Toward Efficient Deep Learning for Graph Drawing (DL4GD)

Loann Giovannangeli, Frederic Lalanne, David Auber, Romain Giot, Romain Bourqui

Toward efficient Deep Learning for Graph Drawing (DL4GD)

Loann Giovannangeli, Frederic Lalanne, David Auber, Romain Giot and Romain Bourqui

Abstract—Due to their great performance in many challenges, Deep Learning (DL) techniques keep gaining popularity in many fields. They have been adapted to process graph data structures to solve various complicated tasks such as graph classification and edge prediction. Eventually, they reached the Graph Drawing (GD) task. This paper is an extended version of the previously published *(DNN)*^{2} and presents a framework to leverage DL techniques for graph drawing (DL4GD). We demonstrate how it is possible to train a Deep Learning model to extract features from a graph and project them into a graph layout. The method proposes to leverage efficient Convolutional Neural Networks, adapting them to graphs using Graph Convolutions. The graph layout projection is learned by optimizing a cost function that does not require any ground truth layout, as opposed to prior work. This paper also proposes an implementation and benchmark of the framework to study its sensitivity to certain deep learning-related conditions. As the field is novel, and many questions remain to be answered, we do not focus on finding the most optimal implementation of the method, but rather contribute toward a better understanding of the approach potential. More precisely, we study different learning strategies relative to the models training datasets. Finally, we discuss the main advantages and limitations of DL4GD.

Index Terms—Graph drawing, Deep learning, Graph neural network, Graph convolution

1 Introduction

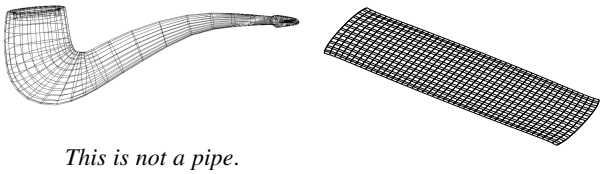
Graphs are common discrete mathematical structures defining relations between entities. Their applications are numerous as many businesses use networks for modeling their technical needs (*e.g.*, traffic of any kind, social network, biology). This structure has been studied for decades to find algorithms that solve a series of tasks efficiently (*e.g.*, search an element, find structure properties). More than ever, with the democratization of *IoT* where every *thing* is smart and communicates with other *things*, data become massive and efficient algorithms are necessary to process them.

This paper focuses on the Graph Drawing task in a context of Node-Link (NL) diagram representations. They consist in representing entities (*i.e.*, nodes) with geometric shapes (*e.g.*, discs, squares) and their relations (*i.e.*, edges) as lines connecting them. The challenge is to find an optimal *layout* (*i.e.*, coordinate for every node and/or edge) that emphasizes the graph structure. The same graph can have different optimal layouts that emphasize (or optimize) different aesthetic criteria, as shown in Figure 1.

Lately Deep Learning (DL) techniques gained popularity thanks to their great performance in many application fields (*e.g.*, Image Processing [2], Natural Language Processing [3]). The main advantages of DL techniques are their ability to learn by themselves an efficient strategy to solve a given task, and their short execution time. At the cost of an expensive training, a forward pass in a DL model is fast with standard architectures since they apply a polynomial transformation to their input with their learned weights. These techniques were adapted to process graphs, but mostly dedicated to classification and labeling on graphs, nodes or edges [4], [5], [6].

This paper is an extended version of *(DNN)*^{2} [1] and presents a framework for leveraging Deep Learning techniques for Graph Drawing (DL4GD). Inspired by Convolutional Neural Networks (CNN) for Image Processing, *(DNN)*^{2} proposes to process graphs with state-of-the-art model architectures. Since these architectures were designed to process images, some adjustments are made to feed them with graph data structure, such as the replacement of standard convolutions with graph convolutions. These model architectures learn by themselves to extract features from their input data through a *feature extraction* stage. Then, a series of fully connected layers learns the model to synthesize these features by regressing them into the desired dimension. By learning graph feature representations by itself, the model can also be considered as a Node Embedding technique, *i.e.*, it produces a descriptor vector of the desired dimension for every node. In this paper, we focus on the projection into a 2D space to visualize graph layouts in static images and we use *(DNN)*^{2} as a Graph Drawing technique.

The first contribution is an extensive presentation of the *(DNN)*^{2} framework, an unsupervised Deep Learning (DL) approach to Graph Drawing. The motivation to find an unsupervised DL



This is not a pipe.

Fig. 1: Two drawings of the same graph. On the left, it is laid out to suggest the abstract concept of a "pipe". On the right, it shows that the graph is a mesh. These representations show that drawings of the same graph can be designed to emphasize different properties. This figure is inspired by R. Magritte surrealist painting "*La trahison des images*" (1929) in which the artist wanted to show that the representation of an object (here, a graph drawing) is different from the actual object (here, a graph).

All authors are with the Univ. Bordeaux, CNRS, Bordeaux INP, INRIA, LaBRI, UMR 5800, F-33400 Talence, France.
E-mail: {firstname}.{lastname}@u-bordeaux.fr
This paper is an extended version of [1] appeared in GD'21.

framework comes from the numerous limitations of the supervised approach in the context of Graph Drawing (see Section 2.3). The second contribution corresponds to several evaluations to study the model performance under various circumstances. These evaluations do not focus on making the most optimal implementation of the *(DNN)²* framework. Instead, many DL-related design choices are fixed and we compare the behavior of this model when trained with various external parameters. More specifically, we study how it is affected by heterogeneity in its training dataset in terms of performance and capability to generalize to unseen data. We also evaluate if the model is able to learn specific graph families topology to generate better layouts dedicated to them. Finally, the evaluation studies the benefits of transfer learning on the framework, and compares *(DNN)²* with other state-of-the-art related algorithms. This paper extends the study of GD'21 [1] by considering more specific cases of evaluation (*e.g.*, specific graph families) with different monitored properties. DL models learning being guided by their training data enables a more fine-grained study of the approach performance.

In this paper, all graphs are considered connected, undirected and unweighted. Unconnected graphs can be handled by individually drawing their connected components. Though we do not consider directed and weighted graphs, the framework design may be extended to handle them as described in Section 3.4.

The remainder of the paper is organized as follows. Section 2 presents related work with a focus on Graph Drawing, structures for learning from graphs, and learning techniques for graph drawing. Section 3 presents the *(DNN)²* framework and Section 4 presents evaluations of an implementation. Finally, Section 5 discusses design choices having a major impact on *(DNN)²* and the main limitations observed, while Section 6 draws conclusions and presents leads for future works.

Notations

Let $G = (V,E)$ be a graph where $V = \{v_0, v_1, ..., v_{N-1}\}, N = |V|$ is its set of nodes and $E \subseteq V \times V$ its set of edges. A graph layout is defined as a vector $X \in \mathbb{R}^{N \times D}$ where X_i is the node v_i projection in D dimensions ($D = 2$ in this paper). The euclidean distance (resp. shortest path length) in the projected (resp. graph theoretical) space between two nodes v_i and v_j is denoted $||X_i - X_j||$ (resp. δ_{ij}).

2 Related Work

This section presents research work related to Graph Drawing, structures and tools for learning from graphs, and learning techniques to process and draw graphs.

2.1 Graph Drawing

Historically, there were three main approaches to Node-Link (NL) diagrams designs. The first was to design algorithms dedicated to graphs with certain properties (*e.g.*, planar graph [7] or tree [8]). This approach enables to offer guarantees about the resulting layout, but cannot be adapted to general graph structures. The second method is the *force-directed* approach [9], [10] which aims at simulating a model of attraction-repulsion (*e.g.*, springs) to converge toward a layout where edge lengths are uniform. This approach is computationally expensive and motivated research around *fast* algorithms (*e.g.*, [11]) to lay large graphs out. Finally, the last approach is the adaptation to a graph context of *multi-dimensionality reduction* algorithms (MDS) [12], [13] who were originally designed to visualize the similarity between rows of data tables with many columns.

All these methods aim at producing *pleasing* layouts, which is measured with aesthetic metrics [14]. These metrics are cornerstone to the Graph Drawing field as they enable the quantitative evaluation of layouts, the comparison of layout techniques, and are admitted to echo human perception [15], [16], [17].

Lately, Gradient Descent and Machine Learning (ML) models were used to optimize these criteria. *GD²* [18] used gradient descent to optimize a combination of several aesthetic metrics whose weights are given by the user. Its flexibility enables to tune the aesthetics weights according to the desired aspects of the layout. *tsNET* [19] also used gradient descent to optimize a Kullback-Leibler based function that focuses on neighborhood preservation, adapting t-SNE [20] to graph drawing. This function will be further presented in Section 3.3 as our method optimizes it as well. Kwon *et al.* [21] used a ML approach to estimate both a graph layout and its corresponding aesthetics at the same time with a new design of graph kernels. Finally, *S_GD²* [22] relies on stochastic gradient descent to optimize *stress* modeled with *constraints* between pairs of nodes. The approach is inspired by cloth behavior simulations where clothes are modeled with a mesh of vertices. To avoid the costly computation of the stress ($\mathcal{O}(N^2)$ if we already know all pairs shortest paths δ) on a whole mesh, they independently relax constraints between pairs of nodes by moving them in opposite directions.

In this paper, we propose another approach to produce graph layouts by learning a Deep Neural Network to embed a theoretical graph structure into an $\mathbb{R}^2$ scalar for every node which we interpret as the graph layout.

2.2 Learning for Graph Processing

The democratization of Deep Learning techniques stimulated the interest in learning techniques for many visualization tasks [23], [24], including Graph Drawing. Since most learning techniques work best with a significant amount of data to learn from, node embedding techniques [25], [26], [27] have been proposed to create descriptor vectors of a graph nodes. These techniques can be used to preprocess graphs in learning frameworks by associating a *features vector* to every node. The vectors are themselves designed to encode node properties and give learning techniques enough input data to learn from.

Neural network structures to learn from graph data emerged in 2008 with the original Graph Neural Network of Scarselli *et al.* [28]. Recently, they regained popularity with the Graph Convolutional Network (GCN) proposed by Kipf and Welling [29] where they adapt the concept of convolutions to a graph context by adding a topological component to the operator. However, GCNs are unable to differentiate nodes with the same local neighborhood [1], [30], [31]. Since then, GCNs have been declined several times for various needs and to overcome different limitations (*e.g.*, FastGCN [32], GraphSAGE [33], P-GNN [30]). Other structures, such as Graph Attention Networks (GAT) [34] and Graph Isomorphism Network (GIN) [31], were also proposed to learn from graph data and overcome some of GCNs drawbacks. Tiezzi *et al.* [35] compared the performance of three Shallow Neural Networks (one for GCN, GAT and GIN). They conclude that GAT is the best performing structure out of the three. In this paper, since we leverage a Convolutional Neural Network architecture from the literature (*i.e.*, *ResNet50* [36]), we focus on Graph Convolutions.

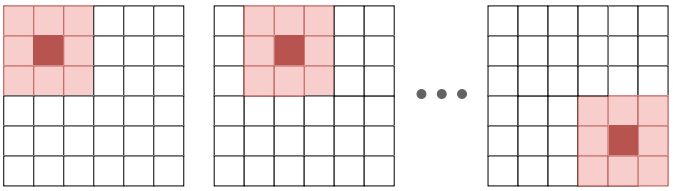
The main tasks studied with these techniques are classification and labeling for graphs, nodes or edges [4], [5], [6]. But recently, they began drawing attention in the Graph Drawing community. For instance, Tiezzi *et al.* [35] compared GCN, GAT and GIN structures on the Graph Drawing task.

2.3 Learning for Graph Drawing

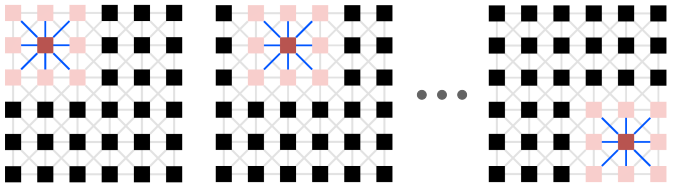
GND [35] and GraphTSNE [37] are two frameworks that use Shallow Neural Networks to draw graphs. While each training of GraphTSNE is done on one specific graph (fed to the model as batches of nodes), GND is trained with batches of graphs to generate layouts of any input graph. The model optimizes a loss function either supervised or unsupervised and the authors found that supervising the training with the result layout from an existing algorithm leads to better performance than unsupervised approaches. However, we argue supervised graph drawing is not satisfactory and will detail why shortly. Another originality of GND was to propose the use of a so-called *Neural Aesthete* model to drive a *Drawer* model learning. Since aesthetic metrics can be very expensive to compute (*e.g.*, stress takes $\mathcal{O}(N^2)$ if we know δ), the idea is to train the *Neural Aesthete* neural network to estimate an aesthetic metric on a graph layout. Then, it can be used in a *Drawer* model loss function as a fast derivable component that estimates the aesthetic metric. The concept is close to that of Haleem *et al.* [38] who estimated aesthetics with a Convolutional Neural Network working on images of graph layouts, but whose approach was sensitive to graphical design choices (*e.g.*, node radius) interfering with the estimation.

