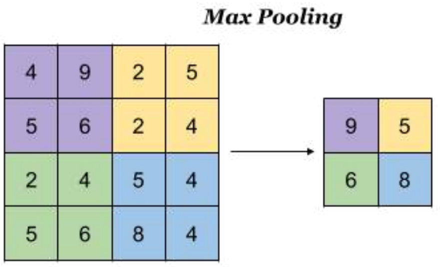
Question #1

1. **Max pooling2d**

   Max pooling is a sample-based discretization process. It calculates the max for each patch of the feature map.



$$out(N_i, C_j, h, w) = \max_{m=0,\ldots,kH-1} \max_{n=0,\ldots,kW-1} \text{input}(N_i, C_j, \text{stride}[0] \times h + m, \text{stride}[1] \times w + n)$$

```
maxpool2d=torch.nn.MaxPool2d(kernel_size=2, stride=1, padding=0,dilation=1, return_indices=False, ceil_mode=False)
torch_out=maxpool2d(Input)
torch_out
```

```
tensor([[[[ 1.0896,  0.8189,  0.8189,  ...,  1.0567,  0.5392,  0.8445],
          [-0.3085,  0.8189,  0.8189,  ...,  1.0567,  0.5392, -0.1451],
          [ 0.6613,  0.6613,  1.1431,  ...,  0.0303,  0.0303, -0.3016],
          ...,
          [ 0.7819,  0.7010,  0.4635,  ...,  0.5238,  1.3032,  1.3032],
          [ 0.7819,  0.4402,  0.2553,  ...,  2.3323,  2.3323,  0.1204],
          [ 1.4825,  1.0374,  0.3730,  ...,  2.3323,  2.3323,  0.8359]],

         [[ 0.8987,  0.8987,  0.5293,  ...,  2.3037,  1.1128,  1.3648],
          [ 0.7848,  0.7848,  0.7252,  ...,  2.3037,  1.1128, -0.0831],
          [ 0.8412,  0.8179,  0.7252,  ...,  1.5340,  0.8336,  0.8336],
```

```
my_maxpool2d=mymaxpool2d(Input,2,1)
my_maxpool2d
```

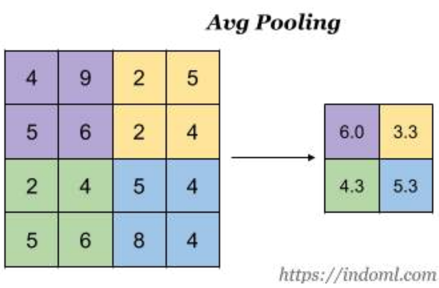
```
out_shape (31, 31)

tensor([[[[ 1.0896,  0.8189,  0.8189,  ...,  1.0567,  0.5392,  0.8445],
          [-0.3085,  0.8189,  0.8189,  ...,  1.0567,  0.5392, -0.1451],
          [ 0.6613,  0.6613,  1.1431,  ...,  0.0303,  0.0303, -0.3016],
          ...,
          [ 0.7819,  0.7010,  0.4635,  ...,  0.5238,  1.3032,  1.3032],
          [ 0.7819,  0.4402,  0.2553,  ...,  2.3323,  2.3323,  0.1204],
          [ 1.4825,  1.0374,  0.3730,  ...,  2.3323,  2.3323,  0.8359]],

         [[ 0.8987,  0.8987,  0.5293,  ...,  2.3037,  1.1128,  1.3648],
          [ 0.7848,  0.7848,  0.7252,  ...,  2.3037,  1.1128, -0.0831],
          [ 0.8412,  0.8179,  0.7252,  ...,  1.5340,  0.8336,  0.8336],
```

2. **Average pooling**

   Average pooling involves calculating the average for each patch of the feature map.



```
avgpool2d=torch.nn.AvgPool2d(kernel_size=2, stride=1, padding=0,ceil_mode=False, count_include_pad=True,divi
torch_out=avgpool2d(Input)
torch_out
```

```
tensor([[[[-0.1664, -0.0071, -0.0327,  ..., -0.2182, -0.5818, -0.2778],
          [-1.0798, -0.5358, -0.5394,  ...,  0.2105, -0.2352, -0.4892],
          [-0.6287, -0.4299, -0.2049,  ..., -0.7921, -0.9628, -0.9398],
          ...,
          [ 0.2801, -0.2668, -0.8818,  ..., -0.2651, -0.0708,  0.1949],
          [-0.2070, -0.4623, -0.6626,  ...,  0.6123, -0.1291, -0.5143],
          [ 0.3845,  0.2588,  0.0465,  ...,  0.5649,  0.2483, -0.1907]],
```

```
my_avgpool2d=myavgpool2d(Input,2,1)
my_avgpool2d
```

```
out_shape (31, 31)

tensor([[[[-0.1664, -0.0071, -0.0327,  ..., -0.2182, -0.5818, -0.2778],
          [-1.0798, -0.5358, -0.5394,  ...,  0.2105, -0.2352, -0.4892],
          [-0.6287, -0.4299, -0.2049,  ..., -0.7921, -0.9628, -0.9398],
          ...,
          [ 0.2801, -0.2668, -0.8818,  ..., -0.2651, -0.0708,  0.1949],
          [-0.2070, -0.4623, -0.6626,  ...,  0.6123, -0.1291, -0.5143],
          [ 0.3845,  0.2588,  0.0465,  ...,  0.5649,  0.2483, -0.1907]],
```

3. **Conv2d-stride=1**

   This layer creates a convolution kernel that is convolved with the layer input to produce a tensor of outputs.

```
Conv2d=torch.nn.Conv2d(in_channels=3, out_channels=6,kernel_size=3, stride=1, padding=0, dilation=1,
                 groups=1,bias=True, padding_mode='zeros')
torch_out=Conv2d(Input)
torch_out
```

```
tensor([[[[ 0.1752, -0.2181,  0.7335,  ...,  0.0169,  1.1395,  0.7046],
          [-0.4693, -0.0181, -1.2356,  ...,  1.2473, -0.6768, -0.1339],
          [ 0.6977,  0.3976,  0.4361,  ..., -0.4231, -0.6272, -0.3523],
          ...,
          [ 0.6289,  0.7697, -0.6303,  ..., -0.4405,  0.9620,  0.1331],
          [-0.4837, -0.3553,  0.2641,  ..., -0.3386,  0.5779,  0.5719],
          [-0.0796, -0.0370, -0.0980,  ...,  1.8525, -0.9600, -0.2636]],
```

```
my_conv2d = myconv2d(Input, conv_weight, conv_bias, 1)
my_conv2d
```

```
tensor([[[[ 0.1752, -0.2181,  0.7335,  ...,  0.0169,  1.1395,  0.7046],
          [-0.4693, -0.0181, -1.2356,  ...,  1.2473, -0.6768, -0.1339],
          [ 0.6977,  0.3976,  0.4361,  ..., -0.4231, -0.6272, -0.3523],
          ...,
          [ 0.6289,  0.7697, -0.6303,  ..., -0.4405,  0.9620,  0.1331],
          [-0.4837, -0.3553,  0.2641,  ..., -0.3386,  0.5779,  0.5719],
          [-0.0796, -0.0370, -0.0980,  ...,  1.8525, -0.9600, -0.2636]],
```

4. **Conv2d-Stride=2**

```
Conv2d_2=torch.nn.Conv2d(in_channels=3, out_channels=6,kernel_size=5, stride=2, padding=0, dilation=2, groups=1,
bias=True, padding_mode='zeros')
torch_out=Conv2d_2(Input)
torch_out
```

```
tensor([[[[-2.0293e-01, -1.0989e+00, -7.6421e-01, -5.7097e-02, -6.1847e-01,
           -3.2147e-01, -8.2310e-01, -1.5520e+00, -1.0093e+00, -2.2810e-01,
            9.2761e-02, -2.0570e-01],
          [ 8.1542e-01,  5.5698e-01,  2.8511e-01, -1.1589e+00, -1.0172e+00,
            5.4716e-01, -1.9442e-02,  4.4153e-01, -1.0483e+00,  1.1606e-01,
           -4.4759e-01, -8.8545e-02],
          [-4.4568e-01, -1.0027e+00,  2.7096e-01,  9.8192e-02, -3.0344e-01,
           -2.3090e-01, -6.0859e-01, -6.0224e-02, -7.3599e-01, -3.7175e-01,
            2.1801e-01, -1.7855e-01],
```

