Simon Fraser University

Image Segmentation

Johannes Merz, Ali Mahdavi Amiri

February 4, 2019

1 U-Net: Segmentation using Deep Learning

In the last years, a number of neural networks for image segmentation we re designed with considerable success. One of the most prominent attempts was the U-Net by Ronneberger et al. (2015) [1]. The name of the network stems from the symmetric shape of the architecture. As can be seen in Figure 1, a contracting path is followed by a symmetric expansive path.

1.1 Contracting Path (3 Points)

The contracting path is a convolutional network which doubles the channels of features to be learned in each downsampling step. Each of these steps applies two 3x3 unpadded convolutions in combination with ReLU layers. Afterwards, a 2x2 max pooling layer with a stride of 2 is used. The framework provides a basis for the implementation of the U-Net. It uses PyTorch, which provides you with a large amount of predefined layers that you can use to build your network. Read up here `https://pytorch.org/docs/stable/nn.html` and implement the contracting path (everything before the first green arrow in Figure 1) of the U-Net. Use the stub function *downStep()* in the file *model.py* for each convolutional layer and build the path with its help.

1.2 Expansive Path (5 Points)

In the expansive part, the data is upsampled using a 2x2 up-convolution (transposed convolution). Afterwards, two 3x3 convolutional layers (+ ReLU) are performed. In contrast to the downsampling, they halve the number of feature channels. In order to give the neural network more high resolution information, skip connections (gray arrows) between the contracting and expansive are utilized. Those are realized by cropping the result from the last

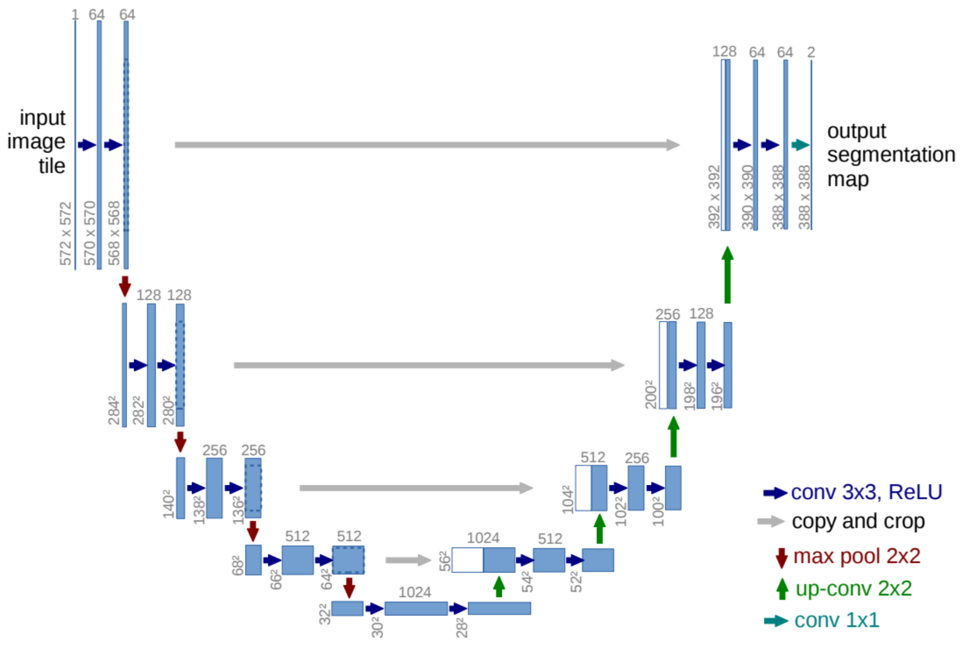

Figure 1.1: U-Net Architecture

convolution of the according contracting step (white box) to the same feature size as the result of the up-convolution (blue box) in the current expansive step. Then concatenate both results and use this as the input for the 3x3 convolutions. Finally, the segmented output is generated by a 1x1 convolution mapping to the defined number of classes / segments. As you have already done with the contracting path, implement the expansive path with the help of the function *upStep()* in the file *model.py*.

