Convolutional graph neural networks for tracking yeast cells

G. Brenna, F. Massard, N. Vadot
EPFL, Switzerland

***Abstract*—We present a graph-based machine learning method that is able to track yeast cells throughout a movie, after having segmented the frames (here, this is done with YeaZ [1]). The method achieves better accuracy than the Hungarian algorithm implemented in YeaZ, thanks to the context-awareness of the graph structure. The method relies on translating the tracking problem to a binary node classification problem, and should generalize to any tracking problem where segmentations are available. The source code, as well as explicative Python notebooks, are available at [2].**

I. Context

The tracking problem emerges when addressing a more specific problem, that of lineage determination of budding yeast : for each new bud, the parent cell needs to be determined. This is made difficult in densely packed environments, but budneck markers can be used to manually determine the parent of each bud. However, it might not always be convenient to use such markers, and the question arises if lineage determination is possible with only a phase contrast image. Buds are most visible when they grow, so a machine learning algorithm might be able to pick up on spatio-temporal correlations in the segmentations to determine the parent. However, feeding a machine learning algorithm coherent temporal information requires a consistent tracking of the cells, which is the main problem this report addresses.

The classical approach to tracking is to extract some features from the segmentation geometry (YeaZ uses cell surface area and center of mass), and minimize the pairwise euclidean distance in the feature space using the Hungarian algorithm. The problem with this method is that it starts to break down for large colonies, since in that regime many cells have similar geometry and entire sections of the colony might drift away. Then, it would be interesting to look at pairwise relationships between cells and their nearest neighbors to track the cells, similarly to how puzzle pieces fall into place given the right context.

A graph-based approach arises, which can naturally encode cells as nodes with certain features, and nearest-neighbor relationships are edge features. The recent development of graph neural networks (GNN) [3] and growing open-source codebase [4], [5] makes it possible to easily implement machine learning methods that are able to learn on graphs.

II. Prior work

The problem of tracking arises in other domains, such as reconstruction of particle trajectory in detectors [6]. In this approach, timestamped detection events in cylindrical coordinates are represented as nodes, and edge features are constructed by considering geometrical differences of the two detection events it connects. Features are then encoded with an MLP, and an edge-classification task is performed in order to attribute multiple detection events to the same particle. In our context, cells are not moving in a highly correlated manner like a particle through a detector, but the idea of encoding geometrical features can be repurposed.

Tracking nodes, as they evolve in time-dependent graphs, can be reformulated as the problem of graph matching. [7] considers the problem of finding a subgraph in a larger graph. It does this by applying graph convolutions, embedding the nodes in a high-dimensional space, and minimizing the total cost of pairwise euclidean distance to match nodes from the two graphs. We try to reformulate the tracking problem as a graph matching problem, and use a similar idea by maximizing a score to assign nodes between two graphs.

III. Data preprocessing

A. Feature extraction and cell graph generation

The full data preprocessing pipeline is schematized in Fig. 1. The data used here consists of 5 colonies of budding yeast, with phase contrast images taken at 5 minute intervals, for a total of 180 images (15 hours of growth time) per colony. The first 4 colonies (in the data, named `colony00[0123]`) are used for generating training (and validation) datasets, and the last colony (named `colony007`) is kept as a final testing set. On average, the colonies started with 2 cells and had 110 cells at the end.

The movies were segmented and tracked semi-automatically using YeaZ [1]. From the segmentation masks, simple geometrical features were extracted from the cells : surface area A, radius r and eccentricity e. To do this, an ellipse was fitted to the cell by using PCA analysis on the cell mask coordinates, which can then be used to obtain the semi-major and minor axes a and b respectively. Then we compute $e = \sqrt{1-(\frac{a}{b})^2}$, and the radius of the circle with the same area as the ellipse $r = \sqrt{ab}$. Fig. 4 shows the distribution of the node features. The distributions are strongly peaked around their mean, respectively $\langle A \rangle \approx 21 \pm 14\ \mu\text{m}^2$, $\langle r \rangle \approx 2.5 \pm 0.8\ \mu\text{m}$ and $\langle e \rangle \approx 0.52 \pm 0.14$. Notice the distribution for r has a few outliers corresponding to very large cells, and these outliers are even more visible for A, explained by the scaling law $A \sim r^2$. It however doesn't make sense to remove these outliers, since that would have the same effect as removing the cell from the colony, impacting its direct neighbors.

