In this talk:
Neural networks for graph-structured data

(Graph Neural Networks; GNNs)
Fantastic GNNs in the Wild
Molecules are graphs!

- A very natural way to represent molecules is as a graph
  - **Atoms** as nodes, **bonds** as edges
  - Features such as **atom type**, **charge**, **bond type**...
GNNs for molecule classification

- Interesting task to predict is, for example, whether the molecule is a potent drug.
  - Can do binary classification on whether the drug will inhibit certain bacteria. \textit{(E.coli)}
  - Train on a \textit{curated dataset} for compounds where response is known.

![Molecule structure]
Follow-up study

- Once trained, the model can be applied to any molecule.
  - Execute on a large dataset of known candidate molecules.
  - Select the ~top-100 candidates from your GNN model.
  - Have chemists thoroughly investigate those (after some additional filtering).

- Discover a previously overlooked compound that is a highly potent antibiotic!

Halicin
...Achieve wide acclaim!

Arguably the most popularised success story of graph neural networks to date!

(Stokes et al., Cell’20)
...Achieve wide acclaim!

Arguably the most popularised **success story** of graph neural networks to date!

(Stokes et al., Cell'20)

**Powerful antibiotics discovered using AI**

Machine learning spots molecules that work even against ‘untreatable’ strains of bacteria.
...Achieve wide acclaim!

Arguably the most popularised success story of graph neural networks to date! (Stokes et al., Cell’20)
...Achieve wide acclaim!

Arguably the most popularised success story of graph neural networks to date! (Stokes et al., Cell’20)
Traffic maps are graphs!

Transportation maps (e.g. the ones found on Google Maps) naturally modelled as graphs.

Nodes could be intersections, and edges could be roads. (Relevant node features: road length, current speeds, historical speeds)
DeepMind’s ETA Prediction using GNNs in Google Maps

Run GNN on supersegment graph to estimate time of arrival (ETA) (graph regression).

Already deployed in several major cities, significantly reducing negative ETA outcomes!
GNNs are a **very hot** research topic.
Rich ecosystem of libraries

- PyTorch geometric: [github.com/rusty1s/pytorch_geometric](https://github.com/rusty1s/pytorch_geometric)
- Spektral: [github.com/deepmind/graph_nets](https://github.com/deepmind/graph_nets)
- JAX: [github.com/deepmind/jraph](https://github.com/deepmind/jraph)
Rich ecosystem of datasets

OGB: ogb.stanford.edu
TUDataset: graphlearning.io

Benchmarkeding Graph Neural Networks: github.com/graphdeeplearning/benchmarking-gnns
2

Talk roadmap
What will we cover today?

- Hopefully I’ve given you a convincing argument for why GNNs are useful to study
  - For more details and applications, please see e.g. my EEML 2020 talk

- **My aim for today:** provide good **blueprints** and **contexts** for studying the field
  - Derive GNNs from first principles
  - Position this in context of several *independently-studied* derivations of GNNs
    - Often using *drastically* different mathematical tools...
  - Look to the *past*: how GNN-like models emerged in historical ML research
  - Look to the *present*: some *immediate* lines of research interest
  - Look to the *future*: how our blueprint *generalises* beyond graph-structured inputs

- Hopefully my perspective is of use both to newcomers and seasoned GNN practitioners
  - *Any* and *all* feedback *very welcome*!
What is the content based on?

- GNN derivation + further horizons inspired by my work on geometric deep learning
  - Ongoing collaboration with Joan Bruna, Michael Bronstein and Taco Cohen

- Various contexts of GNN study inspired by Will Hamilton’s GRL Textbook (esp. Chapter 7)
  - Highly recommended!

- Historical contexts developed with input of several researchers
  - Thanks to Yoshua Bengio, Marco Gori, Jürgen Schmidhuber, Christian Merkwirth and Marwin Segler

- But of course, any errors and omissions are mine alone.
Disclaimer before advancing

- **My talk content** is geared to a *general* Computer Science audience
  - We will construct “useful” functions operating over graphs
  - We will use concepts commonly encountered in a CS curriculum

- **Implementation** requires background in machine learning with deep neural networks
  - Useful resource to get started: “*Deep Learning*” by Goodfellow, Bengio and Courville
    - [https://www.deeplearningbook.org/](https://www.deeplearningbook.org/)

- I recently compiled a list of many useful GNN resources in a *Twitter thread*
  - [https://twitter.com/PetarV_93/status/1306689702020382720](https://twitter.com/PetarV_93/status/1306689702020382720)

- When you feel ready, I **highly** recommend Aleksa Gordić’s GitHub repository on GATs:
  - [https://github.com/gordicaleksa/pytorch-GAT](https://github.com/gordicaleksa/pytorch-GAT)
  - Arguably the most *gentle* introduction to GNN implementations
Towards GNNs from first principles
Towards a neural network for graphs

