CS280 2018 Fall Midterm Project
Striving for Simplicity: the All Convolution Net

Zhijuan Hu
CS280
SIST, ShanghaiTech
huzhj@shanghaitech.edu.cn

Abstract

This project can be divided two parts. The first mainly based on the paper, Striving for Simplicity: the All Convolution Net[2], but what I want to do is to explore the thought of using a convolutional layer with increased stride to replace a max-pooling layer and make experiments to compare the performance of different model not just train the net to get great accuracy by using the methods paper have pointed out. Hence for the controlled experiment I try to make the hyper-parameters same and find out using convolutional layer with increased layer can increase the accuracy. The second part use transfer learning to train the model and find out that when the dataset is similar with original dataset which pre-trained the model, it will convergence quickly and get better performance. Meanwhile, this project explore only using the first step of transfer learning, by training the final FC layer, will make the model get a high accuracy in the beginning, but will stuck after several epoch. It is necessary to train the whole model. Finally, it finds out transfer learning is a good way to train model when the dataset is small and it will get overfitting if train the model directly on the small dataset.

1. Introduction

This project mainly implement ALL Convolutional Network and reproduce experiment results on CIFAR-10 and using transfer learning on CIFAR-100 datasets.

2. Method

2.1. The all convolutional network

First recall the standard CNNs and pooling layer to get the intuitive of the two method. Let f denote a feature map produced by some layer of a CNN. It canbe described as a 3-dimensional array of size $W \times H \times N$ where W and H are the width and height and N is the number of channels (in case f is the output of a convolutional layer, N is the number of filters in this layer). Then p-norm subsampling (or pooling) with pooling size k (or half-length k/2) and stride r applied to the feature map f is a 3-dimensional array $s(f)$ with the following entries:

$$s_{i,j,u}(f) = \left\{ \sum_{h=-\lfloor k/2 \rfloor}^{\lfloor k/2 \rfloor} \sum_{w=-\lfloor k/2 \rfloor}^{\lfloor k/2 \rfloor} |f_{g(h,w,i,j,u)|^p} \right\}^{1/p} \quad (1)$$

where $g(h, w, i, j, u) = (r \cdot i + h, r \cdot j + w, u)$ is the function mapping from positions in s to positions in f respecting the stride, p is the order of the p-norm (for $p \to \infty$, it becomes the commonly used max pooling). If $r > k$, pooling regions do not overlap; however, current CNN architectures typically include overlapping pooling with $k = 3$ and $r = 2$. Let us now compare the pooling operation defined by Eq. 1 to the standard definition of a convolutional layer c applied to feature map f given as:

$$s_{i,j,o}(f) = \sigma \left\{ \sum_{h=-\lfloor k/2 \rfloor}^{\lfloor k/2 \rfloor} \sum_{w=-\lfloor k/2 \rfloor}^{\lfloor k/2 \rfloor} \sum_{u=1}^{N} \theta_{h,w,u,o} \cdot f_{g(h,w,i,j,u)} \right\} \quad (2)$$

where θ are the convolutional weights (or the kernel weights, or filters), $\sigma(\cdot)$ is the activation function, typically a rectified linear activation ReLU $\sigma(x) = \max(x, 0)$, and $o \in [1, M]$ is the number of output feature (or channel) of the convolutional layer.

While a complete answer of this question is not easy to give (see the experiments and discussion for further details and remarks) we assume that in general there exist three possible explanations why pooling can help in CNNs: 1) the p-norm makes the representation in a CNN more invariant; 2) the spatial dimensionality reduction performed by pooling makes covering larger parts of the input in higher layers possible; 3) the feature-wise nature of the pooling operation (as opposed to a convolutional layer where features get mixed) could make optimization easier. Assuming that only the second part the dimensionality reduction performed by pooling is crucial for achieving good performance with CNNs (a hypothesis that we later test in our experiments) one can now easily see that pooling can be removed from a network without abandoning the spatial dimensionality reduction by two means:

- We can remove each pooling layer and increase the stride of the convolutional layer that preceded it accordingly.
- We can replace the pooling layer by a normal convolution with stride larger than one (i.e. for a pooling layer with $k = 3$ and $r = 2$ we replace it with a convolution layer with corresponding stride and kernel size and number of output channels equal to the number of input channels).[2]

Hence, from the thought above and using the model paper provided to compare the performance between max-pooling and convolution layer. Here are the base network on CIFAR-10 and CIFAR-100.

