

Neural Network Homework 1:Hand

Gesture Recognition Report

410421304 資工四 秘子尉

Description of Design:

In this assignment, we are using convolution neural network (CNN) to finish this assignment. CNN trained the model using multiple convolution layers (with addition components), connect with MLP as output, which are more flexible than only use MLP. Beside hand-made structure, we could find some pre-build structures, such as googlenet, alexnet, vgg...etc.

In my design, after several of experiments, and consider my hardware's capability, I decided to use alexnet-like structure network to adjust. In my experiments, there are not only trying self and pre-build model, also some tactics to make accuracy higher.

```
class Net(nn.Module):
    def __init__(self, input_size, hid_size1, hid_size2, hid_size3, hid_size4, num_classes):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(input_size, hid_size1, kernel_size=5, stride=1, padding=2) #convolution layer 1
        self.conv2 = nn.Conv2d(hid_size1, hid_size2, kernel_size=3, padding=1) #convolution layer 2
        self.conv3 = nn.Conv2d(hid_size2, hid_size3, kernel_size=3, padding=1) #convolution layer 3
        self.conv4 = nn.Conv2d(hid_size3, hid_size4, kernel_size=3, padding=1) #convolution layer 4
        self.conv5 = nn.Conv2d(hid_size4, hid_size5, kernel_size=3, padding=1) #convolution layer 5
        self.linear1 = nn.Linear(hid_size5, hid_size6) #mlp layer 1
        self.linear2 = nn.Linear(hid_size6, hid_size7) #mlp layer 2
        self.linear3 = nn.Linear(hid_size7, num_classes) #mlp layer 3
        self.pool = nn.MaxPool2d(2, 2) #max pool
        self.norm = nn.BatchNorm2d(32) #batch normalization
        self.d = nn.Dropout(0.5) #drop 50%

    def forward(self, x):
        hid_out1 = self.pool(F.relu(self.conv1(x))) #conv hidden output 1
        hid_out2 = self.pool(F.relu(self.conv2(hid_out1))) #conv hidden output 2
        hid_out3 = self.pool(F.relu(self.conv3(hid_out2))) #conv hidden output 3
        hid_out4 = self.pool(F.relu(self.conv4(hid_out3))) #conv hidden output 4
        hid_out5 = self.pool(F.relu(self.conv5(hid_out4))) #conv hidden output 5
        hid_out6 = hid_out5.view(x.size(0), -1) #flatten conv output for mlp layers
        hid_out7 = self.linear1(F.relu(self.d(hid_out6))) #mlp hidden output1
        hid_out8 = self.linear2(F.relu(self.d(hid_out7))) #mlp hidden output2
        out = self.linear3(hid_out8) #mlp hidden output3
        prob = F.softmax(out, dim=1) #do softmax, then output
        return out
```

(Fig of alex-like structure. Take the experience on the stack of the network)

First and basic, is the parameters. In my experiments, learning rate would determine huge on the output. Setting learning rate low will make model learn slower, but it can also let the model learn something model with higher learning rate cannot learn. Epochs may be an important parameter either. When we go deeper, the model will be closer to its limit. Though there is pros and cons, which the limit may or may not be the optimal solution (which in this case, usually not).

Second is structure. Everyone know structure plays an important role, but how to adjust? Did there exist some principle? Well maybe not, normally just try and error, but using some pre-build model is just like steps on giant's shoulder, we can use other people's experience to try to apply on our cases, if its worse than just use our original design, but if its better maybe we can try to build our model base on this

design, so it's definitely worth a try.

Third is the early state. I had observed that a better early state of accuracy may lead to a better opportunity of getting higher end(limit) stat, which is just like the gradient descent: one can fall in a local minimum and cannot jump out. This is useful when you try to go deeper, and with additional condition that your learning rate is low enough. If the learning rate is too high, the learning process will more "random", harder to learn a stuff, which may not get a result in a given time.

Now talk about how I adjust my model. Firstly, it was fully base on the result accuracy base on the grading system: if we increase epoch and accuracy become higher, then we keep increase. In this phase I only use self-build model.

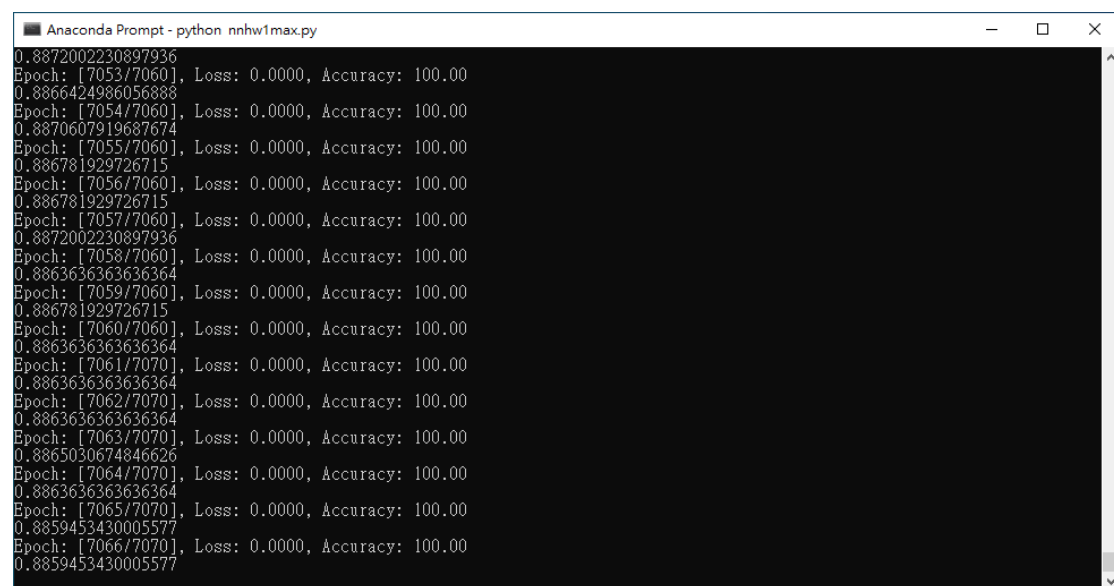
But it has some limitations: my model cannot get a higher accuracy no matter how I adjust. So, in second phase, I try to use the models introduce in the class, and use same model adjustment tactics. But due to the limitation of the device, for most pre-build model I need to lower the layers for a much faster result acquired speed, but it also took a long time to get the result.