DeepDrawing [39] is the first technique to leverage Deep Neural Networks to train a model to project graphs into layouts. The training of DeepDrawing is supervised with the layouts produced by a *groundtruth* algorithm that should be reproduced. That is, the model is trained to minimize the distance (*i.e.*, Procrustes statistic [40], [41]) between the layouts it predicts and the *label layout* produced by the *groundtruth* algorithm. Espadoto *et al.* [42] also studied this approach in their framework which could learn to mimic any multidimensional projection algorithm thanks to Deep Learning. Both works raised limitations to the supervised learning of such models. Like most Deep Learning models, it has to be trained on thousands of samples. However, with such a supervised approach, the data generation requires to generate both the graphs and their corresponding layout with the *groundtruth* algorithm. As such a model is usually trained to reproduce layouts of an efficient but expensive algorithm, computing thousands of *label layouts* can be prohibitive. A second limitation is that the model is very sensitive to diversity in the *label layouts*. If layouts of similar graphs are not approximately the same, the model will have difficulties to converge. Wang *et al.* [39] mentioned having to manually tune some algorithm-specific parameters to obtain desirable visual properties. Since DeepDrawing learns to mimic the *groundtruth* algorithm, it cannot generate better layouts and also learns to reproduce its defects. Finally, as the model is intrinsically related to the *groundtruth* algorithm, its capability to generalize to unseen graph topologies is uncertain.

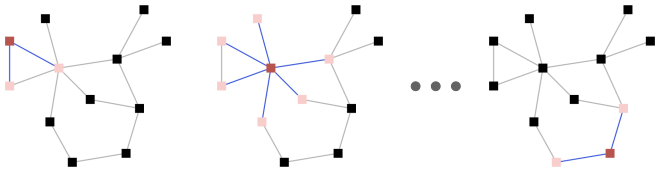
(DNN)^2 [1] and *DeepGD* [43] are the two first unsupervised Deep Learning approaches to graph drawing. Both frameworks were simultaneously published and are actually quite similar in their behavior. The main differences between the two papers are some design choices relative to data structures and deep learning hyperparameters, and the focus of their method evaluation. DeepGD



(a) Standard convolution on an image



(b) Graph-oriented view of a convolution on an image



(c) Graph convolution

Fig. 2: Illustrations of (a) a standard convolution on an image; (b) a standard convolution on an image from a graph perspective (pixels are nodes, nodes are linked when adjacent in the grid); and (c) the generalization of the convolution on a graph.

model architecture gets the graph topology as an adjacency matrix *and* an edge feature matrix. In addition, DeepGD architecture includes the idea of an *Edge Feature Network* that is local to every graph convolution layer and is claimed to make the model able to interpret the graph topology differently according to the depth of the convolution layer. Finally, another distinction between DeepGD and this paper is the focus of their evaluation. The different choices of *(DNN)*^{2} will be detailed in the dedicated Section 3 for the design, and Section 4 for the evaluation. The intuition of the method is that as Graph Convolutions can be similar to standard convolutions on images (as illustrated in Figure 2), we can leverage Deep Learning advances of the Image Processing community and adapt them to extract features from graph data. The advantage of such an approach is that it does not require much information about the graphs to train the model, since it learns by itself to extract features (*i.e.*, feature extraction stage) from the topology before projecting them (*i.e.*, regression stage). By exiting earlier in the network (*e.g.*, at the end of the feature extraction stage), the framework can also be considered as a Node Embedding technique. The approach efficiency for Node Embedding was addressed by *DeepGD* [43] and is not investigated in this paper as we focus on graph drawing.

3 Framework *(DNN)*^{2}

This section describes the *(DNN)*^{2} [1] framework and some design choices made for the evaluation to come in further sections.

3.1 Architecture

Convolutional Neural Networks (CNNs) have proven to be efficient techniques to solve many tasks in the Image Processing community [2]. By design, they convolve pixels with their neighbors

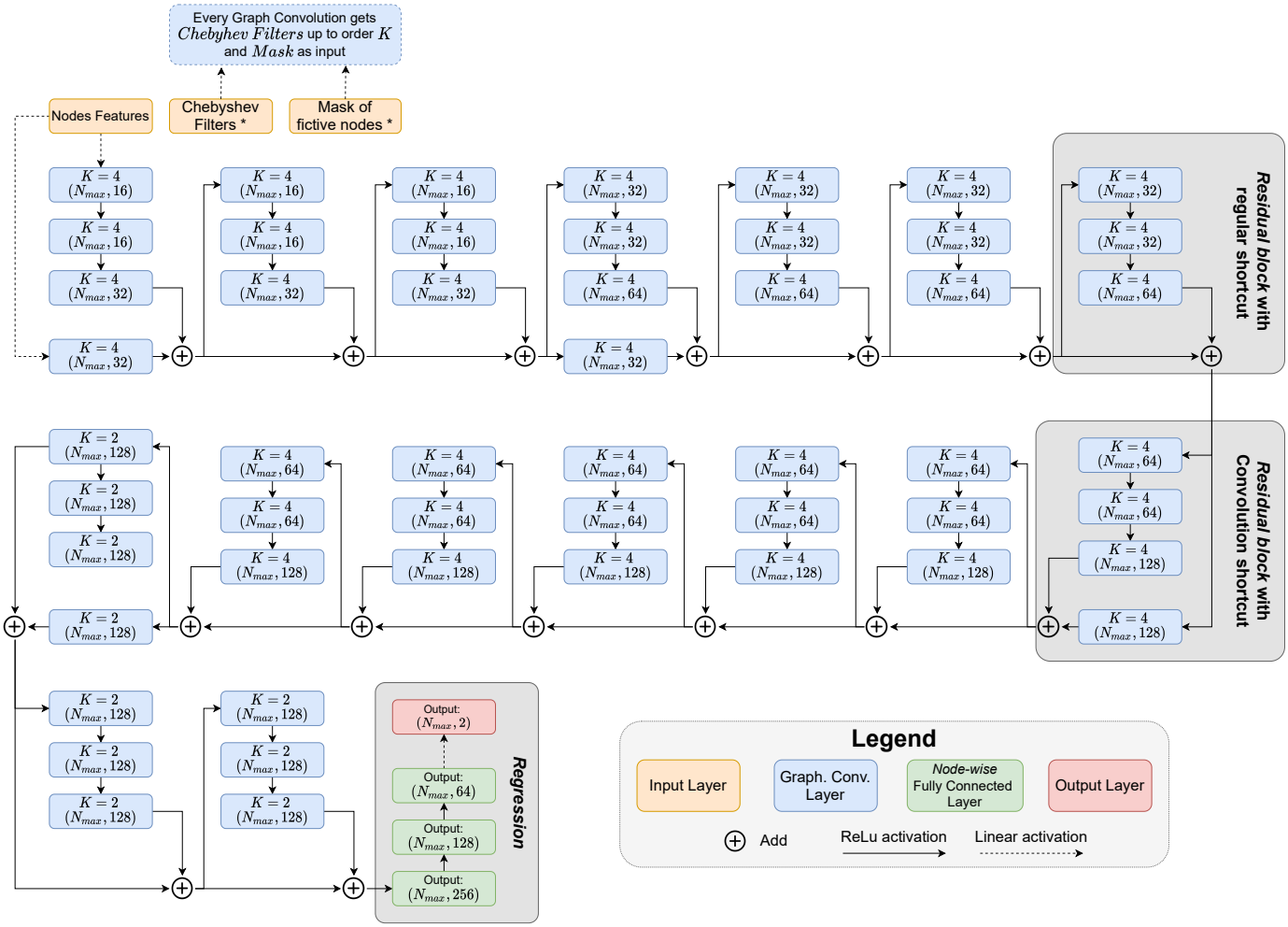


Fig. 3: Schema of *(DNN)*^{2} architecture based on ResNet50. Only 2 out of 16 residual blocks are highlighted for illustration purposes. Chebyshev filters are provided up to order $K = 4$ to all convolution layers except the last 10 (*i.e.*, last 3 residual blocks) where they are up to order $K = 2$. The maximum number of nodes during the training was set to $N_{max} = 128$. At the end of each residual block, node feature vectors are normalized and the mask of real-fictive nodes is applied.

(see Figure 2a). Hence, their adaptation to a graph context is straightforward: pixels can be considered as *nodes* that are linked with an *edge* if they are adjacent in the grid, as in Figure 2b. By providing a data structure that encodes adjacency, can compute convolutions on graph data directly, as illustrated in Figure 2c. The idea of *(DNN)*^{2} is to leverage CNNs architectures to process graph data and infer their layout. Their adaptation to a graph structure is done by replacing the standard convolutions by Graph Convolutions [29], [44].

In our experimentation, we used *ResNet50* [36] as the basis of our model architecture since it is well proven and reaches great performance for various tasks. This architecture groups convolutions in *residual blocks*, themselves grouped in *stacks*, which sequence builds the *feature extraction* stage of the model. A residual block (see Figure 3) is a pattern with two branches: a main branch where the input features are convolved, and a *shortcut* branch where they are (almost) not. At the end of the block, the two branches are aggregated, making the model able to work on different levels of abstraction of the data at the same time. This pattern is the main success factor in *ResNet* architecture. Once features are extracted, they undergo a *regression* stage: a sequence of node-wise fully connected layers that learns to reduce the features to the desired dimension (*e.g.*, 2D or 3D). Each fully connected layer is said *node-wise* because the full connection is within each graph node features and not between different nodes of a graph. The same kernel of learned weights is applied independently to each node feature vector to produce its projection, rather than making *all-against-all* connections between all the feature vectors of all nodes. Having *node-wise* fully connected layers also enables the architecture to be independent of the input graph size.

Among the various implementations of Graph Convolutions (*e.g.*, spatial [45], spectral [29], position-aware [30]), we used the *Spectral* Graph Convolutions. As shown in Figure 3, the model inputs are the graph *signal* and its *spectrum* defined in the next section. The *signal* is the input features for every node that will be extracted and regressed into 2D positions, while the *spectrum* is the topological structure used to convolve nodes with their neighbors.

3.2 Spectral Graph Convolutions

In this paper, we use edge-oriented graph convolutions where each node is described by a feature vector and is convolved with its neighbors to produce its new feature vector. Basically, a graph

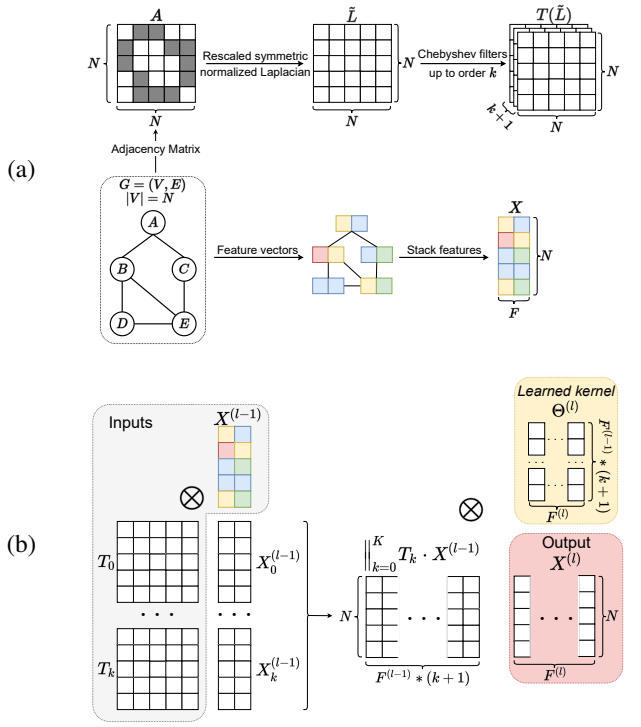


Fig. 4: Illustration of (a) the computation of inputs required to compute a Graph Convolution; and (b) the Graph Convolution operation. In (b), the input feature tensor $X^{(l-1)}$ is a generated vector for the first Graph Convolution (see Section 3.4). For further convolution layers, $X^{(l-1)}$ is the output of the previous layer.

convolution layer l transforms a node feature vector $X_i^{(l-1)}$ in a new vector $X_i^{(l)}$:

$$X_i^{(l)} = X_i^{(l-1)} \cdot \Theta^{(l)} + \sum_{j \in \mathcal{N}(i)} W_{ij} \cdot X_j^{(l-1)} \cdot \Theta'^{(l)} \tag{1}$$

where $X_i^{(l)}$ is the feature vector of node v_i at the layer l and $\mathcal{N}(i)$ is v_i set of neighbors. $W \in \mathbb{R}^{N \times N}$ is a weight factor that encodes how important should the features of node v_j be for the update of node v_i features. This weight matrix is basically defined from the graph topology. Θ and Θ' are the graph convolution weights learned during the training to optimize a cost function. Most Graph Convolutions variants follow this formula and their originality differ on their definition of node neighborhood $\mathcal{N}$, of the weight matrix W and of learnable weights Θ. For example, adjacency or distance matrix can be used as W in *spatial* graph convolutions.

