DEEPIMAGEJ: A USER-FRIENDLY PLUGIN TO RUN DEEP LEARNING MODELS IN IMAGEJ

A PREPRINT

Estibaliz Gómez-de-Mariscal[**]
[**]Equally contributed
Bioengineering and Aerospace Engineering Department
Universidad Carlos III de Madrid and Instituto de Investigación Sanitaria Gregorio Marañón, Spain

Carlos García-López-de-Haro[**]
[**]Equally contributed
Bioengineering and Aerospace Engineering Department
Universidad Carlos III de Madrid and Instituto de Investigación Sanitaria Gregorio Marañón, Spain

Laurène Donati
Biomedical Imaging Group
École polytechnique fédérale de Lausanne (EPFL)
Switzerland

Michael Unser
Biomedical Imaging Group
École polytechnique fédérale de Lausanne (EPFL)
Switzerland

Arrate Muñoz-Barrutia[*]
[*]Corresponding author
Bioengineering and Aerospace Engineering Department
Universidad Carlos III de Madrid and Instituto de Investigación Sanitaria Gregorio Marañón, Spain
mamunozb@ing.uc3m.es

Daniel Sage[*]
[*]Corresponding author
Biomedical Imaging Group
École polytechnique fédérale de Lausanne (EPFL)
Switzerland
daniel.sage@epfl.ch

October 16, 2019

ABSTRACT

DeepImageJ is a user-friendly plugin that enables the generic use in FIJI/ImageJ of pre-trained deep learning (DL) models provided by their developers. The plugin acts as a software layer between TensorFlow and FIJI/ImageJ, runs on a standard CPU-based computer and can be used without any DL expertise. Beyond its direct use, we expect DeepImageJ to contribute to the spread and assessment of DL models in life-sciences applications and bioimage informatics.

Deep learning (DL) models have a profound impact on a wide range of imaging applications, including life-sciences [1, 2, 3]. Unfortunately, their deployment is often riddled with technical challenges for the non-expert user. Their appropriate use requires insights on sophisticated DL concepts and good programming skills.

This situation is in sharp contrast with the philosophy behind ImageJ, the *de-facto* standard image processing software in life-sciences [4]. This open-source package gives biologists access to a wide variety of user-friendly image analysis tools through third-party plugins and macros.

Recent works have aimed at providing a link between TensorFlow [1] and ImageJ [2]. In particular, the CSBDeep team [5], the ImageJ2 team [6], and the Ozcan Research Group [7, 8] have pioneered this connection by making their pre-trained TensorFlow models accessible through ImageJ. Unfortunately, this connection effort has remained restricted to their specific applications.

We present DeepImageJ[3], an open-source plugin of ImageJ that runs a variety of third-party TensorFlow models in a generic way. Keras models can also be integrated whenever properly converted to TensorFlow's format SavedModels. DeepImageJ is designed as a standard ImageJ plugin with the technicalities hidden behind a user-friendly interface.

[1] https://www.tensorflow.org/
[2] https://imagej.nih.gov/ij/
[3] https://deepimagej.github.io/deepimagej/

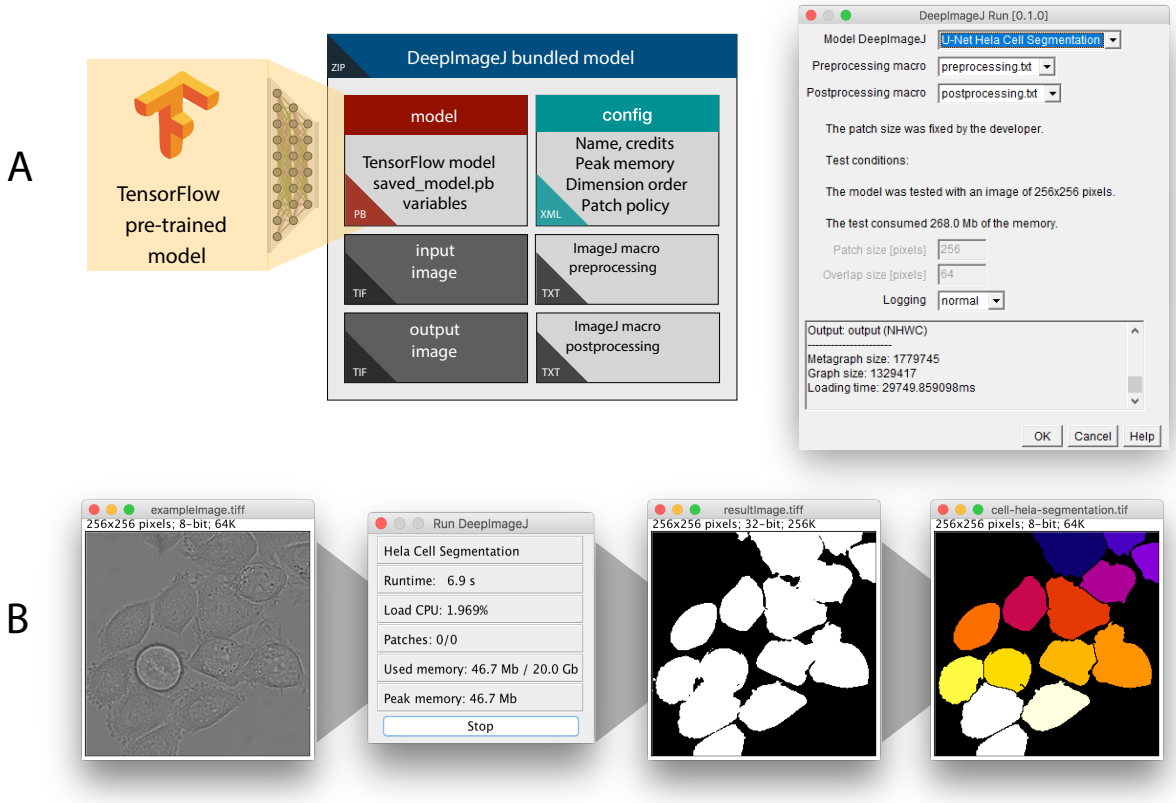


Figure 1: The DeepImageJ plugin. **Panel A**: Each pre-trained TensorFlow model (architecture and weights) is bundled with additional information into a zipped package in DeepImageJ (see Methods). The user can select a network in the list of models and load it in ImageJ. The user can then run the model on a selected image in one click. Pre- or post-processing options can also be added if desired. **Panel B**: Illustration of a typical DeepImageJ workflow with the `U-Net HeLa segmentation` model. From left to right: Input image, monitoring window, network output, post-processed image (optional).

