Graph Neural Networks (GNNs)

Oualid Missaoui

Agenda

- Introduction
- Machine Learning on Graphs
  - Graph properties
  - Adjacency matrix
  - Permutation equivariance
- Neural Message-Passing
  - Convolutional filters
  - Graph convolutional networks
  - Aggregation operators
  - Update operators
  - Node classification
  - Edge classification
  - Graph classification
- General Graph Networks
  - Graph attention networks
  - Edge embeddings
  - Graph embeddings
  - Over-smoothing
  - Regularization
  - Geometric deep learning
- Illustration
  - Synthetic data
    - Regression
    - Classification
  - Emerging Markets Return Predictions
- References

Why Stock Prediction Breaks the IID Comfort Zone

- **Most machine learning loves IID data**
  - Samples are assumed *independent and identically distributed*.
  - Standard theory, model selection, and cross-validation all lean on this.
- In practice, we got reasonably good at handling **multicollinearity**
  - Regularization (ridge, lasso), PCA, factor models, etc.
  - These tame *correlated features* **within** a sample.
- But in cross-sectional stock prediction, IID really fails:
  - Stocks are **strongly dependent**: sector, supply chain, index membership, ownership flows, common factors, etc.
  - Treating each stock-day as an IID sample ignores rich **relational structure** (AAPL is not "just another sample" unrelated to MSFT or NVDA).
- Result:
  - Models see a big, noisy table of returns & features.
  - Cross-sectional dependence is only weakly encoded via a few dummy variables (sector, region) or factors.
  - The **graph of relationships** between stocks is mostly thrown away.

GNNs: Bringing Graph Structure into Excess-Return Prediction

- Key idea: **relax the IID assumption across stocks** by making dependencies **first-class citizens**.
- Represent the equity universe as a **graph**:
  - **Nodes** = stocks.
  - **Edges** encode relationships:
    - Return/covariance/similarity network.
    - Sector / industry links.
    - Supply-chain, ownership, or fundamental similarity links.
- A Graph Neural Network layer:
  - Updates each stock's embedding using **its own features** *and* **messages from its neighbours**.
  - Learns *how much* to trust each neighbour (via learned weights or attention).
- Benefits for excess-return prediction:
  - Information about shocks & signals can **propagate along the graph** (sector moves, factor rotations, supply-chain contagion).
  - The model no longer treats stocks as IID; it exploits **structured dependence** that classical tabular models mostly ignore.
  - Still compatible with all your ML tools:
    - You can plug GNN embeddings into standard prediction heads (for alpha, risk, or regime classification).

Machine Learning on Graphs

- **Tasks on graph-structured data**
  - **Node prediction:** predict node properties (e.g. classify documents using hyperlink/citation graph).
  - **Edge prediction / completion:** predict missing or future edges (e.g. protein–protein interactions).
  - **Graph-level prediction:** predict a property of the *whole* graph (e.g. solubility of a molecule).
    - These tasks are often called **graph regression** or **graph classification**.
    - We typically assume graphs in the dataset are **IID draws** from some distribution.
- **Inductive vs transductive node prediction**
  - **Inductive example:** molecule solubility classification.
    Train on a set of labeled molecules, test on *new* molecules.
  - **Transductive example:** large social network where we know the whole graph but only some node labels.
    During training, we have access to the entire graph; goal is to predict labels for the remaining nodes → **semi-supervised learning**.
- **Graph representation learning & foundation models**
  - Beyond direct prediction, we can train deep models on graphs to learn **useful internal representations**.
  - Example: train a large model on many molecular graphs to build a **foundation model**, then fine-tune on small labeled datasets.
  - Graph neural networks (GNNs):
    - Define an **embedding vector** for each node (and optionally edges and the whole graph).
    - Initialize from observed node properties, then transform through multiple learned layers.
    - Analogy: like word embeddings/transformers, but operating on nodes in a graph rather than tokens in a sequence.

Graph properties

- We focus on **simple graphs**:
  - At most one edge between any pair of nodes.
  - Edges are **undirected**.
  - No self-loops (no edge from a node to itself).
- A graph is written as:

$$\mathcal{G} = (\mathcal{V}, \mathcal{E})$$

  - $\mathcal{V}$: set of **nodes / vertices**.
  - $\mathcal{E}$: set of **edges / links**.
- Nodes are indexed:

$$n = 1, \ldots, N$$

  - An edge from node n to node m is written (n, m).
- **Neighbours**:
  - If two nodes are linked by an edge, they are **neighbours**.
  - Neighbourhood of node n: $\mathcal{N}(n)$.
- **Node features**:
  - For each node n, we have a D-dimensional feature vector $\mathbf{x}_n \in \mathbb{R}^D$.
  - Stack them into a data matrix:

$$\mathbf{X} \in \mathbb{R}^{N \times D}, \quad \text{row } n = \mathbf{x}_n^\top.$$

  - There may also be **edge features**, but we first focus on node features only.

Adjacency matrix

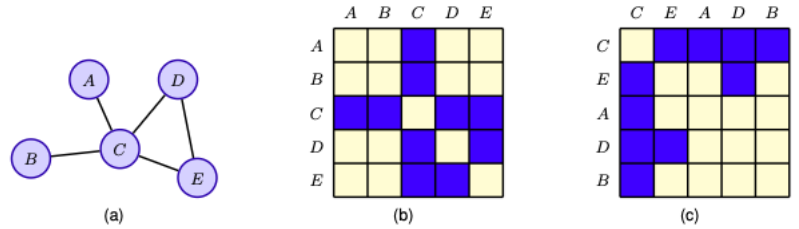


Figure 13.2 An example of an adjacency matrix showing (a) an example of a graph with five nodes, (b) the associated adjacency matrix for a particular choice of node order, and (c) the adjacency matrix corresponding to a different choice for the node order.

- **Adjacency matrix $\mathbf{A}$** is a convenient way to encode edges:
  - For N nodes, $\mathbf{A} \in \mathbb{R}^{N \times N}$.
  - $A_{nm} = 1$ if there is an edge from node n to node m, else 0.
  - For **undirected** graphs: $\mathbf{A}$ is symmetric, $A_{nm} = A_{mn}$.
- Since $\mathbf{A}$ defines the graph structure, one might try:
  - **Flattening $\mathbf{A}$** (e.g., concatenating columns) and feeding it directly to a neural network.
- **Problem: $\mathbf{A}$** depends on an **arbitrary node ordering**.
  - Re-labeling nodes (permuting indices) changes $\mathbf{A}$, even though the underlying graph and its properties (e.g., molecule solubility) are unchanged.
  - Learning permutation invariance via data augmentation is impractical: number of permutations grows **factorially** with N.
- **Conclusion:**
  - Permutation invariance should be treated as an **inductive bias** and built into the network architecture (e.g., via GNNs), rather than learned purely from data.

Permutation Equivariance

- **Permutation of node labels** can be written with a **permutation matrix** $\mathbf{P}$:
  - $\mathbf{P} \in \mathbb{R}^{N \times N}$ has exactly one 1 in each row and each column.
  - A 1 in position (n, m) means: node n is relabelled as node m.
- Build $\mathbf{P}$ using **standard basis vectors** $\mathbf{u}_n$:
  - $\mathbf{u}_n$: column vector with 1 in position n, 0 elsewhere.
  - Identity matrix:

$$\mathbf{I} = \begin{pmatrix} \mathbf{u}_1^\top \\ \mathbf{u}_2^\top \\ \vdots \\ \mathbf{u}_N^\top \end{pmatrix}$$

- Let $\pi(\cdot)$ be a permutation of node indices.
  The permutation matrix is

$$\mathbf{P} = \begin{pmatrix} \mathbf{u}_{\pi(1)}^\top \\ \mathbf{u}_{\pi(2)}^\top \\ \vdots \\ \mathbf{u}_{\pi(N)}^\top \end{pmatrix}.$$

- Effect on data:
  - Node-feature matrix $\mathbf{X}$: permuted version

$$\tilde{\mathbf{X}} = \mathbf{P}\mathbf{X}.$$

  - Adjacency matrix $\mathbf{A}$: rows & columns permuted

$$\tilde{\mathbf{A}} = \mathbf{P}\mathbf{A}\mathbf{P}^\top.$$

Invariance vs equivariance to node relabelling

- When we use graphs in neural networks, we must ensure predictions do **not depend** on arbitrary node ordering.

Graph-level tasks — **invariance**

- For graph-level output $y(\mathbf{X}, \mathbf{A})$ (scalar or vector, but *one per graph*), we require:

$$y(\tilde{\mathbf{X}}, \tilde{\mathbf{A}}) = y(\mathbf{X}, \mathbf{A}),$$

  i.e. **invariant** to node permutation.

Node-level tasks — **equivariance**

- For node-level outputs $\mathbf{y}(\mathbf{X}, \mathbf{A})$ (one element per node), we want predictions to be **relabeled in the same way**:

$$\mathbf{y}(\tilde{\mathbf{X}}, \tilde{\mathbf{A}}) = \mathbf{P}\,\mathbf{y}(\mathbf{X}, \mathbf{A}).$$

- This is **permutation equivariance**:
  - Reordering the nodes reorders the outputs, but does not change their values relative to the nodes they belong to.

Neural Message Passing

- Need **invariance / equivariance** under node label permutations for GNNs.
- Keep the notion of a **layer** as a computational transformation applied repeatedly.
- If each layer is **permutation equivariant**, then stacking layers:
  - preserves equivariance
  - lets each layer use the graph structure.
- **Node-level prediction:** entire network should be equivariant.
- **Graph-level prediction:** add a final **permutation-invariant** layer.
- Each layer should be:
  - a flexible nonlinear function
  - differentiable w.r.t. its parameters → trainable with SGD & autodiff.
- Graphs have **variable sizes** (e.g., molecules with different numbers of atoms), so fixed-length representations are unsuitable.
- Network must handle **variable-length inputs** (as with transformers).
- Some graphs are **very large** (e.g., social networks), so models must scale well.
- **Parameter sharing**:
  - enforces invariance/equivariance in the architecture
  - helps scaling to large graphs.

Convolutional filters

- Use **CNNs** as inspiration for GNN layers.
- View an **image as a graph**:
  - Nodes = pixels
  - Edges = adjacent pixels (horizontal, vertical, and diagonal).
- In a convolutional network:
  - Each pixel at layer $l+1$ computes a function of pixel states in layer l.
  - This local function is called a **filter** (e.g. a 3×3 filter).

