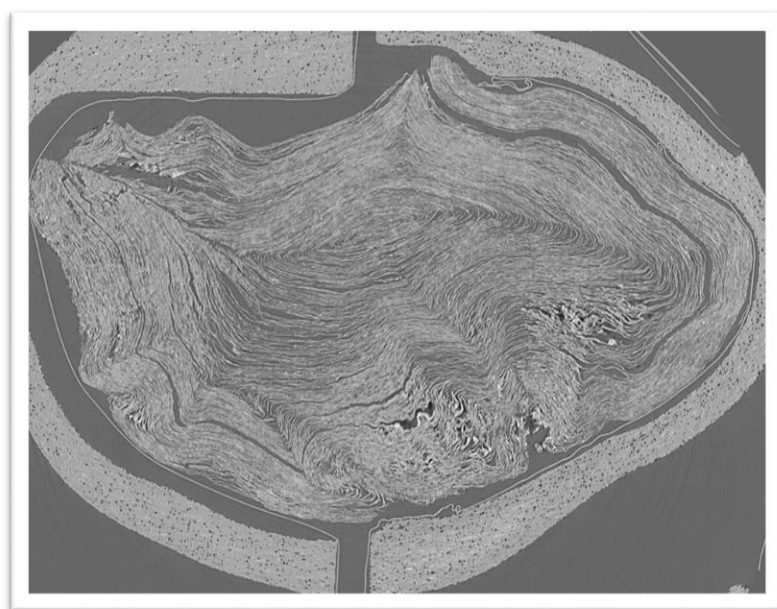


Segmentation of Scroll Surfaces

The Herculaneum papyri have a troubled history of unrolling. Most attempts historically, even those considered to be relatively successful, have rendered them into many thousands of unrecognizable fragments – later to be painstakingly pieced back together by some of society's most patient (and dedicated) researchers. I won't spend time detailing these attempts here. They are well documented and any summarization I make of them is sure to be fraught with inaccuracies or misunderstandings. Instead, I'll discuss the methods and difficulties along the way during the second phase of the Vesuvius Challenge.

The first phase of the challenge, ending December 31st, 2023, resulted in about 1400cm² of readable text spanning fifteen columns– roughly 5% of the pherc. Paris 4. The second phase raised this bar ever so slightly to 95% of four scrolls, with 90% readable text. This slight increase in difficulty influenced us (and many others) to focus our efforts towards fully automated segmentation.

We have come up with a multi-stage pipeline that we believe works very well, and that given a bit more fine tuning, should be capable of segmenting most any scroll (though p.herc Paris 3 might disagree).



