Classifying Graphs as Images with Convolutional Neural Networks

Antoine J.-P. Tixier
École Polytechnique

Giannis Nikolentzos
École Polytechnique and AUEB

Polykarpos Meladianos
École Polytechnique and AUEB

Michalis Vazirgiannis
École Polytechnique and AUEB

Abstract

The task of graph classification is currently dominated by graph kernels, which, while powerful, suffer some significant limitations. Convolutional Neural Networks (CNNs) offer a very appealing alternative. However, processing graphs with CNNs is not trivial. To address this challenge, many sophisticated extensions of CNNs have recently been proposed. In this paper, we show that a classical 2D architecture designed for images can also be used for graph processing in a completely off-the-shelf manner; the only prerequisite being to encode graphs as stacks of two-dimensional histograms of their node embeddings. Despite its simplicity, our method proves very competitive to state-of-the-art graph kernels, and even outperforms them by a wide margin on some datasets. Our approach is also preferable in terms of time complexity. Code and data are publicly available[1].

1 Motivation

Graphs, or networks, are rich, flexible, and universal structures that can accurately represent the interaction among the components of many natural and human-made complex systems [1]. For instance, graphs have been used to describe and analyze the interplay among proteins within cells and the internal structure of proteins themselves [2, 3], the organization of the brain [4], the World Wide Web [5], textual documents [6], and information propagation through a population [7].

Consequently, network processing has been the subject of extensive research in machine learning and artificial intelligence, and still remains today a very active area of investigation. A central graph mining task is that of *graph classification*: its applications range from determining whether a protein is an enzyme or not in bioinformatics to categorizing documents in NLP. Graph classification is the task of interest in this study.

Limitations of graph kernels. The state-of-the-art in graph classification is currently dominated by a family of methods referred to as *graph kernels*, which compute the similarity between two graphs as the sum of the pairwise similarities between some of their substructures, and then pass the similarity matrix computed on the entire dataset to a kernel-based supervised algorithm such as the Support Vector Machine [8] to learn soft classification rules. Graph kernels mainly vary based on the substructures they use, which range from random walks [9] and shortest paths [10] to subgraphs [11], to cite only a few. Graph kernels have been very successful, but they suffer several limitations, among which is their (1) high time complexity. The problem is threefold: first, populating the kernel matrix requires computing the similarity between every two graphs in the training set (say of size N), which amounts to $N(N-1)/2$ operations. Training therefore becomes increasingly more expensive as the dataset gets bigger. Second, computing the similarity between a pair of graphs (i.e., a single operation)

[1] https://github.com/Tixierae/graph_2D_CNN

is itself polynomial in the number of nodes. For instance, the time complexity of the shortest path graph kernel is $\mathcal{O}(|V_1|^2|V_2|^2)$ for two graphs (V_1, V_2), where $|V_i|$ is the number of nodes in graph V_i. Processing large graphs can thus become prohibitive, which is a serious limitation as big networks abounds in practice. Finally, finding the support vectors is $\mathcal{O}(N^2)$ when C is small and $\mathcal{O}(N^3)$ when it gets large [12], which can again pose a problem on big datasets. Another significant limitation of graph kernels is that (2) the computation of the similarity matrix and the learning of the classification rules are two independent steps. In other words, the features are fixed and not optimized for the task. Conversely, CNNs learn their own features directly from the raw data during training to optimize performance on the downstream task. Finally, by comparing graphs based on their substructures, most existing graph kernels focus on local properties and (3) ignore the global structure of graphs [13]. As will be detailed, our method addresses this limitation by using a representation of graphs based on node embeddings that capture both local and global information.

CNNs for graphs. Motivated by the outstanding performance recently reached by Convolutional Neural Networks (CNNs) in computer vision, e.g. [14, 15], many research efforts have been devoted to generalizing CNNs to graphs. Indeed, CNNs offer a very appealing alternative to kernel-based methods. The parsimony achieved through weight sharing makes them very efficient, their time complexity is constant for each training example and linear with respect to the size of the dataset, and the extra expressiveness they bring might translate into significant accuracy gains.

However, since convolution and pooling are natively defined for regular, low-dimensional grids such as images (2D Euclidean space discretized by rectangles), generalizing CNNs to graphs, which are irregular, non-Euclidean objects, is far from trivial. Possible solutions that can be found in the literature fall into two broad categories: *spatial* and *spectral* techniques [16]. Spectral approaches [17, 18] invoke the convolution theorem from signal processing theory to perform graph convolutions as pointwise multiplications in the Fourier domain of the graph. The basis used to send the graph to the Fourier domain is given by the SVD decomposition of the Laplacian matrix of the graph, whose eigenvalues can be viewed as "frequencies". By contrast, spatial methods [19, 20] operate directly on the graph structure.

While these sophisticated frameworks have made great strides, we show in this paper that it is also possible to process graphs by using a standard 2D CNN architecture in a completely off-the-shelf manner, simply by representing graphs as stacks of 2D histograms of their node embeddings. Compared to the more complex approaches that involve different fundamental architectural and/or operational modifications, the main advantage of our method is its simplicity. Furthermore, by using a well-established architecture known to deliver solid performance for image classification, we show that this simplicity can be obtained without giving up accuracy: our results indeed prove competitive to state-of-the-art graph kernel baselines, and are even significantly better on some datasets, by a wide margin.

2 Proposed method

In the following section, we first introduce 2D CNNs and the specific architecture we used in our experiments, and then describe how to transform graphs so as to pass them as input to 2D CNNs.

2.1 2D CNN architecture

Convolutional Neural Networks (CNNs) are feedforward neural networks specifically designed to work on regular grids. A regular grid is the d-dimensional Euclidean space discretized by parallelotopes (rectangles for $d = 2$, cuboids for $d = 3$, etc.). In CNNs, each neuron in a given layer receives input from a neighborhood of the neurons in the previous layer [21]. Those neighborhoods, or *local receptive fields*, allow CNNs to compose higher-level features from lower-level features, and thus to capture patterns of increasing complexity in a hierarchical way. Regular grids satisfy the *spatial dependence*[2] property, which is the fundamental premise on which local receptive fields and hierarchical composition of features in CNNs hold. Next, we pursue our description in the 2D context, where the input are images represented as real-valued tensors of shapes $(n_c, n_{rows}, n_{cols})$; n_c referring to the number of channels (e.g., 1 for grey images, 3 for color images). Furthermore, without loss of generality, we consider square images, for which $n_{rows} = n_{cols} = n$.

