École Polytechnique Fédérale de Lausanne
School of Life Sciences

# Towards Reconstructing Graph-Based Models of Neuronal Arborizations from 3D Microscopy Scans

by Raphaël Mariétan

Master Thesis
Computer Vision Laboratory, EPFL
June 10, 2022

Prof. Pascal Fua & Dr. Mateusz Kozinski
Thesis Advisors

Richard Lengagne
External Expert

Doruk Oner
Thesis Supervisor

**Abstract**

Reconstruction of neuron morphologies has been a longstanding challenge for the computer vision community. The broad spectrum of its applications, ranging from providing more accurate neuron simulation models to neurological disorders identification, highlights the multidisciplinary aspect of this research topic and its significant potential to help science move forward. Considerable effort has been put towards developing tools that can reduce the excessive amount of manual input that has to be provided to complete a reconstruction. However, current methods have struggled to revolutionize the field, mostly failing to keep the tree topology of the reconstructions nor providing the output in the form of a graph, preferred for many uses.

To this end, we propose a new approach to the neuron tracing problem that can be generalized to other applications such as vasculature and road network extraction. The designed method outputs a graph-based reconstruction, which provides branch-specific information, extracts the bifurcation points and preserves the tree structure of the reconstructed neuron (similar to Figure 1). We design a model that can easily be integrated to an annotation pipeline, in which a Reinforcement Learning agent is trained to navigate through neuron images and trace the underlying branch trajectories. We show that the proposed approach can be adapted to work on synthetic data, first by applying in two dimensions, and then by demonstrating its applicability to neuronal structures in three dimensions. The approach is finally tested on predictions coming from real neuron microscopy scans. The agent is able to move along neurites on real data, but missing information and significant gaps in the annotations expose the need for more accurate annotations for the complete assessment of the applicability of the proposed approach to real data use cases.

Figure 1: A reconstructed neuron and its neurites displayed with different colors.

# Table of Contents

# 1 Introduction

Along with the recent technological progress in the field of microscopy, the scientific community has been able to constantly push further the limit in size of the structures that could be observed. As the milestone of acquiring image representations of objects on the nanoscopic scale can be achieved with multiple forms of microscopy, we are already able to observe most structures that are relevant to our current understanding of the human body. The scientific research in biomedical imaging is now at a turning point where it is not limited anymore by the acquisition of images, but by the ability to exploit the extensive amount of information that is concealed in such images.

At the origin of both the most basic and complex processes that provide the human body with the ability to function, the nervous system has been attracting lots of interest from the computer vision community for many years. Imaging of such neural structures has been available for years, but the reconstruction of such complex networks of connected curves from the images can, to this day, still not be efficiently retrieved. Nonetheless, acquiring information about the structures of neurons is crucial to many research subjects, in particular to the study and diagnosis of some poorly understood diseases such as Alzheimer's disease, Parkinson's disease, Amyotrophic Lateral Sclerosis, and other neurological disorders. Neurons are not self-sufficient structures, and their functions highly depend on their connectivity with other pools of neurons or areas of the brain. Dysfunctions in the brain can be the result of slight modifications in the connectome. Therefore, understanding the brain-wise connectivity of neural structures is essential to uncover more about the mechanisms of most brain disorders.

With the future prospect of learning more about the underlying wiring diagram of the brain, reconstruction of individual neurons has to be achieved in the first place. Reconstructing single neurons in itself is also a relevant topic when it comes to neuron morphology classification and analysis of neuron populations across the brain (Peng et al., 2020). However, the complete reconstruction of a single neuron still requires manual intervention to this day, whether it is to correct the output of automatic reconstruction methods, or to assist semi-automatic methods. The reconstruction of a mouse neuron from a microscopy scan can take a very restrictive 8 hours of work, even with the use of an advanced semi-automatic tool (Winnubst et al., 2019). The time and resources that have to be allocated for the reconstruction of a network of a few hundred neurons could quickly add up to exceed the funds of a moderate sized laboratory. For all its potential use cases and the possibility to generalize to other systems such as the vascular system, the need for efficient automatization of neuron reconstruction is real and could save a lot of time and resources that have to be spent acquiring reliable annotations.

Similarly to other problems in the field of image detection, various semi-automatic and automatic approaches involving Deep Learning have been tested on the neuron tracing or similar problems such as vascular system or road system reconstructions. Deep Learning models are at the origin of many algorithms that reached state of the art performance in tasks such as image segmentation of classification. However, these approaches have not yet been able to provide an efficient solution to the current problem, as they come with a range of disadvantages. First, Deep Learning approaches do not guarantee the tree-topology of neuronal arborizations, and maintaining that topology is important for biologists that are willing to use the reconstructions to perform tasks such as neuronal signal propagation simulations. Indeed, common techniques involve segmenting the voxels that should be classified as neuron or background, then skeletonizing that output, and lastly re-connecting the discontinuities that can arise from the

segmentation or the skeletonization processes. Such a method does not allow the reconstruction of the connectivity unless additional manual input is provided. Second, current models do not provide the output in the form of a graph. Indeed, representing neuron reconstructions according to their graph attributes (such as neurite types, trajectories and radii) is more suitable for many applications. Finally, a Deep Learning model, acting as a black box, works very differently from human annotators. It is not possible to affect the output of their network while the prediction is being made, making it hard to integrate a human operator in the reconstruction loop.

In this project, an innovative neuron reconstruction approach is designed, implemented, and tested. This new approach represents the neuronal arborizations as a graph and guarantees the tree topology on its reconstruction. As opposed to previous Deep Learning based approaches, which can not benefit from human input as they build the prediction (only after the segments are predicted can they be manually reconnected), our approach leaves room for external intervention to influence the output of the algorithm. If a branch is modified or interrupted, a new point can be provided and the agent will keep going from there.

The proposed method is based on a Reinforcement Learning agent that reconstructs the branches of a neuron step after step. Starting from a voxel, the agent follows its decision-making policy to iteratively select which connected voxel is the best candidate to be the next voxel of the branch. The implemented game consists in a cursor navigating through the voxels of an input image, in such a way that it builds a trajectory that matches one of the branches contained in the input image. We implement this navigation capacity using a Deep Q-Network, a model combining Deep Learning and Q-Learning, first introduced by Mnih et al. (2013). To that end, we specifically design a reward function and a training framework that fits the goal of tracing neuron branches. The designed reward system compares the reconstructed neurons to the annotations and encourages the agent to build a reconstruction that has a structure similar to the annotations, without enforcing it to be identical.

The implemented neuron tracing method has been progressively adapted to be applied to data of increased complexity. First, the method was shown to be working on simple curves from synthetic 2D data, then it was adapted to yield robust performances with all sorts of curves containing arborizations. The approach was subsequently extended to work on synthetic 3D data, starting with simple curves, to eventually have an agent that is capable of tracing branches and reconstructing trees coming from real neuron data annotations. Finally, the method is applied to input images predicted from real 3D microscopy scans. In this last application, the agent can still follow individual neurites, but the training framework is seriously limited due to anomalies in both the input neuron data and the provided annotations. Significant gaps in both the predictions and the annotations make it impossible to retrieve information about the location of the bifurcation points and of the end points. The affiliation of the annotated points to their respective branches is not retrievable, which also considerably compromises the design of the reward system.

The contribution of this work is three-fold. First, we demonstrate the feasibility of neuron tracing by incremental graph construction. Second, the proposed framework can readily be integrated in an interactive neuron tracing software, where the user selects the starting point and relies on the agent to trace individual neurites, correcting the reconstruction where needed. Last but not least, we highlight the need for a different annotation procedure that would guarantee topological correctness of the ground-truth neuron models.

The main limitation of this work is that, on real microscopy scans, we could not demonstrate the capacity of the agent to switch to a new branch having completed the currently traced neurite. This derives from the fact that the available annotation does not correctly represent the topology of the neurites: branches are often represented as independent and disconnected arborizations, and most neurites are split into multiple segments. Further experiments are therefore needed to assess the performance of the fully automatic tracing on real microscopy scans. Nonetheless, we demonstrated that the method can reliably reconstruct full neuronal arborizations from synthetic data, which suggests that this result could be replicated for real images, if accurate training data is available.

# 2 Related Work

Neuron reconstruction from microscopy scans is a research subject that has been addressed for years (Zhou et al. (2016), Zhou et al. (2018)). Multiple interactive tools and softwares have been developed in an attempt to reduce the required time to reconstruct neurons, and to facilitate human intervention in the reconstruction process (*e.g.* GTree by Zhou et al. (2021), the online tool Imaris, or the online tool Neurolucida). However, most of the existing methods are limited in their contributions, as they do not guarantee the conservation of the topology of the underlying tree, and also because they output probability maps that require later processing, whereas a graph-based reconstruction is preferred.

Extraction of inter-connected curves and the underlying network is a subject of research that is not only relevant to neuron microscopy scans. To name a few, road network and vessel network extractions are some of the typical tasks that are also concerned by this research question, justifying why being able to extract curves from images along with their connectivity is equivalent to finding the solution to a problem that has many applications. There have been multiple research papers on vessel extraction, such as the work by Tetteh et al. (2020), which compares various Deep Learning based extraction methods applied to the vessel extraction problem. Extraction of pulmonary vessels is another variation of this same research problem, which is relevant to several pulmonary vascular diseases (Román et al., 2018).

Most extraction frameworks tend to do a multi-step extraction, for instance by first segmenting the roads from the background, and then by relying on a complex set of heuristics to infer the graph from the segmentation output. In *RoadTracer* (Bastani et al., 2018), the authors attempt to bypass that multi-step extraction by automatically constructing the road network straight from the images. To do so, an iterative graph construction algorithm is used, which adds edges one at a time and identifies nodes when visited. This paper shows that such graph-based iterative approaches can achieve a higher performance than other purely Deep Learning based methods, when applied to the road network extraction problem. Following that idea, it makes sense to implement an iterative algorithm to the neuron tracing problem, as information about the nodes and the edges of the underlying graph is valuable.

In the past decade, there has been growing interest towards developing new Reinforcement Learning algorithms or applying existing RL techniques to improve the solutions of the state of the art algorithms in other domains. Reinforcement Learning has long been associated with video games being at the core of many research papers that attempted to reach superhuman performances with artificial intelligence (AI) models (Mnih et al., 2013). RL approaches are considered as more suited for video games than Deep Learning, as the set of actions and of states that the agent can be in is well defined, and it usually comes with a pre-defined scoring metric, which can be used as part of a reward system. Nevertheless, as opposed to Deep Learning and Machine Learning algorithms, the use of RL techniques for other tasks such as image segmentation or object detection tasks is still not widely accepted as a preferred option. In the emerging field of biomedical imagery in particular, there has been a few papers already attempting to apply a Reinforcement Learning framework to biomedical image processing problems. It has already been shown that a RL agent can be trained to solve tasks such as landmark identification on biomedical images in *An Artificial Agent for Anatomical Landmark Detection in Medical Images* (Ghesu et al., 2016). The authors of *An Artificial Agent for Robust Image Registration* (Liao et al., 2016) also use a RL framework to provide a solution to the image registration problem; which involves aligning two or more images, proving that RL can be used

to improve the current research in biomedical image processing in many aspects.

The goal of this project is to try to experiment if, in a similar manner, RL techniques can be used to provide a new way to extract connectivity from curves on biomedical images. *Deep Reinforcement Learning for Vessel Centerline Tracing in Multi-modality 3D Volumes* (Zhang et al., 2018) managed to show that it is possible to track image structures by iteratively navigating through image pixels. However, this last publication does not address issues such as detecting bifurcations and extracting connectivity between multiple curves from an image, for which a new reward system and a training framework specific and adapted to the neuron tracing task needs to be designed.

# 3 Methods

In contrast to other pixel-wise segmentation approaches, my implementation provides additional information than a binary classification of all pixels. By iteratively building a reconstructed path, the algorithm creates a graph, which has the advantage of extracting the connectivity of the bifurcation points and of separately segmenting the different branches, giving more than a binary output. This section explains the architecture of the algorithm and details the series of improvements that have led to the final implementation of the agent.

## 3.1 Framework

Reinforcement learning is a learning methodology that is based on an agent which learns how to behave in an environment from the rewards or penalties that it gets for the action that it performs. The basic architecture of a RL problem (shown on Figure 2) contains the following elements :

- An agent, which is in charge of deciding which *action* to perform from the set of available actions given a *state*. A state $S_t$ is represented by the information that the agent has about the state of the environment at iteration $t$. The action that the agent chooses to perform given state $S_t$ is denoted as $a_t$.
- An environment, which updates every time an action is performed and provides the agent with a new state $S_{t+1}$.
- A reward $R_t$, which is a numerical feedback from the environment supposed to reflect the quality of a specific action in a certain state.