2.2. Model description

The basenet model is the same as the description of the paper[2] in Table 1. And the variant model is the same as the Table 2. Here for the standard convolutional layer, the stride is 1, padding is the general padding (kernel size for 3 padding 1, 5 padding 2)to make the image size same with the inputs. For the covolutional layer which replace max pooling, the padding is none. Hence in order to make the last layer is 6×6, set the last 3×3 convolutional layer padding 3. Additional, the model ALL CONV A and ALL CONV B is in Table below. Dropout use to the input image as well as after each pooling layer (or after the layer replacing the pooling layer respectively). Much more detail can be found in Appendix.

Table 1. ALL CNN A

Input 32×32 RGB image
5×5 conv. 96 ReLU
3×3 conv. 96 ReLU, stride 2
5×5 conv. 192 ReLU
3×3 conv. 96 ReLU, stride 2
...

2.3. Transfer Learning

Here use the pre-trained model ALL-CNN-C trained on CIFAR-10 as a feature extractor, use transfer learning method to train a new network on class1 and class2 of CIFAR-100. Here are the transfer learning step:

Table 2. ALL CNN B

Input 32×32 RGB image
5×5 conv. 96 ReLU
1×1 conv. 96 ReLU
3×3 conv. 96 ReLU, stride 2
5×5 conv. 192 ReLU
1×1 conv. 96 ReLU
3×3 conv. 96 ReLU, stride 2
...

- use 1×1 kernel size convolutional layer to replace the FC layer and initialize it. Then frozen the parameters before the new layer, using backward and forward training on the last layer parameters for several epochs (here i set 50 epoch).
- Finally, train the whole model several epochs (default 50 epochs).

3. Experiments

The experiment environment is based on python 3.6.6, pytorch 0.4.1, GPU is 1080 GTX, NVIDIA-SMI 396.44, Driver Version: 396.44. CUDA Version 9.0.176, cudnn 7.1.4.

3.1. Experimental Setup

- Preprocess Datum
  While loading datum, using normalize, which mean is $(0.4914, 0.4822, 0.4465)$, variance is $(0.2023, 0.1994, 0.2010)$ and shuffle. For CIFAR-10, divide the datum into training set (49000), validation set (1000), testing set (10000). And while training, when using 1000×4 samples, validate on validation set. After training, test on testing set. For CIFAR-100, divide the datum into training set (5000), testing set (1000). And while training, when using 500×4 samples, check the accuracy on testing set.

- Weight initialization and hyper-parameters.
  The way to initialize the weights and bias use the method[1]. The hyper-parameters are learning rate and epoch. The way to change learning rate is the same as the paper [2]. The experiment set epoch is $[200, 250, 300]$, max epoch is 350. The learning rate can be found in Table. The run_num the subfile name used for saving and distinguishing running results.

3.2. Results

The training results is in the Table.

Table 3. CIFAR-10 resulting

Model	Accu	learning rate	run_num
BaseNet A	80.28	$[0.025, 0.01, 0.005, 0.001]$	3
ALL CNN A	80.05	$[0.025, 0.01, 0.005, 0.001]$	50
BaseNet B	80.97	$[0.1, 0.05, 0.01, 0.001]$	3
ALL CNN B	79.11	$[0.1, 0.05, 0.01, 0.001]$	50
BaseNet C	83.18	$[0.1, 0.05, 0.01, 0.005]$	1
Stride CNN C	80.78	$[0.01, 0.005, 0.001, 0.001]$	15
ConvPool CNN C	85.88	$[0.005, 0.001, 0.0005, 0.0001]$	15
ALL CNN C	83.28	$[0.005, 0.001, 0.0005, 0.0001]$	15

Table 4. CIFAR-100 transfer learning resulting

Model	Accu class1	Accu class2	learning rate	run_num	max epoch
ALL CNN C	68.75	68.45	$[0.025, 0.01, 0.005, 0.001]$	10	350
ALL CNN C Step1	52.10	38.65	$[0.025, 0.01, 0.005, 0.001]$	10	50
ALL CNN C Step2	75.05	75.1	$[0.025, 0.01, 0.005, 0.001]$	10	50

4. Limitation and Discussion

4.1. Comparison between Max-pooling with Convolution layer

- The though about using Convolutional layer replace Max-pooling layer, is derived except the points talked before. We can make use of the fact that if the image area covered by units in the topmost convolutional layer covers a portion of the image large enough to recognize its content (i.e. the object we want to recognize) then fully connected layers can also be replaced by simple 1-by-1 convolutions. From BaseNet A transfrom to BaseNet C (not change max-pooling layer, but using more convolutional layer), the accuracy from 80.28 increase to 80.97, then achieve to 83.18. Although the change is small, in some way it can still mean the convolution can improve model. We can also compare Stride CNN C and ALL CNN C to have the same conclusion.

