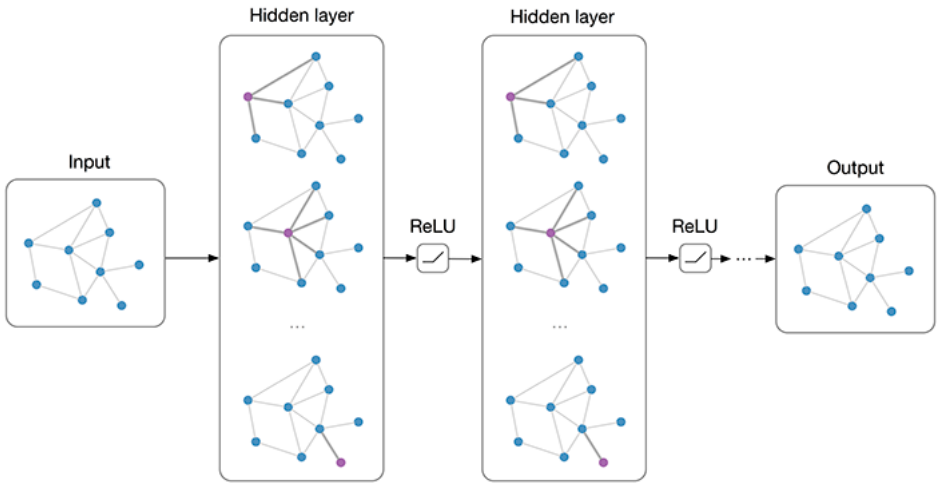
GRAPH CONVOLUTIONAL NETWORKS

THOMAS KIPF, 30 SEPTEMBER 2016



Multi-layer Graph Convolutional Network (GCN) with first-order filters.

Overview

Many important real-world datasets come in the form of graphs or networks: social networks, knowledge graphs, protein-interaction networks, the World Wide Web, etc. (just to name a few). Yet, until recently, very little attention has been devoted to the generalization of neural network models to such structured datasets.

In the last couple of years, a number of papers re-visited this problem of generalizing neural networks to work on arbitrarily structured graphs (Bruna et al., ICLR 2014; Henaff et al., 2015; Duvenaud et al., NIPS 2015; Li et al., ICLR 2016; Defferrard et al., NIPS 2016; Kipf & Welling, ICLR 2017), some of them now achieving very promising results in domains that have previously been dominated by, e.g., kernel-based methods, graph-based regularization techniques and others.

In this post, I will give a brief overview of recent developments in this field and point out strengths and drawbacks of various approaches. The discussion here will mainly focus on two recent papers:

- Kipf & Welling (ICLR 2017), Semi-Supervised Classification with Graph Convolutional Networks (disclaimer: I'm the first author)
- Defferrard et al. (NIPS 2016), Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering

and a review/discussion post by Ferenc Huszar: How powerful are Graph Convolutions? that discusses some limitations of these kinds of models. I wrote a short comment on Ferenc's review here (at the very end of this post).

Outline

- Short introduction to neural network models on graphs
- Spectral graph convolutions and *Graph Convolutional Networks* (GCNs)

- Demo: Graph embeddings with a simple 1st-order GCN model
- GCNs as differentiable generalization of the *Weisfeiler-Lehman* algorithm

If you're already familiar with GCNs and related methods, you might want to jump directly to Embedding the karate club network.

How powerful are Graph Convolutional Networks?

Recent literature

Generalizing well-established neural models like RNNs or CNNs to work on arbitrarily structured graphs is a challenging problem. Some recent papers introduce problem-specific specialized architectures (e.g. Duvenaud et al., NIPS 2015; Li et al., ICLR 2016; Jain et al., CVPR 2016), others make use of graph convolutions known from spectral graph theory[1] (Bruna et al., ICLR 2014; Henaff et al., 2015) to define parameterized filters that are used in a multi-layer neural network model, akin to "classical" CNNs that we know and love.

More recent work focuses on bridging the gap between fast heuristics and the slow[2], but somewhat more principled, spectral approach. Defferrard et al. (NIPS 2016) approximate smooth filters in the spectral domain using Chebyshev polynomials with free parameters that are learned in a neural network-like model. They achieve convincing results on regular domains (like MNIST), closely approaching those of a simple 2D CNN model.

In Kipf & Welling (ICLR 2017), we take a somewhat similar approach and start from the framework of spectral graph convolutions, yet introduce simplifications (we will get to those later in the post) that in many cases allow both for significantly faster training times and higher predictive accuracy, reaching state-of-the-art classification results on a number of benchmark graph datasets.

