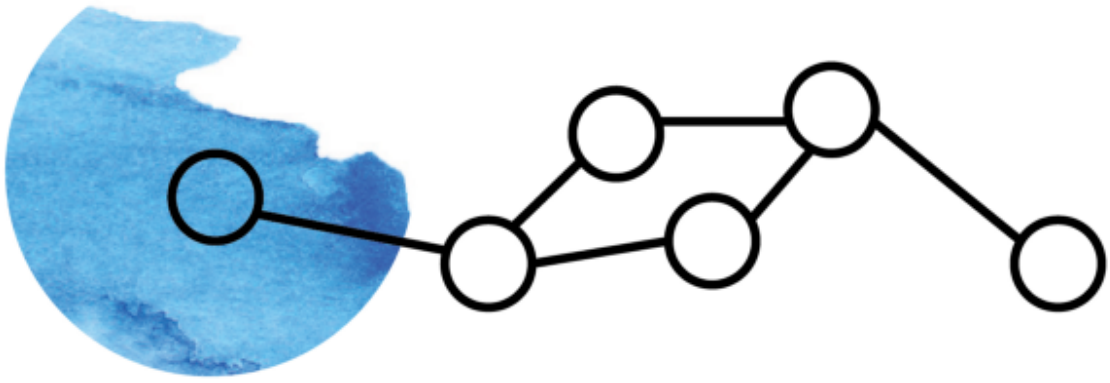
How to get started with machine learning on graphs

David Mack Follow
Dec 6, 2018 · 18 min read

Since our talk at Connected Data London, I've spoken to a lot of research teams who have graph data and want to perform machine learning on it, but are not sure where to start.

In this article, I'll share resources and approaches to get started with machine learning on graphs.



What is graph data?

From talking with research teams, it's really clear how broad and pervasive graph data is — from disease detection, genetics and healthcare to banking and engineering, graphs are emerging as a powerful analysis paradigm for hard problems.

Simply put, a graph is a collection of nodes (e.g. people) and relationships between them (e.g. Fatima is a friend of Jacob). Often those nodes have properties (e.g. Fatima is

age 23).

It's common to store this data in a database. One popular database is Neo4j, in their own words "[the] world's leading graph database, with native graph storage and processing."

Neo4j allows you to query your database with Cypher, the graph equivalent of SQL. In our toy example above, we can get a list of Fatima's friends like so:

```
MATCH (n1)-[:IS_FRIEND_OF]-(n2)
WHERE n1.name = "Fatima"
RETURN n2.name
```

Graphs are a really flexible and powerful way to represent data. Traditional relational databases, with their fixed schemas, make it hard to store connections between different entities, yet such connections are an abundant and vital part of the real world. In a graph database, these connections are easy to store and query. Furthermore, often the relationships between many things (e.g. the connections between family members) collectively provide vital information, and graph databases make this easy to analyze.

> *The term "relationship" and "edge" are used interchangeably in this document. Neo4j uses the former, much of graph theory uses the latter.*

Why use machine learning on graph data ('graph ML')?

First of all, why use machine learning? A great article on this question is "Ways to think about machine Learning" by Benedict Evans, covering the ways that companies are starting to think about and actually use ML.

Distilling Ben's argument to focus on graph ML, there are two major ways that it is useful:

ML can automate functions that are easy for a human to do, but hard to describe to a computer

Real world data is noisy and has many complex sub structures. Tasks such as "outline the people in this image" are easy for a human, but very hard to turn into a discrete algorithm.

Deep learning allows us to transform large pools of example data into effective functions to automate that specific task.

This is doubly true with graphs — they can differ in exponentially more ways than an image or vector thanks to the open-ended relationship structure. Using graph ML we can create functions to spot recurring patterns

ML can transform information on a scale humans cannot

The double-edged sword of computers is that they will do exactly what we tell them — no more, no less (the occasional bug excepted!).

This means that they will perform our exact instructions, with no deviation of improvisation. And they will keep performing them, no matter how long we let them run.

Therefore, computers can process data on a scale no human could (due to the time or attention required). This makes new analysis possible, such as analyzing billions of transaction webs to fingerprint fraud.

What is graph ML?

Our definition is simply "applying machine learning to graph data". This is intentionally broad and inclusive. In this article I'll tend to focus on neural network and deep learning approaches as they're our own focus, however where possible I'll include links to other approaches.

In this article I'm not going to cover "traditional" graph analysis — that's the well known algorithmic techniques like PageRank, clique identification, shortest path, etc. These are very powerful and should be considered a first port-of-call due to their well understood nature and plentiful implementation in public libraries.

What are the challenges of using graph ML?

Whilst an exciting field full of promise, machine learning on graphs is still a nascent technology.

In mainstream areas of ML the community has discovered widely applicable techniques (e.g. transfer learning using ResNet for images or BERT for text) and made them accessible to developers (e.g. TensorFlow, PyTorch, FastAI). However, there are no equivalently easy, universal techniques, nor do any of the popular ML libraries have support for graph data.

Similarly, graph databases such as Neo4j do not offer ways to run machine learning algorithms on their data (although Neo4j are thinking a lot about how to make that possible!)

One of the reasons for the lack of graph support in deep learning libraries is that the flexibility of the data-structure (e.g. any node can have any number of relationships to other nodes) doesn't fit the fixed computation-graph, fixed sized tensor paradigm popular with deep learning libraries and GPU manufacturers.

Put more simply, it's hard to represent and manipulate a sparse graph as a matrix. Not impossible, but definitely harder than working with vectors, text and images.

