Creating and training a U-Net model with PyTorch for 2D & 3D semantic segmentation: Model building [2/4]

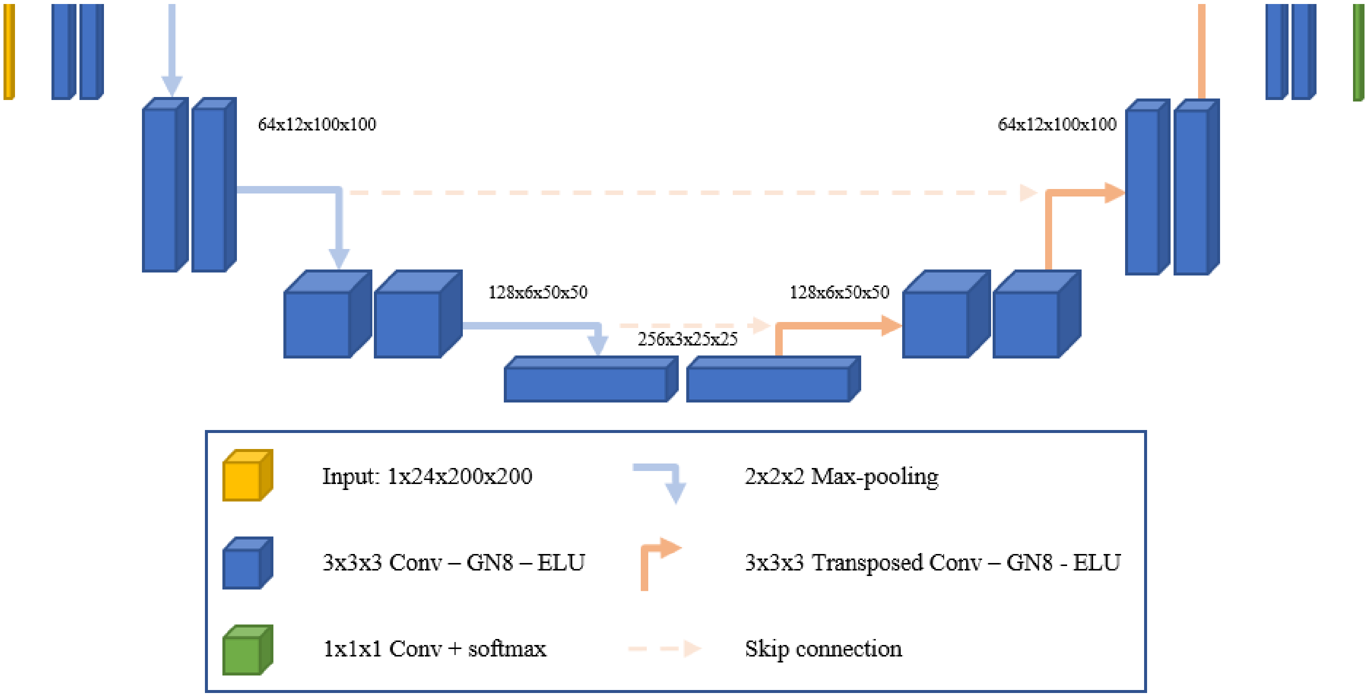
A guide to semantic segmentation with PyTorch and the U-Net

Johannes Schmidt Dec 2, 2020 · 6 min read

In the previous chapter we built a dataloader that picks up our images and performs some transformations and augmentations so that they can be fed in batches to a neural network like the U-Net. In this part, we focus on building a U-Net from scratch with the PyTorch library. The goal is to implement the U-Net in such a way, that important model configurations such as the activation function or the depth can be passed as arguments when creating the model.

About the U-Net

The U-Net is a convolutional neural network architecture that is designed for fast and precise segmentation of images. It has performed extremely well in several challenges and to this day, it is one of the most popular end-to-end architectures in the field of semantic segmentation.



The UNet — Image by Johannes Schmidt — Based on https://arxiv.org/abs/1505.04597

We can split the network into two parts: The encoder path (backbone) and the decoder path. The encoder captures features at different scales of the images by using a traditional stack of convolutional and max pooling layers. Concretely speaking, a block in the encoder consists of the repeated use of two convolutional layers (k=3, s=1), each followed by a non-linearity layer, and a max-pooling layer (k=2, s=2). For every convolution block and its associated max pooling operation, the number of feature maps is doubled to ensure that the network can learn the complex structures effectively.

