Logic, Data Examples and Learning

Lecture 4/5: Graph neural networks and their connections to logic

Balder ten Cate

Tsinghua University, May 23, 2026

How This Lecture Fits into the Series

The first three lectures mainly stayed on the *symbolic* side: our concept classes consists of logical formulas, ot were defined through logic. We then looked at:

- concepts to examples,
- examples to concepts,
- and when fitting concepts generalize.

Today, we instead look at an ML architecture used in practice (Graph Neural Networks) and use logic to study it.

Plan for Today

1. An introduction to ML and graph neural networks.
2. Why logic is useful for understanding GNNs.
3. Three concrete logic connections:
   - graded modal logic,
   - Presburger modal logic,
   - FO+C and guarded fragments as upper bounds.

The overall message is:

logic can help us understand the capabilities and limitations of GNN architectures.

Introduction to Machine Learning and GNNs

Supervised Machine Learning

In supervised learning, we are given examples with labels and we try to learn a rule that predicts labels for new examples.

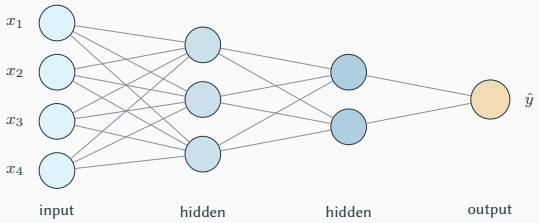
$$\text{training examples} \longrightarrow \text{learned model} \longrightarrow \text{predictions on new inputs}$$

In this lecture, the learned models will be neural networks, and the inputs will often be graphs.

Neural Networks

A feed-forward neural network (FFN), or *multi-layer perceptron*, consists of:

- an input layer,
- one or more hidden layers,
- an output layer.

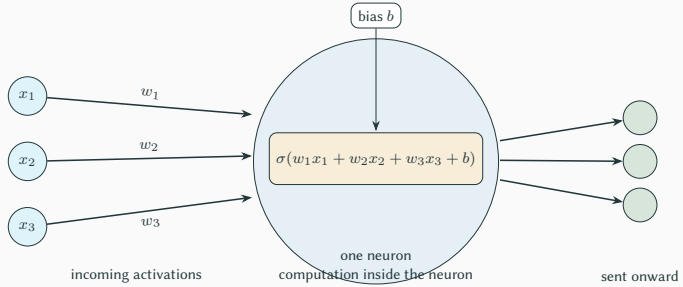


Information flows only from left to right. This is what makes it *feed-forward*.

Historic note. Neural-network ideas go back to McCulloch–Pitts (1943), with the perceptron due to Rosenblatt (1958). Multi-layer networks became practically central in the 1980s.

One Neuron Up Close

Each neuron receives the activations from the previous layer, computes a weighted sum plus a bias, and then applies an "activation function" σ.



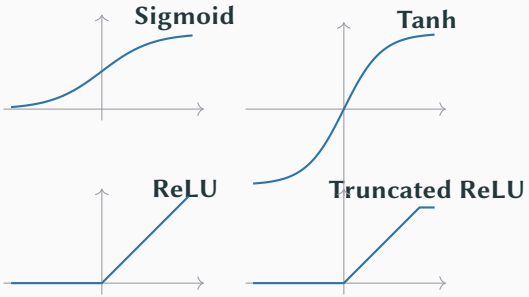
By an *affine map* we simply mean a map of the form

$$x_1, \ldots, x_n \mapsto w_1 \cdot x_1 + \cdots + w_n \cdot x_n + b,$$

that is: a linear map plus a bias term.

Common Activation Functions

Activation functions inject the non-linearity that makes neural networks expressive.

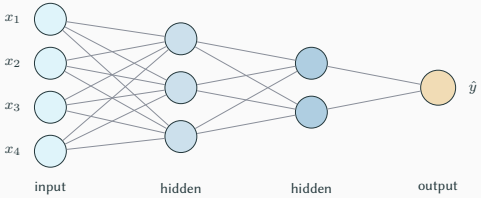


Typical choices include:

- sigmoid and $\tanh$ in older network architectures,
- ReLU and variants of it in much modern deep learning.

Hidden Layers Compute Embedding Vectors

For a fixed input, every hidden layer produces a new vector representation of that input.