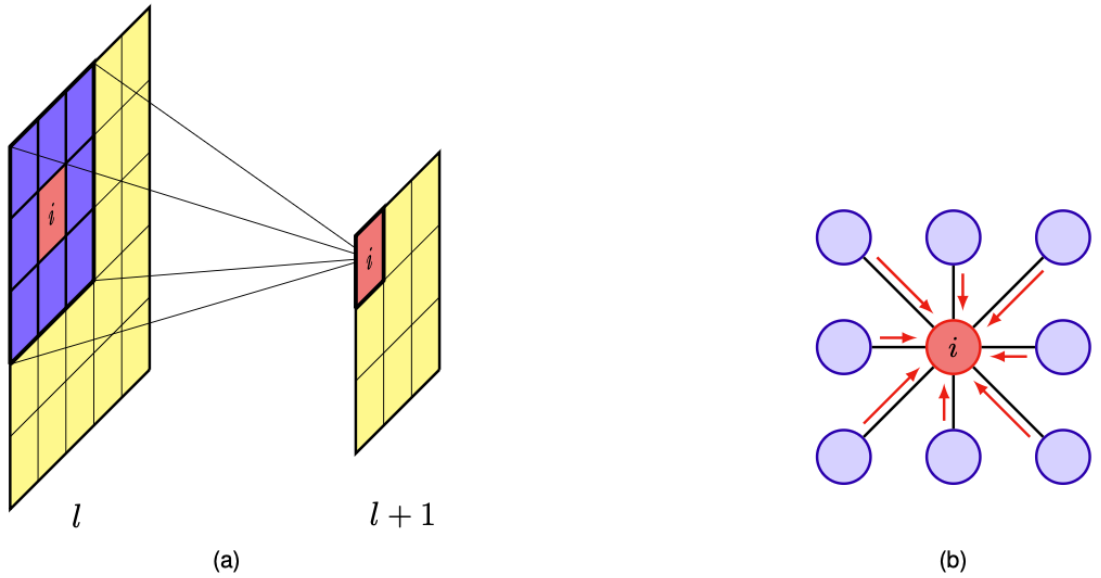


Figure 13.3 A convolutional filter for images can be represented as a graph-structured computation. (a) A filter computed by node i in layer $l+1$ of a deep convolutional network is a function of the activation values in layer l over a local patch of pixels. (b) The same computation structure expressed as a graph showing 'messages' flowing into node i from its neighbours.

- For a single filter at pixel i in layer $l+1$:

$$z_i^{(l+1)} = f\left(\sum_j w_j z_j^{(l)} + b\right)$$

- $f(\cdot)$: differentiable nonlinearity (e.g. ReLU).
- Sum over j: all 9 pixels in the 3×3 patch in layer l.
- Same function is applied at every spatial location:
  - **Weight sharing**: w_j and b are shared across all patches.
- But the above equation is **not equivariant** to reordering nodes in layer l:
  - The weight vector (w_j) changes under permutation of its elements.

Illustration of non-equivariance of convolutional filters

- Convolution update for one pixel i:

$$z_i^{(l+1)} = f\left(\sum_j w_j z_j^{(l)} + b\right)$$

- Toy example with **3 neighbors** instead of 9:

$$F(z_1, z_2, z_3) = w_1 z_1 + w_2 z_2 + w_3 z_3$$

- Now **permute the neighbors** by swapping 1 and 2:
  - Old ordering: (z_1, z_2, z_3)
  - New ordering: (z_2, z_1, z_3)
- Apply the *same* filter to the reordered inputs:

$$F(z_2, z_1, z_3) = w_1 z_2 + w_2 z_1 + w_3 z_3$$

- For permutation **invariance** (and hence equivariance for this scalar output), we would need

$$w_1 z_1 + w_2 z_2 + w_3 z_3 = w_1 z_2 + w_2 z_1 + w_3 z_3 \quad \text{for all } z_1, z_2, z_3,$$

  which only holds if $w_1 = w_2$.

- More generally, for *any* permutation P, we'd need $w = P^\top w$, forcing **all w_j to be identical**.
- But in a normal conv filter we *want* different weights at different relative positions (top-left, center, bottom-right, ...), so $w_1, w_2, \ldots$ are **not** all equal.
- Therefore the conv update depends on the **ordering of neighbors** $\rightarrow$ **not permutation equivariant** as a graph operation.

Making the filter equivariant

- View the filter itself as a **graph** and separate node i from its neighbours:
  - Node i: the center pixel.
  - Neighbours $\mathcal{N}(i)$: the 8 surrounding pixels.
- Impose **shared neighbour weights**:
  - Single parameter w_{neigh} for all neighbours.
  - Separate parameter w_{self} for node i.
- New update rule:

$$z_i^{(l+1)} = f\left(w_{\text{neigh}} \sum_{j \in \mathcal{N}(i)} z_j^{(l)} + w_{\text{self}} z_i^{(l)} + b \right)$$

- Node i still has its own weight w_{self}.

Message-passing interpretation

- The New update rule:

$$z_i^{(l+1)} = f\left(w_{\text{neigh}} \sum_{j \in \mathcal{N}(i)} z_j^{(l)} + w_{\text{self}} z_i^{(l)} + b\right)$$

  can be seen as **message passing**:
  - Neighbouring nodes send messages = their activations $z_j^{(l)}$ to node i.
  - Messages are **aggregated** by a simple sum $\sum_{j \in \mathcal{N}(i)} z_j^{(l)}$.
  - Aggregated neighbour info + self info $z_i^{(l)}$ are transformed by nonlinearity f.
- The summation over neighbours is **permutation-invariant**:
  - Reordering neighbour labels does not change the sum.
- Apply the above equation **synchronously to every node**:
  - If nodes are permuted, outputs are permuted the same way.
  - $\rightarrow$ The operation is **equivariant to node permutations**.
- This equivariance relies on **shared parameters**:
  - $w_{\text{neigh}}, w_{\text{self}}, b$ are shared across all nodes.

Graph Convolutional Networks (GCNs)

- Use the **convolution example** as a template for deep nets on **graph-structured data**.
- Goal: define a **flexible, nonlinear transformation** of node embeddings:
  - Differentiable w.r.t. weights and biases.
  - Maps variables in layer l to variables in layer $l+1$.
- For each node n and layer l, introduce a **D-dimensional embedding**:

$$\mathbf{h}_n^{(l)} \in \mathbb{R}^D, \quad n = 1, \ldots, N,\ l = 1, \ldots, L.$$

- Transformation based on

$$z_i^{(l+1)} = f\left(w_{\text{neigh}} \sum_{j \in \mathcal{N}(i)} z_j^{(l)} + w_{\text{self}}\, z_i^{(l)} + b \right)$$

has two conceptual stages:

1. **Gather / combine** information from neighbours.
2. **Update** the node using its current embedding and the aggregated messages.

Graph Convolutional Networks (GCNs)

Aggregation step

- For a node n, first aggregate messages from its neighbours:

$$\mathbf{z}_n^{(l)} = \text{Aggregate}\big(\{\mathbf{h}_m^{(l)} : m \in \mathcal{N}(n)\}\big).$$

- Requirements for **Aggregate**:
  - Well-defined for a **variable number of neighbours**.
  - **Permutation invariant** to the ordering of neighbours.
  - Can include **learnable parameters**, as long as it is differentiable w.r.t. those parameters to allow gradient-based training.
- Examples: sum, mean, max, attention-style weighted sum, etc.

Update step & message-passing view

- Use another operation to update the embedding at node n:

$$\mathbf{h}_n^{(l+1)} = \text{Update}\big(\mathbf{h}_n^{(l)}, \mathbf{z}_n^{(l)}\big).$$

- **Update**:
  - Differentiable function with learnable parameters.
  - Combines **local state** $\mathbf{h}_n^{(l)}$ and **aggregated messages** $\mathbf{z}_n^{(l)}$.
- Applying **Aggregate** $\rightarrow$ **Update** in parallel to every node defines **one layer** of the network.
- Node embeddings are typically initialized as:

$$\mathbf{h}_n^{(0)} = \mathbf{x}_n$$

  where $\mathbf{x}_n$ are observed node features.
- This framework is called a **message-passing neural network (MPNN)**

Algorithm 13.1: Simple message-passing neural network

Input: Undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$

Initial node embeddings $\{\mathbf{h}_n^{(0)} = \mathbf{x}_n\}$

Aggregate$(\cdot)$ function

Update$(\cdot, \cdot)$ function

Output: Final node embeddings $\{\mathbf{h}_n^{(L)}\}$

```
// Iterative message-passing
```

for $l \in \{0, \ldots, L-1\}$ **do**

$$\mathbf{z}_n^{(l)} \leftarrow \text{Aggregate}\left(\left\{\mathbf{h}_m^{(l)} : m \in \mathcal{N}(n)\right\}\right)$$

$$\mathbf{h}_n^{(l+1)} \leftarrow \text{Update}\left(\mathbf{h}_n^{(l)}, \mathbf{z}_n^{(l)}\right)$$

end for

return $\{\mathbf{h}_n^{(L)}\}$

Aggregation operators overview

- **Aggregate** must:
  - Depend only on the **set** of inputs, not their order → **permutation invariant**.
  - Be well defined for a **variable number of neighbours**.
  - Be **differentiable** (if it has learnable parameters) for gradient-based training.
- Many choices are possible. We start from the simplest: **summation**.

Simple sum aggregation

- Sum aggregation:

$$\text{Aggregate}\big(\{\mathbf{h}_m^{(l)} : m \in \mathcal{N}(n)\}\big) = \sum_{m \in \mathcal{N}(n)} \mathbf{h}_m^{(l)}.$$

- Properties:
  - Clearly **independent of neighbour ordering**.
  - Works for any number of neighbours.
  - Has **no learnable parameters**.
- Drawback:
  - Nodes with **many neighbours** can dominate nodes with few neighbours, especially when degrees vary by orders of magnitude (e.g. social networks).

Mean and degree-normalized aggregations

- **Mean aggregation** normalizes by neighbourhood size:

$$\text{Aggregate}\big(\{\mathbf{h}_m^{(l)} : m \in \mathcal{N}(n)\}\big) = \frac{1}{|\mathcal{N}(n)|} \sum_{m \in \mathcal{N}(n)} \mathbf{h}_m^{(l)}.$$

- Reduces numerical issues from high-degree nodes.
- But discards degree information and is theoretically weaker than sum.
- **GCN-style degree normalization**:

$$\text{Aggregate}\big(\{\mathbf{h}_m^{(l)} : m \in \mathcal{N}(n)\}\big) = \sum_{m \in \mathcal{N}(n)} \frac{\mathbf{h}_m^{(l)}}{\sqrt{|\mathcal{N}(n)|\,|\mathcal{N}(m)|}}.$$

- Accounts for the degrees of **both** node n and neighbour m.
- Still permutation invariant and works for variable neighbourhood sizes.

Other simple aggregators & receptive field

- We can also take **element-wise max or min** over neighbour embeddings:
  - Still permutation invariant and defined for variable neighbourhood sizes.
- In all cases, each node's new embedding is computed from:
  - Its neighbours' embeddings in the **previous layer**.
- This defines a **receptive field** analogous to CNNs:
  - As we stack layers, information propagates further through the graph.
  - In deep layers, a node's representation can depend on a large fraction of the graph.
  - For very large, sparse graphs this may require many layers → some architectures add "super-nodes" or shortcuts for faster propagation.

