The Thirty-Second AAAI Conference
on Artificial Intelligence (AAAI-18)

An End-to-End Deep Learning Architecture for Graph Classification

Muhan Zhang, Zhicheng Cui, Marion Neumann, Yixin Chen
Department of Computer Science and Engineering, Washington University in St. Louis
{muhan, z.cui, m.neumann}@wustl.edu, chen@cse.wustl.edu

Abstract

Neural networks are typically designed to deal with data in tensor forms. In this paper, we propose a novel neural network architecture accepting graphs of arbitrary structure. Given a dataset containing graphs in the form of (G, y) where G is a graph and y is its class, we aim to develop neural networks that read the graphs directly and learn a classification function. There are two main challenges: 1) how to extract useful features characterizing the rich information encoded in a graph for classification purpose, and 2) how to sequentially read a graph in a meaningful and consistent order. To address the first challenge, we design a localized graph convolution model and show its connection with two graph kernels. To address the second challenge, we design a novel SortPooling layer which sorts graph vertices in a consistent order so that traditional neural networks can be trained on the graphs. Experiments on benchmark graph classification datasets demonstrate that the proposed architecture achieves highly competitive performance with state-of-the-art graph kernels and other graph neural network methods. Moreover, the architecture allows end-to-end gradient-based training with original graphs, without the need to first transform graphs into vectors.

1 Introduction

The past few years have seen the growing prevalence of neural networks on application domains such as image classification (Alex, Sutskever, and Hinton 2012), natural language processing (Mikolov et al. 2013), reinforcement learning (Mnih et al. 2013), and time series analysis (Cui, Chen, and Chen 2016). The connection structure between layers makes neural networks suitable for processing signals in tensor forms where the tensor elements are arranged in a meaningful order. This fixed input order is a cornerstone for neural networks to extract higher-level features. For example, if we randomly shuffle the pixels of an image shown in Figure 1, then state-of-the-art convolutional neural networks (CNN) fail to recognize it as an eagle.

Although images and many other types of data are naturally presented with order, there is another major category of structured data, namely graphs, which usually lack a tensor representation with fixed ordering. Examples include molecular structures, knowledge graphs, biological networks, social networks, and text documents with dependencies. The lack of ordered tensor representations limits the applicability of neural networks on graphs. In this paper, we aim at designing novel neural network structures that can accept graphs and learn predictive functions.

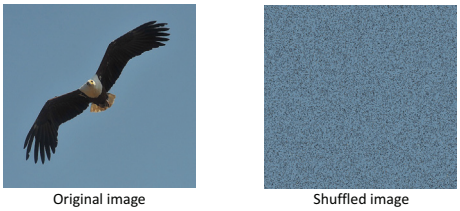

Figure 1: A consistent input ordering is crucial for CNNs.

Recently, there is a growing interest in generalizing neural networks to graphs. (Bruna et al. 2013) generalized convolutional networks to graphs in the spectral domain, where filters are applied on a graph's frequency modes computed by graph Fourier transform. This transformation involves expensive multiplications with the eigenvector matrix of the graph Laplacian. To reduce the computation burden, (Defferrard, Bresson, and Vandergheynst 2016) parameterized the spectral filters as Chebyshev polynomials of eigenvalues, and achieved efficient and localized filters. One limitation of the above spectral formulations is that they rely on the fixed spectrum of the graph Laplacian, and thus are suitable only for graphs with a single structure (and varying signals on vertices). Spatial formulations, on the contrary, are not restricted to a fixed graph structure. To extract local features, several works independently proposed to propagate features between neighboring vertices. (Duvenaud et al. 2015) proposed differentiable Neural Graph Fingerprints, which propagate features between 1-hop neighbors to simulate the traditional circular fingerprint. (Atwood and Towsley 2016) proposed Diffusion-CNN, which propagates neighbors with different hops to the center using different weights. Later, (Kipf and Welling 2016) developed a first-order approximation of the spectral convolution in (Defferrard, Bresson, and Vandergheynst 2016) which also resulted in propagation between neighboring vertices. (Niepert, Ahmed, and Kutzkov 2016) proposed another way of spatial graph convolution by extracting fixed-sized local patches from nodes' neighborhoods and linearizing these

patches with graph labeling methods and graph canonization tools. The resulting algorithm is called PATCHY-SAN.

Since spatial methods do not require a single graph structure, they can be applied to both node classification and graph classification tasks. Although achieving state-of-the-art node classification results (Atwood and Towsley 2016; Kipf and Welling 2016), most previous works have relatively worse performance on graph classification tasks. One reason for this is that after extracting localized vertex features, these features are directly summed up as a graph-level feature used for graph classification (Duvenaud et al. 2015; Defferrard, Bresson, and Vandergheynst 2016).

In this paper, we propose a new architecture that can keep much more vertex information and learn from the global graph topology. A key innovation is a new *SortPooling layer*, which takes as input a graph's unordered vertex features from spatial graph convolutions. Instead of summing up these vertex features, SortPooling arranges them in a consistent order, and outputs a sorted graph representation with a fixed size, so that traditional convolutional neural networks can read vertices in a consistent order and be trained on this representation. As a bridge between graph convolution layers and traditional neural network layers, the SortPooling layer can backpropagate loss gradients through it, integrating graph representation and learning into one end-to-end architecture.

Our contributions in this paper are as follows. 1) We propose a novel end-to-end deep learning architecture for graph classification. It directly accepts graphs as input without the need of any preprocessing. 2) We propose a novel spatial graph convolution layer to extract multi-scale vertex features, and draw analogies with popular graph kernels to explain why it works. 3) We develop a novel SortPooling layer to sort the vertex features instead of summing them up, which can keep much more information and allows us to learn from the global graph topology. 4) Experimental results on benchmark graph classification datasets show that our **Deep Graph Convolutional Neural Network** (DGCNN) is highly competitive with state-of-the-art graph kernels, and significantly outperforms many other deep learning methods for graph classification.

