HOMEWORK 6

Jiazhi Li

jiazhil@usc.edu

April 28, 2019

1 Feedforward-designed convolution neural networks

1.1 Abstract and Motivation

At present, all back-propogation designed CNNs cannot be easily explained in mathematical aspects because of the non-linear activation and back propogation technique. Without reasonable explanation, convolution neural network is just like a black box. By this way, the improvements in this area are just more complicated network or deeper layers. At this time, FF-CNN is a kind of explainable CNN using feedforward instead of back propogation, which is serving as interpretable convolutional neural networks.

1.2 Approach and Procedures

1.2.1 Conv layers using the Saab transforms

From the perspective of feedforward, the convolution layer can be regarded as dimension reduction. Let's have a quick look about convolution layer using the Saab transforms. There are three things we should concerned. They are DC component, AC component and bias [1].
In order to simplify explanation, first we ignore the bias and just concern the output consisted with DC and AC component. For DC component, the formula is quite straight forward, which is shown as followed,

$$\frac{1}{\sqrt{N}}(1, ..., 1)^T$$

If we assign this term to input, we can just get the mean of input. Thus, we consider the DC component as kind of the mean of input. From the aspect of space, the whole output space can be thought as the space spanned by both DC vector and AC vectors. At this time, DC component is easily obtained and the output excluding DC component is AC components. Thus, very reasonably, in order to acquire the AC components, we need to minus the DC component, so-called mean from anchor vectors, which is showing like that,

$$x_{AC} = x - x_{DC}$$

For AC components, let's think about it in a example that there are 60000 images with 32*32 and kernel window size is 4*4 with scanning stride 4. In this example, when we apply this scanning technique, we will get 60000*8*8 windows and each window contains 16 pixels. We can find that the dimension of inputs are huge so that we conduct PCA on all these 60000*8*8 patches and choose first 5 principal components as AC anchor vectors. Having combined these AC vector with DC vector, we eventually get the whole anchor vectors. At this time, five AC 16-D kernels and one DC 16-D kernel are obtained.
Let's come back first to think about the bias term. It is used to compensate the effect of non-linear activation, which is working for solve the problem of sign confusion. There are two guidelines for bias selection. The first is that after applying correction of bias to kernels, the response of input by this kernel should be non-negative. The other is about equality of all bias. Under these two requirements, the selection rule comes out,

$$b_k >= max||x||$$

$$b_0 = b_1 = ... = b_{K-1}$$

The whole flow diagram is shown in **Figure** 1.

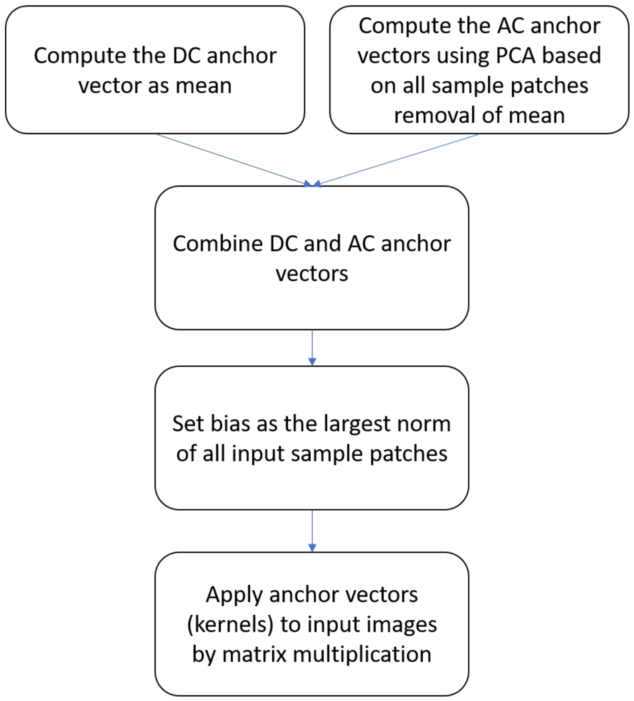

Compute the DC anchor vector as mean
Compute the AC anchor vectors using PCA based on all sample patches removal of mean
Combine DC and AC anchor vectors
Set bias as the largest norm of all input sample patches
Apply anchor vectors (kernels) to input images by matrix multiplication

Figure 1: Flow diagram for Saab transform

That's all about convolution layer and it is followed by max pooling layer. Between two successive Convolutional layer, there is a Pooling layer inserted to subsample the output of previous layer. The function is to progressively reduce the spatial size of the representation and the amount of parameters in the network in order to control overfitting [2]. What the Pooling layer do is shown in **Figure** 2 .

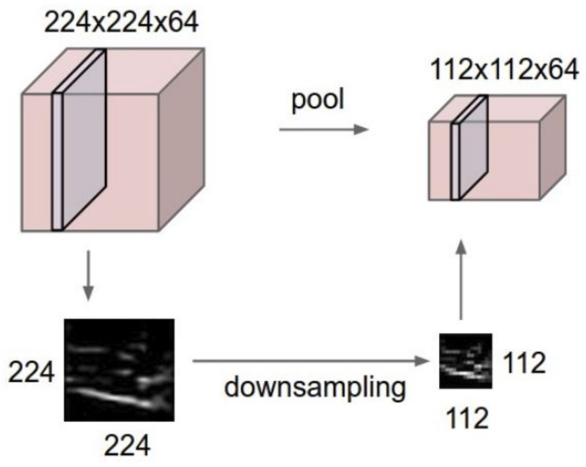


Figure 2: Pooling Layer

In LeNet-5, a 2*2 MAX window with stride = 2 is used to reduce the number of weights for the following layer. The MAX operation is that to keep the maximum in a 2*2 window and discard to other three value, which is shown in **Figure** 3.

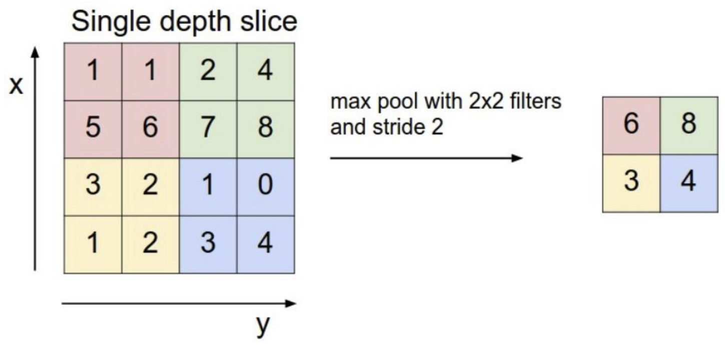


Figure 3: MAX operation

Taking LeNet-5 as a example, two stage Saab transforms where the size of the transform kernels is 4*4 and stride of each is 4. According to the *Getkernel_compact.py*, I can summary the two stage feedforward design convolutional layer as followed in **Figure** 4.

