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.

\[
\begin{align*}
\vec{f}_1 &\to y_1 \\
\vec{f}_2 &\to y_2 \\
\vec{f}_3 &\to y_3 \\
\vec{f}_4 &\to y_4
\end{align*}
\]

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

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

\[ \vec{f}_1, \vec{f}_2, \vec{f}_3, \vec{f}_4 \]
\[ y_1, y_2, y_3, y_4 \]

- Four examples, features \((\vec{f}_i)\) and labels \((y_i)\).
- Good enough for science. **Not Aperture Science!**
- Gentlemen, I give you **graphs**. *The inputs of tomorrow!*
Graphs are everywhere!
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*, \( F \in \mathbb{R}^{N \times F} \), with \( F \) features in each of the \( N \) nodes, and an *adjacency matrix*, \( A \in \mathbb{R}^{N \times N} \).
  - **Output**: a matrix of *node class probabilities*, \( Y \in \mathbb{R}^{N \times C} \), such that \( Y_{ij} = P(\text{Node } i \in \text{Class } j) \).

- We also assume, for simplicity, that the edges are **unweighted** and **undirected**:  
  - That is, \( A_{ij} = A_{ji} = \begin{cases} 1 & i \leftrightarrow j \\ 0 & \text{otherwise} \end{cases} \)

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)*!
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 $F' = \sigma(FW)$, where $W \in \mathbb{R}^{F \times F'}$ is a shared and learnable weight matrix, and $\sigma$ 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)!
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!
Essentially, the loss function to optimise is augmented with a (dis)similarity constraint, $L_{\text{sim}}$:

$$L = L_0 + \lambda L_{\text{sim}}$$

where $L_0$ is the usual supervised learning loss (e.g. cross-entropy), and $\lambda$ is a hyperparameter.

One way to define $L_{\text{sim}}$:

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

where $\mathcal{N}_i$ is the *neighbourhood* of node $i$, $\tilde{h}_i$ is (one of) its hidden layer’s outputs, and $m$ is a hyperparameter.
An alternative to augmenting the loss function is first learning some **structural features**, $\Phi_i$, for each node $i$ (these will not depend on $\vec{f}_i$, but on the graph structure)!

Then, use $\vec{f}_i \parallel \Phi_i$ as the input to the shared classifier (where $\parallel$ 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 $\Phi_i$ for each node $i$.

- Sample a random walk $W_i$, starting from node $i$.

- For node $x$ at step $j$, $x = W_i[j]$, and a node $y$ at step $k \in [j - w, j + w]$, $y = W_i[k]$, modify $\Phi_x$ to maximise $\log P(y|\Phi_x)$ (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.
Expressing the full $P(y|\vec{Φ}_x)$ distribution directly, even for a single layer neural network, where

$$P(y|\vec{Φ}_x) = \text{softmax}(\vec{w}_y^T \vec{Φ}_x) = \frac{\exp (\vec{w}_y^T \vec{Φ}_x)}{\sum_z \exp (\vec{w}_z^T \vec{Φ}_x)}$$

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

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

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 $\Pr(v_3 | \Phi(v_1))$ and $\Pr(v_5 | \Phi(v_1))$ over sequences of probability distributions corresponding to the paths starting at the root and ending at $v_3$ and $v_5$. The representation $\Phi$ is updated to maximize the probability of $v_1$ co-occurring with its context $\{v_3, v_5\}$. 
Incorporating labels and features: \textit{Planetoid}

- Methods such as DeepWalk are still favourable when dealing with \textit{fully unsupervised} graph problems, as they don’t depend on having any labels or features in the nodes!

- However, if we have labels/features, \textbf{why not use them}?

- The essence behind \textbf{Planetoid} \textit{(Predicting Labels And Neighbours with Embeddings Transductively Or Inductively from Data)}, by Yang \textit{et al.} (ICML 2016).
Planetoid’s sampling strategy: *Negative sampling*

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

- Essentially, use a binary classifier:

  $$P(\gamma|\Phi_x, \vec{w}_y) = \sigma(\vec{w}_y^T \Phi_x)$$

  where $\sigma$ 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 \(\gamma = 0\).
Consider this example graph, with three labelled nodes. I will now illustrate the two phases of Planetoid.
Sample from a random walk—can take e.g. nodes 1 and 4 with $\gamma = 1$, and nodes 1 and 5 with $\gamma = 0$. 
Sample given the labels—can take e.g. nodes 2 and 3 with $\gamma = 1$, and nodes 3 and 5 with $\gamma = 0$. 
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. :)
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).
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.
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):

