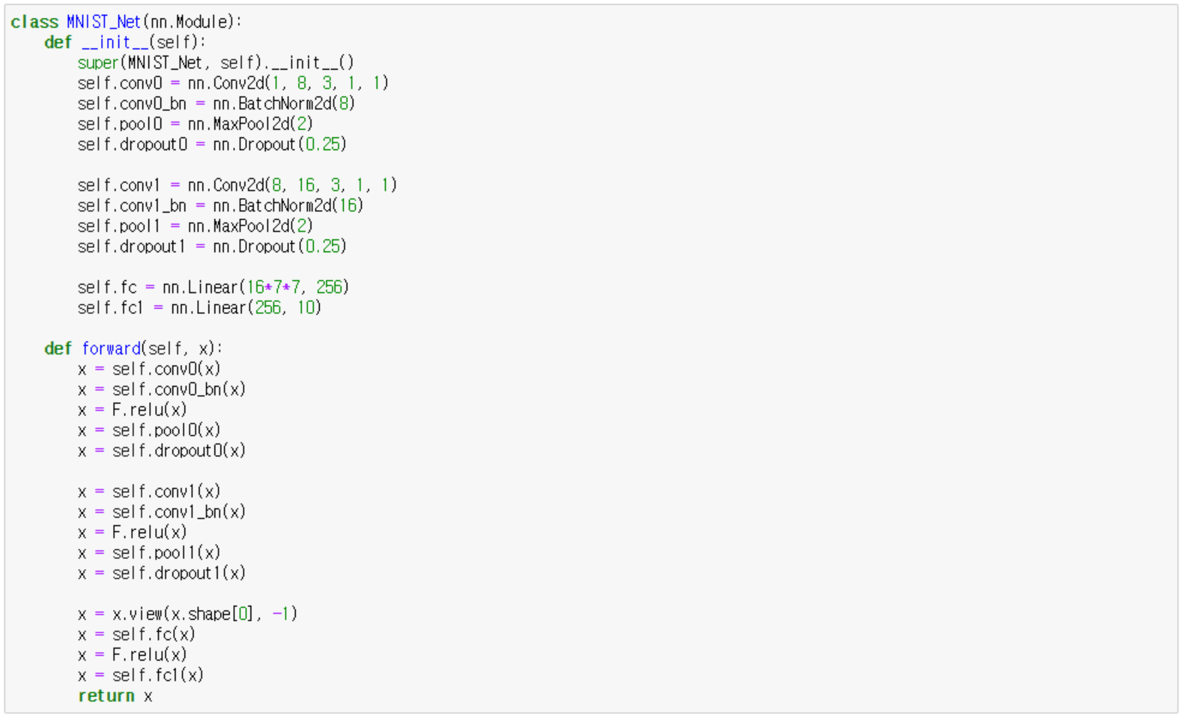
Homework 1. Pytorch Tutorial for CV

Chaelin Kim
(20205096)
GSCT
`chaelin.kim@kaist.ac.kr`

1 Playing around with Neural Network

1.1 Building classifier for MNIST dataset

- Summary of your model architecture and motivation for your design choices



```
class MNIST_Net(nn.Module):
    def __init__(self):
        super(MNIST_Net, self).__init__()
        self.conv0 = nn.Conv2d(1, 8, 3, 1, 1)
        self.conv0_bn = nn.BatchNorm2d(8)
        self.pool0 = nn.MaxPool2d(2)
        self.dropout0 = nn.Dropout(0.25)

        self.conv1 = nn.Conv2d(8, 16, 3, 1, 1)
        self.conv1_bn = nn.BatchNorm2d(16)
        self.pool1 = nn.MaxPool2d(2)
        self.dropout1 = nn.Dropout(0.25)

        self.fc = nn.Linear(16*7*7, 256)
        self.fc1 = nn.Linear(256, 10)

    def forward(self, x):
        x = self.conv0(x)
        x = self.conv0_bn(x)
        x = F.relu(x)
        x = self.pool0(x)
        x = self.dropout0(x)

        x = self.conv1(x)
        x = self.conv1_bn(x)
        x = F.relu(x)
        x = self.pool1(x)
        x = self.dropout1(x)

        x = x.view(x.shape[0], -1)
        x = self.fc(x)
        x = F.relu(x)
        x = self.fc1(x)
        return x
```