Digging Through the Weeds

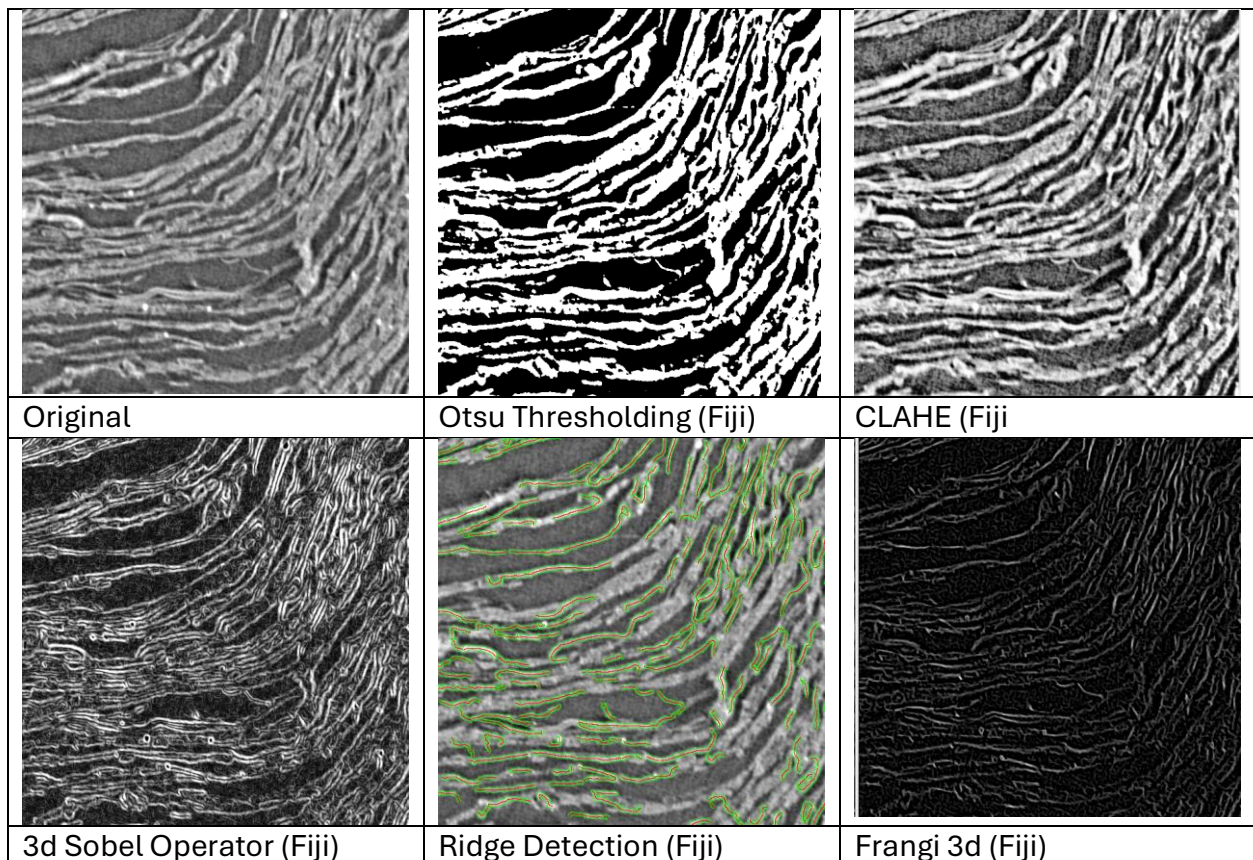
A brief look at the above image makes clear that delineating the boundaries between wrap-to-wrap is exceedingly difficult. Those familiar and experienced in manual

scroll segmentation may be able to work through ten or twenty cm^2 a day, feverishly scanning the volume for any subtle pattern that might indicate just where the hell the sheet has trailed off to.

Many attempts have been made to find classical computer vision algorithms or pipelines that can assist human or machine in tracing these subtle patterns, some even showing great promise. Optical Flow Segmentation(OFS) was one of the first breakthroughs in this respect, allowing for efficient segmentation of sheets large enough to get lost in.

Here's a look at a few common operations performed on images in pursuit of features that can be used in downstream tasks. Over the course of this year I've attempted just about every single non-deep learning algorithm that has either:

- A plug-in/built-in within Fiji(imageJ), a plugin/built-in within 3D slicer, a plugin/built-in within Dragonfly3d
- Or, a scikit-image, opencv, or scipy implementation



Most of these result in images that are no less easy to understand than the raw data they're derived from. There are a few combinations or "filter stacks" a person can do to give understandable results, the primarily through a combination of contrast/histogram normalization to enhance dark edges (which exist near dense objects in CT scans like ours), some sort of edge filter (I'm partial to scharr/prewitt in this task) , and a ridge or

vesselness operator like a hessian or a frangi filter. If followed up with a gradient filter (you can just run another scharr/prewitt) and a removal of the second “edge” or derivative that these filters commonly output, you can end up with a sort of accurate set of points approximating the recto surface.