Pre-trained models can be applied in a few clicks without DL expertise or programming skills, as illustrated in Figure 1. Furthermore, DeepImageJ can be called through ImageJ macros commands and applied to image stacks, which permits its inclusion in image analysis workflows. Finally, the plugin runs on a standard CPU-based computer and hence, does away with expensive GPU hardware.

DeepImageJ currently operates with image-to-image models: networks that take images as inputs and yield images as outputs. Many such models have been recently developed for biomedical imaging tasks such as deconvolution, segmentation and super-resolution. A first selection of such models is available in our online database, from which the user can easily download and apply the model of its choice in ImageJ in one click.

An important aspect to take into consideration when running DL models is image pre-processing (*e.g.*, data normalization). To handle this, DeepImageJ gives the user the flexibility to run any kind of pre- or post-processing routine. Moreover, the plugin can handle multi-channel images and images of arbitrary size whenever permitted by the pre-trained model (see Online Methods).

Most state-of-the-art DL models for image processing are convolutional neural networks (CNN) with numerous layers and scales. Consequently, the running of DL models is significantly costlier than most standard image-processing tasks in terms of computation and memory. Developers of DL networks deal with this by using fast GPU hardware; however, this expensive computational resource is scarcely available in life-science labs. With DeepImageJ, we made the choice to design a plugin that runs on standard CPU-based computers (including personal laptops), those being the computational tools most commonly used by ImageJ users.

The CPU-based Java executions in DeepImageJ are of comparable speed than the original CPU-based Python versions. Moreover, the computation time in DeepImageJ can be drastically reduced by processing images by large tiles whenever possible. For example, DeepImageJ runs `Stardist` in 4.8 seconds when an image of size 256×256 pixels is processed

in tiles of size 128×128 pixels with a padding of 22×22 pixels. However, if the tile covers the entire image (256×256 pixels with no padding), the computation time is reduced to 1.5 seconds. The runtime of the CPU-Python code is 2.62 and 0.63 seconds, respectively. In addition, DeepImageJ provides the user an indication of the peak memory required for processing an image of a given size; memory consumption can thus be accounted for. The models were run on an Intel(R) Core(TM) i7-4712 CPU @ 2.30GHz, 8.0GB (RAM), 64-bit Operating System, x64 based processor machine with Windows 10 operating system.

We validated DeepImageJ by successfully running a selection of pre-trained models in ImageJ. Outputs of these runs are presented in Figure 2. The tested models include home-trained networks and external openly-distributed models (see Table 1). The validation procedure covered applications ranging from super-resolution to segmentation.

Name of the bundled model	Image processing task	Microscopy modality (sample)	Runtime* [sec] (for an image size in pixels)	Source of the TensorFlow network
`FRU-Net sEVs segmentation` [9]	Instance segmentation	TEM (sEV)	174 (1264×1264)	Data: MUNI Training: UC3M
`Noise2Void denoising` [10]	Denoising	Fluorescence; cryo-TEM (Cell culture; sub-cellular structures)	10 (512×512)	CSB Dresden
`Stardist nuclei detection` [11]	Instance segmentation	Fluorescence (Cell culture)	5 (320×256)	CSB Dresden
`CARE isotropic reconstruction` [5]	Isotropic reconstruction	Fluorescence (Zebrafish embryo)	12 (512×512)	CSB Dresden
`CARE deconvolution microtubules` [5]	Deconvolution	Fluorescence (Sub-cellular structures)	8 (1024×1024)	CSB Dresden
`U-Net HeLa segmentation` [12]	Instance segmentation	DIC (Cell culture)	6 (256×256)	Data: CTC Training: BIG EPFL
`U-Net glioblastoma segmentation` [12]	Instance segmentation	Phase contrast (Cell culture)	15 (348×260)	Data: CTC Training: BIG EPFL
`MT3 virtual staining` [7]	Virtual labelling	Light transmission (Kidney tissue)	200 (1224×1224)	ORG UCLA
`Jones virtual staining` [7]	Virtual labelling	Light transmission (Liver tissue)	255 (1224×1224)	ORG UCLA
`Widefield FITC super-resolution` [8]	Super-resolution	Fluorescence (Cell culture)	30 (1024×1024)	ORG UCLA
`Widefield DAPI super-resolution` [8]	Super-resolution	Fluorescence (Cell culture)	30 (1024×1024)	ORG UCLA

Table 1: Available DeepImageJ bundled models and information on the computational cost `https://deepimagej.github.io/deepimagej/`. Note: sEVs: small Extracellular Vesicles; FITC: Fluorescein IsoTioCyanate; DAPI: DNA-specific fluorescent probe; TEM: Transmission Electron Microscopy; DIC: Differential Interference Contrast; MUNI: Masaryk University; UC3M: Universidad Carlos III de Madrid; CSBD: Center for Systems Biology Dresden; CTC: Cell Tracking Challenge `http://celltrackingchallenge.net/` [13, 14]; BIG EPFL: Biomedical Imaging Group EPFL; ORG UCLA: Ozcan Research Group, UCLA. *Run on an Intel(R) Core(TM) i7-4712 CPU @ 2.30GHz, 8.0GB (RAM), 64-bit Operating System, x64 based processor machine with Windows 10 operating system.

Although we made every effort to lay solid foundations to DeepImageJ plugin, the correctness of a model's output still depends on its appropriate usage. Hence, it is critical that the user pays close attention to the information provided by the DL developers (*e.g.*, required pre-processing steps, image format) before running a model.

