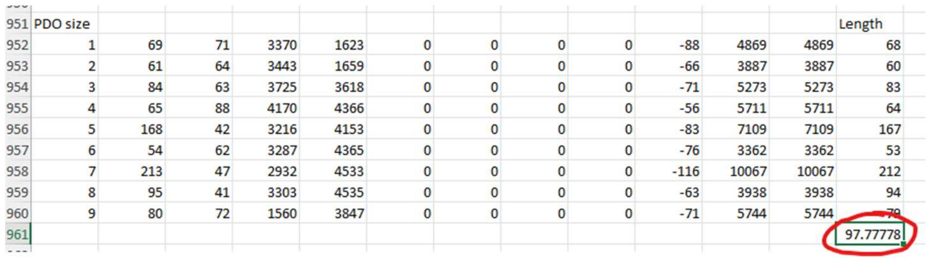
How to set up and run the code.

1. Download files from github: https://github.com/william-wei0/TIL-Analyzer
2. Download and put all data files into a new folder
3. Install packages by running
    a. pip install -r requirements.txt
    b. You will have to install pytorch separately from their website.
    c. https://pytorch.org/get-started/locally/
4. Run generateAnnotations.py to create the annotations for training
    a. Set image_dir to the folder with the TIL maps
    b. Set data_filepath to the clinical_data.csv
    c. Run generateAnnotations.py
5. Train a model by running nnet.py
    a. Check training results in "results" folder
    b. Ensure that there is a spread of survival curves in the training and validation set.
    c. Ensure that the predicted survival curves using chip images follows expected results.
6. Test the new model by running main_test.py
    a. Check the UMAP clustering images in results/Umap
    b. Look for good clustering results where each group is cleanly separated. Does not have to be perfect but at least mostly correct.
    c. Check results/metrics for specific survival curve values and look at p-values. In order to have good results, at least one time interval should have all three p-values be less 0.05
7. Only run generateImage.py if you need to recreate chip images from excel sheet. (the chip images are already included in the provided data folder)
    a. Set annotations_path to the annotated csv with the XY locations of each cell
    b. Set caf_concentration (from the name of the excel file). This is used for the filename of the resulting images.
    c. Set first_CART_concentration as the name of the CART concentration from the first datasheet.
    d. Run generateImage.py for each annotations file.
        i. *Make sure you change caf_concentration each time you run generateImage.py for different CAF densities, otherwise the old images will be overwritten.*

** A note about "annotations_chip_high_surv.csv", "annotations_chip_med_surv.csv", "annotations_chip_low_surv.csv". The first column is the path to the image. The second column is the average PDO size taken from the raw chip annotations file (see below). The third column is just to indicate that the "PDO device" has died and not be treated as censored data.



951	PDO size												Length
952	1	69	71	3370	1623	0	0	0	0	-88	4869	4869	68
953	2	61	64	3443	1659	0	0	0	0	-66	3887	3887	60
954	3	84	63	3725	3618	0	0	0	0	-71	5273	5273	83
955	4	65	88	4170	4366	0	0	0	0	-56	5711	5711	64
956	5	168	42	3216	4153	0	0	0	0	-83	7109	7109	167
957	6	54	62	3287	4365	0	0	0	0	-76	3362	3362	53
958	7	213	47	2932	4533	0	0	0	0	-116	10067	10067	212
959	8	95	41	3303	4535	0	0	0	0	-63	3938	3938	94
960	9	80	72	1560	3847	0	0	0	0	-71	5744	5744	79
961													97.77778

Nnet_survival_pytorch.py, model.py, CustomTransformations.py are utility files for setting the loss functions, setting the model size and structure and creating custom transformations such as the randomCircleCrop.

After running for the first time, run steps 5, 6 to retrain new models and test them. The goal is to get a good UMAP clustering image and get a low p-value when comparing survival proportions.

When I was running the code before, the “GoodUmapandGraph” model had a good survival curve separation until about interval 12, so I tried to get a good p value (less than 0.05) between the survival proportions around interval 12. Right now, one of the p-values is a little high at 0.12 and we were thinking we could get it lower with more samples. The UMAP results with that model were also okay but I think it could be improved.

Our current goals:

1. Generate a good clustering image with UMAP.
2. In metrics#.csv, find a time interval where all three survival proportions have a p-value lower than 0.05.
    a. Adding more samples would improve the UMAP because more samples make it easier to see clusters. It would also decrease the p-value because the p-value is inversely proportional to the number of samples.
3. Clean up the results code:
    a. Use only a single dataloader instead of SurvivalDataset and RawSurvivalDataset in main_test.py
    b. Rewrite the generate_survival_graph_noKMF, getUMAP_all, getUMAP_only_chip so that it doesn't repeat the same code 3 times for the low,med,high survival dataset and just uses a for loop for all the dataloaders.
    c. Rewrite generateImage.py to directly use the excel instead reformatting the excel sheet.