As a reason, I start to use GPU to accelerate the computation, but this may lead to another problem: CUDA may out of memory if the model is too big. So now we can choose to lower the layer counts of the model again, or lower batch size in exchange of lower running speed (which still higher than using CPU to do computation). But if your device is not good enough, you will need to do it both. As for google net, I even need to lower each inception net to 1 layer to let the model to be able to train, so is VGG...etc. The outcome would be predictably not good enough, since these nets were all famous for its high-layer structure. So later-on I am going to focus adjusting alexnet-like structure, as it's is the kind of network best match my device, the structure most similar to my original design which made me feeling familiar, and also the highest outcome at that time.

Decided the main design of the model, I am now designing the tactics to adjust this model. First is by checking its accuracy: by evaluate the model's correct samples compare to testing samples, we can now it's current inner accuracy. I found that 100% inner accuracy may not guarantee an optimal result, but if you cannot even reach 100% of inner accuracy in this state, it is definitely not optimal result, so I set first limitation: training loop can only end when inner accuracy is 100%.

Next is the loss rate. In my experiments, models can reach 100% accuracy in very early epoch, but loss may still decrease. Loss increase when only model accidentally adjust fail (this frequency depends on learning rate), so in my opinion, lower loss means higher accuracy. So now I abandon use the epoch counts as end condition, in exchange using loss as our observation parameter. This strategy really works, which I did not need to take the risk of model under certain epochs accidentally

getting lower accuracy.



```
Anaconda Prompt - python nnhw1max.py
0.8872002230897936
Epoch: [7053/7060], Loss: 0.0000, Accuracy: 100.00
0.8866424986056888
Epoch: [7054/7060], Loss: 0.0000, Accuracy: 100.00
0.8870607919687674
Epoch: [7055/7060], Loss: 0.0000, Accuracy: 100.00
0.886781929726715
Epoch: [7056/7060], Loss: 0.0000, Accuracy: 100.00
0.886781929726715
Epoch: [7057/7060], Loss: 0.0000, Accuracy: 100.00
0.8872002230897936
Epoch: [7058/7060], Loss: 0.0000, Accuracy: 100.00
0.8863636363636364
Epoch: [7059/7060], Loss: 0.0000, Accuracy: 100.00
0.886781929726715
Epoch: [7060/7060], Loss: 0.0000, Accuracy: 100.00
0.8863636363636364
Epoch: [7061/7070], Loss: 0.0000, Accuracy: 100.00
0.8863636363636364
Epoch: [7062/7070], Loss: 0.0000, Accuracy: 100.00
0.8863636363636364
Epoch: [7063/7070], Loss: 0.0000, Accuracy: 100.00
0.8865030674846626
Epoch: [7064/7070], Loss: 0.0000, Accuracy: 100.00
0.8863636363636364
Epoch: [7065/7070], Loss: 0.0000, Accuracy: 100.00
0.8859453430005577
Epoch: [7066/7070], Loss: 0.0000, Accuracy: 100.00
0.8859453430005577
```

(fig shows that as we go deeper, loss and accuracy may always go to its limit. But real challenge is how to adjust after that)

An extra method had come out in some timing: why not use neural network to train neural network? So, I had tried to design a simple MLP model, which can help me to evaluate the model's accuracy. The design is simple: giving an output sequence as training sample, its accuracy given by grading system as regression result, then train this model. Well works in early tries, if the accuracy is originally low enough it can give me a warning which can avoid me using precious-daily-hand-in limitation. But when output's accuracy become higher, it may show at 1~2% of bias, for example if you have a model of 97% accuracy, my model may give it a 96~98% of accuracy, which...definitely not useful enough we want to challenge the top of the leaderboard, so I did not use this so much. The second version of this design is to use 'voting' strategy. As for my every assign output, I can acquire its accuracy, then for each prediction it has it's voting weight same as its accuracy. The label which get a maximum sum vote would be our result label. Then compare this result with our desire output to get our own prediction score. Of course, use in evaluation only, we did not hand in this as any assignment. The result may not be too good, since there may be some label which all my predictions were wrong, but it is useful when we use this data to monitor our training process as a comparison.

```

100 #accs = [] #add accs for observation
101 #loss = [] #add loss for observation
102 while True: #infinite loop
103     for epoch in range(num_epochs): #0 -> 9 run loop count depend on epoch
104         for i, (data, label) in enumerate(trainloader): #watch input data from dataloader
105             trainloss = []
106
107             data = Variable(data.view(-1,1,28,28),requires_grad=False).cuda() #reshape data we get from data loader for cpu
108             label = Variable(label.view(-1,1),requires_grad=False).cuda() #use long to avoid RuntimeError: Expected object of scalar type Long but got scalar type Float for argument #0 'target'
109             optimizer.zero_grad() #set optimizer to grad
110             outputs = model2(data) #get prediction from model
111
112             labels = labels.view(outputs.shape[0]).cuda() #for batch size beside set ones
113
114             loss = criterion(outputs, labels) #get this epoch's loss
115             loss.backward() #back propagation
116             optimizer.step() #optimizer optimize
117
118             labelc = label.cuda() #duplicate label
119
120             trainloss.append(loss.item()) #for calculate loss
121
122             model2.eval() #evaluation mode
123             correct = (outputs == labelc).sum() #calculate correct items
124
125
126             if (i) % 10 == 0 and i != 0: #1000 * 10 -> loader load 10 times -> numbered as 0 - 10 -> max is 10
127                 correct = torch.sum(torch.argmax(outputs,dim=1)==labels) #count the correct classification
128                 print (f'epoch: {i/10}, loss: {loss.item():.4f}, Accuracy: {2 if (epoch*1 + 10*100, num_epochs + 10*100, np.mean(trainloss), (correct * 100 / outputs.shape[0]))} #for our observation
129
130             with torch.no_grad(): #disable data grad
131                 model1 = copy.deepcopy(model2) #use model from current model and set it on CPU, for prevent to much thing in gpu
132                 model1 = model1.cpu() #not on gpu
133                 for i, (data) in enumerate(testloader): #load test data
134                     data = Variable(data,requires_grad=False) #transfer forward to Variable for network
135
136                     outputs = model1(data).cpu() #get prediction of test data
137                     outputs = outputs.detach() #not
138                     outc_index = torch.max(outputs,1) #get predict label
139                     indexc = indexc.cpu() #not on gpu
140                     indexc = indexc.numpy() #change format prepare for observation and output
141                     testy = indexc.data