[2] the concept of spatial dependence is well summarized by: "everything is related to everything else, but near things are more related than distant things" [22]. For instance in images, close pixels are more related than distant pixels.

Convolution and pooling. The standard building block of the CNN, which may be repeated several times, consists in a convolution layer followed by a pooling layer. The convolution layer first applies a filter parameterized by a 3D weight tensor $W \in \mathbb{R}^{n_c \times h \times h}$ over the input image $I \in \mathbb{R}^{n_c \times n \times n}$. This is a linear operation which multiplies W with each instantiation of a 2D window of same shape (h, h) slided over the entire image, extending fully along the n_c channels. Without zero-padding and with a stride of 1 (sliding the window by unit steps), there are $(n - h + 1)^2$ such instantiations. Then, a nonlinear activation function f, such as `ReLU` $\big(\max(0, x)\big)$ or `tanh` $\left(e^{2x}-1/e^{2x}+1\right)$, is applied elementwise to the convolved output, returning the feature map O associated with the filter. Equation 1 summarizes the whole process:

$$o_{i,j} = f(W \cdot I[i : i + h - 1, j : j + h - 1] + b) \tag{1}$$

Where $o_{i,j}$ is the $(i, j)^{th}$ element of the feature map $O \in \mathbb{R}^{(n-h+1)\times(n-h+1)}$ associated with filter W, $\cdot$ is an operator returning the sum of the element-wise products of the two input tensors (summed across channels), and b is the trainable bias of filter W. In practice, $n_f \sim 100$ filters of the same shape are applied to each region of the input, in order to give the model the freedom to learn different and complementary features. Thus, the output of the convolution layer is a volume of n_f 2D feature maps. From a high-level point of view, the convolution operation shrinks the input tensor with respect to its first two dimensions, width and height (spatial dimensions), but increases its depth from n_c to n_f (generally $n_c \ll n_f$). The weights of the filters are initialized randomly and tuned during training via backpropagation. Note that the weights of a given filter are *shared* across the image, endowing the model with the ability to recognize a pattern regardless of its location in the image.

The pooling layer then performs a sub-sampling operation that extracts a quantity, such as the average or the greatest value, independently from each small region of each feature map. Regions of size (2,2) are traditionally used, which downscales the input by a factor of 2 along each spatial dimension, and leaves the depth (number of feature maps) unchanged.

Architecture. We implemented a variant of LeNet-5 [21] with which we reached 99.45% accuracy on the MNIST handwritten digit classification dataset. As illustrated in Figure 1 for an input of shape (5,28,28), this simple architecture deploys four convolutional-pooling layers (each repeated twice) in parallel, with respective region sizes of 3, 4, 5 and 6, followed by two fully-connected layers. The original channels (depth) of the input tensor are replaced with the feature maps by the first convolutional layer, but the spatial dimensions (width and height) are preserved -while being shrunk- throughout all subsequent convolution-pooling operations. Dropout [23] is employed for regularization at every hidden layer. The activations are `ReLU` functions (in that, our model differs from LeNet-5), except for the ultimate layer, which uses a `softmax` to output a probability distribution over classes. The convolution-pooling block is expanded in Figure 2 for a filter of size 3, still using as an example an input of dimensionality (5,28,28). 64 filters are employed at the first level, and as the signal is subsampled through the pooling layer, the number of filters in the subsequent convolutional layer is increased to 96 to compensate for the loss in resolution.

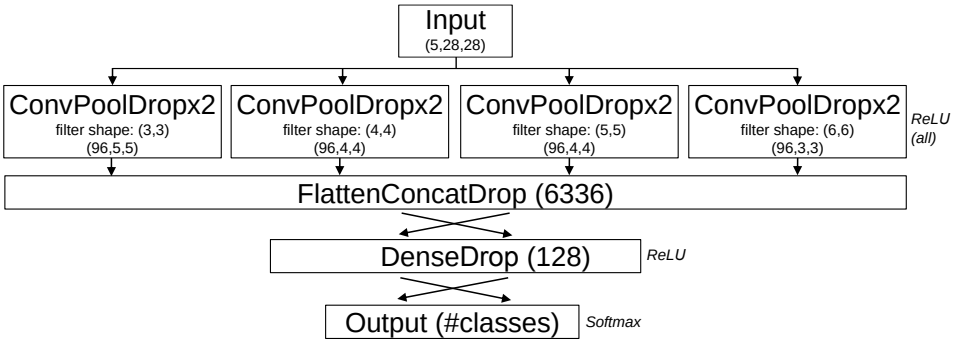

Figure 1: 2D CNN architecture used in our experiments. Numbers within parentheses refer to the *output* dimensions of the tensors. The `ConvPoolDrop`$\times 2$ block of layers is expanded in Figure 2.

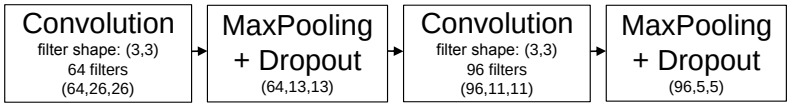

Figure 2: Flow of tensors for a block of convolutional and pooling layers repeated twice with dropout regularization, an input of shape (5,28,28), a filter shape of (3,3), no zero padding, and a stride of 1. Numbers within parentheses refer to the *output* dimensions. Max pooling with a (2,2) window halves the input along each spatial dimension, whereas dropout is a shape-preserving operation (it just randomly sets some of the elements of a tensor to zero).

