4Ward: A relayering strategy for efficient training of arbitrarily complex directed acyclic graphs

Tommaso Boccato [a],[*], Matteo Ferrante [a], Andrea Duggento [a], Nicola Toschi [a],[b]

[a] *Department of Biomedicine and Prevention, University of Rome Tor Vergata, Rome, Italy*
[b] *A.A. Martinos Center for Biomedical Imaging and Harvard Medical School, Boston, USA*

ARTICLE INFO

ABSTRACT

Thanks to their ease of implementation, multilayer perceptrons (MLPs) have become ubiquitous in deep learning applications. The graph underlying an MLP is indeed multipartite, i.e. each layer of neurons only connects to neurons belonging to the adjacent layer. In contrast, in vivo brain connectomes at the level of individual synapses suggest that biological neuronal networks are characterized by scale-free degree distributions or exponentially truncated power law strength distributions, hinting at potentially novel avenues for the exploitation of evolution-derived neuronal networks. In this paper, we present "4Ward", a method and Python library capable of generating flexible and efficient neural networks (NNs) from arbitrarily complex directed acyclic graphs. 4Ward is inspired by layering algorithms drawn from the graph drawing discipline to implement efficient forward passes, and provides significant time gains in computational experiments with various Erdős-Rényi graphs. 4Ward not only overcomes the sequential nature of the *learning matrix* method, by parallelizing the computation of activations, but also addresses the scalability issues encountered in the current state-of-the-art and provides the designer with freedom to customize weight initialization and activation functions. Our algorithm can be of aid for any investigator seeking to exploit complex topologies in a NN design framework at the microscale.

1. Introduction

Thanks to their ease of implementation, multilayer perceptrons (MLPs) have become ubiquitous in deep learning (DL) applications. Virtually all state-of-the-art neural architectures (e.g., convolutional neural networks [4,5], variational autoencoders [6], transformers [7]) include MLPs among their functional blocks. Amongst other advantages, a forward pass in an MLP reduces to a series of matrix multiplications, guaranteeing computational efficiency.

The graph underlying an MLP, however, is multipartite. In other words, each layer of neurons only connects to neurons belonging to the adjacent layer. In this context, studies conducted on in vivo brain connectomes, at the level of individual synapses, suggest that biological neuronal networks are characterized by scale-free degree distributions [8] or exponentially truncated power law strength distributions [9]. These architectural differences between biological and artificial neural networks (NNs) suggest potentially novel avenues for the exploitation of evolution-derived neuronal networks as artificial NNs by relaxing the topological constraints of MLPs.

In the current DL landscape, there is no satisfactory tool that allows to efficiently experiment, on the microscale, with analog (i.e., nonspiking) NNs based on arbitrarily complex topologies. Current strategies to convert complex networks into NNs can be divided into three main categories: topology-dependent, layer-specific and generalist.

Topology-dependent approaches include strategies in which NNs are generated by randomly rewiring a fraction of the edges of a MLP [10–14]. Typically, these models can be trained with standard backpropagation; however, the range of usable topologies is constrained by the random interpolation procedure used on the original MLP. Additionally, the strategy presented in [15] is compatible with Barabási–Albert graphs [16] only, and the difficulties related to the extension of backpropagation toward nonlayered graphs is overcome through the use of evolutionary algorithms for training the networks.

Layer-specific approaches include [17], where the focus is placed on the interlayer connections of MLPs. The procedure employed (sparse evolutionary training - SET), is capable of inducing sparsity in the MLPs' bicliques by pruning the weakest connections. The produced topologies, however, are a result of the SET procedure and, hence, not defined *a priori*. Also the framework presented in [18] allows the definition of a trainable topology through a tool called relational graph. Still, the expressiveness of relational graphs is limited to the bipartite graphs that underlie an MLP.
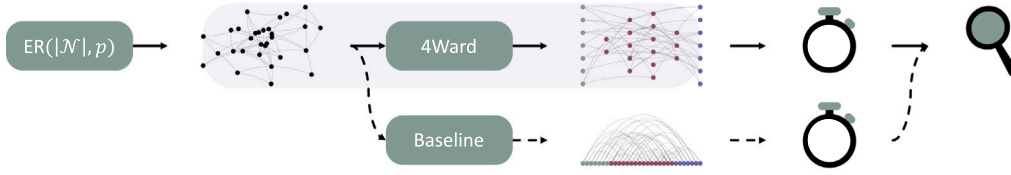
Fig. 1. Time complexity evaluation of the 4Ward algorithm. The process starts with the generation of Erdős-Rényi (ER) [1] graphs from a predetermined parameter grid. Each undirected network is then converted into a DAG following a random heuristic. As highlighted in purple, 4Ward is able to process the DAGs produced and return deployable NNs. Finally, the execution times of the forward passes of the networks are measured and compared with those obtained from the baselines [2,3].

Finally, the generalist category is mostly represented by the *learning matrix method* [2,19]. The learning matrix is an augmented adjacency matrix on which forward and backward operations are performed, i.e. it includes additional rows and columns that contain the state information required to compute the forward/backward passes. The framework accepts directed acyclic graphs (DAGs) represented as upper triangular matrices; however, neuron activations are computed sequentially w.r.t. the network nodes. As a consequence, the method does not scale with the network size. The recent work by Stier et al. `deepstruct` [3,20], faces a similar issue. In the model's forward pass, neural activations are decomposed using a series of matrix multiplications, which are defined based on a pre-calculated partitioning of the NN nodes. This decomposition leads to a reduced number of matrix multiplications when the graph has a density close to zero, demonstrating sublinear dependence on the number of neurons; otherwise, the required number of multiplications can exhibit a quadratic dependence.

Outside the three discussed categories, several NNs architectures have computational graphs defined on a macroscopic scale [21–23]. In these works, nodes typically represent differentiable functional blocks (e.g., linear layers, convolutions), while the edges are tensors. Neural architecture search, in both its "classical" [24] and differentiable [25, 26] variants, also falls within this body of research. Although these works have achieved remarkable results in computer vision and natural language processing, they do not address the requirements for which microscopic-scale models were designed: transforming and combining features without a specific spatial relationship.

In order to overcome the mentioned microscale-related difficulties, in this paper, we present "4Ward", a method and Python library[1] capable of generating flexible and efficient NNs from arbitrary DAGs. The method minimizes the number of matrix multiplications required to perform a forward pass, and is inspired by layering algorithms drawn from the *graph drawing* discipline to implement efficient forward passes. The 4Ward functional blocks are deployable as modules compatible with PyTorch [27]. An overview of the experimental protocol developed to evaluate the time complexity of the 4Ward algorithm is reported in Fig. 1.

2. Methods

The computational graph is a powerful formalism used to represent NNs. Nodes are typically associated with function-variable pairs, (f_v, a_v), $v \in \mathcal{N} \subset \mathbb{N}$, while each edge is associated with a weight w_{uv}, $(u, v) \in \mathcal{E} \subseteq \mathcal{N} \times \mathcal{N}$. When the values of the variables associated with the predecessors of node v are fixed, its activation can be computed through:

$$a_v = f_v \left(\sum_{u:(u,v)\in\mathcal{E}} w_{uv} a_u \right) \tag{1}$$

Graph sources (i.e., $\{v : (u, v) \in \mathcal{E}\} = \emptyset$) are called input nodes since their role is to provide input values for subsequent function evaluations. Similarly, sinks (i.e., $\{v : (v, u) \in \mathcal{E}\} = \emptyset$) are called output nodes

since their activations correspond to the result of previous function evaluations.