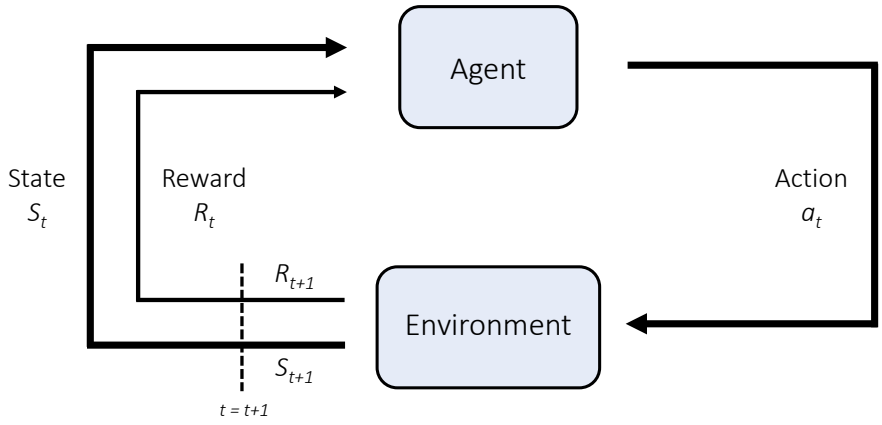



Figure 2: Diagram of a basic RL framework.

RL algorithms have most commonly been applied to games as a way to train a form of artificial intelligence that can ultimately perform at super-human performance when rightly trained. The agent usually consists in the player, which can be substituted with a neural network or various algorithms that are in charge of performing the actions such as requesting the move of a piece in a chess game, or requesting the action of shooting in an arcade game.

Following the example of applying this framework to chess, the environment is the structure that handles starting the game, setting up the positions of all respective pieces, and that evaluates whether the game is finished or not. The definition of a state highly varies from game to game, but it needs to gather the necessary information for the agent to evaluate the current situation of the game. In the game of chess, it can be represented as an 8x8x8 array, with each 8x8 plane keeping track of the positions of all pieces of one type (there are 8 different types of pieces). The environment has to be provided with a reward policy, which gives a numerical feedback on the value of an action given a state; which could be addressed by giving a positive reward proportional to the importance of a piece when an action leads to the capture of a piece in the case of chess. In order to apply this approach to the neuron segmentation problem, we design a game that consists in having a cursor that can navigate through an image, starting at a certain initial point, in such a way that it can build trajectories in space. At each iteration, the environment, which contains the image, is provided with a direction to move its cursor towards. It is expected that the RL learning framework can be applied to such a game design in order to teach the agent how to drive the cursor along the trajectory of neurites contained in the input image. The agent is trained by attempting to trace thousands of images through trial and error, and learns how to adapt its behavior by updating its decision-making policy based on the rewards that it got in its previous tries. The following subsections detail how each component of the framework has been adapted to match the proposed implementation.

### 3.1.1 Agent

At the start of each trial or episode, the agent is initialized in a manually selected starting point. It then assesses the state that it receives from the environment (detailed below) to pick an action. There are two types of actions here : moving actions and bifurcation actions. The set of actions that the agent can perform depends on whether the environment is in 2D or in 3D. For moving actions : if in 2D, the agent can perform 8 different actions (up, up-right, right, right-down, down, down-left, left, left-up), which means moving one pixel away from its previous position in the selected direction. In 3D, the number of possible actions increases to 26, including diagonal moves. Moving actions are fed to the environment in the form of an integer, ranging from 0 to the number of possible actions minus 1. The agent can also detect the end of a neurite and decide to stop, which is also considered a moving action, and is represented by n°9 or 28, respectively in 2D or 3D.

For bifurcation actions, the decision is binary. The agent might predict that the cursor is currently at the intersection of two neurites, and perform the 'bifurcate' action, or predict no bifurcation (n°1 for bifurcate, 0 for no bifurcation). When the bifurcation action is selected, the current location of the cursor is saved into a list of bifurcation points, and whenever the reconstruction of a branch comes to an end, the environment will reset the cursor's position on the last added bifurcation point and center the new state around it. After that, that bifurcation point is removed from the list of points to go back to. A branch reconstruction can either be stopped when the stopping action is triggered, or when the number of steps goes over the number of max steps. An episode is only considered as finished when the reconstruction is stopped and the list of bifurcation points to go back to is empty. As depicted on Fig. 2, the agent receives a new state updated accordingly to the moving action that it decided to take, along with a reward from the environment. The state that the agent receives from the environment is a 2-channel array stacking the 2 following channels :

1. A cropped window of the neuron image that needs to be segmented centered around the agent's pixel position (a square in 2D, a cube in 3D).

2. A crop of the agent's previous paths, named canvas. The agent's path is recorded by having an array of the same size as the cropped image window initialized with 0's, and by filling it with positive values in the position of the voxels that were previously visited by the cursor. The values on this canvas start from 1 for the most recently visited voxel (the cursor) and decrease from there (shown on Figure 3c). This canvas is also centered on the cursor's current position, and is aligned with the cropped image window.

In order to decide which action to perform given its state, an agent needs to follow what is called a decision-making policy. There are multiple possible techniques to design that policy, but Deep Q-Learning, since its introduction in 2013 (Mnih et al., 2013), has attracted a lot of interest thanks to the significant improvements in performance in many tasks, in particular playing Atari games. A Deep Q-Network (DQN) is therefore used in our implementation as a way to map each new state with its preferred action. A DQN is a Neural Network, which in the proposed implementation, takes as input an image of shape $w \times w \times 2$ in 2D, where $w$ is the length of the cube that is cropped around the cursor and fed to the agent, and of shape $w \times w \times w \times 2$ in 3D. That input then goes through a succession of convolutional and linear layers, and finally outputs an array of action probabilities. The action associated with the highest value is the action that should be picked according to the network. The DQN is processing the input state to predict a moving action only, meaning an integer in [0,26] including stopping. The bifurcation detection task is independent, and the location of bifurcations is detected by another convolutional neural network (CNN) which has to be trained before the agent's training. To make it possible for the agent to start learning (to not rely on the output of a randomly initialized DQN) and to keep exploring new behaviors as it progresses into its learning process, the agent does not always perform the action indicated by the DQN during training, but follows an $\epsilon$-greedy policy instead. In $\epsilon$-greedy action selection, the agent has a probability $\epsilon$ of picking a random action (exploration) instead of following the network output (exploitation). Starting with a significant $\epsilon$ and decreasing it over time makes for a good balance between exploration and exploitation, the agent still needs to look for unexplored situations to avoid getting stuck with a non-optimal policy.

To improve the agent's decision-making policy, the DQN is regularly optimized. The weights of the network are updated by iteratively taking steps in the direction of the gradient, which is computed on a batch of previous experiences. After every step taken by the agent, an experience is stored in the experience memory, which is then used to select a batch from. In RL terms, an experience $\tau_t$ (or transition) is a variable summing up the agent-environment situation at time $t$, and it takes the following form :

$$\tau_t = (S_t, a_t, S_{t+1}, R_t, done) \tag{1}$$

with $S_t$ the current state, $a_t$ the moving action performed at time $t$, $S_{t+1}$ the state following action $a_t$, $R_t$ the reward it gets from performing $a_t$, and *done* a boolean signal from the environment telling if the episode is finished or not.

In a conventional RL set-up, a DQN is trained to predict a probability for the whole set of possible actions. In our implementation, the DQN training framework is only applied to the learning of the moving actions (direction or stopping), not for the bifurcation actions, and is

adapted to fulfill the current goal of tracing neurons. Due to the learning of directions being necessary for the learning of stopping, the DQN is trained in a 2-stage fashion. During the first phase, stopping is disabled, and the DQN is optimized according to the values of the direction actions, and the agent does not include stopping as part of its exploration. In the second phase, stopping is enabled, and the DQN is optimized again to include stopping as part of the agent's behavior. The exploration of this second phase only involves randomly stopping with a probability $\epsilon$, while directions are not explored anymore. More about the DQN training is provided in section 3.1.4.

The bifurcation detection task is performed separately by another independent neural network that is in charge of predicting a continuous value between 0-1 for the input state it is provided with (further detailed in 3.1.5). That value represents how likely a window crop is to be centered around a bifurcation point. For instance, a value of 0.8 predicted for a state that the agent is in means that there is a high chance that the agent's cursor is centered on a bifurcation. Based on these values, the locations of the most probable bifurcation points can be retrieved (also detailed later), and added to the list of bifurcation points to return to.

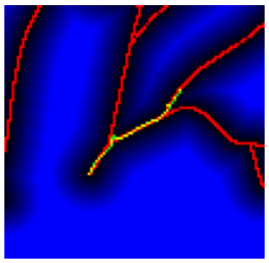
### 3.1.2 Environment

The environment represents the game with which the agent interacts. Applied to the neuron tracing problem, the environment is the unit that is in charge of loading the neuron data, getting it ready for the agent, updating it according to the agent's input and rendering a graphical output that displays the progress of the agent. The environment contains the images that need to be segmented along with the corresponding ground-truth information necessary for training (if available) : the ground-truth segmentation, the coordinates of neurite end points, the coordinates of bifurcation points, and of the desired starting points. The environment is implemented to make an episode start on one of its input starting points. Using branch end points can help with getting better results, compared to starting on a random point, which might be outside the neurite detection scope of the agent. The neuron game environment is able to update and provide the appropriate data to the agent through the implementation of the following methods :

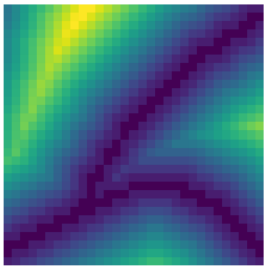
- A $reset()$ function that restarts the game, by selecting and loading a new image to segment along with its ground-truth information, and outputs a 2-channel cropped image (the starting state) centered around the starting point of the agent.
- A $step()$ function, which takes a moving action and a bifurcate action as inputs, and outputs the 2-channel cropped image representing the next state image that results from performing the input action.
- A $take\_action()$ method, in charge of the following : performing the moving action and updating the position of the agent; updating the bifurcation points list if bifurcating; or finishing a branch and going back to a previously detected bifurcation point if stopping.
- A $reward\_function()$, which evaluates the quality of the performed moving action in the given state in the form of a positive or negative numerical feedback. This method is implemented as a set of arbitrary conditions and a more detailed explanation of this function is provided below in section 3.1.3.

- A *render*() function, which provides a graphical output of the current progress of the agent as compared to the image it is trying to reconstruct. The graphical output is distinct from what is being fed to the agent, as the agent only sees a portion of the images and does not have access to the annotations.

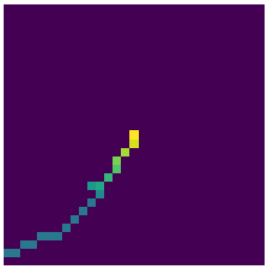
To give a better insight on the distinction between the graphical output rendered by the environment and what is fed to the agent, Figure 8 shows an example of the environment's graphical output of an ongoing episode in 2D (3a), and the two channels fed to the environment at the last step of the ongoing episode. 3b shows the crop of the input image centered around the cursor's position, and 3c shows the corresponding canvas. The 3 channels of (3a) are stacked into a RGB image for rendering purposes.



(a) Environment graphical output, ground-truth segmentation (red), input distance transform (blue), agent's predicted path (green), combination of red and green (yellow).



(b) $1^{st}$ channel of the state fed to the agent, containing a cropped window of size 31x31 pixels of the image to be segmented, centered around the agent.



(c) $2^{nd}$ channel of the state, keeping track of pixels previously covered by the agent, the more recently covered, the higher the value (from 1 (yellow) to 0.4 (green)).

Figure 3: 2D Comparison of the environment graphical rendering (a) and the 2-channel state that is fed to the agent (b and c). In this example, the episode started with the agent at the end of the neurite at the center of the image and meets two bifurcations.

### 3.1.3 Reward

The reward fed to the agent is computed by the environment and its value relies on a set of arbitrary conditions, also named reward function. The set of conditions used to fit this neuron tracing implementation is detailed in Algorithm 1. As explained above, the purpose of our DQN is to output a moving action, and it is not in charge of providing bifurcation detection. The bifurcation detection task is therefore not optimized using the RL framework and is not subject to the attribution of a reward. The reward is only computed based on the value of the moving action and of the state that the agent is in.