Finally, we provided developers with a software tool to bundle their TensorFlow model(s) and necessary additional information about DeepImageJ (see Methods and Supplementary Information). Hence, developers can disseminate their models to the community at large through DeepImageJ, which we hope will encourage constructive feedback and extensive validation of new models. A more systematic procedure to directly upload and assess DL models in the online database will be made available in the near future.

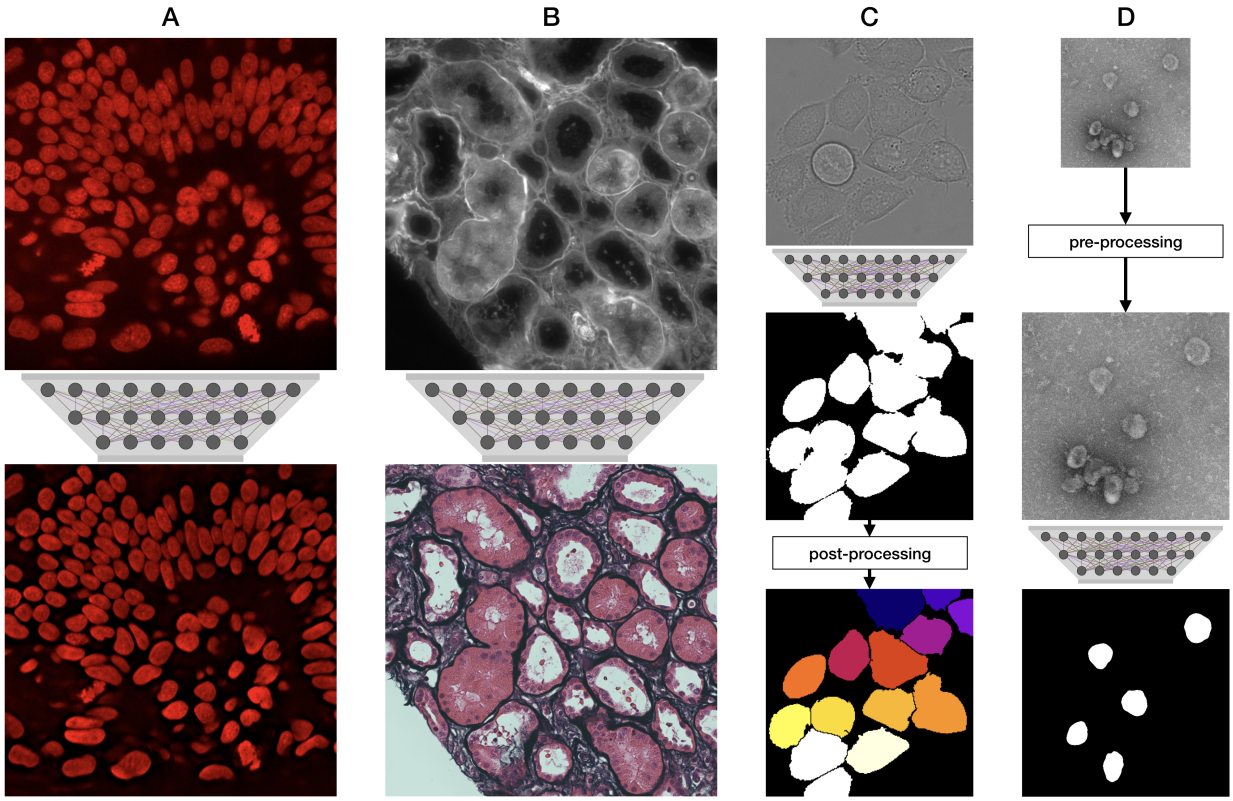


Figure 2: Illustrations of four DeepImageJ runs. **A.** Isotropic resolution recovery in a fluorescence microscopy image (zebrafish retinal cells) with `CARE` [8]. **B.** Virtual staining of unlabelled kidney cells with `VirtualStain` [7]. **C.** Segmentation of HeLa cells in DIC images with a home-trained `U-Net` [12]. The cells were labelled after segmentation with a personalised macro. **D.** Segmentation of extracellular vesicles in TEM images with a home-trained fully residual `U-Net` [9]. The input image was rescaled prior to segmentation.

Online methods

Software/Network compatibility

DeepImageJ is compatible with both FIJI and ImageJ. It is self-sufficient on any operating system: MacOS, Linux and Windows. It supports models from TensorFlow version 1.13 or lower. Keras version 2 or lower is also supported[4] as long as the are compatible with Tensorflow version 1.13 or lower.

DeepImageJ bundled models

DeepImageJ processes folders (models) that contain the files described in Table 2. Those files are usually written by the author of the model, which makes the bundled model self-sufficient. Their content is described in the next paragraph. Further details about loading bundled models in DeepImageJ are given in the Supplementary Material.

Any TensorFlow [15] model is determined by a graph (the architecture of the network) and its weights (specific values for all the parameters in the network obtained after training). The TensorFlow's Java API is only compatible with the SavedModel format which is obtained using an in-house Python routine[5]. Namely, the DeepImageJ models are defined by a Protocol Buffer format file (called `saved_model.pb`) that contains the architecture of the model and a series of text files storing the weights that are kept in a folder called `variables`. ImageJ macros (`preprocessing.txt` and `postprocessing.txt`) are optional processing steps. The pre-processing macro transforms the image into the specific input type for which the model was trained. Typical pre-processing operations are normalization of the pixel intensity values, change of the bit depth and image resizing. The post-processing macro curates the output of the network. Optional post-processing operators are thresholding, resizing and extraction of objects features. Two images, an input (`exampleImage.tiff`) and an output example (`resultImage.tiff`), are also stored in each of the bundled model folder to facilitate model testing.

The configuration file (`config.xml`) has descriptive information about the model:

- General information: Name of the author(s), title, reference to the publication.
- Estimated execution time on the PC with the configuration described in the caption of Table 1.
- Minimum amount of memory required to process the example input image.
- Technical characteristics of the model: Input and output size, dimensions order (height, width, and channels in the first, second and third dimension, respectively), minimum size and padding. See Online Methods, Image size & Tiling strategy.

