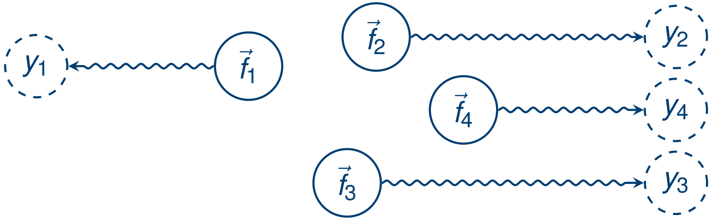
Overview of neural network architectures for graph-structured data analysis

Petar Veličković

Artificial Intelligence Group
Department of Computer Science and Technology, University of Cambridge, UK

Motivation: supervised learning

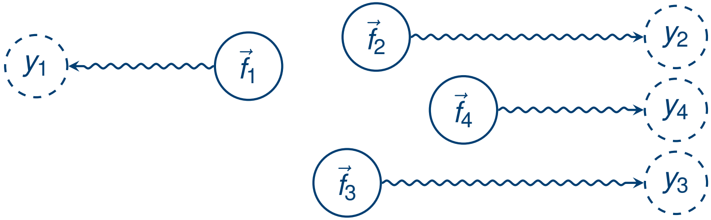
- Petar Veličković here!
- This is a (supervised) machine learning problem.



- Four examples, features ($\vec{f}_i$) and labels (y_i).
- Good enough for science. ✓

Motivation: supervised learning

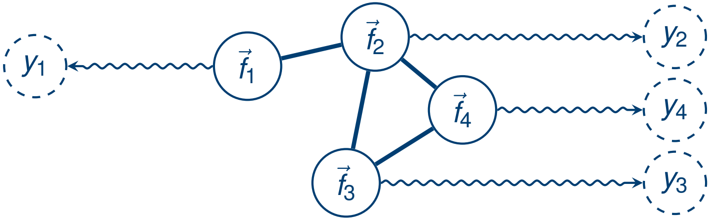
- Petar Veličković here!
- This is a (supervised) machine learning problem.



- Four examples, features ($\vec{f}_i$) and labels (y_i).
- Good enough for science. **Not Aperture Science!** X

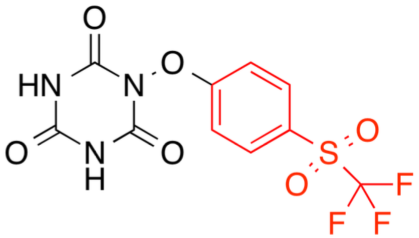
Motivation: supervised learning

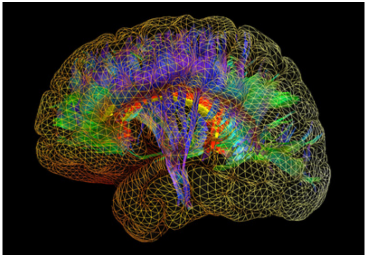
- Petar Veličković here!
- This is a (supervised) machine learning problem.



- Four examples, features ($\vec{f}_i$) and labels (y_i).
- Good enough for science. **Not Aperture Science!** $\boxed{\mathbf{X}}$
- Gentlemen, I give you **graphs**. *The inputs of tomorrow!*

Graphs are **everywhere**!





Introduction

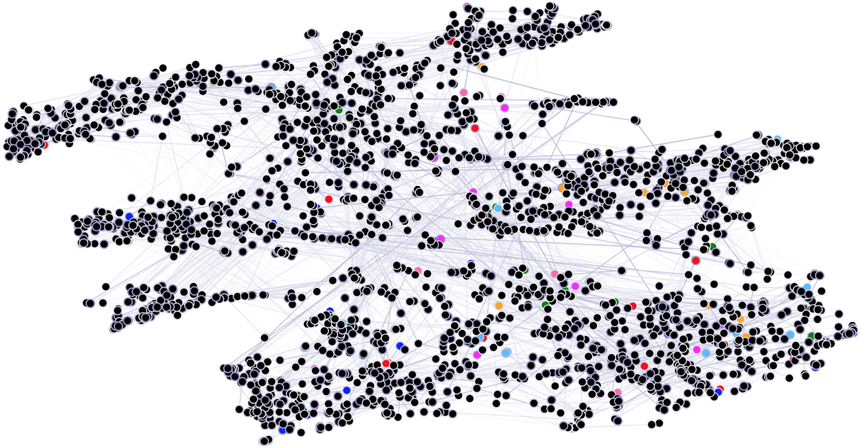
- In this talk, I will demonstrate some of the popular methodologies that leverage **neural networks** for processing **graph-structured inputs**.
- Although the earliest approaches to this problem date to the late 90s, it has caught traction only in the recent five years (with a proper explosion happening throughout 2017)!
  - For early references, you may investigate the works of Sperduti & Starita (1997) and Frasconi *et al.* (1998), IEEE TNNLS.
- There's at least ten submissions *to ICLR 2018 alone* that attempt solving the same graph problems in different ways.

Mathematical formulation

- We will focus on the **node classification** problem:
  - **Input**: a matrix of *node features*, $\mathbf{F} \in \mathbb{R}^{N \times F}$, with F features in each of the N nodes, and an *adjacency matrix*, $\mathbf{A} \in \mathbb{R}^{N \times N}$.
  - **Output**: a matrix of *node class probabilities*, $\mathbf{Y} \in \mathbb{R}^{N \times C}$, such that $Y_{ij} = \mathbb{P}(Node\ i \in Class\ j)$.
- We also assume, for simplicity, that the edges are **unweighted** and **undirected**:
  - That is, $A_{ij} = A_{ji} = \begin{cases} 1 & i \leftrightarrow j \\ 0 & otherwise \end{cases}$

  but many algorithms we will cover are capable of generalising to weighted and directed edges.
- There are **two** main kinds of learning tasks in this space...