2 Deep Graph Convolutional Neural Network (DGCNN)

DGCNN has three sequential stages: 1) *graph convolution layers* extract vertices' local substructure features and define a consistent vertex ordering; 2) a *SortPooling layer* sorts the vertex features under the previously defined order and unifies input sizes; 3) traditional convolutional and dense layers read the sorted graph representations and make predictions. We show the DGCNN architecture in Figure 2.

We use $\mathbf{A}$ to denote the adjacency matrix of a graph, and n the number of vertices. We consider only simple graphs in this paper, i.e., $\mathbf{A}$ is a symmetric 0/1 matrix, and the graph has no self-loops. Suppose each vertex has a c-dimensional feature vector, we use $\mathbf{X} \in \mathbb{R}^{n\times c}$ to denote the graph's node information matrix with each row representing a vertex. For graphs with vertex labels or attributes, $\mathbf{X}$ can be the one-hot encoding matrix of the vertex labels or the matrix of multi-dimensional vertex attributes. For graphs without vertex labels, $\mathbf{X}$ can be defined as a column vector of normalized node degrees. We call a column in $\mathbf{X}$ a *feature channel* of the graph, thus the graph has c initial channels. In the rest of the paper, we use $\mathbf{P}_i$ to denote the i^{th} row of any matrix $\mathbf{P}$, and $\mathbf{P}_{ij}$ to denote the entry (i, j) of $\mathbf{P}$. For a vertex v, we use $\Gamma(v)$ to denote the set of v's neighboring nodes.

2.1 Graph convolution layers

Proposed form Given a graph $\mathbf{A}$ and its node information matrix $\mathbf{X} \in \mathbb{R}^{n\times c}$, our graph convolution layer takes the following form:

$$\mathbf{Z} = f(\tilde{\mathbf{D}}^{-1}\tilde{\mathbf{A}}\mathbf{X}\mathbf{W}), \tag{1}$$

where $\tilde{\mathbf{A}} = \mathbf{A} + \mathbf{I}$ is the adjacency matrix of the graph with added self-loops, $\tilde{\mathbf{D}}$ is its diagonal degree matrix with $\tilde{\mathbf{D}}_{ii} = \sum_j \tilde{\mathbf{A}}_{ij}$, $\mathbf{W} \in \mathbb{R}^{c\times c'}$ is a matrix of trainable graph convolution parameters, f is a nonlinear activation function, and $\mathbf{Z} \in \mathbb{R}^{n\times c'}$ is the output activation matrix.

The graph convolution can be separated into four steps. First, a linear feature transformation is applied to the node information matrix by $\mathbf{XW}$, mapping the c feature channels to c' channels in the next layer. The filter weights $\mathbf{W}$ are shared among all vertices. The second step, $\tilde{\mathbf{A}}\mathbf{Y}$ where $\mathbf{Y} := \mathbf{XW}$ propagates node information to neighboring vertices as well as the node itself. To see this, we notice that $(\tilde{\mathbf{A}}\mathbf{Y})_i = \sum_j \tilde{\mathbf{A}}_{ij}\mathbf{Y}_j = \mathbf{Y}_i + \sum_{j\in\Gamma(i)} \mathbf{Y}_j$, i.e., the i^{th} row of the resulting matrix is the summation of $\mathbf{Y}_i$ itself and $\mathbf{Y}_j$ from i's neighboring nodes. The third step normalizes each row i by multiplying $\tilde{\mathbf{D}}^{-1}_{ii}$, in order to keep a fixed feature scale after graph convolution. The last step applies a point-wise nonlinear activation function f and outputs the graph convolution results.

The graph convolution aggregates node information in local neighborhoods to extract local substructure information. To extract multi-scale substructure features, we stack multiple graph convolution layers (1) as follows

$$\mathbf{Z}^{t+1} = f(\tilde{\mathbf{D}}^{-1}\tilde{\mathbf{A}}\mathbf{Z}^t\mathbf{W}^t), \tag{2}$$

where $\mathbf{Z}^0 = \mathbf{X}$, $\mathbf{Z}^t \in \mathbb{R}^{n\times c_t}$ is the output of the t^{th} graph convolution layer, c_t is the number of output channels of layer t, and $\mathbf{W}^t \in \mathbb{R}^{c_t\times c_{t+1}}$ maps c_t channels to c_{t+1} channels. After multiple graph convolution layers, we add a layer to concatenate the output $\mathbf{Z}^t, t = 1, \ldots, h$ horizontally to form a concatenated output, written as $\mathbf{Z}^{1:h} := [\mathbf{Z}^1, \ldots, \mathbf{Z}^h]$, where h is the number of graph convolution layers and $\mathbf{Z}^{1:h} \in \mathbb{R}^{n\times\sum_1^h c_t}$. In the concatenated output $\mathbf{Z}^{1:h}$, each row can be regarded as a "feature descriptor" of a vertex, encoding its multi-scale local substructure information.

Note that our graph convolution form is similar to the spectral filter proposed in (Kipf and Welling 2016) – it also propagates neighboring nodes to center except for using a different propagation matrix. In fact, our graph convolution form (1) also has a spectral formulation. We will discuss their relations and differences in detail in the supplementary material and show that our graph convolution form is a theoretically closer approximation to the Weisfeiler-Lehman algorithm (Weisfeiler and Lehman 1968). But firstly, let's

look at why (1) can be used for graph classification. Below, we show that $\mathbf{Z}^{1:h}$ contains rich structural information of the graph by drawing analogies with two popular graph kernels. Moreover, we will show that the last graph convolution layer's output $\mathbf{Z}^h$ can be used to sort the graph vertices in a consistent order based on vertices' structural roles.

