Supplemental Material for "LivePyxel: Accelerating image annotations with a Python-integrated webcam live streaming"

Uriel Garcilazo-Cruz,[a,b,†], Joseph O. Okeme[a,b], and Rodrigo A. Vargas-Hernández[a,b,c,†]

The purpose of this supplemental material is to provide additional details about the proposed work in the main draft. Section 3 presents the additional details of the vision model for microscopic organism segmentation, including hardware specifications, model architecture, dataset characteristics, and training performance. Section 4 complements this with a different application case focused on data engineering using snail shell images, detailing its specialized dataset and training results. Each section is organized to provide: (1) hardware/experimental setup details, (2) model architecture specifications (with Table 2 being particularly relevant for technical implementation), (3) dataset composition analysis (including class distributions shown in Figures 2 and 7), and (4) quantitative training evaluations (with F1-score trajectories in Figs. 3 and 6).

1 Dataset preparation

All datasets in the study shared the same preprocessing in preparation to enter the vision model. A python script traversed across all image/mask pairs in the dataset, quantifying the relative frequency of pixels for each category. This information was later used to calibrate the network's cross-entropy loss function and make it frequency-dependent. The dataset was prepared using a custom set of scripts that read the `config.json` file to generate indexes for each label, and mapped it to each of the masks created by LivePyxel, by replacing the color of the mask with the corresponding index, and having the background color of the mask: (0,0,0), as a category. All required scripts that address the dataset, model, and training modules are available in the GitHub repository examples folder. Future releases of LivePyxel are planned to include a pipeline to import a trained dataset that automates the annotation process using models trained by the user.

2 Vision Model Architecture

All datasets in this study were trained using the same U-Net network. The original dataset was split into training and validation, using 90% of image/mask pairs for training and 10% for validation. The network was trained using a custom number of epochs, but using a batch size of 32, a learning rate of 3×10^{-4}, the AdamW optimizer, and the cross-entropy loss function. All training was performed using the Compute Canada resources located in the Narval cluster, using a single node with up to 4 GPUs connected.

The U-Net's encoder consisted of four `downsampling` stages (`downsample1` to `downsample4`), each comprising convolutional layers (with ReLU activation) and max-pooling. The first two stages used double convolutions (e.g., Conv2d(3→64→64) and Conv2d(64→128→128)), while the deeper stages (`downsample3`, `downsample4`) employed quadruple convolutions (e.g., Conv2d(128→256→256→256→256) to capture complex features. The bottleneck (`bottle_neck`) expands the channel dimension to 1024 via two convolutions. The decoder (`upsample1` to `upsample4`) used transposed convolutions for upsampling, followed by double convolutions that halved the channel dimensions (1024→512→512), with skip connections concatenating encoder features. The final 1×1 convolution (out) reduced the output to 8 channels, corresponding to the segmentation classes, each encoding a unique color; see Table 1. The model contained 26.3 million trainable parameters, with the decoder and bottleneck layers fine-tuned (as indicated by True in the parameter table), while the encoder weights remain fixed (False). This design balanced computational efficiency with multi-scale feature learning, suitable for pixel-wise segmentation tasks. We used a U-Net architecture[1], composed of an encoder and decoder portion and skip connections. The encoder, or downsampling module of the architecture, was initialized with the weights and biases of the VGG-19 architecture[2] (Table 2).

Index	Label	Color (RGB)
0	ostracod	(0, 255, 0)
1	rotifer	(211, 179, 145)
2	algae	(164, 251, 233)
3	diatom	(202, 215, 220)
4	square_algae	(230, 226, 246)
5	paramecium	(207, 198, 149)
6	vorticella	(23, 54, 255)
7	tardigrade	(255, 8, 8)

Table 1 Labels and corresponding colors used in the segmentation task.

3 Examples of Usage: Water Tank

In this section, we provide the necessary information for the segmentation task presented in Section 3.1 from the main draft.