Learnable aggregation via MLPs (Deep Sets)

- Let
  - Node embedding at layer l:

$$\mathbf{h}_m^{(l)} \in \mathbb{R}^{D_{\text{in}}}$$

  - First MLP:

$$\text{MLP}_\phi : \mathbb{R}^{D_{\text{in}}} \to \mathbb{R}^{D_{\text{agg}}}$$

  - Second MLP:

$$\text{MLP}_\theta : \mathbb{R}^{D_{\text{agg}}} \to \mathbb{R}^{D_{\text{out}}}$$

- Learnable, permutation-invariant aggregation:

$$\text{Aggregate}\left(\{\mathbf{h}_m^{(l)} : m \in \mathcal{N}(n)\}\right) = \text{MLP}_\theta\left(\sum_{m \in \mathcal{N}(n)} \text{MLP}_\phi\left(\mathbf{h}_m^{(l)}\right)\right).$$

  - Each neighbour:

$$\mathbf{h}_m^{(l)} \in \mathbb{R}^{D_{\text{in}}} \xrightarrow{\text{MLP}_\phi} \tilde{\mathbf{h}}_m \in \mathbb{R}^{D_{\text{agg}}}$$

  - Sum over neighbours:

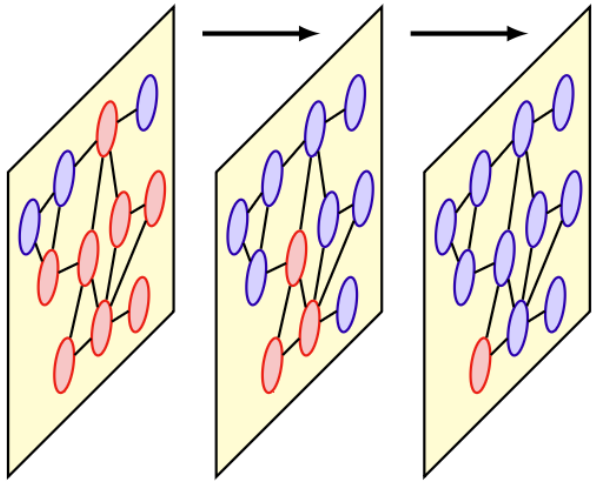
$$\sum_{m \in \mathcal{N}(n)} \tilde{\mathbf{h}}_m \in \mathbb{R}^{D_{\text{agg}}}$$

  - Final transform:

$$\text{MLP}_\theta\left(\sum_{m \in \mathcal{N}(n)} \tilde{\mathbf{h}}_m\right) \in \mathbb{R}^{D_{\text{out}}}$$

- MLP_ϕ and MLP_θ share parameters across the layer.
- Still **permutation invariant** because the inner operation is a sum.
- With suitable D_{agg} and D_{out}, this architecture is a **universal approximator** for permutation-invariant set functions.

Figure 13.4 Schematic illustration of information flow through successive layers of a graph neural network. In the third layer a single node is highlighted in red. It receives information from its two neighbours in the previous layer and those in turn receive information from their neighbours in the first layer. As with convolutional neural networks for images, we see that the effective receptive field, corresponding to the number of nodes shown in red, grows with the number of processing layers.



Update operators – basic form

- After choosing an **Aggregate** operator, we define an **Update** operator.
- Simple choice :

$$\mathrm{Update}\left(\mathbf{h}_n^{(l)}, \mathbf{z}_n^{(l)}\right) = f\left(\mathbf{W}_{\mathrm{self}}\mathbf{h}_n^{(l)} + \mathbf{W}_{\mathrm{neigh}}\mathbf{z}_n^{(l)} + \mathbf{b}\right)$$

- Here:
  - $\mathbf{h}_n^{(l)}$: node n embedding at layer l.
  - $\mathbf{z}_n^{(l)}$: aggregated neighbour message (from Aggregate op).
  - $\mathbf{W}_{\mathrm{self}}, \mathbf{W}_{\mathrm{neigh}}$: learnable weight matrices.
  - $\mathbf{b}$: bias.
  - $f(\cdot)$: nonlinearity (e.g. ReLU) applied element-wise.

Special case: shared weights & sum aggregation

- If Aggregate is the **simple sum**:

$$\mathbf{z}_n^{(l)} = \sum_{m \in \mathcal{N}(n)} \mathbf{h}_m^{(l)}$$

- And if we **share** the same weight matrix for self and neighbours, $\mathbf{W}_{\text{self}} = \mathbf{W}_{\text{neigh}}$, the update becomes

$$\mathbf{h}_n^{(l+1)} = f\left(\mathbf{W}_{\text{neigh}} \sum_{m \in \mathcal{N}(n) \cup \{n\}} \mathbf{h}_m^{(l)} + \mathbf{b}\right).$$

- Interpretation:
  - Sum messages from neighbours **and self**.
  - Apply a **shared linear transform + nonlinearity**.
  - This is a very simple message-passing GNN layer.

Initializing node embeddings

- Typical initialization:

$$\mathbf{h}_n^{(0)} = \mathbf{x}_n$$

  where $\mathbf{x}_n$ are node features.

- If we want an internal dimension different from $\dim(\mathbf{x}_n)$:
  - **Pad with zeros** to increase dimensionality, or
  - Apply a **learnable linear map** to project $\mathbf{x}_n$ to desired size.
- If there are **no node features**, possible initialization:
  - Use a **one-hot encoding of node degree** (number of neighbours), etc.

Matrix form of a GNN & equivariance

- Stack node embeddings into a matrix:

$$\mathbf{H}^{(l)} \in \mathbb{R}^{N \times D_l}, \quad \text{row } n = (\mathbf{h}_n^{(l)})^\top.$$

- Let $\mathbf{X} = \mathbf{H}^{(0)}$ be the data matrix and $\mathbf{A}$ the adjacency matrix.

- A GNN with L layers is written as:

$$\begin{aligned}
\mathbf{H}^{(1)} &= \mathbf{F}\big(\mathbf{X}, \mathbf{A}, \mathbf{W}^{(1)}\big), \\
\mathbf{H}^{(2)} &= \mathbf{F}\big(\mathbf{H}^{(1)}, \mathbf{A}, \mathbf{W}^{(2)}\big), \\
&\vdots \\
\mathbf{H}^{(L)} &= \mathbf{F}\big(\mathbf{H}^{(L-1)}, \mathbf{A}, \mathbf{W}^{(L)}\big),
\end{aligned}$$

where $\mathbf{W}^{(l)}$ collects all weights/biases at layer l.

- Under a **node permutation** represented by a permutation matrix $\mathbf{P}$:

$$\mathbf{P}\mathbf{H}^{(l)} = \mathbf{F}\big(\mathbf{P}\mathbf{H}^{(l-1)}, \mathbf{P}\mathbf{A}\mathbf{P}^\top, \mathbf{W}^{(l)}\big).$$

- Hence each layer, and therefore the **whole network**, is **equivariant to node permutations**: permuting node labels permutes the outputs in the same way.

Node classification – readout layer

- A GNN can be seen as a stack of layers transforming node embeddings $\{\mathbf{h}_n^{(l)}\}$ into $\{\mathbf{h}_n^{(l+1)}\}$.
- After the final layer (L), we have node embeddings $\{\mathbf{h}_n^{(L)}\}$ and want **predictions** for each node.
- For node classification into C classes, we add an **output / readout layer**:
  - For node n, class i:

$$y_{ni} = \frac{\exp(\mathbf{w}_i^\top \mathbf{h}_n^{(L)})}{\sum_j \exp(\mathbf{w}_j^\top \mathbf{h}_n^{(L)})}, \quad i = 1, \ldots, C$$

- Here:
  - $\mathbf{w}_i$: learnable weight vector for class i.
  - $\mathbf{y}_n = (y_{n1}, \ldots, y_{nC})$: softmax probabilities for node n.

Loss function & permutation equivariance

- Let t_{ni} be the **target one-hot label** for node n, class i.
- Cross-entropy loss over training nodes:

$$\mathcal{L} = -\sum_{n \in \mathcal{V}_{\text{train}}} \sum_{i=1}^{C} t_{ni} \log y_{ni}.$$

- Because the same set of weight vectors $\{\mathbf{w}_i\}$ is shared across all nodes:
  - The outputs $\mathbf{y}_n$ are **equivariant** to permutations of node order.
  - The loss $\mathcal{L}$ is therefore **invariant** to node ordering.
- For regression on continuous targets:
  - Replace the softmax layer with a suitable linear/nonlinear head.
  - Use, e.g., a **sum-of-squares** loss instead of cross-entropy.

Train / test node splits: inductive vs transductive

- The sum over n in

$$\mathcal{L} = -\sum_{n \in \mathcal{V}_{\text{train}}} \sum_{i=1}^{C} t_{ni} \log y_{ni}.$$

runs over a subset $\mathcal{V}_{\text{train}}$ of nodes used for training.

- We distinguish three node sets:

1. **Training nodes $\mathcal{V}_{\text{train}}$**
   - Labeled, participate in **message passing**.
   - Used directly in the loss during training.
2. **Transductive test nodes $\mathcal{V}_{\text{trans}}$**
   - Unlabeled during training, not used in the loss.
   - Still included in the graph and in message passing for both train and test.
   - Their labels are predicted at inference time.
   - Presence of such nodes $\rightarrow$ **transductive learning** (a form of *semi-supervised learning*).
3. **Inductive test nodes $\mathcal{V}_{\text{induct}}$**
   - Not present (with their edges) during training.
   - Only appear at inference time, where they join message passing and are labeled.
   - No transductive nodes $\rightarrow$ **inductive learning** (a form of *supervised learning*).

Learnable parameters & latent dimensions in GNNs

- Node features:

$$\mathbf{X} \in \mathbb{R}^{N \times D_{\text{in}}}, \quad \text{row } n = \mathbf{h}_n^{(0)}.$$

- A typical GNN layer (GCN-style) is:

$$\mathbf{H}^{(l+1)} = \sigma\big(\hat{\mathbf{A}}\,\mathbf{H}^{(l)}\mathbf{W}^{(l)} + \mathbf{b}^{(l)}\big),$$

  where
  - $\hat{\mathbf{A}}$: (normalized) adjacency, built from the graph.
  - $\mathbf{H}^{(l)} \in \mathbb{R}^{N \times D_l}$: node embeddings.
  - **Parameters**:
    $\mathbf{W}^{(l)} \in \mathbb{R}^{D_l \times D_{l+1}}$,
    $\mathbf{b}^{(l)} \in \mathbb{R}^{D_{l+1}}$.
- **Key point:**
  - Number of nodes N **does not appear** in the parameter shapes.
  - Latent dimension = D_l (width of embeddings), chosen by us, not tied to N.
  - Parameters are **shared across all nodes**; adjacency $\hat{\mathbf{A}}$ decides *where* each parameter is applied.