Directed graphs with no directed cycles are the backbone of feedforward NNs. DAGs are of particular interest as they admit at least one topological ordering, that is a linear ordering such that $\forall (u, v) \in \mathcal{E}$ $u < v$; this implies the possibility of computing the output values of the graph, given a set of values for the input nodes, evaluating functions in the same order expressed by the topological one. However, this straightforward implementation of the forward pass (i.e., the process that leads to the evaluation of the output nodes) is inefficient due to its sequential nature. All nodes whose predecessors' values are already available can, in principle, be evaluated in parallel.

MLPs, whose underlying graph is a chain of bicliques, are particularly well suited for an efficient implementation of the forward pass. All activations of a layer, a^l, are computed simultaneously from the activations of the previous layer, a^{l-1}, through matrix multiplication[2]:

$$a^l = \sigma(W^l a^{l-1}) \tag{2}$$

where $a^l \in \mathbb{R}^{|L_l|}$, L_l is the set of nodes in layer l, $W^l \in \mathbb{R}^{|L_l| \times |L_{l-1}|}$ represents the weights and σ denotes the activation function. While the computational graph of an MLP is characterized by a specific topology for which a node partition is immediately identifiable, it is worth noting that the same does not hold for arbitrary DAGs. The latter require relayering algorithms to implement efficient forward passes.

2.1. 4Ward

Let the layering $\mathcal{L} = \{L_0, \dots, L_{H-1}\}$ be a partition of $\mathcal{N}$ such that $\forall (u, v) \in \mathcal{E}$ $u \in L_i, v \in L_j \implies i < j$. As in the MLP case, all functions associated with the lth layer can be evaluated in parallel since the required input values are already available at computation time; this is guaranteed by the definition of layering. Consequently, Eq. (2) can be generalized as follows:

$$a^l = \sigma(W^l x^l) \tag{3}$$

where $x^l = [\dots, a_v, \dots]^T$, $v \in P_l$ and $P_l = \{u : \forall v \in L_l (u, v) \in \mathcal{E}\}$. In other words, x^l contains the activations of the predecessors (i.e., P_l) of nodes belonging to layer l. Importantly, in $W^l \in \mathbb{R}^{|L_l| \times |P_l|}$, all entries corresponding to $(\mathcal{N} \times \mathcal{N}) \setminus \mathcal{E}$ must be 0 to preserve the given DAG topology; this constraint is enforced in the implementation phase through the *mask trick*. A major difference from Eq. (2) lies in the ability of nodes belonging to layer l to draw information from the subnetwork $\cup_{i=0}^{l-1} L_i$.

Under the assumption that the overall time complexity of a forward pass is dominated by H, optimizing the graph partitioning becomes crucial.[3] In this context, the *longest-path layering algorithm* [28,29] can be employed (Appendix C). A layering algorithm can be seen as a

---

[1] https://github.com/BoCtrl-C/forward.

[2] The overall time complexity of the forward pass depends on the matrix multiplication algorithm.

[3] It is also important to take the impact of memory accesses and data transfers between devices into account.
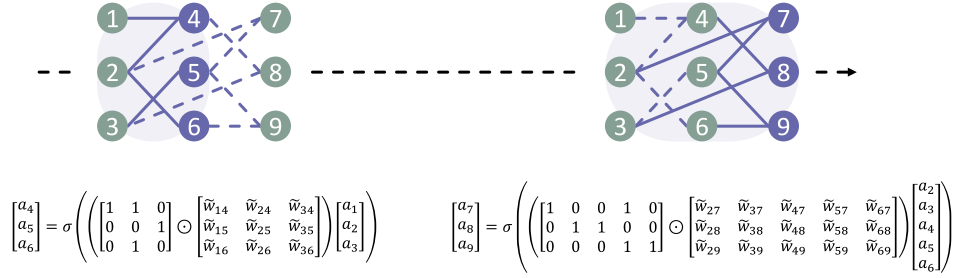
$$\begin{bmatrix} a_4 \\ a_5 \\ a_6 \end{bmatrix} = \sigma\left(\left(\begin{bmatrix} 1 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \odot \begin{bmatrix} \tilde{w}_{14} & \tilde{w}_{24} & \tilde{w}_{34} \\ \tilde{w}_{15} & \tilde{w}_{25} & \tilde{w}_{35} \\ \tilde{w}_{16} & \tilde{w}_{26} & \tilde{w}_{36} \end{bmatrix}\right)\begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix}\right) \qquad \begin{bmatrix} a_7 \\ a_8 \\ a_9 \end{bmatrix} = \sigma\left(\left(\begin{bmatrix} 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \end{bmatrix} \odot \begin{bmatrix} \tilde{w}_{27} & \tilde{w}_{37} & \tilde{w}_{47} & \tilde{w}_{57} & \tilde{w}_{67} \\ \tilde{w}_{28} & \tilde{w}_{38} & \tilde{w}_{48} & \tilde{w}_{58} & \tilde{w}_{68} \\ \tilde{w}_{29} & \tilde{w}_{39} & \tilde{w}_{49} & \tilde{w}_{59} & \tilde{w}_{69} \end{bmatrix}\right)\begin{bmatrix} a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \end{bmatrix}\right)$$

Fig. 2. Forward pass computation for an example NN built through 4Ward (from left to right). **Left**: computation of the activations of layer 1. **Right**: computation of the activations of layer 2. The subnetworks involved are highlighted through the purple overlay. Computations follow $a^l = \sigma((M^l \odot \tilde{W}^l)x^l)$, in accordance with (3).

mapping between graphs and layerings: $\mathcal{G} \mapsto \mathcal{L}$; with $\mathcal{G} = (\mathcal{N}, \mathcal{E})$. In general, DAGs admit more than one layering; the longest-path algorithm, however, returns one of those of minimum height (i.e., H). While it is possible for the minimum height layering to be non-unique (refer to Appendix B), in accordance with the above assumption, different layerings of same height lead to the same time complexity. It is worth noting that topological orderings can be seen as a particular case of layering in which $H = \mathcal{O}(|\mathcal{N}|)$. Instead, in minimum height layerings, H is topology-dependent.

2.2. Implementation

We chose PyTorch [27] as our reference machine learning framework for its ease of use in research prototyping, its automatic differentiation engine, Autograd, and its popularity among scientists. Implementing the forward pass in a PyTorch `Module` is immediate, with the exception of preventing the gradient from updating the zero entries of all W^l. PyTorch tensors, indeed, can be marked as learnable or nonlearnable; however, the topology of the DAG which is object of relayering needs to be preserved.