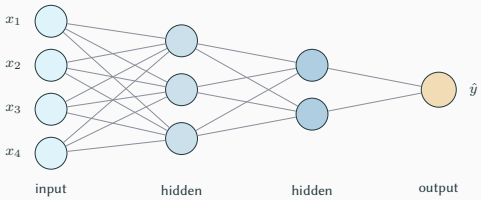
Layer 0

Input vector

$$x = [0, 1, 0, 1]$$

Hidden Layers Compute Embedding Vectors

For a fixed input, every hidden layer produces a new vector representation of that input.



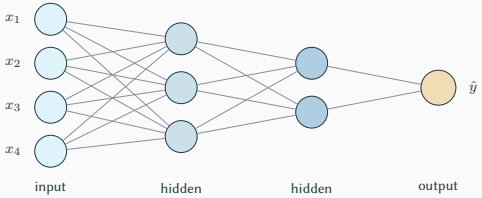
Layer 1

First hidden embedding

$$h^{(1)} = [1.4, 0.0, 0.7]$$

Hidden Layers Compute Embedding Vectors

For a fixed input, every hidden layer produces a new vector representation of that input.



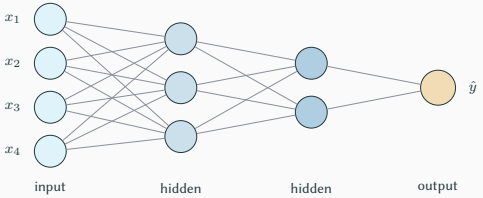
Layer 2

Second hidden embedding

$$h^{(2)} = [0.2, 2.1]$$

Hidden Layers Compute Embedding Vectors

For a fixed input, every hidden layer produces a new vector representation of that input.



Output layer

Final score vector

$$s = [-1.3]$$

Layer by Layer in Linear-Algebra Form

The same feed-forward computation can be viewed more compactly in terms of vectors, matrices, and matrix multiplication.

$$\text{emb}^{(\ell+1)} = \sigma\big(W^{(\ell)}\text{emb}^{(\ell)} + b^{(\ell)}\big).$$

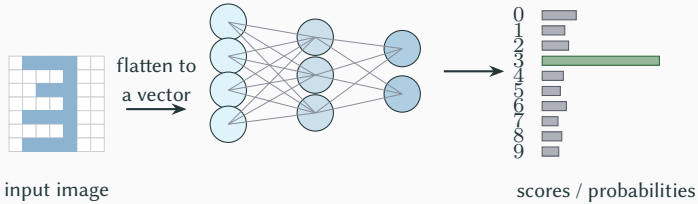
Here:

- $\text{emb}^{(\ell)}$ is the whole embedding vector at layer ℓ,
- $W^{(\ell)}$ is the weight matrix that mixes the coordinates,
- $b^{(\ell)}$ is the bias vector,
- σ is the activation function, applied coordinate-wise.

So a feed-forward network is just:
repeated affine map (matrix multiplication plus a bias term), followed by a non-linear map σ.

Example: Handwritten Digit Recognition

A standard benchmark problem is MNIST: given an image of a handwritten digit, predict whether it is a $0, 1, \ldots, 9$.



Here the input is an image, but the network still just sees a vector of numbers.

Classification vs. Regression

Two common kinds of prediction tasks are:

Classification

Output is chosen from a finite set of labels.

- digit is $0, \ldots, 9$
- email is spam / not spam
- graph node is fraudulent / not fraudulent

Regression

Output is a real number or a real vector.

- predicted temperature
- travel time on a road segment
- a vector of probabilities or scores

The same broad architecture can often be used for both; what changes is how we interpret and train the output layer.

What Are the Parameters?

The “parameters” of the network are:

- the weights on the edges,
- the bias terms in the neurons.

They are often collected together as one big parameter vector θ.

Two different kinds of choices

Parameters: learned from data.

Hyperparameters: chosen by us, such as number of layers, hidden dimension, and activation function.

Python (Pseudo) Code

```
model = nn.FFN(
    layers=[20, 64, 1],
    activation=ReLU,
)

model.train(
    examples,
    labels,
    loss=MeanSquaredError,
    optimizer=Adam(lr=0.001),
    epochs=200,
)
```

How Training Works

Training means adjusting the parameters so that the network's predictions match the labeled data as well as possible.

The basic ingredients are:

- a prediction $\hat{y} = f_\theta(x)$,
- a loss function $L(y, \hat{y})$,
- an optimization method that improves θ.

Backpropagation

Basic training loop:

1. run the network forward to compute the prediction and the loss;
2. compute how the loss changes when each weight changes (the “gradient”);
3. move the weights a little in the direction that makes the loss smaller;
4. repeat many times.

Backpropagation

Backpropagation is the efficient procedure that computes these gradients layer by layer, from the output back to the input.