Nearest neighbor features were extracted by considering the contour of each cell mask, and placing an edge between cells if the minimum distance between their contours is below 1.3 µm (this threshold is arbitrary, and can be changed as needed when calling `bread_nn.graph.build_cellgraph`). Extracted features are ρ, the center of mass (CM) to CM distance between the neighboring cells, θ, the angle of the CM to CM vector with respect to the horizontal axis, and ℓ, the minimal distance between the contours. FIG. 5 shows the distribution of these features. ρ shows a distribution peaked around $\langle\rho\rangle \approx 5.4 \pm 1.2$, which makes sense, as on average cell has a radius of $r \approx 2.5$ µm, meaning two average cells would be separated by approximately $2r \approx 5.0$ µm. θ shows a near-uniform distribution, with mean and standard deviation $\langle\theta\rangle \approx 0 \pm 1.8$. Again, this is intuitive, since we don't expect cells to be preferentially aligned in one direction. Note that we allow θ to take values in $[-\pi, \pi]$ instead of $[0, \pi]$, because edges are not stored bidirectionally (to save memory), so the sign of θ encodes directionality of the edge. ℓ is strongly peaked around $\langle\ell\rangle \approx 0.34 \pm 0.22$ µm. Physically, ℓ is not that representative since membranes are supposed to be in contact, and the mean being slightly offset is an artifact from the segmentation process. Nevertheless, ℓ can be a good distance metric between cells, since it is independent of their sizes.

In total, 720 cell graphs we constructed, for a total of 25510 nodes and 48018 edges (note : edges are considered bidirectional, but are stored as one-directional to save memory). The Python notebook used for this process can be found under `experiment_gnn/build_cellgraphs.ipynb`.

B. Assignment graph generation

We formulate the tracking problem as a graph matching problem. Suppose we have generated two graphs $\mathcal{G}^{(1)}$ and $\mathcal{G}^{(2)}$ storing extracted features from two segmentations, and we want to track cells from segmentation (1) to segmentation (2). For this, we construct an assignment graph $\mathcal{G}^{(a;x,e)}$. Let $\boldsymbol{v_i^{(1)}}, \boldsymbol{e_{i,j}^{(1)}}$ (resp. $\boldsymbol{v_a^{(2)}}, \boldsymbol{e_{a,b}^{(1)}}$) encode the nodes and edges of $\mathcal{G}^{(1)}$ (resp. $\mathcal{G}^{(2)}$). Nodes $\boldsymbol{v_{ia}^{(a)}}$ of $\mathcal{G}^{(a;x,e)}$ are obtained by concatenating features from $\boldsymbol{v_i^{(1)}}$ and $\boldsymbol{v_a^{(2)}}$. Edges $\boldsymbol{e_{ia,jb}^{(a)}}$ of $\mathcal{G}^{(a)}$ link nodes $\boldsymbol{v_{ia}^{(a)}}$ and $\boldsymbol{v_{jb}^{(a)}}$ if $\boldsymbol{e_{i,j}^{(1)}}$ and $\boldsymbol{e_{a,b}^{(2)}}$ exist in $\mathcal{G}^{(1)}$ and $\mathcal{G}^{(2)}$ respectively, in which case edge features are similarly obtained by concatenation.

Finally, we similarly build the corresponding target graph $\mathcal{G}^{(a;y)}$, where the node features are simply 2-dimensional one-hot encoder vectors, indicating whether a node $\boldsymbol{v_{ia}^{(a)}}$ corresponds to a correct tracking. Note that at this point, there is no trace left of the cell ids that were used to differentiate cells in the segmentations.