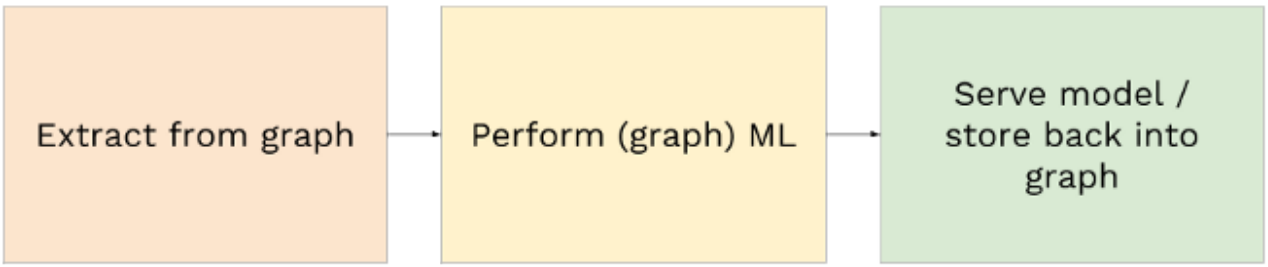
However despite this, there is a strong surge of interest in graph ML. My personal prediction is that this area will become both mainstream and a foundational tool to how we analyze data in many industries.

Note that, just like any other ML technique, most graph ML requires a large volume of training data.

What does a graph ML system look like in practice?

Whilst the answer to this question can vary a lot with the task and dataset, it's helpful to outline what you, the keen adventurer, should expect to encounter.

It's almost certain that you'll be coding up this system yourself — high-level tools for graph ML do not yet exist. It's likely you'll be building the system using Python and an ML library like TensorFlow or PyTorch. Depending on your scale, you may be training your model on a single machine, or using a distributed cluster (interestingly, many graph learning approaches naturally distribute quite well).



You will probably start by extracting your data from a graph — likely stored in CSV files, a graph database like Neo4j, or another format.

Then you'll ingest that data into a machine learning library. In my current work (involving millions of small graphs) I precompile each graph into a TFRecord, with feature vectors storing the nodes, relationships and the adjacency matrix. All node properties and text is tokenized using a common dictionary.

This works for small graphs, but for larger graphs you'll need some sort of scheme to partition the graph into smaller training examples (e.g. you could train on patches, or on individual node-edge-node triples).

Note that some approaches tabularize the data before it reaches the machine learning library. Node2Vec is a good example of this, where random walks are used to transform each node into a vector. These vectors are then fed into the machine learning model as a list.

Once the data is ingested, the actual modeling and learning begins. There is a wide variety of possible approaches here.

Finally, the model needs to be used or served somehow. In some cases a new node/edge/graph property is computed by the model and this can be added to the

original data-store.

In other cases a model is produced for online prediction. In this setup, one needs to build a system to feed the model with any graph data it requires to perform its predictions (possibly once again ingesting it from a graph database) and then finally that prediction can be served up to a user or follow-on system (e.g. an answer spoken by Alexa to a user).

Let's do some machine learning on our graphs!

Alright, so let's look at some of the approaches you can take to perform machine learning on graphs.

I'll outline methods here, point out some of their pros and cons, and link to fuller descriptions. For the sake of time and space, I'll have to sacrifice some detail here.

Despite being a young field, researchers have come up with a dazzling array of approaches and variations. Whilst I've tried to cover the major areas in this article, there is sadly no way to make this list fully exhaustive.

If there is an approach you'd like to see added to this article, please let us know, so we can add it.

What sort of task do you want to do?

There is a wide variety of starting points and overall approaches for graph ML. Therefore, it's helpful to narrow those down by thinking about what is the general task you're trying to achieve.

As with any learning system, your success and development effort will be hugely helped by narrowing down what you want to achieve. By coming up with a minimal, well-specified goal, your model and dataset can be reduced down to something more tractable.

On the last point, graph databases are particularly sirenous, encouraging us towards grand "omnipotent" goals: as the database can represent almost anything, it's tempting to try to build overly general intelligence. You are warned :)

Types of task we'll cover:

1. Predicting if a relationship exists between two nodes
2. Scoring and classification of nodes, edges and whole graphs

This article is intended as a launch-pad for your own research. Just as with any data science, methods will need to be adapted to your individual circumstances. As a lot of graph ML is still in early research, you should expect to try many approaches before finding one that works.

Basic approaches

Before jumping into building a graph ML system (potentially requiring a high investment in infrastructure), it's important to consider if simpler methods could suffice.

There are a few ways to simplify the problem:

- Can you tabularize your data (e.g. could you just look at the node list? could you work on node-edge-node triples?) and use traditional ML approaches (e.g. linear regression, feedforward networks)
- Can you filter the dataset to be smaller (e.g. remove certain nodes)?
- Can you cluster the graph into sub-graphs and treat those as a table?
- Can you use traditional graph metrics (e.g. PageRank) possibly augmented by traditional ML (e.g. apply a linear regression to calculated node properties to classify a node)

I'll refer back to some of these approaches in the following sections where particularly applicable.

General graph ML approaches

Some graph ML approaches can be used for multiple tasks. I've included their full description here. In the later sections I'll make reference to this section and highlight some task-specific details.

Once again, it is not possible to do justice to these large fields of work, the best we can do here is to give you pointers to explore further.

