Forward propagating the output in Python

Before we proceed, note that this section is only here to help you clearly understand how CNNs work. We don't need to perform the following steps in a real-world scenario:

1. Extract the weights and biases of the convolution and linear layers of the architecture that's been defined, as follows:

    - Extract the various layers of the model:

```
list(model.children())
```

    This results in the following output:

```
[Conv2d(1, 1, kernel_size=(3, 3), stride=(1, 1)),
 MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False),
 ReLU(),
 Flatten(),
 Linear(in_features=1, out_features=1, bias=True),
 Sigmoid()]
```

    - Extract the layers among all the layers of the model that have the `weight` attribute associated with them:

```
(cnn_w, cnn_b), (lin_w, lin_b) = [(layer.weight.data, \
                                   layer.bias.data) for layer in \
                                   list(model.children()) \
                                         if hasattr(layer,'weight')]
```

    In the preceding code, `hasattr(layer,'weight')` returns a boolean, regardless of whether the layer contains the `weight` attribute.

    Note that the convolution (`Conv2d`) layer and the `Linear` layer at the end are the only layers that contain parameters, which is why we saved them as `cnn_w` and `cnn_b` for the `Conv2d` layer and `lin_w` and `lin_b` for the `Linear` layer, respectively.

    The shape of `cnn_w` is 1 x 1 x 3 x 3 since we have initialized one filter, which has one channel and a dimension of 3 x 3. `cnn_b` has a shape of 1 as it corresponds to one filter.

2. To perform the `cnn_w` convolution operation over the input value, we must initialize a matrix of zeros for sumproduct (`sumprod`) where the height is *input height - filter height + 1* and the width is *width - filter width + 1*:

```
h_im, w_im = X_train.shape[2:]
h_conv, w_conv = cnn_w.shape[2:]
sumprod = torch.zeros((h_im - h_conv + 1, w_im - w_conv + 1))
```

3. Now, let's fill `sumprod` by convoluting the filter (`cnn_w`) across the first input and summing up the filter bias term (`cnn_b`) after reshaping the filter shape from a 1 x 1 x 3 x 3 shape to a 3 x 3 shape:

```
for i in range(h_im - h_conv + 1):
    for j in range(w_im - w_conv + 1):
        img_subset = X_train[0, 0, i:(i+3), j:(j+3)]
```

```
        model_filter = cnn_w.reshape(3,3)
        val = torch.sum(img_subset*model_filter) + cnn_b
        sumprod[i,j] = val
```

In the preceding code, `img_subset` stores the portion of the input that we would be convolving with the filter and hence we stride through it across the possible columns and then rows. Furthermore, given that the input is 4 x 4 in shape and the filter is 3 x 3 in shape, the output is 2 x 2 in shape.

At this stage, the output of `sumprod` is as follows:

```
tensor([[-0.0143, -0.0636],
        [-0.0834, -0.0500]])
```

4. Perform the ReLU operation on top of the output and then fetch the maximum value of the pool (`MaxPooling`), as follows:

    - ReLU is performed on top of `sumprod` in Python as follows:

```
sumprod.clamp_min_(0)
```

Note that we are clamping the output to a minimum of 0 in the preceding code (which is what ReLU activation does):

```
tensor([[0.,0.],
 [0.,0.]])
```

- The output of the pooling layer can be calculated like so:

```
pooling_layer_output = torch.max(sumprod)
```

The preceding code results in the following output:

```
tensor(0.)
```

5. Pass the preceding output through linear activation:

```
intermediate_output_value = pooling_layer_output*lin_w+lin_b
```

The output of this operation is as follows:

```
tensor([[-1.6398]])
```

6. Pass the output through the `sigmoid` operation:

```
from torch.nn import functional as F # torch library
# for numpy like functions
print(F.sigmoid(intermediate_output_value))
```

Note that we perform `sigmoid` and not `softmax` since the loss function is binary cross-entropy and not categorical cross-entropy like it was in the Fashion-MNIST dataset. As such, the preceding code gives us the following output:

```
tensor([[0.1625]])
```

As you can see, it gives us the same output we obtained using PyTorch's feedforward method, thus strengthening our understanding of how CNNs work.