GCNs Part I: Definitions

Currently, most graph neural network models have a somewhat universal architecture in common. I will refer to these models as *Graph Convolutional Networks* (GCNs); convolutional, because filter parameters are typically shared over all locations in the graph (or a subset thereof as in Duvenaud et al., NIPS 2015).

For these models, the goal is then to learn a function of signals/features on a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ which takes as input:

- A feature description x_i for every node i; summarized in a $N \times D$ feature matrix X (N: number of nodes, D: number of input features)
- A representative description of the graph structure in matrix form; typically in the form of an adjacency matrix A (or some function thereof)

and produces a node-level output Z (an $N \times F$ feature matrix, where F is the number of output features per node). Graph-level outputs can be modeled by introducing some form of pooling operation (see, e.g. Duvenaud et al., NIPS 2015).

Every neural network layer can then be written as a non-linear function

$$H^{(l+1)} = f(H^{(l)}, A)\,,$$

with $H^{(0)} = X$ and $H^{(L)} = Z$ (or z for graph-level outputs), L being the number of layers. The specific models then differ only in how $f(\cdot, \cdot)$ is chosen and parameterized.

GCNs Part II: A simple example

As an example, let's consider the following very simple form of a layer-wise propagation rule:

$$f(H^{(l)}, A) = \sigma\left(AH^{(l)}W^{(l)}\right)\,,$$

where $W^{(l)}$ is a weight matrix for the l-th neural network layer and $\sigma(\cdot)$ is a non-linear activation function like the ReLU. Despite its simplicity this model is already quite powerful (we'll come to that in a moment).

But first, let us address two limitations of this simple model: multiplication with A means that, for every node, we sum up all the feature vectors of all neighboring nodes but not the node itself (unless there are self-loops in the graph). We can "fix" this by enforcing self-loops in the graph: we simply add the identity matrix to A.

The second major limitation is that A is typically not normalized and therefore the multiplication with A will completely change the scale of the feature vectors (we can understand that by looking at the eigenvalues of A). Normalizing A such that all rows sum to one, i.e. $D^{-1}A$, where D is the diagonal node degree matrix, gets rid of this

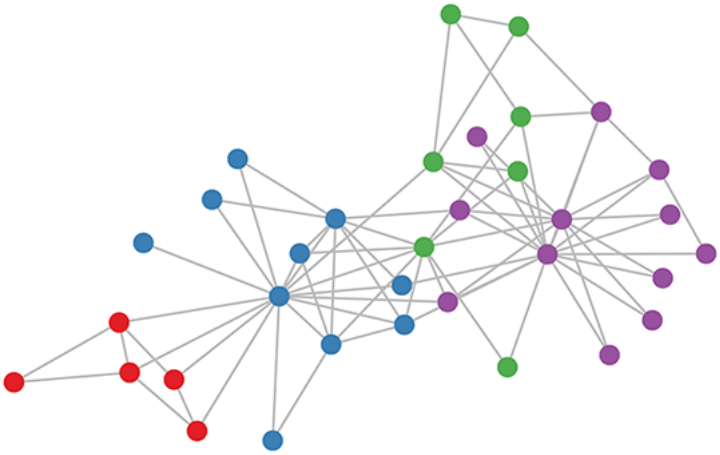
problem. Multiplying with $D^{-1}A$ now corresponds to taking the average of neighboring node features. In practice, dynamics get more interesting when we use a symmetric normalization, i.e. $D^{-\frac{1}{2}}AD^{-\frac{1}{2}}$ (as this no longer amounts to mere averaging of neighboring nodes). Combining these two tricks, we essentially arrive at the propagation rule introduced in Kipf & Welling (ICLR 2017):

$$f(H^{(l)}, A) = \sigma\left(\hat{D}^{-\frac{1}{2}}\hat{A}\hat{D}^{-\frac{1}{2}}H^{(l)}W^{(l)}\right) ,$$

with $\hat{A} = A + I$, where I is the identity matrix and $\hat{D}$ is the diagonal node degree matrix of $\hat{A}$.

In the next section, we will take a closer look at how this type of model operates on a very simple example graph: Zachary's karate club network (make sure to check out the Wikipedia article!).