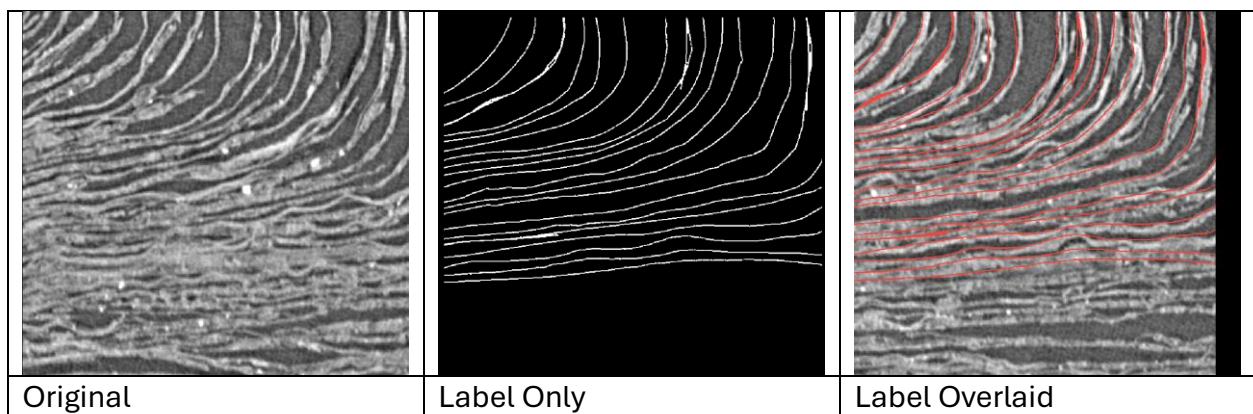
Unfortunately, this is prone to many errors in detecting false surfaces. Enter machine learning.

Abstracting the Data

The primary purpose of any model training / task undertaken in the last year has been to take noisy and unworkable scan data, and output clean and tractable data to be used in automated segmentation.

Hendrik’s post in the discord about using ceres for tracing blank regions prompted me to ask what would happen if he ran instead on my surface models instead of the scan data. It worked reasonably well on a patch he ran it on, we teamed up, and this became the focus of the next two months of models.

Initial training data was gathered by taking the .ppm file from existing grand-prize (and later on some of the additional exploratory segments created by David Josey), and mapping their 2d segmentation to 3d. The initial mapping was created first using a script created by Ryan Chesler for his efforts towards 3d ink detection, which i only slightly adapted to work instead with the segment mask.png (seriously, thats all i did for the first model). Later mapping was done using a script written by James Darby, which borrowed parts from Chuck’s ppm.py script in the khartes repository.



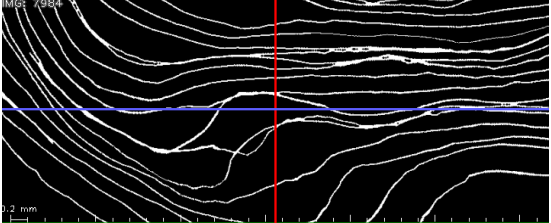
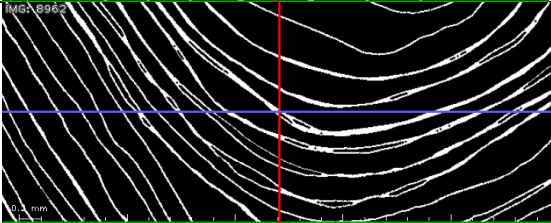
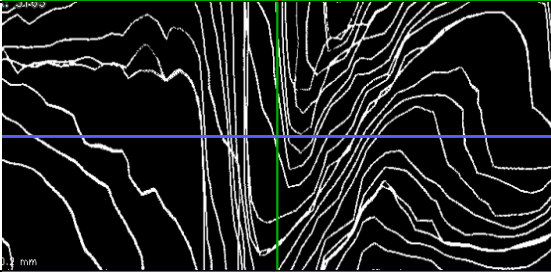
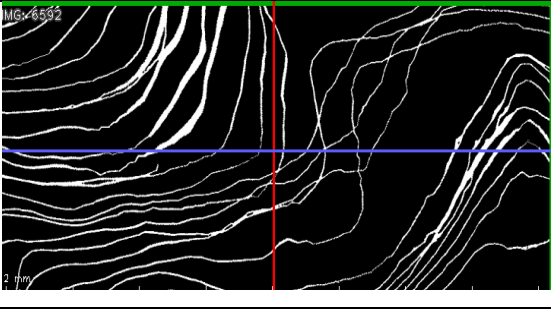
The 2d to 3d mapping works by taking for each point in the UV map, finding its respective zyx coordinates in the .ppm , and placing a point there. Then, for each layer in the mask (if multiple are desired), moving +1 or -1 along the surface normal. The result of