> *Feel free to skip this section and proceed to the task you're interested in solving, then refer back here for detail.*

Node embeddings

Node embeddings were one of the early developments in graph ML, and have remained popular due to their simplicity, robustness and computational efficiency.

A node embedding simply means calculating a vector for each node in the graph. The vector is calculated to have useful properties, for example the dot product of any two nodes' embeddings could tell you if they're from the same community.

In this way, the embedding simplifies the graph data down to something much more manageable: a vector.

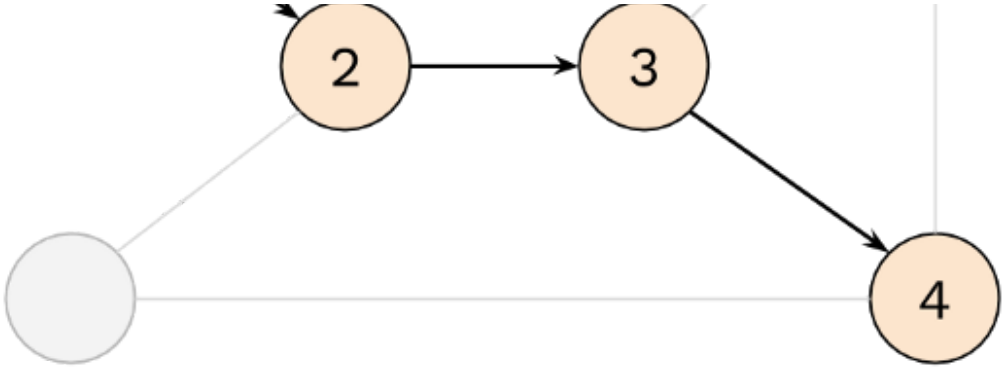
Node embeddings are often calculated by incorporating lots of graph structure together (more on this in a moment).

Their tradeoff is that necessarily information is discarded. A fixed-length vector can rarely represent all of the graph structure around a node. It may or may not incorporate node and relationship properties.

However, with a bit of creativity a node embedding can be used in combination with other graph ML approaches. In this setting, the embedding becomes a node property and can be used as a booster for other techniques, which perhaps don't penetrate as far into the graph structure as the embedding generation did.

Here I'll highlight some of the main embedding approaches. For a comprehensive survey of graph embedding techniques and their comparison, checkout these two recent papers.

Random walks



Random walks are a surprisingly powerful and simple graph analysis technique, backed up by a long lineage of mathematical theory.

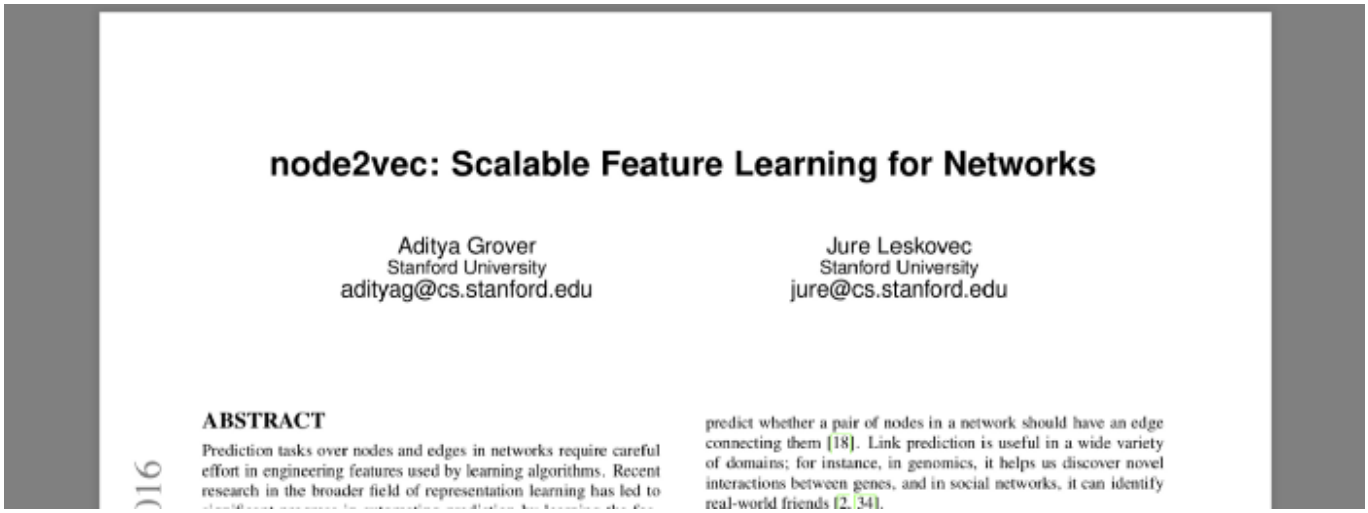
A random walk is where one starts at a node in the graph, randomly chooses an edge, then traverses it. And then repeats the process, until a sufficiently long path is produced.

The genius of a random walk is it turns a many dimensional irregular thing (the graph) into a simple matrix (the list of fixed length paths, each path composed of its nodes).

In large enough volume, it's theoretically possible to reconstruct the basic graph structure from random walks. And random walks play to machine learning's great strength: learning from large volumes of data.

There are many ways to harness random walks to calculate node embeddings. In what follows I'll highlight some of the main approaches.