2.2 How to represent a graph as an image?

Traditionally, a graph $G(V, E)$ is encoded as its adjacency matrix A or Laplacian matrix L. A is a square matrix of dimensionality $|V| \times |V|$, symmetric in the case of undirected graphs, whose $(i, j)^{th}$ entry $A_{i,j}$ is equal to the weight of the edge $e_{i,j}$ between nodes v_i and v_j, if such an edge exists, or to 0 otherwise. On the other hand, the Laplacian matrix L is equal to $D - A$, where D is the diagonal degree matrix. One could initially consider passing one of those structures as input to a 2D CNN. However, unlike images, in which close pixels are more strongly correlated than distant pixels, adjacency and Laplacian matrices are not associated with spatial dimensions and the notion of Euclidean distance, and thus do not satisfy the spatial dependence property. As will be detailed next, we capitalize on *graph node embeddings* to address this issue.

Graph node embeddings. In the node embedding space, the Euclidean distance between two points is meaningful: it is inversely proportional to the similarity of the two nodes they represent. Furthermore, using a representation based on graph node embeddings gives our approach a major advantage over graph kernels, especially on large graphs. Indeed, most graph kernels compare graphs based on their substructures, which are *local features*, while cutting-edge embedding algorithms very effectively encode both *local* and *global* properties of graphs. For instance, two neighboring points in the embedding space might be associated with two nodes very distant in the graph, but playing the same structural role (e.g., of flow control), belonging to the same community, or sharing some other common property.

Alignment and compression. State-of-the-art graph node embeddings techniques are neural, and thus stochastic. More precisely, the D dimensions are recycled from run to run, which means that a given dimension is not associated with the same latent concepts across graphs. Therefore, we ensure that the embeddings of all graphs in the collection are comparable by applying PCA and retaining the first $d \ll D$ principal components. PCA comes at a computational cost: one needs to populate the covariance matrix and carry out its eigenvalue decomposition. Even though the time complexity of each operation is respectively $\mathcal{O}(D^2|V|)$ and $\mathcal{O}(D^3)$, they come almost for free in practice as they are performed in the compact embedding space ($D \sim 100$). Note that in addition to alignment purposes, PCA also serves an information maximization (compression) purpose, which greatly reduces the shape of the tensors fed to the CNN (for reasons that will become clear in what follows), and thus complexity.

Stacking 2D histograms. We then repeatedly extract two-dimensional slices from the d-dimensional PCA node embedding space, and we turn those planes into regular grids by discretizing them into a finite, fixed number of equally-sized bins, where the value associated with each bin is the count of the number of nodes falling into that bin. In other words, we represent a graph as a stack of $d/2$ two-dimensional histograms of its (compressed) node embeddings[3]. As illustrated in Figure 3, the first histogram is computed from the coordinates of the nodes in the plane made of the first two principal directions, the second histogram from directions 3 and 4, and so forth.

Pixels, channels and resolution. Using computer vision vocabulary, bins can be viewed as *pixels*, and the 2D slices of the embedding space as *channels*. However, in our case, instead of having 3 channels (R,G,B) like with color images, we have $d/2$ of them. That is, each pixel (each bin) is associated with a vector of size $d/2$, whose entries are the counts of the nodes falling into that bin in the corresponding 2D slice of the embedding space. Finally, the *resolution* of the image is determined by the number of bins of the histograms, which is constant for a given dataset across all dimensions and channels.

[3] our representation is unrelated to the widespread *color histogram* encoding of images.

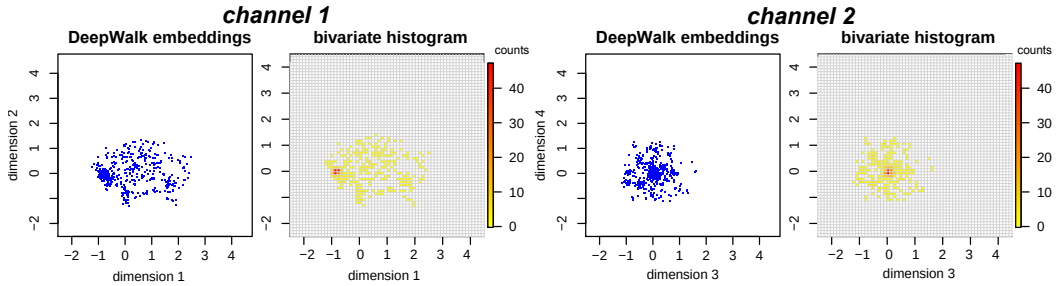

Figure 3: Node embeddings and image representation of graph ID #10001 (577 nodes, 1320 edges) from the REDDIT-12K dataset.

3 Experiments and Results

3.1 Setup

All experiments involved 10-fold cross validation where each fold was repeated 3 times. We used Xavier initialization [24], a batch size of 32, and for regularization, a dropout rate of 0.3 and early stopping with a patience of 5 epochs. The categorical cross-entropy loss was optimized with Adam [25] (default settings). We implemented our model in `Keras` [26] version 1.2.2[4] with `tensorflow` [27] backend. The hardware used consisted in an NVidia Titan Xp GPU with a 4-core Intel Xeon 2.40GHz CPU and 16GB of RAM, under Ubuntu 16.04.2 LTS 64-bit operating system and Python 2.7. The graph kernel baselines (presented in subsection 3.3) were ran on an 8-thread Intel i7 3.4GHz CPU, with 16GB of RAM, under Ubuntu 16.06 LTS 64-bit operating system and Python 2.7.

3.2 Random graphs

Dataset. To quickly test the viability of our pipeline, we created a synthetic dataset of 5 classes containing 2000 undirected and unweighted networks each. For the first and second classes, we generated graphs with the Stochastic Block Model [28], featuring respectively 2 and 3 communities of equal sizes. The in-block and cross-block probabilities were respectively set to 0.1 and 0.7, and the size $|V|$ of each graph was randomly sampled from the Normal distribution with mean 150 and standard deviation 30, i.e., $\mathcal{N}(150, 30)$. The third category was populated with scale-free networks, that is, graphs whose degree distributions follow a power law, using the Barabási-Albert model [29]. The number of incident edges per node was sampled for each graph from $\mathcal{N}(5, 2)$, and the size of each graph was drawn from $\mathcal{N}(150, 30)$ like for the first two classes. Finally, the fourth and fifth classes were filled with Erdös-Rényi graphs [30] whose sizes were respectively sampled from $\mathcal{N}(300, 30)$ and $\mathcal{N}(150, 30)$, and whose edge probabilities were drawn from $\mathcal{N}(0.3, 0.15)$. This overall, gave us a large variety of graphs.

