PA2 CNN Report

计 72　刘光哲　2017011329

2018.10.11

目录

1 Environment

Operating System: macOS 10.14 Mojave
Python: Python 3.6.6
numpy: numpy 1.15.2
scipy: 1.1.0
CPU: Intel Core i7-8750H, Memory: 16 GB

2 Code Review for CNN

2.1 Basic Architecture

The source code of Convolutional Neural Network mainly based on the code of Multi-Layer Perceptron in PA1, so the basic architecture of PA2 is the almost the same as PA1.

I'll only explain the part that is different from MLP.

2.2 My Implement

conv2d_forward The input value of conv2d_forward is *input*, W, b, *kernel_size* and *pad*. As we know, the output value should be the convolve result of *input* and W, that is, $output = convolve(input, rot180(W)) + b$. In the source code, I first pad enough 0 to *input* to get it the correct size, and then do convolution channel by channel.

```
for i in range(shape[0]):
  for ic in range(shape[1]):
    for oc in range(W.shape[0]):
      output[i][oc]+=signal.convolve2d(xx[i][ic],np.rot90(W[oc][ic],2) mode='valid')
for i in range(shape[0]):
        for j in range(shape[1]):
                output[i][j] += b[j]
```

conv2d_backward Let's calculate the gradient of each parameter first:
$\frac{\partial E}{\partial input(i,j)} = \Sigma \frac{\partial E}{\partial output(i+x,j+y)} * W(n-x, m-y)$, that is $\frac{\partial \vec{E}}{\partial input} = convolve(\frac{\partial \vec{E}}{\partial output}, \vec{W})$;
$\frac{\partial E}{\partial W(x,y)} = \Sigma \frac{\partial E}{\partial output(i+x,j+y)} * input(i,j)$, that is $\frac{\partial \vec{E}}{\partial W} = convolve(\vec{input}, rot180(\frac{\partial \vec{E}}{\partial output}))$;
$\frac{\partial E}{\partial b} = \Sigma \frac{\partial E}{\partial output(i,j)}$, that is $\frac{\partial \vec{E}}{\partial b} = \Sigma \frac{\partial \vec{E}}{\partial output(i,j)}$.
Follow the formula to calculate the *grad_input* channel by channel, and throw away the zeros padded to it.

```
for i in range(grad_input1.shape[0]):
  for oc in range(grad_output.shape[1]):
    Wp = np.rot90(grad_output[i][oc], 2)
    for ic in range(grad_input1.shape[1]):
      grad_input1[i][ic] += signal.convolve2d(
            grad_output[i][oc],W[oc][ic],mode='full')
          grad_W[oc][ic] += signal.convolve2d(xx[i][ic], Wp, mode='valid')
grad_input=grad_input1[:,:,pad:pad+grad_input1.shape[2],pad:pad+grad_input1.shape[3]]
grad_b = np.sum(grad_output, axis=(0, 2, 3))
```

avgpool2d_forward Pooling forward is very easy to realize, just summer up each square and divide $kernel\_size^2$.

```
for x in range(output.shape[2]):
  for y in range(output.shape[3]):
    x1 = x * kernel_size
    y1 = y * kernel_size
    output[:,:,x,y]+=np.sum(xx[:,:,x1:x1+kernel_size, y1:y1 +kernel_size],axis=(2,3))
output /= kernel_size ** 2
```

avgpool2d_backward I used a little trick in the backward code. Note that in this case, $\frac{\partial E}{\partial input(i,j)} = \frac{\partial E}{\partial output(int(i/kernel\_size),int(j/kernel\_size))}/(kernel\_size^2)$, and this means $\frac{\partial \vec{E}}{\partial input} = \frac{\partial \vec{E}}{\partial output} \bigotimes \vec{O}nes_{kernel\_size*kernel\_size}/(kernel\_size^2)$, in which $\bigotimes$ represents *Kronecker Product*, or the *Tensor Product*.

```
grad_input=(np.kron(grad_output,np.ones((kernel_size,kernel_size)))/(kernel_size**2))
    [:,:,pad:pad+input.shape[2],pad:pad+input.shape[3]]
```

2.3 Problems Encountered

During the expriment, the following problems happened:

- forgot to rot180(W) in conv2d_forward
- didn't set the maximum and minimum limit in SoftmaxCrossEntropy, and it caused log(0)
- the initial parameter was too big, which caused parameters overflow and gradient explosion
- at first I didn't use the functions in numpy properly, ant it takes a long time to calculate