At every step, the reward is computed after performing the input moving action. If the action is not a stopping action, the position of the cursor is updated accordingly to the action, and the position of the newly covered pixel is compared with the ground-truth pixels. The distance to the closest ground-truth pixel (which has not been visited yet) is then returned and used as part of the reward system. If that distance stays within a tolerance range (named *slack*), the moving action is considered a useful action, and the corresponding ground-truth pixel is removed from the list of ground-truth points. That way, the agent has to keep covering

**Algorithm 1** Set of conditions for the implemented reward function

**if** *action* is 'stop' **then**
  $reward \leftarrow 0.0$
**else** ▷ moving action
  **if** $dist <= slack$ **then**
    positive *reward* inversely proportional to dist
  **else** ▷ useless steps are given a step penalty to learn the shortest path
    $reward \leftarrow -0.1 \times missed\_count$
  **end if**
**end if**

new ground-truth pixels if it wants to get additional reward. The reward given to the agent is proportional to how close it is from the actual newly covered pixel, to favor staying as close as possible to the center of the path. In case the new pixel does not match any of the unvisited ground-truth pixels, the agent is given a negative step penalty, which is proportional to how much time it has spent without performing a good move (*missed_count* represents the number of steps since the last useful move). The step penalty is introduced to satisfy two goals, the first one being to avoid useless steps, and to maximize the smoothness of the agent's path; and the second one is to give a reason for the agent to stop, by fear of accumulating negative rewards in case it stopped collecting single positive step rewards. The action of stopping in itself is given a neutral reward of 0 here, as there are instances of early stopping that can be perceived as negative, but also instances of late stopping, which are perceived as losing the additional step rewards collected as compared to early stopping. Learning of the optimal stopping policy therefore relies on a trade-off between maximizing the step rewards and avoiding the extra step penalty.

### 3.1.4 DQN training

The training of a DQN follows a pattern of trial and error, by optimizing the network step after step according to the previous experiences. The pseudocode in Algorithm 2 sums up a training loop of the agent and explains how the interaction between the environment and the agent is put together in the model. The training is based on simulating an episode many times by collecting the rewards at every step, saving the experiences to the memory along the way, and optimizing the DQN according to the previous experiences sampled from the memory.

The main goal behind Deep Q-learning optimization is to try to maximize what is called the expected return. The expected return $R_{t_0}$ represents the cumulative rewards $r_t$ that the agent will get after a time $t_0$, as described in equation (2). The discount $\gamma$ is a parameter between 0 and 1 that guarantees that this sum converges and that, when close to 1, increases the contribution of rewards that happened at times further in the future than $t_0$.

$$R_t = \sum_{t=t_0}^{\infty} (\gamma^{t-t_0}\ r_t) \tag{2}$$

According to Q-Learning, if we have a function $Q$ that returns the reward for any action-state pair $(S, a)$, then it would be easy to come up with the optimal decision-taking policy, by

**Algorithm 2** DQN training loop

$i \leftarrow 0$
**while** $i < num\_episodes$ **do**
  $S \leftarrow$ env.reset() ▷ start crop from new image
  **while** *done* is $False$ **do**
    update $\epsilon$
    $action \leftarrow$ agent.action$(S, \epsilon)$
    $S_{new}, reward, done \leftarrow$ env.step$(action)$
    $\tau \leftarrow (S, action, S_{new}, reward, done)$
    append $\tau$ to memory
    optimize DQN
    $S \leftarrow S_{new}$
  **end while**
  env.render() ▷ graphical output
  $i \leftarrow i + 1$
**end while**

picking the greedy action $a^*$ associated with the maximum return given a state :

$$a^* = arg\,\max_a Q(S, a) \tag{3}$$

The idea behind Deep Q-Learning is to train a neural network that learns this $Q$ function and that predicts a value for a given state-action pair. The update rule of a DQN is based on the Bellman equation (4), which states that the maximum future reward from state $S$ is the sum of the reward $r$ that the agent's received for entering state $S$ plus the maximum future reward for the next state $S'$.

$$Q(S, a) = r + \gamma \max_{a'} Q(S', a') \tag{4}$$

A DQN is then optimized by taking a step towards minimizing the error $\delta$ between the two sides of equ.(4), also named temporal difference (TD), shown on equ.(5). The optimization step is done by using the neural network to predict a value for the expected return at state $S$ and next state $S'$ and then taking a step towards the difference between these two values.

$$\delta = Q(S, a) - (r + \gamma \max_{a'} Q(S', a')) \tag{5}$$

Applied to the proposed neuron tracing game, an episode represents an attempt of reconstructing the branches of an image. At the start of every episode, a new neuron image is picked along with a starting point. A 2-channel cropped window is initialized by the environment (the starting state), by cropping a section of the starting image around the starting position of the cursor and stacking it with a canvas array only filled with a 1 in the center (the starting cursor position). Following an $\epsilon$-greedy policy, the agent either randomly picks a moving action to explore new areas of the neuron image, or forwards the state to its neural network, which in turn outputs the action that is associated with the maximum output probability. That action is given to the environment, which performs a step by moving the cursor along the input direction. The environment returns the updated state : a crop of the image centered around

the new position, and the updated canvas filled with the new cursor position and re-centered around it. The environment also computes the reward for the performed action and outputs a boolean that tells whether the episode is done or not. An episode is considered as done if the maximum number of steps has been reached and the list of bifurcation points to go back to is empty, or if the stopping action has been selected and the bifurcation point list is empty. All that information is later saved in the memory, and the DQN is optimized. The DQN optimization step is computed by sampling a random batch of 32 past transitions from the experience memory, and by taking a step towards minimizing the TD error.

### 3.1.5 Bifurcation CNN training

The bifurcation CNN (BifNet) in charge of detecting the location of bifurcations performs a regression on a 1-channel cropped cube of the image to predict a value in the range [0.0, 1.0] that relates how likely a cube is to be centered on a bifurcation point. To train the CNN, the exact location of bifurcation points of a few neurons is required to form the training set, which is composed of thousands of cropped cubes taken from the different images, associated with a label. To assign a label to each cube, a 3D Gaussian is first generated around each bifurcation point, with a value of 1 on the bifurcation points themselves and decreasing with the distance, as depicted on Figure 4. The label of a cropped cube is determined by the value assigned by the Gaussian distribution to the pixel at the center of the cube. For instance, a cube that contains a bifurcation, but that has its center slightly shifted from the bifurcation point will be assigned a label slightly lower than 1.

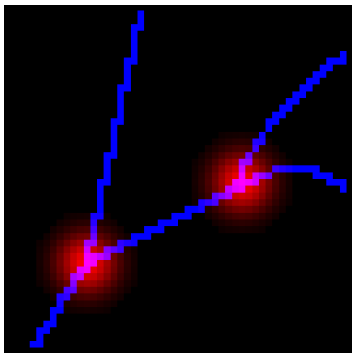


Figure 4: 2D example of the values assigned to all pixels of an image when fitting a Gaussian around every bifurcation point that is proportional to the distance to these points. Red describes the label amplitude at pixel locations. The closer to a bifurcation, the higher the value, and the lighter the red (1 at a bifurcation).

To select all the cubes required for training, a certain amount of center points are selected for each neuron, making sure that a large fraction of them are close to a bifurcation, and that most of them are also close to a branch, as this is the kind of input that the agent will be exposed to. From Figure 4, it is obvious that the points with a value higher than 0 are sparse, even more so when looking at full neurons in 3D. To account for that it is important to balance the cubes so as to have a decent fraction of them with values higher than 0. In addition to that, the cubes are all given random rotations and symmetries to augment the data and make it invariant to the orientation of the branches. Once the network is trained, it can be incorporated into the agent training by detecting the points on the agent's path where the predicted value goes over

a certain threshold. The exact way the network avoids classifying many closely located points as bifurcations and how exactly the location of those points is retrieved is explained in depth in sections 4.2.3 and 5.

## 3.2 Data

To achieve the reconstruction of neuronal shapes in 3 dimensions, the game was first developed in 2 dimensions for obvious reasons. Indeed, the problem being more complex in 3D and harder to visualize, its prior implementation in 2D is important to build a substantial understanding of how to adapt the baseline framework to the neuron tracing problem. Also, it is not possible to project 3D neurons in 2D and use them as images for training, as instances of closed loops could appear in the images. Therefore, the agent was first trained in 2D on a set of hand-drawn curves displaying various shapes, as well as displaying bifurcations to also train for the detection task. Figure 5 below shows an example of some of the graphs that are used to train the model in 2D.

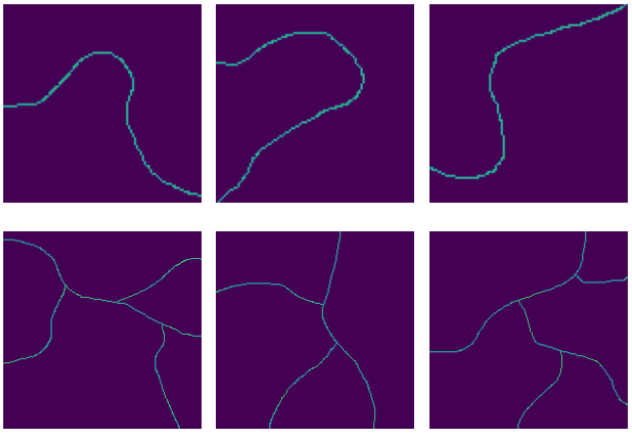


Figure 5: Examples of 2D graphs used in the case of 2D training.

In the case of 3D, the agent has first been trained by turning all these graphs into 3D, giving them a random rotation and a random symmetry. In addition to that, 30 neuron morphologies were retrieved from neuromorpho.org, in particular: morphologies 'Layer-2-3-Ethanol-1.CNG' to 'Layer-2-3-Ethanol-27.CNG' were used for training. Figure 6 shows the 2D projections of one of those morphologies. These morphologies come under the *.swc* format and the position of the ground-truth pixels, as well as the position of the bifurcations and end points can be recovered from it. The NeuroMorpho library can be used to download these *swc* morphologies with python. MorphIO is also a library than can be useful as it comes with a way to store the information contained in the *swc* files in a Class and access some its attributes in a convenient way.

Finally, some large scale 3D neurons are also provided, along with the U-Net prediction for their segmentation. Some ground-truth information is also provided with these neurons, such as the binary labelling of many pixels. The annotations are not continuous, making it impossible to extract the branch-specific segmentation as well as the bifurcation points. These neurons are too large to fit in the memory and have to be loaded in real time in cubes of smaller shape and segmented one after another (detailed in section 6).

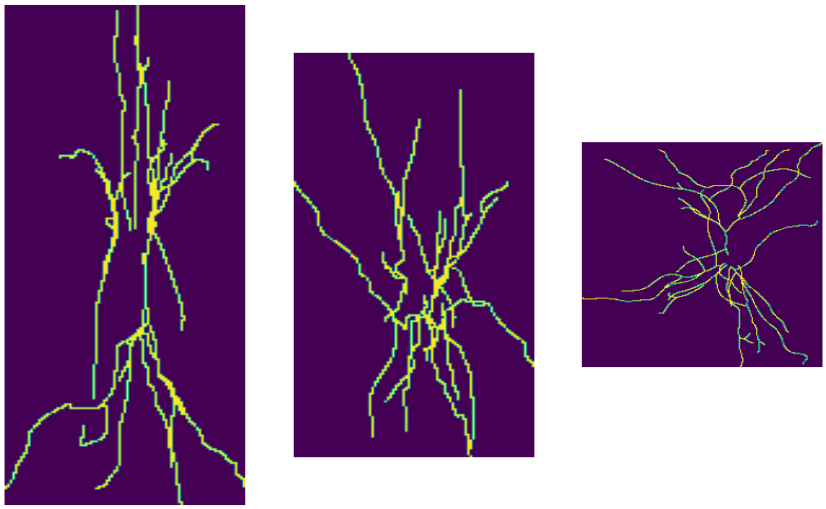
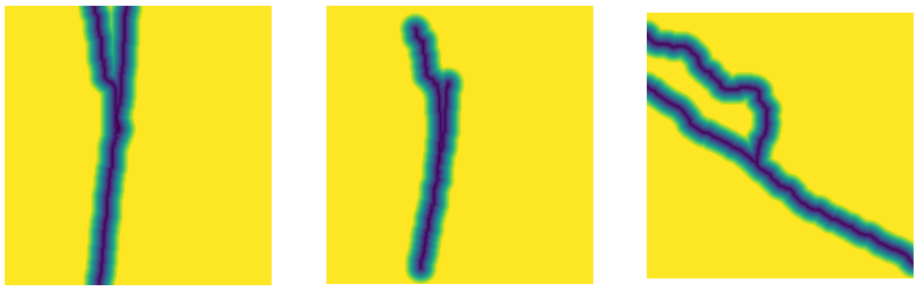
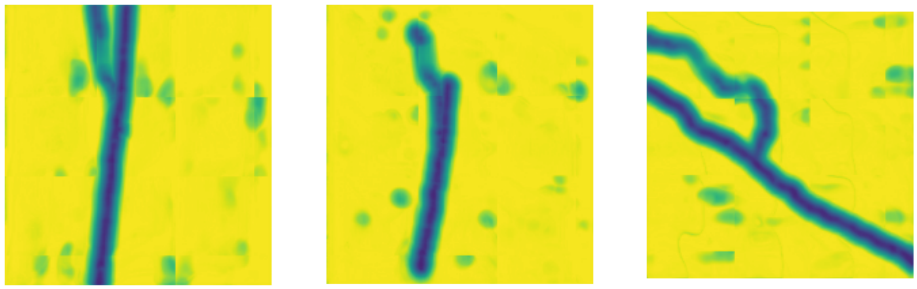


Figure 6: Three 2D projections of a 3D neuron morphology taken from neuromorpho.org.