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

  $$\vec{h}_i^{(t)} = \tanh \left( W \vec{a}_i^{(t)} \right)$$

- 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.
The full propagation model is as follows:

\[
\begin{align*}
\tilde{a}_i^{(t)} &= b_i + \sum_{j \in \mathcal{N}_i} \tilde{r}_j^{(t-1)} \\
\tilde{r}_i^{(t)} &= \sigma \left( \mathbf{W}^r \tilde{a}_i^{(t)} + \mathbf{U}^r \tilde{h}_i^{(t-1)} \right) \\
\tilde{z}_i^{(t)} &= \sigma \left( \mathbf{W}^z \tilde{a}_i^{(t)} + \mathbf{U}^z \tilde{h}_i^{(t-1)} \right) \\
\tilde{h}_i^{(t)} &= \tanh \left( \mathbf{W} \tilde{a}_i^{(t)} + \mathbf{U} \left( \tilde{r}_i^{(t)} \odot \tilde{h}_i^{(t-1)} \right) \right) \\
\tilde{h}_i^{(t)} &= (1 - \tilde{z}_i^{(t)}) \odot \tilde{h}_i^{(t-1)} + \tilde{z}_i^{(t)} \odot \tilde{h}_i^{(t)}
\end{align*}
\]

where \(\odot\) is elementwise vector multiplication, \(\tilde{r}_i\) and \(\tilde{z}_i\) are reset and update gates, and \(\sigma\) is the logistic sigmoid function.
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!
Convolution on images

\[ I \times K = I * K \]
Convolution on images

\[ I \ast K = I \ast K \]

\[
\begin{array}{cccccccc}
0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\
0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]

\[
\begin{array}{cccc}
1 & 0 & 1 \\
0 & 1 & 0 \\
1 & 0 & 1 \\
\end{array}
\]

\[
\begin{array}{cccccccc}
1 & 4 & 3 & 4 & 1 \\
1 & 2 & 4 & 3 & 3 \\
1 & 2 & 3 & 4 & 1 \\
1 & 3 & 3 & 1 & 1 \\
3 & 3 & 1 & 1 & 0 \\
\end{array}
\]
Convolution on images

\[ I \ast K = \begin{array}{cccc}
1 & 4 & 3 & 4 & 1 \\
1 & 2 & 4 & 3 & 3 \\
1 & 2 & 3 & 4 & 1 \\
1 & 3 & 3 & 1 & 1 \\
3 & 3 & 1 & 1 & 0 
\end{array} \]
Convolution on images

\[
I \ast K = \begin{bmatrix}
0 & 1 & 1 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 1 & 1 & 0 \\
0 & 0 & 0 & 1 & 1 & 0 & 0 \\
0 & 0 & 1 & 1 & 0 & 0 & 0 \\
0 & 1 & 1 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 & 0 & 0 & 0 \\
\end{bmatrix} \times \begin{bmatrix}
1 & 0 & 1 \\
0 & 1 & 0 \\
1 & 0 & 1 \\
\end{bmatrix} = \begin{bmatrix}
1 & 4 & 3 & 4 & 1 \\
1 & 2 & 4 & 3 & 3 \\
1 & 2 & 3 & 4 & 1 \\
1 & 3 & 3 & 1 & 1 \\
3 & 3 & 1 & 1 & 0 \\
\end{bmatrix}
\]
Desirable properties for a graph convolutional layer:

- **Computational efficiency** (independent of input size) and **storage efficiency** (on the order of $O(V + E)$);
- **Fixed** number of parameters;
- **Localisation** (acts on a local neighbourhood of a node);
- Specifying **different importances** to different neighbours;
- Applicability to **inductive problems**.

Unfortunately, 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!
A large class of popular approaches attempts to define a convolutional operation by operating on the graph in the \textbf{spectral domain}, leveraging the \textit{convolution theorem}.

These approaches leverage the \textbf{graph Laplacian matrix}, $L$, defined as $L = D - A$, where $D$ is the degree matrix (diagonal matrix with $D_{ii} = \text{deg}(i)$) and $A$ is the adjacency matrix.

Alternately, we may use the \textbf{normalised graph Laplacian}, $	ilde{L} = I - D^{-1/2} AD^{-1/2}$. 
Graph Laplacian example

\[ L = \begin{bmatrix}
2 & -1 & 0 & 0 & -1 & 0 \\
-1 & 3 & -1 & 0 & -1 & 0 \\
0 & -1 & 2 & -1 & 0 & 0 \\
0 & 0 & -1 & 3 & -1 & -1 \\
-1 & -1 & 0 & -1 & 3 & 0 \\
0 & 0 & 0 & -1 & 0 & 1 \\
\end{bmatrix} \]
The Laplacian is symmetric and positive semi-definite; we can therefore diagonalise it as $L = U \Lambda U^T$, where $\Lambda$ is a diagonal matrix of its eigenvalues.

This means that multiplying the feature matrix by $U^T$ 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).
To convolve two signals using the convolution theorem:

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

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!
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 $k$, $T_k$:

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

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

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

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

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

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

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

  The operator is $K$-localised!
To avoid issues with exploding or vanishing signals, typically a scaled version of $L$ is fed into the algorithm:

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

where $\lambda_{\text{max}}$ 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 $3 \times 3$ 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.
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:

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

which improves computational performance on larger graphs and predictive performance on small training sets.

However, the previous issue is *still there*...
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!

To make further progress, we have to move on to non-spectral approaches...
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 $H_d$ 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.
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.
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!
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*
A rather exciting development in this direction concerns **self-attention**; a scenario where the input *attends over itself*:

\[
\alpha_{ij} = a(\vec{h}_i, \vec{h}_j)
\]

\[
\vec{h}'_i = \sum_j \text{softmax}_j(\alpha_{ij}) \vec{h}_j
\]

where \(a(\vec{x}, \vec{y})\) is a simple neural network.