```

(Fig of infinite loop for training)

In the last few weeks, I wants to see each model's limitation: I set up an infinite loop for the model, and after certain accuracy it may output a pair of result and model, so I can check its accuracy performance in each training sub-process. This design is the best design I had made, which if I found an early-low accuracy, I can quickly restart; if we had saw convergence which accuracy stop to adjust or become lower and lower, then we know it is time to stop, and then we can adjust the parameter for another long run. Most importantly, I may not miss the high accuracy model in the mid-way of training, and save my time of adjusting and waiting the result for all day.

Sum up the design, I had use alexnet-like structure to design 5-layer CNN network, mainly focus on adjusting learning rate kernel size and layers, use auto tactics to help me to design the model for this assignment.

Description of code:

(main program: nnhw1final.py)

```

75 class Network(nn.Module): #network structure
76     def __init__(self, input_size, hid_size1, hid_size2, hid_size3, hid_size4, num_classes): # read size when initial calls member for more flexibility
77         super(Network, self).__init__()
78         self.conv1 = nn.Conv2d(input_size, hid_size1, kernel_size = 5, stride = 1, padding = 2) #convolution layer 1
79         self.conv2 = nn.Conv2d(hid_size1, hid_size2, kernel_size = 3, padding = 2) #convolution layer 2
80         self.conv3 = nn.Conv2d(hid_size2, hid_size3, kernel_size = 3, padding = 1) #convolution layer 3
81         self.conv4 = nn.Conv2d(hid_size3, hid_size4, kernel_size = 3, padding = 1) #convolution layer 4
82         self.conv5 = nn.Conv2d(hid_size4, hid_size5, kernel_size = 3, padding = 1) #convolution layer 5
83         self.linear1 = nn.Linear(hid_size5, hid_size7) #mlp layer 1
84         self.linear2 = nn.Linear(hid_size7, hid_size7) #mlp layer 2
85         self.linear3 = nn.Linear(hid_size7, num_classes) #mlp layer 3
86         self.pool = nn.MaxPool2d(3, 2) #maxpool
87         self.norm = nn.BatchNorm2d(32) #batch normalization
88         self.d = nn.Dropout(0.5) #drop 50 %
89
90     def forward(self, x):
91         hid_out1 = self.pool((F.relu(self.conv1(x)))) #conv hidden output 1
92         hid_out2 = self.pool((F.relu(self.conv2(hid_out1)))) #conv hidden output 2
93         hid_out3 = self.pool((F.relu(self.conv3(hid_out2)))) #conv hidden output 3
94         hid_out4 = F.relu(self.conv4(hid_out3)) #conv hidden output 5
95         hid_out5 = self.pool((F.relu(self.conv5(hid_out4)))) #conv hidden output 6
96         fcinput1 = hid_out5.view(x.size(0), -1) #flatten conv output for mlp layers
97         lhidd_out1 = self.linear1(F.relu(self.d(fcinput1))) #mlp hidden output1
98         lhidd_out2 = self.linear2(F.relu(self.d(lhidd_out1))) #mlp hidden output2
99         out = self.linear3(lhidd_out2) #hidden output3
100         prob = F.softmax(out, dim=1) #do softmax, then output
101         return out

```

(Fig of main network structure: hidden size can be adjusted when initial model)

(line 1 to 37)

First we need to prepare the library we are going to use. Then, set up our size of each layer here. We will later use this as the input to initialize a class member of `traindataset` and `testdataset` to load the required data in the program. And also set up the parameters we are going to initial for the CNN network.

Sometimes clean up GPU and prevent us from struggle in “CUDA out of memory” error, so we add a line to clean it up as well. And also, we load an “acc.csv”, which is for use as observe the accuracy change during the process. Now we use the highest result output (99.8% accurate) to observe.

(line 41 to 68)

Initialize two classes for loading training and testing data. For complete the function, I also add the ability to return the length and item. Note that this is a bit different from prepare data for MLP, since CNN need 4-dimensinal data as input, we need to reshape it here.

(line 49 to 54)

Initial class member for the data-load class we had create. For initialize class member, we need the name of the data, when we initialize the member we will load the csv, strip the data inside and put it in list at the same time. Then we use this two classes member to build up two dataloader for loading data in model in batches. Each batch contain 1445 samples, with total 19 batches.

(line 75 to 101)

This is the main model structure part. Each picture with $28 * 28$ size is count as one input, so the input size of CNN is 1. Then we design the network by 2's N-dimension, here is from 64 to 1024, increase two times each layer, totally five layers. For kernel size I set as 5, 3, 3, 3, 3 respectively, with padding size 2, 1, 1, 1, 1, set all stride as 1. The total input data count is 27455. For each layer we would apply a RELU layer, and for 1, 2, 3, 5 layer of CNN model we build we would use MAXPOOL layer to decrease data amount in prevent of CUDA out of memory. MLP structure would be setup as 1024 and 4096 as their hidden size, for each layer apply RELU, and SOFTMAX for output layer.

(line 103 to 110)

We initialize a class member of the Network class we built as the model we are going to use in this assignment. Also initial our loss function as criterion, optimizer as ADAM. And now we initial the variable `ep10`, which is for record “how much loop we had run till now” as this is an infinite loop. Other two list are use to record the loss and accuracy in training process to use to observe the model training.

(line 111 to 176)

We use an infinite loop for the training, there is no terminal condition. We

would decide to terminate or not base on the accuracy outcome. If one model's accuracy is: 1. Get a bad starting state, which may huge influence the later result; 2. Accuracy did not change for a long time, such as for ~5000 epochs did not change or improve, or even worse. Under these two conditions we would stop the model, evaluate this model's output till now and decide whether go for a rerun or change another structure.