this process is a binary 3d zarr volume containing the segmentation lines as white pixels and the background as 0. The mapping process leaves several artifacts, mostly in the form of gaps and small holes, and this is addressed by using a 3x3 ellipsoid kernel and applying a morphological closing operation (James' repository contained a zarr focused closing operation but this did not yield as good of results for me).

Processing existing segmentations is a very fast way to obtain an abundance of training data, leveraging the hours spent already annotating these surfaces. There exist however many issues with this method,

- In almost all grand prize segments there is a not insignificant overlap with the segment that precedes it and the one that follows it. This creates confusing training data for a model. If, in most instances, I've essentially enforced separation between adjacent sheets encouraging the model to also enforce this, but here they intersect and flow around/within each other, the model now is penalized for enforcing this distance – resulting in a model that isn't entirely sure any longer what it is we want
- In areas of high curvature, OFS tends to flow off the sheet, sometimes a rather considerable distance. This is fine for ink detection because we have 64 voxels of play in our surface volume, it is much less workable when trying to train a model to create essentially a 4 voxel thick line representing the surface (and one that stays separated from others)
- Very rarely is there a densely labeled region of a large enough size to contain an entire extracted volume. This necessitates either manually masking this out with an ignore label, extracting a smaller sub volume, or dilating the label some amount to use as a mask (still requiring an ignore label). I had hoped that the last method would work, as it is easily automated, but the model somehow managed to just learn adjacency to the mask i believe. I'm not entirely sure, but it didnt perform well. Options 1 and 2 were most commonly used.
- The most valuable labels, the ones that are in areas that are tightly packed or highly curved, or worse yet – flip up and face the viewer in z – are also those that are mostly unusable as machine learning labels owing to innacurate segmentation and OFS causing adjacent sheets to overlap and cross over eachother and then back as the segmentor snaps the line back to the surface. This is not intended as a shot at the segmentation team, the fact that they were able to segment in these regions at all is a testament to their skill and dedication.
- The labels, being that we are attempting to automate segmentation of the same region, cannot contain too much of that region

- The labels contain ONLY that region, creating a dataset that is not particularly diverse

	Region of intersecting sections
	Region of significant overlap between nearby segments
	Label falling off the surface and multiple segments combing in a region with a “z-flip”
	Segments combining and overlapping

These issues were addressed in several ways. On the data side:

- Manually edited labels of regions I thought were good training regions but had bad labels. This was done primarily using Dragonfly3d.
- Arbitrarily angled bounding boxes were cut from the volume to take advantage of regions with good labels without resorting to an ignore label (which was not compatible with a method I'll discuss later).
- Once reasonable predictions were output from trained models, the softmax predictions were refined manually.
- Several complete run throughs of the data were performed, each pruning volumes found to contain poor labels. The raw number of training volumes here and the size of volumes themselves made it incredibly difficult to properly prune bad labels. I

went through every single one of my training labels at minimum 10 times, and each time found huge amounts of terrible training data containing either unlabeled regions that were not masked, regions of high curvature with poor segmentation, etc. If there is anything i have learned in this challenge it is that this stage is paramount to successful model training, and it happens to be the most painstakingly boring process imaginable.

On the training and architecture side:

- Relatively aggressive label smoothing was applied, as models without this tended to be overconfident and merge regions that should not be merged (likely because this type of thing was contained in the training data from me not properly pruning my labels)

The final training dataset contained 561 volumes typically sized around 220^3 , roughly half of which were from p.herc 1667, the other half being from p.herc Paris 4. The scroll 1 labels represent about 3.9% of the total GP volume, and some of the labels are from outside of the grand prize region.

