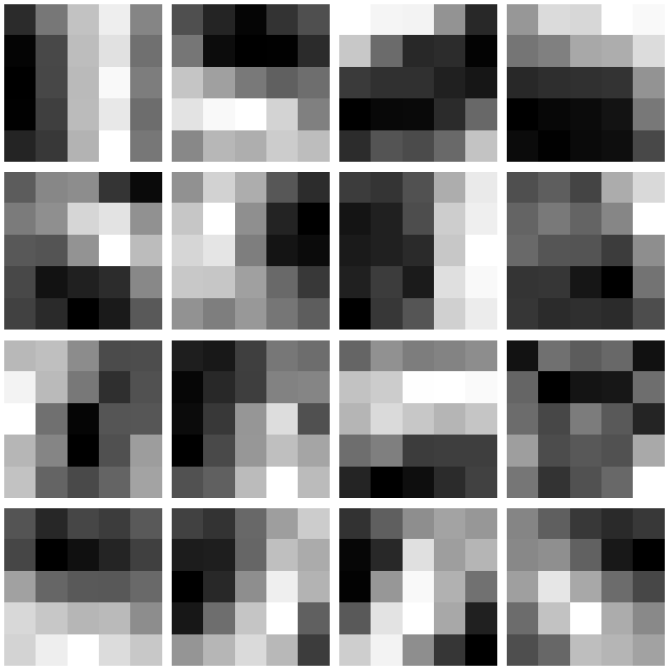

Fig. 7. Kernels of the first convolutional layer in the CNN trained on MNIST dataset.

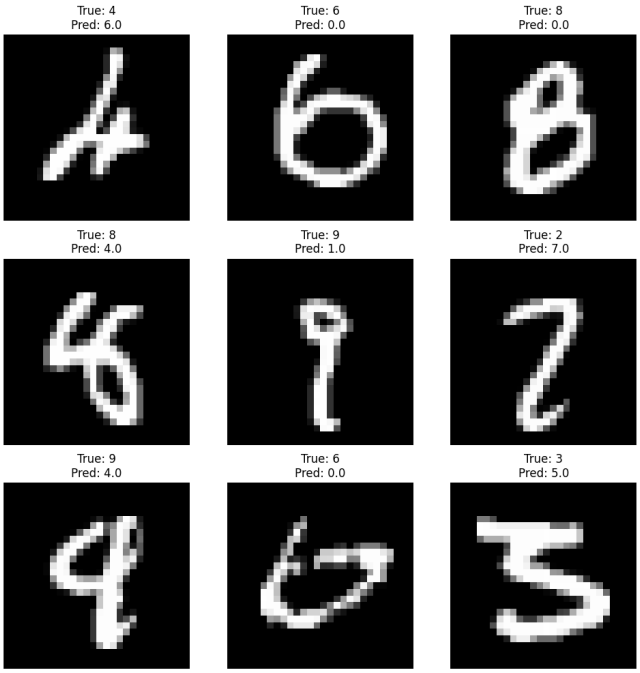

Fig. 8. Examples of misclassified digits from the MNIST test set.

TABLE I
PRECISION AND RECALL FOR EACH DIGIT CLASS IN THE MNIST TEST SET.

Class	Precision	Recall
0	0.9634	0.9939
1	0.9783	0.9947
2	0.9862	0.9690
3	0.9881	0.9851
4	0.9837	0.9837
5	0.9810	0.9865
6	0.9832	0.9781
7	0.9721	0.9844
8	0.9842	0.9610
9	0.9848	0.9663

However, even though the network is effective (98% accuracy in test set), it is not able to classify all digits perfectly; see figure 8 for digits that were missclassified. For a more effective overview, see the confusion matrix (fig 12) that showcases the performance on the test set per digit and what digits that they were misclassified for instead. Lastly, precision and corresponding recall are summarized in Table 1.

