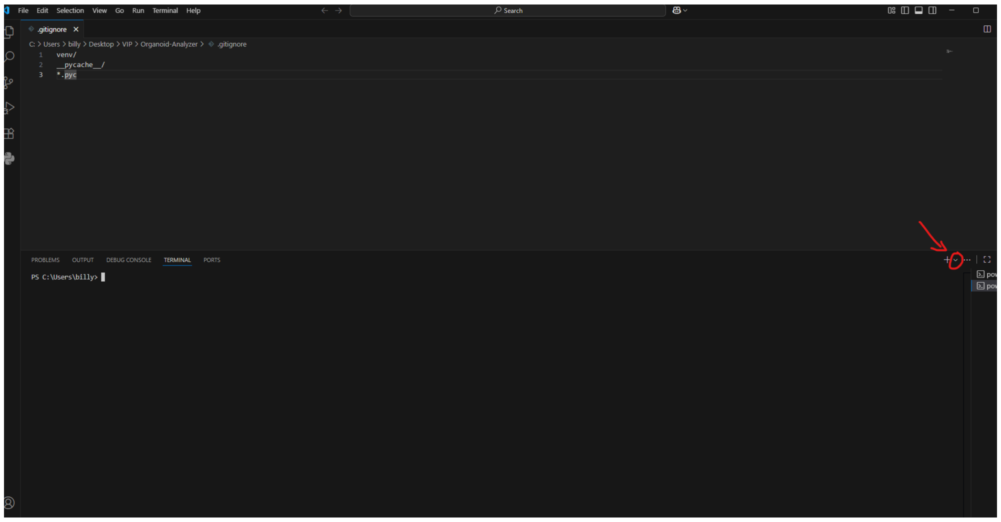
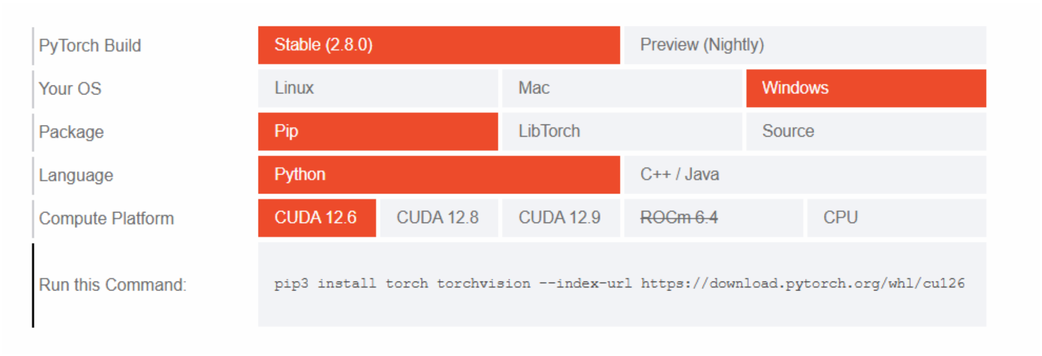
Before running the project:

1. Download files from github https://github.com/william-wei0/Organoid-Analyzer
    a. <> Code > Download ZIP
2. Create a new folder with all of the python files

3. Install python 3.10.11 https://www.python.org/downloads/release/python-31011/
4. Create a new virtual environment:
    a. First, open VSCode
    b. File > Open folder (locate the folder with the python files)
    c. Terminal > New Terminal > dropdown arrow on the right of “powershell” > Command Prompt
    d. 

```
venv/
__pycache__/
*.pyc
```

```
PS C:\Users\billy>
```

    e. In the terminal: python -m venv venv
    f. In the terminal: venv\Scripts\activate
    g. In the terminal: pip install -r requirements.txt (This installs all packages except for pytorch)
    h. Go to pytorch website https://pytorch.org/get-started/locally/
    i. 

