Week 4: Deep neural networks for image segmentation

Artificial Intelligence for Aerospace Engineering

AE-2224II - Q3 2022/3

Guido de Croon

February 9, 2023

Background of the assignment: autonomous drone racing

Artificial Intelligence is a disruptive technology that can profoundly change aspects of our society. Since 2012 there has been great progress in AI, thanks to the rapid developments in "deep neural networks". The world champion in chess was beaten by IBM's Deep Blue in 1997, humans were beaten at Jeopardy by IBM's Walter in 2011, in Go by Google Deepmind's AlphaGo in 2016, and in StarCraft II in 2019. In aerospace, the next achievement may consist of beating human drone race pilots with an autonomous drone.

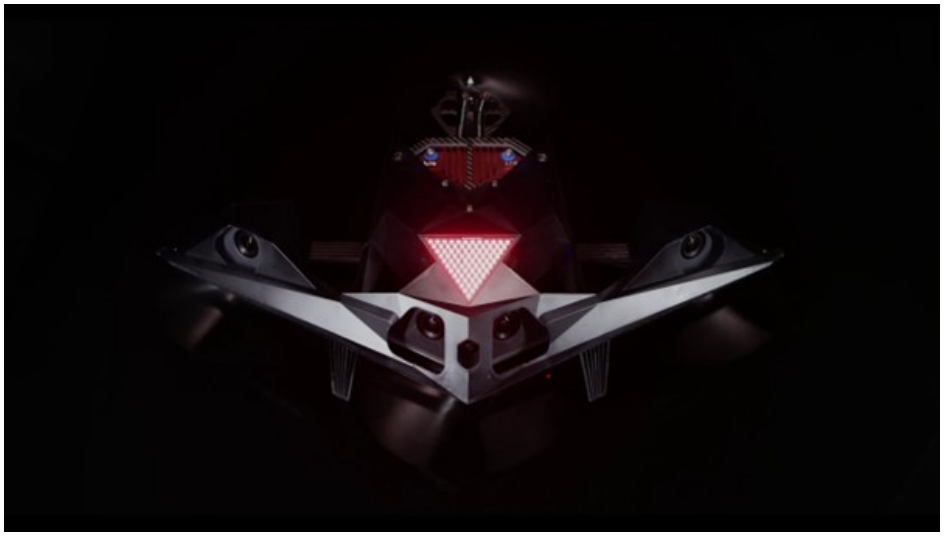

Figure 1: The AIRR racing drone, it has an Inertial Measurement Unit (IMU), a downward pointing laser ranger, 4 high-resolution cameras, and an NVidia Xavier board: 512-Core Volta GPU with Tensor Cores, 8-Core ARM v8.2 64-Bit CPU, 8 MB L2 + 4 MB L3., 16 GB 256-Bit LPDDR4x Memory., 32 GB eMMC 5.1 Flash Storage.

In 2019, the first AI Robotic Racing competition was organized. The goal was to (1) have autonomous

drones race together utilizing different approaches to AI and drone racing, and (2) have the winner compete with one of the world's best human drone race pilots, Gab707. 424 teams from 81 countries participated in the competition. At the end, 9 teams competed against each other in 3 seasonal races and 1 world championship race. The TU Delft team became 1st in the championship but lost against the human pilot. For more details see: http://mavlab.tudelft.nl/mavlab-wins-the-alpha-pilot-challenge/.

In the AIRR races, the drone (shown in Figure 1) had to pass through large gates designed by the competition organizers, the Drone Racing League and Lockheed Martin. Detecting these gates in the images from the drone's cameras was a vital task for the competition.

Aim: The assignment's goal is to design a fully convolutional network that automatically segments gates in drone images.

Dataset: The dataset consists of images and corresponding gate masks. The data is related to the scientific article:
[1] De Wagter, C., Paredes-Vallés, F., Sheth, N., and de Croon, G.C.H.E. (2022). The sensing state-estimation and control behind the winning entry to the 2019 artificial intelligence robotic racing competition. Field Robotics, 2, 1263-1290.

You will find the files necessary to complete this instruction in directory `week4/image-segmentation` of your docker container. From now on, all the paths will be relative to that directory.

Figure 2 shows an example onboard camera image, subject to considerable motion blur, and the corresponding gate mask. The gate masks have been created by team members that manually clicked on the inner and outer corners of the gate.