- We will now work towards defining a GNN from first principles

- What properties are useful for operating meaningfully on graphs?
  - Specifically: what symmetries and invariances must a GNN preserve?
    - Let’s revisit a known example...
Convolution on images

\[
\begin{array}{cccccc}
0 & 1 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 1 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 & 0 & 0 \\
\end{array}
\times
\begin{array}{ccc}
1 & 0 & 1 \\
0 & 1 & 0 \\
1 & 0 & 1 \\
\end{array}
\Rightarrow
\begin{array}{cccccc}
1 & 4 & 3 & 4 & 1 \\
1 & 2 & 4 & 3 & 3 \\
1 & 2 & 4 & 1 & 1 \\
3 & 3 & 1 & 1 & 0 \\
\end{array}
\]
Convolution on images

\[
\begin{array}{cccc}
0 & 1 & 1 & 1 \\
0 & 0 & 1 & 1 \\
0 & 0 & 0 & 1 \\
0 & 1 & 1 & 0 \\
1 & 1 & 0 & 0 \\
\end{array}
\times
\begin{array}{c}
\times_1 \\
\times_0 \\
\times_1 \\
\times_1 \\
\times_1 \\
\end{array}
\times
\begin{array}{c}
1 & 0 & 1 \\
0 & 1 & 0 \\
1 & 0 & 1 \\
\end{array}
= 
\begin{array}{cccc}
1 & 4 & 3 & 4 & 1 \\
1 & 2 & 4 & 3 & 3 \\
1 & 2 & 3 & 4 & 1 \\
1 & 3 & 3 & 1 & 1 \\
3 & 3 & 1 & 1 & 0 \\
\end{array}
\]
Convolution on images

\[
\begin{bmatrix}
0 & 1 & 1 & 1 & 0 & 0 \\
0 & 0 & 1 & 1 & 1 & 0 \\
0 & 0 & 0 & 1 & 1 & 0 \\
0 & 0 & 1 & 1 & 0 & 0 \\
0 & 1 & 1 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 & 0 & 0 \\
\end{bmatrix}
\ast
\begin{bmatrix}
1 & 0 & 1 \\
0 & 1 & 0 \\
1 & 0 & 1 \\
\end{bmatrix}
= 
\begin{bmatrix}
1 & 4 & 3 & 4 & 1 \\
1 & 2 & 4 & 3 & 3 \\
1 & 2 & 3 & 4 & 1 \\
1 & 3 & 3 & 1 & 1 \\
3 & 3 & 1 & 1 & 0 \\
\end{bmatrix}
\]
Convolution on images

\[
\begin{bmatrix}
0 & 1 & 1 & 0 & 0 \\
0 & 0 & 1 & 1 & 0 \\
0 & 0 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 & 0 \\
1 & 1 & 0 & 0 & 0
\end{bmatrix}
\begin{bmatrix}
1 & 0 & 1 \\
0 & 1 & 0 \\
1 & 0 & 1
\end{bmatrix}
\Rightarrow
\begin{bmatrix}
1 & 4 & 3 & 4 & 1 \\
1 & 2 & 4 & 3 & 3 \\
1 & 2 & 3 & 4 & 1 \\
1 & 3 & 3 & 1 & 1 \\
3 & 3 & 1 & 1 & 0
\end{bmatrix}
\]
Convolutional neural network invariances

- Convolutional neural nets respect **translational invariance**
- Patterns are interesting irrespective of *where* they are in the image
- **Locality**: neighbouring pixels relate much more strongly than distant ones
- What about *arbitrary* graphs?
Isomorphism-preserving transformation

- The nodes of a graph are not assumed to be in any order
- That is, we would like to get the same results for two isomorphic graphs
- To see how to enforce this, we will define new terms...
Permutation invariance and equivariance
Learning on sets: Setup

- For now, assume the graph has no edges (e.g. set of nodes, V).

- Let $x_i \in \mathbb{R}^k$ be the features of node $i$.

- We can stack them into a node feature matrix of shape $n \times k$:

$$X = (x_1, \ldots, x_n)^T$$

- That is, the $i$th row of $X$ corresponds to $x_i$.

- Note that, by doing so, we have specified a node ordering! We would like the result of any neural networks to not depend on this.
Permutations and permutation matrices

- It will be useful to think about the operations that change the node order
  - Such operations are known as **permutations** (there are $n!$ of them)
  - e.g. a permutation $(2, 4, 1, 3)$ means $y_1 \leftarrow x_2$, $y_2 \leftarrow x_4$, $y_3 \leftarrow x_1$, $y_4 \leftarrow x_3$.

- To stay within linear algebra, each permutation defines an $n \times n$ matrix
  - Such matrices are called **permutation matrices**
  - They have exactly one 1 in every row and column, and zeros everywhere else
  - Their effect when left-multiplied is to permute the rows of $X$, like so:

\[
P_{(2, 4, 1, 3)}X = \begin{bmatrix}
0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 \\
\end{bmatrix} \quad \begin{bmatrix}
\_ & \_ & \_ & \_ \\
\_ & \_ & \_ & \_ \\
\_ & \_ & \_ & \_ \\
\_ & \_ & \_ & \_ \\
\end{bmatrix} = \begin{bmatrix}
\_ & \_ & \_ & \_ \\
\_ & \_ & \_ & \_ \\
\_ & \_ & \_ & \_ \\
\_ & \_ & \_ & \_ \\
\end{bmatrix}
\]
Permutation invariance

- We want to design functions $f(X)$ over sets that will not depend on the order.