To circumnavigate this issue, we propose a strategy which we term the mask trick. It consists of masking the fully-learnable tensor $\tilde{W}^l$; this is accomplished by pointwise multiplication $W^l = M^l \odot \tilde{W}^l$. Specifically, the lth mask matrix is defined as:

$$m_{uv} = \begin{cases} 1 & \text{if } (u, v) \in \mathcal{E} \\ 0 & \text{otherwise} \end{cases} \qquad (4)$$

where m_{uv} denotes the entry of M^l linked to edge (u, v). The partial derivative of a loss function l w.r.t. $\tilde{w}_{uv}$ immediately follows:

$$\begin{aligned} \frac{\partial l(h_{\boldsymbol{w}}(\boldsymbol{a}^0), y)}{\partial \tilde{w}_{uv}} &= \frac{\partial l}{\partial h_{\boldsymbol{w}}} \frac{\partial h_{\boldsymbol{w}}}{\partial \tilde{w}_{uv}} \\ &= \frac{\partial l}{\partial h_{\boldsymbol{w}}} \frac{\partial h_{\boldsymbol{w}}}{\partial w_{uv}} \frac{\partial w_{uv}}{\partial \tilde{w}_{uv}} \\ &= \frac{\partial l}{\partial h_{\boldsymbol{w}}} \frac{\partial h_{\boldsymbol{w}}}{\partial w_{uv}} m_{uv} \end{aligned} \qquad (5)$$

where $h_{\boldsymbol{w}}$ denotes the NN parameterized by $\{w_{uv}, (u, v) \in \mathcal{E}\}$, $(\boldsymbol{a}^0, y)$ is the generic labeled training sample and $w_{uv} = m_{uv}\tilde{w}_{uv}$. As a consequence, $m_{uv} = 0 \implies \frac{\partial l}{\partial \tilde{w}_{uv}} = 0$.

The pseudocode in Algorithms 1 and 2 provides a detailed description of the 4Ward implementation. Fig. 2, instead, shows an example forward pass computation. In Algorithm 1, $longest\_path()$ wraps the layering algorithm, $push\_sources()$ moves the source nodes within L_0[4] and $init()$ initializes the weights. The $init()$ function accepts M^l as an argument to allow the use of sophisticated initialization methods (e.g., Kaiming Uniform [30]) that take into account the in-degree of nodes. Biases can be easily included in the framework by adding a

column to $\tilde{W}^l$, e.g. replacing M^l with $[M^l, \mathbf{1}]$ and x^l with $[(x^l)^T, 1]^T$. Extending x^l to matrix X^l allows 4Ward to work with batches.

```
𝓛 ← longest_path(𝓖);
𝓛 ← push_sources(𝓖, 𝓛);
for l ← 1 to H − 1 do
    Pˡ ← ∅;
    for v ∈ Lₗ do
        Pˡ ∪ {u : (u, v) ∈ 𝓔}
    end
    Mˡ ← 0;
    W̃ˡ ← 0;
    for v ∈ Lₗ do
        for u ∈ {u : (u, v) ∈ 𝓔} do
            m_uv ← 1
        end
    end
    W̃ˡ ← init(W̃ˡ, Mˡ)
end
```
Algorithm 1: 4Ward instantiation.

```
for l ← 1 to H − 1 do
    aˡ ← σ((Mˡ ⊙ W̃ˡ)xˡ);        /* the mask trick */
end
```
Algorithm 2: 4Ward forward pass.

2.3. Baseline

In addition to 4Ward, we developed a baseline for the experimental protocol described in Section 2.5. The baseline is a PyTorch-compatible extension of the learning matrix method [2,19] which implements the sequential forward pass discussed above. The baseline pseudocode is shown in Algorithms 3 and 4. Algorithm 3 reports the model instantiation while Algorithm 4 describes how the forward pass is computed. In both algorithms, $\boldsymbol{w}_v$ denotes the weight vector of node v, and $\boldsymbol{a}^v$ stores the activations of the v's predecessors. Both vectors share the same dimension: $|\{u : (u, v) \in \mathcal{E}\}|$.

```
sources ← {v : {(u, v) ∈ 𝓔} = ∅};
for v ∈ 𝓝 \ sources do
    w_v ← 0;
    init(w_v);
end
```
Algorithm 3: Baseline instantiation.

---

[4] The considered layering algorithm, by default, places each node in the highest possible layer.

```
for v ∈ 𝒩 \ sources do
    a_v ← σ(w_v^T a^v)
end
```
Algorithm 4: Baseline forward pass.

2.4. Deepstruct

We also incorporated the `deepstruct` methodology, mentioned in Section 1, into our experiments. In the high sparsity regime, the efficiency of this methodology, in terms of time complexity, depends on how the computation of activations is decomposed within the forward pass. Similar to 4Ward, `deepstruct` initializes the input DAG by partitioning it into a layering of minimum height. However, layers are exploited in a different manner during the forward pass. To compute the activations of a specific layer, a^l, `deepstruct` follows the equation:

$$a^l = \sigma\left(\sum_{j=0}^{l-1} W^{j\mapsto l} a^j\right) \tag{6}$$

According to (6), each layer preceding layer l contributes additively to the computation of a^l with its own activations (i.e., a^j) through the matrix multiplication $W^{j\mapsto l} a^j$, where $W^{j\mapsto l}$ represents the synaptic weights connecting neurons in layer j to neurons in layer l. In contrast to (3), where activations are computed all at once, the summation in (6) involves l matrix multiplications. It is important to note that, in the worst case, l corresponds to the number of nodes preceding L_l (i.e., $\left|\cup_{j=0}^{l-1} L_j\right|$, where each L_j contains only one node).

We provide a detailed description of the `deepstruct` implementation of (6) in Algorithm 5. The first for loop iterates over the layers, while the second loop performs the summation in (6). Partial activations are cumulatively stored in the a^l variable. As in Algorithm 2, $\bar{W}^{j\mapsto l}$, $M^{j\mapsto l}$, and σ represent the raw weights between layers j and l, their mask, and the activation function, respectively. If a mask is completely empty, the associated multiplication operation is skipped. When $H \sim |\mathcal{N}|$, the time complexity of the algorithm, in terms of matrix multiplications, becomes quadratic in the number of nodes.

```
for l ← 1 to H − 1 do
    a^l ← 0;
    for j ← 0 to l − 1 do
        if M^{j↦l} ≠ 0 then
            a^l ← a^l + (M^{j↦l} ⊙ W̄^{j↦l})a^j;
        end
    end
    a^l ← σ(a^l);
end
```
Algorithm 5: `deepstruct` forward pass.

2.5. Experiments

Evaluating the time complexity of Algorithm 2 analytically, w.r.t. a traditional set of graph attributes (e.g., $|\mathcal{N}|$ and $|\mathcal{E}|$), is not viable, especially in view of the dependency on low-level operations (e.g., matrix multiplications) and, most importantly, on the topology-dependency of H and $\bar{W}^l$. Instead, we rely on an empirical evaluation of the execution time of our proposed forward pass. Specifically, in a set of experiments, we monitored how long it took neural networks created in accordance with the same generative model to process a predetermined number of batches. Notably, the actual content of a batch is irrelevant for the purposes of time evaluation. Hence, all input tensors were filled with 1s.

We set the Erdős-Rényi (ER) model [1] as our reference generator. Given a graph of size $|\mathcal{N}|$, the ER algorithm randomly places the network edges by sampling from $|\mathcal{E}| = \frac{|\mathcal{N}|(|\mathcal{N}|-1)}{2}$ i.i.d. Bernoulli distributions, $\mathcal{B}(1, p)$. Subsequently, the largest connected component of each network was extracted and converted into a DAG following the procedure in [21].