PyTorch Build	[✓ Stable (2.8.0)]	Preview (Nightly)			
Your OS	Linux	Mac	[✓ Windows]		
Package	[✓ Pip]	LibTorch	Source		
Language	[✓ Python]	C++ / Java			
Compute Platform	[✓ CUDA 12.6]	CUDA 12.8	CUDA 12.9	~~ROCm 6.4~~	CPU
Run this Command:	`pip3 install torch torchvision --index-url https://download.pytorch.org/whl/cu126`				

    j. If your computer can run CUDA, (i.e. is your graphics card an NVIDIA GPU?)

    you should be able to use the default CUDA on the left mostside (CUDA 12.6 at the time of writing)
    k. If your computer cannot run CUDA (i.e. your graphics card not an NVIDIA GPU), then select CPU
    l. Copy the command given on the website and run in the terminal to install pytorch.
    (e.g pip3 install torch torchvision --index-url https://download.pytorch.org/whl/cu126 is the current version at the time of writing)
    m. The virtual environment (venv) is now set up and ready to run the code.
5. To run each file, type “python filename.py” in the terminal
6. *(Keep in mind that the venv must be active to run each file, otherwise the code doesn’t know where to look for packages.)*

In order to run this project, the expected workflow is:

1. Setup configs in Config.py
2. Perform cell tracking for each new set of images with track_cells.py
3. Define how to read the annotations file with load_annoations() in create_dataset.py
4. Create the training/testing dataset with create_dataset.py
5. Set up the train/test annotations excel sheet to define training and test cases
6. Perform multiple model training and testing with train_and_test_models.py

Below are summaries/general ideas of what each file does and what each function does.

Config.py

======= CELL TRACKING SETTINGS =======

IMAGES_FOLDER: This is the folder where the timelapse images are stored

FIJI_PATH: This is the folder with Fiji in it. This is not the path to the app itself (only the folder).

CASE_NAME: This is the name of the output folder used to store the raw tracking results in the DATA folder.

JAVA_ARGUMENTS: Sometimes for large timelapse images, more memory must be allocated to Fiji through Java. Java arugments allows us to increase the memory size for Fiji as well as other Java arguments if needed.

======= DATASET GENERATION SETTINGS =======

After tracking cells, then a training dataset can be generated.

SEQ_LEN: Number of frames to use.

DATASET_CONFIGS: For each new batch of images processed, to create a new dataset, you must let the program know where to find the excel sheet containing the labels (0, 0,5, 1.0) and which folder contains the raw data (spots.xlsx/track.xlsx) extracted from trackmate. The data folder has the same name as the CASE_NAME set in configs when the tracking was done.

The format of DATASET_CONFIGS is a dictionary where each CASE_NAME is associated with a subdictionary containing keys for “annotation_path” and ”data_folder”.

For example, to create a new dataset using CART images, an example of DATASET_CONFIGs could be:

DATASET_CONFIGS = { "CART": {"annotation_path" : f"{DATA_DIR}/CART annotations.xlsx", "data_folder" : f"{DATA_DIR}/CART"} }

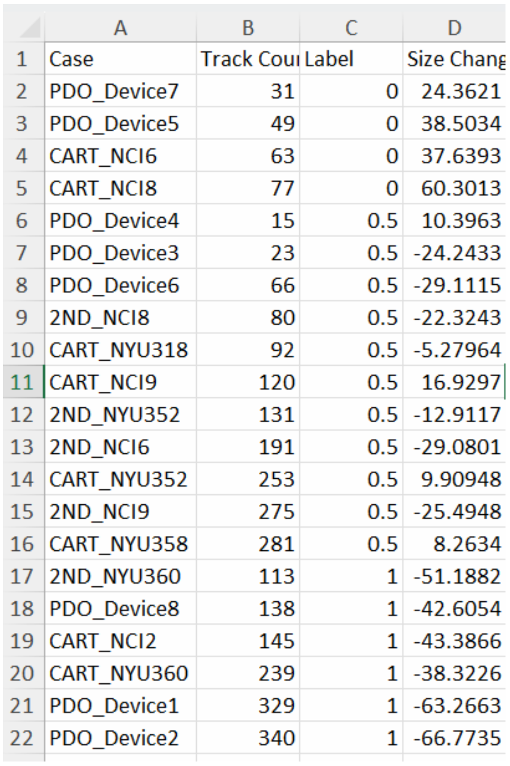
SEQ_DATASET_PREFIX: Only change if you want to change the name of the output sequential dataset files.

TRACK_DATASET_PREFIX: Only change if you want to change the name of the output sequential dataset files.

