ROB313 - TD - Deeplearning

Ye Liu, Xudong zhang

January 23, 2018

1 Original Test

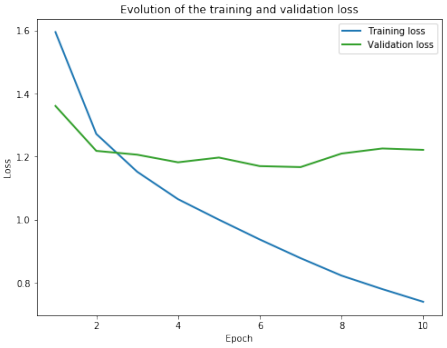
We have tested the CIFAR-10 data with the original structure of the Constitutional Neural Network (CNN). The structure of the CNN is:
Structure:
$\text{Input}(3 \times 32 \times 32) \xrightarrow{Conv1} \text{Feature Map}(18 \times 32 \times 32) \xrightarrow{Pool1} \text{Feature Map}(18 \times 16 \times 16)$
$\xrightarrow{Full\_Connection1} \text{Layer}(64) \xrightarrow{Full\_Connection2} \text{Output}(10)$
Parameter Setting:

```
Conv1: Conv2d(3, 18, kernel_size=3, stride=1, padding=1)
Pool1: MaxPool2d(kernel_size=2, stride=2, padding=0)
Full_Connection1: Linear(18 * 16 * 16, 64)
Full_Connection2: Linear(64, 10)
```

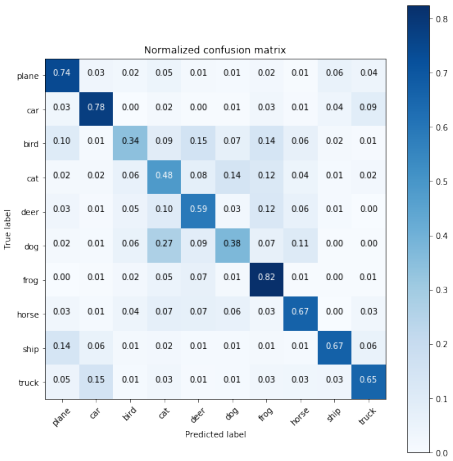
Training Setting:

```
n_training_samples = 20000
n_val_samples = 5000
n_test_samples = 5000
batch_size = 32
n_epochs = 10
learning_rate = 0.001
```

The result for the test is shown below:



1.1 Loss

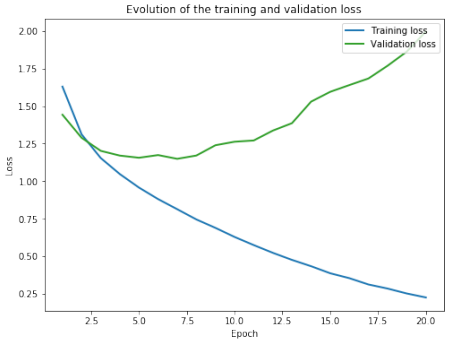


1.2 Confusion Matrix

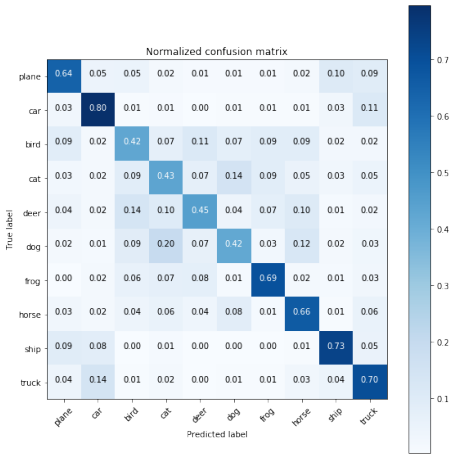
```
Accuracy of the network on the 20000 train images: 73.33 %
Accuracy of the network on the 5000 validation images: 59.64 %
Accuracy of the network on the 5000 test images: 60.16 %
```

2 Dropout

Afterwards, we tested the network with more epochs to see whether more training can reduce the loss further more. We test the CNN with 20 epochs, the result is shown below:



1.3 Loss Without Dropout



1.4 Confusion Matrix Without Dropout

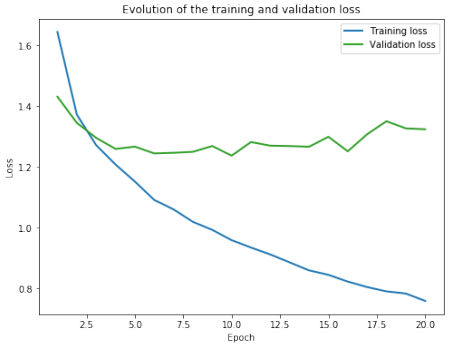
Here we can find that after about 8 epochs, the loss of the validation has grown up. The network has the problem "overfitting". One possible way for solving the problem is Dropout, so we introduced the technique Dropout and test the result. The new structure of the network is shown below:
Structure:

$$\text{Input}(3\times 32\times 32)\xrightarrow{Conv1}\text{Feature Map}(18\times 32\times 32)\xrightarrow{Pool1}\text{Feature Map}(18\times 16\times 16)$$
$$\xrightarrow{Dropout1}\text{Layer}(4608)\xrightarrow{Full\_Connection1}\text{Layer}(64)\xrightarrow{Full\_Connection2}\text{Output}(10)$$

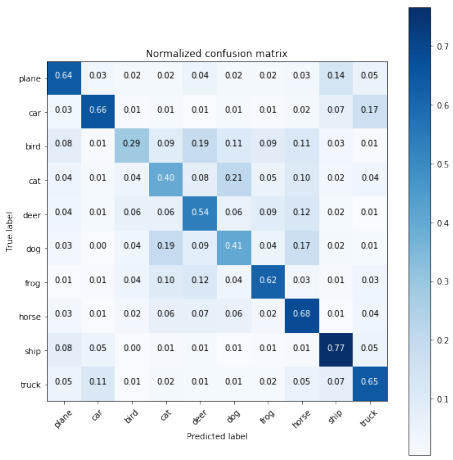
The parameter of dropout here is:

```
Dropout1: nn.Dropout(p=0.5)
```

Here are the results with dropout:



1.5 loss



1.6 Confusion Matrix

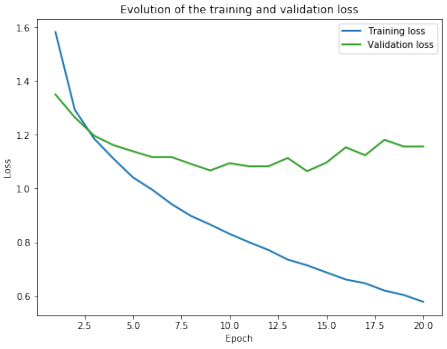
```
Accuracy of the network on the 20000 train images: 67.61 %
Accuracy of the network on the 5000 validation images: 55.90 %
Accuracy of the network on the 5000 test images: 56.70 %
```

We can find that the loss of the validation does not have a strong trend to go up. However, it does not goes down obviously.

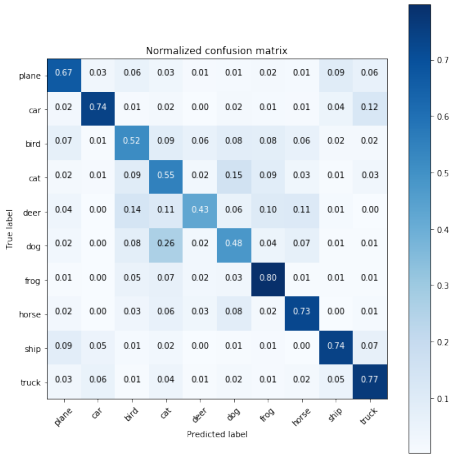
Here we think that the problem is due to the size of the training data. In fact, there are are too few training data, the complexity of the training data and the complexity of the network does not suit with each other. Even if the structure of the network is in fact really simple. So in next section, we plan to use the idea of data augmentation to see whether this can solve this problem.