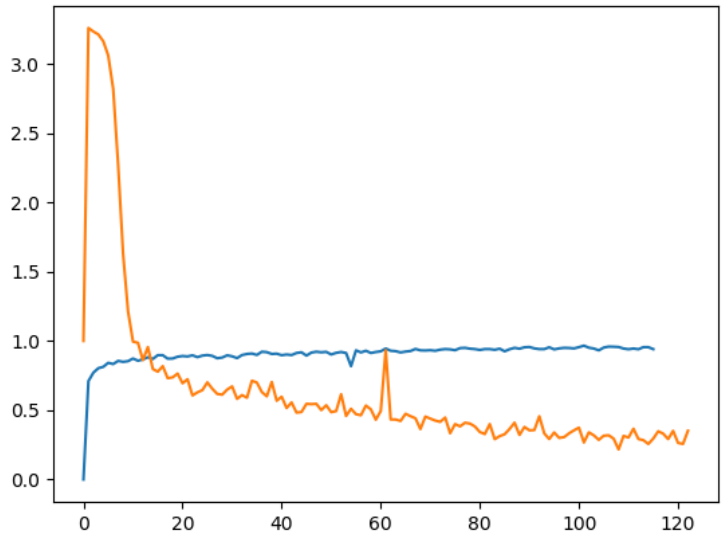
3 Test with CNN

3.1 Test results

The first test is done with the default CNN structure, and the parameters are:

learningrate	weightdecay	momentum	batchsize	init_std
0.015	0.0005	0.9	100	0.2

The training lasts 2 epochs, and the results are like this:



In the graph, the blue curve represents the accuracy of the model, the orange curve represents the Loss value of **each training iter**.

loss	accuracy	time(second)
0.251	0.959	183.273

It's obvious that the CNN didn't converge in 2 epochs, but it does prove that the CNN is working correctly, I will work on parameters to optimize the result of CNN in the next part of the report.

3.2 Working on Parameters

3.2.1 learning rate

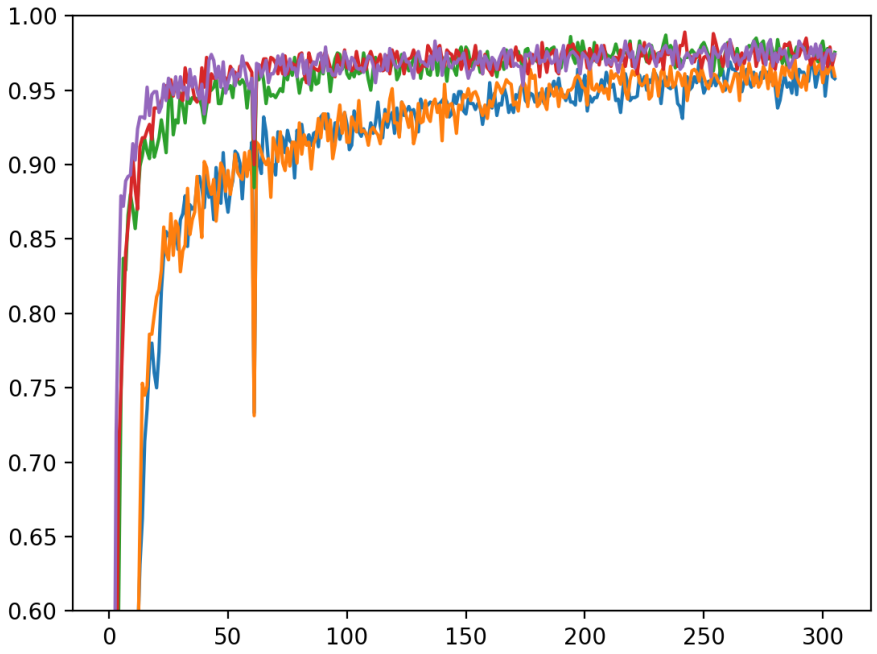
In the exam, I did the following expriments to find out the effect of changing learning rate.