(DNN)^2 derives Equation 1 using the *Spectral* Graph Convolutions as defined by Kipf and Welling [29] and illustrated in Figure 4. In this convolution, node features X are considered as the graph *signal* and are convolved according to the graph *spectrum* (W in (1)). The graph *spectrum* is defined as the eigendecomposion of the rescaled symmetric normalized Laplacian matrix of the graph $\tilde{L}$:

$$\begin{aligned} L &= D - A, \\ L^{sym} &= I_N - D^{-\frac{1}{2}} L D^{-\frac{1}{2}}, \\ \tilde{L} &= \frac{2}{\lambda_{max}} L^{sym} - I_N \end{aligned} \tag{2}$$

where D and A are respectively the degree and adjacency matrices, I_N is the identity matrix of size $N = |V|$, and λ_{max} the highest eigenvalue of L^{sym}.

Since the eigendecomposion of $\tilde{L}$ (*i.e.*, the graph spectrum) is expensive to compute, it is approximated with Chebyshev polynomials [46] up to order K:

$$\begin{aligned} T_0(x) &= 1, \\ T_1(x) &= x, \\ T_k(x) &= 2xT_{k-1}(x) - T_{k-2}(x), \forall k \geq 2 \end{aligned} \tag{3}$$

The order K can thus be seen as the size of the convolution kernel. Finally, the spectral graph convolution layer H^l is defined as:

$$H^l(X) = \left(\Big\|_{k=0}^{K} T_k\left(\tilde{L}\right) \cdot X \right) \cdot \Theta^l \tag{4}$$

where $X \in \mathbb{R}^{N \times F}$ is a stacked tensor of a graph node feature vectors (*i.e.*, graph signal) where each node has F features, $T_k\left(\tilde{L}\right)$ is the edge weight factor for node neighborhood convolutions (*i.e.*, Chebyshev filters) and $\Theta^l \in \mathbb{R}^{(F^{(l-1)} * (K+1)) \times F^{(l)}}$ is the matrix of learned weights in the layer l where $F^{(l)}$ is the size of the desired output feature vector for every node. The symbol $\|$ is used as a *concatenate* operator on all the $T_k(\tilde{L}) \cdot X$ tensors.

3.3 Unsupervised Probability-based Loss

Some approaches consider layout algorithms from the literature as ground truths (*e.g.*, DeepDrawing [39]), but this approach is not satisfactory for multiple reasons (*e.g.*, cost, defects reproduction, see Section 2.3). In addition, there is no such thing as a *ground truth* layout for a graph, since different layouts can optimize various aesthetic metrics, each emphasizing different aspects of the graph (*e.g.*, neighborhood vs. distance preservation). Hence, the training of *(DNN)*^{2} is *unsupervised* as we do not measure its performance in regard of a ground truth layout. Basically, *(DNN)*^{2} can be trained to optimize any smooth function that evaluates how a graph layout is representative of the graph structure. For instance, Ahmed *et al.* [18] proposed smooth expressions of most standard aesthetic metrics to enable their optimization with gradient descent.

Here, we propose to optimize the Kullback-Leibler divergence as proposed by Kruiger *et al.* [19] in *tsNET*. This approach optimizes neighborhood preservation and is an adaptation of the t-SNE [20] algorithm to a graph context defined as:

$$\begin{aligned} C_{comp} &= \frac{1}{2N} \sum_i ||X_i||^2, \\ C_{rep} &= \frac{1}{2N^2} \sum_{i,j \in V, i \neq j} \log\left(||X_i - X_j|| + \varepsilon_r\right), \\ C_{tsNET} &= \lambda_{KL} C_{KL} + \lambda_c C_{comp} - \lambda_r C_{rep} \end{aligned} \tag{5}$$

where C_{KL} (6) is the main topology-related cost term (discussed below). The second term C_{comp} is a *compression* that minimizes the scale of the drawing and is known to accelerate t-SNE convergence. The third term C_{rep} is a *repulsion* that counter-balances the compression effects on the drawing. $(\lambda_{KL}, \lambda_c, \lambda_r)$ are weights used to tune the loss function during the optimization. $\varepsilon_r = \frac{1}{20}$ is a regularization constant.

In the first experimentations of *(DNN)*^{2} [1], we reproduced tsNET protocol by setting a *first stage* training with lambda weights (1, 1.2, 0) to compress the layout and a *second stage* training with (1, 0.01, 0.6). However, we observed that the introduction of the

compression term (and thus the repulsion that counter-balances it) makes it harder for the model to optimize the main C_{KL} term. The original concern that motivated the addition of the compression term was to make convergence faster. But since optimization is done *a priori* in a Deep Learning context, we can afford a longer training convergence as it will not impact the model performance at inference time. Hence, the two additional terms are discarded by setting the λ weights to (1, 0, 0).

The cost function to optimize is then only defined by the Kullback-Leibler (KL) divergence which measures the (dis)similarity between two sets of probabilities:

$$C_{KL} = \sum_{i,j \in V, i \neq j} p_{ij} \log \frac{p_{ij}}{q_{ij}} \tag{6}$$

where p_{ij} and q_{ij} are sets of probabilities that every pair of nodes $(v_i, v_j) \in V^2$ in the graph are connected, based on their location in the graph (*i.e.*, dense or sparse region) and the distance between them. The difference between them is that p_{ij} is computed from the graph distances (*i.e.*, shortest path lengths) while q_{ij} is built upon the euclidean distances in the layout. Evaluating Equation 6 comes down to study if the "*probabilities that all pairs of nodes are neighbors according to their distances* ***in the graph***" are similar to the "*probabilities that all pairs of nodes are neighbors according to their distances* ***in the layout***". If so, the layout is a good representation of the graph. The formal definition of p_{ij} and q_{ij} are postponed to Section 5.1 where we discuss some drawbacks observed during the framework evaluation in Section 4.

3.4 Model Inputs

During the model training, every input tensor is padded with *fictive nodes* to a maximum graph size N_{max}. This enables to have tensors of equal size and greatly eases the model training. Such padding in the model input is commonly used in Deep Learning since samples rarely have the same size (*e.g.*, number of words in sentences for Natural Language Processing, image resolution for Image Processing). The maximum graph size N_{max} is only set during the model training and is not necessary in the inference phase. However, it means that fictive nodes features will be convolved during the model training. To avoid that the model learns from them, a mask of real-fictive nodes is applied after each residual block. Finally, to avoid bias related to node ordering, the model inputs are randomly permuted.

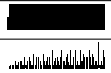
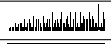
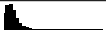
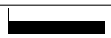
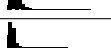
Hence, the first model input is the mask of real-fictive nodes defined as $Mask \in \mathbb{1}^{N_{max}}$ where $Mask_i = 0$ if v_i is a fictive node, 1 otherwise. This tensor is never modified by the model, and is used at different stages in the architecture.

As the model leverages *Spectral* Graph Convolutions, the second input is the graph *spectrum*, approximated with Chebyshev filters up to order K as presented in Section 3.2. This structure is never modified by the model and is provided to graph convolution layers to convolve node features according to the graph spectrum.

The last model input is a tensor that holds the initial node features. That is, the tensor that contains information of each node from which features will be extracted through the first stage of the model and then regressed through the second stage (see Figure 3). There already exist methods to extract features for every node in a graph (*i.e.*, Node Embedding techniques) dedicated to feeding Deep Learning models [25], [26], [27]. But since most layout algorithms from the literature work on the graph topology only (and sometimes include some randomness), we do not use them. As mentioned before, we do not aim at making the most optimal implementation of *(DNN)*^{2} but rather try to keep some design choices simple to limit *black box* effects and enable a proper model performance interpretation. Hence, we create a short vector for every node containing a unique *id* (ranging from 0 to $|V|-1$ for each graph, in generation order) and a random metric. The intuition is that giving a unique *id* to every node can help the model to differentiate nodes having the same local neighborhood and thus avoid some overplotting issues. The random metric provides the randomness factor that is often included in layout algorithms from the literature (*e.g.*, GEM [10]) to avoid ending up in local minima. The node features are then defined as $X \in \mathbb{R}^{N \times 2}$. We also tried a variant of these node features where we added the 2D node positions generated by PivotMDS [12] layout algorithm, as in *tsNET** [19], to check if the model could lead to better results by having the equivalent of a non-random initialization of the node positions. Finally, all the node features from a graph are normalized across that graph dataset to give them the same importance, regardless of their original value domain or the graph size (*e.g.*, nodes *id* feature have higher values with larger graphs).

Graphs are considered unweighted in this study. However, the framework should be able to handle them by encoding weights in the model inputs. For example, node weights can be encoded in the *initial node feature* representation. On the other hand, edge weights can be provided to the model through the topological structure. Here, we used *Chebyshev filters*, but other structures such as weighted adjacency matrix could be experimented. Based on the choice of that topological structure, directed graphs can be handled as well. These ideas could be investigated in future works.

TABLE 1: Distributions of the number of nodes $|V|$, number of edges $|E|$, densities D_2 and distances in the datasets presented in Section 4.1. Distances distribution is the summed count of all distances between all pairs of nodes in all graphs of a dataset. $|E|$ plots X-axis have log scales. All plots of a column have the same range on the X-axis.

Dataset	|V|	|E|	Densities	Distances
HoR				
HeR				
Tree				
Grid				
NN				
Planar				
Commu.				
Rome				

4 Benchmark

4.1 Datasets

This section presents the graph datasets used in the benchmarks. Some of their properties are reported in Table 1. All of them are randomly split into *train* and *validation* subsets for Deep Learning validation purposes. All sets but Her and HoR are split once more for evaluation (*i.e.*, they have a *test* set). Basically, models are trained on a *train* set and a *validation* set is regularly used during training to avoid overfitting on this *train* set. The trained model is then evaluated on unseen data that was set apart from

the beginning: the *test* set. All the graphs in these datasets are connected, undirected, unweighted and have $N \in [2,128]$ nodes.

HoR stands for Homogeneous Random Graphs which is the set of random graphs used in [1]. This dataset is generated with the following constraint: generate 1000 instances of random connected graphs for each number of nodes between 2 and 128. The density is randomly picked and the corresponding number of edges computed following the formula: $D_1 = \frac{2|E|}{|V|*(|V|-1)}$. We do not remove isomorphic graphs which inevitably exist with smaller graph sizes. Having lots of small isomorphic graphs can however be interesting since larger graphs can be decomposed into subsets of small ones, meaning that if the model learns well to draw small graphs, it can help it drawing larger ones containing them.

HeR stands for Heterogeneous Random Graphs. It is generated with the same constraints has HoR except for the graph densities which are randomly taken following $D_2 = \sqrt{\frac{2|E|}{|V|*(|V|-1)}}$. The densities randomness is also controlled to be more uniformly distributed accross the dataset. This enables for more diverse graph structures and leads to a more heterogeneous dataset.

Tree is made of 15 245 tree graphs. Half are shallow with a high maximum degree (see Figure 6b third row), while the other half is deeper (see Figure 6b fourth row).

Grid is the enumeration of all the grids of degree 4, 6 and 8 with $N \leq 128$, $width \geq 2$ and $height \geq 2$.

NN is made of 12 700 Nearest Neighbor graphs generated with the Tulip [47] *Grid Approximation* algorithm. Basically, nodes are randomly projected on a 2D space, then close nodes (*i.e.*, euclidean distance less than a threshold) are linked with an edge.

Planar is made of 19 050 random planar graphs.

Commu are 18 161 gaussian random partition graphs. The generator parameters were set as proposed by Brander *et al.* [48].

Rome is a dataset of 11 531 connected undirected graphs provided by the Graph Drawing symposium[1]. These graph are diverse and of no specific family but are rather sparse (see the high concentration of low densities in Table 1). As it is well-established in the community, it is the main evaluation dataset in this paper and will be used in Section 4.4 and 4.6.

Aside from Rome, all datasets were generated for this study.

4.2 Training Parameters

This section describes the selected hyperparameters to train the models, and whose performances are evaluated later in this study.

Regarding the model, the first design choice is to use ResNet50 [36] as the basis for the architecture. It is selected because it is one of the most used standard architectures and achieves great performance in many challenges. The maximum graph size N_{max} is set to 128 to be larger than the largest Rome graph (see Section 4.1) and to provide a good trade-off between graph size and graph variability. On one hand, N_{max} has to be large enough so that generating thousands of graphs leads to various topologies. On the other hand, N_{max} has to be kept reasonable since the loss function requires to compute the distance matrix of every graph to evaluate the generated layouts. The graph convolutions kernel size (*i.e.*, order of the Chebyshev filters) is set to $K = 4$ for all convolution layers except the last 10 (*i.e.*, last 3 residual blocks) where they are set to $K = 2$ (see Figure 3). Doing so gives more weight to closer neighborhood preservation in the last Graph Convolution layers of the architecture.