The number of parameters of the CNN and the model's general structure that was able to achieve these results are summarized in table 2. From there it is clear that (if it was not already) it is only in the convolutional layers and the dense layer where the model has trainable parameters and deeply learns. Note that the CNN has the classic structure where the number of parameters increases for every convolutional layer as the network extracts more fine features. The last trainable layer is the dense layer and after passing through it "SoftmaxLoss" executes the final step in classification guessing on the most probable class based on the dense layer's output.

Let me make some more comments, starting from 8 because the results are interesting when pondering further. In basically every case of the selected missclassified images you can tell how the network had bad luck in its' "thinking" and why it made the wrong guesses. Take the upper middle one for example, we can tell that it is indeed a 6, but it is unusually round and the "head" of the digit is unusually small, thus resembling the digit 0 more than the digit 6 usually does. The lower middle one in the figure on the other hand has a similar poor shape, but in this case the network found that it most resembled a 6 instead of the true class 0. Even I could have made a wrong guess in this case I feel, the same goes for the lower right digit resembling a 5 as much as a 3 in my opinion.

Recall results provides further insights into the networks performance on the MNIST dataset. Take, for instance, the case of digit 8, which has the lowest recall of 96.10% and a precision of 98.42% (not lowest). This means that out of all actual 8s in the test set, our model correctly identified 96.10%

TABLE II
TRAINED CNN MODEL FOR MNIST CLASSIFICATION.

Layer	Num Weights	Num Biases	Num Trainable Params
Input	0	0	0
Convolution	400	16	416
Relu	0	0	0
MaxPool	0	0	0
Convolution	6400	16	6416
Relu	0	0	0
MaxPool	0	0	0
Dense Layer	7840	10	7850
SoftmaxLoss	0	0	0
Total	**14640**	**42**	**14682**

of them (recall). Conversely, when our model predicted a digit as 8, it was correct 98.42% of the time (precision). The slightly lower recall compared to precision suggests that the model occasionally misses some 8s. However, when it does predict 8, it's highly confident and usually correct.

As a result of the fact that some digits has similarities, we can see in the confusion matrix that for example the digit 2 is mostly misclassified as 7 and vice versa. From table 1 and the confusion matrix we see that the network is the most effective at classifying digit 3, likely because of its' distinct shape. We should also bear in mind that we are not considering any cases where there is ambiguity but the network still is able to make a correct classification despite digit similarities. We are still dealing with an effective network for the given task. Even though MNIST classifiers are something common and pretty well known by now by perhaps most engineers, it still feels impressive seeing it play out like this!

Exercise 7

The goal is to design and train a CNN for classification of the CIFAR dataset and this time around the input data is a lot more complex than written digits of MNIST. The data is multi-channel, in color (R,G,B), and there are many classes that are both distinctly different and also some that are quite similar (at least from a "computer" view). A boat is perhaps distinctly different from an animal, but how does one tell different animals apart from each other?

Development Process: The starting point was the baseline model given from the code stub. Trained on the small dataset the accuracy on the validation data set was 49%. The large dataset replaced the small dataset in training and the development process was on!

When still using the baseline model and training it on the larger dataset the accuracy was again 49%. I had to run a couple of times however, I get weird errors in softmaxloss_backward() sometimes where i tries to access index 255, which is obviously out of bounds when we have 10 classes. This might be related to numerical instability of some kind that I first felt I did not have time to look into. However, after much frustration I re-implemented the function and made sure that the indexing was kept between 0 and 9 and it seemed to fix the problem.

As I am already familiar with CNNs I have several ideas where it could be improved and I will just apply them all at once and continue until the target is reached (55%).

As stated earlier, we have more complex input data, thus it is justified that we should increase the number of convolutional layers to extract more fine details. An additional convolutional layer is added so that I have 3 convolutional layers. The first layer extracts the most rough details of the input picture (could be compared to vehicle/animal). The subsequent layers extract finer details (what animal?) and it is generally a good idea to increase the number of filters for every layer by a constant multiplier. A common choice I think is doubling the number of filters, thus (16, 32, 64). I want to keep things simple as

TABLE III
TRAINED CNN MODEL FOR MNIST CLASSIFICATION.