Loss functions for edge classification / regression

- After L layers we have node embeddings $\mathbf{h}_n^{(L)}$.
- Build an **edge embedding** for an edge (n, m), e.g.

$$\mathbf{e}_{nm}^{(L)} = g\big(\mathbf{h}_n^{(L)}, \mathbf{h}_m^{(L)}\big)$$

where g is a small MLP (shared for all edges).

Edge classification (e.g., link exists / type of relation)

- Logit / probability:

$$p_{nm} = \sigma\big(\mathbf{u}^\top \mathbf{e}_{nm}^{(L)}\big) \quad \text{or} \quad \mathbf{p}_{nm} = \mathrm{softmax}\big(\mathrm{MLP}_{\text{edge}}(\mathbf{e}_{nm}^{(L)})\big).$$

- Cross-entropy loss over a **training set of edges** $\mathcal{E}_{\text{train}}$ (plus sampled non-edges for negatives):

$$\mathcal{L} = -\sum_{(n,m)\in\mathcal{E}_{\text{train}}} \sum_c t_{nm,c} \log p_{nm,c}.$$

Edge regression (e.g., predict correlation, spread, flow)

- Predicted value:

$$\hat{y}_{nm} = f_{\text{edge}}(\mathbf{e}_{nm}^{(L)}).$$

- MSE loss:

$$\mathcal{L} = \sum_{(n,m)\in\mathcal{E}_{\text{train}}} \big(y_{nm} - \hat{y}_{nm}\big)^2.$$

- In all cases, **which pairs** (n, m) exist and are trained on is determined by the **graph structure / adjacency matrix**.

Forward / inference: how the graph drives computation

Given:

- Trained parameters $\{\mathbf{W}^{(l)}, \mathbf{b}^{(l)}, \ldots\}$,
- Node features $\mathbf{X}$,
- Adjacency matrix $\mathbf{A}$ (or its normalized version $\hat{\mathbf{A}}$).

Forward pass:

1. **Initialize node embeddings**

$$\mathbf{H}^{(0)} = \mathbf{X}.$$

2. **Layer-by-layer message passing**

   - For each layer $l = 0, \ldots, L-1$:

$$\mathbf{H}^{(l+1)} = \sigma\big(\hat{\mathbf{A}}\,\mathbf{H}^{(l)}\mathbf{W}^{(l)} + \mathbf{b}^{(l)}\big) \quad \text{(or any Aggregate/Update variant)}.$$

   - Here, $\hat{\mathbf{A}}$ encodes **who talks to whom**; it routes messages along edges and mixes neighbour information.

3. **Prediction head**

   - **Node task:** apply MLP/linear head to each row $\mathbf{h}_n^{(L)}$.
   - **Edge task:** build $\mathbf{e}_{nm}^{(L)}$ from $\mathbf{h}_n^{(L)}, \mathbf{h}_m^{(L)}$ and apply an edge head.
   - **Graph task:** aggregate all $\mathbf{h}_n^{(L)}$ (sum/mean/attention) and apply a graph head.

Dependency on adjacency:

- Change $\mathbf{A} \to$ you change $\hat{\mathbf{A}} \to$ you change the **neighbour sets** $\mathcal{N}(n)$ and the aggregation pattern at every layer.
- Parameters are fixed at inference; **all structural dependence flows through the adjacency matrix**:
  - It determines the receptive field,
  - The paths along which information and shocks propagate,
  - And thus the final node/edge/graph predictions.

Graph attention networks (GATs) - Idea

- **Attention** is powerful in transformers; we can reuse it in GNNs.
- Goal: learn **data-dependent weights** for messages from neighbours.
- For node n at layer l, with neighbour set $\mathcal{N}(n)$:

$$\mathbf{z}_n^{(l)} = \mathrm{Aggregate}\big(\{\mathbf{h}_m^{(l)} : m \in \mathcal{N}(n)\}\big) = \sum_{m \in \mathcal{N}(n)} A_{nm}\, \mathbf{h}_m^{(l)}.$$

- Attention coefficients satisfy:

$$A_{nm} \geq 0, \qquad \sum_{m \in \mathcal{N}(n)} A_{nm} = 1.$$

- This defines a **Graph Attention Network (GAT)**:
  - Some neighbours are weighted more than others.
  - Importance is **learned from data**.

Computing attention coefficients

- Typically use a **softmax** over neighbours.
- Let node embeddings be $\mathbf{h}_n^{(l)}, \mathbf{h}_m^{(l)} \in \mathbb{R}^D$.

Bilinear attention

$$A_{nm} = \frac{\exp\left(\mathbf{h}_n^\top \mathbf{W} \mathbf{h}_m\right)}{\sum_{m' \in \mathcal{N}(n)} \exp\left(\mathbf{h}_n^\top \mathbf{W} \mathbf{h}_{m'}\right)},$$

- $\mathbf{W} \in \mathbb{R}^{D \times D}$ is a learnable matrix.

MLP-based attention

$$A_{nm} = \frac{\exp\left(\mathrm{MLP}(\mathbf{h}_n, \mathbf{h}_m)\right)}{\sum_{m' \in \mathcal{N}(n)} \exp\left(\mathrm{MLP}(\mathbf{h}_n, \mathbf{h}_{m'})\right)}.$$

- **MLP**: neural network with a **single scalar output**.
- MLP is **shared** across all edges:
  - Ensures the aggregation is **permutation equivariant** under node reordering.

Multi-head graph attention

- Extend GAT with **H attention heads**:
  - For each head $h = 1, \ldots, H$, compute its own attention coefficients $A_{nm}^{(h)}$ using one of the mechanisms above (with separate parameters).
- For node n, each head produces an aggregated vector $\mathbf{z}_n^{(l,h)}$.
- Combine heads in the aggregation step:
  - **Averaging**:

$$\mathbf{z}_n^{(l)} = \frac{1}{H} \sum_{h=1}^{H} \mathbf{z}_n^{(l,h)}.$$

  - or **Concatenation + linear projection**:

$$\mathbf{z}_n^{(l)} = \mathbf{W}_{\text{proj}} \left[\mathbf{z}_n^{(l,1)} \parallel \ldots \parallel \mathbf{z}_n^{(l,H)} \right].$$

- For a **fully connected graph**, a multi-head GAT becomes a standard **transformer layer** applied to the set of node embeddings.

Edge Embeddings in Graph Neural Networks

- So far, GNNs used **embeddings on nodes** only:
  - $\mathbf{h}_n^{(l)}$ = embedding of node n at layer l.
- Many graphs also have **data on edges**:
  - We may observe features for relationships (weights, distances, trade flows, $\cdots$).
- Even if there are **no observed edge features**, we can still:
  - Maintain **hidden edge variables**.
  - Update them through layers so they contribute to the learned representation.
- Introduce **edge embeddings**:

$$\mathbf{e}_{nm}^{(l)} \quad \text{for edge } (n, m) \text{ at layer } l.$$

General message passing with edge embeddings

1. **Edge update**

   - Each edge embedding uses its previous state and the incident nodes:

$$\mathbf{e}_{nm}^{(l+1)} = \mathrm{Update}_{\mathrm{edge}}\left(\mathbf{e}_{nm}^{(l)}, \mathbf{h}_{n}^{(l)}, \mathbf{h}_{m}^{(l)}\right).$$

2. **Node aggregation from updated edges**

   - Node n aggregates messages from its incident edges:

$$\mathbf{z}_{n}^{(l+1)} = \mathrm{Aggregate}_{\mathrm{node}}\left(\{\mathbf{e}_{nm}^{(l+1)} : m \in \mathcal{N}(n)\}\right).$$

3. **Node update**

   - Node embedding is then updated using its previous state and aggregate:

$$\mathbf{h}_{n}^{(l+1)} = \mathrm{Update}_{\mathrm{node}}\left(\mathbf{h}_{n}^{(l)}, \mathbf{z}_{n}^{(l+1)}\right).$$

- After L layers, **edge embeddings** $\mathbf{e}_{nm}^{(L)}$ can be:
  - Used directly for **edge-level prediction tasks** (e.g., link classification, rating a relationship, predicting flows).

Graph embeddings – adding a global context

- Beyond **node** and **edge** embeddings, we can learn a **graph-level embedding**:

$$\mathbf{g}^{(l)} \in \mathbb{R}^{D_g}$$

- $\mathbf{g}^{(l)}$ summarizes information about the **entire graph** at layer l:
  - Global properties (market regime, sector-wide shock, volatility state, $\cdots$).
  - Shared context that can influence how nodes and edges are updated.
- With node, edge, and graph embeddings, we can define a **very general message-passing framework** for graph-structured applications.

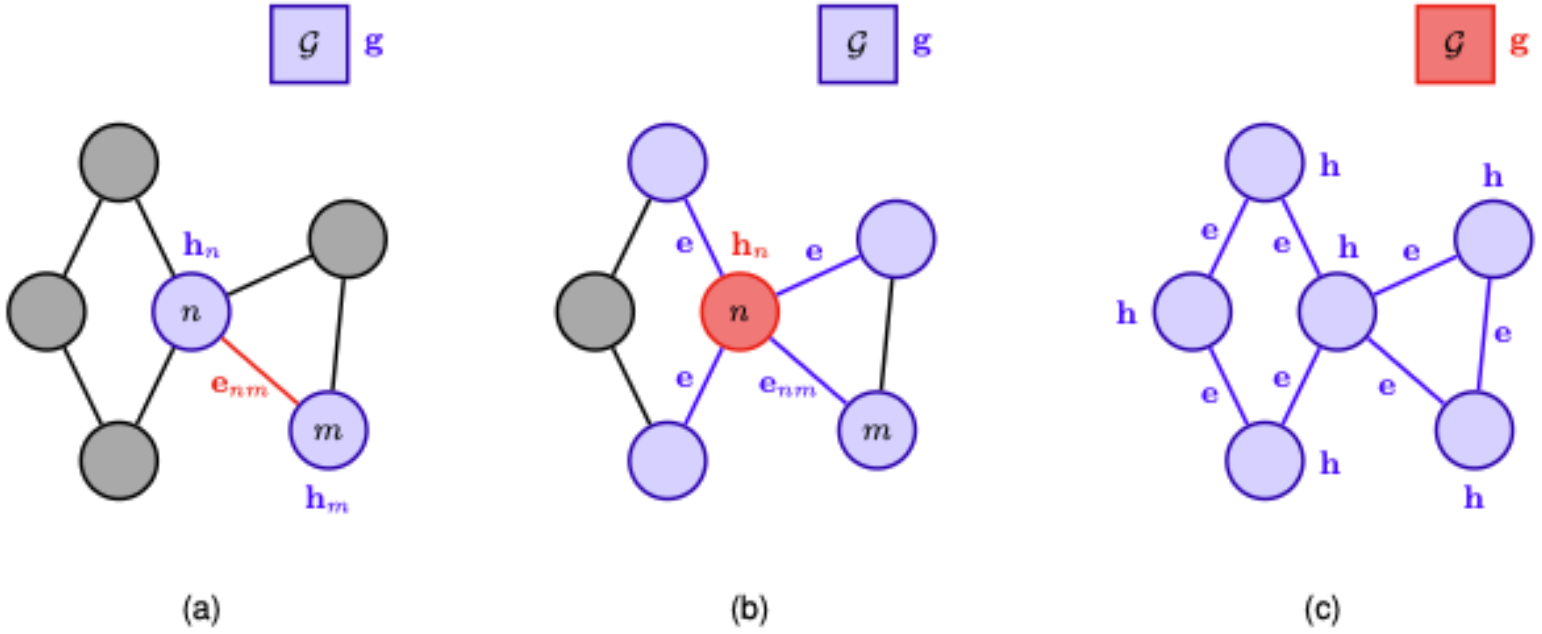


Figure 13.5 Illustration of the general graph message-passing updates defined by (13.32) to (13.35), showing (a) edge updates, (b) node updates, and (c) global graph updates. In each case the variable being updated is shown in red and the variables that contribute to that update are those shown in red and blue.