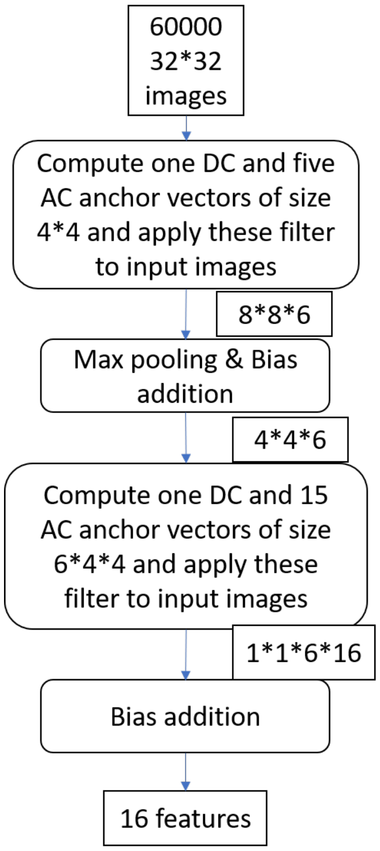

60000 32*32 images
Compute one DC and five AC anchor vectors of size 4*4 and apply these filter to input images
8*8*6
Max pooling & Bias addition
4*4*6
Compute one DC and 15 AC anchor vectors of size 6*4*4 and apply these filter to input images
1*1*6*16
Bias addition
16 features

Figure 4: Flow diagram for two stage FF design convolutional layers

1.2.2 Fully-connected layers using the multi-stage linear least squared regressor

To be mentioned first, if we use the LeNet-5-like architecture, before fully connected layer we will get a 400D feature for each input image. Since after this first FC layer, we want to get 120 nodes as output so that a 120D one hot vector are designed in this stage, which is shown in **Figure** 5.

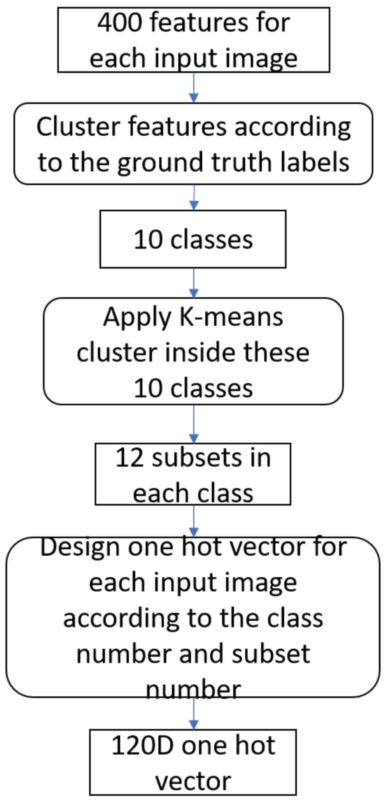
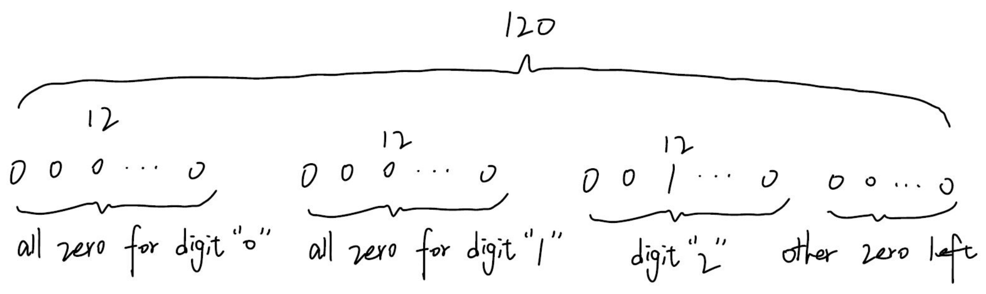
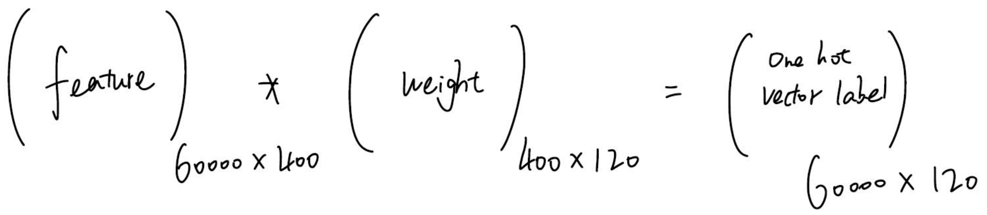

400 features for each input image
Cluster features according to the ground truth labels
10 classes
Apply K-means cluster inside these 10 classes
12 subsets in each class
Design one hot vector for each input image according to the class number and subset number
120D one hot vector

Figure 5: Flow diagram for one hot vector

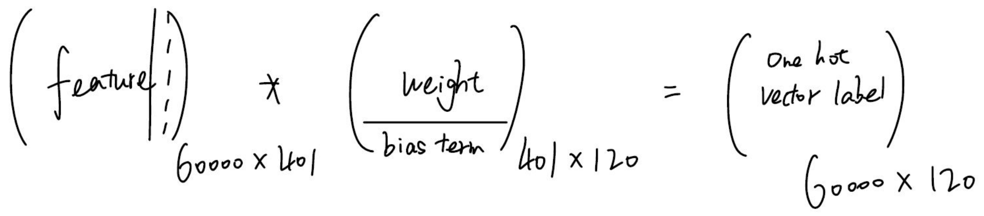
For example, if a 400D feature for one image is belonging to the third subsets of digit ’2’, then its one hot vector will be,



Right there, if there are 60000 images in training data set, we will get 60000 120D one hot vectors corresponding to these images. These 120D one hot vectors is output in this layer and the input is the 400D feature. Thus, we use least-squared regressor to find the transformation function "weight". The forward transformation is shown in **Figure** 6.



(a) Without bias term



(b) With bias term

Figure 6: Forward transformation

Thus, we can get "weight" matrix with the following formula,

$$weight = feature^{-1} * label$$

By this way, "weight" matrix is the transformation matrix or filter in the first FC layer. For each input 400D feature, we can do 2D convolution with this filter to the a 120D one hot vector as output. The three stage fully connected layer is shown in **Figure** 7.

