Homework #1

Siyeol Choi
Endri Taka

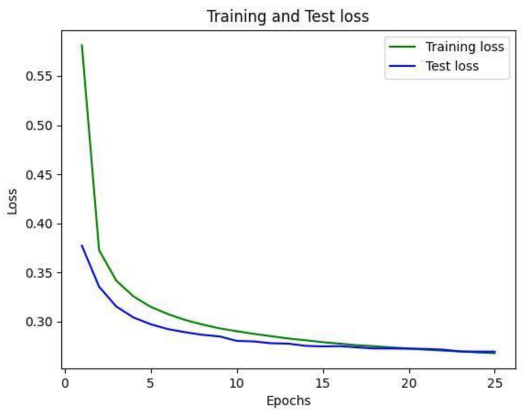
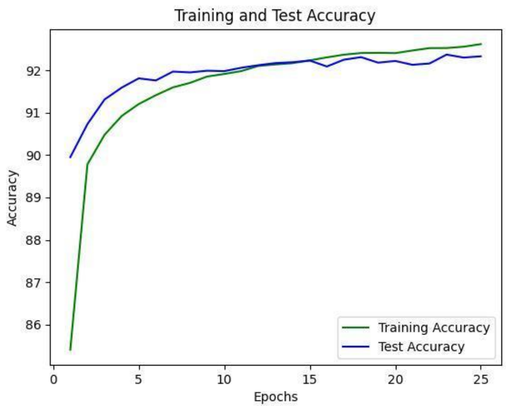
Problem 1.

Question 1:

In addition to putting the model to cuda by “model.to(device)”, images and labels for training and testing should also be moved to cuda. Therefore, in line 75 and 113 (code *p1_q1.py*), “images, labels = images.to(device), labels.to(device) “are added.

Question 2:

Since Training Loss, Test Loss, Training Accuracy, Training Loss are all being calculated and were being printed into the console, we saved into four lists saving each of the results. Then, we plotted it using matplotlib.pyplot displaying Training data in green and Test data in blue. The overall plots are as follows.

[Loss plot]	[Accuracy plot]
	

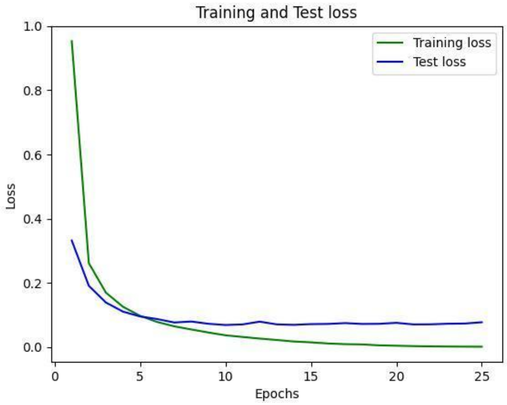
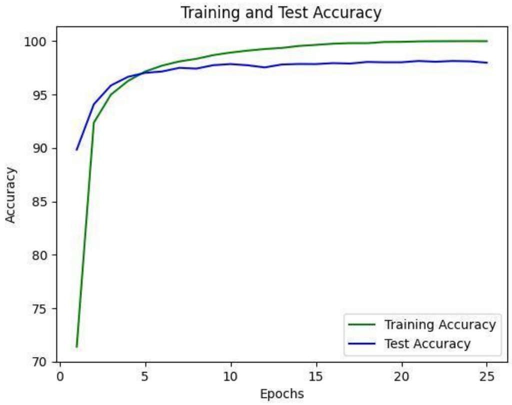
Question 3:

To fill the table required in this question we add some code to measure the times for training and inference (codes *p1_q3_training_time.py* and *p1_q3_inference_time.py* respectively). Because GPU operations are asynchronous and thus computations can occur in parallel, we use *torch.cuda.synchronize()*, so that the time recording takes place only when the process running on the GPU has finished its execution. To measure the time we use torch.cuda.event().

