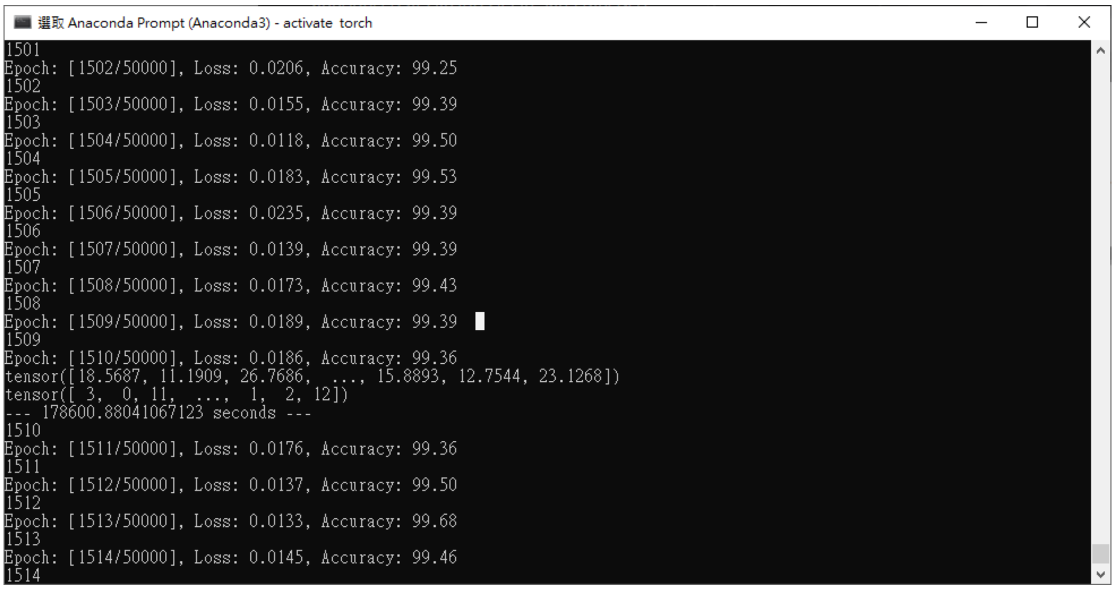
Selected Topics in Visual Recognition using Deep Learning
Homework 1 Assignment Report
STID: 0856619 秘子尉

Link of Code:

You can find final version of code at:
https://github.com/ndhu410421304/VRDL2019FALL_Image-Recognition/blob/master/Code%20and%20files/CS_IOC5008_0856619_HW1_program-FinalVersion.py
And well documented older version at:
https://github.com/ndhu410421304/VRDL2019FALL_Image-Recognition/blob/master/Code%20and%20files/old_version_code/CS_IOC5008_0856619_HW1_program-OldVersion.py
(History contains other older code versions, but some not well documented and formatted.)

Brief Introduction:

In order to use powerful library of deep learning, I used pytorch as the library. The final version of my code includes data augmentation for each batch before used for training in model, includes random resize clip, random horizontal flip, and try other augmentations as well. And I used pretrain model of DenseNet with a few adjustments in classifier as my highest result (96%), after I tried multiple powerful pretrain models like GoogleNet, ResNext, etc.

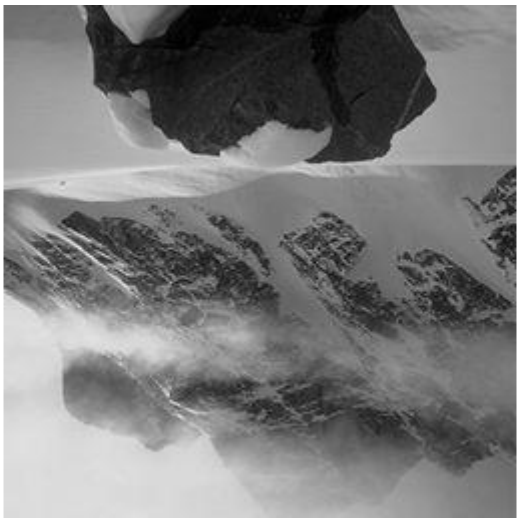


```
1501
Epoch: [1502/50000], Loss: 0.0206, Accuracy: 99.25
1502
Epoch: [1503/50000], Loss: 0.0155, Accuracy: 99.39
1503
Epoch: [1504/50000], Loss: 0.0118, Accuracy: 99.50
1504
Epoch: [1505/50000], Loss: 0.0183, Accuracy: 99.53
1505
Epoch: [1506/50000], Loss: 0.0235, Accuracy: 99.39
1506
Epoch: [1507/50000], Loss: 0.0139, Accuracy: 99.39
1507
Epoch: [1508/50000], Loss: 0.0173, Accuracy: 99.43
1508
Epoch: [1509/50000], Loss: 0.0189, Accuracy: 99.39
1509
Epoch: [1510/50000], Loss: 0.0186, Accuracy: 99.36
tensor([18.5687, 11.1909, 26.7686,  ..., 15.8893, 12.7544, 23.1268])
tensor([ 3,  0, 11,  ...,  1,  2, 12])
--- 178600.88041067123 seconds ---
1510
Epoch: [1511/50000], Loss: 0.0176, Accuracy: 99.36
1511
Epoch: [1512/50000], Loss: 0.0137, Accuracy: 99.50
1512
Epoch: [1513/50000], Loss: 0.0133, Accuracy: 99.68
1513
Epoch: [1514/50000], Loss: 0.0145, Accuracy: 99.46
1514
```

▲ Though densenet gets good result, it takes a lots of time to run.