Below you will find some general explanations about the purpose of each function. Please look at the code directly to understand how they are implemented. Feel free to message me on Slack if you want clarification about certain lines.

nnet.py

Dataset Class

- **SurvivalDataset**
  Custom torch.utils.data.Dataset that loads medical image paths, applies transformations, and returns:
  - Image (tensor)
  - Survival label array
  - Survival time
  - Censoring status (alive/dead).

Data Loading

- **get_data(annotations_path)**
  Reads a CSV file with annotations and returns three lists:
  - Image file paths
  - Survival times
  - Event labels (1 = dead, 0 = censored).

Class Weight Calculations

- **get_weights_by_class(dataloader)**
  Iterates through dataset and counts how many censored vs. dead samples. Returns class weights (to balance training if classes are imbalanced).
- **get_weights_by_image(image_tensor)**
  Computes per-image weights based on how many pixels are non-black.
  - Mostly black images are weighted less because they have less information.
  - This is done because I want the training images to look like the chip images so I randomly crop them. But because they are randomly cropped, sometimes the picture is mostly black which has no information and should thus be weighted less.

Metrics & Results

- **save_results(train_loss, val_loss, train_accuracy, valid_accuracy, train_c_index, val_c_index, excel_filepath)**
  Saves training/validation metrics and concordance indices into an Excel file.
  If file already exists, appends new results.

Training

- **train_model(model, train_loader, val_loader, num_epochs, breaks, loss_fn, learning_rate, class_weights, save_dir, save_period=50)**
  Core training loop:
  - Loads batches of images/labels.
  - Computes custom weights (class & pixel-wise).
  - Optimizes model with AdamW + learning rate scheduler.
  - Every 10 epochs → evaluates model, saves metrics/plots, runs external test.
  - Saves best-performing models (by accuracy/loss).

Evaluation

- **calculate_c_index(dataloader, model, loss_fn, breaks)**
  Evaluates model on a dataset.
  - Computes loss and accuracy.
  - Uses **concordance index (c-index)** to measure how well predictions match actual survival order at different time points (3y, 2.5y, 2y, 1.5y, 1y).
  - Returns accuracy, loss, and c-index values.
- **generate_survival_graph_with_KMF(dataloader, model, epoch, dataset_name, save_dir="./results")**
  Plots survival curves:
  - Model predictions (different colored lines depending on survival probability).
  - Kaplan-Meier estimate (true survival curve in black).

    - Saves figure per epoch.

Main Execution

- **main()**
  Coordinates the pipeline:
  1. Defines save paths and hyperparameters.
  2. Loads annotations and splits train/validation sets.
  3. Creates survival arrays for training & validation.
  4. Builds DataLoaders with transformations.
  5. Initializes model, loss function, and class weights.
  6. Calls train_model() to start training loop.

main_test.py

Dataset Classes

- **SurvivalDataset**
  Custom PyTorch dataset that loads images and their *processed survival arrays* (created by nnet_survival_pytorch).
- **RawSurvivalDataset**
  Custom PyTorch dataset that loads images along with their *raw survival times* and *censoring status*.
- These datasets should really be one dataset class, but I didn't have time to rewrite the training functions. You can change this easily though. Just modify SurvivalDataset to also output the raw survival times and censoring and make it so that the for loop can handle the extra outputs.

Data Utility Functions

- **get_data(annotations_path)**
  Reads a CSV annotation file and extracts image filenames, survival times, and censoring events. Returns them as lists.
- **save_results(survival_times, excel_filepath)**
  Saves survival-related results into an Excel file. If the file already exists, appends the new data; otherwise, creates a new file.

Survival Analysis & Visualization

- **generate_survival_graph_noKMF(low_dataloader, med_dataloader, high_dataloader, model, save_dir)**
  Runs a trained model on three datasets (low, medium, high CAF groups).
  - Predicts survival curves.
  - Plots all individual curves and median/average survival curves.
  - Performs **log-rank tests** to compare survival distributions.
  - Saves results, plots, and statistics to csv in results/metrics/metrics#.csv

        - The first 15 lines are the survival curves for high CAF chip images (low survival)
        - The next 15 lines are the survival curves for med CAF chip images (med survival)
        - The last 15 lines are the survival curves for low CAF chip images (high survival)
        - The last 6 rows are the p-values calculated using a z-test of two proportions, comparing the survival proportion at each time interval from between each of the median/average survival curve of the three CAF densities. The excel cells to the far right of each row shows which groups are being compared.
    - The code for the 3 dataloaders can be reduced into a list of dataloaders and a for loop but I didn't have time to change it.
- **validate_during_testing(low_dataloader, med_dataloader, high_dataloader, model, epoch)**
  Similar to generate_survival_graph_noKMF, but designed for validation during model training/testing. Saves epoch-specific survival curves and prints log-rank p-values.