Training accuracy [%]	Testing accuracy [%]	Total time for training [s]	Total time for inference [s]	Average time for inference per image [ms]	GPU memory during training [MB]
92.60	92.33	166.39	1.069210	0.106921	689

<u>Problem 2.</u>

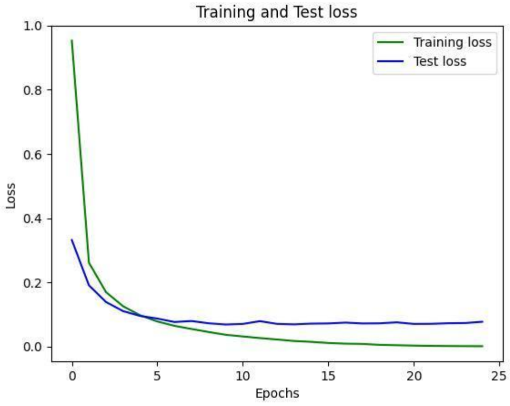
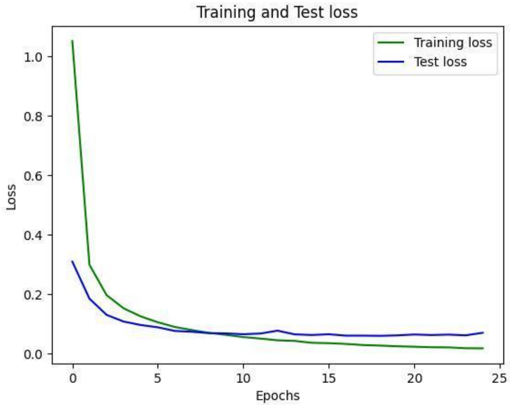
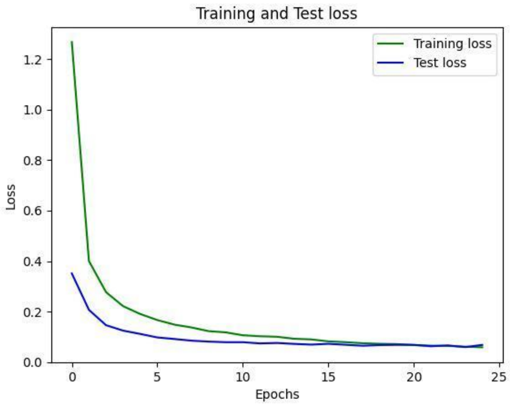
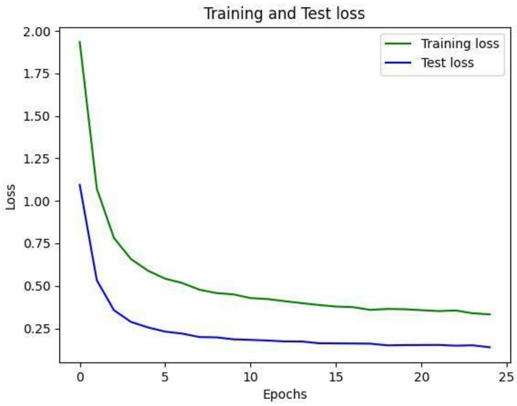
Question 1:

[Loss plot]	[Accuracy plot]
	

The model is overfitting because the test loss is bigger than the training loss, thus it doesn’t generalize well on “unseen” data. More specifically we notice that after five epochs, the test loss remains almost constant while the training gets lower with the increment of epochs (as expected).

Question 2:

Dropout	Test Loss	Test Accuracy	Analysis
0.0	0.0776	97.98%	Overfitting
0.2	0.0703	98.12%	Slightly overfitting after 8 epochs
0.5	0.0682	98.18%	Best result (Smallest Loss)
0.8	0.1384	96.45%	Underfitting

Dropout 0.0	Dropout 0.2
	
Dropout 0.5	Dropout 0.8
	

Added variable "dropout_prob" to test 0.0, 0.2, 0.5, and 0.8 probability of dropouts in line 46. Since drop-out is a method of dropping random neurons between connected layers in a designated probability, the higher the assigned probability is, the fewer remaining neurons would be. Therefore, 0.0 probability did not drop any neuron between layers and it resulted in overfitting. On the other hand, 0.8 dropped too many neurons, so the test loss became lower than the train loss. This happens because dropout is enabled only during training and not in the testing phase. In between, there were 0.2 and 0.5 probability of drop-outs which had seemingly a good result, but 0.5 probability was better since it did not underfit or overfit within 25 epochs and it had the smallest loss in the end.