Transductive learning



Training algorithm sees *all features* (**including test nodes**)!

Inductive learning

- Now, the algorithm *does not have access to all nodes upfront*!
- This often implies that either:
  - Test nodes are (incrementally) inserted into training graphs;
  - Test graphs are **disjoint** and *completely unseen*!
- A much harder learning problem (requires generalising across *arbitrary graph structures*), and many transductive methods will be inappropriate for inductive problems!

Simplest approach: a per-node classifier

- Completely **drop** the graph structure, and classify each node individually, with a shared deep neural network classifier. :)
- In fact, this is how *most of deep learning is done*, even if there might be relationships between training examples!
- A single layer of the network computes $\mathbf{F}' = \sigma(\mathbf{FW})$, where $\mathbf{W} \in \mathbb{R}^{F \times F'}$ is a shared and learnable *weight matrix*, and σ is an *activation function* (e.g. logistic/tanh/ReLU)—ignoring biases.
- The final layer will use the *softmax* function and optimise the *cross-entropy* loss in each training node (usual classification).
- Simple, but very cheap (and should always be a baseline)!

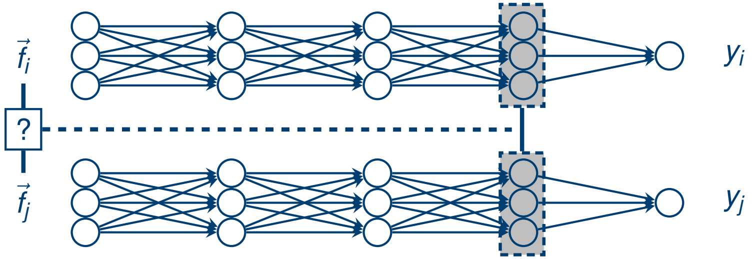
Augmenting the per-node classifier

- Many earlier approaches to incorporating graph structure will retain the per-node shared classifier, but incorporate graph structure by either:
  - *constraining its learnt features* depending on the graph edges;
  - *augmenting the input layer* with structural node features.

- I will now briefly cover both of those approaches.

Injecting structure: *semi-supervised embedding*

- Introduced by Weston *et al.* (ICML 2008), generalising the work of Zhu *et al.* (ICML 2003) and Belkin *et al.* (JMLR 2006) to neural networks.



- Under the assumption that the edges encode *node similarity*, further constrain the learnt representations of nodes to be close/distant depending on presence of edge!

Semi-supervised embedding loss

- Essentially, the loss function to optimise is augmented with a (dis)similarity constraint, $\mathcal{L}_{sim}$:

$$\mathcal{L} = \mathcal{L}_0 + \lambda \mathcal{L}_{sim}$$

  where $\mathcal{L}_0$ is the usual supervised learning loss (e.g. cross-entropy), and λ is a hyperparameter.
- One way to define $\mathcal{L}_{sim}$:

$$\mathcal{L}_{sim} = \sum_i \left(\sum_{j \in \mathcal{N}_i} \|\vec{h}_i - \vec{h}_j\|^2 + \sum_{j \notin \mathcal{N}_i} \max\left(0, m - \|\vec{h}_i - \vec{h}_j\|^2\right) \right)$$

  where $\mathcal{N}_i$ is the *neighbourhood* of node i, $\vec{h}_i$ is (one of) its hidden layer's outputs, and m is a hyperparameter.

Inserting structure: *DeepWalk*

- An alternative to augmenting the loss function is first learning some **structural features**, $\vec{\Phi}_i$, for each node i (these will not depend on $\vec{f}_i$, but on the graph structure)!
- Then, use $\vec{f}_i \| \vec{\Phi}_i$ as the input to the shared classifier (where $\|$ is concatenation).
- Typically, **random walks** are used as the primary input for analysing the structural information of each node.
- The first method to leverage random walks efficiently is *DeepWalk* by Perozzi *et al.* (KDD 2014)

Overview of DeepWalk

- Start by random features $\vec{\Phi}_i$ for each node i.
- Sample a random walk $\mathcal{W}_i$, starting from node i.
- For node x at step j, $x = \mathcal{W}_i[j]$, and a node y at step $k \in [j - w, j + w]$, $y = \mathcal{W}_i[k]$, modify $\vec{\Phi}_x$ to maximise $\log \mathbb{P}(y|\vec{\Phi}_x)$ (obtained from a neural network classifier).
- Inspired by **skip-gram models** in natural language processing: to obtain a good vector representation of a word, its vector should allow us to easily predict the words that *surround* it.

Overview of DeepWalk, *cont'd*

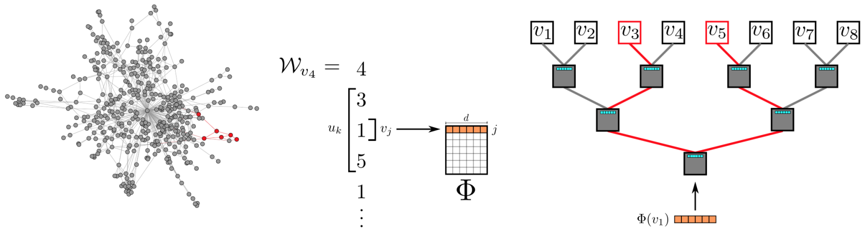
- Expressing the full $\mathbb{P}(y|\vec{\Phi}_x)$ distribution directly, even for a single layer neural network, where

$$\mathbb{P}(y|\vec{\Phi}_x) = \mathit{softmax}(\vec{w}_y^T \vec{\Phi}_x) = \frac{\exp\left(\vec{w}_y^T \vec{\Phi}_x\right)}{\sum_z \exp\left(\vec{w}_z^T \vec{\Phi}_x\right)}$$

  is prohibitive for large graphs, as we need to normalise across the entire space of nodes—making most updates *vanish*.

