Lab 7 - Convolutional Neural Networks (CNNs)

The building blocks of CNNs

Convolutional layers

The basic building block for a CNN is the convolutional layer, accessible as `torch.nn.Conv<s>d`, where `<s>` represents the number of **spatial dimensions** of our data:

- `Conv1d` for 1 dimensional sequences. Example: audio. Audio is organized as a sequence of a given length (the single spatial dimension), where each single value in this sequence represent the intensity/amplitude of the signal for a given time point. Audio data can be organized in multiple **channels** (e.g., stereo data has 2 channels). The convolution opration is represented by a one-dimensional kernel;
- `Conv2d` for 2 dimensional data, like images.
- `Conv3d` for 3 dimensional data. An example might be a 3D reconstruction of an image. A convolution in that domain might equate to sliding a cubic kernel along all three dimensions.

Parameters for constructors:

```
Conv2d(in_channels: int, out_channels: int, kernel_size: Union[int, Tuple[int, int]], stride: Union[int, Tuple[int, int]] = 1, padding: Union[int, Tuple[int, int]] = 0, dilation: Union[int, Tuple[int, int]] = 1, groups: int = 1, bias: bool = True, padding_mode: str = 'zeros')
```

- in_channels: the number of channels of the incoming data
- out_channels: the number of channels for the output data, i.e., the number of convolutions that are operated
- kernel_size: the kernel size of each convolution. An int k is interpreted as a tuple (k, k) (i.e., a square kernel); for a rectangular kernel, pass a tuple.
- stride, padding, dilation: trivial

Note that the convolution does NOT require a specific spatial dimension as input/output, as convolution is oblivious to these factors:

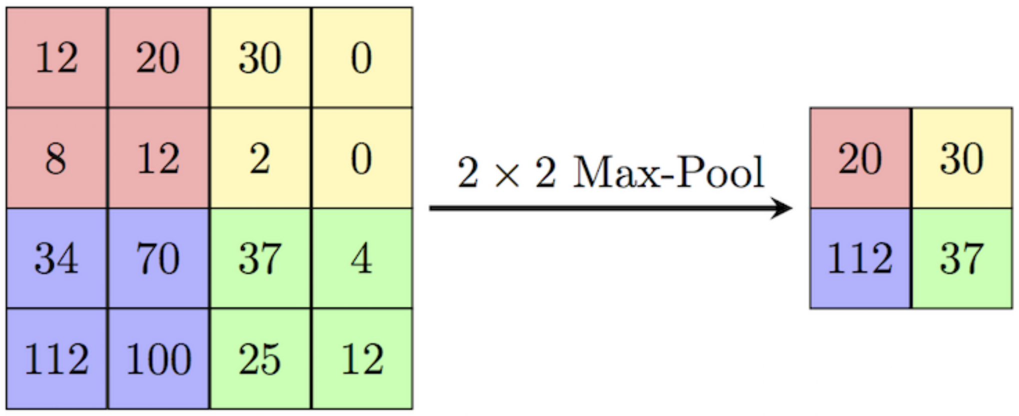
In [2]:
```python
import torch
import torch.nn as nn
import torch.nn.functional as F

conv_layer = nn.Conv2d(in_channels=3, out_channels=32, kernel_size=3)
print("Parameters of convolution\n", "Weights\n", conv_layer.weight.shape, "\nBias\n", conv_layer.bias.shape)

print("\nConv2d is applied independently of the input spatial dimension")
y = conv_layer(torch.rand(1,3,32,32))
print("Shape of y ", y.shape)

z = conv_layer(torch.rand(1,3,12,12))
print("Shape of z ", z.shape)
```

```
Parameters of convolution
 Weights
 torch.Size([32, 3, 3, 3])
Bias
 torch.Size([32])

Conv2d is applied independently of the input spatial dimension
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
<ipython-input-2-59331ee7c149> in <module>
      7
      8 print("\nConv2d is applied independently of the input spatial dimension")
----> 9 y = conv_layer(torch.rand(3,32,32))
     10 print("Shape of y ", y.shape)
     11

~\Anaconda3\lib\site-packages\torch\nn\modules\module.py in _call_impl(self, *input, **kwargs)
    887             result = self._slow_forward(*input, **kwargs)
    888         else:
--> 889             result = self.forward(*input, **kwargs)
    890         for hook in itertools.chain(
    891                 _global_forward_hooks.values(),

~\Anaconda3\lib\site-packages\torch\nn\modules\conv.py in forward(self, input)
    397
    398     def forward(self, input: Tensor) -> Tensor:
--> 399         return self._conv_forward(input, self.weight, self.bias)
    400
    401 class Conv3d(_ConvNd):

~\Anaconda3\lib\site-packages\torch\nn\modules\conv.py in _conv_forward(self, input, weight, bias)
    394                             _pair(0), self.dilation, self.groups)
    395         return F.conv2d(input, weight, bias, self.stride,
--> 396                         self.padding, self.dilation, self.groups)
    397
    398     def forward(self, input: Tensor) -> Tensor:

RuntimeError: Expected 4-dimensional input for 4-dimensional weight [32, 3, 3, 3], but got 3-dimensional input of siz
e [3, 32, 32] instead
```