Features, track_features: This is a list of the features that will be computed for the new dataset. If you want include or exclude certain features when making a new dataset, change it here.

======= TRAINING SETTINGS =======

TEST_TRAIN_SPLIT_ANNOTATION_PATH: In order to define which cases are used for training and which are for testing, an excel sheet is used to allow for easy modification of the train and test cases. This path is to that annotations file. The annotations file must follow the same format as image below.



	A	B	C	D
1	Case	Track Cou	Label	Size Chang
2	PDO_Device7	31	0	24.3621
3	PDO_Device5	49	0	38.5034
4	CART_NCI6	63	0	37.6393
5	CART_NCI8	77	0	60.3013
6	PDO_Device4	15	0.5	10.3963
7	PDO_Device3	23	0.5	-24.2433
8	PDO_Device6	66	0.5	-29.1115
9	2ND_NCI8	80	0.5	-22.3243
10	CART_NYU318	92	0.5	-5.27964
11	CART_NCI9	120	0.5	16.9297
12	2ND_NYU352	131	0.5	-12.9117
13	2ND_NCI6	191	0.5	-29.0801
14	CART_NYU352	253	0.5	9.90948
15	2ND_NCI9	275	0.5	-25.4948
16	CART_NYU358	281	0.5	8.2634
17	2ND_NYU360	113	1	-51.1882
18	PDO_Device8	138	1	-42.6054
19	CART_NCI2	145	1	-43.3866
20	CART_NYU360	239	1	-38.3226
21	PDO_Device1	329	1	-63.2663
22	PDO_Device2	340	1	-66.7735

SEQ_DATASET_PATH, TRACK_DATASET_PATH: Only change if you want to change the path to a specific dataset instead of the default one.

DROPOUT: Amount of dropout to use. Used to reduce overfitting by deleting a percentage of neurons weights (e.g 30%)

MAX_EPOCHS: We use early stopping to prevent overfitting. This value is just the maximum number of epochs to use increase early stopping doesn't happen.

BATCH_SIZE: What batch size to use. Usually higher batch sizes result in faster training but require more memory.

MIN_POW_FUSION, MAX_POW_FUSION: Instead of training one model at a time, the program trains several programs with varying sizes each incrementing by a power of 2.

These define the upper and lower bounds of the model sizes of the Linear layer. For example, using 2 and 10 would train model sizes of 4, 8,16, 32, 64,128, 512, 1024.

MIN_POW_HIDDEN, MAX_POW_HIDDEN: Similar to the fusion, this modifies the hidden size of the LSTM.

ABLATION_CONFIGS: If you want to try training models with a different set of features but don't want to create a new dataset ABLATION_CONFIGS allows you to create a dictionary with the test name as the key and the features/trackfeatures as the values.

Track_cells.py

This script processes time-lapse image folders using Fiji's TrackMate plugin to automatically detect and track objects. It first checks whether an output folder exists and handles duplicates by either deleting the folder and creating a new one or exiting the code. Fiji is then launched and TrackMate initialized, after which each image folder is loaded and converted into a hyperstack suitable for tracking. The program sets detection and tracking parameters, calculates an automatic threshold, and attaches analyzers to extract both spot-level and track-level features. TrackMate is executed to perform the tracking, and the resulting measurements for tracks and spots are collected. Finally, the results are saved as CSV files in the designated output folder for further analysis.

Check if output folder can be created

This section checks whether a folder with the given case name already exists. If it does, the user is asked whether to delete the old folder or stop the program to avoid overwriting results. If the user chooses deletion, the folder and its contents are removed and recreated. If the folder does not exist, it is simply created to store the new outputs.

Launch Fiji with TrackMate

This part starts Fiji with the TrackMate plugin using jpype and imagej. It sets Java options, initializes the Fiji application, and imports the necessary TrackMate Java classes that will be used later in the workflow.

Iterate through image folders

For each subfolder inside the main images directory, the code checks that the folder exists, then loads the image stack using Fiji's FolderOpener. The stack is converted into a hyperstack so that Z-slices are treated as time frames, making it suitable for time-lapse tracking.

Initialize TrackMate settings and auto threshold

In this section the tracking model and settings are prepared. A new TrackMate Model and Settings object are created, and the image intensity threshold is automatically calculated using Otsu's method. The code then configures the detector for spot detection and sets the tracker parameters for linking objects across frames, including distance thresholds and rules for splitting and merging tracks.

