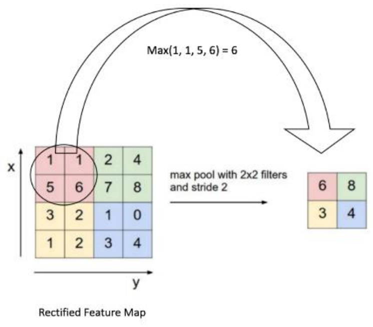
The pixel I use is randomly generated by torch.randn(3,32,32)

1.MaxPooling



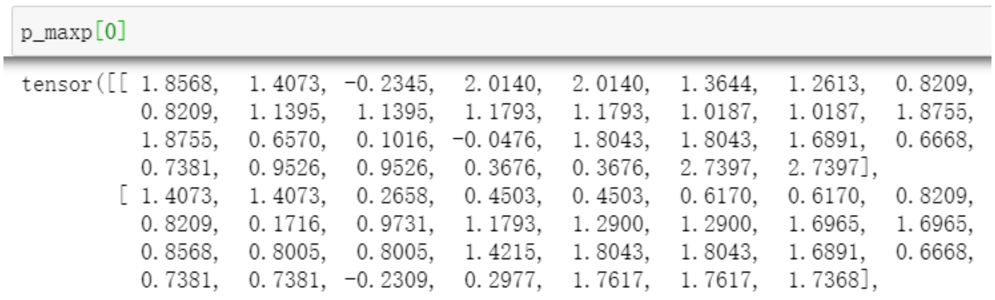
Maxpooling will choose the max value in a limited kernel size shows above and the size of the output can be calculated by the equation shows below.

- Input: (N, C, H_{in}, W_{in})
- Output: (N, C, H_{out}, W_{out}), where

$$H_{out} = \left\lfloor \frac{H_{in} + 2 * \text{padding}[0] - \text{dilation}[0] \times (\text{kernel\_size}[0] - 1) - 1}{\text{stride}[0]} + 1 \right\rfloor$$

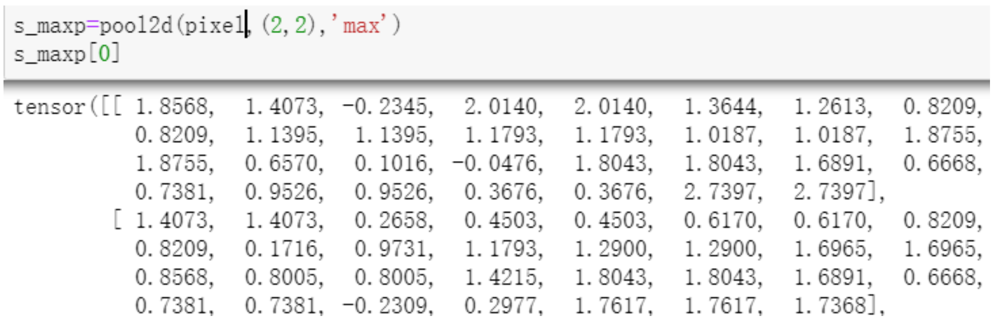
$$W_{out} = \left\lfloor \frac{W_{in} + 2 * \text{padding}[1] - \text{dilation}[1] \times (\text{kernel\_size}[1] - 1) - 1}{\text{stride}[1]} + 1 \right\rfloor$$

As for this question, the output size is (3,31,31). Then the result is shown below, top graph is from pytorch and the down graph is from scratch.(only the first channel shows here for comparison), the result is totally the same in 4 decimal places.



```
p_maxp[0]
```

```
tensor([[ 1.8568,  1.4073, -0.2345,  2.0140,  2.0140,  1.3644,  1.2613,  0.8209,
          0.8209,  1.1395,  1.1395,  1.1793,  1.1793,  1.0187,  1.0187,  1.8755,
          1.8755,  0.6570,  0.1016, -0.0476,  1.8043,  1.8043,  1.6891,  0.6668,
          0.7381,  0.9526,  0.9526,  0.3676,  0.3676,  2.7397,  2.7397],
        [ 1.4073,  1.4073,  0.2658,  0.4503,  0.4503,  0.6170,  0.6170,  0.8209,
          0.8209,  0.1716,  0.9731,  1.1793,  1.2900,  1.2900,  1.6965,  1.6965,
          0.8568,  0.8005,  0.8005,  1.4215,  1.8043,  1.8043,  1.6891,  0.6668,
          0.7381,  0.7381, -0.2309,  0.2977,  1.7617,  1.7617,  1.7368],
```