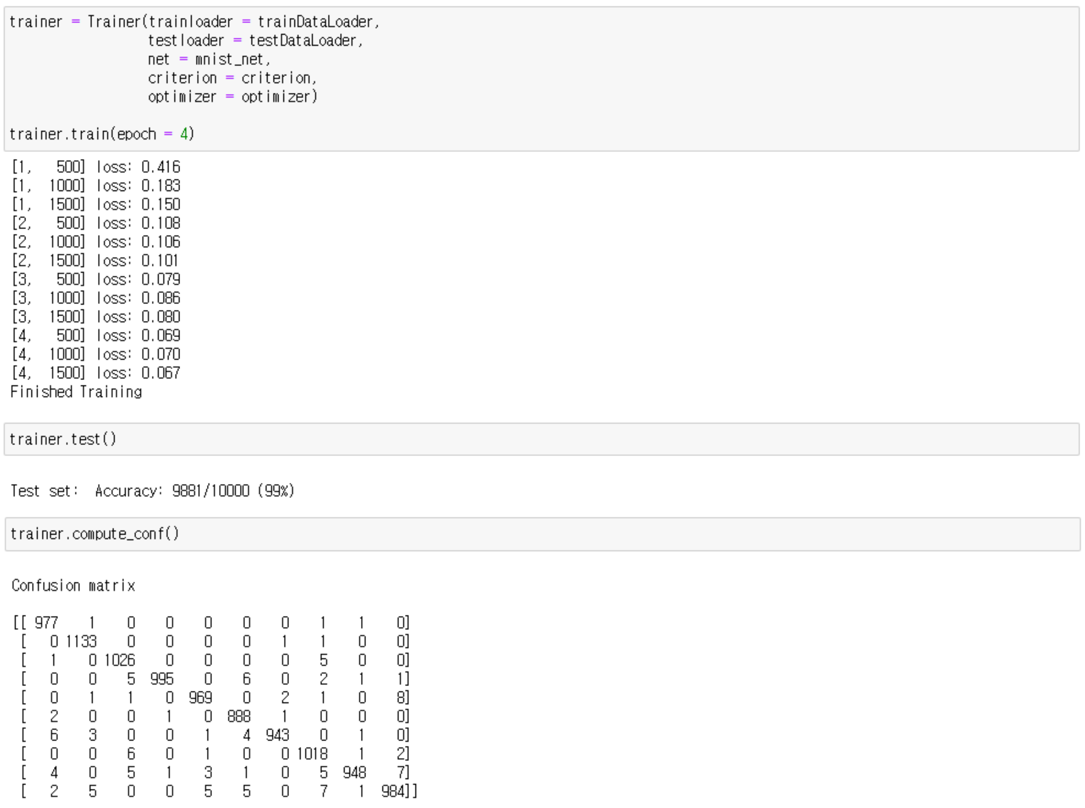
Figure 1: Model architecture of classifier for MNIST dataset

Motivation

- Convolution layer: The spatial information of the image can be maintained, features of the image can be extracted with multiple filters, and the number of parameters are small by sharing the filters.
- Pooling layer: to reduce the number of parameters for preventing overfitting problem
- Relu: to prevent saturated neuron that makes the gradients zero
- Batch normalization: to improve the gradient flow and stabilize the whole training process by making the input unit gaussian

  - Dropout: to prevent overfitting and make the network learn more general by deactivating some neurons during learning and forcing the network to have a redundant representation.

- Classification result including the confusion matrix for each class



```
trainer = Trainer(trainloader = trainDataLoader,
                  testloader = testDataLoader,
                  net = mnist_net,
                  criterion = criterion,
                  optimizer = optimizer)

trainer.train(epoch = 4)
```

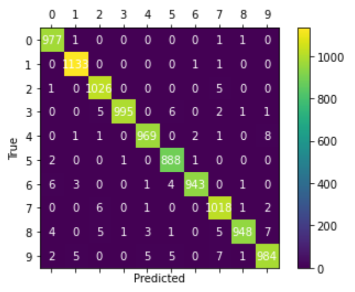
```
[1,   500] loss: 0.416
[1,  1000] loss: 0.183
[1,  1500] loss: 0.150
[2,   500] loss: 0.108
[2,  1000] loss: 0.106
[2,  1500] loss: 0.101
[3,   500] loss: 0.079
[3,  1000] loss: 0.086
[3,  1500] loss: 0.080
[4,   500] loss: 0.069
[4,  1000] loss: 0.070
[4,  1500] loss: 0.067
Finished Training
```

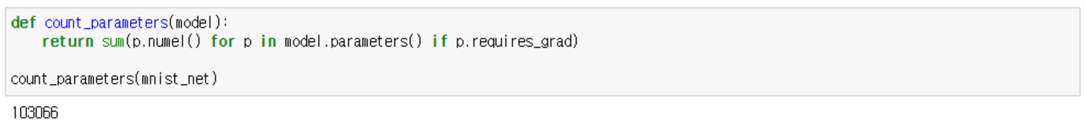
```
trainer.test()
```

```
Test set:  Accuracy: 9881/10000 (99%)
```

```
trainer.compute_conf()
```

```
Confusion matrix

[[ 977    1    0    0    0    0    0    1    1    0]
 [    0 1133    0    0    0    0    1    1    0    0]
 [    1    0 1026    0    0    0    0    5    0    0]
 [    0    0    5  995    0    6    0    2    1    1]
 [    0    1    1    0  969    0    2    1    0    8]
 [    2    0    0    1    0  888    1    0    0    0]
 [    6    3    0    0    1    4  943    0    1    0]
 [    0    0    6    0    1    0    0 1018    1    2]
 [    4    0    5    1    3    1    0    5  948    7]
 [    2    5    0    0    5    5    0    7    1  984]]
```





```
def count_parameters(model):
    return sum(p.numel() for p in model.parameters() if p.requires_grad)

count_parameters(mnist_net)
```

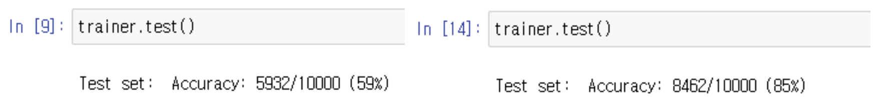
```
103066
```

Figure 2: Classification result (top: training and test, middle: confusion matrix, bottom: the number of parameters)

- Analysis on the most confusing case and the possible reasons
  ⇒ In the confusion matrix above, the case that predicted 4 as 9 came out the highest. This may have been predicted that the long bar on the right in 4 and the bar on 9 are similar. Also, since the mnist dataset is composed with human handwriting, some errors can be made from the dataset itself (e.g. 4 that looks very similar to 9 in the dataset.)