- Only looking for BaseNet C and its variants. BaseNet C and ConvPool CNN C both use max-pooling and convolutional layer and both them get accuracy a little high. Stride CNN C is derived from BaseNet C by using the former convolutional layer (it equal to delete one layer) with stride 2. But ALL CNN C (only using convolutional layer with stride 2 to replace the max-pooling layer) can also get better, and ConvPool CNN C equal to add one layer. Hence to get the results is not so surprised.

4.2. Comparison with transfer learning

- Comparison on results of transfer learning step1 and step2.
  For first step, we can find only training the last layer can also get about 50 accuracy only using 10 epochs. And for second step, it final performance is better than train a new model on the class1 which is only 68.75.

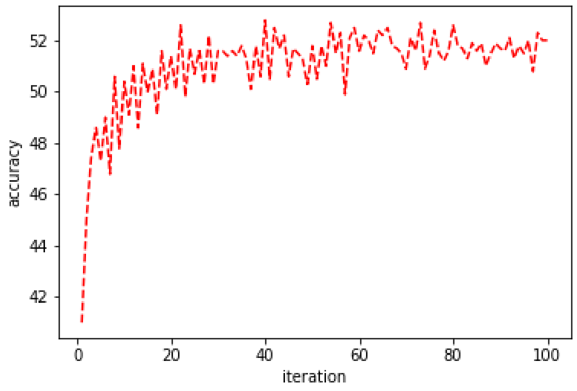


Figure 1. ALL CNN C step1 calss1 accuracy on testing dataset

- Compare transfer learning on dataset with different features Only looking ALL CNN C Step1 on class1 and class2, we can find the accuracy class1 is highly greater than the class2. It means the extraction features is not very useful for the new dataset. But finally its performance is still greater higher than the model training only on the small dataset class2. In some way, it means transfer learning which model train on big and with all kinds of variant image can have better performance and avoid overfitting.

4.3. Further Work

However, this project did not achieve the accuracy as high as the paper mentioned. It may because I try to use the same hyper-parameters, but some may not work on the

