PA 2: Convolutional Neural Networks

20222421 GSAI SimJaeYoon

0. Overview

This project is about Convolutional Neural Networks(CNN). We will train and evaluate performance using several CNN networks and several image datasets that are important in the history of deep learning. In this assignment, we have to use MNIST dataset, CIFAR-10 dataset, and Pascal VOC 2007 dataset to verify our networks.

1. What you learned through this project

We were able to learn a lot of things in this assignment. We first learned about the overall CNN concept and learning procedure. We learned the structure and features of LeNet, AlexNet, ResNet, and ImageNet, which are representative networks of CNN. We were able to study more complex models, especially like yolo and ssd.

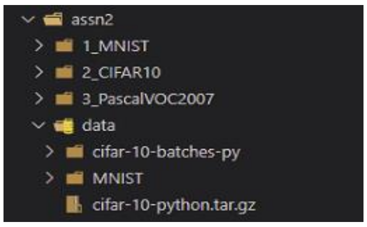
And we also learned about the overall process of implementing and learning these CNNs using the pytorch framework. We were able to learn the basics of how to use Pytorch, how to load data, and how to use a tensor. And we were able to learn various things such as network class configuration, activation function, auto-grad, automatic updating of gradients. We have determined and implemented optimizers and learned these overalls through GPU.

In the case of learning, it was also possible to learn how to improve the results through the parameter tuning process. We could learn that the speed or performance of learning changes according to the learning rate or epoch. After that, in order to check and compare the results, it was also possible to measure the accuracy and check the loss value through a graph.

While the entire process of implementing a CNN model and measuring its performance was a good experience, not all processes were smooth. In particular, in the case of models that are a little complicated to implement, such as yolo or ssd, the method of handling was not familiar, so it used a lot of time. However, based on these experiences, it will be of great help in future research and projects.

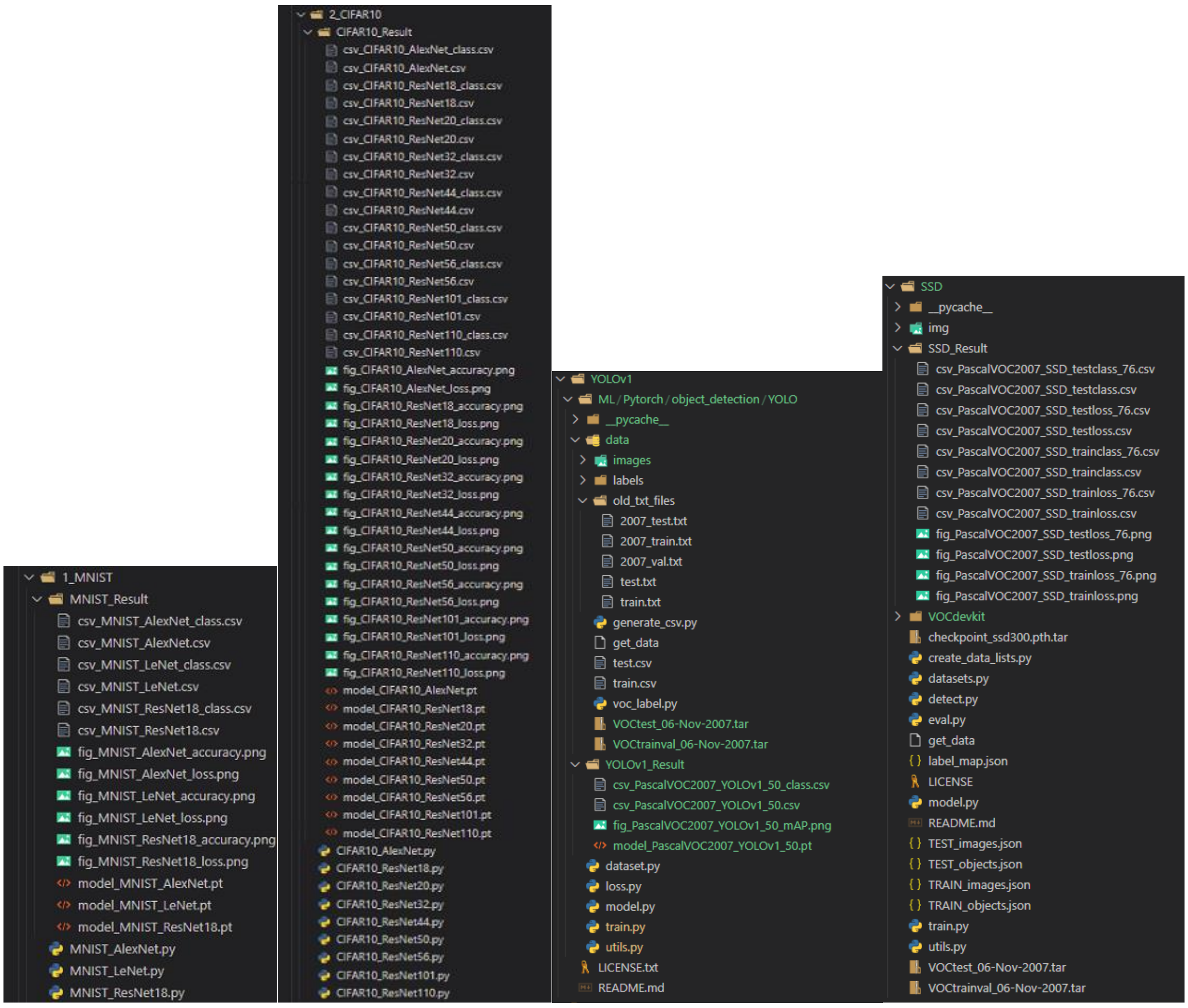
2. How to implement a simple neural network and convolutional neural network architecture and training algorithm. If you use your own code, you will get extra credits for it

The overall task was done in a lab server equipped with the NVIDIA RTX A6000 and cluster server with several GPUs. And according to the three dataset, the folders were organized and used as follows, and there are folders that organize the results obtained using each model separately in the middle.



```
assn2
  1_MNIST
  2_CIFAR10
  3_PascalVOC2007
  data
    cifar-10-batches-py
    MNIST
    cifar-10-python.tar.gz
```