1.2 Written Questions

1. What if we didn't clear up the gradients?
   ⇒ Whenever .background() is called, Pytorch accumulates the gradients in buffers on subsequent backward pass. If we didn't clear up them, it can be mixed with the previous gradients, and values can diverge when the number of iterations increases. So, if you don't want to mix up gradients between minibatches, you have to zero them out at the start of a new minibatch.
2. Why should we change the network into eval-mode?
   ⇒ model.eval() sets the module in evaluation mode. It is a kind of switch for some specific layers/parts (e.g. Dropout, BatchNorm) of the model that behave differently during training and inference time. For example, in eval mode the Dropout is deactivated and BatchNorm uses the parameters saved in training.
3. Why do we need to call the constructor, nn.Module?
   ⇒ torch.nn.Module is the base class for all neural network modules in Pytorch. If we want to create a class that holds our weights, bias, and method for the forward step, then nn.Module provides a number of attributes and methods for them. In order to create a custom network using Pytorch, therefore, the network need to inherit nn.Module.
4. Is there any difference in performance according to the activation function?
   ⇒ Yes. When comparing inference of ReLU and Sigmoid in the same Network, the accuracy of ReLU is about 85%, which is better than 59% of Sigmoid. Sigmoid function has a problem in that when the output approaches 1 or 0 (saturated), the gradient approaches 0. Then, the gradients can't backpropagate through the network. The ReLU function is not saturated in positive region, so there is no vanishing gradient problem in positive region. It is also said that the convergence speed is about 6 times faster than that of the sigmoid function.



```
In [9]:  trainer.test()

         Test set:  Accuracy: 5932/10000 (59%)
```

```
In [14]:  trainer.test()

          Test set:  Accuracy: 8462/10000 (85%)
```

Figure 3: Test time accuracy comparison (Left: Sigmoid / Right: ReLU)

5. Is training gets done easily? If it doesn't, why not?
   ⇒ No, it isn't. As mentioned in the answer of Q5, when using the sigmoid function saturated neuron kills the gradient. As the layer becomes deeper, therefore, layers at the front may not be updated due to the problem of vanishing gradient during backpropagation.
6. Is training gets done easily compared to experiment (2)? If it doesn't, why not?
   ⇒ Yes. As mentioned in the answer of Q5, when using the ReLU function neurons are not saturated in positive region.
7. What would happen if there is no activation function?
   ⇒ If there is no activation function, the output signal would simply be a single linear function and it would be equal to linear classifier. Linear equation is easy to solve but it is limited in its complexity and hard to learn complex functional mapping from data. Introducing non-linear activation functions allows for the network to solve a larger variety of problems.
8. Is there any performance difference before/after applying the batch-norm?
   ⇒ A little bit. In the case of 2-layer network (comparing (7) with (6)), the performance of both is similar at 96%. In the case of a 3-layer network (comparing (8) with (5)), there is a 1% improvement in the performance from 97% to 98%.
9. How did 2-layer neural network and 3-layer neural network behave differently after applying the batch-norm?
   ⇒ Comparing (8) with (7), there is a 2% improvement in the performance from 96% to 98%. As the number of layers increases, a vanishing gradient problem in which the gradient is not transmitted during backpropagation may occur. For this, there is a batch normalization technique that normalizes the distribution of batches in each layer. As the layer becomes deeper, the probability of occurrence of vanishing gradient problem increases, so the effect of batch normalization increases.

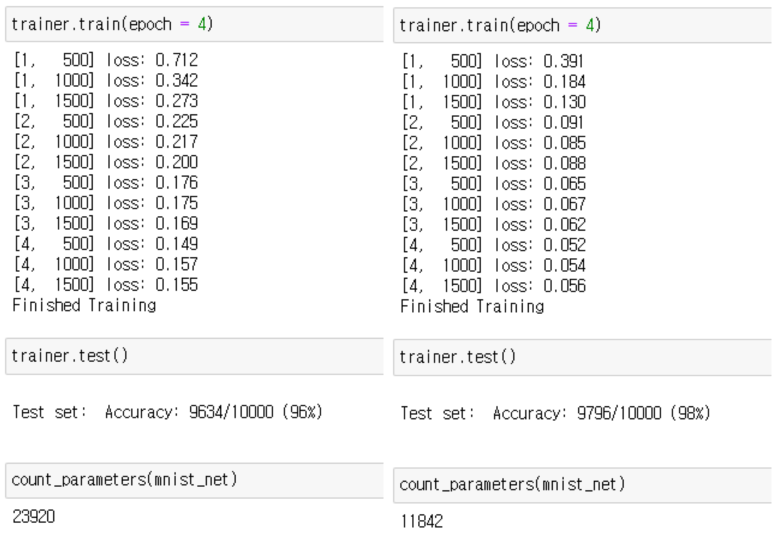
10. What are the problems with the large number of parameters?
⇒ Network that uses many parameters can represent more complicated functions because more can be expressed. Therefore, normally the more parameters, the better to the network. As the network gets bigger, however, the memory usage increases and . So, we need to adjust the network size to suit our resources. Also, overfitting can occur as the number of parameters increases. In this case, regularization should be strongly applied.

11. Given input image with shape:(H, W, C1), what would be the shape of output image after applying 2 (F * F) convolutional filters with stride S?
⇒ ((H-F)/S + 1, (W-F)/S + 1, 2)

Output volume size

- (Output Height) = (H-F+2*P)/S + 1
- (Output Width) = (W-F+2*P)/S + 1
- (Output Channel) = (# of filters)

