Combining Optimal Path Search With Task-Dependent Learning in a Neural Network

Tomas Kulvicius, Minija Tamosiunaite, and Florentin Wörgötter

***Abstract*— Finding optimal paths in connected graphs requires determining the smallest total cost for traveling along the graph's edges. This problem can be solved by several classical algorithms, where, usually, costs are predefined for all edges. Conventional planning methods can, thus, normally not be used when wanting to change costs in an adaptive way following the requirements of some task. Here, we show that one can define a neural network representation of path-finding problems by transforming cost values into synaptic weights, which allows for online weight adaptation using network learning mechanisms. When starting with an initial activity value of one, activity propagation in this network will lead to solutions, which are identical to those found by the Bellman–Ford (BF) algorithm. The neural network has the same algorithmic complexity as BF, and, in addition, we can show that network learning mechanisms (such as Hebbian learning) can adapt the weights in the network augmenting the resulting paths according to some task at hand. We demonstrate this by learning to navigate in an environment with obstacles as well as by learning to follow certain sequences of path nodes. Hence, the here-presented novel algorithm may open up a different regime of applications where path augmentation (by learning) is directly coupled with path finding in a natural way.**

***Index Terms*— Graph neural networks (GNNs), navigation, reinforcement learning (RL), sequences, shortest path problem.**

I. Introduction

THIS study addresses the so-called single-source shortest-path (SSSP) problem. Possibly, the most prominent goal for any path-finding problem is to determine a path between two vertices (usually called source and destination) in a graph, such that the total summed costs for traveling along this path are minimized. Usually, (numerical) costs are associated with the edges that connect the vertices, and in this way, costs are accumulated when traveling along a set of them. In the SSSP problem, the aim is to find the shortest paths (SPs) from one vertex (source) to all other remaining vertices in the graph.

The SSSP problem has a wide range of applications, e.g., in computer networks (SP between two computers), social networks (SP between two persons or linking between two persons), trading and finance (e.g., currency exchange), multiagent systems, such as games, task, and path planning in robotics, and so on, just to name a few.

The most general way to solve the SSSP problem is the Bellman–Ford (BF) algorithm [1], [2], [3], which can also deal with graphs that have some negative cost values. In this work, we present a neural implementation, which is mathematically equivalent to the BF algorithm with an algorithmic complexity, which is the same as for BF.

The neural implementation relies on the multiplication of activities (instead of adding of costs). As a consequence, it is directly compatible with (Hebbian) network learning, which is not the case for any additively operating path-planning algorithm. To demonstrate this, we are using Hebbian-type three-factor learning [4], [5], [6] to address SSSP tasks under some additional constraints, solved by the three-factor learning.

Our paper is structured as follows. First, we will provide an overview of the state-of-the-art methods and state our contribution with respect to that. Next, we will describe the details of the BF algorithm and the proposed neural network (NN-BF). This will be followed by a comparison between the BF algorithm and NN-BF and by examples of combining NN-BF-based planning with three-factor learning. Finally, we will conclude our study with a summary and provide an outlook for future work.

II. Related Work

A. State of the Art

There are mainly two types of approaches to solve the SSSP problem: classical algorithms and approaches based on artificial neural networks. Classical algorithms exist with different complexity and properties. The simplest and fastest algorithm is breadth-first search (BFS) [7], [8]. However, it can only solve the SSSP problem for graphs with uniform costs, i.e., all costs equal to 1. Dijkstra's algorithm [9] as well as the BF algorithm [1], [2] can deal with graphs that have arbitrary costs. From an algorithmic point of view, Dijkstra's algorithm is faster than the BF algorithm; however,

Manuscript received 24 May 2022; revised 12 July 2023; accepted 15 October 2023. This work was supported in part by the European Community's H2020 Program [Future and Emerging Technologies (FET)] under Grant 899265 (ADOPD) and in part by the German Science Foundation (DFG) under Grant WO 388/16-1. *(Corresponding author: Tomas Kulvicius.)*

Tomas Kulvicius is with the Department for Computational Neuroscience, and the University Medical Center Göttingen, Systemic Ethology and Developmental Science, Child and Adolescent Psychiatry and Psychotherapy, University of Göttingen, 37073 Göttingen, Germany (e-mail: tomas.kulvicius@phys.uni-goettingen.de).

Minija Tamosiunaite is with the Department for Computational Neuroscience, University of Göttingen, 37073 Göttingen, Germany, and also with the Faculty of Computer Science, Vytautas Magnus University, 53361 Kaunas, Lithuania.

Florentin Wörgötter is with the Department for Computational Neuroscience, University of Göttingen, 37073 Göttingen, Germany.

Color versions of one or more figures in this article are available at https://doi.org/10.1109/TNNLS.2023.3327103.

Digital Object Identifier 10.1109/TNNLS.2023.3327103

Dijkstra only works on graphs with positive costs, whereas the BF algorithm can also solve graphs where some of the costs are negative. Furthermore, Dijkstra is a greedy algorithm and requires a priority queue, which makes it not so well suited for parallel implementation as compared with BF.

It is worth noting that there are two versions of the BF algorithm (see [3]). Both versions have the same basic algorithmic complexity but version 2 operates on graph nodes and not on graph edges (as version 1), which allows implementing version 2 in a totally asynchronous way where computations at each node are independent of other nodes [3]. In spite of this, version 2 is largely ignored in the literature, and users will usually be directed to version 1 when doing a (web-)search.

Some heuristic search algorithms, such as A* [10] and its variants (see [11], [12], [13], [14]), exist, which are faster than Dijkstra or BF, but they can only solve the single-source single-target SP problem.

The most general, but slowest, algorithm to solve the SSSP problem is the Floyd–Warshall algorithm [15], [16], which finds all-pairs SPs (APSPs), i.e., all SPs from each vertex to all other vertices. Another way to solve the APSP problem is by using Johnson's algorithm [17], which utilizes BF and Dijkstra's algorithms and is—under some conditions—faster than the Floyd–Warshall algorithm (essentially, this is the case for sparse graphs).

Many different algorithms exist, which utilize artificial neural networks to solve SP problems. Some early approaches, which were dealing with relatively small graphs (below 100 nodes), were based on Hopfield networks [18], [19], [20] or Potts neurons and a mean field approach [21]. These approaches, however, may not always give optimal solutions or may fail to find solutions at all, especially for larger graphs.

Some other bioinspired neural networks were proposed [22], [23], [24], [25], [26], [27], [28] for solving path-planning problems. These approaches work on grid structures, where activity in the network is propagated from the source neuron to the neighboring neurons until activity propagation within the whole network is finished. SPs can then be reconstructed by following the activity gradients. The drawback of these approaches is, however, that they are specifically designed for grid structures and cannot be applied at general graphs with arbitrary costs.

Several deep learning approaches were proposed to solve SP problems, for example, using a deep multilayer perceptron (DMLP; [29]), fully convolutional networks [30], [31], [32], or a long short-term memory (LSTM) network [33]. Inspired by classical reinforcement learning (RL) algorithms, e.g., Q learning, SARSA [34], some deep RL algorithms had been employed [35], [36], [37] too. These approaches are designed to solve path-planning problems in 2-D or 3-D spaces and cannot deal with graphs with arbitrary costs.

In addition to this, deep learning approaches based on graph neural networks (GNNs) have been employed to solve path problems, too [38], [39], [40], [41], [42], [43], mostly in the context of relation and link predictions. While deep learning approaches may lead to a better run-time performance as compared with classical approaches due to fast inference, all deep learning-based approaches need to learn their (many) synaptic weights before they are functional. This usually requires a large amount of training data. Another disadvantage of these approaches is that optimal solution is not guaranteed, since networks intrinsically perform function approximation based on the training data. Moreover, in some of the cases, networks may even fail to find a solution at all, especially, in cases where training data are quite different from new cases (generalization issue for out-of-distribution data).

Recently, graph-based neural networks have been proposed too, such as spatial–temporal graph convolutional networks (ST-GCN) for skeleton-based action recognition [44], spatial–temporal graph network for video supported machine translation [45], and graph transformer network for zero-shot temporal activity detection (TN-ZSTAD) from videos [46]. Other deep networks are based on scene graph generation (SGG), e.g., conditional random field (CRF)-based SGG [47], TransE-based SGG [48], convolutional neural network (CNN)-based SGG [49], recurrent neural network (RNN)/LSTM-based SGG [50], [51], [52], and GNN-based SGG [53], [54]. While these approaches have been mostly applied for text and image/video analysis, such as text-to-image generation, image/video captioning, image retrieval, scene understanding, and human–object interaction (for a comprehensive survey, see [55]), none of them have been exploited to specifically address SP/SSSP problems.

Furthermore, some approaches exist for an automated search (tuning) of generative adversarial network (GAN) architectures, such as neural architecture search (NAS) [56], [57], AutoGAN [58], and automated generative adversarial network (AGAN) [59], and for the simultaneous learning of network architecture parameters and weights (ZeroNAS, [60]). Although these approaches are related to our proposed method, there are two principle differences. First, the aforementioned approaches are used to optimize network architectures or architecture and weights to perform discriminating tasks, e.g., image classification or object detection, whereas in our approach, we are dealing with the SSSP problem, where we build the network architecture and update connection weights based on local learning rules. Hence, we do not specifically optimize network architecture and weights to map input to outputs. Second, in contrast to NAS approaches, weights in our network correspond to costs, which allow using activity propagation within the network to find optimal solutions for the SSSP tasks.