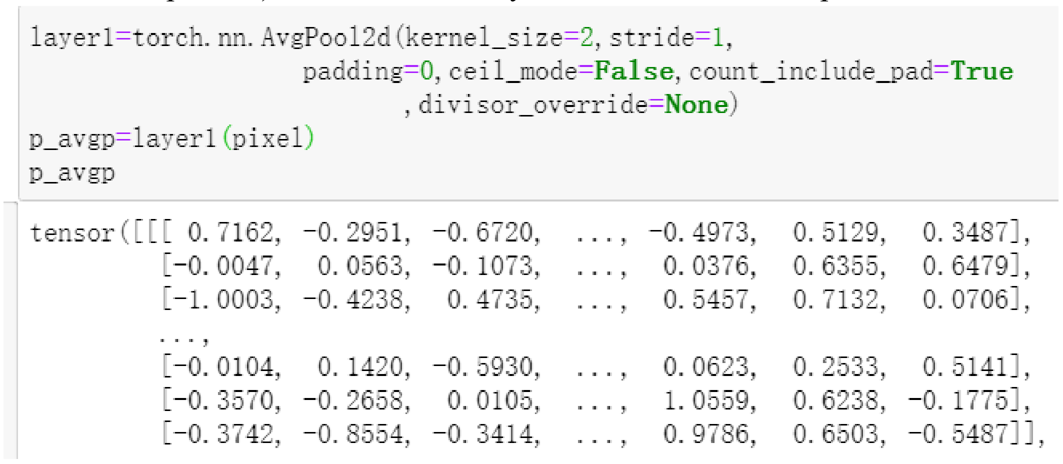


```
s_maxp=pool2d(pixel, (2,2),'max')
s_maxp[0]
```

```
tensor([[ 1.8568,  1.4073, -0.2345,  2.0140,  2.0140,  1.3644,  1.2613,  0.8209,
          0.8209,  1.1395,  1.1395,  1.1793,  1.1793,  1.0187,  1.0187,  1.8755,
          1.8755,  0.6570,  0.1016, -0.0476,  1.8043,  1.8043,  1.6891,  0.6668,
          0.7381,  0.9526,  0.9526,  0.3676,  0.3676,  2.7397,  2.7397],
        [ 1.4073,  1.4073,  0.2658,  0.4503,  0.4503,  0.6170,  0.6170,  0.8209,
          0.8209,  0.1716,  0.9731,  1.1793,  1.2900,  1.2900,  1.6965,  1.6965,
          0.8568,  0.8005,  0.8005,  1.4215,  1.8043,  1.8043,  1.6891,  0.6668,
          0.7381,  0.7381, -0.2309,  0.2977,  1.7617,  1.7617,  1.7368],
```

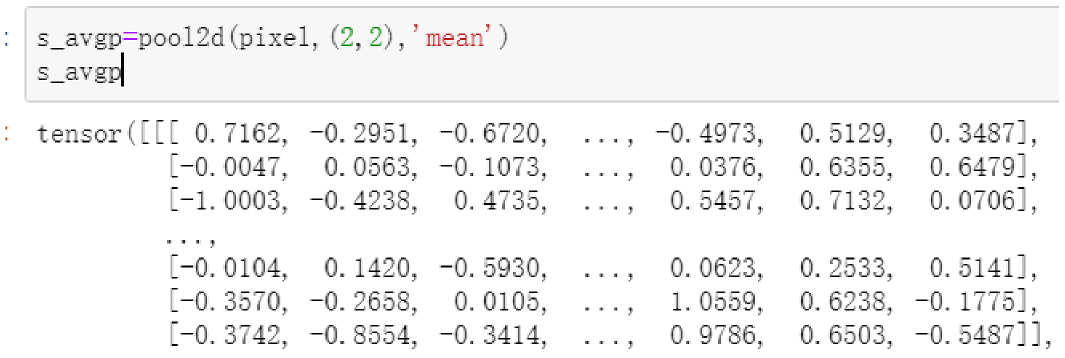
2.AvgPooling

The mechanism of AvgPooling is just like Maxpooling, and the only different way is the output is calculated by average from the kernel size. The result shows below, top graph is from pytorch and the down graph is from scratch.(only the first channel shows here for comparison), the result is totally the same in 4 decimal places.



```
layer1=torch.nn.AvgPool2d(kernel_size=2,stride=1,
                              padding=0,ceil_mode=False,count_include_pad=True
                                       ,divisor_override=None)
p_avgp=layer1(pixel)
p_avgp
```