File name	Content
`saved_model.pb` variables	TensorFlow model in Protocol Buffer format.
preprocessing.txt postprocessing.txt	ImageJ macros written by the model's author.
exampleImage.tif resultImage.tif	Example of input and output images.
config.xml	Details about the related publication, technical characteristics of the model, execution time and required memory.

Table 2: List of the required files to run a model in DeepImageJ.

Image size and tiling strategy

The model developer needs to specify the following information when uploading their Convolutional Neural Network (CNN) model:

Q, I: Whether the input size of the model (Q) is predetermined or not. If it is predetermined, Q needs to be provided, and it will be compared with the size of the image to process (I).

[4] `https://github.com/deepimagej/python4deepimagej/`
[5] `https://github.com/deepimagej/python4deepimagej/`

m: If the network has an auto-encoder architecture, the size of each dimension of the input image, has to be multiple of a minimum size m defined as $m = p^d$ where d is the number of poolings (down-sampling operations) and p their size.

P: To preserve the input size at the output, convolutions are usually calculated using zero padding boundary conditions (Figure 3). Namely, additional void pixel values are added along the borders of the image. Hence, the size (per dimension) of the receptive field of a convolution (the valid domain of the output) is given by $R = Q - 2P$ with Q, the model input size and P, the size of the network padding. It is computed as

$$P = \sum_{i=1}^{n} \left(\frac{k_i - 1}{2} \right). \tag{1}$$

where n is the number of convolutional layers and k_i is the size of the kernel in each convolutional layer i. Usually, k_i is an odd number. If the kernel is not square, then P and R have different values on each dimension.

To handle input images with a large size, DeepImageJ follows a common strategy called tiling:

- If the network has not a predetermined input size (Q), the algorithm calculates what is the smallest multiple s of the minimum size m that is still larger or equal to the size of the input image I and the total padding $2P$:

$$s = \underset{\substack{s \in \mathbb{N} - \{0\} \\ mod(s,m)=0}}{\operatorname{argmin}} \{s \geq (I + 2P)\} \tag{2}$$

  Then, the image is augmented by mirroring along the borders up to a size s per dimension, and it is processed. Finally the output is cropped to the initial size I. See Figure 3 for an illustration.

- If the network input size (Q) is predetermined, then the algorithm compares the size of the image (I) with it taking into account the padding (P). If it is smaller ($I + 2P \leq Q$), then the image is augmented by mirroring until the desired size Q is reached. If the opposite is true ($I + 2P > Q$), the optimal number of patches to process (p) is calculated as follows:

$$p = ceil \left(\frac{I}{Q - 2P} \right) \tag{3}$$

  where the function $ceil(x)$ outputs the smallest integer number that is equal or larger than x. Note that p can vary on each dimension. Then, the image will be covered by patches of size

$$T = floor \left(\frac{I}{p} \right) \tag{4}$$

  where the function $floor(x)$ outputs the largest integer number that is equal or smaller than x and $T \leq (Q - 2P)$. From each processed patch of size Q, a patch of size T is cropped and placed accordingly to reconstruct a valid output (tiling strategy). The patches along the borders are filled by mirroring as shown in Figure 3. As the quotient in Equation 4 may not be an entire number, the last patch on each dimension has exactly size $I - (p - 1)T$.

Both the input size of the network (Q) and the receptive field (R) are critical parameters for good results and they are directly related to the time spent by the plugin to process one image. Large input images and small receptive fields imply longer computations.

Example of a complete image analysis pipeline

In the following lines, we describe a generic workflow for image analysis using DL models and DeepImageJ plugin (see Figure 4). We use a toy example in which two U-Net [12] models are trained to segment cells on 2D phase contrast microscopy images. The entire code was written in Python using the Keras library[6] and it was run in the Google Colaboratory environment, which supplies a free Tesla K80 GPU service. The code is freely distributed[7].

The data used was made available by the Cell Tracking Challenge initiative (CTC)[8] [13, 14]. They are $2D$ phase contrast microscopy videos of HeLa cells cultured on a flat glass and Glioblastoma-astrocytoma U373 cells grown on a

[6] https://github.com/zhixuhao/unet

[7] https://github.com/deepimagej/python4deepimagej/

[8] http://celltrackingchallenge.net/

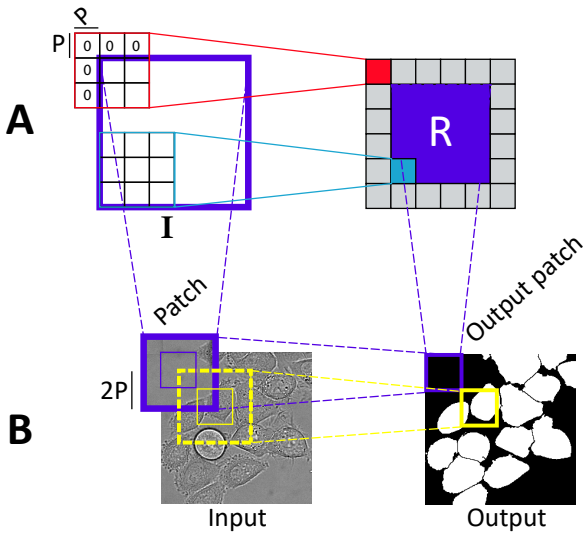


Figure 3: Tiling strategy with convolutions. **Panel A:** Convolution of size 3×3 pixels. The receptive field (R) of a convolution (blue squares) is smaller than its input and it is determined by the size of the padding (red operation). **Panel B:** Image processing using tiles (patches). The overlap $2P$ is calculated to avoid artifacts in the reconstruction of the output.