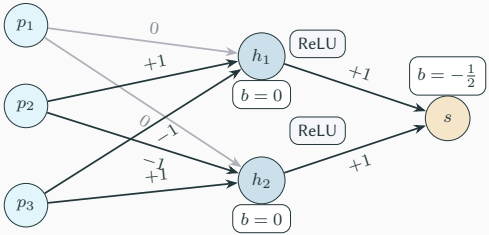
Relies on differentiable activations.

Historic note. The standard backpropagation story is classically associated with Rumelhart, Hinton, and Williams (1986), though the underlying differentiation ideas are older.

Quiz: What Boolean Function Is This?

Consider a feed-forward classifier whose input is a bit-vector encoding a valuation for proposition letters $\{p_1, p_2, p_3\}$.

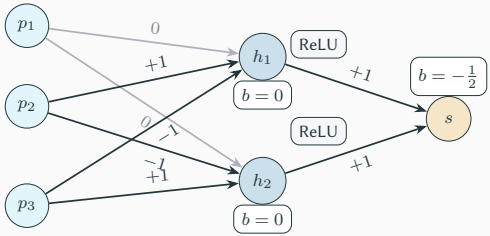
Final classification is YES if the output score is > 0, and NO otherwise.



Quiz: What Boolean Function Is This?

Consider a feed-forward classifier whose input is a bit-vector encoding a valuation for proposition letters $\{p_1, p_2, p_3\}$.

Final classification is YES if the output score is > 0, and NO otherwise.



Answer

It computes p_2 XOR p_3.

From Vector Learning to Graph Learning

What MLPs Are Good At

For fixed-length vectors, MLPs are an extremely flexible architecture.

- Inputs have a fixed dimension.
- The order of the coordinates is built in.
- With enough hidden units, they can represent a huge range of functions.

So this is already a powerful model. But it assumes the input comes as a fixed vector with a built-in coordinate system.

Different Architectures for Different Types of Inputs

- For fixed length sequences of numbers, MLPs are a natural starting point.
- For images, convolutional neural networks exploit the 2-dimensional geometry of pixels.
- For graphs, we should likewise exploit graph structure.

Lesson from deep learning: the architecture should reflect the structure of the input.

Why Graphs Need Something Different

A learning architecture for graphs should respect at least three basic facts:

- graphs can have different sizes;
- each node only has local neighbors, not a fixed coordinate position;
- isomorphic graphs should not be distinguished just because we renamed the vertices.

This is the starting point for graph neural networks.

Examples of Graph Learning Tasks

Graph learning shows up in many places:

- molecules and reaction prediction,
- social and citation networks,
- recommender systems,
- knowledge graphs,
- program-analysis graphs.

Typical tasks are:

- classify a node,
- classify a whole graph,
- predict a missing edge or label.

How GNNs Work

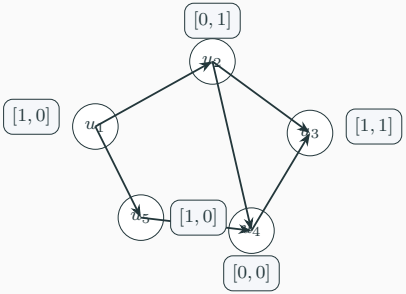
Input: A Featured Graph

A graph neural network takes as input a graph whose nodes already carry feature vectors.

For the logic connection, it is often enough to think of:

- Boolean node labels,
- directed edges,
- and sometimes edge labels for different relation types.

We will consider node-labeled directed graphs here (to make the connection with logic the easiest to explain)



Historic note. Graph neural networks first appeared in the late 2000s, beginning with Scarselli et al. (2008/2009). The message-passing viewpoint was further crystallized in the 2010s, for instance by Gilmer et al. (2017).

A GNN Runs for a Fixed Number of Rounds

We will focus on message-passing GNNs that run for some fixed number of update rounds, say N.

Each node v starts with an initial embedding vector $\mathrm{emb}_0(v) \in \mathbb{R}^d$.