- Equivalently, applying a permutation matrix shouldn’t modify the result!

- We arrive at a useful notion of permutation invariance. We say that $f(X)$ is permutation invariant if, for all permutation matrices $P$:

$$f(PX) = f(X)$$

- One very generic form is the Deep Sets model (Zaheer et al., NeurIPS’17): $f(X) = \phi \left( \sum_{i \in V} \psi(x_i) \right)$ where $\psi$ and $\phi$ are (learnable) functions, e.g. MLPs.
  - The sum aggregation is critical! (other choices possible, e.g. max or avg)
Permutation **equivariance**

- Permutation-*invariant* models are a good way to obtain set-level outputs

- What if we would like answers at the node level?
  - We want to still be able to identify node outputs, which a permutation-invariant aggregator would destroy!

- We may instead seek functions that don’t change the node order
  - i.e. if we permute the nodes, it doesn’t matter if we do it before or after the function!

- Accordingly, we say that $f(X)$ is **permutation equivariant** if, for all permutation matrices $P$:
  
  $$f(PIX) = P f(X)$$
General blueprint for learning on sets

- Equivariance mandates that each node's row is unchanged by $f$. That is, we can think of equivariant set functions as transforming each node input $x_i$ into a latent vector $h_i$:

$$h_i = \psi(x_i)$$

where $\psi$ is any function, applied in isolation to every node. Stacking $h_i$ yields $H = f(X)$.

- We arrive at a general blueprint: (stacking) equivariant function(s), potentially with an invariant tail---yields (m)any useful functions on sets!

$$f(X) = \phi \left( \bigoplus_{i \in V} \psi(x_i) \right)$$

Here, $\bigoplus$ is a permutation-invariant aggregator (such as sum, avg or max).

(remark: this is typically as far as we can get with sets, without assuming or inferring additional structure)
5 Learning on graphs
Learning on graphs

- Now we augment the set of nodes with **edges** between them.
  - That is, we consider general $E \subseteq V \times V$.

- We can represent these edges with an **adjacency matrix**, $A$, such that:

$$
a_{ij} = \begin{cases} 
1 & (i, j) \in E \\
0 & \text{otherwise}
\end{cases}
$$

- Further additions (e.g. **edge features**) are possible but ignored for simplicity.

- Our main desiderata (**permutation** {in,equi}**variance**) still hold!
**Permutation invariance and equivariance on graphs**

- The main difference: node permutations now also accordingly act on the *edges*

- We need to appropriately permute both **rows** and **columns** of $A$
  - When applying a permutation matrix $P$, this amounts to $PAP^T$

- We arrive at updated definitions of suitable functions $f(X, A)$ over graphs:

  **Invariance:** \[ f(PX, PAP^T) = f(X, A) \]

  **Equivariance:** \[ f(PX, PAP^T) = Pf(X, A) \]
Locality on graphs: **neighbourhoods**

- On **sets**, we enforced equivariance by applying functions to every node in isolation.

- **Graphs** give us a broader context: a node’s **neighbourhood**
  - For a node $i$, its (1-hop) neighbourhood is commonly defined as follows:
    $$\mathcal{N}_i = \{ j : (i, j) \in \mathcal{E} \lor (j, i) \in \mathcal{E} \}$$
    
    **N.B.** we do not explicitly consider directed edges, and often we assume $i \in \mathcal{N}_i$

- Accordingly, we can extract the **multiset** of **features** in the neighbourhood
  $$X_{\mathcal{N}_i} = \{ x_j : j \in \mathcal{N}_i \}$$
  
  and define a **local** function, $g$, as operating over this multiset: $g(x_i, X_{\mathcal{N}_i})$. 
A recipe for **graph** neural networks

- Now we can construct permutation equivariant functions, $f(\mathbf{X}, \mathbf{A})$, by appropriately applying the local function, $g$, over *all* neighbourhoods:

$$f(\mathbf{X}, \mathbf{A}) = \begin{bmatrix}
- g(x_1, \mathbf{X}_{N_1}) & - \\
- g(x_2, \mathbf{X}_{N_2}) & - \\
\vdots & \\
- g(x_n, \mathbf{X}_{N_n}) & - 
\end{bmatrix}$$

- To ensure equivariance, we need $g$ to not depend on the **order** of the vertices in $\mathbf{X}_{Ni}$
  - Hence, $g$ should be permutation *invariant*!
A recipe for **graph** neural networks, visualised

\[ X_{\mathcal{N}_b} = \{x_a, x_b, x_c, x_d, x_e\} \]
How to use GNNs?
How to use GNNs?
How to use GNNs?

Inputs $(X, A)$

Latents $(H, A)$

Node classification $z_i = f(h_i)$
How to use GNNs?