- To rectify, DeepWalk expresses it as a *hierarchical softmax*—a tree of binary classifiers, each halving the node space.

DeepWalk in action



$\mathcal{W}_{v_4} = 4, 3, 1, 5, 1, \vdots$
u_k
v_j
d
j
Φ
v_1 v_2 v_3 v_4 v_5 v_6 v_7 v_8
$\Phi(v_1)$

(a) Random walk generation. (b) Representation mapping. (c) Hierarchical Softmax.

Figure 3: Overview of DeepWalk. We slide a window of length $2w + 1$ over the random walk $\mathcal{W}_{v_4}$, mapping the central vertex v_1 to its representation $\Phi(v_1)$. Hierarchical Softmax factors out $\Pr(v_3 \mid \Phi(v_1))$ and $\Pr(v_5 \mid \Phi(v_1))$ over sequences of probability distributions corresponding to the paths starting at the root and ending at v_3 and v_5. The representation Φ is updated to maximize the probability of v_1 co-occurring with its context $\{v_3, v_5\}$.

Later improved by *LINE* (Tang *et al.*, WWW 2015) and *node2vec* (Grover & Leskovec, KDD 2016), but main idea stays the same.

Incorporating labels and features: *Planetoid*

- Methods such as DeepWalk are still favourable when dealing with *fully unsupervised* graph problems, as they don't depend on having any labels or features in the nodes!
- However, if we have labels/features, **why not use them**?
- The essence behind **Planetoid** (*Predicting Labels And Neighbours with Embeddings Transductively Or Inductively from Data*), by Yang *et al.* (ICML 2016).

Planetoid's sampling strategy: *Negative sampling*

- Addresses the issue with $\mathbb{P}(y|\vec{\Phi}_x)$ by employing **negative sampling**; predict instead $\mathbb{P}(\gamma|\vec{\Phi}_x, \vec{w}_y)$, where $\gamma \in \{0, 1\}$.

- Essentially, use a binary classifier:

$$\mathbb{P}(\gamma|\vec{\Phi}_x, \vec{w}_y) = \sigma\left(\vec{w}_y^T \vec{\Phi}_x\right)$$

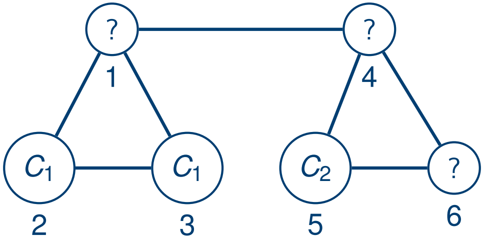
  where σ is the logistic sigmoid function. Now each update will focus only on *one* node's weight vector rather than all of them!

- $\gamma = 1$ implies that nodes x and y are a "positive" pair (more detail in the next slide).

Planetoid's sampling strategy: *Sampling pairs*

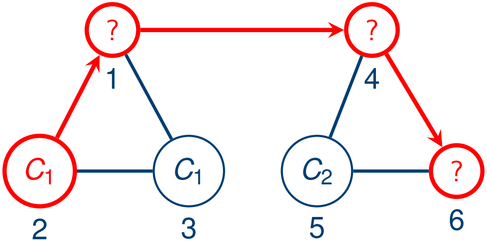
- Planetoid retains DeepWalk's idea of predicting proximal nodes in random walks.
  - Sample two nodes *a* and *b* that are close enough in a random walk, optimise the classifier to predict $\gamma = 1$.
  - Sample two nodes *a* and *b* uniformly at random, optimise the classifier to predict $\gamma = 0$.
- It also injects **label information**:
  - Sample two nodes *a* and *b* with same labels ($y_a = y_b$), optimise the classifier to predict $\gamma = 1$.
  - Sample two nodes *a* and *b* with different labels ($y_a \neq y_b$), optimise the classifier to predict $\gamma = 0$.

Planetoid in action



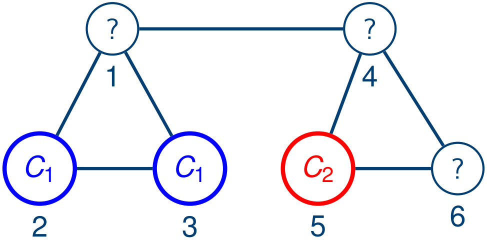
Consider this example graph, with three labelled nodes.
I will now illustrate the two phases of Planetoid.

Planetoid in action: Random walk-based sampling



Sample from a random walk—can take e.g. nodes 1 and 4 with $\gamma = 1$, and nodes 1 and 5 with $\gamma = 0$.

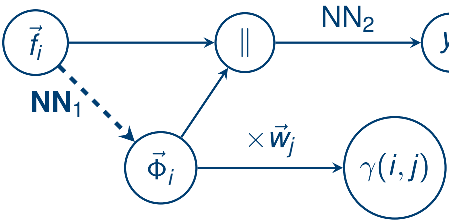
Planetoid in action: Label-based sampling



Sample given the labels—can take e.g. nodes 2 and 3 with $\gamma = 1$, and nodes 3 and 5 with $\gamma = 0$.

Planetoid's inductive dataflow

- In an inductive setting, the structural features $\vec{\Phi}_i$ can no longer be independently learned—need to adapt to **unseen** nodes!
- The inductive version of Planetoid forces $\vec{\Phi}_i$ to directly depend on $\vec{f}_i$—you guessed it—by employing a neural network. :)



$$\begin{tikzcd}
\vec{f}_i \arrow[r] \arrow[dr, dashed, "\mathbf{NN}_1"'] & \| \arrow[r, "\mathrm{NN}_2"] & y_i \\
 & \vec{\Phi}_i \arrow[u] \arrow[r, "\times \vec{w}_j"] & \gamma(i,j)
\end{tikzcd}$$

Explicit graph neural network methodologies

- All methods covered so far have used a shared classifier that classifies each node independently, with graph structure injected only *indirectly*.
- We will from now restrict our attention solely to methods that *directly* leverage the graph structure when computing intermediate features.
- **Main idea**: Compute node representations $\vec{h}_i$ based on the initial features $\vec{f}_i$ and the graph structure, and then use $\vec{h}_i$ to classify each node independently (as before).