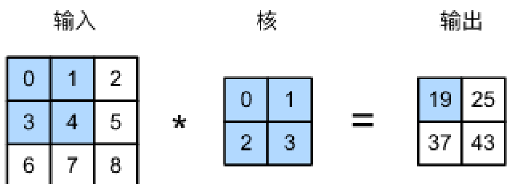
```
tensor([[[ 0.7162, -0.2951, -0.6720,  ..., -0.4973,  0.5129,  0.3487],
         [-0.0047,  0.0563, -0.1073,  ...,  0.0376,  0.6355,  0.6479],
         [-1.0003, -0.4238,  0.4735,  ...,  0.5457,  0.7132,  0.0706],
         ...,
         [-0.0104,  0.1420, -0.5930,  ...,  0.0623,  0.2533,  0.5141],
         [-0.3570, -0.2658,  0.0105,  ...,  1.0559,  0.6238, -0.1775],
         [-0.3742, -0.8554, -0.3414,  ...,  0.9786,  0.6503, -0.5487]],
```



```
s_avgp=pool2d(pixel,(2,2),'mean')
s_avgp
```

```
tensor([[[ 0.7162, -0.2951, -0.6720,  ..., -0.4973,  0.5129,  0.3487],
         [-0.0047,  0.0563, -0.1073,  ...,  0.0376,  0.6355,  0.6479],
         [-1.0003, -0.4238,  0.4735,  ...,  0.5457,  0.7132,  0.0706],
         ...,
         [-0.0104,  0.1420, -0.5930,  ...,  0.0623,  0.2533,  0.5141],
         [-0.3570, -0.2658,  0.0105,  ...,  1.0559,  0.6238, -0.1775],
         [-0.3742, -0.8554, -0.3414,  ...,  0.9786,  0.6503, -0.5487]],
```

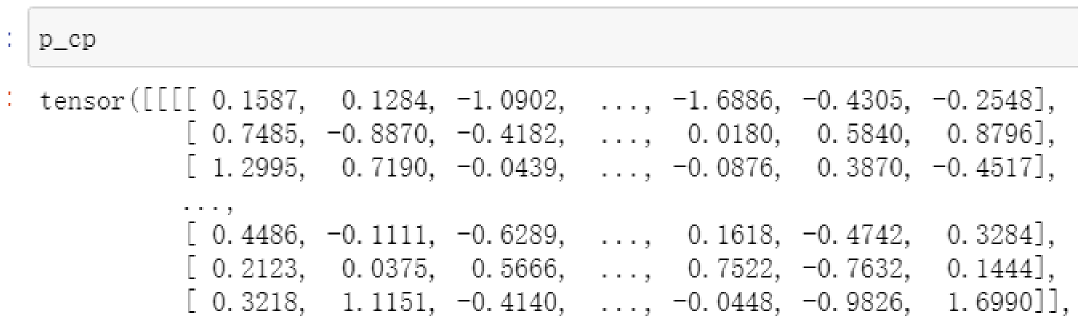
3.Conv2d-stride=1



Conv2d is used to calculate the sum between the input and kernel, the kernel is like the weight, so the function is to do element-wise product between input and kernel, the formula is show below.

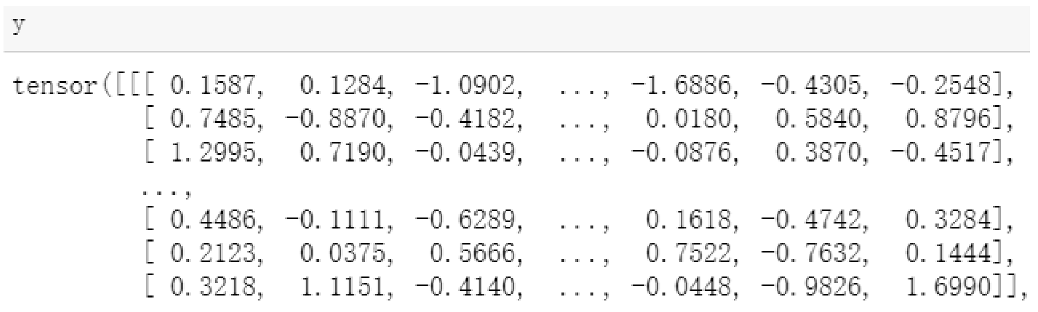
$$\text{out}(N_i, C_{\text{out}_j}) = \text{bias}(C_{\text{out}_j}) + \sum_{k=0}^{C_{\text{in}}-1} \text{weight}(C_{\text{out}_j}, k) \star \text{input}(N_i, k)$$

And the stride means the step kernel move to right and down. Then the result shows below. the result is totally the same in 4 decimal places.



```
p_cp
```

```
tensor([[[[ 0.1587,   0.1284,  -1.0902,  ...,  -1.6886,  -0.4305,  -0.2548],
          [ 0.7485,  -0.8870,  -0.4182,  ...,   0.0180,   0.5840,   0.8796],
          [ 1.2995,   0.7190,  -0.0439,  ...,  -0.0876,   0.3870,  -0.4517],
          ...,
          [ 0.4486,  -0.1111,  -0.6289,  ...,   0.1618,  -0.4742,   0.3284],
          [ 0.2123,   0.0375,   0.5666,  ...,   0.7522,  -0.7632,   0.1444],
          [ 0.3218,   1.1151,  -0.4140,  ...,  -0.0448,  -0.9826,   1.6990]],
```