Training and Model Configuration

The architecture used for all the models we ran through the solver was a 3d U-Net with residual encoder blocks, implemented through the nnUNetv2 framework using the (M) planner. The choice of nnUNet was a pragmatic one, as I'm by no means a machine learning expert, and nnUNet provides an immediately accessible framework for training state-of-the-art 2d and 3d semantic segmentation models (like the ones we need for this task). Its primary calling card is medical image segmentation, which has many parallels to our task in that the sensor which has collected the data is typically either CT – like the scrolls – or MRI. All-in, about 55 models were trained to detect either fibers or surfaces in 2d or 3d.

Almost every model had the same basic architecture as far as layers, number of features, patch size, kernel size, etc. This is due to the self-configuring nature of nnUNet, which selects these parameters based on a dataset fingerprint it extracts from the input data, and the target VRAM amount. I typically trained with the same or similar data, and mostly on my home setup which contains two rtx 3090s.

It must be stated before going to deep into this that very little quantitative performance analysis was conducted in any of these training runs, as the scarcity of good training data (not counting the effort required to obtain it) and the pace of the challenge made dedicating a significant amount of time in this direction a difficult choice to make. This is

an area I hope to focus on in the next year. It's also one that is unfortunately done very poorly even in the academic side of medical image segmentation, and thus it is incredibly difficult for a newcomer to this field to parse out what is good information, and what is going to send me down a 20 hour rabbit hole of time I cannot afford to lose.

Several different preprocessing steps were attempted, none of which had a detectable qualitative effect on model performance. On validation DICE score, these all performed *worse* than their non-preprocessed counterparts. I believe this is because the “noise” in a CT scan contains local information about orientation and density, and the model learns from these features.

- CLAHE only
- Non-local means denoising
- CLAHE + NLM denoising
- Sobel filtering

Normalization

Two normalization schemes were used, Normalization did not have a large effect on validation DICE, but later models exclusively used CT normalization.

- CT normalization, which uses global statistics (mean, std, and percentiles) calculated across all training cases' foreground regions to normalize each case, making it ideal for physical quantities like CT scans where absolute values matter.
- Z-score normalization, which normalizes each training case independently by subtracting its own mean and dividing by its own standard deviation, making it suitable when absolute values aren't meaningful or when inter-case standardization is desired.

Augmentation

Heavy augmentations were necessary throughout training due to the scarcity of training data, particularly in damaged and warped regions. All were wrapped in `RandomTransform`, so were called during data loading with the noted probability. The additional augmentations greatly improved performance in warped regions.

- The following nnUNet defaults were applied in all training runs:
 - Random Rotation (20%)
 - Random Scaling (20%)
 - Gaussian Noise (10%)
 - Gaussian Blur (20%)
 - Multiplicative Brightness (15%)
 - Contrast Adjustment (15%)
 - Simulation of Low Resolution (25%)
 - Gamma Modifications (10%)
- I added the following augmentations for most training (and all of the later ones). I borrowed most of these ideas from another MIC-DKFZ repository, from a project in which they segmented lung airways (<https://github.com/MIC-DKFZ/MurineAirwaySegmentation>)
 - Random Blank Rectangles (50%) -- these are just rectangles of image mean intensity
 - Inhomogeneous Lighting (25%) -- this transform simulates lighting effects that vary strongly across slices -- which seemed particularly useful for our scans that mirror this effect
 - Elastic Deformation (30%)

Loss Functions

Several loss functions were used, the final model trained used a compound loss of DICE/Cross-Entropy/Distance Adjusted, and the final ensemble used for our submission had DICE/Cross-Entropy/Distance/Skeleton-Recall