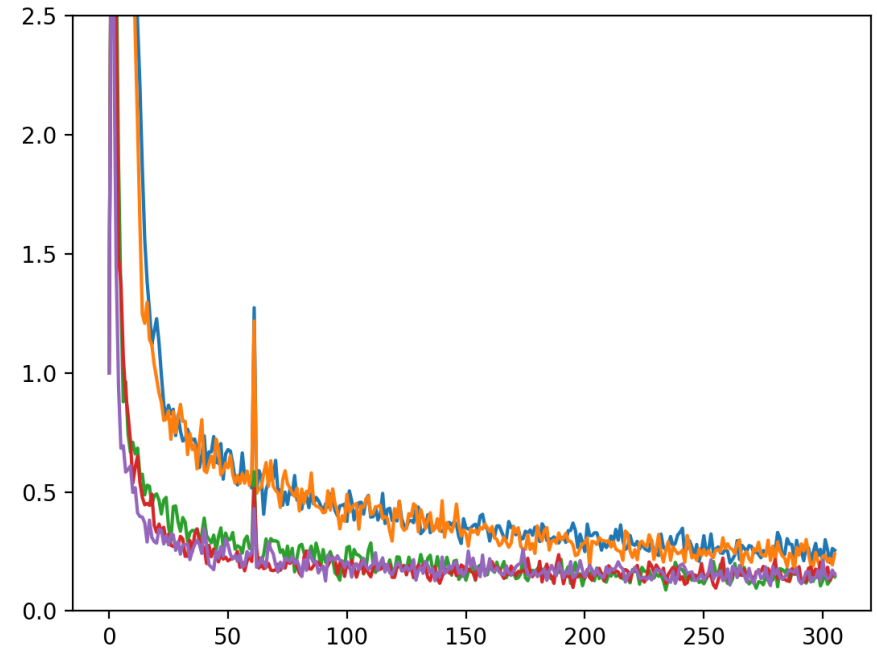
I used 0.005, 0.01, 0.05, 0.1 and a dynamic-learning_rate plan given by myself to train the CNN for 5 epochs, the results are like this:

$(dynamic = loss > 1?0.2 : loss > 0.5?0.1 : loss > 0.2?0.05 : max(0.005, \frac{loss}{5}))$

learning rate	loss	accuracy	time(second)	color
0.005	0.242	0.956	449.478	blue
0.01	0.202	0.965	447.014	orange
0.05	0.121	0.978	453.529	green
0.1	0.126	0.977	450.358	red
dynamic	0.127	0.979	497.508	purple

The first graph represents the change of accuracy, the second graph represents the change of loss value.