Inputs
(X, A)

GNN

Latents
(H, A)

Node classification
\( z_i = f(h_i) \)

Graph classification
\( z_G = f(\bigoplus_{i \in V} h_i) \)
How to use GNNs?

Inputs $(X, A)$

Latents $(H, A)$

Node classification $z_i = f(h_i)$

Graph classification $z_G = f(\bigoplus_{i \in V} h_i)$

Link prediction $z_{ij} = f(h_i, h_j, e_{ij})$
Message passing on graphs
What’s in a GNN layer?

- As mentioned, we construct permutation-equivariant functions $f(X, A)$ over graphs by shared application of a local permutation-invariant $g(x_i, X_{Ni})$.
  - We often refer to $f$ as “GNN layer”, $g$ as “diffusion”, “propagation”, “message passing”

- Now we look at ways in which we can actually concretely define $g$.
  - Very intense area of research!

- Fortunately, almost all proposed layers can be classified as one of three spatial “flavours”.
The three “flavours” of GNN layers

Convolutional:
\[ h_i = \phi \left( x_i, \bigoplus_{j \in \mathcal{N}_i} c_{ij} \psi(x_j) \right) \]

Attentional:
\[ h_i = \phi \left( x_i, \bigoplus_{j \in \mathcal{N}_i} a(x_i, x_j) \psi(x_j) \right) \]

Message-passing:
\[ h_i = \phi \left( x_i, \bigoplus_{j \in \mathcal{N}_i} \psi(x_i, x_j) \right) \]
Convolutional GNN

- Features of neighbours aggregated with fixed weights, $c_{ij}$
  \[ h_i = \phi \left( \bigoplus_{j \in N_i} c_{ij} \psi(x_j) \right) \]

- Usually, the weights depend directly on $A$.
  - ChebyNet (Defferrard et al., NeurIPS’16)
  - GCN (Kipf & Welling, ICLR’17)
  - SGC (Wu et al., ICML’19)

- Useful for **homophilous** graphs and **scaling up**
  - When edges encode *label similarity*
Attentional GNN

- Features of neighbours aggregated with \textit{implicit} weights (via attention)
  \[ h_i = \phi \left( x_i, \bigoplus_{j \in \mathcal{N}_i} a(x_i, x_j)\psi(x_j) \right) \]

- Attention weight computed as \( a_{ij} = a(x_i, x_j) \)
  - MoNet (Monti et al., CVPR’17)
  - GAT (Veličković et al., ICLR’18)
  - GaAN (Zhang et al., UAI’18)

- Useful as “middle ground” w.r.t. capacity and scale
  - Edges need not encode homophily
  - But still compute scalar value in each edge
Message-passing GNN

- Compute arbitrary vectors ("messages") to be sent across edges
  \[ h_i = \phi \left( \bigoplus_{j \in \mathcal{N}_i} \psi(x_i, x_j) \right) \]

- Messages computed as \( m_{ij} = \psi(x_i, x_j) \)
  - Interaction Networks (Battaglia et al., NeurIPS’16)
  - MPNN (Gilmer et al., ICML’17)
  - GraphNets (Battaglia et al., 2018)

- Most generic GNN layer
  - May have scalability or learnability issues
  - Ideal for computational chemistry, reasoning and simulation
7 Perspectives on GNNs
This framework looks quite clean, *but*...

- We didn’t **start** researching GNNs from a blueprint like this.

- Graphs naturally arise **across** the sciences
  - Different disciplines found different tools to process them

- To give you a feel of the scale of diversity, I will now **survey** several prior and concurrent approaches to graph representation learning + to what extent they map to this blueprint.

- If you’ve read up on graph machine learning before, there’s a good chance you will have seen at least some of these.
Node embedding techniques
Node embedding techniques

- Some of the earliest “successes” of deep learning on graphs relied on finding good ways to embed nodes into vectors $h_u$ (below: $z_u$) using an encoder function
  - At the time, implemented as a look-up table!

Credits to Will Hamilton
What’s in a good representation?

- What makes an embedding “good”?
  - Graphs carry interesting **structure**!
  - Good node representations should **preserve** it.

- Simplest notion of graph structure is an **edge**.
  - Features of nodes \( i \) and \( j \) should be predictive of existence of edge \((i, j)\)!

- Yields a straightforward unsupervised objective
  - Optimise \( h_i \) and \( h_j \) to be **nearby** iff \((i, j) \in E\).
  - Can use standard **binary cross-entropy** loss:

\[
\sum_{(i,j) \in E} \log \sigma \left( h_i^T h_j \right) + \sum_{(i,j) \notin E} \log \left( 1 - \sigma \left( h_i^T h_j \right) \right)
\]
Random-walk objectives

- This link-prediction objective is a special case of random-walk objectives.

- **Redefine** the condition from \((i, j) \in E\) to \(i\) and \(j\) co-occur in a (short) random walk.