Graph Neural Networks

- The first prominent example of such an architecture are **Graph Neural Networks** (GNNs) presented first in Gori *et al.* (IJCNN 2005) and then in Scarselli *et al.* (TNNLS 2009).
- Start with randomly initialised $\vec{h}_i^{(0)}$, then at each timestep propagate as follows (slightly different than original paper, assuming only undirected edges of one type):

$$\vec{h}_i^{(t)} = \sum_{j \in \mathcal{N}_i} f\left(\vec{h}_j^{(t-1)}\right)$$

where f is a *propagation model*, expressed as a usual neural network linear layer:

$$f(\vec{h}_i) = \mathbf{W}\vec{h}_i + \vec{b}$$

where $\mathbf{W}$ and $\vec{b}$ are learnable weights and biases, respectively.

Graph Neural Networks, *cont'd*

- As backpropagating through time is expensive, the authors of GNNs further constrain f to be a **contractive map.** This implies that the $\vec{h}_i$ vectors will always converge to a *unique fixed point*!
- Iterate until convergence (for T steps), then classify using $\vec{h}_i^{(T)}$. Train using the Almeida-Pineda extension of backpropagation (Almeida, 1990; Pineda, 1987).
- Arguably, too restrictive. Also, impossible to inject problem-specific information into $\vec{h}_i^{(0)}$ (as will always converge to same value regardless of initialisation).

Gated Graph Neural Networks

- An extension to GNNs, known as **Gated** Graph Neural Networks (GGNNs) by Li *et al.* (ICLR 2016), brought the bleeding-edge deep learning practices to GNNs.
- Propagate for a **fixed number of steps**, and do not restrict the propagation model to be contractive.
  - This enables conventional backpropagation.
  - It also allows us to meaningfully initialise the model!
- Leverage a more sophisticated propagation model (employing techniques such as *gating*) to surpass GNN performance.

GGNN propagation rule

- Initialise as $\vec{h}_i^{(0)} = \vec{f}_i \| \vec{0}$ (append zeroes for extra capacity).
- Then propagate as follows (slightly different than original paper, assuming only undirected edges of one type):

$$\vec{a}_i^{(t)} = b_i + \sum_{j \in \mathcal{N}_i} \vec{h}^{(t-1)}$$
$$\vec{h}_i^{(t)} = \tanh\left(\mathbf{W}\vec{a}_i^{(t)}\right)$$

- Now, extend this to incorporate *gating mechanisms*, to prevent full overwrite of $\vec{h}_i^{(t-1)}$ by $\vec{h}_i^{(t)}$.
  - Basically, learn (from $\vec{a}_i^{(t)}$ and $\vec{h}_i^{(t-1)}$) how much to overwrite.

Full GGNN propagation rule

- The full propagation model is as follows:

$$\vec{a}_i^{(t)} = b_i + \sum_{j \in \mathcal{N}_i} \vec{h}_j^{(t-1)}$$

$$\vec{r}_i^{(t)} = \sigma\left(\mathbf{W}^r \vec{a}_i^{(t)} + \mathbf{U}^r \vec{h}_i^{(t-1)}\right)$$

$$\vec{z}_i^{(t)} = \sigma\left(\mathbf{W}^z \vec{a}_i^{(t)} + \mathbf{U}^z \vec{h}_i^{(t-1)}\right)$$

$$\vec{\tilde{h}}_i^{(t)} = \tanh\left(\mathbf{W}\vec{a}_i^{(t)} + \mathbf{U}\left(\vec{r}_i^{(t)} \odot \vec{h}_i^{(t-1)}\right)\right)$$

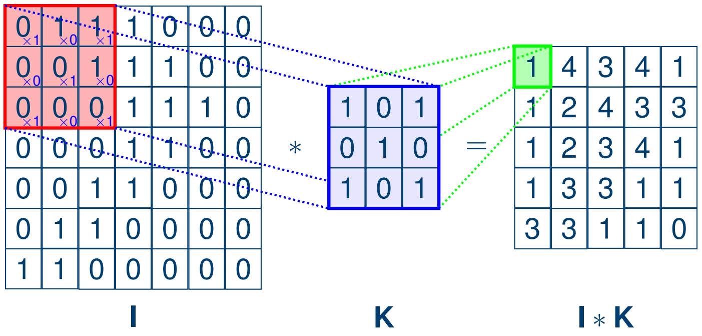
$$\vec{h}_i^{(t)} = (1 - \vec{z}_i^{(t)}) \odot \vec{h}_i^{(t-1)} + \vec{z}_i^{(t)} \odot \vec{\tilde{h}}_i^{(t)}$$

where $\odot$ is elementwise vector multiplication, $\vec{r}_i$ and $\vec{z}_i$ are *reset* and *update* gates, and σ is the logistic sigmoid function.

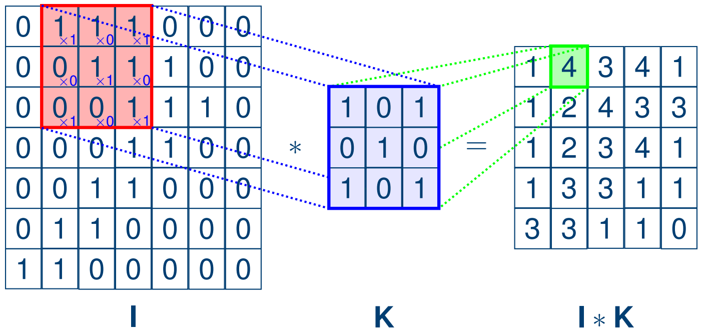
The silver bullet—a *convolutional* layer

- GGNNs feature a “time-step” operation which should be very familiar to those of you who have already worked with *recurrent neural networks* (such as LSTMs).
- These are designed for data that changes *sequentially*; however, our graphs have **static** features!
- It would be more appropriate if we could somehow generalise the *convolutional operator* (as used in CNNs) to operate on arbitrary graphs!
- An excellent “common framework” for many of the approaches to be listed now has been presented in *“Neural Message Passing for Quantum Chemistry”*, by Gilmer *et al.* (ICML 2017).