Concretely, the assignment graph dataset was obtained by considering all pairs of segmentations separated by at most 20 frames (1h40), constructing the assignment graphs, and repeating this for each colony. The Python notebook used for this process can be found under `experiment_gnn/build_assgraphs.ipynb`.

IV. METHODS

A. GNN structure

The GNN structure is summarized in FIG. 2. Node and edge features of $\mathcal{G}^{(a;x,e)}$ are embedded by two MLPs with ReLU activation. The resulting embedded graph is then convolved by multiple DeepGCN layers [8]. One DeepGCN layer consists of a batch normalization layer (LayerNorm), an activation function (ReLU), one dropout layer, one GENConv layer, then finally a residual connection layer. In particular, the GENConv layer constructs messages $\boldsymbol{x_i}'$ from node and edge features $\boldsymbol{x_i}, \boldsymbol{e_{i,j}}$ as

$$\boldsymbol{x_i}' = \mathrm{MLP}(\boldsymbol{x_i}' + \mathrm{AGG}(\{\mathrm{ReLU}(\boldsymbol{x_i} + \boldsymbol{e_{i,j}}) + \epsilon \mid i \in \mathcal{N}(j)\})),$$

where $\mathcal{N}(j)$ denotes indices of the nodes connected to node j, AGG is a permutation-invariant aggregation function (in this case, a softmax), and ϵ is a small learnable constant. The MLP has 2 layers, and preserves the number of channels of the input.

The final linear layer maps the embedded node channels to two channels, and the resulting graph $\mathcal{G}^{(a;\hat{y})}$ is then passed to CrossEntropyLoss along with the ground truth $\mathcal{G}^{(a;y)}$ for backpropagation.

B. Making predictions

Let n_1 and n_2 be the number of cells in the first and second images respectively. We define an assignment matrix A with components $A_{i,j} = 1$ if cell i in the first image is the same as cell j in the second image, else $A_{i,j} = 0$. For the example in FIG. 1, the assignment matrix is

$$A = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}.$$

From the predicted graph $\mathcal{G}^{(a;\hat{y})}$, we can construct an assignment matrix by reshaping the node features into a matrix Z of shape $(n_1,\ n_2,\ 2)$, where the last dimension stores weights for the classification decision. There are multiple ways of transforming Z into A, but we describe only the method that empirically worked the best (other methods are presented and discussed in `experiment_gnn/pipeline.ipynb`).

By taking differences along the last axis of Z, we obtain a score matrix S of shape $(n_1,\ n_2)$, where each entry's magnitude can be interpreted as the confidence of the GNN of the assignment. The goal is then to find $A^\star$ as such to maximize the assignment score, mathematically

$$A^\star = \arg \max_{A \in \{0,1\}^{n_1 \times n_2}} \sum_{i,j} S_{i,j} A_{i,j}.$$

This linear sum assignment (linsum) is easily solved by using `scipy.optimize.linear_sum_assignment` from the Scipy package [9].