With the prospect of being suitable for application on real data, the agent is first being trained on perfect distance maps to prove the reliability of the concept. Figure 7 below shows the comparison of the perfect input and of the predicted input for a cube taken from a large scale neuron. The predicted distance map is obtained by running cropped sections of a microscopy scan through a U-Net to end up with an output similar to 7b, which can in turn be used as an input to the agent. The synthetic input is obtained by computing the Euclidian distance transform (EDT) and is clipped with a value of 15.



(a) Synthetic distance map.



(b) Predicted distance map.

Figure 7: Comparison of the synthetic (7a) and predicted (7b) distance maps for a section of a 3D neuron. The 2D projections along all 3 axes are displayed in both cases.

# 4 Experimental Procedure Leading to Final Model

This section details the multiple contributions that had to be added to the basic game in order to reach a working solution in 3D, as well as what motivated such additions. The game development first started in 2D by achieving reconstruction of simple curves using the ground-truth as input, then showing that the approach could also be applied to reconstruct the same curves using synthetic distance maps as input. The method was then adapted to work in 3D and the bifurcation detection task was integrated into the framework. Finally, the approach was tested on distance maps predicted from real neuron microscopy scans.

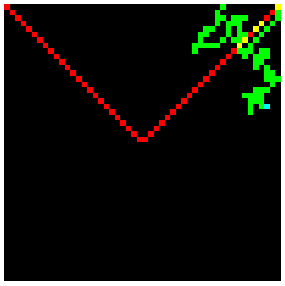
## 4.1 In 2 Dimensions

### 4.1.1 Implementation of the Game and Following a Single Line

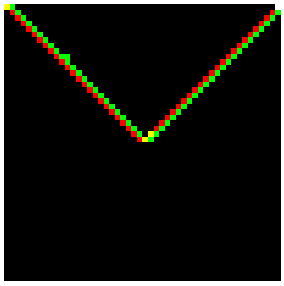
The implementation started with the design of a basic game working on a small 50x50 image. The agent has first been trained to reconstruct using the ground-truth and not the distance map to test a reinforcement learning framework that can successfully lead to following a path along an image. The environment is providing the DQN with a 3-channel input containing :

- The full ground-truth image to reconstruct.
- A canvas containing the pixels previously covered by the agent. This canvas is an array of the same size as the full image filled with 1's on pixels that were previously covered by the cursor, and with 0's elsewhere.
- A canvas of the same size as the original image filled with 0's except at the location of the current cursor position, where it is filled with a 1.

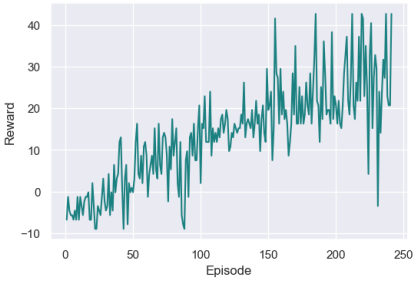
The metric used to qualify the quality of a move is based on the completeness; which is the percentage of ground-truth pixels that are covered by the canvas pixels. A basic reward system of $+1$ for a step causing an increase in completeness and $-0.1$ otherwise allows the agent to reconstruct the most basic path as shown on Figures 8a and 8b (with a slack of 1).



(a) Early episode of the agent following the path, almost complete exploration.



(b) Final episode of training. Agent managing to track the path.




(c) Reward as a function of the number of episodes. Maximum possible reward of 50.

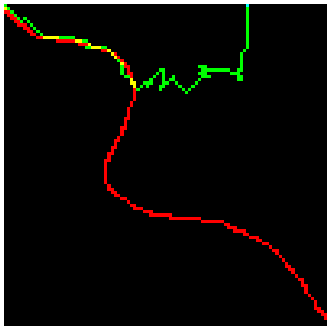
Figure 8: Results of the first basic game development.

At this point, the agent's input is not centered around the cursor's position. The current position is only encoded through the pixel filled in the $3^{rd}$ channel. This first approach shows

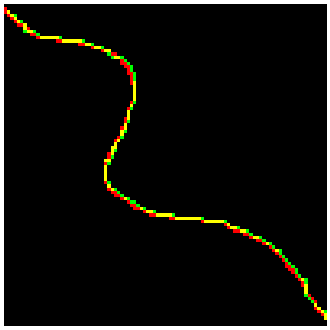
that the agent can learn how to follow a straight line as well as a bend. It takes about 250 episodes to do so by looking at the number of episodes it takes for the reward curve to reach a plateau, and takes 6 minutes of training with 50 steps per episode to complete.

### 4.1.2 Tracing Simple Curves

It takes many episodes to train a similar agent on longer curves with various shapes as the ones shown on Figure 5. 20 such curves are used, 10 for training and 10 for testing. Sticking to more complex bends in all directions can not be achieved as depicted on Figure 9a. Playing with the parameters and with the neural network setup is not sufficient to positively alter the performance of the agent. A massive leap forward in performance can be achieved when providing a slight change in how the image is fed to the agent. The training performance on all graphs is close to perfect on all graphs when switching to taking a crop centered around the current position of the agent. Figure 9b shows the results of the agent when tested on the same graphs as before with the new input. The $3^{rd}$ channel of the input only containing the current position of the agent can therefore be discarded, as this information is already implicitly covered by centering the crop around it.



(a) Testing the agent on a simple curve with the primary setup.



(b) Testing the agent on the same curve after centering the crop around the agent's position.

Figure 9: Comparison of performance on a curve before (9a) and after (9b) centering the agent input around the position of the cursor after 300 episodes of training.

With the ultimate prospect of being applicable to real neuron data, which comes in the form of distance map predictions, it is necessary to show that this first approach is also suited for the use of distance maps as input. Up to this point, the ground-truth annotation is still used as input to the agent. As the agent mastered path following using the ground-truth segmentation as input, it can next be tried using distance maps. Synthetic distance maps can be generated by taking the EDT of the ground-truth graphs. By using the same approach to train the agent, path following can also be achieved, as suggested by Figure 10. A slight drop in stability and in convergence time can be observed when going from using the ground-truth to using the distance map as input. When trying the model on the 10 testing graphs, only 60% of the graphs were successfully reconstructed, as compared to 100% of the graphs when using the ground-truth input. A graph is considered as successfully reconstructed when the completeness of the reconstruction reaches 95% (meaning that 95% of the ground-truth pixels are covered by the agent's path). Adding a data augmentation step, which consists in duplicating each training graph 10 times with a random rotation and a symmetry along a random axis, it is possible to reduce the

overfit to the training data and the agent becomes more robust to variations in graphs, going from correctly reconstructing 60% of the testing graphs to 95%.

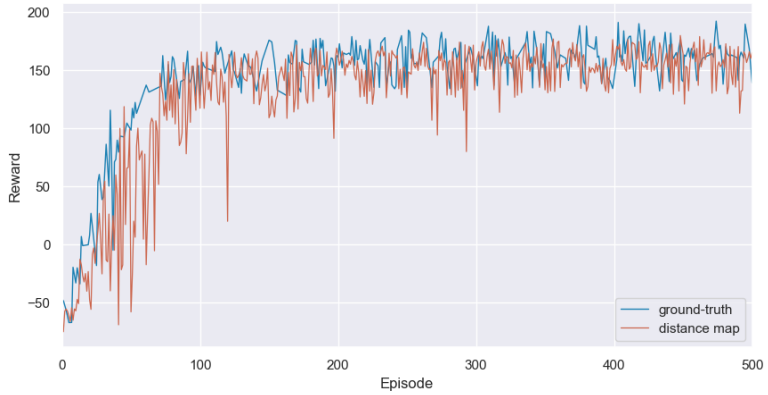



Figure 10: Comparison of the reward curves during training when using the ground-truth (blue) or the perfect distance map (orange) as part of the input to the agent.

### 4.1.3 DQN Extensions Implementation

Since the early developments of Deep Q-learning agents, many improvements have been proposed to Deep Q-Networks. In the basic DQN implemented in 4.1.1, two slight modifications are already implemented in the DQN : the double DQN as well as the dueling DQN. The first one works by decoupling the action evaluation from the action selection during the optimization step, by using two different networks and updating one less frequently than the other. This addition is a way to improve the stability of the learning, to avoid evaluating with the same network that is being updated. The latter helps with the generalization across actions, by adding a separate channel in the linear layer stage of the neural network and combining it with the output. As detailed in *Rainbow: Combining Improvements in Deep Reinforcement Learning* (Hessel et al., 2017), two other extensions stand out as real improvements to the classical DQN framework in multiple tested game set-ups. Prioritized Experience Replay (PER) and N-step learning seem to be reasonable additions when trying to improve learning convergence speed and stability. This paper reviews the changes in performance of 7 variations of a standard DQN and compares the performance when adding and removing each of the 7 extensions in various set-ups.

PER works by sampling more often transitions from the replay memory from which there is more to learn (Schaul et al., 2016). N-step or multi-step learning consists in using more than a single reward $r$ in the update rule from equ.(5), by replacing it with the discounted sum of the next $n$ rewards. When combined with the right choice of $n$, multi-step learning is associated with faster learning. The comparison of the reward curves with and without both of these extensions is plotted below (Fig. 11). From that plot, we observe that the baseline model is the most stable one displaying the smoothest curve, especially when compared to N-step learning. There is also an increase of 20% in computational time when using PER, which is not a significant problem as of now in 2D, but which quickly starts being an issue when moving to 3D as explained below. Due to the lack of positive contribution towards the model and the

increase in the complexity of the model in consequence to their implementation, the use of these two extensions is discarded in the next implementations.

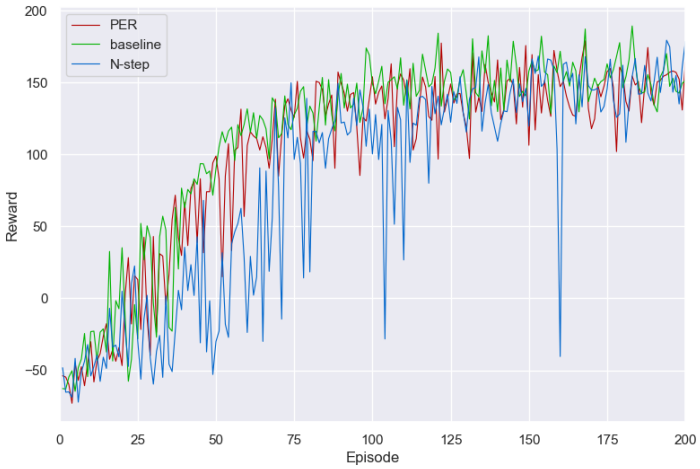



Figure 11: Comparison of the reward curves during training with and without the two Rainbow extensions, with baseline in green, multi-step in blue (with n set to 3), and PER in red.

