ECOLE POLYTECHNIQUE FEDERALE DE LAUSANNE

SEMESTER PROJECT :

AUTOMATIC SEGMENTATION OF LIGHT-SHEET ZEBRAFISH IMAGES

ALY ELBINDARY (300247)

MICROBS LAB

Supervising Professor : Selman Sakar

Supervising TA : Artur Krzysztof Banach

June 6, 2024

Table of Contents

1 Background and Objectives

In this project, we have at our disposal numerous amounts of light-sheet zebrafish scans. Our aim is to segment these scans in a particular manner. The aim would be to highlight specific features of the zebrafish through these segmentations (somites & notochord, which we will look at later), and then potentially create 3D renderings of certain parts of the zebrafish embryo. The difficulty of this task is that the manual segmentation of these scans is often a extremely bothersome task, taking a significant amount of time per zebrafish scan.

Thus, the aim would be to automate this process, through the fine-tuning and usage of already existing state-of-the art segmentation models. Before diving into the machine learning (ML) models, we will first look at the process of manually segmenting these scans, in order to better understand the data that we have at our disposal, the expected segmentations as well as the difficulty of manually labeling the data for future ML tasks.

The main focus of this project will be to develop this automated process through one particular transformer model : Segment Anything Model [5] (developed by Meta). We will be looking at how the model itself is structured, its benefits when it comes to our specific use-case as well as its limitations.

2 Methodology

In this part, we will be looking at the methods used to segment light-sheet zebrafish scans.

We will first look at manual segmentation using the 3D Slicer software, and we will the discuss the implementation of the ML pipeline using SAM.

2.1 Light-Sheet Scans and Segmentation Labels

2.1.1 Manual Segmentation with 3D Slicer Software

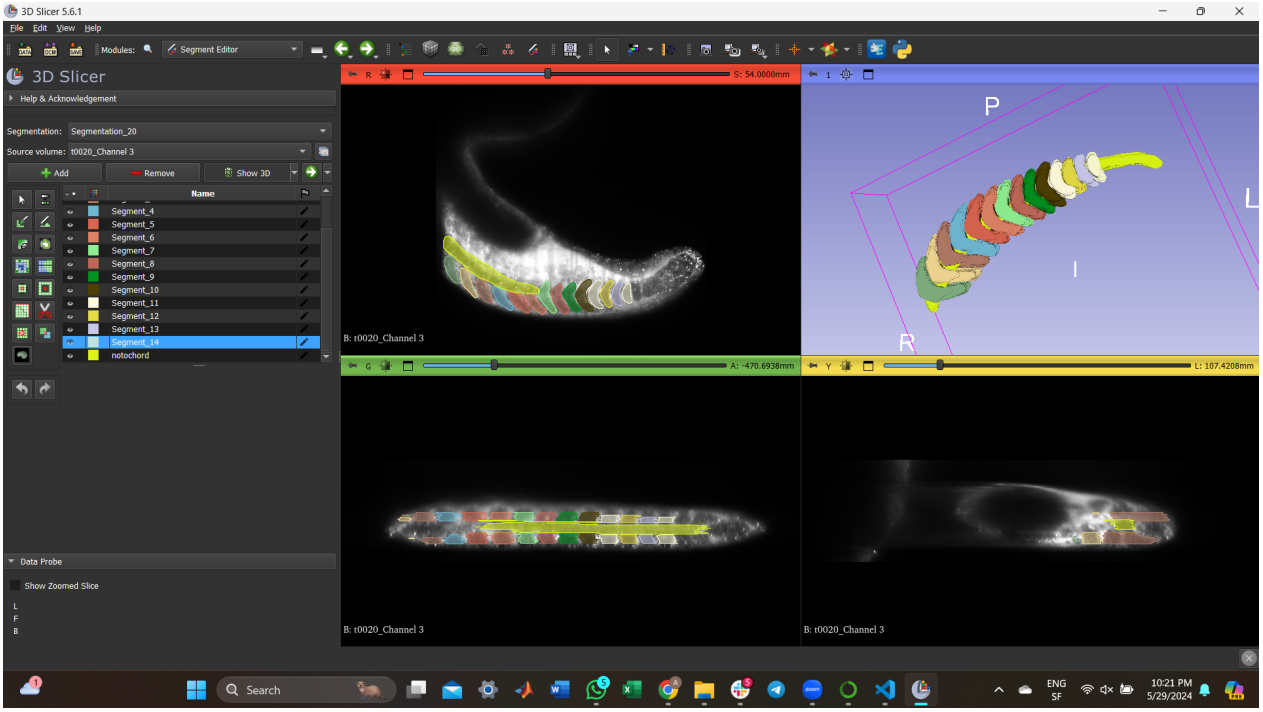


Figure 2.1.1: Screenshot Captured from 3D Slicer 5.6.1

In figure 2.1.1 above, we can see the interface of a software called 3D Slicer, which allows us to manually segment different parts of our image. In the top-left-hand widget of figure 2.1.1, we can navigate through all of the individual frames contained within the original ".tif" zebrafish scan. In this case they have already been segmented, and we can see the highlighted features that we are looking for :

- the somites ;
- the notochord (in fluorescent yellow).

The top-right-hand image contains the 3D rendering that has been generated after segmenting the frames. The two bottom widgets contain projections that have been created from the original frames of the ".tif" file.

After viewing multiple documentations and tutorials, particularly the 3D Slicer Tutorial Playlist by Jan Witowski [4], there are multiple methods that allow us to segment each frame of the scan through a relatively easy (but time consuming) process, such as "thresholding" and "grow from seeds". The main method that has been used for this process is the "grow from seeds" method.

"Grow From Seeds" is a segmentation algorithm that allows a semi-automatic segmentation of all frames within a scan by laying a foundation (seed) into the regions we would like to segment, and adding a contour around them in order to indicate where the background is. This is done for

just a couple of frames, and then we launch the algorithm so that the process is done for all frames. We can see an example of these seeds and their contour in figure 2.1.2.