Add spot and track analyzers

Here the script attaches additional analyzers to TrackMate. Spot analyzers measure shape, intensity, and contrast for each detected spot, while track analyzers compute features such as speed, branching, duration, and spatial location for each track.

Run TrackMate and validate

The configured TrackMate instance is then executed. The input is first checked for validity, and if the processing fails, an error message is displayed and the program skips to the next folder.

Extract track-level features

This section gathers numerical descriptors for each track, such as the number of detected spots, the track duration, displacement, mean speed, distance traveled, and other statistical features. Each track's data is stored as a row in a growing list.

Extract spot-level features

For each detected spot, the script records measurements such as its position, radius, intensity values, and shape characteristics. If a spot is linked to a track, its track ID is included as well. All spot information is stored in a list of rows for later export.

Save results

Finally, the collected track and spot data are converted into pandas DataFrames. The track features are saved as one CSV file and the spot features as another. Each pair of CSV files is named according to the subfolder being processed and written into the designated output folder.

Create_dataset.py

save_unscaled_spot_features
Takes the spot-level dataframe, extracts the important unscaled features, and saves them into a CSV file for later use.

save_unscaled_track_features
Takes the track-level dataframe, merges it with label information from the datasets, extracts unscaled track features, and saves them into a CSV file.

load_annotations
Reads the annotation Excel file for a dataset, depending on the dataset type (CART, 2ND, PDO, CAF), extracts IDs and labels, and builds a mapping between sample IDs and their labels.

load_tracks_and_spots
Loads track and spot CSV files for each dataset. It matches them with the correct labels from annotations, formats the data, and combines everything into two dataframes: one for tracks and one for spots.

filter_valid_trajectories
Filters out trajectories that are too short by keeping only those tracks that last at least a minimum number of frames.

compute_msd
Given X and Y positions of a trajectory, computes the mean squared displacement (MSD) for different time lags.

compute_features
Calculates new features for each spot, such as velocity, speed, direction, and mean squared displacement. It also cleans up the dataframe by removing unnecessary columns and handling missing or invalid values.

align_and_save_dataset
Aligns trajectory data into fixed-length sequences. Short sequences are padded with zeros, and long ones are trimmed. The aligned data is saved as both a .npz file (for model training) and a .csv file (for inspection).

build_track_level_dataset
Creates a dataset from track-level features. It attaches labels, collects the features, and saves the dataset in both .csv and .npz formats.

filter_outer
Removes invalid spot trajectories based on shape features, for example, if ellipse size is zero or the aspect ratio is unrealistic. It removes entire bad tracks and reports how many were discarded.

Train_fusion_model.py

SubsetDataset class
This defines a PyTorch dataset that loads and organizes sequence data, track data, and identifiers for specific cases. It uses the helper function select_specific_cases to filter

out only the cases you want and makes them accessible in a way that can be used by PyTorch's DataLoader.

select_specific_cases
This function loads sequence and track data along with annotations, and then selects only the samples belonging to a chosen case (for example, “train” or “test” cases). It aligns sequence and track features by their track IDs, and returns the matched features, labels, and identifiers.

train_test_split_by_case
This function takes the entire dataset and splits it into training and testing sets based on case annotations. It matches sequence and track data by track IDs, organizes them into train/test groups, and prints how many samples ended up in each. It also keeps track of label distributions for debugging.

run_inference
This runs the model in evaluation mode and calculates probabilities for each class, and returns the predicted class labels.

Train_UnifiedFusionModel
This is the main training function. It loads and splits the data, encodes labels, prepares PyTorch datasets and dataloaders, builds the unified fusion model, and trains it over several epochs. It monitors validation performance, uses early stopping, and saves the best model. After training, it generates graphs of accuracy/loss, evaluates the model on train/val/test sets, computes metrics like accuracy, F1, and AUC, performs case-level analysis, and saves everything to disk.