12. How did the performance and the number of parameters change after using the Convolution operation? Why did these results come out?
⇒ Comparing (9) with (7), the performance improved by 2% (96% $\rightarrow$ 98%) and the number of parameters decreased by about 2 times. (23,920 $\rightarrow$ 11,842) In the linear layer, as the size of the input increases, the number of weights used by the hidden layer increases. This can cause the problems mentioned in Q10. When the convolution layer is used, parameters are used as much as the size and the number of filters, so the number of parameters can be reduced while extracting features.



```
trainer.train(epoch = 4)

[1,   500] loss: 0.712
[1,  1000] loss: 0.342
[1,  1500] loss: 0.273
[2,   500] loss: 0.225
[2,  1000] loss: 0.217
[2,  1500] loss: 0.200
[3,   500] loss: 0.176
[3,  1000] loss: 0.175
[3,  1500] loss: 0.169
[4,   500] loss: 0.149
[4,  1000] loss: 0.157
[4,  1500] loss: 0.155
Finished Training

trainer.test()

Test set:  Accuracy: 9634/10000 (96%)

count_parameters(mnist_net)

23920
```

```
trainer.train(epoch = 4)

[1,   500] loss: 0.391
[1,  1000] loss: 0.184
[1,  1500] loss: 0.130
[2,   500] loss: 0.091
[2,  1000] loss: 0.085
[2,  1500] loss: 0.088
[3,   500] loss: 0.065
[3,  1000] loss: 0.067
[3,  1500] loss: 0.062
[4,   500] loss: 0.052
[4,  1000] loss: 0.054
[4,  1500] loss: 0.056
Finished Training

trainer.test()

Test set:  Accuracy: 9796/10000 (98%)

count_parameters(mnist_net)

11842
```

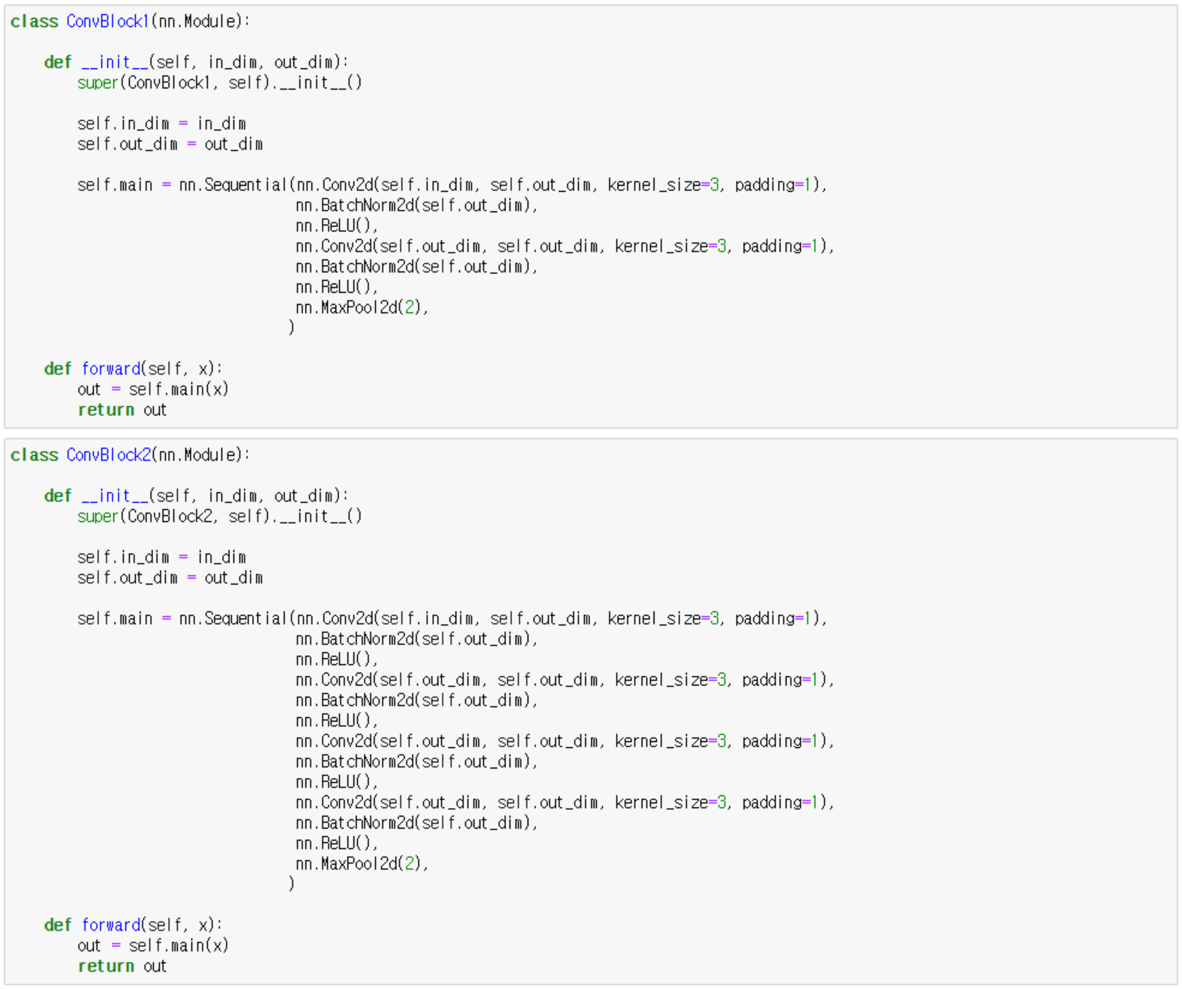
Figure 4: Comparision of training result and the number of parameters (Left: Linear / Right: Conv)

13. How did the performance change after using the Convolution operation? Why did these results come out?
⇒ Comparing (10) with (9), the performance is similar, but the number of parameters is significantly reduced. (11842 $\rightarrow$ 3202) When using a 2D pooling layer, the number of channels is maintained, and the width and height of the input are reduced by extracting the features of each part of the input. This reduces the number of parameters of the network to suppress overfitting problem and reduce memory usage. Also, since the pooling layer has no parameters to be trained, it does not affect the total number of parameters of the network.