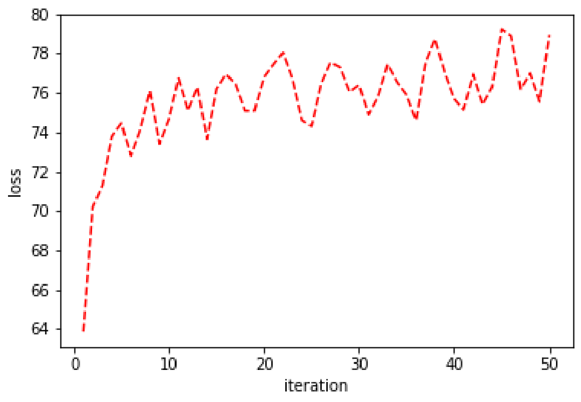


Figure 2. ALL CNN C step2 class1 accuracy on testing dataset

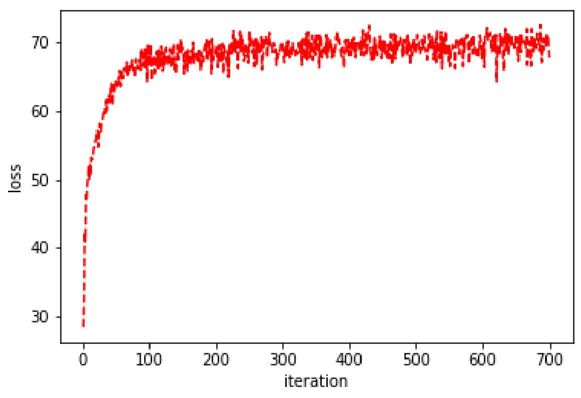


Figure 3. ALL CNN C class1 accuracy on testing dataset

model. Looking at the BaseNet A loss figure. We can find when epoch is 100-150, the loss will not decrease even later on decrease learning rate which epoch getting to 200, 250, 300. Hence if we want to make better performance, I think there are big space to take action. Also, for Plob3 Bonus, I only achieve ALL CNN A and ALL CNN B, but not using data augmentation. It accuracy is not very high. As for other transfer learning to use, we can not only change the last layer, but also to change more layers and add more power layer to get better performance. I think there are two ways to improve it, one is get better feature extracting(better network or better dataset), another is making a better mapping from feature to class (for example if we only use linear layer to replace the last layer, the performance is bad, but CNN is a nonlinear mapping which explain its performance better than linear layer). This thought, we can use more power nonlinear mapping such as Gaussian neuron network to replace the mapping.

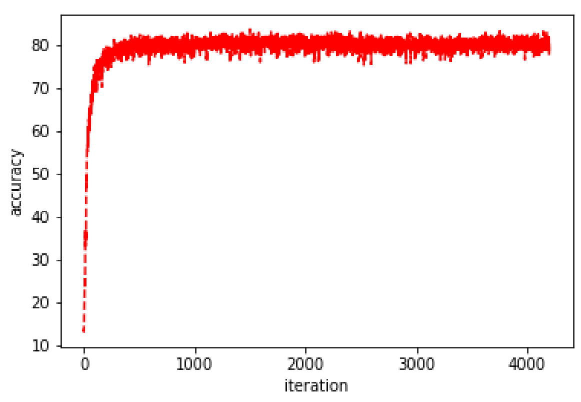


Figure 4. BaseNet A accuracy on testing dataset

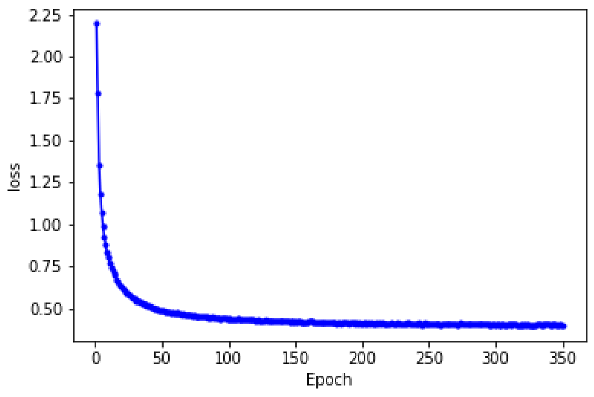


Figure 5. BaseNet A loss on testing dataset

References

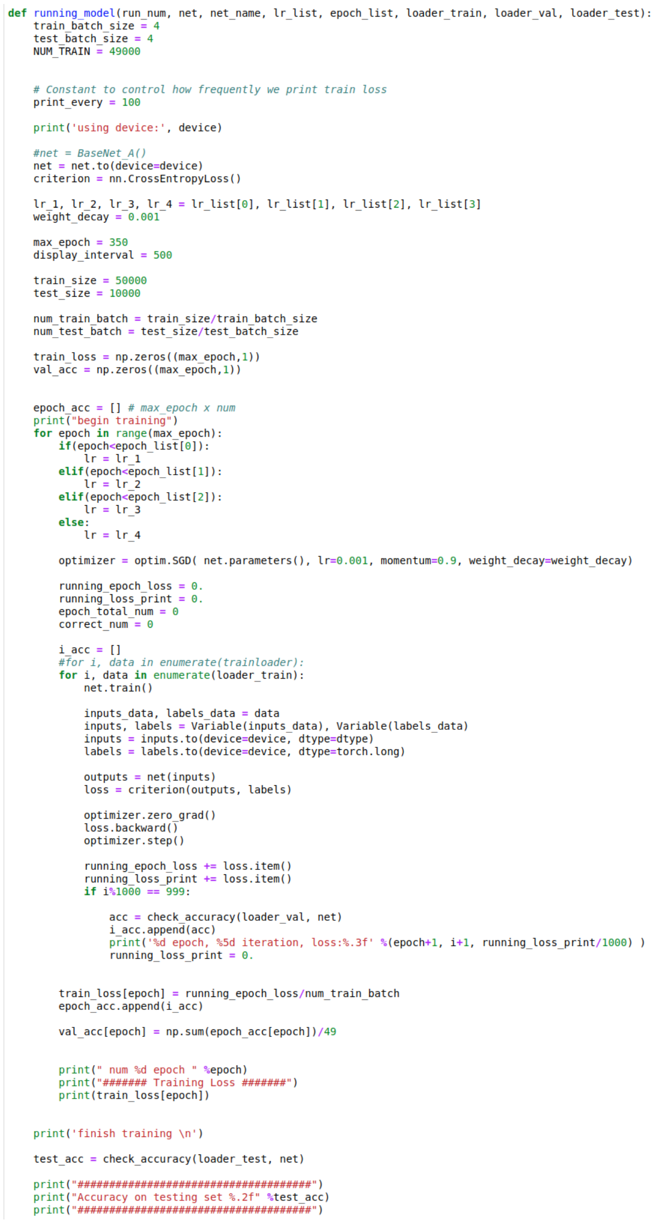
[1] K. He, X. Zhang, S. Ren, and J. Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *CoRR*, abs/1502.01852, 2015.