3 Data Augmentation

Firstly, we try to augment the data by using Horizon Random Flip. With probability 0.5, we will Horizontally flip the given image. The test is done without dropout. The result is shown below:



1.7 loss



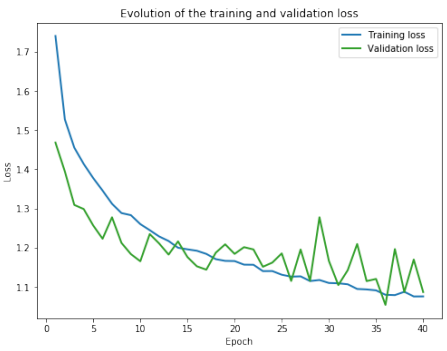
1.8 Confusion Matrix

```
Accuracy of the network on the 20000 train images: 77.72 %
Accuracy of the network on the 5000 validation images: 63.64 %
Accuracy of the network on the 5000 test images: 64.10 %
```

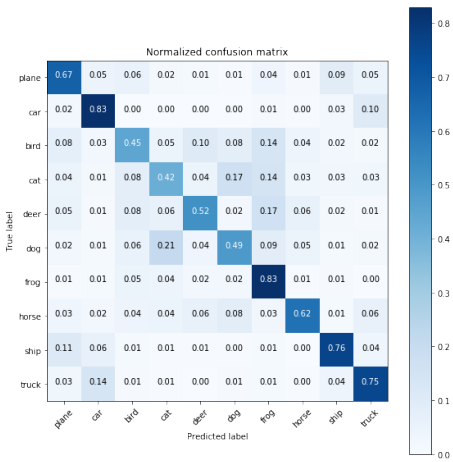
We can find that the loss of validation does not go up apparently after 10 epochs. The accuracy becomes much better. However, the old problem still exists: the loss of validation does not go down apparently after 10 epochs. This indicates that the data is still not big enough. So we add more augmentation strategies to see the result. This time, we add RandomResizedCrop. This action will randomly resize the image to the size of the original size and a random aspect ratio of the original aspect ratio. Here we use the parameter shown below:

```
RandomResizedCrop(32, scale=(0.5, 1.0), ratio=(0.75, 1.33))
```

The results after 40 epochs is shown below:



1.9 loss



1.10 Confusion Matrix

```
Accuracy of the network on the 20000 train images: 61.85 %
Accuracy of the network on the 5000 validation images: 64.48 %
Accuracy of the network on the 5000 test images: 63.38 %
```

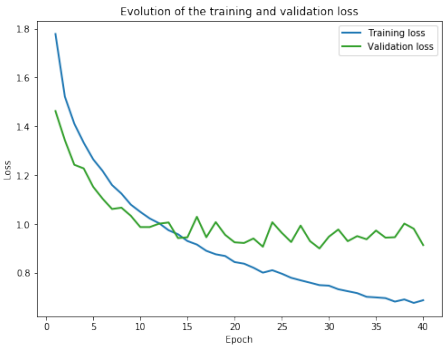
From the image of the loss, we can find that this time, the loss of validation and the loss of training is similar to each other. And at the end of the training, the loss of validation starts to vibrate. We think that it is due to the problem that now the network is too simple compared with the complexity of the data. So in the next section, we try to construct more complex structure and test the network based on data augmentation.