[1]Rome graphs: http://www.graphdrawing.org/data.html

For training on small datasets (*e.g.*, Grid, see Section 4.1), the whole *train* and *validation* sets are repeated until their size is at least 8800 and *1760* respectively. It means the models will see the same graph several times with various random node orders in the *train* and *validation* sets. We repeat the entire dataset to preserve their distribution (see Table 1). This data augmentation is designed to give the models enough data to learn from. However, the very consideration of a *validation* set can be questioned on such datasets: what are the benefits of preventing overfitting if we can enumerate and learn how to draw *all* the possible graphs? This paper does not study this question, but it is an interesting lead of investigation for future work.

Regarding the training hyper-parameters; the models were trained with Adam optimizer [49] with a starting learning rate of 10^{-3}. The trainings were set to last at most 200 epochs, with an early stop if the validation loss did not improve during 20 consecutive epochs. This choice was made after the experimental verification that training the models for 1000 epochs straight did not provide significant benefit. In practice, most of the $(DNN)^2$ instances training early stopped in fewer than 100 epochs. Trainings on the HoR and HeR datasets were made with a batch size set to 400 on a CPU cluster with 20 workers for a training time of several hours each. On all others datasets, the batch size was set to 32 and trainings were conducted on a NVIDIA 2080 Super Max-Q (Mobile) GPU for a training time of about one hour each. That is to say, it does not require heavy resources. As for inference time, it is not monitored since it is almost instantaneous compared to graph pre-processing (*e.g.*, loading, Chebyshev filters computation).

4.3 Evaluation Metrics and Protocol

The aesthetic metrics used to compare the graph layout techniques are described in the following and formally defined in Table 2. Those noted with a * have been inverted so that all are oriented *lower is better*.

Aspect Ratio* [18] is defined as the worst ratio between the drawing width and height after a series of rotations.

Angular resolution* [18] is the ratio between the minimum angle formed by two edges on a node in the drawing and the optimal angle that should be formed by the edges on the node with the maximum degree.

Edge crossings number [14], [18] is the number of times edges cross each other in the drawing.

Cluster overlap [50] corresponds to the normalized sum of distances between nodes that are at a distance smaller than r and not in the same cluster. The metric requires a neighborhood radius r in the drawing and a clustering algorithm. Here, $r = 0.2$ and the clustering algorithm is MCL [51], an efficient deterministic clustering algorithm.

Neighborhood preservation* [19] is the sum, for each node, of the size of the intersection over union between its k-hop-neighborhood U in the graph and the set of its $|U|$ nearest nodes in the layout.

Stress [9], [18] measures how the euclidean distances in the layout deviate from the distances in the theoretical space (*i.e.*, shortest path lengths), with a weighting inversely proportional to the theoretical distance. Here, stress is normalized to obtain an average *stress per node* and fairly compare graphs of different sizes.

Statistical validation should support the results interpretation as we work with thousands of graphs. To assert whether the

TABLE 2: Quality metrics definitions. Those noted with a * have been inverted so that all are oriented *lower is better*. Notations that are not in the table were defined in Section 1.

Metric	Definition	Notations
Aspect ratio*	$1-\min_\theta \frac{\min(w_\theta,h_\theta)}{\max(w_\theta,h_\theta)}$	$\theta \in \{\frac{2\pi k}{N}, k \in [0,1,...,N-1]\}$ w_θ and h_θ layout width and height after rotation θ
Angular resolution*	$1-\frac{\min_{(i,j),(j,k)\in E}\theta_{ijk}}{\theta_G}$	$\theta_G=\frac{2\pi}{d_{max}}$, optimal angle; d_{max}, maximum degree
Edge crossings number	$\sum_{e1,e2\in E,e1\neq e2}\mathbb{1}\{hasCrossing(e1,e2)\}$	*hasCrossing*, geometric function testing if two segments intersect according to the source and target node positions of the edge parameters.
Cluster overlap	$\sum_{i\in V}\frac{\sum_{u\in U_i}(1-\lvert\lvert X_u-X_i\rvert\rvert)*\mathbb{1}\{MCL_i\neq MCL_u\}}{\sum_{u\in U_i}(1-\lvert\lvert X_u-X_i\rvert\rvert)}$	$r=0.2$; U_i, set of nodes v_u for which $\lvert\lvert X_i-X_u\rvert\rvert<r$; MCL_i, the cluster of node i
Neighborhood preservation*	$1-\frac{1}{\lvert V\rvert}\sum_{i\in V}\frac{\lvert U_i\cap Y_i\rvert}{\lvert U_i\cup Y_i\rvert}$	U_i, k-hop-neighborhood of v_i in the theoretical space; Y_i set of $\lvert U_i\rvert$ neareast nodes in the projected space.
Stress	$\frac{1}{N}\sum_{i,j\in V}w_{ij}(\lvert\lvert X_i-X_j\rvert\rvert-\delta_{ij})^2$	$w_{ij}=\delta_{ij}^{-2}$

performance between layout techniques is significantly different, we first conduct a Friedman test [52] to verify if changes in the condition variable (*i.e.*, layout technique) significantly affect the metric results, with an acceptance threshold of $\alpha = 0.05$. If so, post-hoc pairwise comparison tests (Nemenyi [53]) are conducted, also with $\alpha = 0.05$. For every pair of layout techniques that passes the test, we can safely assert that the difference between their metric results is significant. Pairwise significant differences can be observed even with close means and standard deviations if the two techniques do not perform well on the same graphs. Since all metrics are oriented *lower is better*, a significant difference is in favor of the technique with the lowest mean value.

4.4 Dataset Heterogeneity and KL Optimization

The importance of training dataset heterogeneity is crucial in Deep Learning techniques. Because the model learns to optimize a cost function for a whole dataset, it may overfit if the dataset is too homogeneous, and not be able to learn if the dataset is too heterogeneous. It is then important to keep balance between enough homogeneity so that the model can recognize patterns in the data to learn the task, while keeping enough heterogeneity so that it can still generalize to unseen data.

This section presents the comparison of *(DNN)*^{2} performance trained on two different sets whose main difference is the heterogeneity: HoR and HeR (see Section 4.1). The models are then evaluated on the Rome dataset since we want to compare their performance and capability to generalize to other graph structures. Table 1 shows the differences of these three sets regarding distances and densities distribution. The two first *(DNN)*^{2} models in this section optimize the C_{tsNET} (5) loss with λ weights set according to the original tsNET implementation [1], [19]. The goal is to study how the models are affected by heterogeneity in their training dataset. In addition, we compare them to a third *(DNN)*^{2} instance trained to optimize C_{KL} (6) on HeR. Since we do not need to speed up the convergence of any gradient descent, we want to observe how a model specifically dedicated to C_{KL} optimization performs compared to models that optimize C_{tsNET} including additional terms (*i.e.*, compression and repulsion). We only do this second comparison on the HeR dataset as we will see that it leads the model toward significantly better performance.

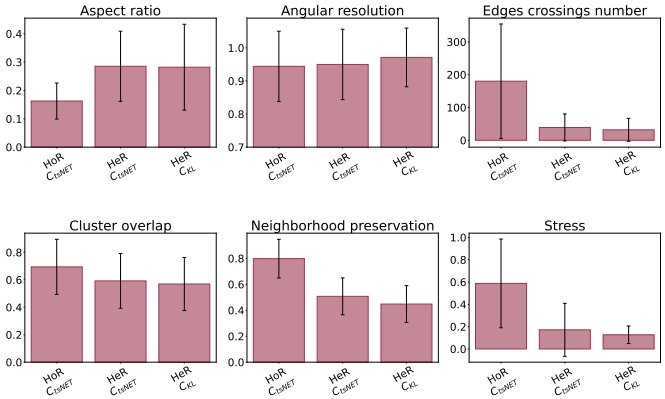


Fig. 5: Performance of *(DNN)*^{2} models trained on HeR or HoR dataset and evaluated on Rome. The two first models optimize C_{tsNET} (5) while the last optimizes C_{KL} (6). All statistical test passed, meaning that the pairwise difference between the three model performances on every metric is significant.

The performance of these three models on the Rome test set is reported in Figure 5. All the statistical tests (see Section 4.3) passed, meaning that all the models have significantly different performances on every metric. First, we compare *(DNN)*^{2} trained on HoR and HeR respectively. The HoR-C_{tsNET} instance is only better on Aspect Ratio and Angular Resolution (by a short margin). On all other metrics, HeR-C_{tsNET} is far better. This confirms that having a more heterogeneous training dataset improves the model performance and capability to generalize.

The two last bars of each plot in Figure 5 compare *(DNN)*^{2} models trained on HeR to optimize C_{tsNET} and C_{KL}. Except on Angular Resolution, the model that optimizes C_{KL} (*i.e.*, HeR-C_{KL}) always performs better. In addition, we can be certain HeR-C_{KL} optimized neighborhood preservation, while it cannot be sure with HeR-C_{tsNET} since C_{tsNET} requires to optimize two additional terms. Optimizing C_{KL} seems to make more sense with a Deep Learning approach since we do not need to speed up the learning convergence, the model reaches better performance and we know exactly what it tries to optimize (which improves results

understanding and further re-design). Since explainability is a major concern with Deep Learning techniques, the advantages of the latter point cannot be neglected.

In the prior version of the study [1], it was claimed that finetuning models improved their performance. This conclusion was drawn after comparison with a generic model trained on HoR. Yet, as we just saw, a model trained on HeR has a significantly greater capability to generalize to unseen data. Hence, it is not certain that finetuning models would be beneficial.

At the same time, it seems intuitive that no matter what a model has been pretrained on, it will benefit from being finetuned on specific datasets. In fact, graphs from a specific family (*e.g.*, grids, trees) have common and typical internal structures we call *topological patterns*. We believe that a model trained on a dataset only made of graphs from a specific family can learn to leverage these topological patterns to achieve better drawings.

These two hypotheses echo layout algorithm approaches from the literature (see Section 2.1) where we can oppose algorithms dedicated to specific graph families with generic ones. Dedicated algorithms assume that the graph follows some properties and leverages them to produce a layout. On the other hand, generic methods generally aim at optimizing an energy function, whatever the properties of the graph are. With the Deep Learning approach, the models learn by themselves how to extract features from the graph. Hence, the same architecture can learn either to layout generic graphs (*i.e.*, with no common graph topological pattern) or graphs from the same family (*i.e.*, that share topological patterns), based on the dataset it is trained on. The next section investigates how *(DNN)*^{2} Deep Learning approach fits to the two categories.

4.5 Specific Graph Families

This section presents the evaluation of three *(DNN)*^{2} training approaches on specific *graph families*: Trees, Grids, Nearest-Neighhbor (NN), Planars and Community (Commu.) graphs; each being represented by a dataset presented in Section 4.1. These graph families were selected as they cover a large variety of topologies. The first hypothesis studied in this section is that training *(DNN)*^{2} on a specific graph family leads to better layouts of that family than training *(DNN)*^{2} on random graphs. This hypothesis is motivated by the assumption that Deep Learning models should learn to recognize and leverage topological patterns to produce better layouts. This study will also clarify whether *(DNN)*^{2} is more prone to fit in the category of generic algorithms or those who are dedicated to specific topologies. Finally, it will also extend the comparison of pretraining, finetuning or training *(DNN)*^{2} fromscratch presented in [1].

The other question this section aims at answering is: does *(DNN)*^{2} benefit from having an existing layout as input? Or more generally, meaningful input node features? Many layout algorithms initialize node positions with fast techniques rather than randomly (*e.g.*, tsNET [19]), before conducing their own optimization. Since *(DNN)*^{2} directly infers graph layouts, there is no such thing as node position initialization. Instead, we provide the 2D positions of a fast layout technique, PivotMDS [12], to the model input node features (see Section 3.4). That way, we can verify (i) that the model benefits from having knowledge of another layout; and (ii) that adding meaningful features to the input node feature vectors improves the produced layouts quality.

Hence, for each graph family, the performances of 6 instances of *(DNN)*^{2} that optimize C_{KL} (6) are compared: the three approaches pretrained, finetuned (Ftune) and fromscratch (Fscratch), each trained twice: with and without PivotMDS (PMDS) input features. For all the studied families, the Pretrained instance is the HeR-C_{KL} of Section 4.4 which was trained on the dataset of Heterogeneous Random graphs (see Section 4.1). Fscratch instances of a family are trained on their corresponding dataset with random learnable weights initialization. Ftune instances are trained as Fscratch ones, but learnable weights are initialized to Pretrained instance weights.