3.1 Hardware Setup

The imaging setup consisted of a custom configuration using a Zeiss compound microscope (model X-100) with a 10x achromatic objective lens (NA 0.25) and bright-field (Koehler) illumination. Images were captured using a modified GoPro camera equipped with an M12 CCTV lens, affixed to a standard 10x wide-angle eyepiece. A video capture card streamed the camera out-

[a] Department of Chemistry and Chemical Biology, McMaster University, Hamilton, ON, Canada
[b] School of Computational Science and Engineering, McMaster University, Hamilton, ON, Canada
[c] Brockhouse Institute for Materials Research, McMaster University, Hamilton, ON, Canada
† garcilau@mcmaster.ca, vargashr@mcmater.ca

Layer	Configuration	Trainable
downsample1	Conv2d(3→64→64) MaxPool2d	False
downsample2	Conv2d(64→128→128) MaxPool2d	False
downsample3	Conv2d(128→256×4) MaxPool2d	False
downsample4	Conv2d(256→512×4) MaxPool2d	False
bottle_neck	Conv2d(512→1024→1024) ReLU(inplace=True)	True
upsample1	ConvTranspose2d(1024→512) Conv2d(1024→512×2)	True
upsample2	ConvTranspose2d(512→256) Conv2d(512→256×2)	True
upsample3	ConvTranspose2d(256→128) Conv2d(256→128×2)	True
upsample4	ConvTranspose2d(128→64) Conv2d(128→64×2)	True
out	Conv2d(64→8, kernel=1×1)	True

Table 2 U-Net architecture. Encoder layers (fixed weights) downsample via convolutions and max-pooling; decoder layers (trainable) upsample with transposed convolutions and skip connections. ReLU activations follow all convolutions except the final layer. Total parameters: 26.3M.

put as a webcam, ensuring compatibility with OpenCV. A drawing tablet model XP-PEN 15.6 Pro was used to assist with annotations, by duplicating the display and projecting LivePyxel as a secondary screen. Fig. 1 depicts the complete imaging setup.

3.2 Dataset Information

The dataset was composed of 1.25K image/mask pairs, and included a wide asymmetric distribution of categories, as seen in the relative frequency of such classes Fig. 2). The three main classes in the dataset were samples of paramecia, vorticella, and green algae, which constitute 79% of the number of pixels in pixels in any category other than background.

3.3 Model Architecture

The water tank dataset was trained on 1,250 images, each with an original resolution of 720×480 pixels. Training was carried out over 131 epochs, with an additional set of 200 images reserved for testing and detection, none of which were used during training or validation. All training was performed using two NVIDIA V100 16 GB GPUs, totaling 8 hours of computation.

Fig. 4 illustrates the model's performance by plotting the training and validation losses. The plot clearly shows a sharp decline in both training and validation loss from epoch 0 to 20, followed by a gradual increase in validation loss afterward. This pattern suggests the model began overfitting, learning noise from the training data rather than generalizing effectively.

3.4 Training

In addition to the pattern seen in the loss function, the performance of the model can be seen in Fig. S3 with the F1 scores of every class, which rapidly increases in the early training stages and begins to plateau around epoch 20. While the F1 scores for most classes stabilize at high values, minor fluctuations and slower improvements occur in some classes beyond epoch 20. Together, the loss and F1 score trends indicate that the model reached its optimal generalization capability around epoch 20, making it the most reliable checkpoint for downstream tasks.

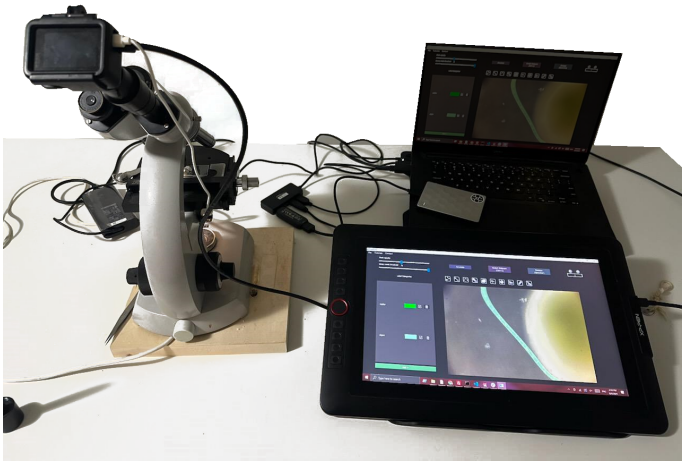

Fig. 1 Custom camera setup for deploying LivePyxel. The configuration includes a microscope equipped with a mounted digital camera, connected to a laptop and a pen display tablet for real-time image acquisition and analysis.