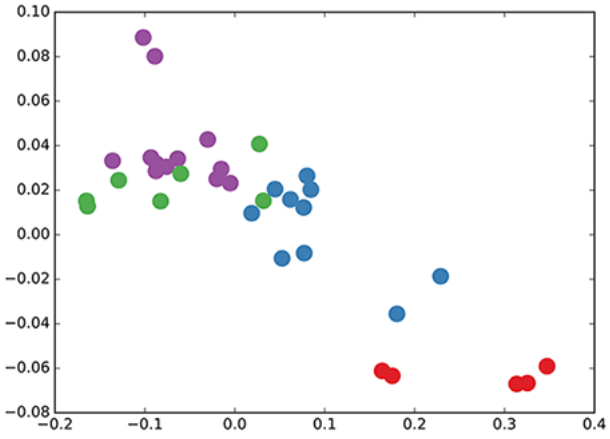
GCNs Part III: Embedding the karate club network



Karate club graph, colors denote communities obtained via modularity-based clustering (Brandes et al., 2008).

Let's take a look at how our simple GCN model (see previous section or Kipf & Welling, ICLR 2017) works on a well-known graph dataset: Zachary's karate club network (see Figure above).

We take a 3-layer GCN with randomly initialized weights. Now, even before training the weights, we simply insert the adjacency matrix of the graph and $X = I$ (i.e. the identity matrix, as we don't have any node features) into the model. The 3-layer GCN now performs three propagation steps during the forward pass and effectively convolves the 3rd-order neighborhood of every node (all nodes up to 3 "hops" away). Remarkably, the model produces an embedding of these nodes that closely resembles the community-structure of the graph (see Figure below). Remember that we have initialized the weights completely at random and have not yet performed any training updates (so far)!



GCN embedding (with random weights) for nodes in the karate club network.

This might seem somewhat surprising. A recent paper on a model called DeepWalk (Perozzi et al., KDD 2014) showed that they can learn a very similar embedding in a complicated unsupervised training procedure. How is it possible to

get such an embedding more or less "for free" using our simple untrained GCN model?

We can shed some light on this by interpreting the GCN model as a generalized, differentiable version of the well-known Weisfeiler-Lehman algorithm on graphs. The (1-dimensional) Weisfeiler-Lehman algorithm works as follows[3]:

For all nodes $v_i \in \mathcal{G}$:

- Get features[4] $\{h_{v_j}\}$ of neighboring nodes $\{v_j\}$
- Update node feature $h_{v_i} \leftarrow \text{hash}\left(\sum_j h_{v_j}\right)$, where $\text{hash}(\cdot)$ is (ideally) an injective hash function

Repeat for k steps or until convergence.

In practice, the Weisfeiler-Lehman algorithm assigns a unique set of features for *most* graphs. This means that every node is assigned a feature that uniquely describes its role in the graph. Exceptions are highly regular graphs like grids, chains, etc. For most irregular graphs, this feature assignment can be used as a check for graph isomorphism (i.e. whether two graphs are identical, up to a permutation of the nodes).

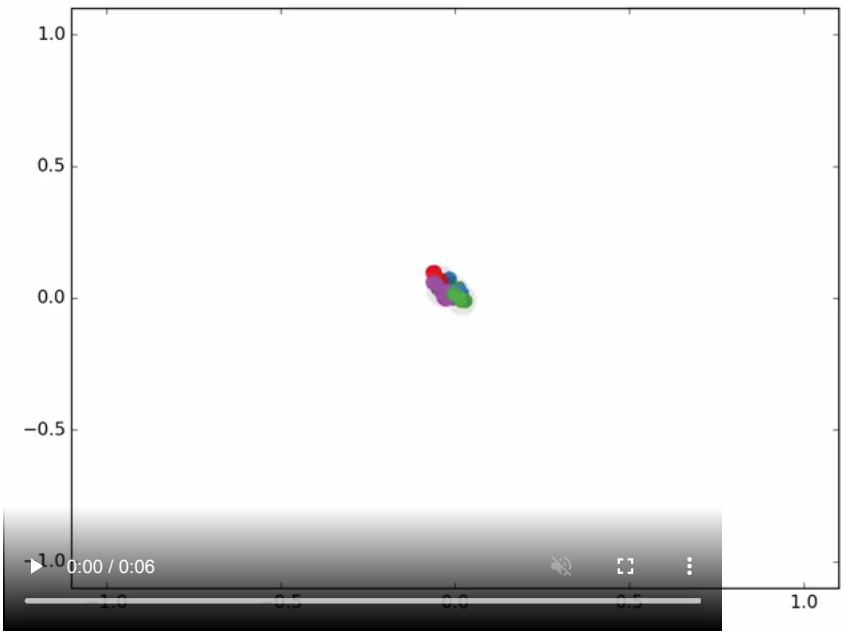
Going back to our Graph Convolutional layer-wise propagation rule (now in vector form):

