Multicell-fold Project 3

Jiahui Zhu

November 2024

Contents

1 Introduction

This project aims to replicate the findings presented in the paper titled *Multicell-Fold: Geometric Learning in Folding Multicellular Life* by Yang et al. [2]. The primary objective is to validate whether the geometric deep-learning workflow described can accurately predict local cell-cell junction rearrangements. The dataset employed for this purpose was initially generated for the project *Deconstructing Gastrulation at Single-Cell Resolution*, carried out at Princeton University in 2021 by Tomer Stern, Stanislav Y. Shvartsman, and Eric F. Wieschaus [1].

The raw data for this study were directly sourced from the aforementioned publication, which also provides access to some preprocessing scripts [1]. I utilized specific functions from the provided code package, namely `DG_load_segmented_embryo.m` and `DG_calc_cell_areas.m`, and developed additional MATLAB scripts to extract critical features as described in the proposed model architecture. All codes utilized in the analysis are available in the GitHub repository detailed in the Code Availability section.

2 Overview of Dual Graph Structure proposed in Multicell-fold Paper

In the study of *Drosophila* developmental dynamics, a dual graph data structure that categorizes the information into three key types: cell geometries, cell-edge geometries, and dynamics information was introduced. Ablation studies indicate that removing any one of these types significantly degrades model performance, while using any single type as the sole predictor fails to yield satisfactory results.

The initial data representation included two types of graphs: the cell-cell adjacency graph and the cell-edge graph. These featured two kinds of nodes—cells and cell junctions (vertices)—and three kinds of edges: cell-cell, cell-vertices, and vertices-vertices.

In an effort to optimize the data representation, they identified an efficient representation that focuses on the concatenation of cell-edge information instead on cell-cell adjacency graph, leveraging the mathematical properties of duals. This modified approach retains only the features that most significantly impact performance and reduces the dataset to one-third of its original size. Although this streamlined representation performs marginally worse, it largely maintains the predictive quality of the model.

2.1 Efficient Representation

- **Nodes (N)**: Each node represents a cell, denoted as $\{C\}$. The nodes are embedded with features that include the area a, the perimeter p, and their respective general displacements ($\dot{a}$, $\dot{p}$) between consecutive frames.

- **Edges (E)**: The edges connect pairs of cells, indicated as $\{e_{c-c}\}$, where each edge represents a relationship (junction) between two cells. The edges are attributed with features such as the exponential of the negative edge length $l^*_{v-u} = \exp(-l_{v-u})$. This transformation is used to emphasize shorter edges, presumably to prioritize interactions between closely positioned cells. In addition to the edge length feature, the difference in the edge lengths between frames ${l^*_{v-u}}'$ is also included. This feature captures the dynamic changes in the spatial relationships between cells.

- **Criteria for Identifying the Loss of Cell Junctions** The loss of cell-cell junctions $\{R\}_t$ is computed as the difference between $\{e_{c-c}\}_t$ and $\{e_{c-c}\}_{t+1}$,

$$\{R\}_t = \text{diff}(\{e_{c-c}\}_t, \{e_{c-c}\}_{t+1}),$$

  meaning that a pair of cells lose the shared cell-cell junction if it appears as a neighboring pair in frame t but not in frame $t+1$.

3 Methodology and Analysis

3.1 Data Preprocessing

Data preprocessing includes computing node and edge features, as well as determining junction loss between nodes as a target, to align with the specific requirements of the model architecture.

3.1.1 Loading Data

- **Embryo 1620 (Intercalations):** This dataset represents an embryo undergoing intercalation events and is used for training purposes.
- **Embryo 1830 (Divisions):** This dataset represents a different embryo, characterized by cellular division events. It is utilized as the test dataset to evaluate the robustness and predictive power of the models trained on the Embryo 1620 dataset.

For each embryo, mesh data representing cells and vertices were loaded:

```
[C_1, E_1, V_1] = DG_load_segmented_embryo(embryo1)  % Embryo 1620
[C_2, E_2, V_2] = DG_load_segmented_embryo(embryo2)  % Embryo 1830
```

These arrays, C, E, and V, represent cells, edges, and vertices, respectively.