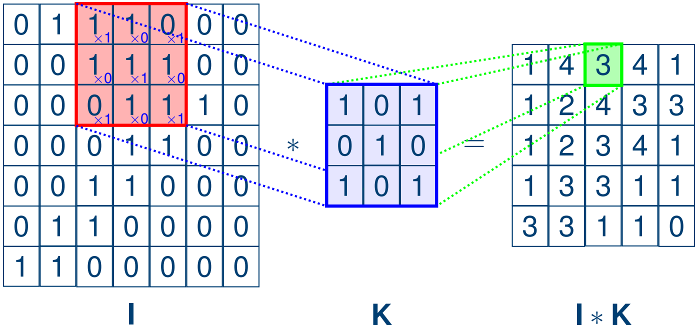
Convolution on images



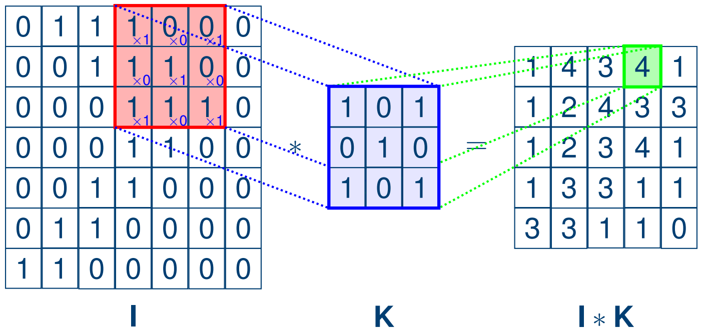
Convolution on images



Convolution on images



Convolution on images



UNIVERSITY OF CAMBRIDGE

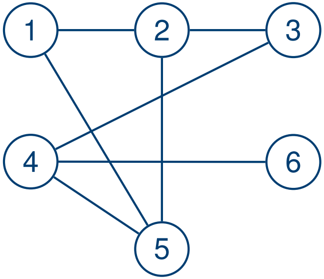
Challenges with graph convolutions

- Desirable properties for a graph convolutional layer:
  - **Computational and storage efficiency** ($\sim O(V + E)$);
  - **Fixed** number of parameters (independent of input size);
  - **Localisation** (acts on a *local neighbourhood* of a node);
  - Specifying **different importances** to different neighbours;
  - Applicability to **inductive problems**.
- Fortunately, images have a highly rigid and regular connectivity pattern (each pixel “connected” to its eight neighbouring pixels), making such an operator trivial to deploy (as a small kernel matrix which is slided across).
- Arbitrary graphs are a **much harder** challenge!

Spectral graph convolution

- A large class of popular approaches attempts to define a convolutional operation by operating on the graph in the **spectral domain**, leveraging the *convolution theorem*.
- These approaches utilise the **graph Laplacian matrix**, $\mathbf{L}$, defined as $\mathbf{L} = \mathbf{D} - \mathbf{A}$, where $\mathbf{D}$ is the degree matrix (diagonal matrix with $D_{ii} = deg(i)$) and $\mathbf{A}$ is the adjacency matrix.
- Alternately, we may use the **normalised graph Laplacian**, $\tilde{\mathbf{L}} = \mathbf{I} - \mathbf{D}^{-1/2}\mathbf{A}\mathbf{D}^{-1/2}$.

Graph Laplacian example



$$\mathbf{L} = \begin{bmatrix} 2 & -1 & 0 & 0 & -1 & 0 \\ -1 & 3 & -1 & 0 & -1 & 0 \\ 0 & -1 & 2 & -1 & 0 & 0 \\ 0 & 0 & -1 & 3 & -1 & -1 \\ -1 & -1 & 0 & -1 & 3 & 0 \\ 0 & 0 & 0 & -1 & 0 & 1 \end{bmatrix}$$

Graph Fourier Transform

- The Laplacian is symmetric and positive semi-definite; we can therefore diagonalise it as $\mathbf{L} = \mathbf{U}\boldsymbol{\Lambda}\mathbf{U}^T$, where $\boldsymbol{\Lambda}$ is a diagonal matrix of its eigenvalues.
- This means that multiplying the feature matrix by $\mathbf{U}^T$ allows us to enter the *spectral domain* for the graph! Therein, convolution just amounts to pointwise multiplication.
- This “Graph Fourier Transform” is the essence of the work of Bruna *et al.* (ICLR 2014).

Graph Fourier Transform, *cont'd*

- To convolve two signals using the convolution theorem:

$$conv(\vec{x}, \vec{y}) = \mathbf{U}\left(\mathbf{U}^T\vec{x} \odot \mathbf{U}^T\vec{y}\right)$$

- Therefore, a *learnable convolutional layer* amounts to:

$$\vec{h}'_i = \mathbf{U}\left(\vec{w} \odot \mathbf{U}^T\mathbf{W}\vec{h}_i\right)$$

  where $\vec{w}$ is a learnable vector of weights, and $\mathbf{W} \in \mathbb{R}^{F' \times F}$ is a shared, learnable, feature transformation.
- Downsides:
  - Computing $\mathbf{U}$ is $O(V^3)$—*infeasible* for large graphs!
  - One independent weight per node—not fixed!
  - Not localised!