Then, in each round, all node embeddings are updated in parallel. So for each node v we obtain a sequence

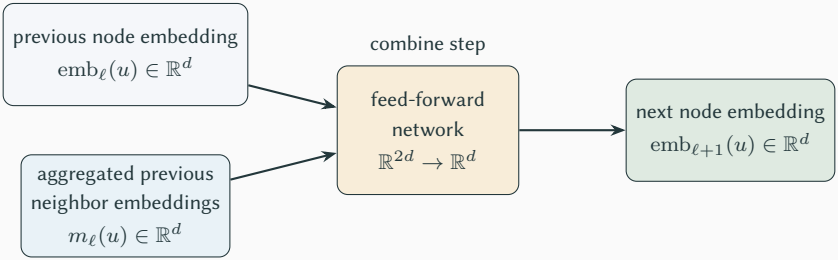
$$\mathrm{emb}_0(v),\ \mathrm{emb}_1(v),\ \ldots,\ \mathrm{emb}_N(v) \in \mathbb{R}^d.$$

Intuitively:

- round 0 is the input description of the node;
- round 1 incorporates information aggregated from immediate neighbors;
- later rounds propagate information further through the graph.

The Update at One Node Is Just an FFN

After aggregation, the update at a single node is an ordinary feed-forward computation.



So for fixed feature dimension d, a message-passing layer is:

aggregate neighbor information, then apply the same small neural network at every node.

The Aggregate-Combine Template

At each layer, a node updates its feature vector by looking at:

- its own current embedding vector;
- the multiset of embedding vectors of its neighbors.

The schematic form is:

$$\mathrm{emb}_{\ell+1}(v) = \mathrm{FFN}\Big(\mathrm{emb}_\ell(v), \mathrm{agg}\{\!\{\mathrm{emb}_\ell(u) \mid v \text{ is a neighbor of } u\}\!\}\Big).$$

The design choices now become crucial:

- what the aggregation function is;
- what kind of "neighbors" we consider (successors, predecessors, global, ...)
- what kind of FFNs are allowed (e.g., which activation functions).

The Simple GNN Model We Will Focus On

In this lecture, we will focus on:

- "LocalSum" aggregation: aggregate embeddings of *successors* by *summing*).
- Activation function: either ReLU or Truncated ReLU.

So the update has the form

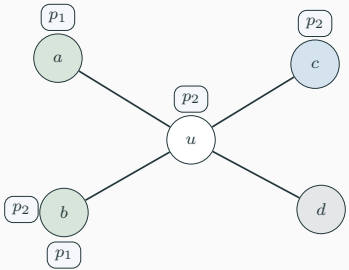
$$\mathrm{emb}_{\ell+1}(v) = \textsf{ReLU-FFN}\Big(\mathrm{emb}_\ell(v), \textsf{sum}\{\!\{\mathrm{emb}_\ell(u) \mid (u,v) \in E\}\!\}\Big).$$

To simplify the exposition, for the rest of the talk, we will only consider such GNNs.

(Other options include aggregation by taking the max; aggregating predecessors, global aggregation, etc.)

Worked Example: A Node-Labeled Graph

Suppose our graph has binary node labels, namely proposition letters p_1 and p_2.

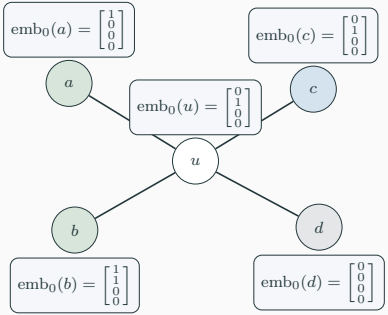


Initial Embeddings Encode the Node Labels

Let the embedding dimension be

$$d = 4.$$

We choose the initial embeddings based on the proposition letters that are satisfied.

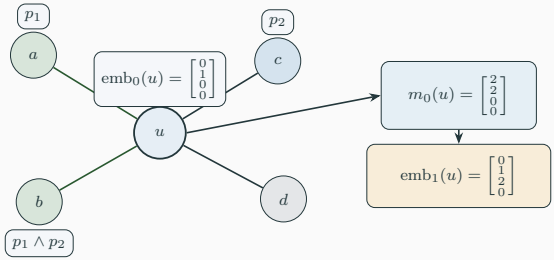


What We Want One Update Round to Compute

Focus on the node u.

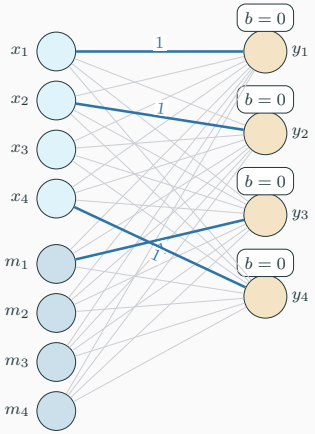
Say we want the next embedding to store the number of p_1-successors in coordinate 3.

In this example, the neighbors of u contain exactly two p_1-nodes, namely a and b.