Question 3:

Since the best dropout rate of Problem 2, Question 2 was 0.5, we conducted the experiment with the dropout rate (in this case X) of 0.5 which had the smallest loss. On top of that, we added an extra experiment which had an extra step of Normalization at line 39 (for training datasets) and 40 (for test datasets) which helps the model learn faster.

The following result is as follows.

dropout	training acc [%]	test acc [%]	total time [s]	first epoch 96%
0.5	98.19	98.18	244.93	8
0.5+norm	98.51	98.32	359.38	6

The total time with the normalization technique seemed to increase, but the first epoch that reached the accuracy of 96% percent was two epochs smaller than the experiment without the normalization. Our team got curious about why the total time increased since normalization is intended to make the training fast, so we conducted another experiment with more normalization with parameters of (mean=0.456, std=0.224). The total time even increased to 375.27 second from 359.38 which was experimented with given mean and std. Therefore, we made an assumption that the reason total time increased is because the computation of the gradients during training becomes more difficult to compute. Presumably the values of the data play a role in the gradients calculation process. With input normalization, the values of gradients during back propagation should change, compared to having “raw” data. This can lead to timing differences, as we noticed during our experimentations. Furthermore, we checked that the data type of the inputs and the model’s parameters (floating point 32-bits) didn’t change with normalization, so we can eliminate a hardware (GPU) execution issue.

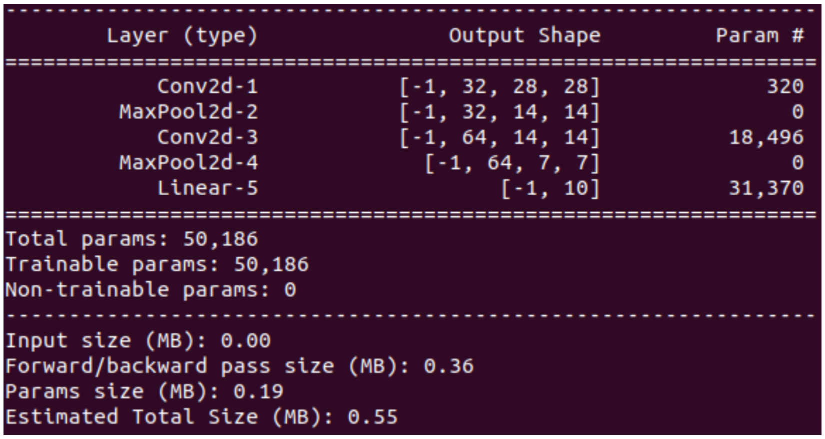
However, the first epoch of 96% accuracy was smaller with normalization, so we can conclude that the normalization actually helps the model to learn faster as expected. Moreover, both training and testing accuracy after 25 epochs is greater when adding normalization. This means that the desired accuracy can be achieved with fewer epochs (model learns faster). Hence, when training for the same number of epochs (25 in our case) we can achieve even better accuracy.

<u>Problem 3.</u>

Question 1:
By utilizing the *torchsummary* and *thop* libraries and the fact that one MAC is the equivalent of two FLOPS, we fill the following Table:

Model name	MACs	FLOPs	# parameters	Model size [MB]	Saved model size [KB]
SimpleCNN	3907456	7814912	50186	0.55	200

As we notice in the table, the “Model size” and the “Saved model size” are different. To explain why this happens, we present the output of the *torchsummary*:



```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 32, 28, 28]             320
         MaxPool2d-2           [-1, 32, 14, 14]               0
            Conv2d-3           [-1, 64, 14, 14]          18,496
         MaxPool2d-4             [-1, 64, 7, 7]               0
            Linear-5                   [-1, 10]          31,370
================================================================
Total params: 50,186
Trainable params: 50,186
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.36
Params size (MB): 0.19
Estimated Total Size (MB): 0.55
```

The total estimated size in the *torchsummary* (as found in the code *torchsummary.py* at github), is the summation of the input size, the forward/backward pass size and the parameters size. Since in our case the input size is 28x28 pixels, *torchsummary* considers it very small, thus reporting 0.00. Moreover, it calculates the amount of memory required for the forward and the backward pass, as well as the parameters size, resulting in an estimated total size of 0.55MB.