- 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.
My recent ICLR 2018 submission, in collaboration with MILA, on **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:

\[ e_{ij} = a(W\hat{h}_i, W\hat{h}_j) \]
\[ \alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k \in N_i} \exp(e_{ik})} \]
\[ \hat{h}_i' = \sigma \left( \sum_{j \in N_i} \alpha_{ij} W\hat{h}_j \right) \]

Some further optimisations (like multi-head attention and dropout on the \( \alpha_{ij} \) 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!
- (Theoretically*) **storage efficient**—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 found our “silver bullet” convolution!

- How well does it perform?
### Datasets under study

Table: Summary of the datasets used in our experiments.

<table>
<thead>
<tr>
<th>Task</th>
<th>Cora</th>
<th>Citeseer</th>
<th>PPI</th>
</tr>
</thead>
<tbody>
<tr>
<td># Nodes</td>
<td>2708 (1 graph)</td>
<td>3327 (1 graph)</td>
<td>56944 (24 graphs)</td>
</tr>
<tr>
<td># Edges</td>
<td>5429</td>
<td>4732</td>
<td>818716</td>
</tr>
<tr>
<td># Features/Node</td>
<td>1433</td>
<td>3703</td>
<td>50</td>
</tr>
<tr>
<td># Classes</td>
<td>7</td>
<td>6</td>
<td>121 (multilabel)</td>
</tr>
<tr>
<td># Training Nodes</td>
<td>140</td>
<td>120</td>
<td>44906 (20 graphs)</td>
</tr>
<tr>
<td># Validation Nodes</td>
<td>500</td>
<td>500</td>
<td>6514 (2 graphs)</td>
</tr>
<tr>
<td># Test Nodes</td>
<td>1000</td>
<td>1000</td>
<td>5524 (2 graphs)</td>
</tr>
</tbody>
</table>
Results on Cora/Citeseer

<table>
<thead>
<tr>
<th>Method</th>
<th>Cora</th>
<th>Citeseer</th>
</tr>
</thead>
<tbody>
<tr>
<td>MLP</td>
<td>55.1%</td>
<td>46.5%</td>
</tr>
<tr>
<td>ManiReg</td>
<td>59.5%</td>
<td>60.1%</td>
</tr>
<tr>
<td>SemiEmb</td>
<td>59.0%</td>
<td>59.6%</td>
</tr>
<tr>
<td>LP</td>
<td>68.0%</td>
<td>45.3%</td>
</tr>
<tr>
<td>DeepWalk</td>
<td>67.2%</td>
<td>43.2%</td>
</tr>
<tr>
<td>ICA</td>
<td>75.1%</td>
<td>69.1%</td>
</tr>
<tr>
<td>Planetoid</td>
<td>75.7%</td>
<td>64.7%</td>
</tr>
<tr>
<td>Chebyshev</td>
<td>81.2%</td>
<td>69.8%</td>
</tr>
<tr>
<td>GCN</td>
<td>81.5%</td>
<td>70.3%</td>
</tr>
<tr>
<td><strong>GAT (ours)</strong></td>
<td><strong>83.3%</strong></td>
<td><strong>74.0%</strong></td>
</tr>
</tbody>
</table>

Improvement w.r.t GCN: 1.8% for Cora, 3.7% for Citeseer.
## Results on PPI

### Inductive

<table>
<thead>
<tr>
<th>Method</th>
<th>PPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random</td>
<td>0.396</td>
</tr>
<tr>
<td>MLP</td>
<td>0.422</td>
</tr>
<tr>
<td>GraphSAGE-GCN</td>
<td>0.500</td>
</tr>
<tr>
<td>GraphSAGE-mean</td>
<td>0.598</td>
</tr>
<tr>
<td>GraphSAGE-LSTM</td>
<td>0.612</td>
</tr>
<tr>
<td>GraphSAGE-pool</td>
<td>0.600</td>
</tr>
<tr>
<td><strong>GAT (ours)</strong></td>
<td><strong>0.942</strong></td>
</tr>
</tbody>
</table>

Improvement w.r.t GraphSAGE: 33.0%
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 submission)
Molecular fingerprinting

Fragments most activated by pro-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

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

<table>
<thead>
<tr>
<th>Method</th>
<th>Jaccard</th>
</tr>
</thead>
<tbody>
<tr>
<td>NodeMLP</td>
<td>0.56 (0.06)</td>
</tr>
<tr>
<td>NodeAVG</td>
<td>0.66 (0.06)</td>
</tr>
<tr>
<td>Jakobsen et al. [9]</td>
<td>0.68 (0.06)</td>
</tr>
<tr>
<td>GraphMLP</td>
<td>0.68 (0.06)</td>
</tr>
<tr>
<td>GCN</td>
<td>0.73 (0.06)</td>
</tr>
</tbody>
</table>

Table 1: Cross-validation dice scores on the HPC.

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

Questions?

petar.velickovic@cst.cam.ac.uk