For each loop it contains only 10 epochs. It is originally use for terminate program under a super low loss (like 0,0000000001), but now use this to lower the memory usage. For each epoch we input data in batches, for each batch we will do the data prediction and output under certain condition

(line 113 to 132)

First is the normal start, we load batches sequentially in the model, from these batches we strip the data and label from inside, transform to format of variable for CNN model, use long for label to avoid error, do optimizer step and back propagation, and prepare the data we are going to use in next step.

(line 134 to 137)

Use loss and accuracy we had acquire from before, we output the result to screen once it reached "end of this loop", when the epoch count had met the setting condition. This step is also a standard step.

```

113 if (i) % 10 == 0 and i != 0: #end of 10 epochs loop, transfer back to train, transfer back to 0 - 10 epochs loop
114     correct = torch.sum(torch.argmax(outputs, dim=1) == labels) # count the correct classification
115     print("Epoch: %d, Loss: %.4f, Accuracy: %.2f" % (epoch+1, loss, correct * 100 / outputs.shape[0])) #for our observation
116     % (epoch+1, loss, correct * 100 / outputs.shape[0])) #for our observation
117
118 with torch.no_grad(): # disable auto grad
119     model2 = copy.deepcopy(model2) #copy model from current model and set it as CPU, for prevent to much thing in gpu
120     model2 = model2.cpu() #set on cpu
121     for i, (data) in enumerate(testloader): #load test data
122         data = Variable(data, requires_grad=False) #transfer format to Variable for network
123
124         outputs = model2(data).cpu() #get prediction of test data
125         outputs = outputs.detach() #for
126         out, index = torch.max(outputs, 1) #get predict label
127         index = index.cpu() #back to cpu
128         index = index.numpy() #change format prepare for observation and output
129         testy = index #copy
130
131         dif = acclabel - testy #calculate how many times difference between our observation result and acquired output
132         hid_acc = (np.count_nonzero(dif == 0) / 7172) #divide by total
133         print(hid_acc)
134         hloss.append(hid_acc) #record local accuracy
135         hloss.append(trainloss) #record loss of this round
136         if (hid_acc > ori_acc): #get better accuracy - become better
137             print("New record!")
138             torch.save(model2, "hid_net.pt") #save the network
139             ori_acc = hid_acc #reset highest accuracy till now
140             output = pd.DataFrame({"smp_id": range(1, len(testy)+1), "label": testy}) #output result when we get current best
141             output.to_csv("mhulid10421304.csv", columns=["smp_id", "label"], index=False) #output result to csv
142         if (epoch) % 9 == 0 and (epoch) != 0: #start epoch in our loop
143             if epoch % 100 == 0 and epoch != 0: #HIDDEN
144                 #save per 100
145                 output = pd.DataFrame({"smp_id": range(1, len(testy)+1), "label": testy})
146                 output.to_csv("mhulid10421304.csv", columns=["smp_id", "label"], index=False) #output result to csv
147                 torch.save(model2, "hid_net.pt") #epoch * 10 + epoch + 1 - 10))
148                 #save data also
149                 print(hloss)
150                 hlossarray = np.asarray(hloss).ravel() #move accuracy
151                 hlossarray = np.asarray(hloss).ravel() #move loss
152                 output2 = pd.DataFrame({"smp_id": range(1, len(hlossarray)+1), "loss": hlossarray, "acc": hlossarray})
153                 output2.to_csv("hidob10421304.csv", columns=["smp_id", "loss", "acc"], index=False) #output result to csv for observation

```

(Fig of savepoints and observation part of program.)

(139 to 174)

Now is the important part of my design. When It is at the end of one loop (10 epoch), we would do a model evaluation. We had already setup model to evaluation model in previous step, so its no worry of break the original trained model.

We load the test data in model to see the result. Of course, it contains some data format change, which had noted in code. Then we would compare this prediction to "acc.csv", the observation-usage prediction set. Originally it was

generated by the result of various submitted outputs from various models, but now it was change into highest accuracy (99.8%) output, as its accuracy is even higher than the observe one we generate (use cross-evaluation). By comparison we could get another local accuracy, if the local accuracy is higher than record we would output the result prediction and model. And when it is after thousands epoch, it would also output prediction and model no matter what, we would also output the record of local accuracy and loss in order to observe the change of model in the whole training process.