Node2Vec



node2vec: Scalable Feature Learning for Networks

Aditya Grover
Stanford University
adityag@cs.stanford.edu

Jure Leskovec
Stanford University
jure@cs.stanford.edu

ABSTRACT

Prediction tasks over nodes and edges in networks require careful effort in engineering features used by learning algorithms. Recent research in the broader field of representation learning has led to

predict whether a pair of nodes in a network should have an edge connecting them [18]. Link prediction is useful in a wide variety of domains; for instance, in genomics, it helps us discover novel interactions between genes, and in social networks, it can identify real-world friends [2, 34].

Node2Vec is a popular and fairly generalized embedding technique using random walks.

The way to turn these random walks into an embedding is with a clever optimization objective. First assign each node a random embedding (e.g. gaussian vector of length N). Then for each pair of source-neighbor nodes in each walk, we want to maximize the dot-product of their embeddings by adjusting those embeddings. Finally, we simultaneously minimize the dot-product of random node pairings. The effect of this is that we learn a set of embeddings that tend to have high dot products for nodes in the same walks, e.g. in the same community/structures.

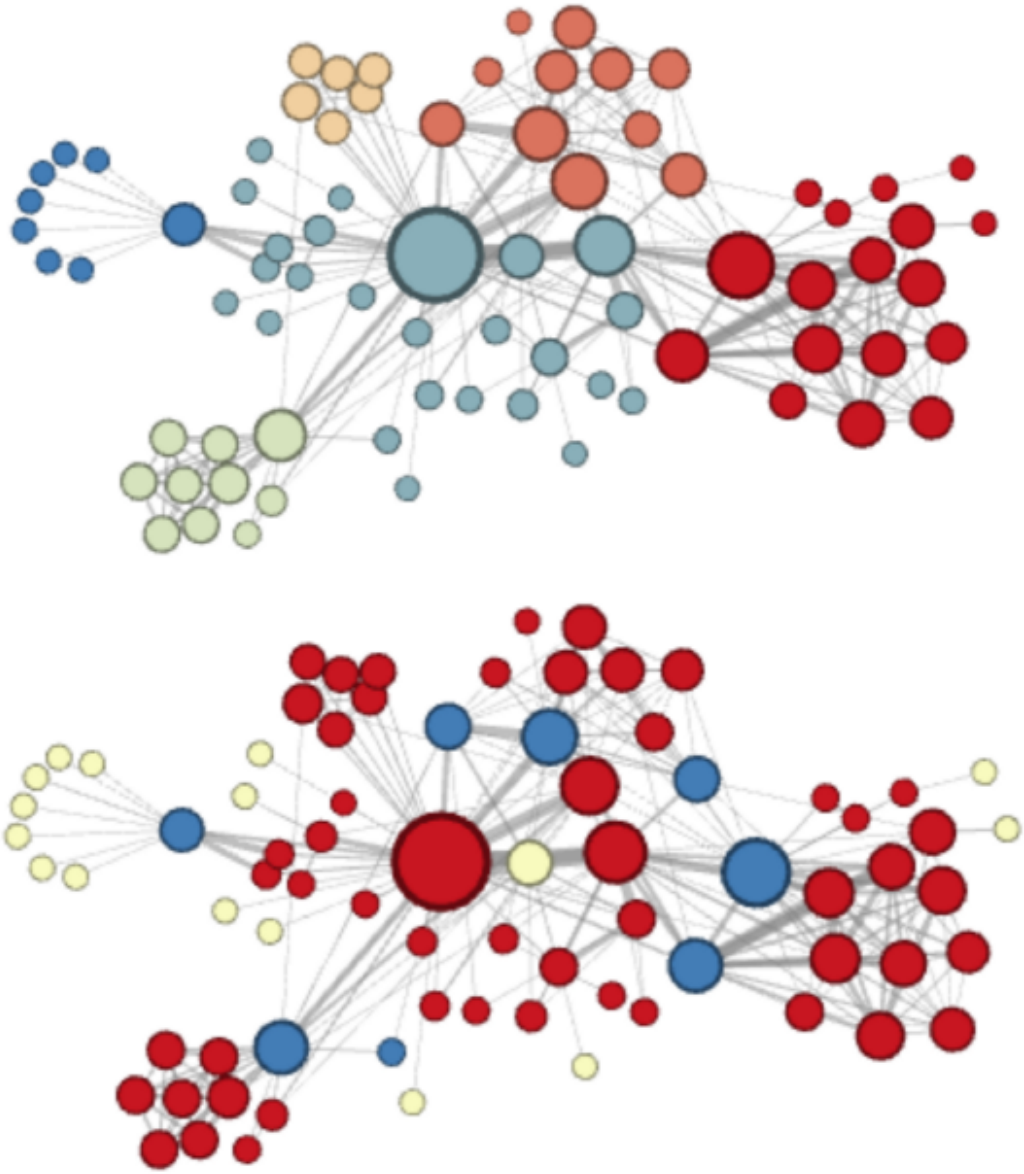


Figure 3: Complementary visualizations of Les Misérables coappearance network generated by *node2vec* with label colors reflecting homophily (top) and structural equivalence (bottom)

ing homophily (top) and structural equivalence (bottom).

From the original Node2Vec paper demonstrating “In-Out” hyper-parameter

The final bit of Node2Vec is that it has parameters to shape the random walks. Using the “In-out” hyper-parameter you can prioritise whether the walks focus on small local areas (e.g. are these nodes in the same small community?) or whether the walks wander broadly across the graph (e.g. are these nodes in the same type of structure?).

Node2Vec extensions

Node2Vec’s strength is its simplicity, but that is also its downfall. The standard algorithm does not incorporate node properties or edge properties, amongst other desirable pieces of information.