4 Examples of Usage: Data Engineering with snail shells

In this section, we provide the necessary information for Section 3.2 in the main draft. The snail shell dataset incorporated an 'engineered' technique in data augmentation (see Fig. 8 in the main manuscript) that expanded an original dataset with 1.4K image/mask pairs into a dataset of size 10,000, making it a contrastingly larger dataset than the water tank section. The motivation for this dataset was to evaluate the capacity of LivePyxel's binary mask to greatly automate the gathering of data of image/mask pairs for segmentation tasks, and to evaluate its effects in the capacity of the U-Net model to accurately predict and identify among the 4 different classes of shells.

4.1 Dataset Information

The augmented dataset was produced via data engineering, using the masks to extract pixels from the original images as described in Section 3.2 in the main manuscript. The produced dataset was weighted and prepared as described in Section 1. A total of 9K images were used for training, while 1K were used for validation purposes.

4.2 Model Architecture

The snail-shell dataset was trained on 10K images/mask pairs, each with an original resolution of 720×480 pixels. Training was carried out over 10 epochs. An additional set of 100 images, not

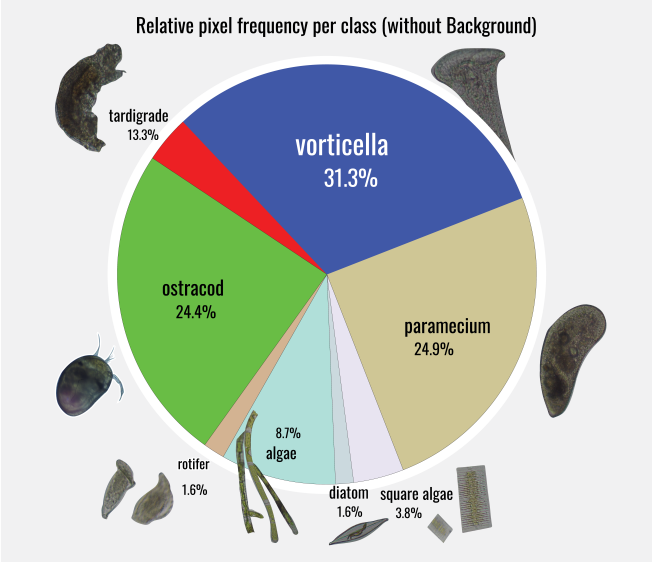

Fig. 2 Relative pixel frequency distribution for each annotated class in the EM dataset (N = 1,250 images), excluding background pixels (RGB 0,0,0). Percentages were computed from annotation masks after normalization, showing the proportional representation of *Vorticella*, *Ostracod*, *Paramecium*, and other taxa.

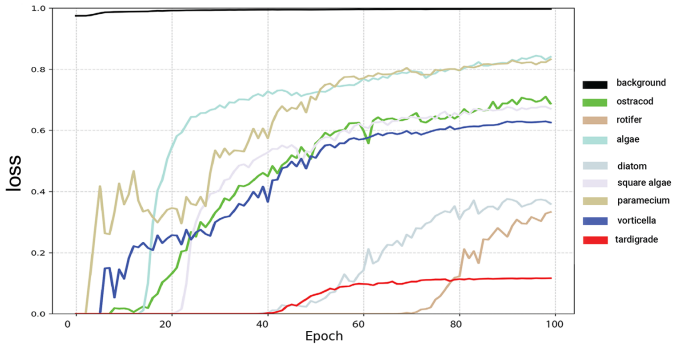

Fig. 3 F1 score progression during U-Net training for the nine annotated categories in the EM dataset. The model was initialized with weights from a pretrained VGG19 network. Each curve represents the per-class F1 score over 100 training epochs.

used during training or validation, were reserved for testing and detection. All training was performed using four NVIDIA V100 16 GB GPUs for a total duration of training that spanned 10 hours.