Configure the entire folder



```
1_MNIST
  MNIST_Result
    csv_MNIST_AlexNet_class.csv
    csv_MNIST_AlexNet.csv
    csv_MNIST_LeNet_class.csv
    csv_MNIST_LeNet.csv
    csv_MNIST_ResNet18_class.csv
    csv_MNIST_ResNet18.csv
    fig_MNIST_AlexNet_accuracy.png
    fig_MNIST_AlexNet_loss.png
    fig_MNIST_LeNet_accuracy.png
    fig_MNIST_LeNet_loss.png
    fig_MNIST_ResNet18_accuracy.png
    fig_MNIST_ResNet18_loss.png
    model_MNIST_AlexNet.pt
    model_MNIST_LeNet.pt
    model_MNIST_ResNet18.pt
  MNIST_AlexNet.py
  MNIST_LeNet.py
  MNIST_ResNet18.py
```

```
2_CIFAR10
  CIFAR10_Result
    csv_CIFAR10_AlexNet_class.csv
    csv_CIFAR10_AlexNet.csv
    csv_CIFAR10_ResNet18_class.csv
    csv_CIFAR10_ResNet18.csv
    csv_CIFAR10_ResNet20_class.csv
    csv_CIFAR10_ResNet20.csv
    csv_CIFAR10_ResNet32_class.csv
    csv_CIFAR10_ResNet32.csv
    csv_CIFAR10_ResNet44_class.csv
    csv_CIFAR10_ResNet44.csv
    csv_CIFAR10_ResNet50_class.csv
    csv_CIFAR10_ResNet50.csv
    csv_CIFAR10_ResNet56_class.csv
    csv_CIFAR10_ResNet56.csv
    csv_CIFAR10_ResNet101_class.csv
    csv_CIFAR10_ResNet101.csv
    csv_CIFAR10_ResNet110_class.csv
    csv_CIFAR10_ResNet110.csv
    fig_CIFAR10_AlexNet_accuracy.png
    fig_CIFAR10_AlexNet_loss.png
    fig_CIFAR10_ResNet18_accuracy.png
    fig_CIFAR10_ResNet18_loss.png
    fig_CIFAR10_ResNet20_accuracy.png
    fig_CIFAR10_ResNet20_loss.png
    fig_CIFAR10_ResNet32_accuracy.png
    fig_CIFAR10_ResNet32_loss.png
    fig_CIFAR10_ResNet44_accuracy.png
    fig_CIFAR10_ResNet44_loss.png
    fig_CIFAR10_ResNet50_accuracy.png
    fig_CIFAR10_ResNet50_loss.png
    fig_CIFAR10_ResNet56_accuracy.png
    fig_CIFAR10_ResNet56_loss.png
    fig_CIFAR10_ResNet101_accuracy.png
    fig_CIFAR10_ResNet101_loss.png
    fig_CIFAR10_ResNet110_accuracy.png
    fig_CIFAR10_ResNet110_loss.png
    model_CIFAR10_AlexNet.pt
    model_CIFAR10_ResNet18.pt
    model_CIFAR10_ResNet20.pt
    model_CIFAR10_ResNet32.pt
    model_CIFAR10_ResNet44.pt
    model_CIFAR10_ResNet50.pt
    model_CIFAR10_ResNet56.pt
    model_CIFAR10_ResNet101.pt
    model_CIFAR10_ResNet110.pt
  CIFAR10_AlexNet.py
  CIFAR10_ResNet18.py
  CIFAR10_ResNet20.py
  CIFAR10_ResNet32.py
  CIFAR10_ResNet44.py
  CIFAR10_ResNet50.py
  CIFAR10_ResNet56.py
  CIFAR10_ResNet101.py
  CIFAR10_ResNet110.py
```

```
YOLOv1
  ML / Pytorch / object_detection / YOLO
    __pycache__
    data
      images
      labels
      old_txt_files
        2007_test.txt
        2007_train.txt
        2007_val.txt
        test.txt
        train.txt
      generate_csv.py
      get_data
      test.csv
      train.csv
      voc_label.py
      VOCtest_06-Nov-2007.tar
      VOCtrainval_06-Nov-2007.tar
    YOLOv1_Result
      csv_PascalVOC2007_YOLOv1_50_class.csv
      csv_PascalVOC2007_YOLOv1_50.csv
      fig_PascalVOC2007_YOLOv1_50_mAP.png
      model_PascalVOC2007_YOLOv1_50.pt
    dataset.py
    loss.py
    model.py
    train.py
    utils.py
  LICENSE.txt
  README.md
```

```
SSD
  __pycache__
  img
  SSD_Result
    csv_PascalVOC2007_SSD_testclass_76.csv
    csv_PascalVOC2007_SSD_testclass.csv
    csv_PascalVOC2007_SSD_testloss_76.csv
    csv_PascalVOC2007_SSD_testloss.csv
    csv_PascalVOC2007_SSD_trainclass_76.csv
    csv_PascalVOC2007_SSD_trainclass.csv
    csv_PascalVOC2007_SSD_trainloss_76.csv
    csv_PascalVOC2007_SSD_trainloss.csv
    fig_PascalVOC2007_SSD_testloss_76.png
    fig_PascalVOC2007_SSD_testloss.png
    fig_PascalVOC2007_SSD_trainloss_76.png
    fig_PascalVOC2007_SSD_trainloss.png
  VOCdevkit
  checkpoint_ssd300.pth.tar
  create_data_lists.py
  datasets.py
  detect.py
  eval.py
  get_data
  label_map.json
  LICENSE
  model.py
  README.md
  TEST_images.json
  TEST_objects.json
  TRAIN_images.json
  TRAIN_objects.json
  train.py
  utils.py
  VOCtest_06-Nov-2007.tar
  VOCtrainval_06-Nov-2007.tar
```

Configure each folder : (1)MNIST(Left) / (2)CIFAR10(Center) / (3, 4)PascalVOC2007(Right)

The neural network can be directly implemented, the model provided by pytorch can be imported and used, or the code implemented by another person can be used by github clone. The tendency to study using well-written codes as open sources has increased. Even so, in the case of a simple model, it may be directly implemented. And for convenience, there are many models that can be used as modules in pytorch itself. In the case of this task, it was conducted in various ways depending on the dataset used and the model architecture to be implemented.