Spectral embeddings. We started with the most naive way to embed the nodes of a graph, that is, using the eigenvectors of its adjacency or Laplacian matrix (we experimented with both and observed no difference). Here, no PCA-based compression was necessary. We retained the eigenvectors associated with the 10 largest eigenvalues in magnitude, thus embedding the nodes into a 10-dimensional space. Since it was giving very good results, we did not experiment with any other embedding method.

Resolution and channels. Because we computed the unit-normed eigenvectors, the coordinates of any node in any dimension belonged to the $[-1, 1]$ range. Furthermore, inspired by the MNIST images which are 28×28 in size, and on which we initially tested our 2D CNN architecture, we decided to learn 2D histograms featuring 28 bins in each direction. This gave us a resolution of $^{28}/_{(1-(-1))}$, that is, 14 pixels per unit, which we write 14:1 for brevity in the remainder of this paper. Finally, we decided to make use of the information carried out by all 10 eigenvectors, and thus kept 5 channels. Any graph in the dataset was thus represented as a tensor of shape (5,28,28).

[4] `https://faroit.github.io/keras-docs/1.2.2/`

Results. With the CNN architecture introduced in subsection 2.1, and under the experimental setup previously described, we reached a mean classification accuracy of $99.08\%(\pm 3.21)$. That level of accuracy was usually reached within 3 epochs. This very good performance can be explained by the abundance of training data and the perfect between-class balance of our synthetic dataset. But above all, it can be explained by the very specific, distinct patterns associated with each category, as illustrated in Figure 4 for the first four classes. Even though we tried to inject some variance, discriminating between categories is easy to the point that clear differences can be seen with the naked eye. Of course, by jointly processing all five channels, the CNN can detect and learn more complex and even more discriminative multidimensional patterns.

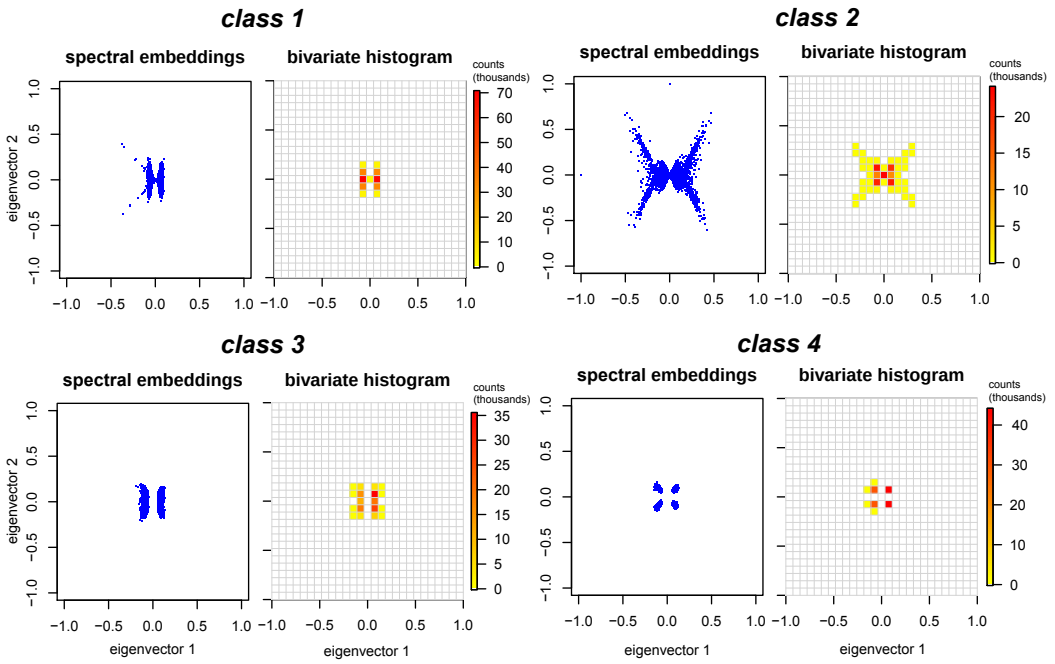

Figure 4: Overlayed node embeddings in the space made of the first two eigenvectors of the adjacency matrices (first channel) and associated overlayed bivariate histograms for all graphs in the first four classes of the *random* dataset.

These positive results provided empirical evidence for the validity of our approach, and were the impetus for moving forward with evaluating it on real-world datasets in comparison to state-of-the-art graph kernel baselines, which is the topic of the next subsection.

3.3 Real-world graphs

Datasets. We used publicly available[5] datasets from [31] which we briefly describe in what follows, and in Table 1. We refer the reader to the original paper for more information. In all datasets, graphs are unweighted, undirected, with unlabeled nodes, and the task is to predict the class they belong to. Classes are mutually exclusive. REDDIT-B, REDDIT-5K, and IMDB-B are perfectly balanced, whereas REDDIT-12K and COLLAB feature a maximum class imbalance ratio of 1:5 and 1:3.4, respectively. Note that graphs with less than 10 nodes were removed because having 5 channels requires at least a 10-dimensional embedding space, which is impossible to obtain with less than 10 nodes. However, this represented only a few graphs per dataset, for some datasets.

In all REDDIT datasets, a graph corresponds to a thread where nodes represent users, and there is an edge between two nodes if one of the two users responded to a comment from the other user. More precisely, graphs in **REDDIT-B** are labeled according to whether they were constructed from Q&A communities or discussion communities, and **REDDIT-5K** and **REDDIT-12K** respectively feature graphs from 5 and 11 forums dedicated to specific topics (those forums are known as "subreddits"). In **COLLAB**, graphs are hop-1 neighborhoods of researchers from a scientific collaboration network

[5] `http://www.mit.edu/~pinary/kdd/datasets.tar.gz`

Dataset	IMDB-B	COLLAB	REDDIT-B	REDDIT-5K	REDDIT-12K
Max # vertices	136	492	3782	3648	3782
Min # vertices	12	32	6	22	2
Average # vertices	19.77	74.49	429.61	508.50	391.40
Max # edges	1249	40120	4071	4783	5171
Min # edges	26	60	4	21	1
Average # edges	96.53	2457.78	497.75	594.87	456.89
# graphs	1000	5000	2000	4999	11929
# classes	2	3	2	5	11
Max class imbalance	1:1	1:3.4	1:1	1:1	1:5

Table 1: Statistics of the social network datasets used in our experiments.