Connection with Weisfeiler-Lehman subtree kernel The Weisfeiler-Lehman subtree kernel (Shervashidze et al. 2011) is a state-of-the-art graph kernel, which leverages the Weisfeiler-Lehman (WL) algorithm (Weisfeiler and Lehman 1968) as a subroutine to extract multi-scale subtree features for graph classfication.

The basic idea of WL is to concatenate a vertex's color with its 1-hop neighbors' colors as the vertex's WL signature, and then sort the signature strings lexicographically to assign new colors. Vertices with the same signature are assigned the same new color. A WL signature characterizes the height-1 subtree rooted at a vertex. The procedure is repeated until the colors converge or reaching some maximum iteration h. In the end, vertices with the same converged color share the same structural role within the graph and cannot be further distinguished. A vertex color at any iteration t uniquely corresponds to a height-t subtree rooted at the vertex.

WL is widely used in graph isomorphism checking: if two graphs are isomorphic, they will have the same multiset of WL colors at any iteration. The WL subtree kernel uses this idea to measure the similarity between two graphs G and G' as follows:

$$k(G, G') = \sum_{t=0}^{h} \sum_{v \in V} \sum_{v' \in V'} \delta(\mathrm{c}^t(v), \mathrm{c}^t(v')), \qquad (3)$$

where $\mathrm{c}^t(v)$ is the (integer) color of vertex v in the t^{th} iteration, and $\delta(x, y) = 1$ if $x = y$ and 0 otherwise. That is, it counts the common colors of two graphs in all iterations. The intuition is that two graphs are similar if they have many common subtrees rooted at their vertices, which are characterized by colors (same color $\Leftrightarrow$ same WL signature $\Leftrightarrow$ same rooted subtree). The WL subtree kernel counts the common colors until iteration h in order to compare two graphs at multiple scales.

To show the relation between the graph convolution in (1) and the WL subtree kernel, we rewrite $\mathbf{Y} := \mathbf{XW}$, and decompose (1) row-wise as follows:

$$\mathbf{Z}_i = f([\tilde{\mathbf{D}}^{-1}\tilde{\mathbf{A}}]_i \mathbf{Y}) = f(\tilde{\mathbf{D}}_{ii}^{-1}(\mathbf{Y}_i + \textstyle\sum_{j \in \Gamma(i)} \mathbf{Y}_j)). \qquad (4)$$

In (4), we can view $\mathbf{Y}_i$ as a **continuous color** of vertex i. In analogy to WL, (4) also aggregates $\mathbf{Y}_i$ and its neighboring colors $\mathbf{Y}_j$ into a **WL signature vector** $\tilde{\mathbf{D}}_{ii}^{-1}(\mathbf{Y}_i + \sum_{j \in \Gamma(i)} \mathbf{Y}_j)$. The nonlinear function f maps unique WL signature vectors to unique continuous new colors if f is injective. Therefore, the graph convolution (1) may be viewed as a "soft" version of the WL algorithm.

Now we have seen that the graph convolution output $\mathbf{Z}_i^t$ for vertex i is a continuous and vectorized version of the integer colors $\mathrm{c}^t(v)$ in (3). DGCNN differs from the WL subtree kernel in how it uses the colors. DGCNN horizontally concatenates these colors $\mathbf{Z}^t$ as $\mathbf{Z}^{1:h}$, and trains models on them instead of calculating kernel functions as in (3).

The soft version of WL has two benefits over the original WL. First, the convolution parameters $\mathbf{W}$ allow hierarchical feature extraction of nodes' original information matrix $\mathbf{X}$, and are trainable through backpropagation, enabling better expressing power than the WL subtree kernel. Second, the soft WL is easy to compute using sparse matrix multiplication, avoiding the need to read and sort the possibly very long WL signature strings.

Connection with propagation kernel Instead of operating on integer node labels, the propagation kernel (PK) (Neumann et al. 2012; 2016) compares the label distributions between two graphs. PK is equipped with a diffusion update scheme:

$$\mathbf{L}^{t+1} = \mathbf{T}\mathbf{L}^t, \text{ where } \mathbf{T} = \mathbf{D}^{-1}\mathbf{A}. \qquad (5)$$

$\mathbf{T} = \mathbf{D}^{-1}\mathbf{A}$ is the transition matrix of a random walk on graph $\mathbf{A}$, and $\mathbf{L}^t \in \mathbb{R}^{n \times c}$ contains the c-dimensional label distribution vectors of the n vertices in the t^{th} iteration. In PK, initial labels are diffused over iterations. The final similarity is computed by mapping distribution vectors from all iterations into discrete bins based on locality-sensitive hashing, and counting common integer bins. PK has similar graph classification performance to WL and even better efficiency.

Our proposed graph convolution (1) adopts a propagation matrix $\tilde{\mathbf{D}}^{-1}\tilde{\mathbf{A}}$ which is very similar to that of PK, except for preserving nodes' self-information over iterations.

In summary, our graph convolution model (1) effectively mimics the behavior of two popular graph kernels, which helps explain its graph-level classification performance. The connection between discrete WL, continuous optimization, and random walks have also been studied in (Boldi et al. 2006; Kersting et al. 2014; Kipf and Welling 2016) etc.

2.2 The SortPooling layer

The main function of the SortPooling layer is to sort the feature descriptors, each of which represents a vertex, in a consistent order before feeding them into traditional 1-D convolutional and dense layers.