General message passing with node, edge, and graph embeddings

1. **Edge update** (use old edge, its incident nodes, and global state):

$$\mathbf{e}_{nm}^{(l+1)} = \text{Update}_{\text{edge}}\left(\mathbf{e}_{nm}^{(l)}, \mathbf{h}_n^{(l)}, \mathbf{h}_m^{(l)}, \mathbf{g}^{(l)}\right).$$

2. **Node aggregation from updated edges**:

$$\mathbf{z}_n^{(l+1)} = \text{Aggregate}_{\text{node}}\left(\{\mathbf{e}_{nm}^{(l+1)} : m \in \mathcal{N}(n)\}\right).$$

3. **Node update** (combine node, aggregated edges, and global state):

$$\mathbf{h}_n^{(l+1)} = \text{Update}_{\text{node}}\left(\mathbf{h}_n^{(l)}, \mathbf{z}_n^{(l+1)}, \mathbf{g}^{(l)}\right).$$

4. **Graph update** (update global embedding using all nodes & edges):

$$\mathbf{g}^{(l+1)} = \text{Update}_{\text{graph}}\left(\mathbf{g}^{(l)}, \{\mathbf{h}_n^{(l+1)} : n \in \mathcal{V}\}, \{\mathbf{e}_{nm}^{(l+1)} : (n,m) \in \mathcal{E}\}\right).$$

- Intuition of the update cycle:
  1. **Edges** are refreshed using local node information **and** global context.
  2. **Nodes** aggregate updated edges and update their own states, again conditioned on global context.
  3. **Graph-level embedding** is updated from the full set of node & edge embeddings, plus its previous state.
- The final graph embedding $\mathbf{g}^{(L)}$ can be used directly for **graph-level prediction tasks** (e.g. classify a molecule, predict network risk, summarize an equity universe, etc.).

Algorithm 13.2: Graph neural network with node, edge, and graph embeddings

Input: Undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$
Initial node embeddings $\{\mathbf{h}_n^{(0)}\}$
Initial edge embeddings $\{\mathbf{e}_{nm}^{(0)}\}$
Initial graph embedding $\mathbf{g}^{(0)}$
Output: Final node embeddings $\{\mathbf{h}_n^{(L)}\}$
Final edge embeddings $\{\mathbf{e}_{nm}^{(L)}\}$
Final graph embedding $\mathbf{g}^{(L)}$

// Iterative message-passing

for $l \in \{0, \ldots, L-1\}$ **do**

$$\mathbf{e}_{nm}^{(l+1)} \leftarrow \text{Update}_{\text{edge}}\left(\mathbf{e}_{nm}^{(l)}, \mathbf{h}_n^{(l)}, \mathbf{h}_m^{(l)}, \mathbf{g}^{(l)}\right)$$

$$\mathbf{z}_n^{(l+1)} \leftarrow \text{Aggregate}_{\text{node}}\left(\left\{\mathbf{e}_{nm}^{(l+1)} : m \in \mathcal{N}(n)\right\}\right)$$

$$\mathbf{h}_n^{(l+1)} \leftarrow \text{Update}_{\text{node}}\left(\mathbf{h}_n^{(l)}, \mathbf{z}_n^{(l+1)}, \mathbf{g}^{(l)}\right)$$

$$\mathbf{g}^{(l+1)} \leftarrow \text{Update}_{\text{graph}}\left(\mathbf{g}^{(l)}, \{\mathbf{h}_n^{(l+1)}\}, \{\mathbf{e}_{nm}^{(l+1)}\}\right)$$

end for

return $\{\mathbf{h}_n^{(L)}\}, \{\mathbf{e}_{nm}^{(L)}\}, \mathbf{g}^{(L)}$

Over-smoothing in GNNs

- **Problem:** in many GNNs, repeated message passing causes node embeddings to become **too similar**.
  - After several layers, $\mathbf{h}_n^{(l)} \approx \mathbf{h}_m^{(l)}$ for many nodes n, m.
  - The network effectively **loses discriminative power** and cannot benefit from additional depth.
- This phenomenon is called **over-smoothing**:
  - Information is mixed so many times across the graph that all nodes end up with nearly the same representation.
  - Limits how deep we can make a message-passing network.

Mitigating over-smoothing: residual & multi-layer readout

1. Residual connections in the node update

- Modify the node update to add a **skip connection**:

$$\mathbf{h}_n^{(l+1)} = \mathrm{Update}_{\mathrm{node}}\left(\mathbf{h}_n^{(l)}, \mathbf{z}_n^{(l+1)}, \mathbf{g}^{(l)}\right) + \mathbf{h}_n^{(l)}.$$

- Keeps some of the **original node signal** at each layer, slowing down the tendency toward uniform embeddings.

2. Readout from all layers (not only the last)

- Let the output layer see **representations from every layer**:

$$\mathbf{y}_n = f\left(\mathbf{h}_n^{(1)} \oplus \mathbf{h}_n^{(2)} \oplus \cdots \oplus \mathbf{h}_n^{(L)}\right),$$

where $\oplus$ denotes **concatenation**.

- Variants:
  - Replace concatenation by **element-wise max pooling** over $\{\mathbf{h}_n^{(l)}\}_{l=1}^{L}$.
- Intuition:
  - Earlier layers keep **local, less-smoothed** information.
  - Later layers capture **more global context**.
  - The readout learns how to combine them, reducing the impact of over-smoothing in deep GNNs.

Regularization in Graph Neural Networks

- Standard NN regularization tools still apply:
  - Add **penalty terms** to the loss (e.g. sum-of-squares of parameters, L2).
  - Early stopping, weight decay, etc.
- Some methods are particularly natural for GNNs.

- **Weight sharing**
  - GNNs already share weights across nodes/edges to enforce permutation equivariance / invariance.
  - Typically, each layer has its **own** parameters.
  - Further regularization: **share weights across layers** to reduce the total number of free parameters.

Dropout on nodes and edges

- **Node dropout** for GNNs:
  - During training, randomly omit **subsets of nodes** from the graph on each forward pass.
  - Forces the network to rely on multiple paths / neighbours, improving robustness.
- **Edge dropout** (a.k.a. drop-edges):
  - Randomly remove or **mask entries in the adjacency matrix** during training.
  - Encourages the model not to overfit to any specific edge pattern.
- Both node and edge dropout are GNN-specific analogues of standard dropout, tailored to graph structure.

Geometric Deep Learning - Motivation

- So far, we enforced **permutation symmetry** in GNNs:
  - Reordering node labels permutes outputs in the same way.
  - Acts as a strong **inductive bias**, reducing data needs.
- Many graph problems also have **geometric symmetries**:
  - Molecules, meshes, fluids, 3D scenes.
  - Their properties are **invariant** to:
    - Translations (moving the whole object),
    - Rotations,
    - Reflections (mirroring).
- Example: predicting a molecule's properties
  - Nodes = atoms $(C, H, N, \cdots)$.
  - Each atom n has a 3D coordinate $\mathbf{r}_n^{(0)} \in \mathbb{R}^3$.
  - Property (e.g. solubility) does **not** change if we rotate / translate / reflect the whole molecule.
- Goal of **geometric deep learning**:
  - Design update and aggregation rules so that learned representations respect these geometric invariances / equivariances.

Back to the stock return prediction problem: Why not just use sector dummies?

Sector dummies = extremely coarse graph

- A sector dummy says only:
  "Stock i and j share the same label (Energy, Tech, $\cdots$)."
- Limitations:
  - **Binary, unweighted:** all pairs in a sector treated equally similar.
  - **No within-sector topology:** no notion of who is central / peripheral.
  - **No cross-sector structure:** banks ↔ homebuilders ↔ REITs, refiners ↔ airlines, etc.
  - **Static:** sector classifications change slowly, but correlations, supply chains, and flows can change weekly.

Intuition:
Using sector dummies to model dependence is like doing image recognition with a feature saying "this pixel is in the top half of the image" instead of using a convolution over actual neighbouring pixels.

What GNNs buy you beyond sector dummies

1. Richer, *learned* similarity instead of fixed buckets

- Build a graph with **weighted edges** from:
  - Return/covariance similarity,
  - Fundamental similarity (valuation, balance sheet, ESG, etc.),
  - Supply-chain links, ownership overlap, news co-mentions.
- GNN learns how much each edge type/weight matters for predicting excess return, instead of hard-coding "same sector = 1, else 0".

2. Local shock propagation & higher-order effects

- Sector dummy: every stock in the bucket gets the *same* treatment.
- GNN:
  - A shock at node A can propagate to neighbours, then neighbours-of-neighbours, with **learned decay and directionality**.
  - Captures patterns like *"semis → cloud → ad-tech"* or *"oil → airlines"* that don't align cleanly with GICS sectors.

3. Overlapping structures, not just one partition

- A stock can be simultaneously:
  - in a sector,
  - in a factor cluster (value, quality, carry),
  - in a supply-chain cluster,
  - in a "same owner / same ETF" cluster.
- Dummies can only encode one or a few of these;
  a graph can represent **multiple edge types** at once, and GNNs can attend to whichever structure is most predictive in the current regime.

4. Regime-dependent, time-varying dependence

- Correlation graphs **change over time** (crisis vs calm, value vs growth rotations).
- GNNs can be fed **time-varying graphs**:
  - e.g., rolling covariance network, dynamic co-movement graph.
- Sector dummies are frozen; they can't express that "these two tech names move together *now* in this regime, but not six months ago."