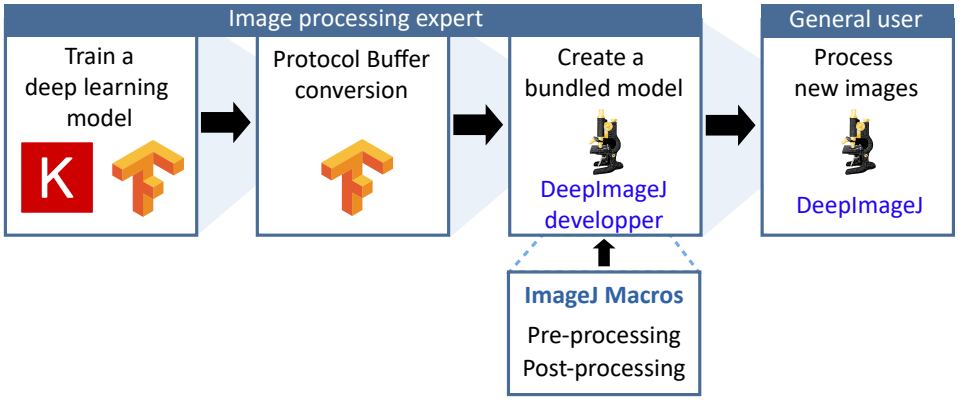


Figure 4: Proposed pipeline for image analysis using deep-learning models and DeepImageJ plugin.

polyacrylamide substrate. We refer to the model trained on the HeLa cells as `U-Net HeLa segmentation` and on the U373 cells as `U-Net glioblastoma segmentation`. Only a small portion of the data was chosen to train and test the models (see Table 3 for details). Moreover, the image size was halved to shorten the computational time during training. The Keras `ImageDataGenerator` class was used to perform data augmentation with random rotations of $\pm 40°$, shifts, shear deformations, zooms, vertical and horizontal flips.

Models were trained with the binary cross-entropy loss function, a learning rate of $1e^{-04}$ and a weight decay of $5e^{-07}$ during 10 epochs of 500 steps each. Altogether, it took 17 minutes to train each of them. The model finally chosen in both cases was the one that resulted in the lowest validation loss during training. The probability output maps from the inference were thresholded at 0.5 to get the final binary masks. The segmentation accuracy was assessed using the percentage of correct pixel assignments (accuracy) and the Jaccard index as computed by the code provided at the CTC web page (SEG) [13, 14]. Table 3 summarizes the segmentation accuracy results.

A short script along with the code translates `U-Net HeLa segmentation` and `U-Net glioblastoma segmentation` models from the widely used Keras format HDF5 (.hdf5) to the TensorFlow SavedModel one. Note that the script can be easily adapted to translate other models given in Keras format. Then, the generated models were easily converted to DeepImageJ bundled models using the provided builder module (DeepImageJ Build Bundled Model). Subsequently, the models were loaded in FIJI/ImageJ so they could be applied to the rest of the images. An

Name of the bundled model	Images (training)	Images (test)	Loss (training)	Loss (test)	Accuracy (training)	Accuracy (test)	SEG (CTC)	Python runtime* [sec]
`U-Net HeLa segmentation`	8	9	0.245	0.231	0.900	0.905	0.830	9
`U-Net glioblastoma segmentation`	24	10	0.061	0.054	0.985	0.984	0.795	19

Table 3: Summary of U-Net training. *Run on an Intel(R) Core(TM) i7-4712 CPU @ 2.30GHz, 8.0GB (RAM), 64-bit Operating System, x64 based processor machine with Windows 10 operating system.

optional post-processing macro to analyze all segmented objects was also implemented and it is provided with the rest of the code. Due to the large overlap between cells, the output binary mask from the `U-Net HeLa segmentation` model was first processed using a Watershed transform to split cellular clusters. Then, both models output an image of uniquely labelled cells that were further processed using the implemented user interface for object analysis. The described DeepImageJ workflow with the `U-Net HeLa segmentation` model is illustrated in panel B of Figure 1.

Data availability statement

The web page: `https://deepimagej.github.io/deepimagej/` provides free access to the plugin, along with the bundled models and user guide for image processing.

Acknowledgements

We would like to thank João Luis Soares Lopes, Rémy Pétremand and Halima Hannah Schede from École polytechnique fédérale de Lausanne (EPFL) for writing the complete Python pipeline and providing ready-to-use U-Net models. Daniel Wüstner from the University of Southern Denmark, Odense, provided membrane fluorescence images to test Noise2Void model. We would like also to thank Pedro M. Gordaliza for fruitfully discussions.

This work is partially supported by the Spanish Ministry of Economy and Competitiveness (TEC2015-73064-EXP, TEC2016–78052-R) and by a 2017 Leonardo Grant for Researchers and Cultural Creators, BBVA Foundation. This work is part of the EPFL initiative "imaging@EPFL". We thanks the program "Short Term Scientific Missions" of NEUBIAS (network of European bioimage analysists). We also want to acknowledge the support of NVIDIA Corporation with the donation of the Titan X (Pascal) GPU card used for this research.

Author contributions

E.G.M. and C.G.L.H. contributed to the design of the experimental framework, reviewed, trained and exported existing image processing methods. C.G.L.H. and D.S. developed and implemented the plugin and worked on the supporting documentation with input from the rest of the authors. E.G.M. and L.D. wrote the manuscript with help from A.M.B. and D. S.. E.G.M., A.M.B. and D.S. created the web page dedicated to the plugin. All the authors contributed to the conception of the study, the design of the experimental framework and took part in the literature review. All authors revised the manuscript.

Competing interests

The authors declare that they have no competing interests.

References

[1] George Barbastathis, Aydogan Ozcan, and Guohai Situ. On the use of deep learning for computational imaging. *Optica*, 6(8):921–943, Aug 2019.

[2] Fuyong Xing, Yuanpu Xie, Hai Su, Fujun Liu, and Lin Yang. Deep Learning in Microscopy Image Analysis: A Survey. *IEEE Trans. Neural Networks Learn. Syst.*, 29(10):4550–4568, oct 2018.

[3] Geert Litjens, Thijs Kooi, Babak Ehteshami Bejnordi, Arnaud Arindra Adiyoso Setio, Francesco Ciompi, Mohsen Ghafoorian, Jeroen A.W.M. van der Laak, Bram van Ginneken, and Clara I. Sánchez. A survey on deep learning in medical image analysis. *Med. Image Anal.*, 42(December 2012):60–88, dec 2017.