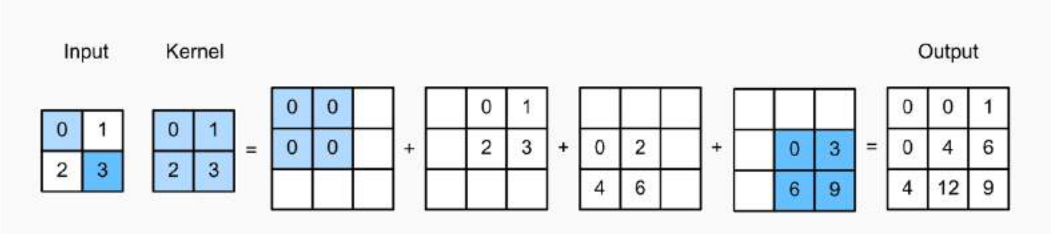
```
myconv2D = myconv2d_2(Input,3, 6,2,  conv_bias,  2)

print(myconv2D)
```

```
tensor([[[[-2.0293e-01, -1.0989e+00, -7.6421e-01, -5.7097e-02, -6.1847e-01,
           -3.2147e-01, -8.2310e-01, -1.5520e+00, -1.0093e+00, -2.2810e-01,
            9.2761e-02, -2.0570e-01],
          [ 8.1542e-01,  5.5698e-01,  2.8511e-01, -1.1589e+00, -1.0172e+00,
            5.4716e-01, -1.9442e-02,  4.4153e-01, -1.0483e+00,  1.1606e-01,
           -4.4759e-01, -8.8545e-02],
          [-4.4568e-01, -1.0027e+00,  2.7096e-01,  9.8192e-02, -3.0344e-01,
           -2.3090e-01, -6.0859e-01, -6.0224e-02, -7.3599e-01, -3.7175e-01,
            2.1801e-01, -1.7855e-01],
```

5. Transpose 2d

Compared to convolutions that reduce through kernels, transposed convolutions broadcast inputs.
If a convolution layer reduces the input width and height by nw and hh time, respectively. Then a transposed convolution layer with the same kernel sizes, padding and strides will increase the input width and height by nw and nh, respectively




```
transpose2d=torch.nn.ConvTranspose2d(in_channels=3, out_channels=4,kernel_size=3, stride=1, padding=0, output_padding=
groups=1, bias=True, dilation=1, padding_mode='zeros')
torch_out=transpose2d(Input)
torch_out
```

```
tensor([[[[ 0.1184, -0.0216,  0.5006,  ...,  0.0365, -0.0865,  0.0470],
          [-0.2145, -0.3770,  0.0592,  ..., -0.0968,  0.1344,  0.2888],
          [-0.0900,  0.0140,  0.6085,  ..., -0.8772, -0.0056, -0.4345],
          ...,
          [-0.2172, -0.1211, -0.0900,  ...,  0.0951,  0.1038, -0.0617],
          [-0.3840, -0.1787, -0.4393,  ..., -0.3100, -0.5051, -0.1049],
          [-0.3018, -0.2052,  0.0081,  ...,  0.3402,  0.0176, -0.0979]],
```

```
myconv2D = mytranspose2d(3, 4, 1,conv_bias, 1,Input)

print(myconv2D)
```

```
tensor([[[[ 0.1184, -0.0216,  0.5006,  ...,  0.0365, -0.0865,  0.0470],
          [-0.2145, -0.3770,  0.0592,  ..., -0.0968,  0.1344,  0.2888],
          [-0.0900,  0.0140,  0.6085,  ..., -0.8772, -0.0056, -0.4345],
          ...,
          [-0.2172, -0.1211, -0.0900,  ...,  0.0951,  0.1038, -0.0617],
          [-0.3840, -0.1787, -0.4393,  ..., -0.3100, -0.5051, -0.1049],
          [-0.3018, -0.2052,  0.0081,  ...,  0.3402,  0.0176, -0.0979]],
```

6. Flatten

```
torch_out=torch.flatten(Input, start_dim=0, end_dim=-1)

torch_out
```

```
tensor([0.5758, 0.0858, 0.4262,  ..., 0.4947, 0.6854, 0.8425])
```

```
def flatten(Input):
    return Input.reshape(-1)

flatten(Input)
```

```
tensor([0.5758, 0.0858, 0.4262,  ..., 0.4947, 0.6854, 0.8425])
```

7. Sigmoid

```
torch_out=torch.sigmoid(Input,out=None)
torch_out
```

```
tensor([[[[0.6401, 0.5214, 0.6050,  ..., 0.3127, 0.6251, 0.6445],
          [0.2329, 0.4549, 0.3390,  ..., 0.5525, 0.3490, 0.5403],
          [0.3744, 0.8827, 0.9422,  ..., 0.3092, 0.2450, 0.4588],
          ...,
          [0.2944, 0.5662, 0.3257,  ..., 0.4068, 0.2828, 0.3058],
          [0.5595, 0.5199, 0.6894,  ..., 0.2664, 0.8526, 0.4319],
          [0.8154, 0.5449, 0.5182,  ..., 0.1524, 0.5534, 0.5306]],

         [[0.4455, 0.7976, 0.8279,  ..., 0.8376, 0.5053, 0.2074],
          [0.5662, 0.4012, 0.5992,  ..., 0.8269, 0.5659, 0.7461],
          [0.3781, 0.5856, 0.6447,  ..., 0.2381, 0.5683, 0.6588],
```

```python
def my_sigmoid(x):
    return 1 / (1+torch.exp(-x))
```

```python
mysigmoid=my_sigmoid(Input)
mysigmoid
```

```
tensor([[[[0.6401, 0.5214, 0.6050,  ..., 0.3127, 0.6251, 0.6445],
          [0.2329, 0.4549, 0.3390,  ..., 0.5525, 0.3490, 0.5403],
          [0.3744, 0.8827, 0.9422,  ..., 0.3092, 0.2450, 0.4588],
          ...,
          [0.2944, 0.5662, 0.3257,  ..., 0.4068, 0.2828, 0.3058],
          [0.5595, 0.5199, 0.6894,  ..., 0.2664, 0.8526, 0.4319],
          [0.8154, 0.5449, 0.5182,  ..., 0.1524, 0.5534, 0.5306]],

         [[0.4455, 0.7976, 0.8279,  ..., 0.8376, 0.5053, 0.2074],
          [0.5662, 0.4012, 0.5992,  ..., 0.8269, 0.5659, 0.7461],
          [0.3781, 0.5856, 0.6447,  ..., 0.2381, 0.5683, 0.6588],
```

8. Roi pool

```python
boxes = torch.Tensor([[0, 0, 0, 6.5, 6.5]])
torchvision.ops.roi_pool(Input,boxes,output_size=(3,3))
```

```
tensor([[[[2.7906, 2.7906, 1.0249],
          [2.7906, 2.7906, 0.6814],
          [1.4357, 0.8984, 0.3990]],

         [[1.5710, 1.5710, 1.9034],
          [1.5640, 1.1976, 2.9541],
          [1.5640, 0.7777, 1.4045]],

         [[0.8038, 1.9377, 2.1746],
          [1.4872, 1.9377, 2.1746],
          [2.1326, 2.4090, 2.2931]]]])
```

```python
my_roi_pool(Input,boxes, (3,3))
```

```
tensor([[[[2.7906, 2.7906, 1.3240],
          [2.7906, 2.7906, 0.6814],
          [0.9861, 0.8984, 0.6814]],

         [[1.5710, 1.5710, 1.4848],
          [1.0722, 1.0722, 2.9541],
          [1.5640, 1.0722, 1.1651]],

         [[0.8038, 1.2169, 2.1746],
          [1.4872, 1.4872, 2.1746],
          [1.4872, 2.4090, 2.4090]]]])
```