5. Empirical precedent (analogy with CNNs)

- We once could have said for images:
  "Why not just use pixel-location dummies in an MLP?"
  – technically possible, but we know CNNs win because they **respect structure**.
- GNNs play the same role for securities:
  - take the cross-sectional dependence seriously, not as a couple of one-hot flags.

> So the pitch:
> **Sector dummies give you a coarse partition; GNNs give you a flexible, time-varying, multi-relational graph where information can propagate.**
> That's the gap we're trying to close when we move from tabular models to GNNs for excess-return prediction.

Illustration: GNN for regression

1.1 Graph Construction: Stochastic Block Model (SBM)

We use the **Stochastic Block Model** to generate a graph with realistic community structure, mimicking how financial assets cluster into sectors.

Definition: Given n nodes partitioned into K communities, the SBM generates edges independently:

$$P(A_{ij} = 1) = \begin{cases} p_{\text{in}} & \text{if } c_i = c_j \text{ (same community)} \\ p_{\text{out}} & \text{if } c_i \neq c_j \text{ (different communities)} \end{cases}$$

where:

- A_{ij} is the adjacency matrix entry
- c_i is the community assignment of node i
- $p_{\text{in}} > p_{\text{out}}$ ensures dense within-community connections

Parameters used:

- $n = 300$ nodes (assets)
- $K = 5$ communities (sectors)
- $p_{\text{in}} = 0.3$ (30% chance of edge within sector)
- $p_{\text{out}} = 0.02$ (2% chance of edge across sectors)

This creates a **block-diagonal** adjacency matrix structure visible when nodes are sorted by community.

1.2 Node Features

Each node i has a feature vector $\mathbf{x}_i \in \mathbb{R}^5$ drawn i.i.d. from a standard normal:

$$\mathbf{x}_i \sim \mathcal{N}(\mathbf{0}, \mathbf{I}_5)$$

These features are **independent** of the graph structure. In a financial context, these could represent:

- Firm characteristics (size, value, momentum scores)
- Recent return statistics
- Fundamental ratios

1.3 Target Generation: The Key Design

The target y_i is constructed using **all features** from both self and neighbors:

$$y_i = \alpha \cdot \tilde{s}_i + \beta \cdot \tilde{h}_i + \epsilon_i$$

where:

- $\tilde{s}_i$ = scaled sum of own features (own signal)
- $\tilde{h}_i$ = scaled sum of neighbor-averaged features (neighbor signal)
- $\epsilon_i \sim \mathcal{N}(0, \sigma^2)$ = noise

Own signal (sum over all features):

$$s_i = \sum_{k=0}^{4} x_i^{(k)} = \mathbf{1}^\top \mathbf{x}_i$$

Neighbor signal (sum of mean neighbor features):

$$h_i = \sum_{k=0}^{4} \bar{x}_{\mathcal{N}(i)}^{(k)} = \mathbf{1}^\top \bar{\mathbf{x}}_{\mathcal{N}(i)}$$

where $\bar{\mathbf{x}}_{\mathcal{N}(i)} = \frac{1}{|\mathcal{N}(i)|} \sum_{j \in \mathcal{N}(i)} \mathbf{x}_j$ is the mean feature vector of neighbors.

Variance Scaling: Both signals are scaled to unit variance:

$$\tilde{s}_i = \frac{s_i}{\text{std}(s)}, \quad \tilde{h}_i = \frac{h_i}{\text{std}(h)}$$

This ensures α and β directly reflect variance contributions.

Parameters used:

- $\alpha = 0.3$ (own signal weight)
- $\beta = 0.7$ (neighbor signal weight)
- $\sigma = 0.1$ (noise std)

This means **~84% of signal variance** comes from neighbors ($0.7^2/(0.3^2 + 0.7^2) = 0.84$).

1.5 Why This Design Creates GNN Advantage

Model	Can access $\mathbf{x}_i$?	Can access $\{\mathbf{x}_j : j \in \mathcal{N}(i)\}$?	Learnable signal
Ridge/MLP	Yes	**No**	Only $\alpha \cdot \tilde{s}_i$ (~16% of variance)
GCN/GAT	Yes	**Yes** (via message passing)	Both components (~100% of variance)

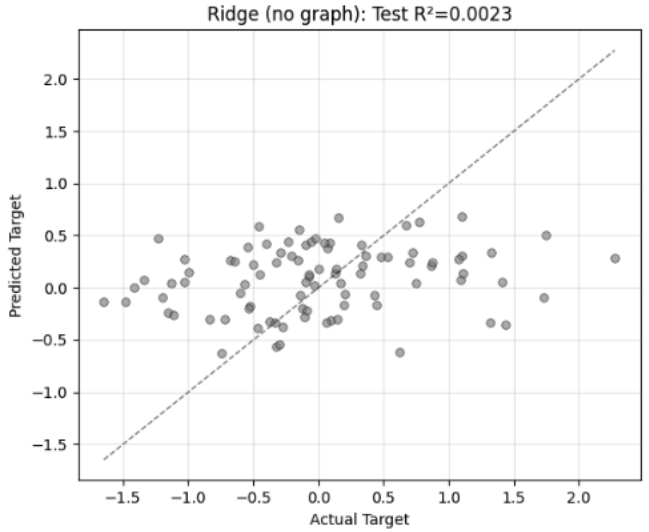
The baseline models are **structurally limited** - they cannot access neighbor features regardless of model capacity.

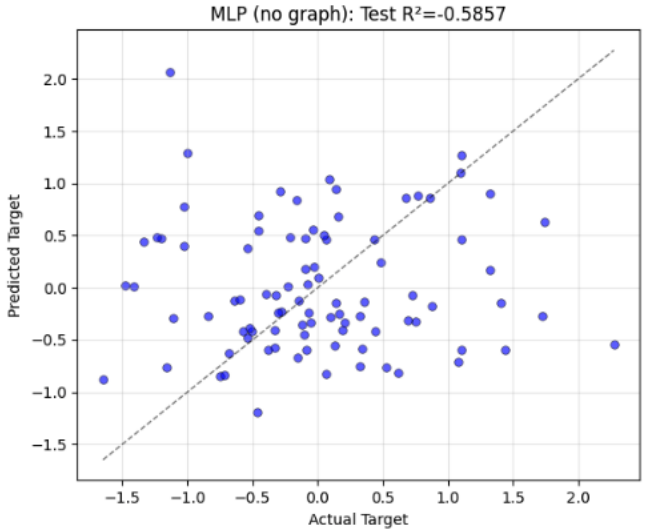
Illustration: GNN for regression

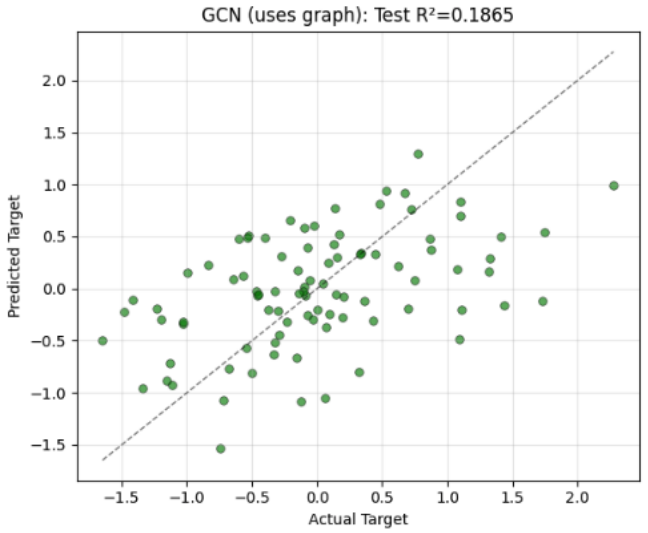
```
======================================================================
RESULTS SUMMARY
======================================================================

Signal: 30% own features + 70% neighbor features
Baselines can only capture the 30% own-feature component.
GNNs can capture both components via message passing.

----------------------------------------------------------------------
Model          Train R²      Test R²     Test Corr   Uses Graph
----------------------------------------------------------------------
Ridge            0.1913       0.0023        0.2162           No
MLP              0.9370      -0.5857        0.0148           No
GCN              0.6634       0.1865        0.4786          Yes
GAT              0.9539       0.8735        0.9388          Yes
----------------------------------------------------------------------
```







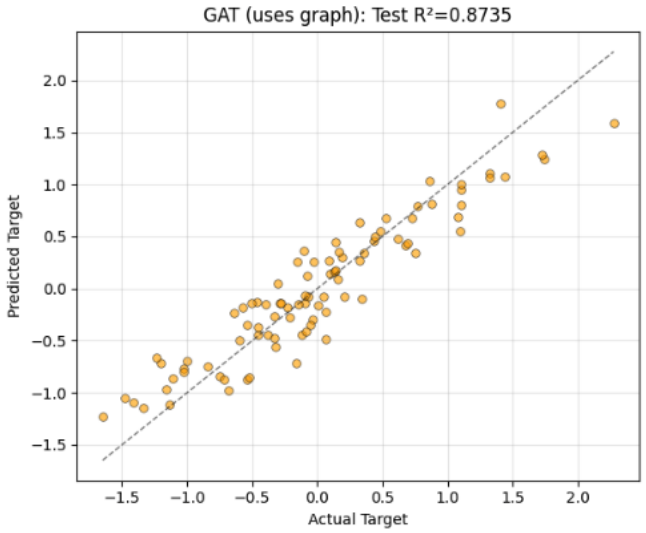


Illustration: GNN for classification

1.1 Graph Construction: Stochastic Block Model (SBM)

We use the **Stochastic Block Model** to generate a graph with realistic community structure, mimicking how financial assets cluster into sectors.

Definition: Given n nodes partitioned into K communities, the SBM generates edges independently:

$$P(A_{ij} = 1) = \begin{cases} p_{\text{in}} & \text{if } c_i = c_j \text{ (same community)} \\ p_{\text{out}} & \text{if } c_i \neq c_j \text{ (different communities)} \end{cases}$$

where:

- A_{ij} is the adjacency matrix entry
- c_i is the community assignment of node i
- $p_{\text{in}} > p_{\text{out}}$ ensures dense within-community connections

Parameters used:

- $n = 300$ nodes (assets)
- $K = 5$ communities (sectors)
- $p_{\text{in}} = 0.3$ (30% chance of edge within sector)
- $p_{\text{out}} = 0.02$ (2% chance of edge across sectors)

1.2 Node Features

Each node i has a feature vector $\mathbf{x}_i \in \mathbb{R}^5$ drawn i.i.d. from a standard normal:

$$\mathbf{x}_i \sim \mathcal{N}(\mathbf{0}, \mathbf{I}_5)$$

1.3 Target Generation: Binary Classification

We first compute a continuous latent score using **all features** from both self and neighbors:

$$z_i = \alpha \cdot \tilde{s}_i + \beta \cdot \tilde{h}_i$$

where:

- $\tilde{s}_i$ = scaled sum of own features (own signal)
- $\tilde{h}_i$ = scaled sum of neighbor-averaged features (neighbor signal)

Own signal (sum over all features):

$$s_i = \sum_{k=0}^{4} x_i^{(k)} = \mathbf{1}^\top \mathbf{x}_i$$

Neighbor signal (sum of mean neighbor features):

$$h_i = \sum_{k=0}^{4} \bar{x}_{\mathcal{N}(i)}^{(k)} = \mathbf{1}^\top \bar{\mathbf{x}}_{\mathcal{N}(i)}$$

Binary label via thresholding:

$$y_i = \mathbf{1}[z_i > 0]$$

This creates balanced classes (approximately 50% positive).