4.5.1 Results Reports

For each dataset and metric, the mean performance and corresponding standard deviation of the 6 models are reported on one barplot (see Section 4.3). Each barplot reports the mean and standard deviation value of its corresponding metric for the 6 models. In addition, 4 graphs (with $N \approx \{10, 50, 75, 100\}$) from the *test* set of the graph family are drawn and presented.

Significance in barplots: For each barplot in the next sections, the statistical validation protocol defined in Section 4.3 is applied. At first, a Friedman test was conducted and rejected the null hypothesis, meaning that the differences between the drawing techniques performances were significant. Since the differences are not due to randomness, further Nemenyi pairwise tests are conducted. Originally, bars are colored blue and the significant difference between two model instance performances is encoded with an arc between their label. To alleviate plots, model instances whose performance is different to all other ones are colored red and their incident arcs are removed. For example, in Figure 6a Aspect Ratio plot, both Pretrain and Pretrain PMDS instance performances are significantly different to all others. On the other hand, Ftune instance performance is only different from instances it has an arc with (*i.e.*, Ftune PMDS and Fscratch PMDS) and the red bars (*i.e.*, Pretrain and Pretrain PMDS). It means the difference is *not* significant between Ftune and Fscratch on that metric.

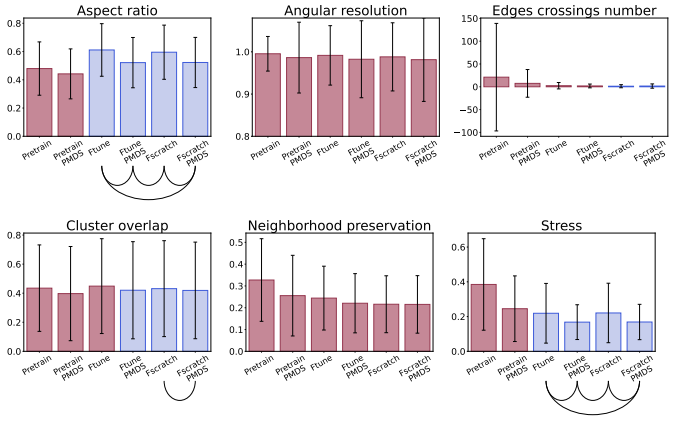
4.5.2 Trees

The performances of Pretrain, Ftune and Fscratch model instances with and without PMDS are reported in Figure 6a and visual examples are proposed in Figure 6b. We can see that learning from a Trees dataset (*i.e.*, Ftune and Fscratch instances) leads the model toward better Stress, Edge crossing number and Neighborhood preservation, while it has a smaller effect on Angular resolution and Cluster overlap. As there is almost no difference between Ftune and Fscratch instances, using the prior knowledge of the pretrained instance does not help to achieve better layouts. The visual aspects of graph samples in Figure 6b confirms that it is very beneficial for the model to be trained specifically on trees if it aims at drawing them. To conclude, transfer learning from random graphs knowledge only has very little interest.

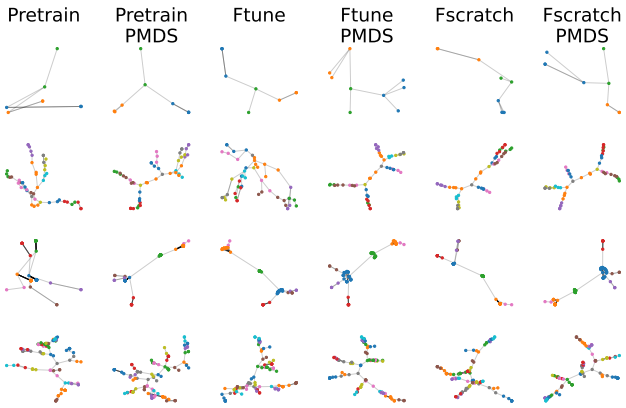
As for having PMDS positions as input node features, we can see it consistently improves the layout quality according to all the metrics. Visually, PMDS seems to improve Angular Resolution and avoid the overplot of some leaves.

4.5.3 Grids

Performance on Grid graphs is reported in Figure 7a with visual examples in Figure 7b. Training the models on the specific Grids dataset significantly improves its Neighborhood Preservation, Edge crossing number and Stress aesthetics. Overall, the Ftune instance underperforms, while Ftune PMDS is best in terms of aesthetics. Visually, Pretrain is the only instance to draw the small grid well. On the other hand, it folds larger ones on themselves if it does



(a) Comparison of Pretrain, Ftune and Fscratch models with and without PMDS input feature on Tree graphs. Barplots report mean and standard deviation of their respective aesthetic metric. Bars color and arcs encode pairwise significance (see Section 4.5.1).

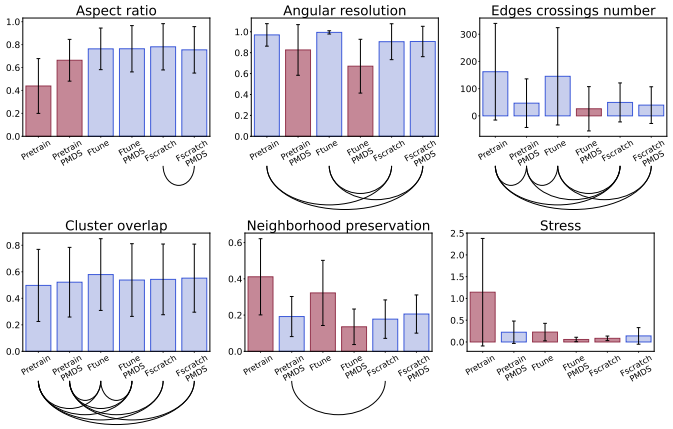


(b) Trees drawings examples with $N \approx \{10, 50, 75, 100\}$.

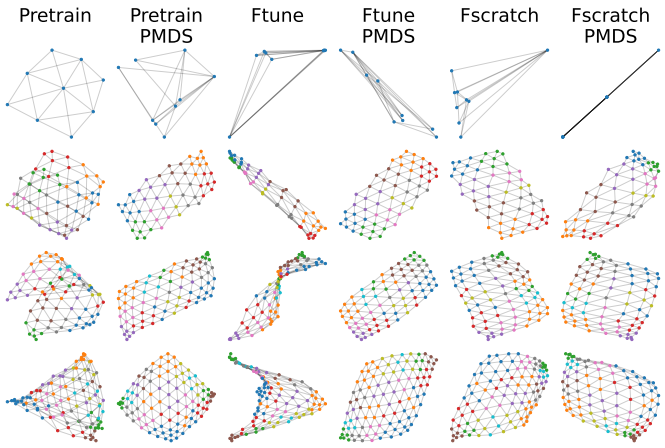
Fig. 6: Pretrain, Ftune and Fscratch instances (a) aesthetics and (b) layout samples on **Tree** graphs with and without PMDS input features.

not have PMDS input features. On the other hand, drawings of instances trained on the Grid dataset are satisfactory on larger graphs except for the Ftune instance. We can see that Fscratch succeeds at laying out the grids even without PMDS input features. Since Grid graphs have a very specific topology, we expected that models trained on Grid would be more efficient than Pretrain.

Having PMDS as input features significantly improved Ftune PMDS instance performance, but deteriorates Fscratch ones. Both visually and in terms of aesthetics, Ftune PMDS is better than Ftune, while Fscratch PMDS is worse than Fscratch. Having PMDS features was expected to be beneficial for the models considering that PivotMDS is a layout algorithm well suited to grids. Hence, it probably means that the transfer learning helps Ftune PMDS model to rely on the PMDS features, while it provides nothing to the Ftune instance. As for Fscratch, there is not much difference between having PMDS input features or not. The only significant differences are a better Aspect Ratio with PMDS input features, but a better Stress without them. Both Fscratch instances drawings on larger grids are satisfactory and show that it could leverage grids topological patterns to achieve better layouts than Pretrain.



(a) Comparison of the 6 model instances on Grid graphs. Barplots are detailed in Section 4.5.1).



(b) Grids drawings examples with $N \approx \{10, 50, 75, 100\}$.

Fig. 7: Pretrain, Ftune and Fscratch instances (a) aesthetic metrics and (b) layout samples on **Grid** graphs.

4.5.4 *Nearest Neighbor (NN) graphs*

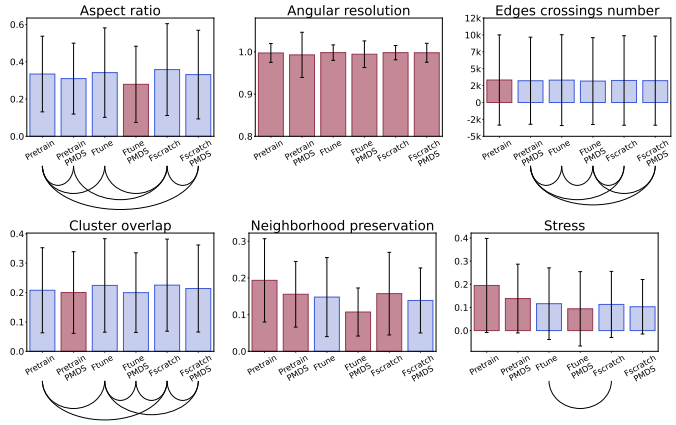
Figure 8a and 8b present respectively the models performance and some visual examples of the models instances on the Nearest Neighbor (NN) dataset. We can see that training the models on specific NN graphs (*i.e.*, Ftune and Fscratch instances) only positively affects Neighborhood Preservation and Stress aesthetics. Again, Ftune PMDS instance is the best performing overall and Ftune instance is better than Pretrain and Fscratch. Visually, all the Ftune and Fscratch layouts are satisfactory, while Pretrain instances fail at laying out larger graphs.

Having PMDS as input features improves Aspect Ratio, Cluster overlap, Neighborhood Preservation and Stress aesthetics. Yet, this aesthetic gain is not visually relevant as we do not observe much difference between layouts from instances with or without PMDS.

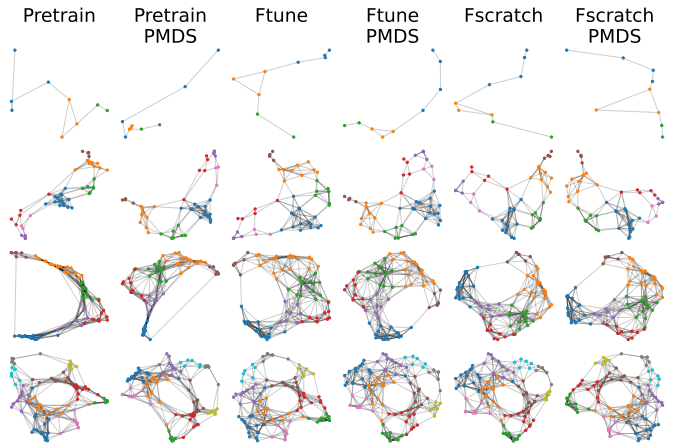
4.5.5 *Planars*

The models performance and visual examples on Planar graphs are presented in Figure 9a and 9b respectively. As opposed to planar drawing algorithms from the literature that are formally proven to produce crossings-free layouts (*e.g.*, [7]), *(DNN)*^{2} does not guarantee that its produced layouts do not have edge crossings.

According to aesthetics, there is no strong benefits in learning specifically from Planar graphs to lay them out well. In fact, Pretrain

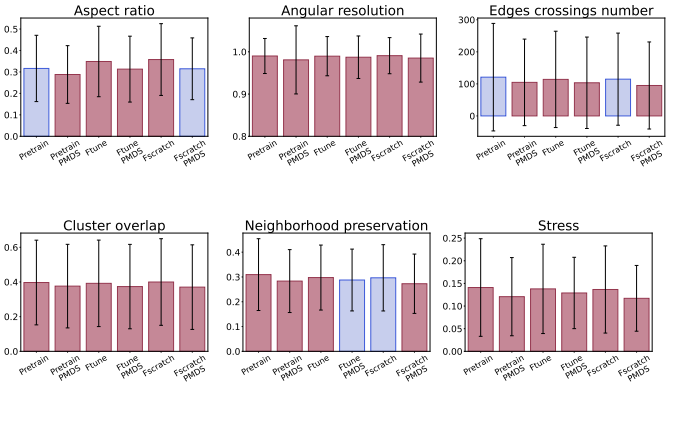


(a) Comparison of the 6 model instances on NN graphs. Barplots are detailed in Section 4.5.1).