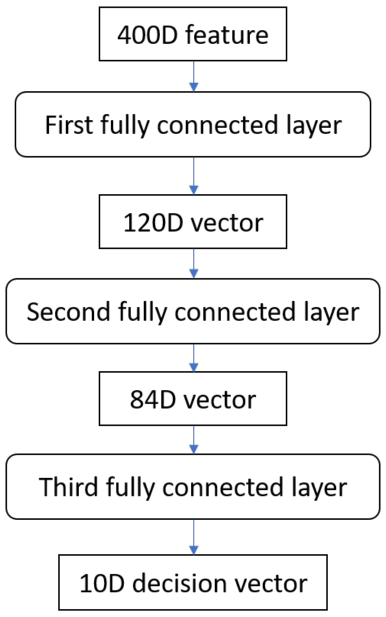

400D feature
First fully connected layer
120D vector
Second fully connected layer
84D vector
Third fully connected layer
10D decision vector

Figure 7: Flow diagram for fully connected layers

1.3 Discussion

1.3.1 Similarities between FF-CNNs and backpropagation-designed CNNs

The similarity between FF-CNN and BP-CNN is quite straight forward. Both of them tend to own the same structure. They are two kinds of layers, convolution layer and fully-connected layer. From the experimental result, both of them are vulnerable to attacks.

1.3.2 Differences between FF-CNNS and backpropagation-designed CNNs

There are too many differences between FF-CNN and BP-CNN. The whole bunch of filter weights in BP-CNN are obtained by non-convex optimization. However, FF-CNN is based on the input data. In more details, the filter weights in convolution layer of FF-CNN don't depend on labels but statistics of input image. Only fully-connected layer is based on labels. What's more, the main method in convolution layer for FF-CNN is PCA and that in fully-connected layer is k-means and linear regression, which can be easily explained in mathematical aspect. However, in BP-CNN, the use of non-linear activation and back propogation make it difficult to interpret.

2 Image reconstructions from Saab coefficients

2.1 Approach and Procedures

The first step is to use 60000 training images to get the kernels and bias in each stage. The flow diagram is shown in **Figure** 8.

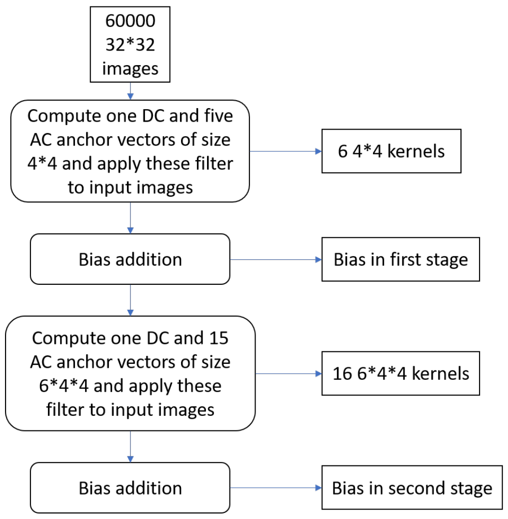

60000 32*32 images
Compute one DC and five AC anchor vectors of size 4*4 and apply these filter to input images
6 4*4 kernels
Bias addition
Bias in first stage
Compute one DC and 15 AC anchor vectors of size 6*4*4 and apply these filter to input images
16 6*4*4 kernels
Bias addition
Bias in second stage

Figure 8: Flow diagram for getting kernels and bias

After that, we get the kernels and bias needed for Saab transformation. By this way, we can use these kernels to apply Saab transformation for the following image shown in **Figure** 9.

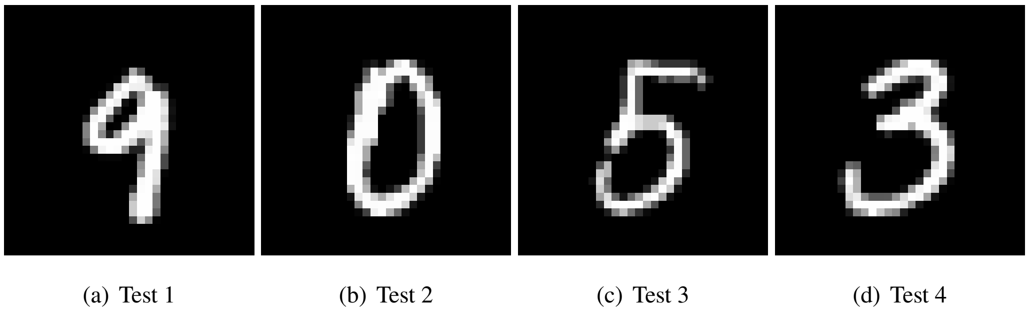

(a) Test 1 (b) Test 2 (c) Test 3 (d) Test 4

Figure 9: Origin images

The flow diagram to get Saab coefficients is shown in **Figure** 10.

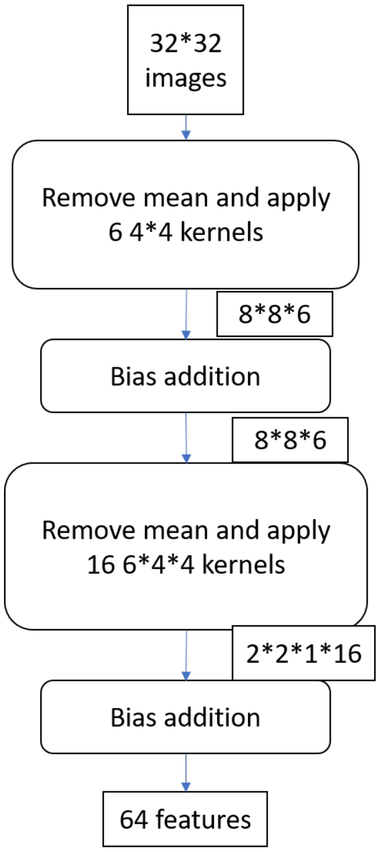
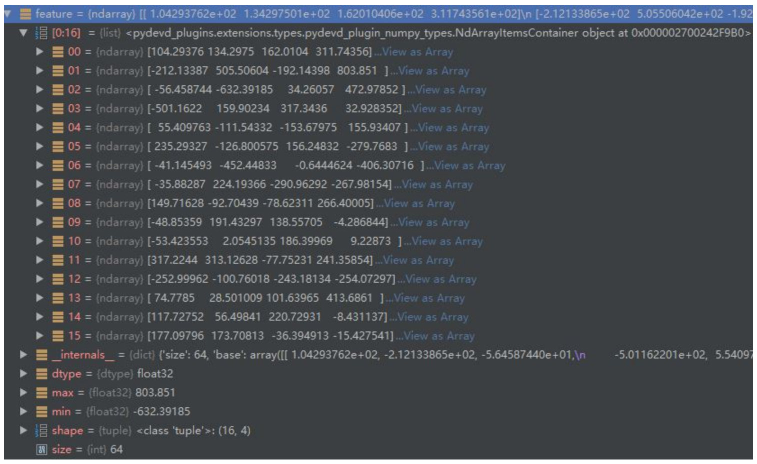