The question is by what order should we sort the vertices? In image classification, pixels are naturally arranged with some spatial order. In text classification, we can use dictionary order to sort words. In graphs, we can sort vertices according to their **structural roles** within the graph. (Niepert, Ahmed, and Kutzkov 2016) used graph labeling methods, especially WL, to sort vertices in a preprocessing step, since the final WL colors define an ordering based on graph topology. This vertex ordering imposed by WL is consistent across graphs, meaning that vertices in two different graphs will be assigned similar relative positions if they have similar structural roles in their respective graphs. Consequently, neural networks can read graph nodes in sequence and learn meaningful models.

In DGCNN, we also aim to use such WL colors to sort vertices. Luckily, we have seen that the outputs of the graph convolution layers are exactly the **continuous WL colors** $\mathbf{Z}^t$, $t = 1, \ldots, h$. We can use them to sort the vertices.

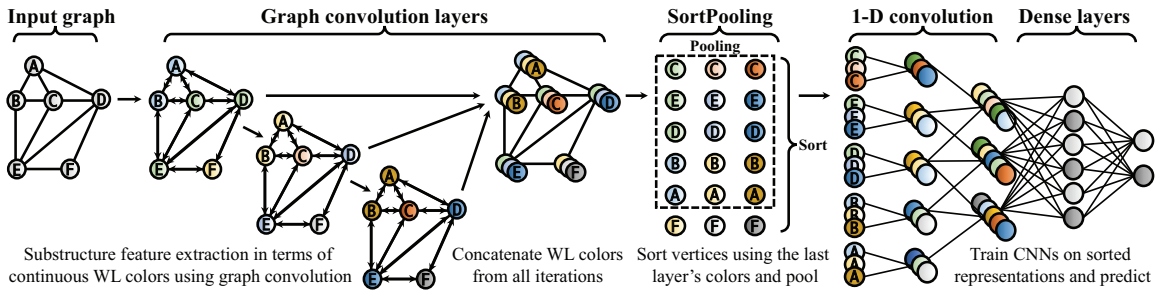

Figure 2: The overall structure of DGCNN. An input graph of arbitrary structure is first passed through multiple graph convolution layers where node information is propagated between neighbors. Then the vertex features are sorted and pooled with a SortPooling layer, and passed to traditional CNN structures to learn a predictive model. Features are visualized as colors.

Following this idea, we invent a novel SortPooling layer. In this layer, the input is an $n \times \sum_1^h c_t$ tensor $\mathbf{Z}^{1:h}$, where each row is a vertex's feature descriptor and each column is a feature channel. The output of SortPooling is a $k \times \sum_1^h c_t$ tensor, where k is a user-defined integer. In the SortPooling layer, the input $\mathbf{Z}^{1:h}$ is first sorted row-wise according to $\mathbf{Z}^h$. We can regard this last layer's output as the vertices' **most refined continuous WL colors**, and sort all the vertices using these final colors. This way, a consistent ordering is imposed for graph vertices, making it possible to train traditional neural networks on the sorted graph representations. Ideally, we need the graph convolution layers to be deep enough (meaning h is large), so that $\mathbf{Z}^h$ is able to partition vertices into different colors/groups as finely as possible.

The vertex order based on $\mathbf{Z}^h$ is calculated by first sorting vertices using the last channel of $\mathbf{Z}^h$ in a descending order. If two vertices have the same value in the last channel, the tie is broken by comparing their values in the second to last channel, and so on. If ties still exist, we continue comparing their values in $\mathbf{Z}_i^{h-1}$, $\mathbf{Z}_i^{h-2}$, and so on until ties are broken. Such an order is similar to the lexicographical order, except for comparing sequences from right to left.

In addition to sorting vertex features in a consistent order, the next function of SortPooling is to unify the sizes of the output tensors. After sorting, we truncate/extend the output tensor in the first dimension from n to k. The intention is to unify graph sizes, making graphs with different numbers of vertices unify their sizes to k. The unifying is done by deleting the last $n - k$ rows if $n > k$, or adding $k - n$ zero rows if $n < k$.

As a bridge between graph convolution layers and traditional layers, SortPooling has another great benefit in that it can **pass loss gradients back** to previous layers by remembering the sorted order of its input, making the training of previous layers' parameters feasible. In comparison, since (Niepert, Ahmed, and Kutzkov 2016) sorts vertices in the preprocessing step, their parameter training cannot take place before sorting. We show how to do backpropagation for SortPooling in the supplementary material.

2.3 Remaining layers

After SortPooling, we get a tensor $\mathbf{Z}^{\text{sp}}$ of size $k \times \sum_1^h c_t$ with each row representing a vertex and each column representing a feature channel. To train CNNs on them, we first reshape $\mathbf{Z}^{\text{sp}}$ into a $k(\sum_1^h c_t) \times 1$ vector row-wise. Then we add a 1-D convolutional layer with filter size and step $\sum_1^h c_t$, in order to sequentially apply filters on vertices' feature descriptors. After that, several MaxPooling layers and 1-D convolutional layers are added in order to learn local patterns on the node sequence. Finally, we add a fully-connected layer followed by a softmax layer.