(two researchers are linked if they co-authored a paper), and are labeled according to the subfield of Physics the corresponding researcher belongs to. Finally, the **IMDB-B** dataset features hop-1 neighborhoods of actors and actresses selected from two movie collaboration networks corresponding to specific genres (action and romance), in which two actors are linked if they starred in the same movie. Graphs are labeled according to the genre they were sampled from.

Baselines. We compared our model to two state-of-the-art graph kernels, the graphlet kernel [11] and the Weisfeiler-Lehman (WL) kernel [32]. The **graphlet** kernel computes the similarity between two graphs as the cosine of their count vectors. These vectors encode how many subgraphs of size up to a certain threshold can be found in each graph (each entry is an occurrence count). We sampled 2000 graphlets of size up to 6 from each graph. The **WL** kernel is actually a framework that operates on top of any graph kernel accepting node labels and boosts its performance by using the relabeling procedure of the WL test of isomorphism. More precisely, following the computation of the kernel value between the two graphs, vertex labels are updated based on the labels of their neighbors. This two-step process repeats for a certain number of iterations. The final kernel value is the sum of the values at each iteration. Since our graphs have unlabeled nodes, we set the degrees of the nodes as their labels. Furthermore, we used the WL framework with the subtree graph kernel [9], as it is very efficient with this kernel [32]. For both baselines, we used a C-SVM classifier[6] [33]. The C parameter of the SVM and the number of iterations in WL were jointly optimized on a 90-10 % partition of the training set of each fold by searching the grid $\big\{(10^{-4}, 10^4, \text{len} = 10); (2, 7, \text{step} = 1)\big\}$.

Neural embeddings. We started with spectral embeddings like for the random dataset, but observed that neural models were giving much better accuracy. More specifically, we used `node2vec` [34], which applies the very fast Skip-Gram language model [35] to truncated biased random walks performed on the graph. `node2vec` scales linearly with the number of nodes in the network. We leveraged the `igraph` module [36] and the publicly available high performance C++ implementation[7] of `node2vec`.

Resolution, channels, and p and q parameters. While performance on the random dataset was very satisfying out-of-the-box and required no optimization, we expected that at least some rough tuning would be necessary on the real-world datasets. With the p and q parameters of `node2vec` held constant and equal to 1, we conducted a search on the coarse grid $\big\{$(14,9);(2,5)$\big\}$ to get more insights about the impact of resolution and number of channels (respectively). On a given dataset, image size is calculated as the range $|max(\text{coordinates}) - min(\text{coordinates})|$ multiplied by the resolution (where "coordinates" are the node loadings flattened across all dimensions of the embedding space). For instance, on COLLAB with a resolution of 9:1, image size is equal to 37×37, since $|2.78 - (-1.33)| \times 9 \approx 37$. Optimal values for each dataset are summarized in Table 2. With the best resolution and number of channels, we then tuned the return and in-out parameters p and q of `node2vec`. Those parameters respectively bias the random walks towards exploring larger areas of the graph or staying in local neighborhoods, allowing the embeddings to encode a similarity that interpolates between structural equivalence (two nodes acting as, e.g., flow controllers, are close to each other) and homophily (two nodes belonging to the same community are close to each other). We tried 5 combinations of values for (p, q): $\big\{(1, 1); (0.25, 4); (4, 0.25); (0.5, 2); (2, 0.5)\big\}$. $p = q = 1$ is equivalent to `DeepWalk` [37].

Data augmentation. Generating more training examples by altering the input images is known to improve performance in image classification [15]. Since we had direct access to the underlying data

[6] `http://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html`

[7] `https://github.com/snap-stanford/snap/tree/master/examples/node2vec`

	REDDIT-B	REDDIT-5K	REDDIT-12K	COLLAB	IMDB-B
best resolution	9:1	9:1	9:1	9:1	14:1
best #channels	5	2	5	5	5
best p,q	2,0.5	4,0.25	1,1	0.25,4	1,1

Table 2: Best resolution, number of channels, and (p, q) for each dataset.

that were used to generate our images, which is typically not the case in computer vision, we thought it would be more sensible to implement a synthetic data generation scheme at the *node embeddings* level rather than at the image (2D histograms) level. More precisely, we used a simple nonparametric technique known as the *smoothed bootstrap with variance correction* [38]. This generator has been successfully used in the field of hydroclimatology to improve modeling of precipitation [39] and streamflow [40], and in risk analysis [41]. Unlike the traditional bootstrap [42] which simply draws with replacement from the initial set of observations, the smoothed bootstrap can generate values outside of the original range while still being faithful to the structure of the underlying data [43]. Actually, generating new observations with the smoothed bootstrap scheme can be viewed as sampling from the Kernel Density Estimate (KDE) of the sample, which is consistent with our way of representing graphs as images, since a KDE is nothing more than a smoothed histogram. The pseudo-code for the smoothed bootstrap algorithm is available in the Appendix, while a visual example can be seen in Figure 5.

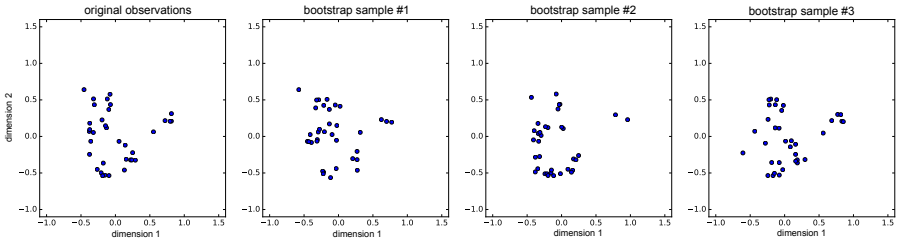

Figure 5: Original DeepWalk embeddings on PCA dimensions 1 and 2 (first channel) and synthetic values generated with 3 runs of the smoothed bootstrap generator, for graph ID #3 of IMDB-B dataset (35 nodes, 133 edges).