However, it’s quite straightforward to extend Node2Vec to incorporate more information. Simply alter the loss function. For example:

- Instead of dot product between node embeddings, try a different / learned function
- Instead of only using node embeddings, incorporate their properties as well

For inspiration, check out the large number of papers citing Node2Vec.

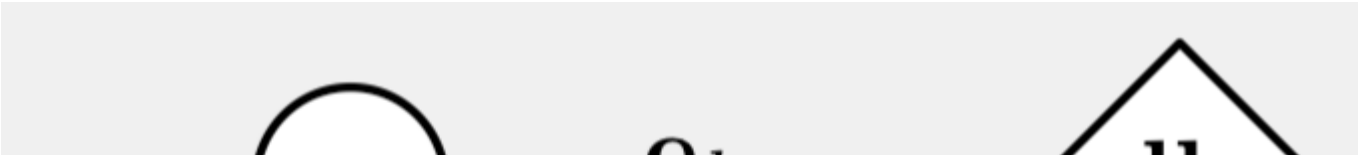
Collaborative filtering using random walks

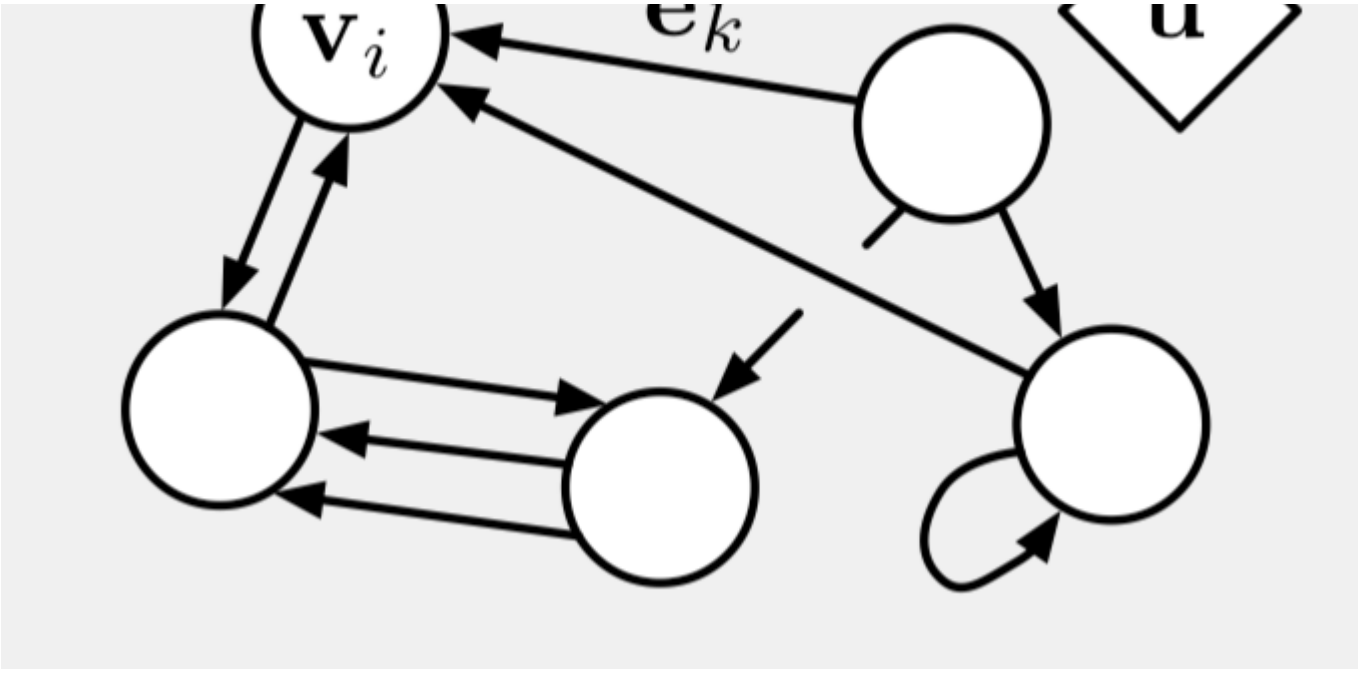
A very simple example use of random walks is to solve the collaborative filtering problem e.g. given users’ reviews of products, which other products will a user like?.

This broadly follows the same scheme as node2vec, although has been simplified even further. You can see the entire implementation and an explanation in our article.

Graph Networks (aka Graph Convolutional Networks)

Graph Networks are a rich and important area of graph ML. The basic premise is to embed a neural network into the graph structure itself:





From relational inductive biases paper

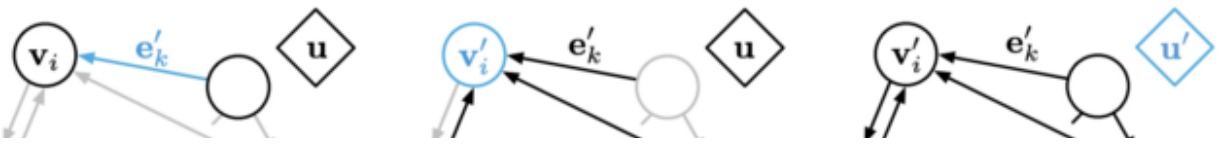
Typically this involves storing states for each node and using an adjacency matrix to propagate those states to the nodes' neighbors.

A good overview paper is "Relational inductive biases, deep learning, and graph networks" from DeepMind which both surveys the history of this sub-field and also proposes a unifying approach for comparing different Graph Networks and neural networks in general.