Parameters used:

- $\alpha = 0.3$ (own signal weight)
- $\beta = 0.7$ (neighbor signal weight)

This means **~84% of signal** comes from neighbors ($0.7^2/(0.3^2 + 0.7^2) = 0.84$).

Model	Can access $\mathbf{x}_i$?	Can access $\{\mathbf{x}_j : j \in \mathcal{N}(i)\}$?	Learnable signal
LogReg/MLP	Yes	**No**	Only $\alpha \cdot \tilde{s}_i$ (~16% of signal)
GCN/GAT	Yes	**Yes** (via message passing)	Both components (~100% of signal)

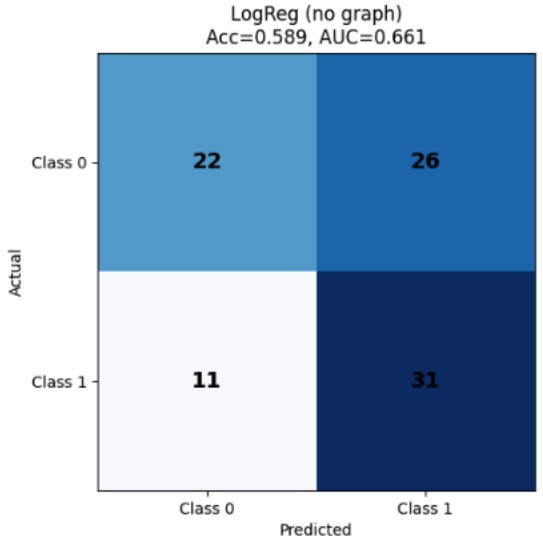
The baseline models are **structurally limited** - they cannot access neighbor features regardless of model capacity.

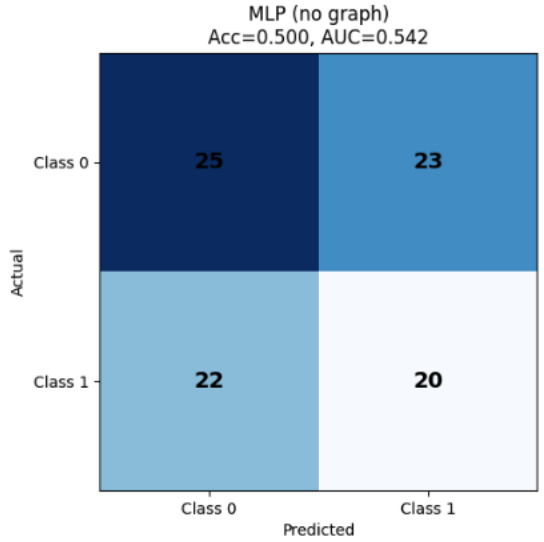
Illustration: GNN for classification

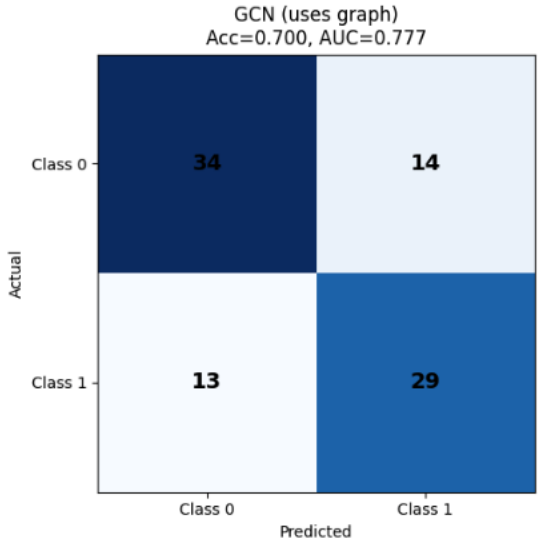
```
================================================================================
RESULTS SUMMARY
================================================================================

Signal: 30% own features + 70% neighbor features
Baselines can only capture the 30% own-feature component.
GNNs can capture both components via message passing.

--------------------------------------------------------------------------------
Model         Train Acc     Test Acc      Test F1      Test AUC   Uses Graph
--------------------------------------------------------------------------------
LogReg           0.7000       0.5889       0.6263       0.6607           No
MLP              1.0000       0.5000       0.4706       0.5424           No
GCN              0.8143       0.7000       0.6824       0.7773          Yes
GAT              0.9810       0.8889       0.8684       0.9653          Yes
--------------------------------------------------------------------------------
```







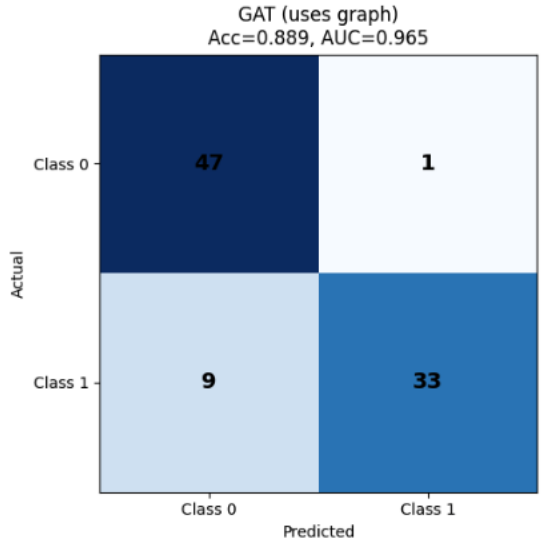
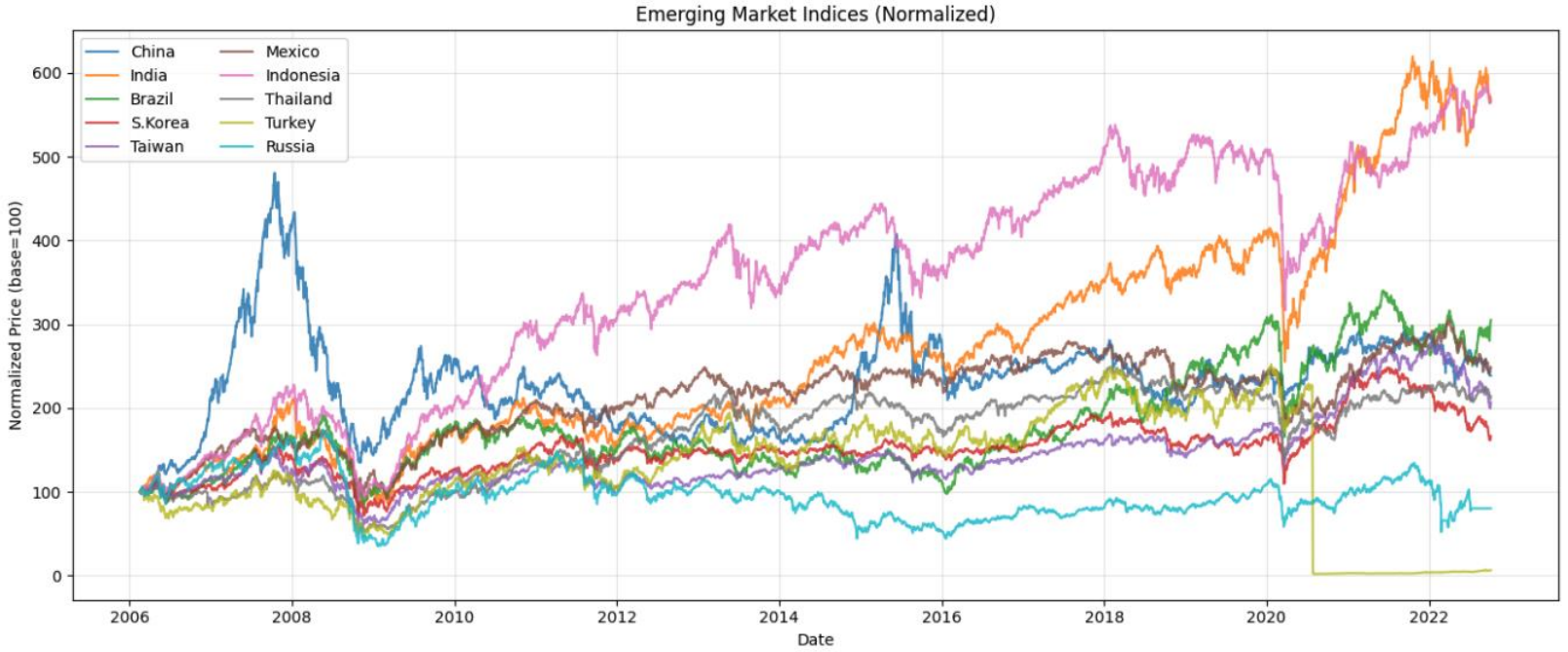


Illustration: GNN for Emerging Markets Return Prediction



Ticker	Index	Country
I2CHN003	Shanghai SE Composite	China
I2IND007	S&P CNX Nifty	India
I4BRA002	Bovespa	Brazil
I2KOR003	KOSPI	South Korea
I2TWN002	Taiwan Weighted	Taiwan
I4MEX005	IPC	Mexico
I2IDN003	Jakarta Composite	Indonesia
I2THA002	SET Index	Thailand
I3TUR002	ISE 30	Turkey
I3RUS002	RTS	Russia

