The aim of this laboratory is to extend and—potentially—improve the performance of the U-Net based pipeline that you implemented in your previous lab.

You will work on three main improvements:

1. Introducing **K-fold cross-validation** during training
2. Implementing a **weighted Dice coefficient loss function** that makes the training focus more on the boundaries of the ROIs to be segmented
3. Integrating **auto-context** into the pipeline by adding one extra input channel

For all tasks, please keep the same architecture that you have already implemented and just add all the requested extensions. Moreover, you can go through all the exercises by always keeping the following parameters:

- Image size = 240 × 240
- Number of filters at the first layer = 8
- Optimizer = Adam
- Learning rate = 1e-4
- Drop-out rate = 0.2
- Batch size = 8
- Evaluation metrics = Dice coefficient, precision, recall
- Batch normalization = True

General recommendation regarding the number of epochs: you are free to set the number as you wish in order to optimize your network (generally between 100 and 200 epochs may be required). However, please make sure to preliminary test your pipelines by setting a low number of epochs, even though this might of course give you a bad result. Once everything is working with no errors until the very end of the pipeline, you can start running longer training phases with more epochs.

Task 1: K-fold cross validation

With K-fold cross validation, you do not simply split the dataset into one training and one validation set, but rather test K different configurations of splitting between training and validation sets.

You can start by setting K = 3. You will need to implement a loop that goes through each of the 3 folds. At the beginning of each loop, you will call a function (which you need to define) that appropriately splits your image and mask files into the training and validation sets for that specific fold. If you have n images in total, then—for the first fold—you will use the first $n/3$ images as validation set and the remaining $2*n/3$ as training set. Instead, for the second fold, the first $n/3$ images and the last $n/3$ images will be your training set. For the last fold, the first $2*n/3$ images of the list will be used as training set.

Train your network in this manner on the brain tumor MRI dataset that you have already worked on in the previous lab (path: */DL_course_data/Lab3/MRI/Image/* for the MRI slices, */DL_course_data/Lab3/MRI/Mask/* for the tumor binary masks). Is the performance consistent

across all folds? How would you deal with datasets for which some folds have a different performance than others?

> **Bonus task:** you can run more tests by increasing the number of folds (up to a maximum of 10 folds) and analyze the performance.

Task 2: Introducing weight maps

In some segmentation tasks, some parts of the ROI can be more difficult to segment than others. For instance, in the task of tumor segmentation, the boundary of the tumor is not always very well defined and visible. Thus, it could be helpful to define weight maps that make it harder to minimize the loss function in such boundary regions, so that the learning process can focus more on this challenging part. You can now implement and test this strategy also on your brain tumor dataset.

First of all, you need to create binary masks on the boundaries of the tumor regions. You can define a function for this purpose. For every mask saved in the folder */DL_course_data/Lab3/MRI/Mask/*, the function will 1) dilate the mask, 2) erode it, 3) subtract the eroded mask from the dilated mask, 4) save the new binary boundary mask (1 along the boundaries, 0 in the rest of the image). You can use a radius of 2 for both the dilation and erosion operations. Before proceeding with the next step, please check whether your algorithm is actually working.

You can then store the list of the generated boundary mask files (that correspond to your "weight maps"), which you will use together with the image and mask lists during training. You can then start your cross-validation by first selecting the appropriate training and validation images, masks and weight maps.

After doing so, you should define the inputs for the model. Remember that you need to define two different types of inputs: one is the actual input image to be segmented, and the other represents the weight maps to be integrated in the loss function.