A Single-Layer FFN Already Suffices

After LocalSum aggregation, the FFN only needs to:



all other weights are 0

A Second Example: A Two-Step Successor Satisfying p_1

Now we want to compute a different node property:

does the current node have a two-step neighbor satisfying p_1?

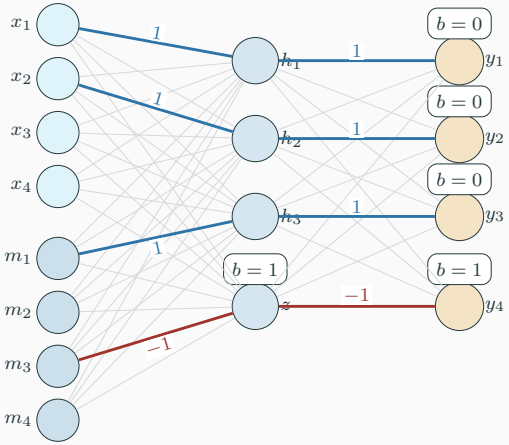
We will store the answer as a Boolean value in coordinate 4.

The idea is to build on the previous example.

This time, we need two rounds of message passing:

- after one round, coordinate 3 stores the number of p_1-neighbors;
- after two rounds, the coordinate 4 will contain the right value.

The Same FFN, with One Extra Hidden Neuron



extra hidden neuron

$z = \mathrm{ReLU}(1 - m_3)$

$y_4 = \mathrm{ReLU}(1 - z)$

Why Two Rounds Are Needed

In this case, we can apply the same FFN for every round.

What happens?

- After the first round, coordinate 3 stores the number of p_1-neighbors, while coordinate 4 is still 0.
- In the second round, the aggregated value m_3 counts how many two-step walks from u end in a p_1-node.
- Therefore coordinate 4 becomes 1 exactly when u has a two-step neighbors satisfying p_1.

So this property is still computable by a GNN, but now we need two rounds of message passing.

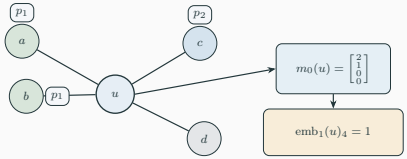
A Last Example: More p_1-neighbors than p_2-neighbors

We can also ask for a simple comparison property:

does the current node have more p_1-neighbors than p_2-neighbors?

After one round, the aggregated vector already contains the relevant counts:

- $m_1 = \#\, p_1$-neighbors,
- $m_2 = \#\, p_2$-neighbors.



With a small ReLU network, we can set

$$\mathrm{emb}_1(u)_4 = \mathrm{ReLU}\big(1 - \mathrm{ReLU}(1 + m_2 - m_1)\big),$$

which is 1 iff $m_1 > m_2$.

Why We Are Doing This

Of course, the point is *not* that we want to design GNNs by hand.

We want to learn them from data.

The point of these examples is to give a sense of the expressive power of GNNs.

GNNs Deliver the Kind of Model We Wanted

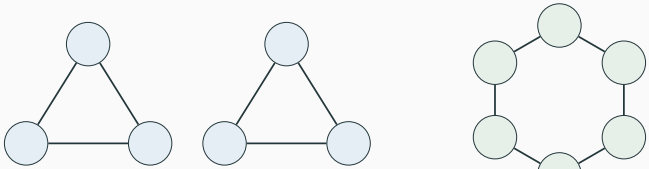
Compared with ordinary feed-forward networks, GNNs do two important things:

- they can process graphs of arbitrary size;
- they are automatically isomorphism invariant.

(They are also successfully trained and used in practice, with many applications.)

GNNs Are Coarser Than Isomorphism

Isomorphism invariance is only a *minimum* requirement.



These two graphs are not isomorphic, but GNNs cannot distinguish them. All nodes (in both graphs) have the same embedding avery every round.

Therefore, the node property "x lies on a triangle" cannot be tested by a GNN.

Why Two Triangles and One Hexagon Look the Same to a GNN

Assume all nodes start with the same initial embedding.

Then, in both graphs:

- every node has exactly two neighbors;
- if all nodes have embedding e_ℓ at round ℓ, then every node aggregates

$$m_\ell = e_\ell + e_\ell = 2e_\ell;$$

- therefore every node computes the same next embedding

$$e_{\ell+1} = \mathrm{FFN}(e_\ell, 2e_\ell).$$

By induction, after any number of rounds, *all* nodes in both graphs still agree on their embeddings.

(The same holds even for global aggregation since both graphs have six nodes.)