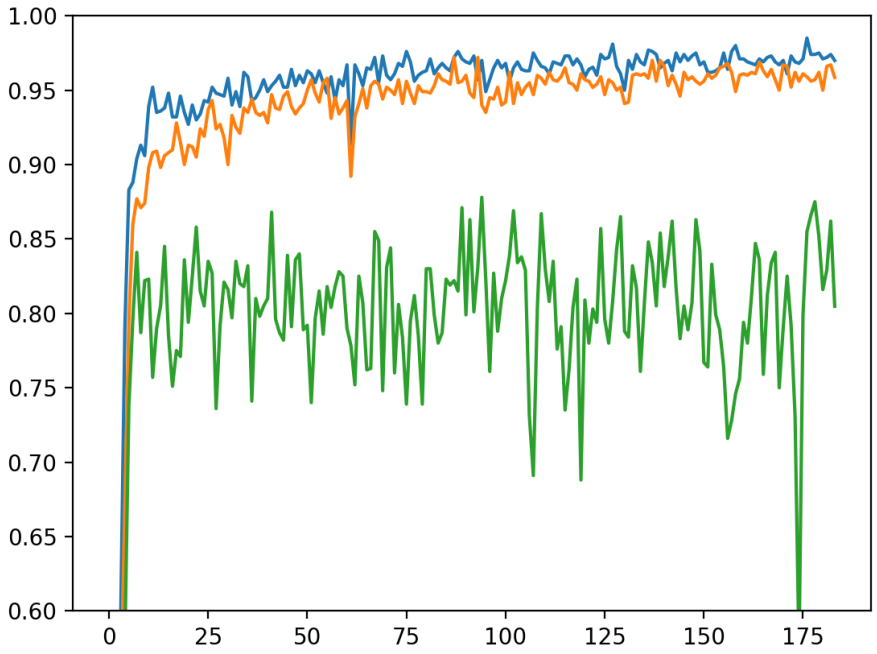


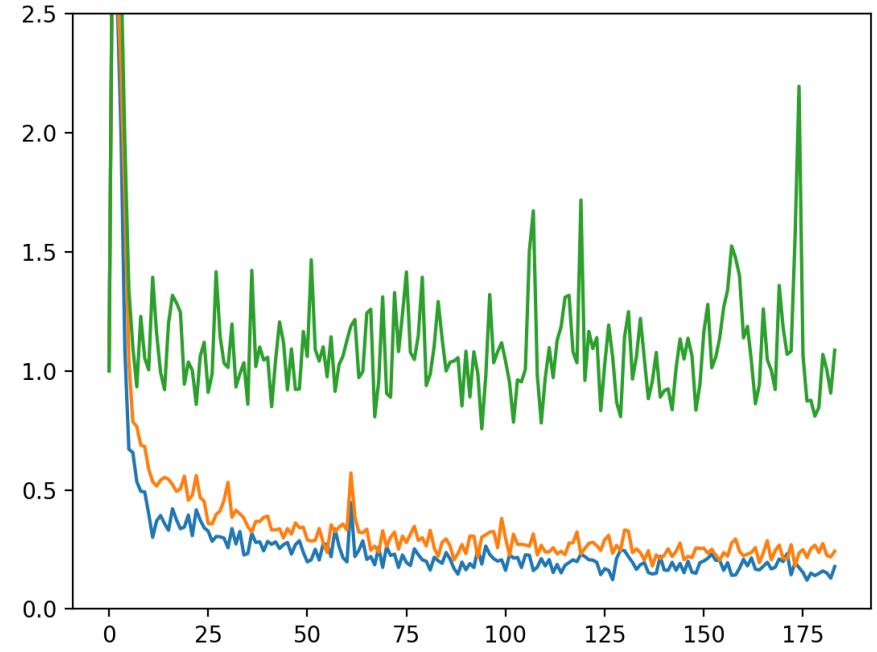
The bigger the learning rate is, the faster it converges, and the lower accuracy it converges to. In order to converge faster and keep its accuracy, *dynamic* learning rate is a useful trick.

3.2.2 weight decay

I used 0.00005, 0.0005 and 0.05 to train 3 epochs, the results are like this:

weight decay	loss	accuracy	time(second)	color
0.00005	0.140	0.975	283.867	blue
0.0005	0.208	0.963	281.673	orange
0.05	0.958	0.829	289.794	green