Layer	Num Weights	Num Biases	Num Trainable Params
Input	0	0	0
Convolution	432	16	448
Relu	0	0	0
MaxPool	0	0	0
Convolution	4608	32	4640
Relu	0	0	0
MaxPool	0	0	0
Convolution	18432	64	18496
Relu	0	0	0
MaxPool	0	0	0
Fully Connected	262144	256	262400
Relu	0	0	0
Fully Connected	2560	10	2570
SoftmaxLoss	0	0	0
Total	**288176**	**378**	**288554**

well, so I reduced the kernels to 3x3 size instead of 5x5 and I adjusted padding. Since we have more convolutional layers and filters, I find it appropriate to add another dense layer at the end before the last dense layer + softmax loss so the network can interpret the output from the convolutional layers more sophisticated. Lastly I feel that regularization techniques are missing except for weight decay, so I would really like to add dropout to every learnable layer except the last dense layer, but I was dumbfounded when I realized that it is not implemented... Let us see how this goes.

After these changes the performance increased to 56%. Since the accuracy is strong enough (56% 55%), I will end the development process early here even though I have not experimented with the hyperparameters further as that would require tiresome re-runs and I have already put some time in re-implementing softmaxloss_backward() so it does not crash in the middle of a run. On the internet it is not difficult to find examples that have accuracies far above reaching as high as in 90% area (this one included a 10-conv later CNN) so there is definitely room for a network to deeply learn more about the features in data. The plot in figure 13 also alludes to that the regularization is poor, implementing dropout would improve.

Final model: The final model is structured as detailed in Table III. The model comprises three convolutional layers, each followed by a ReLU activation function and a max-pooling layer. This is followed by two fully connected layers. The total number of trainable parameters in the model is 288,554, comprising 288,176 weights and 378 biases and it is thus **alot** bigger than the previous one (unsurprisingly).

The confusion matrix is seen in fig 12 and the first convolutional layer filters are seen in fig 9. The kernels are smaller, the data is complex with noise and colour channels this time and it is difficult to see any pattern in how the kernels ”specialize”. It seems that the kernels separate colours maybe because some of them are almost entirely one color. The table with precision and recall is found in table IV. A plot with some examples of missclassifications are seen in figure 10.

TABLE IV
PRECISION AND RECALL FOR EACH CLASS.

Class	Precision	Recall
0 (airplane)	0.6283	0.5730
1 (automobile)	0.7388	0.5940
2 (bird)	0.4587	0.4110
3 (cat)	0.4144	0.3630
4 (deer)	0.4548	0.4930
5 (dog)	0.5600	0.3920
6 (frog)	0.5904	0.7150
7 (horse)	0.5136	0.7200
8 (ship)	0.6535	0.6940
9 (truck)	0.6249	0.6580

As argued earlier and correctly predicted apprently (wohoo!) the model was able to differentiate vehicles from animals which is clear from the confusion matrix and these were not as commonly missclassified against each other. Overall however, the accuracy is much lower than it was in the previous exercise, as this requires a more sophisticated model and further development to reach the highest levels of accuracies akin to the previous results.

Glancing over the table IV we see that there is a larger difference in precision and recall comparing the different classes as well. Apparently cats are the most unlikely to get right and cars the most likely to get right by the model. Having this in mind looking at some examples of misclassified images we see 2 incorrect cat predictions in middle picture and upper middle. These have in common that it is a lot of white color in the picture which I suspect might affect the result (to be fair the dog in the middle picture are more cat-alike than many types of dogs). This dog is also held in someone’s arms, and the enviornment might have an effect here too.