3.1.2 Feature Calculation

Cell Features:

- **Areas:**

```
areas_1 = DG_calc_cell_areas(C_1, V_1)
areas_2 = DG_calc_cell_areas(C_2, V_2)
```

- **Perimeters:**

```
perimeters_1 = calc_cell_perimeters(C_1, V_1)
perimeters_2 = calc_cell_perimeters(C_2, V_2)
```

Edge Features:

- **Edge Lengths:**

```
edge_lengths_1 = calculate_edge_lengths(C_1, E_1, V_1)
edge_lengths_2 = calculate_edge_lengths(C_2, E_2, V_2)
```

Loss of Junction (target):

```
junction_loss_1 = calculate_junction_loss(E_1)
junction_loss_2 = calculate_junction_loss(E_2)
```

3.1.3 Constructing Graphs

With node and edge features extracted, the next step involved constructing graph representations for each developmental stage. These graphs integrate both spatial and temporal attributes of cellular structures:

- **Node Features:** Calculated areas and perimeters for each cell at different time points, including temporal changes (delta area and delta perimeter). These features are stored using:

```
node_features_1 = save_node_features(C_1, areas_1, perimeters_1)
node_features_2 = save_node_features(C_2, areas_2, perimeters_2)
```

- **Edge Features:** Calculated edge lengths along with temporal changes in these lengths (delta weighted edge length), The features are saved using:

```
edge_features_1 = save_edge_features(E_1, edge_lengths_1)
edge_features_2 = save_edge_features(E_2, edge_lengths_2)
```

- **Graph Construction:** All extracted features, including junction loss data, are then concatenated into comprehensive graph structures for each developmental stage:

```
save_graph(node_features_1, edge_features_1, junction_loss_1, 'train')
save_graph(node_features_2, edge_features_2, junction_loss_2, 'test')
```

The 'train' and 'test' designations for the graphs indicate their use in training predictive models and evaluating their performance, respectively.

3.1.4 Output Graph Data Structure

The dataset comprises two distinct sets of graphs corresponding to the developmental stages of the embryos: the train_embryo, which includes 197 timepoints, and the test_embryo, which encompasses 59 timepoints. Each timepoint is encapsulated as a subgraph within the dataset, organized into four primary components:

Time: Each subgraph is associated with a specific timepoint, labeled as *time*.

Node Features: It is structured as $[cell\_idx, area, \Delta area, perimeter, \Delta perimeter]$, where Δ denotes the change relative to the **previous** timepoint

Edge Features: It is structured as $[cell1, cell2, edge\_idx, weighted\_length, \Delta weighted\_length]$.

Junction Loss Data: This is formatted as $[edge\_index, binary\_variable]$, with the binary variable indicating a loss (1) or no loss (0) compare to **next** timepoint

3.1.5 Code Availability

The code used for analysis is available at: GitHub Repository. For detailed descriptions of all MATLAB functions used , please refer to the supplementary material section. **Note:** Functions beginning with `"DG_"` are sourced directly from the paper [1].

3.2 Exploratory data analysis

3.2.1 Removal of Nodes with Zero Features

Upon examination, a significant number of nodes were found to have zero features (i.e., all feature values are zero). Specifically, there were 14,070 nodes with zero features in the training graph and 12,638 in the testing graph. These nodes were removed to maintain data consistency. Furthermore, node indices were remapped to ensure consistency, and the indices in the edge features were also updated accordingly.

Additionally, tests were conducted to determine whether there were any edges connecting nodes with zero features. The results confirmed that no such edges exist.

3.2.2 Data Statistics

Table 1: Statistical Overview of Training and Testing Graphs

Graph	Parameters	Values
Training Graph	Number of Nodes	6000-7000
	Number of Edges	10000-17500
	Percentage of Positive Targets (Junction Loss)	0.5%-4%
	Number of Time Points	198
Test Graph	Number of Nodes	5000-8000
	Number of Edges	10000-16000
	Percentage of Positive Targets (Junction Loss)	1%-8%
	Number of Time Points	60

3.2.3 Visualization of Feature Distribution

- **Area, Perimeter, Edge Length, and Delta Edge Length:** These features exhibit right-skewed distributions, indicating that the majority of the data points are concentrated on the left side of the distribution, with a long tail to the right.
- **Delta Area and Delta Perimeter:** These distributions are more centralized and resemble a normal distribution, suggesting a symmetric spread of data around the mean.
- **Distribution of Positive Targets:** Upon examination of the distribution of positive targets and comparison with the original distributions, it is observed that they mostly follow a similar trend with no distinctive deviations noted.