- Focal Loss + DICE-- attempting to force the model to learn harder regions, one was trained with a compound Focal and DICE loss. This model really struggled to converge and performed poorly.
- IoU/Jaccard loss + CE -- Model maintained better connectivity than DICE/CE alone, but struggled to generalize
- Skeletonization Recall Loss + CE -- (<https://github.com/MIC-DKFZ/Skeleton-Recall>) - "Skeleton Recall Loss operates by performing a tubed skeletonization on the ground truth segmentation and then computing a soft recall loss against the predicted segmentation output." This was a favorite of mine throughout the challenge. On its face, it would seem that computing a tubed skeletonize of a curvilinear *plane* would not be smart, and I would tend to agree, but it enhanced

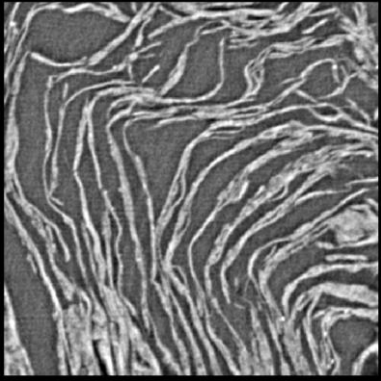
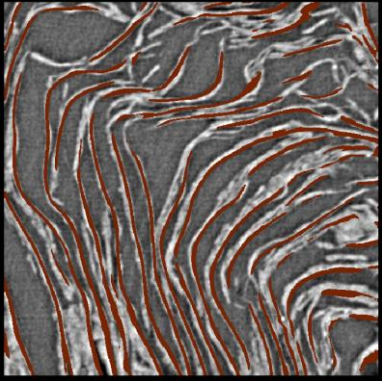
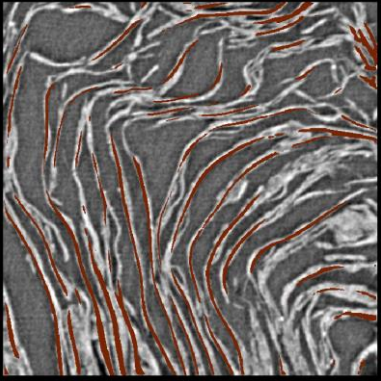
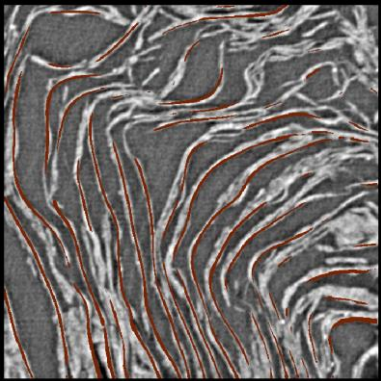
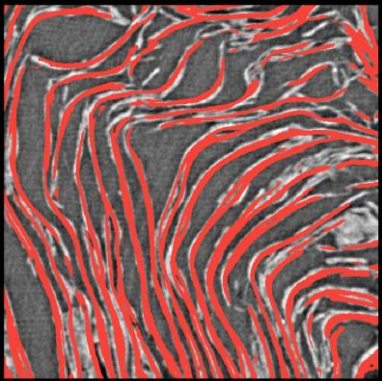
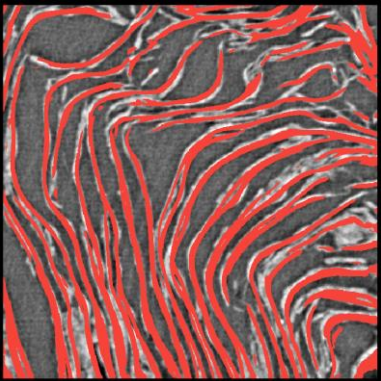
connectivity greatly, forcing the model to learn even in regions of great difficulty. It does come with a cost of greater tendency for the model to merge nearby surfaces.