B. Contribution

The comparison of the above discussed approaches for solving SP/SSSP problems is graphically summarized in Fig. 1, and details are provided in Table I. Here, we compare our approach with respect to the following criteria: 1) ability to deal with positive and negative costs; 2) solution optimality; and 3) algorithmic complexity. Given these criteria, one can see that our approach most closely relates to the neural approaches by Xia and Wang [61] and Zhang and Li [62]. The approach by Xia and Wang [61] utilizes a relatively complex RNN architecture. It can deal with optimal and negative costs and find optimal solutions, but it has a higher complexity

TABLE I
COMPARISON OF DIFFERENT METHODS FOR SOLVING SP AND SSSP PROBLEMS

Algorithm	**Problem: SP/SSSP**	**Numerical calculation basis**	**Cost type**	**Algorithmic procedure**	**Training**	Optimality	**Complexity**	**Online learning**
A* [10]	SP	Cost values	Only positive	Additive cost summation	NO	Optimal	$\mathcal{O}(bd)$	Not directly possible
Dijkstra [9]	SSSP	Cost values	Only positive	Additive cost summation	NO	Optimal	$\mathcal{O}(\|E\| + \|V\|^2)$ or $\mathcal{O}(\|E\| + \|V\| log\|V\|)$	Not directly possible
BF [1], [2]	SSSP	Cost values	Positive and negative	Additive cost summation	NO	Optimal	$\mathcal{O}(\|V\|\|E\|)$	Not directly possible
Recurrent neural network Xia, 2000 [61]	SP	Weight values = cost values	Positive and negative	Additive/Multiplicative activity propagation	NO	Optimal	$\mathcal{O}(\|V\|^2)$ per interation	Not shown
Neural network Zhang, 2017 [62]	SSSP	Weight values = cost values	Positive and negative	Additive activity propagation	NO	Optimal	Not stated	Not shown
DMLP [29]	SP	Weight values*	n/a	Additive/Multiplicative activity propagation	YES	Optimality not guaranteed	Not stated	n/a
FCN [30]–[32]	SP	Weight values*	n/a	Additive/Multiplicative activity propagation	YES	Optimality not guaranteed	Not stated	n/a
LSTM [33]	SP	Weight values*	n/a	Additive/Multiplicative activity propagation	YES	Optimality not guaranteed	Not stated	n/a
RL [34]	SSSP	Value tables	Rewards or punishments	Reward propagation	NO	Optimality not guaranteed**	$\mathcal{O}(n^2)$ or $\mathcal{O}(n^3)$ [63]	Reinforcement learning
Deep-RL [35]–[37]	SSSP	Weight values*	n/a	Additive/Multiplicative activity propagation	YES	Optimality not guaranteed	Not stated	Reinforcement learning
GNN [38]–[43]	SSSP	Weight values*	n/a	Additive/Multiplicative activity propagation	YES	Optimality not guaranteed	Not stated	n/a
T-POOL [28]	SSSP	Weight values = cost values	Only positive (uniform)	Additive/Multiplicative activity propagation	NO	Optimal	$\mathcal{O}(\|V\|\|L\|)$	n/a
NN-BF (this study)	SSSP	Weight values = inverse of cost values	Positive and negative	Multiplicative activity propagation	NO	Optimal	$\mathcal{O}(\|V\|\|E\|)$	Hebbian learning, Reinforcement learning

SP – single-source – single-target; SSSP – Sigle-Source Shortest Paths (single-source – multi-targets)
b – branching factor; d – depth of the solution: $|V|$ – number of nodes, $|E|$ – number of edges, $|L|$ – number of layers
*Weight values do not correspond to the costs
**RL usually does not converge to optimal solutions in high dimensional spaces within reasonable time

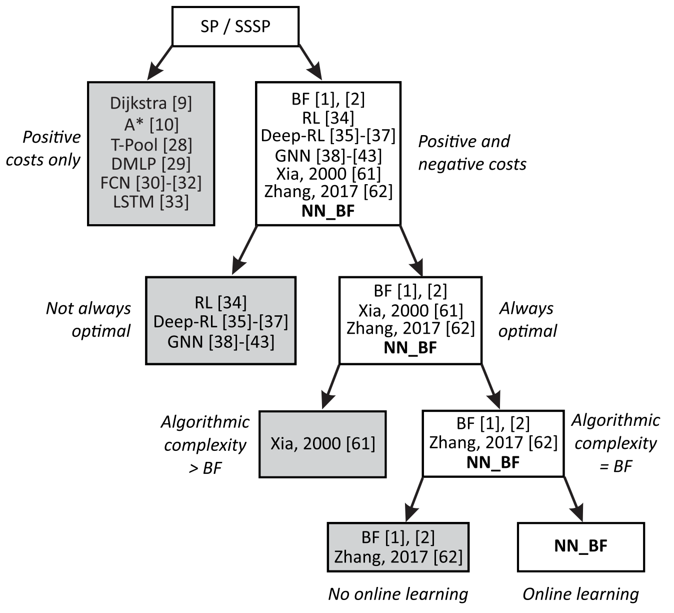

Fig. 1. Comparison of different methods for solving SP and SSSP problems.

as compared with the BF algorithm. Zhang and Li [62], as in our approach, proposed a neural implementation of the BF algorithm with the same complexity; however, it uses additive activity propagation (akin to additive cost propagation in the BF algorithm) as compared with multiplicative activity propagation as proposed by our algorithm. Moreover, neither Xia and Wang [61] nor Zhang and Li [62] show a combination of addressing the SP problem together with neural learning.

In summary, we present a neural implementation of the BF algorithm, which finds optimal solutions for graphs with arbitrary positive and (some) negative costs with the same algorithmic complexity as BF. The advantage of this is that, different from BF and other neural implementations of the BF algorithm [61], [62], we show that we can also directly apply neuronal learning rules that can dynamically change the activity in the network and lead to rerouting of paths according to some requirements. We show this by two cases where neuronal learning is used to structure the routing of the paths in a graph in different ways.

III. METHODS

A. Definitions

We denote a weighted graph as $G(V, E, C)$ with vertices V, edges E, and corresponding edge costs C. In the first version of the BF algorithm, a so-called relaxation procedure (cost minimization) is performed for all edges, whereas in the second version of the BF algorithm, relaxation is performed for all nodes. The relaxation procedure is performed for a number of iterations until convergence, i.e., approximate distances are replaced with shorter distances until all distances finally converge.

The BF algorithm is guaranteed to converge if relaxation of all edges is performed $|V| - 1$ times.[1] After convergence, SPs (sequences of nodes) can be found from a list of predecessor nodes backward from target nodes to the source node.

[1]Note that this corresponds to the worst case. See also algorithmic complexity analysis below.

Algorithm 1 Pseudocode of Neural Network Algorithm

Input: list of neuron connections (i, j) with weights $(w_{i,j})$, source neuron (s), and costs $(c_{i,j})$
Output: list of neuron activations (outputs), list of input nodes with maximal activity

```
for each neuron $i$ do
    $w[i, j] \leftarrow 1 - c[i, j]/K$                      ▷ Initialise connection weights based on costs
    $activations[i] \leftarrow 0$                           ▷ Initialise all neuron activations with 0
    $max\_inputs[i] \leftarrow null$                        ▷ Initialise all maximal input nodes with null
end for
$activations[s] \leftarrow 1$                               ▷ Initialise activation of the source neuron with 1

for $k \leftarrow 1$ to $N - 1$ do                          ▷ Repeat $N - 1$ times ($N$ - number of neurons)
    for each neuron $i$ do                                  ▷ For each neuron compute its activation
        for each input $j$ of neuron $i$ with weight $w[i, j]$ do
            $weighted\_input \leftarrow activations[j] \times w[i, j]$
            if $weighted\_input > activations[i]$ then
                $activations[i] \leftarrow weighted\_input$
                $max\_inputs[i] \leftarrow j$
            end if
        end for
    end for
    $activations[s] \leftarrow 1$                           ▷ Set activation of the source neuron to 1
end for

return $activations, max\_inputs$                           ▷ Return list of activations and list of max inputs
```

B. Neural Implementation of the BF Algorithm

In this section, we will describe our neural implementation of BF algorithm. Similar to BF, the proposed neural network finds SPs in a weighted graph from a source node (vertex) to all other nodes; however, SPs here are defined in terms of maximal activations from the source neuron to all other neurons.

The neural network has N neurons, where N corresponds to the number of vertices $|V|$ in a graph, connections between neurons i and j $(i, j = 1, \ldots, N)$ correspond to the edges, and connection weights correspond to the (inverse) costs of edges, which are computed by

$$w_{i,j} = 1 - \frac{c_{i,j}}{K} \tag{1}$$

where $K > 0$. We can prove that the solutions obtained with the neural network are identical to the solutions obtained with BF under certain constraints (see Section IV-A1).