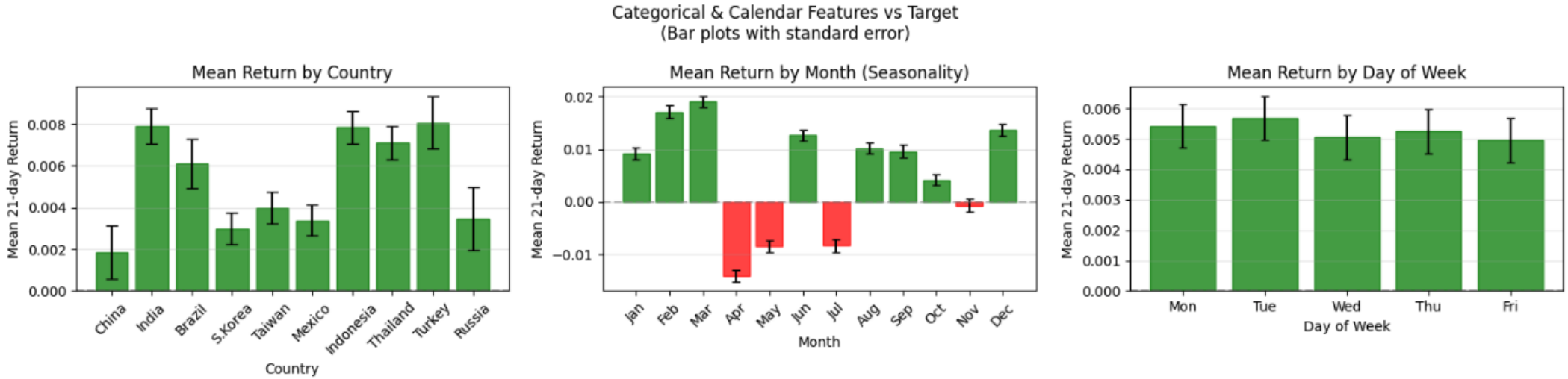


Illustration: GNN for Emerging Markets Return Prediction

Technical Indicators (10)

1. **Momentum (21-day)**: Return over past month
2. **Momentum (63-day)**: Return over past quarter
3. **Volatility (21-day)**: Std of returns over past month
4. **RSI (14-day)**: Relative Strength Index
5. **MA Ratio (50-day)**: Price / 50-day moving average
6. **MACD**: 12-day EMA - 26-day EMA
7. **Bollinger Position**: (Price - MA20) / (2 * std20)
8. **ATR (14-day)**: Average True Range (using close-to-close)
9. **Volatility Ratio**: Rolling std ratio (recent vs longer-term)
10. **Drawdown (63-day)**: Distance from 63-day high

Mean-Reversion Features (3)

11. **Distance from 252-day high**: How far below annual high
12. **Distance from 252-day low**: How far above annual low
13. **Return Z-score**: How unusual is recent return vs rolling history

Cross-Asset/Macro Proxies (3)

14. **Correlation with EM average**: Rolling correlation with average EM return
15. **Cross-sectional dispersion**: Dispersion across all EM indices (risk-on/off proxy)
16. **Beta to EM average**: Rolling beta to the EM average return

Calendar Features (4)

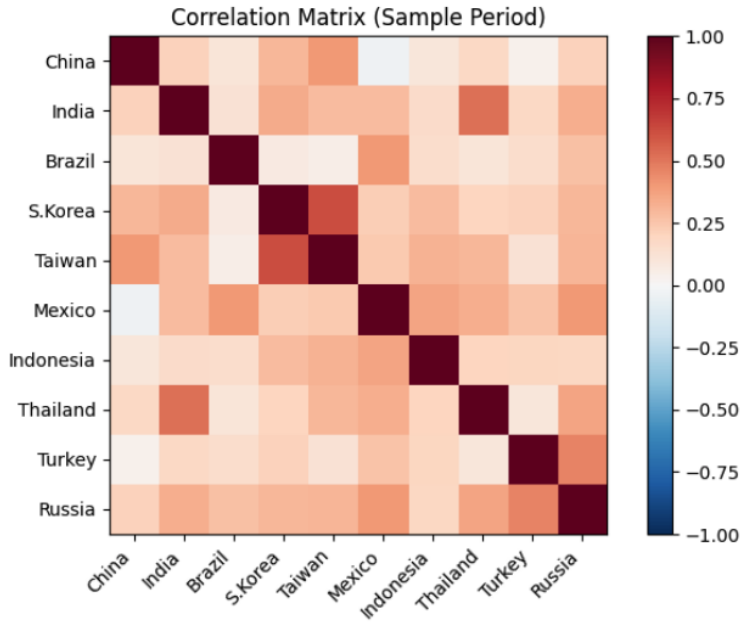
17-18. **Month of year**: Cyclical encoding (sin/cos) for seasonality 19-20. **Day of week**: Cyclical encoding (sin/cos) for intra-week effects

Country One-Hot Encoding (10)

21-30. **Country indicators**: One-hot encoding for each of the 10 countries

```
Feature Importance Summary (correlation with target):
--------------------------------------------------
  drawdown_63               -0.0852
  mom_63                    -0.0807
  month_cos                 +0.0803
  dist_from_high_252        -0.0711
  macd                      -0.0661
  ma_ratio_50               -0.0634
  atr_14                    +0.0442
  vol_21                    +0.0406
  dist_from_low_252         -0.0392
  mom_21                    -0.0354
```

Illustration: GNN for Emerging Markets Return Prediction



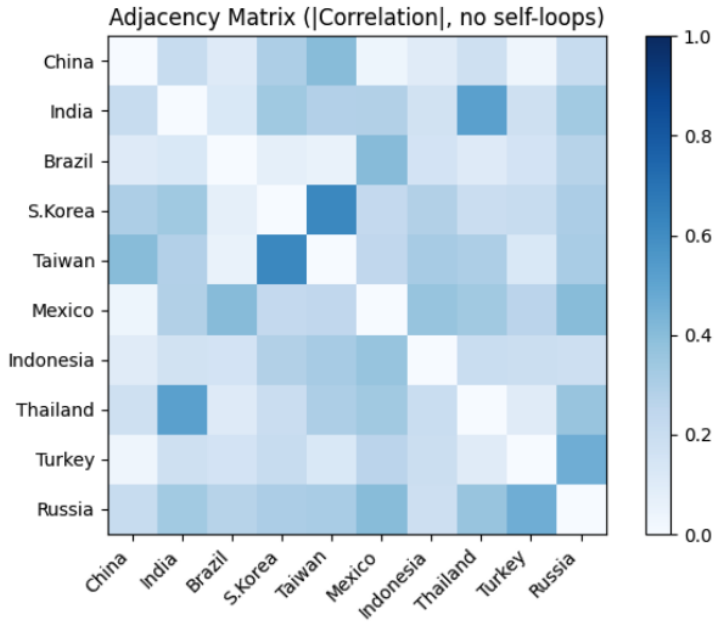
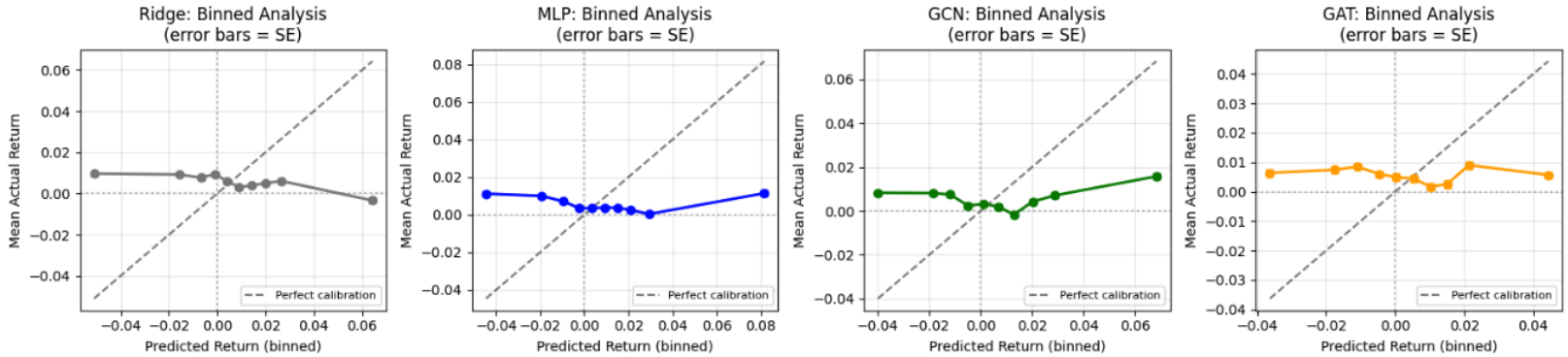


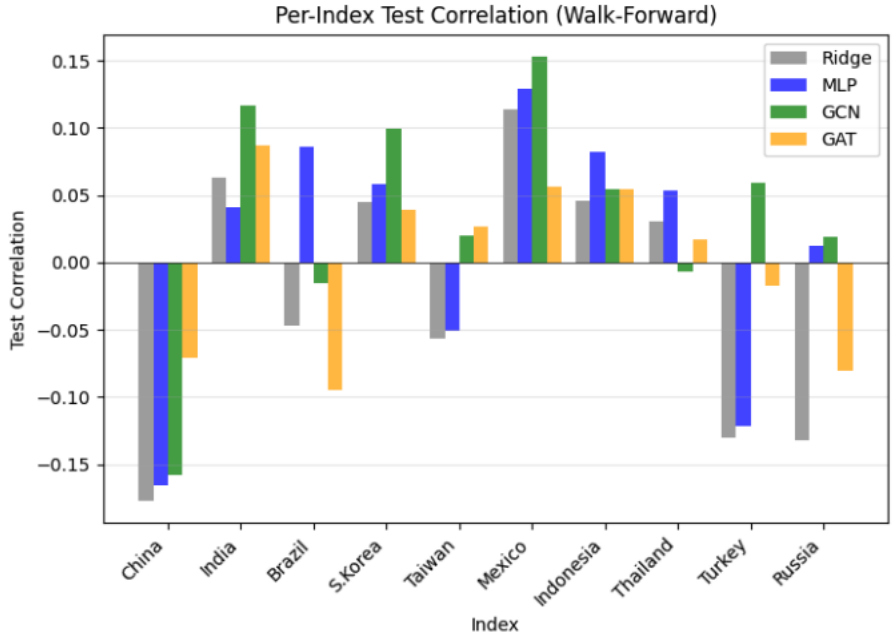
Illustration: GNN for Emerging Markets Return Prediction

```
======================================================================
WALK-FORWARD RESULTS SUMMARY
======================================================================

Predicting next 21-day (1 month) returns of 10 emerging market indices
Using 30 technical features and covariance-based graph
Walk-forward: 99 folds, 504-day training window, 21-day test, 21-day buffer
Total test samples: 2079 days x 10 indices
----------------------------------------------------------------------
Model           Test R2    Test Corr    Test RMSE   Uses Graph
----------------------------------------------------------------------
Ridge           -0.2394      -0.0599       0.0560           No
MLP             -0.2362      -0.0105       0.0559           No
GCN             -0.1745       0.0192       0.0545          Yes
GAT             -0.1535      -0.0143       0.0540          Yes
----------------------------------------------------------------------

GNN improvement over best baseline:
  R2:   +0.0826
  Corr: +0.0296
======================================================================
```





References

- Chris Bishop, Hugh Bishop (2023) Deep Learning, The MIT Press.
- Franco Scarselli; Marco Gori; Ah Chung Tsoi; Markus Hagen-buchner; Gabriele Monfardini (2009), The Graph Neural Network Model IEEE Transactions on Neural Networks