GNN Classifiers

A *GNN (node) classifier* consists of:

- a GNN N that computes a final real vector for each node, or for the whole graph after readout;
- an *acceptance condition* telling us when that output counts as YES.

An example acceptance condition

- Last coordinate of the output vector is > 0

The precise acceptance condition won't matter too much.

So, once the final vector has been computed, a classifier turns it into a Boolean or multi-class decision. We can even write $G, v \models N$.

Logic

Next up: connections to logic.

Historic note. The modern logic-for-GNNs line is mostly a 2019–2025 story, building bridges to modal logic, counting logic, and finite model theory.

Why Logic?

The Main Idea

We will discuss connections between

- classes of graph neural networks, and
- logical languages (e.g., modal logics, logics with counting)

Historic note. For graph isomorphism, logical expressiveness, and counting, there is older background going back to Weisfeiler–Leman (1968), Immerman–Lander (1990), Tinhofer (1991), and Cai–Fürer–Immerman (1991).

Why These Characterizations Matter

They are useful for at least four reasons.

- **Capabilities and limitations.** What can this architecture express, and what can it never express?
- **Architecture design.** If the logic is too weak or too strong, that suggests how to extend or restrict the architecture.
- **Explainability.** Logical formulas are often more interpretable than raw learned weights.
- **Static analysis and verification.** Once the behavior is tied to a logical language, we can import decidability and complexity results.

Uniform vs. Non-Uniform Expressive Power

There are two related questions one can ask.

Non-uniform

Given two concrete nodes or graphs, can some GNN in a class distinguish them?

Uniform

Which queries or classification functions can the whole class of GNNs implement?

Examples

- Non-uniform: can some GNN distinguish these two specific pointed graphs?
- Uniform: is there one fixed GNN classifier that recognizes, on every graph, the nodes with at least two outgoing p-neighbors?

In this lecture, we mainly focus on the *uniform* viewpoint.

Roadmap for the Rest of the Lecture

We now look at three progressively stronger counting logics.

1. **Graded modal logic.** A first modal-language view of what message passing can and cannot see.
2. **Presburger modal logic.** This captures the arithmetic flavor introduced by sum aggregation.
3. **FO+C and guarded fragments.** These provide broader upper bounds for more expressive GNN families.

1. Graded Modal Logic

A Tiny Reminder on GML

Graded modal logic extends ordinary modal logic with counting modalities.

Typical syntax

$$\varphi ::= p \mid \neg\varphi \mid \varphi \wedge \psi \mid \Diamond_{\geq k}\varphi$$

Here, $\Diamond_{\geq k}\varphi$ says:

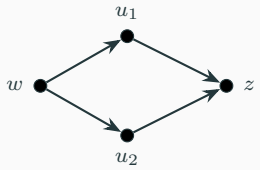
there are at least k outgoing neighbors satisfying φ.

So GML is already a natural language for talking about local graph neighborhoods with counting. For instance, $\Diamond_{\geq 2}p$ says that the current node has at least two outgoing neighbors satisfying p.

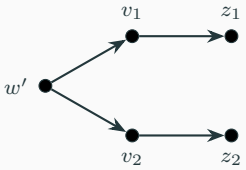
A First Non-Definability Example

Consider the node property:

the current node has more than one distinct two-step successor.



only one two-step successor



two distinct two-step successors

These pointed graphs are *graded bisimilar*. Intuitively, GML can count one step at a time, but it cannot remember that two different paths later merge again.

Graded Bisimulations

A graded bisimulation is like an ordinary bisimulation, but it must preserve *how many* matching successors there are.

Definition

A relation Z between nodes of two directed graphs is a graded bisimulation if whenever uZv:

- u and v satisfy the same atomic propositions;
- for every finite set X of successors of u, there is a set Y of successors of v and a bijection $f : X \to Y$ with $xZf(x)$ for all $x \in X$;
- and symmetrically from v back to u.

Invariance and a van Benthem-Style Theorem

Invariance

If two pointed graphs are graded bisimilar, then they satisfy exactly the same GML formulas.

van Benthem-style theorem (cf. De Rijke 2000, Otto 2019, van Benthem-tC-Väänänen 2009)

A first-order definable node property is definable in GML iff it is invariant under graded bisimulation.

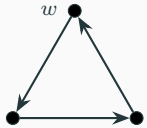
In other words,

> *to show that an FO property is not expressible in* GML*, it is necessary and sufficient that there are two (finite) graded-bisimilar pointed graphs that disagree on the property.*

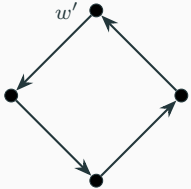
A Second Non-Definability Example

Consider the node property:

the current node lies on a directed triangle.



w lies on a triangle



w' does not lie on a triangle

These pointed graphs are again graded bisimilar: every node has exactly one outgoing edge, so graded bisimulation only sees an infinite-looking one-way unfolding. It does not see the difference between cycle length 3 and cycle length 4.