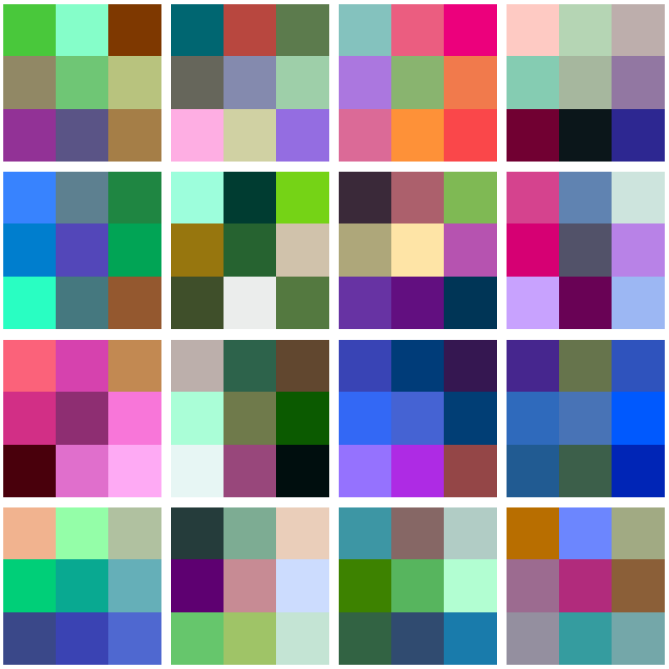

Fig. 9. Kernels of the first convolutional layer in the CNN trained on CIFAR dataset (RGB).

II. REFERENCES

Fig. 10. Examples of misclassified objects in images from the CIFAR test set.