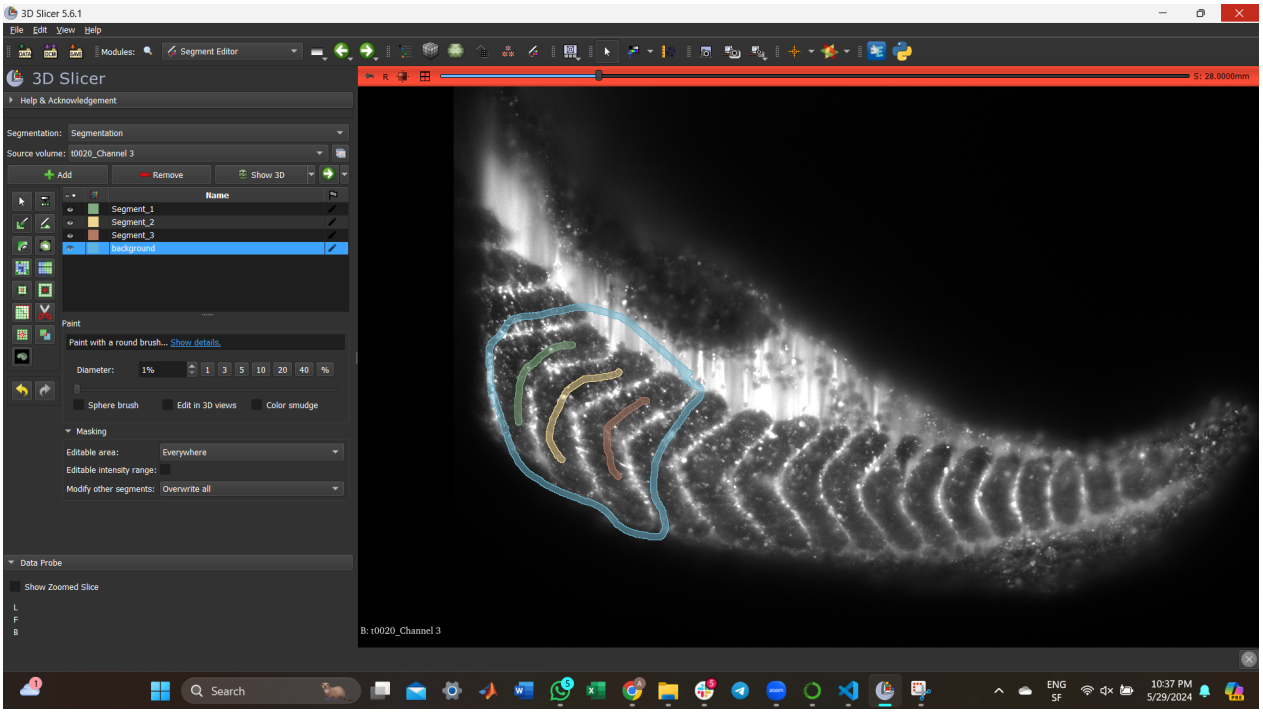
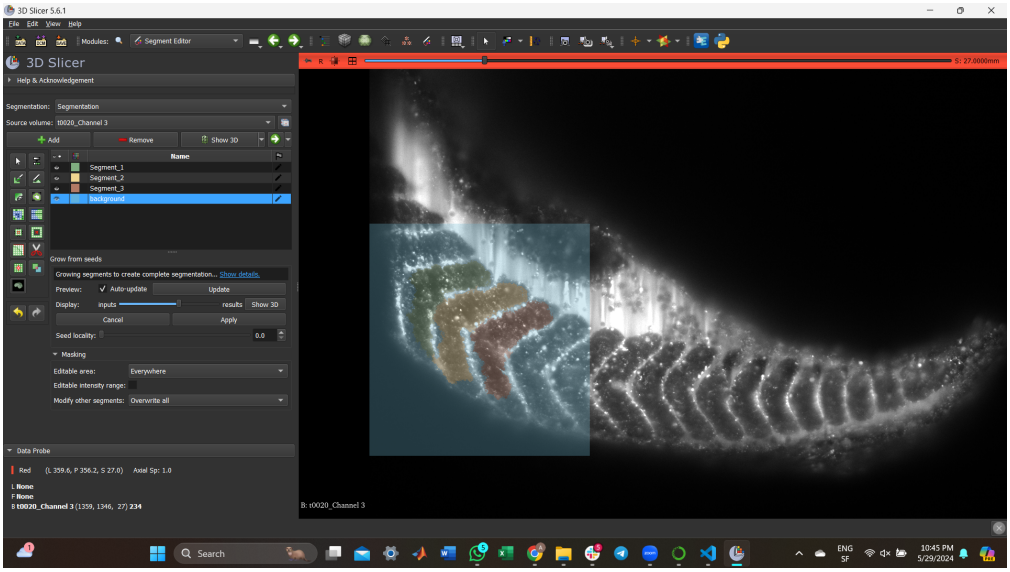
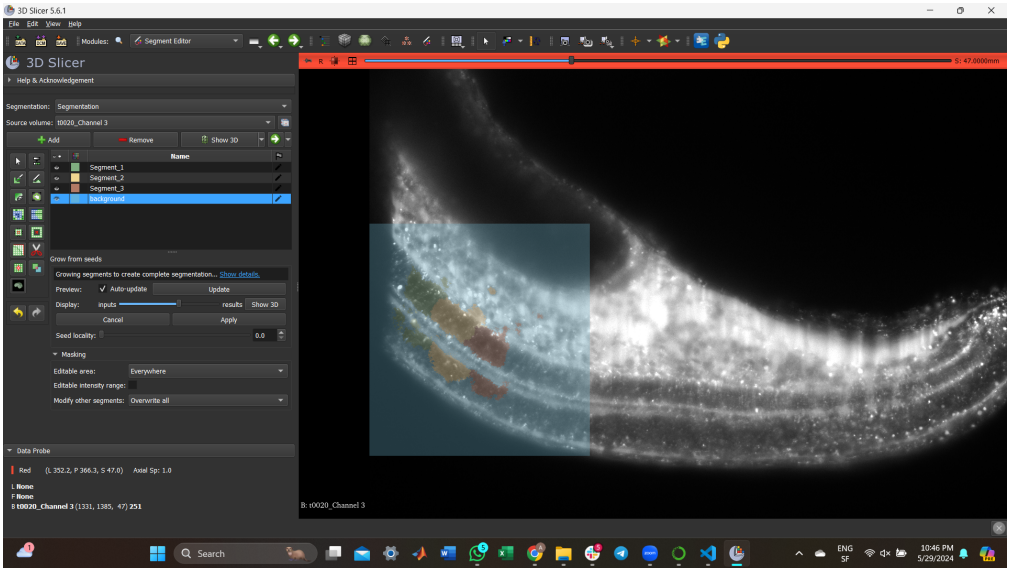


Figure 2.1.2: Laying Seeds for ”Grow from Seeds” Algorithm

Once these seeds are created, we initialize the algorithm and then preview the final results. In figure 2.1.3 below, we show some of the frames containing these segmentation previews. We can see that some of the frames are segmented in a very accurate manner, such as the frame shown in sub-figure 2.1.3a, however there are still a lot frames such as the one in sub-figure 2.1.3b. So once we apply this algorithm, not only would we need to eliminate the segmentations from the frames where they should not be at all, we would also need to ”clean” the frames where the segmentations are not entirely incorrect, but require nevertheless some polishing. This can be done with the ”Scissors” tool, which allows us to edit out specific parts of the segmentations.



(a) Accurate Segmentation



(b) Inaccurate Segmentation

Figure 2.1.3: Example of some Frames with Segmentation Preview

Combining all of these steps together leads to a relatively straightforward manual segmentation process, however this is extremely time-consuming, especially if we would like to segment all of the somites within the scans accurately as well as the notochord.

This is the motivation behind the automatic segmentation that we would like to implement, so we will now see how we can do so using SAM.

2.1.2 Datasets for Model Training & Testing

Now that we have seen the labeling method within the previous subsection 2.1.1, we will now go into the specifics of the dataset that we will be using for our project.

There will be a total of 4 Labeled zebrafish scans(4×135 frames roughly), each having 10 segmented somites as well as the notochord (making it a total of 11 classes that would need to be segmented). Only one of these scans will be used for testing whilst the rest will be used for training.

2.2 Automatic Segmentation using SAM

2.2.1 Literature Review

Looking at the massive success of certain chat-bots over recent years, such as ChatGPT, it is clear that Large Language Models (LLMs), particularly transformers, have revolutionized the field of natural language processing. Although transformers have been extremly successful specifically within the context of NLP, these architectures have also been deployed recently within the computer vision field, through models such as Stable Diffusion for image generation, Detection Transformer for object detection and, more recently, SAM for image segmentation (2023) [2]. In figure 2.2.1 below, we can see a brief overview of the transformer architecture of SAM.

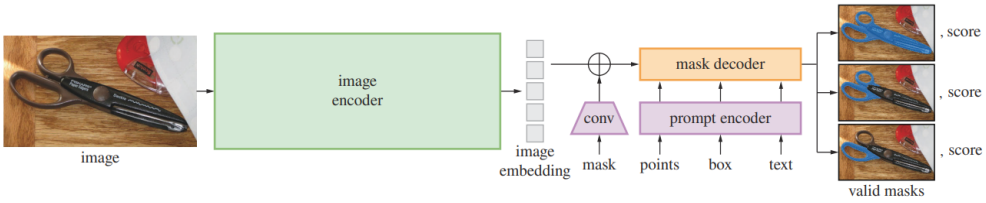


Figure 2.2.1: Segment Anything Model (SAM) overview. A heavyweight image encoder outputs an image embedding that can then be efficiently queried by a variety of input prompts to produce object masks at amortized real-time speed. For ambiguous prompts corresponding to more than one object, SAM can output multiple valid masks and associated confidence scores [5].