[2] J. T. Springenberg, A. Dosovitskiy, T. Brox, and M. A. Riedmiller. Striving for simplicity: The all convolutional net. *CoRR*, abs/1412.6806, 2014.

A. Appendix

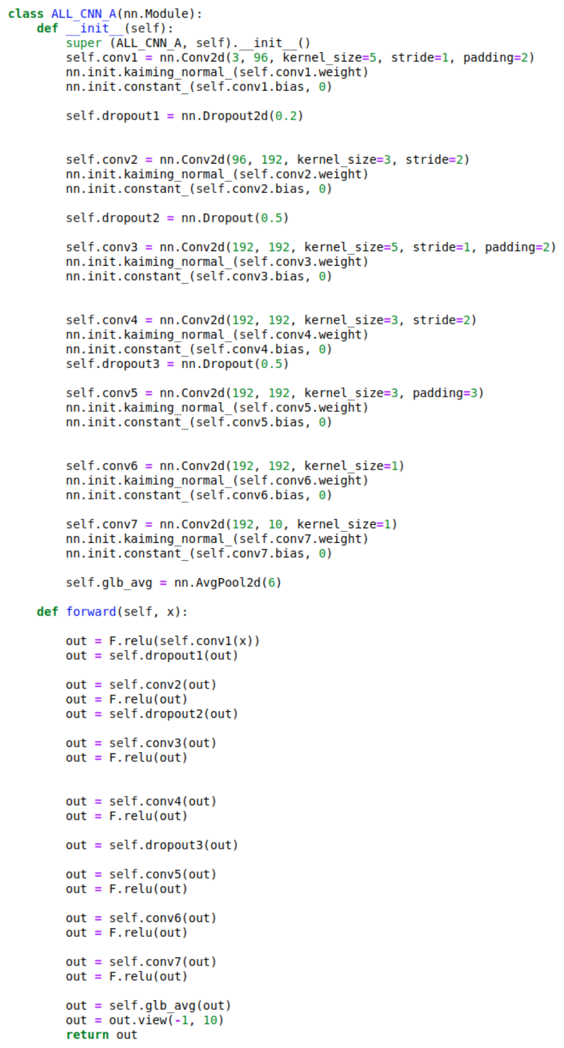
The code mainly parts are :