Chebyshev networks

- These issues have been overcome by *ChebyNets*, the work of Defferrard *et al.* (NIPS 2016).
- Rather than computing the Fourier transform, use the related family of *Chebyshev polynomials* of order k, T_k:

$$\vec{h}'_i = \sum_{k=0}^{K} w_k T_k(\mathbf{L})\mathbf{W}\vec{h}_i$$

- These polynomials have a recursive definition, highly simplifying the computation:

$$T_0(x) = 1 \qquad T_1(x) = x \qquad T_k(x) = 2xT_{k-1}(x) - T_{k-2}(x)$$

Properties of Chebyshev networks

- Owing to its recursive definition, we can compute the output iteratively as $\sum_{k=0}^{K} w_k \vec{t}_k$, where:

$$\vec{t}_0 = \mathbf{W}\vec{h}_i \qquad \vec{t}_1 = \mathbf{LW}\vec{h}_i \qquad \vec{t}_k = 2\mathbf{L}\vec{t}_{k-1} - \vec{t}_{k-2}$$

  where each step constitutes a **sparse** multiplication with **L**.

- The number of parameters is **fixed** (equal to K weights).

- Note that $T_k(\mathbf{L})$ will be a (weighted) sum of all powers of **L** up to $\mathbf{L}^k$. This means that $T_k(\mathbf{L})_{ij} = 0$ if $dist(i, j) > k$!

$\implies$ The operator is **K-localised**!

Properties of Chebyshev networks, *cont'd*

- To avoid issues with exploding or vanishing signals, typically a scaled version of **L** is fed into the algorithm:

$$\tilde{\mathbf{L}} = \frac{2\mathbf{L}}{\lambda_{max}} - \mathbf{I}$$

  where λ_{max} is the largest eigenvalue of **L**.

- This constrains all eigenvalues to lie in the range $[-1, 1]$, therefore making the norm of all results controllable.
- Major limitation: *unable to specify* ***different weights*** *to* ***different nodes*** *in a neighbourhood*! All k-hop neighbours will receive weight $w_k + w_{k+1} + \cdots + w_K$.

Limited filters

Going back to the image scenario, under the assumption that each pixel of an image is connected to its immediate four neighbours, this would constrain our 3×3 convolutional kernel to be of the form:

$$\begin{bmatrix} w_2 & w_1 + w_2 & w_2 \\ w_1 + w_2 & w_0 + w_1 + w_2 & w_1 + w_2 \\ w_2 & w_1 + w_2 & w_2 \end{bmatrix}$$

severely limiting the variety of patterns that can be usefully extracted from the image.

GCNs

- Arguably the most popular approach in recent months has been the **Graph Convolutional Network** (GCN) of Kipf & Welling (ICLR 2017).
- The authors further simplify the Chebyshev framework, setting $K = 1$ and assuming $\lambda_{max} \approx 2$, allowing them to redefine a single convolutional layer as simply:

  $$\vec{h}'_i = \tilde{\mathbf{D}}^{-1/2}\tilde{\mathbf{A}}\tilde{\mathbf{D}}^{-1/2}\mathbf{W}\vec{h}_i$$

  which improves computational performance on larger graphs and predictive performance on small training sets.
- However, the previous issue is *still there...*

Applicability to inductive problems

- Another *fundamental* constraint of all spectral-based methods is that the learnt filter weights are assuming a particular, fixed, graph Laplacian.
- This makes them theoretically inadequate for arbitrary **inductive** problems!
- We have to move on to non-spectral approaches…

Molecular fingerprinting networks

- An early notable approach towards such methods is the work of Duvenaud *et al.* (NIPS 2015).
- Here, the method adapts to processing with various degrees by learning a *separate* weight matrix $\mathbf{H}_d$ for each node degree d.
- The authors dealt with an extremely specific domain problem (*molecular fingerprinting*), where node degrees could never exceed five; this **does not scale** to graphs with *very wide degree distributions*.

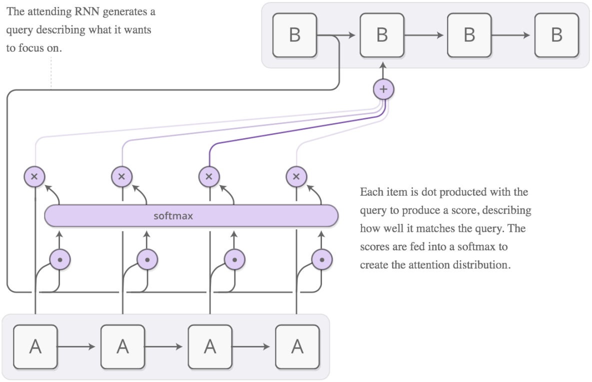
GraphSAGE

- Conversely, the recently-published **GraphSAGE** model by Hamilton *et al.* (NIPS 2017) aims to **restrict every degree to be the same** (by sampling a *fixed-size* set of neighbours of every node, during both training and inference).
- Inherently **drops relevant data**—limiting the set of neighbours visible to the algorithm.
- Impressive performance was achieved across a variety of inductive graph problems. However, the best results were often achieved with an LSTM-based aggregator, which is unlikely to be optimal.

Attentional mechanisms

- One of the latest non-spectral techniques leverages an *attentional mechanism* (originally published by Bahdanau *et al.* (ICLR 2015)), which is now a *de facto* standard for sequential processing tasks.
- Computes *linear combinations* of the input features to generate the output. The coefficients of these linear combinations are parametrised by a **shared neural network**!
- Intuitively, allows each component of the output to generate its own combination of the inputs—thus, different outputs *pay different levels of attention* to the respective inputs.

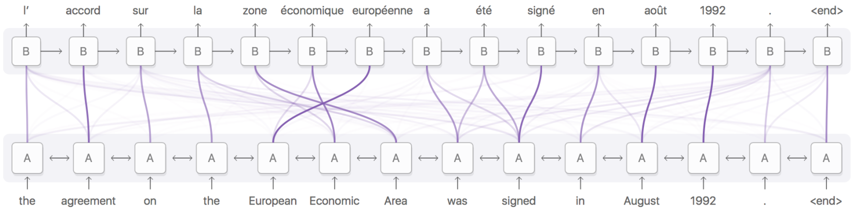
Attention in action: *a potential mechanism*



The attending RNN generates a query describing what it wants to focus on.