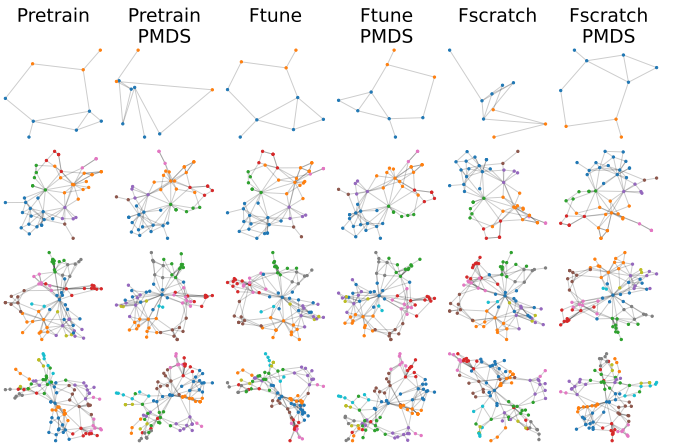


(b) NN graphs drawings examples with $N \approx \{10, 50, 75, 100\}$.

Fig. 8: Pretrain, Ftune and Fscratch instances (a) aesthetic metrics and (b) layout samples on **NN** graphs.



(a) Comparison of the 6 model instances on Planar graphs. Barplots are detailed in Section 4.5.1).



(b) Planar drawings examples with $N \approx \{10, 50, 75, 100\}$.

Fig. 9: Pretrain, Ftune and Fscratch instances (a) aesthetic metrics and (b) layout samples on **Planar** graphs.

layouts are visually similar to Ftune and Fscratch ones, if not better. Hence, despite small improvements in the Edge crossing number, the model was not able to learn topological patterns dedicated to planar graphs to improve their layouts.

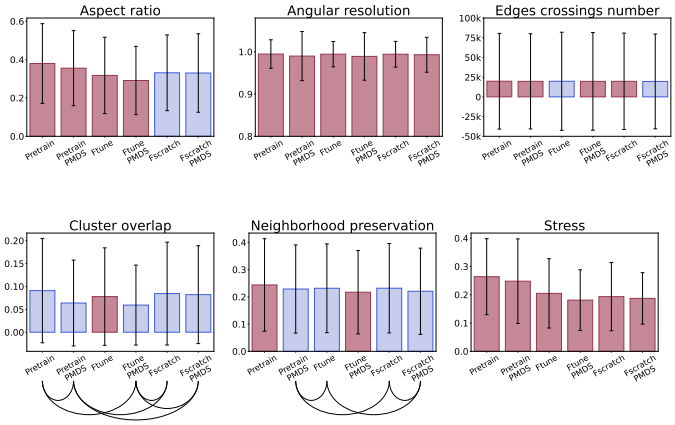
Adding PMDS input features slightly improves all aesthetics but does not seem to improve layouts drawings that much.

4.5.6 *Communities*

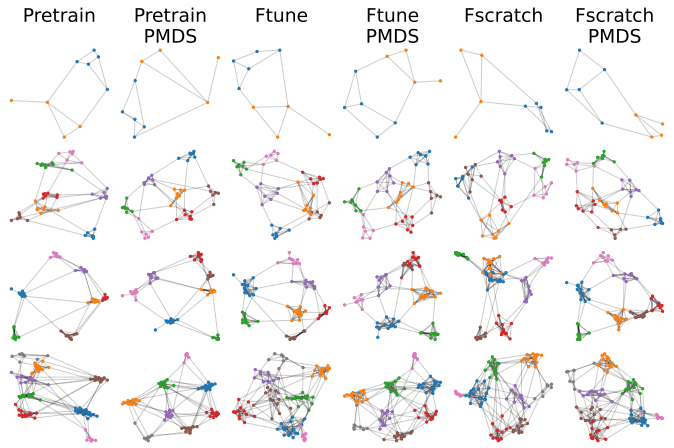
Communities (Commu.) is the last specific dataset we study in this Section. The performance and examples of layouts on this dataset are reported in Figure 10a and 10b. In terms of aesthetics, training on this specific family improves the layouts Aspect ratio, Cluster overlap and Stress, but have almost no effect on other metrics. However, Cluster overlap is probably the most meaningful metric for Communities graphs. In fact, we can see on the visual examples that all the methods are able to identify communities and organize the layout in the same way. The main difference between the Pretrain model that has only seen Random graphs and the Ftune-Fscratch instances is their capability to represent intra-cluster structures. While Pretrain almost completely overplots nodes of the same community, Ftune and Fscratch better emphasize the intra-community structures.

Having PMDS as input features improves Pretrain and Ftune results, while it has a smaller positive effect on Fscratch. Visually, the layouts of models with PMDS features are not much improved compared to their without-PMDS counterparts.

To conclude, this section aimed at studying two hypotheses. First, that training model instances on specific graph families makes them better at laying out graphs of that family by learning specific topological patterns. This hypothesis also studies whether the framework is better at laying out generic graphs, or tuned for specific graph topologies. Overall, training on a specific dataset does improve the models capability to layout the graph family it is trained on. On graph families with a very specific topology (*i.e.*, Trees, Grids), training the model on samples of that family is almost mandatory to achieve good layouts. On the other families, the difference is less pronounced. On one hand, we would like to remind that the Pretrain and Pretrain PMDS instances used in this section were the same accross all the datasets, and its results are overall pleasing despite some problematic cases. On the other hand, Ftune and Fscratch instances constantly achieve good layouts, valuing the training on specific topologies. In the end, the Pretrain model is not optimal in terms of pure aesthetics but it is a good generic model.

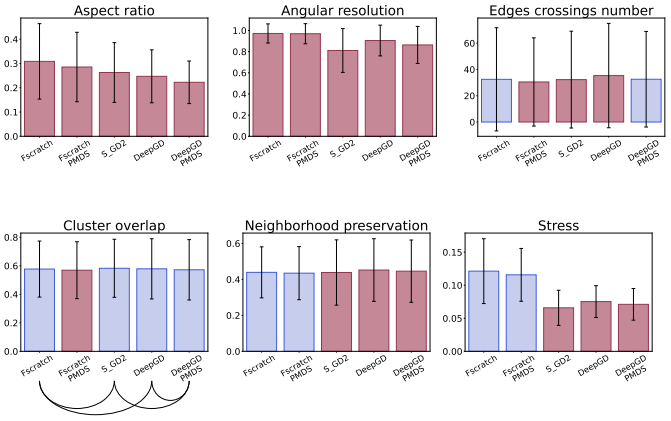


(a) Comparison of the 6 model instances on Commu graphs. Barplots are detailed in Section 4.5.1).

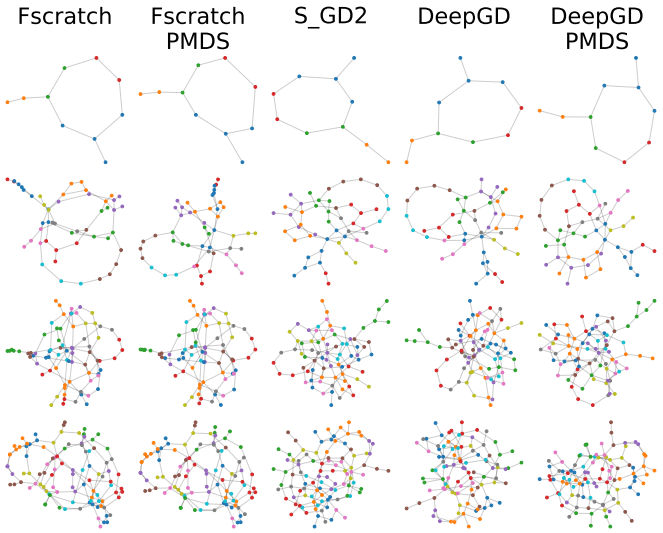


(b) Commu. drawings examples with $N \approx \{10, 50, 75, 100\}$.

Fig. 10: Pretrain, Ftune and Fscratch instances (a) aesthetic metrics and (b) layout samples on **Commu** graphs.



(a) Comparison of $(DNN)^2$ Fscratch, S_GD^2 and *DeepGD* with and without PMDS input feature on Rome graphs. Barplots report mean and standard deviation of their respective aesthetic metric. Bars color and arcs encode pairwise significance (see Section 4.5.1).



(b) Rome drawings examples with $N \approx \{10, 50, 75, 100\}$.

Fig. 11: $(DNN)^2$ Fscratch, S_GD^2 and *DeepGD* (a) aesthetic metrics and (b) layout samples on **Rome** graphs.

The second hypothesis was that adding the 2D positions of an algorithms from the literature as *meaningful* input node features would help the model to achieve better layouts. Transfer learning consistently leads to better results for instances with PMDS input features, meaning that it improves the feature extraction stage of the model architecture, while the regression is more sensitive to the *train* dataset graph topologies. On the contrary, even though Fscratch PMDS performs better than Fscratch on most families, the difference is less significant. At the very least, as long as the PivotMDS node positions are different (*i.e.*, there is no overplot), adding PMDS features helps the model to differentiate nodes that have a local isomorphic neighborhood, thus overcoming a typical flaw of Graph Convolutional Networks (as seen in Section 2.2).

4.6 Comparison to literature algorithms

In this section, we compare $(DNN)^2$ with two methods from the literature, S_GD^2 [22] and *DeepGD* [43]. S_GD^2 is an efficient algorithm that leverages stochastic gradient descent for each graph drawing to optimize stress modeled with independent constraints between pairs of nodes. On the other hand, *DeepGD* is a Deep Neural Network approach to Graph Drawing and probably the closest framework to $(DNN)^2$ (see Section 2.3).

The performance and visual examples of the methods on the Rome *test* graphs are presented in Figure 11a and 11b respectively. Fscratch instances refer to the results of $(DNN)^2$ trained on the Rome *train* dataset to optimize C_{KL} (6). DeepGD instances are trained for 1000 epochs to optimize *Stress* with random weights initialization on the Rome *train* graphs, with and without PivotMDS node positions in their input.

Regarding aesthetic metrics, $(DNN)^2$ Fscratch instances are slightly better on Edges Crossing Number, Cluster Overlap and Neighborhood Preservation. The better performance on the last two can be explained by $(DNN)^2$ cost function C_{KL} that optimizes neighborhood preservation whereas stress focuses on distance preservation. On the other hand, $(DNN)^2$ Fscratch is not as good on Aspect Ratio, Angular Resolution and Stress (which is directly optimized by both S_GD^2 and *DeepGD*).

In the visual examples, we can see that all layouts are satisfactory, but the $(DNN)^2$ Fscratch ones are less pleasing. Indeed, the edge length between nodes of a sparse region is quite high,

whereas it is very small in dense regions. In fact, the drawings tend to emphasize inter-community structures representation at the cost of the intra-community representation. Though C_{KL} focuses on neighborhood preservation, this is a problematic behavior we observed several times. By design, C_{KL} assumes that the distances distribution around each node follows a Gaussian centered on 0. However, this assumption often leads to the deterioration of close neighborhood preservation, as we will detail in Section 5.1.

To conclude, *(DNN)²* and *DeepGD* Deep Learning (DL) approaches do not produce better layouts than recent algorithms from the literature yet. However, they are steps toward efficient DL for Graph Drawing techniques that could soon outperform the state-of-the-art, as DL already revolutionized other research fields.

5 Discussion

This section addresses some limiting aspects that were observed in the previous sections about *(DNN)²* framework or its implementation. In a first part, we detail the limitations dedicated to the loss function used in the evaluations, while the second part comes back on several limitations relative to DL4GD in general.

5.1 KL optimization for Graph Drawing

In this section, we discuss the limitations of optimizing C_{KL}, the Kullback-Leibler divergence cost function defined by Kruiger *et al.* [19]. The loss function design was described in Section 3.3, but we detail all the formula in the following:

$$C_{KL} = \sum_{i,j \in V, i \neq j} p_{ij} \log \frac{p_{ij}}{q_{ij}} \tag{6}$$

where p_{ij} (7) and q_{ij} (9) are probabilities that every pair of nodes in the graph are connected. Basically, p_{ij} is computed from shortest path lengths in the graph, while q_{ij} is computed from euclidean distances in the projected space. Thus, a model that generates layouts to optimize C_{KL} will look for node positions which, once turned into probabilities q_{ij}, match the theoretical probabilities p_{ij}.

$$p_{ij} = p_{ji} = \frac{p_{i|j} + p_{j|i}}{2N}, p_{ii} = 0 \tag{7}$$

where $p_{j|i}$ is defined in Equation 8

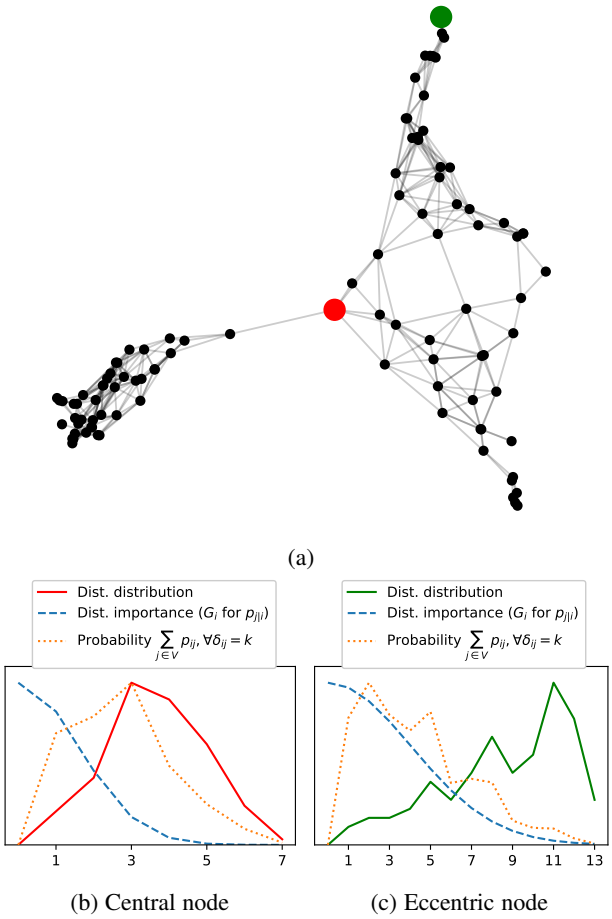
$$p_{j|i} = \exp\left(-\frac{\delta_{ij}^2}{2\sigma_i^2}\right) / \sum_{k \in V, k \neq i} \exp\left(-\frac{\delta_{ik}^2}{2\sigma_i^2}\right), p_{i|i} = 0 \tag{8}$$

where σ_i is found by binary search so that the perplexity $\kappa_i = 2^{\sum_{j \in V} p_{j|i} \log_2 p_{j|i}}$ matches a given value.