Why GML Feels Close to Message Passing

A bounded-depth GNN also reasons locally, layer by layer.

- one layer corresponds to one round of neighborhood aggregation;
- several layers correspond to bounded modal depth;
- neighbor multisets matter, so plain bisimulation is too weak;
- graded bisimulation is the right refinement.

Indeed, it turns out that GNNs are invariant for graded bisimulations.

Key Results (Barcelo et al, 2019)

Fact 1

Every GML-formula can be translated to an equivalent GNN classifier.

Fact 2

On finite graphs, FO-definable node properties recognized by these GNN classifiers are already definable in GML.

This holds both on (finite) directed and on (finite) undirected graphs.

Intuition: suppose a node property is:

- computed by such a GNN classifier,
- and also definable by a first-order formula.

Then, on finite structures, graded-bisimulation invariance forces that property to be expressible in GML.

Historic note. For GNNs, this connection is due to Barceló, Kostylev, Monet, Pérez, Reutter, and Silva (2019/2020).

Beyond FO: a Simple Example

Some node properties testable by GNN classifiers are not first-order definable.

Cardinality comparisons

The property "*There are more p_1-neighbors than p_2-neighbors*" is expressible as a GNN-classifier but is not first-order definable.

What This Tells Us

The graded-modal perspective gives us:

- a clean explanation of the locality of message passing;
- a limitation result: GNNs cannot see beyond graded-bisimulation types;
- a first bridge from neural architectures to classical modal logic.

But GML only counts with threshold-like operators, and the exact logical characterization of all GNN-definable properties seems much harder once we move beyond the FO fragment.

The next two parts give partial answers to that harder question.

To understand the full power of GNNs, we need a richer arithmetic language.

2. Presburger Modal Logic

Modal Logic with Presburger Quantifiers

We use a "modal" language in which formulas still talk about one node x, but can contain arithmetic counting conditions.

Typical shape

$$\sum_{i=1}^{m} \lambda_i \cdot \#y \left[\varepsilon_i(x,y) \wedge \varphi_i(y)\right] \;\leq\; \delta$$

Here:

- λ_i, δ are integers,
- $\varepsilon_i(x,y)$ is an atomic formula (eg. the edge relation of the graph),
- $\varphi_i(y)$ is a formula describing the counted neighbors.

We will call this language MP.

Example

The formula

$$3 \cdot \#y\,[E(x,y) \wedge Red(y)] - 2 \cdot \#y\,[E(x,y) \wedge Blue(y)] \geq 5$$

says:

> *three times the number of red outgoing neighbors, minus two times the number of blue outgoing neighbors, is at least five.*

This is exactly the kind of linear arithmetic that GNNs can do using their FFNs.

Expressive Correspondence

Theorem (Benedikt et al. 2024)

GNN classifiers with rational weights and truncated RelU activation are expressively equivalent to local Presburger modal logic.

Shows that *truncated ReLU* it is expressive enough to be interesting, but still tame enough to admit a clean logical characterization.

Why the Translation Works

The two directions have very different flavors.

- From logic to GNN: build one feature for each subformula and use one message passing round for each nesting of modal operators.
- From GNN to logic: show that, when we fix the number of message passing rounds, there are only finitely many “types” and each type can be described by a formula.

The second direction works because every FFN layer only performs linear combinations of counts followed by a controlled activation.

Verification Consequences

Once we have a translation into MP and its local variants, we can import logical decidability results.

- Satisfiability is decidable for GNNs with truncated ReLU. In fact, it is PSPACE-complete. Likewise for equivalance.
- For GNNs with ordinary ReLU activations, satisfiability is undecidable. Some variants with additional restrictions can recover decidability.

This is a good example of logic serving as a tool for static analysis of neural architectures.

Historic note. The decidability transfer uses logical results such as Bednarczyk et al. (2021). The undecidability side ultimately connects to classical negative results in the spirit of Hilbert's tenth problem from the 1970s.

3. FO+C as an Upper Bound

Why We Need a Stronger Logic

Presburger modal logic is already quite expressive, but there are two reasons to look higher:

- In general, GNN are more expressive than Presburger modal logic;
- for complexity-theoretic and uniformity questions, a first-order language with explicit number variables provides a helpful perspective.