Pooling layers

Pooling layers are essentially convolutions without trainable kernels. For each overlap between the image and the kernel, they output the maximum (→ *maxpooling*) or the average (→ *avgpooling*) of the image in that specific region.



`MaxPool2d(kernel_size: Union[int, Tuple[int, ...]], stride: Union[int, Tuple[int, ...], NoneType] = None, padding: Union[int, Tuple[int, ...]] = 0, dilation: Union[int, Tuple[int, ...]] = 1, return_indices: bool = False, ceil_mode: bool = False)`

Notice that now we have no input or output channels as parameter, because MaxPool/AvgPool act independently on each channel, so `in_channels=out_channels`

Adaptive Pooling

Adaptive (Max/Average) Pooling is still a pooling layer, but we have the option to specify the desired spatial dimension of the output instead of the parameters like kernel size, padding...

PyTorch works out by itself the params which are required in order for the pooling to produce an output of the desired size.

Maybe the most common application of this layer is when operating the channel-wise average pooling at the end of the cascade of convolutional layers. In this case, we specify a fixed size of $(1, 1)$, s.t. PyTorch will essentially operate an average of each whole channel.

In [5]:
```
layer = nn.AdaptiveAvgPool2d(output_size=(1, 1))
layer(torch.rand(1,3,32,32)).shape # 1 batch size, 3 channels, 32x32 img
```

Out[5]:
```
torch.Size([1, 3, 1, 1])
```

Q: what do we need to push this data through a Linear layer? We need to flatten the data

In [7]:
```
nn.Flatten()(layer(torch.rand(1,3,32,32))).shape
```

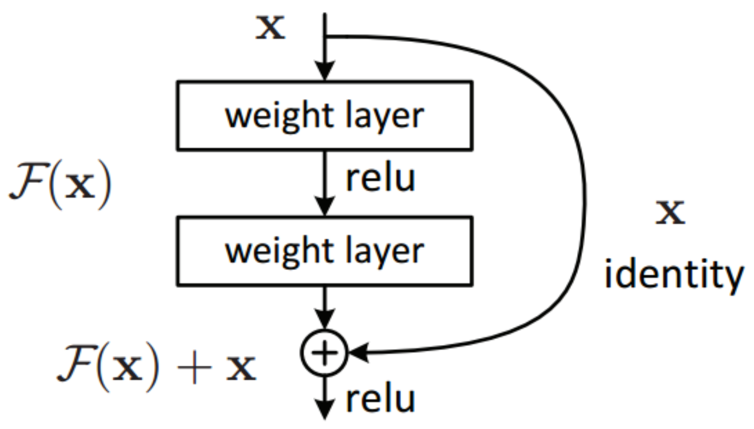
Out[7]:
```
torch.Size([1, 3])
```

Now that we have all the building blocks of a CNN, let's put it in practice and build something

Let's build a ResNet18!

A ResNet is a CNN architecture which is composed by a cascade of "residual blocks", which are themselves cascades of convolutions, plus skip connections.