For completeness, we repeated the exact same temporal assessment for all the $(|\mathcal{N}|, p)$ pairs belonging to a 2D grid defined over the ER's size-probability parameter space, averaging, for each parameter pair, several execution times linked to a fixed number of ER generation seeds: $\mathbb{E}\left[\Delta T^{\bullet}_{|\mathcal{N}|,p}\right]$ where • can take on the values $4W$, SQ or DS depending on the methodology considered - 4Ward, sequential baseline or `deepstruct`. The mean gains were calculated by running the same experimental protocol on $\mathbb{E}\left[\Delta T^{SQ}_{|\mathcal{N}|,p}/\Delta T^{4W}_{|\mathcal{N}|,p}\right]$ and $\mathbb{E}\left[\Delta T^{DS}_{|\mathcal{N}|,p}/\Delta T^{4W}_{|\mathcal{N}|,p}\right]$. All experiments where run on a workstation equipped with an Intel Xeon Gold 6326 CPU, 512 GB of RAM and an NVIDIA RTX A6000 GPU (driver version 510.47, CUDA version 11.6).

3. Results

We implemented the experimental protocol described in Section 2.5 while setting $|\mathcal{N}| \in \{2^6, 2^7, 2^8, 2^9, 2^{10}\}$ and $p \in \{0.2, 0.4, 0.6, 0.8, 1.0\}$. For each $(|\mathcal{N}|, p)$ pair, 30 graphs were generated starting from different random seeds. We tested three batch sizes: 32, 128 and 512; all computations involved a sequence of 100 batches. The experiments were repeated for all the implementations (i.e., baseline, `deepstruct` and 4Ward).

The computed mean execution times (baseline and 4Ward) are reported in Fig. 3. Once the probability p is fixed, function $\mathbb{E}\left[\Delta T_{|\mathcal{N}|,p}\right]$ grows with the network size; this applies to both the baseline and our method. However, the two implementations behave differently as the ER probability varies. The execution time of the baseline's forward pass does not appear to depend on p, while that of 4Ward increases with p. In other words, the gradient of $\mathbb{E}\left[\Delta T^{SQ}_{|\mathcal{N}|,p}\right]$ w.r.t. p and $|\mathcal{N}|$ is almost vertical, the one of $\mathbb{E}\left[\Delta T^{4W}_{|\mathcal{N}|,p}\right]$ points toward the top-right corner of the investigated parameter space. No differences emerged from the data collected for the different batch sizes.

Fig. 4 shows the results obtained in terms of mean gain (Section 2.5). In the top panel, the gradient of the metric points toward the left boundary of the parameter space. The isolines corresponding to the unity gain vertically cut the tested spaces in a low-density region (i.e., $p \sim 0.8$). 4Ward decreases computation time by up to a factor 2.80, with higher gains at higher sparsity levels. In the bottom panel, instead, gradients point toward the top-right corners of the drawn parameter spaces, and therefore present a non-zero horizontal component. In this comparison, mean gains span across a minimum of 2.47 and a maximum of 160.37. Minor differences emerge for the different batch sizes.

4. Discussion

Our results verified the assumption that the overall time complexity of a forward pass is dominated by H. For the baseline, the result can be directly inferred from the mean times reported in Section 3, which depend only on the number of nodes that characterizes the graphs. Our sequential implementation is indeed a special case of 4Ward in which the layering $\mathcal{L}$ always has one node per hidden layer (i.e., $H \sim |\mathcal{N}|$). The same conclusion can be reached for 4Ward (see Appendix A). The left panel of Fig. A.5 reports function $\mathbb{E}\left[H_{|\mathcal{N}|,p}\right]$, which demonstrates in how many layers, on average, a graph corresponding to a precise $(|\mathcal{N}|, p)$ pair is partitioned by the longest-path layering algorithm. Unsurprisingly, $\mathbb{E}\left[\Delta T^{4W}_{|\mathcal{N}|,p}\right]$ and $\mathbb{E}\left[H_{|\mathcal{N}|,p}\right]$ exhibit the same behavior. The batch size, which was set to values commonly used in DL literature, has little impact on the analyzed execution times, potentially due to the comparatively high computational power of the A6000 unit.
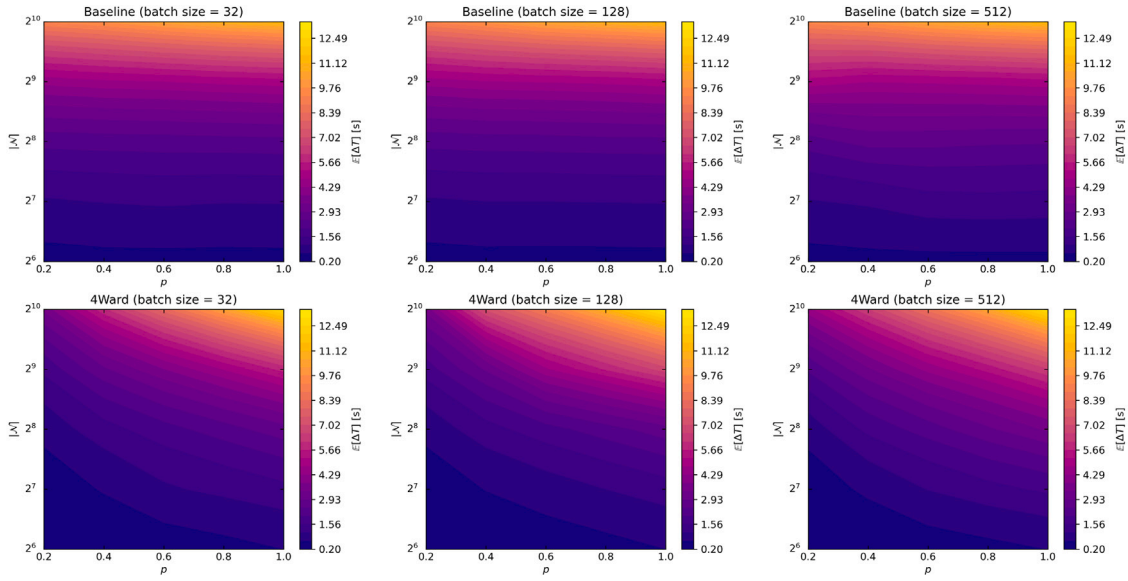
Fig. 3. Mean execution times in seconds. Timings were examined on 100 forward passes. **Top**: baseline. **Bottom:** 4Ward. Each column refers to a specific batch size. Plots span the investigated ER parameter space. Network sizes are represented on the *y*-axis, while probabilities are represented on the *x*-axis. Elapsed times are color-coded.
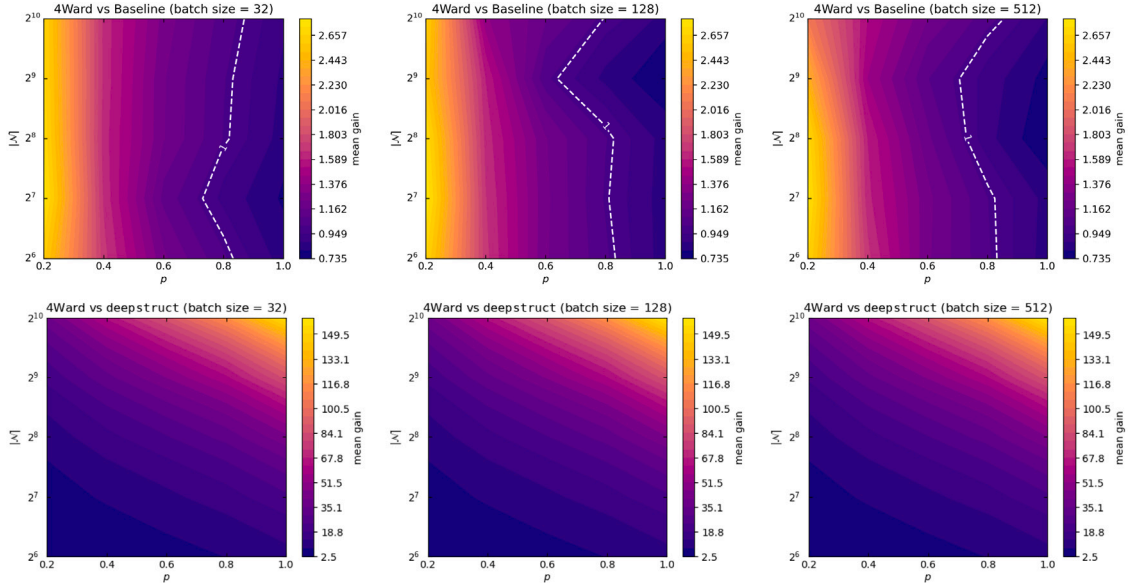
Fig. 4. Mean gains. **Top**: 4Ward vs. baseline. **Bottom**: 4Ward vs. deepstruct. Each column refers to a specific batch size. Plots span the same size-probability parameter space of Fig. 3. Gain values are color-coded. The isolines corresponding to the unity gain are denoted in white.