For models using MNIST dataset, model architecture was directly implemented by referring to the official document pytorch.org. In the case of ResNet, there were many cases of pretraining, so the functions stored in torchvision.model were available. In the case of models without pretraining, it was implemented by referring to github and papers. In addition, the process of storing loss and acuity as a graph and storing the acuity for each class separately as a csv file was implemented by studying directly. In the case of the model parameter learned, it was also saved separately as a .pt file. The following is AlexNet, one of several direct network architectures. In this way, simple models are not difficult to implement directly.

```
# AlexNet architecture
class AlexNet(nn.Module):
    def __init__(self):
        super(AlexNet, self).__init__()
        self.conv1 = nn.Conv2d(3, 96, kernel_size = 5, stride = 1, padding = 2)
        self.conv2 = nn.Conv2d(96, 256, kernel_size = 5, stride = 1, padding = 2)
        self.conv3 = nn.Conv2d(256, 384, kernel_size = 3, stride = 1, padding = 1)
        self.conv4 = nn.Conv2d(384, 384, kernel_size = 3, stride = 1, padding = 1)
        self.conv5 = nn.Conv2d(384, 256, kernel_size = 3, stride = 1, padding = 1)
        self.fc1 = nn.Linear(3 * 3 * 256, 4096)
        self.dropout1 = nn.Dropout(0.5)
        self.fc2 = nn.Linear(4096, 10)
        self.dropout2 = nn.Dropout(0.5)

    def forward(self, x):
        x = self.conv1(x) # Layer(2)
        x = F.relu(x)    # Layer(3)
        x = local_contrast_normalization(x, radius = 5)    # Layer(4)
        x = F.max_pool2d(x, kernel_size = 3, stride = 2)    # Layer(5)
        x = self.conv2(x) # Layer(6)
        x = F.relu(x)    # Layer(7)
        x = local_contrast_normalization(x, radius = 5)    # Layer(8)
        x = F.max_pool2d(x, kernel_size = 3, stride = 2)    # Layer(9)
        x = self.conv3(x) # Layer(10)
        x = F.relu(x)    # Layer(11)
        x = self.conv4(x) # Layer(12)
        x = F.relu(x)    # Layer(13)
        x = self.conv5(x) # Layer(14)
        x = F.relu(x)    # Layer(15)
        x = F.max_pool2d(x, kernel_size = 3, stride = 2)    # Layer(16)
        x = x.view(-1, 3 * 3 * 256)
        x = self.fc1(x) # Layer(17)
        x = F.relu(x)    # Layer(18)
        x = self.dropout1(x)    # Layer(19)
        x = self.fc2(x)    # Layer(20)
        x = self.dropout2(x)    # Layer(21)
        output = F.softmax(x, dim = 1)    # Layer(22)
        return output
```

AlexNet using CIFAR10 dataset and several ResNet models were also implemented directly by referring to pytorch.org and GitHub. Similarly, models were called up using built-in functions, and graphs and result information were stored separately at the end. In addition, model parameters were saved as .pt file.
Finally, yolo and ssd model using Pascal VOC 2007 dataset did not need to be implemented directly from the beginning. The git clone was directly performed through the GitHub link in the assignment pdf and modified and used as desired. In the case of dataset, the existing MNIST or CIFAR10 had instructions to download in pytorch, but Pascal VOC 2007 dataset was downloaded directly using wget and pre-processed. In the process of modifying the code, the code was written separately to store information such as the result of train and test mAP or mAP for each class.

3. Hyper-parameters of each network architecture and its training method

In the case of Hyper-parameters, each network architecture was set in various ways, and the training method was also conducted differently depending on the network. First, we will explain the network architecture using MNIST dataset. For batch size, LeNet and ResNet using MNIST dataset were set to 64 and AlexNet was set to 128. Epoch set all three networks to 30. In the case of learning rate, all three networks were set to 0.005. For Momentum, LeNet was set to 0.8 and the rest to 0.9. In the case of weight decay, all three were fixed values of 0.0001, which was set to a very small value. In all three models, SGD optimizer was used to update parameters, and cross entropy loss was used for loss.

Second, we will look at AlexNet using CIFAR10 and eight types of ResNet. Of these, three ResNet were pretrained using ImageNet and then fine-tuned. Most network architectures used the same hyper-parameter. For batch size, 128 was set, and epoch used 50. For the learning rate of all ResNet except AlexNet, a large value of 0.1 was used, and the learning rate was reduced using the learning rate scheduler. AlexNet used a small value of 0.005 from the beginning. In the case of Momentum, AlexNet used 0.9 and the remaining ResNet used 0.8. The weight decay values were all 0.0001. This time, the parameter was updated using SGD optimizer, and cross entropy loss was used as the loss function.

In the case of Yolo model, it takes a lot of time to learn one epoch. The reason is that the amount of dataset is enormous and the amount of computation in the model is quite large. First, in the case of batch size, it was set to 16 and in the case of learning rate, it was set to 0.00002 very small. The weight decay value was set to 0, and in the case of epoch, learning was conducted by setting it from 50 to 1000. In the learning process, the results of bounding box and mean average precision could be obtained, and in particular, the mAP value for each class could be obtained in the process of obtaining mean average precision. In the case of SSD, the test was conducted using the model stored after learning. During training, the batch size was set to 8, the learning rate was set to 0.003, the momentum was set to 0.9, and the weight decay value was set to 0.0005. In order to achieve better performance, learning was conducted by modifying epoch and existing hyperparameters. As a result of subsequent learning, in the test process, the batch size was increased to 64, and then the mAP was immediately calculated.

4. History of efforts to improve the performance on your network model

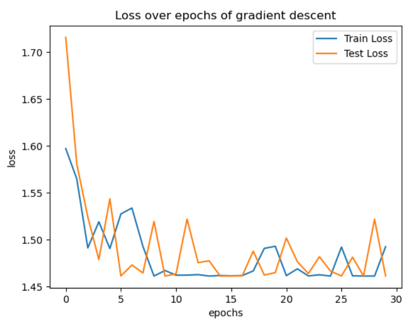
To improve the performance, hyperparameters such as learning rate and epochs were modified. Various combinations of hyperparameters were used to conduct as many experiments as possible. In the case of epoch, for models using the same dataset, the same epoch was used, and the appropriate epoch was determined by looking at the graph of the validation loss. In the case of the learning rate, a fixed small learning rate was used in some models, and in another model, the performance was improved by starting with 0.1 and gradually decreasing. In the case of the original learning rate, it is more effective to reduce it appropriately than to fix and use one value. And it was learning rate scheduler that made this possible, and it was actively utilized. In addition, several and multi GPUs were used to try various things.

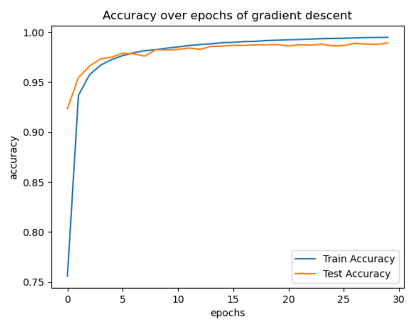
5. Learning curves for training and validation in one graph and the classification result

We would like to check the classification results and performance of several models for a total of three datasets. For each model, we want to check the loss graph, accuracy or mAP graph according to the epoch. For MNIST dataset, epoch was set to 30, and for CIFAR10 dataset, epoch was set to 50. Finally, for the Pascal VOC 2007 dataset, the epoch was set to a fluidly wide range.