2 Playing around with Convolutional Neural Network

2.1 Implementation: VGG-19

- Summary of the implemented model

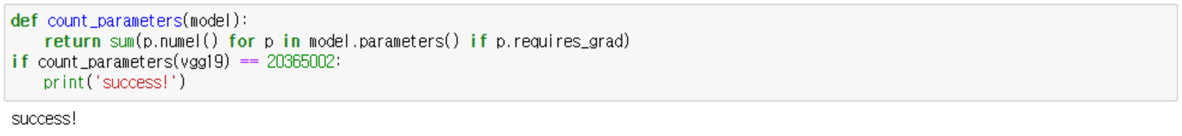


```
class ConvBlock1(nn.Module):

    def __init__(self, in_dim, out_dim):
        super(ConvBlock1, self).__init__()

        self.in_dim = in_dim
        self.out_dim = out_dim

        self.main = nn.Sequential(nn.Conv2d(self.in_dim, self.out_dim, kernel_size=3, padding=1),
                                  nn.BatchNorm2d(self.out_dim),
                                  nn.ReLU(),
                                  nn.Conv2d(self.out_dim, self.out_dim, kernel_size=3, padding=1),
                                  nn.BatchNorm2d(self.out_dim),
                                  nn.ReLU(),
                                  nn.MaxPool2d(2),
                                 )

    def forward(self, x):
        out = self.main(x)
        return out
```

```
class ConvBlock2(nn.Module):

    def __init__(self, in_dim, out_dim):
        super(ConvBlock2, self).__init__()

        self.in_dim = in_dim
        self.out_dim = out_dim

        self.main = nn.Sequential(nn.Conv2d(self.in_dim, self.out_dim, kernel_size=3, padding=1),
                                  nn.BatchNorm2d(self.out_dim),
                                  nn.ReLU(),
                                  nn.Conv2d(self.out_dim, self.out_dim, kernel_size=3, padding=1),
                                  nn.BatchNorm2d(self.out_dim),
                                  nn.ReLU(),
                                  nn.Conv2d(self.out_dim, self.out_dim, kernel_size=3, padding=1),
                                  nn.BatchNorm2d(self.out_dim),
                                  nn.ReLU(),
                                  nn.Conv2d(self.out_dim, self.out_dim, kernel_size=3, padding=1),
                                  nn.BatchNorm2d(self.out_dim),
                                  nn.ReLU(),
                                  nn.MaxPool2d(2),
                                 )

    def forward(self, x):
        out = self.main(x)
        return out
```

Figure 5: Model architecture of ConvBlocks of my VGG-19 model

- Number of the parameters



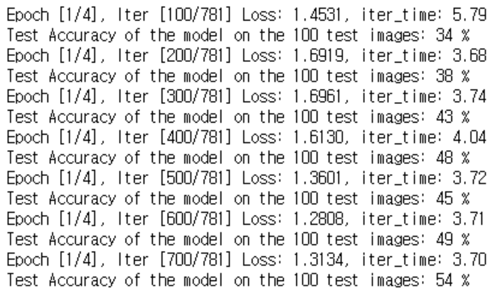
```
def count_parameters(model):
    return sum(p.numel() for p in model.parameters() if p.requires_grad)
if count_parameters(vgg19) == 20365002:
    print('success!')
```

```
success!
```

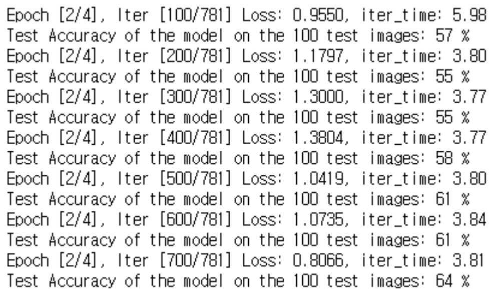
Figure 6: Counting the number of parameters of my VGG-19 model

2.2 Building classifier for CIFAR-10 dataset

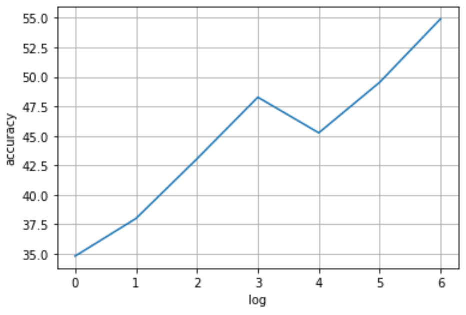
- Classification result for your model and the pre-trained model

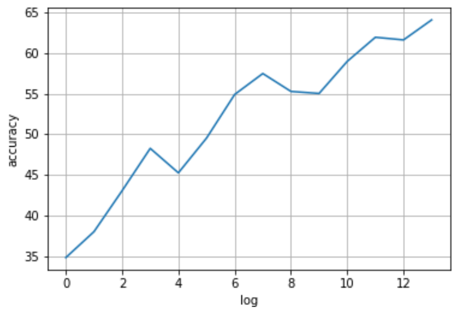


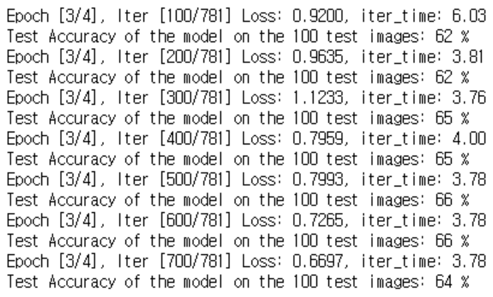
```
Epoch [1/4], Iter [100/781] Loss: 1.4531, iter_time: 5.79
Test Accuracy of the model on the 100 test images: 34 %
Epoch [1/4], Iter [200/781] Loss: 1.6919, iter_time: 3.68
Test Accuracy of the model on the 100 test images: 38 %
Epoch [1/4], Iter [300/781] Loss: 1.6961, iter_time: 3.74
Test Accuracy of the model on the 100 test images: 43 %
Epoch [1/4], Iter [400/781] Loss: 1.6130, iter_time: 4.04
Test Accuracy of the model on the 100 test images: 48 %
Epoch [1/4], Iter [500/781] Loss: 1.3601, iter_time: 3.72
Test Accuracy of the model on the 100 test images: 45 %
Epoch [1/4], Iter [600/781] Loss: 1.2808, iter_time: 3.71
Test Accuracy of the model on the 100 test images: 49 %
Epoch [1/4], Iter [700/781] Loss: 1.3134, iter_time: 3.70
Test Accuracy of the model on the 100 test images: 54 %
```