The decoder path is a symmetric expanding counterpart that uses transposed convolutions. This type of convolutional layer is an up-sampling method with trainable parameters and performs the reverse of (down)pooling layers such as the max pool. Similar to the encoder, each convolution block is followed by such an up-convolutional layer. The number of feature maps is halved in every block. Because recreating a segmentation mask from a small feature map is a rather difficult task for the network, the output after every up-convolutional layer is appended by the feature maps of the corresponding encoder block. The feature maps of the encoder layer are cropped if the dimensions exceed the one of the corresponding decoder layers.

In the end, the output passes another convolution layer (k=1, s=1) with the number of feature maps being equal to the number of defined labels. The result is a u-shaped convolutional network that offers an elegant solution for good localization and use of context. Let’s take a look at the code.

The code

This code is based on https://github.com/ELEKTRONN/elektronn3/blob/master/elektronn3/models/unet.py (c) 2017 Martin Drawitsch, released under MIT License, which implements a configurable (2D/3D) U-Net with user-defined network depth and a few other improvements of the original architecture. They themselves actually used the 2D code from Jackson Huang https://github.com/jaxony/unet-pytorch.

Here is a simplified version of the code — saved in a file `unet.py` :

```
from torch import nn
import torch


@torch.jit.script
def autocrop(encoder_layer: torch.Tensor, decoder_layer: torch.Tensor):
    """
    Center-crops the encoder_layer to the size of the decoder_layer,
    so that merging (concatenation) between levels/blocks is possible.
    This is only necessary for input sizes != 2**n for 'same' padding and always requir
    """
    if encoder_layer.shape[2:] != decoder_layer.shape[2:]:
        ds = encoder_layer.shape[2:]
        es = decoder_layer.shape[2:]
        assert ds[0] >= es[0]
        assert ds[1] >= es[1]
        if encoder_layer.dim() == 4:  # 2D
            encoder_layer = encoder_layer[
                            :,
                            :,
                            ((ds[0] - es[0]) // 2):((ds[0] + es[0]) // 2),
                            ((ds[1] - es[1]) // 2):((ds[1] + es[1]) // 2)
                            ]
        elif encoder_layer.dim() == 5:  # 3D
            assert ds[2] >= es[2]
            encoder_layer = encoder_layer[
                            :,
                            :,
                            ((ds[0] - es[0]) // 2):((ds[0] + es[0]) // 2),
```

```
                            ((ds[0] - es[0]) // 2):((ds[0] + es[0]) // 2),
                            ((ds[1] - es[1]) // 2):((ds[1] + es[1]) // 2),
                            ((ds[2] - es[2]) // 2):((ds[2] + es[2]) // 2),
                            ]
    return encoder_layer, decoder_layer


def conv_layer(dim: int):
    if dim == 3:
        return nn.Conv3d
    elif dim == 2:
        return nn.Conv2d


def get_conv_layer(in_channels: int,
                   out_channels: int,
                   kernel_size: int = 3,
                   stride: int = 1,
                   padding: int = 1,
                   bias: bool = True,
                   dim: int = 2):
    return conv_layer(dim)(in_channels, out_channels, kernel_size=kernel_size, stride=s
                           bias=bias)


def conv_transpose_layer(dim: int):
    if dim == 3:
        return nn.ConvTranspose3d
    elif dim == 2:
        return nn.ConvTranspose2d


def get_up_layer(in_channels: int,
                 out_channels: int,
                 kernel_size: int = 2,
                 stride: int = 2,
                 dim: int = 3,
                 up_mode: str = 'transposed',
                 ):
    if up_mode == 'transposed':
        return conv_transpose_layer(dim)(in_channels, out_channels, kernel_size=kernel_
    else:
        return nn.Upsample(scale_factor=2.0, mode=up_mode)