- Define Class for each network (all network in $Nets/new\_ALL\_Conv.py$); the function $running\_model$ to train the model and plot the accuracy, loss and save the datum (include parameters in model, .png, etc). See Figure 8 and Figure 9, 10. And for transfer learning the important thing is define optimzer, See Figure 14.



```
def running_model(run_num, net, net_name, lr_list, epoch_list, loader_train, loader_val, loader_test):
    train_batch_size = 4
    test_batch_size = 4
    NUM_TRAIN = 49000


    # Constant to control how frequently we print train loss
    print_every = 100

    print('using device:', device)

    #net = BaseNet_A()
    net = net.to(device=device)
    criterion = nn.CrossEntropyLoss()

    lr_1, lr_2, lr_3, lr_4 = lr_list[0], lr_list[1], lr_list[2], lr_list[3]
    weight_decay = 0.001

    max_epoch = 350
    display_interval = 500

    train_size = 50000
    test_size = 10000

    num_train_batch = train_size/train_batch_size
    num_test_batch = test_size/test_batch_size

    train_loss = np.zeros((max_epoch,1))
    val_acc = np.zeros((max_epoch,1))


    epoch_acc = [] # max_epoch x num
    print("begin training")
    for epoch in range(max_epoch):
        if(epoch<epoch_list[0]):
            lr = lr_1
        elif(epoch<epoch_list[1]):
            lr = lr_2
        elif(epoch<epoch_list[2]):
            lr = lr_3
        else:
            lr = lr_4

        optimizer = optim.SGD( net.parameters(), lr=0.001, momentum=0.9, weight_decay=weight_decay)

        running_epoch_loss = 0.
        running_loss_print = 0.
        epoch_total_num = 0
        correct_num = 0

        i_acc = []
        #for i, data in enumerate(trainloader):
        for i, data in enumerate(loader_train):
            net.train()

            inputs_data, labels_data = data
            inputs, labels = Variable(inputs_data), Variable(labels_data)
            inputs = inputs.to(device=device, dtype=dtype)
            labels = labels.to(device=device, dtype=torch.long)

            outputs = net(inputs)
            loss = criterion(outputs, labels)

            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            running_epoch_loss += loss.item()
            running_loss_print += loss.item()
            if i%1000 == 999:

                acc = check_accuracy(loader_val, net)
                i_acc.append(acc)
                print('%d epoch, %5d iteration, loss:%.3f' %(epoch+1, i+1, running_loss_print/1000) )
                running_loss_print = 0.


        train_loss[epoch] = running_epoch_loss/num_train_batch
        epoch_acc.append(i_acc)

        val_acc[epoch] = np.sum(epoch_acc[epoch])/49


        print(" num %d epoch " %epoch)
        print("####### Training Loss #######")
        print(train_loss[epoch])


    print('finish training \n')

    test_acc = check_accuracy(loader_test, net)

    print("####################################")
    print("Accuracy on testing set %.2f" %test_acc)
    print("####################################")
```

Figure 6. partly running model



```
class ALL_CNN_A(nn.Module):
    def __init__(self):
        super (ALL_CNN_A, self).__init__()
        self.conv1 = nn.Conv2d(3, 96, kernel_size=5, stride=1, padding=2)
        nn.init.kaiming_normal_(self.conv1.weight)
        nn.init.constant_(self.conv1.bias, 0)

        self.dropout1 = nn.Dropout2d(0.2)


        self.conv2 = nn.Conv2d(96, 192, kernel_size=3, stride=2)
        nn.init.kaiming_normal_(self.conv2.weight)
        nn.init.constant_(self.conv2.bias, 0)

        self.dropout2 = nn.Dropout(0.5)

        self.conv3 = nn.Conv2d(192, 192, kernel_size=5, stride=1, padding=2)
        nn.init.kaiming_normal_(self.conv3.weight)
        nn.init.constant_(self.conv3.bias, 0)


        self.conv4 = nn.Conv2d(192, 192, kernel_size=3, stride=2)
        nn.init.kaiming_normal_(self.conv4.weight)
        nn.init.constant_(self.conv4.bias, 0)
        self.dropout3 = nn.Dropout(0.5)

        self.conv5 = nn.Conv2d(192, 192, kernel_size=3, padding=3)
        nn.init.kaiming_normal_(self.conv5.weight)
        nn.init.constant_(self.conv5.bias, 0)


        self.conv6 = nn.Conv2d(192, 192, kernel_size=1)
        nn.init.kaiming_normal_(self.conv6.weight)
        nn.init.constant_(self.conv6.bias, 0)

        self.conv7 = nn.Conv2d(192, 10, kernel_size=1)
        nn.init.kaiming_normal_(self.conv7.weight)
        nn.init.constant_(self.conv7.bias, 0)

        self.glb_avg = nn.AvgPool2d(6)

    def forward(self, x):

        out = F.relu(self.conv1(x))
        out = self.dropout1(out)

        out = self.conv2(out)
        out = F.relu(out)
        out = self.dropout2(out)

        out = self.conv3(out)
        out = F.relu(out)


        out = self.conv4(out)
        out = F.relu(out)

        out = self.dropout3(out)

        out = self.conv5(out)
        out = F.relu(out)

        out = self.conv6(out)
        out = F.relu(out)

        out = self.conv7(out)
        out = F.relu(out)

        out = self.glb_avg(out)
        out = out.view(-1, 10)
        return out
```