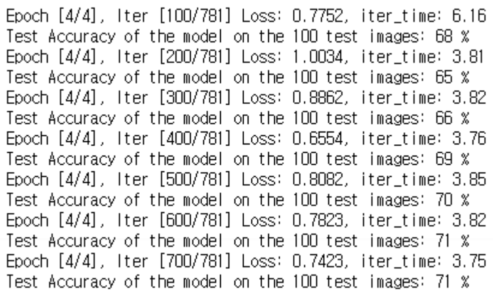
```
Epoch [2/4], Iter [100/781] Loss: 0.9550, iter_time: 5.98
Test Accuracy of the model on the 100 test images: 57 %
Epoch [2/4], Iter [200/781] Loss: 1.1797, iter_time: 3.80
Test Accuracy of the model on the 100 test images: 55 %
Epoch [2/4], Iter [300/781] Loss: 1.3000, iter_time: 3.77
Test Accuracy of the model on the 100 test images: 55 %
Epoch [2/4], Iter [400/781] Loss: 1.3804, iter_time: 3.77
Test Accuracy of the model on the 100 test images: 58 %
Epoch [2/4], Iter [500/781] Loss: 1.0419, iter_time: 3.80
Test Accuracy of the model on the 100 test images: 61 %
Epoch [2/4], Iter [600/781] Loss: 1.0735, iter_time: 3.84
Test Accuracy of the model on the 100 test images: 61 %
Epoch [2/4], Iter [700/781] Loss: 0.8066, iter_time: 3.81
Test Accuracy of the model on the 100 test images: 64 %
```



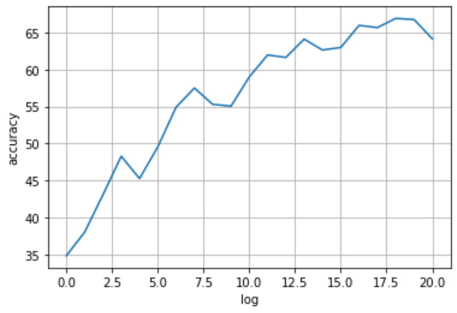




```
Epoch [3/4], Iter [100/781] Loss: 0.9200, iter_time: 6.03
Test Accuracy of the model on the 100 test images: 62 %
Epoch [3/4], Iter [200/781] Loss: 0.9635, iter_time: 3.81
Test Accuracy of the model on the 100 test images: 62 %
Epoch [3/4], Iter [300/781] Loss: 1.1233, iter_time: 3.76
Test Accuracy of the model on the 100 test images: 65 %
Epoch [3/4], Iter [400/781] Loss: 0.7959, iter_time: 4.00
Test Accuracy of the model on the 100 test images: 65 %
Epoch [3/4], Iter [500/781] Loss: 0.7993, iter_time: 3.78
Test Accuracy of the model on the 100 test images: 66 %
Epoch [3/4], Iter [600/781] Loss: 0.7265, iter_time: 3.78
Test Accuracy of the model on the 100 test images: 66 %
Epoch [3/4], Iter [700/781] Loss: 0.6697, iter_time: 3.78
Test Accuracy of the model on the 100 test images: 64 %
```



```
Epoch [4/4], Iter [100/781] Loss: 0.7752, iter_time: 6.16
Test Accuracy of the model on the 100 test images: 68 %
Epoch [4/4], Iter [200/781] Loss: 1.0034, iter_time: 3.81
Test Accuracy of the model on the 100 test images: 65 %
Epoch [4/4], Iter [300/781] Loss: 0.8862, iter_time: 3.82
Test Accuracy of the model on the 100 test images: 66 %
Epoch [4/4], Iter [400/781] Loss: 0.6554, iter_time: 3.76
Test Accuracy of the model on the 100 test images: 69 %
Epoch [4/4], Iter [500/781] Loss: 0.8082, iter_time: 3.85
Test Accuracy of the model on the 100 test images: 70 %
Epoch [4/4], Iter [600/781] Loss: 0.7823, iter_time: 3.82
Test Accuracy of the model on the 100 test images: 71 %
Epoch [4/4], Iter [700/781] Loss: 0.7423, iter_time: 3.75
Test Accuracy of the model on the 100 test images: 71 %
```



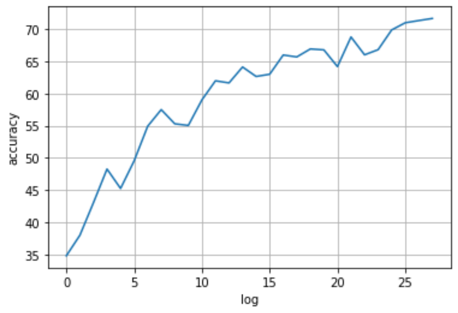
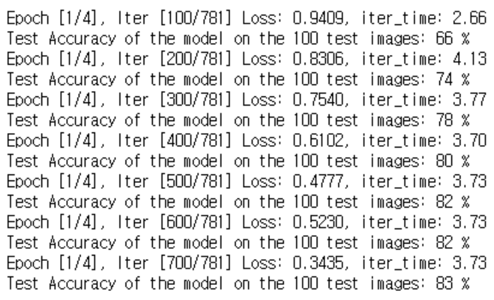
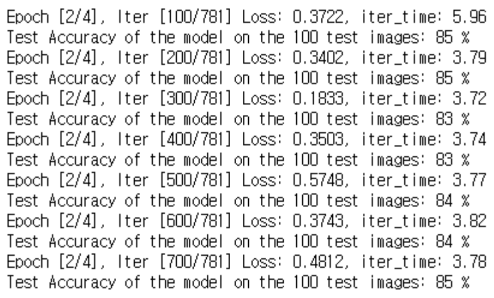