### 4.1.4 Stopping Action Training

In all previous training, stopping was not enabled and the episodes would either stop when reaching high completeness, or when going over the maximum number of steps. Mastering the directions prior to the stopping is necessary for the agent to reach the end of branches and to be exposed with situations in which it should stop. Hence, training for stopping is performed in a 2-stage fashion. First, by training a model for the directions, and then, by performing fine-tuning of the pre-trained model to tune the part of its output which is associated with stopping. Having never been exposed to stopping, the agent does not start stopping by itself once allowed to do so. The exploration of the direction actions is not useful anymore, hence the $\epsilon$-greedy policy now only needs to be applied to the stopping action selection (agent always exploiting the output of the DQN, except that it can pick the stopping with a probability of $\epsilon$ at every step). Starting off with a very low $\epsilon$ of 0.004 (1 stopping every 250 steps in average) to fill the memory with a few instances of stopping, the agent quickly learns to exploit the stopping action right at the end of a branch as a way to avoid the step penalty of the extra steps and in order to maximize the reward that it gets by sticking to the path until the end of the branch (Figure 12).

The stopping training was made by training the agent on 5 different graphs, augmented to 50 graphs after the data augmentation step. To assess the quality of the model, the agent is tested on the 15 remaining graphs, also augmented 10-fold. The trained model is able to stop within a distance of 3 pixels from the stopping point on 98% of the graphs. This shows that the agent effectively learned that tracing the few pixels leading to the end of a branch still constitute a valuable reward, and that it successfully learned when to stop, with no cases of early stopping observed during testing.

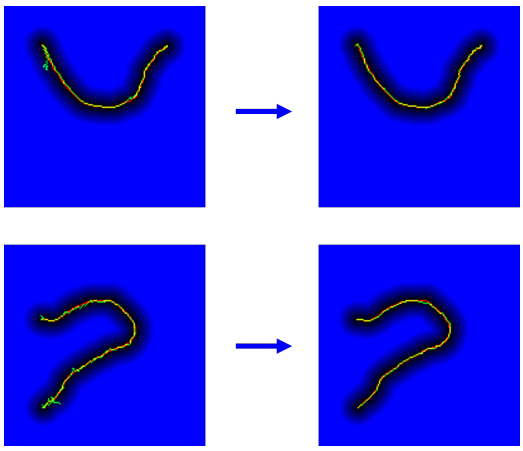


Figure 12: Comparison of stopping on two graphs at the start of stopping training (left) versus at the end of training (right).

#### 4.1.5 Bifurcation Action Training

Bifurcating was first being tried as an additional action that can be outputted from the same DQN. When the agent performs the bifurcation action, it stays in the same position and a reward is given according to the accuracy of the bifurcation detection. A true positive bifurcation detection is rewarded with $+5$, a false positive with $-5$ and an already detected bifurcation with $-20$. Training for the bifurcation action requires to already be able to reach bifurcations, so the bifurcation training is done by fine-tuning the previously trained model that was shown to successfully move and stop. After re-training the model, two main problems stand out :

1. The agent gets stuck into picking bifurcations indefinitely, despite the negative reward. This is due to the fact that it is always being provided with the same image crop with the same canvas as input, hence logically outputting the same action.

2. Unlearning the ability to move and stop in a smooth and efficient way, most likely as a result of trying to adapt the weights of the DQN towards improving the detection of the bifurcations.

Proposed solutions:

1. Adding a $3^{rd}$ channel to the stacked input, a bifurcation canvas, which keeps track of the previously detected bifurcation points by filling the canvas with disks of a small radius centered around the detected bifurcation point (see Figure 13). Unfortunately, the additional channel is not sufficient to completely dissuade the agent from over-detecting the same bifurcation point multiple times in a row and get stuck in this situation. The DQN is not able to link the negative reward, that it gets when detecting the same bifurcation point twice, with the spatial information contained in the newly added channel, which clearly indicates that the cursor is in a region already detected as a bifurcation.

2. Freezing parts of the neural network layers as an attempt to reduce the extent of the damage caused to the direction and stopping learning. By first going back and experimenting this idea to the stopping training (as mastered to a higher extent), the agent is

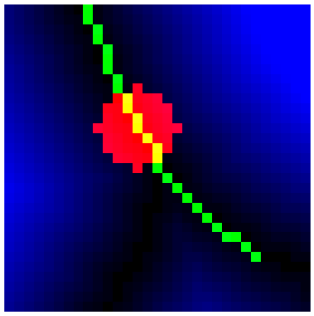


Figure 13: Example crop containing a bifurcation with the addition of the $3^{rd}$ channel (bifurcation canvas, in red). The 3 channels are stacked into a RGB image for rendering purposes. Green: canvas, blue: distance map.

able to learn to use the stopping action correctly by using the network trained for the directions and training it again keeping the convolutional layers frozen. When applied to bifurcation training and loading the model trained for directions and stopping in a similar fashion, the rate of correct bifurcation detection almost drops to 0. Therefore, the level of abstraction contained in the fully-connected layers seems to be too advanced to extract information relevant to the bifurcations from it. Sequential unfreezing of all higher convolutional layers one by one showed that the best results are obtained when keeping all the layers as unfrozen during optimization.

However, the performance of the bifurcation detection task after the last modifications is still not deemed sufficient and reliable at this point to continue the training and test enabling going back to all the previously detected bifurcations. There are too many instances of points being repeatedly picked as bifurcations and of irrelevant points being labelled as bifurcations to go further with the training. The bifurcation detection training is also too damaging as it is for an agent that is already efficient when it comes to moving and stopping along a path. For these reasons, the alternative approach of scanning for bifurcations independently from the agent is considered in further developments. The 3D implementation having been started in parallel with the 2D one, the alternative bifurcation detection approach is directly tested in 3D and will be detailed in section 4.2.3.

## 4.2 In 3 Dimensions

Now that the agent is able to move and stop in a reliable fashion, the environment can be adapted to be used with 3D images. The DQN has to be adapted accordingly, as it is now receiving input cubes in 3D and performing 3D convolutions is therefore required.

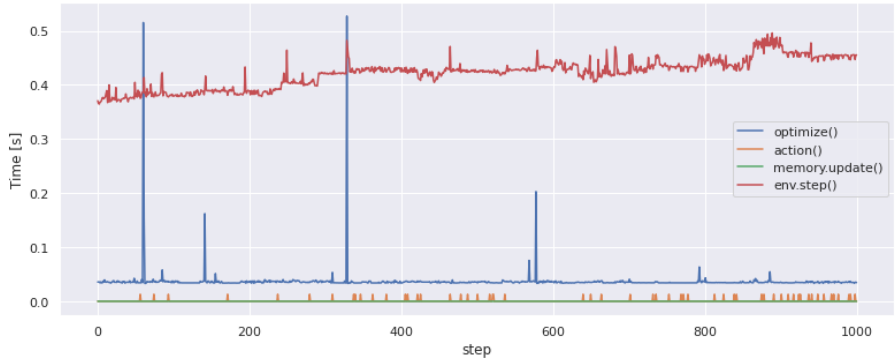
### 4.2.1 Sparsity and Computational Issues

When applying the same sets of parameters as the best working solution in 2D to the 3D environment, a massive drop in performance is observed. Applying the same method is not sufficient to teach the agent to follow simple curves in 3D. When going from 2D to 3D, the sparsity of positive ground-truth pixels increases, and so does the number of possible output actions, rising from 9 to 27. This causes the problem to be more imbalanced, as the agent is

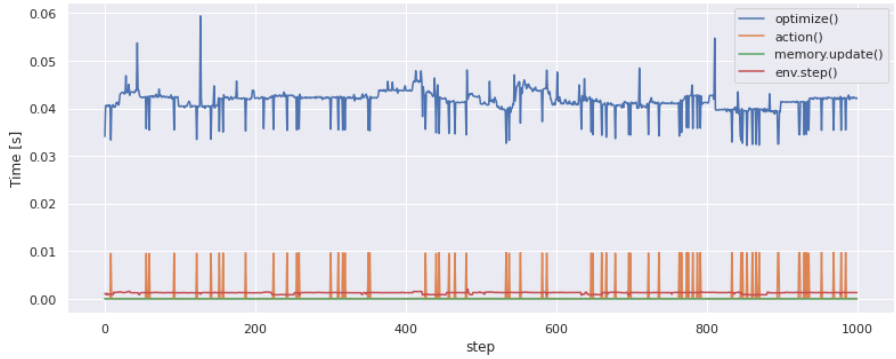
far less likely to take an action that results in catching a positive reward. Positive rewards are necessary for the agent to learn any positive behavior, so failing to produce positive behavior during exploration can not result in reproducing positive behavior during exploitation.

In addition to that, there is a strong increase in computational resources required for training, both on the physical memory side, due to having to keep 3D cubes in the RAM as part of the buffer memory; and on the GPU side, now having to compute 3D convolutions, which takes up a lot more resources than previously. The time taken to compute the same training loop goes from 2 seconds per episode in 2D to almost 3 minutes in 3D, causing a moderate training to take close to 15 hours to complete, which is not a sustainable training framework. Further analysis of the computation times allowed to identify the bottlenecks. Figure 14 details the computational time taken by the different elements of a training loop.

According to Fig. 14a, optimizing and taking a step (environment's step method) are the two operations that take the longest. The *optimize()* step computation time can not be further optimized by keeping the DQN as it is, as most of the time is spent processing the batch of experience and moving it to the GPU, which can not be avoided. The *step()* duration, on the other hand, when analyzed in depth, is mostly due to the completeness calculation and to the way the cubes are cropped from the image. Coming up with a new way to compute the completeness allowed to reduce the duration of one *step()* function call from 0.45s to less than 0.002s (shown on Figure 14b), dividing the overall training loop time 10-fold. Also, reducing the window size from 51 to 31 voxels, as well as lowering the memory capacity (dropping random




(a) Previous methods.




(b) New methods.

Figure 14: Comparison of the function calls computation times before (14a) and after (14b) their optimization.

samples when it reaches full capacity) and reducing the complexity of the network allowed to bring the computation time down to have a convenient training framework to work with.

### 4.2.2 Following and Stopping Along Simple Curves and Real Neuron Branches

As mentioned before, the primary problem when switching to 3D is the lack of exposition to positive rewards. Multiple methods are tested as a way to tackle this issue. The ground-truth is first dilated prior to computing the EDT (the ground-truth itself is not dilated), so as to make the tubes thicker, and the slack is increased (the distance threshold for a cursor to still be considered as matching a ground-truth voxel). By doing that, the agent is more likely to start collecting positive rewards during early exploration. By implementing this method, the agent is now able to follow simple curves such as the one shown on Figure 15. However, the model as it is is still slow at training and does not display high accuracy when testing on new graphs. When training on 13 graphs augmented to 91 graphs and testing on 7 graphs also augmented with a factor of 7, the agent is only able to reach a completeness higher than 30% in 14% of the graphs, and a completeness higher than 60% on a single graph. On top of that, dilation is not an option that can be applied to real neuron predicted distance maps, so it should not be a preferred solution.

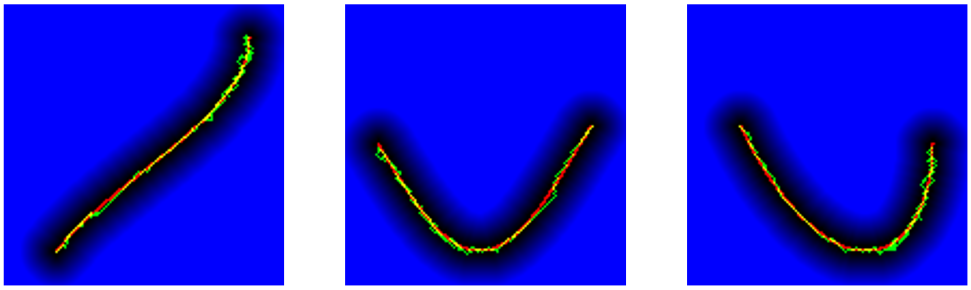


Figure 15: 2D projections on all 3 axes of an episode in 3D with increased slack and dilated EDT tubes.

From the numerous previous training attempts, the main take-away is that early training (the first 20 to 30 episodes) is absolutely crucial in determining the outcome of the learned behavior. Varying the learning rate or other learning parameters is usually not sufficient to unlearn the behavior that the agent picks within the first few episodes. Being exposed to a very sparse reward environment in 3D, the agent spends the vast majority of its steps collecting negative rewards in early training. This creates a bias towards optimizing the network according to those negative reward, to which the agent is always being exposed, causing the agent to learn to behave in an unpredictable way. As a consequence, it can be observed that the agent quickly learns to follow a single direction, suggesting that the weights of the models have updated in such a way that any input always returns the same maximum value for its output actions. In such training experiments, even if the agent meets positive reward at later times, the agent remains unable to unlearn the bad behavior that it picked up by spending all its early experiences away from the rewards. This suggests that it is extremely inefficient and counter-productive to let the agent perform steps once it is away from the centerline of a branch.