The ResNet18 architecture is composed of 18 convolutional layers arranged in "basic" residual blocks with this shape:

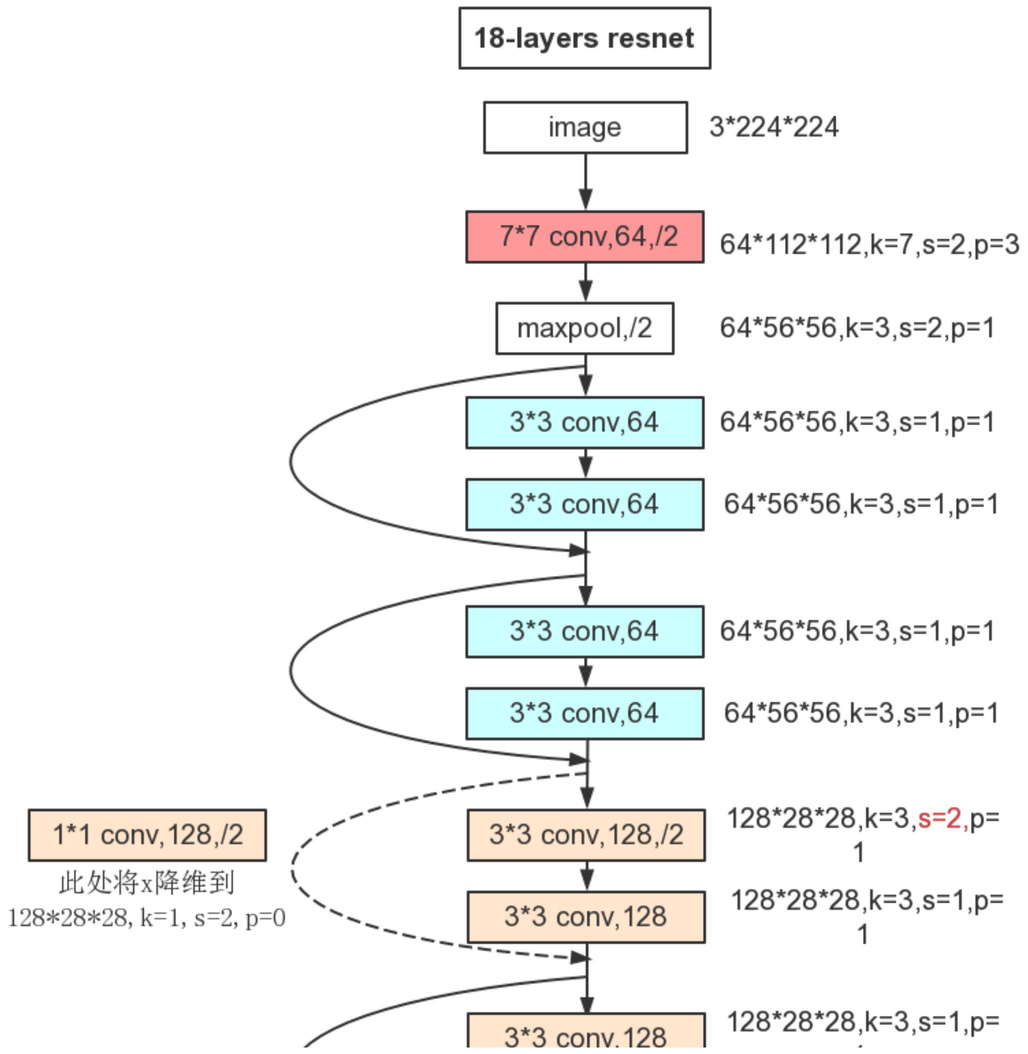


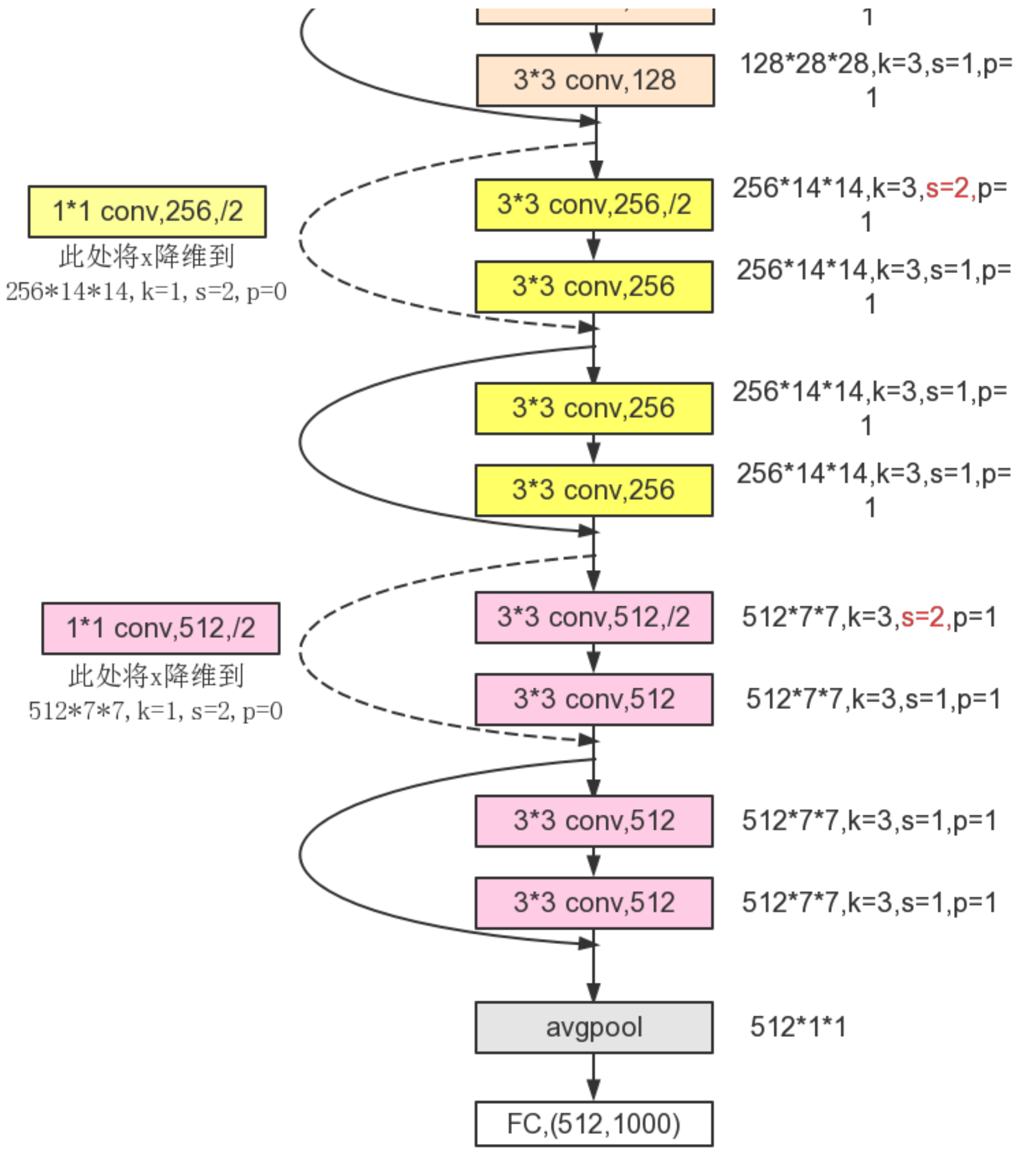
x
weight layer
$\mathcal{F}(\mathbf{x})$
relu
weight layer
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

The incoming information gets duplicated: one flow passes through multiple convolutional layers (`weight layer` in the image above), while the other "skips" these layers and is summed up to the output of the last conv layer, *before* the activation function of this layer is applied.

Skip connections avoid the dilemma of the "vanishing gradient", where large neural network architectures are basically untrainable due to the first layers having extremely small gradients which limits weight update.

This, instead, is the general scheme of a ResNet18:





Additionally, notice that all convolutional layers within residual blocks have **no bias**.

Let us build, in order,

1. The preparatory block
2. The classifier block (pooling + linear)
3. The basic residual block without downsampling
   - then add downsampling

In [11]:
```
class PreparatoryBlock(nn.Module):
    def __init__(self, in_channels=3):
        super().__init__()
        # conv with 7x7 kernel, stride 2, padding 3
        # maxpool with 3x3 kernel, stride 2, padding 1
        self.mod = nn.Sequential(
            nn.Conv2d(in_channels=in_channels, out_channels=64, kernel_size=7, stride=2, padding=3),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        )

    def forward(self, data):
        return self.mod(data)

block = PreparatoryBlock()
block(torch.rand(1,3,224,224)).shape
```