32*32 images
Remove mean and apply 6 4*4 kernels
8*8*6
Bias addition
8*8*6
Remove mean and apply 16 6*4*4 kernels
2*2*1*16
Bias addition
64 features

Figure 10: Flow diagram for getting Saab coefficients

If the kernel number is (6,16), then for each 32*32 image, 64 features will be obtained, which is shown in **Figure** 11.



```
▼ feature = {ndarray} [[ 1.04293762e+02  1.34297501e+02  1.62010406e+02  3.11743561e+02]\n [-2.12133865e+02  5.05506042e+02 -1.92
  ▼ [0:16]  = {list} <pydevd_plugins.extensions.types.pydevd_plugin_numpy_types.NdArrayItemsContainer object at 0x000002700242F9B0>
    ▶ 00 = {ndarray} [104.29376 134.2975  162.0104  311.74356] ...View as Array
    ▶ 01 = {ndarray} [-212.13387  505.50604 -192.14398  803.851  ] ...View as Array
    ▶ 02 = {ndarray} [ -56.458744 -632.39185    34.26057   472.97852 ] ...View as Array
    ▶ 03 = {ndarray} [-501.1622    159.90234   317.3436     32.928352] ...View as Array
    ▶ 04 = {ndarray} [  55.409763 -111.54332  -153.67975   155.93407 ] ...View as Array
    ▶ 05 = {ndarray} [ 235.29327  -126.800575  156.24832  -279.7683  ] ...View as Array
    ▶ 06 = {ndarray} [ -41.145493  -452.44833     -0.6444624 -406.30716  ] ...View as Array
    ▶ 07 = {ndarray} [ -35.88287  224.19366 -290.96292 -267.98154] ...View as Array
    ▶ 08 = {ndarray} [149.71628 -92.70439 -78.62311 266.40005] ...View as Array
    ▶ 09 = {ndarray} [-48.85359  191.43297  138.55705   -4.286844] ...View as Array
    ▶ 10 = {ndarray} [-53.423553    2.0545135 186.39969     9.22873  ] ...View as Array
    ▶ 11 = {ndarray} [317.2244  313.12628 -77.75231 241.35854] ...View as Array
    ▶ 12 = {ndarray} [-252.99962 -100.76018 -243.18134 -254.07297] ...View as Array
    ▶ 13 = {ndarray} [ 74.7785    28.501009 101.63965  413.6861  ] ...View as Array
    ▶ 14 = {ndarray} [117.72752   56.49841  220.72931   -8.431137] ...View as Array
    ▶ 15 = {ndarray} [177.09796  173.70813  -36.394913 -15.427541] ...View as Array
  ▶ _internals_ = {dict} {'size': 64, 'base': array([[ 1.04293762e+02, -2.12133865e+02, -5.64587440e+01,\n        -5.01162201e+02,  5.54097
  ▶ dtype = {dtype} float32
  ▶ max = {float32} 803.851
  ▶ min = {float32} -632.39185
  ▶ shape = {tuple} <class 'tuple'>: (16, 4)
    size = {int} 64
```

Figure 11: Saab coefficients

The procedures for reconstruction is shown in **Figure** 12.

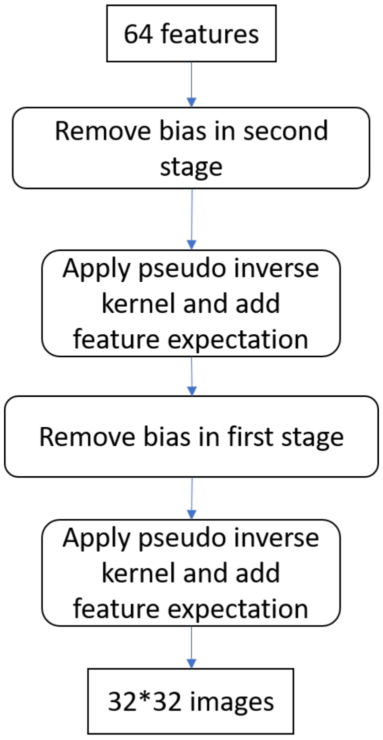
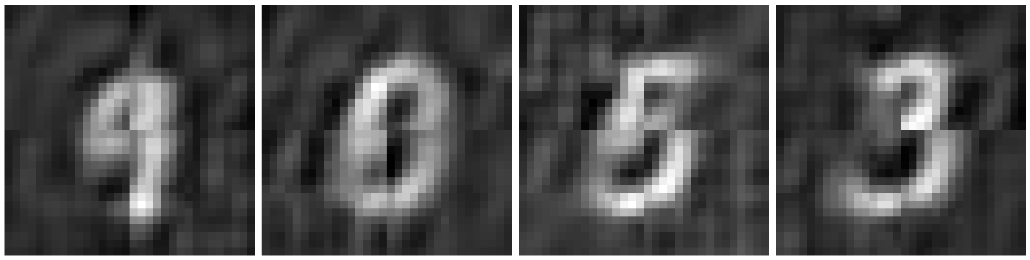

64 features
Remove bias in second stage
Apply pseudo inverse kernel and add feature expectation
Remove bias in first stage
Apply pseudo inverse kernel and add feature expectation
32*32 images

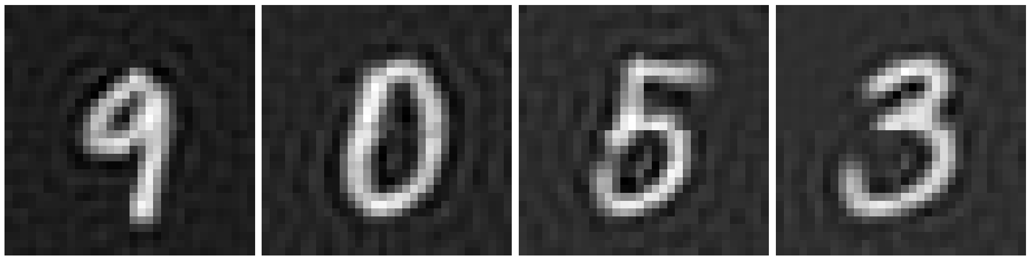
Figure 12: Flow diagram for reconstruction

