University of Pisa
Master of Science in Artificial Intelligence and Data Engineering

Semantic Segmentation of MRI scans

Matteo Del Seppia
Federico Minniti

Accademic Year 2022-2023

Summary

1. Introduction

Radiation oncologists try to deliver high doses of radiation using X-ray beams pointed to tumors while avoiding the stomach and intestines. With newer technology such as integrated magnetic resonance imaging and linear accelerator systems, also known as MR-Linacs, oncologists can visualize the daily position of the tumor and intestines, which can vary day to day. In these scans, radiation oncologists must manually outline the position of the stomach and intestines to adjust the direction of the x-ray beams to increase the dose delivery to the tumor and avoid the stomach and intestines. This is a time-consuming and labor-intensive process that can prolong treatments from 15 minutes a day to an hour a day, which can be difficult for patients to tolerate—unless deep learning could help automate the segmentation process. A method to segment the stomach and intestines would make treatments much faster and would allow more patients to get more effective treatment.

Two networks will be developed to address this problem:

- A newly designed network built and trained from scratch.
- A pre-trained network, fine-tuned on this dataset.

This project is based on the UW-Madison GI Tract Image Segmentation competition.

2. Problem statement

Our goal is to address a semantic segmentation task, considering MRI images that may contain up to three possible organs: small bowel, large bowel and stomach.
The network hence must learn to recognize the presence of these organs and create some segmentation masks to prevent them being targeted by X-ray beams.

2.1 Dataset Description

The dataset is available [on Kaggle](). It contains 16590 labeled images of MRI scans, with masks for stomach and/or large bowel and/or small bowel.
In our dataset, an image mask is a binary image where each pixel is either 0 or 1, representing the presence or absence of an organ. The masks are stored in the dataset using Run-Length Encoding, a simple compression algorithm that is often used in image processing and computer graphics. It works by encoding a sequence of identical values (in our case, pixels that are "on") as a single value and a count of how many times the value appears in the sequence.

2.2 Dataset Preprocessing

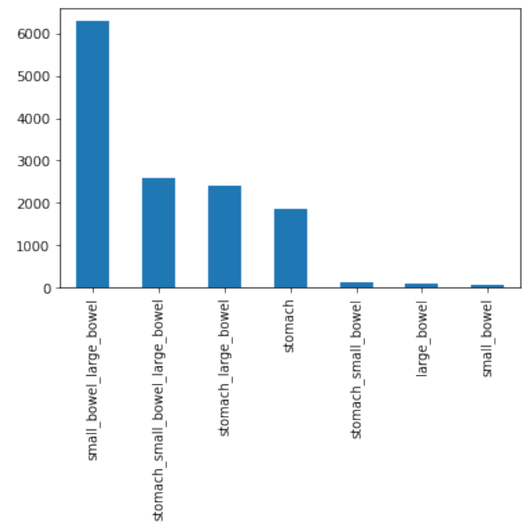
In the dataset, the same "id" or "case" could be found in multiple rows, one for each possible organ. Also, many cases didn't have any labels at all, because they served as a test set for the Kaggle competition, hence they have been discarded.

```
(115488, 3)
(None,                            id        class  segmentation
 0  case123_day20_slice_0001  large_bowel          NaN
 1  case123_day20_slice_0001  small_bowel          NaN
 2  case123_day20_slice_0001      stomach          NaN
 3  case123_day20_slice_0002  large_bowel          NaN
 4  case123_day20_slice_0002  small_bowel          NaN)
```

This was not ideal for our data loading process, and the dataset had to be heavily restructured. In our final dataset we have one row for each case having at least one label, and one column for each possible labeled organ. A new, additional column has been added to identify which labels were associated with each case, easing the process of checking data balancing.

	id	large_bowel	small_bowel	stomach	path	width	height	organ
0	case124_day20_slice_0081	28568 5 28833 7 29098 9 29363 10 29629 10 2989...	33943 7 34168 3 34208 11 34433 4 34473 13 3469...	28088 11 28353 14 28617 17 28882 19 29148 20 2...	uw-madison-gi-tract-image-segmentation/train/c...	266	266	stomach_small_bowel_large_bowel
1	case9_day0_slice_0086	22451 1 22804 10 22827 10 23163 12 23185 13 23...	27877 5 28236 7 28595 11 28955 11 29315 11 296...	NaN	uw-madison-gi-tract-image-segmentation/train/c...	360	310	small_bowel_large_bowel
2	case9_day22_slice_0077	20285 1 20626 10 20644 4 20985 12 21003 5 2103...	49066 3 49425 6 49784 9 50144 10 50504 11 5086...	NaN	uw-madison-gi-tract-image-segmentation/train/c...	360	310	small_bowel_large_bowel
3	case115_day0_slice_0098	11569 8 11831 15 11853 5 12094 21 12118 13 123...	17477 2 17720 2 17728 9 17741 6 17977 2 17983 ...	NaN	uw-madison-gi-tract-image-segmentation/train/c...	266	266	small_bowel_large_bowel
4	case54_day39_slice_0069	26030 4 26294 8 26560 9 26825 10 27091 10 2735...	NaN	17719 14 17983 18 18249 20 18514 23 18779 26 1...	uw-madison-gi-tract-image-segmentation/train/c...	266	266	stomach_large_bowel

The organs present in each case were not balanced, hence some kind of oversampling was needed during training. For the moment, the dataset was split into training, test, and validation with a proportion of 80%-10%-10% respectively, maintaining the class imbalance across the three splits.



2.3 Data Augmentation and Balancing

The problem of class imbalance on the training set was addressed using a Weighted Random Sampler. To each sample in the training set was assigned a weight, inversely proportional to the number of samples in its corresponding class. These weights determine the probability that each data point will be included in a batch during training. The higher the weight, the higher the probability that the data point will be included in the batch. By using a Weighted Random Sampler, we can ensure that the model sees a more balanced representation of the dataset during training, which can lead to better performance and prevent bias towards the majority classes.

Note that while it is true that the weights of a Weighted Random Sampler can be thought of as relative probabilities, they are not probabilities in the strict sense. The weights can be thought of as a measure of the "importance" of each data point.

```
[ ] classes = np.unique(y_train)
    class_sample_count = np.array([len(np.where(y_train == t)[0]) for t in np.unique(y_train)])
    class_weight = 1. / class_sample_count

    weights = []
    for i in range(len(y_train)):
      label = y_train.iloc[i]
      idx = np.where(classes == label)[0]
      weights.append(class_weight[idx][0])

    # weights for weighted random sampler
    weights[0:10]

    [0.0003856536829926726,
     0.00015867978419549348,
     0.00015867978419549348,
     0.00015867978419549348,
     0.00041425020712510354,
     0.00038565368299267226,
     0.00038565368299267226,
     0.00015867978419549348,
     0.00015867978419549348,
     0.00015867978419549348]
```

Furthermore, online data augmentation such as rotation, scaling, flipping has been used to randomly apply transformations to the input images and their corresponding masks. These transformations result in a larger and more diverse dataset, which can improve the model's ability to generalize to new data and can help prevent overfitting.

```python
data_transforms = {
    "train": A.Compose([
      A.CLAHE(clip_limit=2.0, always_apply=True),
      A.ShiftScaleRotate(shift_limit=0.03, rotate_limit=20, border_mode=cv2.BORDER_CONSTANT, value=0, p=0.85),
      A.OneOf([
                A.ElasticTransform(alpha=1, sigma=50, alpha_affine=10,
                                   border_mode=cv2.BORDER_CONSTANT, value=0, p=0.5),
                A.GridDistortion(num_steps=5, distort_limit=0.1,
                                 border_mode=cv2.BORDER_CONSTANT, value=0, p=0.5)
            ], p=0.2),

      A.OneOf([
                A.GaussNoise(var_limit=(0.0001, 0.004), p=0.7),
                A.Blur(blur_limit=3, p=0.3)
            ], p=0.5),

      A.Flip(p=0.5)
    ]),
    "test": A.Compose([A.CLAHE(clip_limit=2.0, always_apply=True)])
}
```

CLAHE (Contrast Limited Adaptive Histogram Equalization) is an image processing technique that is used to enhance the contrast of images by redistributing the pixel values. The technique is particularly useful for images with low contrast, like our MRI scans, where details may be difficult to discern.
CLAHE was used as a mandatory preprocessing step for both training and test images. Apart from the standard data augmentation techniques like flipping, rotating, and shifting, we also used grid distortions, elastic distortions, blurring and noising, as we have found they are commonly used in medical image segmentation tasks.