III. Appendix

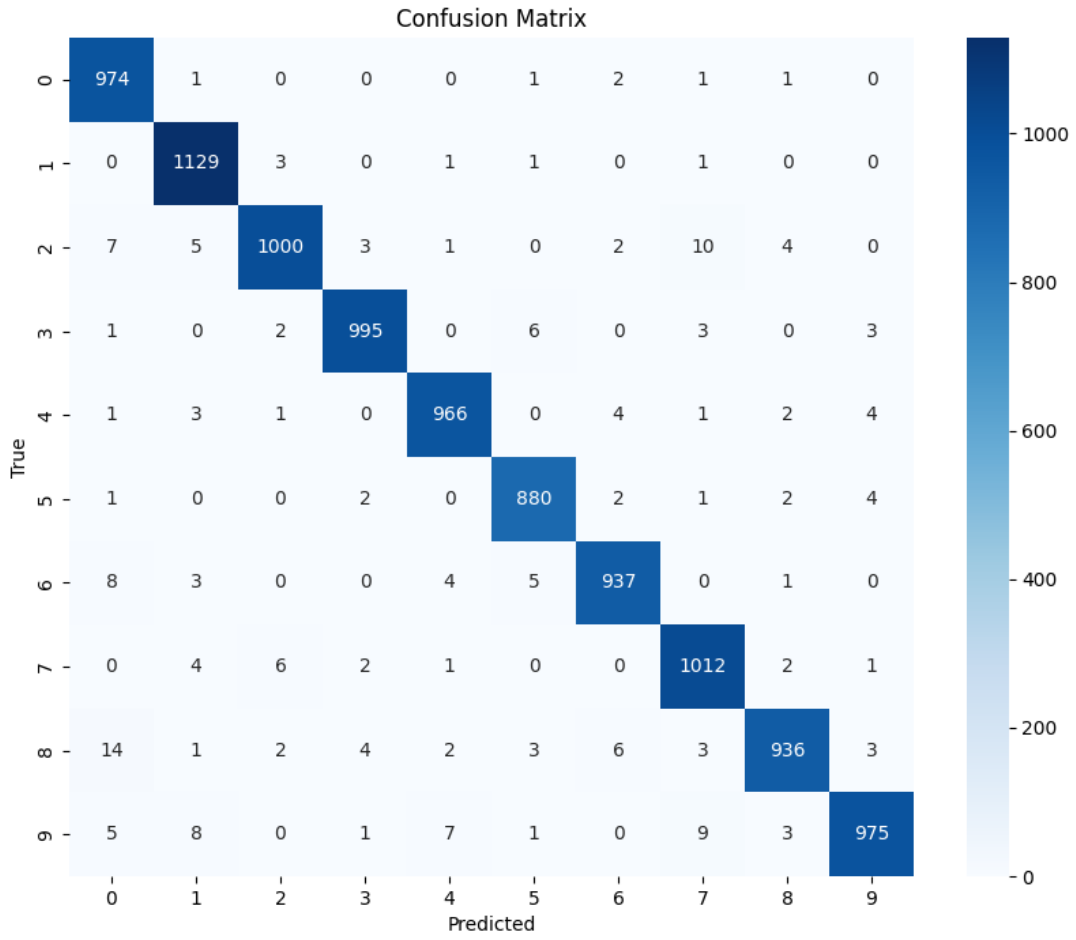

Fig. 11. Confusion matrix showing the performance of the CNN on the MNIST test set.