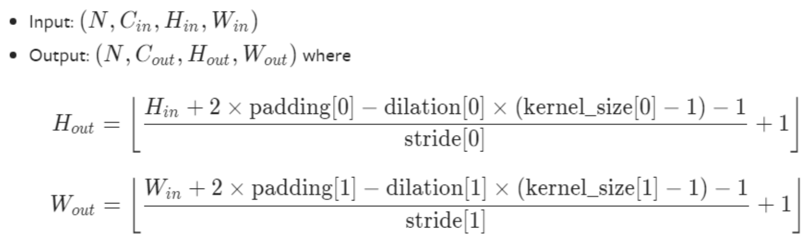


```
y
```

```
tensor([[[ 0.1587,   0.1284,  -1.0902,  ...,  -1.6886,  -0.4305,  -0.2548],
         [ 0.7485,  -0.8870,  -0.4182,  ...,   0.0180,   0.5840,   0.8796],
         [ 1.2995,   0.7190,  -0.0439,  ...,  -0.0876,   0.3870,  -0.4517],
         ...,
         [ 0.4486,  -0.1111,  -0.6289,  ...,   0.1618,  -0.4742,   0.3284],
         [ 0.2123,   0.0375,   0.5666,  ...,   0.7522,  -0.7632,   0.1444],
         [ 0.3218,   1.1151,  -0.4140,  ...,  -0.0448,  -0.9826,   1.6990]],
```

4.Conv2d-stride=2

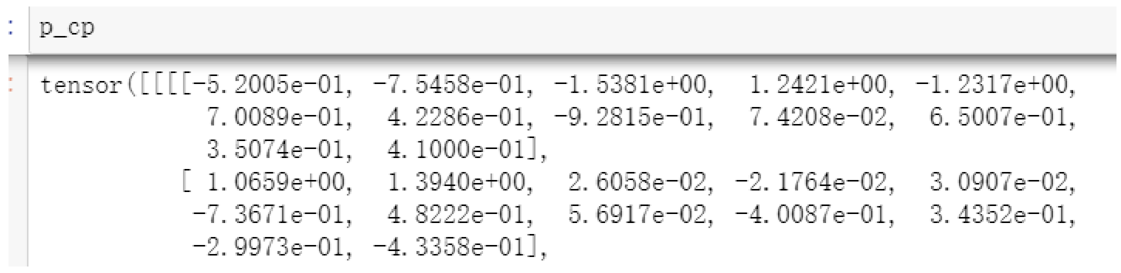
When the stride changes, the size of output will also change based on the formula shows below, and the other is the same of the previous question. Then the results between pytorch and scratch are also the same in 4 decimal places.



- Input: $(N, C_{in}, H_{in}, W_{in})$
- Output: $(N, C_{out}, H_{out}, W_{out})$ where

$$H_{out} = \left\lfloor \frac{H_{in} + 2 \times \text{padding}[0] - \text{dilation}[0] \times (\text{kernel\_size}[0] - 1) - 1}{\text{stride}[0]} + 1 \right\rfloor$$

$$W_{out} = \left\lfloor \frac{W_{in} + 2 \times \text{padding}[1] - \text{dilation}[1] \times (\text{kernel\_size}[1] - 1) - 1}{\text{stride}[1]} + 1 \right\rfloor$$



```
p_cp
```

```
tensor([[[[-5.2005e-01, -7.5458e-01, -1.5381e+00,  1.2421e+00, -1.2317e+00,
            7.0089e-01,  4.2286e-01, -9.2815e-01,  7.4208e-02,  6.5007e-01,
            3.5074e-01,  4.1000e-01],
          [ 1.0659e+00,  1.3940e+00,  2.6058e-02, -2.1764e-02,  3.0907e-02,
           -7.3671e-01,  4.8222e-01,  5.6917e-02, -4.0087e-01,  3.4352e-01,
           -2.9973e-01, -4.3358e-01],
s_cp=poo2d_multi(pixel, layer1.weight, layer1.bias, 6, 2, 2)
s_cp
```

```
tensor([[[[-5.2005e-01, -7.5458e-01, -1.5381e+00,  1.2421e+00, -1.2317e+00,
            7.0089e-01,  4.2286e-01, -9.2815e-01,  7.4208e-02,  6.5007e-01,
            3.5074e-01,  4.1000e-01],
          [ 1.0659e+00,  1.3940e+00,  2.6058e-02, -2.1764e-02,  3.0907e-02,
           -7.3671e-01,  4.8222e-01,  5.6917e-02, -4.0087e-01,  3.4352e-01,
           -2.9973e-01, -4.3358e-01],
```

5.ConvTranSpose2d

The transposed convolution is to make the size of output larger than input. The mechanism of transpose convolution in pytorch is show below, every value in input will make an element-wise product with the kernel, and then put the result in the larger output, then sum the value in same corresponding places, then the result is totally the same.