2.4 Training and Evaluation

In deep learning-based segmentation tasks, the choice of loss function plays a critical role in the performance of the model. A compound loss function that combines multiple individual loss functions can provide several benefits, such as improved performance and better convergence, over a single loss function.
After some trials, we found that a good loss function for our problem was a compound loss, mixing Dice Loss and binary cross entropy:

```python
dice_loss = smp.losses.DiceLoss(mode='multilabel')

bce_loss = smp.losses.SoftBCEWithLogitsLoss()

def compound_loss_function(y_pred, groundtruth):
  return 0.4*dice_loss(y_pred, groundtruth) + 0.6*bce_loss(y_pred, groundtruth)
```

The Dice loss function is based on the Dice coefficient and is popular in medical image segmentation tasks, as it can handle imbalance between foreground and background pixels better than other loss functions. On the other hand, it is more difficult to optimize due its unstable gradient. Therefore, it is often combined with other, more smooth loss functions like (binary) cross entropy.
To evaluate the performance of our model on validation/test set we used the Dice coefficient, which is positively correlated with the well-known Jaccard.
Moreover, the Dice coefficient is defined as the ratio of the intersection of two sets A and B to their average size: Dice (A, B) = 2 * $|A \cap B| / (|A| + |B|)$, where in our case A is the predicted area and B is the target area.
The Dice coefficient is commonly used in medical image segmentation tasks, such as segmenting tumors, lesions, and organs, as well as in other computer vision applications. It is a simple and intuitive measure that provides a quantitative assessment of the performance of the segmentation model.

```python
def dice_coefficient(groundtruth, y_pred, thr=0.5, epsilon=0.001):
    groundtruth = groundtruth.to(torch.float32)

    # round elements of y_pred
    y_pred = (y_pred>thr).to(torch.float32)

    # dimensions (NxCxWxH)
    # N: images' index in a batch
    # C: depth (channel)
    # W: width
    # H: height

    # (groundtruth AND prediction) of each layer of the mask torch tensor
    # cardinality of the intersection
    intersection = (groundtruth*y_pred).sum(dim=(2,3))
    denominator = groundtruth.sum(dim=(2,3)) + y_pred.sum(dim=(2,3))
    dice = ((2*intersection+epsilon)/(denominator+epsilon)).mean(dim=(1,0))
    return dice
```

In all our trainings, early stopping was implemented simply by saving on disk the parameters of the best model after each epoch, judged by looking at the mean Dice on validation. We generally found that convergence was fast and required no more than 15-25 epochs, probably due to the small size of our training set. The convergence of the pretrained model was even faster in terms of epochs.

3. Models

3.1 Models built from scratch

In order to create our network we developed an encoder-decoder, which is the standard architecture for semantic segmentation tasks.

The encoder takes the input image and processes it through a series of convolutional layers; these layers gradually reduce the resolution of the image, increasing the number of output channels and extracting more and more low-level details. In addition to the convolutional layers, we also used pooling layers to reduce the size of the input image.

The decoder receives the low-resolution representation of the image from the encoder and processes it through a series of deconvolutional (up sampling) layers to obtain the segmented output image. To do that, these deconvolutional layers gradually increase the resolution of the image, reduce the number of output channels, and combine information from the encoder and the decoder to obtain a segmentation map of the input image.

Initially, for the first three versions, which were not too complex, we considered the following data augmentations:

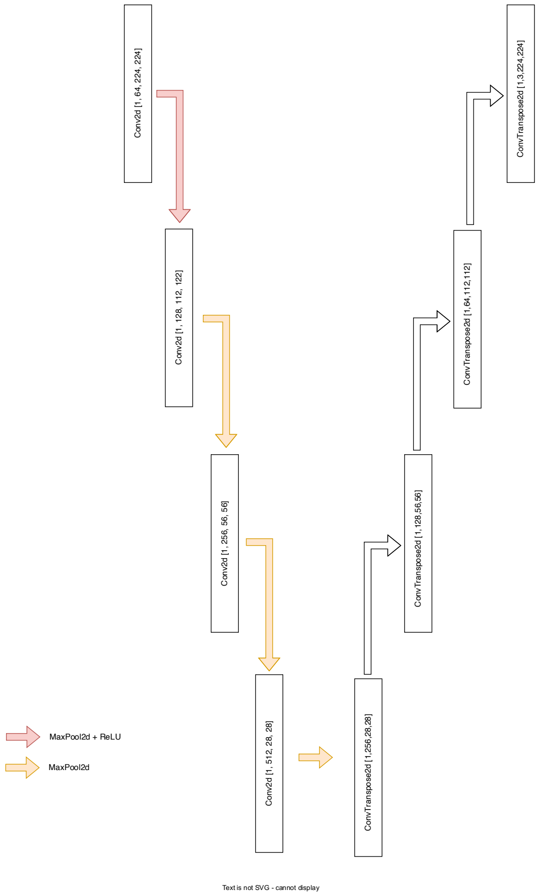
```
data_transforms = {
    "train": A.Compose([
      A.CLAHE(clip_limit=2.0, always_apply=True),
      A.ShiftScaleRotate(shift_limit=0.03, rotate_limit=20, border_mode=cv2.BORDER_CONSTANT, value=0, p=0.85),
      A.OneOf([
                A.GaussNoise(var_limit=(0.0001, 0.004), p=0.7),
                A.Blur(blur_limit=3, p=0.3)
            ], p=0.5),

      A.Flip(p=0.5)
    ]),
    "test": A.Compose([A.CLAHE(clip_limit=2.0, always_apply=True)])
}
```

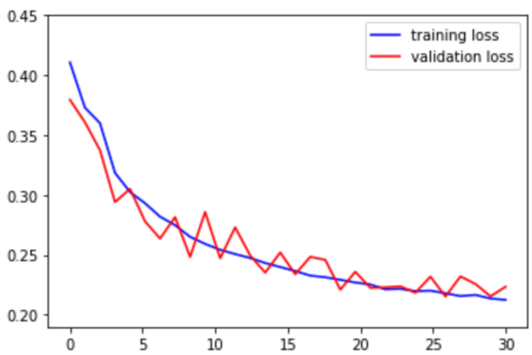
Afterwards we considered different approaches, modifying existing layers and adding new layers, to enhance the capabilities of our final model from scratch.

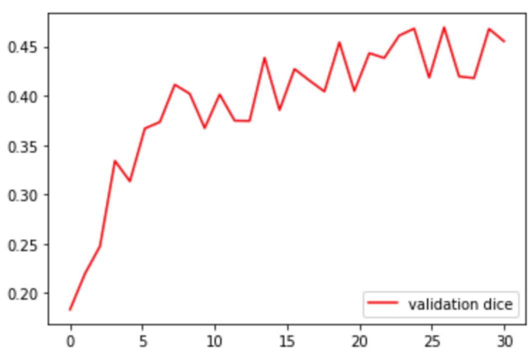
3.1.1 Simple CNN

We first developed a basic model with an encoder composed of convolutional layers and max pooling layers; and a decoder with only deconvolutional layers