Out[11]:
```
torch.Size([1, 64, 56, 56])
```

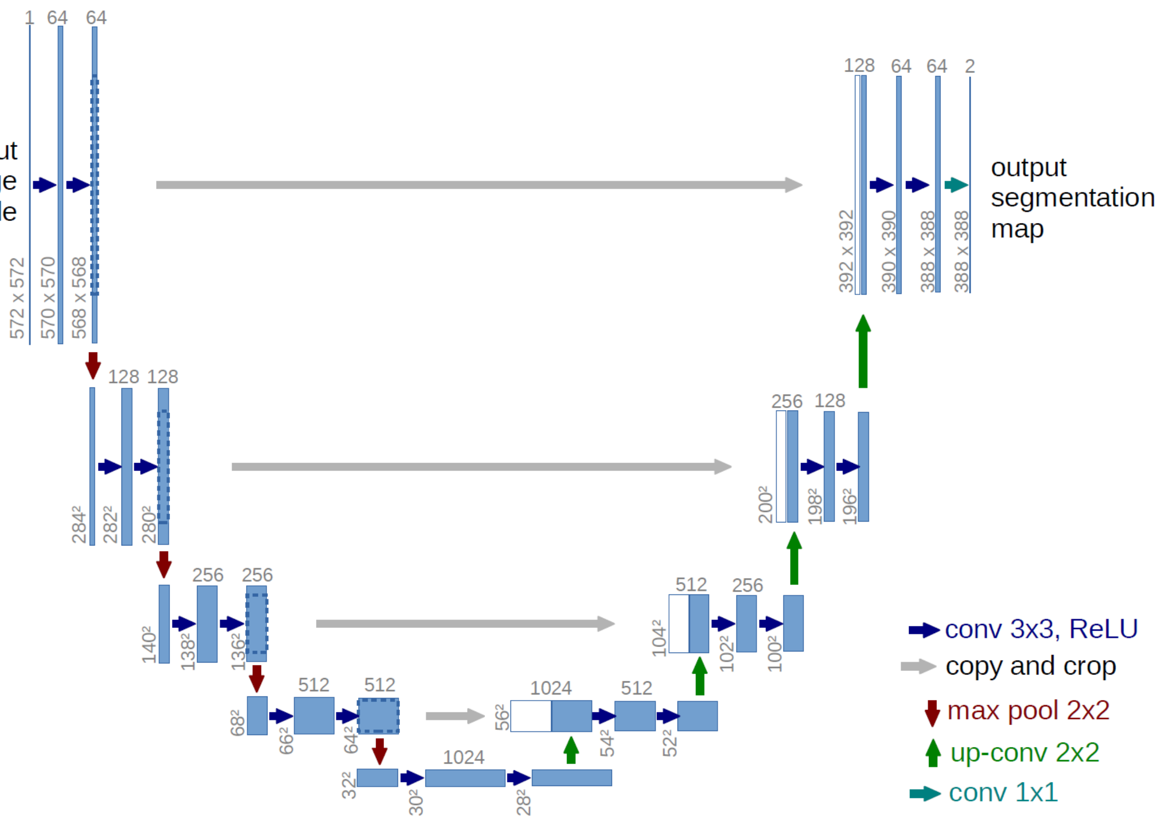
In [13]:
```
class ClassifierBlock(nn.Module):
    def __init__(self, in_channels, num_classes=10):
        super().__init__()
        self.mod = nn.Sequential(
            nn.AdaptiveAvgPool2d((1, 1)),
            nn.Flatten(),
            nn.Linear(in_channels, num_classes)
        )

    def forward(self, data):
        return self.mod(data)

block = ClassifierBlock(512)
block(torch.rand(1, 512, 7, 7)).shape
```

Out[13]:
```
torch.Size([1, 10])
```

In [15]:
```
class BasicResidualBlock(nn.Module):
    def __init__(self, in_channels, out_channels, downsampling=False):
        super().__init__()
        stride_first_layer = (2 if downsampling else 1)
        self.conv1 = nn.Conv2d(in_channels=in_channels, out_channels=out_channels, kernel_size=3, padding=1, stride=s
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.conv2 = nn.Conv2d(in_channels=out_channels, out_channels=out_channels, kernel_size=3, padding=1, bias=Fa
        self.bn2 = nn.BatchNorm2d(out_channels)

        self.downsampling = downsampling
        if downsampling:
            self.conv_skip = nn.Conv2d(in_channels=in_channels, out_channels=out_channels, kernel_size=1, stride=2, p

    def forward(self, data):
        out1 = data
        if self.downsampling:
            out1 = self.conv_skip(out1)
        out2 = self.conv1(data)
        out2 = self.bn1(out2)
        out2 = F.relu(out2)
        out2 = self.conv2(out2)
        out2 = self.bn2(out2)
        out2 += out1
        return F.relu(out2)

block = BasicResidualBlock(64, 64)
block(torch.rand(1, 64, 56, 56)).shape
```

Out[15]:
```
torch.Size([1, 64, 56, 56])
```

Let's build a U-Net for image segmentation!