- Dominated unsupervised graph representation learning prior to GNNs!
  - DeepWalk (Perozzi et al., KDD’14)
  - node2vec (Grover & Leskovec, KDD’16)
  - LINE (Tang et al., WWW’15)
Local objectives **emulate** Conv-GNNs

- Random walk objectives inherently capture **local** similarities.
- But a (convolutional) GNN summarises local patches of the graph!
  - Neighbouring nodes tend to highly overlap in $n$-step neighbourhoods;
  - Therefore, a conv-GNN enforces similar features for neighbouring nodes **by design**.
Local objectives **emulate** Conv-GNNs

- Random walk objectives inherently capture **local** similarities.
- But a (convolutional) GNN **summarises local patches** of the graph!
  - Neighbouring nodes tend to highly overlap in $n$-step neighbourhoods;
  - Therefore, a conv-GNN enforces similar features for neighbouring nodes **by design**.

- From a representation perspective, DeepWalk-style models **emulate** a convolutional GNN!

**Corollary 1:** Random-walk objectives can **fail** to provide useful signal to GNNs!
  - First spotted within DGI (Veličković et al., ICLR’19)
  - Independently verified by SGC (Wu et al., ICML’19)
Parallels to NLP

- Note clear **correspondence** between *node embedding* techniques and *word embedding* techniques in NLP
  - nodes ~ words
  - random walks ~ sentences
  - “node2vec” ~ “word2vec”
  - The optimisation objectives are *near-equal*

- This correspondence continues even nowadays, with recent unsupervised graph representation learning techniques borrowing concepts from BERT.
  ("Strategies for pre-training graph neural networks"; Hu, Liu et al., ICLR’20)

- Speaking of NLP...
Natural Language Processing
Parallels from NLP

- It’s not only that NLP feeds into GNN design...
  - Words in a sentence **interact**
    - Nontrivially and **non-sequentially**
    - We may want to use a **graph** over them
  - But *what* is this graph?

- A common assumption is to assume a **complete graph**
  - Then let the network **infer** relations

- If you’re at all involved with NLP, this should sound **familiar**...
A note on Transformers

Transformers are Graph Neural Networks!
- Fully-connected graph
- Attentional flavour

The sequential structural information is injected through the **positional embeddings**. Dropping them yields a fully-connected GAT model.

Attention can be seen as inferring **soft adjacency**.

See Joshi (The Gradient; 2020).
Spectral GNNs
Look to the Fourier transform

The **convolution theorem** defines a very attractive identity:

\[
(x * y)(\xi) = \hat{x}(\xi) \cdot \hat{y}(\xi)
\]

"convolution in the time domain is multiplication in the frequency domain"

- This could give us a ‘detour’ to defining convolutions on graphs
  - Pointwise multiplication is **easy**!
  - But what are the ‘domains’ in this case?

- We will first see how graphs arise in **discrete** sequences.
Rethinking the convolution on sequences

- We can imagine a sequence as a cyclical grid graph, and a convolution over it:

- NB this defines a circulant matrix $C([b, c, 0, 0, ..., 0, a])$ s.t. $H = f(X) = CX$

$$f(X) = \begin{bmatrix} b & c & a \\ a & b & c \\ \vdots & \vdots & \vdots \\ a & b & c \\ c & a & b \end{bmatrix} \begin{bmatrix} \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \end{bmatrix}$$
Properties of circulants, and their eigenvectors

● Circulant matrices **commute**!
  ○ That is, $C(v)C(w) = C(w)C(v)$, for any parameter vectors $v, w$.

● Matrices that commute are **jointly diagonalisable**.
  ○ That is, the eigenvectors of one are eigenvectors of all of them!

● Conveniently, the eigenvectors of circulants are the *discrete Fourier basis*

$$\phi_\ell = \frac{1}{\sqrt{n}} \left(1, e^{\frac{2\pi i \ell}{n}}, e^{\frac{4\pi i \ell}{n}}, \ldots, e^{\frac{2\pi i (n-1)\ell}{n}}\right)^\top, \quad \ell = 0, 1, \ldots, n - 1$$

● This can be easily computed by studying $C([0, 1, 0, 0, 0, \ldots])$, which is the **shift matrix**.
The DFT and the convolution theorem

- If we stack these Fourier basis vectors into a matrix: $\Phi = (\phi_0, \ldots, \phi_{n-1})$
  - We recover the discrete Fourier transform (DFT), as multiplication by $\Phi^*$. (conjugate transpose)

- We can now eigendecompose any circulant as $C(\theta) = \Phi \Lambda \Phi^*$
  - Where $\Lambda$ is a diagonal matrix of its eigenvalues, $\hat{\theta}$

- The convolution theorem naturally follows:

$$f(X) = C(\theta)X = \Phi \Lambda \Phi^* X = \Phi \begin{bmatrix} \hat{\theta}_0 \\ \vdots \\ \hat{\theta}_{n-1} \end{bmatrix}$$

$$\Phi^* X = \Phi (\hat{\theta} \circ \hat{X})$$

- Now, as long as we know $\Phi$, we can express our convolution using $\hat{\theta}$ rather than $\theta$
What we have covered so far

Key idea: we don’t need to know the circulant if we know its eigenvalues!
What about graphs?