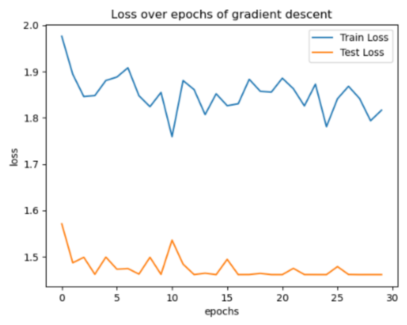
5.1. MNIST dataset

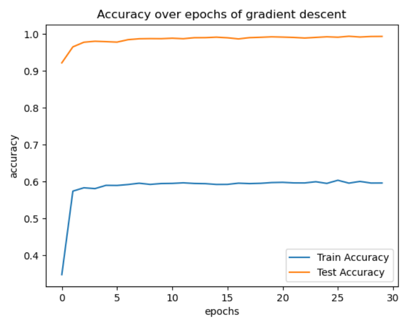
- LeNet



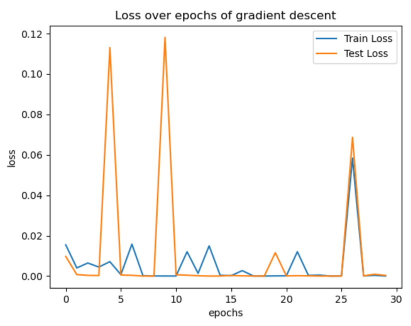


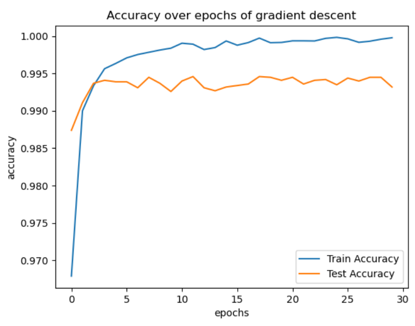
- AlexNet





- ResNet-18(Pretrained)



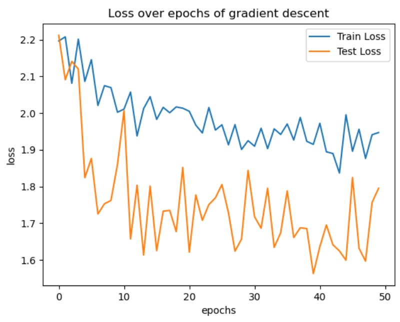


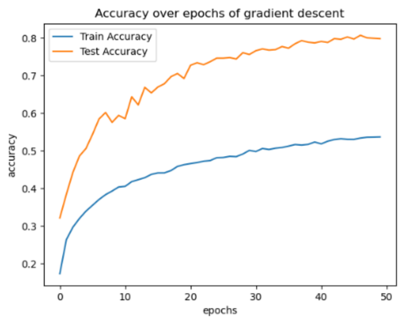
Accuracy Precision(%)		0	1	2	3	4	5	6	7	8	9	Total
LeNet	Train	99.74	99.70	99.34	99.26	99.55	99.42	99.79	99.53	99.31	99.10	99.48
	Test	99.38	99.64	98.83	99.10	98.69	98.87	98.32	98.34	98.56	98.41	98.92
AlexNet	Train	99.49	74.53	61.54	55.35	53.04	50.15	51.03	50.00	49.20	49.62	59.63
	Test	99.59	99.91	99.70	99.40	99.49	99.66	99.26	98.83	98.56	98.71	99.32
ResNet-18 (Pretrained)	Train	99.98	99.97	100.0	100.0	99.94	100.0	99.98	100.0	99.98	99.93	99.98
	Test	99.89	99.91	99.32	99.70	99.38	99.10	98.74	99.31	99.48	98.21	99.32