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1         [-1, 64, 224, 224]           9,472
         MaxPool2d-2         [-1, 64, 112, 112]               0
              ReLU-3         [-1, 64, 112, 112]               0
            Conv2d-4        [-1, 128, 112, 112]          73,856
         MaxPool2d-5          [-1, 128, 56, 56]               0
            Conv2d-6          [-1, 256, 56, 56]         295,168
         MaxPool2d-7          [-1, 256, 28, 28]               0
            Conv2d-8          [-1, 512, 28, 28]       1,180,160
         MaxPool2d-9          [-1, 512, 14, 14]               0
           Encoder-10          [-1, 512, 14, 14]               0
  ConvTranspose2d-11          [-1, 256, 28, 28]         524,544
  ConvTranspose2d-12          [-1, 128, 56, 56]         131,200
  ConvTranspose2d-13         [-1, 64, 112, 112]          32,832
  ConvTranspose2d-14          [-1, 3, 224, 224]             771
           Decoder-15          [-1, 3, 224, 224]               0
================================================================
Total params: 2,248,003
Trainable params: 2,248,003
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.57
Forward/backward pass size (MB): 77.33
Params size (MB): 8.58
Estimated Total Size (MB): 86.48
----------------------------------------------------------------
```



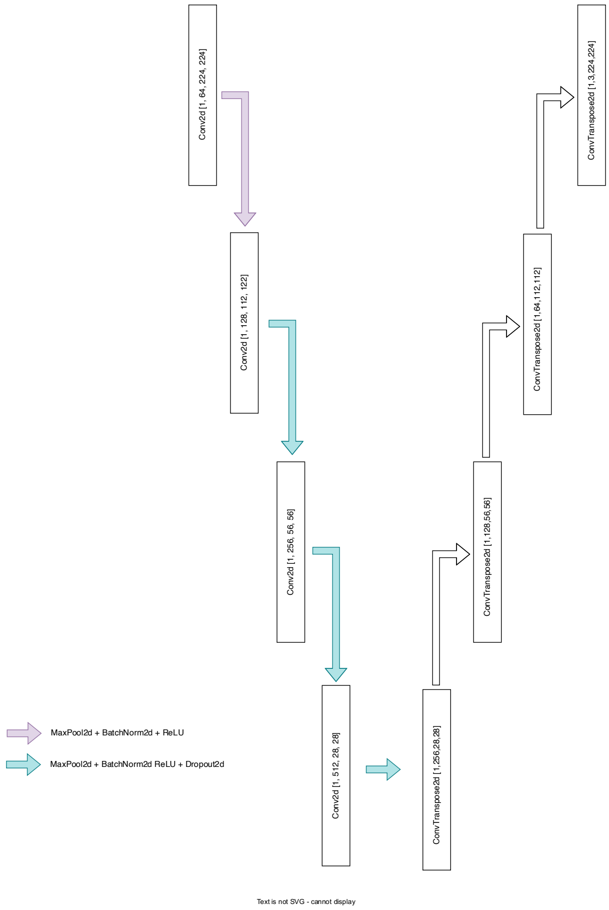


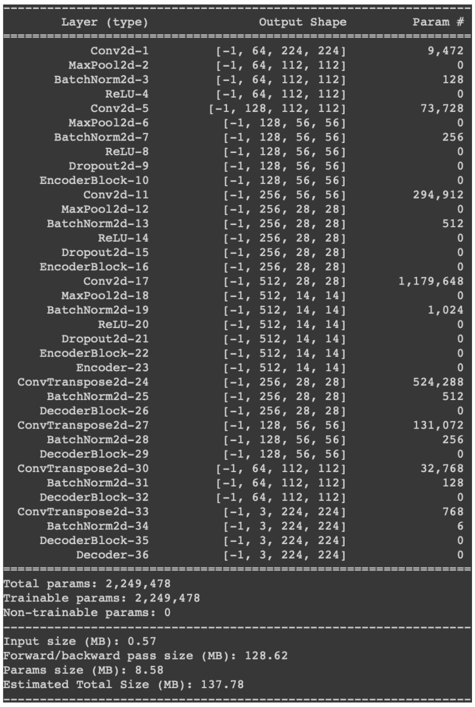


The best model obtains a mean Dice of 0.4677 on the validation set.

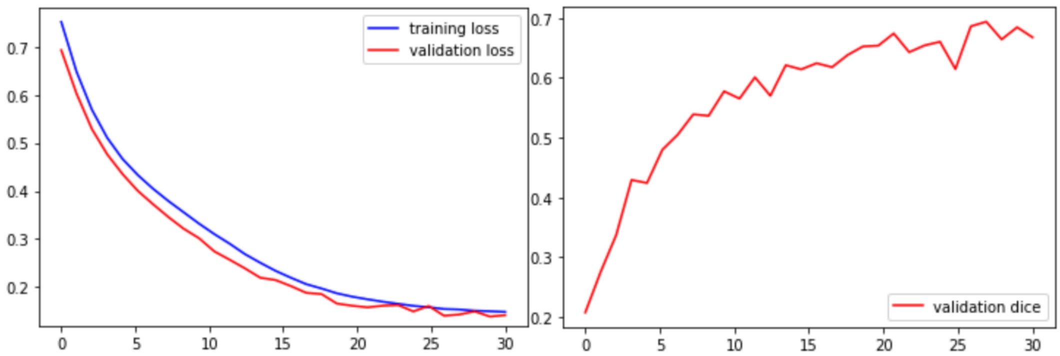
3.1.2 SimpleCNN with dropout and batch normalization

In order to enhance generalization and gradients stability we introduced dropout layers after the activations with $p = 0.1$ and batch normalizations after each convolutional layer.





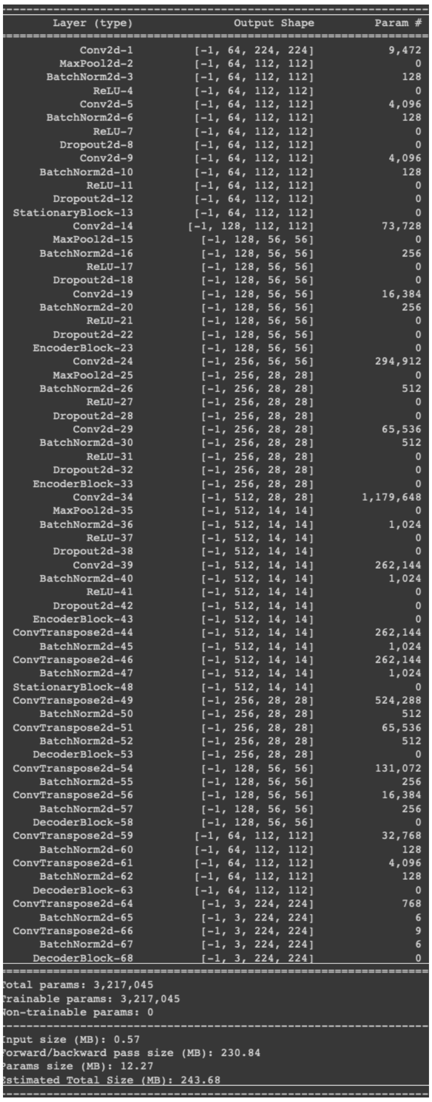
```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1         [-1, 64, 224, 224]           9,472
         MaxPool2d-2         [-1, 64, 112, 112]               0
       BatchNorm2d-3         [-1, 64, 112, 112]             128
              ReLU-4         [-1, 64, 112, 112]               0
            Conv2d-5        [-1, 128, 112, 112]          73,728
         MaxPool2d-6          [-1, 128, 56, 56]               0
       BatchNorm2d-7          [-1, 128, 56, 56]             256
              ReLU-8          [-1, 128, 56, 56]               0
         Dropout2d-9          [-1, 128, 56, 56]               0
     EncoderBlock-10          [-1, 128, 56, 56]               0
           Conv2d-11          [-1, 256, 56, 56]         294,912
        MaxPool2d-12          [-1, 256, 28, 28]               0
      BatchNorm2d-13          [-1, 256, 28, 28]             512
             ReLU-14          [-1, 256, 28, 28]               0
        Dropout2d-15          [-1, 256, 28, 28]               0
     EncoderBlock-16          [-1, 256, 28, 28]               0
           Conv2d-17          [-1, 512, 28, 28]       1,179,648
        MaxPool2d-18          [-1, 512, 14, 14]               0
      BatchNorm2d-19          [-1, 512, 14, 14]           1,024
             ReLU-20          [-1, 512, 14, 14]               0
        Dropout2d-21          [-1, 512, 14, 14]               0
     EncoderBlock-22          [-1, 512, 14, 14]               0
          Encoder-23          [-1, 512, 14, 14]               0
  ConvTranspose2d-24          [-1, 256, 28, 28]         524,288
      BatchNorm2d-25          [-1, 256, 28, 28]             512
     DecoderBlock-26          [-1, 256, 28, 28]               0
  ConvTranspose2d-27          [-1, 128, 56, 56]         131,072
      BatchNorm2d-28          [-1, 128, 56, 56]             256
     DecoderBlock-29          [-1, 128, 56, 56]               0
  ConvTranspose2d-30         [-1, 64, 112, 112]          32,768
      BatchNorm2d-31         [-1, 64, 112, 112]             128
     DecoderBlock-32         [-1, 64, 112, 112]               0
  ConvTranspose2d-33          [-1, 3, 224, 224]             768
      BatchNorm2d-34          [-1, 3, 224, 224]               6
     DecoderBlock-35          [-1, 3, 224, 224]               0
          Decoder-36          [-1, 3, 224, 224]               0
================================================================
Total params: 2,249,478
Trainable params: 2,249,478
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.57
Forward/backward pass size (MB): 128.62
Params size (MB): 8.58
Estimated Total Size (MB): 137.78
----------------------------------------------------------------
```



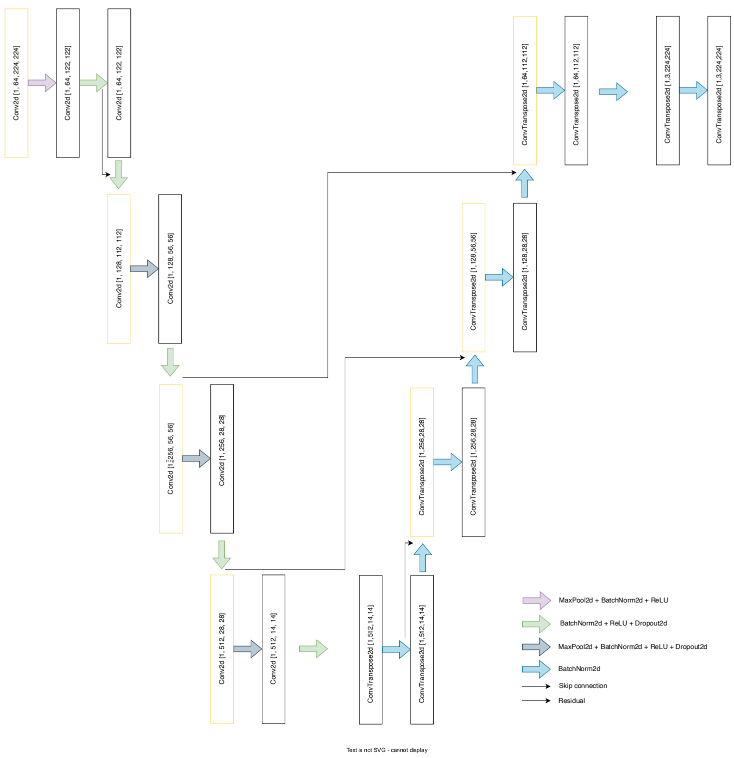
The best model obtains a mean Dice of 0.6940 on the validation set.

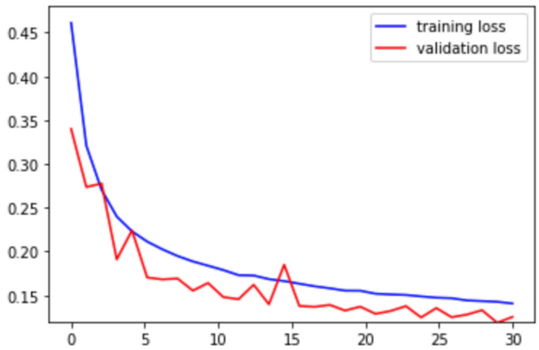
3.1.3 Adding residuals and skip connections

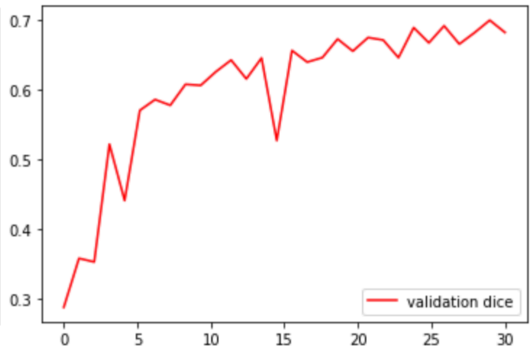
In the third version we have added more convolutional/deconvolutional layers to increase the model complexity. We also add some residuals and skip connections, which connect some layers of the encoder to those of the decoder, as well as some internal layers of the encoder. This allows low-level information from the original image to flow through the network to combine with high-level information processed by the convolutional layers. This should help with the generalization accuracy and the stability of gradients.