9. Batch norm

```python
torch.nn.functional.batch_norm(Input, running_mean=mean, running_var=var,weight=None, bias=None, training=False, momen
```

```
tensor([[[[ 0.6508,  0.1384,  0.4944,  ..., -0.7752,  0.5832,  0.6707],
          [-1.1982, -0.1405, -0.6499,  ...,  0.2689, -0.6033,  0.2176],
          [-0.4881,  2.1593,  2.9672,  ..., -0.7923, -1.1282, -0.1241],
          ...,
          [-0.8656,  0.3274, -0.7124,  ..., -0.3458, -0.9248, -0.8087],
          [ 0.2986,  0.1318,  0.8826,  ..., -1.0109,  1.8846, -0.2382],
          [ 1.6019,  0.2371,  0.1248,  ..., -1.7464,  0.2727,  0.1766]],

         [[-0.2048,  1.4223,  1.6264,  ...,  1.6974,  0.0409, -1.3530],
          [ 0.2915, -0.3905,  0.4305,  ...,  1.6190,  0.2902,  1.1218],
          [-0.4903,  0.3728,  0.6286,  ..., -1.1712,  0.3003,  0.6922],
```

```python
def batch_norm(Input,mean,var, momentum, eps):
    output = torch.zeros(Input.shape)
    for i in range(Input.shape[1]):
        output[0][i] = (Input[0][i] - mean[i])/(np.sqrt(var[i])-eps)
    return output
batch_norm(Input, mean, var,momentum=0.1, eps=1e-05)
```

```
tensor([[[[ 0.6508,  0.1384,  0.4944,  ..., -0.7752,  0.5833,  0.6707],
          [-1.1982, -0.1405, -0.6499,  ...,  0.2689, -0.6033,  0.2176],
          [-0.4881,  2.1593,  2.9672,  ..., -0.7923, -1.1282, -0.1241],
          ...,
          [-0.8657,  0.3274, -0.7124,  ..., -0.3458, -0.9248, -0.8087],
          [ 0.2986,  0.1318,  0.8826,  ..., -1.0109,  1.8847, -0.2382],
          [ 1.6019,  0.2371,  0.1248,  ..., -1.7464,  0.2727,  0.1766]],

         [[-0.2048,  1.4224,  1.6264,  ...,  1.6974,  0.0409, -1.3530],
          [ 0.2915, -0.3905,  0.4305,  ...,  1.6190,  0.2902,  1.1218],
          [-0.4903,  0.3728,  0.6286,  ..., -1.1712,  0.3003,  0.6922],
```

10. Cross_entropy

Cross-entropy is the function which combines softmax, logarithm and negative log likelihood.

```
torch_out= torch.nn.functional.cross_entropy(Input, target, weight=None,
size_average=None, ignore_index=-100, reduce=None,reduction='mean')
torch_out
```

```
tensor(1.3397)
```

```
cross_entropy(Input,target)
```

```
tensor([1.3397])
```

11. Mse-loss

```
target = torch.randn(1,3,32,32)
torch.nn.functional.mse_loss(Input, target, size_average=None,
reduce=None, reduction='mean')
```

```
tensor(2.0003)
```

```
def mse(Input,target):
    MSE = ((Input-target)**2).mean()
    return MSE
```

```
mse(Input,target)
```

```
tensor(2.0003)
```

Question #2a

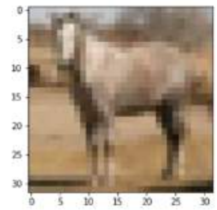
Model Structure:

