Fred Hamprecht, Alberto Bailoni
cvflecture.s20@gmail.com

General Regulations

- No individual submission! Hand in your solutions in groups of two or three people (preferred).
- Only include a single jupyter notebook per submission. Not one per person or separate ones per exercise.
- All outputs, in particular images, should be visible in the notebook. We should not have to run your notebook or uncomment lines / change some code to see your results!
- When asked, remember to comment / interpret your results.
- Every result or image you provide should be at least pointed to in the notebook (by file / directory name).
- Include all files/code that include your work in the zip file. Otherwise, it will receive no marks.
- Submit all your files in a single `.zip` archive (not `.rar` or `.tar`) and upload it using the Physics platform https://uebungen.physik.uni-heidelberg.de/h/1176. **Only one person in the group should upload the zip archive**: in the upload interface you will see the option to indicate your exercise-partners.

1 Train a CNN for Semantic Segmentation (Part 2) (12+8pt)

Finally, we get to do the real thing and train a CNN! The goal is to perform a foreground-background segmentation of images of yeast cells. In the jupyter notebook `ex04.ipynb` you find code for the first part of the exercise: loading the dataset, normalizing it and applying data augmentation.

As a semantic classification model we will use the UNet architecture described in the lecture. In the provided code you will find a highly configurable implementation of a UNet model[1] (see Fig. 1), so that you will be able to compare different architecture hyperparameters and test which combination performs best.

The output of the fully convolutional model should be an image with only one channel and the same spatial dimension as the input. For each pixel of the output, the model should predict a value of 1 if the pixel belongs to a yeast cell and a value of 0 otherwise.

(a) Loss for semantic segmentation (2pt) We will consider two types of loss function to train our semantic segmentation model: binary cross entropy[2] (already implemented in PyTorch `torch.nn.BCELoss`) and Dice loss. The second one is based on the Søresent-Dice coefficient, which is similar to the Intersection Over Union (IoU) score we have seen in the last assignment:

$$SDL(p, \hat{p}) = \frac{1}{N} \sum_{n=1}^{N} \left(1 - \frac{2 \sum_{xy} p_n(x,y) \cdot \hat{p}_n(x,y)}{\sum_{xy} p_n^2(x,y) + \sum_{xy} \hat{p}_n^2(x,y)} \right). \tag{1}$$

In this formula, $p_n(x,y)$ is the prediction of the model for pixel (x,y) and the image n in the training batch, whereas $\hat{p}_n(x,y)$ is the associated ground truth label (equal to 1 if the pixel belongs to a yeast cell and 0 otherwise). As compared to binary cross entropy, Dice loss was shown to produce crisper outputs and perform better in the presence of class imbalance[3]. Follow the instructions in `ex04.ipynb` to implement the loss functions.

[1] Implementation adapted from https://github.com/imagirom/ConfNets.
[2] More details about cross-entropy here: https://en.wikipedia.org/wiki/Cross_entropy
[3] See for example this paper for a comparison between the two loss functions: https://arxiv.org/abs/1807.10097

(b) Training a UNet model (10pt) In the jupyter notebook, we provide PyTorch code to train a CNN model. Read the python documentation strings of the `UNet` class in `cvf20/models/unet.py` to understand the meaning of each argument in the initialization method: in the notebook you can find an example of UNet model with depth 5 from which you can start your experiments.

Then choose one of the two losses we introduced in the previous task and train the model with Adam optimizer and a batch-size of 4: you can start using a learning rate of 10^{-4}, but you could need to experiment with this parameter and change it depending on the loss function you use.

Use `tensorboard` (see code in the notebook) to monitor the training loss and the output predictions throughout the training. Which of the two loss functions does perform better? What if you add some kind of normalization layer after each convolution in the model?

Report your training and validation set IoU scores in the different setups and comment your results by plotting some predictions on the validation set or by showing some screenshots of the tensorboard-plots. If you note recurrent mistakes in the predictions, comment them. At least one of your tested models should achieve IoU$= 0.91$ on the validation set.

(c) Get creative (8pt bonus) Now you can get creative and see if you can improve the performance of the previous semantic segmentation model! For each new experiment you run, do not forget to summarize in the notebook the changes you implemented together with comments and/or plots about the predictions and achieved validation scores. Keep in mind that even if you do not observe any noticeable improvement after testing your modified model, this is also a result.

In the following, we provide a few examples of things you could try out, but feel free to implement your own ideas:

- For each `encoder` and `decoder` module of the UNet model (see Fig. 1), add residual connections[4];
- Experiment with basic parameters of the `UNet` class, e.g. number of features, model depth, activation and normalization types, downscaling and upscaling modules;
- Augment images by rotating them by a random angle (and not only multiples of 90 degrees);
- Augment images with Elastic transformations: you can easily find on-line python code for this[5];
- Try different normalization procedures of the input images, e.g. subtracting background values from each image (`numpy.median` could be helpful for this).

Remark – If you implement your own type of model, please write its code in the file `cvf20/models/your_models.py` and try to start from the code we provide. By simply reimplementing some of the `construct_NAME_MODULE()` methods in the `UNet` class you will be able to build a highly personalized model with custom versions of the encoder, decoder, downsampling, upsampling modules shown in Fig. 1.

[4]You can find more informations on residual blocks here: https://towardsdatascience.com/residual-blocks-building-blocks-of-resnet-fd90ca15d6ec

[5]See for example the following implementation of elastic transformation: https://gist.github.com/chsasank/4d8f68caf01f041a6453e67fb30f8f5a

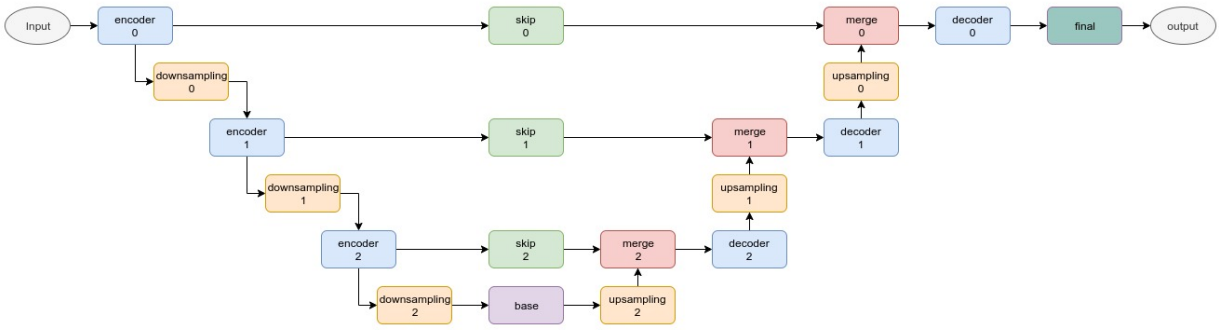

Figure 1: **Highly configurable UNet architecture** – Each of the blocks in the image represents a module of the architecture that can be easily configured by tweaking most of the hyperparameters, such as number of features, type of activation and normalization, number of convolutional layers in each encoder/decoder module, model depth (i.e. how many downscaling levels there are in the hierarchy of the UNet model). You can find the implementation of the `UNet` class in the file `cvf20/models/unet.py`