```

```
def maxpool_layer(dim: int):
    if dim == 3:
        return nn.MaxPool3d
    elif dim == 2:
        return nn.MaxPool2d


def get_maxpool_layer(kernel_size: int = 2,
                      stride: int = 2,
                      padding: int = 0,
                      dim: int = 2):
    return maxpool_layer(dim=dim)(kernel_size=kernel_size, stride=stride, padding=paddi


def get_activation(activation: str):
    if activation == 'relu':
        return nn.ReLU()
    elif activation == 'leaky':
        return nn.LeakyReLU(negative_slope=0.1)
    elif activation == 'elu':
        return nn.ELU()


def get_normalization(normalization: str,
                      num_channels: int,
                      dim: int):
    if normalization == 'batch':
        if dim == 3:
            return nn.BatchNorm3d(num_channels)
        elif dim == 2:
            return nn.BatchNorm2d(num_channels)
    elif normalization == 'instance':
        if dim == 3:
            return nn.InstanceNorm3d(num_channels)
        elif dim == 2:
            return nn.InstanceNorm2d(num_channels)
    elif 'group' in normalization:
        num_groups = int(normalization.partition('group')[-1])  # get the group size fr
        return nn.GroupNorm(num_groups=num_groups, num_channels=num_channels)


class Concatenate(nn.Module):
    def __init__(self):
        super(Concatenate, self).__init__()

```

```
    def forward(self, layer_1, layer_2):
        x = torch.cat((layer_1, layer_2), 1)

        return x


class DownBlock(nn.Module):
    """
    A helper Module that performs 2 Convolutions and 1 MaxPool.
    An activation follows each convolution.
    A normalization layer follows each convolution.
    """

    def __init__(self,
                 in_channels: int,
                 out_channels: int,
                 pooling: bool = True,
                 activation: str = 'relu',
                 normalization: str = None,
                 dim: str = 2,
                 conv_mode: str = 'same'):
        super().__init__()

        self.in_channels = in_channels
        self.out_channels = out_channels
        self.pooling = pooling
        self.normalization = normalization
        if conv_mode == 'same':
            self.padding = 1
        elif conv_mode == 'valid':
            self.padding = 0
        self.dim = dim
        self.activation = activation

        # conv layers
        self.conv1 = get_conv_layer(self.in_channels, self.out_channels, kernel_size=3,
                                    bias=True, dim=self.dim)
        self.conv2 = get_conv_layer(self.out_channels, self.out_channels, kernel_size=3
                                    bias=True, dim=self.dim)

        # pooling layer
        if self.pooling:
            self.pool = get_maxpool_layer(kernel_size=2, stride=2, padding=0, dim=self.
```

```
        # activation layers
        self.act1 = get_activation(self.activation)
        self.act2 = get_activation(self.activation)

        # normalization layers
        if self.normalization:
            self.norm1 = get_normalization(normalization=self.normalization, num_channe
                                           dim=self.dim)
            self.norm2 = get_normalization(normalization=self.normalization, num_channe
                                           dim=self.dim)

    def forward(self, x):
        y = self.conv1(x)  # convolution 1
        y = self.act1(y)  # activation 1
        if self.normalization:
            y = self.norm1(y)  # normalization 1
        y = self.conv2(y)  # convolution 2
        y = self.act2(y)  # activation 2
        if self.normalization:
            y = self.norm2(y)  # normalization 2

        before_pooling = y  # save the outputs before the pooling operation
        if self.pooling:
            y = self.pool(y)  # pooling
        return y, before_pooling


class UpBlock(nn.Module):
    """
    A helper Module that performs 2 Convolutions and 1 UpConvolution/Upsample.
    An activation follows each convolution.
    A normalization layer follows each convolution.
    """

    def __init__(self,
                 in_channels: int,
                 out_channels: int,
                 activation: str = 'relu',
                 normalization: str = None,
                 dim: int = 3,
                 conv_mode: str = 'same',
                 up_mode: str = 'transposed'
                 ):
        super().__init__()