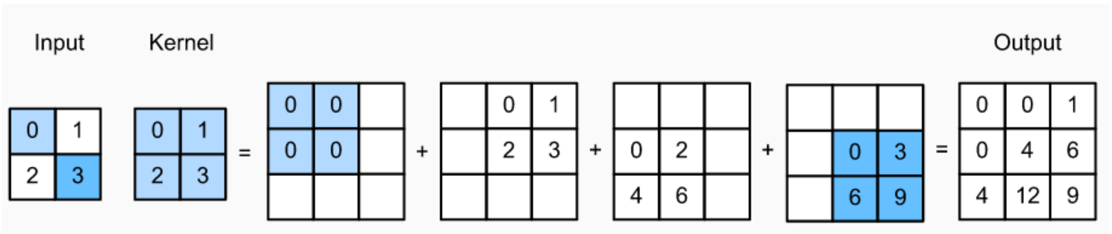


Fig. 13.10.1 Transposed convolution layer with a 2×2 kernel.

```
layer1=torch.nn.ConvTranspose2d(in_channels=3, out_channels=4, kernel_s
                                stride=1, padding=0, output_padding=0, groups=
p_ct=layer1(pixel.view(-1, 3, 32, 32))
```

```
p_ct
```

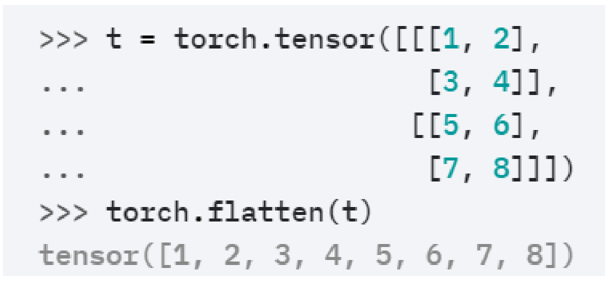
```
tensor([[[[ 5.1801e-01,  2.9314e-02, -3.1701e-01,  ..., -3.7691e-01,
           -1.5012e-01,  2.0831e-01],
          [ 2.4386e-01,  1.4466e-01,  2.4619e-01,  ...,  1.3239e-01,
            3.5772e-01, -2.7334e-01],
          [ 2.1303e-01,  6.5138e-02, -1.0414e-02,  ..., -2.8771e-01,
            3.2126e-01, -4.0548e-01],
```

```
p_conv_trans=poo2d_multi(pixel, layer1.weight, layer1.bias, 4)
p_conv_trans
```

```
tensor([[[[ 5.1801e-01,  2.9314e-02, -3.1701e-01,  ..., -3.7691e-01,
           -1.5012e-01,  2.0831e-01],
          [ 2.4386e-01,  1.4466e-01,  2.4619e-01,  ...,  1.3239e-01,
            3.5772e-01, -2.7334e-01],
          [ 2.1303e-01,  6.5138e-02, -1.0414e-02,  ..., -2.8771e-01,
            3.2126e-01, -4.0548e-01],
```

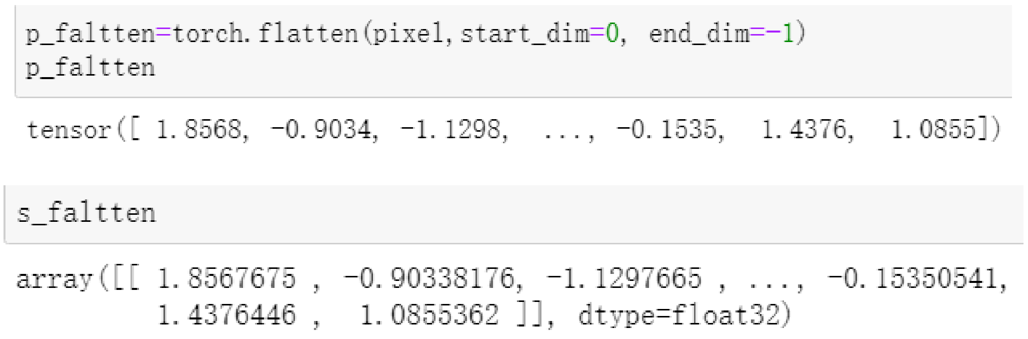
6.Flatten

The flatten function is to flatten all the numbers in different channels and make every different row into one row. The function sample can be seen here.



```
>>> t = torch.tensor([[[1, 2],
...                    [3, 4]],
...                   [[5, 6],
...                    [7, 8]]])
>>> torch.flatten(t)
tensor([1, 2, 3, 4, 5, 6, 7, 8])
```

Then the result can be seen here, the results are also the same in 4 decimal places.



```
p_faltten=torch.flatten(pixel,start_dim=0, end_dim=-1)
p_faltten
```