4 More complex network

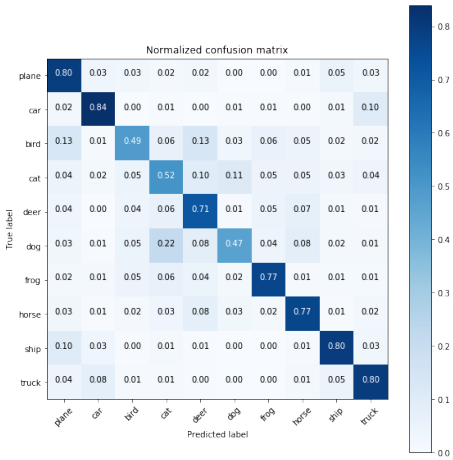
In this section, we construct a more complex network and test the performance based on the data augmentation. We show the structure of the network below:
Structure:
$\text{Input}(3 \times 32 \times 32) \xrightarrow{Conv1} \text{Feature Map}(18 \times 32 \times 32) \xrightarrow{Pool1} \text{Feature Map}(18 \times 16 \times 16)$
$\text{Feature Map}(18 \times 16 \times 16) \xrightarrow{Conv2} \text{Feature Map}(18 \times 16 \times 16) \xrightarrow{Pool2} \text{Feature Map}(18 \times 8 \times 8)$
$\xrightarrow{Full\_Connection1} \text{Layer}(576) \xrightarrow{Full\_Connection2} \text{Layer}(64) \xrightarrow{Full\_Connection3} \text{Output}(10)$
Parameter Setting:

```
Conv1: Conv2d(3, 18, kernel_size=3, stride=1, padding=1)
Pool1: MaxPool2d(kernel_size=2, stride=2, padding=0)
Conv2: Conv2d(3, 18, kernel_size=3, stride=1, padding=1)
Pool2: MaxPool2d(kernel_size=2, stride=2, padding=0)
Full_Connection1: Linear(18 * 8 * 8,576)
Full_Connection2: Linear(576, 64)
Full_Connection3: Linear(64, 10)
```

The run the network with 40 epochs and the result is shown below:



1.11 loss



1.12 Confusion Matrix

```
Accuracy of the network on the 20000 train images: 73.92 %
Accuracy of the network on the 5000 validation images: 70.52 %
Accuracy of the network on the 5000 test images: 69.48 %
```

Here we find that the result becomes much better. However, at the end of training, it still shows the trend of over-fitting: the validation loss does not go down along with the training loss and the vibration still exists.

For solving this problem, there are two ways, firstly, we can add more data augmentation to improve the complexity of the data. Another way is to tune the learning rate. The parameters are changed according to gradient algorithm. So at the end of the training, the learning rate may become too big so that the loss function will vibrate around the local minimum. In the next

section, we will test the idea of learning rate adaptation and improve the complexity of training data by adding random rotation.

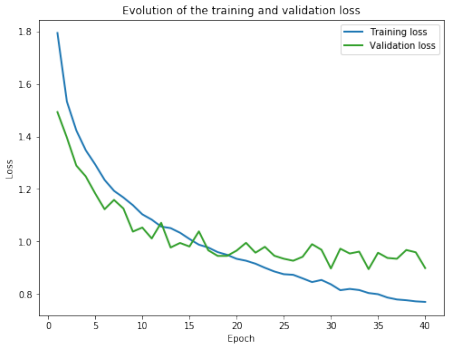
5 Final Optimization–Learning Rate Adaptation and Stronger Data Augmentation

Firstly, we use a stronger Data Augmentation by adding random rotation. The parameter that we use is shown below:

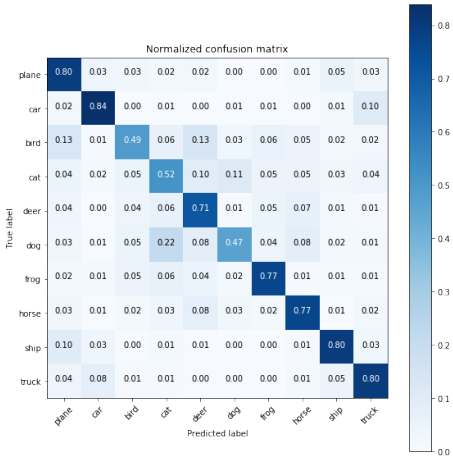
```
transforms.RandomRotation(10)
```

Here we want to rotate the original image with a random degree between $[-10, 10]$

After adding this for transformation, the result after 40 epochs is shown below:



1.13 loss

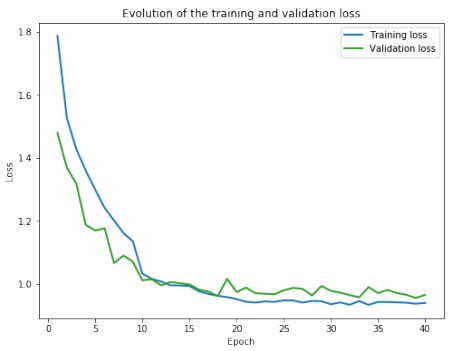


1.14 Confusion Matrix

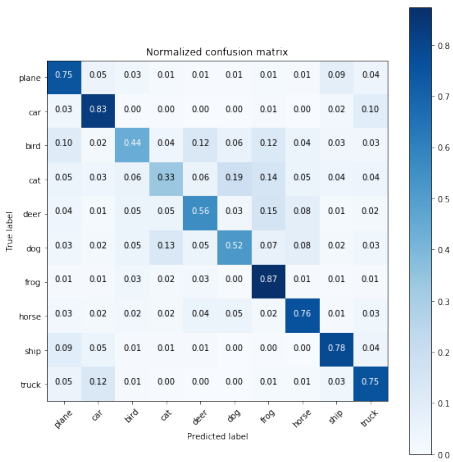
```
Accuracy of the network on the 20000 train images: 72.47 %
Accuracy of the network on the 5000 validation images: 70.28 %
Accuracy of the network on the 5000 test images: 69.78 %
```

The result becomes a little better, but still at the end of the training, the vibration exists and the validation loss stops to go down.

Now we try to add the technique of learning rate adaptation. There are many different methods for learning rate adaptation. Here we just use one simple method (Maybe the simplest): We initialize the learning rate with 0.001 and we decay the learning rate by multiplying the initial learning rate with 0.1. The result of the test is shown below:



1.15 loss



1.16 Confusion Matrix

```
Accuracy of the network on the 20000 train images: 67.12 %
Accuracy of the network on the 5000 validation images: 66.84 %
Accuracy of the network on the 5000 test images: 66.06 %
```

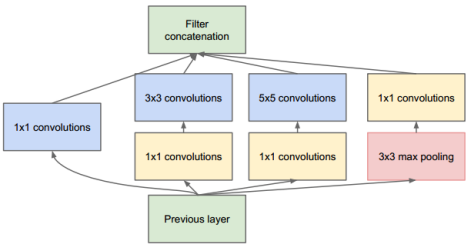
We can see from the image that the validation loss becomes more smooth. But the result becomes lower even with the similar validation loss at the end of training. This may due to the problem that decay rate is too high. However, we can find that at the end of training, we still can not find apparent descending trend. So the limit of the network is around 66%.

In all, the best accuracy that we get now on the test data is 69.78%.

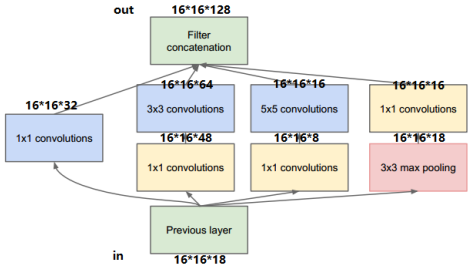
6 Deeper network

Inception network proposed by Google had great performance for ImageNet. It concatenates four outputs as one final output for each unit, so that the network can get the information of the squares of (1,1), (3,3), and (5,5) pixels (as shown in the figure below). And inception network does not use fully connected layer, which works more like a real human brain.

So we decided to introduce the framework to this TD. To better understand it and to test its performance. The picture in the right below is the first inception unit we had used. And we add another similar units, and between each two units we have an average pooling layer(the code can be found later). We tested the network with dropout and data augmentation.