We present the pseudocode of the neural network algorithm (NN-BF) in Algorithm 1. Similar to BF version 2 (see [3]), we run NN-BF for a number of iterations until convergence operating on the graph nodes (neurons). In every iteration, we update the activation (output) of each neuron by

$$a_i = \max_j \{a_j w_{i,j}\} \tag{2}$$

where a_j corresponds to the inputs of neuron i and $w_{i,j}$ to the connection weights between input neurons j and output neuron i. To start this process, we set the activation of the source neuron to 1. Thus, in this way, activity from the source neuron is propagated to all other neurons until activation has converged everywhere. After convergence, SPs (sequences of neurons) can be found from a list of maximal input activations backward from target neurons to the source neuron (similar to BF algorithm).[2]

The NN-BF algorithm is guaranteed to converge if activity of the network is updated $N - 1$ $(N = |V|)$ times. This can be shown by the worst-case scenario where neurons would be connected in a sequence, i.e., $1 \rightarrow 2 \rightarrow 3 \rightarrow \cdots \rightarrow N$. Thus, to propagate activity from the first neuron to the last neuron in the sequence, one would need to perform $N - 1$ updates.

A comparison of solving a simple graph using BF (version 1) and NN-BF is presented in Fig. 2. Here, we used a simple directed weighted graph with four nodes and six edges with one negative cost and five positive costs. We show the resulting distances in Fig. 2(a) and network activations in Fig. 2(b) after each iteration (numbers inside the nodes/neurons of the graph/neural network). Red edges/connections denote distance/activity propagation from the source node/neuron to all other nodes/neurons. For more details, please refer to the figure caption. In this case, the algorithm converged after two iterations when using BF and after one iteration when using NN-BF.

C. Performance Evaluation of the Neural Planner

We evaluated the performance of the neural network and compared it against the performance of the BF algorithm (version 1) with respect to the following: 1) solution equality; 2) algorithmic complexity; and 3) number of iterations until convergence.

[2]The source code is available at https://doi.org/10.5281/zenodo.10029797.

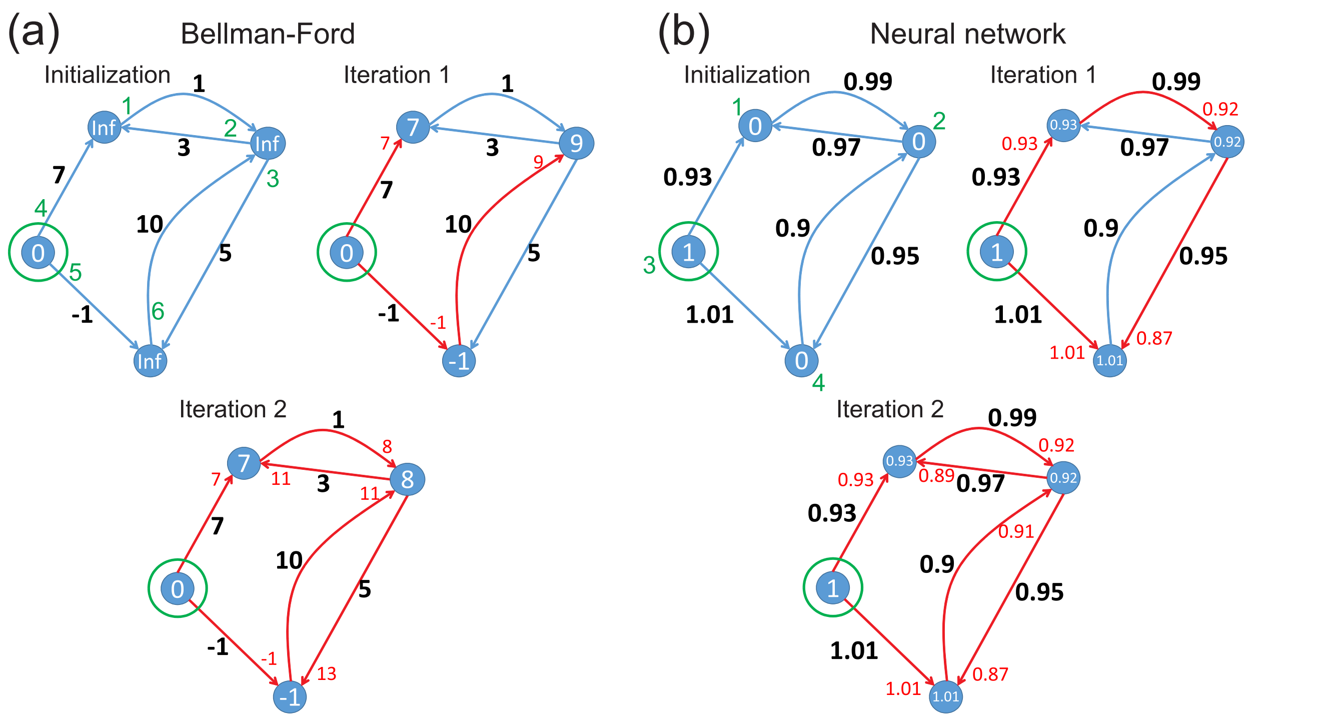

Fig. 2. Example of solving a directed weighted graph with four vertices and six edges using (a) BF and (b) NN-BF. Green circle denotes the source vertex/neuron. Black numbers correspond to the edge/connection weights. Red numbers correspond to costs/activations from the source vertex/neuron for the respective edges/inputs. Numbers inside graph vertices/neurons correspond to costs/activations from the source to the respective nodes/neurons. We used $K = 100$ to convert edge costs to connections weights, i.e., $w_i = 1 - c_i/100$ ($i = 1$–6). Green numbers in (a) correspond to the numbers of edges and the order in which they were processed, whereas green numbers in (b) correspond to numbers of neurons and their order in which their were processed.

For the numerical evaluation, we generated random directed weighted graphs of different density, from sparse graphs (only few connections per node) to fully connected graphs (every node connects to all other nodes). To generate graphs of different densities, we used the probability p_e to define whether two nodes will be connected or not. For example, a graph with 100 nodes and $p_e = 0.1$ will obtain a 10% connectivity rate and, thus, to ≈10 edges per node and ≈1000 edges in total.

Costs for edges were assigned randomly from specific ranges (e.g., $[-10, -1]$ and $[1, 100]$), where negative weights were assigned with probability p_n (e.g., $p_n = 0.1$) and positive weights with probability $1 - p_n$. The specific parameters for each evaluation case will be provided in Section IV.

D. Combining Neural Planning Network With Neural Plasticity

The neural implementation of the BF algorithm allows us to use neuronal learning rules in order to dynamically change connection weights between neurons in the network and, thus, the outcome of the neural planner. The methods applied to achieve this will be presented directly in front of the corresponding results to allow for easier reading.

IV. Results

A. Comparison of BF and NN-BF

In the following, we first provide a comparison between BF and NN-BF, and then, we will show results for NN-BF also on network learning.

1) Solution Equality: To prove equivalence, one can pairwisely compare two paths A and B between some source and a target, where A is the best path under the BF condition and B is any another path. Path A is given by nodes a_i and B by b_i. As described above, for a BF cost of a_i, we define the weight of the NN-BF connection as $w_i = 1 - (a_i/K)$ and likewise for b_i. We need to show that

$$\text{if } \sum_{i=1}^{n} a_i < \sum_{i=1}^{n+m} b_i \text{ then } \prod_{i=1}^{n}\left(1 - \frac{a_i}{K}\right) > \prod_{i=1}^{n+m}\left(1 - \frac{b_i}{K}\right). \tag{3}$$

Note that, in general, we can assume, for both sides, a path length of n, because—if one path is shorter than the other—we can just extend this by some dummy nodes with zero cost, such that the BF costs still add up correctly to the total.

First, we analyze the simple case of $n = 2$. The general proof, below, makes use of the structural similarity to the simple case. Thus, we show

$$\begin{aligned} a_1 + a_2 &< b_1 + b_2 \\ &\Leftrightarrow \left(1 - \frac{a_1}{K}\right)\left(1 - \frac{a_2}{K}\right) \\ &> \left(1 - \frac{b_1}{K}\right)\left(1 - \frac{b_2}{K}\right). \end{aligned} \tag{4}$$

We start from the second line in (4) and rewrite it as follows:

$$\begin{aligned} &\frac{1}{K^2}\left[K^2 - K(a_1 + a_2) + a_1 a_2\right] \\ &\qquad > \frac{1}{K^2}\left[K^2 - K(b_1 + b_2) + b_1 b_2\right]. \end{aligned} \tag{5}$$

We simplify and divide by K, where $K > 0$, to

$$-a_1 - a_2 + \frac{a_1 a_2}{K} > -b_1 - b_2 + \frac{b_1 b_2}{K}. \tag{6}$$

Now, we let $K \to \infty$ and get

$$a_1 + a_2 < b_1 + b_2 \tag{7}$$

which proves conjecture (4) for large enough K.

To generalize this, we need to show that (3) holds. For this, we analyze the second inequality given by

$$\prod_{i=1}^{n}\left(1 - \frac{a_i}{K}\right) > \prod_{i=1}^{n}\left(1 - \frac{b_i}{K}\right). \tag{8}$$

From the structure of (5), we can deduce that

$$\begin{aligned}\prod_{i=1}^{n}\left(1 - \frac{a_i}{K}\right) &= \frac{1}{K^n}\Bigg(K^n - K^{n-1}\sum_i a_i \\ &\quad + K^{n-2}\sum_{i\neq j} a_i a_j - K^{n-3}\sum_{i\neq j\neq k} a_i a_j a_k \pm \cdots\Bigg)\end{aligned} \tag{9}$$

and likewise for b_i, which allows writing out both sides of inequality (3). When doing this (not shown to save space), we see that the fore factor $(1/K^n)$ can be eliminated. Then, the term K^n also subtracts away from both sides. After this, we can divide the remaining inequality as in (6), here using K^{n-1} as divisor. This renders

$$\begin{aligned}&-\sum_i a_i + \frac{1}{K}\sum_{i\neq j} a_i a_j - \frac{1}{K^2}\sum_{i\neq j\neq k} a_i a_j a_k \pm \cdots \\ &> -\sum_i b_i + \frac{1}{K}\sum_{i\neq j} b_i b_j - \frac{1}{K^2}\sum_{i\neq j\neq k} b_i b_j b_k \pm \cdots \\ &\leftrightarrow \sum_i a_i + D_a < \sum_i b_i + D_b\end{aligned} \tag{10}$$

using the correct individual signs for D. Now, for $K \to \infty$, the disturbance terms D vanish on both sides, and we get

$$\sum_i a_i < \sum_i b_i \tag{11}$$

as needed.