Thanks to SAM's transformer architecture, with the image encoder, mask decoder and prompt encoder as main components, high-level segmentations have demonstrated the capabilities of this model. We can see these examples in figure 2.2.2 below :

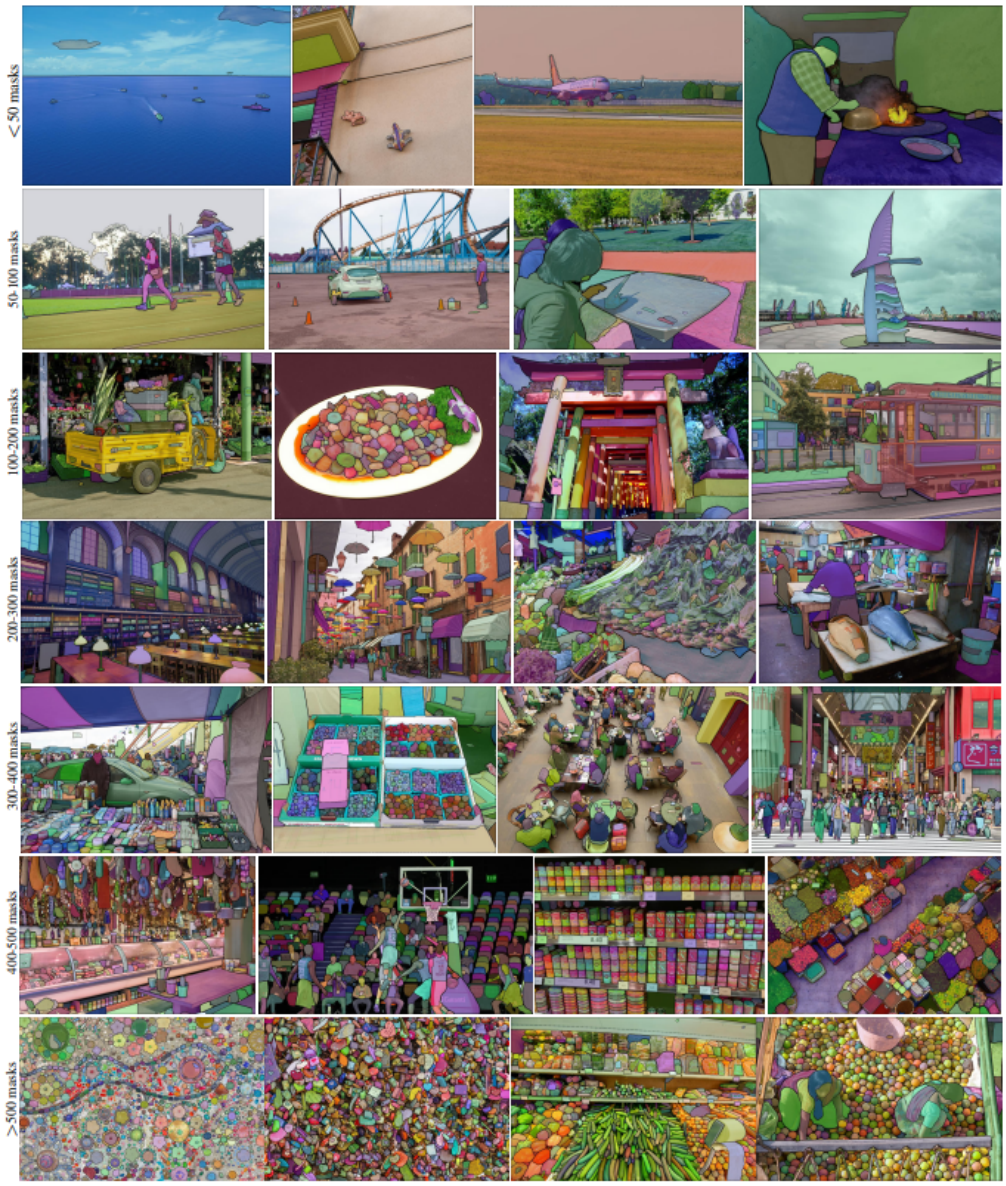


Figure 2.2.2: Example images with overlaid masks. These masks were annotated fully automatically by SAM [5].

The reason why large transformer models such as SAM can perform on a wide variety of data, is because these models were already pretrained on a massive amount of data : in the case of SAM, it has been pretrained on around 11M images and more than 1B masks [1]. In order to then deploy this model on a speific task, we would need to finetune it, which means that we would need to train the pretrained model on task-specific data (in our case : the light-sheet scans with manually generated masks) : this is what we call finetuning.

We will now see how we can implement this finetuning pipeline, but first we will look at how we will need to adapt the raw data (the ".tif" light-sheet scans as well as the ".nrrd" segmentation files generated through 3D Slicer).

2.2.2 Raw Data Adaptaion

Often times when it comes to training a model, the data needs to be of a certain format first, and thus we would need to understand the initial format of our scans as well as our segmentation masks, and then we can see how we would need to adapt it. In this project, the finetuning pipelines that we will be utilizing are based of an already existing pipeline that has been specifically created for segmentation of human brain MRI scans [8].

The data format that is required is to split the singular ".tif" file into multiple ".tif" files, one for each frame. Similarly, the ".nrrd" file containing the segmentations must also be split into multiple ".tif" files. These files will then be subsequently converted into regular images within our pipeline.

When doing these conversions within our `python` environment, there are two main problems that appear :

- incorrect pixel values when converting a ".tif" frame file into a ".png" as in figure 2.2.3 below:

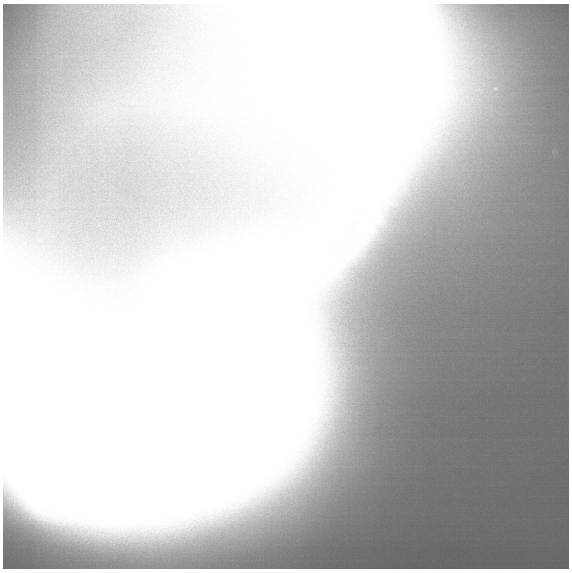


Figure 2.2.3: Incorrect Conversion of ".tif" Frame into a ".png"

- incorrect orientation of the masks as in figure 2.2.4 below (somites are located in the top-right side of the image, whereas they should be in the bottom-left, as seen in section 2.1.1):

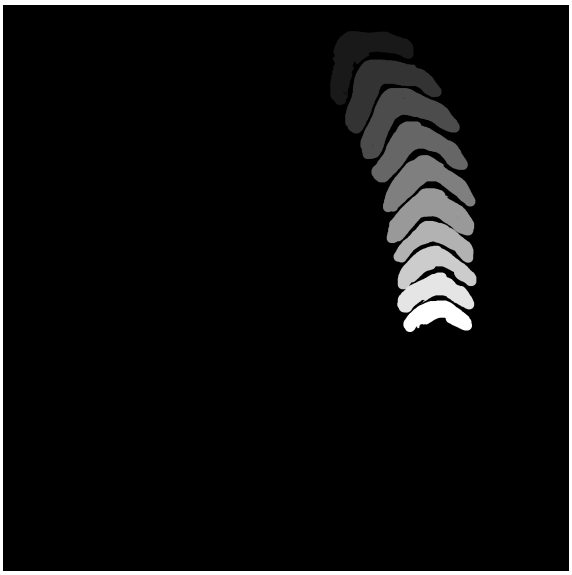


Figure 2.2.4: Incorrect Conversion of ".nrrd" Frame into a ".png"

In order to solve these conversion issues, we developed some functions that allow us to do the proper pixel value conversion (from `uint16` to `uint8`) for the ".tif" frames, as well as properly rotating the ".nrrd" masks. The code that allows this pixel value conversion can be found in the function `adjust_jpg()` within listing 1 of the appendix 6. The mask images are initially rotated around their own center by 90°, and are then pivoted around the middle horizontal axis of the image by 180°. The new images that we obtain can be seen in figures 2.2.5 and 2.2.6 below :