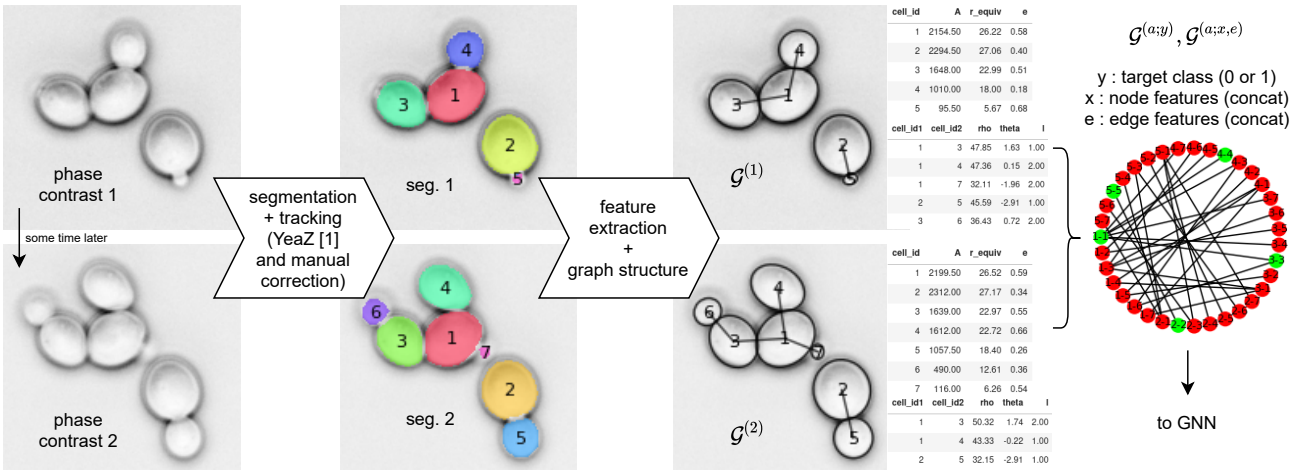

FIGURE 1: Data preprocessing pipeline. A movie of a budding yeast colony is taken (in phase contrast, as is the case here), then YeaZ [1] is used to segment individual frames, and semi-automatically track cells. Manual corrections to segmentation and tracking are applied as required. From the segmentations, geometric features are extracted, as well as features describing the neighborhood of each cell. This is stored in a graph structure $\mathcal{G}^{(1)}$ and $\mathcal{G}^{(2)}$, from which an assignment graph $\mathcal{G}^{(a;x,e)}$ is built. A ground truth assignment graph $\mathcal{G}^{(a;y)}$ is also built. Both assignment graphs are then used by the GNN for training.

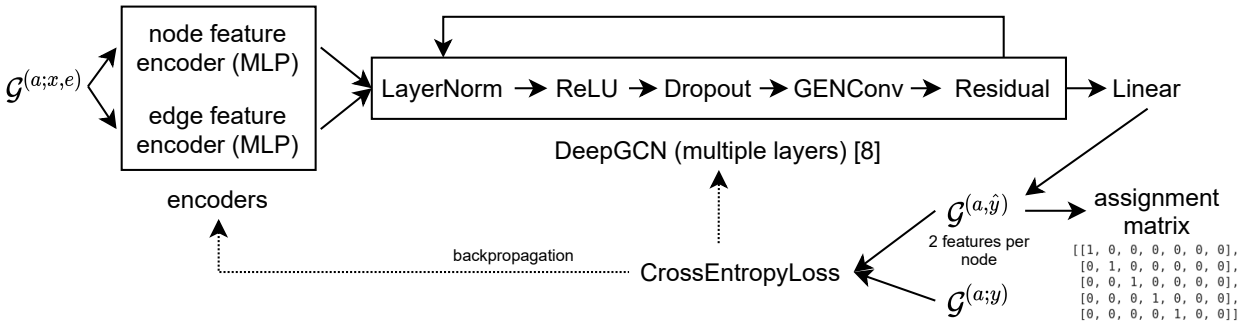

FIGURE 2: The GNN structure. For a given input graph, the node and edge features are encoded by MLPs. The resulting graph is then convolved multiple times following DeepGCN [8]. A final linear layer maps the encoded node features down to two, and CrossEntropyLoss is computed for backpropagation. For evaluation, the node features of $\mathcal{G}^{(a;\hat{y})}$ can be reduced and reshaped to build an assignment matrix.

V. RESULTS

This section discusses performance on the models trained with the constructed graphs, as described in section III. All models are trained with the Adam optimizer with varible exponentially decaying learning rate, adjusted manually depending on the batch size used. Unless specified otherwise, all layers use dropout (if it applies) during training with probability parameter 0.1. The random seed is fixed at the start of each training session. Training and validation datasets are shuffled at each epoch, and make up 80% and 20% of the total dataset respectively.

For evaluation, models predicting assignment matrix $\hat{A}$ are compared to the ground truth A using the accuracy metric defined as

$$\mathrm{acc}(\hat{A}, A) = \frac{1}{n_1 n_2} \sum_{i,j} \delta_{A_{i,j}, \hat{A}_{i,j}}.$$

A. MLP baseline

As a baseline for the GNN, we train a MLP on the node features $\boldsymbol{X}$ of $G^{(a;x,e)}$ to evaluate if the graph structure (that is, addition of edge features) really adds valuable information for learning.