The most distinctive feature of 4Ward, however, is its efficiency, as shown by the gain analysis of Section 3. According to Fig. 4, the mean gain function computed on the times of 4Ward and the baseline, $\mathbb{E}\left[\Delta T^{SQ}_{|\mathcal{N}|,p}/\Delta T^{4W}_{|\mathcal{N}|,p}\right]$, presents a series of vertical bands characterized by the same gain. The mean gain increases as the ER probability decreases. In other words, it is particularly advantageous to rely on the forward pass defined by our library when dealing with sparse neural networks. An undirected network is defined to be sparse when $|\mathcal{E}| \ll \frac{|\mathcal{N}|(|\mathcal{N}|-1)}{2}$, and $\mathbb{E}[|\mathcal{E}|] = p\frac{|\mathcal{N}|(|\mathcal{N}|-1)}{2}$ for the ER model. In our experiments, the highest percentage gain recorded is 280% ($p = 0.2$), and it can reasonably be assumed that this figure would increase further with increasing sparsity. In the examined density regime, the baseline and 4Ward become equivalent when the DAG is fully-connected, because in this case it only admits one topological ordering in which each node is connected to the entire subnetwork that precedes it. This is experimentally confirmed in Fig. 4 (aside from a minor shift to the left

in unity gain isolines potentially due to low-level implementation differences). The comparison between 4Ward and deepstruct yielded even more impressive results, showing a remarkable highest percentage gain of approximately 16 000%. Moreover, within the tested parameter space, deepstruct consistently exhibited forward pass times that were at least 2.4 times slower than those of 4Ward. Notably, the plots demonstrate diagonal gain "isobands", indicating that mean gains are influenced by both the size of the network, $|\mathcal{N}|$, and the ER probability, p. Specifically, as network size and density increase, the mean gains also increase. We hypothesize that the two models will perform similarly along a curve that lies entirely within the narrow unexplored region of the parameter space defined by $p < 0.2$. At the lowest density regime tested (i.e., $p = 0.2$), the multiplication operations performed by deepstruct already result in a significant slowdown compared to 4Ward. Lastly, it is important to acknowledge that slight differences in the implementation of the two architectures might have introduced a minor bias in the measured timings.

5. Conclusions

In this paper we have presented and released 4Ward, a scalable network generator capable of converting arbitrary DAGs into complex NNs. 4Ward effectively addresses the sequential limitations of the *learning matrix* method and resolves scalability concerns by introducing a novel parallelization method for computing activations. Furthermore, 4Ward provides the designer with freedom to customize weight initialization and activation functions. The modules produced within the library can be trained using all PyTorch optimizers (e.g., Adam [31], stochastic gradient descent with momentum [32]) due to their compatibility with the machine learning framework. 4Ward provides significant computational time gains and can be of aid for any investigator seeking to exploit complex topologies in a NN design framework at the microscale.

Appendix A. Height attenuation

We define two new graph attributes useful to understand the temporal performance of 4Ward. The graph *height*, H, tells us in how many layers a layering of minimum height partitions the graph. The *height attenuation*, instead, is defined as $\frac{|\mathcal{N}|}{H}$.

Appendix B. Non-uniqueness of minimum height layering

As anticipated in Section 2.1, there could be multiple minimum height layerings associated to the same DAG. Since the implementation of 4Ward, described in Algorithms 1 and 2, relies on the specific layering returned by the employed *longest-path* algorithm, it is legitimate to ask what is the impact of different possible partitions on the time complexity of our methodology. In order to address this question, we designed an additional experiment in which we measured the forward pass execution times for different implementations, corresponding to different minimum height layerings, of the same NNs.

Similarly to the experiments described in Section 2.5 and reported in Section 3, we set $|\mathcal{N}| = 64$, a batch size of 512, and for each density regime $p \in \{0.2, 0.4, 0.6, 0.8, 1.0\}$, we generated 30 ER graphs. Then, for each graph, 10 variants of the same NN were initialized; variants were

produced by a modified version of Algorithm 1 in which the stochastic node reassignment procedure of Algorithm 6 is executed after the call to the *push_sources*() function. The algorithm starts from the layering returned by *longest_path*(), that is the one in which hidden nodes are placed as close to sinks as possible. Hence, it attempts to randomly reassign hidden nodes to new layers without altering the underlying computational graph. This is done by sampling the new layer of a node from the set of layers that stand between the lowest layer that is higher than those of the node predecessors and the layer of the node itself. Inside the algorithm, *copy*() clones the set that is passed as an argument while l_u denotes the layer index of node u. It is important to highlight that the procedure only changes the order according to which node activations are computed; indeed, forward passes computed feeding several models generated from the same DAG with the same input batch produce exactly the same output.

---

for $L \in \{L_2, \ldots, L_{H-2}\}$ **do**
    $L' \leftarrow copy(L)$;
    for $u \in L'$ **do**
        preds $\leftarrow \{v : (v, u) \in \mathcal{E}\}$;
        lower $\leftarrow \max\{l_v : v \in \text{preds}\} + 1$;
        upper $\leftarrow l_u$;
        $l \sim \{\text{lower}, \ldots, \text{upper}\}$;
        $L \leftarrow L \setminus \{u\}$;
        $L_l \leftarrow L_l \cup \{u\}$;
        $l_u \leftarrow l$;
    end
end

Algorithm 6: Node reassignment.

The results obtained are shown in the left panel of Fig. B.6, where, for each density, we report mean ($\mathbb{E}[\Delta T]$) and standard deviation of the produced distributions of forward pass times (100 batches per measured time). Each distribution has been computed from graphs characterized by the same number of nodes $|\mathcal{N}|$, edges $|\mathcal{E}|$ and layers H. This can be accomplished by generating multiple graphs from the same pair $(|\mathcal{N}|, p)$ and keeping only those whose height equals the mean one (pre-computed) that emerges from the generation hyper-parameters considered. The standard deviation reported for $p = 1.0$ constitutes our baseline since it is only due to the noise introduced by the hardware and software of the machine on which the experiments have been run; being fully-connected, all the 30×10 NNs generated in this regime present indeed the same computational DAG and $\mathcal{L}$. When $0.4 \leq p \leq 0.8$, standard deviations become, counterintuitively, even smaller. The result suggests that the observed variability is again dominated by machine-related noise. We motivate the higher standard deviation of the baseline conjecturing that each matrix multiplication performed could contribute with its own additive noise component to the measured times. And, as shown by the trend depicted in the top-right panel of Fig. B.6, the number of matrix multiplications performed by a model (i.e., $H - 1$) increases with p. The high sparsity regime ($p = 0.2$), instead, presents the highest standard deviation; on average, about 1/4 of the hidden nodes are in this case reassigned to a new layer (Fig. B.6, bottom-right panel) by Algorithm 6. Overall, the standard deviations are small compared to the values of the computed mean execution times, and, as expected, times, on average, increase with p.