In contrast, *torch.save()* saves the architecture of the model as well as the parameters (weights and biases). In our case, the saved model size is 200KB. This is reasonable because the total parameters size is 50186*4/1024 = 196.04KB (or 0.19MB as reported from *torchsummary*), thus the rest 3.96KB is utilized for the model architecture.

Questions 2 & 3:

First of all, we tested various models with different channels of convolutional layers and we kept one linear layer as the baseline model. We tested and compared the following combinations of output channels:

1st Conv/ 2nd Conv Layers output channels: **8/16**, **8/32**, **16/32**, and **32/64**.

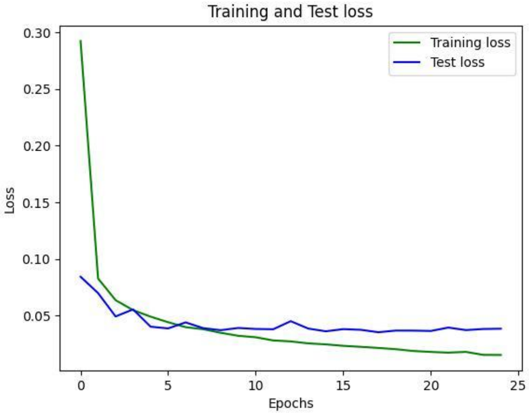
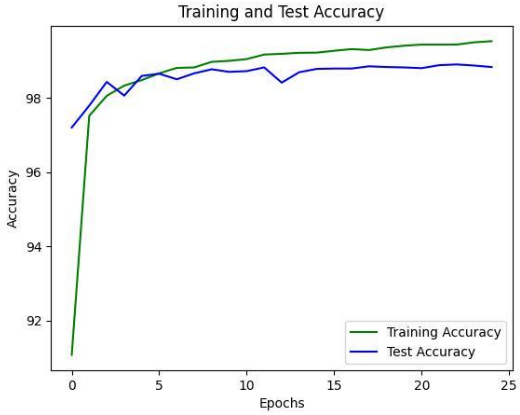
We found that the smallest model (**8/16**) has slightly lower accuracy compared to the biggest one (**32/64)**. More specifically, the training accuracy (10 epochs) for the 8/16 was 98.78% and testing accuracy 98.57%, while for the 32/64 was 99.35% and 98.89%, respectively. For completeness purposes, the 8/32 had 99.2% and 98.71%, while the 16/32 had 99.14% and 98.73% for training and testing, respectively. Hence, we can conclude that the difference among all of them is very small.

Furthermore, by comparing the model sizes, 8/16 has 9098 parameters with total size of 0.13 MB, while the 32/64 has 50186 and 0.15MB, respectively. Thus, the smaller model has significantly lower size, and it is preferable. For the above reasons, we choose the **8/16** model. Note that we also tested smaller models, but the accuracy was competitive leading us to use the aforementioned model for our exploration.

We name our model as ***CNN_2_8_16***, where 2 stands for the number of convolutional layers, and 8, 16 the number of output channels for each layer. The extended table is:

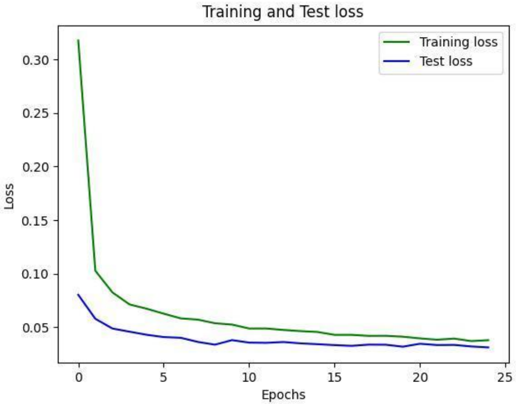
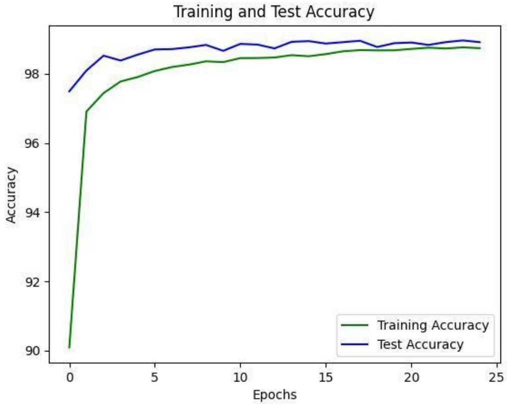
Model name	MACs	FLOPs	# parameters	Model size [MB]	Saved model size [KB]
SimpleCNN	3907456	7814912	50186	0.55	200
CNN_2_8_16	299488	598976	9,098	0.13	40K

