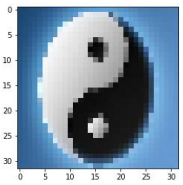
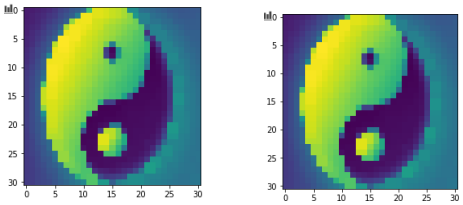
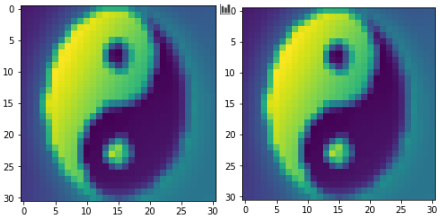
We resize the right-side image as our test image, which is resized as 32*32*3.



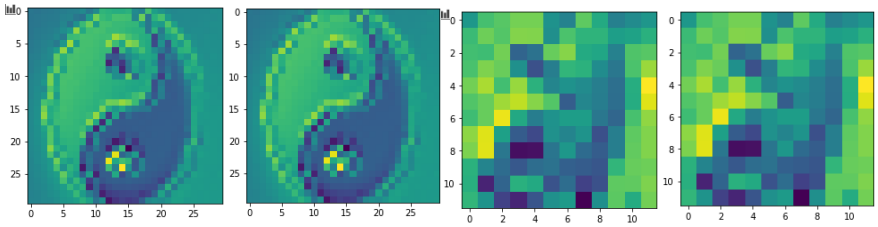
1.1 Max Pool 2D: The original function will do a max pool for all the 3 channels of the image. The output image is as the below left. We then implement the Max Pool using numpy array, and the result is as the below right. Identified by program, they are the same.



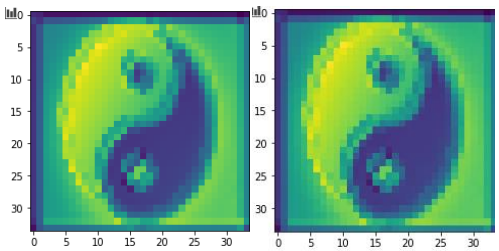
1.2 Average Pool 2D: The operation for this function is similar to Max Pool, the only difference is we are only extracting the average here other than the maxinum. The below left is the system output, while the below right is the ouput from our self defined function, which were identified that they are the same by program.



1.3 & 1.4 Conv 2D: The operation for this function is to do the convolutional tranformation for the image. The difference between the convolutional transformation between 1.3 & 1.4 are the kernel size, which controlling the kernel matrix to process the image; the stride, which is the step the kernel move on each calculation on the image; and the dilation, which controlling the gap between image points to be multiply to the corresponding point in the kernel matrix. The below 1st & 2nd images are the example of the output for **1.3,** whcih from the program & our scratch, locating at left & right respectively. The 3rd & 4th images are generated for 1.4, from pytorch and our self defined method respecitvely. The program has indentified that the difference between both pairs of images are all below 0.00013(considering as the small numerical value).

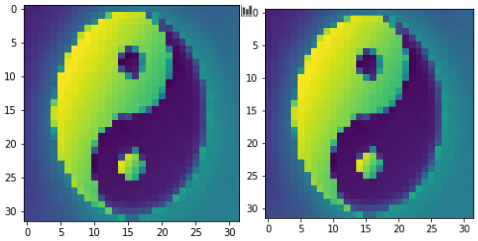


1.5 Conv Transpose 2D: This operation is similar as the reverse of the normal convolutional process. The below left image is one example generated by the program, while the right is generated by our self defined program. Both of the images have been identified to have the numerical diference lower than 0.00013.



2.1 Flatten: The process here is just to resize our original input image to 1*3072, aranging the value by sequence. Based on that, we have self defined a method with the same principle, and generated the exactly the same result.

2.2 Sigmoid: The process here is just to transform each pixel in the input image using Sigmoid activation function. The below left image is one example generated by the program, while the right is generated by our self defined program. Both of the images have been identified to have the numerical diference lower than 0.00013.



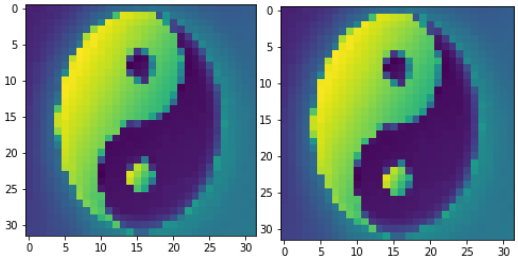
2.3 ROI Pooling: The process here is to pool a selected region in the image to our expected size. The principle of the operation in pytorch is assign the value of the element at the 1st column & the last row in our selected region, to each pixel of our expected new-sized image for each channel. The left sied is the example to generate a 4*4 new sized image. Our self defined program has been implemented based on the sampe principle.

```
[ 58., 112., 159.]
[ 58., 112., 159.]
[ 58., 112., 159.]
[ 58., 112., 159.]
```

2.4 Batch Normalization: The process here is exactly the same principle as the Mini-Batch Normalization. The formula is as shown at the right side.

$$\hat{x}_i^{(k)} = \frac{x_i^{(k)} - \mu_B^{(k)}}{\sqrt{{\sigma_B^{(k)}}^2 + \epsilon}},$$

For the example, the below left image is one example generated by the program, while the right is generated by our self defined program. Both of the images have been identified to have the numerical diference lower than 0.00013.



2.5 & 2.6 Cross Entropy Loss & MSE Loss: The cross entropy loss in pytorch is actually the log softmax, having the formula as:

$$\text{LogSoftmax}(x_i) = \log\left(\frac{\exp(x_i)}{\sum_j \exp(x_j)}\right)$$

The MSE loss in pytorch here have the same similar as Least Square:

$$\ell(x, y) = L = \{l_1, \ldots, l_N\}^\top, \quad l_n = (x_n - y_n)^2$$

We have implemented the calculation using the self defined methods, and generated random number as the expected value for testing purposes. The cross entropy loss in our case is calculated approximately as 1.1025 for both pytorch function and ours, and the MSE loss is calculated approximately as 1.0483 for both pytorch function and ours.