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 16, 32, 32]             448
              ReLU-2           [-1, 16, 32, 32]               0
            Conv2d-3           [-1, 16, 32, 32]           2,320
              ReLU-4           [-1, 16, 32, 32]               0
         MaxPool2d-5           [-1, 16, 16, 16]               0
            Conv2d-6           [-1, 32, 16, 16]           4,640
              ReLU-7           [-1, 32, 16, 16]               0
            Conv2d-8           [-1, 32, 16, 16]           9,248
              ReLU-9           [-1, 32, 16, 16]               0
        MaxPool2d-10             [-1, 32, 8, 8]               0
           Conv2d-11             [-1, 64, 8, 8]          18,496
             ReLU-12             [-1, 64, 8, 8]               0
           Conv2d-13             [-1, 64, 8, 8]          36,928
             ReLU-14             [-1, 64, 8, 8]               0
        MaxPool2d-15             [-1, 64, 4, 4]               0
           Conv2d-16            [-1, 128, 4, 4]          73,856
             ReLU-17            [-1, 128, 4, 4]               0
           Conv2d-18            [-1, 128, 4, 4]         147,584
             ReLU-19            [-1, 128, 4, 4]               0
        AvgPool2d-20            [-1, 128, 1, 1]               0
          Flatten-21                  [-1, 128]               0
           Linear-22                   [-1, 10]           1,290
================================================================
Total params: 294,810
Trainable params: 294,810
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.01
Forward/backward pass size (MB): 0.99
Params size (MB): 1.12
Estimated Total Size (MB): 2.13
----------------------------------------------------------------
```

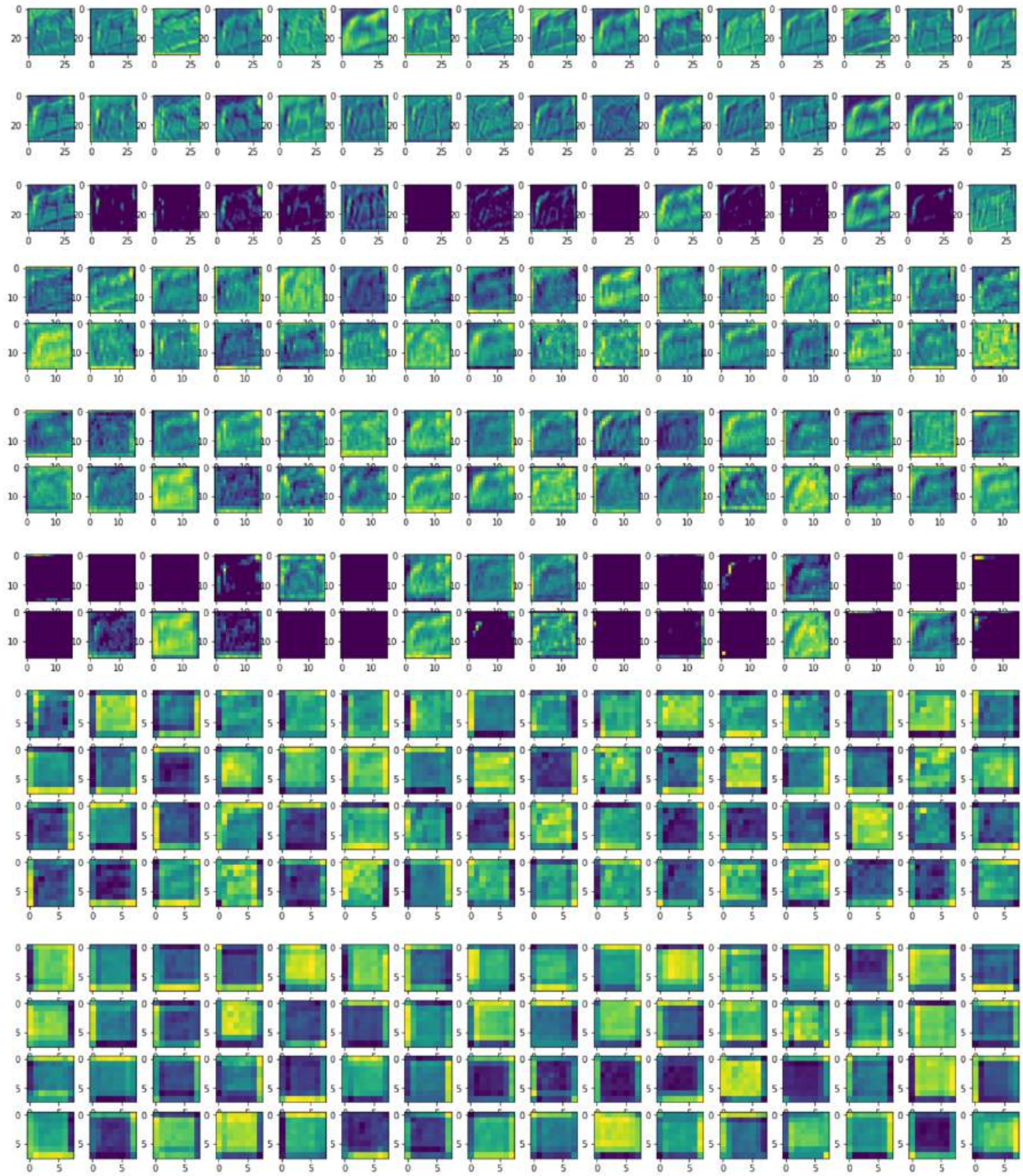
I have tried to change the learning rate [0.1,**0.01**], batch size [32,**64**,128], optimizer [**Adam**, SGD]. The model with the best accuracy is using lr=0.01, batch size=64, optimizer=Adam, and the accuracy reached 82.7%.

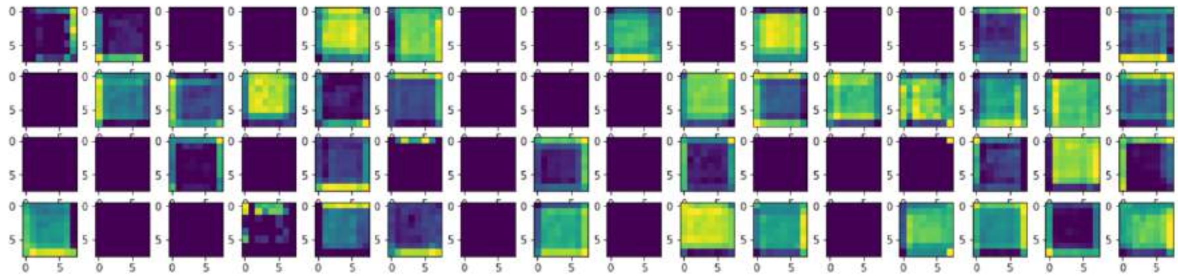
I also tried different transformation methods, such as RandomHorizontalFlip, RandomRotation, they can also substantially improve model accuracy.

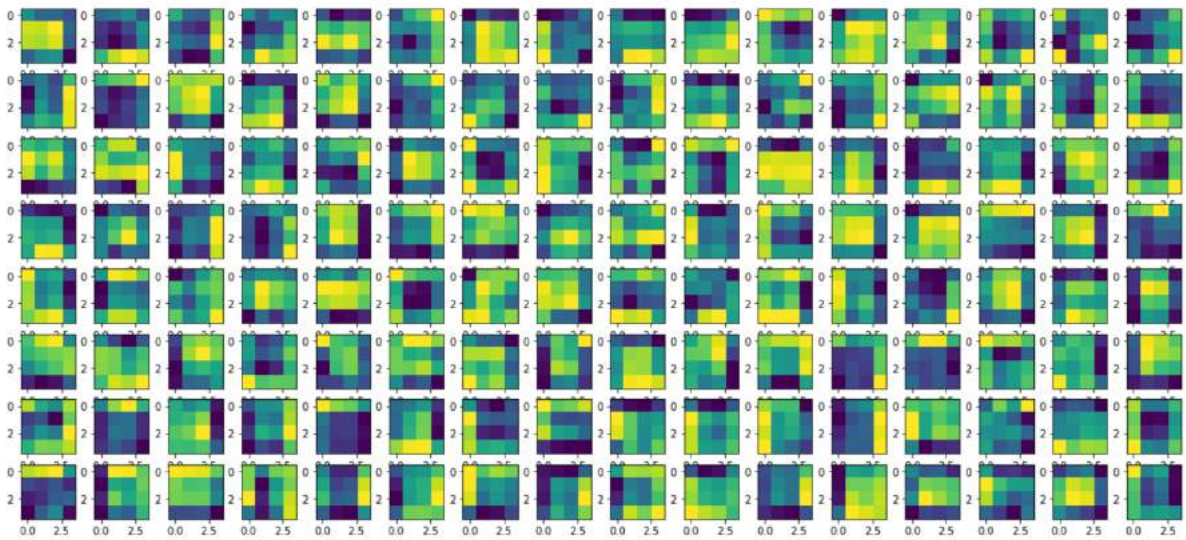
Example image:

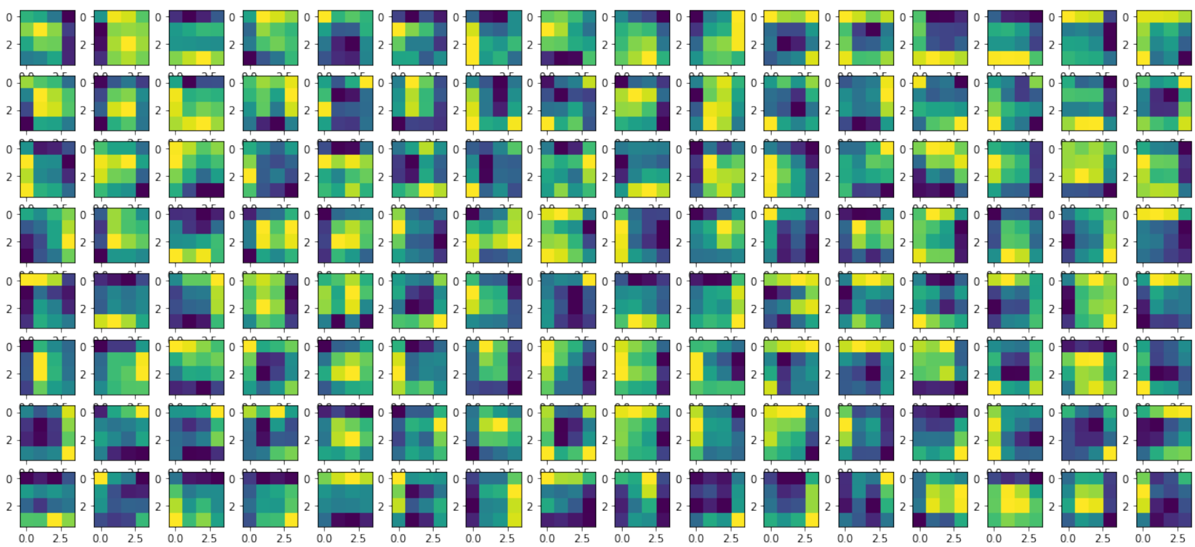


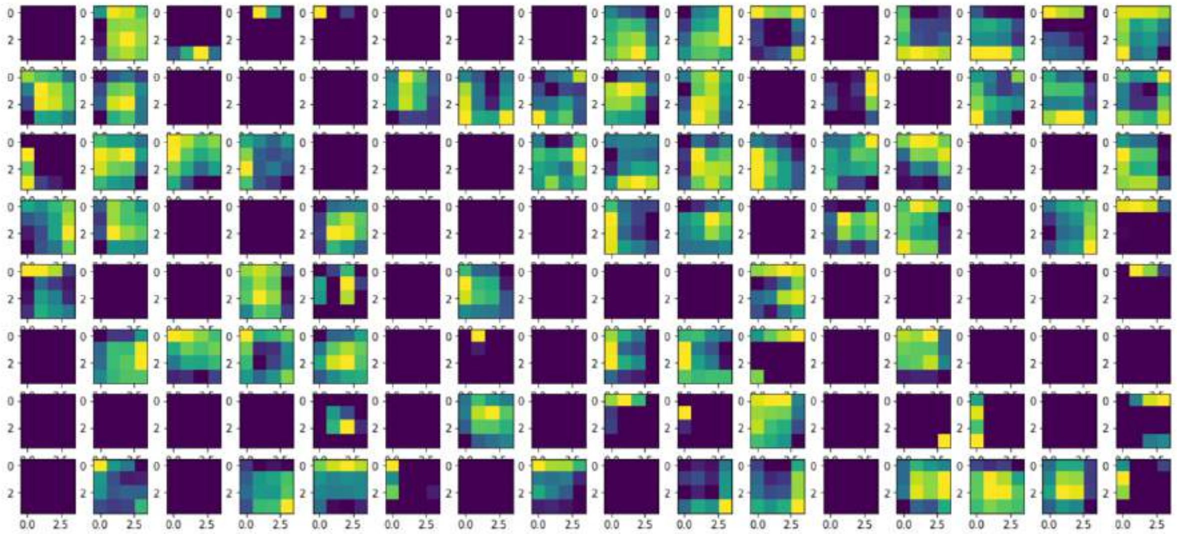
Without absolute average activation function:



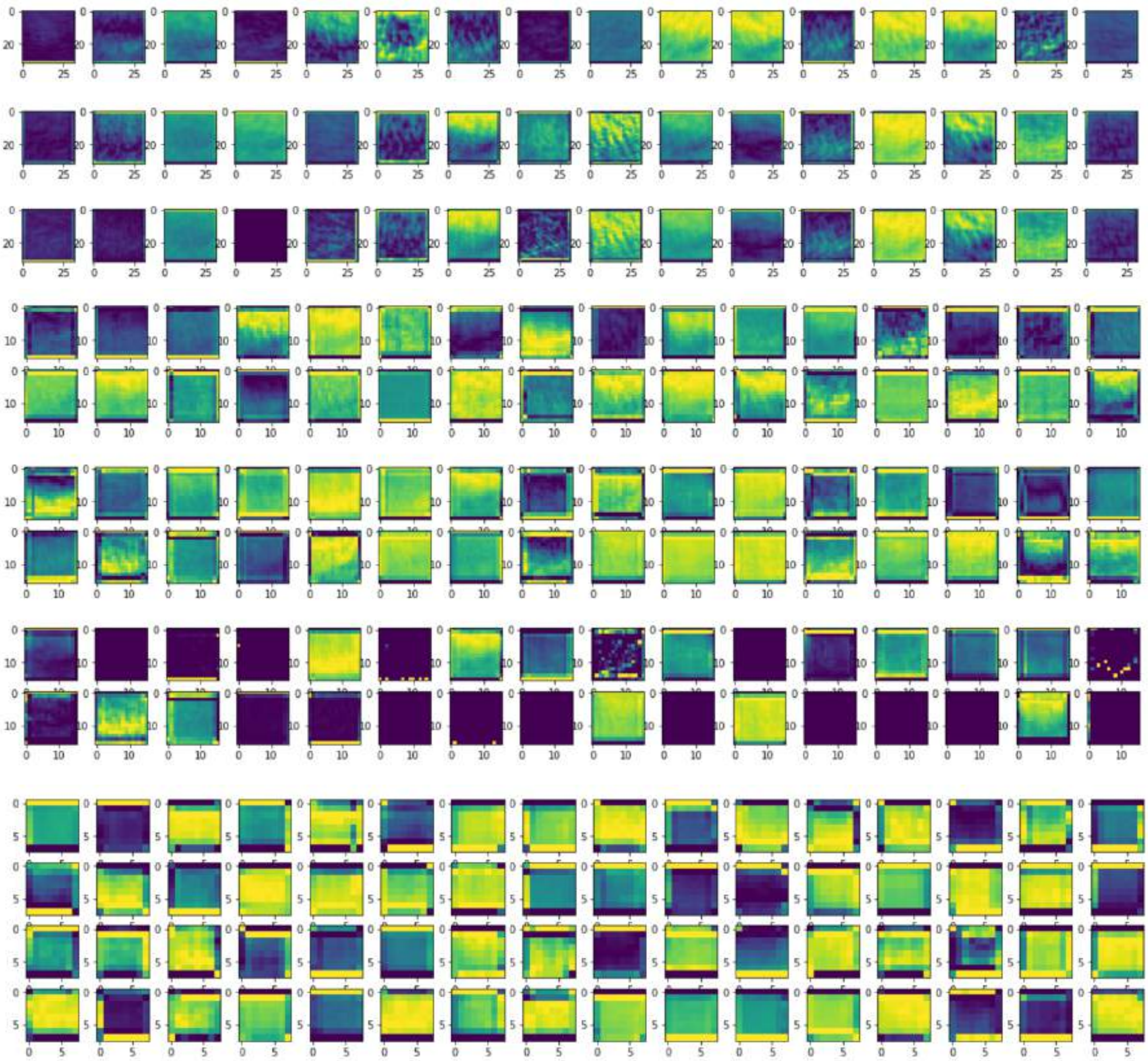


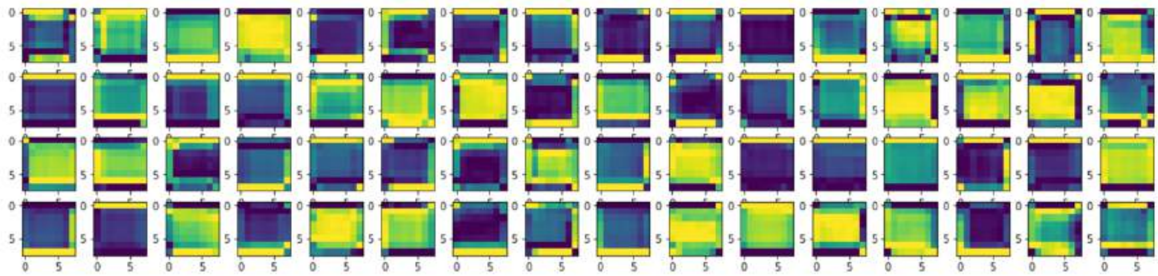


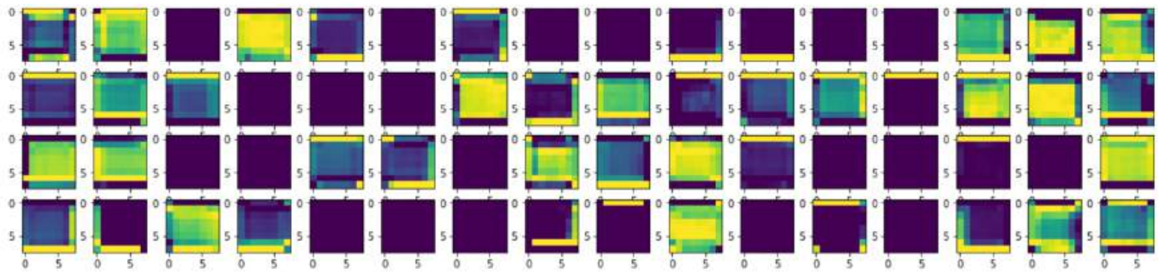


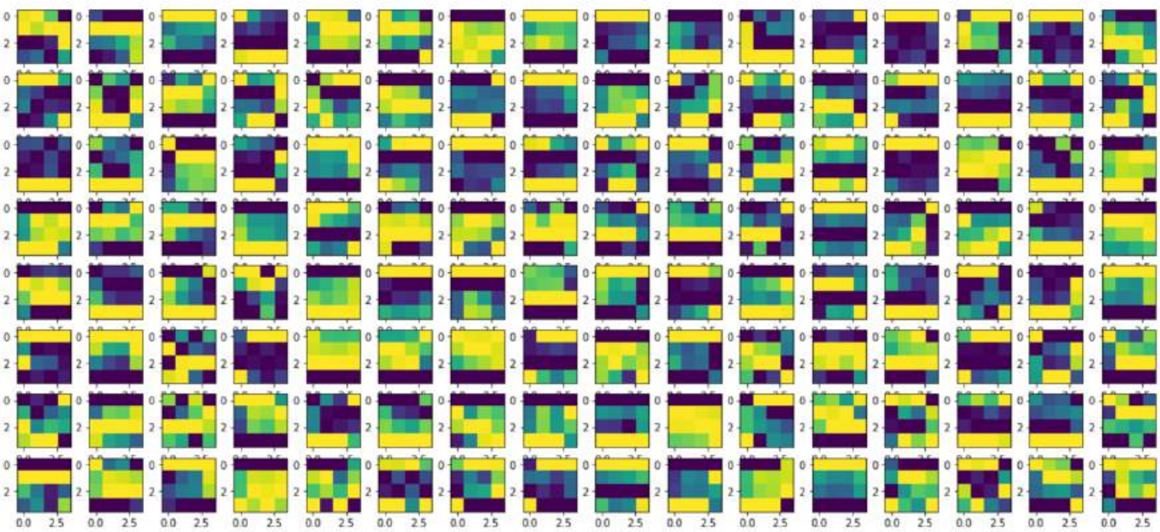


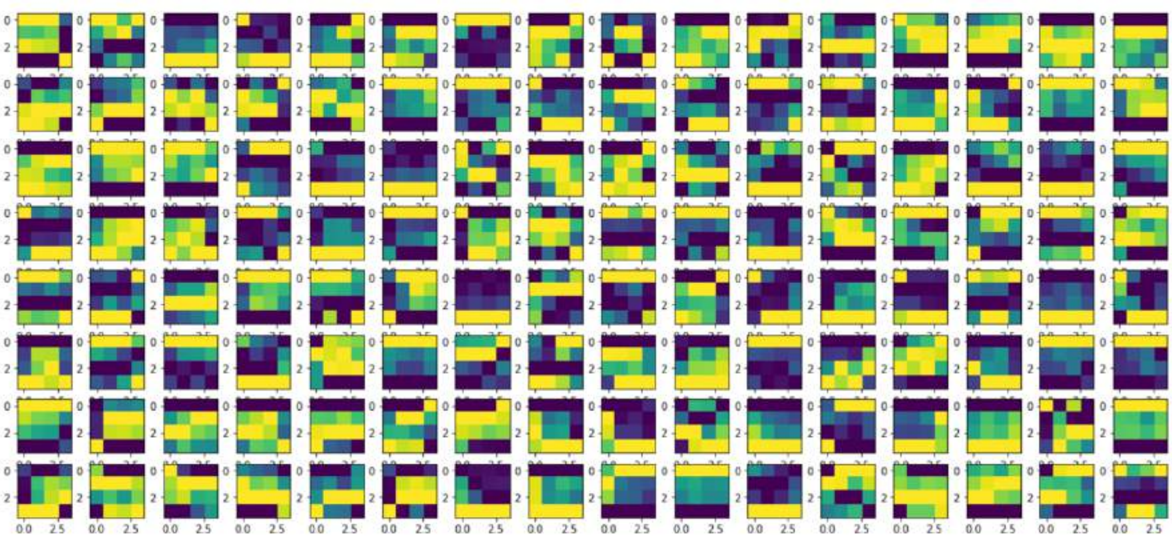