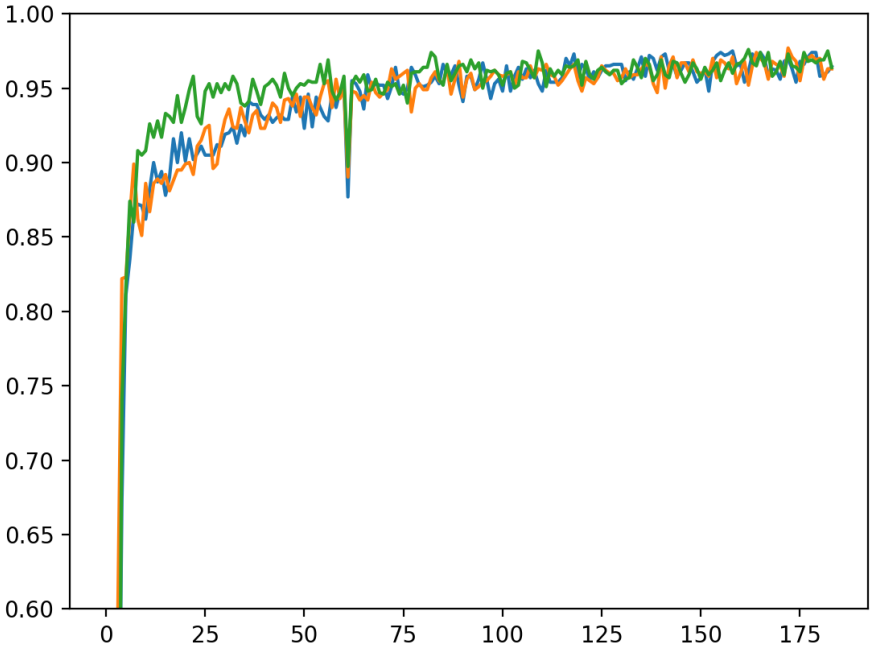


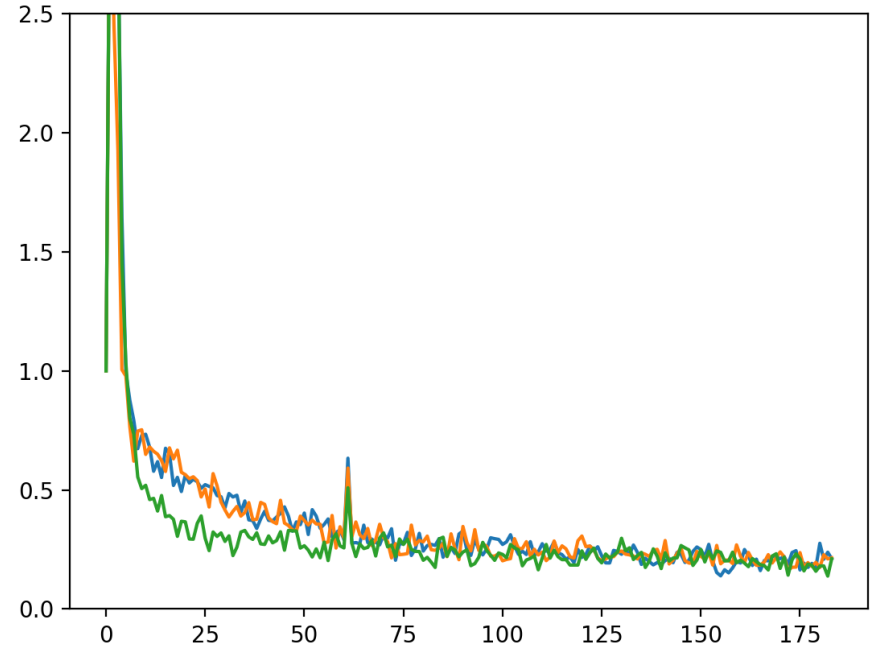
Weight decay is set to prevent overfitting, but if the weight decay is too big, it will affect the converge speed and accuracy of the network. From my expriment, 0.00005 seems to be the best weight decay value.

3.2.3 momentum

I used 0.9, 0.85 and 0.80 to train 3 epochs, the results are like this:

weight decay	loss	accuracy	time(second)	color
0.80	0.189	0.968	281.843	blue
0.85	0.197	0.964	277.002	orange
0.90	0.164	0.972	274.881	green