```

```
        self.in_channels = in_channels
        self.out_channels = out_channels
        self.normalization = normalization
        if conv_mode == 'same':
            self.padding = 1
        elif conv_mode == 'valid':
            self.padding = 0
        self.dim = dim
        self.activation = activation
        self.up_mode = up_mode

        # upconvolution/upsample layer
        self.up = get_up_layer(self.in_channels, self.out_channels, kernel_size=2, stri
                               up_mode=self.up_mode)

        # conv layers
        self.conv0 = get_conv_layer(self.in_channels, self.out_channels, kernel_size=1,
                                    bias=True, dim=self.dim)
        self.conv1 = get_conv_layer(2 * self.out_channels, self.out_channels, kernel_si
                                    padding=self.padding,
                                    bias=True, dim=self.dim)
        self.conv2 = get_conv_layer(self.out_channels, self.out_channels, kernel_size=3
                                    bias=True, dim=self.dim)

        # activation layers
        self.act0 = get_activation(self.activation)
        self.act1 = get_activation(self.activation)
        self.act2 = get_activation(self.activation)

        # normalization layers
        if self.normalization:
            self.norm0 = get_normalization(normalization=self.normalization, num_channe
                                           dim=self.dim)
            self.norm1 = get_normalization(normalization=self.normalization, num_channe
                                           dim=self.dim)
            self.norm2 = get_normalization(normalization=self.normalization, num_channe
                                           dim=self.dim)

        # concatenate layer
        self.concat = Concatenate()

    def forward(self, encoder_layer, decoder_layer):
        """ Forward pass
        Arguments:
            encoder_layer: Tensor from the encoder pathway
```

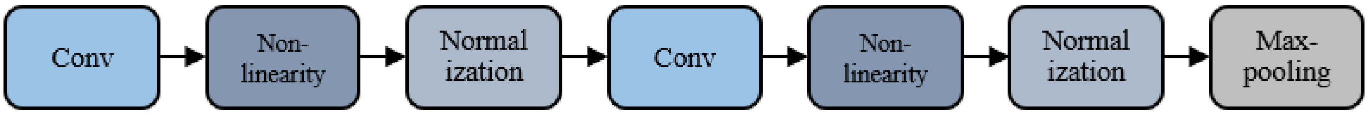
```
            encoder_layer: Tensor from the encoder pathway
            decoder_layer: Tensor from the decoder pathway (to be up'd)
        """
        up_layer = self.up(decoder_layer)  # up-convolution/up-sampling
        cropped_encoder_layer, dec_layer = autocrop(encoder_layer, up_layer)  # croppin

        if self.up_mode != 'transposed':
            # We need to reduce the channel dimension with a conv layer
            up_layer = self.conv0(up_layer)  # convolution 0
        up_layer = self.act0(up_layer)  # activation 0
        if self.normalization:
            up_layer = self.norm0(up_layer)  # normalization 0

        merged_layer = self.concat(up_layer, cropped_encoder_layer)  # concatenation
        y = self.conv1(merged_layer)  # convolution 1
        y = self.act1(y)  # activation 1
        if self.normalization:
            y = self.norm1(y)  # normalization 1
        y = self.conv2(y)  # convolution 2
        y = self.act2(y)  # acivation 2
        if self.normalization:
            y = self.norm2(y)  # normalization 2
        return y


class UNet(nn.Module):
    def __init__(self,
                 in_channels: int = 1,
                 out_channels: int = 2,
                 n_blocks: int = 4,
                 start_filters: int = 32,
                 activation: str = 'relu',
                 normalization: str = 'batch',
                 conv_mode: str = 'same',
                 dim: int = 2,
                 up_mode: str = 'transposed'
                 ):
        super().__init__()

        self.in_channels = in_channels
        self.out_channels = out_channels
        self.n_blocks = n_blocks
        self.start_filters = start_filters
        self.activation = activation
        self.normalization = normalization
```

```
        self.conv_mode = conv_mode
        self.dim = dim
        self.up_mode = up_mode

        self.down_blocks = []
        self.up_blocks = []

        # create encoder path
        for i in range(self.n_blocks):
            num_filters_in = self.in_channels if i == 0 else num_filters_out
            num_filters_out = self.start_filters * (2 ** i)
            pooling = True if i < self.n_blocks - 1 else False

            down_block = DownBlock(in_channels=num_filters_in,
                                   out_channels=num_filters_out,
                                   pooling=pooling,
                                   activation=self.activation,
                                   normalization=self.normalization,
                                   conv_mode=self.conv_mode,
                                   dim=self.dim)

            self.down_blocks.append(down_block)

        # create decoder path (requires only n_blocks-1 blocks)
        for i in range(n_blocks - 1):
            num_filters_in = num_filters_out
            num_filters_out = num_filters_in // 2

            up_block = UpBlock(in_channels=num_filters_in,
                               out_channels=num_filters_out,
                               activation=self.activation,
                               normalization=self.normalization,
                               conv_mode=self.conv_mode,
                               dim=self.dim,
                               up_mode=self.up_mode)

            self.up_blocks.append(up_block)

        # final convolution
        self.conv_final = get_conv_layer(num_filters_out, self.out_channels, kernel_siz
                                         bias=True, dim=self.dim)

        # add the list of modules to current module
        self.down_blocks = nn.ModuleList(self.down_blocks)
        self.up_blocks = nn.ModuleList(self.up_blocks)
```

```python

        # initialize the weights
        self.initialize_parameters()

    @staticmethod
    def weight_init(module, method, **kwargs):
        if isinstance(module, (nn.Conv3d, nn.Conv2d, nn.ConvTranspose3d, nn.ConvTranspo
            method(module.weight, **kwargs)  # weights

    @staticmethod
    def bias_init(module, method, **kwargs):
        if isinstance(module, (nn.Conv3d, nn.Conv2d, nn.ConvTranspose3d, nn.ConvTranspo
            method(module.bias, **kwargs)  # bias

    def initialize_parameters(self,
                              method_weights=nn.init.xavier_uniform_,
                              method_bias=nn.init.zeros_,
                              kwargs_weights={},
                              kwargs_bias={}
                              ):
        for module in self.modules():
            self.weight_init(module, method_weights, **kwargs_weights)  # initialize we
            self.bias_init(module, method_bias, **kwargs_bias)  # initialize bias

    def forward(self, x: torch.tensor):
        encoder_output = []

        # Encoder pathway
        for module in self.down_blocks:
            x, before_pooling = module(x)
            encoder_output.append(before_pooling)

        # Decoder pathway
        for i, module in enumerate(self.up_blocks):
            before_pool = encoder_output[-(i + 2)]
            x = module(before_pool, x)

        x = self.conv_final(x)

        return x
```