4.3 Training

The training and validation loss curves 5 indicate a rapid decrease in both losses during the first few epochs, followed by a steady convergence. By epoch 10, the training loss continues to decrease slightly, while the validation loss flattens, showing minimal signs of overfitting. This suggests that the model generalizes well within this training window. Furthermore, the F1 scores per class 6 improve sharply in the early epochs and stabilize above 0.95 for all classes in epoch 3, demonstrating consistent and balanced performance across shell categories. Based on this performance, we selected the final model weights of epoch 8-10, where both the loss and the F1 scores suggest optimal training without overfitting.

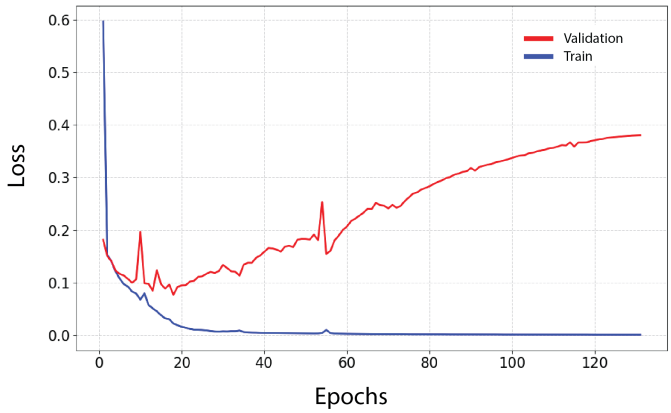

Fig. 4 Cross-entropy loss curves for the training and validation sets during U-Net model training on the water tank dataset. The rapid decrease in both losses during the initial epochs is followed by a divergence after approximately epoch 20, indicating the beginning of overfitting.

Index	Label	Color (RGB)
0	smooth_tiger	(0, 255, 0)
1	sierpinsky	(0, 47, 255)
2	silky	(255, 255, 19)
3	toque	(255, 64, 255)

Table 3 Labels and corresponding RGB color codes used in the segmentation task to train the Snail-shells dataset.

5 Benchmarking Annotation Tools

The benchmark comparing LivePyxel with other annotation tools was performed using a reference image containing five distinct polygonal shapes with varying curvature complexity (see the Original panel in Fig. 10 of the main paper). LivePyxel achieved balanced performance, with 5.7% false positives and 0.5% false negatives—comparable to other software. CVAT produced the lowest overall error (2.5% false positives, 1.3% false negatives) by leveraging an AI segmentation tool, but required a more complex setup and data management workflow. VIA and LabelMe showed slightly higher false-positive rates, while COCO Annotator exhibited the strongest tendency toward over-segmentation. To assess efficiency, we measured the time required to label this set of shapes across different annotation tools. The labeling times are reported in Table 4, and Fig. 8 displays zoomed views of the annotated boundary for five different annotation tools. Except for LivePyxel, which used Bézier splines to better capture curved contours, polygons were used in all other software. Notably, CVAT integrates the Segment Anything Model (SAM), which can accelerate the annotation process.

Data availability

LivePyxel and all trained vision models and data presented in this paper are freely available at `https://github.com/UGarCil/LivePyxel` and can be installed through PyPI. All datasets used in this study are available at `https://doi.org/10.5281/zenodo.17858610`.

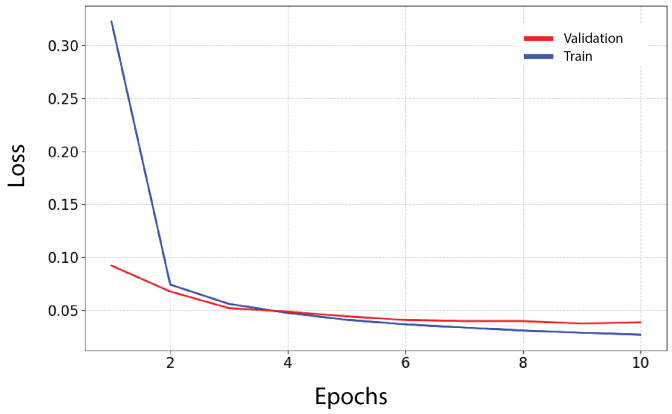

Fig. 5 Cross-entropy loss curves for the training and validation sets during 10 epochs of U-Net model training on the snail-shell dataset. Both losses decrease rapidly in the initial epochs and converge steadily, indicating good generalization without signs of overfitting.