2.2 Experimental Results

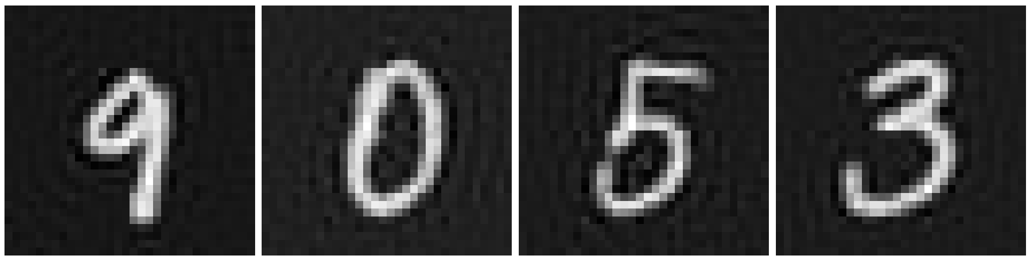
The reconstructed images are shown in **Figure** 13. To be mentioned, the notation is like that (kernel number in first stage, kernel number in second stage). And the following number is about PSNR.



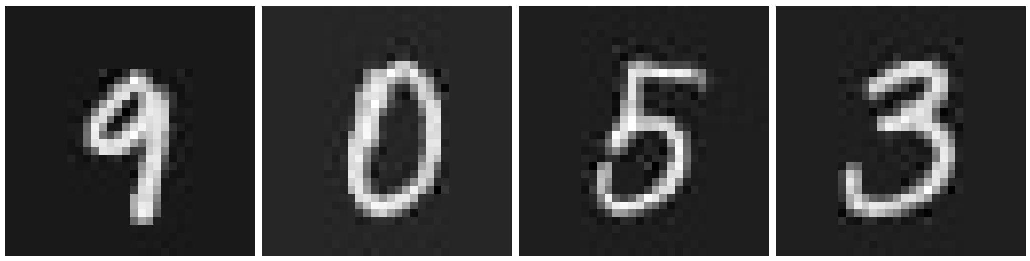
(a) "Digit 9" (6,16) 13.40 (b) "Digit 0" (6,16) 12.21 (c) "Digit 5" (6,16) 11.24 (d) "Digit 3" (6,16) 12.91



(e) "Digit 9" (10,60) 17.73 (f) "Digit 0" (10,60) 16.00 (g) "Digit 5" (10,60) 14.70 (h) "Digit 3" (10,60)14.55



(i) "Digit 9"(10,100)20.00 (j) "Digit 0" (10,100)17.54 (k) "Digit5"(10,100)18.84 (l) "Digit 3" (10,100)18.78



(m) "9"(10,160)25.43 (n) "0"(10,160)20.69 (o) "5"(10,160)21.32 (p) "3"(10,160)21.37

Figure 13: Reconstruction result

2.3 Discussion

From these images, we can obviously find that the digits image especially with small number of kernels look like a combination of four separable parts, shown in the following **Figure** 14.

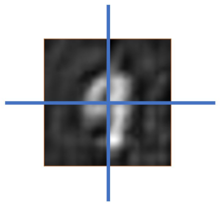


Figure 14: Four separable parts for reconstruction images

I think one reason of these "edge" is that we use 4*4 kernel window and 4 stride. By this way, it just extract each window separably without any correlation information between "edge". When we do reconstruction from features, lack of those edge correlation information makes it look like this way.
Another observation is about relation between two stage kernel number and PSNR. We know PCA is used in each stage to maintain the principle component of input data. Although PCA is very strong technique for dimension reduction without any loss of important information, it will definitely eliminate some necessary information. By this way, when we use more kernels to record more principle components of input data, the high fidelity will be remaining so that higher PSNR is obtained.

3 Handwritten digits recognition using ensembles of feedforward design

3.1 Abstract and Motivation

The main idea of ensemble learning is to generate multiple learners through certain rules, and then combine them with some integration strategy, and finally judge the output and final results. In general, the multiple learners in the ensemble learning are all homogeneous "weak learners". Based on the weak learners, multiple learners are generated by sample set perturbation, input feature perturbation, output representation perturbation and algorithm parameter perturbation. After that, a "strong learner" with better precision is obtained after integration. With the deepening of integrated learning research, its broad definition has gradually been accepted by scholars. It refers to the way of learning multiple sets of learners without distinguishing the nature of learners. According to this definition, multiple fields such as multi-classifier system, mixture of experts, and committee-based learning can be incorporated into ensemble learning. However, there is still a lot of ensemble learning research with homogeneous classifiers.

3.2 Approach and Procedures

Before showing my strategy to build the ensemble systems of FF-CNNs, let's look at the procedures of training process and testing process, which are shown in **Figure** 15 and **Figure** 16.

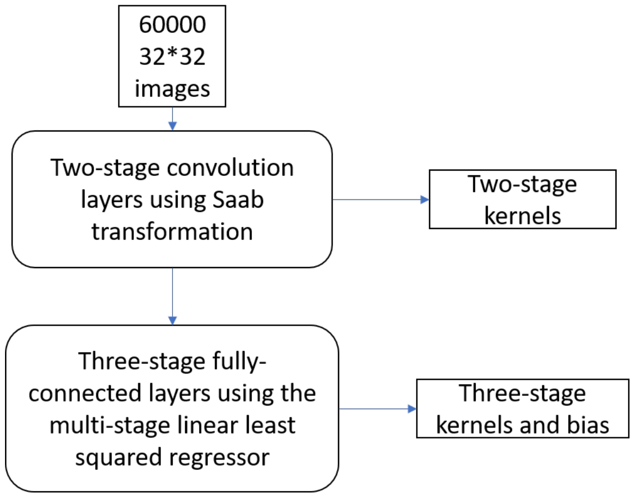

60000 32*32 images
Two-stage convolution layers using Saab transformation
Two-stage kernels
Three-stage fully-connected layers using the multi-stage linear least squared regressor
Three-stage kernels and bias