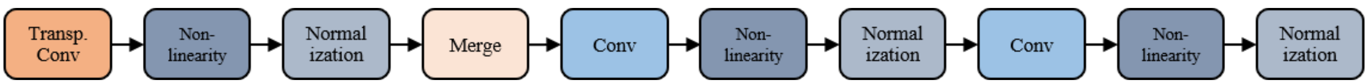
path. As you can see in `unet.py` the `DownBlock` and the `UpBlock` help to build the architecture. Both use smaller helper functions that return the correct layer, depending on what arguments are passed , e.g. if a 2D (`dim=2`) or 3D (`dim=3`) network is wanted. The number of blocks is defined by the `depth` of the network.

A `DownBlock` generally has the following scheme:



DownBlock — Image by Johannes Schmidt

A `UpBlock` has the following layers:



UpBlock — Image by Johannes Schmidt

For our `Unet` class we just need to combine these blocks and make sure that the correct layers from the encoder are concatenated to the decoder (skip pathways). These layers have to be cropped if their sizes do not match with the corresponding layers from the decoder. In such cases, the `autocrop` function is used. For merging, I concatenate along the channel dimension (see `Concatenate`). Instead of transposed convolutions we could also use upsampling layers (interpolation methods) that are followed by a 1x1 or 3x3 convolution block to reduce the channel dimension. Using interpolation generally gets rid of the checkerboard artifact. For 3D input consider using trilinear interpolation.

At the end we just need to think about the parameter initialization. By default, the weights are initialized with `torch.nn.init.xavier_uniform_` and the biases are initialized with zeros using `torch.nn.init.zeros_` .

For details and the available parameter options, I encourage you to take a look at the code. Feel free to change the code to your needs or expand e.g. the number of activation functions.

Creating a U-Net model

Let's create such a model and use it to make a prediction on some random input:

```
from unet import UNet

model = UNet(in_channels=1,
             out_channels=2,
             n_blocks=4,
             start_filters=32,
             activation='relu',
             normalization='batch',
             conv_mode='same',
             dim=2)

x = torch.randn(size=(1, 1, 512, 512), dtype=torch.float32)
with torch.no_grad():
    out = model(x)

print(f'Out: {out.shape}')
```