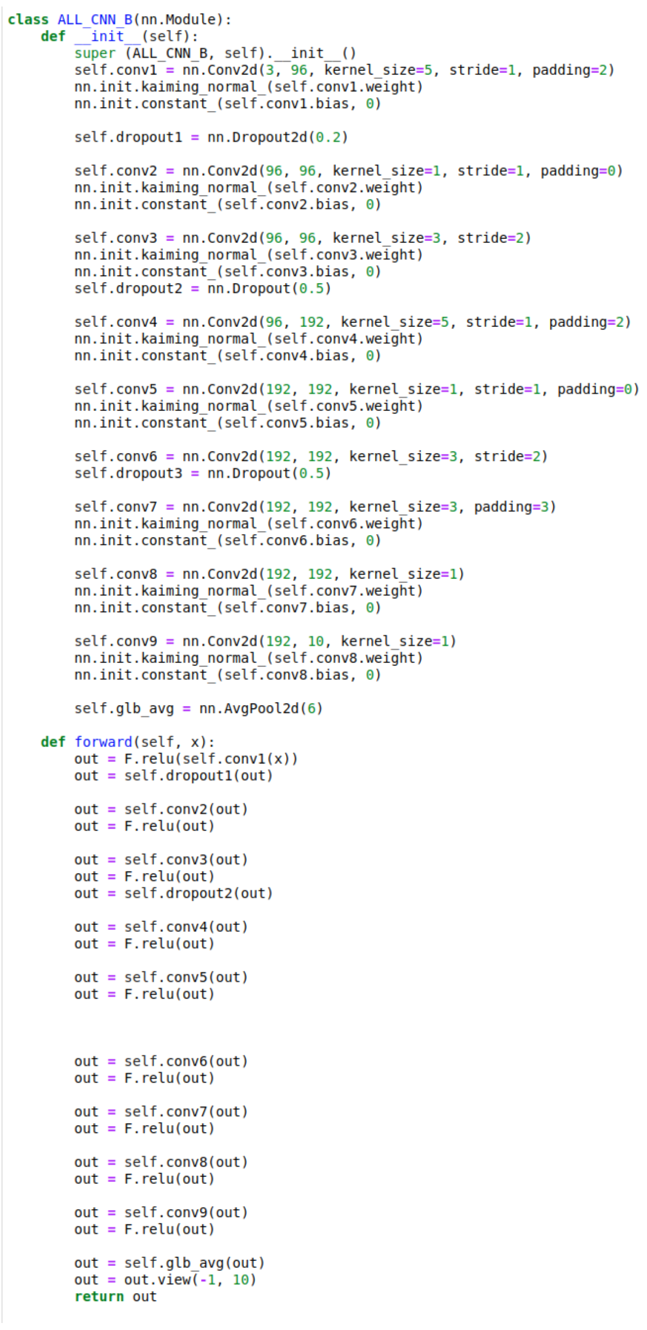
Figure 7. all cnn A

B. Appendix

When I run BaseNet B, the same learning rate with BaseNet A and B always make the loss be constant. Finally, i decrease the learning rate. The log is Figure 11. The wrong loss is Figure 12, after change the learning rate the right loss image is Figure 13.

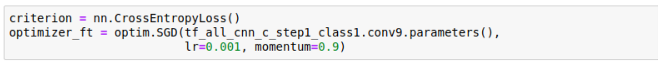
C. Appendix

All the curves path I have written in the report in run_num. You can use this information to find it.