Each item is dot producted with the query to produce a score, describing how well it matches the query. The scores are fed into a softmax to create the attention distribution.

Attention in action: *machine translation*



Self-attention

- A rather exciting development in this direction concerns **self-attention**; a scenario where the input *attends over itself*:

$$\alpha_{ij} = a(\vec{h}_i, \vec{h}_j)$$
$$\vec{h}'_i = \sum_j \mathit{softmax}_j(\alpha_{ij})\vec{h}_j$$

  where $a(\vec{x}, \vec{y})$ is a neural network (the *attention mechanism*).
- Critically, this is **parallelisable** across all input positions!
- Vaswani et al. (NIPS 2017) have successfully demonstrated that this operation is self-sufficient for achieving state-of-the-art on machine translation.

Graph Attention Networks

- My recent ICLR 2018 publication—in collaboration with the Montréal Institute for Learning Algorithms (MILA)—proposing **Graph Attention Networks** (GATs), leverages exactly the self-attention operator!
- In its naïve form, the operator would compute attention coefficients *over all pairs of nodes*.
- To inject the graph structure into the model, we *restrict* the model to only attend over a node's neighbourhood when computing its coefficient!

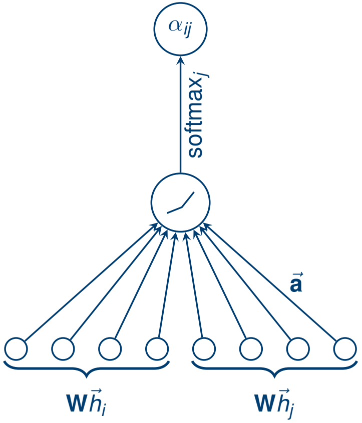
GAT equations

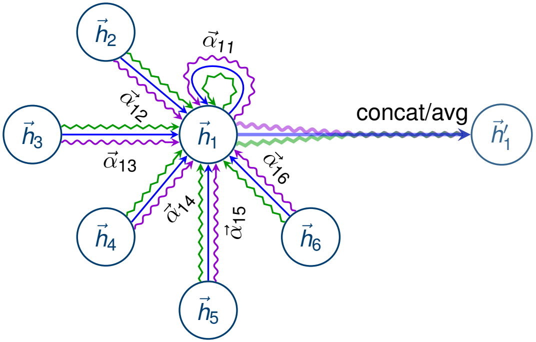
- To recap, a single attention head of a GAT model performs the following computation:

$$e_{ij} = a(\mathbf{W}\vec{h}_i, \mathbf{W}\vec{h}_j)$$
$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k \in \mathcal{N}_i} \exp(e_{ik})}$$
$$\vec{h}'_i = \sigma\left(\sum_{j \in \mathcal{N}_i} \alpha_{ij}\mathbf{W}\vec{h}_j\right)$$

- Some further optimisations (like *multi-head attention* and *dropout* on the α_{ij} values) help further *stabilise* and *regularise* the model.

A single GAT step, visualised





GAT analysis

- **Computationally efficient**: attention computation can be parallelised across all edges of the graph, and aggregation across all nodes!
- **Storage efficient**—a sparse version does not require storing more than $O(V + E)$ entries anywhere;
- **Fixed** number of parameters (dependent only on the desirable feature count, not on the node count);
- Trivially **localised** (as we aggregate only over neighbourhoods);
- Allows for (implicitly) specifying **different importances** to **different neighbours.**
- Readily applicable to **inductive problems** (as it is a shared *edge-wise* mechanism)!

GAT performance

- **It seems that we have finally satisfied all of the major requirements for our convolution!**

- How well does it perform?

Datasets under study

Table: Summary of the datasets used in our experiments.

	Transductive			*Inductive*
	Cora	**Citeseer**	**Pubmed**	**PPI**
# Nodes	2708	3327	19717	56944 (24 graphs)
# Edges	5429	4732	44338	818716
# Features/Node	1433	3703	500	50
# Classes	7	6	3	121 (multilabel)
# Training Nodes	140	120	60	44906 (20 graphs)
# Validation Nodes	500	500	500	6514 (2 graphs)
# Test Nodes	1000	1000	1000	5524 (2 graphs)

Results on Cora/Citeseer/Pubmed

Transductive

Method	**Cora**	**Citeseer**	**Pubmed**
MLP	55.1%	46.5%	71.4%
ManiReg	59.5%	60.1%	70.7%
SemiEmb	59.0%	59.6%	71.7%
LP	68.0%	45.3%	63.0%
DeepWalk	67.2%	43.2%	65.3%
ICA	75.1%	69.1%	73.9%
Planetoid	75.7%	64.7%	77.2%
Chebyshev	81.2%	69.8%	74.4%
GCN	81.5%	70.3%	**79.0%**
MoNet	$81.7 \pm 0.5\%$	—	$78.8 \pm 0.3\%$
GCN-64*	$81.4 \pm 0.5\%$	$70.9 \pm 0.5\%$	$\mathbf{79.0} \pm 0.3\%$
GAT (ours)	$\mathbf{83.0} \pm 0.7\%$	$\mathbf{72.5} \pm 0.7\%$	$\mathbf{79.0} \pm 0.3\%$

Results on PPI

Inductive

Method	**PPI**
Random	0.396
MLP	0.422
GraphSAGE-GCN	0.500
GraphSAGE-mean	0.598
GraphSAGE-LSTM	0.612
GraphSAGE-pool	0.600
GraphSAGE*	0.768
Const-GAT (ours)	0.934 ± 0.006
GAT (ours)	**0.973** $\pm$ 0.002