Results. The classification accuracy of our approaches and the baselines we implemented are reported in Table 3. Even though we did not re-implement those models, we also display for comparison purposes the performance reported in [31] (Deep Graph Kernels) and [19] (Graph CNN, PSCN $k = 10$), since the experimental setting is the same. Our approach shows significantly better than all baselines on the REDDIT-12K and REDDIT-B datasets, with large improvements of 6.81 and 3.15 in accuracy over the best performing competitor, respectively. We also reach best performance on the REDDIT-5K dataset, with an improvement in accuracy of 1.34 over the best performing baseline. However, the difference is not statistically significant. Finally, on the IMDB-B dataset, we get third place, very close (≤ 1.13) to the top performers, and again without statistically significant differences. Actually, the only dataset on which a baseline proved significantly better than our approach is COLLAB (WL graph kernel). Results also show that the smoothed bootstrap can bring some extra skill, as it beats our model without synthetic data generation on 3 datasets (REDDIT-B, COLLAB and IMDB-B). However, the performance improvements are modest and further research is necessary to make our data augmentation strategy more effective.

Runtimes. Even if not directly comparable, we report in Table 4 kernel matrix computation time for the two graph kernel baselines, along with the time required by our 2D CNN model to perform one pass over the entire training set, i.e., the time per epoch. With respects to time complexity, our method is superior to graph kernels on several counts: first, unlike graph kernels, the time required by the 2D CNN to process one training example is constant (all images for a given dataset have the same dimensions), while computing the kernel value for a pair of graphs depends on their size (polynomial in the number of nodes). It is true that a prerequisite for our approach is an embedding for all the graphs in the dataset, but `node2vec` scales linearly with the number of nodes in the graph. Therefore, on big graphs, our method is still usable, while graph kernels may not be. Second, with a 2D CNN, the time necessary to go through the entire training set only increases linearly with the size of the set, while populating the kernel matrix is quadratic, and finding the support vectors is then again at

Table 3: 10-fold CV average test set classification accuracy of our proposed methods (lower section) compared to state-of-the-art graph kernels and graph CNN (upper section). $\pm$ is standard deviation. Best performance per column in **bold**, among our methods underlined. *indicates stat. sign. at the $p < 0.05$ level (best 2D CNN vs. WL) using the Mann-Whitney U test (`https://docs.scipy.org/doc/scipy-0.19.0/reference/generated/scipy.stats.mannwhitneyu.html`).

Method \ Dataset	REDDIT-B (size=2,000;nclasses=2)	REDDIT-5K (4,999;5)	REDDIT-12K (11,929;11)	COLLAB (5,000;3)	IMDB-B (1,000;2)
Graphlet Shervashidze2009	77.26 ($\pm$ 2.34)	39.75 ($\pm$ 1.36)	25.98 ($\pm$ 1.29)	73.42 ($\pm$ 2.43)	65.40 ($\pm$ 5.95)
WL Shervashidze2011	78.52 ($\pm$ 2.01)	50.77 ($\pm$ 2.02)	34.57 ($\pm$ 1.32)	**77.82*** ($\pm$ 1.45)	**71.60** ($\pm$ 5.16)
Deep GK Yanardag2015	78.04 ($\pm$ 0.39)	41.27 ($\pm$ 0.18)	32.22 ($\pm$ 0.10)	73.09 ($\pm$ 0.25)	66.96 ($\pm$ 0.56)
PSCN $k = 10$ Niepert2016	86.30 ($\pm$ 1.58)	49.10 ($\pm$ 0.70)	41.32 ($\pm$ 0.42)	72.60 ($\pm$ 2.15)	71.00 ($\pm$ 2.29)
2D CNN	89.12 ($\pm$ 1.70)	**52.11** ($\pm$ 2.24)	**48.13*** ($\pm$ 1.47)	70.28 ($\pm$ 1.21)	70.40 ($\pm$ 3.85)
2D CNN + boot .05	87.75 ($\pm$ 7.62)	51.80 ($\pm$ 1.89)	47.51 ($\pm$ 1.29)	70.37 ($\pm$ 1.60)	70.33 ($\pm$ 4.19)
2D CNN + boot .1	87.98 ($\pm$ 8.24)	51.79 ($\pm$ 1.98)	47.60 ($\pm$ 1.30)	70.25 ($\pm$ 1.55)	70.47 ($\pm$ 3.24)
2D CNN + boot .2	**89.45*** ($\pm$ 1.64)	51.61 ($\pm$ 2.05)	47.34 ($\pm$ 1.15)	70.31 ($\pm$ 1.19)	70.13 ($\pm$ 3.65)
2D CNN + boot .5	89.30 ($\pm$ 1.40)	51.44 ($\pm$ 2.02)	46.95 ($\pm$ 1.35)	70.29 ($\pm$ 1.64)	69.53 ($\pm$ 3.08)

least quadratic. This means that on large datasets, our approach is also preferable to graph kernels. Examples of 2D CNN architectures much more complex than ours applied to millions of images in reasonable time abounds in the recent computer vision literature. Processing such big datasets with graph kernels would simply be intractable. In addition, not only do neural models allow training on very large datasets, but they also usually benefit a great deal from it.

	REDDIT-B	REDDIT-5K	REDDIT-12K	COLLAB	IMDB-B
Size, average (# nodes, # edges)	2000, (430,498)	4999, (509,595)	11929, (391,457)	5000, (74,2458)	1000, (20,97)
Input shapes (for our approach)	(5,62,62)	(2,65,65)	(5,73,73)	(5,36,36)	(5,37,37)
Graphlet Shervashidze2009	551	5,046	12,208	3,238	275
WL Shervashidze2011	645	5,087	20,392	1,579	23
2D CNN (our approach)	6	16	52	5	1

Table 4: Runtimes. For the graph kernel baselines, time necessary to populate the Kernel matrix (8-thread 3.4GHz CPU). For our model, time per epoch (Titan Xp GPU). Times are in seconds, rounded to the nearest integer.