Two components are therefore essential to solve the low reward exposure in early training and reach high performance in tracing paths in 3D :

- First, the introduction of an "abort episode" condition to the environment, by counting the number of steps spent since last collecting a positive reward. When triggered, it causes the episode to stop (when the agent spends more than 10 steps off-path). This way, the unpredictable behavior acquired by spending extended time collecting negative rewards is avoided.

- Second, the trade-off between exploration and exploitation defined by the $\epsilon$ parameter plays a strong role in the outcome of the learning. If scaled too high, the agent spends too long acting randomly, and the successful events at the start of training get diluted amongst all the other saved experiences. This causes the agent to miss the optimal learning window and to end up picking up wrong behavior. With a too low $\epsilon$, it becomes really unlikely for the agent to perform a sequence of action leading to a series of positive rewards, and learning of positive behavior won't be achievable.

By introducing the abort condition and finely tuning the value of $\epsilon$ and the rate at which it decreases, performance similar to previous performance in 2D can be achieved for moving actions. As a result, when running the same training as before on the 91 training graphs and testing it on the 49 testing graphs, the performance drastically improves. The 95% completeness mark is reached on 90% of the tested graphs.

Similarly, when performing the 2-stage training for the stopping training by fine-tuning the model trained for the directions, convincing performance can be achieved as in the 2D case. The abort condition needs to be disabled so that the agent experiences occurrences of late stopping and learns to stop right at the end of branches. Stopping within a 5 voxel distance to the stopping point is achieved in 100% of the tested graphs for which the 95% completeness score has been obtained.

Even when applying the newly trained model to data coming from real neuron annotations retrieved from neuromorpho.org, the agent is able to move along branches and bifurcations, and can successfully stop at the end, as shown with an example on Figure 16.

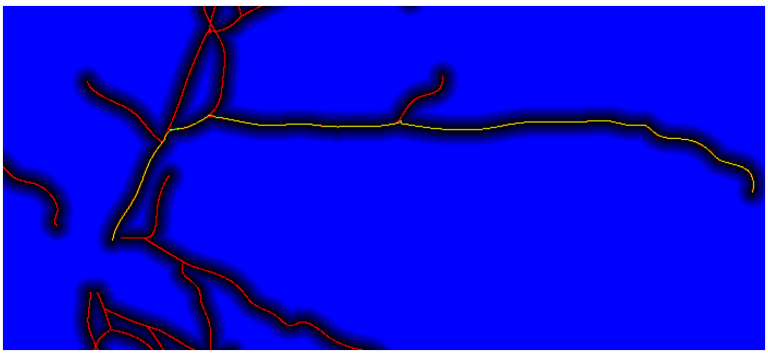


Figure 16: Example of the agent being able to move and stop along a real neuron 3D branch with bifurcations using synthetic distance map as input.

### 4.2.3 Bifurcation detection

The last step that needs to be incorporated to the agent in order to have a working solution is to add the bifurcation detection component to the model. To do so and as explained in section

3.1.5, a bifurcation network (BifNet) is trained by selecting many cubes from different neurons, some centered on bifurcations, others away from them, and assigning them a labels depending on their distance to the closest bifurcation point. Once trained, BifNet can be used to assign a value at each step of the agent by cropping a cube centered around the current cursor position and predicting its output value. The choice for a regression over a voxel-wise classification, breaking up which voxels represent bifurcations and which voxels do not, is justified by the fact that a series of very similar inputs will be assigned a prediction, which will hopefully keep rising as the inputs get closer to a bifurcation. Keeping the continuous nature of that prediction is helpful to identify where these predicted values increase and reach a peak, which is really likely to represent a bifurcation. Figure 17 shows the predicted values obtained when testing every voxel of the full branch of a neuron. Accurate detection of the peaks in regression values over a branch allows to retrieve the locations of bifurcation points. The next section (section 5) details

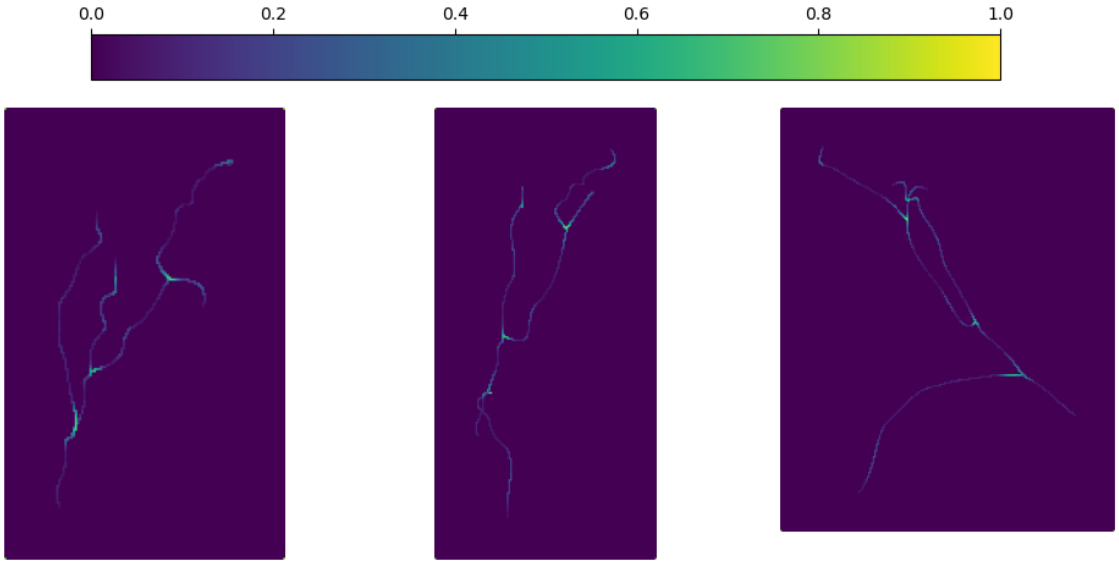



Figure 17: Combined output on a full branch when running cubes centered around each voxel of the ground-truth through BifNet and predicting their regression values. The values range from 0 (darker) to 1 (lighter), with a value of 1 representing a high chance of the input cube being centered on a bifurcation point.

how the bifurcation detection task is performed after each branch reconstruction to detect the main peaks in regression values. The location of these main peaks is then retrieved and updated in the environment so that every time a stopping action is performed, the environment will update the episode to make it start again from the last added bifurcation point. The agent can then continue its reconstruction from that new starting point, building a new branch but keeping track of what bifurcation point it is affiliated to. This way, it is possible to extract a graph structure from the output of the algorithm, with the bifurcation points representing the nodes, and the branches representing the edges.

To pick the best working model at the bifurcation detection task, different models were trained on 14000 training cubes and then tested on 6000 cubes. Despite predicting continuous values in order to simplify the detection of the peaks, this task remains a classification problem. To assess of the quality of the models, a testing class threshold of 0.4 was used to classify the test samples and compute the weighted F1-score of the models. Table 1 compares the weighted

F1-scores of the baseline implemented model with the scores that it achieves when adding more complexity. It shows that the baseline CNN, containing 3 convolutional layers and 1 hidden layer gives the best results during testing. The extra added complexity of the other models reduces the performance of the model, hence the baseline CNN is kept as the main model.

| Model | Weighted F1-score (%) |
|---|---|
| Baseline CNN | 92.34 |
| Baseline + extra hidden layer | 91.85 |
| Baseline + extra hidden layer + dropout | 91.47 |
| Baseline + extra convolutional layer | 91.61 |

Table 1: Weighted F1-scores of the different BifNet variations.

# 5 Proof of Concept

To come up with a model that can successfully achieve the reconstruction of 3D neurons, a 2-stage training is required. The training data used here are the 3D neurons collected from neuromorpho.org, which are then turned into distance maps. During the first training, the agent learns to move along the neurites and to handle bifurcations correctly, meaning not getting stuck at bifurcation points. Indeed, the main problem with bifurcations is that the DQN can sometimes predict a cyclic output, *i.e.* it can output an action that implies taking a step in a direction and subsequently output the action corresponding to the opposite direction, thus getting stuck oscillating between the same two voxels. 150-200 episodes are enough for the agent to learn how to properly track branches. Stop needs to be disabled, and the abort condition activated, so as to favor efficient learning in the early training.

In the second training, stopping is enabled, as well as the going back condition in the environment, which states that if the stopping action is selected and the list of detected bifurcations is not empty, the agent's position is updated to the last added bifurcation point. That point is then removed from that list of points to go back to. Even if stopping was not enabled during the first training, the model weights involved in stopping action selection have already started updating as a consequence of the overall network optimization. It is therefore important to reset the output layer element associated with the stopping action to avoid it starting with a high value as soon as it is enabled, which would result in the agent always picking stopping on the first step. In addition to that, double perfect bifurcation action input is provided. It means that every time the agent goes within a close range to a bifurcation, the action "bifurcate" is automatically selected. This way, the agent will be going back to all bifurcations that it visited, and can train to get moving again in the right direction when back at a bifurcation point. The agent is able to learn to not go down the same branch that it already covered by associating the path that is not covered yet in the canvas with potential reward. In addition to providing the perfect bifurcation input a first time for that training, every visited bifurcation is added a second time to the list of points to go back to. That extra input will cause the agent to go back to the same bifurcation twice, learning to go down the second branch on the first time, and also learning to stop when coming back to a completed bifurcation on the second time. This is essential when considering a non-perfect bifurcation input coming from BifNet, as the agent is expected to learn to stop when going back to a wrongly detected bifurcation.

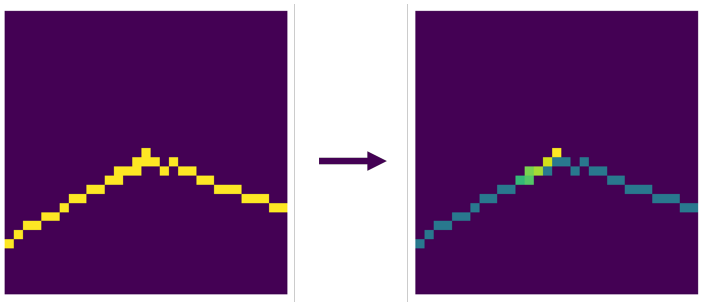
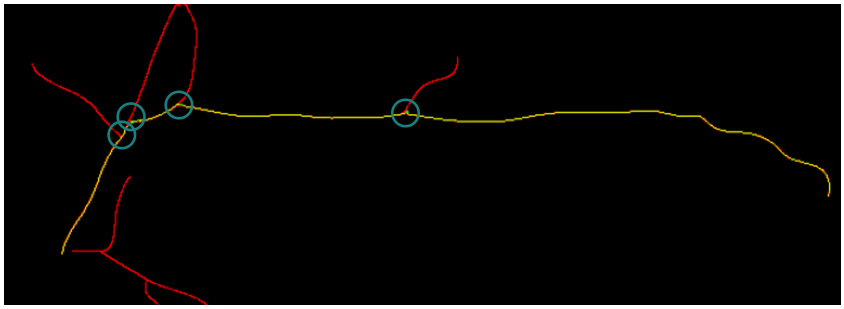


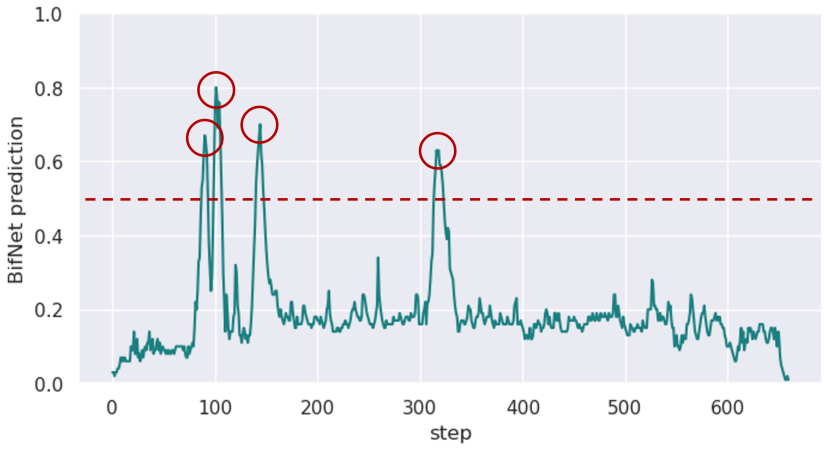
Figure 18: New version of the voxel canvas, also encoding the order at which voxels were covered. Central voxel (most recent) has a value of 1 and it incrementally decreases to a minimal value of 0.4. In this example, the agent came back to a bifurcation 6 steps before this observation, as reflected by the lightest voxels starting from the middle of the crop.