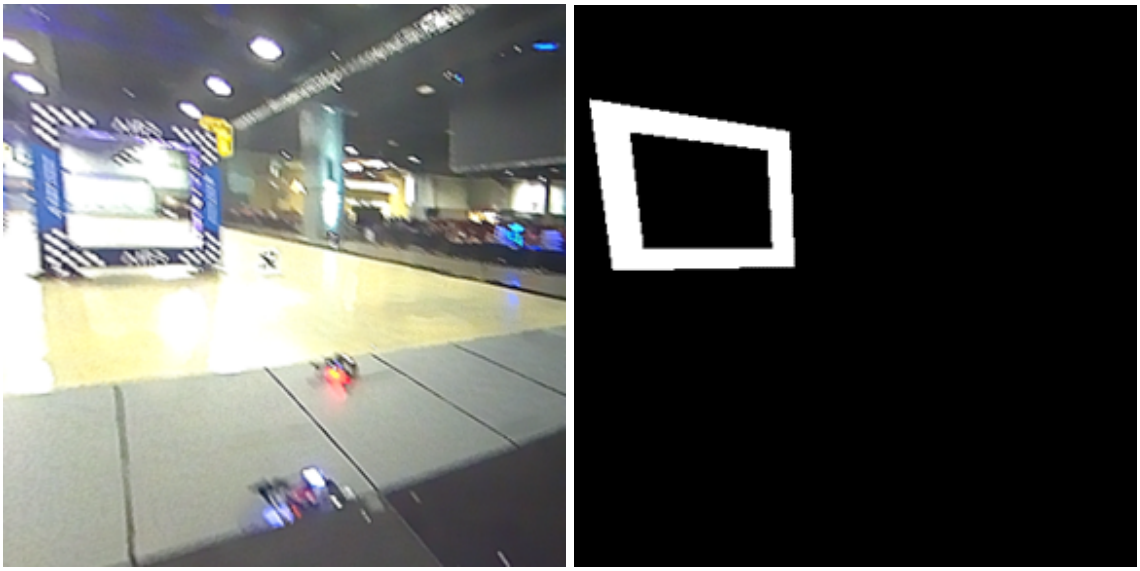

Figure 2: Example image and corresponding gate mask.

This lab session you will create a fully convolutional network that takes images as input and outputs a segmentation mask that should match the ground-truth gate masks as good as possible. You will also evaluate the performance of your method on a validation set. The assignment is accepted when the performance is of sufficient quality.

Neural network structure

For segmentation, you will use a U-Net. An example U-Net, as introduced in Ronneberger et al. 2015, is shown in Figure 3. Please note that the U-net is a *fully convolutional network* with an encoding part consisting of convolutions and a decoding part consisting of deconvolutions.

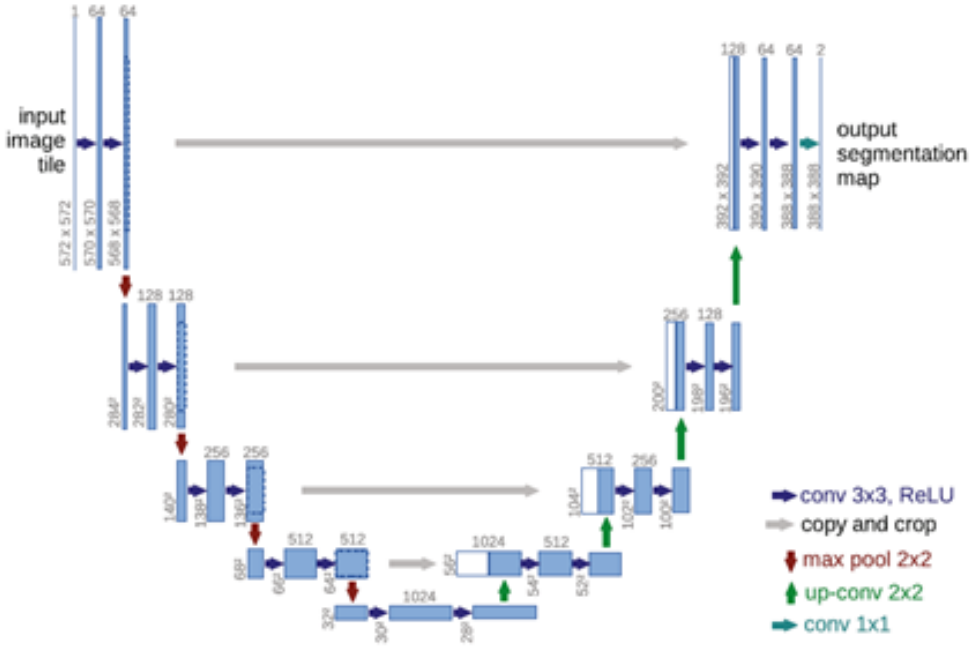

Figure 3: U-Net structure, where blue boxes represent multi-channel feature maps and white boxes represent copies of the dashed feature maps at the start of the arrow.