Discussion. Replacing the raw counts by the empirical joint probability density function, either by normalizing the histograms, or with a KDE, significantly deteriorated performance. This suggests that keeping the absolute values of the counts is important, which makes sense, because some categories might be associated with larger or smaller graphs, on average. Therefore, preventing the model from using size information is likely to decrease accuracy. We also observed that increasing the number of channels to more than 5 does not yield better results, but that reducing this number improves performance in some cases, probably because it plays a regularization role.

The main contribution of our study is a novel method for *representing* graphs as images from their node embeddings. How the embeddings are computed, and which classifier is used, does not matter. We hold this flexibility to be a major strength. First, the *embedding-agnostic* nature of our method means that it can be seamlessly extended to directed, weighted, or labeled graphs with continuous or categorical node/edge attributes, simply by using an embedding algorithm that accepts such graphs, e.g. [44–46]. The independence of our approach with respect to the image classification model used is another advantage. Here, we employed a vanilla 2D CNN architecture as it was offering an excellent trade-off between accuracy and simplicity, but more recent models, such as [47], may yield even better results. Above all, performance should improve as graph node embedding algorithms and CNN architectures for images improve in the future.

Even though results are very good out-of-the-box in most cases, finding an embedding algorithm that works well, or the right combination of parameters for a given dataset, can require some efforts. For instance, on COLLAB, we hypothesize that our results are inferior to that observed on the other datasets because optimizing p and q for COLLAB may require more than a coarse grid search, or because `node2vec` may not be well-suited to very dense graphs in the first place.

Finally, generating a different number of bootstrap samples for each class could also work better on the imbalanced datasets.

4 Conclusion

We showed that CNN architectures designed for images can be used for graph processing in a completely off-the-shelf manner, by representing graphs as stacks of two-dimensional histograms of their node embeddings. Despite the simplicity of our approach, we showed it to be very competitive to the state-of-the-art, sometimes outperforming the baselines with wide margins. Furthermore, these good results were obtained with limited parameter tuning and by using a basic 2D CNN model. From a time complexity perspective, our approach is preferable to graph kernels too, allowing to process larger datasets featuring bigger graphs.

Acknowledgments

We gratefully acknowledge the support of the NVIDIA Corporation, who donated the Titan X Pascal GPU that was used for this research in the context of the NVIDIA GPU Grant program.

References

[1] Sergei N Dorogovtsev and José FF Mendes. *Evolution of networks: From biological nets to the Internet and WWW*. OUP Oxford, 2013.

[2] Karsten M Borgwardt, Hans-Peter Kriegel, SV Vishwanathan, and Nicol N Schraudolph. Graph kernels for disease outcome prediction from protein-protein interaction networks. In *Pacific symposium on biocomputing*, volume 12, pages 4–15, 2007.

[3] Karsten M Borgwardt, Cheng Soon Ong, Stefan Schönauer, SVN Vishwanathan, Alex J Smola, and Hans-Peter Kriegel. Protein function prediction via graph kernels. *Bioinformatics*, 21(suppl 1):i47–i56, 2005.

[4] Ed Bullmore and Olaf Sporns. Complex brain networks: graph theoretical analysis of structural and functional systems. *Nature Reviews Neuroscience*, 10(3):186–198, 2009.

[5] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

[6] Rada Mihalcea and Paul Tarau. Textrank: Bringing order into texts. Association for Computational Linguistics, 2004.

[7] Maksim Kitsak, Lazaros K Gallos, Shlomo Havlin, Fredrik Liljeros, Lev Muchnik, H Eugene Stanley, and Hernán A Makse. Identification of influential spreaders in complex networks. *Nature physics*, 6(11):888–893, 2010.

[8] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

[9] Thomas Gärtner, Peter Flach, and Stefan Wrobel. On graph kernels: Hardness results and efficient alternatives. In *Learning Theory and Kernel Machines*, pages 129–143. Springer, 2003.

[10] K. M. Borgwardt and H. Kriegel. Shortest-path kernels on graphs. In *Proceedings of the 5th International Conference on Data Mining*, pages 74–81, 2005.

[11] N. Shervashidze, T. Petri, K. Mehlhorn, K. M. Borgwardt, and S.V.N. Vishwanathan. Efficient Graphlet Kernels for Large Graph Comparison. In *Proceedings of the International Conference on Artificial Intelligence and Statistics*, pages 488–495, 2009.

[12] Léon Bottou and Chih-Jen Lin. Support vector machine solvers. *Large scale kernel machines*, 3(1):301–320, 2007.

[13] Giannis Nikolentzos, Polykarpos Meladianos, and Michalis Vazirgiannis. Matching node embeddings for graph similarity. In *AAAI*, pages 2429–2435, 2017.

[14] Oriol Vinyals, Alexander Toshev, Samy Bengio, and Dumitru Erhan. Show and tell: A neural image caption generator. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3156–3164, 2015.

[15] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[16] Joan Bruna, Wojciech Zaremba, Arthur Szlam, and Yann LeCun. Spectral networks and locally connected networks on graphs. *arXiv preprint arXiv:1312.6203*, 2013.

[17] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. In *Advances in Neural Information Processing Systems*, pages 3837–3845, 2016.

[18] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[19] Mathias Niepert, Mohamed Ahmed, and Konstantin Kutzkov. Learning convolutional neural networks for graphs. In *Proceedings of the 33rd annual international conference on machine learning. ACM*, 2016.

[20] Jean-Charles Vialatte, Vincent Gripon, and Grégoire Mercier. Generalizing the convolution operator to extend cnns to irregular domains. *arXiv preprint arXiv:1606.01166*, 2016.

[21] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[22] Waldo R Tobler. A computer movie simulating urban growth in the detroit region. *Economic geography*, 46(sup1):234–240, 1970.

[23] Nitish Srivastava, Geoffrey E Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[24] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Aistats*, volume 9, pages 249–256, 2010.

[25] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[26] François Chollet et al. Keras. `https://github.com/fchollet/keras`, 2015.