With absolute average activation function:

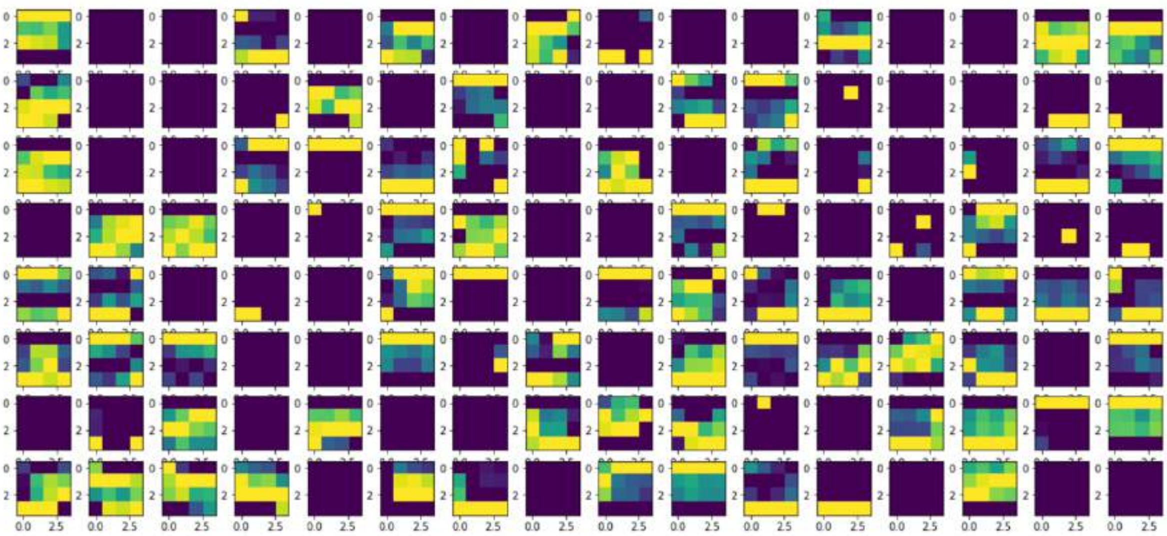




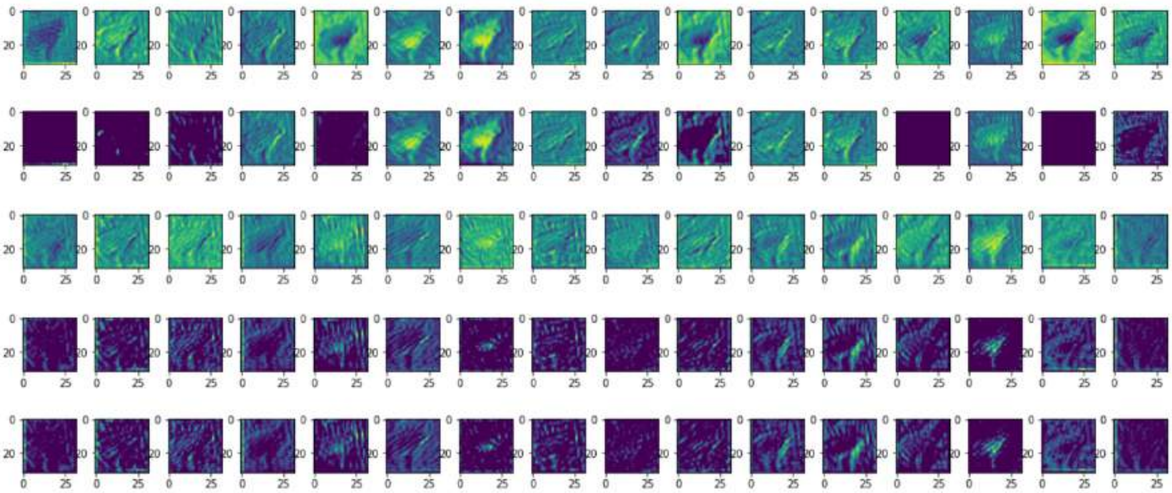


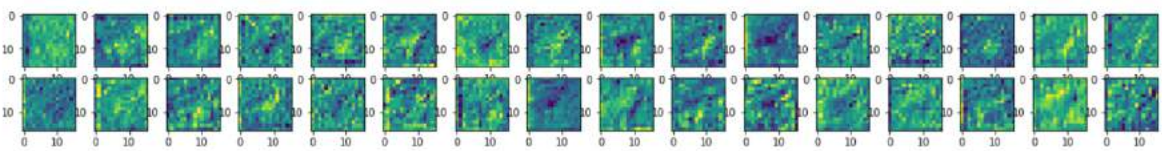


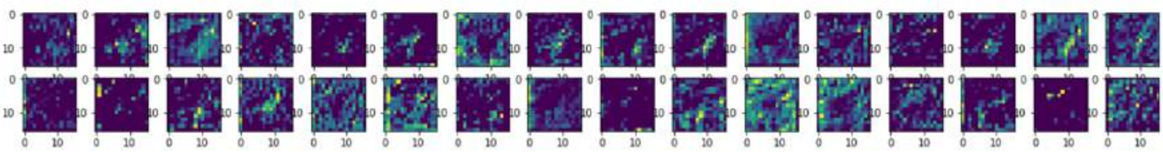


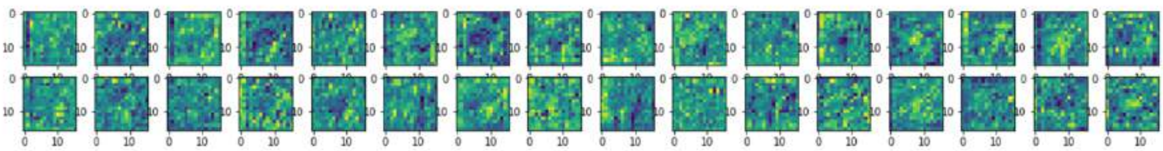


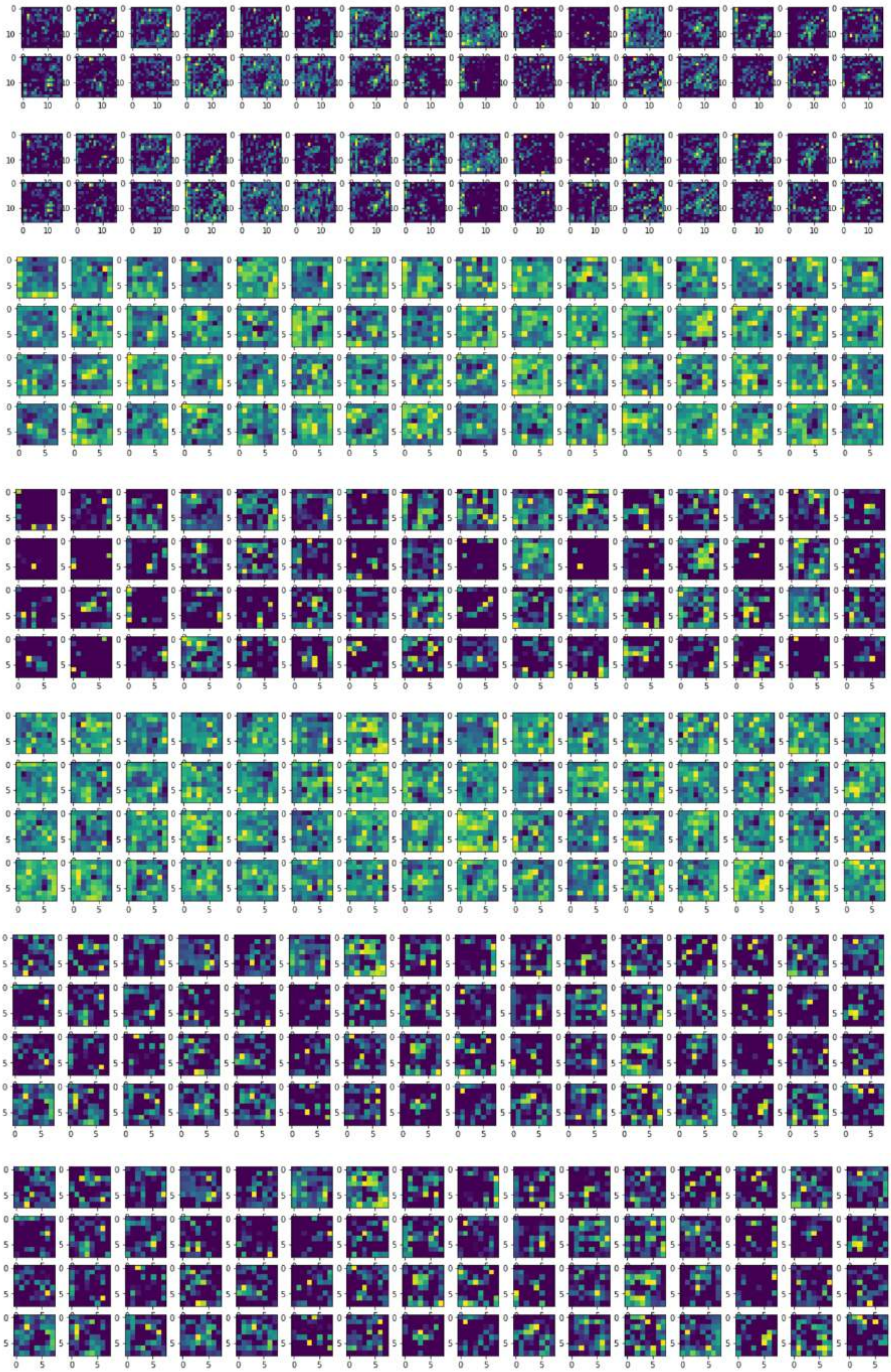
Question #2b

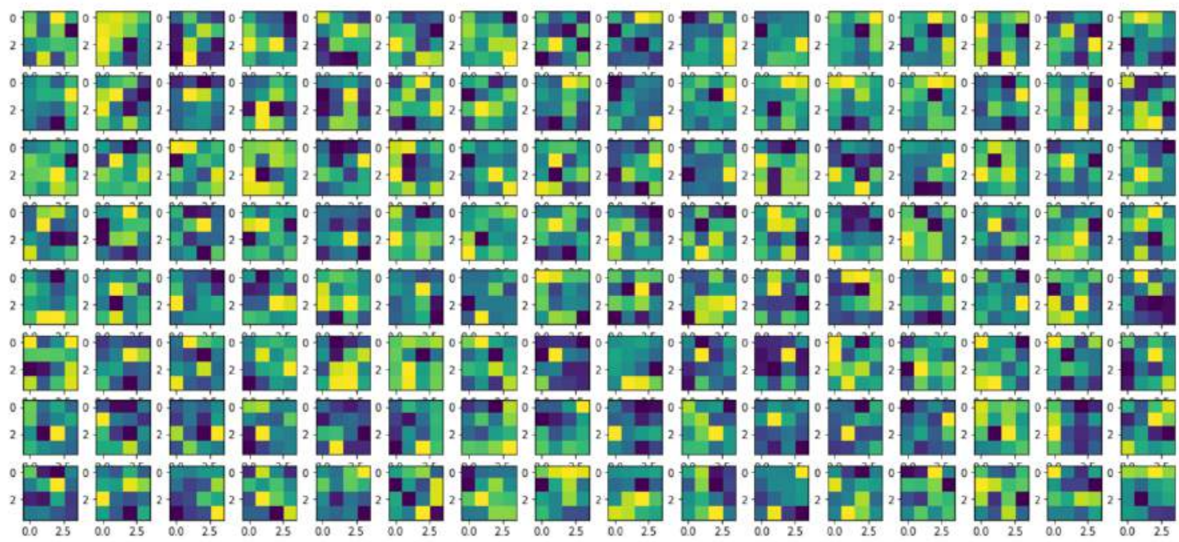


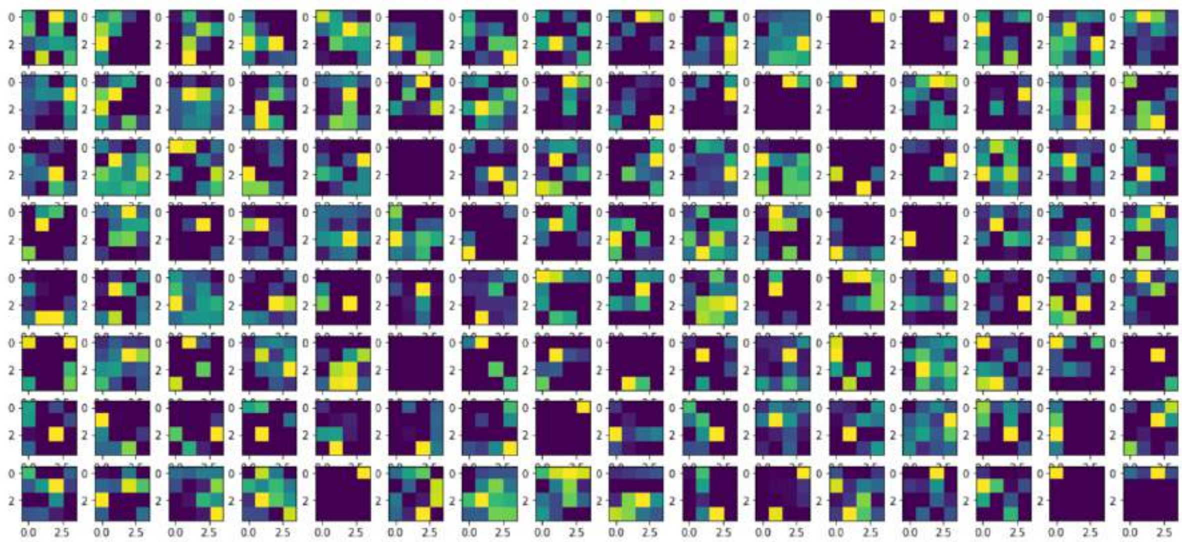


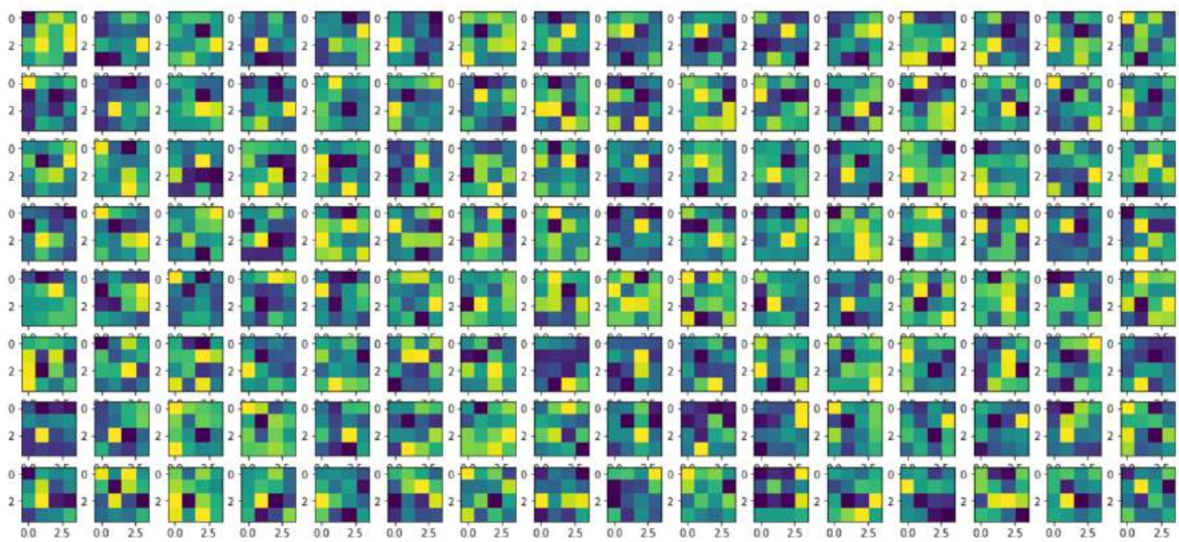


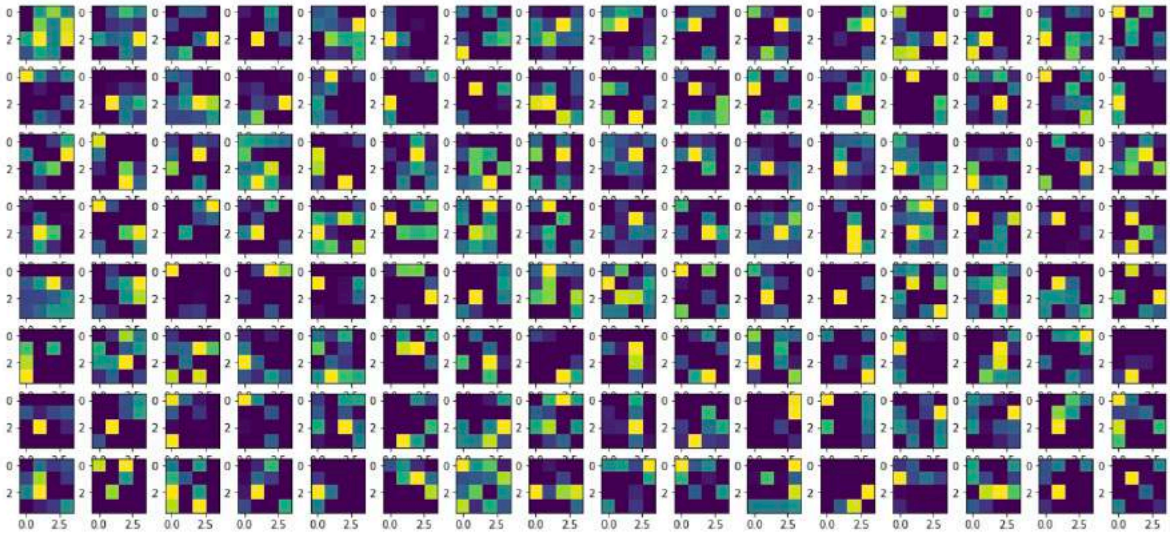


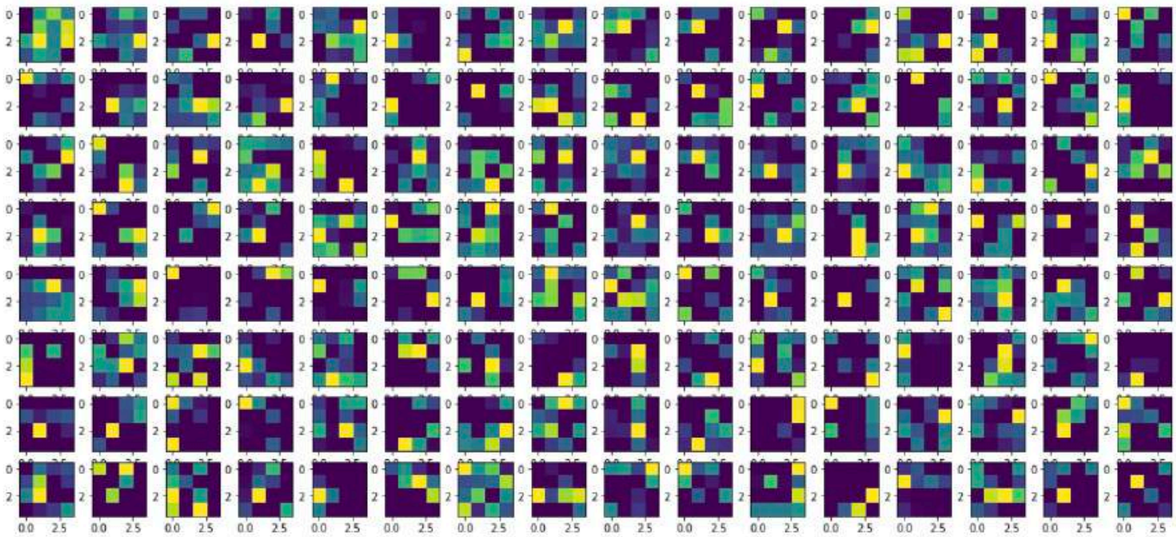






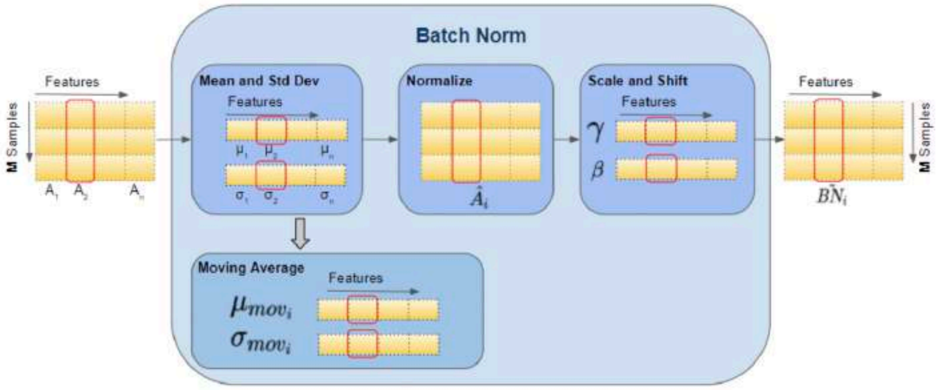






Model accuracy: 83.8

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 16, 32, 32]             448
              ReLU-2           [-1, 16, 32, 32]               0
       BatchNorm2d-3           [-1, 16, 32, 32]               0
            Conv2d-4           [-1, 16, 32, 32]           2,320
              ReLU-5           [-1, 16, 32, 32]               0
       BatchNorm2d-6           [-1, 16, 32, 32]               0
         MaxPool2d-7           [-1, 16, 16, 16]               0
            Conv2d-8           [-1, 32, 16, 16]           4,640
              ReLU-9           [-1, 32, 16, 16]               0
      BatchNorm2d-10           [-1, 32, 16, 16]               0
           Conv2d-11           [-1, 32, 16, 16]           9,248