$$h_{v_i}^{(l+1)} = \sigma\left(\sum_j \frac{1}{c_{ij}} h_{v_j}^{(l)} W^{(l)}\right),$$

where j indexes the neighboring nodes of v_i. c_{ij} is a normalization constant for the edge (v_i, v_j) which originates from using the symmetrically normalized adjacency matrix $D^{-\frac{1}{2}} A D^{-\frac{1}{2}}$ in our GCN model. We now see that this propagation rule can be interpreted as a differentiable and parameterized (with $W^{(l)}$) variant of the hash function used in the original Weisfeiler-Lehman algorithm. If we now choose an appropriate non-linearity and initialize the random weight matrix such that it is orthogonal (or e.g. using the initialization from Glorot & Bengio, AISTATS 2010), this update rule becomes stable in practice (also thanks to the normalization with c_{ij}). And we make the remarkable observation that we get meaningful smooth embeddings where we can interpret distance as (dis-)similarity of local graph structures!

GCNs Part IV: Semi-supervised learning

Since everything in our model is differentiable and parameterized, we can add some labels, train the model and observe how the embeddings react. We can use the semi-supervised learning algorithm for GCNs introduced in Kipf & Welling (ICLR 2017). We simply label one node per class/community (highlighted nodes in the video below) and start training for a couple of iterations[5]:



Semi-supervised classification with GCNs: Latent space dynamics for 300 training iterations with a single label per class. Labeled nodes are highlighted.

Note that the model directly produces a 2-dimensional latent space which we can immediately visualize. We observe that the 3-layer GCN model manages to linearly separate the communities, given only one labeled example per class. This is a somewhat remarkable result, given that the model received no feature description of the nodes. At the same time, initial node features *could* be provided, which is exactly what we do in the experiments described in our paper (Kipf & Welling, ICLR 2017) to achieve state-of-the-art classification results on a number of graph datasets.

Conclusion

Research on this topic is just getting started. The past several months have seen exciting developments, but we have probably only scratched the surface of these types of models so far. It remains to be seen how neural networks on graphs can be further taylored to specific types of problems, like, e.g., learning on directed or relational graphs, and how one can use learned graph embeddings for further tasks down the line, etc. This list is by no means exhaustive and I expect further interesting applications and extensions to pop up in the near future. Let me know in the comments below if you have some exciting ideas or questions to share!

THANKS TO THE FOLLOWING PEOPLE:

Max Welling, Taco Cohen, Chris Louizos and Karen Ullrich (for many discussions and feedback both on the paper and this blog post). Also I'd like to thank Ferenc Huszar for highlighting some drawbacks of these kinds of models.

A NOTE ON COMPLETENESS

This blog post constitutes by no means an exhaustive review of the field of neural networks on graphs. I have left out a number of both recent and older papers to make this post more readable and to give it a coherent story line. The papers that I mentioned here will nonetheless serve as a good start if you want to dive deeper into this topic and get a complete overview of what is around and what has been tried so far.

CITATION

If you want to use some of this in your own work, you can cite our paper on Graph Convolutional Networks:

```
@article{kipf2016semi,
  title={Semi-Supervised Classification with Graph Convolutional Networks},
  author={Kipf, Thomas N and Welling, Max},
  journal={arXiv preprint arXiv:1609.02907},
  year={2016}
}
```

SOURCE CODE

We have released the code for Graph Convolutional Networks on GitHub: https://github.com/tkipf/gcn.

You can follow me on Twitter for future updates.

THE ISSUE WITH REGULAR GRAPHS

In the following, I will briefly comment on the statements made in How powerful are Graph Convolutions?, a recent blog post by Ferenc Huszar that provides a slightly negative view on some of the models discussed here. Ferenc considers the special case of regular graphs. He correctly points out that Graph Convolutional Networks (as introduced in this blog post) reduce to rather trivial operations on regular graphs when compared to models that are specifically designed for this domain (like "classical" 2D CNNs for images). It is indeed important to note that current graph neural network models that apply to arbitrarily structured graphs typically share some form of shortcoming when applied to regular graphs (like grids, chains, fully-connected graphs etc.). A localized spectral treatment (like in Defferrard et al., NIPS 2016), for example, reduces to rotationally symmetric filters and can never imitate the operation of a "classical" 2D CNN on a grid (exluding border-effects). In the same way, the Weisfeiler-Lehman algorithm will not converge on regular graphs. What this tells us, is that we should probably look beyond regular grids when trying to evaluate the usefulness of a specific graph neural network model, as there are specific trade-offs that have to be made when designing such models for arbitary graphs (yet it is of course important to make people aware of these trade-offs) - that is, unless we can come up with a universally powerful model at some point, of course.