3 Related Work

Graph Kernels Graph kernels make kernel machines such as SVMs feasible for graph classification by computing some positive semidefinite graph similarity measures, which have achieved state-of-the-art classification results on many graph datasets (Vishwanathan et al. 2010; Shervashidze et al. 2011). A pioneering work was introduced as the convolution kernel in (Haussler 1999), which decomposes graphs into small substructures and computes kernel functions by adding up the pair-wise similarities between these components. Common types of substructures include walks (Vishwanathan et al. 2010), subgraphs (Kriege and Mutzel 2012), paths (Borgwardt and Kriegel 2005), and subtrees (Shervashidze et al. 2011; Neumann et al. 2016). (Orsini, Frasconi, and De Raedt 2015) reformulated many well-known substructure-based kernels in a general way called graph invariant kernels. (Yanardag and Vishwanathan 2015) proposed deep graph kernels which learn latent representations of substructures to leverage their dependency information. Convolution kernels compare two graphs based on all pairs of their substructures. Assignment kernels, on the other hand, tend to find a correspondence between parts of two graphs. (Bai et al. 2015) proposed aligned subtree kernels incorporating explicit subtree correspondences. (Kriege, Giscard, and Wilson 2016) proposed the optimal assignment kernels for a type of hierarchy-induced kernels. Most existing graph kernels focus on comparing small local patterns. Recent studies show comparing graphs more globally can improve the performance (Kondor and Pan 2016;

Morris, Kersting, and Mutzel 2017). (Dai, Dai, and Song 2016) represented each graph using a latent variable model and then explicitly embedded them into feature spaces in a way similar to graphical model inference. The results compared favorably with standard graph kernels in both accuracy and efficiency.

DGCNN is closely related to a type of graph kernels based on structure propagation, especially the Weisfeiler-Lehman (WL) subtree kernel (Shervashidze et al. 2011) and the propagation kernel (PK) (Neumann et al. 2016). To encode the structural information of graphs, WL and PK iteratively update a node's feature based on its neighbors' features. WL operates on hard vertex labels, while PK operates on soft label distributions. As this operation can be efficiently implemented as a random walk, these kernels are efficient on large graphs. Compared to WL and PK, DGCNN has additional parameters $\mathbf{W}$ between propagations which are trained through end-to-end optimization. This allows supervised feature learning from the label information, making it different from the two-stage framework of graph kernels.

Neural networks for graphs There are two lines of research on generalizing neural networks to graphs: 1) given a **single graph structure**, infer signals in graph forms or infer labels of individual nodes (Scarselli et al. 2009; Bruna et al. 2013; Henaff, Bruna, and LeCun 2015; Li et al. 2015; Defferrard, Bresson, and Vandergheynst 2016; Kipf and Welling 2016); and 2) given a set of graphs with **different structure and sizes**, learn to predict the class labels of unseen graphs (the graph classification problem) (Duvenaud et al. 2015; Atwood and Towsley 2016; Niepert, Ahmed, and Kutzkov 2016; Simonovsky and Komodakis 2017; Lei et al. 2017). In this paper, we focus on the *second* problem, which is more difficult because the graph structure is not fixed, nor is the number of nodes within each graph. Moreover, unlike the first problem where vertices from different graphs have fixed indices or are in correspondence, a problem-specific vertex ordering is often unavailable in the second problem.

Our work is related to a pioneering work using CNNs for graph classification, called PATCHY-SAN (Niepert, Ahmed, and Kutzkov 2016). To mimic the behavior of CNNs on images, PATCHY-SAN first extracts fixed-sized local patches from vertices' neighborhoods as the receptive fields for convolution filters. Then, in order apply CNNs on these patches, PATCHY-SAN uses **external software**, such as the graph canonization tool NAUTY (McKay and Piperno 2014), to define a global vertex order for the whole graph and a local order for each patch in a preprocessing step. After that, graphs are transformed to ordered tensor representations and a CNN is trained on these tensors. Although achieving competitive results with graph kernels, the drawbacks of this approach include heavy data preprocessing and reliance on external software. Our DGCNN inherits its idea of imposing an order for graph vertices, but integrates this step into the network structure, namely the SortPooling layer.

DGCNN is also related to the Graph neural networks (GNNs) (Scarselli et al. 2009; Li et al. 2015), Diffusion-CNN (Atwood and Towsley 2016), and Neural Graph Fingerprints (Duvenaud et al. 2015) in how to extract node features. However, to perform graph-level classification, GNNs supervise a single node, and Diffusion-CNN and Neural Graph Fingerprints use summed node features. In comparison, DGCNN sorts vertices with some order and applies traditional CNNs on the ordered representations, which keeps much more information and enables learning from global graph topology.

4 Discussion

One important criterion of graph neural network design is that the network should map isomorphic graphs to the same representation and output the same prediction, otherwise any permutation in the adjacency matrix could result in a different prediction for a same graph. For summing-based methods, this is not an issue, as summing is invariant to vertex permutation. However, for sorting-based methods DGCNN and PATCHY-SAN (Niepert, Ahmed, and Kutzkov 2016), additional care is required. To ensure that isomorphic graphs are preprocessed to the same tensor, PATCHY-SAN first uses the WL algorithm and then leverages NAUTY, a graph canonization tool (McKay and Piperno 2014). Although NAUTY is efficient enough for small graphs, the problem of graph canonization is theoretically at least as computationally hard as graph isomorphism checking.

In comparison, we show that such a graph canonization step can be avoided in DGCNN. DGCNN sorts vertices using the last graph convolution layer's outputs, which we show can be viewed as the continuous colors output by a "soft" WL. Thus, DGCNN is able to sort vertices as a **by-product** of graph convolution, which avoids explicitly running the WL algorithm like PATCHY-SAN. Moreover, due to the sorting scheme in SortPooling, graph canonization is no longer needed.