Methodlogy:

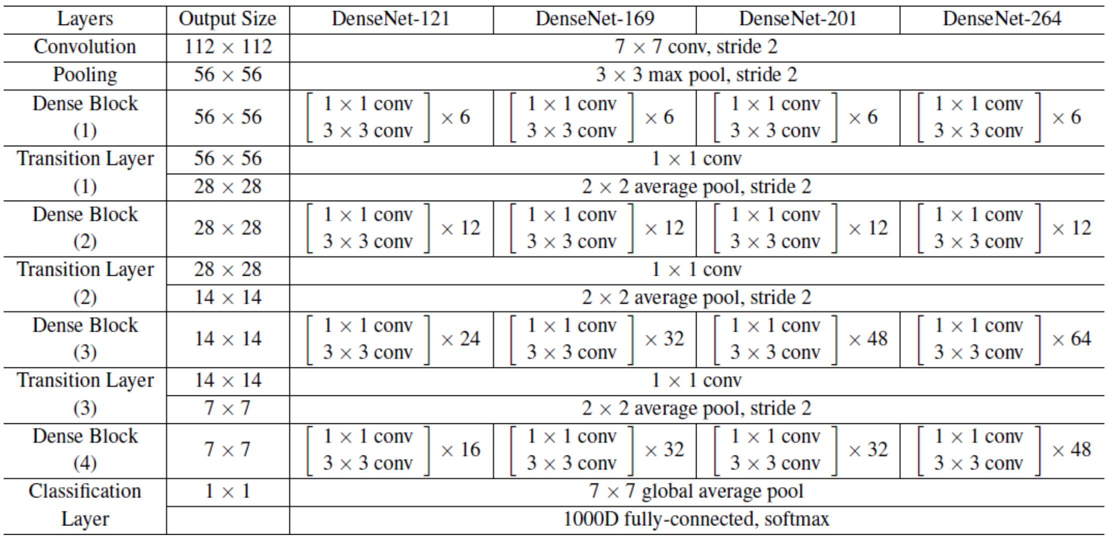
My main methodlogy of this assignment is to "try more possibilities", so I had tried out use traditional approach (machine learning methods) in first few days, then try to stack up convolutional network structure similar to Alexnet, and then when I found that I can use pretrained model, I start to tryout those models which had trained by Imagenet. The result is significant even without any change to them, then I try to think if there is any way to further improve the accuracy. Then I come up with data augmentation, which can give me more training datas from small inputs.



▲ Checking training data may also have some surprise, for example test image no 0778 is an upside-down image.

The structure of my code is as following: first we use transform.compose to define the augmentations we want to deal with the input data. Then we load these data by dataloader, so the training / testing data would be separate to small batches, so gpu only needs to deal with small amount of data, which code prevent OOM (out of memory) issue. Have a loop to run same time as epoch count, which in each epoch we would load in the training data by batches. We print out the loss and local accuracy information for each epoch, output current prediction for few epochs, and output final model after last epoch. For detailed description, please check the comment in code file. There is also detailed commented code for older version.

The main network part of my network is to use pretrained DenseNet which had trained on Imagenet for transfer learning. Densenet, different from ResNet, which have connected layer with each other layer, benefit from reusing same resources, and can reduce the amount of parameter. The version I had used is DenseNet-201, and I had modified the final layer of classification for our prediction task.



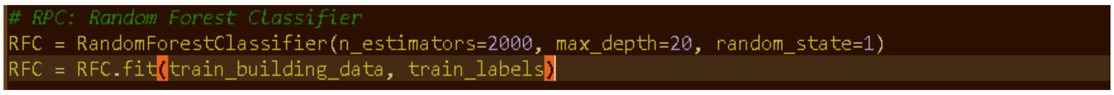
Layers	Output Size	DenseNet-121	DenseNet-169	DenseNet-201	DenseNet-264
Convolution	112×112	7×7 conv, stride 2			
Pooling	56×56	3×3 max pool, stride 2			
Dense Block (1)	56×56	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 6$
Transition Layer (1)	56×56	1×1 conv			
	28×28	2×2 average pool, stride 2			
Dense Block (2)	28×28	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 12$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 12$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 12$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 12$
Transition Layer (2)	28×28	1×1 conv			
	14×14	2×2 average pool, stride 2			
Dense Block (3)	14×14	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 24$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 32$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 48$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 64$
Transition Layer (3)	14×14	1×1 conv			
	7×7	2×2 average pool, stride 2			
Dense Block (4)	7×7	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 16$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 32$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 32$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 48$
Classification Layer	1×1	7×7 global average pool			
		1000D fully-connected, softmax			

▲ Spec of the DenseNet for ImageNet be like. I had used DenseNet-201 one.

Phase1 - Machine Learning Method:

Day 1

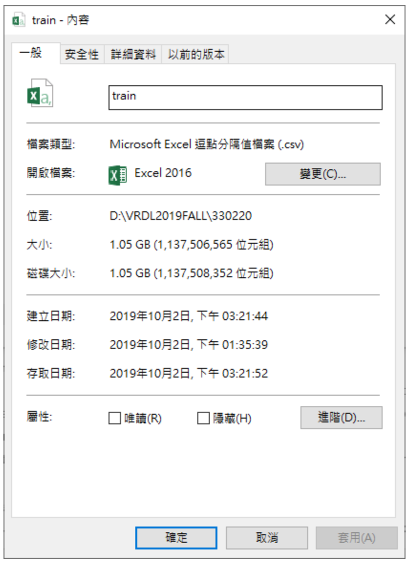
First, I had tried the traditional way to get familiar with dataset. I had tried Random Forest Classifier, Two kinds of SVM Classifier, and MLP Classifier. Unfortunately, since the data amount and varieties are not that much, the accuracy we get by using traditional method is not good enough, around 40 % for Random Forest Classifier at best.



```
# RPC: Random Forest Classifier
RFC = RandomForestClassifier(n_estimators=2000, max_depth=20, random_state=1)
RFC = RFC.fit(train_building_data, train_labels)
```

▲Phase1 main part of code be like. Using library sklearn for some machine learning method, and get best accuracy around 40%.