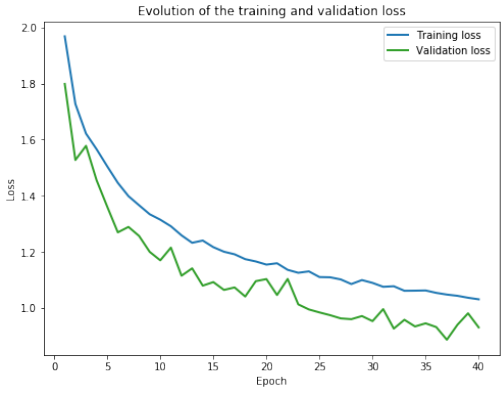


1.17 inception network

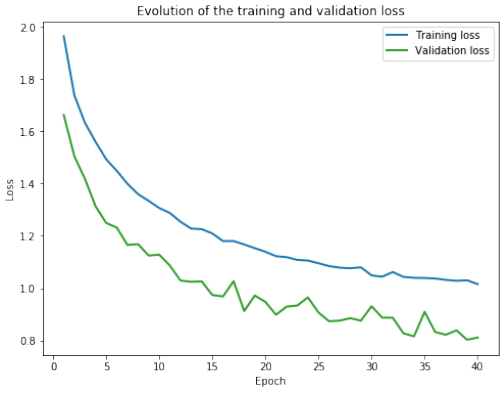


1.18 inception network

Firstly, I want the network to train fast, so we had done a RandomResizedCrop(24) in data augmentation section. And the result for 40 epochs is shown in the lift picture below (with test accuracy = 61.12%). The accuracy is disappointing, I thought the network can be better with a larger input: RandomResizedCrop(30). The test accuracy is 62.86%.



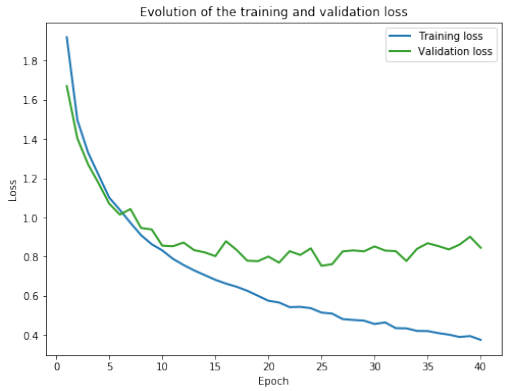
1.19 inception network input:24*24



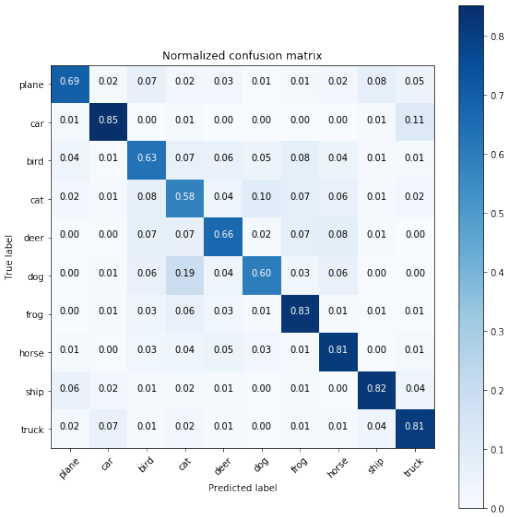
1.20 inception network input:30*30

However from the results we can see that, the validation loss was still decreasing (with vibration). It may require more epochs to get the best result. And to make the network converge faster, we reset the batch size as 128.

Finally, we got our best result: test accuracy 73.52 %.



1.21 inception network input:24*24



1.22 inception network input:30*30

```
Accuracy of the network on the 20000 train images: 83.42 %
Accuracy of the network on the 5000 validation images: 75.80 %
Accuracy of the network on the 5000 test images: 73.52 %
```