U-Net is a network architecture for image segmentation (but it has worked well also in other domains, as image inpainting and image generation) which is composed of two main substructures:

1. a shrinking phase, where a cascade of convolutions and maxpooling progressively reduces the size of the images

    - each convolution has a padding of 0, so effectively also the regular convolution reduces the size of the image
2. a growing phase, where convolutions are intertwined with transposed convolutions to increase the size of the images
    - to mimic the behavior of the maxpooling in the shrinking phase, the transposed convolution has the same parameters: stride and kenrel size of 2, padding of 0.

The two phases are organized in levels, s.t. the output of the final convolution of a shrinking level (before maxpool) gets cropped, copied, and concatenated to the input of the corresponding level of the growing phase (after transposed convolution).

After the growing phase, we have the segmentation is performed by classifying each pixel of the resulting output image: in this setting, we have an image of size $h' \times w' \times C$ where C is the number of classes, and we do a pixel-level softmax, so we're able to classify each pixel.

To build a UNet, we will first build two structure:

1. The Shrinking block
2. The Growing block

Then put all the pieces together in a final class

In [18]:
```python
class ConvBlockDownsample(nn.Module):
    def __init__(self, in_channels, out_channels):
        super().__init__()
        self.mod = nn.Sequential(
            nn.Conv2d(in_channels=in_channels, out_channels=out_channels, kernel_size=3),
            nn.ReLU(),
            nn.Conv2d(in_channels=out_channels, out_channels=out_channels, kernel_size=3),
            nn.ReLU(),

        )
        self.downsample = nn.MaxPool2d(kernel_size=2)

    def forward(self, data):
        out = self.mod(data)
        out2 = self.downsample(out)
        return out, out2

block = ConvBlockDownsample(3, 64)
output_for_upsample, output = block(torch.rand(1, 3, 572, 572))
print("Shape of output: ", output.shape)
print("Shape of output for upsample: ", output_for_upsample.shape)
```

```
Shape of output:  torch.Size([1, 64, 284, 284])
Shape of output for upsample:  torch.Size([1, 64, 568, 568])
```

In [25]:
```python
class ConvBlockUpsample(nn.Module):
    def __init__(self, in_channels, out_channels):
        super().__init__()
        self.upsample = nn.ConvTranspose2d(in_channels=in_channels, out_channels=out_channels, kernel_size=2, stride=
        self.conv = nn.Sequential(
            nn.Conv2d(in_channels=in_channels, out_channels=out_channels, kernel_size=3),
            nn.ReLU(),
            nn.Conv2d(in_channels=out_channels, out_channels=out_channels, kernel_size=3),
            nn.ReLU()
        )

    def forward(self, data, data_from_downsample:torch.Tensor):
        out = self.upsample(data)
        # data has dimension ch x h x w
        # data_from_downsample has dimension ch x H x W
        # with H > h, W > w
        h, w = out.shape[2], out.shape[3]
        H, W = data_from_downsample.shape[2], data_from_downsample.shape[3]
        # do a center crop of data_from_downsample
        # (starting from H//2, W//2, the center pixel of the larger image)
        cropped_data_from_downsample = data_from_downsample[:, :, H//2-h//2 : H//2+(h//2 + h%2), W//2-w//2 : W//2+(w/
        out = torch.cat([out, cropped_data_from_downsample], dim=1)
        return self.conv(out)

block = ConvBlockUpsample(128, 64)
data_before_upsample = torch.rand((1, 128, 196, 196))
# we do the forward pass by output_for_upsample from before
block(data_before_upsample, output_for_upsample).shape
```

```
torch.Size([1, 64, 392, 392])
392 392
568 568
torch.Size([1, 64, 392, 392])
```

Out[25]:
```
torch.Size([1, 64, 388, 388])
```

References and additional material

- tutorial for building a LeNet CNN on KMNIST (+ training)
- ResNets: Deep Residual Networks for Image Recognition
- UNet: U-Net: Convolutional Networks for Biomedical Image Segmentation