[4] Caroline A Schneider, Wayne S Rasband, and Kevin W Eliceiri. NIH Image to ImageJ: 25 years of image analysis. *Nat. Methods*, 9(7):671–675, jul 2012.

[5] Martin Weigert, Uwe Schmidt, Tobias Boothe, Andreas Müller, Alexandr Dibrov, Akanksha Jain, Benjamin Wilhelm, Deborah Schmidt, Coleman Broaddus, Siân Culley, Mauricio Rocha-Martins, Fabián Segovia-Miranda, Caren Norden, Ricardo Henriques, Marino Zerial, Michele Solimena, Jochen Rink, Pavel Tomancak, Loic Royer, Florian Jug, and Eugene W. Myers. Content-aware image restoration: pushing the limits of fluorescence microscopy. *Nat. Methods*, 15(12):1090–1097, dec 2018.

[6] Samuel J. Yang, Marc Berndl, D. Michael Ando, Mariya Barch, Arunachalam Narayanaswamy, Eric Christiansen, Stephan Hoyer, Chris Roat, Jane Hung, Curtis T. Rueden, Asim Shankar, Steven Finkbeiner, and Philip Nelson. Assessing microscope image focus quality with deep learning. *BMC Bioinformatics*, 19:77, 2018.

[7] Yair Rivenson, Hongda Wang, Zhensong Wei, Kevin de Haan, Yibo Zhang, Yichen Wu, Harun Günaydın, Jonathan E. Zuckerman, Thomas Chong, Anthony E. Sisk, Lindsey M. Westbrook, W. Dean Wallace, and Aydogan Ozcan. Virtual histological staining of unlabelled tissue-autofluorescence images via deep learning. *Nat. Biomed. Eng.*, 3(6):466–477, jun 2019.

[8] Hongda Wang, Yair Rivenson, Yiyin Jin, Zhensong Wei, Ronald Gao, Harun Günaydın, Laurent A. Bentolila, Comert Kural, and Aydogan Ozcan. Deep learning enables cross-modality super-resolution in fluorescence microscopy. *Nat. Methods*, 16(1):103–110, jan 2019.

[9] Estibaliz Gómez-de Mariscal, Martin Maška, Anna Kotrbová, Vendula Pospíchalová, Pavel Matula, and Arrate Muñoz Barrutia. Deep-learning-based segmentation of small extracellular vesicles in transmission electron microscopy images. *Scientific Reports*, 2019.

[10] Alexander Krull, Tim-Oliver Buchholz, and Florian Jug. Noise2void-learning denoising from single noisy images. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2129–2137, 2019.

[11] Uwe Schmidt, Martin Weigert, Coleman Broaddus, and Gene Myers. Cell detection with star-convex polygons. In *Medical Image Computing and Computer Assisted Intervention - MICCAI 2018 - 21st International Conference, Granada, Spain, September 16-20, 2018, Proceedings, Part II*, pages 265–273, 2018.

[12] Thorsten Falk, Dominic Mai, Robert Bensch, Özgün Çiçek, Ahmed Abdulkadir, Yassine Marrakchi, Anton Böhm, Jan Deubner, Zoe Jäckel, Katharina Seiwald, Alexander Dovzhenko, Olaf Tietz, Cristina Dal Bosco, Sean Walsh, Deniz Saltukoglu, Tuan Leng Tay, Marco Prinz, Klaus Palme, Matias Simons, Ilka Diester, Thomas Brox, and Olaf Ronneberger. U-Net: deep learning for cell counting, detection, and morphometry. *Nat. Methods*, 16(1):67–70, jan 2019.

[13] Martin Maška, Vladimír Ulman, David Svoboda, Pavel Matula, Petr Matula, Cristina Ederra, Ainhoa Urbiola, Tomás España, Subramanian Venkatesan, Deepak MW Balak, et al. A benchmark for comparison of cell tracking algorithms. *Bioinformatics*, 30(11):1609–1617, 2014.

[14] Vladimír Ulman, Martin Maška, Klas EG Magnusson, Olaf Ronneberger, Carsten Haubold, Nathalie Harder, Pavel Matula, Petr Matula, David Svoboda, Miroslav Radojevic, et al. An objective comparison of cell-tracking algorithms. *Nature methods*, 14(12):1141, 2017.

[15] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: Large-scale machine learning on heterogeneous systems. *arXiv*, 1603.04467, 2016.

Supplementary Information

In this section, we have included the User Manual of the three modules delivered within the DeepImageJ plugin.

DeepImageJ Install

Download the plugin (`DeepImageJ.zip`) from the DeepImageJ's web site[9]. Unzip it and store the 5 .jar files it contains inside the plugins folder of FIJI/ImageJ (".../ImageJ/plugins/" or ".../Fiji/plugins/"). These files can also be dragged and dropped directly into FIJI/ImageJ. Create a new folder (`models`) in /ImageJ/ or /Fiji/. Finally, restart FIJI/ImageJ to see the new plugin: `ImageJ > Plugins > DeepImageJ`. See Figure 5a.