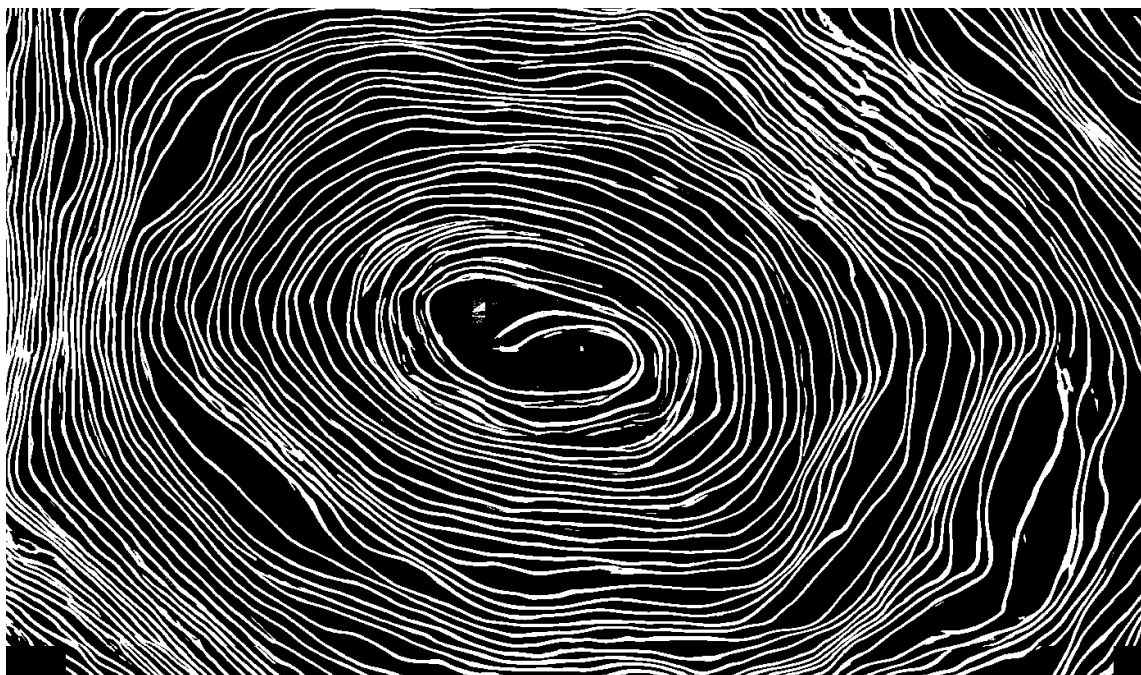
- Distance based Boundary losses – using a precomputing distance transform, loss is weighted according to the distance from the label, with a higher weight being placed on the region nearest to the boundary. This was another high performer. Enhanced boundaries between nearby fibers.

Postprocessing

The ensemble used for our final trace was created by taking the softmax predictions of 4 models. For each slice of each volume tif, a 9x9 sliding window computes the variance within it, and then weighs them from highest to lowest variance for each window. The slices from each are then blended, and the next slice is computed. The reason for this custom ensembling rather than the default nnUNet mean ensembling is because a simple arithmetic mean for each prediction results in over confident models dominating the ensemble, even if their respective predictions are poor. In this method a model that predicts a solid white block is penalized accordingly, because its prediction for our region is poor. Most of the time a large segmented region is desired, but for fine and detailed structures, this results in bad models ruining the ensemble. The resultant volumes from the two methods are very similar, but in regions where it counts most, this led to better results for us.

Ensembled predictions are not always better than single model predictions, however in this challenge each model i trained did better in some spots than others, and worse in some. Overconfident but frequently inaccurate models paradoxically performed well in areas that more conservative but significantly stronger models did not. By adaptively merging these and capping the downside of the “weaker” models, we were able to achieve a relatively thorough recto surface segmentation of the entirety of the scroll, allowing a surface optimization algorithm to effectively patch and trace the surface into a mesh.

		
Original	050 Extra Augs + Distance + DICE + CE	043 Extra Augs + Skeleton Loss + CE
		
Skeleton + CE	036, 043, 044	036, 043, 044, 050



Final Thoughts

As is always the case with machine learning, **the data is the bottleneck**. The models we used for this result are not great. They are trained on relatively poor data, and not a lot of it. Producing training data is the worst part of machine learning, but we need more of it. Much of it will likely come from the segmentations produced with this or other automated methods, as it typically more faithfully follows the surface than manual segmentation and has none of the limitations of view or exhaustion a human does. With this oncoming wave of data though, there must then be a wave of time spent pruning and adjusting this data, and ensuring its accuracy.