6.1 code

https://github.com/gggliuye/pytorch/blob/master/cifar10_inception_capsule.ipynb

```
class MyConvolutionalNetwork(nn.Module):
    def __init__(self):
        super(MyConvolutionalNetwork, self).__init__()
        self.conv1 = nn.Conv2d(3, 18, kernel_size=3, stride=1, padding=1)
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2, padding=0)

        #### START CODE: ADD NEW LAYERS ####
        #IN_V1
        self.v1pool1 = nn.MaxPool2d(kernel_size=3, stride=1, padding=1)
        self.v1conv1 = nn.Conv2d(18, 16, kernel_size=1, stride=1, padding=0)
        self.v1conv2 = nn.Conv2d(18, 8, kernel_size=1, stride=1, padding=0)
        self.v1conv22 = nn.Conv2d(8, 16, kernel_size=5, stride=1, padding=2)
        self.v1conv3 = nn.Conv2d(18, 48, kernel_size=1, stride=1, padding=0)
        self.v1conv32 = nn.Conv2d(48, 64, kernel_size=3, stride=1, padding=1)
        self.v1conv4 = nn.Conv2d(18, 32, kernel_size=1, stride=1, padding=0)
        self.pool2 = nn.MaxPool2d(kernel_size=3, stride=3, padding=1)
        #IN_V2 6 6 128    6 6 256
        self.v2pool1 = nn.MaxPool2d(kernel_size=3, stride=1, padding=1)
        self.v2conv1 = nn.Conv2d(128, 32, kernel_size=1, stride=1, padding=0)
        self.v2conv2 = nn.Conv2d(128, 16, kernel_size=1, stride=1, padding=0)
        self.v2conv22 = nn.Conv2d(16, 32, kernel_size=5, stride=1, padding=2)
        self.v2conv3 = nn.Conv2d(128, 96, kernel_size=1, stride=1, padding=0)
        self.v2conv32 = nn.Conv2d(96, 128, kernel_size=3, stride=1, padding=1)
        self.v2conv4 = nn.Conv2d(128, 64, kernel_size=1, stride=1, padding=0)
        self.pool3 = nn.MaxPool2d(kernel_size=3, stride=3, padding=0)
        #IN_V3 2 2 256 2 2 512
        self.v3pool1 = nn.MaxPool2d(kernel_size=3, stride=1, padding=1)
        self.v3conv1 = nn.Conv2d(256, 64, kernel_size=1, stride=1, padding=0)
        self.v3conv2 = nn.Conv2d(256, 32, kernel_size=1, stride=1, padding=0)
        self.v3conv22 = nn.Conv2d(32, 64, kernel_size=5, stride=1, padding=2)
        self.v3conv3 = nn.Conv2d(256, 192, kernel_size=1, stride=1, padding=0)
```

```
        self.v3conv32 = nn.Conv2d(192, 256, kernel_size=3, stride=1, padding=1)
        self.v3conv4 = nn.Conv2d(256, 128, kernel_size=1, stride=1, padding=0)
        self.pool4 = nn.MaxPool2d(kernel_size=2, stride=2, padding=0)
        # Size of the output of the last convolution:
        self.flattened_size = 512
        ### END CODE ###
        self.fc1 = nn.Linear(self.flattened_size, 10)

    def forward(self, x):
        # shape : 3x32x32 -> 18x32x32
        x = F.relu(self.conv1(x))
        # 18x32x32 -> 18x16x16
        x = self.pool1(x)
        #IN_V1
        x1 = self.v1pool1(x)
        x1 = self.v1conv1(x1)
        x2 = self.v1conv2(x)
        x2 = self.v1conv22(x2)
        x3 = self.v1conv3(x)
        x3 = self.v1conv32(x3)
        x4 = self.v1conv4(x)
        x = F.relu(torch.cat((x1,x2,x3,x4),1))
        x = self.pool2(x)
        #IN_V2
        x1 = self.v2pool1(x)
        x1 = self.v2conv1(x1)
        x2 = self.v2conv2(x)
        x2 = self.v2conv22(x2)
        x3 = self.v2conv3(x)
        x3 = self.v2conv32(x3)
        x4 = self.v2conv4(x)
        x = F.relu(torch.cat((x1,x2,x3,x4),1))
        x = self.pool3(x)
        #IN_V3
        x1 = self.v3pool1(x)
        x1 = self.v3conv1(x1)
        x2 = self.v3conv2(x)
        x2 = self.v3conv22(x2)
        x3 = self.v3conv3(x)
        x3 = self.v3conv32(x3)
        x4 = self.v3conv4(x)
        x = F.relu(torch.cat((x1,x2,x3,x4),1))
        x = self.pool4(x)

        x = x.view(-1, self.flattened_size)
        x = F.dropout(x, training=self.training)
        x = self.fc1(x)
        return x
```

7 Capsule Network

In the last year, Geoffrey E Hinton has proposed a new framework of neural network, which is capsule network, so we decided to test the model in our case. However, we run into a problem of memory size.

RuntimeError: $ Torch: not enough memory: you tried to allocate 1GB. Buy new RAM! at /pytorch/torch/lib/TH/THGeneral.c:270