- On graphs, convolutions of interest need to be more generic than circulants.
  - But we can still use the concept of joint eigenbases!
  - If we know a “graph Fourier basis”, $\Phi$, we can only focus on learning the eigenvalues.

- For grids, we wanted our convolutions to commute with shifts.
  - We can think of the shift matrix as an adjacency matrix of the grid
  - This generalises to non-grids!
  - For the grid convolution on $n$ nodes, $\Phi$ was always the same ($n$-way DFT).
  - Now every graph will have its own $\Phi$!

- Want our convolution to commute with $A$, but we cannot always eigendecompose $A$!
  - Instead, use the graph Laplacian matrix, $L = D - A$, where $D$ is the degree matrix.
    - Captures all adjacency properties in mathematically convenient way!
Example Laplacian

\[
L = \begin{bmatrix}
2 & -1 & 0 & 0 & -1 & 0 \\
-1 & 3 & -1 & 0 & -1 & 0 \\
0 & -1 & 2 & -1 & 0 & 0 \\
0 & 0 & -1 & 3 & -1 & -1 \\
-1 & -1 & 0 & -1 & 3 & 0 \\
0 & 0 & 0 & -1 & 0 & 1 \\
\end{bmatrix}
\]
Graph Fourier Transform

Assuming undirected graphs, $L$ is:

- **Symmetric** ($L^T = L$)
- **Positive semi-definite** ($x^T L x \geq 0$ for all $x \in \mathbb{R}^{|V|}$)
- This means we will be able to *eigendecompose* it!

This allows us to re-express $L = \Phi \Lambda \Phi^*$, as before.

- Changing the eigenvalues in $\Lambda$ expresses *any* operation that commutes with $L$.
- Commonly referred to as the **graph Fourier transform** (Bruna et al., ICLR’14)

Now, to convolve with some feature matrix $X$, do as follows (the diagonal can be *learnable*):

$$f(X) = \Phi \begin{bmatrix} \hat{\theta}_0 \\ \vdots \\ \hat{\theta}_{n-1} \end{bmatrix} \Phi^* X$$
Spectral GNNs in practice

- However, directly learning the eigenvalues is typically inappropriate:
  - Not **localised**, doesn’t **generalise** to other graphs, computationally **expensive**, etc.

- Instead, a common solution is to make the eigenvalues related to $\Lambda$, the eigenvalues of $L$
  - Commonly by a degree-$k$ polynomial function, $p_k$
  - Yielding $f(x) = \Phi p_k(\Lambda) \Phi^* x = p_k(L)x$
  - Popular choices include:
    - **Cubic splines** (Bruna et al., ICLR’14)
    - **Chebyshev polynomials** (Defferrard et al., NeurIPS’16)
    - **Cayley polynomials** (Levie et al., Trans. Sig. Proc.’18)

- **NB** by using a polynomial in $L$, we have defined a **conv-GNN**!
  - With coefficients defined by $c_{ij} = (p_k(L))_{ij}$
  - Most efficient spectral approaches “**spatialise**” themselves in similar ways
  - The “spatial–spectral” divide is often **not really a divide**!
The Transformer positional encodings and beyond

- Lastly, another look at Transformers.
  - Transformers signal that the input is a **sequence** of words by using *position embeddings*
    - Sines/cosines sampled depending on position
  - Very similar to the DFT eigenvectors!

- Positional embeddings could hence be interpreted as eigenvectors of the grid graph
  - Which is the only assumed ‘underlying’ connectivity between the words

- We can use this idea to run Transformers over *general* graph structures!
  - Just feed some eigenvectors of the graph Laplacian (columns of \( \Phi \))
  - See the **Graph Transformer** from Dwivedi & Bresson (2021)
Probabilistic Graphical Models
Probabilistic modelling

- We’ve so far used edges in a graph to mean any kind of relation between nodes.
- Taking a more probabilistic view, we can treat nodes as random variables, and interpret edges as dependencies between their distributions.
  - This gives rise to probabilistic graphical models (PGMs).
  - They help us ignore relations between variables when computing joint probabilities.
Markov random fields

- One particular PGM of interest here is the Markov random field (MRF).
  - It allows us to decompose the joint into a product of edge potentials

- Specifically, we assume nodes are represented by inputs $X$ and latents $H$
  - Inputs and latents are related for every node in isolation
  - Latents are related according to the edges of the graph

- This yields the following decomposition of the joint

\[
p(X, H) \propto \prod_{u \in V} \Phi(x_u, h_u) \prod_{(u, v) \in E} \Psi(h_u, h_v)
\]

where $\Phi$ and $\Psi$ are real-valued potential functions.
Mean-field inference

- To embed nodes, we need to sample from the posterior, $p(H | X)$.
  - Generally *intractable*, even if we know the exact potential functions.

- One popular method of resolving this is **mean-field variational inference**
  - Assume that posterior can be approximated by a product of node-level densities
    $$p(H|X) \approx \prod_{u \in \mathcal{V}} q(h_u)$$
    where $q$ is a well-defined density, that is easy to compute and sample (e.g. Gaussian).