MNIST Performance

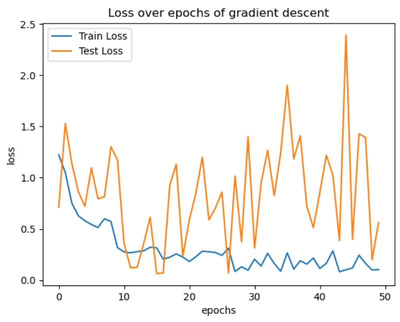
5.2. CIFAR-10 dataset

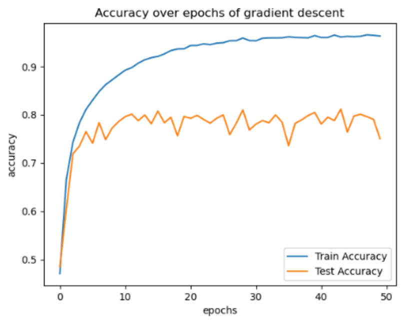
- AlexNet



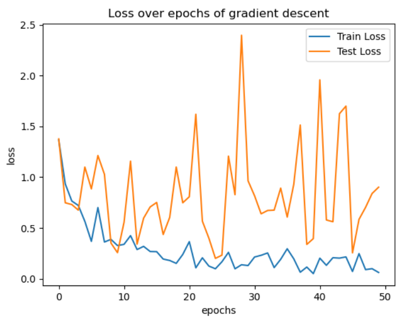


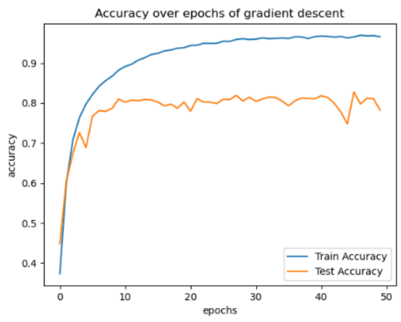
- ResNet-20



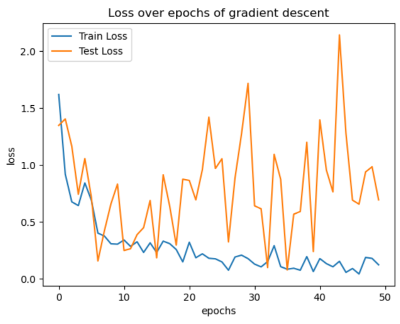


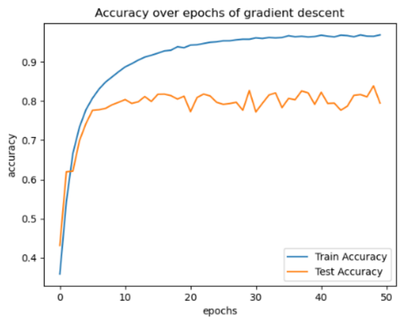
- ResNet-32



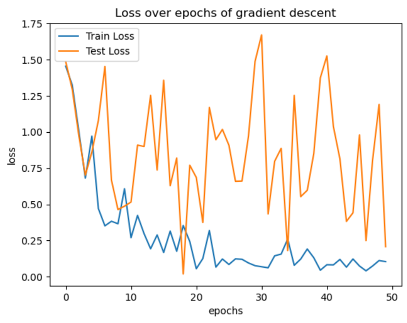


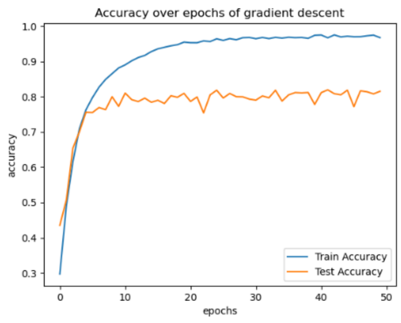
- ResNet-44



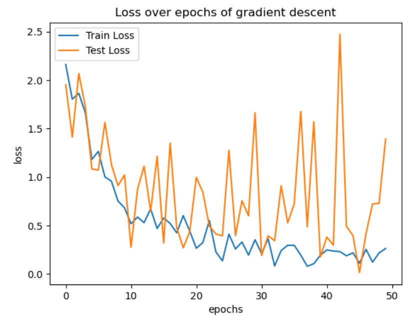


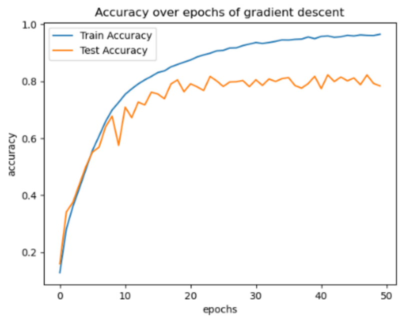
- ResNet-56



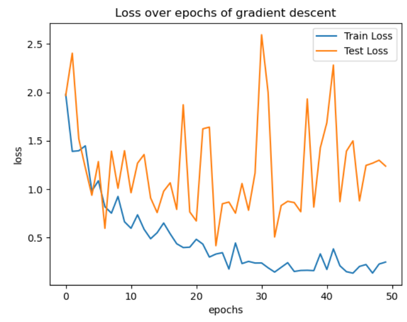


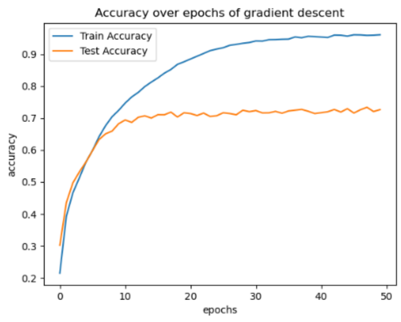
- ResNet-110



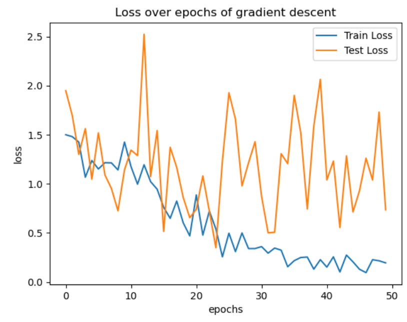


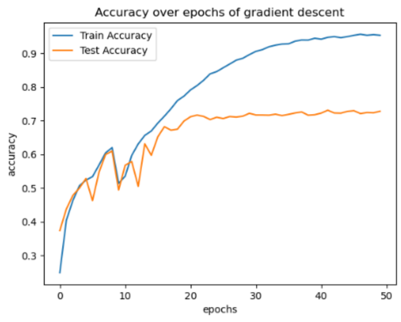
- ResNet-18(Pretrained)



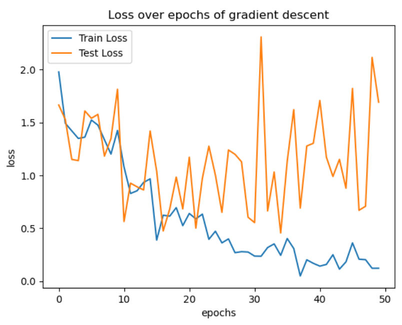


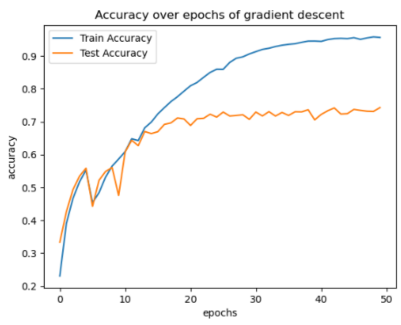
- ResNet-50(Pretrained)





- ResNet-101(Pretrained)