3.3 Model Architecture

3.3.1 Graph Autoencoder Architecture

The Graph Autoencoder uses a two-component approach: the Graph Transformer Encoder and the Inner Product Decoder.

3.3.2 Graph Transformer Encoder

The encoder utilizes the TransformerConv layer from the PyTorch Geometric library. The architecture includes an initial TransformerConv layer that manages node features and edge dimensions with parameters aligned with those in the cited paper: hidden dimension size $c_{\text{hidden}} = 24$, number of layers num_layers $= 6$, number of attention heads heads $= 4$, and dropout rate dropout $= 0.5$. Each TransformerConv layer in the sequence is followed by a LayerNorm layer, ensuring consistent normalization.

3.3.3 Inner Product Decoder

The decoder component computes the pairwise inner product of the node embeddings to predict edge scores, reconstructing the graph's adjacency matrix. The activation function used is sigmoid, ensuring that the output probabilities of edge existence range between 0 and 1.

3.3.4 Operational Flow

The model's operational flow begins with the encoder transforming the input graph's node features and edge attributes into a set of latent space embeddings. These embeddings are then utilized by the decoder to reconstruct the graph's edges.

3.4 Training

The models were trained using a learning rate of 5×10^{-4} and a weight decay of 1×10^{-8}. Training was conducted over 700 epochs.

3.4.1 Regularization Term

It is defined as:

$$L_{\text{regularization}} = \left(\epsilon - \frac{N_{\text{pos}}^{(c)}}{N_{\text{tot}}^{(c)}} \right)^2$$

where ϵ represents the average predicted probability of cell rearrangement across the entire batch. This average serves as a benchmark for expected positive outcomes. Here, $N_{\text{pos}}^{(c)}$ is the number of rearranged cell pairs identified as positive, and $N_{\text{tot}}^{(c)}$ is the total number of cell pairs in the batch.

3.4.2 Loss Function

The total loss function, L_{total}, combines the binary cross-entropy loss, L_{BCE}, with the regularization term:

$$L_{\text{total}} = L_{\text{BCE}} + L_{\text{regularization}}$$

4 Results and Discussion

4.1 Comparison of Confusion Matrices

The performance of the model in predicting cell rearrangement in the *Drosophila* embryo is quantitatively assessed through confusion matrices. The reproduced results indicate a precision of 95.53% for correctly predicting cells that remain static ('Stay') and a precision of 26.53% for cells that undergo rearrangement ('Rearrange'). In contrast, the results reported in the original paper show a higher accuracy for the 'Rearrange' predictions at 82.67%, but a similar accuracy for the 'Stay' predictions at 83.37% (Figure 1 and Figure 2). This discrepancy highlights a potential overfitting issue or differences in the model configuration or dataset between the original study and the reproduction attempt.

In the reproduced results, the ratio of correct predictions for 'Stay' cells is consistently high, while the 'Rearrange' predictions reveal considerable fluctuations, with a large proportion of false positives.

4.2 Accuracy, Precision, Recall and F1-Score

The analysis reveals a relatively high accuracy of 91.8%, which suggests that the model performs well overall in distinguishing between the 'Stay' and 'Rearrange' cell states. However, the precision of 25.4% alongside a recall of 26.5% for the 'Rearrange' predictions indicates significant challenges in accurately identifying true positive cases of cell rearrangements. This is further confirmed by an F1 Score of 26.0%, highlighting a balance between precision and recall that is suboptimal. These results suggest that while the model is proficient at identifying cells that remain static, it struggles to reliably identify cells undergoing rearrangements. This could be due to the model's sensitivity to the imbalanced nature of the data or possibly the need for further tuning of the model's parameters to better capture the characteristics of rearranging cells.

Table 2: Performance Metrics

Metric	Value
Accuracy	0.918
Precision	0.254
Recall	0.265
F1 Score	0.260

4.3 Discussion

Overall, the comparison of the reproduced results with those from the paper reveals that while the ability to predict static cells ('Stay') remains robust, there is a noticeable decline in the accuracy of predicting rearranging cells ('Rearrange'). This divergence may be attributed to variations in the training process, model parameters, or differences in the handling of the dataset. Further investigation into these factors is recommended to align the reproduced results more closely with the original findings and improve the reliability of predictions for cell dynamics in *Drosophila* embryos.