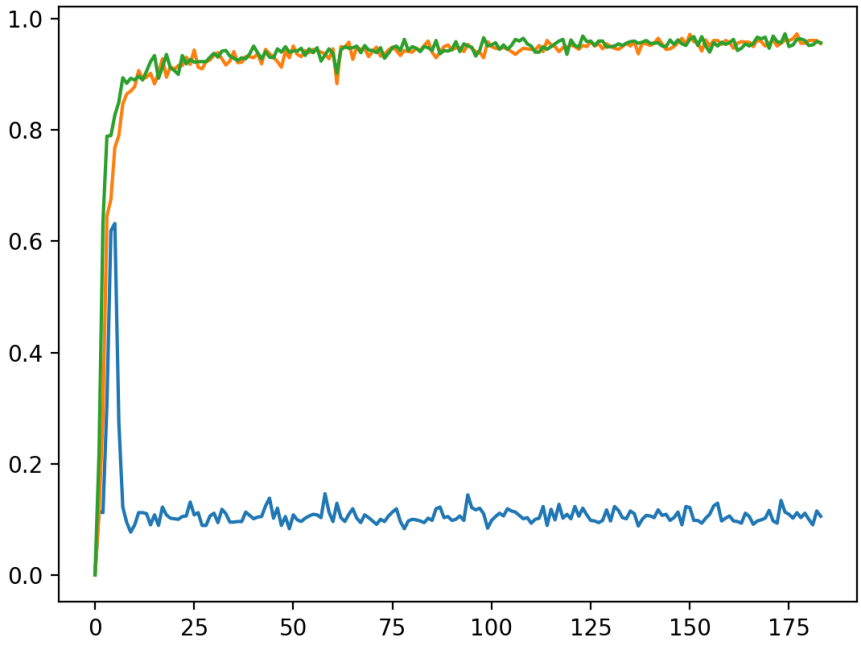


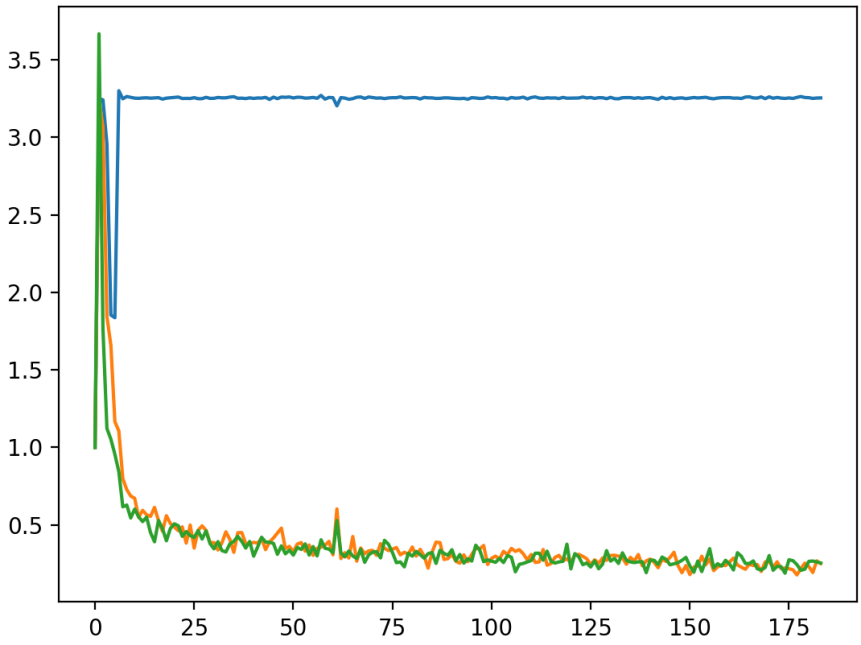
The effect of change momentum isn't as big as imagined. But we can still find that in general, neural networks with bigger momentum converges faster than others, but as for accuracy, there's no exprimental evidence that can show the relationship between momentum and accuracy. They all converge to the same limit.

3.2.4 initial state

I used 0.01, 0.1 and 1, the results are like this:

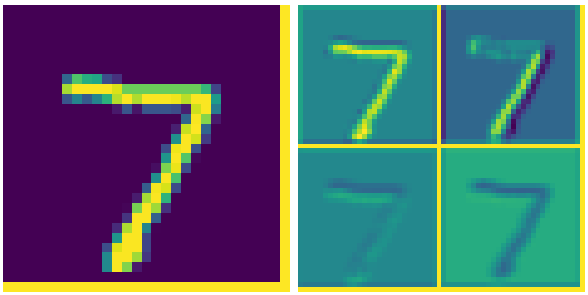
initial state	loss	accuracy	time(second)	color
0.01	3.257	0.098	275.912	blue
0.1	0.192	0.966	270.686	orange
1	0.226	0.961	282.762	green





Initial state, in fact, is not a very important factor to the result of CNN. CNN can adjust its parameters to fit in any initial state, unless it is too inproper. In my expriment, 0.1 and 1 didn't show much difference in performance, but 0.01 is exactly too small and cause gradient vanishing.

3.3 Visualization result



The left one is the original graph, and the right one is the visualization of the graph after the first convolution layer. The important features are figured out after the first convolution.

4 Comparison with MLP

4.1 Performance

CNN runs much slower than normal MLP because it has more nodes in the whole neural network. The calculation of convolution is the main time consumer in CNN, and MLP doesn't need to do so.

4.2 Analysis

According to the running results of CNN and MLP, here's my conclusion:

- CNN has more parameters and more Layers than MLP, so it covnerges slower.
- CNN has lower possibility of overfitting, for testing accuracy is often higher than training accuracy
- CNN can finally converge to a higher accuracy than MLP when the parameters are optimized.

5 Summary

For neural network, my conclusions are:

- The neural networks with more parameters converges slower, but are more likely to converge to a higher limit.
- SoftmaxCrossEntropy function may cause overflow in calculation, so measures like *add* ϵ and *minus* the greatest dimension value should be taken to make it safer.

For coding, my conclusions are:

- Make sure the details before coding, and don't use tricks that may cause wrong results.
- numpy and scipy are much more efficient than normal python calculations
- intel-numpy(numpy with mkl) is much faster than normal numpy