Figure 15: Flow diagram for training process

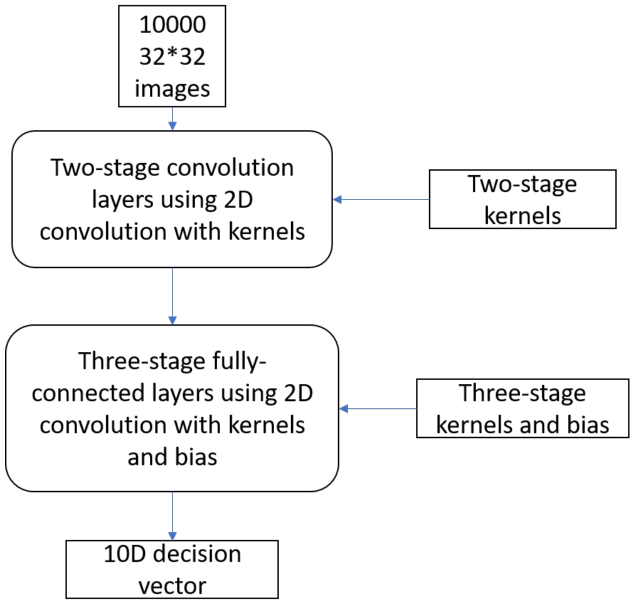

10000 32*32 images
Two-stage convolution layers using 2D convolution with kernels
Two-stage kernels
Three-stage fully-connected layers using 2D convolution with kernels and bias
Three-stage kernels and bias
10D decision vector

Figure 16: Flow diagram for testing process

By this way, I can simplify the training process and testing process in only one block, which are shown in **Figure** 17 and **Figure** 18. In order to increase the diversity of input images, I apply 3*3 Laws filter to them. Since there are 9 Laws filters, we can get 9 filtered images after 2D convolution with Laws filters. After appending the input images without any changes, we get 10 different input images so that 10 different FF-CNNs will be trained.

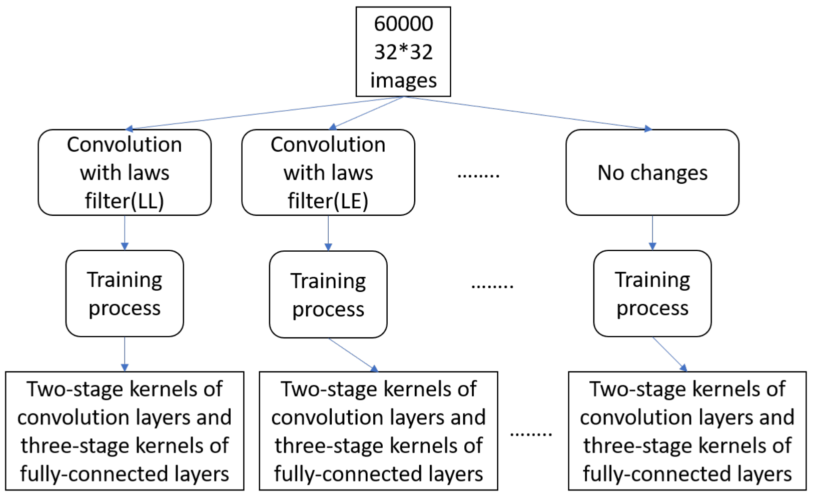

60000 32*32 images
Convolution with laws filter(LL)
Convolution with laws filter(LE)
....
No changes
Training process
Training process
....
Training process
Two-stage kernels of convolution layers and three-stage kernels of fully-connected layers
Two-stage kernels of convolution layers and three-stage kernels of fully-connected layers
....
Two-stage kernels of convolution layers and three-stage kernels of fully-connected layers

Figure 17: Flow diagram for training process of weak classifier

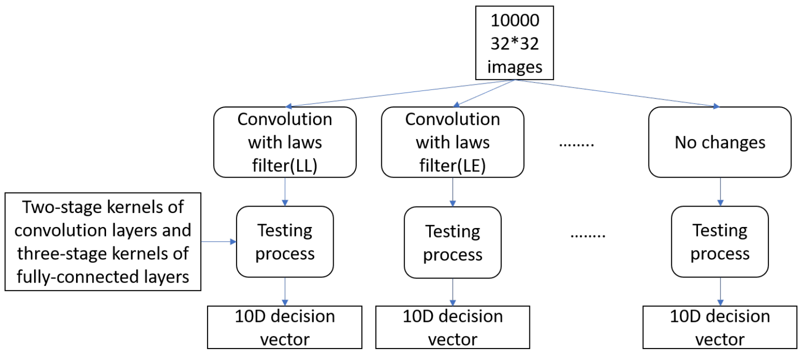

10000 32*32 images
Convolution with laws filter(LL)
Convolution with laws filter(LE)
....
No changes
Two-stage kernels of convolution layers and three-stage kernels of fully-connected layers
Testing process
Testing process
....
Testing process
10D decision vector
10D decision vector
10D decision vector

Figure 18: Flow diagram for test process of weak classifier

At this time, I have trained 10 weak classifiers for first step classification. And then, the strong classier SVM is used for the final classification. The training and testing process for SVM is shown in **Figure** 19 and **Figure** 20.

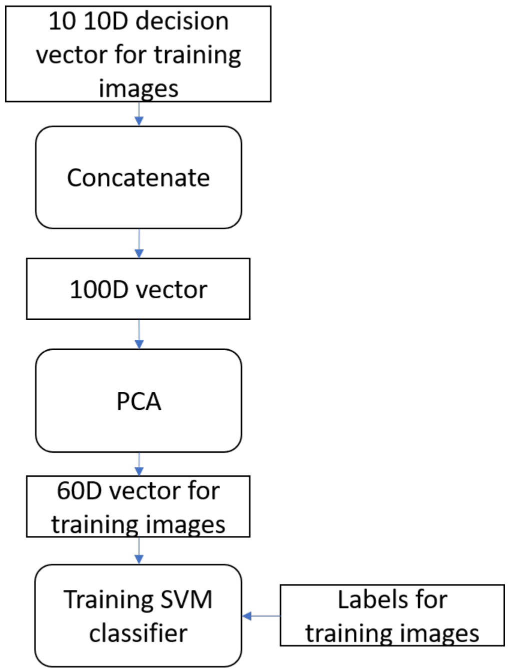

10 10D decision vector for training images
Concatenate
100D vector
PCA
60D vector for training images
Training SVM classifier
Labels for training images

Figure 19: Flow diagram for training process of ensemble classifier

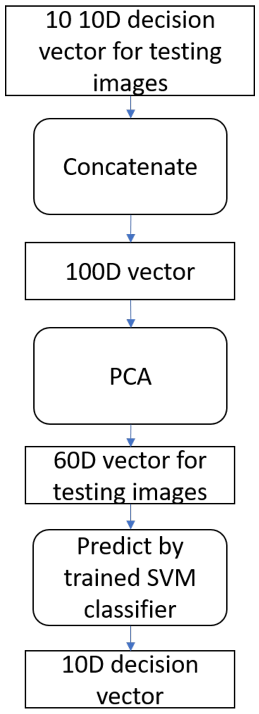
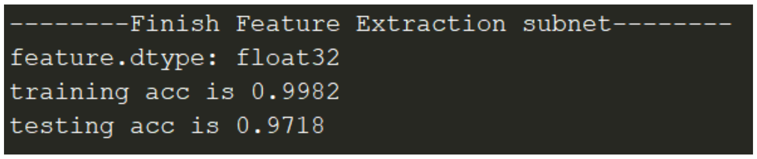

10 10D decision vector for testing images
Concatenate
100D vector
PCA
60D vector for testing images
Predict by trained SVM classifier
10D decision vector

Figure 20: Flow diagram for test process of ensemble classifier

3.3 Experimental Results

Training and testing classification accuracy for individual FF-CNN on the MNIST dataset is shown in **Figure** 21.