In practice, the agent struggles to get moving again after going back to a previous bifurcation. A change in the canvas channel of the input is performed to make it reflect the order of the moves by filling the canvas with higher values for more recent moves (up to 1), as shown on Figure 18. The going back event occurring in the example of Fig. 18 is only deducible on the new version of the canvas, as the superimposition of voxels in the binary canvas makes it impossible to know what voxels were covered during the reconstruction of the last branch. It is fair to assume that the agent is also less capable of learning from the left version of the canvas if missing that piece of information.

To incorporate the input from BifNet for the bifurcation detection task, a branch is first fully reconstructed, and the values predicted by BifNet on each cursor position is collected. Once the branch is finished, the predicted values are thresholded and the main peaks are detected. By retrieving the index at which those peak occurs, it is possible to access the location of those predicted bifurcation points. Figure 19 depicts the reconstruction of a branch (19a) and its corresponding BifNet predicted values (19b). The location of the peaks and the corresponding location of the predicted bifurcation points are highlighted with circles.



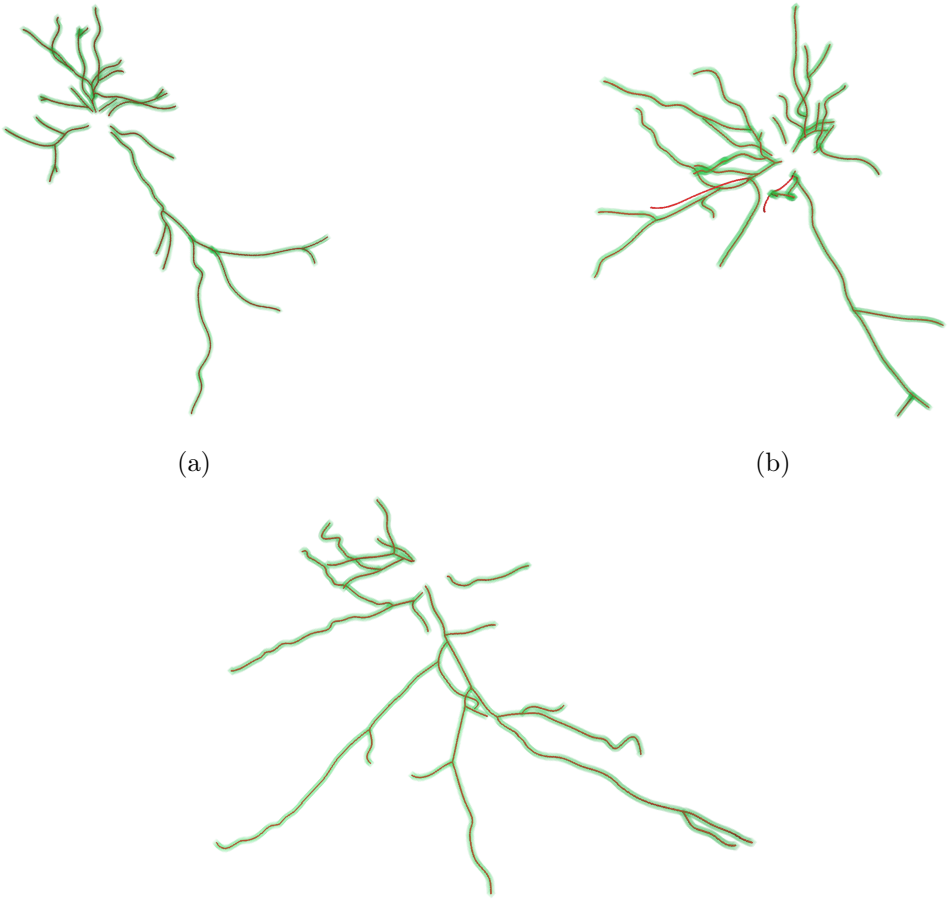
(a) Reconstruction of a branch before stopping. Bifurcation points are highlighted with circles.




(b) BifNet predictions, highlighting the peaks in red corresponding to bifurcation points, and drawing the peak threshold line.

Figure 19: Example of a reconstructed branch (19a) along with the predicted values for the bifurcation detection task (19b). For each step performed by the agent (using the distance map as input, although not displayed here), a cropped section of the input neuron distance map is fed to BifNet, which in turn outputs a predicted value.

Finally, when combining the contribution of the agent and of the bifurcation detection network, the proposed implementation can be tested on 3D neurons. The algorithm still requires starting points to be provided (unless starting at a random point is desired). Figure 20 below shows the predicted reconstruction on 3 testing neurons when providing the soma end points as starting points. All branches from neurons 20a and 20c were correctly identified and reconstructed when testing the model on them. In some instances, and as in the case of neuron 20b, the agent can get stuck at bifurcation points; either before, not being able to commit to a branch, and just oscillating between the two options, or when going back to a visited bifurcation and not being able to find a way to the unsegmented branch. This will cause the children branches downstream from this node to not be segmented. In neuron 20b the agent gets stuck on two different branches and leaves them unsegmented if no other starting point or no further assistance is provided, which is the main drawback of such iterative methods. On the other hand, if manual intervention is integrated to the framework, a starting segment could be added where the agent gets stuck, and the agent could keep going down the next branch.



(a) (b)

(c)

Figure 20: Testing the model on 3 neurons to obtain their reconstructed branches and connectivity. The reconstruction (in green) is stacked on top of the ground-truth of the neuron (in red). The model is only provided with the soma points as starting points in these tests. All branches from neurons (20a) and (20c) were accurately reconstructed. The agent got stuck on two branches on neuron (20b).

# 6 Application to Real Data

## 6.1 Individual Cubes

Being too large to fit in the memory, the large scale neuron images provided here are cropped into cubes of smaller shapes (around 250x250x250 pixels) and processed independently. For each cube, a distance map prediction is obtained from the U-Net and stored for faster accessibility. The same is done for the annotations, which are split into cubes. Cubes are all associated to a cube number along the 3 dimensions. For example the cube identifier [4,0,3] is used for the $5^{th}$ cube along the first axis, $1^{st}$ cube along the second, and $4^{th}$ cube along the last axis. Knowing the coordinates of the starting indices off all cubes, it is possible to navigate between an image-wise and a cube-wise system of coordinates. The agent is first trained and tested on some individual cubes which display quality predictions and are not crowded with branches. Figure

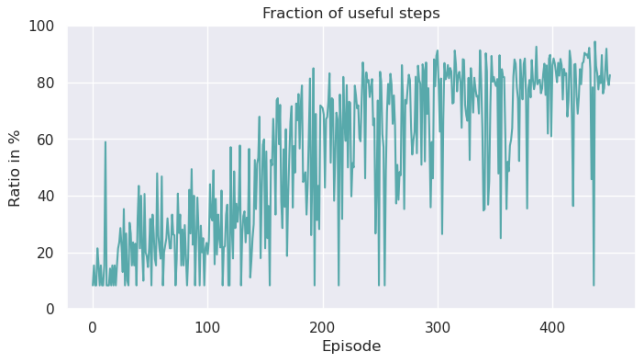
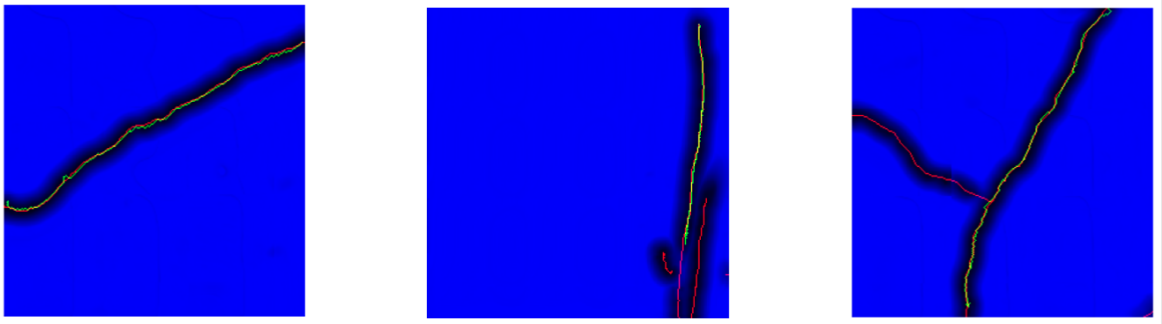



Figure 21: Number of useful steps over the total number of steps across episodes during training. A step is deemed as useful when reaching a ground-truth voxel that has not been previously visited.

21 shows the fraction of steps that contributed to increase the match between the agent's path and the ground-truth voxels during training. After 150 episodes, the agent stops diverging off-path and starts having a higher proportion of useful steps than unsuccessful steps, to eventually reach high proportions of on-path steps after 400 episodes. Note that the abort condition is activated during this training, meaning that this plot does only reflect the ability of the agent to stay on-path, but it does not reflect the cases when the agent got stuck on a branch. As



(a) (b) (c)

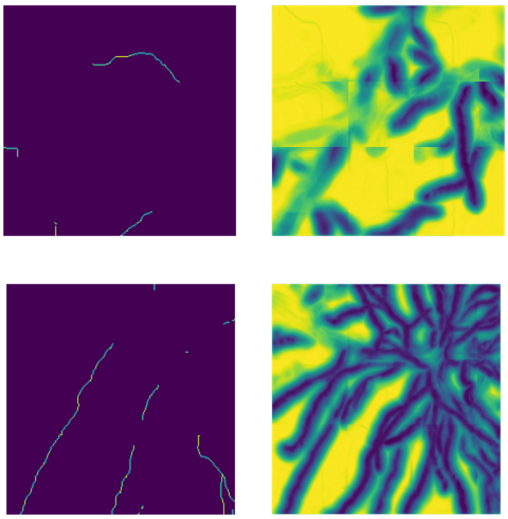
Figure 22: 3 examples of tests when using the prediction as input.

opposed to when using synthetic distance map input, the performance is a lot more unstable, even after a long training. The agent is still able to move along a branch, as depicted by a few examples on Figure 22a and 22c, but it can easily get stuck on many branches due to the high variability in the input that it receives. 22b is a typical example of how the agent could get tricked into thinking that the branch is stopping due to a discontinuity in the prediction.

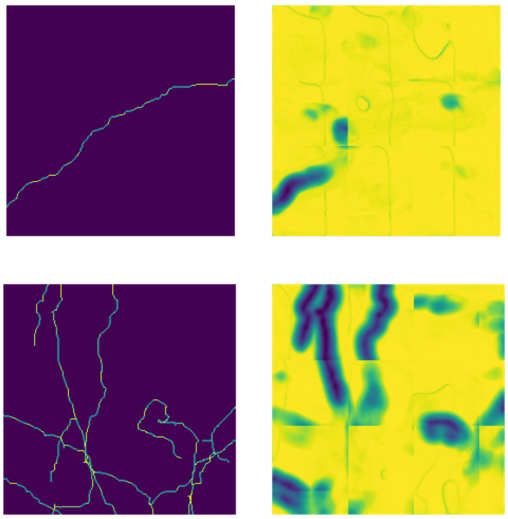
## 6.2 Large Scale Images

Moving onto the large scale neuron images, the environment has to be adapted in order to handle the loading of cropped predicted cubes of the full image. To deal with the large image size, the environment only loads the cubes necessary to start an episode, and other cubes are only loaded when reached by a sliding window around the agent's position. Because of the fact that a small crop is taken around the cursor's position, there will be missing information once that crop reaches the border of the loaded cube. As information from a neighboring cube might still be relevant to the current crop, the cubes surrounding the starting cube also need to be loaded. Although significantly increasing the time it takes to restart an episode, the loading of the 26 cubes around the starting cube is necessary and deemed more reliable than other available options such as padding the starting cube with a constant value (which would lead to the agent stopping when it should not). Once those cubes are loaded and combined into one single larger image, the agent can start moving the same way it does on smaller cubes. If the cursor goes beyond any of the 6 borders of the central cube, the next 9, 15 or 19 cubes (depending on whether 1, 2 or 3 borders were crossed at the same time by the moving cursor window) are loaded and a new large image combining those cubes is initialized for the agent to keep going.