- We then obtain the parameters of $q$ by minimising the distance (e.g. KL-divergence) to the true posterior, $KL(\prod_u q(h_u) \| p(H | X))$.

- Minimising the KL is intractable, but it admits a favourable approximate algorithm.
GNNs strike again!

- Using variational inference techniques (out of scope), we can iteratively update $q$, starting from some initial guess $q^{(0)}(h)$, as follows:

$$\log q^{(t+1)}(h_i) = c_i + \log \Phi(x_i, h_i) + \sum_{j \in \mathcal{N}_i} \int_{\mathbb{R}^k} q^{(t)}(h_j) \log \Psi(h_i, h_j) \, dh_i$$

- See anything familiar? :)

Using variational inference techniques (out of scope), we can iteratively update $q$, starting from some initial guess $q^{(0)}(h)$, as follows:

$$
\log q^{(t+1)}(h_i) = c_i + \log \Phi(x_i, h_i) + \sum_{j \in N_i} \int_{\mathbb{R}^k} q^{(t)}(h_j) \log \Psi(h_i, h_j) \, dh_i
$$

This aligns very nicely with computations of a (message-passing) GNN!
GNNs and PGMs, more broadly

- Based on this idea, **structure2vec** (Dai et al., ICML’16) embed mean-field inference within computations of a GNN.
  - Key difference: in PGMs, we expect potential functions specified and known upfront
  - Here, they are defined implicitly, within the latents of a GNN.

- The structure2vec GNN itself is not unlike a typical MPNN.

- Recently, there are other approaches that unify GNNs with PGM-like computations:
  - CRF-GNNs (Gao et al., KDD’19)
  - GMNNs (Qu et al., ICML’19)
  - ExpressGNN (Zhang et al., ICLR’20)
  - Tail-GNNs (Spalević et al., ICML’20 GRL+)
How powerful are Graph Neural Networks?

- GNNs are a powerful tool for processing real-world graph data
  - But they won’t solve any task specified on a graph accurately!

- Canonical example: deciding graph isomorphism
  - Am I able to use my GNN to distinguish two non-isomorphic graphs? ($h_{G_1} \neq h_{G_2}$)
  - If I can’t, any kind of task discriminating them is hopeless

- Permutation invariance mandates that two isomorphic graphs will always be indistinguishable, so this case is OK.
Weisfeiler-Leman Test

- Simple but powerful way of distinguishing: pass random hashes of sums along the edges
  - Connection to conv-GNNs spotted very early; e.g. by GCN (Kipf & Welling, ICLR’17)
- It explains why untrained GNNs work well!
  - Untrained ~ random hash
- The test does **fail** at times, however:

---

Algorithm 1: WL-1 algorithm (Weisfeiler & Lehmann, 1968)

**Input:** Initial node coloring \((h_1^{(0)}, h_2^{(0)}, \ldots, h_N^{(0)})\)

**Output:** Final node coloring \((h_1^{(T)}, h_2^{(T)}, \ldots, h_N^{(T)})\)

\[
t \leftarrow 0;
\]

**repeat**

\[
\begin{align*}
    h_i^{(t+1)} &\leftarrow \text{hash}\left(\sum_{j \in N_i} h_j^{(t)}\right);
    \\
t &\leftarrow t + 1;
\end{align*}
\]

**until** stable node coloring is reached;

\[
h_i = \phi \left( x_i^{(0)} \bigoplus \bigoplus_{j \in N_i} c_{ij} \psi(x_j) \right)
\]
GNNs are **no more powerful** than 1-WL

- Over *discrete features*, GNNs can only be **as powerful** as the 1-WL test described before!

- One important condition for maximal power is an *injective* aggregator (e.g. *sum*)

  ![Diagram showing the comparison of operations: sum - multiset, mean - distribution, max - set.]

- Graph isomorphism network (**GIN**; Xu *et al.*, ICLR’19) proposes a simple, maximally-expressive GNN, following this principle

\[
h_v^{(k)} = \text{MLP}^{(k)} \left( \left(1 + \epsilon^{(k)}\right) \cdot h_v^{(k-1)} + \sum_{u \in \mathcal{N}(v)} h_u^{(k-1)} \right)
\]
Higher-order GNNs

- We can make GNNs stronger by analysing failure cases of 1-WL!
- For example, just like 1-WL, GNNs cannot detect closed triangles
  - Augment nodes with randomised/positional features
    - Explored by RP-GNN (Murphy et al., ICML’19) and P-GNN (You et al., ICML’19)
    - See also: Sato et al. (SDM’21)
  - Can also literally count interesting subgraphs (Bouritsas et al., 2020)
- $k$-WL labels subgraphs of $k$ nodes together.
  - Exploited by 1-2-3-GNNs (Morris et al., AAAI’19)
- Further avenues of interest:
  - Invariant and equivariant GNNs (Maron et al. (ICLR’19))
  - Directional graph networks (DGNs) (Beaini, Passaro et al. (2020))
Going beyond \textit{discrete} features

- What happens when features are \textit{continuous}? (real-world apps / latent GNN states)
  - \ldots the proof for injectivity of sum (hence GINs’ expressivity) \textbf{falls apart}

Graph 1:

- Node receiving the message
- Message of neighbour node #2

Graph 2:

Simple aggregators that can differentiate graph 1 and 2:

<table>
<thead>
<tr>
<th>Mean</th>
<th>Min</th>
<th>Max</th>
<th>STD</th>
</tr>
</thead>
<tbody>
<tr>
<td>VS</td>
<td>2</td>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>

- Aggregators that fail:
Which is best? **Neither.**

- There doesn’t seem to be a clear single “winner” aggregator here...