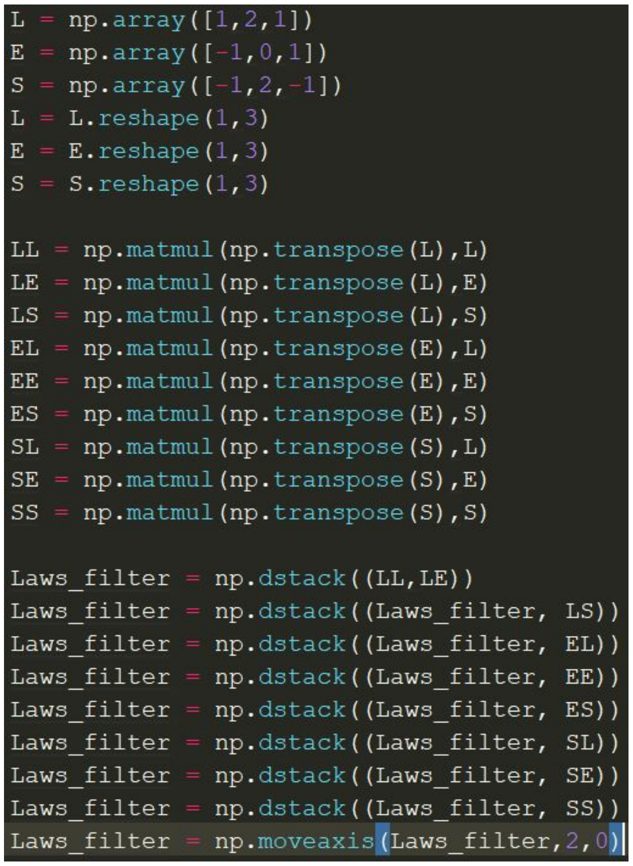
```
--------Finish Feature Extraction subnet--------
feature.dtype: float32
training acc is 0.9982
testing acc is 0.9718
```

Figure 21: Training and testing classification accuracy

3.4 Discussion

3.4.1 Strategy for ensemble system design

In my ensemble system, I use 9 3*3 laws filters to add diversity to the ensemble system. There are 3 basic vectors for 3*3 laws filter, which are shown in **Figure** 22. They are level(L), edge(E) and spot(S).



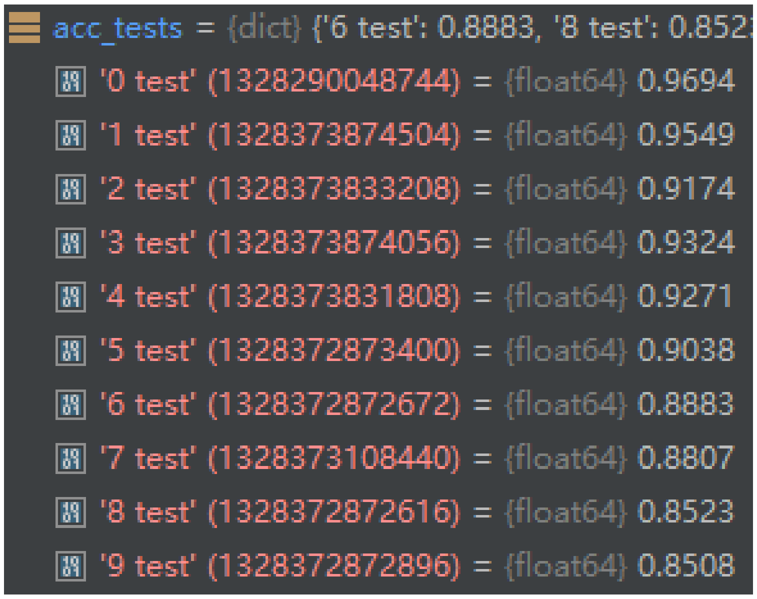
```
L = np.array([1,2,1])
E = np.array([-1,0,1])
S = np.array([-1,2,-1])
L = L.reshape(1,3)
E = E.reshape(1,3)
S = S.reshape(1,3)

LL = np.matmul(np.transpose(L),L)
LE = np.matmul(np.transpose(L),E)
LS = np.matmul(np.transpose(L),S)
EL = np.matmul(np.transpose(E),L)
EE = np.matmul(np.transpose(E),E)
ES = np.matmul(np.transpose(E),S)
SL = np.matmul(np.transpose(S),L)
SE = np.matmul(np.transpose(S),E)
SS = np.matmul(np.transpose(S),S)

Laws_filter = np.dstack((LL,LE))
Laws_filter = np.dstack((Laws_filter, LS))
Laws_filter = np.dstack((Laws_filter, EL))
Laws_filter = np.dstack((Laws_filter, EE))
Laws_filter = np.dstack((Laws_filter, ES))
Laws_filter = np.dstack((Laws_filter, SL))
Laws_filter = np.dstack((Laws_filter, SE))
Laws_filter = np.dstack((Laws_filter, SS))
Laws_filter = np.moveaxis(Laws_filter,2,0)
```

Figure 22: Laws filter

Although digit images from MNIST are not the real texture images, the specific shape of each digit number can be regarded as kind of texture. For example, after applying edge filters, the high response will be obtained in specific region for each digit number. By this way, when I apply 9 different Laws filter before training the FF-CNN, different responses increase the diversity in the ensemble system of FF-CNNs. The testing accuracy for each individual FF-CNNs in ensemble is shown in **Figure** 23.



```
acc_tests = {dict} {'6 test': 0.8883, '8 test': 0.852
    '0 test' (1328290048744) = {float64} 0.9694
    '1 test' (1328373874504) = {float64} 0.9549
    '2 test' (1328373833208) = {float64} 0.9174
    '3 test' (1328373874056) = {float64} 0.9324
    '4 test' (1328373831808) = {float64} 0.9271
    '5 test' (1328372873400) = {float64} 0.9038
    '6 test' (1328372872672) = {float64} 0.8883
    '7 test' (1328373108440) = {float64} 0.8807
    '8 test' (1328372872616) = {float64} 0.8523
    '9 test' (1328372872896) = {float64} 0.8508
```