Loss function

We advise you to try the following cost functions, potentially using a weighted sum of these costs:

Binary cross-entropy loss:
The binary cross-entropy loss function for a single sample is defined as:

$$\mathcal{L}_{\text{BCE}} = -\left(t\log(p) + (1-t)\log(1-p)\right), \tag{1}$$

where p is the output of the network interpreted as a probability. So if the target t is 1, the cost function will be minimal when the probability is also one, and viceversa for a target of 0. Please note that the PyTorch implementation deals with the case in which p or $(1-p)$ are zero, preventing infinite values.

Part 1: Refining the parameters of a pretrained network

Given the time restrictions of the lab session, we advise you to start by refining a pretrained network. In particular, you will load a pretrained UNet as follows:

```
model = torch.hub.load('mateuszbuda/brain-segmentation-pytorch', 'unet',
        in_channels=3, out_channels=1, init_features=32, pretrained=True)
```

As the name suggests, this network was pretrained on brain MRI images. Hence, the original task for which it was trained is different from what we want to do today. Still, the lower level features learned by the network are likely to be useful for our task. Hence, we load the network and use it as a starting point for training.

Please do the following:

1. Study the code. Pay particular attention to the fact that we extended the `Dataset` class of PyTorch for our specific problem. This requires the implementation of an `__init__`, `__len__`, and `__getitem__` functions. For more details, see: https://pytorch.org/tutorials/beginner/basics/data_tutorial.html. Also note that we use the `DataLoader` class of PyTorch to load the data in batches, using `SubsetRandomSampler`. For more details, see: https://pytorch.org/tutorials/beginner/data_loading_tutorial.html. Also have a look at the training and validation loop.

2. Run the code with `train = True`. This will refine the UNet for our segmentation task on the training set. The training will run for 50 epochs. It should end up at a training loss of ≈ 0.10.

3. Run the code with `train = False`. This will allow you to inspect the outputs of the network on the images. You should press enter in the terminal to see the next image, mask, and predicted mask (see Fig. 4). You have to close the windows to get back to the input prompt in the terminal.

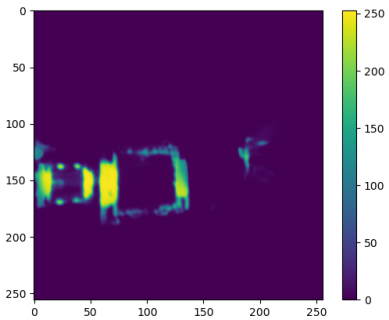

Figure 4: Example predicted mask after refinement.

Part 2: Changing the loss function.

You will now attempt to improve the performance in a number of ways. Please do the following:

Changing the loss function

In the article mentioned above[1], the authors use a weighted sum of the Dice loss and the binary cross-entropy loss.

Dice loss:
The Dice loss function for a single sample is defined as:

$$\mathcal{L}_{\text{Dice}} = 1 - \frac{2\sum_{i=1}^{N} p_i t_i}{\sum_{i=1}^{N} p_i^2 + \sum_{i=1}^{N} t_i^2}, \tag{2}$$

It has been developed to deal with skewed distributions, in which one class is more common than the other. In our case, there are many more background pixels than gate pixels. Hence, the Dice loss may be more appropriate than the binary cross-entropy loss. Please do and answer the following:

1. Implement the Dice loss function in the `loss_dice` function in file `loss_functions.py`. *Hint:* You can implement the loss function as a typical function, i.e., with `def dice_loss(pred, target)`, and assign it to the loss function in the `train_U_Net` function: `loss_fn = dice_loss`. That is, without using brackets.

2. Run the code with `train = True`. **Does the dice loss lead to better segmentation?** How can you tell? (As a sanity check: The training loss we get is ≈ 0.14 and Figure 5 shows an example output at the end of training.)