This leads us to FO + C: first-order logic with counting and arithmetic.

FO+C in More Detail

FO + C is a *two-sorted* logic.

- **Element variables** $x, y, z, \ldots$ range over graph nodes.
- **Number variables** $i, j, k, \ldots$ range over natural numbers.

In the standard finite-model-theory setup, on an input with n nodes, you can think of the number sort as an initial segment of $\mathbb{N}$ together with built-in arithmetic:

$$0,\ 1,\ <,\ +,\ \times.$$

The logic then allows:

- ordinary first-order quantification over nodes,
- counting terms such as $\#x\,\varphi(x)$,
- Bounded quantification over numbers of the form $\exists n < m \cdots$ and $\forall n < m \cdots$.
- comparisons and arithmetic combinations of the resulting numbers.

What FO+C Has That Modal Presburger Logic Lacks

Both MP and FO + C can count, but FO + C is much more flexible.

- MP stays centered at one current node and forms arithmetic combinations of *local* counts.
- FO + C can quantify over nodes and number variables globally, compare counts taken at different places, and nest these constructions.

In slogan form:

> MP *is arithmetic modal logic;* FO + C *is full first-order logic with arithmetic and counting.*

Historic note. Counting logics became central in finite model theory around 1990; see for instance Immerman–Lander (1990), Tinhofer (1991), and the Cai–Fürer–Immerman separation result (1991).

Example Going Beyond Presburger Modal Logic

Comparing counts at different centers

$$\exists y \Big(E(x,y) \ \wedge \ \#z\,[E(y,z) \wedge p(z)] = \#z\,[E(x,z) \wedge q(z)] \Big)$$

This example already shows the extra power quite clearly: we quantify over a neighbor y and compare a count around y with a different count around x.

FO + C as an upper bound

The connection between GNNs and FO + C can be stated in several ways. The result that is easiest to state is:

Theorem (Grohe 2023)

GNN with ReLU activations and rational weights are epressively contained in (a fragment of) FO + C.

More about FO + C

The reason FO + C is so important in descriptive complexity is the following classical theorem.

Barrington–Immerman–Straubing (1990)

Over ordered finite structures:

- uniform version: a language is in dlogtime-uniform TC^0 iff it is definable in FO + C;
- nonuniform version: a language is in nonuniform TC^0 iff it is definable in FO + C with suitable built-in relations.

So when FO + C appears as a logical upper bound for GNN-like computation, it is not just a random logic: it is the logic classically associated with threshold-circuit computation.

Interlude: Descriptive-Complexity

Descriptive complexity asks:

> *which computational complexity classes correspond to which logical languages?*

Instead of presenting an algorithm for a graph problem, we present a formula that defines the property.

Typical examples:

- existential second order logic captures NP (Fagin)
- least fixed-point logic captures polynomial time on ordered structures (Immerman-Vardi)
- counting logics capture circuit classes with counting power.

This situates GNNs between logic, circuits, and learnable architectures.

What It Means for a Logic to Capture a Complexity Class

Let $\mathcal{L}$ be a logic and $\mathcal{C}$ a complexity class.

Capturing

We say that $\mathcal{L}$ captures $\mathcal{C}$ over a class of finite structures if:

- every property definable in $\mathcal{L}$ is decidable in $\mathcal{C}$;
- every property decidable in $\mathcal{C}$ is definable in $\mathcal{L}$.

In practice, one usually needs an ordering or other built-in relations on the input structures to talk about *uniform* computation.

So a capture theorem says:

> *this logic is neither too weak nor too strong for a certain computational resource bound.*

Why TC^0

TC^0 is the class of Boolean problems solvable by polynomial-size, bounded-depth threshold circuits.

This is the natural neighborhood for GNNs because:

- threshold gates already resemble neurons with threshold-like activations;
- bounded depth matches the fixed number of layers in a constant-depth architecture;
- counting and arithmetic behavior become central.

Historic note. The circuit class TC^0 was introduced in the 1980s. A classical neural-network connection is due to Maass (1997), who characterized TC^0 via bounded-depth polynomial-size feedforward networks with suitable activations.

Takeaway

We saw that different variants of GNNs connect to different counting logics (GML, MP, FO + C). These are just a sample of recent results.

Similar connections have been established with other counting logics, such as finite variable fragments of FO with counting quantifiers, and graded hybrid logics.

Over the last few years, GNNs have flourished into a lively area where logicians and AI researchers meet and interact.

See for example `https://pwalega.github.io/logic-gnn-2026/`