Figure 23: Testing accuracy for each individual FF-CNNs in ensemble system

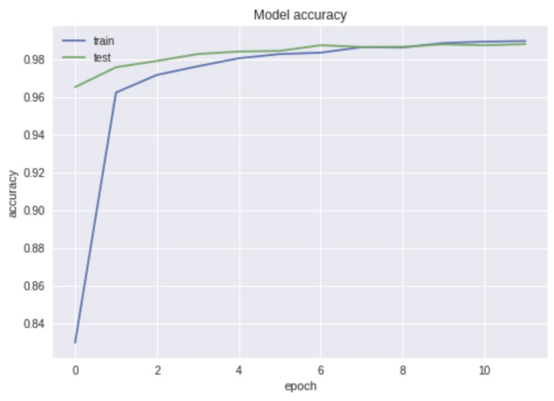
Finally, when I use the ensemble system introduced in last section, we get training and testing classification accuracy of my ensemble system shown in **Table** 1.

Table 1: Accuracy for individual classifier in ensemble system

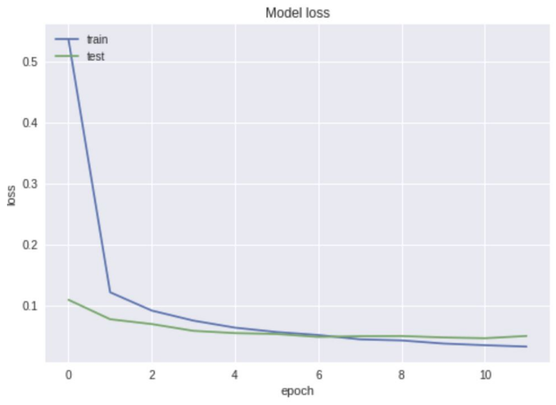
	Origin	LL	LE	LS	EL	EE	ES	SL	SE	SS	Ensemble
Testing	96.94%	95.49%	91.74%	93.24%	92.71%	90.38%	88.83%	88.07%	85.23%	85.08%	97.90%
Training	96.94%	98.43%	97.24%	98.56%	98.43%	99.10%	98.34%	98.43%	98.65%	97.32%	99.12%

3.4.2 Error analysis

The best result for BP-CNNs is obtained in setting (12,128,0.2,0.01,1e-6,0.9) with notation (epoch,batch size,dropout,learning rate,decay,momentum). The best result is shown in **Figure** 24.



(a) Accuracy curve



(b) Loss curve

Figure 24: Best BP-CNN with parameter (12,128,0.2,0.01,1e-6,0.9)

Having compared classification error cases arising from BP-CNNs and FF-CNNs, I got the union of error images. At this time, the accuracy of BP-CNNs is 98.8% and the accuracy of FF-CNNs is 97.2%. I got 70 union error images for both BP-CNNs and FF-CNNs. Part of these images are shown in **Figure** 25.

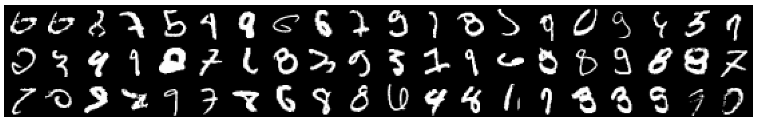


Figure 25: Union error images

Some statistics for percentages of error are shown in **Figure** 26 and **Table** 2.

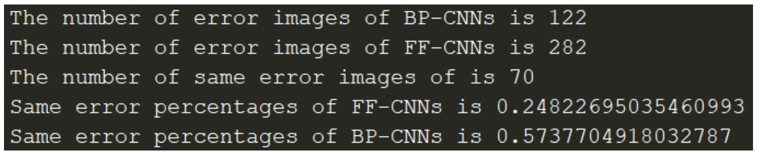

```
The number of error images of BP-CNNs is 122
The number of error images of FF-CNNs is 282
The number of same error images of is 70
Same error percentages of FF-CNNs is 0.24822695035460993
Same error percentages of BP-CNNs is 0.5737704918032787
```
Figure 26: Printed statistics in Python console

Table 2: Statistics for percentages of errors

	Number of errors	Percentages for same error	Percentages for different error
FF-CNNs	282	24.8%	75.2%
BP-CNNs	122	57.4%	42.6%

From the **Figure** 25, we can easily find that these images are very strange. Even for human beings, some of them are difficult to recognize. Since we know whatever the method they use to get the weights of filter in each layer, both BP-CNNs and FF-CNNs tend to learn the "similarity" from training data set. For example, if the task is about recognition of dog in whole bunch of data set, the images with grass as background and the images without grass can be recognized out as dog, which means that grass is not the necessary part or feature for dog. And then, after training with such images, grass or other background will be ignored by network. Les's come back to our example of MNIST. We know lots of data samples are written by kind of standard style so that the network really learn a lot from this standard style. By this way, when the images with strange writing style come out, the network just gets into trouble because it just rarely meet this style image. All in all, it means the generalization ability of this network is not enough.

3.4.3 Ideas to improve BP-CNNs and FF-CNNs

For BP-CNNs, there is no very good proposal for improvement because of the lack in mathematical explanation. One way that may be reasonable is to add more ad hoc tricks on it. It means that for different data set, we design different networks using different specific rules.

For FF-CNNs, one way to improve the performance is using ensembles of multiple FF-CNNs which is justified in last section. Another way I think is to deepen the layers of FF-CNNs. Just like what is talking in `PCANet` [3], appropriate deeper network essentially have a low-rank factorization, possibly resulting in a lower change of over-fitting the dataset, which means that the ability of generalization is improved.

Reference

[1] Kuo C C J , Zhang M , Li S , et al. Interpretable Convolutional Neural Networks via Feedforward Design[J]. 2018.
[2] El-Sawy A, El-Bakry H, Loey M. CNN for Handwritten Arabic Digits Recognition Based on LeNet-5[J]. 2016.
[3] Chan T H, Jia K, Gao S, et al. PCANet: A Simple Deep Learning Baseline for Image Classification?[J]. IEEE Transactions on Image Processing, 2015, 24(12):5017-5032.