Theorem 1. *In* DGCNN*, if two graphs G_1 and G_2 are isomorphic, their graph representations after SortPooling are the same.*

Proof. Notice that the first phase's graph convolutions are invariant to vertex indexing. Thus if G_1 and G_2 are isomorphic, they will have the same multiset of vertex feature descriptors after graph convolution. Since SortPooling sorts vertices in such a way that two vertices have a tie if and only if they have exactly the same feature descriptor, the sorted representation is invariant to which of the two vertices is ranked higher. Hence, G_1 and G_2 have the same sorted representation after SortPooling. □

Therefore, DGCNN needs to explicitly run **neither** WL **nor** NAUTY, which frees us from data preprocessing and external software, and provides a pure neural network architecture for end-to-end graph classification. We also comment that although DGCNN sorts vertices dynamically during training, the vertex order will gradually become stable with increasing training epochs. This is because the parameters $\mathbf{W}$ are shared among all vertices. The updating of $\mathbf{W}$ will increase or decrease the continuous WL colors of all vertices simultaneously. Moreover, the learning rate of $\mathbf{W}$ is iteratively decayed during training, making the overall vertex order stable over the course.

Table 1: Comparison with graph kernels.

Dataset	MUTAG	PTC	NCI1	PROTEINS	D&D
Nodes (max)	28	109	111	620	5748
Nodes (avg.)	17.93	25.56	29.87	39.06	284.32
Graphs	188	344	4110	1113	1178
DGCNN	**85.83±1.66**	58.59±2.47	74.44±0.47	**75.54±0.94**	**79.37±0.94**
GK	81.39±1.74	55.65±0.46	62.49±0.27	71.39±0.31	74.38±0.69
RW	79.17±2.07	55.91±0.32	>3 days	59.57±0.09	>3 days
PK	76.00±2.69	**59.50±2.44**	82.54±0.47	73.68±0.68	78.25±0.51
WL	84.11±1.91	57.97±2.49	**84.46±0.45**	74.68±0.49	78.34±0.62

5 Experimental Results

We conduct experiments on benchmark datasets to evaluate the performance of DGCNN against state-of-the-art graph kernels and other deep learning approaches. The code and data are available at https://github.com/muhanzhang/DGCNN.

5.1 Comparison with graph kernels

Datasets We use five benchmark bioinformatics datasets to compare the graph classification accuracy of DGCNN with graph kernels. The datasets are: MUTAG, PTC, NCI1, PROTEINS, D&D. We include detailed dataset information in the supplementary material. All the five datasets are vertex-labeled.

Baselines and experimental setting We compare DGCNN with four graph kernels: the graphlet kernel (GK) (Shervashidze et al. 2009), the random walk kernel (RW) (Vishwanathan et al. 2010), the propagation kernel (PK) (Neumann et al. 2012), and the Weisfeiler-Lehman subtree kernel (WL) (Shervashidze et al. 2011). Due to the large literature, we could not compare to every graph kernel, but to some classical ones and those closely related to our approach. Following the conventional settings, we performed 10-fold cross validation with LIBSVM (Chang and Lin 2011) (9 folds for training and 1 fold for testing) using one training fold for hyperparameter searching, and repeated the experiments for 10 times (thus **100 runs per dataset**). The average accuracies and their standard deviations are reported. We searched the height parameter of WL and PK in $\{0, 1, 2, 3, 4, 5\}$, and set the bin width w of PK to 0.001. We set the size of the graphlets for GK to 3. We set the decay parameter λ of RW to the largest power of 10 that is smaller than the reciprocal of the squared maximum node degree as suggested in (Shervashidze et al. 2011).

For the proposed DGCNN, to make a fair comparison with graph kernels, we used a **single network structure** on all datasets, and ran DGCNN using exactly the same folds as used in graph kernels in all the **100 runs** of each dataset. The network has four graph convolution layers with 32, 32, 32, 1 output channels, respectively. For convenience, we set the last graph convolution to have one channel and only used this single channel for sorting. We set the k of SortPooling such that 60% graphs have nodes more than k. The remaining layers consist of two 1-D convolutional layers and one dense layer. The first 1-D convolutional layer has 16 output channels followed by a MaxPooling layer with filter size 2 and step size 2. The second 1-D convolutional layer has 32 output channels, filter size 5 and step size 1. The dense layer has 128 hidden units followed by a softmax layer as the output layer. A dropout layer with dropout rate 0.5 is used after the dense layer. We used the hyperbolic tangent function (tanh) as the nonlinear function in graph convolution layers, and rectified linear units (ReLU) in other layers. Stochastic gradient descent (SGD) with the ADAM updating rule (Kingma and Ba 2014) was used for optimization. The only hyperparameters we optimized are the learning rate and the number of training epochs (details in the supplementary material). We implemented SortPooling and graph convolution layers using Torch (Collobert, Kavukcuoglu, and Farabet 2011) as standard *nn* modules, which can be seamlessly added to existing Torch architectures.

Results Table 1 lists the results. As we can see, although a single structure was used for all datasets, DGCNN achieved highly competitive results with the compared graph kernels, including achieving the highest accuracies on MUTAG, PROTEINS, and D&D. Compared to WL, DGCNN has higher accuracies on all datasets except for NCI1, indicating that DGCNN is able to utilize node and structure information more effectively. Note that the height parameters in PK and WL were tuned individually for each dataset by searching from $\{0,1,2,3,4,5\}$, while DGCNN used a **single height** $h=4$ for **all datasets**. Thus, we expect better performance if we use different structures for different datasets.

We compare the efficiency of DGCNN with one of the most efficient graph kernels, the WL kernel, on D&D, the benchmark dataset with the largest graph size. We omit the SVM training time of WL, since the computing time is dominated by the kernel computation. WL takes 252 seconds to construct the kernel matrix. For DGCNN, the training time varies with the iteration number. Here we limit the iteration number to 10, under which condition DGCNN already achieves comparable or better accuracy than WL. DGCNN takes 156 seconds, meaning that it is able to achieve competitive efficiency with the fastest graph kernels. Moreover, DGCNN is trained through SGD, avoiding the at least quadratic complexity w.r.t. the number of graphs required for graph kernels. Therefore, we expect to see a much greater advantage when applying to industrial-scale graph datasets.