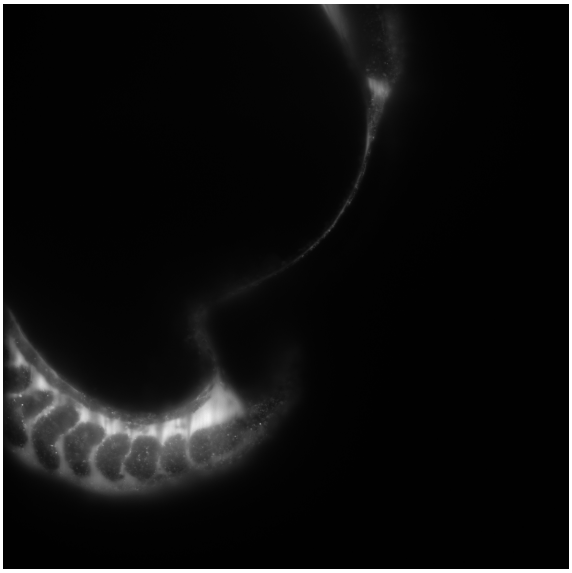
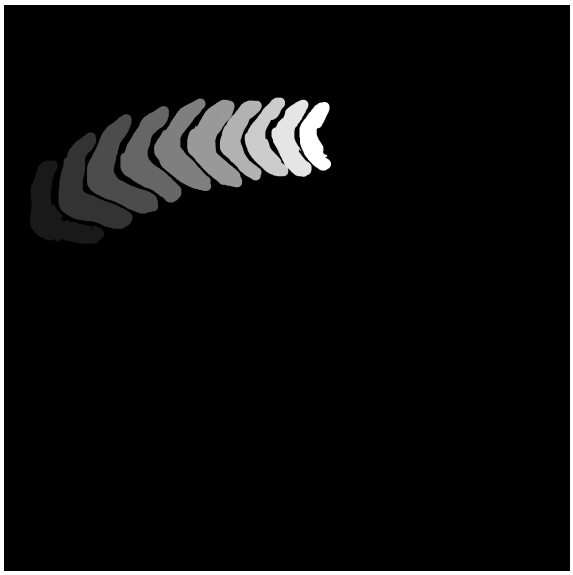
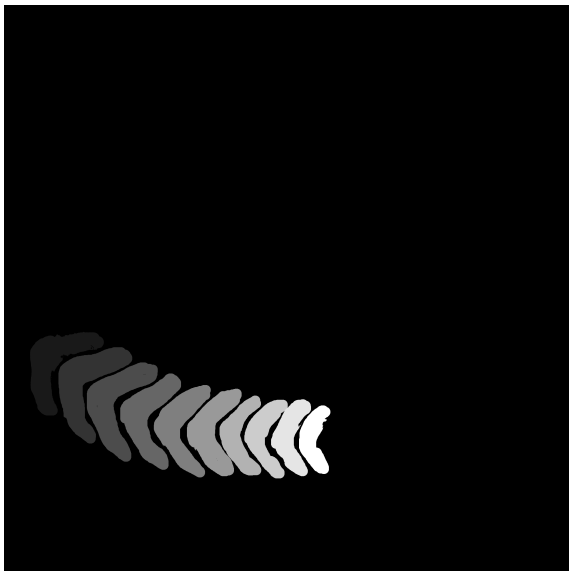


Figure 2.2.5: Correct Conversion of ".tif" Frame into a ".png"



(a) Mask after Rotating by 90° around its own Center



(b) Final Mask after Rotating by 180° Middle Horizontal Axis

Figure 2.2.6: Main figure caption describing both subfigures

The code that was used in order to obtain these rotations can be found in listing 1 within the appendix 6, more specifically with the functions `rotate_around_middle_horizontal_axis()` and `rotate_data()`.

Now that we have adapted the data, we will be moving onto the training pipelines.

2.2.3 Finetuning SAM

The implementation of the SAM architecture that we will be using will be the one provided by HuggingFace [1], since not only do we get a model implementation that is suited for training (i.e. model parameters can be updated through backpropagation of a chosen loss), but there is also an image processor (called SamProcessor) which converts the input image and the prompt into their corresponding embeddings for the model.

Important criterias that must be selected for proper training are the loss function and the evaluation metric. Let's start with the loss function.

When it comes to image segmentation, there exists a wide variety of loss functions that can be used, such as Binary Cross-Entropy, Weighted Cross-Entropy, Dice Loss and Focal Loss just to name a few. Although using any appropriate image-segmentation loss would yield similar results, we decided to select the Focal Loss, since it works best with highly-imbalanced dataset by down-weighting the contribution of easy examples, enabling our model to learn hard examples [3]. This would be particularly useful for segments such as the notochord, which doesn't necessarily appear as often as the somites do within the frames. Focal Loss is defined from the Cross-Entropy (CE) loss for binary classification :

[1] SAM HuggingFace

$$\text{CE}(p, y) = \begin{cases} -\log(p) & \text{if } y = 1 \\ -\log(1-p) & \text{otherwise.} \end{cases} \tag{2.1}$$

where $y \in \{0, 1\}$ represents the ground-truth class and $p \in [0, 1]$ is the model's estimated probability for the class with label $y = 1$. To make this notation simpler, we define p_t as :

$$p_t = \begin{cases} p & \text{if } y = 1 \\ 1-p & \text{otherwise,} \end{cases} \tag{2.2}$$

and now combining equations 2.1 and 2.2, we can now define the focal loss as :

$$\text{FL}(p_t) = (1-p_t)^{\gamma} CE(p_t), \tag{2.3}$$

where γ is a tunable parameter. The term $(1-p_t)^{\gamma}$ is the modulating factor [3] that allows for down-weighting easy examples as discussed previously.

The evaluation metric that we will be using will be "Intersection over Union" (IoU). IoU allows us to quickly understand how "similar" the model predicted mask is to the ground truth mask. It's a percentage representing the ratio between the intersection of the segmented regions of the ground truth mask and the predicted mask and their union. In figure 2.2.7 below, we can see a diagram representing this metric :

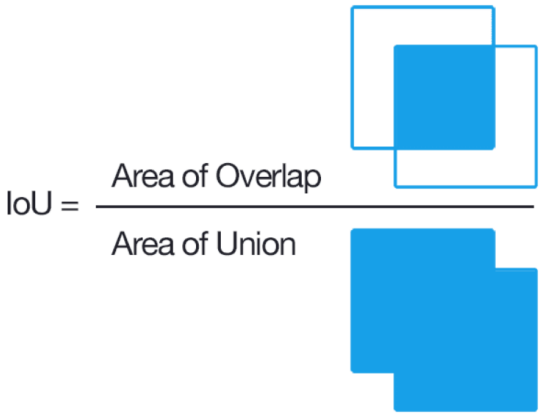


Figure 2.2.7: Diagram illustrating the Intersection over Union (IoU) evaluation metric [7]

Thus, the closer this metric is to 1, the more accurate the model is.

Now that the criterias have been selected, we will now move onto the training setup itself. In an initial part, we will be finetuning SAM by considering ourselves in a simple binary image segmentation task (i.e. only one class : all somites as well as the notochord will have the same color). Currently, at the time of writing this report, due to the recent release of SAM, the most notable finetuning pipelines that exist online of SAM only consider the binary image segmentation case, even when looking at the training script provided by HuggingFace. Thus, once we test our model in the binary case, we will attempt to create our own model structure utilizing SAM in order to achieve multi-class image segmentation, and see how feasible it is.

The way we will structure our multi-class segmentation model is by adding a convolutional layer on top of our imported HuggingFace model. This will allow us to generate a number of masks equal to the number of classes, and thus finetuning the parameters of this augmented model will include not only the parameters of just SAM, but also those of the convolutional layer. This custom model inherits from the `torch.Module` class and can be seen in listing 2 within the appendix 6.

3 Results

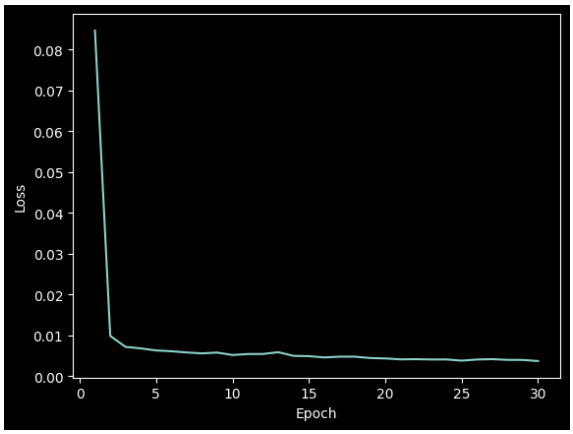
In this section, we will be looking the evolution of the train/test loss and IoU over the number of epochs. This will be done for different settings that we have chosen, in order to see if some configurations make a difference.

For these results, all of the segmentations contain the same number of somites (10) as well as the notochord. Additionally, HuggingFace provides different versions of SAM [2] that have a different amounts of parameters and that could potentially be task-specific.