For this, we need to equilibrate the labels $\boldsymbol{y}$, because asymptotically the number of nodes in the assignment graph grows as $n_1 n_2$, but the number of assignments where $y = 1$ grows as n_1. Therefore, labels $y = 1$ are underrepresented in large graphs, and equilibration is done by removing samples where $y = 0$ from the training batch, until both classes are equally represented.

Furthermore, batch normalization is applied before the MLP in order to improve training, as described in [10].

A hyperparameter scan was performed on the number of MLP layers and hidden channels, and found that the best performing MLP obtained an accuracy of approximately 0.88. More layers and hidden channels were found to marginally improve the accuracy, before it started to drop

due to overfitting.

B. GNN performance

We perform a scan on the following hyperparameters : N_{enc} number of layers in both encoder MLPs, N_{conv} number of DeepGCN layers, and N_{dim} the number of hidden channels. Other hyperparameters, such as dropout, learning rate, exponential learning rate decay and training batch size were found to not significatively improve final performance, but can decrease training time if correctly chosen (here they were set to 0.1, $\sim 10^{-4}$, ~ 0.98, ~ 32 respectively). For the study, we chose $N_{\text{enc}} \in \{1, 2, 3, 4, 5\}$, $N_{\text{conv}} \in \{1, 2, 3, 4, 5, 8, 11\}$ and $N_{\text{dim}} \in \{30, 60, 90, 120\}$. FIG. 6 plots the results of the hyperparameter scan in parallel coordinates.

Models with less than 0.99 accuracy are obtained with few parameters overall, $N_{\text{enc}} \in \{1, 2, 3\}$, $N_{\text{conv}} \in \{1, 2, 3, 4\}$ and $N_{\text{dim}} \in \{30, 60\}$. For better accuracy, it was found most effective to increase the number of encoder layers $N_{\text{enc}} \in \{2, 3\}$, which systematically drove up the accuracy up to 0.999. Following this, increasing N_{enc} only marginally improved accuracy, but allowing up to 5 convolution layers gave models obtaining accuracy of up to 0.9999. In this regime, the graph neural network is able to capture most of the complexity of the problem, but still struggles with edge cases (for instance, a cell moving a large amount). Further increasing the number of parameters (via N_{conv} or N_{dim}, they seem to compensate each other) results in even better accuracy, where edge cases are now correctly handled. The best models obtained an accuracy of 0.99997, and utilized $N_{\text{enc}} \geq 3$, $N_{\text{conv}} \geq 8$ and $N_{\text{dim}} \geq 80$.

Intuitively, the performance gain of increasing N_{enc} eventually caps off, because each cell graph node or edge is decribed with only 3 features, and the MLP can only transform information it is given. Increasing N_{conv} intuitively can be seen as increasing the "communication distance" between cells, meaning each node has a "better view" of the whole graph structure. This of course comes with the cost of requiring more dimensions to store the information in, explaining the need to increase N_{dim}, and how both compensate each other. It is not useful to be able to "see" further in the graph (N_{conv} large) if you cannot "store" the information (N_{dim} small), and vice-versa.

We now compare the performance of the best GNN model ($N_{\text{enc}} = 5$, $N_{\text{conv}} = 11$, $N_{\text{dim}} = 120$) against the built-in tracking system of YeaZ, using the test dataset. From the 180 frames in the dataset, 885 pairs are tested, corresponding to frames separated by 5 to 25 minutes. Numerical results are presented in TAB. I. The linear sum assignment method works the best, and performs better than YeaZ, especially for pairs of frames separated by a long time difference, and in situations where cells move around a lot (specific examples are shown in FIG. 3). The dataset's frames were separated by 5 minutes here, however one should expect approximately 15 minutes of frame separation in real applications, in which case this method has been shown to be performant.