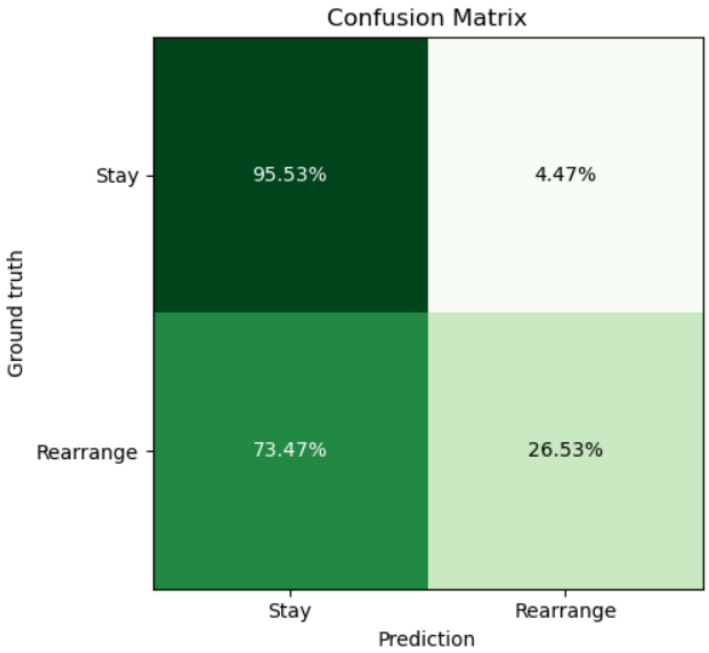

Figure 1: My reproduced result

5 Conclusion and Future Direction

In conclusion, this project has established a comprehensive pipeline that encompasses all stages from data preprocessing through exploratory data analysis (EDA), to developing the model architecture, training, and final evaluation. While the results did not meet all expectations, they provide a solid foundation for future improvements.

The analysis indicates that the model performs adequately in identifying static cells but falls short in accurately predicting cell rearrangements, reflecting a significant imbalance in class representation. This imbalance likely skews the model's ability to generalize well across different cell states, particularly affecting the precision and recall of the 'Rearrange' predictions.

For future work, it is imperative to focus on methods to rectify this imbalance. Techniques such as oversampling the minority class, synthesizing new examples through SMOTE (Synthetic Minority Oversampling Technique), or applying tailored loss functions that penalize misclassification of the minority class more heavily could be explored.

Further, continuous evaluation through a more extensive cross-validation framework across different datasets could provide deeper insights and a more robust validation of the model's capabilities.

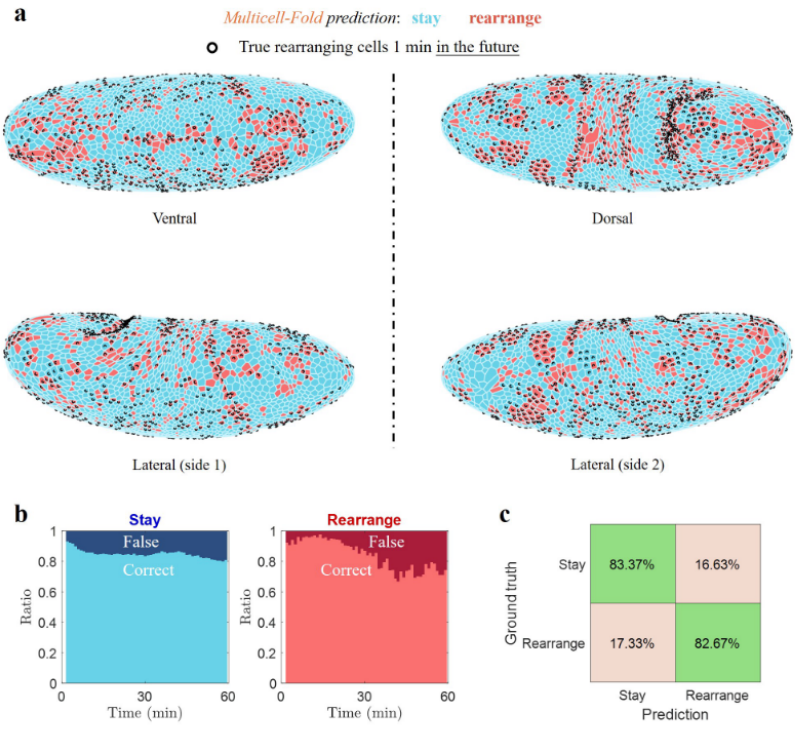

Figure 2: Results from paper

6 Supplementary Materials

6.1 Raw Dataset Structures - Description of the Segmented Embryo

The following describes the dataset structure used to store a segmented and tracked embryo. The command:

```
[C, E, V] = DG_load_segmented_embryo(...);
```

loads the segmented and tracked embryo in Matlab. This dataset contains all the necessary information for analyzing planar behaviors of the embryo (excluding volumes). The structures are organized as follows:

- **C**: A cell array of size $(t, 1)$, where t is the number of time points. Each cell $C\{t\}$ contains a struct array of size $(c, 1)$, where c is the number of cells at time point t. Each entry $C\{t\}(c)$ contains:
  - `C{t}(c).vertices`: A row vector containing indices (positive integers) of the vertices surrounding the cell
  - `C{t}(c).edges`: A row vector containing indices (non-zero integers) of the edges surrounding the cell. The sign indicates the traversal direction along the edges
  - `C{t}(c).cells`: A row vector containing indices (positive integers) of the immediate neighbor cells
  - `C{t}(c).centroid`: A row vector of size $(1, 3)$ with the XYZ coordinates of the cell centroid. Here, it is all zero.

  Notes:
  - The `vertices`, `edges`, and `cells` fields are ordered for traversal around the cell
  - $C\{t\}(c)$ and $C\{t+1\}(c)$ refer to the same cell at consecutive time points for tracking
  - If cell c does not exist at time t, all fields in $C\{t\}(c)$ are set to `[]`
- **E**: A cell array of size $(t, 1)$, where t is the number of time points. Each cell $E\{t\}$ contains a struct array of size $(e, 1)$, where e is the number of edges at time t. Each entry $E\{t\}(e)$ contains:
  - `E{t}(e).pixels`: Ignored, set to `[]`
  - `E{t}(e).vertices`: A row vector of size $(1, 2)$ with indices of the edge's two end vertices
  - `E{t}(e).edges`: A row vector of size $(1, n)$, $n \geq 4$, containing indices (positive integers) of the edge's immediate neighbor edges
  - `E{t}(e).cells`: A row vector of size $(1, 2)$ with indices of the two cells interfaced through the edge
- **V**: A cell array of size $(t, 1)$, where t is the number of time points. Each cell $V\{t\}$ contains a struct array of size $(v, 1)$, where v is the number of vertices at time t. Each entry $V\{t\}(v)$ contains:
  - `V{t}(v).coords`: A row vector of size $(1, 3)$ with the XYZ coordinates of the vertex. Here, $Z = 0$
  - `V{t}(v).vertices`: A row vector of size $(1, n)$, $n \geq 3$, containing indices (positive integers) of the neighboring vertices
  - `V{t}(v).edges`: A row vector of size $(1, n)$, $n \geq 3$, containing indices (positive integers) of the edge's immediate neighbor edges
  - `V{t}(v).cells`: A row vector of size $(1, n)$, $n \geq 3$, containing indices (positive integers) of the vertex's neighboring cells

6.2 Description of Matlab function used for preprocessing

6.2.1 DG_calc_cell_areas

Inputs:

- `C` - A cell array containing information about the cells. Each element `C{t}(c).centroid` represents the centroid of cell `c` at time point `t`.
- `V` - A cell array containing information about the vertices.

Outputs:

- `cell_area` - A numeric matrix with dimensions [total time points, total cells] storing the areas of the cells.

Procedure:

1. Determine the number of cells and time points.
2. Initialize the `cell_area` matrix with `NaN` values of type `single`.
3. Adjust the `C` array to have a uniform length by filling in default structures where necessary.
4. Use a parallel for-loop (`parfor`) to:
   (a) Retrieve vertices of each cell.
   (b) Extract coordinates from `V` based on these vertices.
   (c) Perform PCA on the coordinates to simplify the geometry.
   (d) Calculate the polygon area with `polyarea` assuming a closed loop.

6.2.2 calc_cell_perimeters

Inputs:

- `C` - A cell array containing information about cells at each time point, with each cell's vertices indexed.
- `V` - A cell array containing vertex coordinates at each time point.

Outputs:

- `cell_perimeter` - A numeric matrix with dimensions [n_time_points, n_cells] that stores the perimeters of the cells.