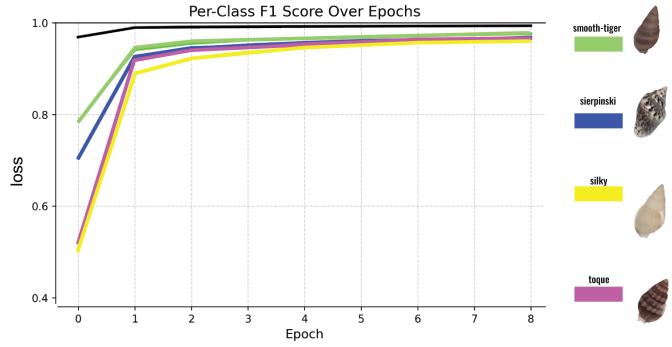

Fig. 6 Per-class F1 score progression during U-Net training on the snail-shell dataset. All categories show rapid improvement in the first two epochs, stabilizing above 0.95 by epoch 3, indicating balanced and consistent segmentation performance across classes.

Software	Time [s]
COCO	180
LabelMe	254
VIA	248
CVAT*	60
LivePyxel†	173

Table 4 Time taken to annotate Fig. 10 in the main paper. *CVAT incorporates SAM. † Bézier splines.

References

1 O. Ronneberger, P. Fischer and T. Brox, Medical image computing and computer-assisted intervention–MICCAI 2015: 18th international conference, Munich, Germany, October 5-9, 2015, proceedings, part III 18, 2015, pp. 234–241.

2 K. Simonyan and A. Zisserman, *arXiv preprint arXiv:1409.1556*, 2014, N/A.

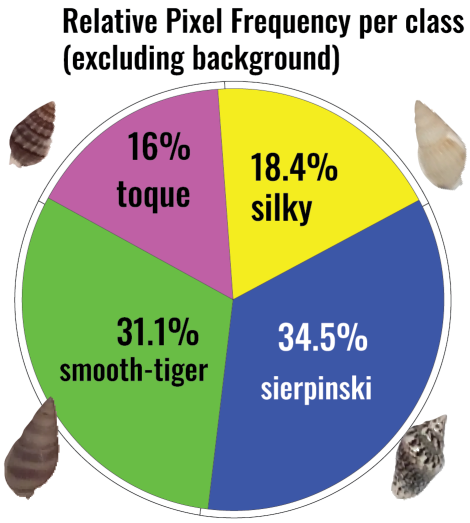

Fig. 7 Relative pixel frequency distribution (excluding background) for the four classes in the snail-shell dataset. Percentages were calculated from annotation masks after normalization, highlighting the proportional representation of each shell category.

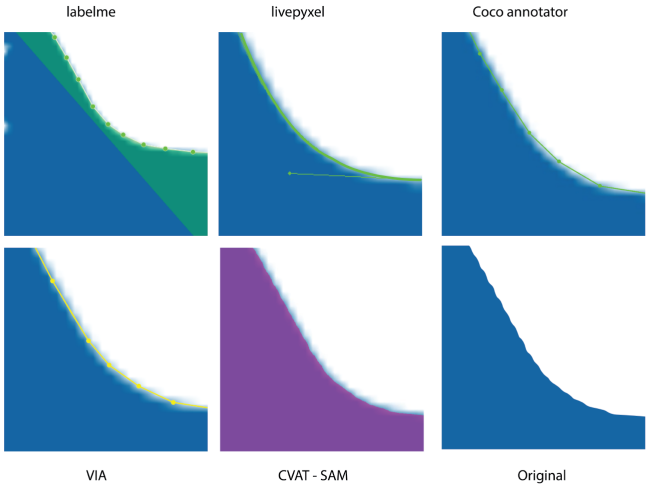

Fig. 8 Differences in boundary smoothness and contour accuracy illustrate the variability in mask generation between traditional polygons, splines, and AI-based annotation tools; LivePyxel uses Bézier splines, CVAT uses SAM, and Labelme, COCO, and VIA use polygons.