We conclude that for $K \to \infty$, the neural network is equivalent to the BF algorithm.

In the following, we will show by an extensive statistical evaluation that the distribution of K for different realistic graph configurations is well behaved, and we never found any case where K had to be larger than 10^6.

Evidently, K can grow if $\sum_i a_i \approx \sum_i b_i$. This can be seen easily, as, in this case, we can express $\sum_i b_i = \sum_i a_i + \epsilon$, with ϵ a small positive number. When performing this setting, we get from (10) that $\epsilon > D_b - D_a$, which can be fulfilled with large values for K.

We have set the word "can" above in italics, because a broader analysis shows that—even for similar sums—K gets somewhat larger only for very few cases only as can be seen from Fig. 3.

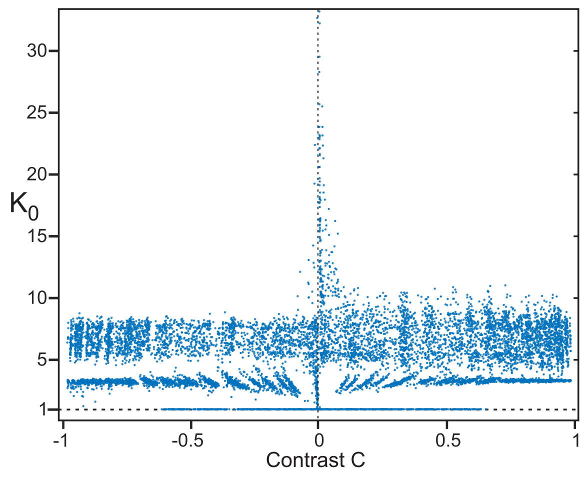

Fig. 3. Statistics for K computed from a large number of two-path combinations. Contrast $C = (\sum_i a_i - \sum_i b_i)/(\sum_i a_i + \sum_i b_i)$. For $K > K_0$, NN-BF renders identical results as BF.

For this, we performed the following experiment. We defined two paths A and B with different lengths L_A and L_B, where L was taken from $L \in \{2, 3, 4, 5, 6, 7, 8, 10, 20, 30, 40, 50, 100, 200\}$, and all combinations of L_A and L_B were used. We evaluated 60 instances for each path pair (in total 11 760 path pairs). To obtain those, we generated the cost for all edges in both paths using Gaussian distributions, where mean m and variance σ are chosen for each trial randomly and separately for A and B with $1 \leq m \leq 21$ and $0 \leq \sigma \leq 5$.

In Fig. 3, we are plotting on the horizontal axis the "contrast" between the summed cost of paths A and B given by $C = ((\sum_i a_i - \sum_i b_i)/(\sum_i a_i + \sum_i b_i))$. This allows comparing sums with quite different values. The vertical axis shows K_0, where for $K > K_0$, the neural network renders identical results as BF.

Note that we have truncated this plot vertically, but only 15 values of K_0 had to be left out, all with contrast values $C \simeq 0$, where the largest was $K_0 = 812$ with a contrast of $C = 5.3349 \times 10^{-5}$. Hence, only, for small contrast, a few larger values of K_0 are found. Note also that the asymmetry with respect to positive versus negative contrast is expected, because fewer negative costs exist in paths A and B due to our choice of $1 \leq m \leq 21$.

Considering only pairwise paths is of limited practical use. From such a perspective, it is better to consider different graphs with certain cost ranges and different connection densities where one should now ask how likely it would be that results of BF and NN-BF are mismatched due to a possible K divergence. To assess this, we had calculated the statistics for three different graphs with sizes 500, 1000, and 2000 nodes and randomly created costs taken from three different uniform cost distributions with intervals: [1, 10], [1, 100], and [1, 1000]. We considered four different connectivity densities for each graph with 5%, 10%, 50%, and 100% connections each. Fig. 4 shows that the maximal K_0 found was 10^6; hence, that in all cases $K \geq 10^6$ will suffice, where such a high value is only needed for sparse graphs and large costs.

In addition, we have also performed experiments where we analyzed the influence of the values of K on the success

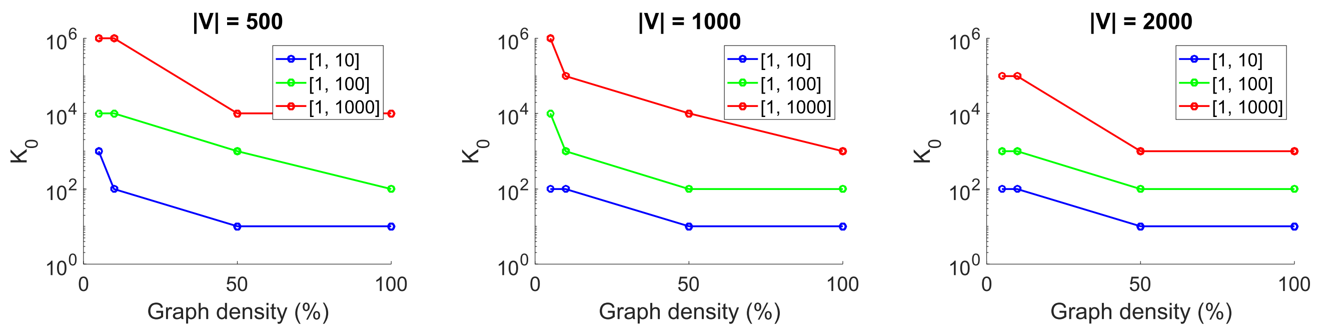

Fig. 4. Analysis of K_0 values for graphs of different sizes (500, 1000, and 2000), different densities (5%, 10%, 50%, and 100%), and different cost ranges ([1, 10], [1, 100], and [1, 1000]). For cases $|V| = 500$ with graph densities 5% and 10% and cost range [1, 1000], and for $|V| = 1000$ with graph density 5% and cost range [1, 1000], 10 000 random graphs were used, whereas in all other cases, 1000 graphs were used.

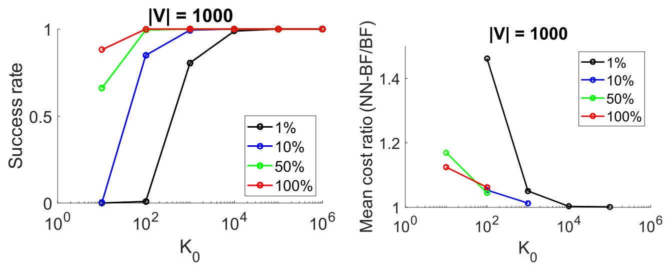

Fig. 5. Success rate of optimal solutions (left) and mean cost ratio (NN-BF cost/BF cost) of suboptimal solution versus K_0 value. The 1000 randomly generated graphs with costs in range [1, 100] were used with four different densities (1%, 10%, 50%, and 100%). Note that for the graphs with 1% density and with $K_0 = 10$, solutions were not found, and for the graphs with 10% density and with $K_0 = 10$, solutions were found for four graphs and were optimal, whereas for all other remaining graphs, solutions were not found.

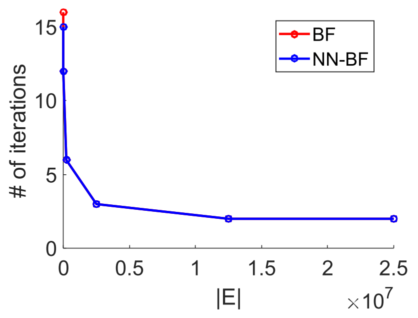

Fig. 6. Results of convergence analysis. Maximal number of iterations until convergence of BF version 1 (BF) or NN-BF versus number of edges $|E|$. The 500 randomly generated graphs with 5000 nodes were analyzed for each case.

rate for finding optimal solutions and on the cost ratio of the suboptimal solutions as compared with the solutions found by the BF algorithm. For this, we randomly generated 1000 graphs with 1000 nodes and four different densities: 1%, 10%, 50%, and 100% connections (fully connected graph), and cost range [1, 1000]. Statistics are shown in Fig. 5 where we show success rate of optimal solutions (left) and mean cost ratio (NN-BF cost/BF cost) of suboptimal solutions for $K_0 = 10, 10^2, 10^3, 10^4, 10^5, 10^6$. Results demonstrate that $K_0 \geq 10^4$ values will lead to convergence to optimal solutions in all cases except for very sparse graphs (1% and 10% connectivity); however, even in these cases, the number of suboptimal solutions is very low (below 1% and suboptimal solutions are very close to the optimal solutions (cost ratio < 1.004). With $K_0 = 10^6$, optimal solutions were obtained in all cases. This shows that the choice of K is noncritical in almost all cases.