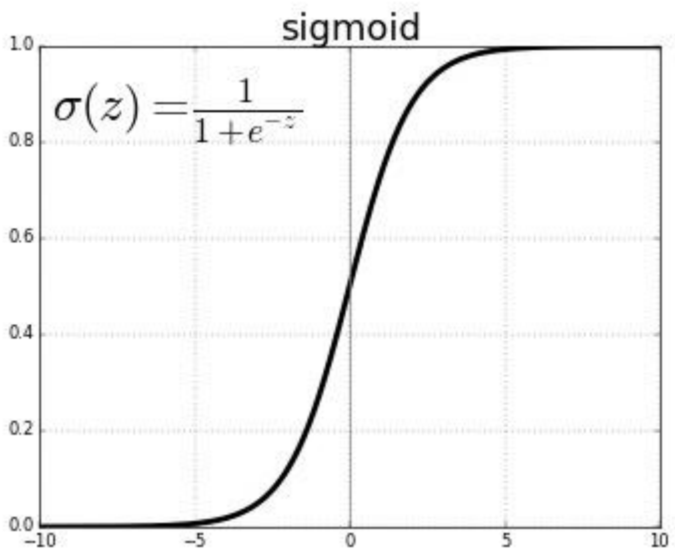
```
tensor([ 1.8568, -0.9034, -1.1298,  ..., -0.1535,  1.4376,  1.0855])
```

```
s_faltten
```

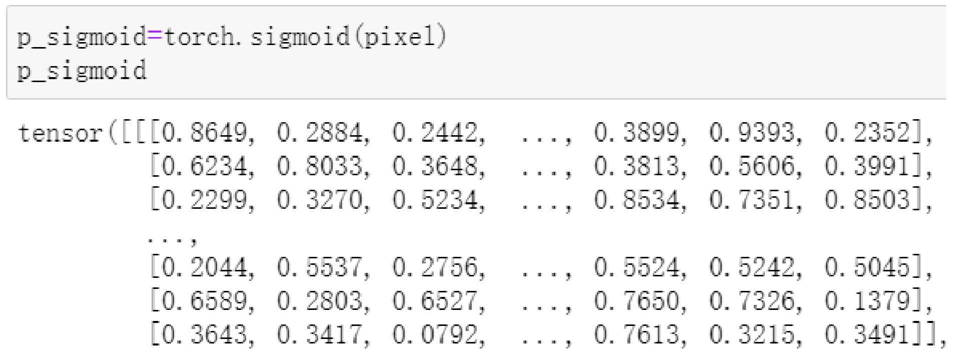
```
array([[ 1.8567675 , -0.90338176, -1.1297665 , ..., -0.15350541,
         1.4376446 ,  1.0855362 ]], dtype=float32)
```

7. Sigmoid

Sigmoid function is well-known in machine learning field. And the function of it shows below, the scratch codes only generate the formula and input the pixel as z.

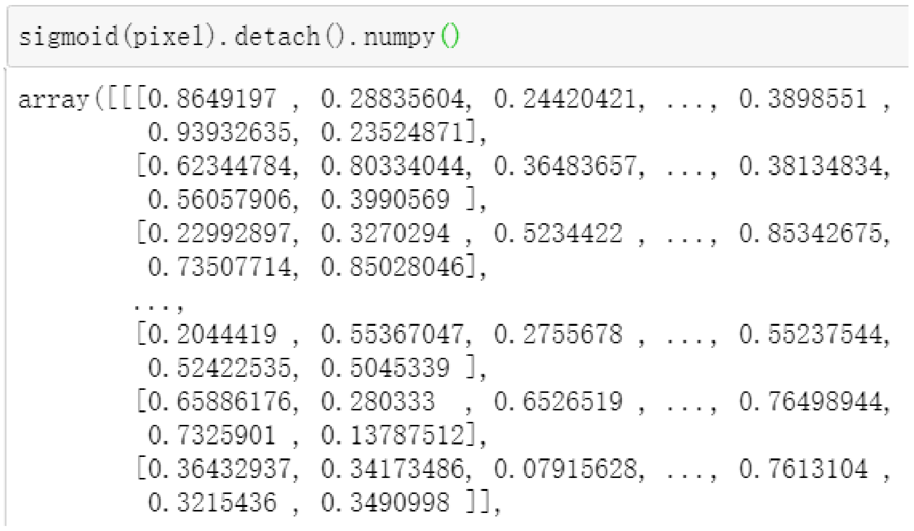


Then the results below between scratch and pytorch.



```
p_sigmoid=torch.sigmoid(pixel)
p_sigmoid
```

```
tensor([[[0.8649, 0.2884, 0.2442,  ..., 0.3899, 0.9393, 0.2352],
         [0.6234, 0.8033, 0.3648,  ..., 0.3813, 0.5606, 0.3991],
         [0.2299, 0.3270, 0.5234,  ..., 0.8534, 0.7351, 0.8503],
         ...,
         [0.2044, 0.5537, 0.2756,  ..., 0.5524, 0.5242, 0.5045],
         [0.6589, 0.2803, 0.6527,  ..., 0.7650, 0.7326, 0.1379],
         [0.3643, 0.3417, 0.0792,  ..., 0.7613, 0.3215, 0.3491]],
```