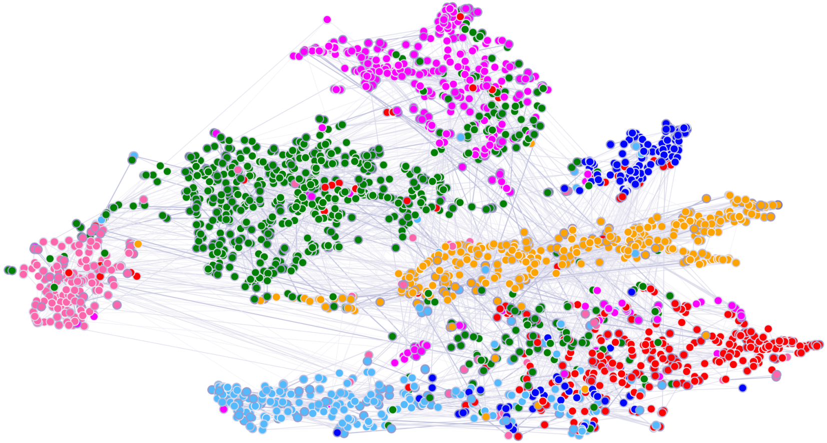
Here, *Const-GAT* is a GCN-like inductive model.

Applications

- I will conclude with an overview of a few interesting applications of GCN- and GAT-like models.

- This list is by no means exhaustive, and represents only what I have been able to find thus far. :)

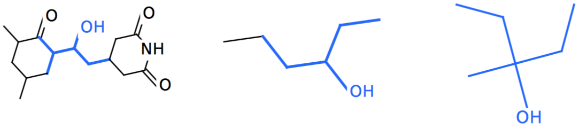
Citation networks



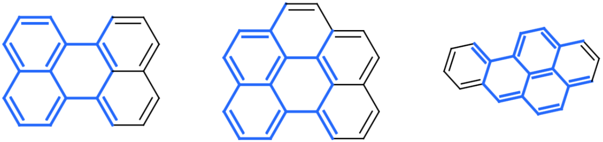
Veličković *et al.* (ICLR 2018)

Molecular fingerprinting

Fragments most activated by pro-solubility feature

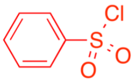
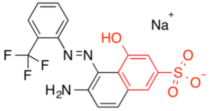
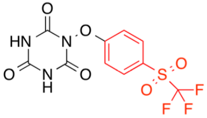
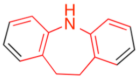
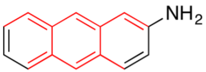
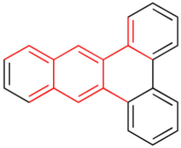


Fragments most activated by anti-solubility feature



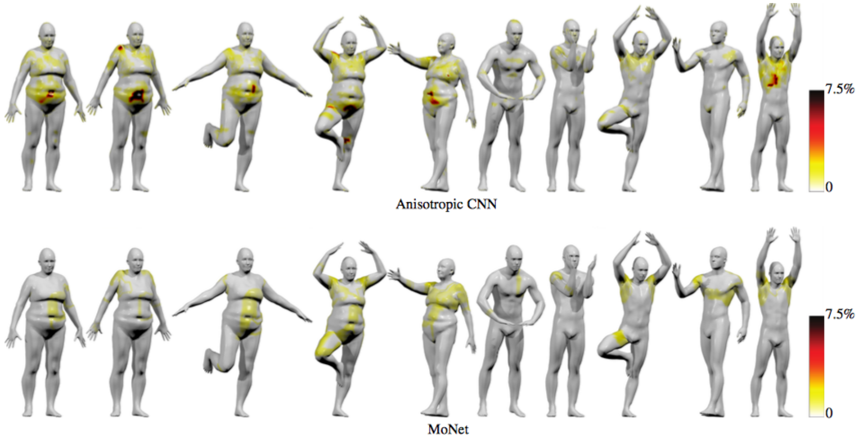
Duvenaud *et al.* (NIPS 2015)

Molecular fingerprinting, *cont'd*

Fragments most activated by toxicity feature on SR-MMP dataset			
Fragments most activated by toxicity feature on NR-AHR dataset			

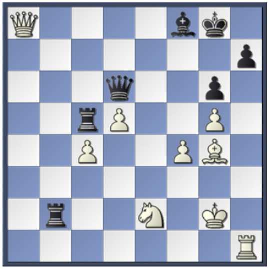
Duvenaud *et al.* (NIPS 2015)

Learning on manifolds



The *MoNet* framework, by Monti *et al.* (CVPR 2017)

Modelling multi-agent interactions



The *VAIN* framework, by Hoshen (NIPS 2017)

Cortical mesh segmentation

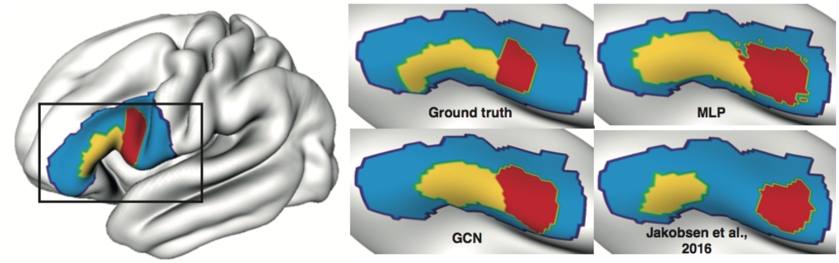


Figure 1: Region segmentation produced by the different models evaluated on the same validation sample.

Cucurull *et al.* (NIPS BigNeuro 2017)
Currently preparing an extended version to submit to MICCAI. . .

Thank you!

Questions?

`petar.velickovic@cst.cam.ac.uk`

`http://www.cst.cam.ac.uk/~pv273/`

`https://github.com/PetarV-/GAT`