2) Algorithmic Complexity of the Algorithms: In the following, we will show that all three algorithms have the same algorithmic complexity.

BF version 1 consists of two loops (see [3]), where the outer loop runs in $\mathcal{O}(|V|)$ time and the inner loop runs in $\mathcal{O}(|E|)$, where $|V|$ and $|E|$ define the number of graph vertices (nodes) and edges (links), respectively. In the worst case, BF needs to be run for $|V| - 1$ iterations and in the best case only for one iteration. Thus, the worst algorithmic complexity of BF is $\mathcal{O}(|V||E|)$, whereas the best is $\mathcal{O}(|E|)$.

The algorithmic structure of the second version of the BF algorithm (see [3]) is the same as that of our NN-BF algorithm (Algorithm 1). Hence, their complexity is the same, and we will provide the analysis for the NN-BF algorithm. NN-BF operates on neurons and their inputs and not on edges of the graph as in BF version 1. Hence, NN-BF is similar to BF version 2 that operates on the nodes. The outer loop, as in BF version 1, runs in $\mathcal{O}(N)$, where N is the number of neurons and corresponds to the number of vertices $|V|$. The second loop iterates through all neurons, and the third loop iterates through all inputs of the respective neuron and runs in $\mathcal{O}(N)$ and $\mathcal{O}(I_j)$, respectively. Here, I_j corresponds to the number of inputs of a particular neuron j. Given the fact that $N = |V|$ and $\sum_j I_j = |E|$ $(j = 1, \dots, N)$, we can show that the worst and the best algorithmic complexity of the NN-BF algorithm is the same as for BF version 2, i.e., $\mathcal{O}(|V||E|)$ and $\mathcal{O}(|E|)$, respectively. Hence, this is the same complexity as for BF version 1, too.

3) Convergence Analysis: The worst-case scenario would be to run the algorithms for $|V|-1$ iterations, where $|V|$ is the number of graph nodes. However, in practice, this will be not needed, and significantly, fewer iterations will usually suffice. Thus, we tested how many iterations are needed until cost convergence of the BF algorithm (version 1) as compared with activation convergence of the neural network. In both cases, we stopped the respective algorithm, as soon as its node costs or neuron activations were not changing anymore.

For this analysis, we used 500 randomly generated graphs with 5000 nodes with positive costs ($p_n = 0$) from the range [1, 10] and connectivity densities corresponding to approximately 12.5×10^3, 25×10^3, 250×10^3, 2.5×10^6, 12.5×10^6, and 24.995×10^6 edges.

The results are shown in Fig. 6, where we show the maximal number of iterations obtained from 500 tested graphs until

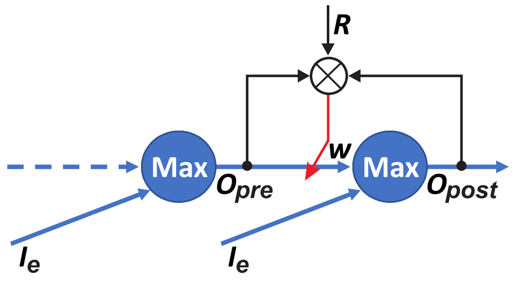

Fig. 7. Schematic of the learning architecture. I_e—external input. R—reward signal. O_{pre} and O_{post}—output of presynaptic and postsynaptic neuron, respectively. w—synaptic weight between pre- and postsynaptic neuron. The multiplication symbol followed by a red arrow indicates that the synaptic weight w is changed by a three-factor multiplicative (correlation) process [see (12)].

convergence. One can see that in case of sparse graphs 16 for BF and 15 for NN-BF iterations suffice for convergence and for denser graphs (>10% connectivity), only three iterations are needed. The resulting run time per one iteration on a sparse graph with 5000 nodes and 12 500 edges is 21 and 59 μs for BF (version 1) and NN-BF, respectively. Run time on a fully connected graph with 24 995 000 edges is 26 and 14 ms for BF and NN-BF, respectively.[3] Thus, relatively large graphs can be processed in less than 100 ms.

B. Network Learning Rule

As described above, the SP in the network between a source and a target neuron is given by that specific sequence of connected neurons, which leads to the largest activation at the target neuron. Hence, learning that modifies connection weights (and, thus, also the resulting activations) can alter path-planning sequences.

In this section, we, thus, define a Hebbian-type synaptic learning rule to modify connection weights [4], [6], [64], which will later be used in some examples.

In general, Hebbian plasticity is defined as a multiplicative process where a connection between two neurons—its synapse—is modified by the product of incoming (presynaptic) with outgoing (postsynaptic) activity of this neuron [4]. Often, a third factor R, by means of an additional incoming activation, enters this correlation process also in a multiplicative way to provide additional control over the weight change, leading to a three-factor rule (see [5] for a discussion of three-factor learning). We now first present our three-factor Hebbian learning scheme and rule, and then, we present two learning scenarios, namely, navigation learning and sequence learning.

To allow for learning (see schematic in Fig. 7), neurons will also receive additional external input $I_e = 1$ signaling occurrence of specific events, for instance, if an agent is at the specific location in an environment (otherwise, it will be set to $I_e = 0$). The network will receive external inputs I_e only during learning but not during planning. The output of a postsynaptic neuron, O_{post}, is computed as described above using (2). Hence, $O_{\text{post}} = \max\{O_{\text{pre}}, I_e\}$.

[3] C++ CPU implementation on Intel Core i9-9900 CPU, 3.10 GHz, and 32.0-GB RAM.

The synaptic weight w between pre- and postsynaptic neuron is then changed according to

$$\Delta w = \alpha O_{\text{pre}} O_{\text{post}} R \tag{12}$$

where $\alpha > 0$ is the learning rate, usually set to small values to prevent too fast learning. The driving force of the learning process is the reward signal R, which can be positive or negative and will lead to weight increase [long-term potentiation (LTP)] or decrease [long-term depression (LTD)], respectively. This learning rule represents, thus, a three-factor rule [5], where, here, the third-factor is the reward signal R.

A diagram of the learning scheme is presented in Fig. 7, where we show the components of the learning rule to change the synaptic weight between two neurons in the planning network.

Note that in our learning scheme, weights were bounded between 0 and 1, i.e., $0 \le w < 1$.

C. Combining Learning With Path Planning—Two Examples

In the following, we will show the examples of network learning combined with path finding. Examples stem from two common problem sets (navigation learning and sequence learning), and we can demonstrate here the main advantage of NN-BF, which is that learning and path finding rely on the same numerical representation. To achieve the same with BF (or any other additively operating algorithm), one would have to transform costs (for path finding) to activations (for network learning) back and forth. Examples are kept small, but upscaling to large problems is straightforward.

1) Navigation Learning: In the first learning scenario, we considered a navigation task in 2-D space, where an agent had to explore an unknown environment and plan the SP from a source location to a destination.

We assume a rectangular arena; see Fig. 8(a) and (b), iteration 0, with some locations (blue dots) and obstacles (gray boxes). The locations are represented by neurons in the planning network, whereas connections between neurons correspond to possible transitions from one location to other neighboring locations. Note that some transitions between neighboring locations, and, thus, connections between neurons are not possible due to obstacles. The positions of locations and possible transitions between them were generated in the following way. The position of a location k is defined by

$$(x_k, y_k) = \left(hi + \xi_i, h\ j + \xi_j\right) \tag{13}$$

where $i, j = 1, 2, \ldots, m$ and ξ is noise from a uniform distribution $\mathcal{U}(a, b)$. In this study, we used $m = 5$ ($k = m^2 = 25$), $h = 0.2$, $a = -0.05$, and $b = 0.05$.

Possible transitions between neighboring locations and, thus, possible connections between neurons were defined based on a distance threshold θ_d between locations, i.e., connections were only possible if the Euclidean distance between two locations $d_{k,l} = ||(x_k, y_k) - (x_l, y_l)|| \le \theta_d$. In one case [Fig. 8(a)], we used $\theta_d = 0.25$, and in another case [Fig. 8(b)], we used $\theta_d = 0.35$.

Synaptic weights between neurons were updated after each transition (called iteration) from the current location k (presynaptic neuron) to the next neighboring location l