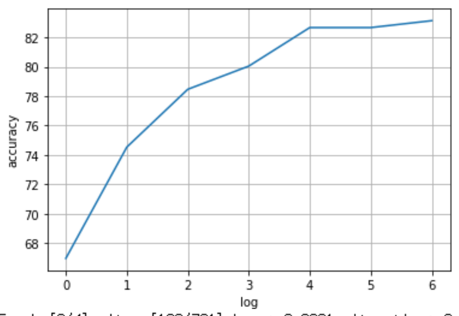
Figure 7: Classification result of my VGG-19 model

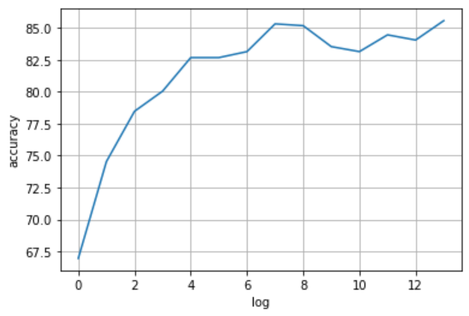


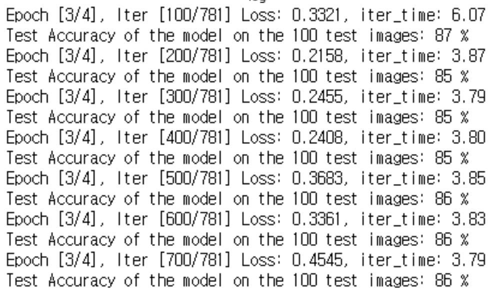
```
Epoch [1/4], Iter [100/781] Loss: 0.9409, iter_time: 2.66
Test Accuracy of the model on the 100 test images: 66 %
Epoch [1/4], Iter [200/781] Loss: 0.8306, iter_time: 4.13
Test Accuracy of the model on the 100 test images: 74 %
Epoch [1/4], Iter [300/781] Loss: 0.7540, iter_time: 3.77
Test Accuracy of the model on the 100 test images: 78 %
Epoch [1/4], Iter [400/781] Loss: 0.6102, iter_time: 3.70
Test Accuracy of the model on the 100 test images: 80 %
Epoch [1/4], Iter [500/781] Loss: 0.4777, iter_time: 3.73
Test Accuracy of the model on the 100 test images: 82 %
Epoch [1/4], Iter [600/781] Loss: 0.5230, iter_time: 3.73
Test Accuracy of the model on the 100 test images: 82 %
Epoch [1/4], Iter [700/781] Loss: 0.3435, iter_time: 3.73
Test Accuracy of the model on the 100 test images: 83 %
```



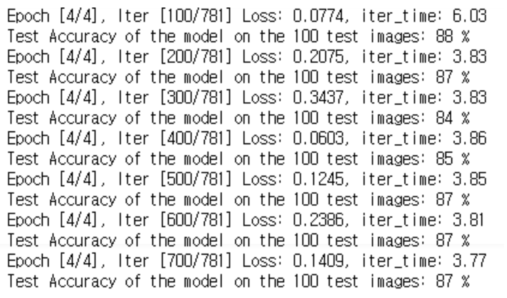
```
Epoch [2/4], Iter [100/781] Loss: 0.3722, iter_time: 5.96
Test Accuracy of the model on the 100 test images: 85 %
Epoch [2/4], Iter [200/781] Loss: 0.3402, iter_time: 3.79
Test Accuracy of the model on the 100 test images: 85 %
Epoch [2/4], Iter [300/781] Loss: 0.1833, iter_time: 3.72
Test Accuracy of the model on the 100 test images: 83 %
Epoch [2/4], Iter [400/781] Loss: 0.3503, iter_time: 3.74
Test Accuracy of the model on the 100 test images: 83 %
Epoch [2/4], Iter [500/781] Loss: 0.5748, iter_time: 3.77
Test Accuracy of the model on the 100 test images: 84 %
Epoch [2/4], Iter [600/781] Loss: 0.3743, iter_time: 3.82
Test Accuracy of the model on the 100 test images: 84 %
Epoch [2/4], Iter [700/781] Loss: 0.4812, iter_time: 3.78
Test Accuracy of the model on the 100 test images: 85 %
```