Finally, q_{ij} is defined as:

$$q_{ij} = q_{ji} = \frac{(1 + ||X_i - X_j||^2)^{-1}}{\sum_{k,l \in V k \neq l} (1 + ||X_k - X_l||^2)^{-1}}, q_{ii} = 0 \tag{9}$$

The first concern about C_{KL} optimization was raised by a fisheye effect we observed in *(DNN)²* drawings. The graphs central parts are correctly drawn, while peripheral parts are flattened. This can typically be observed in Figure 12a and in the Grid graphs drawings (see Figure 7b) where the corners are contracted. This fisheye distortion was already observed by Kruiger *et al.* [19] but they did not really relate it with C_{KL}. We believe this behavior comes from the combination of two limitations of optimizing a t-SNE [20] based cost function for graph drawing. The second concern regards Deep Learning model capabilities to optimize such a function as we will see that it is intrinsically related with each data sample. In the next, we detail why these concerns are problematic in the context of DL4GD.



(b) Central node (c) Eccentric node

Fig. 12: (a) NN graph drawing by *(DNN)²* Ftune on NN graphs with PMDS features. (b) (resp. (c)) presents the distances distribution centered on a central node (resp. eccentric node) v_i, the Gaussian G_i of distances importance centered on v_i (*i.e.*, Equation 8 numerator) and the resulting sum of probabilities p_{ij} (7) around v_i for each distance k. The orange curve is an aggregation that represents what a model optimizing C_{KL} would converge toward in terms of distance preservation around node v_i. Y-scales are different for the three curves and were adapted to emphasize trends.

As defined by Kruiger *et al.* [19], C_{KL} is an adaptation of the t-SNE [20] algorithm to a graph context. t-SNE works by turning highly-dimensional distances between datapoints into probabilities modeling their (dis)similarities. The probability that two points x_i and x_j are similar is picked assuming that x_i probability to be similar with any other point follows a Gaussian G_i centered on x_i. More specifically, the probability that x_i andx_j are neighbors is relative to $G_i(\delta_{ij})$ and $G_j(\delta_{ij})$ where δ_{ij} is the distance between x_i and x_j in a high-dimensional space.

The adaptation to a graph context is done by considering *nodes* instead of *datapoints* and setting d_{ij} to the shortest path length between nodes v_i and v_j (*i.e.*, δ_{ij}). For each node v_i, the Graph Drawing view of $G_i(k)$ can be interpreted as "how well should the k-hop-neighborhood of v_i be preserved in the layout". By centering

each Gaussian G_i on 0, the highest importance is given to distances $k = 1$ and the importance of distances $k > 1$ decreases according to the Gaussian slope (*i.e.*, relative to the standard deviation), see the blue curves in Figure 12b and 12c.

As mentioned earlier, we believe that the first concern (*i.e.*, fisheye effect) is raised by two limitations related to the function design. The first one comes from the way G_i standard deviation σ_i is picked. As defined in Equation 8, σ_i is found by binary search so that central (resp. eccentric) nodes get low (resp. high) values. Overall, this tells the model that higher distances are more important to preserve around eccentric nodes than around central ones. However, around eccentric nodes, higher distances are also more numerous (see the distances distribution in Figure 12c). Hence, increasing σ_i comes down to giving more weight to more numerous higher distances. In the end, the probability $p_{j|i}$ (8) is a ratio of $G_i(\delta_{ij})$ with $\sum_{k \in V} G_i(\delta_{ik})$. Thus, increasing σ_i reduces all the probabilities associated with all distances around eccentric nodes. In the example provided in Figure 12, the sum of probabilities p_{ij} (*i.e.*, sum of orange curve values) around the central node is 0.014 against 0.007 around the eccentric node. If we look back at the C_{KL} (6) formula, p_{ij} is also a weight factor of the ratio between p_{ij} and q_{ij} logarithm. It means that the preservation of distances around the eccentric node are globally weighted two times less than around the central node. In an ideal case, that importance should be $\frac{1}{N}$ for all nodes, *i.e.*, 0.01 in the Figure 12 graph example that has 100 nodes. In addition, giving more importance to more numerous higher distances also breaks the original will to make distances $k = 1$ the most important, and distances $k > 1$ importance decrease according to the G_i slope. For example, with the central node in Figure 12b, we can se that although the most important distance should be $k = 1$ (*cf.*, the blue curve G_i), the probabilities associated with $k = 2$ and $k = 3$ are higher (*cf.*, orange curve). Eventually, p_{ij} trends of both the central and the eccentric nodes are affected by the distances distribution centered on them, which leads us toward our second concern.

The second concern regards the capability of a Deep Learning (DL) model to optimize C_{KL}. As opposed to standard optimization methods that work on a single sample at a time, a DL model is trained to optimize the function on a dataset of samples. Hence it has to find a strategy (*i.e.*, combination of weights) that can be applied to *any* input graph and that optimizes the cost function overall. According to what we studied for the first concern, we see at least two risks that a DL model would fail at correctly laying out graphs by optimizing C_{KL}. The first risk comes from the different overall weight that is given to central *vs.* eccentric nodes. As observed in Figure 12 example, the sum of p_{ij} was 0.014 for the central node against 0.007 for the eccentric one, making the eccentric node weight less than the central one in the p_{ij} matrix of probabilities. In regards to DL techniques training behaviors, there is a high chance that a DL model strategy to optimize C_{KL} on any graph would ignore most eccentred nodes since they weight significantly less in the layout evaluation. This could explain the fisheye effect observed, *i.e.*, why central nodes are more correctly drawn than eccentric ones. The second risk we identified was also raised by the study of the first concern. As mentioned earlier, the consequences of increasing σ_i as nodes get eccentric breaks the initial assumption that importance of distance preservation around a node v_i follows a gaussian centered on 0 and makes p_{ij} probabilities sensitive to the distances distribution around v_i. This behavior makes the evaluation of every node position depend on the distances distribution in the graph itself. Thus, it becomes arduous for a DL model to find a generic strategy that produces layouts optimizing the function on a dataset of various graphs, since each sample is evaluated according to different criteria. In this regard, we can oppose C_{KL} with, for instance, the optimization of Stress (see Table 2) which completely sets aside the notion of *graph* and *distances distribution* to only evaluate the differences between distances in two spaces (*i.e.*, projected and high-dimension).

Visually, the mentioned concerns lead to overplots between eccentric nodes as it is clearly observable in Figure 12a where the center of the graph is correctly drawn, while the three eccentric parts suffer from overplots. It can also be observed with Grid graph drawings (see Figure 7b) where grids corners are contracted. This effect is part of the causes of the *fisheye distortion* mentioned by Kruiger *et al.* [19] and which they counter balance by adding a repulsion post-process (see Equation 5).

5.2 Limitations

Our evaluation has shown that *(DNN)*^{2} is able to learn from graph data to produce layouts without ground truth by optimizing a Kullback-Leibler (KL) divergence based function. Though some flaws of the approach have been discussed throughout the evaluation, there remain some that we discuss in this section.

In the previous Section 5.1, we have seen that the node overplots in *(DNN)*^{2} drawings are mainly due to the KL loss design. However, we cannot ignore that *(DNN)*^{2} is also built with Graph Convolutions which are known to produce overplots. As mentioned in prior works [1], [30], [31] and in Section 2.2, they hardly distinguish nodes that have the same local neighborhood (*i.e.*, locally isomorphic) and tend to produce the same embeddings for them. Even though the loss function penalizes these overplots, the model architecture does not have many tools to learn how to differentiate locally isomorphic nodes. For instance, we set a unique *id* to each node in its initial feature vector (see Section 3.4) to guide the model toward an extraction of features that preserves each node identity, but we cannot be certain it was sufficient.

Another limitation is that the measured performance of *(DNN)*^{2} is sensitive to their training setup. During the experimentation phase, we fixed some design choices even though we expected them to be suboptimal (*e.g.*, using Graph Convolutions, *KL* loss, minimalist input features) to limit black box effects and be able to relate shifts in performance to specifically tuned properties. Yet, some Deep Learning related hyperparameters were found by a trial and error process, which led to a common problematic in Deep Learning. Sometimes, tweaking hyperparameters (*e.g.*, divide learning rate by 10) have more effects than changing some structures that we could expect to strongly impact the model behavior. Obviously, we presented *(DNN)*^{2} results with the hyperparameters that led to the best results we explored, but we cannot be certain that our setup is the most efficient. Though Deep Learning is a promising approach to Graph Drawing, it also comes with its own limits such as the difficult understanding of the models behaviors. The good side of this limitation is that the tuning of the model architecture with the selection of hyperparameters is done *a priori*, before the model training. In a Deep Learning approach, all the parametrization heavy-lifting is done by an expert before the training, as opposed to many algorithms that require the end-user to set various parameters (which optimal value depend on the graph).

Finally, *(DNN)*^{2} capability to generalize to various graph sizes is complicated. Execution time is a major advantage since Deep

Neural Networks architectures such as *(DNN)*^{2} sequentially apply polynomial transformations in shape of matrices to their input, which is almost instantaneous on a GPU. However, training the model on graphs of significantly different sizes becomes a challenge as it increases the possible variations of all other properties and makes it harder for the model to find a generic strategy to optimize a given aesthetic. That raises again the question of how to build a *train* dataset that enables a model to produce layouts optimizing the desired loss function while maintaining a good capacity to layout unseen samples. On the other hand, the notion of *train-validation-test* datasets split can be questioned for some graph families. For example, with *Cliques* or *Grids*, since it is possible to enumerate all the graphs up to a given size, the concept of generalization to *unseen* samples (*i.e.*, the need for *validation*) might not be suited. Overall, this paper has studied some dataset properties effects on *(DNN)*^{2} performance, but more research is necessary to understand how to correctly build a dataset to train a Deep Neural Network to layout general graphs.

6 Conclusion

This paper has presented *(DNN)*^{2}, a Deep Learning approach to graph drawing. It proposes to leverage efficient Deep Neural Network architectures in a context of graphs through Graph Neural Networks. In this paper, we evaluate the framework by adapting a state-of-the-art Convolutional Neural Network to a graph context through Graph Convolutions. The model learns by itself how to extract features from the graph and then projects it into a layout. The produced layout optimizes a loss function without any notion of *ground truth* layout (*i.e.*, unsupervised). Throughout several evaluations, we demonstrated the strengths and limitations of Deep Learning for Graph Drawing (DL4GD).

Research in the DL4GD thematic have already started with the emergence of several papers during the 2021 summer and future work leads to study are numerous. Regarding the *(DNN)*^{2} implementation that was used in our evaluations, the overall setup can be improved. We fixed some potentially suboptimal design choices to limit black box effects and enable to relate the statistical performance shifts to monitored conditions. Hence, the implementation can be improved by searching for more optimal hyperparameters, using node embedding techniques for the input node feature representation, changing the loss function, the model architecture, the data structures, etc. The main concern with this lead is to solve the generalizability issues of the model architecture by making it less sensitive to topologies or graph size variations outside the scope of its *train* dataset.

Finally, other leads for future work would be to design other DL4GD frameworks. For example, since we have seen that the model learned differently based on the graph topologies of its *train* dataset, it would be interesting to design an *aggregator* meta-model that learns to merge the drawings of *drawer* models, themselves trained on a different graph family each. That way, the framework would leverage the topology-dedicated feature extraction of specialized models, while maintaining a good capacity to adapt to unseen graphs.

Acknowledgments

We would like to thank X. Wang from the Ohio State University for sharing the original implementation of DeepGD framework [43].