Phase2 – Build my own Convolutional network Structure:



train

檔案類型:	Microsoft Excel 逗點分隔值檔案 (.csv)
開啟檔案:	Excel 2016
位置:	D:\VRDL2019FALL\330220
大小:	1.05 GB (1,137,506,565 位元組)
磁碟大小:	1.05 GB (1,137,508,352 位元組)
建立日期:	2019年10月2日, 下午 03:21:44
修改日期:	2019年10月2日, 下午 01:35:39
存取日期:	2019年10月2日, 下午 03:21:52
屬性:	☐ 唯讀(R) ☐ 隱藏(H)

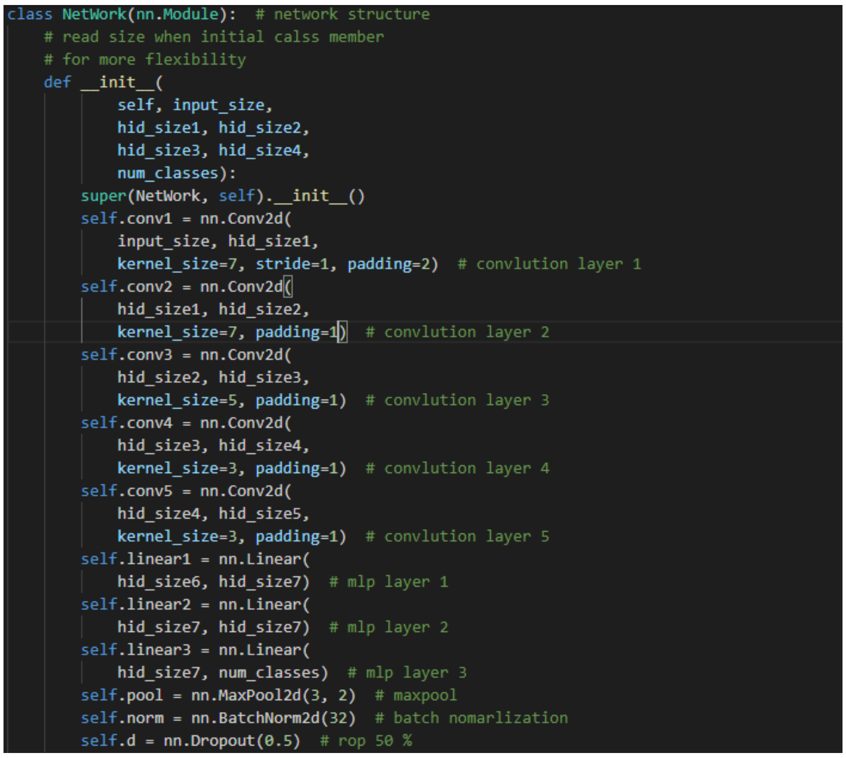
▲ Originallly transform images into csv files before input. For a better solution, I had resized them to 320 * 320, and the size of csv file become very big.

Day 2 – 4

Then I realize I need a more powerful way to build a classifier, so I start to build up a convolutional structure by myself. The structure I had use is like famous AlexNet structure: 5 convolutional layers and 3 full connected layers (check older version code). After finish the structure, I need to deal with input. Since the convolutional network, unlike traditional machine learning method, may use a lots of memory space, so if the image is too large with too many images, the memory may not be enough to support for even 1 epoch. And also, I had found the original way I use for reading image one by one had taken too much time for each try, so I had written a program to first transform the image into square and rescale, then output as csv file. Though it still took lots of time for input if I resized them too big, says 256 * 256, the csv file of training data may take like 1GB storage, and the working efficiency isn't high enough: it may take like few hours to even finish 10 epochs. But since I had found that higher image resolution may leads to better result (try from 46 * 46 to 256 * 256), I still need to use the higher resolution ones. Due to that, I would need to use mini batch method to save memory space in order the neural network could work, and also do more dropout, increase kernel size and stride, remove padding or half the hidden layer size to increase the working efficiency, then finally reach the baseline: at best accuracy 60. The hyperparameter of hidden layers size here I would like to build the structure in a triangle shape: from start would be ascending order, with each layer of 2's exponentiation of size, then in full connected layer would be descending order, finally output with size of same class.