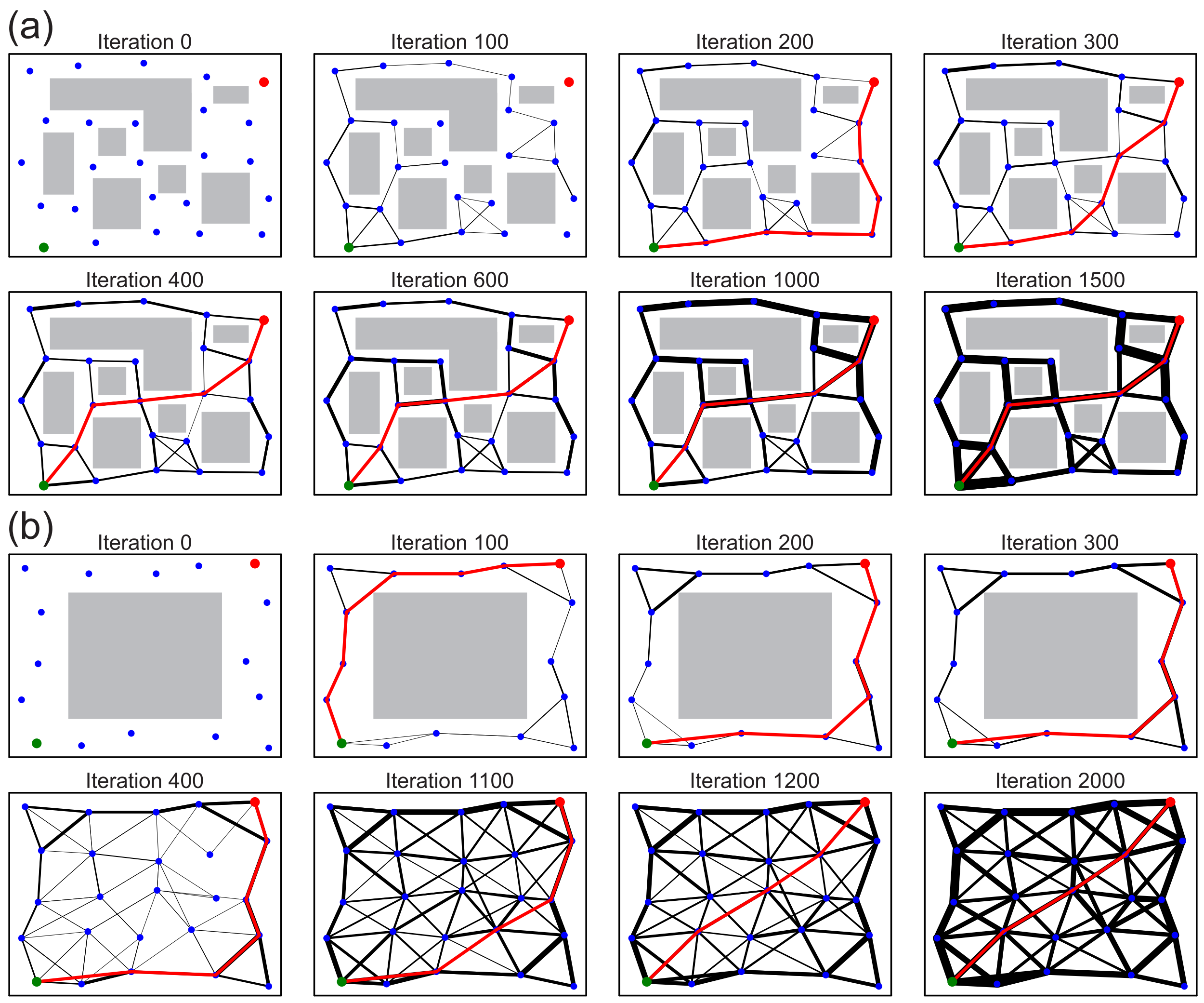

Fig. 8. Results of navigation learning. (a) Static environment. (b) Dynamic environment. Blue dots correspond to locations represented by neurons, and black lines correspond to connections between neurons (paths between locations) where line thickness is proportional to the synaptic weight strength. Gray blocks denote obstacles. Green and red dots correspond to the start point and the endpoint of the path marked by the red trajectory.

(postsynaptic neuron) according to the learning rule as described above, where pre- and postsynaptic neurons obtain an external input $I_e = 1$ during this transition (otherwise $I_e = 0$). In the navigation scenario, the reward signal R was inversely proportional to the distance between neighboring locations $d_{k,l}$, i.e., $R = 1 - \beta d_{k,l}$ (here, we used $\beta = 1$). Thus, smaller distances between locations lead to larger reward values and larger weight updates, and vice versa. In the navigation learning scenario, initially, all weights were set to 0, and the learning rate was set to $\alpha = 0.02$. The learning and planning processes were repeated many times, where the learning process was performed for 100 iterations, and then, a path from the source (bottom-left corner) to the destination location (top-right corner) was planned using current state of the network.

We used two navigation learning cases. In the first case, a static environment with multiple obstacles was used where obstacles were present during the whole experiment [see Fig. 8(a)], whereas in the second case, a dynamic environment was simulated, where one big obstacle was used for some period of time and then was removed some time later [see Fig. 8(b)].

Results for navigation learning in a static environment are shown in Fig. 8(a), where we show the development of the network's connectivity (black lines; line thickness is proportional to the weight strength) and the planned path (red trajectory) based on the network's connectivity during the learning process. After 100 learning iterations, the environment was only partially explored. Therefore, the full path could not yet be planned. After 200 iterations, only a suboptimal path was found, since the environment was still not completely explored. As exploration and learning process proceeds, connections between different locations get stronger (note thicker lines in the second row), and eventually, the planned path converges to the SP (after 400 iterations).

In Fig. 8(b), we show results for navigation in a dynamic environment. Here, we first run the learning process until the path has converged (see the top row) and then remove the obstacle after 300 learning iterations. As expected, after obstacle removal, the middle part of the environment is explored, and new connections between neurons are built. Eventually, as the learning proceeds, the output of the planning network converges to the SP (after 1200 iterations).

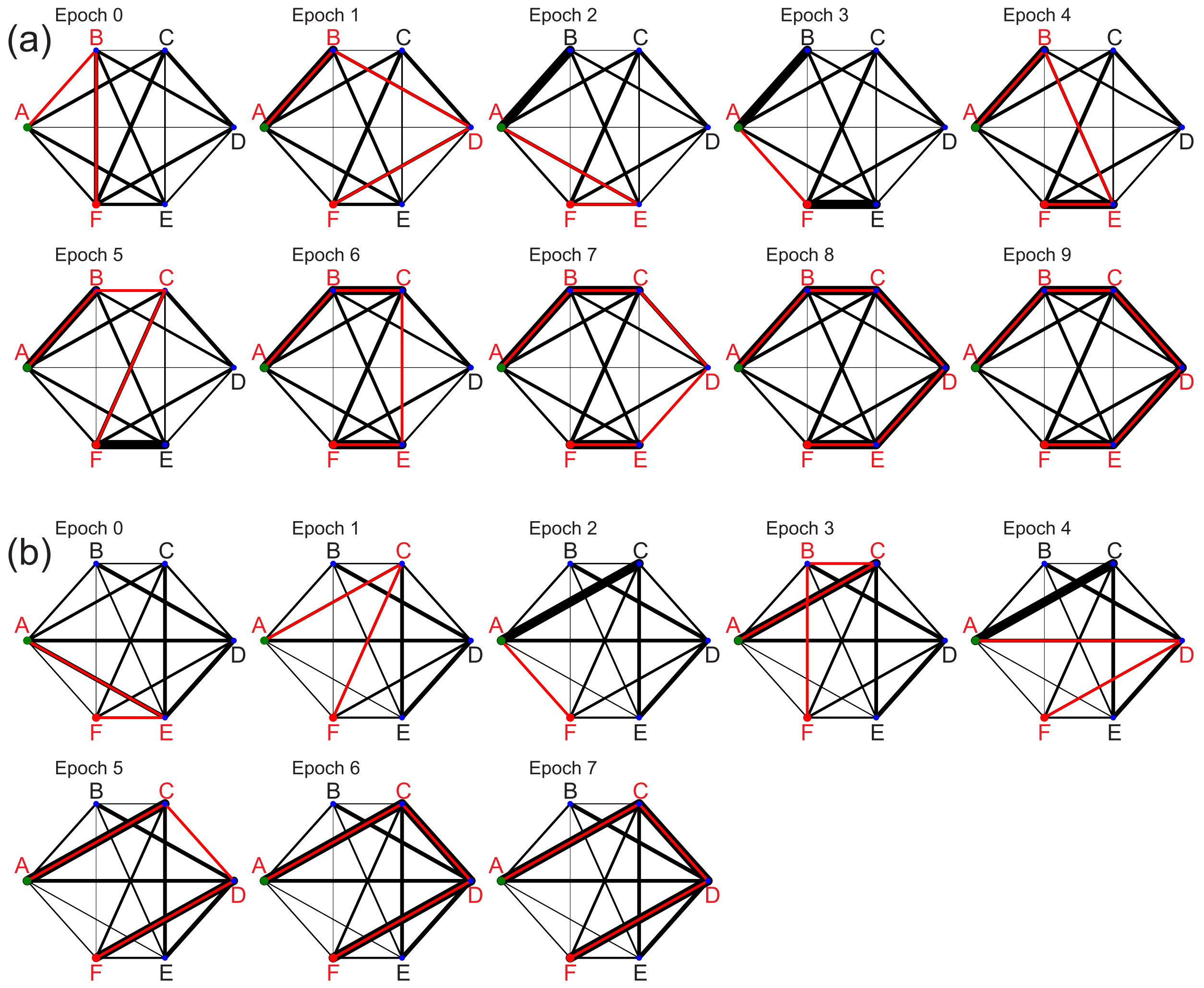

Fig. 9. Results of sequence learning. (a) Learning of sequence $A\ B\ C\ D\ E\ F$. (b) Learning of sequence $A\ C\ D\ F$. Blue dots correspond to letters represented by neurons, and black lines correspond to connections between neurons where line thickness is proportional to the synaptic weight strength. Green and red dots correspond to the start point and endpoint of the sequence marked by the red trajectory.

Due to the learning rule used, in both cases, systems converge to the shortest Euclidean distance paths.

2) Sequence Learning: In the second learning scenario, we were dealing with sequence learning, where the task was to learn executing a predefined sequence of arbitrary events, which, in our study, were denoted by letters. For instance, this could correspond to an action sequence in order to execute some task by a robot and could be used for task planning in robotics to generate optimal plans. Note that sequences do not necessary have to be predefined (as in our example), but optimal sequences can be learned based on obtained rewards during action execution.