Finally, we specify that not all layerings of minimum height, computable from the same DAG, have the same probability of being generated by Algorithm 6; however, we believe that this does not impact what was concluded thanks to the experiments. We conclude reporting in Fig. B.7 a visual example of layerings produced by Algorithm 6 from the same 32-node DAG/layering pair.
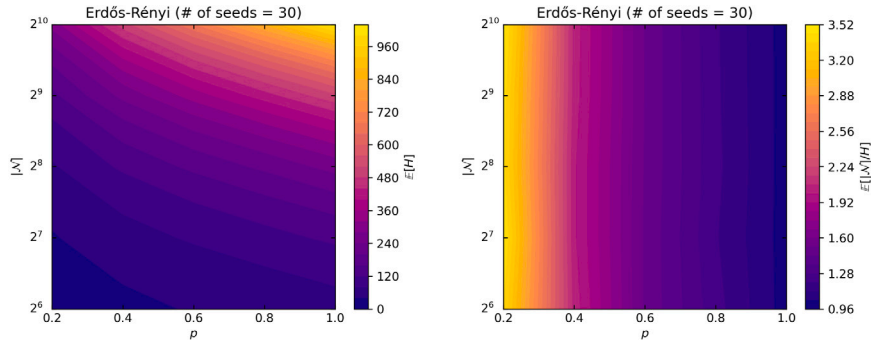
Fig. A.5. Summary of the defined graph attributes over the investigated ER parameter space. **Left**: mean height. **Right**: mean height attenuation. Plots span the same size-probability space as Figs. 3 and 4. Attribute values are color-coded.
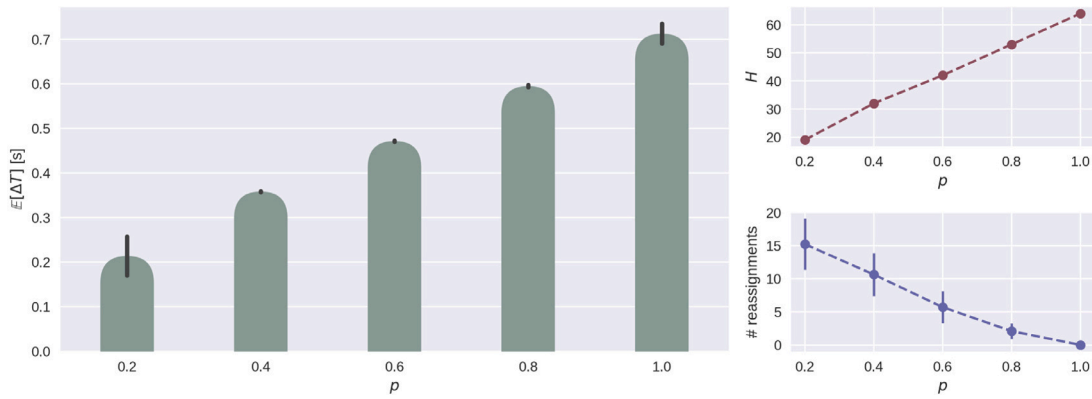

Fig. B.6. Results of the node reassignment experiment. **Left**: mean forward pass execution times and standard deviations (black error-bars). Each green bar refers to a specific ER probability/density, p. **Top-right**: height H of the tested computational DAGs. **Bottom-right**: averge number of node reassignments, with confidence interval (standard deviation), performed by Algorithm 6 per density regime.
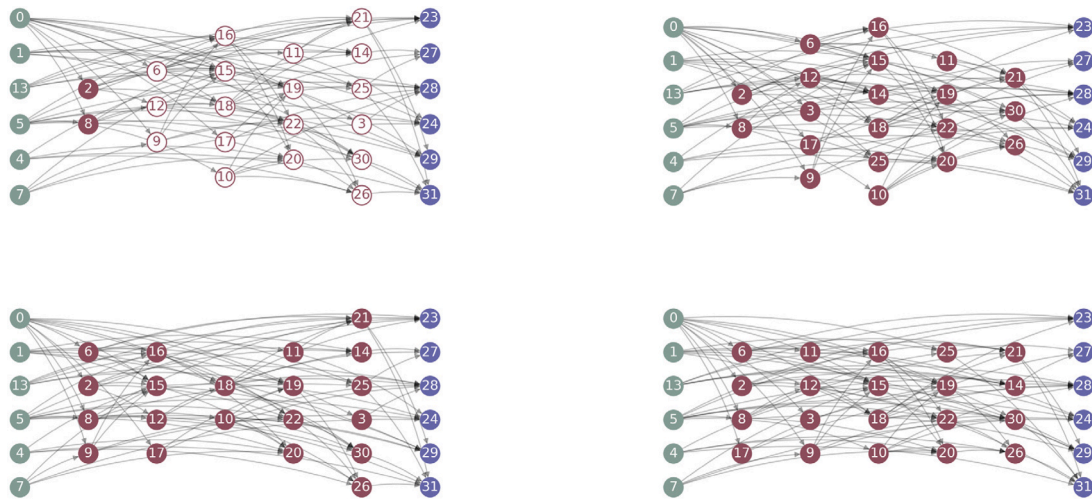

Fig. B.7. Layering visualizations. All partitions refer to the same 32-node DAG. The top-left layering has been produced through the *longest-path algorithm*; all others were processed by Algorithm 6. Sources, hidden nodes and sinks are represented in green, red and violet, respectively. The white vertices in the top-left panel denote nodes that can be moved according to the presented node reassignment procedure.

Appendix C. Longest-path algorithm

To ensure comprehensiveness, we report in Algorithm 7 the *longest-path layering algorithm* for partitioning nodes into layerings of minimum height. This algorithm utilizes two sets of vertices, namely U and Z, which are initially empty. The variable l represents the label of the layer currently under construction, $\tilde{L}_l$. Whenever a node is assigned to a layer, it is also included in the set U, which contains all the

vertices already assigned to a layer. On the other hand, Z consists of the vertices assigned to a layer below the current one. To assign a new node v to the current layer, we select it from the vertices that have not yet been assigned to any layer and whose immediate successors are already assigned to the layers below the current one. The final layering $\mathcal{L} = \{L_0, \dots, L_{H-1}\}$ is obtained by reversing layer indices: $L_i = \tilde{L}_{H-i}$, $0 \leq i \leq H - 1$.

The *longest-path algorithm* possesses two notable advantages: simplicity and linear time complexity. By employing Algorithm 7, nodes are positioned as close as possible to the sinks.

