Bing

Sure, I can give you an example of the attention map and multiplication. Let’s assume that the input tensor x has a shape of (1, 4, 2, 2, 2), which means it has one batch, four channels, and two units in each spatial dimension. The input tensor x can be visualized as a 4x2x2x2 cube, where each cell represents a feature value. Here is an example of the input tensor x:

x = torch.tensor([[[[[0.1, 0.2], [0.3, 0.4]],

```
                [[ 0.5,  0.6],
                 [ 0.7,  0.8]]],


               [[[ 0.9,  1.0],
                 [ 1.1,  1.2]],

                [[ 1.3,  1.4],
                 [ 1.5,  1.6]]],


               [[[ 1.7,  1.8],
                 [ 1.9,  2.0]],

                [[ 2.1,  2.2],
                 [ 2.3,  2.4]]],


               [[[ 2.5,  2.6],
                 [ 2.7,  2.8]],

                [[ 2.9,  3.0],
                 [ 3.1,  3.2]]]]])
```

The spatial attention branch performs a 1x1x4 convolution on the input tensor x, which reduces the number of channels to 1. The output of the convolution is then passed through a sigmoid function to obtain the spatial attention map, which has a shape of (1, 1, 2, 2, 2). The spatial attention map can be visualized as a 1x2x2x2 cube, where each cell represents a weight between 0 and 1. Here is an example of the spatial attention map:

spa_se = torch.tensor([[[[[0.6, 0.7], [0.8, 0.9]],

```
                     [[ 0.9,  0.8],
                      [ 0.7,  0.6]]]]])
```

The channel attention branch performs an adaptive average pooling on the input tensor x, which reduces the spatial dimensions to 1x1x1. The output of the pooling is then fed to a neural network, which consists of two linear layers and a ReLU activation. The neural network learns a channel attention vector, which has a shape of (1, 4, 1, 1, 1). The channel attention vector can be visualized as a 4x1x1x1 cube, where each cell represents a weight between 0 and 1. Here is an example of the channel attention vector:

chn_se = torch.tensor([[[[[0.4]],

```
                    [[[ 0.5]],

                    [[[ 0.6]],

                    [[[ 0.7]]]]])
```

The spatial attention map and the channel attention vector are then multiplied element-wise with the input tensor x, respectively, to perform spatial-wise and channel-wise feature recalibration. The recalibrated tensors have the same shape as the input tensor x, which is (1, 4, 2, 2, 2). The recalibrated tensors can be visualized as 4x2x2x2 cubes, where each cell represents a recalibrated feature value. Here are the examples of the recalibrated tensors:

spa_se_x = torch.tensor([[[[[0.06, 0.14], [0.24, 0.36]],

```
                   [[ 0.45,  0.48],
                    [ 0.49,  0.48]]],


                  [[[ 0.54,  0.70],
                    [ 0.88,  1.08]],

                   [[ 1.17,  1.12],
                    [ 1.05,  0.96]]],


                  [[[ 1.02,  1.44],
                    [ 1.52,  1.80]],

                   [[ 1.89,  1.76],
                    [ 1.61,  1.44]]],


                  [[[ 1.50,  1.82],
                    [ 1.89,  2.52]],

                   [[ 2.61,  2.40],
                    [ 2.17,  2.16]]]]])
```

chn_se_x = torch.tensor([[[[[0.04, 0.08], [0.12, 0.16]],

```
                   [[ 0.20,  0.24],
                    [ 0.28,  0.32]]],


                  [[[ 0.45,  0.50],
                    [ 0.55,  0.60]],

                   [[ 0.65,  0.70],
                    [ 0.75,  0.80]]],


                  [[[ 1.02,  1.08],
                    [ 1.14,  1.20]],

                   [[ 1.26,  1.32],
                    [ 1.38,  1.44]]],


                  [[[ 1.75,  1.82],
                    [ 1.89,  1.96]],

                   [[ 2.03,  2.10],
                    [ 2.17,  2.24]]]]])
```

The recalibrated tensors are then added together with the input tensor x, forming a skip connection. This reduces the sparsity and singularity caused by the parallel excitations, and preserves the original information of the input. The skip connection also makes the learning more generic and enhances the segmentation prediction. The final output tensor has the same shape as the input tensor x, which is (1, 4, 2, 2, 2). The final output tensor can be visualized as a 4x2x2x2 cube, where each cell represents a final feature value. Here is an example of the final output tensor:

net_out = torch.tensor([[[[[0.16, 0.30], [0.48, 0.72]],

```
                  [[ 0.85,  0.96],
                   [ 1.01,  1.04]]],


                 [[[ 1.08,  1.40],
                   [ 1.64,  1.92]],

                  [[ 1.97,  1.92],
                   [ 1.85,  1.76]]],


                 [[[ 2.12,  2.52],
                   [ 2.66,  3.00]],
                     [[ 3.15,   3.08],
                      [ 2.99,   2.88]]],


                    [[[ 3.30,   3.60],
                      [ 3.66,   4.28]],

                     [[ 4.64,   4.40],
                      [ 4.35,   4.32]]]]])
```

I hope this example helps you understand the attention map and multiplication. 😊