UMAP & Clustering Analysis

- **getUMAP_all(model, high_surv_dataloader, med_surv_dataloader, low_surv_dataloader, validation_dataloader)**
  Extracts feature embeddings from the model for all datasets (chip images+ training images).
    - Groups patients into categories based on survival time and censoring.
    - Runs **UMAP** for dimensionality reduction.
    - Plots 2D embeddings colored by group.
    - Applies **KMeans clustering** and elbow analysis to assess cluster quality.
    - The code runs a **UMAP** per num _neighbors and per minimum cosine_similiarity
    - You want to see 3 clear clusters for each CAF density

- **getUMAP_only_chip(model, high_surv_dataloader, med_surv_dataloader, low_surv_dataloader)**
  Runs UMAP only on chip datasets (high, med, low CAF).
    - Plots embeddings grouped by CAF condition.
    - Runs KMeans clustering with varying cluster numbers.
    - Saves UMAP visualizations and clustering results.

Training & Testing

- **main()**
  Full pipeline runner:
    - Loads datasets (chip + training).
    - Prepares dataloaders with transformations.
    - Loads pretrained models.
    - Generates survival curves and runs UMAP analyses.
- **test_on_chip(test_model, epoch)**
  Evaluates a model on chip datasets (high, medium, low CAF).
    - Loads annotations and datasets.
    - Prepares survival arrays and dataloaders for evaluation.
    - Calls validation/survival graph plotting functions.

CustomTransformation.py

Random Seed Setup

```
random.seed(seed)

torch.manual_seed(seed)

torch.cuda.manual_seed_all(seed)
```

Ensures reproducibility by setting the same seed for Python’s random, PyTorch CPU, and PyTorch CUDA random number generators.

CenterCircleCrop

```
class CenterCircleCrop(torch.nn.Module):

  def forward(self, img):

    ...
```

- Takes an input image and masks it with a **circle covering the whole image** (fits the largest possible circle within the image bounds).
- The background outside the circle is set to black.
- Output: an image where only the circular region is visible.

CenterCircleCrop2

```
class CenterCircleCrop2(torch.nn.Module):

  def forward(self, img):

    ...
```

- Similar to CenterCircleCrop, but instead of filling the entire image, it crops to a **user-defined radius circle centered in the middle** of the image.
- Useful when you want precise control of the circular crop size.
- Background outside the circle is black.

ImageTransformations

```
class ImageTransformations():
  def __init__(...):
    ...
```

- A wrapper class that defines different **image preprocessing and augmentation pipelines** for training, validation, and downstream tasks.
- Takes two arguments:
  - original_image_size: the starting size of the images (default 640).
  - cropped_image_size: the target cropped size (default 512).
- Has two possible modes:

1. **Using cropped size directly** (using_cropped_size_directly = True):
    - Resize image → Random crop → Circular crop → Random flips/rotations → Normalize.
    - Keeps final image size = cropped_image_size.
2. **Not using cropped size directly** (using_cropped_size_directly = False):
    - Similar, but resizes back to original_image_size after cropping.

- **Key attributes:**
  - train_transformations: Augmentation pipeline for training (resize, crop, circle crop, flips, rotations, normalize).
  - validation_transformations: Preprocessing pipeline for validation (resize + normalize, no heavy augmentation).
  - umap_transformations: Used for dimensionality reduction visualization (resize, center crop, normalize, but no augmentations).

Model.py

Identity

```
class Identity(nn.Module):
  def forward(self, x):
    return x
```

- A placeholder module that simply returns the input without changes.
- Used here to **replace ResNet18's final fully connected layer** so you can use the feature embeddings directly instead of classification logits.

DifferentResNet186

(I'm using this to test new models with different architectures)

```
class DifferentResNet186(nn.Module):
```

- A **custom ResNet18-based model** for survival prediction.
- Architecture:
    1. **Feature extractor**: ResNet18 backbone (pretrained), with final FC layer replaced by Identity.
    2. **Two hidden fully connected layers** (fc1, fc2) each followed by:
        - Sigmoid activation
        - Batch normalization
        - Dropout (for regularization)
    3. **Output layer**: survival → predicts values for num_intervals (time intervals in survival analysis).
    4. Final **Sigmoid** ensures outputs are in [0,1].
- Summary: ResNet18 → two-layer MLP → survival probabilities per interval.
- More complex than the second model, allows deeper feature transformation.

GoodUmapandGraph

(This is the model of the 'epoch 75.pt' file and has good results)

class GoodUmapandGraph(nn.Module):

- A **simpler ResNet18-based model** for survival prediction.
- Architecture:
    1. **Feature extractor**: ResNet18 backbone with Identity replacing the final FC layer.
    2. **One hidden fully connected layer** (fc) with:
        - Sigmoid activation
        - Batch normalization
        - Dropout
    3. **Output layer**: survival → predicts values for num_intervals.
    4. Final **Sigmoid** outputs probabilities.
- Summary: ResNet18 → single-layer MLP → survival probabilities per interval.
- Lighter and faster, with fewer parameters than DifferentResNet186.

Key difference:

- **DifferentResNet186** has **two intermediate fully connected layers**
- **GoodUmapandGraph** has **only one intermediate fully connected layer**

generateImage.py