$U \leftarrow \emptyset$;
$Z \leftarrow \emptyset$;
$l \leftarrow 1$;
while $U \neq \mathcal{N}$ **do**
  set $\leftarrow \{v \in \mathcal{N} \setminus U : \{u : (v, u) \in \mathcal{E}\} \subseteq Z\}$;
  if set $\neq \emptyset$ **then**
    $v \sim$ set;
    $\tilde{L}_l \leftarrow \tilde{L}_l \cup \{v\}$;
    $U \leftarrow U \cup \{v\}$;
  end
  if set $= \emptyset$ **then**
    $l \leftarrow l + 1$;
    $Z \leftarrow Z \cup U$;
  end
end
$\mathcal{L} \leftarrow \{\tilde{L}_H, \dots, \tilde{L}_1\}$;

Algorithm 7: The *longest-path layering algorithm*.

References

[1] P. Erdős, A. Rényi, et al., On the evolution of random graphs, Publ. Math. Inst. Hung. Acad. Sci. 5 (1) (1960) 17–60.

[2] R.L.S. Monteiro, T.K.G. Carneiro, J.R.A. Fontoura, V.L. da Silva, M.A. Moret, H.B.d.B. Pereira, A model for improving the learning curves of artificial neural networks, in: D. Li (Ed.), PLoS One 11 (2) (2016) e0149874, http://dx.doi.org/10.1371/journal.pone.0149874, URL https://dx.plos.org/10.1371/journal.pone.0149874.

[3] J. Stier, M. Granitzer, Deepstruct – linking deep learning and graph theory, Softw. Impacts 11 (2022) 100193, http://dx.doi.org/10.1016/j.simpa.2021.100193, URL https://www.sciencedirect.com/science/article/pii/S2665963821000841.

[4] K. Simonyan, A. Zisserman, Deep convolutional networks for large-scale image recognition, 2014, arXiv e-prints arXiv:1409.1556.

[5] K. He, X. Zhang, S. Ren, J. Sun, Deep residual learning for image recognition, in: 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016, pp. 770–778, http://dx.doi.org/10.1109/CVPR.2016.90.

[6] D.P. Kingma, M. Welling, Auto-encoding variational Bayes, in: Y. Bengio, Y. LeCun (Eds.), 2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings, 2014, URL http://arxiv.org/abs/1312.6114.

[7] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A.N. Gomez, Ł. Kaiser, I. Polosukhin, Attention is all you need, in: I. Guyon, U.V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, R. Garnett (Eds.), Advances in Neural Information Processing Systems, Vol. 30, Curran Associates, Inc., 2017, URL https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

[8] L.R. Varshney, B.L. Chen, E. Paniagua, D.H. Hall, D.B. Chklovskii, Structural properties of the caenorhabditis elegans neuronal network, in: O. Sporns (Ed.), PLoS Comput. Biol. 7 (2) (2011) e1001066, http://dx.doi.org/10.1371/journal.pcbi.1001066, URL https://dx.plos.org/10.1371/journal.pcbi.1001066.

[9] L.K. Scheffer, C.S. Xu, M. Januszewski, Z. Lu, S.-y. Takemura, K.J. Hayworth, G.B. Huang, K. Shinomiya, J. Maitlin-Shepard, S. Berg, J. Clements, P.M. Hubbard, W.T. Katz, L. Umayam, T. Zhao, D. Ackerman, T. Blakely, J. Bogovic, T. Dolafi, D. Kainmueller, T. Kawase, K.A. Khairy, L. Leavitt, P.H. Li, L. Lindsey, N. Neubarth, D.J. Olbris, H. Otsuna, E.T. Trautman, M. Ito, A.S. Bates, J. Goldammer, T. Wolff, R. Svirskas, P. Schlegel, E. Neace, C.J. Knecht, C.X. Alvarado, D.A. Bailey, S. Ballinger, J.A. Borycz, B.S. Canino, N. Cheatham, M. Cook, M. Dreher, O. Duclos, B. Eubanks, K. Fairbanks, S. Finley, N. For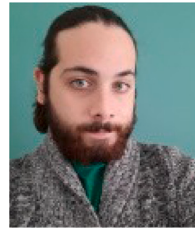knall, A. Francis, G.P. Hopkins, E.M. Joyce, S. Kim, N.A. Kirk, J. Kovalyak, S.A. Lauchie, A. Lohff, C. Maldonado, E.A. Manley, S. McLin, C. Mooney, M. Ndama, O. Ogundeyi, N. Okeoma, C. Ordish, N. Padilla, C.M. Patrick, T. Paterson, E.E. Phillips, E.M. Phillips, N. Rampally, C. Ribeiro, M.K. Robertson, J.T. Rymer, S.M. Ryan, M. Sammons, A.K. Scott, A.L. Scott, A. Shinomiya, C. Smith, K. Smith, N.L. Smith, M.A. Sobeski, A. Suleiman, J. Swift, S. Takemura, I. Talebi, D. Tarnogorska, E. Tenshaw, T. Tokhi, J.J. Walsh, T. Yang, J.A. Horne, F. Li, R. Parekh, P.K. Rivlin, V. Jayaraman, M. Costa, G.S. Jefferis, K. Ito, S. Saalfeld, R. George, I.A. Meinertzhagen, G.M. Rubin, H.F. Hess, V. Jain, S.M. Plaza, A connectome and analysis of the adult Drosophila central brain, eLife 9 (2020) e57443, http://dx.doi.org/10.7554/eLife.57443, URL https://elifesciences.org/articles/57443.

[10] D. Simard, L. Nadeau, H. Kröger, Fastest learning in small-world neural networks, Phys. Lett. A 336 (1) (2005) 8–15, http://dx.doi.org/10.1016/j.physleta.2004.12.078, URL https://linkinghub.elsevier.com/retrieve/pii/S0375960105000022.

[11] O. Erkaymaz, M. Özer, N. Yumuşak, Performance analysis of a feed-forward artifical neural network with small-world topology, Proc. Technol. 1 (2012) 291–296, http://dx.doi.org/10.1016/j.protcy.2012.02.062, URL https://linkinghub.elsevier.com/retrieve/pii/S2212017312000631.

[12] O. Erkaymaz, M. Özer, N. Yumuşak, Impact of small-world topology on the performance of a feed-forward artificial neural network based on 2 different real-life problems, p. 12.

[13] O. Erkaymaz, M. Ozer, Impact of small-world network topology on the conventional artificial neural network for the diagnosis of diabetes, Chaos Solitons Fractals 83 (2016) 178–185, http://dx.doi.org/10.1016/j.chaos.2015.11.029, URL https://linkinghub.elsevier.com/retrieve/pii/S096007791500394X.

[14] O. Erkaymaz, M. Ozer, M. Perc, Performance of small-world feedforward neural networks for the diagnosis of diabetes, Appl. Math. Comput. 311 (2017) 22–28, http://dx.doi.org/10.1016/j.amc.2017.05.010, URL https://linkinghub.elsevier.com/retrieve/pii/S0096300317302989.

[15] M. Annunziato, I. Bertini, M. De Felice, S. Pizzuti, Evolving complex neural networks, in: R. Basili, M.T. Pazienza (Eds.), AI*IA 2007: Artificial Intelligence and Human-Oriented Computing, in: Series Title: Lecture Notes in Computer Science, vol. 4733, Springer Berlin Heidelberg, Berlin, Heidelberg, 2007, pp. 194–205, http://dx.doi.org/10.1007/978-3-540-74782-6_18, URL http://link.springer.com/10.1007/978-3-540-74782-6_18.