---

1. A spectral graph convolution is defined as the multiplication of a signal with a filter in the Fourier space of a graph. A graph Fourier transform is defined as the multiplication of a graph signal X (i.e. feature vectors for every node) with the eigenvector matrix U of the graph Laplacian L. The

(normalized) graph Laplacian can be easily computed from the symmetrically normalized graph adjacency matrix $\tilde{A}$: $L = I - \tilde{A}$.↩

2. Using a spectral approach comes at a price: Filters have to be defined in Fourier space and a graph Fourier transform is expensive to compute (it requires multiplication of node features with the eigenvector matrix of the graph Laplacian, which is a $O(N^2)$ operation for a graph with N nodes; computing the eigenvector matrix in the first place is even more expensive).↩

3. I'm simplifying things here. For an extensive mathematical discussion of the Weisfeiler-Lehman algorithm, have a look at the paper by Douglas (2011).↩

4. Node features for the Weisfeiler-Lehman algorithm are typically chosen as scalar integers and are often referred to as colors.↩

5. As we don't anneal the learning rate in this example, things become quite "jiggly" once the model has converged to a good solution.↩

Hi Chris, thanks for your questions! The code to reproduce the Karate club examples is available here: https://github.com/tkipf/gc.... For unsupervised embeddings, Graph Auto-Encoders and their variants are usually the way to go (essentially: GCN as encoder model and some scoring function as decoder) for unsupervised learning on graphs. We have some code for this (incl. relevant references) here: https://github.com/tkipf/gae

1 ^ | v • Reply • Share ›

Konrad Wagstyl • 3 years ago

Hi Thomas,

We really like the idea of gcns as a way to do CNNs on meshes, commonly used in to represent neuroimaging data. We're trying to apply your toolbox to a fully supervised classification with multiple examples. Is there a way to adapt your gcn code to run on multiple examples of labelled data?
To clarify, we have identical NxN connectivity matrices per example, but NxDxS features (S = number of subjects) and NxExS label classes.

2 ^ | v 1 • Reply • Share ›

Thomas Kipf Mod → Konrad Wagstyl • 3 years ago

Hi Konrad, thanks for your comment! The current version of our code doesn't explicitly support such a scenario, but you can make it work by passing your S (identical) connectivity matrices as a single (S*N x S*N) block-diagonal matrix (with NxN blocks) to the model. In the same way you then need to pass the feature and label matrices as (S*N x D) and (S*N x E) "block" matrices, where the feature/label blocks are aligned with the respective connectivity matrix block. A related approach was discussed here: https://github.com/tkipf/gc...

^ | v • Reply • Share ›

Kludge • 8 months ago • edited

Nice article. I was wondering have you tried to approximate eigenvector of graph laplacian using your model? Let's say you computed a few eigenvectors, those are the functions on the nodes. If you use only a few nodes as training example, can GCN roughly approximate the eigenvectors?

I guess for eigenvectors corresponds to small eigenvalues, this is possible, but what about less smooth eigenvectors?

1 ^ | v • Reply • Share ›

Thomas Kipf Mod → Kludge • 8 months ago

Sounds like an interesting experiment - but no, I haven't tried anything like this.

^ | v • Reply • Share ›

F.A.Rezaur Rahman • a year ago

Hi Thomas, this is very cool. I am new to this area and currently getting familiar with it.

I can see from your github link that it can take a lot of memory as it uses the adjacency matrix (NxN dimension). Do you think there can be a way to do it with something like adjacency list/sparse matrix representation to help with the memory issue?

-Reza

1 ^ | v • Reply • Share ›

Thomas Kipf Mod → F.A.Rezaur Rahman • a year ago

A good way to address the memory issue is to use some form of mini-batching, as e.g. in GraphSAGE: https://arxiv.org/abs/1706....

2 ^ | v • Reply • Share ›

F.A.Rezaur Rahman → Thomas Kipf • a year ago

Thanks for the reply and sharing the GraphSAGE work,

Basically, I am currently working with graphs of million nodes. How much memory do you think I will need to run on graph of that scale?

Another question is what kind of input features will you recommend here for each node? May be some example from something you used yourself will be helpful.

Thanks in advance
-Reza

^ | v • Reply • Share ›

Thomas Kipf Mod → F.A.Rezaur Rahman • a year ago

You should be able to run this on CPU on a machine with 64GB RAM or more (without using any mini-batching). If your data doesn't come with any input features, you might benefit from initializing these with DeepWalk (https://arxiv.org/abs/1403.... features.