	mean	std	25%	50%	75%
naive	0.986984	0.043199	0.997253	1.0	1.0
fw	0.998341	0.014823	1.000000	1.0	1.0
bw	0.986895	0.040118	0.993590	1.0	1.0
bw+corr	0.995342	0.017675	1.000000	1.0	1.0
linsum	**0.999913**	0.000645	1.000000	1.0	1.0
yeaz	0.994201	0.059083	1.000000	1.0	1.0

TABLE I: Test accuracy of different methods of generating the assignment matrices from the GNN's output. The testing set (`colony007`) was split into pairs of 885 interframes, separated in real time by 5 to 25 minutes. "naive" refers to classifying each node simply based on the weights for each class. "fw" refers to the forward method, which forces each cell in the first frame to match one in the second frame. "bw" refers to the backwards method, which forces each cell in the second frame to match one in the first frame. "bw+corr" is a corrected version of the backward method, which accounts for buds. "linsum" uses the linear sum optimization method, as presented in subsection IV-B. "yeaz" is the baseline tracking method. See `experiment_gnn/pipeline.ipynb` for more information.

VI. CONCLUSION

We have shown that this graph neural network model is able to learn the structure of a yeast cell colony, and is able to reliably track cells from frame to frame. It has been shown that the model exceeds YeaZ's integrated tracker performance, using a relatively small training dataset of only 4 colonies growing over 15 hours and imaged every 5 minutes. If implemented in YeaZ's GUI, tracking tasks will become easier and significantly less time-consuming, especially for large colonies.

One of the limitations of this model is the need to have segmented the images and verified the segmentations. We however argue that thanks to advancements in this domain [1], [11], this has become a trivial task. Nevertheless, it is a good practice to verify the segmentations manually, and correct as needed.

The model presented in this report essentially only uses two graphs as an input, and could benefit from having a "longer history" of the colony in order to make better tracking predictions. Multiple papers [12], [13], [14] address the problem of temporal graphs, and it might be worth investigating to further the research. In summary, using temporal graph networks are a promising approach to constructing lineages from only phase contrast (or bright field) microscopy movies. This would likely have to be coupled to an attention system, for which literature and implementations for graph neural networks already exist [15], [16].

References

[1] N. Dietler, M. Minder, V. Gligorovski, A. M. Economou, D. A. H. L. Joly, A. Sadeghi, C. H. M. Chan, M. Koziński, M. Weigert, A.-F. Bitbol, and S. J. Rahi, "A convolutional neural network segments yeast microscopy images with high accuracy," *Nature Communications*, vol. 11, no. 1, p. 5723, nov 2020. [Online]. Available: https://www.nature.com/articles/s41467-020-19557-4

[2] N. Vadot, "bread-tracking source code," Dec. 2021. [Online]. Available: https://github.com/ninivert/bread-tracking

[3] J. Zhou, G. Cui, S. Hu, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li, and M. Sun, "Graph Neural Networks: A Review of Methods and Applications," *arXiv:1812.08434 [cs, stat]*, oct 2021, arXiv: 1812.08434. [Online]. Available: http://arxiv.org/abs/1812.08434

[4] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d. Alché-Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[5] M. Fey and J. E. Lenssen, "Fast Graph Representation Learning with PyTorch Geometric," *arXiv:1903.02428 [cs, stat]*, apr 2019, arXiv: 1903.02428. [Online]. Available: http://arxiv.org/abs/1903.02428

[6] G. DeZoort, S. Thais, J. Duarte, V. Razavimaleki, M. Atkinson, I. Ojalvo, M. Neubauer, and P. Elmer, "Charged particle tracking via edge-classifying interaction networks," *arXiv:2103.16701 [hep-ex]*, nov 2021, arXiv: 2103.16701. [Online]. Available: http://arxiv.org/abs/2103.16701

[7] T. N. Kipf and M. Welling, "Semi-Supervised Classification with Graph Convolutional Networks," *arXiv:1609.02907 [cs, stat]*, feb 2017, arXiv: 1609.02907. [Online]. Available: http://arxiv.org/abs/1609.02907

[8] G. Li, C. Xiong, A. Thabet, and B. Ghanem, "DeeperGCN: All You Need to Train Deeper GCNs," *arXiv:2006.07739 [cs, stat]*, jun 2020, arXiv: 2006.07739. [Online]. Available: http://arxiv.org/abs/2006.07739

[9] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors, "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python," *Nature Methods*, vol. 17, pp. 261–272, 2020.