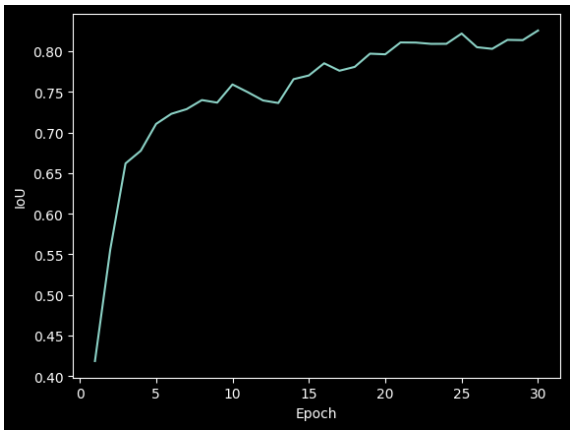
3.1 Binary Mask Segmentation - Model : facebook/sam-vit-base

This model is a lighter version of the larger SAM model that can also be found on HuggingFace. The finetuning results can be seen below :

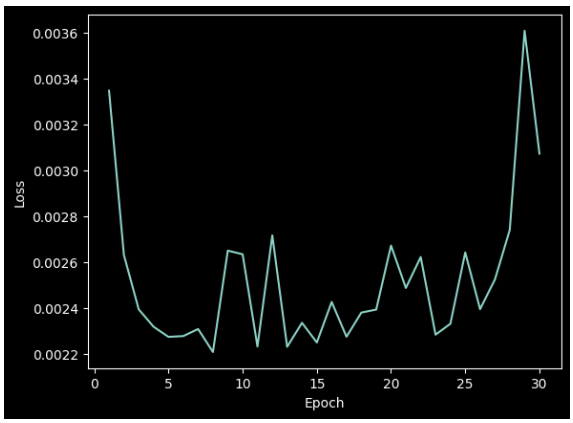
[2] SAM Models



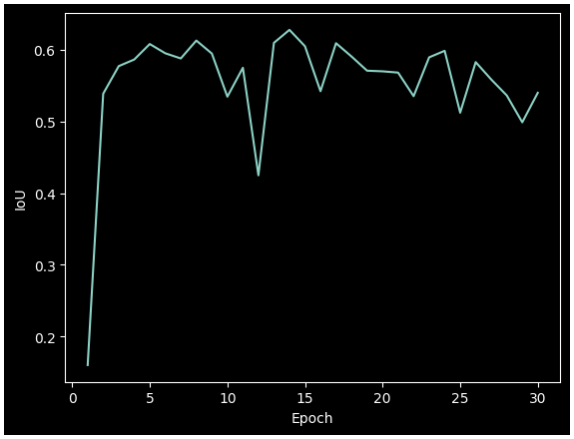
(a) Evolution of Train Focal Loss over Number of Epochs



(b) Evolution of Train IoU over Number of Epochs

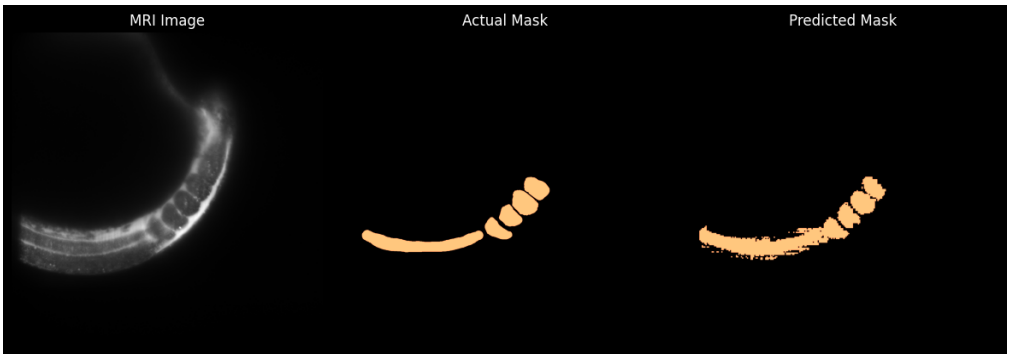


(c) Evolution of Test Focal Loss over Number of Epochs

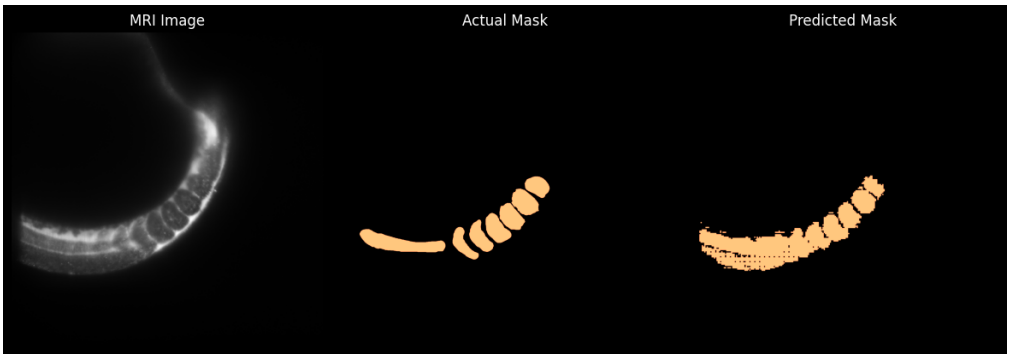


(d) Evolution of Test IoU over Number of Epochs

Figure 3.1.1: Focal Loss and Iou Evolution over Number of Epochs



(a) Prediction 1

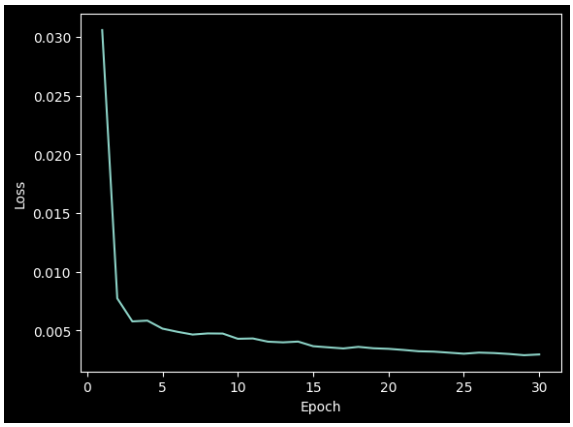


(b) Prediction 2

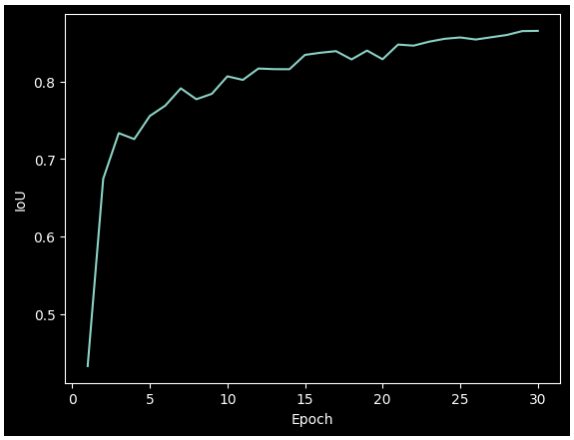
Figure 3.1.2: Some Examples of Inferences on Test Set

3.2 Binary Mask Segmentation - Model : facebook/sam-vit-huge

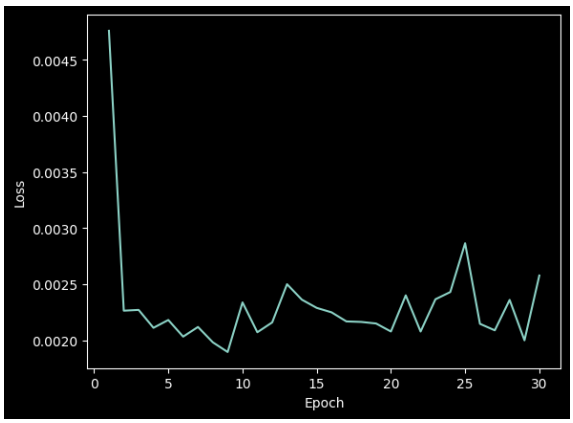
This model is the larger version of the SAM model that can be found on HuggingFace. It is the most downloaded version on the website. The finetuning results can be seen below :



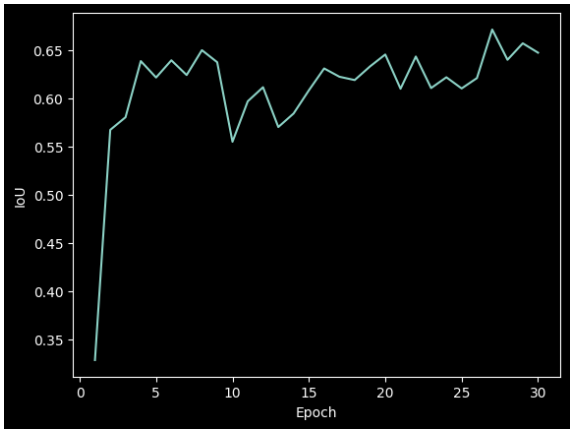
(a) Evolution of Train Focal Loss over Number of Epochs