We have made so many mistakes along the way, and the poor early decisions made by me to not come up with more strict quantitative measures have resulted in difficult to measure progress, along with very hand-wavy “it seems better” results. There is no doubt that model performance has gotten better, but improving from here will require a framework from which to build upon it. Just a few simple yet unanswered questions we must find answers to :

- Do we need to scan the scrolls at such high resolution, or do we need to use such high resolution for our tasks?
 - This question left unanswered has significant unseen downsides. Given that the only current (fast) method of assessing segmentation quality is ink detection, and the only way to assess ink detection is to look at essentially a GP sized banner. Similarly, the only real way to know if a model’s performance will influence patch and trace quality is to test it on regions it performs poorly, but it cannot perform well without a dense prediction. This means that every single time we want to test a new loss function, preprocessing step, anything, we must perform segmentation on 754 volumes. On 8 rtx 4090s this takes an hour and a half to download and about 4 hours to infer at a cost of roughly 50 dollars if you manage to stop your instance immediately when its complete, but in most cases you’d like to run overnight, so you either eat an extra 50 and sleep 8 hours, or get 4 hours of sleep that night. Another hour and a half then is spent uploading then another hour to reconstruct, and if you’re working with someone else add their time for upload/download as well. The result of this is if you wish to improve a models performance with a change in some step, you must dedicate a minimum of 6 hours and just under 60 dollars, and likely that nights sleep.

- What are some ways to measure a “good surface” without ink detection or manual inspection? Are there geometric or distance checks that indicate what a “good result is?”
- What loss functions, augmentations, preprocessing steps are appropriate for our data? I don't believe I have a good answer to any of those yet
- Are we leaving anything on the table when it comes to ink detection?
 - Given that most contestants this year have chased automated segmentation, and because last year with the focus on ink detection for the challenge many contestants (including myself) did a poor job of gathering quantitative metrics, we still as far as I can tell have no real answers to the following:
 - Would true 3d labels help model performance? Is putting the same label through a 3d model for 20 layers training the model on the wrong features?
 - Are we potentially overfit to the grand prize region? What does a model trained much less ink from there, and more diversely throughout the scroll look like?
 - Does our patch size of 64x64 limit a model's ability to learn important features w.r.t ink detection? In almost any other semantic segmentation task, the preference tends to lean towards as large a patch size as possible
 - Is our ink detection model robust to segmentations from different approaches? (khartes, thaumato, etc)

We all must make a better effort this year to be better researchers -- even if it comes at a short term cost of time. My lackadaisical attitude towards it has already cost me much more time now when I need it most than it would have cost me in January.

I'm absolutely certain that 1-click whole scroll unrolling and ink detection are no longer the pipe dream of classicists and those of us who have been blessed enough to participate in this challenge, and I'm confident that with a bit more tuning we will have it very , very soon.

I'm honored to be in the same company as all of you, who know as well as anyone the sacrifices necessary to effectively make progress these past two years.

I'd also like to make a quick shoutout here to the nnUNet developers and MIC-DKFZ. The successes I've had in training these models is almost entirely accreditable to them. Their approach to data-centric and quantitatively reproducible machine learning is a shining light in a field that is absolutely filled to the brim with daily papers announcing the “newest latest SoTA architecture!” while hiding behind countless errors in measurement and reproducibility. Common are unfair comparisons w.r.t model size or training time, scarcely are extra training datasets or pretraining mentioned, and if they are somewhere

as a footnote. As an individual without a doctorate in machine learning (or a bachelors in anything for that matter), the field is nearly impenetrable due to an inability to even assess what is accurate information and what is not, and a tendency for the code accompanying the rare paper that decides to publish their code being difficult to set and run without a thorough understanding of their codebase. Many, many thanks to these guys for making machine learning approachable, reproducible, and most importantly – exceptionally accurate.