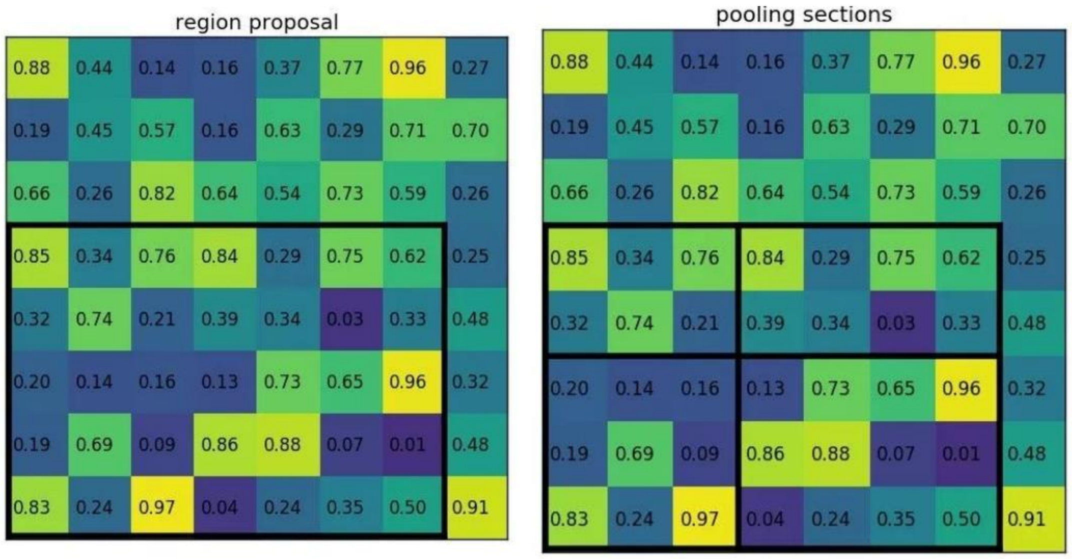


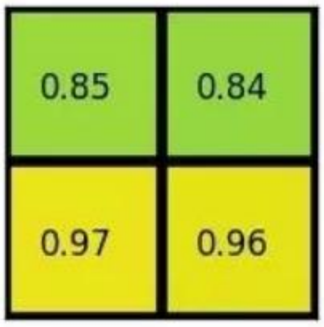
```
sigmoid(pixel).detach().numpy()
```

```
array([[[0.8649197 , 0.28835604, 0.24420421, ..., 0.3898551 ,
          0.93932635, 0.23524871],
         [0.62344784, 0.80334044, 0.36483657, ..., 0.38134834,
          0.56057906, 0.3990569 ],
         [0.22992897, 0.3270294 , 0.5234422 , ..., 0.85342675,
          0.73507714, 0.85028046],
         ...,
         [0.2044419 , 0.55367047, 0.2755678 , ..., 0.55237544,
          0.52422535, 0.5045339 ],
         [0.65886176, 0.280333  , 0.6526519 , ..., 0.76498944,
          0.7325901 , 0.13787512],
         [0.36432937, 0.34173486, 0.07915628, ..., 0.7613104 ,
          0.3215436 , 0.3490998 ]],
```

8. Roi-Pooling

Roi-pooling is used to make different size of graph into the same size. Because the input of the Nerual Network is always fixed and the sizes of dataset sometimes are different. Just like the sample show below, first choose the range we need and split the set into the dimension we use, here is (2,2). The split the range into 4 pieces, finally extract the maximum number in each piece to generate the input of the neural network.