Afterwards, we test if our chosen model is overfitting. Thus, we plot the loss and accuracy plot for 25 epochs. As shown below, our model is overfitting, and to solve that we test various methods.

[Loss plot]	[Accuracy plot]
	

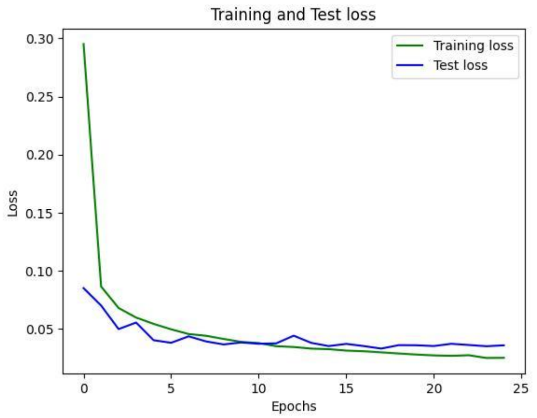
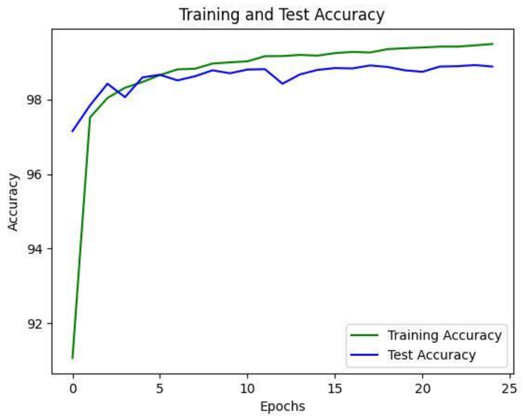
1) <u>Dropout method</u>

In this method, we applied a dropout layer between the second Maxpool layer and the Linear layer in line 73. Default probability 0.5 led to an underfitting, so we tried 0.2 instead and it produced a reasonable outcome, as can be shown below. The total reported accuracy at 25 epochs was 98.71% and 98.95% for training and testing, respectively.

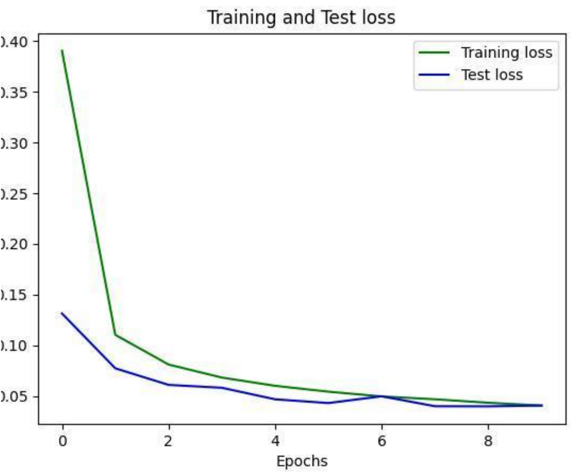
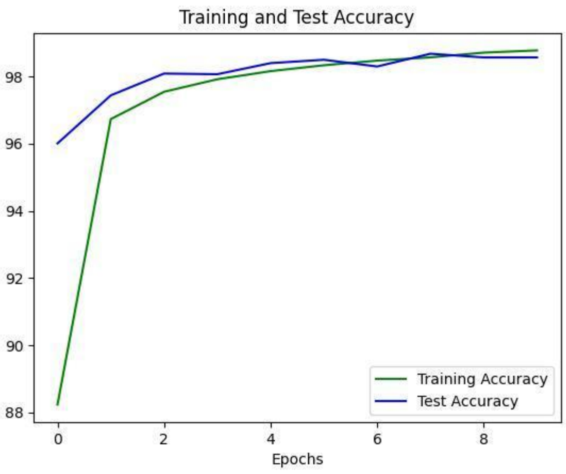
[Loss plot]	[Accuracy plot]
	