3. In the article, a weighting is used of 0.5 to combine cross-entropy and the dice-loss. Implement this weighted loss. **Does this weighting improve the segmentation?** (As a sanity check: the error we get on the training set ≈ 0.12)

Evaluation of results:
A first way in which the results can be evaluated is by looking at the segmentation masks. We call this a *qualitative* evaluation, as it is subjective and cannot be expressed with a number. A second way is to perform a *quantitative* evaluation, which can be expressed with a number. For segmentation, the most common metric is the *intersection over union* (IoU), which is defined as:

$$\text{IoU} = \frac{\sum_{i=1}^{N} p_i \cap t_i}{\sum_{i=1}^{N} p_i \cup t_i}, \tag{3}$$

where p_i and t_i are the predicted and target masks, respectively, $\cap$ the logical 'and', and $\cup$ the logical 'or'. The difference with the loss functions discussed above is that this function does not take floats. This is especially relevant to the floating point predictions by the neural network. We need to use a threshold on the prediction mask in order to calculate the IoU. We can use the IoU to compare the outcomes of the different training runs. However, this raises the following question: If we care most about IoU, why not use it directly as a cost function? Since it is a quantitative measure, it can also be used as a loss function during training. **Optional: implement the IoU function for quantitative comparison.**

Finally, the segmentations produced by the network were used by a downstream procedure of functions to detect the gates and hence position of the drone in the track. So an ideal final evaluation of the trained neural networks would actually involve the whole pipeline.

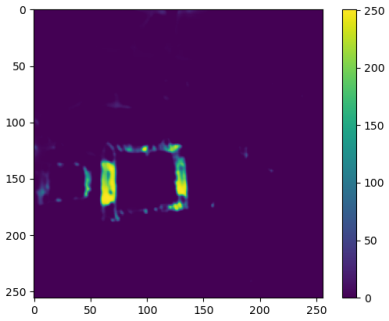

Figure 5: Example predicted mask after refinement, using the Dice loss.

Part 3: Data augmentation.

The availability of sufficient labelled data is essential for the success of supervised learning. However, such data is often scarce, as labelling requires human effort. In our case, the MAVLab drone race team members have labelled the images during the competition, which was very limited in time. Hence, the total dataset was already quite small, and - due to the restrictions for this lab session - has been even further reduced.

One major approach to deal with this problem is *data augmentation*, i.e., the automatic generation of new labelled data from the existing data set. This can be done in a number of ways, e.g., by rotating, flipping, or scaling the images. Other options include changes to the light or color properties of the images. When implementing data augmentation techniques, we do have to be careful not to destroy properties of the data that are important for the task at hand. For example, in our task a vertical flip does not make sense, as the drone will not fly completely upside down. Moreover, for the task of segmentation, we have to make sure that the mask is still correct: When applying a horizontal flip, the mask should be flipped as well.

Please do the following:

1. Implement a horizontal flip in the `__getitem__` function. (Why shouldn't you do it in the `load_AIRR_images` function?) Apply it with a probability of $p = 0.5$. **Does this improve the segmentation?** (Sanity check: it actually worsened our results with the BCE loss, ≈ 0.12. Data augmentation will mostly be beneficial for generalization to an unseen test set, given also longer training times.)

Copy your code for Parts 2 and 3 in weblab and submit it. Any training error ≤ 0.09 will be accepted. The following part is optional.

OPTIONAL - Part 4: Freezing weights.

Since the start of the lab session, we have been refining all the weights of a pretrained UNet architecture. However, it is also possible to *freeze* some of the weights, i.e., not to update them during training. Namely, we may expect that the low-level visual features in the earlier layers are common to many tasks, and that these features are already well learned. Hence, we may not want to update these weights during training. In particular, for many classification tasks, it may even be sufficient to freeze all the weights except for the last layer.

Please do the following:

1. Implement freezing of the weights until the last layer in the `train_U_Net` function. *Hint:* Freeze the weights after loading in the UNet model. Freezing parameters can be done with `param.requires_grad = False`. Layer parameters can be accessed with the help of a model's `children()` method. **Does this lead to good segmentation?**

2. What if you freeze the layers to about halfway? **Does this lead to good segmentation?** (It leads to a ≈ 0.09 BCE loss in our case.)

OPTIONAL - Part 5: Optimizing IoU performance and / or computational efficiency.

If you still have time and are passionate about the topic, try to optimize the IoU performance of a network on the dataset. You can do this by changing the training settings such as the number of epochs and the learning rate. But you can also use early stopping with a validation set, or change the network architecture, the loss function, the data augmentation, etc. An additional objective can be to use as small a network as possible, in order to promote computational efficiency for onboard the drone.