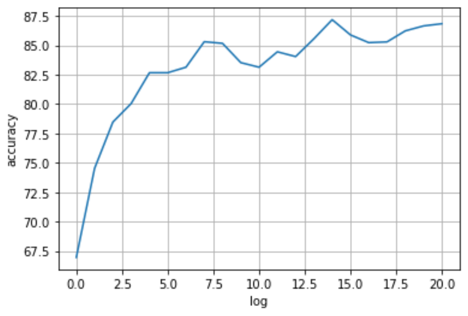




```
Epoch [3/4], Iter [100/781] Loss: 0.3321, iter_time: 6.07
Test Accuracy of the model on the 100 test images: 87 %
Epoch [3/4], Iter [200/781] Loss: 0.2158, iter_time: 3.87
Test Accuracy of the model on the 100 test images: 85 %
Epoch [3/4], Iter [300/781] Loss: 0.2455, iter_time: 3.79
Test Accuracy of the model on the 100 test images: 85 %
Epoch [3/4], Iter [400/781] Loss: 0.2408, iter_time: 3.80
Test Accuracy of the model on the 100 test images: 85 %
Epoch [3/4], Iter [500/781] Loss: 0.3683, iter_time: 3.85
Test Accuracy of the model on the 100 test images: 86 %
Epoch [3/4], Iter [600/781] Loss: 0.3361, iter_time: 3.83
Test Accuracy of the model on the 100 test images: 86 %
Epoch [3/4], Iter [700/781] Loss: 0.4545, iter_time: 3.79
Test Accuracy of the model on the 100 test images: 86 %
```



```
Epoch [4/4], Iter [100/781] Loss: 0.0774, iter_time: 6.03
Test Accuracy of the model on the 100 test images: 88 %
Epoch [4/4], Iter [200/781] Loss: 0.2075, iter_time: 3.83
Test Accuracy of the model on the 100 test images: 87 %
Epoch [4/4], Iter [300/781] Loss: 0.3437, iter_time: 3.83
Test Accuracy of the model on the 100 test images: 84 %
Epoch [4/4], Iter [400/781] Loss: 0.0603, iter_time: 3.86
Test Accuracy of the model on the 100 test images: 85 %
Epoch [4/4], Iter [500/781] Loss: 0.1245, iter_time: 3.85
Test Accuracy of the model on the 100 test images: 87 %
Epoch [4/4], Iter [600/781] Loss: 0.2386, iter_time: 3.81
Test Accuracy of the model on the 100 test images: 87 %
Epoch [4/4], Iter [700/781] Loss: 0.1409, iter_time: 3.77
Test Accuracy of the model on the 100 test images: 87 %
```



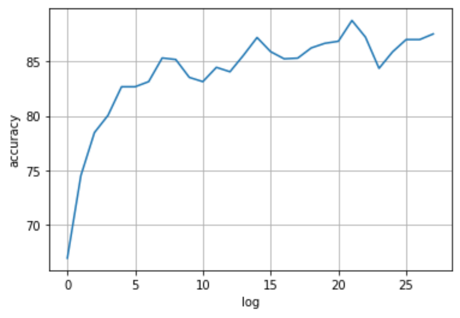
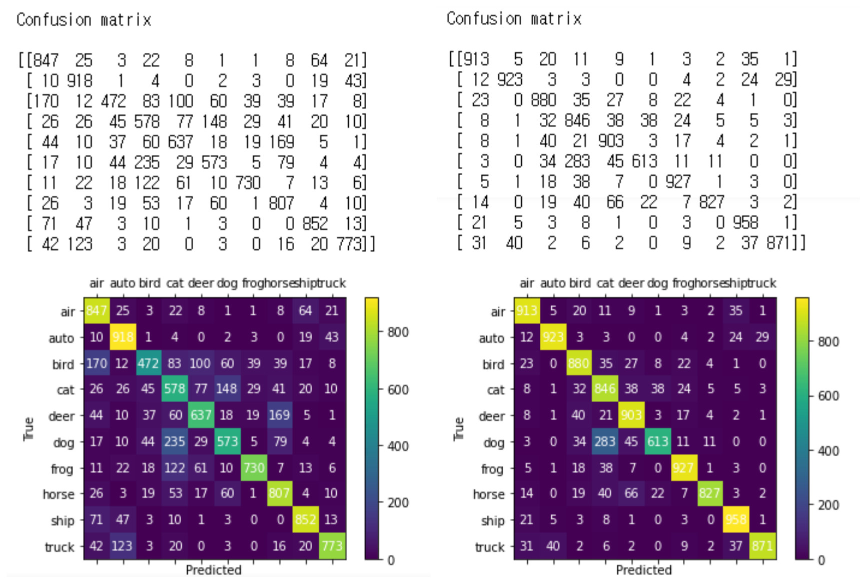


Figure 8: Classification result of the pre-trained model

- Confusion matrix and its analysis for each model



```
Confusion matrix

[[847  25   3  22   8   1   1   8  64  21]
 [ 10 918   1   4   0   2   3   0  19  43]
 [170  12 472  83 100  60  39  39  17   8]
 [ 26  26  45 578  77 148  29  41  20  10]
 [ 44  10  37  60 637  18  19 169   5   1]
 [ 17  10  44 235  29 573   5  79   4   4]
 [ 11  22  18 122  61  10 730   7  13   6]
 [ 26   3  19  53  17  60   1 807   4  10]
 [ 71  47   3  10   1   3   0   0 852  13]
 [ 42 123   3  20   0   3   0  16  20 773]]
```

```
Confusion matrix

[[913   5  20  11   9   1   3   2  35   1]
 [ 12 923   3   3   0   0   4   2  24  29]
 [ 23   0 880  35  27   8  22   4   1   0]
 [  8   1  32 846  38  38  24   5   5   3]
 [  8   1  40  21 903   3  17   4   2   1]
 [  3   0  34 283  45 613  11  11   0   0]
 [  5   1  18  38   7   0 927   1   3   0]
 [ 14   0  19  40  66  22   7 827   3   2]
 [ 21   5   3   8   1   0   3   0 958   1]
 [ 31  40   2   6   2   0   9   2  37 871]]
```

Figure 9: Comparison of confusion matrices (Left: my VGG-19, Right: pre-trained VGG-19)

When using the pre-trained model, our network is trained based on the network that has already been learned. Therefore, the learning speed to converge is faster and the accuracy is higher than that of leraning from scratch. In the classification result, the accuracy of my VGG-19 model increases slowly from the beginning, whereas the accuracy of the pre-trained model is more than 50% from the beginning and the result converges to some extent in only one epoch. Even in the confusion matrix, the pre-trained model showed better learning.