(support program: nnchange.py)

```

1 import torch
2 import torch.nn as nn
3 import torch.utils.data as data
4 import torchvision.transforms as transforms
5 from torch.autograd import Variable
6 import matplotlib.pyplot as plt
7 import torch.nn.functional as F
8 import numpy as np
9 import pandas as pd
10 import csv
11 import copy as copy
12
13 num = 27455
14 test_size = 7172
15 train_label = np.zeros((7172, 25))
16 csvcount = 151 #how many submissions to generate one observation usage data
17
18 oacc = [0.82043, 0.82691, 0.88933, 0.78083, 0.83542, 0.71025, 0.81345, 0.88578, 0.83403, 0.88613, 0.89225, 0.76394, 0.88892, 0.79672, 0.89539, 0.76569, 0.95746, 0.94638,
19         0.91084, 0.90167, 0.90167, 0.75139, 0.82217, 0.827029, 0.96164, 0.78033, 0.95222, 0.96822, 0.96549, 0.97489, 0.96896, 0.96199, 0.96722, 0.97289, 0.97245, 0.97871,
20         0.95771, 0.96896, 0.92115, 0.95711, 0.91215, 0.93770, 0.91019, 0.95815, 0.91657, 0.83954, 0.83926, 0.83815, 0.79457, 0.94735, 0.93479, 0.96361, 0.98221, 0.98779,
21         0.99476, 0.98925, 0.94979, 0.92503, 0.97315, 0.94281, 0.96164, 0.96443, 0.96582, 0.98326, 0.98221, 0.97803, 0.96465, 0.97873, 0.98438, 0.96465, 0.96465, 0.98535,
22         0.99058, 0.98779, 0.98570, 0.98186, 0.98847, 0.97556, 0.98865, 0.98744, 0.97768, 0.98849, 0.98117, 0.98865, 0.96617, 0.97175, 0.96881, 0.97768, 0.97842, 0.98326,
23         0.99267, 0.99258, 0.95815, 0.96722, 0.95900, 0.96896, 0.96896, 0.97245, 0.97723, 0.98087, 0.93981, 0.97519, 0.98082, 0.96921, 0.97429, 0.97350, 0.93540, 0.93667,
24         0.99225, 0.99085, 0.95327, 0.96931, 0.97977, 0.98012, 0.98117, 0.98500, 0.97838, 0.98500, 0.97768, 0.97315, 0.96582, 0.96443, 0.98256, 0.97803, 0.97594, 0.98186,
25         0.98221, 0.98291, 0.96025, 0.98291, 0.98326, 0.97873, 0.96827, 0.98500, 0.98256, 0.98152, 0.98012, 0.98152, 0.98779, 0.95955, 0.93165, 0.96164, 0.98152, 0.94665,
26         0.98478, 0.96617, 0.96757, 0.95990, 0.96199, 0.95936, 0.95487] #evaluation accuracy given by graded system
27
28 guesslabel = np.zeros((7172))
29 print(guesslabel.shape)
30
31 csvs = ['nnhd410421304(30).csv'%(n) for n in range(1, csvcount+1)]
32
33 for n in range(csvcount): #each csv
34     pd_data = pd.read_csv(csvs[n]).values
35     label = pd_data[:,12] #new cases
36     for m in range(test_size):
37         lab = label[m] #get the predict label
38         train_label[n][lab] += oacc[n] #vote for that label with its weight
39
40 tlist = train_label.tolist() #change data format
41 for o in range(test_size):
42     guesslabel[o] = np.argmax(tlist[o]) #then each label, observe which prediction got more votes, then set output as that prediction
43
44 output = pd.DataFrame({"samp_id": range(1,len(guesslabel)+1), "label": guesslabel})
45 output.to_csv("acc.csv", columns=["samp_id", "label"], index=False) #output an observation usage prediction

```

(Fig of nnechange.py. The accuracies were acquired from submit.)

(1 to 29)

This program is use to generate the voting result from numerous submits. We need to give each submit a voting weight, which this weight is same as their result accuracy, total 151 submits. We rename these submit by their order of submit in order to sequentially read in these data. (Note: all these submit were same as the one I submit to Kaggle)

(31 to 36)

Now for these inputs, we deal with them one by one. First, we found the prediction of this submit for one sample. Then we let this submit vote for this prediction for this sample. For example, submit 1's prediction of sample 1 is 20, then it will vote 20 for sample 1. After this step we complete create a voting pool.

(38 to 40)

Now for each sample, we are going to find out which prediction got most vote, then set up this as the prediction of the output observe result. For example, for sample 1, 10 of submits vote for prediction 10, 141 of submits vote for prediction 20, then the output prediction would be set as 20. Then finally we output the result,

named its as acc.py (line42,43). (Note that this observe data is not used later since we have already got a high enough accuracy of submit, we use that as observe data from that timepoint.

(Support program: acccheck.py)

```
1 import torch
2 import torch.nn as nn
3 import torch.utils.data as data
4 import torchvision.transforms as transforms
5 from torch.autograd import Variable
6 import matplotlib.pyplot as plt
7 import torch.nn.functional as F
8 import numpy as np
9 import pandas as pd
10 import csv
11 import copy as copy
12 import time
13 start_time = time.time()
14
15 num = 27455
16 test_size = 7172
17 train_label = np.zeros((7172, 25))
18 csvcount = 24
19
20 guesslabel = np.zeros(7172)
21 print(guesslabel.shape)
22
23 acc = pd.read_csv('acc.csv').values #read observation usage prection data
24 testcsv = pd.read_csv('nnhw1410421304.csv').values #read our desire-submit data to check
25 acclabel = acc[:,1:2] #second row is label
26 testlabel = testcsv[:,1:2]
27
28 dif = acclabel - testlabel
29 print(np.count_nonzero(dif == 0)/ 7172) #show accuracy
```

(fig of acccheck.py. This had later been implemented in main program, so only use for double check.)

(1 to 26)

This program is simple. First, we import library and initialize some empty arrays for later storage as well, then in 23 to 26, we read in our observe data and our desire submit data. Strip labels out and store in arrays.

(28 to 29)

We do division between these two arrays. If the result is zero it's means theirs predict were the same, otherwise different. By counting zeros and divide by length of array we would know the local accuracy (Our own predict accuracy). (Note that since we had later implemented this part in the main program for observation, this is only use for double check.)

Note: For detail comment check the code and note inside folder.

Discussion of the result:

In this assignment, I had learned to use some additional tactics to help me to adjust the model. Mostly the optimal performance is come from try and error, and since model training required a lot of times, it becomes more difficult to do these “tries”. So, I think “why not let these data be useful?” and design another model to help me to adjust the model, and I found that this is a good idea, no matter the result is not good enough, it is not too bad at least, like ~90% of accuracy. Its better than just blind-guess which kind of parameter would give me a best result: if it made the result better, that’s good; if it’s not, at least I learn another way to solve the problem for myself. And also, since the limitation of the device, it takes almost 1 day to run around 5000 epochs of a 5 layers model, so it is a waste if you did not save the observation in this training process, so I let the training unstopped, only stop when I observer that this model is dead (not glowing, or worse), and save model during save point for better observation. By this process it helps my models’ accuracy become higher, as I can know performance of each part of training process, stop the model if it had a bad start, also knows that each model would finally reach its limitation and stop growing.

nnhw1410421304(37000).csv 3 days ago by 410421304 0607 stop growing	0.98152	<input type="checkbox"/>
nnhw1410421304(36000).csv 3 days ago by 410421304 0607	0.98152	<input type="checkbox"/>
nnhw1410421304(35000).csv 3 days ago by 410421304 0607	0.98152	<input type="checkbox"/>
nnhw1410421304(34000).csv 3 days ago by 410421304 0607	0.98012	<input type="checkbox"/>
nnhw1410421304(33000).csv 3 days ago by 410421304 0607	0.98152	<input type="checkbox"/>

(fig of example of model dead: as can see above, the model merely adjusted for 5000 epochs (from 33000 ~ 37000). This happens when we go deeper)

Comparing each model, as I had said before, due to the limitation of device, it is hard for me to try the model which contains to many layers, though I had implemented them as well, but they are the simple-simple version of the original design, and they are not good enough as well. So, I go for a traditional way, see how alexnet set up its parameter, try this parameter and start to adjust from this. I had found that first, loss is important as it means the way the model goes: lower better. Second is accuracy, don’t stop training until it is stable 100% accuracy (of the model),

or the result may not be good enough for using. Try to lower the learning rate if loss did not become lower; if the learning speed is too slow for a model, try to increase it. If also not work, try to print out the shape of the model's output. If it is too small, which is neuron counts too small, may lead to the model become hard to learn.