In the above paper, graph networks are thought of as a collection of functions, for propagating state and for aggregating state across nodes, edges and entire graphs. In this way, many different architectures from the literature are comparable. Here's an extract showing these functions in action:

A GN block contains three "update" functions, ϕ, and three "aggregation" functions, ρ,

$$\begin{aligned} \mathbf{e}'_k &= \phi^e\left(\mathbf{e}_k, \mathbf{v}_{r_k}, \mathbf{v}_{s_k}, \mathbf{u}\right) & \bar{\mathbf{e}}'_i &= \rho^{e\to v}\left(E'_i\right) \\ \mathbf{v}'_i &= \phi^v\left(\bar{\mathbf{e}}'_i, \mathbf{v}_i, \mathbf{u}\right) & \bar{\mathbf{e}}' &= \rho^{e\to u}\left(E'\right) \\ \mathbf{u}' &= \phi^u\left(\bar{\mathbf{e}}', \bar{\mathbf{v}}', \mathbf{u}\right) & \bar{\mathbf{v}}' &= \rho^{v\to u}\left(V'\right) \end{aligned} \tag{1}$$



(a) Edge update (b) Node update (c) Global update

Figure 3: Updates in a GN block. Blue indicates the element that is being updated, and black indicates other elements which are involved in the update (note that the pre-update value of the blue element is also used in the update). See Equation 1 for details on the notation.

A graph network has many possible outputs:

- Node states
- Edge states
- Entire graph states

These can then be used like embeddings for tasks like classification, scoring and link-prediction.

Graph Networks are very general and powerful — they've been used to analyze many things from natural language, 3d scenes to biology. Our recent work has been showing that they can implement many common traditional graph algorithms.

Some starting points to understand more about graph networks and their capabilities:

- "How powerful are Graph Convolutional Networks" by Thomas Kipf shows how an untrained network can perform impressive analysis thanks to structure alone
- GraphSAGE is a framework for inductive representation learning on large graphs
- Deep Learning for Network Biology gives a great overview of graph ML techniques applied to biological problems

Our extensions to graph networks

Under our MacGraph research project, we've been experimenting with a number of extensions to graph networks. Whilst we're still refining the approaches, they've shown enough promise that they're worth including here for other researchers.

We're attempting to learn different reasoning algorithms, extracting and transforming data from the graph. To do this we've added to the aforementioned network a series of components that make it akin to a graph-based Turing machine:

- GRU cells at each node to enable each node to better retain state over multiple iterations
- Attention-based read and write of node state from a global "control cell"
- A control cell (essentially a LSTM cell) that takes the current task as input and outputs a sequence of control signals to the rest of the network guiding their actions — this is based on the MACnet architecture.
- Node and edge tables that are read from using attention guided by the "control cell"
- A working memory that is read and written to by the aforementioned components

Predicting if a relationship exists between two nodes ('link prediction')

This is a common task and quite well studied. The basic formulation is:

What is the probability p(A,R,B) that node *A* has relationship *R* to node *B*?

Examples are knowledge graph completion (e.g. if Michelangelo is a painter born in Tuscany, is he Italian?) and predicting protein interaction. This can be used both for trying to predict new, unknown things (e.g. what drugs might be effective?) as well as improving existing imperfect data (e.g. which project does this task belong to?).

> *More information on many of the approaches can be found in the earlier section "General graph ML approaches"*

Node embeddings and random walks

Node embeddings (often generated using random walks) are frequently used for link prediction.

Embeddings are often generated such that nearby nodes in the graph have similar embedding tensors. Therefore a comparison between (e.g. dot product or euclidean distance) provides a likelihood of linkage. Some methods like Node2Vec actually directly train the embedding on the presence/absence of links.

Graph networks can be used to generate node embeddings for link prediction. In this case, incorporate link prediction capability into the network's loss function.