[10] S. Ioffe and C. Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," *arXiv:1502.03167 [cs]*, Mar. 2015, arXiv: 1502.03167. [Online]. Available: http://arxiv.org/abs/1502.03167

[11] C. Stringer, M. Michaelos, and M. Pachitariu, "Cellpose: a generalist algorithm for cellular segmentation," Tech. Rep., feb 2020, type: article. [Online]. Available: https://www.biorxiv.org/content/10.1101/2020.02.02.931238v1

[12] U. Singer, I. Guy, and K. Radinsky, "Node Embedding over Temporal Graphs," *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence*, pp. 4605–4612, aug 2019, arXiv: 1903.08889. [Online]. Available: http://arxiv.org/abs/1903.08889

[13] W. Jin, M. Qu, X. Jin, and X. Ren, "Recurrent Event Network: Autoregressive Structure Inference over Temporal Knowledge Graphs," *arXiv:1904.05530 [cs, stat]*, oct 2020, arXiv: 1904.05530. [Online]. Available: http://arxiv.org/abs/1904.05530

[14] E. Rossi, B. Chamberlain, F. Frasca, D. Eynard, F. Monti, and M. Bronstein, "Temporal Graph Networks for Deep Learning on Dynamic Graphs," *arXiv:2006.10637 [cs, stat]*, oct 2020, arXiv: 2006.10637. [Online]. Available: http://arxiv.org/abs/2006.10637

[15] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph Attention Networks," *arXiv:1710.10903 [cs, stat]*, Feb. 2018, arXiv: 1710.10903. [Online]. Available: http://arxiv.org/abs/1710.10903

[16] S. Brody, U. Alon, and E. Yahav, "How Attentive are Graph Attention Networks?" *arXiv:2105.14491 [cs]*, Oct. 2021, arXiv: 2105.14491. [Online]. Available: http://arxiv.org/abs/2105.14491