[27] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[28] Paul W Holland, Kathryn Blackmond Laskey, and Samuel Leinhardt. Stochastic blockmodels: First steps. *Social networks*, 5(2):109–137, 1983.

[29] Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *science*, 286(5439):509–512, 1999.

[30] Paul Erdös and Alfréd Rényi. On random graphs, i. *Publicationes Mathematicae (Debrecen)*, 6:290–297, 1959.

[31] Pinar Yanardag and SVN Vishwanathan. Deep graph kernels. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1365–1374. ACM, 2015.

[32] N. Shervashidze, P. Schweitzer, E. J. Van Leeuwen, K. Mehlhorn, and K. M. Borgwardt. Weisfeiler-Lehman Graph Kernels. *The Journal of Machine Learning Research*, 12:2539–2561, 2011.

[33] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[34] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 855–864. ACM, 2016.

[35] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[36] Gabor Csardi and Tamas Nepusz. The igraph software package for complex network research. *InterJournal*, Complex Systems:1695, 2006.

[37] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 701–710. ACM, 2014.

[38] Bernard W Silverman. *Density estimation for statistics and data analysis*, volume 26. CRC press, 1986.

[39] Upmanu Lall, Balaji Rajagopalan, and David G Tarboton. A nonparametric wet/dry spell model for resampling daily precipitation. *Water resources research*, 32(9):2803–2823, 1996.

[40] Ashish Sharma, David G Tarboton, and Upmanu Lall. Streamflow simulation: A nonparametric approach. *Water resources research*, 33(2):291–308, 1997.

[41] Antoine J-P Tixier, Matthew R Hallowell, and Balaji Rajagopalan. Construction safety risk modeling and simulation. *Risk analysis*, 2017.

[42] Bradley Efron. Bootstrap methods: another look at the jackknife. In *Breakthroughs in Statistics*, pages 569–593. Springer, 1992.

[43] B Rajagopalan, U Lall, David G Tarboton, and DS Bowles. Multivariate nonparametric resampling scheme for generation of daily weather variables. *Stochastic Hydrology and Hydraulics*, 11(1):65–93, 1997.

[44] Lizi Liao, Xiangnan He, Hanwang Zhang, and Tat-Seng Chua. Attributed social network embedding. *arXiv preprint arXiv:1705.04969*, 2017.

[45] Zhilin Yang, William W Cohen, and Ruslan Salakhutdinov. Revisiting semi-supervised learning with graph embeddings. *arXiv preprint arXiv:1603.08861*, 2016.

[46] Cheng Yang, Zhiyuan Liu, Deli Zhao, Maosong Sun, and Edward Y Chang. Network representation learning with rich text information. In *IJCAI*, pages 2111–2117, 2015.

[47] Gao Huang, Zhuang Liu, Kilian Q Weinberger, and Laurens van der Maaten. Densely connected convolutional networks. *arXiv preprint arXiv:1608.06993*, 2016.

Appendix

Algorithm 1: Multivariate smoothed bootstrap with variance correction

Input : list L of M arrays $A_m \in \mathbb{R}^{|V_{A_m}| \times d}$ of original observations, $p_{boot} \in \mathbb{R}^+$
Output : list L' of $M' = \mathrm{int}\big(p_{boot} \times M\big)$ arrays of synthetic observations

1 $L'' \leftarrow$ select M' elements from L at random without replacement
2 **for** $A \in \mathbb{R}^{|V_A| \times d}$ *in* L'' **do**
3 $\quad \mu_A \in \mathbb{R}^d \leftarrow \mathrm{apply}(\mathrm{mean}, \mathrm{A})$
4 $\quad \sigma_A^2 \in \mathbb{R}^d \leftarrow \mathrm{apply}(\mathrm{var}, \mathrm{A})$
5 $\quad h_A \in \mathbb{R}^d \leftarrow \mathrm{apply}(\mathrm{compute\_kde\_bandwidth}, \mathrm{A})$
6 $\quad A' \leftarrow$ empty array
7 $\quad$ **for** $j \leftarrow 1$ **to** $\mathcal{N}(|V_A|, \frac{|V_A|}{5})$ **do**
8 $\quad\quad$ sample i from $U(1, |V_A|)$
9 $\quad\quad \epsilon \in \mathbb{R}^d \leftarrow \mathrm{apply}(\mathcal{N}(0, \sqrt{\mathrm{x}}), \mathrm{h_A})$
10 $\quad\quad$ **for** $k \leftarrow 1$ **to** d **do**
11 $\quad\quad\quad A'[j,k] \leftarrow \mu_A[k] + {}^{(A[i,k] - \mu_A[k] + \epsilon[k])}/_{\sqrt{1 + {}^{h_A[k]}/_{\sigma_A^2[k]}}}$
12 $\quad\quad$ **end**
13 $\quad$ **end**
14 $\quad$ **store** A' in L'
15 **end**
16 **return** L'

The algorithm takes an input a list of graph node embeddings, each of dimension (number of vertices in the graph) $\times\, d$, where d is the dimensionality of the (compressed) embedding space. It returns a list of M' synthetic d-dimensional graph node embeddings, where M' is determined by the augmentation ratio p_{boot}, typically in $]0, 1]$. Line 5 computes the bandwidths of the columnwise Kernel Density

Estimates (KDEs), which control the degree of smoothing. The bandwidth of the KDE is subject to the classical bias-variance trade-off and is thus estimated from grid searching `logspace(-1, -0.5, num=10)` on an equal train/test split. Line 7 ensures that the size of the new graph (number of rows in the new array) slightly differs from the size of the original graph.

Line 8 selects a row (i.e., a node) uniformly at random from the embeddings, and line 9 samples some d-dimensional Gaussian noise, with, for each dimension, standard deviation equal to the square root of the KDE bandwidth for that dimension (the bandwidth can be viewed as the variance of the KDE).

Finally, lines 10 and 11 simulate the new d-dimensional observation. Simply adding noise $\epsilon[k]$ to the original observation $A[i, k]$ would yield realizations with an increased variance compared to the original sample [38], hence the additional correction terms in line 11.