2 ^ | v • Reply • Share ›

Meenakshi Khosla • 2 years ago

Hi Thomas,
Great work! I was wondering where does the 'weight-sharing' in the context of regular CNNs factor in the case of Graph CNN? How are the filter parameters being shared across the graph?
Thanks.

1 ^ | v • Reply • Share ›

Thomas Kipf Mod → Meenakshi Khosla • 2 years ago

Hi Meenakshi, thanks! In graph CNNs the update function for a node in the graph is typically parameterized the same way across every location in the graph (the model by Duvenaud et al. is an exception), i.e. we share parameters just like in regular CNNs.

^ | v • Reply • Share ›

Meenakshi Khosla → Thomas Kipf • 2 years ago

Hi Thomas! Thanks for your reply--it makes sense. It seems that your work can be extended for graph-level classification on graphs with varying structure using some sort of pooling. I was wondering in what class would you actually put your convolution scheme--spatial or spectral convolutions? I was thinking of ways to extended Deferrard's scheme to handle graphs of varying structure (for graph level classification). But it seems it's not so straightforward. Intuitively, I'm unable to understand why that is considering your method is quite similar to the one proposed for Fast local filtering. Much thanks!

^ | v • Reply • Share ›

Thomas Kipf Mod → Meenakshi Khosla • 2 years ago • edited

Even though the paper has a section that explains a relation to earlier spectral approaches, I would classify our work primarily as a spatial approach, where messages are propagated along graph edges (therefore the name 'propagation model'). Nomenclature in this field is admittedly quite confusing at the moment. Many recent papers confuse spatial with spectral methods, where by spectral I mean methods that involve explicit computation in the spectral domain of the graph (involving eigendecomposition of the graph Laplacian or other related operators on the graph).

Related to your other question: it is not well understood at this moment how different parameterizations of filters (or propagation functions) affect the ability to generalize across graphs. The only thing we know is that linear models in the spectral domain generalize very poorly (often referred to as 'spectral methods don't generalize across graphs'). For neural network-based models the story is a lot more complicated and has to do with overfitting/underfitting, regularization etc.

^ | v • Reply • Share ›

Arun Mallya • 2 years ago

Thanks a lot for this well-written writeup. What would you say is the key difference between the Graph Convolutional Network (GCN) and the Gated Graph Sequence NN (GGSN)?
A few off the top of my head:
1) GCN can have different functions at different layers, while GGSN applies the same layer over time
2) GCN is limited by number of layers, ie one neighborhood propagation per layer, while GGSN can keep going over time, with one neighborhood propagation per time step.

Are there any key differences in representational or modeling power? Overall, both seem to run some function over the graph and produce embeddings per node.
Thanks!

1 ^ | v 1 • Reply • Share ›

Thomas Kipf Mod → Arun Mallya • 2 years ago • edited

Hi Arun, thanks for your question! Both models are indeed quite similar. We highlighted some of the specific differences in our rebuttal for ICLR2017: https://openreview.net/foru...

^ | v • Reply • Share ›

Devansh Shah • 18 days ago

Hi Thomas,
I wanted to know if we can use a CNN instead of the weight matrix W for images. The input nodes are images and there is a graph structure defined on the image. It would be better to use a CNN in this case?

^ | v • Reply • Share ›

Thomas Kipf Mod → Devansh Shah • 18 days ago

Yes, it's possible to couple CNNs and GCNs/GNNs, like in this paper https://arxiv.org/abs/1711...., but you can run into memory issues.

^ | v • Reply • Share ›

Murphy Huang • 19 days ago

Another question is, what if there were some edges that had negative attributes. The normalization of Laplacian matrix would cause some problem in calculation the negative square root of (pseudo-) degree matrix
I believe it is very common in the case of brain connectome, where some regions are positively correlated with some regions and negatively correlated with other regions.
I am looking forward to your reply.

^ | v • Reply • Share ›

Murphy Huang • 19 days ago

Hi, Thomas,

Nice article. I am wondering whether I could use GCN to do the graph classification. In this case, these graphs have different adjacency matrices. From your paper, I think that GCN is based on the idea of decomposition of Laplacian matrix and training a model/filter on that domain. Hence, all graphs should share the same Laplacian matrix, otherwise the filter could not be transferred from one domain to the other.

Reply • Share ›

Cong Rao • 2 months ago

Hi, Thomas.