Deeper did not guarantee a better result. Surely 2-layers CNN is better than 1-layer CNN, but as the layer increase, this rule may not be as useful as before, if the model is too complicate for the problem, it may learn some un-need stuff as the layer grows, which may lead to poor outcome. And after several loops, the model should reach its final state, which it may stop learning (or very slow), usually base on my experience this is not the best result of a model, just local minima. My additional observation tactic can help me to verify where the model stopped learning.

As a result, the best score I had acquired is accuracy around 99.8%. The accuracy acquired by blind guess is 70~89%, plain Alexnet is around 95%, plain VGG around 90%, plain Resnet (with lower layers) 82~87%, plain Googlenet around 91~94%. Its really hard to compare which kind of network is good, because it may need to consider the situation, dataset, device, and many other conditions. But common rules are: Don't go too shallow; Don't go too deep. Don't set learning rate too high or too low. Don't set epochs too high or too low. Standard of "too" is depend on your conditions of model, which means you need to try, and the first thing is try to understand how does parameters do it's part.

So, these are my experience of training a CNN model for recognition. Surely luck could take an important part of the training, but experience and auto training can save us a lot of time from the impossible outcomes. I had originally learned how to use CNN before, and now I feel like I had learned more adjusting strategy, and other tactics of support training, and also some skills of how to deal with the data's format, which may be helpful for myself if I am going to use neural network or image recognition.

Appendix:

Day1~ Day2: Self build network, 2-layers CNN, acc 70~82%

```
class Network(nn.Module):
    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, 3, 1)
        self.conv2 = nn.Conv2d(hid_size1, hid_size2, 5, 1)
        self.linear1 = nn.Linear(hid_size2, num_classes)
        self.pool = nn.MaxPool2d(2, 2)
        self.norm = nn.BatchNorm2d(32)

    def forward(self, x):
        hid_out1 = self.pool((F.relu(self.conv1(x))))
        hid_out2 = self.pool((F.relu(self.conv2(hid_out1))))
        hid_out3 = self.pool((F.relu(self.conv3(hid_out2))))
        fcinput1 = hid_out3.view(x.size(0), -1)
        out = self.linear1(fcinput1)
        #out = self.linear2(fcinput1)
        prob = F.softmax(out, dim=1)
        return out
```

(sample of self-build CNN)

Day3: Self build network, 3-layers CNN, acc 71~83%

Day4~ Day7: Self build network, 2-layers CNN, acc 71~89%

Day8~ Day9: Self build network, 6-layers CNN, acc ~79%

```
class Network(nn.Module):
    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 = 1)
        self.conv2 = nn.Conv2d(hid_size1, hid_size2, kernel_size = 5, stride = 1, padding = 1)
        self.conv3 = nn.Conv2d(hid_size2, hid_size3, kernel_size = 3, padding = 1)
        self.conv4 = nn.Conv2d(hid_size3, hid_size4, kernel_size = 3, padding = 1)
        self.conv5 = nn.Conv2d(hid_size4, hid_size5, kernel_size = 3, padding = 1)
        self.linear1 = nn.Linear(hid_size5, hid_size7)
        self.linear2 = nn.Linear(hid_size7, hid_size7)
        self.linear3 = nn.Linear(hid_size7, num_classes)
        self.pool = nn.MaxPool2d(3, 2)
        self.norm = nn.BatchNorm2d(32)
        self.d = nn.Dropout()

    def forward(self, x):
        hid_out1 = self.pool((F.relu(self.conv1(x))))
        hid_out2 = self.pool((F.relu(self.conv2(hid_out1))))
        hid_out3 = F.relu(self.conv3(hid_out2))
        hid_out4 = F.relu(self.conv4(hid_out3))
        hid_out5 = self.pool((F.relu(self.conv5(hid_out4))))
        hid_out6 = self.pool((F.relu(self.conv6(hid_out5))))
        fcinput1 = hid_out6.view(x.size(0), -1)
        hid_out7 = self.linear1(F.relu(self.d(fcinput1)))
        hid_out8 = self.linear2(F.relu(self.d(hid_out7)))
        out = self.linear3(hid_out8)
        #out = self.linear2(fcinput1)
        prob = F.softmax(out, dim=1)
        return out
```

(sample of 6-layers self-build XNN)

Day10: AlexNet type, 5-layers CNN, acc 94~95%

```
class Network(nn.Module):
    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)
        self.conv2 = nn.Conv2d(hid_size1, hid_size2, kernel_size = 5, padding = 2)
        self.conv3 = nn.Conv2d(hid_size2, hid_size3, kernel_size = 3, padding = 1)
        self.conv4 = nn.Conv2d(hid_size3, hid_size4, kernel_size = 3, padding = 1)
        self.conv5 = nn.Conv2d(hid_size4, hid_size5, kernel_size = 3, padding = 1)
        self.linear1 = nn.Linear(hid_size5, hid_size7)
        self.linear2 = nn.Linear(hid_size7, num_classes)
        self.pool = nn.MaxPool2d(3, 2)
        self.norm = nn.BatchNorm2d(32)
        self.d = nn.Dropout(0.5)

    def forward(self, x):
        hid_out1 = self.pool((F.relu(self.conv1(x))))
        hid_out2 = self.pool((F.relu(self.conv2(hid_out1))))
        hid_out3 = F.relu(self.conv3(hid_out2))
        hid_out4 = F.relu(self.conv4(hid_out3))
        hid_out5 = self.pool((F.relu(self.conv5(hid_out4))))
        hid_out6 = self.pool((F.relu(self.conv6(hid_out5))))
        fcinput1 = hid_out6.view(x.size(0), -1)
        hid_out7 = self.linear1(F.relu(self.d(fcinput1)))
        hid_out8 = self.linear2(F.relu(self.d(hid_out7)))
        out = self.linear3(hid_out8)
        #out = self.linear2(fcinput1)
        prob = F.softmax(out, dim=1)
        return out
```