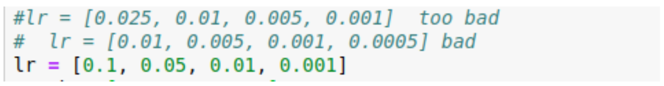
```
class ALL_CNN_B(nn.Module):
    def __init__(self):
        super (ALL_CNN_B, self).__init__()
        self.conv1 = nn.Conv2d(3, 96, kernel_size=5, stride=1, padding=2)
        nn.init.kaiming_normal_(self.conv1.weight)
        nn.init.constant_(self.conv1.bias, 0)

        self.dropout1 = nn.Dropout2d(0.2)

        self.conv2 = nn.Conv2d(96, 96, kernel_size=1, stride=1, padding=0)
        nn.init.kaiming_normal_(self.conv2.weight)
        nn.init.constant_(self.conv2.bias, 0)

        self.conv3 = nn.Conv2d(96, 96, kernel_size=3, stride=2)
        nn.init.kaiming_normal_(self.conv3.weight)
        nn.init.constant_(self.conv3.bias, 0)
        self.dropout2 = nn.Dropout(0.5)

        self.conv4 = nn.Conv2d(96, 192, kernel_size=5, stride=1, padding=2)
        nn.init.kaiming_normal_(self.conv4.weight)
        nn.init.constant_(self.conv4.bias, 0)

        self.conv5 = nn.Conv2d(192, 192, kernel_size=1, stride=1, padding=0)
        nn.init.kaiming_normal_(self.conv5.weight)
        nn.init.constant_(self.conv5.bias, 0)

        self.conv6 = nn.Conv2d(192, 192, kernel_size=3, stride=2)
        self.dropout3 = nn.Dropout(0.5)

        self.conv7 = nn.Conv2d(192, 192, kernel_size=3, padding=3)
        nn.init.kaiming_normal_(self.conv6.weight)
        nn.init.constant_(self.conv6.bias, 0)

        self.conv8 = nn.Conv2d(192, 192, kernel_size=1)
        nn.init.kaiming_normal_(self.conv7.weight)
        nn.init.constant_(self.conv7.bias, 0)

        self.conv9 = nn.Conv2d(192, 10, kernel_size=1)
        nn.init.kaiming_normal_(self.conv8.weight)
        nn.init.constant_(self.conv8.bias, 0)

        self.glb_avg = nn.AvgPool2d(6)

    def forward(self, x):
        out = F.relu(self.conv1(x))
        out = self.dropout1(out)

        out = self.conv2(out)
        out = F.relu(out)

        out = self.conv3(out)
        out = F.relu(out)
        out = self.dropout2(out)

        out = self.conv4(out)
        out = F.relu(out)

        out = self.conv5(out)
        out = F.relu(out)


        out = self.conv6(out)
        out = F.relu(out)

        out = self.conv7(out)
        out = F.relu(out)

        out = self.conv8(out)
        out = F.relu(out)

        out = self.conv9(out)
        out = F.relu(out)

        out = self.glb_avg(out)
        out = out.view(-1, 10)
        return out
```

Figure 8. all cnn B



```
criterion = nn.CrossEntropyLoss()
optimizer_ft = optim.SGD(tf_all_cnn_c_step1_class1.conv9.parameters(),
                         lr=0.001, momentum=0.9)
```

Figure 9.



```
#lr = [0.025, 0.01, 0.005, 0.001]  too bad
#  lr = [0.01, 0.005, 0.001, 0.0005] bad
lr = [0.1, 0.05, 0.01, 0.001]
```

Figure 10.

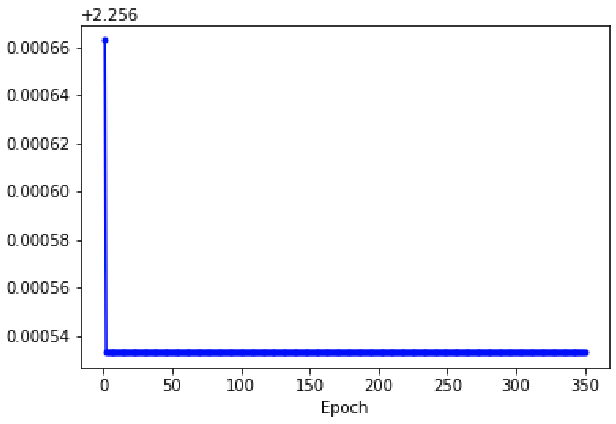


Figure 11.

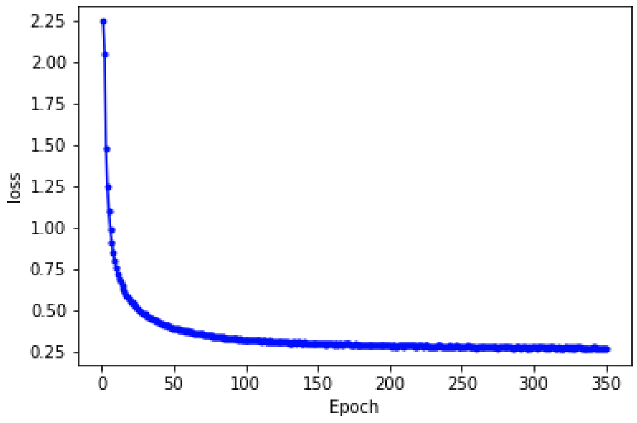


Figure 12.