By taking a closer look at the large scale neurons that are provided, they contain areas that are very crowded with branches (as in 23a). Such populated volumes actually account for most of the neurons total volume, excluding the axon and a few external neurites. Unfortu-



(a) Two examples of cubes with apparent gaps in the labels (projected in 2D).



(b) Two examples of cubes with apparent gaps in the predictions (projected in 2D).

Figure 23: Depictions of anomalies in real data in both labels (23a) and predictions (23b).

nately, anomalies on both the annotation side and the prediction side strongly challenge the applicability of the iterative framework, as exemplified on Figure 23. 23a displays two cases of substantially incomplete annotations, and 23b two cases of missing parts in the predicted distance maps. The discontinuity of the ground-truth poses a problem for an efficient training for three different reasons. First, the use of the abort condition, which was shown to be essential for training on synthetic data is compromised by gaps within individual branches, as it causes the agent to stop while in the middle of a branch. Second, the reward system as a whole is also compromised, because the agent can be negatively rewarded for actions that it rightly pursued, or the opposite, and will end up going against any previous learning. Finally, the fact that the branches are not fully connected and continuous make it impossible to extract the bifurcation points, required for training. Information about what points belong to what branch is also not made available through the discontinuous annotations, which is limiting when trying to train the agent in such crowded environments, as it needs to be dissuaded from jumping to a branch that it is not currently reconstructing. On the other hand, the missing parts in the predictions make it hard for the agent to reconstruct branches with such large gaps. When trained on a few instances of areas showing a combination of clean prediction and accurate annotation, the agent manages to follow through parts of branches that are closely bounded by other branches as in Figure 24.

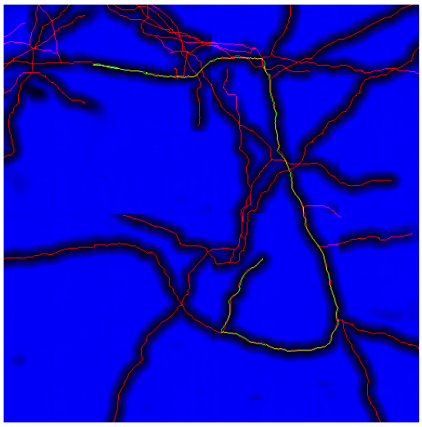


Figure 24: 2D projected example of the agent following a branch on a large scale 3D neuron using as input the U-Net prediction, with the ground-truth in red, the prediction in blue, and the agent's path in yellow/green.

Overall, the agent struggles to adapt to the high variability in the input data when using the predicted distance maps as input. The agent often gets stuck in the middle of branches, most likely due to the variations in tube thickness, tube contrast, and to sudden drops in intensity of the tubes that were not to observed using synthetic data. The unexpected changes occurring on cube borders, easily visible on Figure 23b, also confuse the agent, making it hard for the agent to replicate the performance that it can achieve on perfect distance maps.

To address that issue, having branch specific annotations could be a starting point; that is, having points in the annotations that are affiliated to their respective branches and ordered from start to end of the branch. The basic *swc* file format used to save neuron morphologies already accepts such valuable pieces of information so it could be easily integrated in any annotation process. Having such information would help with the design of a reward system that can

address the main issue of the agent in the proposed implementation. Considering that the agent getting stuck at a point remains the most common issue, both on synthetic and real data use cases, the need for a reward system that can prevent such behavior is strong. This way, information about the directionality of the next point can be integrated in the reward function (done in a similar framework by Zhang et al. (2018)) and behaviors that do not involve moving forward along branches could be penalized.

Another approach that is worth mentioning is that instead of using the distance maps predicted from the U-Net as inputs, we could imagine using the output from higher U-Net layers. The information contained in the higher layers, which normally gets processed on the way to the output layer, could potentially be relevant to the agent's decision making process. The higher level of abstraction from the higher layers might help the agent with its ability to handle variations in tube sizes and contrasts. However, the approach implemented here is more intuitive when it comes to adapting the model in function of a behavior observed while testing the agent. Indeed, any observed behavior can be put into perspective with its probable cause by looking at the distance map, *e.g.* by checking if there was an unexpected end to the traced tube in the input image, which would justify a stopping behavior in the reconstruction.

# 7 Conclusion

In this project, a novel Reinforcement Learning game was designed to match the application of neuron reconstruction. It was demonstrated that, starting with simple curves in 2 dimensions, an agent could eventually be trained to trace neurites by moving along their trajectories in 3 dimensions. Using the output from an independent bifurcation detection network, the proposed approach is able to reconstruct arborizations of neurites from synthetic images of full-neurons. Our implementation has the two advantages of representing the neuronal arborizations as a continuous graph structure and of guaranteeing the reconstruction of the tree topology.

Moreover, the implemented framework could easily be embedded into an interactive neuron reconstruction software. Indeed, the iterative nature of our reconstruction method leaves room for external interventions after any step. A user could be asked to select starting points and simply rely on the agent to reconstruct the downstream branches from there. If at any point, the agent gets stuck, the user could provide the next point to restart the agent's motion.

It was then demonstrated that, although with less stability, the agent can still be trained to successfully trace neurites when using data predictions generated from real microscopy scans. Due to discontinuities in the branch segments and the absence of bifurcation point locations in the annotations, the complete training framework that was shown to work on synthetic data could not be replicated. This project therefore highlights the need for a different annotation pipeline that would involve labelling the bifurcations and that would also guarantee the topology of the underlying tree structure. Having such detailed annotations is necessary to drive the research towards building models that can extract additional information about the connectivity of the reconstructed structures.

Recommendations for the continuation of the project are therefore three-fold :

- Improve the data annotation regime to better represent the topology of the neurons, by providing ordered points and continuous branches. That way, the location of the bifurcation points can be easily retrieved, and branch-wise annotations will be available.
- Once complete annotations are available, adapt the model to favor always moving forward along branches and to avoid getting stuck as much as possible. Integrating the ordering of the ground-truth points to the reward system should reduce the frequency of such issues.
- Integrate the proposed approach to a semi-automatic annotation pipeline to assess if the reconstruction process can be significantly sped up.

# 8 References


Bastani, F., He, S., Abbar, S., Alizadeh, M., Balakrishnan, H., Chawla, S., Madden, S., and DeWitt, D. (2018). RoadTracer: Automatic Extraction of Road Networks from Aerial Images. *arXiv:1802.03680 [cs]*. arXiv: 1802.03680.

Ghesu, F. C., Georgescu, B., Mansi, T., Neumann, D., Hornegger, J., and Comaniciu, D. (2016). An Artificial Agent for Anatomical Landmark Detection in Medical Images. In Ourselin, S., Joskowicz, L., Sabuncu, M. R., Unal, G., and Wells, W., editors, *Medical Image Computing and Computer-Assisted Intervention - MICCAI 2016*, Lecture Notes in Computer Science, pages 229–237, Cham. Springer International Publishing.

Hessel, M., Modayil, J., van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M., and Silver, D. (2017). Rainbow: Combining Improvements in Deep Reinforcement Learning. *arXiv:1710.02298 [cs]*. arXiv: 1710.02298.

Liao, R., Miao, S., de Tournemire, P., Grbic, S., Kamen, A., Mansi, T., and Comaniciu, D. (2016). An Artificial Agent for Robust Image Registration. *arXiv:1611.10336 [cs]*. arXiv: 1611.10336.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. (2013). Playing Atari with Deep Reinforcement Learning. *arXiv:1312.5602 [cs]*. arXiv: 1312.5602.

Peng, H., Xie, P., Liu, L., Kuang, X., Wang, Y., Qu, L., Gong, H., Jiang, S., Li, A., Ruan, Z., Ding, L., Chen, C., Chen, M., Daigle, T. L., Ding, Z., Duan, Y., Feiner, A., He, P., Hill, C., Hirokawa, K. E., Hong, G., Huang, L., Kebede, S., Kuo, H.-C., Larsen, R., Lesnar, P., Li, L., Li, Q., Li, X., Li, Y., Li, Y., Liu, A., Lu, D., Mok, S., Ng, L., Nguyen, T. N., Ouyang, Q., Pan, J., Shen, E., Song, Y., Sunkin, S. M., Tasic, B., Veldman, M. B., Wakeman, W., Wan, W., Wang, P., Wang, Q., Wang, T., Wang, Y., Xiong, F., Xiong, W., Xu, W., Yao, Z., Ye, M., Yin, L., Yu, Y., Yuan, J., Yuan, J., Yun, Z., Zeng, S., Zhang, S., Zhao, S., Zhao, Z., Zhou, Z., Huang, Z. J., Esposito, L., Hawrylycz, M. J., Sorensen, S. A., Yang, X. W., Zheng, Y., Gu, Z., Xie, W., Koch, C., Luo, Q., Harris, J. A., Wang, Y., and Zeng, H. (2020). Brain-wide single neuron reconstruction reveals morphological diversity in molecularly defined striatal, thalamic, cortical and claustral neuron types. Technical report, bioRxiv. Section: New Results Type: article.

Román, K. L.-L., de La Bruere, I., Onieva, J., Andresen, L., Holsting, J. Q., Rahaghi, F. N., Macía, I., González Ballester, M. A., and José Estepar, R. S. (2018). 3D Pulmonary Artery Segmentation from CTA Scans Using Deep Learning with Realistic Data Augmentation. *Image analysis for moving organ, breast, and thoracic images: third International Workshop, RAMBO 2018, fourth International Workshop, BIA 2018, and first International Workshop, TIA 2018, held in conjunction with MICCAI 2018, Granada,...*, 11040:225–237.

Schaul, T., Quan, J., Antonoglou, I., and Silver, D. (2016). Prioritized Experience Replay. Technical Report arXiv:1511.05952, arXiv. arXiv:1511.05952 [cs] type: article.

Tetteh, G., Efremov, V., Forkert, N. D., Schneider, M., Kirschke, J., Weber, B., Zimmer, C., Piraud, M., and Menze, B. H. (2020). DeepVesselNet: Vessel Segmentation, Centerline Prediction, and Bifurcation Detection in 3-D Angiographic Volumes. *Frontiers in Neuroscience*, 14:592352.

Winnubst, J., Bas, E., Ferreira, T. A., Wu, Z., Economo, M. N., Edson, P., Arthur, B. J., Bruns, C., Rokicki, K., Schauder, D., Olbris, D. J., Murphy, S. D., Ackerman, D. G., Arshadi, C., Baldwin, P., Blake, R., Elsayed, A., Hasan, M., Ramirez, D., Dos Santos, B., Weldon, M., Zafar, A., Dudman, J. T., Gerfen, C. R., Hantman, A. W., Korff, W., Sternson, S. M., Spruston, N., Svoboda, K., and Chandrashekar, J. (2019). Reconstruction of 1,000 Projection Neurons Reveals New Cell Types and Organization of Long-Range Connectivity in the Mouse Brain. *Cell*, 179(1):268–281.e13.

Zhang, P., Wang, F., and Zheng, Y. (2018). Deep Reinforcement Learning for Vessel Centerline Tracing in Multi-modality 3D Volumes. In Frangi, A. F., Schnabel, J. A., Davatzikos, C., Alberola-López, C., and Fichtinger, G., editors, *Medical Image Computing and Computer Assisted Intervention – MICCAI 2018*, volume 11073, pages 755–763. Springer International Publishing, Cham. Series Title: Lecture Notes in Computer Science.

Zhou, H., Li, S., Li, A., Huang, Q., Xiong, F., Li, N., Han, J., Kang, H., Chen, Y., Li, Y., Lin, H., Zhang, Y.-H., Lv, X., Liu, X., Gong, H., Luo, Q., Zeng, S., and Quan, T. (2021). GTree: an Open-source Tool for Dense Reconstruction of Brain-wide Neuronal Population. *Neuroinformatics*, 19(2):305–317.

Zhou, Z., Kuo, H.-C., Peng, H., and Long, F. (2018). DeepNeuron: an open deep learning toolbox for neuron tracing. *Brain Informatics*, 5(2):3.

Zhou, Z., Liu, X., Long, B., and Peng, H. (2016). TReMAP: Automatic 3D Neuron Reconstruction Based on Tracing, Reverse Mapping and Assembling of 2D Projections. *Neuroinformatics*, 14(1):41–50.