In this case, we used a fully connected network (without self-connections) where neurons correspond to letters. In total, we used six different events (letters), i.e., A–F [see Fig. 9(a)]. Initially, all weights were set to random values from a uniform distribution $\mathcal{U}(0, 0.5)$. Here, the planning and learning procedures are performed in epochs, where we execute the sequence based on the planner's outcome always starting with letter A and ending with letter F and perform weight updates for each pair in the planned sequence. Suppose that, at the beginning, the planner generates a sequence $A\ B\ F$, then there will be two learning iterations in this epoch, i.e., weight update for the transition from A to B and from B to F. As in the navigation learning scenario, pre- and postsynaptic neurons will receive external input $I_e = 1$ whenever a certain transition from one event to the next event happens, e.g., from A (presynaptic neuron) to B (postsynaptic neuron).

Different from the navigation scenario, here, we used positive and negative rewards, i.e., $R = 1$ if a sequence pair is correct and $R = -1$ otherwise. In case of our previous $A\ B\ F$ example, this would lead to an increase of the synaptic weight between neurons A and B (correct) and to a decease of the synaptic weight between neurons B and F (incorrect; after B should be C). The learning procedure is repeated for several learning epochs until convergence. Here, we used the relatively high learning rate $\alpha = 0.9$, which allows fast convergence to the correct sequence.

We performed two sequence learning cases, where in the first case, we were learning a sequence consisting of all possible six events $A\ B\ C\ D\ E\ F$, and in the second case, a shorter sequence $A\ C\ D\ F$ had to be learned.

Results for learning of the full sequence $A\ B\ C\ D\ E\ F$ and of the partial sequence $A\ C\ D\ F$ are presented in Fig. 9(a) and (b), respectively. As in the navigation learning example, we show the development of the network's connectivity during the learning process. In Fig. 9(a), we can see that before learning (epoch 0), due to random weight initialization, the generated sequence is $A\ B\ F$, which is neither correct nor complete. However, after the first learning epoch, we can see that the connection weight between neurons A and B was increased (note thicker line between A and B as compared with epoch 0), whereas the connection weight between neurons B and F was decreased, which then led to a different sequence $A\ B\ D\ F$. After epoch 3, the connection weight between neurons E and F was increased, since in previously generated sequence $A\ E\ F$ (epoch 2), the transition from E to F was correct. Finally, after learning epoch 7, the network generates the correct sequence.

Similarly, in case of learning of the partial sequence $A\ C\ D\ F$ [see Fig. 9(b)], learning converges to the correct sequence already after epoch 5, since the sequence is shorter.

V. Conclusion and Outlook

Finding the optimal path in a connected graph is a classical problem, and, as discussed in Section II-A, many different algorithms exist to address this. Here, we proposed a neural implementation of the BF algorithm. The main reason, as we would think, for translating BF into a network algorithm is the now-arising possibility to directly use network learning algorithm on a path-finding problem. To do this with BF or other cost-based algorithm, you would have to switch back and forth between a cost- and a weight-based representation of the problem. This is not needed for NN-BF. Given that BF and NN-BF have the same algorithmic complexity, computational speed of the path-finding step is similar, too, where learning epochs can be built into NN-BF as needed. The examples shown above rely on a local learning rule that alters the weights only between directly connected nodes. Globally acting rules could, however, also be used to address different problems, too, and—as far as we see—there are no restrictions with respect to possible network learning mechanisms, because NN-BF is a straightforward architecture for activity propagation without any special limitations.

Hence, network algorithm with NN-BF allows for new and different applications where learned path augmentation is directly coupled with path finding in a natural way.

So far, we only investigated the performance of our method on specific environments and tasks. Some of these task addressed the problem of unseen environments using our learning methods. In our future work, we will investigate the capability of our approach to generalize across different environments and/or different tasks, and we can also extend the experiments into a deeper investigation of complex unknown environments to even more strongly emphasize the power of being able to combine path finding directly with learning.

References

[1] R. Bellman, "On a routing problem," *Quart. Appl. Math.*, vol. 16, no. 1, pp. 87–90, 1958.

[2] L. R. Ford, "Network flow theory," Rand Corp., Santa Monica, CA, USA, Tech. Rep. P-923, 1956.

[3] J. S. Baras and G. Theodorakopoulos, "Path problems in networks," *Synth. Lectures Commun. Netw.*, vol. 3, no. 1, pp. 1–77, Jan. 2010.

[4] D. O. Hebb, *The Organization of Behavior*. New York, NY, USA: Wiley, 2005.

[5] L. Kusmierz, T. Isomura, and T. Toyoizumi, "Learning with three factors: Modulating Hebbian plasticity with errors," *Current Opinion Neurobiol.*, vol. 46, pp. 170–177, Oct. 2017.

[6] B. Porr and F. Wörgötter, "Learning with 'relevance': Using a third factor to stabilize Hebbian learning," *Neural Comput.*, vol. 19, no. 10, pp. 2694–2719, 2007.

[7] E. F. Moore, "The shortest path through a maze," in *Proc. Int. Symp. Theory Switching*, 1959, pp. 285–292.

[8] D. Merrill, M. Garland, and A. Grimshaw, "Scalable GPU graph traversal," *ACM SIGPLAN Notices*, vol. 47, no. 8, pp. 117–128, Sep. 2012.

[9] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Math.*, vol. 1, no. 1, pp. 269–271, Dec. 1959.

[10] P. Hart, N. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Trans. Syst. Sci. Cybern.*, vol. SSC-4, no. 2, pp. 100–107, Jul. 1968.

[11] R. E. Korf, "Depth-first iterative-deepening: An optimal admissible tree search," *Artif. Intell.*, vol. 27, no. 1, pp. 97–109, 1985.

[12] S. Koenig, M. Likhachev, and D. Furcy, "Lifelong planning A*," *Artif. Intell.*, vol. 155, nos. 1–2, pp. 93–146, 2004.

[13] X. Sun, S. Koenig, and W. Yeoh, "Generalized adaptive A*," in *Proc. 7th Int. J. Conf. Auto. Agents Multiagent Syst.*, 2008, pp. 469–476.

[14] D. D. Harabor and A. Grastien, "Online graph pruning for path finding on grid maps," in *Proc. AAAI*, 2011, pp. 1114–1119.

[15] R. W. Floyd, "Algorithm 97: Shortest path," *Commun. ACM*, vol. 5, no. 6, p. 345, Jun. 1962.

[16] S. Warshall, "A theorem on Boolean matrices," *J. ACM*, vol. 9, no. 1, pp. 11–12, Jan. 1962.

[17] D. B. Johnson, "Efficient algorithms for shortest paths in sparse networks," *J. ACM*, vol. 24, no. 1, pp. 1–13, Jan. 1977.

[18] M. K. M. Ali and F. Kamoun, "Neural networks for shortest path computation and routing in computer networks," *IEEE Trans. Neural Netw.*, vol. 4, no. 6, pp. 941–954, Nov. 1993.

[19] D.-C. Park and S.-E. Choi, "A neural network based multi-destination routing algorithm for communication network," in *Proc. IEEE Int. Joint Conf. Neural Netw.*, vol. 2, May 1998, pp. 1673–1678.

[20] F. Araujo, B. Ribeiro, and L. Rodrigues, "A neural network for shortest path computation," *IEEE Trans. Neural Netw.*, vol. 12, no. 5, pp. 1067–1073, Sep. 2001.

[21] J. Häkkinen, M. Lagerholm, C. Peterson, and B. Söderberg, "A Potts neuron approach to communication routing," *Neural Comput.*, vol. 10, no. 6, pp. 1587–1599, Aug. 1998.

[22] R. Glasius, A. Komoda, and S. C. A. M. Gielen, "Neural network dynamics for path planning and obstacle avoidance," *Neural Netw.*, vol. 8, no. 1, pp. 125–133, Jan. 1995.

[23] R. Glasius, A. Komoda, and S. C. A. M. Gielen, "A biologically inspired neural net for trajectory formation and obstacle avoidance," *Biol. Cybern.*, vol. 74, no. 6, pp. 511–520, Jun. 1996.

[24] N. Bin, C. Xiong, Z. Liming, and X. Wendong, "Recurrent neural network for robot path planning," in *Proc. Int. Conf. Parallel Distrib. Comput., Appl. Technol.*, 2004, pp. 188–191.

[25] S. X. Yang and M. Meng, "Neural network approaches to dynamic collision-free trajectory generation," *IEEE Trans. Syst., Man Cybern., B*, vol. 31, no. 3, pp. 302–318, Jun. 2001.

[26] H. Qu, S. X. Yang, A. R. Willms, and Z. Yi, "Real-time robot path planning based on a modified pulse-coupled neural network model," *IEEE Trans. Neural Netw.*, vol. 20, no. 11, pp. 1724–1739, Nov. 2009.

[27] J. Ni, L. Wu, P. Shi, and S. X. Yang, "A dynamic bioinspired neural network based real-time path planning method for autonomous underwater vehicles," *Comput. Intell. Neurosci.*, vol. 2017, pp. 1–16, Feb. 2017.

[28] T. Kulvicius, S. Herzog, M. Tamosiunaite, and F. Wörgötter, "Finding optimal paths using networks without learning—Unifying classical approaches," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 33, no. 12, pp. 7877–7887, Dec. 2022.