References

[1] Giovannangeli, Loann and Lalanne, Frederic and Auber, David and Giot, Romain and Bourqui, Romain, "Deep Neural Network for DrawiNg Networks, (DNN)^2," in *International Symposium on Graph Drawing*. Springer, 2021.

[2] Krizhevsky, Alex and Sutskever, Ilya and Hinton, Geoffrey E, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, pp. 1097–1105, 2012.

[3] Vaswani, Ashish and Shazeer, Noam and Parmar, Niki and Uszkoreit, Jakob and Jones, Llion and Gomez, Aidan N and Kaiser, Łukasz and Polosukhin, Illia, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[4] Xie, Yu and Yao, Chuanyu and Gong, Maoguo and Chen, Cheng and Qin, A Kai, "Graph convolutional networks with multi-level coarsening for graph classification," *Knowledge-Based Systems*, vol. 194, p. 105578, 2020.

[5] Abu-El-Haija, Sami and Kapoor, Amol and Perozzi, Bryan and Lee, Joonseok, "N-gcn: Multi-scale graph convolution for semi-supervised node classification," in *uncertainty in artificial intelligence*. PMLR, 2020, pp. 841–851.

[6] Kim, Jongmin and Kim, Taesup and Kim, Sungwoong and Yoo, Chang D, "Edge-labeling graph neural network for few-shot learning," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2019, pp. 11–20.

[7] Gutwenger, Carsten and Mutzel, Petra, "Planar polyline drawings with good angular resolution," in *International Symposium on Graph Drawing*. Springer, 1998, pp. 167–182.

[8] Jankun-Kelly, TJ and Ma, Kwan-Liu, "MoireGraphs: Radial focus context visualization and interaction for graphs with visual nodes," in *IEEE Symposium on Information Visualization 2003 (IEEE Cat. No. 03TH8714)*. IEEE, 2003, pp. 59–66.

[9] Kamada, Tomihisa and Kawai, Satoru and others, "An algorithm for drawing general undirected graphs," *Information processing letters*, vol. 31, no. 1, pp. 7–15, 1989.

[10] Frick, Arne and Ludwig, Andreas and Mehldau, Heiko, "A fast adaptive layout algorithm for undirected graphs (extended abstract and system demonstration)," in *International Symposium on Graph Drawing*. Springer, 1994, pp. 388–403.

[11] Meidiana, Amyra and Hong, Seok-Hee and Cai, Shijun and Torkel, Marnijati and Eades, Peter, "Sublinear-Time Attraction Force Computation for Large Complex Graph Drawing," in *2021 IEEE 14th Pacific Visualization Symposium (PacificVis)*. IEEE, 2021, pp. 146–150.

[12] Brandes, Ulrik and Pich, Christian, "Eigensolver methods for progressive multidimensional scaling of large data," in *International Symposium on Graph Drawing*. Springer, 2006, pp. 42–53.

[13] Klimenta, Mirza and Brandes, Ulrik, "Graph drawing by classical multidimensional scaling: new perspectives," in *International Symposium on Graph Drawing*. Springer, 2012, pp. 55–66.

[14] Purchase, Helen C, "Metrics for graph drawing aesthetics," *Journal of Visual Languages & Computing*, vol. 13, no. 5, pp. 501–516, 2002.

[15] Purchase, Helen C and Cohen, Robert F and James, Murray, "Validating graph drawing aesthetics," in *International Symposium on Graph Drawing*. Springer, 1995, pp. 435–446.

[16] Purchase, Helen, "Which aesthetic has the greatest effect on human understanding?" in *International Symposium on Graph Drawing*. Springer, 1997, pp. 248–261.

[17] Ware, Colin and Purchase, Helen and Colpoys, Linda and McGill, Matthew, "Cognitive measurements of graph aesthetics," *Information visualization*, vol. 1, no. 2, pp. 103–110, 2002.

[18] R. Ahmed, F. De Luca, S. Devkota, S. Kobourov, and M. Li, "Graph Drawing via Gradient Descent, (GD)^2," *arXiv preprint arXiv:2008.05584*, 2020.

[19] Kruiger, Johannes F and Rauber, Paulo E and Martins, Rafael M and Kerren, Andreas and Kobourov, Stephen and Telea, Alexandru C, "Graph Layouts by t-SNE," in *Computer Graphics Forum*, vol. 36, no. 3. Wiley Online Library, 2017, pp. 283–294.

[20] Van der Maaten, Laurens and Hinton, Geoffrey, "Visualizing data using t-SNE." *Journal of machine learning research*, vol. 9, no. 11, 2008.

[21] Kwon, Oh-Hyun and Crnovrsanin, Tarik and Ma, Kwan-Liu, "What would a graph look like in this layout? a machine learning approach to large graph visualization," *IEEE transactions on visualization and computer graphics*, vol. 24, no. 1, pp. 478–488, 2017.

[22] Zheng, Jonathan X and Pawar, Samraat and Goodman, Dan FM, "Graph drawing by stochastic gradient descent," *IEEE transactions on visualization and computer graphics*, vol. 25, no. 9, pp. 2738–2748, 2018.

[23] Wang, Qianwen and Chen, Zhutian and Wang, Yong and Qu, Huamin, "Applying Machine Learning Advances to Data Visualization: A Survey on ML4VIS," *IEEE Transactions on Visualization and Computer Graphics*, 2021.
[24] Wu, Aoyu and Wang, Yun and Shu, Xinhuan and Moritz, Dominik and Cui, Weiwei and Zhang, Haidong and Zhang, Dongmei and Qu, Huamin, "Survey on Artificial Intelligence Approaches for Visualization Data," *IEEE Transactions on Visualization and Computer Graphics*, 2021.
[25] Perozzi, Bryan and Al-Rfou, Rami and Skiena, Steven, "Deepwalk: Online learning of social representations," in *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2014, pp. 701–710.
[26] Tang, Jian and Qu, Meng and Wang, Mingzhe and Zhang, Ming and Yan, Jun and Mei, Qiaozhu, "Line: Large-scale information network embedding," in *Proceedings of the 24th international conference on world wide web*, 2015, pp. 1067–1077.
[27] Grover, Aditya and Leskovec, Jure, "node2vec: Scalable feature learning for networks," in *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, 2016, pp. 855–864.
[28] Scarselli, Franco and Gori, Marco and Tsoi, Ah Chung and Hagenbuchner, Markus and Monfardini, Gabriele, "The graph neural network model," *IEEE transactions on neural networks*, vol. 20, no. 1, pp. 61–80, 2008.
[29] Kipf, Thomas N and Welling, Max, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.
[30] You, Jiaxuan and Ying, Rex and Leskovec, Jure, "Position-aware graph neural networks," in *International Conference on Machine Learning*. PMLR, 2019, pp. 7134–7143.
[31] Keyulu Xu and Weihua Hu and Jure Leskovec and Stefanie Jegelka, "How Powerful are Graph Neural Networks?" in *International Conference on Learning Representations*, 2019.
[32] Jie Chen and Tengfei Ma and Cao Xiao, "FastGCN: Fast Learning with Graph Convolutional Networks via Importance Sampling," in *International Conference on Learning Representations*, 2018.
[33] Hamilton, William L and Ying, Rex and Leskovec, Jure, "Inductive representation learning on large graphs," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, 2017, pp. 1025–1035.
[34] Petar Veličković and Guillem Cucurull and Arantxa Casanova and Adriana Romero and Pietro Liò and Yoshua Bengio, "Graph Attention Networks," in *International Conference on Learning Representations*, 2018.
[35] Tiezzi, Matteo and Ciravegna, Gabriele and Gori, Marco, "Graph Neural Networks for Graph Drawing," *IEEE Transactions on Neural Networks and Learning Systems*, 2022.
[36] K. He and X. Zhang and S. Ren and J. Sun, "Deep Residual Learning for Image Recognition," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016, pp. 770–778.
[37] Leow, Yao Yang and Laurent, Thomas and Bresson, Xavier, "GraphTSNE: a visualization technique for graph-structured data," *arXiv preprint arXiv:1904.06915*, 2019.
[38] Haleem, Hammad and Wang, Yong and Puri, Abishek and Wadhwa, Sahil and Qu, Huamin, "Evaluating the readability of force directed graph layouts: A deep learning approach," *IEEE computer graphics and applications*, vol. 39, no. 4, pp. 40–53, 2019.
[39] Wang, Yong and Jin, Zhihua and Wang, Qianwen and Cui, Weiwei and Ma, Tengfei and Qu, Huamin, "DeepDrawing: A deep learning approach to graph drawing," *IEEE Transactions on Visualization and Computer Graphics*, vol. 26, no. 1, pp. 676–686, 2019.
[40] Goodall, Colin, "Procrustes methods in the statistical analysis of shape," *Journal of the Royal Statistical Society: Series B (Methodological)*, vol. 53, no. 2, pp. 285–321, 1991.
[41] Hu, Yifan and Kobourov, Stephen G and Veeramoni, Sankar, "Embedding, clustering and coloring for dynamic maps," in *2012 IEEE Pacific Visualization Symposium*. IEEE, 2012, pp. 33–40.
[42] Espadoto, Mateus and Hirata, Nina Sumiko Tomita and Telea, Alexandru C, "Deep learning multidimensional projections," *Information Visualization*, vol. 19, no. 3, pp. 247–269, 2020.
[43] Wang, Xiaoqi and Yen, Kevin and Hu, Yifan and Shen, Han-Wei, "DeepGD: A Deep Learning Framework for Graph Drawing Using GNN," *IEEE Computer Graphics and Applications*, vol. 41, no. 5, pp. 32–44, 2021.
[44] Defferrard, Michaël and Bresson, Xavier and Vandergheynst, Pierre,
[45] J. Bruna, W. Zaremba, A. Szlam, and Y. LeCun, "Spectral networks and locally connected networks on graphs," *arXiv preprint arXiv:1312.6203*, 2013.
"Convolutional neural networks on graphs with fast localized spectral filtering," *arXiv preprint arXiv:1606.09375*, 2016.
[46] Hammond, David K and Vandergheynst, Pierre and Gribonval, Rémi, "Wavelets on graphs via spectral graph theory," *Applied and Computational Harmonic Analysis*, vol. 30, no. 2, pp. 129–150, 2011.
[47] Auber, David and Archambault, Daniel and Bourqui, Romain and Delest, Maylis and Dubois, Jonathan and Lambert, Antoine and Mary, Patrick and Mathiaut, Morgan and Melançon, Guy and Pinaud, Bruno and others, "TULIP 5," 2017.
[48] Brandes, Ulrik and Gaertler, Marco and Wagner, Dorothea, "Experiments on graph clustering algorithms," in *European Symposium on Algorithms*. Springer, 2003, pp. 568–579.
[49] Kingma, Diederik P and Ba, Jimmy, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.
[50] Wang, Yong and Shen, Qiaomu and Archambault, Daniel and Zhou, Zhiguang and Zhu, Min and Yang, Sixiao and Qu, Huamin, "Ambiguityvis: Visualization of ambiguity in graph layouts," *IEEE Transactions on Visualization and Computer Graphics*, vol. 22, no. 1, pp. 359–368, 2015.
[51] S. V. Dongen, "Graph clustering by flow simulation," Ph.D. dissertation, University of Utrecht, 2000.
[52] Friedman, Milton, "The use of ranks to avoid the assumption of normality implicit in the analysis of variance," *Journal of the american statistical association*, vol. 32, no. 200, pp. 675–701, 1937.
[53] Nemenyi, Peter Bjorn, *Distribution-free multiple comparisons.* Princeton University, 1963.

Loann Giovannangeli is a PhD. student at the LaBRI, University of Bordeaux, France. He worked one year as a research engineer in the LaBRI. He obtained his Master of Science degree in 2019 from the University of Bordeaux. His research interest include Information Visualization, Machine Learning and especially the applications of Artificial Intelligence for visualizations generation and evaluation.

Frederic Lalanne was graduated from Enseirb-Matmeca in 2013 and joined the LaBRI, University of Bordeaux in 2014 where he has been working mostly on large data analysis, visualization and LaBRI's large data platform.

David Auber received his PhD degree from the University of Bordeaux I in 2003. He has been an assistant professor in the University of Bordeaux Department of Computer Science since 2004. His current research interests include information visualization, graph drawing, bioinformatics, databases, and software engineering

Romain Giot received his Ph.D. degree in biometric authentication at the University of Caen in 2012 and is an associate professor at the University of Bordeaux in a big-data visualization team since 2013. His researches are dedicated to visualization, machine learning and their junction in eXplainable AI (XAI). He co-authored dozens of peer-reviewed papers and is involved in several Program Committees.

Romain Bourqui received his Master and PhD degrees in Computer Science from the University Bordeaux I in 2005 and 2008. He has been an associate professor at the University of Bordeaux since 2009. His research interests include Information Visualization, Large Data Visualization, Explainable Machine Learning.