Day 5-6

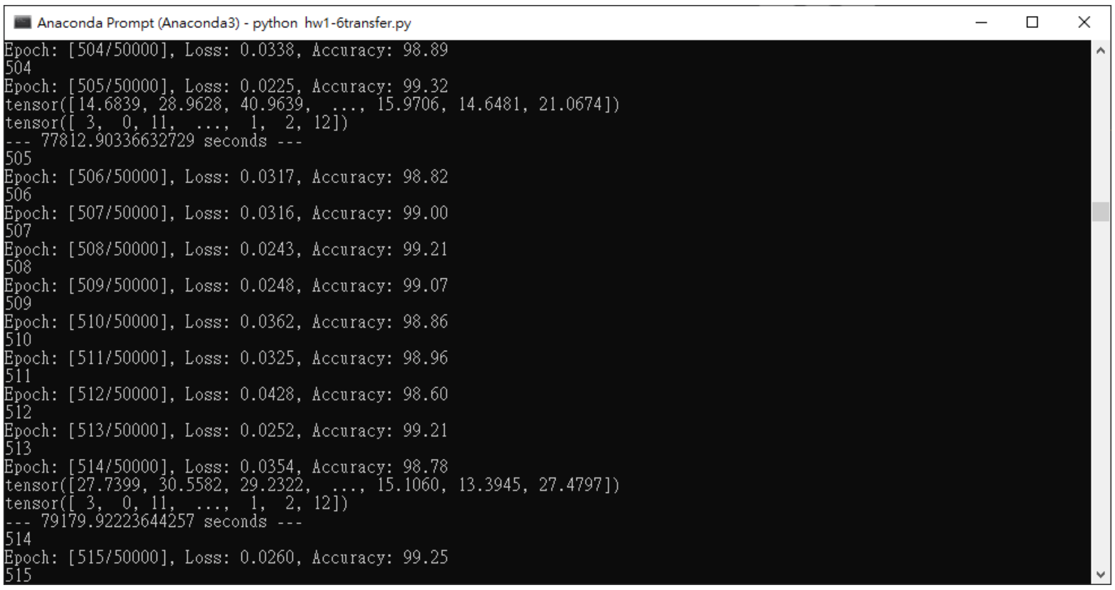
After the basic structure of my network works., I had start to find ways to improve the efficiency and accuracy. I had found that I can use cuda to speedup the training process, though it may also have memory issue – gpu may not have enough space. So, I try to add a little hidden size (like from 16's exponent to 20's exponent), decrease kernel size (like from 7 7 5 3 3 to 5 5 3 3 3) to not let image become too small while in memory limit, and go deeper epochs for higher chance to have its stable output. But since I don't know when the network would converge, so I let the network to output its current prediction after several epoch. The result is significant, I had increased the accuracy to 77 %, but there is another bottleneck appears: since I can now reach the convergence point of the network, but I cannot do further huge improvement by hardware limit. So, then I try to use simple ResNet-like structure to prevent overfit. But for ResNet, although it may not easily overfit, it may also converge, it had reached its max possible accuracy of local accuracy 100 in few hundred epochs, and at its highest accuracy at around 73, near to our original network structure. Bottleneck of accuracy appears again.



```python
class NetWork(nn.Module):  # network structure
    # read size when initial calss member
    # for more flexibility
    def __init__(
            self, input_size,
            hid_size1, hid_size2,
            hid_size3, hid_size4,
            num_classes):
        super(NetWork, self).__init__()
        self.conv1 = nn.Conv2d(
            input_size, hid_size1,
            kernel_size=7, stride=1, padding=2)  # convlution layer 1
        self.conv2 = nn.Conv2d(
            hid_size1, hid_size2,
            kernel_size=7, padding=1)  # convlution layer 2
        self.conv3 = nn.Conv2d(
            hid_size2, hid_size3,
            kernel_size=5, padding=1)  # convlution layer 3
        self.conv4 = nn.Conv2d(
            hid_size3, hid_size4,
            kernel_size=3, padding=1)  # convlution layer 4
        self.conv5 = nn.Conv2d(
            hid_size4, hid_size5,
            kernel_size=3, padding=1)  # convlution layer 5
        self.linear1 = nn.Linear(
            hid_size6, hid_size7)  # mlp layer 1
        self.linear2 = nn.Linear(
            hid_size7, hid_size7)  # mlp layer 2
        self.linear3 = nn.Linear(
            hid_size7, num_classes)  # mlp layer 3
        self.pool = nn.MaxPool2d(3, 2)  # maxpool
        self.norm = nn.BatchNorm2d(32)  # batch nomarlization
        self.d = nn.Dropout(0.5)  # rop 50 %
```

▲ Structure of network use on day6. This give me an accuracy of 68% at 700th epoch.

Phase3-Pretrain Model


Anaconda Prompt (Anaconda3) - python hw1-6transfer.py

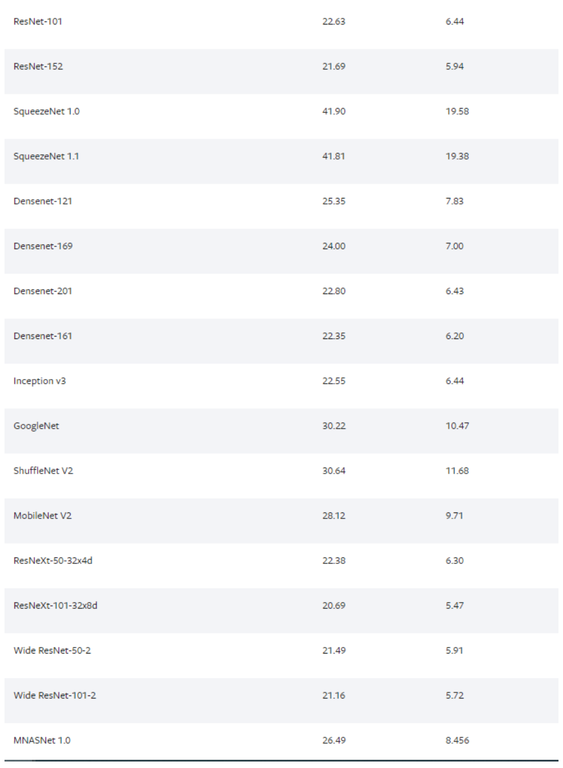
```
Epoch: [504/50000], Loss: 0.0338, Accuracy: 98.89
504
Epoch: [505/50000], Loss: 0.0225, Accuracy: 99.32
tensor([14.6839, 28.9628, 40.9639,  ..., 15.9706, 14.6481, 21.0674])
tensor([ 3,  0, 11,  ...,  1,  2, 12])
--- 77812.90336632729 seconds ---
505
Epoch: [506/50000], Loss: 0.0317, Accuracy: 98.82
506
Epoch: [507/50000], Loss: 0.0316, Accuracy: 99.00
507
Epoch: [508/50000], Loss: 0.0243, Accuracy: 99.21
508
Epoch: [509/50000], Loss: 0.0248, Accuracy: 99.07
509
Epoch: [510/50000], Loss: 0.0362, Accuracy: 98.86
510
Epoch: [511/50000], Loss: 0.0325, Accuracy: 98.96
511
Epoch: [512/50000], Loss: 0.0428, Accuracy: 98.60
512
Epoch: [513/50000], Loss: 0.0252, Accuracy: 99.21
513
Epoch: [514/50000], Loss: 0.0354, Accuracy: 98.78
tensor([27.7399, 30.5582, 29.2322,  ..., 15.1060, 13.3945, 27.4797])
tensor([ 3,  0, 11,  ...,  1,  2, 12])
--- 79179.92223644257 seconds ---
514
Epoch: [515/50000], Loss: 0.0260, Accuracy: 99.25
515
```

▲ Running ResNext 101 32x8d Pretrain model with default value.