             ReLU-12           [-1, 32, 16, 16]               0
      BatchNorm2d-13           [-1, 32, 16, 16]               0
        MaxPool2d-14             [-1, 32, 8, 8]               0
           Conv2d-15             [-1, 64, 8, 8]          18,496
             ReLU-16             [-1, 64, 8, 8]               0
      BatchNorm2d-17             [-1, 64, 8, 8]               0
           Conv2d-18             [-1, 64, 8, 8]          36,928
             ReLU-19             [-1, 64, 8, 8]               0
      BatchNorm2d-20             [-1, 64, 8, 8]               0
        MaxPool2d-21             [-1, 64, 4, 4]               0
           Conv2d-22            [-1, 128, 4, 4]          73,856
             ReLU-23            [-1, 128, 4, 4]               0
      BatchNorm2d-24            [-1, 128, 4, 4]               0
           Conv2d-25            [-1, 128, 4, 4]         147,584
             ReLU-26            [-1, 128, 4, 4]               0
      BatchNorm2d-27            [-1, 128, 4, 4]               0
AdaptiveAvgPool2d-28            [-1, 128, 1, 1]               0
          Flatten-29                  [-1, 128]               0
           Linear-30                   [-1, 10]           1,290
================================================================
Total params: 294,810
Trainable params: 294,810
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.01
Forward/backward pass size (MB): 1.46
Params size (MB): 1.12
Estimated Total Size (MB): 2.60
----------------------------------------------------------------
```



Batch Norm is proposed to address Internal Covariate Shift, which means that the distribution of the output data of each layer is constantly changing as the parameters are continuously updated, resulting in the need for the latter layer to re-fit the new distribution, making network learning difficult. Therefore, without Batch Norm, a smaller learning rate and slow update are required, and thus the learning efficiency is relatively low. With Batch Norm, a larger learning rate can be used to speed up the convergence, and the convergence process will be more stable and not particularly sensitive to the initial values.

During training, each batch of training data is normalized, i.e., the mean and variance of each batch of data is used. When testing, for a sample prediction, there is no concept of batch, so the mean and variance used at this time are the mean and variance of the full training data, which can be obtained by moving average. For Batch Normalization, when a model is trained, all its parameters are determined, including the mean and variance, γ and β.