[29] A. H. Qureshi, M. J. Bency, and M. C. Yip, "Motion planning networks," 2018, *arXiv:1806.05767*.

[30] N. Pérez-Higueras, F. Caballero, and L. Merino, "Learning human-aware path planning with fully convolutional networks," in *Proc. IEEE Int. Conf. Robot. Autom. (ICRA)*, May 2018, pp. 5897–5902.

[31] Y. Ariki and T. Narihira, "Fully convolutional search heuristic learning for rapid path planners," 2019, *arXiv:1908.03343*.

[32] T. Kulvicius, S. Herzog, T. Luddecke, M. Tamosiunaite, and F. Wörgötter, "One-shot multi-path planning using fully convolutional networks in a comparison to other algorithms," *Frontiers Neurorobotics*, vol. 14, p. 115, Jan. 2021.

[33] M. J. Bency, A. H. Qureshi, and M. C. Yip, "Neural path planning: Fixed time, near-optimal path generation via Oracle imitation," 2019, *arXiv:1904.11102*.

[34] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: MIT Press, 2018.

[35] L. Tai, G. Paolo, and M. Liu, "Virtual-to-real deep reinforcement learning: Continuous control of mobile robots for mapless navigation," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst. (IROS)*, Sep. 2017, pp. 31–36.

[36] A. I. Panov, K. S. Yakovlev, and R. Suvorov, "Grid path planning with deep reinforcement learning: Preliminary results," *Proc. Comput. Sci.*, vol. 123, pp. 347–353, Jan. 2018.

[37] A. Banino et al., "Vector-based navigation using grid-like representations in artificial agents," *Nature*, vol. 557, no. 7705, pp. 429–433, May 2018.

[38] M. Schlichtkrull, T. N. Kipf, P. Bloem, R. Van Den Berg, I. Titov, and M. Welling, "Modeling relational data with graph convolutional networks," in *Proc. Eur. Semantic Web Conf.*, 2018, pp. 593–607.

[39] M. Zhang and Y. Chen, "Link prediction based on graph neural networks," in *Proc. NIPS*, vol. 31, 2018, pp. 5165–5175.

[40] P. Velickovic, R. Ying, M. Padovano, R. Hadsell, and C. Blundell, "Neural execution of graph algorithms," in *Proc. Int. Conf. Learn. Represent.*, 2020, pp. 1–14.

[41] K. Teru, E. Denis, and W. Hamilton, "Inductive relation prediction by subgraph reasoning," in *Proc. Int. Conf. Mach. Learn.*, 2020, pp. 9448–9457.

[42] Z. Zhu, Z. Zhang, L.-P. Xhonneux, and J. Tang, "Neural bellman-ford networks: A general graph neural network framework for link prediction," in *Proc. Neural Inf. Process. Syst.*, 2021, pp. 1–15.

[43] Y. Yang et al., "Graph neural networks inspired by classical iterative algorithms," in *Proc. Int. Conf. Mach. Learn.*, 2021, pp. 1–11.

[44] S. Yan, Y. Xiong, and D. Lin, "Spatial temporal graph convolutional networks for skeleton-based action recognition," in *Proc. AAAI Conf. Artif. Intell.*, 2018, vol. 32, no. 1, pp. 1–9.

[45] M. Li, P.-Y. Huang, X. Chang, J. Hu, Y. Yang, and A. Hauptmann, "Video pivoting unsupervised multi-modal machine translation," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 45, no. 3, pp. 3918–3932, Mar. 2023.

[46] L. Zhang et al., "TN-ZSTAD: Transferable network for zero-shot temporal activity detection," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 45, no. 3, pp. 3848–3861, Mar. 2023.

[47] W. Cong, W. Wang, and W.-C. Lee, "Scene graph generation via conditional random fields," 2018, *arXiv:1811.08075*.

[48] A. Bordes, N. Usunier, A. Garcia-Duran, J. Weston, and O. Yakhnenko, "Translating embeddings for modeling multi-relational data," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 26, 2013, pp. 1–9.

[49] S. Woo, D. Kim, D. Cho, and I. S. Kweon, "LinkNet: Relational embedding for scene graph," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 31, 2018, pp. 1–11.

[50] D. Xu, Y. Zhu, C. B. Choy, and L. Fei-Fei, "Scene graph generation by iterative message passing," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jul. 2017, pp. 3097–3106.

[51] R. Zellers, M. Yatskar, S. Thomson, and Y. Choi, "Neural motifs: Scene graph parsing with global context," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, Jun. 2018, pp. 5831–5840.

[52] W. Gao, Y. Zhu, W. Zhang, K. Zhang, and H. Gao, "A hierarchical recurrent approach to predict scene graphs from a visual-attention-oriented perspective," *Comput. Intell.*, vol. 35, no. 3, pp. 496–516, Aug. 2019.

[53] Y. Li, W. Ouyang, B. Zhou, J. Shi, C. Zhang, and X. Wang, "Factorizable Net: An efficient subgraph-based framework for scene graph generation," in *Proc. Eur. Conf. Comput. Vis. (ECCV)*, 2018, pp. 335–351.

[54] J. Yang, J. Lu, S. Lee, D. Batra, and D. Parikh, "Graph R-CNN for scene graph generation," in *Proc. Eur. Conf. Comput. Vis. (ECCV)*, 2018, pp. 670–685.

[55] X. Chang, P. Ren, P. Xu, Z. Li, X. Chen, and A. Hauptmann, "A comprehensive survey of scene graphs: Generation and application," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 45, no. 1, pp. 1–26, Jan. 2023.

[56] M. Zhang et al., "One-shot neural architecture search: Maximising diversity to overcome catastrophic forgetting," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 43, no. 9, pp. 2921–2935, Sep. 2021.

[57] C. Li, G. Wang, B. Wang, X. Liang, Z. Li, and X. Chang, "DS-Net++: Dynamic weight slicing for efficient inference in CNNs and vision transformers," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 45, no. 4, pp. 4430–4446, Apr. 2023.

[58] X. Gong, S. Chang, Y. Jiang, and Z. Wang, "AutoGAN: Neural architecture search for generative adversarial networks," in *Proc. IEEE/CVF Int. Conf. Comput. Vis. (ICCV)*, Oct. 2019, pp. 3223–3233.

[59] H. Wang and J. Huan, "AGAN: Towards automated design of generative adversarial networks," 2019, *arXiv:1906.11080*.

[60] C. Yan et al., "ZeroNAS: Differentiable generative adversarial networks search for zero-shot learning," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 44, no. 12, pp. 9733–9740, Dec. 2022.

[61] Y. Xia and J. Wang, "A discrete-time recurrent neural network for shortest-path routing," *IEEE Trans. Autom. Control*, vol. 45, no. 11, pp. 2129–2134, 2000.

[62] Y. Zhang and S. Li, "Distributed biased min-consensus with applications to shortest path planning," *IEEE Trans. Autom. Control*, vol. 62, no. 10, pp. 5429–5436, Oct. 2017.

[63] S. Koenig and R. G. Simmons, *Complexity Analysis of Real-Time Reinforcement Learning Applied to Finding Shortest Paths in Deterministic Domains*. Pittsburgh, PA, USA: Carnegie-Mellon Univ., 1993.

[64] B. Porr and F. Wörgötter, "Isotropic sequence order learning," *Neural Comput.*, vol. 15, no. 4, pp. 831–864, Apr. 2003.

Tomas Kulvicius received the Ph.D. degree in computer science from the University of Göttingen, Göttingen, Germany, in 2010. His Ph.D. thesis was on the development of receptive fields in closed-loop learning systems.

From 2010 to 2015, he was a Researcher at the University of Göttingen, where he worked on trajectory generation and motion control for robotic manipulators. From 2015 to 2017, he was appointed as an Assistant Professor at the Centre for Bio Robotics, University of Southern Denmark, Odense, Denmark. He is currently a Research Assistant at the University of Göttingen. His research interests include modeling of closed-loop behavioral systems, robotics, artificial intelligence, machine learning algorithms, movement generation and trajectory planning, action recognition and prediction, and movement analysis and diagnostics.

Minija Tamosiunaite received the Ph.D. degree in informatics from Vytautas Magnus University, Kaunas, Lithuania, in 1997.

She is currently working as a Senior Researcher at the Bernstein Center for Computational Neuroscience, Third Institute of Physics, University of Göttingen, Göttingen, Germany. Her research interests include machine learning, biological signal analysis, and application of learning methods in robotics.

Florentin Wörgötter received the Diplom degree in biology with specialization in mathematics from the University of Düsseldorf, Düsseldorf, Germany, in 1985, and the Ph.D. degree from the University of Essen, Essen, Germany, in 1988, with a focus on the visual cortex.

From 1988 to 1990, he was engaged in computational studies with the California Institute of Technology, Pasadena, CA, USA. From 1990 to 2000, he was a Researcher at the University of Bochum, Bochum, Germany, where he investigated the experimental and computational neuroscience of the visual system. From 2000 to 2005, he was a Professor of computational neuroscience with the Psychology Department, University of Stirling, Stirling, U.K., where his interests strongly turned toward learning in neurons. Since July 2005, he has been the Head of computational neuroscience at the Bernstein Center for Computational Neuroscience, Third Institute of Physics, University of Göttingen, Göttingen, Germany. His current research interests include information processing in closed-loop perception action systems, sensory processing, motor control, and learning/plasticity, which are tested in different robotic implementations.