1.3 Loss Function (4 Points)

The loss function that is minimized by the optimization is a combination of a pixel-wise softmax over the output image and the cross entropy loss function. The former is defined as:

$$p_k(x) = \frac{e^{a_k(x)}}{\sum_{k'=1}^{K} e^{a_{k'}(x)}} \tag{1.1}$$

where $a_k(x)$ is the activation function (the output of the last layer) and K the number of classes for the segmentation. This function approximates 1 for the most likely segment class (k with the highest activation) and 0 for all others. The cross entropy can now be used to penalize the deviation of each pixel from its true label:

$$E = \frac{1}{M} \sum_{x \in \Omega} -\log(p_{l(x)}(x)). \tag{1.2}$$

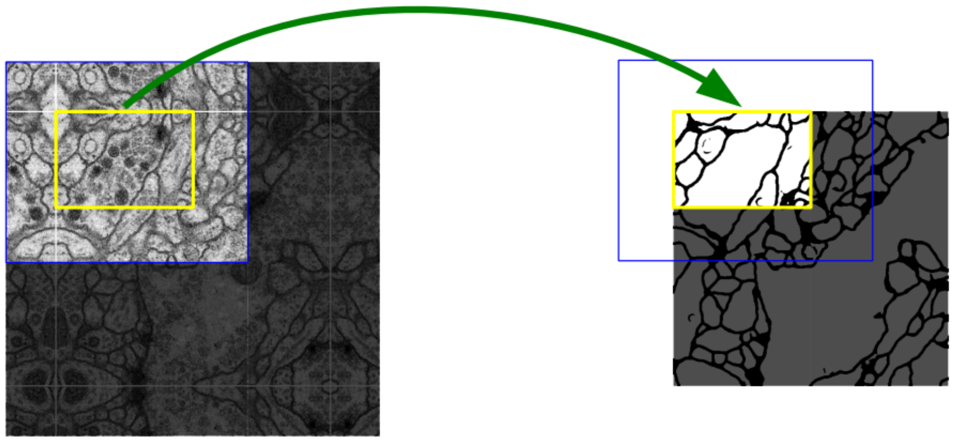

Figure 1.2: Overlap-tile strategy for seamless segmentation of arbitrary large images

Because $l(x)$ returns the true label of pixel x (known from the training dataset), $p_{l(x)}(x)$ should be 1. The cross entropy captures the pixel-wise difference of $p_{l(x)}$ to 1. M is the number of pixels in the image. Implement the function *getLoss()* in *train.py* to calculate the loss during the training. Make sure to only use mathematical functions provided by PyTorch (e.g. *torch.mean()*) and not by other libraries like Numpy (e.g. *numpy.mean()*) in order to stay compatible with the framework.

1.4 Data Loader (5 Points)

Now that we have the model of the network and the loss function set up, we can prepare the data that we want to use for training. The framework already provides you with a dataset. The dataset includes a number of images and corresponding label masks (targets). It is your task to load them by completing the code in *dataloader.py*. When loading the dataset there are some things that need to be considered. We are using greyscale images, so there is only one color channel. Also, before using the images, they need to be normalized to [0,1]. This improves the learning performance and the stability of our network. Lastly, in the original U-Net paper, the authors provide the following comment on the used data:

To allow a seamless tiling of the output segmentation map (see Figure 2), it is important to select the input tile size such that all 2x2 max-pooling operations are applied to a layer with an even x- and y-size.

Having the network architecture in mind, think about what this means for the data loader. While implementing the loader, make sure all the mentioned conditions are met.

1.5 Experiments (3 Points)

Run experiments with the provided dataset and the architecture you built and report your results. How are your test results? How high is your accuracy? For how many epochs did you train the network and how long did it take? Report your resulting segmentations of 5 test images and the achieved accuracy.

1.6 Data Augmentation (5 Points)

Now try the following two things and repeat your experiments. How do the results change?

Data Augmentation

Data Augmentation is a technique to *fake* more training data, by applying transformations to the training dataset. Implement the function *applyDataAugmentation()* in the data loader. You could try some of the following techniques:

- Flip the image horizontally or vertically
- Zoom images
- Rotate Images
- Apply gamma correction
- The authors of U-Net heavily relied on applying elastic deformations on the input data. In their paper they explain their approach like this:

  We generate smooth deformations using random displacement vectors on a coarse 3 by 3 grid. The displacements are sampled from a Gaussian distribution with 10 pixels standard deviation.

Implement above improvements and rerun your experiments. Did the results change?

References

[1] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*, pages 234–241. Springer, 2015.