This will give us:

```
Out: torch.Size([1, 2, 512, 512])
```

To check weather our model is correct, we can get the model's summary with this package pytorch-summary:

```
from torchsummary import summary
summary = summary(model, (1, 512, 512))
```

which prints out a summary like this:

```
================================================================================
Layer (type:depth-idx)                 Output Shape              Param #
================================================================================
├─ModuleList: 1                        []                        --
|    └─DownBlock: 2-1                  [-1, 32, 256, 256]        --
|    |    └─Conv2d: 3-1                [-1, 32, 512, 512]        320
|    |    └─ReLU: 3-2                  [-1, 32, 512, 512]        --
|    |    └─BatchNorm2d: 3-3           [-1, 32, 512, 512]        64
|    |    └─Conv2d: 3-4                [-1, 32, 512, 512]        9,248
|    |    └─ReLU: 3-5                  [-1, 32, 512, 512]        --
|    |    └─BatchNorm2d: 3-6           [-1, 32, 512, 512]        64
```

```
|    |    └─BatchNorm2d: 3-6              [-1, 32, 512, 512]       64
|    |    └─MaxPool2d: 3-7                [-1, 32, 256, 256]       --
|    └─DownBlock: 2-2                     [-1, 64, 128, 128]       --
|    |    └─Conv2d: 3-8                   [-1, 64, 256, 256]       18,496
|    |    └─ReLU: 3-9                     [-1, 64, 256, 256]       --
|    |    └─BatchNorm2d: 3-10             [-1, 64, 256, 256]       128
|    |    └─Conv2d: 3-11                  [-1, 64, 256, 256]       36,928
|    |    └─ReLU: 3-12                    [-1, 64, 256, 256]       --
|    |    └─BatchNorm2d: 3-13             [-1, 64, 256, 256]       128
|    |    └─MaxPool2d: 3-14               [-1, 64, 128, 128]       --
|    └─DownBlock: 2-3                     [-1, 128, 64, 64]        --
|    |    └─Conv2d: 3-15                  [-1, 128, 128, 128]      73,856
|    |    └─ReLU: 3-16                    [-1, 128, 128, 128]      --
|    |    └─BatchNorm2d: 3-17             [-1, 128, 128, 128]      256
|    |    └─Conv2d: 3-18                  [-1, 128, 128, 128]      147,584
|    |    └─ReLU: 3-19                    [-1, 128, 128, 128]      --
|    |    └─BatchNorm2d: 3-20             [-1, 128, 128, 128]      256
|    |    └─MaxPool2d: 3-21               [-1, 128, 64, 64]        --
|    └─DownBlock: 2-4                     [-1, 256, 64, 64]        --
|    |    └─Conv2d: 3-22                  [-1, 256, 64, 64]        295,168
|    |    └─ReLU: 3-23                    [-1, 256, 64, 64]        --
|    |    └─BatchNorm2d: 3-24             [-1, 256, 64, 64]        512
|    |    └─Conv2d: 3-25                  [-1, 256, 64, 64]        590,080
|    |    └─ReLU: 3-26                    [-1, 256, 64, 64]        --
|    |    └─BatchNorm2d: 3-27             [-1, 256, 64, 64]        512
├─ModuleList: 1                           []                       --
|    └─UpBlock: 2-5                       [-1, 128, 128, 128]      --
|    |    └─ConvTranspose2d: 3-28         [-1, 128, 128, 128]      131,200
|    |    └─ReLU: 3-29                    [-1, 128, 128, 128]      --
|    |    └─BatchNorm2d: 3-30             [-1, 128, 128, 128]      256
|    |    └─Concatenate: 3-31             [-1, 256, 128, 128]      --
|    |    └─Conv2d: 3-32                  [-1, 128, 128, 128]      295,040
|    |    └─ReLU: 3-33                    [-1, 128, 128, 128]      --
|    |    └─BatchNorm2d: 3-34             [-1, 128, 128, 128]      256
|    |    └─Conv2d: 3-35                  [-1, 128, 128, 128]      147,584
|    |    └─ReLU: 3-36                    [-1, 128, 128, 128]      --
|    |    └─BatchNorm2d: 3-37             [-1, 128, 128, 128]      256
|    └─UpBlock: 2-6                       [-1, 64, 256, 256]       --
|    |    └─ConvTranspose2d: 3-38         [-1, 64, 256, 256]       32,832
|    |    └─ReLU: 3-39                    [-1, 64, 256, 256]       --
|    |    └─BatchNorm2d: 3-40             [-1, 64, 256, 256]       128
|    |    └─Concatenate: 3-41             [-1, 128, 256, 256]      --
|    |    └─Conv2d: 3-42                  [-1, 64, 256, 256]       73,792
|    |    └─ReLU: 3-43                    [-1, 64, 256, 256]       --
|    |    └─BatchNorm2d: 3-44             [-1, 64, 256, 256]       128
```

```
|    |    └─Conv2d: 3-45                [-1, 64, 256, 256]        36,928
|    |    └─ReLU: 3-46                  [-1, 64, 256, 256]        --
|    |    └─BatchNorm2d: 3-47           [-1, 64, 256, 256]        128
|    └─UpBlock: 2-7                     [-1, 32, 512, 512]        --
|    |    └─ConvTranspose2d: 3-48       [-1, 32, 512, 512]        8,224
|    |    └─ReLU: 3-49                  [-1, 32, 512, 512]        --
|    |    └─BatchNorm2d: 3-50           [-1, 32, 512, 512]        64
|    |    └─Concatenate: 3-51           [-1, 64, 512, 512]        --
|    |    └─Conv2d: 3-52                [-1, 32, 512, 512]        18,464
|    |    └─ReLU: 3-53                  [-1, 32, 512, 512]        --
|    |    └─BatchNorm2d: 3-54           [-1, 32, 512, 512]        64
|    |    └─Conv2d: 3-55                [-1, 32, 512, 512]        9,248
|    |    └─ReLU: 3-56                  [-1, 32, 512, 512]        --
|    |    └─BatchNorm2d: 3-57           [-1, 32, 512, 512]        64
├─Conv2d: 1-1                           [-1, 2, 512, 512]         66
==========================================================================================
Total params: 1,928,322
Trainable params: 1,928,322
Non-trainable params: 0
Total mult-adds (M): -14267.05
==========================================================================================
Input size (MB): 1.00
Forward/backward pass size (MB): 1156.00
Params size (MB): 7.36
Estimated Total Size (MB): 1164.36
==========================================================================================
```