[16] R. Albert, A.-L. Barabási, Statistical mechanics of complex networks, Rev. Modern Phys. 74 (2002) 47–97, http://dx.doi.org/10.1103/RevModPhys.74.47, URL https://link.aps.org/doi/10.1103/RevModPhys.74.47.

[17] D.C. Mocanu, E. Mocanu, P. Stone, P.H. Nguyen, M. Gibescu, A. Liotta, Scalable training of artificial neural networks with adaptive sparse connectivity inspired by network science, Nature Commun. 9 (1) (2018) 2383, http://dx.doi.org/10.1038/s41467-018-04316-3, URL http://www.nature.com/articles/s41467-018-04316-3.

[18] J. You, J. Leskovec, K. He, S. Xie, Graph structure of neural networks, 2020, URL http://arxiv.org/abs/2007.06559 arXiv:2007.06559 [cs, stat].

[19] G.M. Platt, X.-S. Yang, A.J. Silva Neto (Eds.), Computational Intelligence, Optimization and Inverse Problems with Applications in Engineering, Springer International Publishing, Cham, 2019, http://dx.doi.org/10.1007/978-3-319-96433-1, URL http://link.springer.com/10.1007/978-3-319-96433-1.

[20] J. Stier, M. Granitzer, Structural analysis of sparse neural networks, Procedia Comput. Sci. 159 (2019) 107–116, http://dx.doi.org/10.1016/j.procs.2019.09.165, URL https://www.sciencedirect.com/science/article/pii/S1877050919313432, Knowledge-Based and Intelligent Information & Engineering Systems: Proceedings of the 23rd International Conference KES2019.

[21] S. Xie, A. Kirillov, R. Girshick, K. He, Exploring randomly wired neural networks for image recognition, in: 2019 IEEE/CVF International Conference on Computer Vision (ICCV), 2019, pp. 1284–1293, http://dx.doi.org/10.1109/ICCV.2019.00137.

[22] M. Wortsman, A. Farhadi, M. Rastegari, Discovering neural wirings, in: H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché Buc, E. Fox, R. Garnett (Eds.), Advances in Neural Information Processing Systems, Vol. 32, Curran Associates, Inc., 2019, URL https://proceedings.neurips.cc/paper_files/paper/2019/file/d010396ca8abf6ead8cacc2c2f2f26c7-Paper.pdf.

[23] N. Roberts, D.A. Yap, V.U. Prabhu, Deep connectomics networks: Neural network architectures inspired by neuronal networks, 2019, URL http://arxiv.org/abs/1912.08986 arXiv:1912.08986 [cs, stat].

[24] B. Zoph, Q.V. Le, Neural architecture search with reinforcement learning, in: 5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings, OpenReview.net, 2017, URL https://openreview.net/forum?id=r1Ue8Hcxg.

[25] H. Liu, K. Simonyan, Y. Yang, DARTS: Differentiable architecture search, in: 7th International Conference on Learning Representations, ICLR 2019, New Orleans, la, USA, May 6-9, 2019, OpenReview.net, 2019, URL https://openreview.net/forum?id=S1eYHoC5FX.

[26] Y.-C. Gu, L.-J. Wang, Y. Liu, Y. Yang, Y.-H. Wu, S.-P. Lu, M.-M. Cheng, DOTS: Decoupling operation and topology in differentiable architecture search, in: 2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), IEEE, Nashville, TN, USA, 2021, pp. 12306–12315, http://dx.doi.org/10.1109/CVPR46437.2021.01213, URL https://ieeexplore.ieee.org/document/9578254/.

[27] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, S. Chintala, PyTorch: An imperative style, high-performance deep learning library, in: H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché Buc, E. Fox, R. Garnett (Eds.), Advances in Neural Information Processing Systems, Vol. 32, Curran Associates, Inc., 2019, pp. 8024–8035, URL https://proceedings.neurips.cc/paper/2019.

[28] R. Tamassia, Handbook of Graph Drawing and Visualization, first ed., Chapman & Hall/CRC, 2016.

[29] P. Healy, N.S. Nikolov, How to layer a directed acyclic graph, in: P. Mutzel, M. Jünger, S. Leipert (Eds.), Graph Drawing, Springer Berlin Heidelberg, Berlin, Heidelberg, 2002, pp. 16–30.

[30] K. He, X. Zhang, S. Ren, J. Sun, Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification, in: 2015 IEEE International Conference on Computer Vision (ICCV), 2015, pp. 1026–1034, http://dx.doi.org/10.1109/ICCV.2015.123.

[31] D.P. Kingma, J. Ba, Adam: A method for stochastic optimization, in: Y. Bengio, Y. LeCun (Eds.), 3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings, 2015, URL http://arxiv.org/abs/1412.6980.

[32] I. Sutskever, J. Martens, G. Dahl, G. Hinton, On the importance of initialization and momentum in deep learning, in: S. Dasgupta, D. McAllester (Eds.), Proceedings of the 30th International Conference on Machine Learning, in: Proceedings of Machine Learning Research, vol. 28, PMLR, Atlanta, Georgia, USA, 2013, pp. 1139–1147, URL https://proceedings.mlr.press/v28/sutskever13.html.

Matteo Ferrante is a Ph.D. candidate at the National AI Ph.D. – Health and Life Sciences program, studying neuromorphic architectures and generative therapeutic "telepathy". He has degrees in Physics and Biomedical Physics from the University of Pavia. His interests lie in artificial intelligence, precision medicine, and neuroscience. His Ph.D. project focuses on decoding visual stimuli in the brain and generating activation maps using mappings between latent spaces of the brain and artificial neural networks.
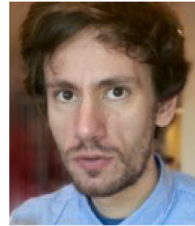
Andrea Duggento, Assistant Professor of Medical Physics at UNITOV, holds bachelor's and master's degrees in Theoretical Physics from the University of Pisa and a Ph.D. in Physics from Lancaster University. With a Medical Physics Degree from the University of Rome "Tor Vergata", Andrea's research focuses on nonlinear dynamical systems, statistical analysis, and information processes in biological networks. His recent work explores directed functional networks in the brain, with publications in prestigious journals such as Physical Review Letters and Philosophical Transactions of the Royal Society.

Nicola Toschi is an Associate Professor in Medical Physics at the University of Rome "Tor Vergata" and Research Staff and Faculty at the Athinoula A. Martinos Center for Biomedical Imaging (Harvard Medical School). He has previously worked as a strategy consultant at McKinsey & Company, as a facilitator for the United Nations convention on Climate Change, with the Italian National Television (RAI) and as a project coordinator with AMREF. His research is interdisciplinary, with a focus on scientific and technological solutions for the deployment of advanced physical and mathematical techniques in order to extract quantitative information of investigative, diagnostic and prognostic value in a clinical context. He is a senior member of the IEEE society, an active member of ISMRM and OHBM, a founding member of the Alzheimer's Precision Medicine Initiative (AMPI) a member of the Technical Committee on Cardiopulmonary Systems.

Tommaso Boccato is a student in the Italian National Ph.D. Program in Artificial Intelligence at the Tor Vergata Medical Physics Section. He holds a bachelor's degree in Information Engineering and a master's degree in ICT, both from the University of Padova. His interests include neural networks, deep learning, network science, and computer vision. Tommaso's current research focuses on bio-inspired neural architectures.