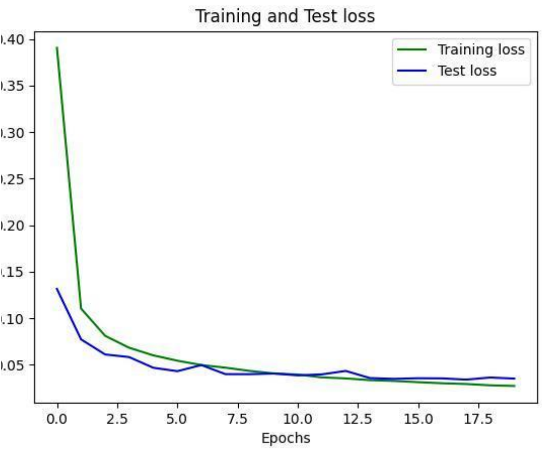
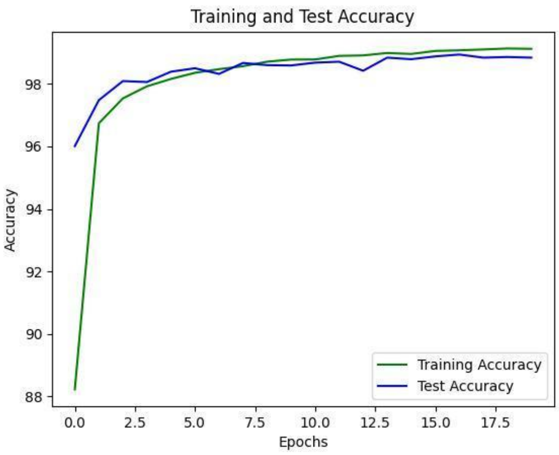
2) L2 normalization

In this method we add to the loss function the sum of the squared norms of our weights and multiply by a constant, lambda. In our case, we tested various values of lambda, but we didn't find any significant improvement. Below we present the plots for lambda=0.0001. Clearly, L2 normalization doesn't seem to adequately solve the overfitting problem. For 25 epochs, the accuracy was 99.48% and 98.88% for training and testing, respectively.

[Loss plot]	[Accuracy plot]
	

In our quest to build a model that regularizes well, we had the idea of changing the learning rate. By testing that, we found that a learning rate of 0.005 instead produces good results without any technique for overfitting. Below we show the plots for 10 and 20 epochs. As we can see, this model (with the chosen hyperparameters) generalizes very well even for a small number of epochs. The training and testing accuracy for 10 epochs is 98.78% and 98.55%, respectively, while for 20 epochs is 99.15% and 98.89%, respectively. We notice that accuracy is improving at a relatively small rate by increasing the number of epochs. Since our model generalizes well, we conclude that if we want better accuracy we can achieve it by training for more epochs.

[Loss plot] 10 epochs	[Accuracy plot] 10 epochs
	

[Loss plot] 20 epochs	[Accuracy plot] 20 epochs
	

The reason we wanted to use a model that is smaller in size, thus having less parameters and requiring less memory, is because edge devices are computationally limited platforms. Thus, if the same ML task can be solved with a smaller size, that can have significant benefits implementing the inference on edge. More specifically, the application’s restrictions, e.g., latency or throughput, can be more easily met (lower number of FLOPS), while at the same time the energy and power consumption are lower.

Discussion

Contribution:
We solved every task together. Siyeol concentrated more on plotting and visualizing the result, while Endri concentrated more on estimating the time and memory usage. For the last two questions, Siyeol conducted an experiment mainly about adding dropouts and Endri experimented with regularization and manipulating learning rates. Moreover, Endri tested a CNN with batch normalization layers but without better results compared to the presented model, therefore we didn’t include that on our report.

What we learned:
1) Importance of avoiding overfitting and underfitting. Plus, the methods to avoid overfitting and underfitting regarding the model size and accuracy.
2) Training an AI model is very time consuming and it takes a lot of time to debug, so we have to concentrate when writing codes. Moreover, the process of finding the perfect situation is quite empirical.