Day 7 – 15

I had asked someone who had previously learned deep learning for some advice, and he asked me about the train data's size. After knowing my structure and the providing data, he told me that he thought the training data may not be enough, it is better to use transfer learning method. The transfer learning models used here were all provided by model zoo, and can be access by calling functions in pytorch. So, I try to start from pretrain Resnet network, it is so astonishing that even for only running 5 epochs, it could reach like 92% accuracy. Then I start wondering why transferlearning is so powerful, it is said that the pretrained model had learned a lot of low-level features, which these low-level features could be the general features of many category, so it could also be apply in our cases. And also, pretrain model would also hard to be oevrfit, so we can go deeper. I want to know which pretrain model would have best accuracy for this assignment, so I had tried multiple models which had already get low top-1 error and top-5 error, like DenseNet 161 and 201, GoogleNet, wide ResNet 50-2 and 101-2, ResNext 101 32x8d, and ResNet 152. Base on result on Kaggle, it seems that most of the model can reach 95% accuracy if we simply use one layer of fully connected layer as output, and model beside DenseNet and GoogleNet can convergence in a few hundred epochs, since network like GoogleNet have a larger and deeper structure itself, and in result DenseNet get its highest accuracy around 96%.

Also, in this modification, with requirement to access the pretrain model, I had changed the way to read data, which had highly increased the efficiency of reading data. Now we did not to read image as image file, but to read them as pure data, which can save time for data processing, and to preserve the originality of each image.



ResNet-101	22.63	6.44
ResNet-152	21.69	5.94
SqueezeNet 1.0	41.90	19.58
SqueezeNet 1.1	41.81	19.38
Densenet-121	25.35	7.83
Densenet-169	24.00	7.00
Densenet-201	22.80	6.43
Densenet-161	22.35	6.20
Inception v3	22.55	6.44
GoogleNet	30.22	10.47
ShuffleNet V2	30.64	11.68
MobileNet V2	28.12	9.71
ResNeXt-50-32x4d	22.38	6.30
ResNeXt-101-32x8d	20.69	5.47
Wide ResNet-50-2	21.49	5.91
Wide ResNet-101-2	21.16	5.72
MNASNet 1.0	26.49	8.456

▲ A small part of pytorch supported pretrained model. They all have good top-1 (mid) error and top-5 error (right), I just try few of them with running only few nodes and it already works really well.

Phase 4-Data Augmentation:
Day 16-20

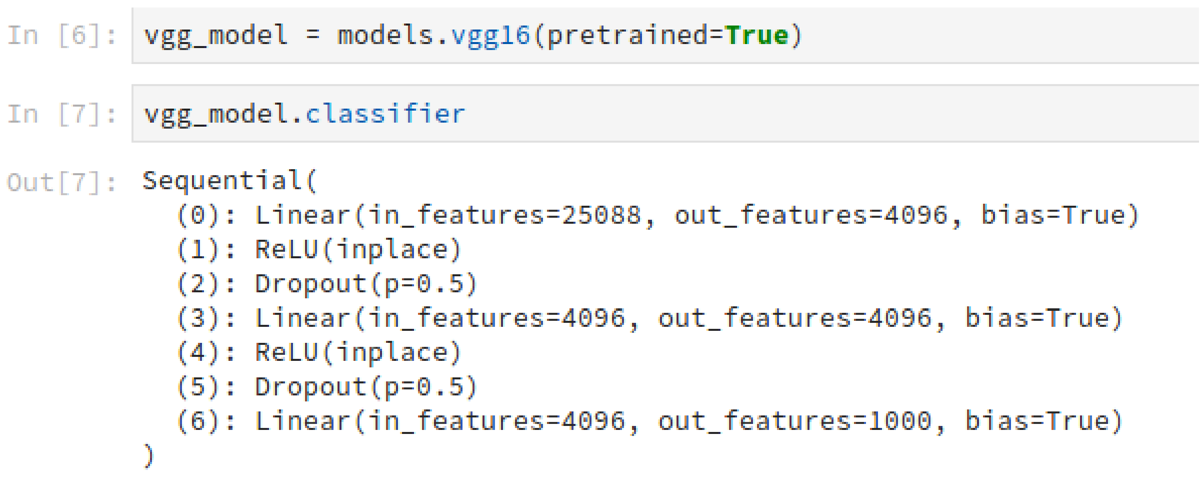
Since now we had now the power of pretrain model, and know it had gone to its limit, we now need further improve for our model. But I don’t want to break the pretrain structure then so I think maybe I can do some modification on the training data.

So, I try to add some additional data transform to make more additional data. Each time when dataloader is going to load image’s data, we randomly do the data augmentation: training data has chance to flip horizontally, and has chance to rotate left or right within angle of 5. I had tried random flip vertically, but since I had found that it is unreasonable for a building to be upside down, and the accuracy is not increase after test as well, so I had gotten rid of it; I had tried the rotation degree as well, but seems it did not greatly improve the performance (though rotation 5 did not improve much as well).

In my opinion, I think when we are taking a picture of building, normally we would posing the camera correctly, so maybe the rotation is not helpful in this case.

Horizontally flip camera can generate 2x of data already, so I had saved this part.