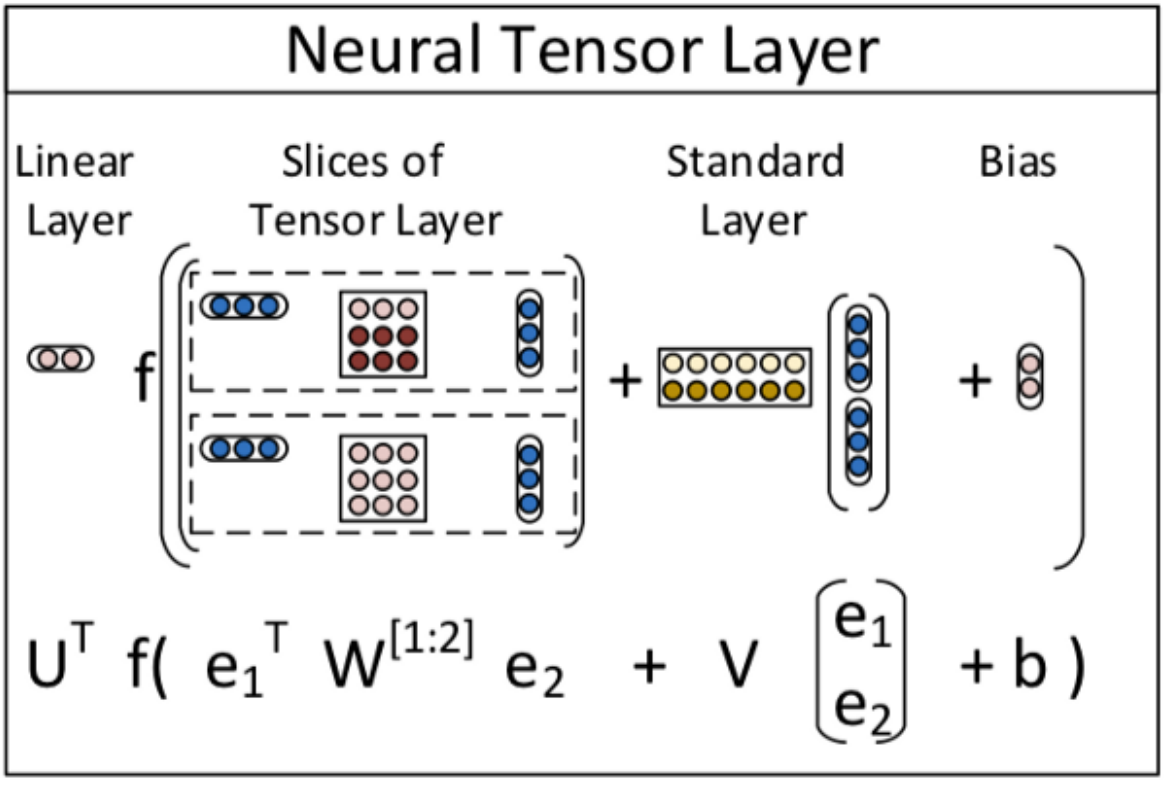
Deep learning with graph features

This means to tabularize the graph data, then run a traditional feed-forward network on it.

For example, each node could be represented by its properties (concatenated into a tensor). Each training example has two nodes and relationship type as features and existence of edge as label. Remember to balance the label classes.

This simple approach can work well when a lot of the graph structure is reflected in the properties (e.g. a graph of streets and each node has its GPS location)

Neural Tensor Networks



This interesting approach from Stanford essentially memorizes the graph into tensors and matrices. “Our model outperforms previous models and can classify unseen relationships in WordNet and FreeBase with an accuracy of 86.2% and 90.0%, respectively.”

Reinforcement learning

Reinforcement learning can also be used for link prediction. In this approach, the network learns to extract a series of facts from the graph which it can combine to produce a link prediction.

One example of this approach is “Multi-hop knowledge graph reasoning with reward shaping” in which the network learns to walk the graph and use that information to produce a link prediction.

Scoring and classification of nodes, edges and whole graphs

Another common task is to try to classify or score part of the graph. For example, trying to find how relevant proteins are to a certain gene. Or trying to cluster students into their schools based on their friendship groups.

Classification means to output a probability distribution across potential labels, *scoring* means to output a scalar that might be used for comparison with others. Both are conceptually similar, classification involves more dimensions.

Formally, the task is to define one of the following functions, where **Output** is the set of possible output class distributions or the set of possible output scores:

f(n:Node) → r ∈ Output

g(e:Edge) → r ∈ Output

h(g:Graph) → r ∈ Output

Most approaches to performing this have two steps:

1. Perform some computation on the graph, possibly combining multiple elements of its nodes and edges into state stored in nodes, edges and/or globally across the graph

2. Extract, aggregate and transform the state into the desired output

Step 1. can be performed using many different approaches, which I'll list below.

Step 2. is often performed using a feed-forward neural network (FFN). The extraction and aggregation are either done using hand-crafted functions (e.g. read out specific nodes, sum together specific edges) or learned functions (e.g. attention for extraction, convolution for aggregation).

The choice of both steps is a matter of data science and experimentation, there has yet to emerge any clear "one size fits all" solution.

> *More information on many of the approaches can be found in the earlier section "General graph ML approaches"*

Node embeddings and random walks

Node embeddings provide a rich source of node-state for classification and scoring.

When using embeddings, often the node(s) under inspection will have their embeddings passed through a small FFN to produce the desired output. Depending on the use case, the node's properties can also be included in the FFN's input.

If the node embeddings are created using random-walks (e.g. using Node2Vec) they will incorporate local structural information (for example, which community the node is in, or what super-structure it is part of) that may be relevant to the classification or scoring being performed (e.g. clustering different graph sub-communities).

Graph Networks

Graph Networks are a versatile, generalized approach to embedding a neural network into the graph itself.

A graph network computes node, edge and graph states (although some of these can be omitted depending on the application).

These states can then be transformed to produce the final output. For instance, the graph state could be passed through a FFN to create an overall graph categorization.

There are many different examples in the literature of graph networks, see the introductory section for a brief survey of them.

Attention sequences

An interesting approach is outlined in "Graph classification using Structural Attention". In this work, the graph is repeatedly read from using attention to extract nodes.