About input sizes

To ensure correct semantic concatenations, it is advised to use input sizes that return even spatial dimensions in every block but the last in the encoder. For example: An input size of 120^2 gives intermediate output shapes of [60^2, 30^2, 15^2] in the encoder path for a U-Net with `depth=4` . A U-Net with `depth=5` with the same input size is not recommended, as a maxpooling operation on odd spatial dimensions (e.g. on a 15^2 input) should be avoided.

To make our lives easier, we can numerically compute the maximum network depth for a given input dimension with a simple function:

```
shape = 1920
```

```
def compute_max_depth(shape, max_depth=10, print_out=True):
    shapes = []
    shapes.append(shape)
    for level in range(1, max_depth):
        if shape % 2 ** level == 0 and shape / 2 ** level > 1:
            shapes.append(shape / 2 ** level)
            if print_out:
                print(f'Level {level}: {shape / 2 ** level}')
        else:
            if print_out:
                print(f'Max-level: {level - 1}')
            break

    return shapes


out = compute_max_depth(shape, print_out=True, max_depth=10)
```

This will output

```
Level 1: 960.0
Level 2: 480.0
Level 3: 240.0
Level 4: 120.0
Level 5: 60.0
Level 6: 30.0
Level 7: 15.0
Max-level: 7
```

which tells us that that we can design a U-Net as deep as this without having to worry about semantic mismatches. Conversely, we can also numerically determine the possible input shapes dimensions for a given depth:

```
low = 10
high = 512
depth = 8


def compute_possible_shapes(low, high, depth):
    possible_shapes = {}
    for shape in range(low, high + 1):
        shapes = compute_max_depth(shape,
                                   max_depth=depth,
                                   print_out=False)
```

```
            if len(shapes) == depth:
                possible_shapes[shape] = shapes

        return possible_shapes

    possible_shapes = compute_possible_shapes(low, high, depth)
```

This will output

```
{256: [256, 128.0, 64.0, 32.0, 16.0, 8.0, 4.0, 2.0],
 384: [384, 192.0, 96.0, 48.0, 24.0, 12.0, 6.0, 3.0],
 512: [512, 256.0, 128.0, 64.0, 32.0, 16.0, 8.0, 4.0]}
```

which tells us that we can have 3 different input shapes with such a level 8 U-Net architecture. But I dare to say that such a network with this input size is probably not useful in practice.

Summary

In this part we created a configurable UNet model for the purpose of semantic segmentation. Now that we have built our model, it is time to create a training loop in the next chapter.

Unet Python Pytorch Deep Learning Semantic Segmentation