Phase 5-Modify the last part of Network



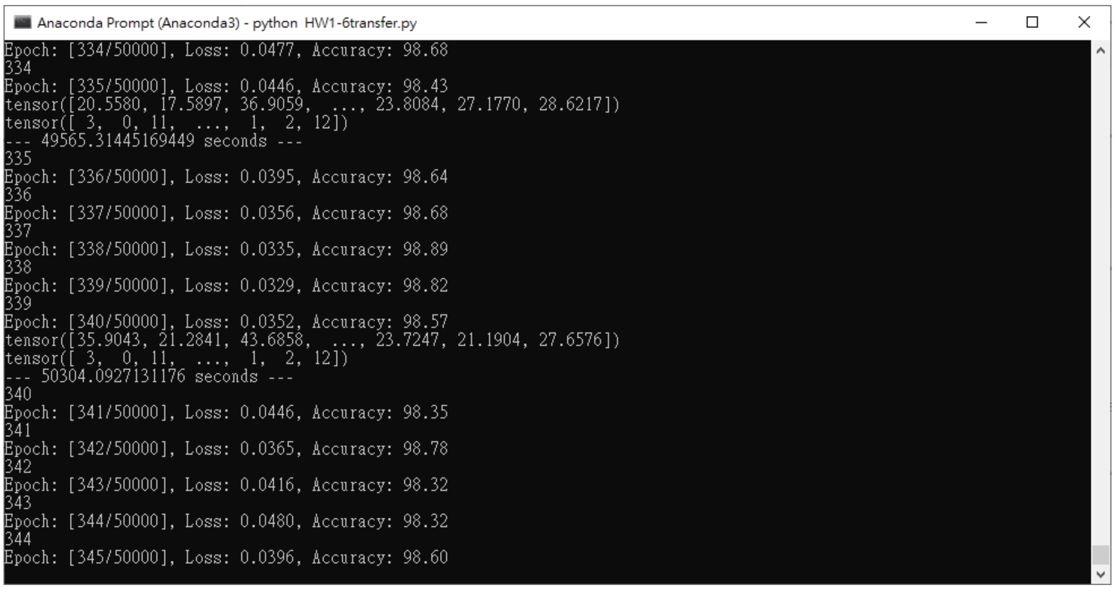
```
In [6]: vgg_model = models.vgg16(pretrained=True)

In [7]: vgg_model.classifier

Out[7]: Sequential(
          (0): Linear(in_features=25088, out_features=4096, bias=True)
          (1): ReLU(inplace)
          (2): Dropout(p=0.5)
          (3): Linear(in_features=4096, out_features=4096, bias=True)
          (4): ReLU(inplace)
          (5): Dropout(p=0.5)
          (6): Linear(in_features=4096, out_features=1000, bias=True)
        )
```

▲ What vgg's classifier be like. I had constructed my classifier as follow same rule.

In the last few days, I had tried to modify the full connected layers. I want to see if we have more layers on output, will the accuracy increase? I had tried to add the output mlp layers to 3, the time to convergence take a lot more, but result did not improve much, but the used time for training absolutely increased. And I also try some other data augmentation methods, like color jitter, to see whether the result would become better.



Anaconda Prompt (Anaconda3) - python HW1-6transfer.py

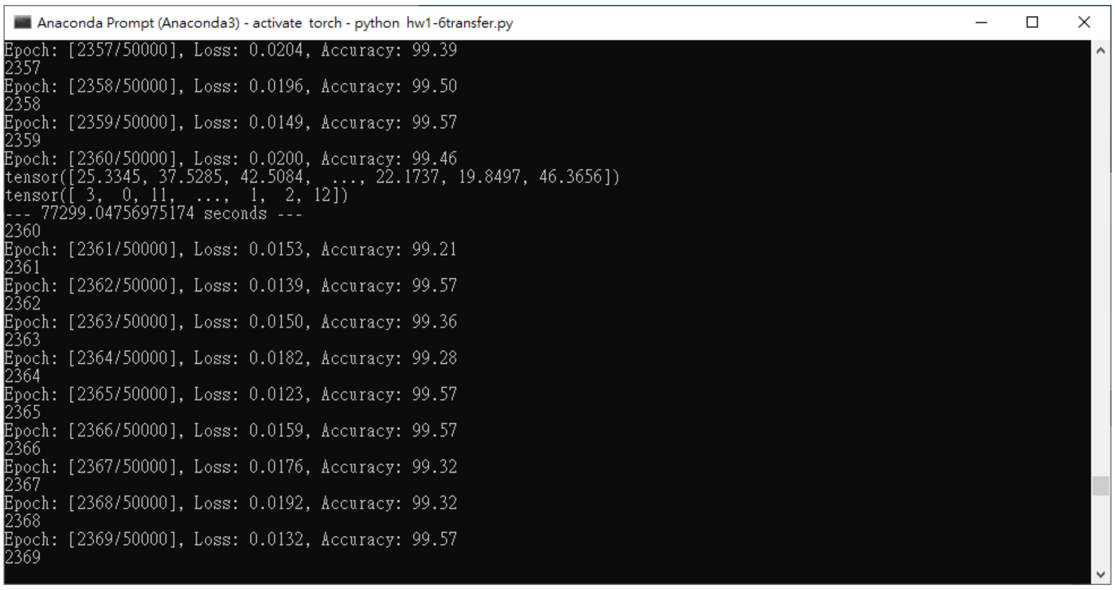
```
Epoch: [334/50000], Loss: 0.0477, Accuracy: 98.68
334
Epoch: [335/50000], Loss: 0.0446, Accuracy: 98.43
tensor([20.5580, 17.5897, 36.9059,  ..., 23.8084, 27.1770, 28.6217])
tensor([ 3,  0, 11,  ...,  1,  2, 12])
--- 49565.31445169449 seconds ---
335
Epoch: [336/50000], Loss: 0.0395, Accuracy: 98.64
336
Epoch: [337/50000], Loss: 0.0356, Accuracy: 98.68
337
Epoch: [338/50000], Loss: 0.0335, Accuracy: 98.89
338
Epoch: [339/50000], Loss: 0.0329, Accuracy: 98.82
339
Epoch: [340/50000], Loss: 0.0352, Accuracy: 98.57
tensor([35.9043, 21.2841, 43.6858,  ..., 23.7247, 21.1904, 27.6576])
tensor([ 3,  0, 11,  ...,  1,  2, 12])
--- 50304.0927131176 seconds ---
340
Epoch: [341/50000], Loss: 0.0446, Accuracy: 98.35
341
Epoch: [342/50000], Loss: 0.0365, Accuracy: 98.78
342
Epoch: [343/50000], Loss: 0.0416, Accuracy: 98.32
343
Epoch: [344/50000], Loss: 0.0480, Accuracy: 98.32
344
Epoch: [345/50000], Loss: 0.0396, Accuracy: 98.60
```

▲ Data augmentation with jitter apply. Takes a lot of time for few epochs.

