10 Convolutional neural networks

Convolutional neural networks (CNNs) have many aspects in common with multilayer perceptrons (MLPs – fully connected feed-forward neural networks) such as being a directed acyclic graph with weighted edges and non-linear activations. CNNs use convolutions, or more precisely correlation filters, and the weights of the convolution kernels are the parameters that are optimized when training the network.

Since the edge weights of the CNN are implemented using convolutions, many of the edge weights have the same value. This is called weight sharing, which means that only one edge weight is learned for many edges. This results in a significant reduction in the number of model parameters, and therefore makes it possible to have much larger input data compared to MLP networks. Furthermore, the use of convolutions is very efficient and is used in both the forward pass and the backpropagation. During backpropagation, the weights of the convolution kernels are updated.

Working with images in 2D makes it possible to apply additional operations to the hidden layers. This includes for example a pooling step typically combined with a down-scaling step. Max-pooling is an example of a widely used pooling method, where pixels are replaced by the maximum in a local neighborhood. Max-pooling ensures that only the important features are kept, and makes the analysis robust to small translations. Several other operations can be applied, and an overview is given in chapter 9 in Goodfellow et al.[1].

Despite that CNNs have many aspects in common with MLPs, they are typically more complicated and therefore not as simple to implement. A large number of software frameworks are available, and training neural networks for many engineering applications will involve GPU processing. This is, however, implemented in a user-friendly way in many of the software packages and highly sophisticated neural networks can be developed using high-level programming using e.g. Python which makes these frameworks easy to use. We will work with PyTorch, which makes it easy to create models due to features

[1] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. *Deep learning*, volume 1. MIT press Cambridge, 2016

such as automatic gradient computation and easy utilization of GPUs (Graphics Processing Units).

10.1 Exercise

In this exercise, we will work with a model for segmentation, which has a structure similar to the U-net[2]. Our model is implemented in PyTorch and requires that you code in Python. Setting up the model and making sure that everything runs as expected is time-consuming, so we have prepared some code for you.

The U-net is designed for image segmentation and is a Fully Convolutional Network (FCN), meaning that all operations from the input to the output are convolutions. The name sounds like a fully connected network (MLP), but it is not the same. The FCN is a CNN containing convolutions, pooling, up-convolutions, and skip connections. The purpose of this exercise is to work with our segmentation network to understand the function of all the elements in the network, such that you will be able to use and modify the network for your application.

We will work with segmenting microscopy data of glands from patients with colon cancer and healthy controls. The microscopy images are shown in Figure 10.1. The data has been reshaped into 128-by-128 images in RGB and the labels are binary with the value 0 in the background and 255 in pixels labeled to contain glands. The task is to train our CNN to take an RGB image as input and give an image with labels as output.

[2] 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

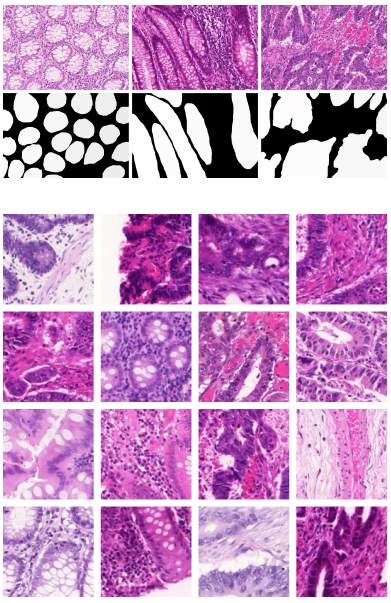

Figure 10.1: Data examples for the segmentation exercise. The top image shows an example of the original data. The task is to segment individual glands in the histological colon tissue. We have down-sampled to half size and cropped out 128-by-128 images for this exercise. The cropped images and their labels are shown in the middle and bottom. Data from: https://warwick.ac.uk/fac/cross_fac/tia/data/glascontest/

10.2 Suggested procedure

The exercise contains some steps to investigate the model. First, you will investigate a model with pre-trained weights and later you will train the model yourself. Since the model and the dataset is relatively small, the model can be trained within some minutes.

Taks 1: Load the model First you should follow the link to the code and run the cell that downloads the data. After that, you should run the cells that load the data, set up the model, and load the pre-trained model.

Task 2: Model overview To have an overview of the model, it is a good idea to make a sketch of the model on a piece of paper, which is similar to the drawing of the U-net. This means drawing boxes for the input image, internal neurons (images inside the network), and the output image, and write up their dimension. Also, draw arrows illustrating how layers are connected.

Through this drawing, you will see what the output image size is if you have the 128-by-128 input image. The model can also take images of other sizes, but not all sizes are valid. What is required for the sizes of images in this model?

Task 3: Run the model You can run the model on a test image that you can get from the data homepage. These images have the original shape, so you need to crop out a size that fits the model.

In cases where you have a large image, you will sometimes need to crop out smaller images and segment them individually. To see the effect of this, you should take one of the test images and crop it into 128-by-128 images, segment them individually, and put them together into a full image. You can choose to have the segmentations overlapping to smooth out boundary effects.

Visualize how a segmentation with the largest crop compares to a segmentation assembled from smaller crops.

Task 4: Weights You can look at the weights in the trained convolutional kernels and you can try to visualize some of them. They are, however, very small, so they might not be very informative. You should also investigate the size of the convolutional kernels. What happens to their sizes as you go deeper into the network? What is the number of learnable parameters?

Task 5: Hidden layers You can also investigate the hidden layers in the network. You should visualize the output of the hidden layers as images. What do you see? What is the size of the hidden layers? What is the number of hidden layers?

Task 6: Training You should now train the model. First, you will create random model parameters by creating a new model object. What is the output, when you have not trained the model?

You should run the model for several epochs. When should you stop training? How many epochs are needed?

You can try changing the learning rate, and see how that affects the model.

Task 7: Modify the model You can try changing the model. For example, you can try another optimizer, change the activation functions, or modify the architecture of the model.

You can also try working with data augmentation and dropout etc.