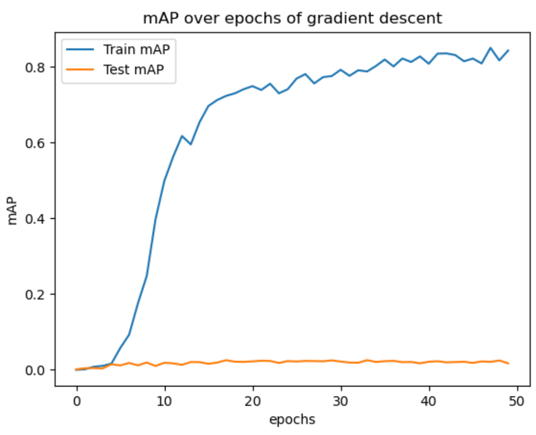


Accuracy Precision(%)		Airplane	Automobile	Bird	Cat	Deer	Dog	Frog	Horse	Ship	Truck	Total
AlexNet	Train	88.82	70.16	53.23	46.50	48.00	41.82	47.68	45.73	47.94	46.83	59.67
	Test	81.60	91.70	59.19	73.70	75.50	67.50	86.90	82.50	94.30	85.30	79.83
ResNet-20	Train	96.3	98.46	95.36	93.42	96.04	94.58	97.00	96.66	97.80	97.84	96.34
	Test	79.9	80.60	74.50	70.10	63.70	76.80	77.90	54.70	91.30	80.70	75.02
ResNet-32	Train	96.64	98.28	95.60	94.30	96.38	94.92	96.88	97.18	97.88	97.98	96.60
	Test	81.30	98.10	60.50	81.80	71.70	68.60	87.10	83.10	80.70	69.50	78.24
ResNet-44	Train	96.98	98.61	96.10	94.19	96.60	95.62	97.16	97.22	98.20	98.10	96.88
	Test	81.00	83.10	81.20	62.30	80.00	83.20	85.00	73.60	69.00	96.50	79.49
ResNet-56	Train	96.86	98.26	95.66	94.67	96.39	95.30	97.38	97.28	97.68	98.04	96.75
	Test	88.70	91.70	73.50	66.90	76.90	68.70	91.10	88.20	87.50	82.19	81.54
ResNet-110	Train	96.46	98.48	95.46	94.34	96.28	94.86	97.06	97.20	97.60	97.54	96.52
	Test	82.69	81.80	75.70	84.50	66.10	54.70	84.89	84.30	73.60	95.60	78.39
ResNet-18 (Pretrained)	Train	96.67	97.36	94.44	93.82	95.08	94.80	96.74	96.74	97.86	97.14	96.06
	Test	79.70	83.89	61.10	51.20	73.30	64.30	79.60	79.10	78.40	75.80	72.64
ResNet-50 (Pretrained)	Train	95.84	96.94	93.58	93.00	94.32	94.22	96.28	96.32	96.58	96.14	95.32
	Test	74.90	83.20	67.50	56.80	65.70	62.20	79.80	71.10	82.10	84.50	72.78
ResNet-101 (Pretrained)	Train	96.00	97.00	94.74	92.92	95.08	93.97	96.50	96.76	96.92	96.22	95.61
	Test	76.60	90.90	71.39	61.90	71.30	50.70	82.280	76.40	81.69	79.20	74.29

CIFAR-10 Performance

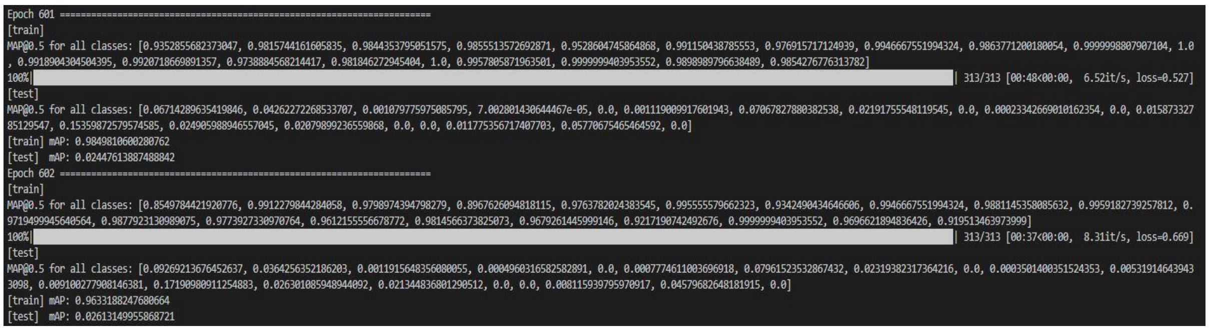
5.3. Pascal VOC 2007

- YOLOv1



YOLOv1 mAP result with 50 epochs

In the case of test mAP, performance was getting better at 600 epochs than at 50 epochs, and better performance would have been recorded if it had reached 1000 epochs. Afterwards, the overfitting results can be improved, but the assignment deadline is insufficient, which will remain an improvement in the future. After turning 50 epochs, I watched the progress by turning 1000 epochs. The results were definitely improving, but the server was shut down at 600 epochs and we didn't have any more time to improve the results further. So, here's the result of 600 epoch performance improvement over 50 epochs.



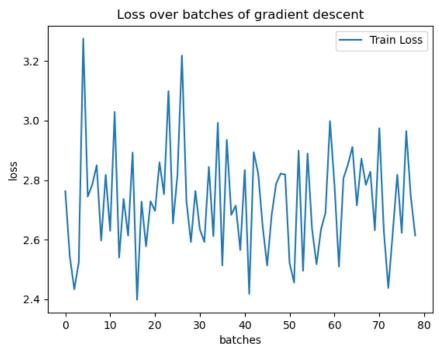
```
Epoch 601 ========================================================================
[train]
MAP@0.5 for all classes: [0.9352855682373047, 0.9815744161605835, 0.9844353795051575, 0.9855513572692871, 0.9528604745864868, 0.991150438785553, 0.976915717124939, 0.9946667551994324, 0.9863771200180054, 0.9999998807907104, 1.0
, 0.9918904304504395, 0.9920718669891357, 0.9738884568214417, 0.981846272945404, 1.0, 0.9957805871963501, 0.9999999403953552, 0.9898989796638489, 0.9854276776313782]
100%|██████████████████████████████████████████████████████████████████████████████| 313/313 [00:48<00:00,  6.52it/s, loss=0.527]
[test]
MAP@0.5 for all classes: [0.06714289635419846, 0.04262272268533707, 0.001079775975085795, 7.002801430644467e-05, 0.0, 0.001119009917601943, 0.07067827880382538, 0.02191755548119545, 0.0, 0.00023342669010162354, 0.0, 0.015873327
85129547, 0.15359872579574585, 0.02490598894655704 5, 0.02079899236559868, 0.0, 0.0, 0.011775356717407703, 0.05770675465464592, 0.0]
[train] mAP: 0.9849810600280762
[test]  mAP: 0.02447613887488842
Epoch 602 ========================================================================
[train]
MAP@0.5 for all classes: [0.8549784421920776, 0.9912279844284058, 0.9798974394798279, 0.896762609481815, 0.9763782024383545, 0.995555579662323, 0.9342490434646606, 0.9946667551994324, 0.9881145358085632, 0.9959182739257812, 0.
9719499945640564, 0.9877923130989075, 0.9773927330970764, 0.9612155556678772, 0.9814566373825073, 0.9679261445999146, 0.9217190742492676, 0.9999999403953552, 0.9696621894836426, 0.919513463973999]
100%|██████████████████████████████████████████████████████████████████████████████| 313/313 [00:37<00:00,  8.31it/s, loss=0.669]
[test]
MAP@0.5 for all classes: [0.09269213676452637, 0.0364256352186203, 0.0011915648356080055, 0.0004960316582582891, 0.0, 0.0007774611003696918, 0.07961523532867432, 0.02319382317364216, 0.0, 0.0003501400351524353, 0.00531914643943
3098, 0.009100277908146381, 0.17190980911254883, 0.026301085948944092, 0.021344836801290512, 0.0, 0.0, 0.008115939795970917, 0.04579682648181915, 0.0]
[train] mAP: 0.9633188247680664
[test]  mAP: 0.02613149955868721
```