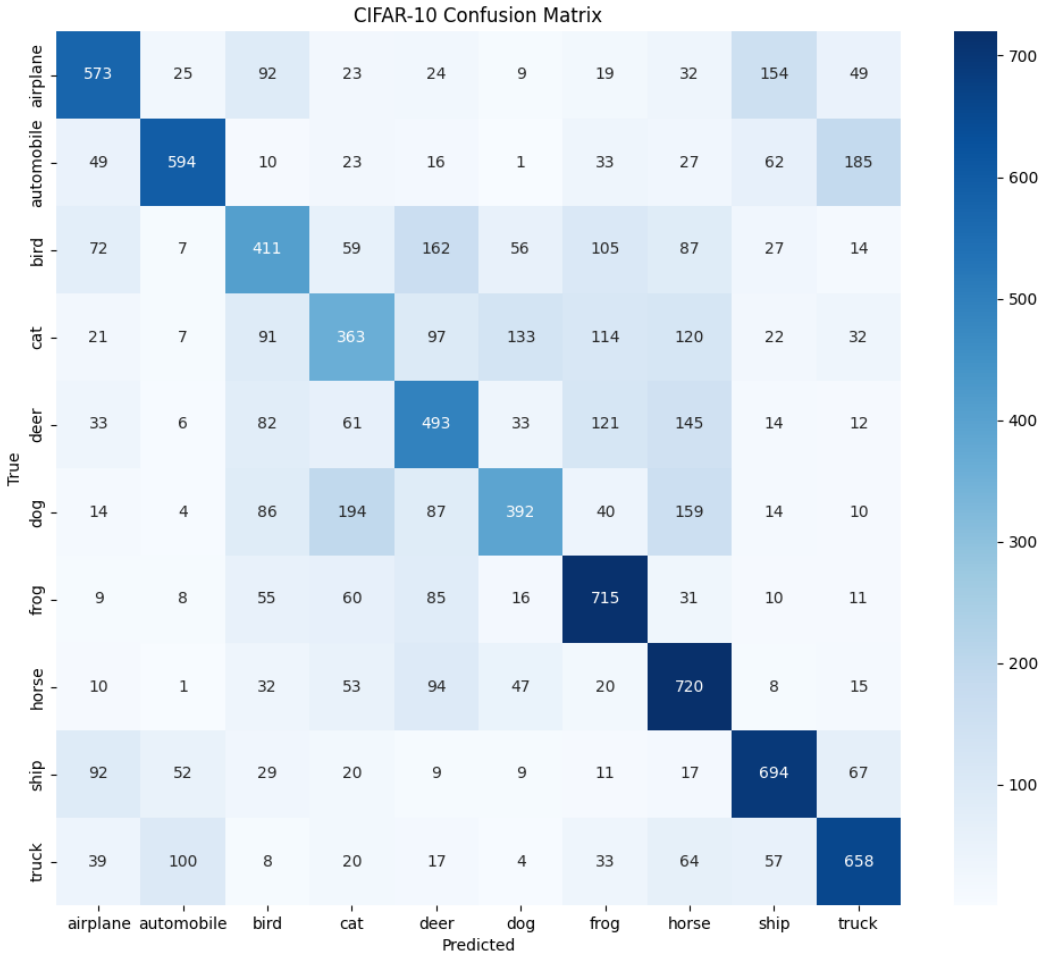

Fig. 12. Confusion matrix showing the performance of the CNN on the CIFAR test set. The overall accuracy was 56%

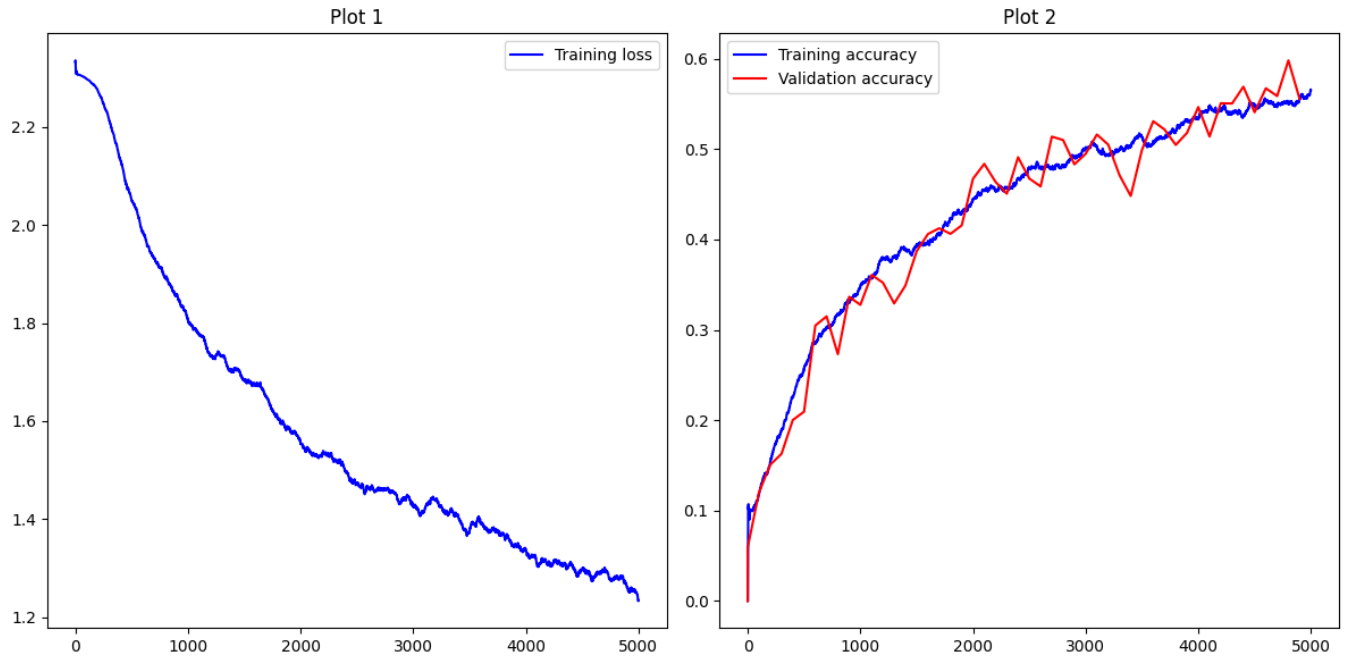

Fig. 13. Plot VS iteration and accuracy VS iteration for training and validation data. The jumpy validation data accuracy is a sign of poor regularization I think (volatile results due to learning noise in training).