(b) Evolution of Train IoU over Number of Epochs

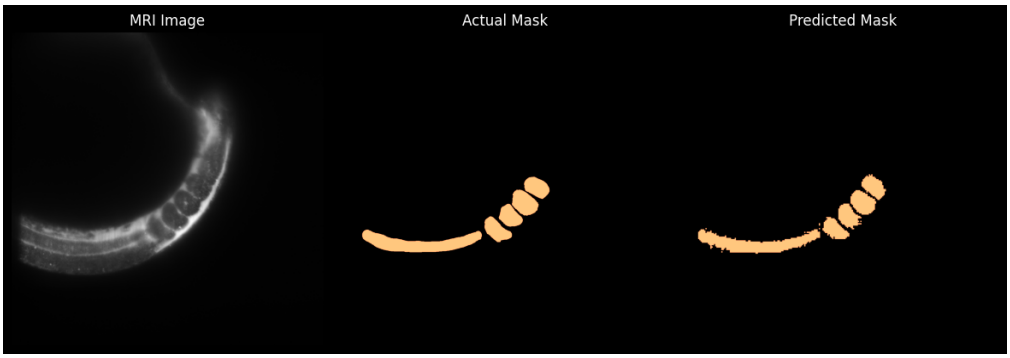


(c) Evolution of Test Focal Loss over Number of Epochs

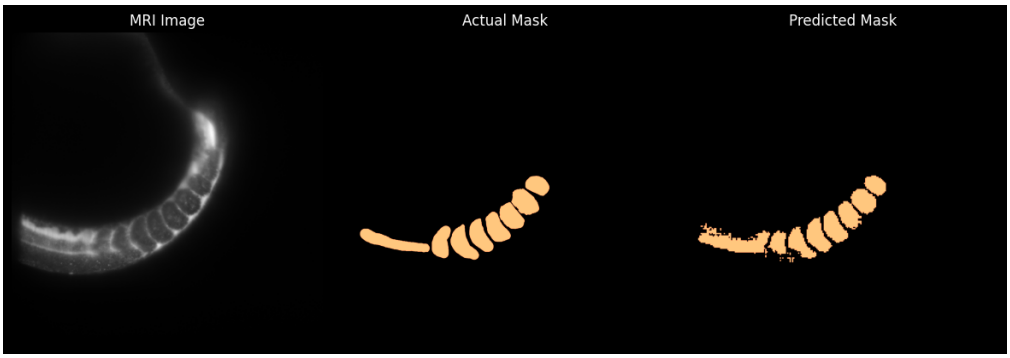


(d) Evolution of Test IoU over Number of Epochs

Figure 3.2.1: Focal Loss and Iou Evolution over Number of Epochs



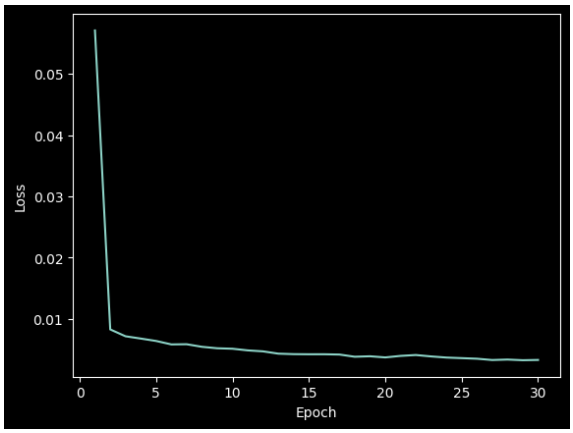
(a) Prediction 1



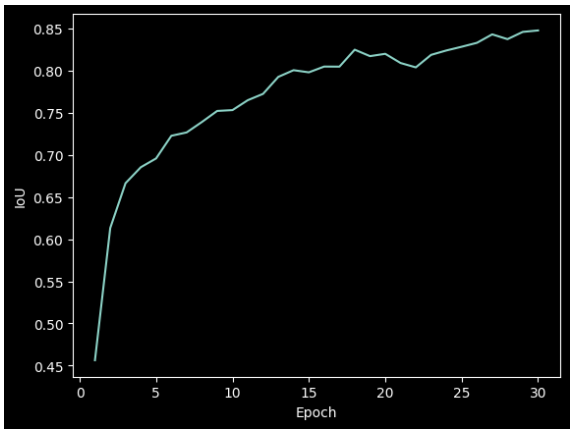
(b) Prediction 2

Figure 3.2.2: Some Examples of Inferences on Test Set

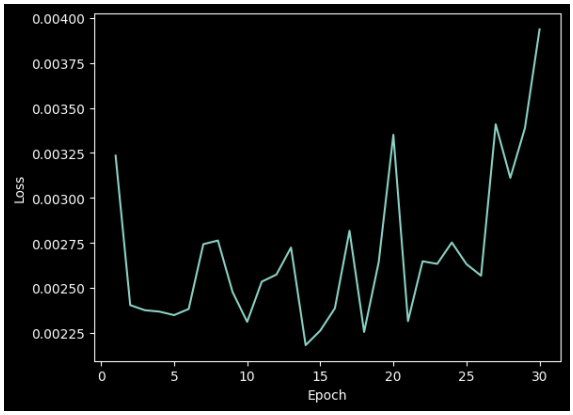
3.3 Binary Mask Segmentation - Model : wanglab/medsam-vit-base



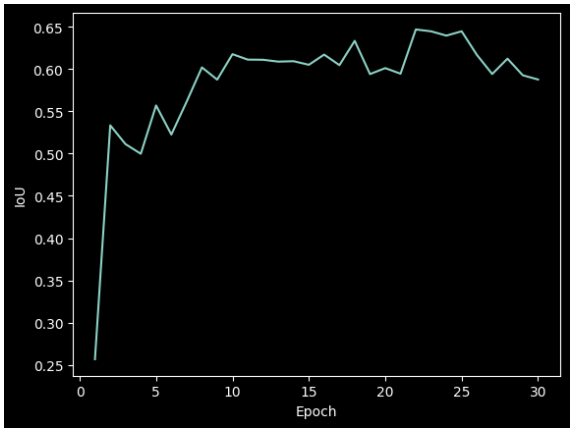
(a) Evolution of Train Focal Loss over Number of Epochs