Procedure:

1. Initialize variables for the number of cells and time points.
2. Preallocate the output matrix with NaN values.
3. Ensure uniformity in the structure array `C` by filling with default structures.
4. Use a parallel for-loop (`parfor`) to:
   (a) Retrieve vertices and their coordinates for each cell.
   (b) Close the polygon by repeating the first vertex at the end.
   (c) Calculate the perimeter as the sum of Euclidean distances between consecutive vertices.
   (d) Store the calculated perimeter in the matrix.

6.2.3 `calc_edge_lengths`

Inputs:

- `C` - Cell array containing cell data structure across time points.
- `E` - Cell array containing edge data structure across time points.
- `V` - Cell array containing vertex data structure across time points.

Outputs:

- `cellEdgeDistances` - Matrix containing time point, indices of two connected cells, edge index, and distance information. [timepoint, cell_1, cell_2, edge_index, edge_distance]

Procedure:

1. Initialize the number of time points and a temporary storage array.
2. Use a parallel loop (`parfor`) to:
   (a) Process each time point independently.
   (b) For each edge, check connectivity and validity of the vertices.
   (c) Retrieve vertex coordinates and calculate the Euclidean distance.
   (d) Store results in a temporary array specific to that time point.
3. Concatenate all temporary arrays into the main output array.

6.2.4 `calculate_junction_loss`

Inputs:

- `E` - Cell array of edge data structures across time points, each structure detailing indices of connected cells.

Outputs:

- `junction_loss` - Matrix containing time point index, edge index, and a binary indicator for edge loss (1 if lost, 0 if not).

Procedure:

1. Initialize temporary storage for edge loss data, excluding the last time point.
2. Use parallel processing to analyze each time point up to the second-to-last:
   (a) Extract edges for the current and next time points.
   (b) Identify edges that are lost by the next time point.
   (c) Record each edge's loss status in a matrix.
3. Concatenate all time point loss data into the output matrix.

6.2.5 `save_node_features`

Inputs:

- `C` - Cell array containing data for each cell across time points.
- `cell_areas` - Numeric matrix with the areas of each cell at each time point.
- `perimeters` - Numeric matrix with the perimeters of each cell at each time point.

Outputs:

- `node_features` - Cell array, each cell containing a matrix of node features per time point. Features include cell index, area, change in area, perimeter, and change in perimeter.

Procedure:

1. Initialize the output cell array based on the number of time points.
2. Use parallel processing to compute features for each time point:
   (a) For each cell, handle NaN values by replacing them with zero for area and perimeter.
   (b) Compute the difference in area and perimeter from the previous time point.
   (c) Store these features in a local matrix.
3. Assign the computed matrix to the corresponding element of the output cell array.

6.2.6 `save_edge_features`

Inputs:

- `E` - Cell array of edge data structures across time points, each structure containing indices of connected cells.
- `edge_lengths` - Numeric matrix with edge lengths including time point, cell indices, edge index, and length.

Outputs:

- `edge_features` - Cell array, each element containing a matrix of edge features per time point. Features include indices of connected cells, edge index, weighted length, and delta of weighted length.

Procedure:

1. Initialize edge feature storage and a lookup map for edge lengths.
2. Populate the lookup map with edge lengths for efficient retrieval.
3. Use parallel processing to compute features for each edge at each time point:
   (a) Retrieve or default the edge length.
   (b) Calculate weighted length and its change from the previous time point.
   (c) Store computed features in a preallocated matrix.
4. Trim and assign the feature matrices to the corresponding output cell array element.

6.2.7 `save_graph`

Inputs:

- `node_features` - Cell array of node features per time point.
- `edge_features` - Cell array of edge features per time point.
- `junction_loss` - Matrix indicating junction losses across time points.
- `type` - String specifying the type of graph, used for naming the output file.

Outputs:

- None returned, but the graph data is saved to a file.

Procedure:

1. Initialize storage for graph data up to the second-to-last time point.
2. For each time point, integrate node features, edge features, and junction loss data into a single structure.
3. Extract and adjust junction loss data specific to each time point.
4. Construct a filename using the input type and save the graph data in MAT-file format using version 7.3.

References

[1] Tomer Stern, Stanislav Y. Shvartsman, and Eric F. Wieschaus. Deconstructing gastrulation at single-cell resolution. *Current Biology*, 32(8):1861–1868.e7, 2022.

[2] Haiqian Yang, Anh Q. Nguyen, Dapeng Bi, Markus J. Buehler, and Ming Guo. Multicell-fold: geometric learning in folding multicellular life, 2024.