Finding and Summary:

Go deeper?

I had found that even if we let the model go deeper, it is not guarantee to get a better result. Usually we can get our best result while the model is not statistic with its accuracy, after that the model's accuracy would not change a lot. I think It may be issue of overfit. But this is a tradeoff, since the convergence model would have a favorable performance, though it may not be best but its good enough; on the contrary, if we try to use a model is not converge yet, it maybe will have a better performance by chance, but usually not as good as convergence ones.



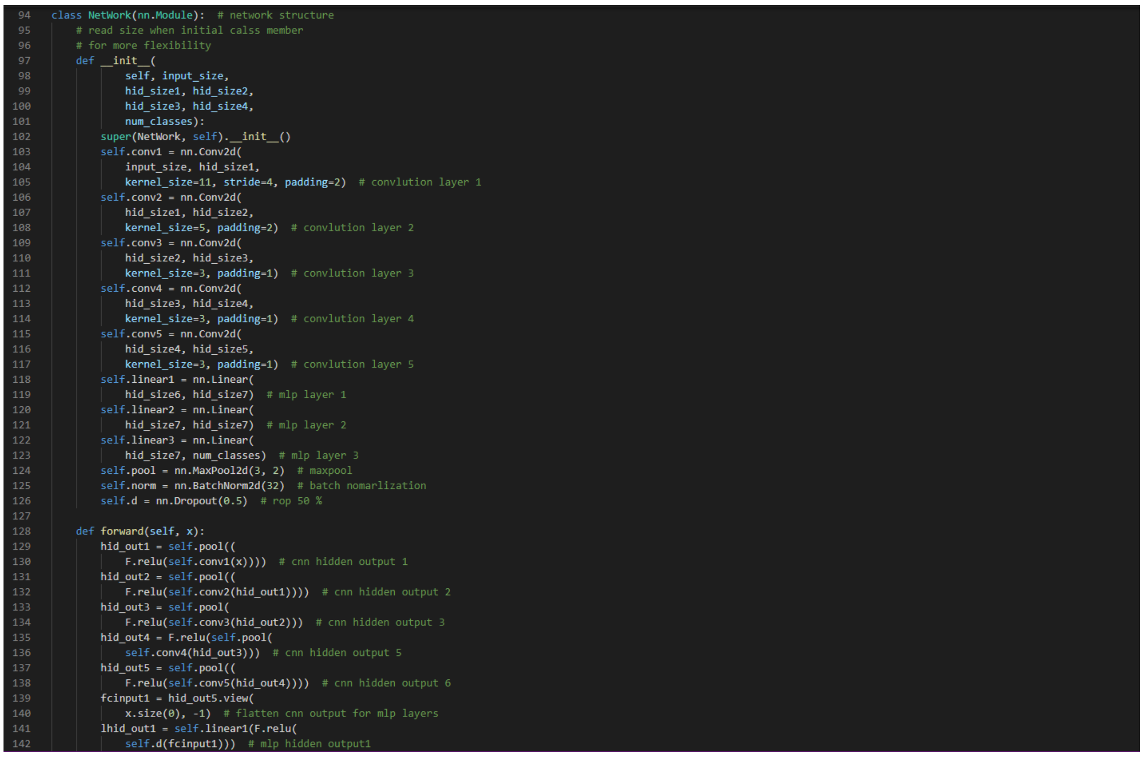
Anaconda Prompt (Anaconda3) - activate torch - python hw1-6transfer.py

```
Epoch: [2357/50000], Loss: 0.0204, Accuracy: 99.39
2357
Epoch: [2358/50000], Loss: 0.0196, Accuracy: 99.50
2358
Epoch: [2359/50000], Loss: 0.0149, Accuracy: 99.57
2359
Epoch: [2360/50000], Loss: 0.0200, Accuracy: 99.46
tensor([25.3345, 37.5285, 42.5084,  ..., 22.1737, 19.8497, 46.3656])
tensor([ 3,  0, 11,  ...,  1,  2, 12])
--- 77299.04756975174 seconds ---
2360
Epoch: [2361/50000], Loss: 0.0153, Accuracy: 99.21
2361
Epoch: [2362/50000], Loss: 0.0139, Accuracy: 99.57
2362
Epoch: [2363/50000], Loss: 0.0150, Accuracy: 99.36
2363
Epoch: [2364/50000], Loss: 0.0182, Accuracy: 99.28
2364
Epoch: [2365/50000], Loss: 0.0123, Accuracy: 99.57
2365
Epoch: [2366/50000], Loss: 0.0159, Accuracy: 99.57
2366
Epoch: [2367/50000], Loss: 0.0176, Accuracy: 99.32
2367
Epoch: [2368/50000], Loss: 0.0192, Accuracy: 99.32
2368
Epoch: [2369/50000], Loss: 0.0132, Accuracy: 99.57
2369
```

▲ Run pretrain googlenet for 2300+ epochs, but result is not better than first 500 epochs.

How to adjust hyperparameter?

Since I did not adjust a lot on the pretrain model, I would like to talk about my experience of using self-structured convolutional neural network. In my opinion, the data shape size is important, since we can calculate how an image size may become after doing convolution, we need to be careful not to make it too small, in other word is to try not to let the kernel size of convolutional layer too big, and the stride should not set too big as well. It would be better to have smaller kernel size and with more layers, rather than to skip a lot of possible features in early stage. The hidden layer size should not set too small, or the parameter may not be enough. It depends whether to set them high, if you want a descent output you can try more; for more possibility you can try less.