I was learning to use the model recently. I found that in the code, the adjacent matrix A is constructed over the labeled, unlabeled and test data. So for a (single) new test instance (node) that is not in the graph, can we compute its feature using the model? Namely, how do we compute the feature of unseen data?

Reply • Share ›

Rick Nueve • 3 months ago

Hi, I'm trying to rebuild this network in numpy. I've come upon an issue. How do you perform back-propogation on a GCN? For every node input of a graph, we get a corresponding output with n-dim features. However, we only want to take into account the select nodes which we provide labels for during training. So what happens to the rest of the nodes outputs? Even though we do not have labels the network still outputs a value. So what gets backpropgated for those unlabeled samples? I thought to multiply the error calculation of the outputs without labels by zero, 'a mask', so that they would have no influence during backprop, however this causes underflow errors to occur in a few epochs? If you do not mind, could you explain how to negate the values of the outputs which we do not give labels for during training?

Reply • Share ›

Thomas Kipf Mod ➚ Rick Nueve • 3 months ago

Hi Rick — the solution is indeed to ignore the gradients coming from unlabeled nodes. Masking the loss should work fine, but if you have underflow issues for some reason, you can try to mask the gradients directly.

Reply • Share ›

Rick Nueve ➚ Thomas Kipf • 3 months ago

Thank you for the suggestion. Applying the mask to the gradients directly was able to fix the issue. Thanks!

Reply • Share ›

Chen Tang • 5 months ago

Hi Thomas, thank you for the excellent and informative article! I am curious about how well the trained GCNs could be generalized across different graph structures. For instance, if only a portion of the graph is used for training, will the GCNs have considerably good performance on another portion of the graph?

Reply • Share ›

Thomas Kipf Mod ➚ Chen Tang • 5 months ago

Absolutely - GCNs can be applied both in a transductive setting (which the original paper mostly focused on) and in inductive settings, where training and testing are performed on disjunct sets of nodes. Whether the model will generalize well heavily depends on the data and on the particular model architecture, so it is difficult to give a general answer.

Reply • Share ›

Oren Wright • 5 months ago

Excellent read. I'm trying to reconcile it with part of a recent ICLR paper by Xu, et al. (https://arxiv.org/pdf/1810.00826.pdf)

In that paper they claim that GCNs are ***not*** as powerful as the Weisfeiler-Lehman test, because they use a mean pooling operation which is not injective. Really curious to hear your thoughts. Thanks!

Reply • Share ›

Thomas Kipf Mod ➚ Oren Wright • 5 months ago • edited

Yes, that's an interesting paper. Their claim however does not apply to the GCN formulation from our paper as they mistakenly assumed that we use a mean pooling aggregation. The symmetric normalization of the adjacency matrix ensures that the aggregation is sensitive to both the node degree of the node itself and the neighboring node.

In addition, there is a very simple trick that can fix this issue which is commonly used in practice: one can augment node features with graph statistics such as node degree, and then a relatively simple GNN architecture will do.

Reply • Share ›

Sudipan Saha • 6 months ago

Is there a restriction on the possible feature value? Do they need to be normalized in some way? My features have value between -1 to +1.

Reply • Share ›

Thomas Kipf Mod ➚ Sudipan Saha • 6 months ago

No restrictions, but using standard techniques for feature normalization is recommended.

Reply • Share ›

Wei Liu • 6 months ago

Thank you for this very informative article! However I am struggling to understand how GCN can be used in a supervised setting? You trained a model and at the end you have an embedding for all your (training data) nodes, now you have a unseen new node (e.g. a new karate club member), how do you go about inferencing it's class label?

Reply • Share ›

Thomas Kipf Mod ➚ Wei Liu • 6 months ago

Inductive predictions (i.e. for unseen nodes) are only possible if you have some transferrable feature description of a node (e.g. bag of words vectors in the citation network examples)

(e.g. bag of words vectors in the citation network examples).

▲ | ▼ • Reply • Share ›

Samila Salam • 6 months ago

how to train and test and validate using dgl library gcn implementation

▲ | ▼ • Reply • Share ›

Samila Salam • 6 months ago

how can we validate and test the data with dgl library.in there documentation only training is done

▲ | ▼ • Reply • Share ›

Lucas Tecot • 7 months ago

What does the symmetric normalization do in contrast to the regular normalization? If A is symmetric aren't they identical?

▲ | ▼ • Reply • Share ›

Thomas Kipf Mod ➚ Lucas Tecot • 7 months ago

Row-wise normalization (sometimes also called random walk normalization) does not retain the symmetry of A, hence symmetric normalization is often preferred in spectral approaches. In practice, there are subtle differences in performance when using either of them for semi-supervised learning, so sometimes it is worth trying both.