```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1         [-1, 64, 224, 224]           9,472
         MaxPool2d-2         [-1, 64, 112, 112]               0
       BatchNorm2d-3         [-1, 64, 112, 112]             128
              ReLU-4         [-1, 64, 112, 112]               0
            Conv2d-5         [-1, 64, 112, 112]           4,096
       BatchNorm2d-6         [-1, 64, 112, 112]             128
              ReLU-7         [-1, 64, 112, 112]               0
         Dropout2d-8         [-1, 64, 112, 112]               0
            Conv2d-9         [-1, 64, 112, 112]           4,096
      BatchNorm2d-10         [-1, 64, 112, 112]             128
             ReLU-11         [-1, 64, 112, 112]               0
        Dropout2d-12         [-1, 64, 112, 112]               0
 StationaryBlock-13          [-1, 64, 112, 112]               0
           Conv2d-14        [-1, 128, 112, 112]          73,728
        MaxPool2d-15          [-1, 128, 56, 56]               0
      BatchNorm2d-16          [-1, 128, 56, 56]             256
             ReLU-17          [-1, 128, 56, 56]               0
        Dropout2d-18          [-1, 128, 56, 56]               0
           Conv2d-19          [-1, 128, 56, 56]          16,384
      BatchNorm2d-20          [-1, 128, 56, 56]             256
             ReLU-21          [-1, 128, 56, 56]               0
        Dropout2d-22          [-1, 128, 56, 56]               0
     EncoderBlock-23          [-1, 128, 56, 56]               0
           Conv2d-24          [-1, 256, 56, 56]         294,912
        MaxPool2d-25          [-1, 256, 28, 28]               0
      BatchNorm2d-26          [-1, 256, 28, 28]             512
             ReLU-27          [-1, 256, 28, 28]               0
        Dropout2d-28          [-1, 256, 28, 28]               0
           Conv2d-29          [-1, 256, 28, 28]          65,536
      BatchNorm2d-30          [-1, 256, 28, 28]             512
             ReLU-31          [-1, 256, 28, 28]               0
        Dropout2d-32          [-1, 256, 28, 28]               0
     EncoderBlock-33          [-1, 256, 28, 28]               0
           Conv2d-34          [-1, 512, 28, 28]       1,179,648
        MaxPool2d-35          [-1, 512, 14, 14]               0
      BatchNorm2d-36          [-1, 512, 14, 14]           1,024
             ReLU-37          [-1, 512, 14, 14]               0
        Dropout2d-38          [-1, 512, 14, 14]               0
           Conv2d-39          [-1, 512, 14, 14]         262,144
      BatchNorm2d-40          [-1, 512, 14, 14]           1,024
             ReLU-41          [-1, 512, 14, 14]               0
        Dropout2d-42          [-1, 512, 14, 14]               0
     EncoderBlock-43          [-1, 512, 14, 14]               0
 ConvTranspose2d-44           [-1, 512, 14, 14]         262,144
      BatchNorm2d-45          [-1, 512, 14, 14]           1,024
 ConvTranspose2d-46           [-1, 512, 14, 14]         262,144
      BatchNorm2d-47          [-1, 512, 14, 14]           1,024
 StationaryBlock-48           [-1, 512, 14, 14]               0
 ConvTranspose2d-49           [-1, 256, 28, 28]         524,288
      BatchNorm2d-50          [-1, 256, 28, 28]             512
 ConvTranspose2d-51           [-1, 256, 28, 28]          65,536
      BatchNorm2d-52          [-1, 256, 28, 28]             512
     DecoderBlock-53          [-1, 256, 28, 28]               0
 ConvTranspose2d-54           [-1, 128, 56, 56]         131,072
      BatchNorm2d-55          [-1, 128, 56, 56]             256
 ConvTranspose2d-56           [-1, 128, 56, 56]          16,384
      BatchNorm2d-57          [-1, 128, 56, 56]             256
     DecoderBlock-58          [-1, 128, 56, 56]               0
 ConvTranspose2d-59         [-1, 64, 112, 112]          32,768
      BatchNorm2d-60         [-1, 64, 112, 112]             128
 ConvTranspose2d-61         [-1, 64, 112, 112]           4,096
      BatchNorm2d-62         [-1, 64, 112, 112]             128