Test_UnifiedFusionModel
This function loads a pre-trained model and evaluates it on a test dataset. It generates accuracy, F1, AUC scores, and confusion matrices, and produces visualizations such as ROC curves. It also performs case-level analysis and correlates predictions with size-change data.

train_models_and_shap

This function automates running multiple training experiments (ablation studies). It iterates over different model configurations and hidden/fusion sizes, trains models for each setting, and optionally runs SHAP analysis to interpret feature importance. It collects and saves performance results (train, val, test accuracy, loss differences, R^2 scores) into structured CSV files for later comparison.

Shap_analysis.py

load_and_align_data
This function loads two .npz files, one containing sequence data and the other containing track data. It extracts the features, labels, and track IDs from both sources and aligns them by matching track IDs. If the sequence data has a specific shape, it transposes it to the expected format. Finally, it returns the matched sequence data, track data, labels, and a list of combined prefix/track IDs.

SHAP_UnifiedFusionModel
This function runs SHAP analysis on a trained Unified Fusion Model, which combines sequence and track features. It first loads the model and the data, splits the data into training and test sets, and defines a wrapper to make the model compatible with SHAP. It then uses SHAP to compute feature importance values for both sequence and track

features. The function saves results as CSV files and creates multiple plots: bar charts of signed and absolute feature importance (including time-summed versions) and a SHAP summary plot.

Results_Utils.py

This is just a utilities file with the functions to create the result graphs

compute_metrics

This function calculates evaluation metrics for classification. It computes accuracy, macro F1-score, and the ROC AUC score (if probabilities are provided). It also converts true labels into a one-hot encoded format for use in ROC analysis.

plot_roc

This function plots ROC curves for each class in a multi-class classification problem. It shows how well the model separates classes by comparing true positive rates and false positive rates, and saves the plot as an image.

plot_confusion_matrix

This function creates and saves a confusion matrix heatmap. It shows how many samples were correctly or incorrectly classified for each class, helping to visualize errors.

fusion_weight_analysis

This function tests how the accuracy of the model changes when adjusting the balance (weight) between two inputs: sequence features and track features. It evaluates performance across a range of weights and saves a plot of accuracy vs. weighting.

compute_case_proportions
This function calculates the distribution of predicted classes for each case in the dataset. It groups results by case ID, computes proportions of each class (Progressive, Stable, Responsive), and saves a stacked bar plot of class proportions.

correlate_with_size_change
This function compares predicted scores with actual experimental size change data. It fits a line to the relationship, calculates the R^2 value (goodness of fit), and saves a scatter plot with the best-fit line.

plot_loss_curve
This function plots training and validation loss over time, showing how well the model is learning. The curve helps detect overfitting or underfitting, and the result is saved as an image.

plot_accuracies
This function plots training, validation, and test accuracies over time, showing how performance changes during training. It saves the accuracy curve as an image.

UnifiedFusionModel.py

1. Attention Module

- **Purpose:** Learns to focus on the most relevant time steps in the LSTM output.
- **How it works:**
    1. Applies a linear layer (nn.Linear(hidden_dim, 1)) across the hidden states of the sequence.
    2. Uses a softmax to turn the scores into normalized attention weights.
    3. Applies dropout to the weights for regularization.
    4. Computes a **context vector** as the weighted sum of LSTM outputs.

This lets the model selectively emphasize important parts of the input sequence.

2. Sequence Encoder (BiLSTM + Attention + LayerNorm)

- The sequence input (x_seq) goes through a **bidirectional LSTM**. This captures both past and future context in the sequence.
- The LSTM outputs are normalized (LayerNorm) to stabilize training.
- The **Attention module** reduces the variable-length sequence into a fixed-size vector (lstm_feat).
- A final **layer normalization** is applied again before weighting the LSTM features.

3. Track Encoder (Optional)

- If additional **track-level features** are provided (x_track), they go through a fully connected network:
    1. Linear → ReLU → Dropout → Linear
- This maps track features into the same dimension as the LSTM output (hidden_size*2).
- If no track features are available, the model simply skips this branch.

4. Fusion

- The sequence and track features are **combined**:
    - The model scales them with a tunable factor (lstm_weight vs 1 - lstm_weight).
    - They are **concatenated** into a fused representation.
- If no track features exist, the fused vector is just the LSTM representation.

5. Classifier Head

- A fully connected block processes the fused features:
    - Linear → ReLU → Dropout → Linear
- The output dimension is 3, meaning this is set up for a **3-class classification task**.