▲ | ▼ • Reply • Share ›

pooja mehta • 7 months ago

Thank you for such a nice blog which is very useful and applicability of GCN is in many places. I have one query can we use weighted graph in GCN if then what changes needs to be done for doing so??

▲ | ▼ • Reply • Share ›

Kludge • 7 months ago

I was wondering can GCN fit the random label? I tried a few experiments but not able to fit the random labels. Maybe I didn't do the optimization carefully(I just use the default hyperparameters for optimization). I asked this because GCN is relatively small compared to CNN, and I am interested in knowing if it is still overly parameterized.

▲ | ▼ • Reply • Share ›

Thomas Kipf Mod ➚ Kludge • 7 months ago

That's an interesting question, and something I haven't experimented with yet (with default GCNs). I suppose that a more general architecture like in this paper: http://papers.nips.cc/paper... should be able to fit random labels perfectly (for any kind of graph structure).

▲ | ▼ • Reply • Share ›

Kludge ➚ Thomas Kipf • 7 months ago

Hi Thomas, thanks for the quick response. Actually after some experiments, I am able to fit the random labels using GCN.

9 ▲ | ▼ • Reply • Share ›

hjin_cs • 8 months ago

Hi Thomas, I looked the GCN paper in ICLR 2017, and found a question:
In eq (7), you just set the free parameter \theta = \theta_0' = -\theta_1', and then apply the renormalization trick.
While if we set \theta = \theta_0' = \theta_1, then it becomes the exact normalized graph Laplacian. While in such kind of setting, the performance is worse than the normalized adjacency matrix, even after adding the self-loop.

Can you explain why? Or provide some intuitions. It's kind of similar question of explanation of performance in Table 3.

▲ | ▼ • Reply • Share ›

Thomas Kipf Mod ➚ hjin_cs • 7 months ago

Both should perform similarly (with proper hyper parameter tuning) in practice, I guess. The normalized Laplacian and the normalized adjacency matrix share the same eigenbasis, so if you would do a spectral graph convolution (GFT -> filtering -> inverse GFT), then the result should be the same. I guess the difference happens in the way you (re-)normalize the final operator (after adding self-connections etc.), as then the two are no longer the same.

▲ | ▼ • Reply • Share ›

Samila Salam • 8 months ago

HOW WE CAN CONVERT NODE TO VECTOR?IS node2vec IS USEFUL FOR THIS PURPOSE.i have lncrna-lncrna coexpression network.what kind of feature can i add to each node?

▲ | ▼ • Reply • Share ›

Thomas Kipf Mod ➚ Samila Salam • 8 months ago

If you do not have any feature information on nodes then starting with DeepWalk/node2vec embeddings is a reasonable option, yes.

▲ | ▼ • Reply • Share ›

Octavian Ganea • 9 months ago • edited

Hi Thomas. Cool stuff! May I ask for a reference for these claims:

"If we now choose an appropriate non-linearity and initialize the random weight matrix such that it is orthogonal (or e.g. using the initialization from Glorot & Bengio, AISTATS 2010), this update rule becomes stable in practice (also thanks to the normalization with cii) "

and for "And we make the remarkable observation that we get meaningful smooth embeddings where we can interpret distance as (dis-)similarity of local graph structures!"

Thanks!

▲ | ▼ • Reply • Share ›

Thomas Kipf Mod ➚ Octavian Ganea • 9 months ago

Hi Octavian, thanks for your comment! In terms of the capabilities of this class of models to find smooth embeddings of local graph structures, the GraphSAGE paper has some theoretical analysis in this direction: https://arxiv.org/abs/1706.... (Theorem 1).

For the first claim: there are convergence proofs for repeated application of e.g. a lazy random walk matrix (degree-normalized adjacency matrix + self-connections) - see here: https://cs.stanford.edu/~mp... . Once we start introducing learnable weight matrices and non-linearities then this doesn't hold anymore, but with suitable initialization things are usually quite stable - which you can verify by running many of the available implementations of these types of models (e.g. https://github.com/tkipf/gcn) :-) The usual Deep Learning tricks (residual connections, gating, gradient clipping...) to make training stable apply here as well of course.

▲ | ▼ • Reply • Share ›

Load more comments

GET IN TOUCH

THOMAS [DOT] KIPF [AT] GMAIL [DOT] COM
Amsterdam Machine Learning Lab, University of Amsterdam
Science Park 904, 1098 XH Amsterdam, The Netherlands