     DecoderBlock-63         [-1, 64, 112, 112]               0
 ConvTranspose2d-64          [-1, 3, 224, 224]             768
      BatchNorm2d-65          [-1, 3, 224, 224]               6
 ConvTranspose2d-66          [-1, 3, 224, 224]               9
      BatchNorm2d-67          [-1, 3, 224, 224]               6
     DecoderBlock-68          [-1, 3, 224, 224]               0
================================================================
Total params: 3,217,045
Trainable params: 3,217,045
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.57
Forward/backward pass size (MB): 230.84
Params size (MB): 12.27
Estimated Total Size (MB): 243.68
----------------------------------------------------------------
```







The best model obtained a mean Dice of 0.6995 on the validation set.

3.1.4 Final CNN from scratch

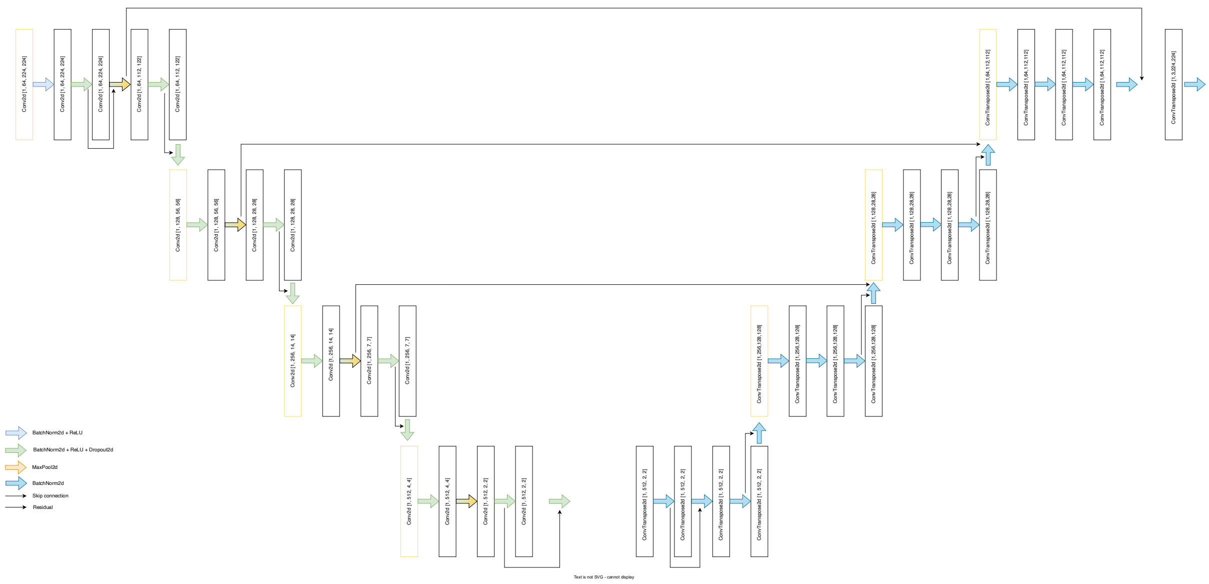
Finally, we added additional convolutional layers to try to increase the network's capacity to learn complex information from the input. To improve generalization and counter possible overfitting due to the relatively large size of the model, we added elastic and grid distortions among the possible augmentations (as described in the section 2.2).

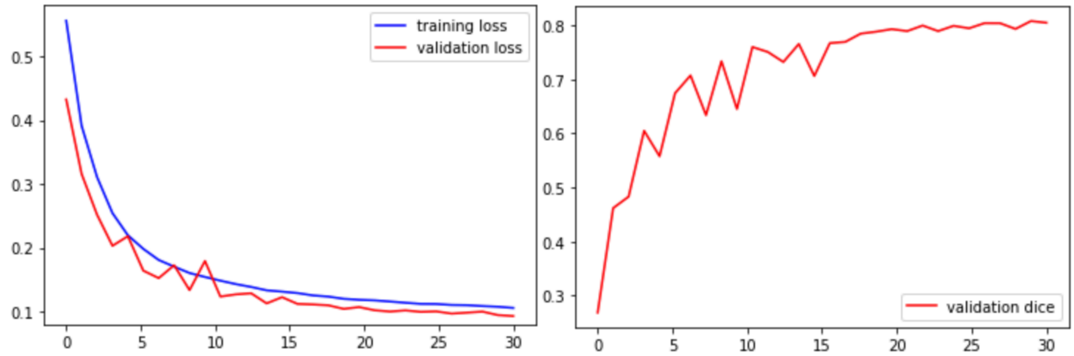
Encoder:

Layer (type)	Output Shape	Param #
Conv2d-1	[-1, 64, 224, 224]	9,408
BatchNorm2d-2	[-1, 64, 224, 224]	128
ReLU-3	[-1, 64, 224, 224]	0
Conv2d-4	[-1, 64, 224, 224]	36,864
BatchNorm2d-5	[-1, 64, 224, 224]	128
ReLU-6	[-1, 64, 224, 224]	0
Dropout2d-7	[-1, 64, 224, 224]	0
Conv2d-8	[-1, 64, 224, 224]	36,864
BatchNorm2d-9	[-1, 64, 224, 224]	128
ReLU-10	[-1, 64, 224, 224]	0
Dropout2d-11	[-1, 64, 224, 224]	0
EncoderBlock-12	[-1, 64, 224, 224]	0
MaxPool2d-13	[-1, 64, 112, 112]	0
Conv2d-14	[-1, 64, 112, 112]	36,864
BatchNorm2d-15	[-1, 64, 112, 112]	128
ReLU-16	[-1, 64, 112, 112]	0
Dropout2d-17	[-1, 64, 112, 112]	0
Conv2d-18	[-1, 64, 112, 112]	36,864
BatchNorm2d-19	[-1, 64, 112, 112]	128
ReLU-20	[-1, 64, 112, 112]	0
Dropout2d-21	[-1, 64, 112, 112]	0
EncoderBlock-22	[-1, 64, 112, 112]	0
Conv2d-23	[-1, 128, 56, 56]	73,728
BatchNorm2d-24	[-1, 128, 56, 56]	256
ReLU-25	[-1, 128, 56, 56]	0
Dropout2d-26	[-1, 128, 56, 56]	0
Conv2d-27	[-1, 128, 56, 56]	147,456
BatchNorm2d-28	[-1, 128, 56, 56]	256
ReLU-29	[-1, 128, 56, 56]	0
Dropout2d-30	[-1, 128, 56, 56]	0
EncoderBlock-31	[-1, 128, 56, 56]	0
MaxPool2d-32	[-1, 128, 28, 28]	0
Conv2d-33	[-1, 128, 28, 28]	147,456
BatchNorm2d-34	[-1, 128, 28, 28]	256
ReLU-35	[-1, 128, 28, 28]	0
Dropout2d-36	[-1, 128, 28, 28]	0
Conv2d-37	[-1, 128, 28, 28]	147,456
BatchNorm2d-38	[-1, 128, 28, 28]	256
ReLU-39	[-1, 128, 28, 28]	0
Dropout2d-40	[-1, 128, 28, 28]	0
EncoderBlock-41	[-1, 128, 28, 28]	0
Conv2d-42	[-1, 256, 14, 14]	294,912
BatchNorm2d-43	[-1, 256, 14, 14]	512
ReLU-44	[-1, 256, 14, 14]	0
Dropout2d-45	[-1, 256, 14, 14]	0
Conv2d-46	[-1, 256, 14, 14]	589,824
BatchNorm2d-47	[-1, 256, 14, 14]	512
ReLU-48	[-1, 256, 14, 14]	0
Dropout2d-49	[-1, 256, 14, 14]	0
EncoderBlock-50	[-1, 256, 14, 14]	0
MaxPool2d-51	[-1, 256, 7, 7]	0
Conv2d-52	[-1, 256, 7, 7]	589,824
BatchNorm2d-53	[-1, 256, 7, 7]	512
ReLU-54	[-1, 256, 7, 7]	0
Dropout2d-55	[-1, 256, 7, 7]	0
Conv2d-56	[-1, 256, 7, 7]	589,824
BatchNorm2d-57	[-1, 256, 7, 7]	512
ReLU-58	[-1, 256, 7, 7]	0
Dropout2d-59	[-1, 256, 7, 7]	0
EncoderBlock-60	[-1, 256, 7, 7]	0
Conv2d-61	[-1, 512, 4, 4]	1,179,648
BatchNorm2d-62	[-1, 512, 4, 4]	1,024
ReLU-63	[-1, 512, 4, 4]	0
Dropout2d-64	[-1, 512, 4, 4]	0
Conv2d-65	[-1, 512, 4, 4]	2,359,296
BatchNorm2d-66	[-1, 512, 4, 4]	1,024
ReLU-67	[-1, 512, 4, 4]	0
Dropout2d-68	[-1, 512, 4, 4]	0
EncoderBlock-69	[-1, 512, 4, 4]	0
MaxPool2d-70	[-1, 512, 2, 2]	0
Conv2d-71	[-1, 512, 2, 2]	2,359,296
BatchNorm2d-72	[-1, 512, 2, 2]	1,024
ReLU-73	[-1, 512, 2, 2]	0
Dropout2d-74	[-1, 512, 2, 2]	0
Conv2d-75	[-1, 512, 2, 2]	2,359,296
BatchNorm2d-76	[-1, 512, 2, 2]	1,024
ReLU-77	[-1, 512, 2, 2]	0
Dropout2d-78	[-1, 512, 2, 2]	0
EncoderBlock-79	[-1, 512, 2, 2]	0

Decoder:

```
  ConvTranspose2d-80            [-1, 512, 2, 2]       2,359,296
       BatchNorm2d-81            [-1, 512, 2, 2]           1,024
  ConvTranspose2d-82            [-1, 512, 2, 2]       2,359,296
       BatchNorm2d-83            [-1, 512, 2, 2]           1,024
      DecoderBlock-84            [-1, 512, 2, 2]               0
  ConvTranspose2d-85            [-1, 512, 2, 2]       2,359,296
       BatchNorm2d-86            [-1, 512, 2, 2]           1,024
  ConvTranspose2d-87            [-1, 512, 2, 2]       2,359,296
       BatchNorm2d-88            [-1, 512, 2, 2]           1,024
      DecoderBlock-89            [-1, 512, 2, 2]               0
  ConvTranspose2d-90            [-1, 256, 7, 7]       1,179,648
       BatchNorm2d-91            [-1, 256, 7, 7]             512
  ConvTranspose2d-92            [-1, 256, 7, 7]         589,824
       BatchNorm2d-93            [-1, 256, 7, 7]             512
      DecoderBlock-94            [-1, 256, 7, 7]               0
  ConvTranspose2d-95            [-1, 256, 7, 7]         589,824
       BatchNorm2d-96            [-1, 256, 7, 7]             512
  ConvTranspose2d-97            [-1, 256, 7, 7]         589,824
       BatchNorm2d-98            [-1, 256, 7, 7]             512
      DecoderBlock-99            [-1, 256, 7, 7]               0
 ConvTranspose2d-100          [-1, 128, 28, 28]       1,179,648
      BatchNorm2d-101          [-1, 128, 28, 28]             256
 ConvTranspose2d-102          [-1, 128, 28, 28]         147,456
      BatchNorm2d-103          [-1, 128, 28, 28]             256
     DecoderBlock-104          [-1, 128, 28, 28]               0
 ConvTranspose2d-105          [-1, 128, 28, 28]         147,456
      BatchNorm2d-106          [-1, 128, 28, 28]             256
 ConvTranspose2d-107          [-1, 128, 28, 28]         147,456
      BatchNorm2d-108          [-1, 128, 28, 28]             256
     DecoderBlock-109          [-1, 128, 28, 28]               0
 ConvTranspose2d-110         [-1, 64, 112, 112]         294,912
      BatchNorm2d-111         [-1, 64, 112, 112]             128
 ConvTranspose2d-112         [-1, 64, 112, 112]          36,864
      BatchNorm2d-113         [-1, 64, 112, 112]             128
     DecoderBlock-114         [-1, 64, 112, 112]               0