(b) Evolution of Train IoU over Number of Epochs

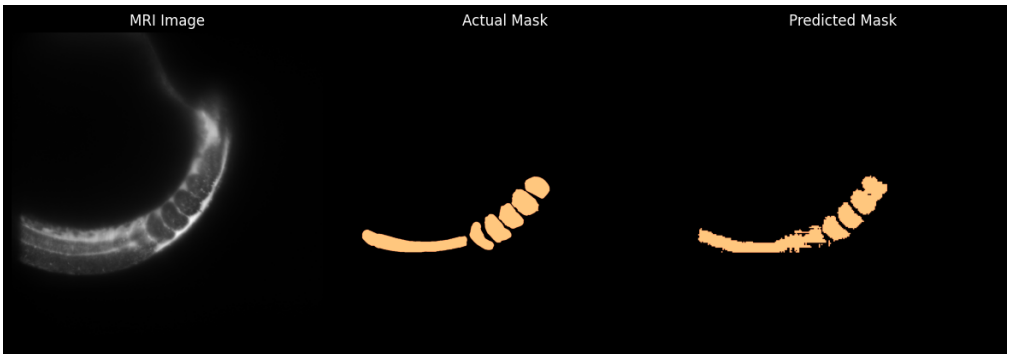


(c) Evolution of Test Focal Loss over Number of Epochs

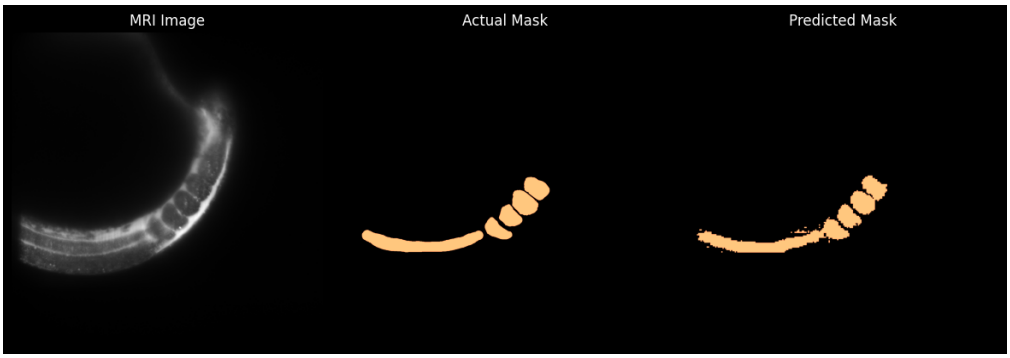


(d) Evolution of Test IoU over Number of Epochs

Figure 3.3.1: Focal Loss and Iou Evolution over Number of Epochs



(a) Prediction 1

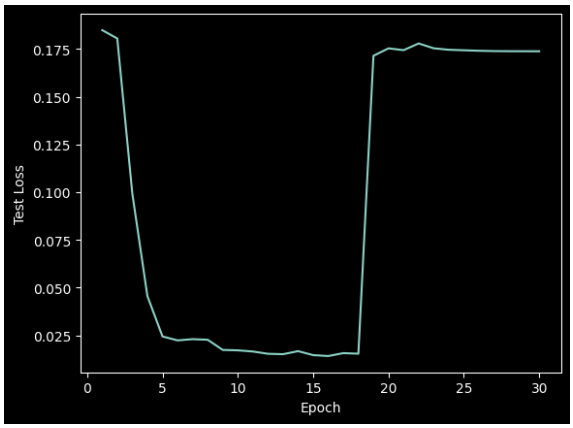


(b) Prediction 2

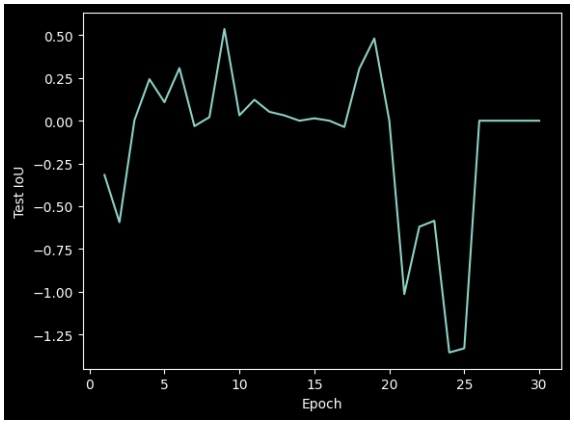
Figure 3.3.2: Some Examples of Inferences on Test Set

3.4 Multi-Class Image Segmentation

For this part we will be using the SAM-vit-base model along with the aforementioned multi-class structure which can be found within the appendix in listing 2. Additionally, the way we generate the dataset has been changed : instead of having a input/mask pair for each frame, for each input image we now have a folder containing 11 different mask frames (one for each of the classes), that way the focal loss can be computed individually for each class. Subsequently, the training pipeline we use previously for binary segmentation was also altered in order to accommodate for these changes.



(a) Evolution of Test Loss over Number of Epochs



(b) Evolution of Test IoU over Number of Epochs

Figure 3.4.1: Evolution of Train and Test Loss over Number of Epochs

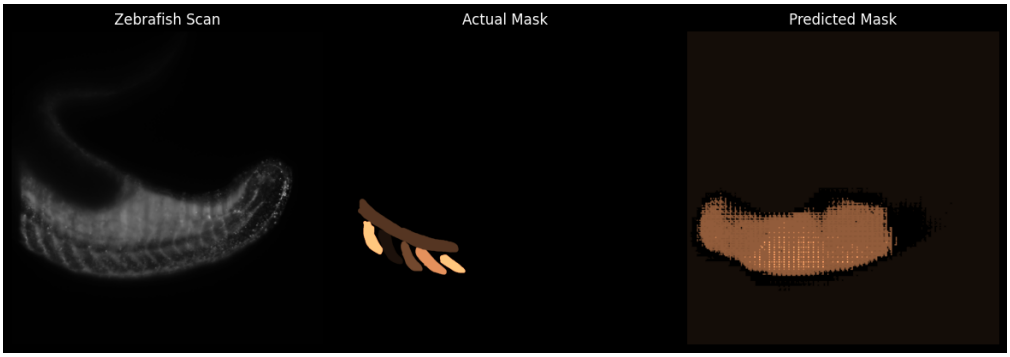


Figure 3.4.2: Example of Inferences on Test Set

4 Discussion

In this part, we will discuss the different results that we have obtained in section 3 for each of the different models and methods and compare them together in order to determine the one with the best results. We will then look into potential improvements that can be done within the context of this project.

4.1 Results Analysis

Firstly, when looking at the different binary-segmentation results, we can see that after a relatively low number of epochs (30 in this case), the evolution of the tesing IoU starts stagnatin, indicating that training for a higher number of epochs would not be useful. More specifically, the reduced version of the HggingFace SAM model (sam-vit-base) with around 93.7M parameters [3] reaches a peak testing IoU of around 61% when looking at plot 3.1.1d. However, the evolution of the test

[3] HuggingFace Base SAM Model Size

loss for both the larger SAM model (with 641M parameters [4]) and the medical SAM model (model size not available on HuggingFace) in plots 3.2.1d and 3.3.1d respectively, we can see that these models reach a higher peak IoU of around 67%. This 7% difference in IoU is very much significant in terms of the model's performance : when comparing the base model's inferences in plot 3.1.2 to the ones of the other models in plots 3.2.2 and 3.3.2, we can see that not only are the segmentations of the somites more distinguised from each other and the notochord, the individual segmentation of each part of the zebrafish is much more defined and accurate. This shows that the usage of larger models (or the more task-specific one with MedSAM) has a significant benefit on performance.

When it comes to the results of the multi-class segmentation, the model's performance is not good at all. This can be seen directly from test loss and test IoU plots in figure 3.4.1, particularly with the IoU metric which has very inconsistent numbers. This also translates to the inferences of the model : an example of these inferences can be also be seen in figure 3.4.2. These erroneous results could be due to a multitude of reasons, particularly with the training pipeline that has been utilized, even if it is based off of the successful pipeline that was utilized for the binary segmentation. However, due to the lack of resources relative to SAM specifically due to its recent release, it is difficult to know exactly the issue with the pipeline and thus analysis would need to be conducted.

4.2 Potential Improvements

Now that a baseline has been established for the segmentation of light-sheet zebrafish scans, there are some improvements that can still be done within this project that have not been implemented yet, since the main focus of the project was to understand and implement a finetuning strategy for a newly released segmentation model in SAM that has not yet been documented as much as other traditional segmentation models :

- **Hyperparameter Search** : for all of the different arbitrarily chosen factors in order to train the model (learning rate, batch size, loss function, weight decay...), it would be beneficial to run our finetuning pipeline over a range of different values for each of these parameters, and determine if some configurations could yield higher IoU scores. For this project, we decided to apply the commonly used configurations of these hyperparameters by looking at other implementations of computer-vision training pipelines, such as the article that was used as a basis for this project [8] ;

- **Multi-Class Segmentation Fix** : in practice, in order to go from the basic binary segmentation task to the multi-class one, a convolutional layer is simply added on top of the base model used for binary segmentation, however this was not sufficient in order achieve this objective.

- **Data Improvement** : it is clear that better data labeling using 3D Slicer (more accurate segmentations) would improve our results, however what would be more impactful is the amount of data we need. In ML, it is widely recognized that the more data we have, the more capable the model will become when finetuned onto this specific data. Therefore more labeled zebrafish scans would help improve performance. However, one could achieve this

[4] HuggingFace Huge SAM Model Size

increase in data not just through more labeling, but also through different data preprocessing techniques, such as duplication of certain images. However, with this increase in data, certain challenges such as class imbalance become more prevalent [6], and would need to be taken into account.

4.3 Drawbacks of SAM

Due to the relatively recent release of SAM (2023), the are not many documentations or use cases of this model specifically when it comes to the multi-class segmentation task. This makes it difficult to be able to not only figure out how to adapt the model onto a specific use-case, but also understand certain problems or limitations that might arise within those contexts, like we have seen with the multi-class task.