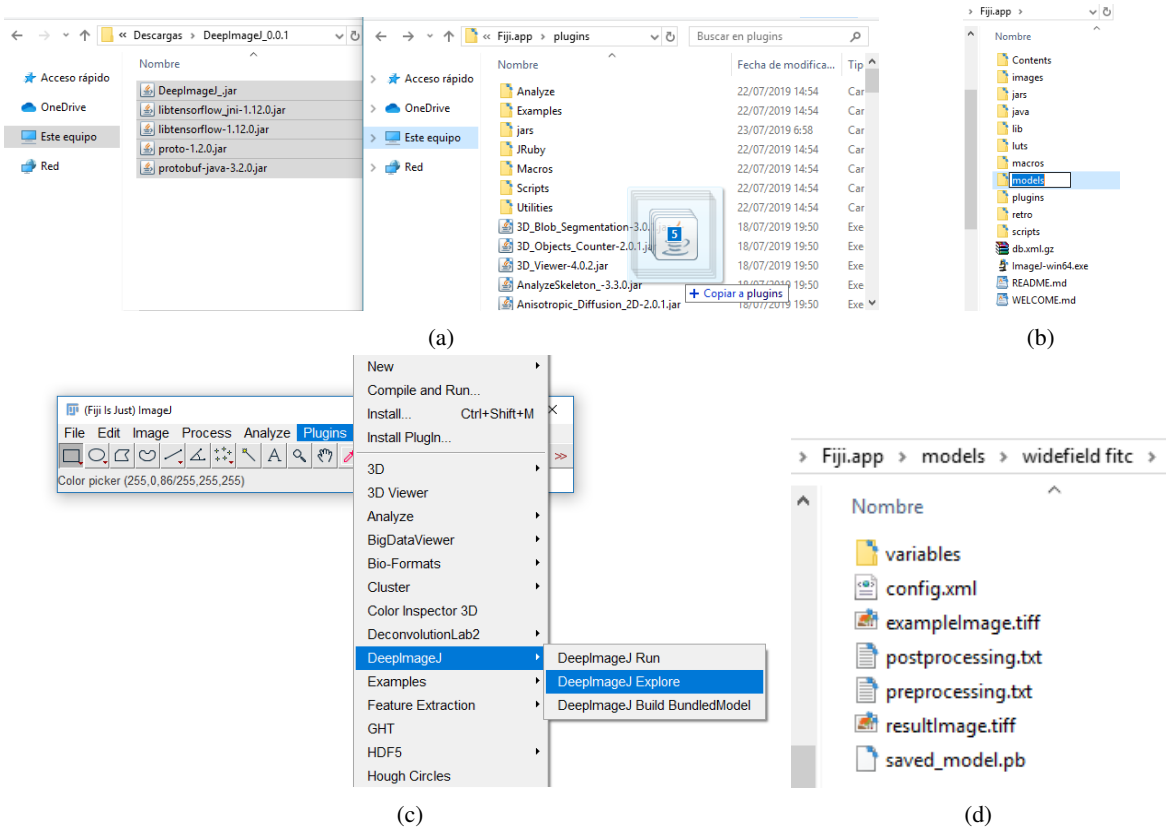


Figure 5: Installation steps: (a) Copy all the files contained in the zip file to the plugins folder. (b) Create a new folder called `models` to store the models. (c) Open FIJI/ImageJ and run DeepImageJ. (d) Example of a valid model for DeepImageJ.

There are three different options to work with DeepImageJ:

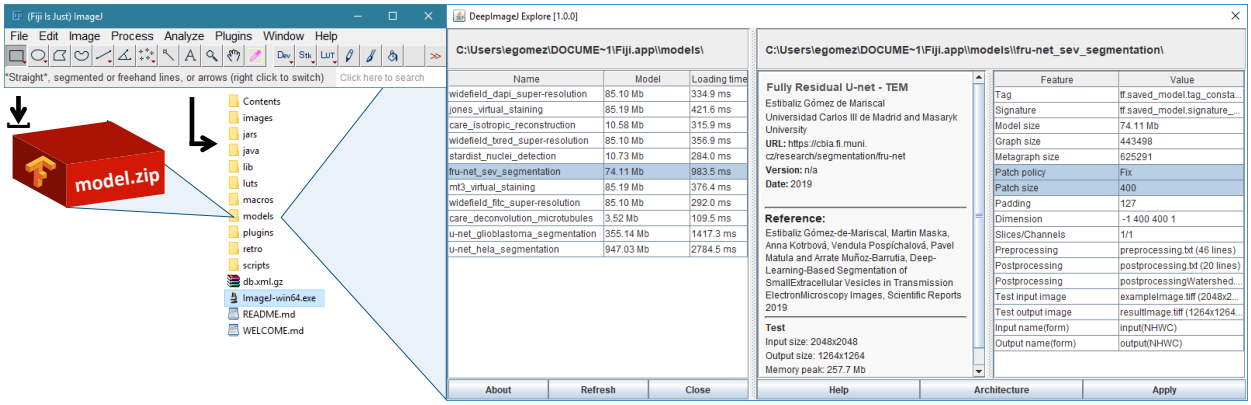
- Run: Runs a specific model to process an image already opened in FIJI/ImageJ.
- Explore: Displays the models available in the local machine.
- Build Bundled Model: user interface to convert TensorFlow SavedModels models into DeepImageJ bundled models. .

Both DeepImageJ modalities look for the folder called `models` inside FIJI/ImageJ directory and load the valid models. See Online Methods. Example models can be downloaded from DeepImageJ's web site as a zip file. Once it is unziped, move the inner folder to `models`. Please, make sure that it looks like in Figure 5d.

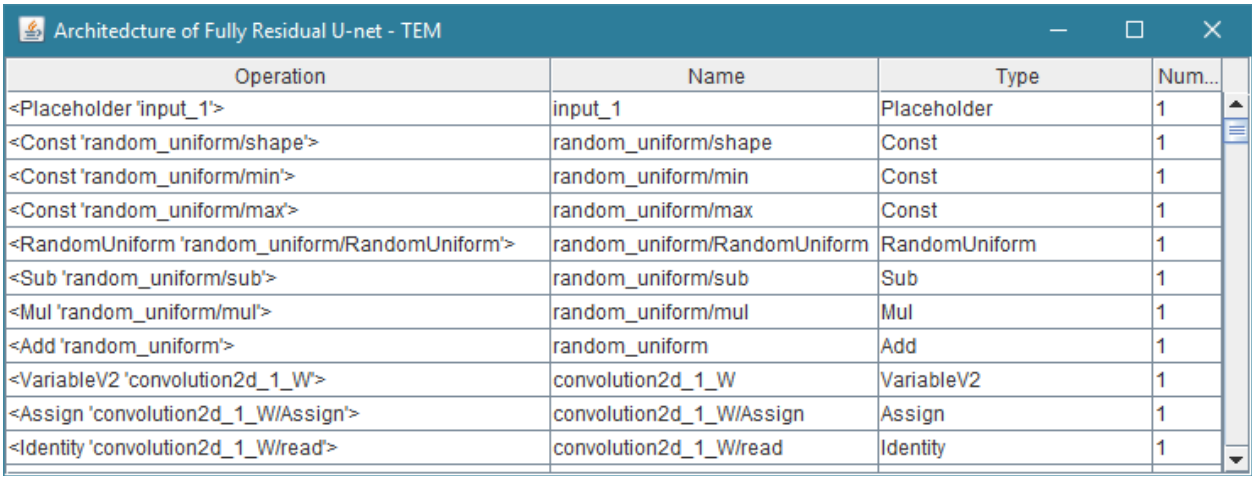
DeepImageJ Run

The steps to make inference with a model over an image is as follows:

[9] `https://deepimagej.github.io/deepimagej/`



(a)



(b)

Figure 6: (a) DeepImageJ reads the information from each model and displays it in a new window. (b) When clicking on `Architecture` the operations performed by the network are given.

1. Open the image to process.
2. Click on `ImageJ > DeepImageJ > Run`.
3. Choose the model to use.
4. Choose whether pre- and post-processing should be applied to the image.
5. Write the patch and padding size. If you do not have the means to know these values, use the values given by default. See the Online Methods for technical details.
6. Click 'Ok'.

The running window displayed during inference shows in real time the following information: A chronometer, the number of already processed patches, the current memory usage, the memory allocated in FIJI/ImageJ. If the allocated memory is smaller than the peak memory (see Online Methods), it will not be possible to run the model. This issue can be solved by increasing the allocated memory in FIJI/ImageJ[10]. Once the network run has been finalized, the input and output images are available for the user to further manipulate them or just to store the result.

DeepImageJ Explorer

A quick snapshot of the available models is obtained with `ImageJ > Plugins > DeepImageJ > DeepImageJ Explore` (see Figure 6a). When clicking on `Apply` in this window, the main processing workflow (Figure 1 in the main manuscript) is repeated automatically using the example image provided in the bundled model. If the user clicks on `Architecture`, the plugin recovers technical details about the network architecture, Figure 6b.

[10] `https://imagejdocu.tudor.lu/faq/technical/how_do_i_increase_the_memory_in_imagej`

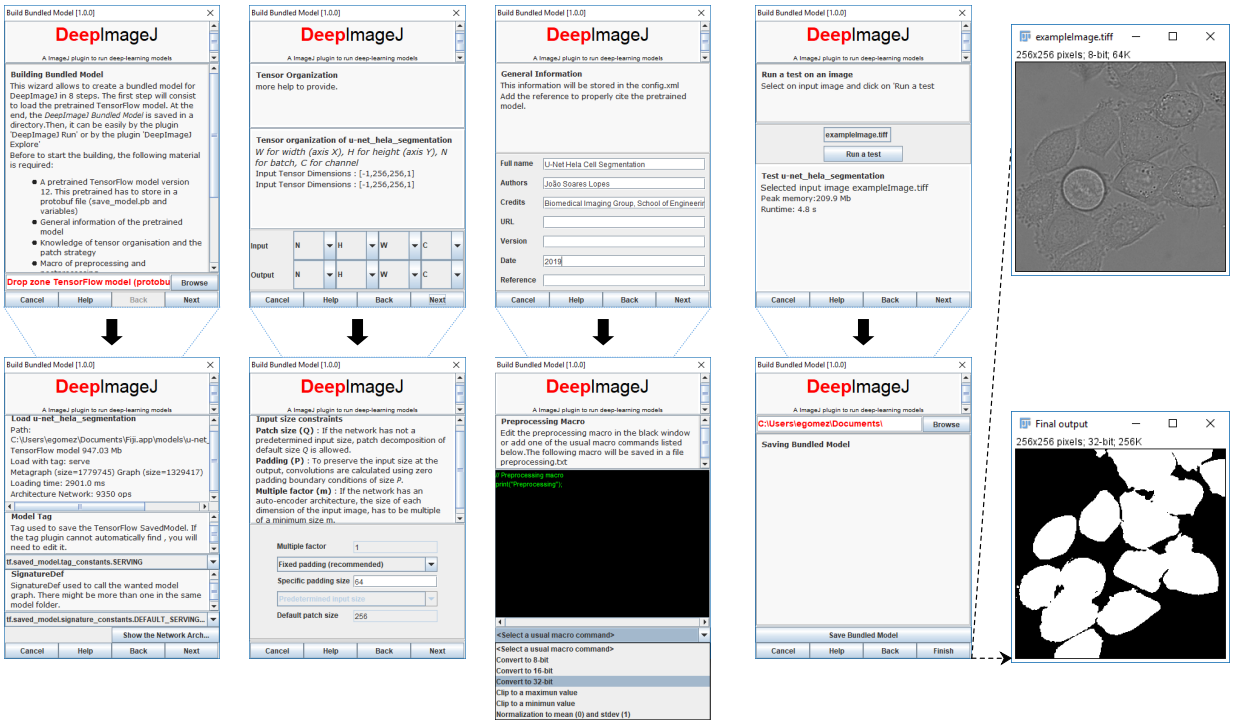


Figure 7: The developer module of DeepImageJ loads a model from the specified path to the SavedModel directory (variables folder and `saved_model.pb` file) and it asks for TensorFlow session tag constant name. Once the model is loaded, input and output dimension order (N: batch number, H: height, W: width, C: channels), input size and padding need to be specified. The developer may also include optional pre- and post-processing macro routines. A sample image must be provided, so DeepImageJ can test the model. If the output is correct, the model is ready to be exported as a DeepImageJ bundled model.

DeepImageJ Build Bundled Models

The developer module of DeepImageJ has a user-friendly interface in FIJI/ImageJ that allows to convert TensorFlow SavedModel models into DeepImageJ compatible format (bundled models) (see Figure 7). The input is a directory with variables sub-folder and `saved_model.pb` file. The output is the bundled model described in the online methods.