Table 2: Comparison with other deep learning approaches.

Dataset	NCI1	PROTEINS	D&D	COLLAB	IMDB-B	IMDB-M
Nodes (max)	111	620	5748	492	136	89
Nodes (avg.)	29.87	39.06	284.32	74.49	19.77	13.00
Graphs	4110	1113	1178	5000	1000	1500
DGCNN	74.44±0.47	**75.54±0.94**	**79.37±0.94**	**73.76±0.49**	70.03±0.86	**47.83±0.85**
PSCN	76.34±1.68	75.00±2.51	76.27±2.64	72.60±2.15	**71.00±2.29**	45.23±2.84
DCNN	56.61±1.04	61.29±1.60	58.09±0.53	52.11±0.71	49.06±1.37	33.49±1.42
ECC	**76.82**	–	72.54	–	–	–
DGK	62.48±0.25	71.68±0.50	–	73.09±0.25	66.96±0.56	44.55±0.52
DGCNN (sum)	69.00±0.48	**76.26±0.24**	78.72±0.40	69.45±0.11	51.69±1.27	42.76±0.97

5.2 Comparison with other deep approaches

Datasets We compare DGCNN with other deep learning approaches for graph classification on six datasets, including three benchmark bioinformatics datasets: NCI1, PROTEINS, and D&D, as well as three social network datasets: COLLAB, IMDB-B, IMDB-M (Yanardag and Vishwanathan 2015). Graphs in these social network datasets do not have vertex labels, thus are pure structures. We exclude the two smallest bioinformatics datasets: MUTAG and PTC, which only have a few hundred examples, since deep learning methods easily overfit them, reporting abnormally high variance in previous works (Niepert, Ahmed, and Kutzkov 2016).

Baselines and experimental setting We compare DGCNN with four other deep learning approaches: including three recent neural network approaches for graph classification (PSCN, DCNN, and ECC), and a deep graph kernel approach (DGK). Among them, PATCHY-SAN (PSCN) (Niepert, Ahmed, and Kutzkov 2016) is the closest to ours. Diffusion-CNN (DCNN) (Atwood and Towsley 2016) uses diffusion graph convolutions to extract multi-scale substructure features. ECC (Simonovsky and Komodakis 2017) can be viewed as a hierarchical version of the Neural Fingerprints (Duvenaud et al. 2015). Both DCNN and ECC use summed node features for graph classification. The Deep Graphlet Kernel (DGK) (Yanardag and Vishwanathan 2015) learns substructure similarities via word embedding techniques. For PSCN, ECC, and DGK, we report the best results from the papers, as they were under the same setting as ours. For DCNN, since the original experiment split the training/validation/testing data equally, we redid the experiment using the standard setting. Among these methods, PSCN and ECC can leverage additional edge features. These augmented results are not reported here since edge features are missing from most graph datasets and all the other compared methods do not leverage edge features.

For DGCNN, we still use the same structure as when comparing with graph kernels, in order to show its robust performance across different datasets under a single structure. Since the new added social network datasets do not contain node labels, we set the k of SortPooling such that 90% graphs have nodes more than k in order to compensate the loss of node features.

Results Table 2 lists the results. DGCNN shows the highest accuracies on PROTEINS, D&D, COLLAB, and IMDB-M. Compared to PATCHY-SAN, the improvement of DGCNN can be explained as follows. 1) By letting gradient information backpropagate through SortPooling, DGCNN enables parameter training even before the sorting begins, thus achieving better expressibility. 2) By sorting nodes on the fly, DGCNN is less likely to overfit a particular node ordering. In comparison, PATCHY-SAN sticks to a predefined node ordering. Another huge advantage of DGCNN is that it provides a unified way to integrate preprocessing into a neural network structure. This frees us from using any external software.

DGCNN shows significant accuracy improvement over DCNN which uses summed node features for classification. Another summing-based method, ECC, is slightly better on NCI1, but much worse on D&D. These results meet our expectation since summing will lose much individual node and global topology information. Compared to DGK, DGCNN shows better performance on all the reported datasets.

To demonstrate the advantage of SortPooling over summing, we further list the results of DGCNN (sum), which replaces the SortPooling and later 1-D convolution layers in DGCNN with a summing layer. As we can see, the performance worsens a lot in most cases. We also conducted some supplementary experiments in order to further understand the effect of SortPooling, included in the supplementary material.

6 Conclusions

In this paper, we have proposed a novel neural network architecture, DGCNN, for graph classification. DGCNN has a number of advantages over existing graph neural networks. First, it directly accepts graph data as input without the need of first transforming graphs into tensors, making end-to-end gradient-based training possible. Second, it enables learning from global graph topology by sorting vertex features instead of summing them up, which is supported by a novel SortPooling layer. Finally, it achieves better performance than existing methods on many benchmark datasets. In the future, we would like to study different sorting schemes and apply our method to more datasets.

7 Acknowledgments

The work is supported in part by the DBI-1356669, SCH-1343896, III-1526012, and SCH-1622678 grants from the

National Science Foundation and grant 1R21HS024581 from the National Institute of Health.

References

Alex, K.; Sutskever, I.; and Hinton, G. E. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25*. 1097–1105.

Atwood, J., and Towsley, D. 2016. Diffusion-convolutional neural networks. In *Advances in Neural Information Processing Systems*, 1993–2001.

Bai, L.; Rossi, L.; Zhang, Z.; and Hancock, E. 2015. An aligned subtree kernel for weighted graphs. In *International Conference on Machine Learning*, 30–39.