Additionally, although this does not have a significant impact in our project, it is nevertheless important to note that prompting and the prompt encoder of SAM plays a significant role when it comes to the model's performance. The prompt that we give the model is bounding box region which indicates to the model the area within the image in which it should segment. An example of this can be seen in figure 4.3.1 below :

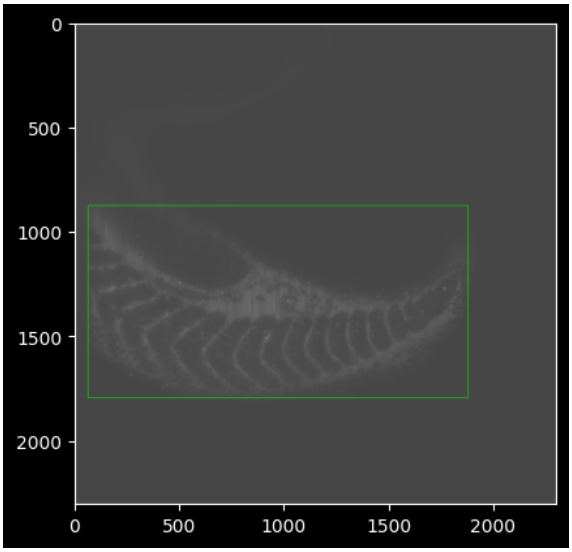


Figure 4.3.1: Example of a Prompt Bounding Box for a Given Input Image (Image Brightness has been Altered for Better Visualization).

As we can see, these bounding boxes only serve as a contour surrounding the body of the zebrafish embryo. Therefore, as long as the embryo does not significantly change positions from frame to frame, then one general bounding box can be generated from the very beginning for all input images, and the performance would remain the same. This detail is important to note since the presence of a bounding box as an input is crucial for the model's performance, meaning that if future scans are added, then the zebrafish embryo would have to remain relatively static across all frames.

5 Conclusion

Within this project, we have had the opportunity to explore different manual image-segmentation methods such as the ”Grow from Seeds”, which have proven to be very time-consuming. Thus motivating the exploration of different ML automatic segmentation methods.

The usage of SAM, one of the first vision-transformers, has been particularly interesting. This is because we are exploring the application of the architecture that has been very successful with LLMs such as ChatGPT, but within the context of computer vision, which makes SAM unique in regards of its structure relative to other traditional segmentation models, such as U-Net.

The results that we have produced show that the model is able to be relatively quickly finetuned onto our dataset and is able to recognize certain feature rapidly, however the reliance on the prompt mechanism as well as the difficulty in the implementation of the multi-class segmentation hold it back.

6 Appendix

```python
def adjust_jpg(image):
    """
    Takes in a jpeg that has been extracted from a .tif file and
    adjusts the pixel intensity in order to get a readable image

    Inputs :
        - image : original jpeg to be adjusted
    Outputs :
        - image : adjusted jpeg
    """
    # Convert image to uint8 type
    if image.dtype != np.uint8:
        image = (image / (np.max(image) / 255)).astype(np.uint8)

    return image

"""
Rotate .nrrd data in different ways
"""
def rotate_around_middle_horizontal_axis(data, angle):
    # Get the height and width of each frame
    height, width, num_frames = data.shape

    # Calculate the middle index along the horizontal axis for each
   frame
    middle_horizontal_indices = width // 2

    # Initialize an array to store the rotated data
    rotated_data = np.empty_like(data)

    # Iterate over each frame
    for i in range(num_frames):
        # Rotate the data within the frame around the middle horizontal
    axis
        rotated_frame = np.rot90(data[:, :, i], k=angle // 90, axes=(1,
    0))

        # Flip the rotated frame along the horizontal axis to align
   with the original orientation
        rotated_frame = np.flip(rotated_frame, axis=1)

        # Store the rotated frame in the output array
        rotated_data[:, :, i] = rotated_frame

```

```
    return rotated_data

def rotate_data(data, angle):
    # Rotate the data by the specified angle
    rotated_data = np.rot90(data, k=angle//90, axes=(0, 1))  # Adjust axes if needed
    return rotated_data

"""
TIF input files
"""
def split_multi_page_tiff(input_file, output_folder, file_prefix):
    """
    This function takes in a zebrafish.tif file, which contains hundreds of frames,
    and generates an individual .tif file for each frame (in order to match
    the training pipeline data structure). Saves them in a folder

    Inputs :
        - input_file : zebrafish tif file path
        - output_folder : folder path where the files will be saved
        - file_prefix : string representing the prefix with which we're naming the
                        files


    """
    # Create the output folder if it doesn't exist
    if not os.path.exists(output_folder):
        os.makedirs(output_folder)

    with WandImage(filename=input_file) as img:
        for i, page in enumerate(img.sequence):
            with WandImage(page) as single_img:
                output_path = os.path.join(output_folder, f"{file_prefix}_frame_{i}.tif")
                single_img.save(filename=output_path)
                single_img.clear()


"""
TIF Mask Labels
"""
def convert_nrrd_to_tiffs(input_file_path, output_folder, file_prefix):
    """
    This function takes in a segmentation.nrrd file, which contains hundreds of frames,
```

```
    (corresponding to segmentation masks) and generates an individual tif file for each
    mask (in order to match the training pipeline data structure). Saves them in a folder

    Inputs :
        - input_file : segmentation.nrrd file path
        - output_folder : folder path where the files will be saved
        - file_prefix : string representing the prefix with which we're naming the
                        files


    """
    # Get data in .nrrd file
    data, _ = nrrd.read(input_file_path)

    # PROPER ROTATION CONFIGURATION
    data = rotate_data(data, 90)
    data = rotate_around_middle_horizontal_axis(data, 180)

    # Create the output folder if it doesn't exist
    if not os.path.exists(output_folder):
        os.makedirs(output_folder)

    num_frames = data.shape[2]
    for i in range(num_frames):
        frame_data = data[:, :, i]

        # Normalize data to 0-255 range
        normalized_frame = ((frame_data - np.min(frame_data)) / (np.max(frame_data) - np.min(frame_data)) * 255).astype(np.uint8)

        # Convert to PIL Image
        frame_image = PilImage.fromarray(normalized_frame)

        # Save as TIFF file
        frame_image.save(os.path.join(output_folder, f"{file_prefix}_frame_{i}_mask.tif"))
```

Listing 1: Data Adaptation Code

```
class MultiSAM(nn.Module):
    def __init__(self, num_classes):
        super(MultiSAM, self).__init__()

        self.sam = SamModel.from_pretrained("facebook/sam-vit-base")
```

```
        self.conv_layer = nn.Conv2d(in_channels=1, out_channels=
    num_classes, kernel_size=1)

    def forward(self, x):

        x = self.sam(pixel_values = x["pixel_values"],
                     input_boxes = x["input_boxes"],
                     multimask_output=False).pred_masks.squeeze(1).to


        x = self.conv_layer(x)

        return x
```

Listing 2: Multi-Class Segmentation Model

References

[1] Meta AI. *Segment Anything*. 2023.

[2] Rafael Guedes. *SAM: Segment Anything Model - Quickly customize your product landing page with SAM*. https://towardsdatascience.com/sam-segment-anything-model-4b25a47245f2. 2021.

[3] Shruti Jadon. "A survey of loss functions for semantic segmentation". In: *2020 IEEE Conference on Computational Intelligence in Bioinformatics and Computational Biology (CIBCB)*. 2020, pp. 1–7. DOI: 10.1109/CIBCB48159.2020.9277638.

[4] Jan Witowski. *3D Slicer Tutorial*. YouTube playlist. 2019. URL: https://www.youtube.com/playlist?list=PLeaIM0zUlEqswa6Pskg9uMq15LiWWYP39.

[5] Alexander Kirillov et al. *Segment Anything*. 2023. arXiv: 2304.02643 [cs.CV]. URL: https://arxiv.org/pdf/2304.02643.pdf.

[6] Alexandra L'Heureux et al. "Machine Learning With Big Data: Challenges and Approaches". In: *IEEE Access* 5 (2017), pp. 7776–7797. DOI: 10.1109/ACCESS.2017.2696365.

[7] Adrian Rosebrock. *Intersection over Union (IoU) for object detection*. URL: https://pyimagesearch.com/2016/11/07/intersection-over-union-iou-for-object-detection/.

[8] Saskia Dwi Ulfah. *Brain MRI Segmentation with Segment Anything Model (SAM)*. 2023. URL: https://medium.com/@sdwiulfah/brain-mri-segmentation-with-segment-anything-model-sam-16d0b4101a85.