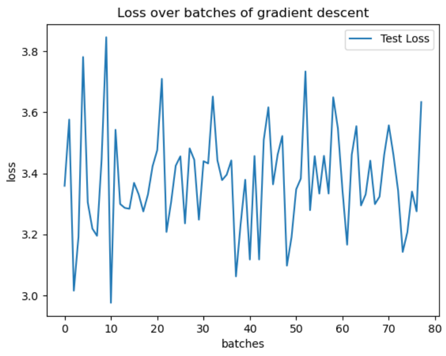
Average Precision(%)	Airplane	Bicycle	Bird	Boat	Bottle	Bus	Car	Cat	Chair	Cow	
Train	85.49	99.12	97.98	89.67	97.63	99.55	93.42	99.46	98.81	99.59	
Test	9.26	3.64	1.19	0.04	0.00	0.07	7.96	2.31	0.00	0.03	
Average Precision(%)	Dining Table	Dog	Horse	Motorbike	Person	Potted Plant	Sheep	Sofa	Train	TV / Monitor	Total (mAP)
Train	97.19	98.77	97.73	96.12	98.14	96.79	92.17	99.99	96.96	91.95	96.33
Test	0.53	0.91	17.19	2.63	2.13	0.00	0.00	0.81	0.45	0.00	2.61

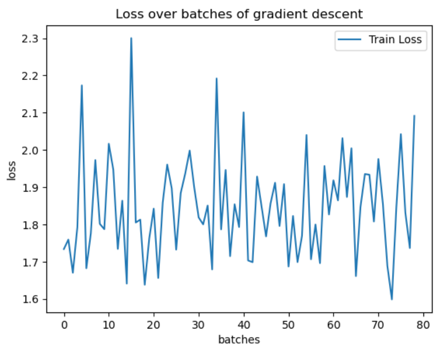
Pascal VOC 2007 Performance (YOLOv1)

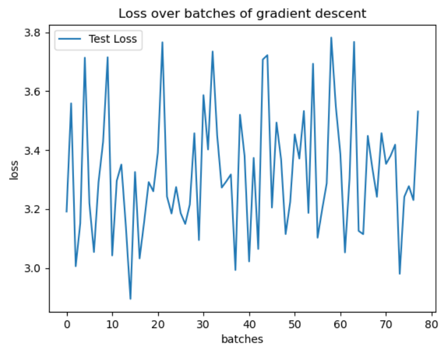
- SSD

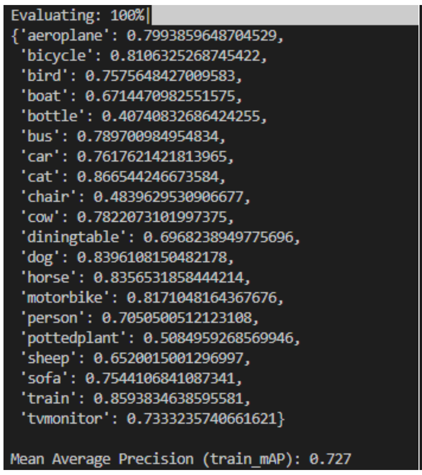
In the case of SSD models, it was confirmed that there was a clear difference in performance according to epoch. After proceeding with the learning as much as desired, the model parameter was saved to check its performance for train dataset and test dataset. First of all, when about 70 epochs were learned, some good results could be confirmed. Certainly, it is a significantly better result than when you learn a little.





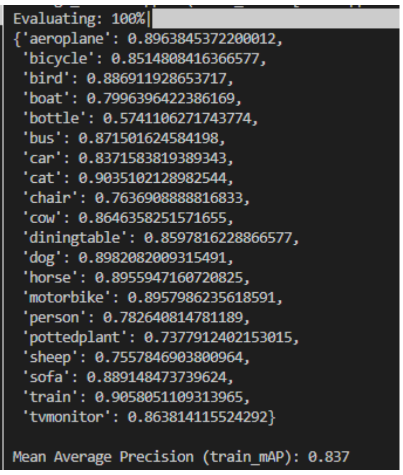






```
Evaluating: 100%|
{'aeroplane': 0.7993859648704529,
 'bicycle': 0.8106325268745422,
 'bird': 0.7575648427009583,
 'boat': 0.6714470982551575,
 'bottle': 0.40740832686424255,
 'bus': 0.789700984954834,
 'car': 0.7617621421813965,
 'cat': 0.866544246673584,
 'chair': 0.4839629530906677,
 'cow': 0.7822073101997375,
 'diningtable': 0.6968238949775696,
 'dog': 0.8396108150482178,
 'horse': 0.8356531858444214,
 'motorbike': 0.8171048164367676,
 'person': 0.7050500512123108,
 'pottedplant': 0.5084959268569946,
 'sheep': 0.6520015001296997,
 'sofa': 0.7544106841087341,
 'train': 0.8593834638595581,
 'tvmonitor': 0.7333235740661621}

Mean Average Precision (train_mAP): 0.727
```

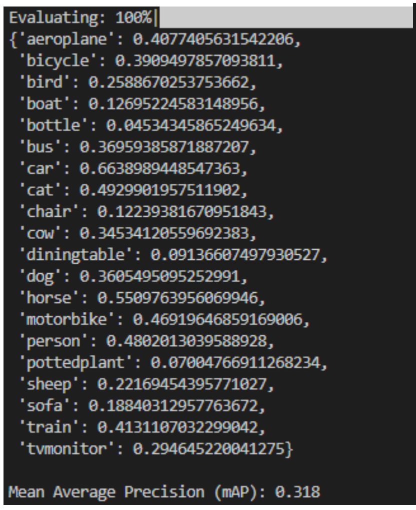


```
Evaluating: 100%|
{'aeroplane': 0.8963845372200012,
 'bicycle': 0.8514808416366577,
 'bird': 0.886911928653717,
 'boat': 0.7996396422386169,
 'bottle': 0.5741106271743774,
 'bus': 0.871501624584198,
 'car': 0.8371583819389343,
 'cat': 0.9035102128982544,
 'chair': 0.7636908888816833,
 'cow': 0.8646358251571655,
 'diningtable': 0.8597816228866577,
 'dog': 0.8982082009315491,
 'horse': 0.8955947160720825,
 'motorbike': 0.8957986235618591,
 'person': 0.782640814781189,
 'pottedplant': 0.7377912402153015,
 'sheep': 0.7557846903800964,
 'sofa': 0.889148473739624,
 'train': 0.9058051109313965,
 'tvmonitor': 0.863814115524292}

Mean Average Precision (train_mAP): 0.837
```