(sample of alex-like network. Basically, it's same as self-build, but imitate the structure)

Day11~12: VGG type, 6-layers CNN, acc 90~91%

```
10 class Net(nn.Module):
11     def __init__(self, input_size, hid_size1, hid_size2, hid_size3, hid_size4, num_classes):
12         super(Net, self).__init__()
13         self.conv1 = nn.Conv2d(input_size, hid_size1, kernel_size=3, padding=1)
14         self.conv2 = nn.Conv2d(hid_size1, hid_size2, kernel_size=3, padding=1)
15         self.conv3 = nn.Conv2d(hid_size2, hid_size3, kernel_size=3, padding=1)
16         self.conv4 = nn.Conv2d(hid_size3, hid_size3, kernel_size=3, padding=1)
17         self.conv5 = nn.Conv2d(hid_size3, hid_size4, kernel_size=1, padding=1)
18         self.conv6 = nn.Conv2d(hid_size4, hid_size4, kernel_size=1, padding=1)
19
20         self.linear1 = nn.Linear(hid_size4, hid_size7)
21         self.linear2 = nn.Linear(hid_size7, num_classes)
22         self.pool = nn.MaxPool2d(2, 2)
23         self.norm = nn.BatchNorm2d(32)
24         self.d = nn.Dropout()
25
26     def forward(self, x):
27         hid_out1 = self.pool(F.relu(self.conv1(x)))
28         hid_out2 = self.pool(F.relu(self.conv2(hid_out1)))
29         hid_out3 = F.relu(self.conv3(hid_out2))
30         hid_out4 = self.pool(F.relu(self.conv4(hid_out3)))
31         hid_out5 = F.relu(self.conv5(hid_out4))
32         hid_out6 = self.pool(F.relu(self.conv6(hid_out5)))
33         hid_out7 = F.relu(self.conv6(hid_out6))
34         hid_out8 = self.pool(F.relu(self.conv6(hid_out7)))
35         fcinput1 = hid_out8.view(x.size(0), -1)
36         lhid_out1 = F.relu(self.linear1(fcinput1))
37         out = self.linear2(lhid_out1)
38         prob = F.softmax(out, dim=1)
39         return out
```

(sample of VGG-like network. Basically, it's same as self-build, but imitate the structure)

Day13: ResNet type, 5 blocks, acc 82~87%

```
10 class Net(nn.Module):
11     def __init__(self, in_channel, num_classes, verbose=False):
12         super(Net, self).__init__()
13         self.verbose = verbose
14
15         self.block1 = nn.Conv2d(in_channel, hid_size1, 2, 2)
16
17         self.block2 = nn.Sequential(
18             nn.MaxPool2d(2, 2),
19             residual_block(hid_size1, hid_size1),
20             residual_block(hid_size1, hid_size1)
21         )
22
23         self.block3 = nn.Sequential(
24             residual_block(hid_size1, hid_size2, False),
25             residual_block(hid_size2, hid_size2)
26         )
27
28         self.block4 = nn.Sequential(
29             residual_block(hid_size2, hid_size3, False),
30             residual_block(hid_size3, hid_size3)
31         )
32
33         self.block5 = nn.Sequential(
34             residual_block(hid_size3, hid_size4, False),
35             residual_block(hid_size4, hid_size4),
36             nn.AvgPool2d(1)
37         )
38
39         self.classifier = nn.Linear(512, num_classes)
40
41     def forward(self, x):
42         x = self.block1(x)
43         x = self.block2(x)
44         x = self.block3(x)
45         x = self.block4(x)
46         x = self.block5(x)
47         x = x.view(x.size[0], -1)
48         x = self.classifier(x)
49         return x
```

(from default structure, but lower layers for device to capable to run for testing performance)

Day14~18: AlexNet type, 5-layers CNN, acc 91~97% (adjust params)

Day19: VGG type, 5-layers CNN, acc 91~92%

Day20: AlexNet type, 5-layers CNN, acc 95% (adjust params)

Day20: ResNet type, 5 blocks, acc 84%

Day20~21: GoogleNet type, 5 blocks, acc 91~94%

```
10 class Net(nn.Module):
11     def __init__(self, in_channel, num_classes, verbose=False):
12         super(Net, self).__init__()
13         self.verbose = verbose
14
15         self.block1 = nn.Sequential(
16             conv_relu(in_channel, out_channel=64, kernel=3, stride=1, padding=1),
17             #nn.MaxPool2d(2, 2)
18         )
19
20         self.block2 = nn.Sequential(
21             #conv_relu(64, 64, kernel=1),
22             conv_relu(64, 192, kernel=3, padding=1),
23             nn.MaxPool2d(3, 2)
24         )
25
26         self.block3 = nn.Sequential(
27             inception(192, 64, 96, 128, 16, 32, 32),
28             #inception(192, 128, 128, 192, 16, 96, 64),
29             nn.MaxPool2d(3, 2)
30         )
31
32         self.block4 = nn.Sequential(
33             inception(256, 128, 128, 192, 192, 32, 96, 64),
34             #inception(256, 128, 128, 192, 128, 16, 96, 64),
35             #inception(512, 192, 128, 128, 256, 24, 64, 64),
36             #inception(512, 128, 192, 192, 128, 32, 64, 64),
37             #inception(512, 256, 192, 128, 128, 32, 128, 128),
38             nn.MaxPool2d(3, 2)
39         )
40
41         self.block5 = nn.Sequential(
42             #inception(512, 256, 192, 128, 128, 32, 128, 128),
43             #inception(512, 384, 192, 192, 128, 48, 128, 128),
44             #nn.AvgPool2d(2)
45         )
```

(from default structure, but lower layers for device to capable to run for testing performance)