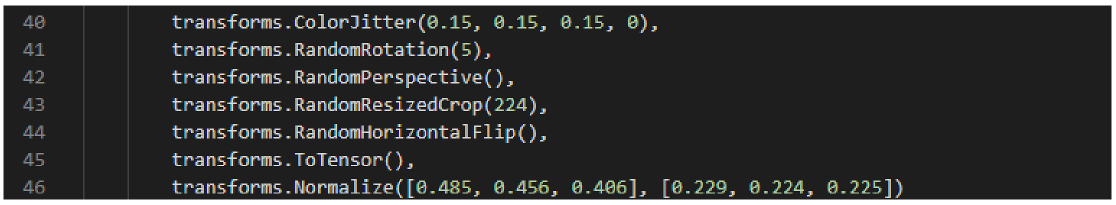
```python
class NetWork(nn.Module):  # network structure
    # read size when initial calss member
    # for more flexibility
    def __init__(
            self, input_size,
            hid_size1, hid_size2,
            hid_size3, hid_size4,
            num_classes):
        super(NetWork, self).__init__()
        self.conv1 = nn.Conv2d(
            input_size, hid_size1,
            kernel_size=11, stride=4, padding=2)  # convlution layer 1
        self.conv2 = nn.Conv2d(
            hid_size1, hid_size2,
            kernel_size=5, padding=2)  # convlution layer 2
        self.conv3 = nn.Conv2d(
            hid_size2, hid_size3,
            kernel_size=3, padding=1)  # convlution layer 3
        self.conv4 = nn.Conv2d(
            hid_size3, hid_size4,
            kernel_size=3, padding=1)  # convlution layer 4
        self.conv5 = nn.Conv2d(
            hid_size4, hid_size5,
            kernel_size=3, padding=1)  # convlution layer 5
        self.linear1 = nn.Linear(
            hid_size6, hid_size7)  # mlp layer 1
        self.linear2 = nn.Linear(
            hid_size7, hid_size7)  # mlp layer 2
        self.linear3 = nn.Linear(
            hid_size7, num_classes)  # mlp layer 3
        self.pool = nn.MaxPool2d(3, 2)  # maxpool
        self.norm = nn.BatchNorm2d(32)  # batch nomarlization
        self.d = nn.Dropout(0.5)  # rop 50 %

    def forward(self, x):
        hid_out1 = self.pool((
            F.relu(self.conv1(x))))  # cnn hidden output 1
        hid_out2 = self.pool((
            F.relu(self.conv2(hid_out1))))  # cnn hidden output 2
        hid_out3 = self.pool(
            F.relu(self.conv3(hid_out2)))  # cnn hidden output 3
        hid_out4 = F.relu(self.pool(
            self.conv4(hid_out3)))  # cnn hidden output 5
        hid_out5 = self.pool((
            F.relu(self.conv5(hid_out4))))  # cnn hidden output 6
        fcinput1 = hid_out5.view(
            x.size(0), -1)  # flatten cnn output for mlp layers
        lhid_out1 = self.linear1(F.relu(
            self.d(fcinput1)))  # mlp hidden output1
```

▲ For hardware issue, the last version of my self-sturctured network have kernel size in each layer of 11 5 3 3 3 to save memory space of GPU

About data augmentation:

Prior adjust image is a way to improve performance or to save some memory size, especially when you have the issue of memory space shortage on gpu. In my observation, it is possible to slightly decrease the image size: but if image size is lower than 200 * 200, the accuracy may start to strongly decrease, since a lot of information may be missing, and some fake information may be generated while resizing. To save the size of dataset and also benefit from augmentation, I would like to recommend to randomly do the data transform on data loader. Horizontal flip may be a best choice, since it is so trivial that it would not generate any fake information and also can increase the data amount by 2x. Rotation in my opinion is not a must do for this case. I had not tried yet but maybe randomly set crop center of image would be a good idea, since we can generate loads of image from the original image, just they are parts of original image, so the information may also be correct. It may be risky to generate fake data, since we would not know whether these data will or will not influence our result.



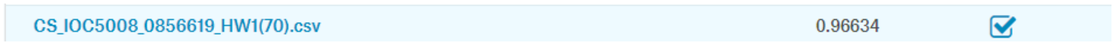
```
            transforms.ColorJitter(0.15, 0.15, 0.15, 0),
            transforms.RandomRotation(5),
            transforms.RandomPerspective(),
            transforms.RandomResizedCrop(224),
            transforms.RandomHorizontalFlip(),
            transforms.ToTensor(),
            transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
```

▲ Try to stack a lots of augmentation together. It did not guarantee to get good performance, but it would let your model hard to converge.

Summary:

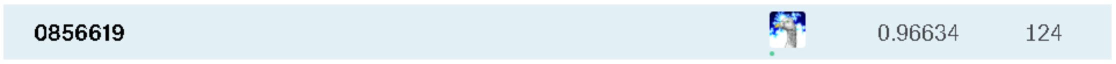
For this assignment, we are required to use 2810 image of 13 categories to classify 1040 input. For the training data, it may not be enough. So, in this case, we may need more data, but not allow to add additional train data images. Using the techniques of data augmentation to larger the dataset, and to use the power of transfer learning to get general low-level feature had get pre-trained model which had trained on loads of image already may be a great aid to help us to well-classify the testing data. Finally, if it is possible, we can try to improve the structure, to add few more layers to try whether it would be matched for this test case.

What I had learned in this assignment is that try not to overfitting the model. Once its overfitting, the accuracy would not increase anymore. Complicated structure may take more time before converge. And doing data augmentation may be a good strategy to increase training data without any more image into dataset. If I am going train another deep learning network in the future, I would like to try more data augmentation method.



CS_IOC5008_0856619_HW1(70).csv	0.96634	✓

▲ Best result currently.



0856619		0.96634	124

▲ Total submits.