Boldi, P.; Lonati, V.; Santini, M.; and Vigna, S. 2006. Graph fibrations, graph isomorphism, and pagerank. *RAIRO-Theoretical Informatics and Applications* 40(2):227–253.

Borgwardt, K. M., and Kriegel, H.-P. 2005. Shortest-path kernels on graphs. In *5th IEEE International Conference on Data Mining*, 8–pp. IEEE.

Bruna, J.; Zaremba, W.; Szlam, A.; and LeCun, Y. 2013. Spectral networks and locally connected networks on graphs. *arXiv preprint arXiv:1312.6203*.

Chang, C.-C., and Lin, C.-J. 2011. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology* 2:27:1–27:27. Software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm.

Collobert, R.; Kavukcuoglu, K.; and Farabet, C. 2011. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*.

Cui, Z.; Chen, W.; and Chen, Y. 2016. Multi-scale convolutional neural networks for time series classification. *arXiv preprint arXiv:1603.06995*.

Dai, H.; Dai, B.; and Song, L. 2016. Discriminative embeddings of latent variable models for structured data. In *Proceedings of The 33rd International Conference on Machine Learning*, 2702–2711.

Defferrard, M.; Bresson, X.; and Vandergheynst, P. 2016. Convolutional neural networks on graphs with fast localized spectral filtering. In *Advances in Neural Information Processing Systems*, 3837–3845.

Duvenaud, D. K.; Maclaurin, D.; Iparraguirre, J.; Bombarell, R.; Hirzel, T.; Aspuru-Guzik, A.; and Adams, R. P. 2015. Convolutional networks on graphs for learning molecular fingerprints. In *Advances in neural information processing systems*, 2224–2232.

Haussler, D. 1999. Convolution kernels on discrete structures. Technical report, Citeseer.

Henaff, M.; Bruna, J.; and LeCun, Y. 2015. Deep convolutional networks on graph-structured data. *arXiv preprint arXiv:1506.05163*.

Kersting, K.; Mladenov, M.; Garnett, R.; and Grohe, M. 2014. Power iterated color refinement. In *AAAI*, 1904–1910.

Kingma, D., and Ba, J. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Kipf, T. N., and Welling, M. 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*.

Kondor, R., and Pan, H. 2016. The multiscale laplacian graph kernel. In *Advances in Neural Information Processing Systems*, 2982–2990.

Kriege, N., and Mutzel, P. 2012. Subgraph matching kernels for attributed graphs. In *Proceedings of the 29th International Conference on Machine Learning (ICML-12)*, 1015–1022.

Kriege, N. M.; Giscard, P.-L.; and Wilson, R. 2016. On valid optimal assignment kernels and applications to graph classification. In *Advances in Neural Information Processing Systems*, 1615–1623.

Lei, T.; Jin, W.; Barzilay, R.; and Jaakkola, T. 2017. Deriving neural architectures from sequence and graph kernels. In Precup, D., and Teh, Y. W., eds., *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, 2024–2033. International Convention Centre, Sydney, Australia: PMLR.

Li, Y.; Tarlow, D.; Brockschmidt, M.; and Zemel, R. 2015. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493*.

McKay, B. D., and Piperno, A. 2014. Practical graph isomorphism, {II}. *Journal of Symbolic Computation* 60(0):94 – 112.

Mikolov, T.; Chen, K.; Corrado, G.; and Dean, J. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Graves, A.; Antonoglou, I.; Wierstra, D.; and Riedmiller, M. 2013. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.

Morris, C.; Kersting, K.; and Mutzel, P. 2017. Global weisfeiler-lehman graph kernels. *arXiv preprint arXiv:1703.02379*.

Neumann, M.; Patricia, N.; Garnett, R.; and Kersting, K. 2012. Efficient graph kernels by randomization. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, 378–393. Springer.

Neumann, M.; Garnett, R.; Bauckhage, C.; and Kersting, K. 2016. Propagation kernels: efficient graph kernels from propagated information. *Machine Learning* 102(2):209–245.

Niepert, M.; Ahmed, M.; and Kutzkov, K. 2016. Learning convolutional neural networks for graphs. In *Proceedings of the 33rd annual international conference on machine learning. ACM*.

Orsini, F.; Frasconi, P.; and De Raedt, L. 2015. Graph invariant kernels. In *IJCAI*, 3756–3762.

Scarselli, F.; Gori, M.; Tsoi, A. C.; Hagenbuchner, M.; and Monfardini, G. 2009. The graph neural network model. *IEEE Transactions on Neural Networks* 20(1):61–80.

Shervashidze, N.; Vishwanathan, S.; Petri, T.; Mehlhorn, K.; and Borgwardt, K. M. 2009. Efficient graphlet kernels for large graph comparison. In *AISTATS*, volume 5, 488–495.

Shervashidze, N.; Schweitzer, P.; Leeuwen, E. J. v.; Mehlhorn, K.; and Borgwardt, K. M. 2011. Weisfeiler-lehman graph kernels. *Journal of Machine Learning Research* 12(Sep):2539–2561.

Simonovsky, M., and Komodakis, N. 2017. Dynamic edge-conditioned filters in convolutional neural networks on graphs. *arXiv preprint arXiv:1704.02901*.

Vishwanathan, S. V. N.; Schraudolph, N. N.; Kondor, R.; and Borgwardt, K. M. 2010. Graph kernels. *Journal of Machine Learning Research* 11(Apr):1201–1242.

Weisfeiler, B., and Lehman, A. 1968. A reduction of a graph to a canonical form and an algebra arising during this reduction. *Nauchno-Technicheskaya Informatsia* 2(9):12–16.

Yanardag, P., and Vishwanathan, S. 2015. Deep graph kernels. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 1365–1374. ACM.