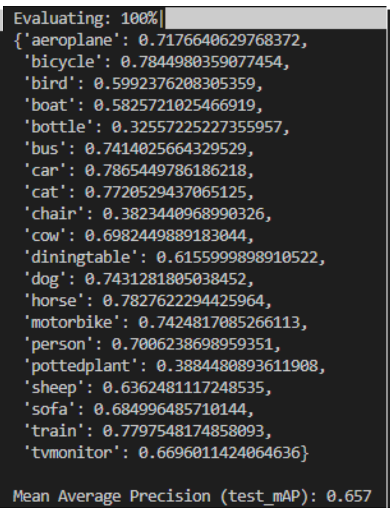
Train Evaluation with enough amount (ex. 76 epochs) of learning (left)/

Train Evaluation with much enough amount (ex. 383 epochs) of learning (right)



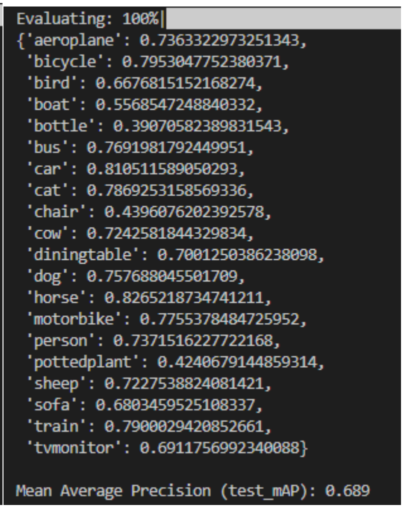
```
Evaluating: 100%|
{'aeroplane': 0.4077405631542206,
 'bicycle': 0.3909497857093811,
 'bird': 0.2588670253753662,
 'boat': 0.12695224583148956,
 'bottle': 0.04534345865249634,
 'bus': 0.36959385871887207,
 'car': 0.6638989448547363,
 'cat': 0.4929901957511902,
 'chair': 0.12239381670951843,
 'cow': 0.34534120559692383,
 'diningtable': 0.09136607497930527,
 'dog': 0.3605495095252991,
 'horse': 0.5509763956069946,
 'motorbike': 0.46919646859169006,
 'person': 0.4802013039588928,
 'pottedplant': 0.07004766911268234,
 'sheep': 0.22169454395771027,
 'sofa': 0.18840312957763672,
 'train': 0.4131107032299042,
 'tvmonitor': 0.294645220041275}

Mean Average Precision (mAP): 0.318
```



```
Evaluating: 100%|
{'aeroplane': 0.7176640629768372,
 'bicycle': 0.7844980359077454,
 'bird': 0.5992376208305359,
 'boat': 0.5825721025466919,
 'bottle': 0.32557225227355957,
 'bus': 0.7414025664329529,
 'car': 0.7865449786186218,
 'cat': 0.7720529437065125,
 'chair': 0.3823440968990326,
 'cow': 0.6982449889183044,
 'diningtable': 0.6155999898910522,
 'dog': 0.7431281805038452,
 'horse': 0.7827622294425964,
 'motorbike': 0.7424817085266113,
 'person': 0.7006238698959351,
 'pottedplant': 0.3884480893611908,
 'sheep': 0.636248111724853 5,
 'sofa': 0.684996485710144,
 'train': 0.7797548174858093,
 'tvmonitor': 0.6696011424064636}

Mean Average Precision (test_mAP): 0.657
```



```
Evaluating: 100%|
{'aeroplane': 0.7363322973251343,
 'bicycle': 0.7953047752380371,
 'bird': 0.6676815152168274,
 'boat': 0.5568547248840332,
 'bottle': 0.39070582389831543,
 'bus': 0.7691981792449951,
 'car': 0.810511589050293,
 'cat': 0.786925315856933 6,
 'chair': 0.4396076202392578,
 'cow': 0.7242581844329834,
 'diningtable': 0.7001250386238098,
 'dog': 0.757688045501709,
 'horse': 0.8265218734741211,
 'motorbike': 0.7755378484725952,
 'person': 0.7371516227722168,
 'pottedplant': 0.4240679144859314,
 'sheep': 0.7227538824081421,
 'sofa': 0.6803459525108337,
 'train': 0.7900029420852661,
 'tvmonitor': 0.6911756992340088}

Mean Average Precision (test_mAP): 0.689
```

Test Evaluation with a small amount (ex. 10 epochs) of learning (Left) /

Test Evaluation with enough amount (ex. 76 epochs) of learning (center) /

Test Evaluation with much enough amount (ex. 383 epochs) of learning (right)

Average Precision(%)	**Airplane**	**Bicycle**	**Bird**	**Boat**	**Bottle**	**Bus**	**Car**	**Cat**	**Chair**	**Cow**	
Train	79.93	81.06	75.75	67.14	40.74	78.97	76.17	86.65	48.39	78.22	
Test	71.76	78.44	59.92	58.25	32.55	74.14	78.65	77.20	38.23	69.82	
Average Precision(%)	**Dining Table**	**Dog**	**Horse**	**Motorbike**	**Person**	**Potted Plant**	**Sheep**	**Sofa**	**Train**	**TV / Monitor**	**Total (mAP)**
Train	69.68	83.96	83.56	81.71	70.50	50.84	65.20	75.44	85.93	73.33	72.70
Test	61.55	74.31	78.27	74.24	70.06	38.84	63.62	68.49	77.97	66.96	65.70

Pascal VOC 2007 Performance 1 (SSD)

Average Precision(%)	**Airplane**	**Bicycle**	**Bird**	**Boat**	**Bottle**	**Bus**	**Car**	**Cat**	**Chair**	**Cow**	
Train	89.63	85.14	88.69	79.96	57.41	87.15	83.71	90.35	76.36	86.46	
Test	73.63	79.53	66.76	55.68	39.07	76.91	81.05	78.69	43.96	72.42	
Average Precision(%)	**Dining Table**	**Dog**	**Horse**	**Motorbike**	**Person**	**Potted Plant**	**Sheep**	**Sofa**	**Train**	**TV / Monitor**	**Total (mAP)**
Train	85.97	89.82	89.55	89.57	78.26	73.77	75.57	88.91	90.58	86.38	83.70
Test	70.01	75.76	82.65	77.55	73.71	42.40	72.27	68.03	79.00	69.11	68.90

Pascal VOC 2007 Performance 2 (SSD)