Finally, to be able to compile the model, you should define your weighted Dice loss function in the same way as you saw in the lecture. The weight map should be the first input to the loss function, and a weight strength can be integrated as well. You can start by setting a weight strength of 1. Once everything works, you can also try to test different strengths too in order to check whether the accuracy changes. Please also notice that, in the binary weight map defined above, higher values are assigned to the boundary regions (compared to the rest of the image that is set to 0). However, when weighting the Dice loss function, you are rather interested in assigning lower weights to the boundary regions. Thus, remember to **appropriately adapt the weight values to your task** (e.g. by adding 1 and then computing their inverse) before multiplying them:

```
def weighted_loss(weight_map, weight_strength):
    def weighted_dice_loss(y_true, y_pred):
        y_true_f = K.flatten(y_true)
        y_pred_f = K.flatten(y_pred)

        weight_f = K.flatten(weight_map)
        weight_f = weight_f * weight_strength
```

```
        weight_f = ## complete!

        weighted_intersection = K.sum(weight_f * (y_true_f * y_pred_f))

        return -(2. * weighted_intersection + K.epsilon()) / (K.sum(y_true_f)
+ K.sum(y_pred_f) + K.epsilon())
    return weighted_dice_loss
```

If you want to perform also data augmentation, you can get an image generator that also includes the weights with the following code (and functions defined below):

```
train_generator = generator_with_weights(x_train, y_train, weight_train,
Batch_size)
model_history = model.fit_generator(
            train_generator, steps_per_epoch=steps_per_epoch, epochs=n_epoch,
            verbose=1, max_queue_size=1, validation_steps=len(x_val),
            validation_data=([x_val, weight_val], y_val),
            shuffle=True, class_weight='auto')
```

```
def combine_generator(gen1, gen2, gen3):
    while True:
        x = gen1.next()
        y = gen2.next()
        w = gen3.next()
        yield([x, w], y)

def generator_with_weights(x_train, y_train, weights_train, batch_size):
    background_value = x_train.min()
    data_gen_args = dict(rotation_range=10.,
                         width_shift_range=0.1,
                         height_shift_range=0.1,
                         cval=background_value,
                         zoom_range=0.2,
                         horizontal_flip=True)

    image_datagen = ImageDataGenerator(**data_gen_args)
    mask_datagen = ImageDataGenerator(**data_gen_args)
    weights_datagen = ImageDataGenerator(**data_gen_args)

    image_generator = image_datagen.flow(x_train, shuffle=False,
                                         batch_size=batch_size,
                                         seed=1)

    mask_generator = mask_datagen.flow(y_train, shuffle=False,
                                       batch_size=batch_size,
                                       seed=1)

    weight_generator = weights_datagen.flow(weights_train, shuffle=False,
                                         batch_size=batch_size,
                                         seed=1)

    train_generator = combine_generator(image_generator, mask_generator,
                                        weight_generator)

    return train_generator
```

You can then finally start training your model. Can you observe a discrepancy between loss function and the traditional (un-weighted) dice coefficient evaluation metrics? Is it how you expected it to be? Which accuracy do you achieve?

Task 3: Adding autocontext

For this last task, you will implement the cyclical autocontext architecture that was discussed during the lecture. You can disregard the inclusion of weight maps for this task.

At each step s of the cycle (with s between 0 and T), you will need to run your K-fold cross-validation training by feeding the U-Net architecture with two input channels. The first channel is the original MRI slice to be segmented. The second channel is, instead, obtained as:

- If $s > 0$: the output segmentation for that specific image estimated during cross-validation at the step $s-1$.
- If $s=0$: an array having the exact same size as the input image but with all elements equal to 0.5.

At the end of every fold, you should compute the model predictions on the current validation set (which can be estimated with the command *model.predict*, as seen in the lecture) and store them in an array.

The final array including the predictions for the entire dataset (i.e. at the end of the very last fold) can then be saved before moving from step s to step $s+1$. In this way, at step $s+1$, the model predictions (segmentation outputs) obtained from step s can simply be loaded again and used for the new cycle of training as context layers (i.e. inputs of the second channel).

Please train the network with at least two cycles of training ($T=1$). Did the autocontext layer help for this specific segmentation task?

> **Bonus task**: you can also increase the number of cycles and see whether any differences in the performance can be observed.

Final observations:

Which training modality led to the best performance on your brain tumor dataset (between simple U-Net, U-Net with weighted dice loss and U-Net with autocontext channel)? What do you think could be additionally tested to further improve the performance?

Good luck!