- In fact, we prove in the PNA paper (Corso, Cavalleri et al., NeurIPS’20) that there isn’t one!

**Theorem 1** (Number of aggregators needed). *In order to discriminate between multisets of size $n$ whose underlying set is $\mathbb{R}$, at least $n$ aggregators are needed.*

- The proof is (in my opinion) really cool! (relies on Borsuk–Ulam theorem)

- PNA proposes empirically powerful **combination** of aggregators for general-purpose GNNs:
Geometric Deep Learning
Remark on **geometric** deep learning

- We used the blueprint of *invariances* and *equivariances* to describe GNNs

- In fact, it is remarkably powerful! By combining an appropriate
  - **Local** and **equivariant** layer specified over *neighbourhoods*
  - Activation functions
  - (Potentially: **pooling** layers that coarsen the structure)
  - **Global** and **invariant** layer over the entire domain

  we recover many standard architectures (including CNNs and Transformers!)

- But also a more general class of **geometric** deep learning architectures
The “Four Gs” of geometric deep learning

Credits to Michael Bronstein
<table>
<thead>
<tr>
<th>Geometric Deep Learning Blueprint instances of interest</th>
<th>Architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Domain, Ω</strong></td>
<td><strong>CNNs</strong></td>
</tr>
<tr>
<td><strong>Gauges</strong></td>
<td>Spherical CNNs</td>
</tr>
<tr>
<td><strong>Graphs</strong></td>
<td>Gauge Equivariant Mesh CNNs</td>
</tr>
<tr>
<td><strong>Sets</strong></td>
<td>GNNs</td>
</tr>
<tr>
<td><strong>SO(3)</strong></td>
<td>Deep Sets</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Domain</th>
<th>Metric, g</th>
<th>Symmetry group, G</th>
<th>Architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grids</td>
<td>$L_\infty$</td>
<td>Translations</td>
<td>Spherical CNNs</td>
</tr>
<tr>
<td>Spheres</td>
<td>Great circle</td>
<td>3D rotations, SO(3)</td>
<td></td>
</tr>
<tr>
<td>SO(3)</td>
<td>Geodesic</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gauges</td>
<td>Geodesic</td>
<td>{Gauge transform.}</td>
<td></td>
</tr>
<tr>
<td>Graphs</td>
<td>Shortest path</td>
<td>Permutations, $\Sigma_n$</td>
<td></td>
</tr>
<tr>
<td>Sets</td>
<td>0</td>
<td>Permutations, $\Sigma_n$</td>
<td>Deep Sets</td>
</tr>
</tbody>
</table>

Credits to Taco Cohen
VII

Historical concepts
Where did GNNs come from?

- Early forms can be traced to the **early 1990s**, often involving DAG structures.
  - Labeling RAAM *(Sperduti, NeurIPS’94)*
  - Backpropagation through structure *(Goller & Küchler, ICNN’96)*
  - Adaptive structure processing *(Sperduti & Starita, TNN’97; Frasconi et al., TNN’98)*

- First proper treatment of **generic** graph structure processing happens in the 2000s:
  - The **GNN framework** *(Gori et al., IJCNN’05; Scarselli et al., TNN’08)*
  - The **NN4G framework** *(Micheli, TNN’09)*

- The GNN model of Gori, Scarselli *et al.* used primarily *recurrent*-style updates
  - Updated for modern best practices by **gated GNNs** *(Li et al., ICLR’16)*
Computational Chemistry
“Chemistry disrupts ML, not the other way around”

• Important and concurrent GNN development line came from computational chemistry
  ○ Very relevant to the area, as molecules are naturally modelled as graphs

• GNN-like models of *molecular property prediction* arise, also, in the 1990s
  ○ Examples include **ChemNet** (Kireev, CICS’95) and (Baskin *et al.*, CICS’97)

• “**Molecular Graph Networks**” (Merkwirth and Lengauer, CIM’05) already propose many elements commonly found in modern MPNNs

• This drive continued well into the 2010s:
  ○ GNNs for molecular fingerprinting (Duvenaud *et al.*, NeurIPS’15)
  ○ GNNs for quantum chemistry (Gilmer *et al.*, ICML’17)

• Lastly, recall (Stokes *et al.*, Cell’20): chemistry is to-this-day a leading outlet for GNNs!
Thank you!

Questions?

petarv@google.com | https://petar-v.com

With many thanks to Will Hamilton, Joan Bruna, Michael Bronstein and Taco Cohen