 ConvTranspose2d-115         [-1, 64, 112, 112]          36,864
      BatchNorm2d-116         [-1, 64, 112, 112]             128
 ConvTranspose2d-117         [-1, 64, 112, 112]          36,864
      BatchNorm2d-118         [-1, 64, 112, 112]             128
     DecoderBlock-119         [-1, 64, 112, 112]               0
 ConvTranspose2d-120          [-1, 3, 224, 224]          12,288
      BatchNorm2d-121          [-1, 3, 224, 224]               6
================================================================
Total params: 25,436,486
Trainable params: 25,436,486
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.57
Forward/backward pass size (MB): 467.91
Params size (MB): 97.03
Estimated Total Size (MB): 565.51
----------------------------------------------------------------
```



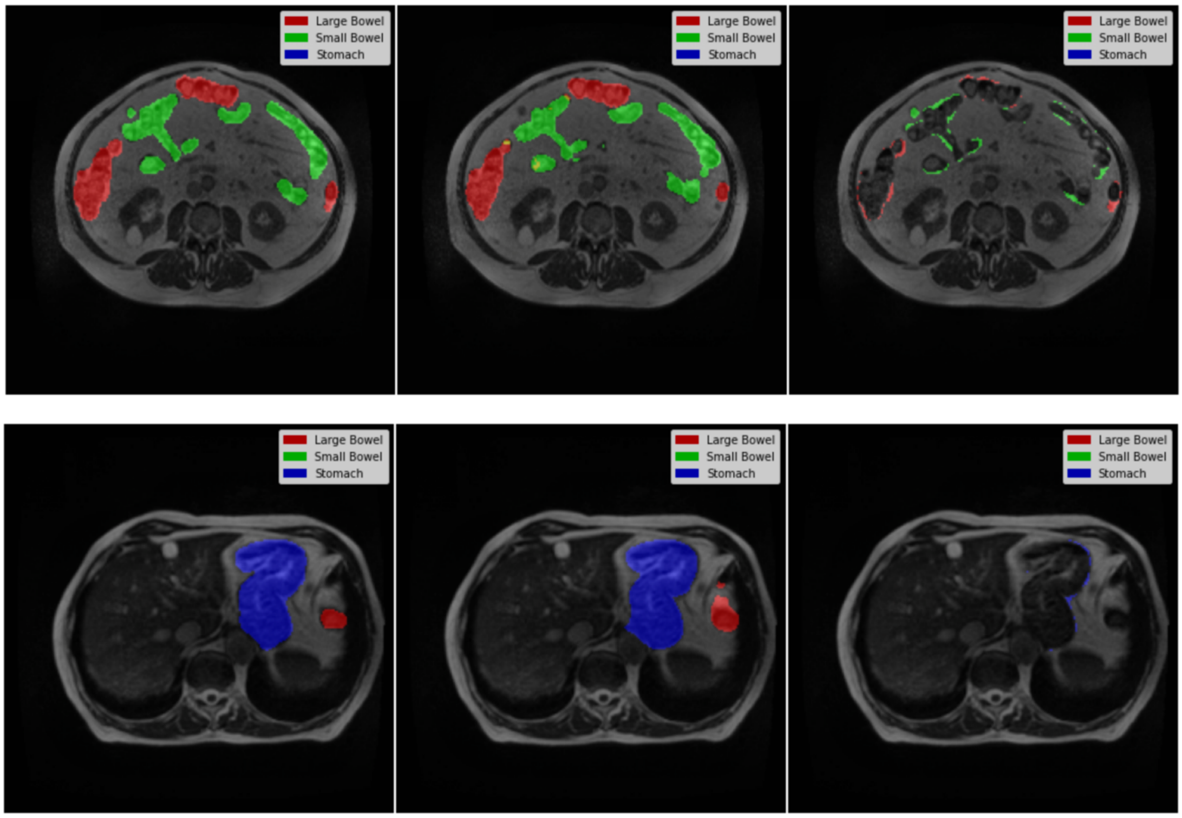


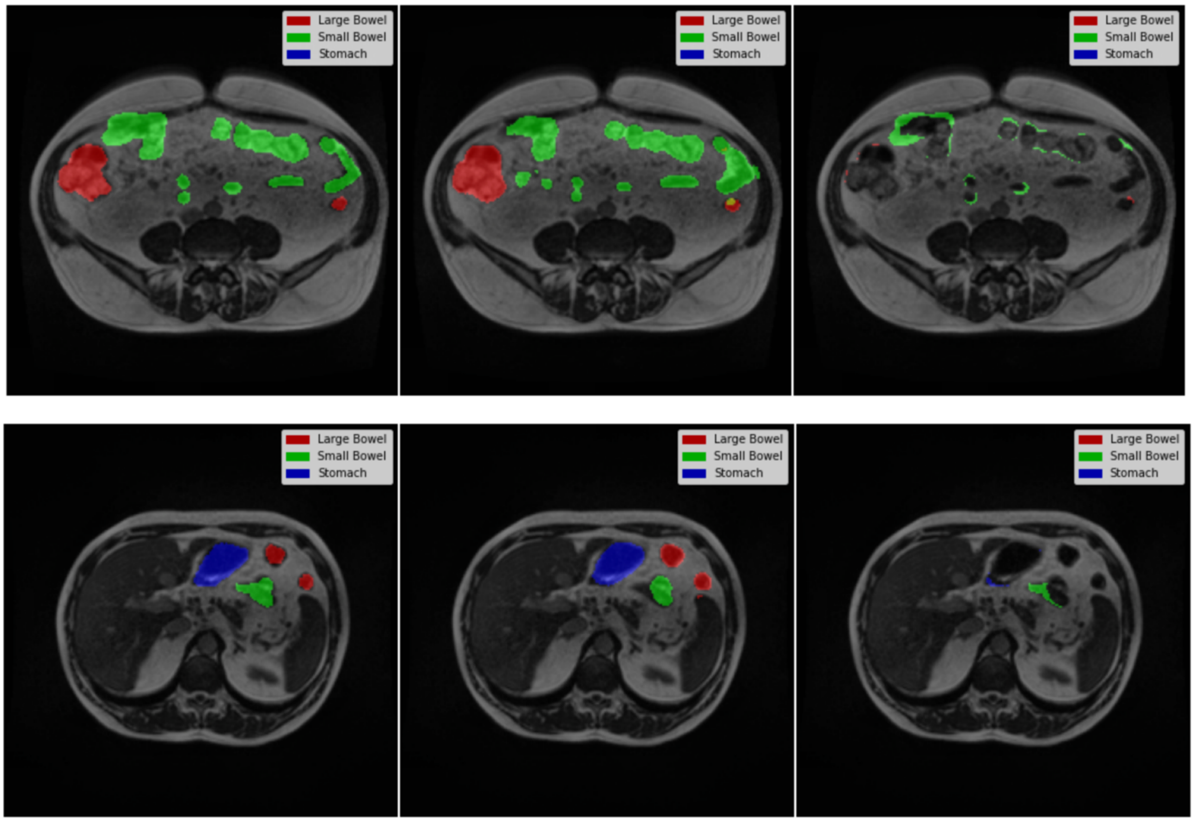
The best model after the fine-tuning obtained a mean Dice of 0.8054 on the test set.

The final model has been trained with the following parameters:

- Max epochs: 30
- Batch size: 16
- RMSprop optimizer with initial learning rate=5e-4 and momentum=0.9
- CosineAnnealingLR scheduler with *T_max*=50 and *eta_min*=1e-5

Some predictions obtained using the final model (left is real mask, middle is prediction, right is the difference):

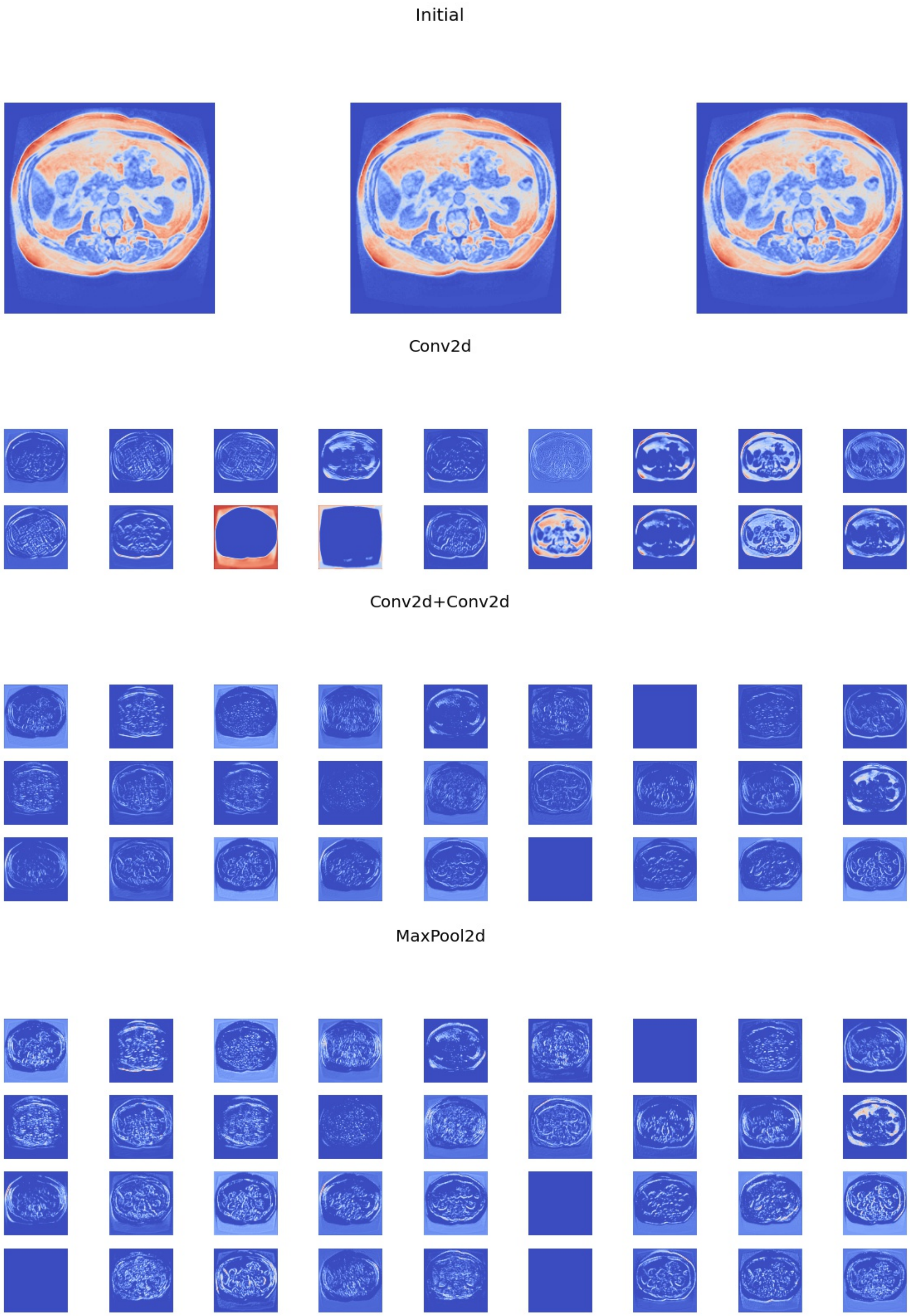




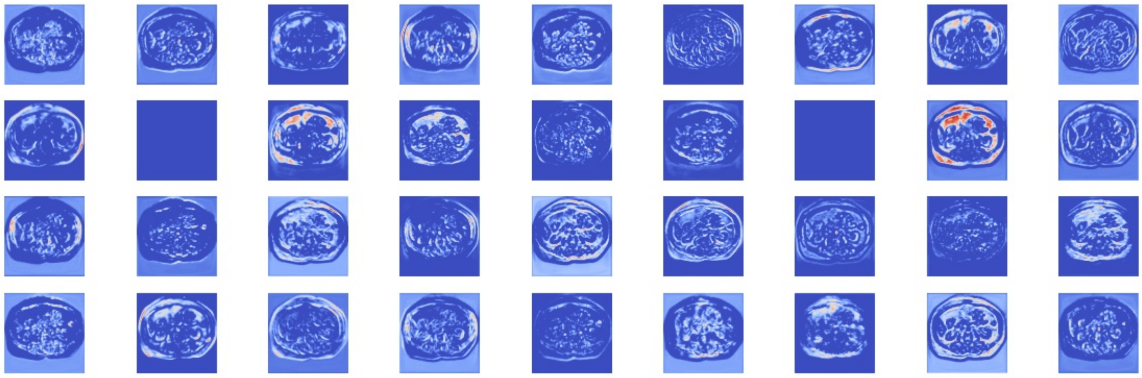
3.1.4.1 Activation maps

By visualizing the activation maps generated by the CNN, we can get an idea of what kind of features the network is looking for in the input image to make segmentation predictions. This can help us understand how the network is interpreting the image and what features are important for different semantic classes.

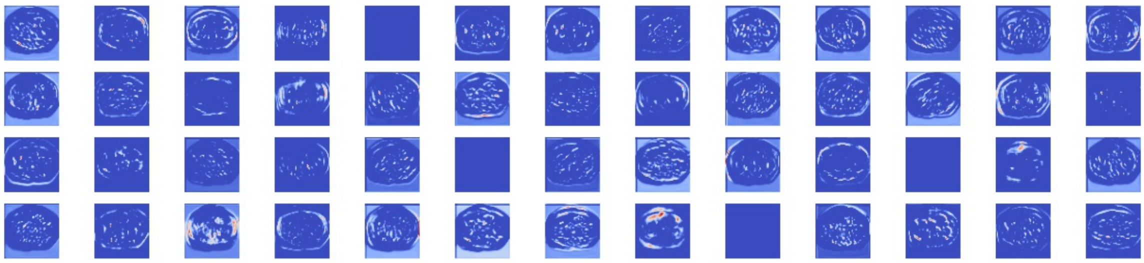
The following activation maps are not complete (they have been cropped for readability of the documentation).


Initial
Conv2d
Conv2d+Conv2d
MaxPool2d

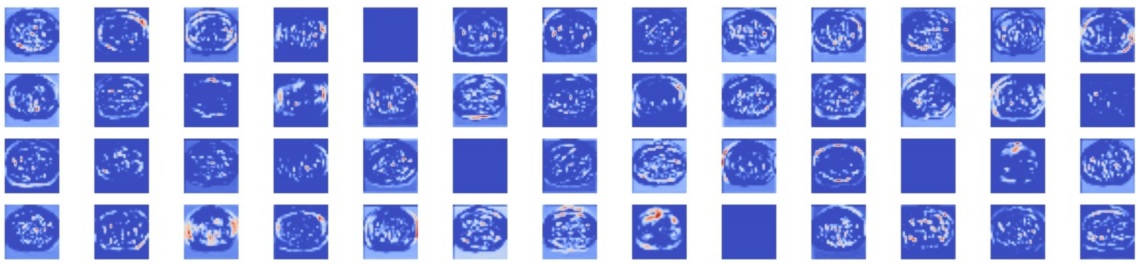
Conv2d+Conv2d



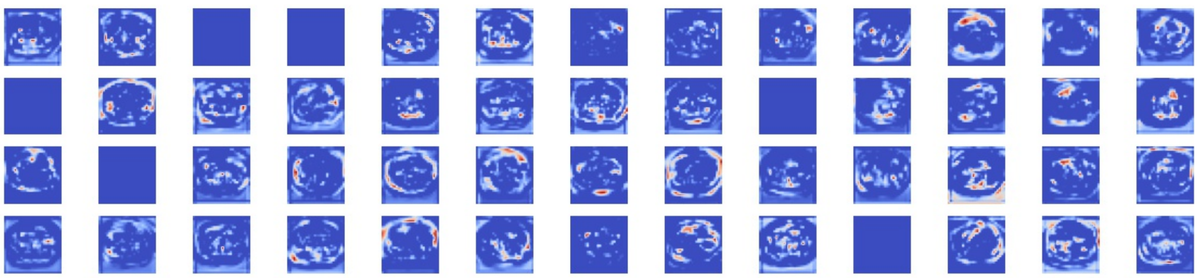
Conv2d+Conv2d



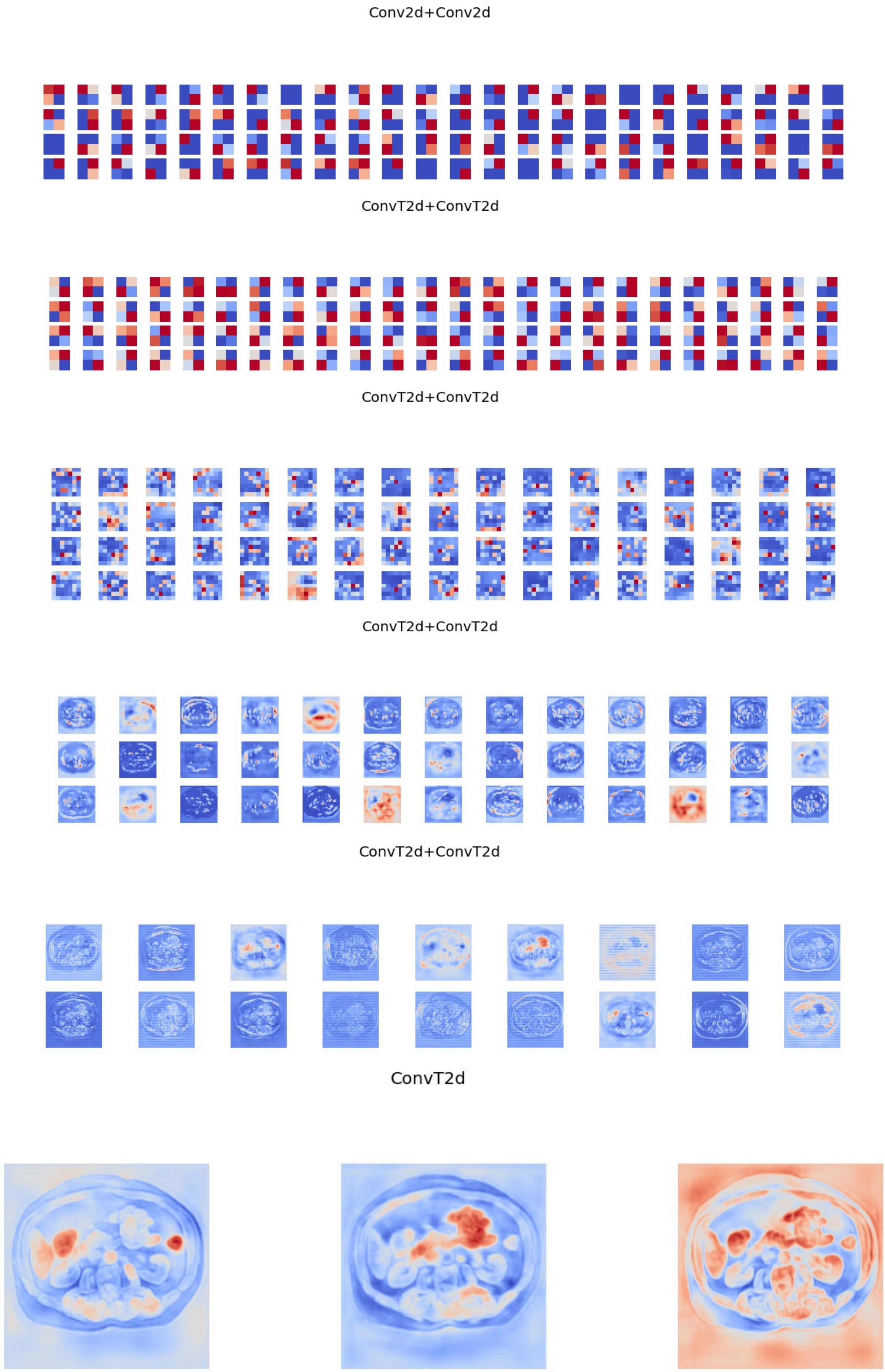
MaxPool2d



Conv2d+Conv2d



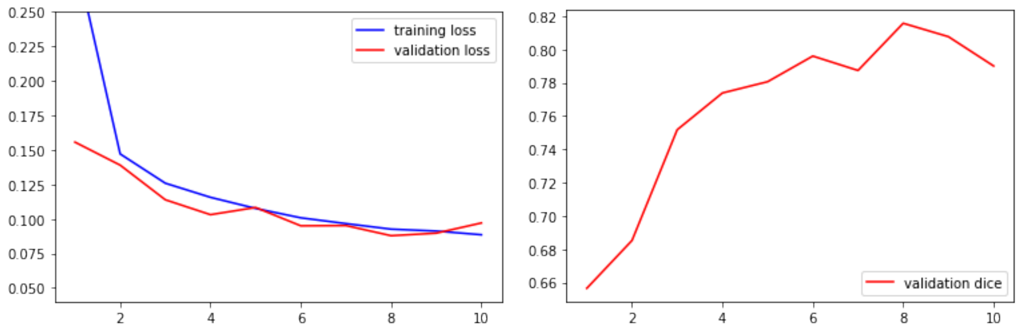

Conv2d+Conv2d
MaxPool2d
Conv2d+Conv2d
Conv2d+Conv2d
MaxPool2d


Conv2d+Conv2d
ConvT2d+ConvT2d