Then the result of pytorch and scratch shows below, it can be seen that the result is totally the same in 4 decimal places.

```
inp=pixel.view(-1,3,32,32)
box=torch.tensor([[0,0,0,20,20]]).float()
p_roi=torchvision.ops.roi_pool(inp,box,output_size=(6,6))
p_roi
```

```
tensor([[[[1.8568,  2.0140,  1.1395,  1.2900,  1.8755,  1.4215],
          [1.3422,  1.9971,  0.9666,  2.2221,  1.8594,  2.6306],
          [1.6153,  1.7532,  1.9006,  1.9436,  1.4912,  2.6692],
          [3.7132,  3.7132,  1.2993,  1.7878,  1.4880,  1.4880],
          [1.3881,  1.5390,  2.4791,  0.5354,  1.4634,  2.0017],
          [1.3881,  2.4800,  2.4791,  1.6747,  1.4634,  2.7278]],
:  s_poi=np.array(_result)
   s_poi[0]
```

```
:  array([[1.8567675 , 2.0140462 , 1.1395307 , 1.2899902 , 1.875452  ,
           1.4215219 ],
          [1.3421711 , 1.9970928 , 0.96655166, 2.2220724 , 1.8593836 ,
           2.630614  ],
          [1.6153022 , 1.7531915 , 1.9006094 , 1.9435998 , 1.4912332 ,
           2.6691873 ],
          [3.7132497 , 3.7132497 , 1.2992598 , 1.7877973 , 1.488033  ,
           1.488033  ],
          [1.3881211 , 1.5389556 , 2.479112  , 0.5354247 , 1.463366  ,
           2.00165   ],
          [1.3881211 , 2.479967  , 2.479112  , 1.6747336 , 1.463366  ,
           2.7278185 ]], dtype=float32)
```

9. Batch-norm

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

Where E(x) is the mean and Var[x] is the variance, γ and β are default set to 1 and 0 respectively.

Then the result of pytorch and scratch shows below, it can be seen that the result is totally the same in 4 decimal places.

```
p_batch_norm
```

```
tensor([[[[ 1.0714, -0.5221, -0.6528,  ..., -0.2592,  1.5812, -0.6812],
          [ 0.2905,  0.8119, -0.3207,  ..., -0.2799,  0.1400, -0.2369],
          [-0.6984, -0.4172,  0.0536,  ...,  1.0166,  0.5886,  1.0022],
          ...,
          [-0.7851,  0.1238, -0.5586,  ...,  0.1208,  0.0554,  0.0099],
          [ 0.3794, -0.5449,  0.3636,  ...,  0.6808,  0.5813, -1.0589],
          [-0.3219, -0.3791, -1.4173,  ...,  0.6691, -0.4317, -0.3603]],
```

```
norm(pixel)
```

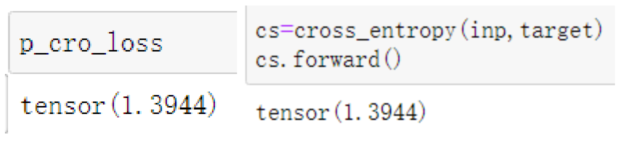
```
tensor([[[ 1.0714, -0.5221, -0.6528,  ..., -0.2592,  1.5812, -0.6812],
         [ 0.2905,  0.8119, -0.3207,  ..., -0.2799,  0.1400, -0.2369],
         [-0.6984, -0.4172,  0.0536,  ...,  1.0166,  0.5886,  1.0022],
         ...,
         [-0.7851,  0.1238, -0.5586,  ...,  0.1208,  0.0554,  0.0099],
         [ 0.3794, -0.5449,  0.3636,  ...,  0.6808,  0.5813, -1.0589],
         [-0.3219, -0.3791, -1.4173,  ...,  0.6691, -0.4317, -0.3603]],
```

10. Cross-entropy

Cross-entropy is the function which aggregates soft-max, logarithm and negative log likelihood loss (nll loss). Soft max is carried out in different channels, and nll-loss is the loss got from the corresponding class(channel) which the target provides. The total formula is show below.

$$\operatorname{loss}(x, class) = -\log\left(\frac{\exp(x[class])}{\sum_j \exp(x[j])}\right)$$

Then the result (left is got from pytorch, right from scratch) is totally the same.



```
p_cro_loss

tensor(1.3944)
```

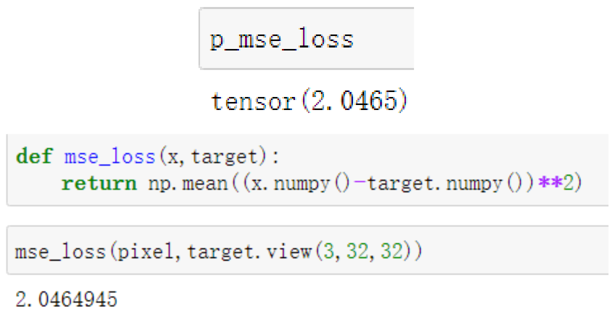
```
cs=cross_entropy(inp, target)
cs.forward()

tensor(1.3944)
```

11. Mse-Loss

Mse loss is the mean square error loss, the formula shows below:

$$MSELoss = \frac{\sum_{i=1}^{m}(x_i - y_i)^2}{m}$$

The result can be seen below, it is the same in 4 decimal places.



```
p_mse_loss

tensor(2.0465)
```

```
def mse_loss(x, target):
    return np.mean((x.numpy()-target.numpy())**2)
```

```
mse_loss(pixel, target.view(3, 32, 32))

2.0464945
```