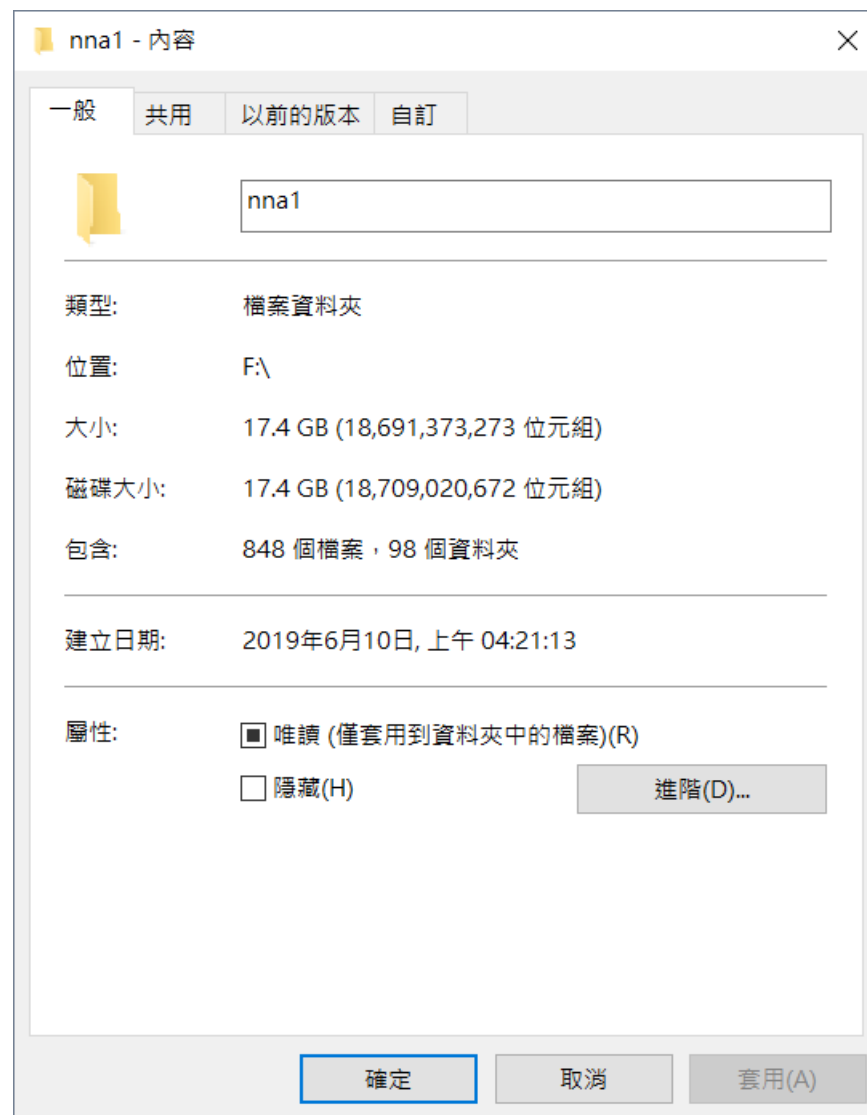
Day21: Self build network, 5-layers CNN, acc ~95%

Day22~28: AlexNet type, 5-layers CNN, acc 96~99% (adjust params, auto save)

Day28: GoogleNet type, 5 blocks, acc 90~93%

Day28: GoogleNet type, 5 blocks, acc 93% (auto save)

Day29~37: AlexNet type, 5-layers CNN, acc 95~98% (adjust params, auto save, infinite loop)



(total size of all model saves)

4	410421304		0.99825	178	13h
Your Best Entry ↑					
Your submission scored 0.96687, which is not an improvement of your best score. Keep trying!					

(current score)

```

logconv) P:\nnhmlfinal>python nnhmlfinal.py
(27455, 1, 28, 28)
Epoch: [3/10], Loss: 3.0610, Accuracy: 11.00
0.598600509202454
Net saved.
Epoch: [2/10], Loss: 2.8726, Accuracy: 37.00
0.2826268822013383
Net saved.
Epoch: [3/10], Loss: 2.5764, Accuracy: 33.00
0.3289180145008366
Net saved.
Epoch: [4/10], Loss: 2.2015, Accuracy: 43.00
0.409230402119333
Net saved.
Epoch: [5/10], Loss: 1.8480, Accuracy: 52.00
0.447713206151013
Net saved.
Epoch: [6/10], Loss: 1.5467, Accuracy: 59.00
0.4991634132758427
Net saved.
Epoch: [7/10], Loss: 1.3032, Accuracy: 69.00
0.5488000923591746
Net saved.
Epoch: [8/10], Loss: 1.1093, Accuracy: 75.00
0.6090351366424986
Net saved.
Epoch: [9/10], Loss: 0.9547, Accuracy: 79.00
0.663836125959004
Net saved.
Epoch: [10/10], Loss: 0.8170, Accuracy: 83.00
0.693251133742313
Net saved.
Epoch: [11/10], Loss: 0.7003, Accuracy: 85.00
0.7267150027886224
Net saved.
Epoch: [12/10], Loss: 0.6012, Accuracy: 88.00
0.7520914668153932
Net saved.
Epoch: [13/10], Loss: 0.5150, Accuracy: 90.00
0.774930170665921
Net saved.
Epoch: [14/10], Loss: 0.4433, Accuracy: 92.00
0.7923870691919687
Net saved.
Epoch: [15/10], Loss: 0.3847, Accuracy: 94.00
0.804238706971969

```

(Sample running screen)

camp_id	loss	acc
1	3.061034	0.059816
2	2.872808	0.262627
3	2.576427	0.339918
4	2.201453	0.40923
5	1.848019	0.447713
6	1.546734	0.499163
7	1.303240	0.548801
8	1.109305	0.609035
9	0.954727	0.663832
10	0.817001	0.693252
11	0.700319	0.726715
12	0.601188	0.752091
13	0.514974	0.774958
14	0.443086	0.792387
15	0.384734	0.804239
16	0.337013	0.820552
17	0.296648	0.834077
18	0.259615	0.845929
19	0.226974	0.861684
20	0.199496	0.87507
21	0.176973	0.88232
22	0.158462	0.889292
23	0.142391	0.896124

(Sample observe data. Useful for observing accuracy change.)