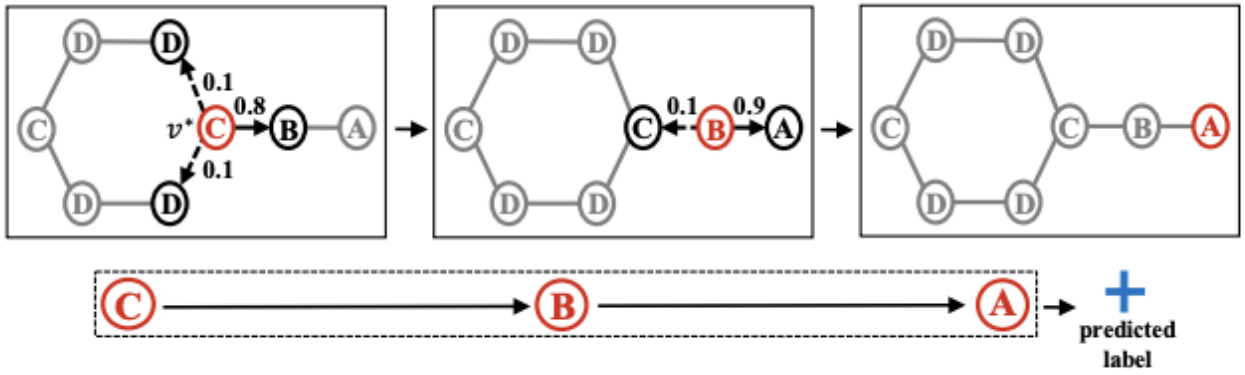


Figure 1: Attention-based graph classification. Given a starting node v^* and a budget $T = 3$ nodes to select for graph classification, attention is used to guide the walk towards more informative parts of the graph.

The center of the network is an LSTM cell which controls which nodes are read from, incorporates the attention read data into its internal state, then outputs a prediction of the graph's classification.

This is similar to the approach (although not on graph data) used in Compositional Attention Networks for Machine Reasoning, where a central RNN cell guides attentional reading and composition of the read data.

Traditional deep learning on node properties or on patches

Simplifying the problem down to a tabular dataset opens up many better-researched approaches (e.g. feed-forward and convolutional neural networks).

One way to do this is to treat each node with its properties as a training example. This may involve hand-crafting additional properties that you believe will assist in the classification/scoring.

Another way to tabularise the graph is to extract fixed sized patches. In this model, a node, its edges, and potentially its neighbors are extracted into a fixed sized table. The fixed size means that a maximum number of edges and nodes can be stored, and if more exist in the table, they must be randomly sampled. Furthermore, if there are fewer nodes and edges than the fixed table can store, it will need padded out with a designated null value. Finally, one must choose how to select patches — a simple model is to extract one per node or edge.

Tabularization discards potentially valuable network information, but simplifies engineering and model research.

Search-engine techniques using node properties

Finally, a non machine learning approach is worth consideration. If nodes are being scored on some sort of linkage, content or textual basis, a search-engine/document retrieval inspired approach may work.

This ranges from simple fuzzy text matching, through to PageRank, phrase and concept matching.

This is a very mature field of computer science. For more pointers, check out Wikipedia's Information retrieval article.

Further reading

Throughout this article I've linked to many resources, they are all listed here for convenience. Additionally, I've added further items of interest not listed earlier.

Surveys of the field

- A good survey of the different structural approaches to graph machine learning (I'd recommend starting with this one): Graph Neural Networks: A Review of Methods and Applications

- A Comprehensive Survey of Graph Embedding: Problems, Techniques and Applications
- Graph Embedding Techniques, Applications, and Performance: A Survey
- A Survey on Network Embedding
- Attention Models in Graphs: A Survey
- Deep Learning for Network Biology
- Representation Learning on Graphs: Methods and Applications
- Network Representation Learning: A Survey
- Graph Summarization Methods and Applications: A Survey
- Must-read papers on knowledge representation learning (KRL) / knowledge embedding (KE)

Random walks / embeddings

- Node2Vec
- Review prediction with Neo4j and TensorFlow
- Knowledge graph embedding: a survey of approaches and applications
- A novel embedding model for knowledge base completion based on CNN
- GEMSEC: graph embedding with self clustering

Graph Networks

- Relational inductive biases
- Graph convolutional networks
- GraphSAGE
- Smart Reply: Automated Response Suggestion for Email
- 3D Graph Neural Networks for RGBD Semantic Segmentation

Knowledge graphs

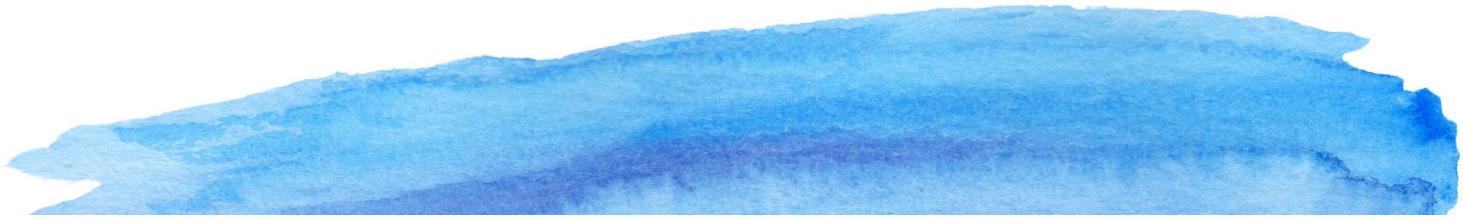
- DeepPath: A Reinforcement Learning Method for Knowledge Graph Reasoning
- Multi-hop knowledge graph reasoning with reward shaping
- Neural Tensor Networks
- MacGraph — Iterative reasoning on knowledge graphs
- KBGAN: Adversarial Learning for Knowledge Graph Embeddings

Miscellaneous

- Compositional Attention Networks for Machine Reasoning
- Graph classification using Structural Attention
- GAMEnet: graph augmented memory networks for recommending medication combination
- Modeling Relational Data with Graph Convolutional Networks
- Answering questions using knowledge graphs and sequence translation

Octavian's research

Octavian's mission is to develop systems with human-level reasoning capabilities. We believe that graph data and deep learning are key ingredients to making this possible. If you interested in learning more about our research or contributing, get in touch.



Machine Learning Graphs Graphml Neo4j AI