ConvT2d+ConvT2d
ConvT2d+ConvT2d
ConvT2d+ConvT2d
ConvT2d

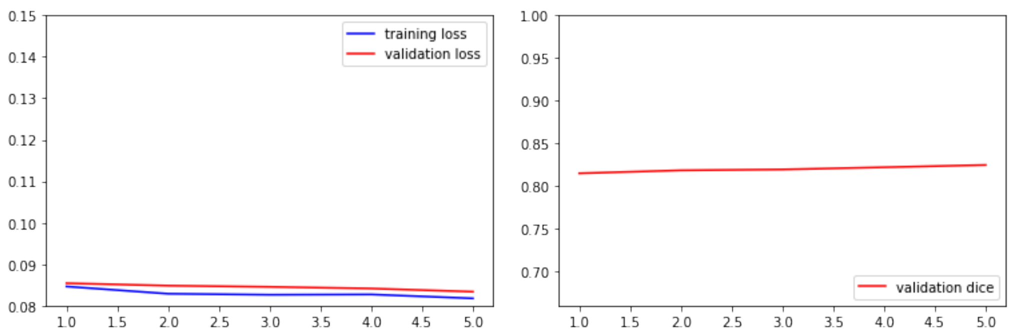
3.2 Transfer learning

For the fine-tuning of the pre-trained model, we selected a one of the many models available in the *segmentation-models-pytorch* library. A Unet++ architecture incorporating an EfficientNetB3 encoder pretrained on ImageNet has been chosen for fine-tuning which has a total of 10.6 million parameters for the encoder and 3 million parameters for the decoder.

The encoder has been initially frozen, and the decoder layers were trained on our dataset for 10 epochs. This first part involved an RMSprop optimizer with starting learning rate of 5e-5 and momentum set to 0.9. The scheduler was CosineAnnealingLR, with *T_max=50* and *eta_min=1e-6.*

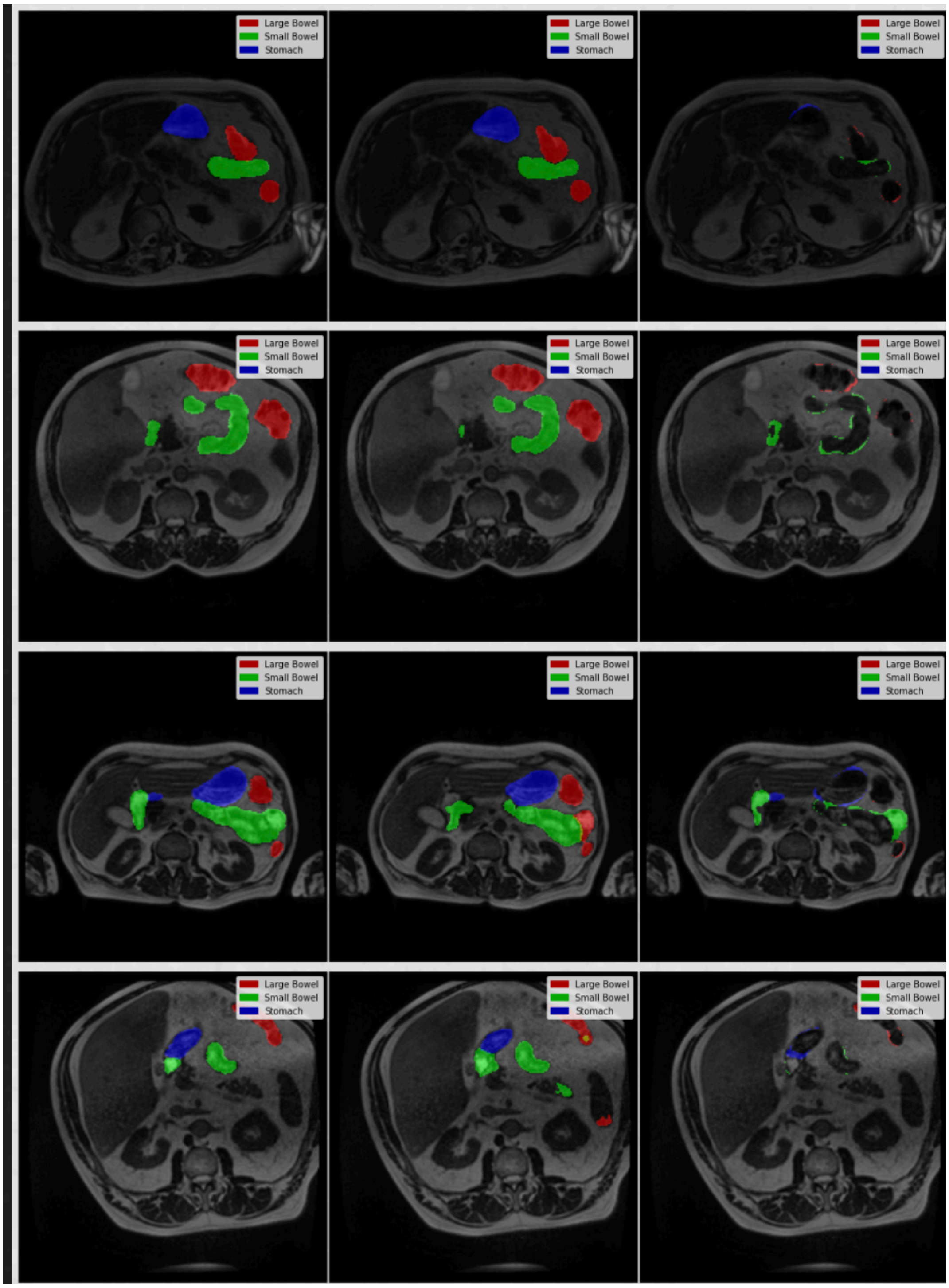


Afterwards, the last convolutional layer of the encoder has been unfrozen and the training continued for another 5 epochs, which were enough to see a small improvement and the beginning of a plateau. This time, the learning rate was set to 1e-6 as starting point, and a StepLR scheduler decreased the learning rate by 10% after each epoch.



The best model after the fine-tuning obtained a mean Dice of 0.8248 on the test set.

These are some predictions obtained using that model (left is real mask, middle is prediction, right is the difference):



4. Conclusions

The results of our study showed that the fine-tuned model we picked slightly outperformed the model built from scratch, even though it had much fewer parameters overall, probably due to the highly efficient architecture of its EfficientNetB3 encoder. This suggests that the pre-trained model was able to leverage the pre-existing knowledge learned from ImageNet, which allowed it to converge faster and perform better on our specific task. Furthermore, our results suggest that a more complex, pre-trained fine-tuned model, maybe employing a Unet++ with EfficientNetB7 as encoder, may have yielded even better results.

Another consideration we feel it's possible to make is that the training set was not large, and this probably hampered the generalization capacity of our models.

In conclusion, the project highlights the advantages of using pre-trained models for medical image segmentation.