APPENDIX

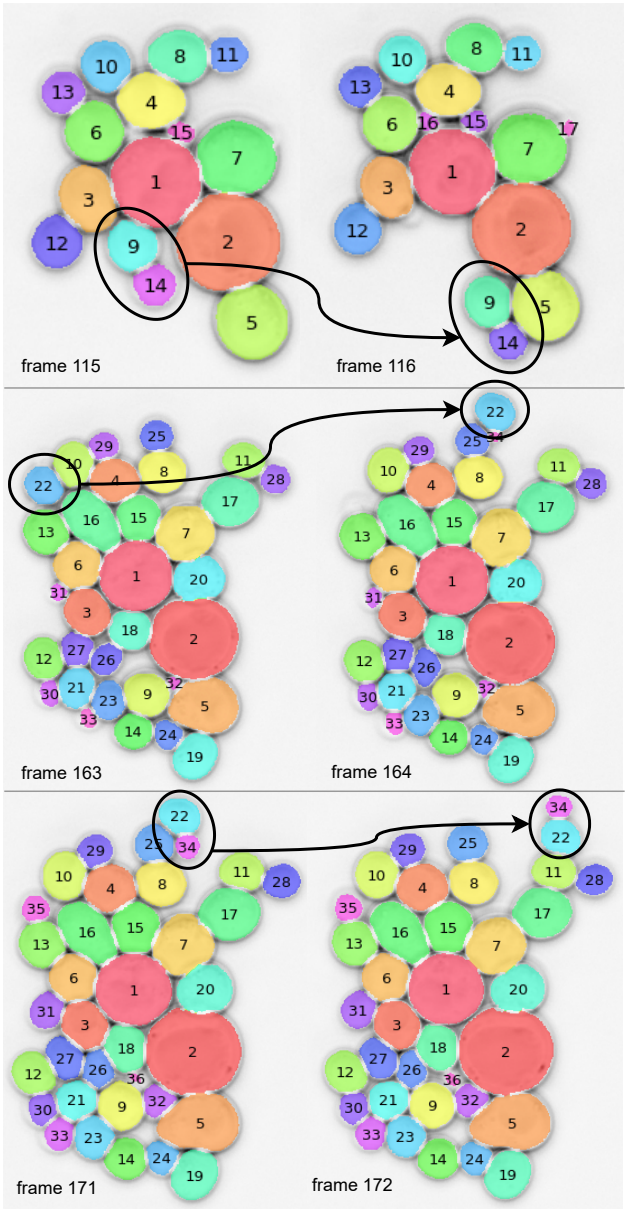

FIGURE 3: Segmentations from `colony007` highlighting moving cells. Out of the 179 interframes, YeaZ makes 3 tracking mistakes. On frame 115, cells 9 and 14 move together towards cell 5, and YeaZ tracks cell 9 to cell 16 and cell 14 to cell 9. On frame 163, cell 22 moves around the top of the colony, and YeaZ tracks cell 10 to 22, cell 10 to 29, cell 29 to cell 25 and cell 25 to 34. On frame 171, cells 22 and 34 move together around the top while swapping places, and YeaZ tracks cell 22 to 34 and vice-versa.

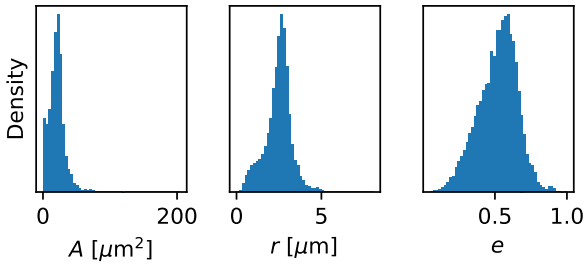

FIGURE 4: Distribution of node features.

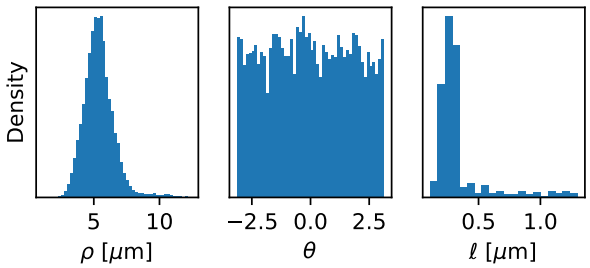

FIGURE 5: Distribution of edge features.

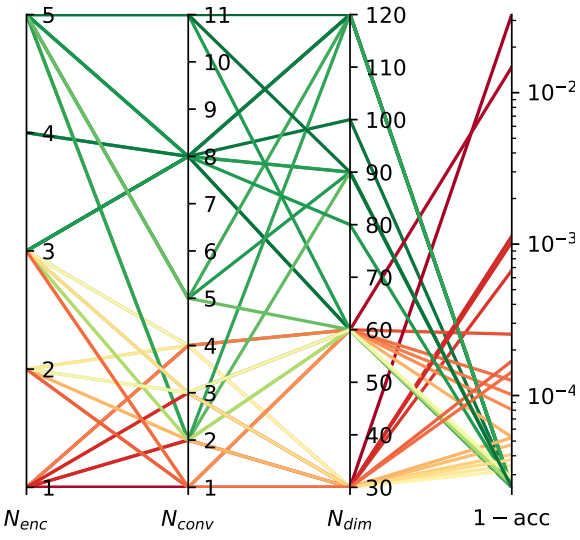

FIGURE 6: Parallel coordinates plot of the hyperparameter scan. The best performing model uses $N_{\text{enc}} = 5$, $N_{\text{conv}} = 11$, $N_{\text{dim}} = 120$, but a very similar performance can be achieved for less parameters, $N_{\text{enc}} \geq 3$, $N_{\text{conv}} \geq 8$ and $N_{\text{dim}} \geq 80$.