This is a comprehensive, step-by-step tutorial designed for a graduate-level Graph ML course. It creates a bridge between formal mathematical definitions and their direct implementation in **PyTorch Geometric (PyG)**.

You can structure this content into a single comprehensive Jupyter Notebook or split it into two separate notebooks (one for foundations, one for advanced pipelines).

Graduate Tutorial: Heterogeneous & Knowledge Graph Neural Networks

Objective: Understand the mathematical generalization of GNNs to heterogeneous structures and implement them for Node Classification and Link Prediction.

Part 1: Theoretical Foundations

1.1 From Homogeneous to Heterogeneous

In a standard (homogeneous) graph, we have a single graph $G = (V, E)$. In a **Heterogeneous Graph**, we introduce types.

Mathematical Definition: A heterogeneous graph is defined as $G = (V, E, \mathcal{A}, \mathcal{R})$ where:

- V: Set of nodes.
- E: Set of edges.
- $\mathcal{A}$: Set of node types. Each node $v \in V$ has a type mapping $\phi(v) \in \mathcal{A}$.
- $\mathcal{R}$: Set of relation types. Each edge $e \in E$ has a type mapping $\psi(e) \in \mathcal{R}$.

1.2 The Message Passing Challenge

In a standard GCN, the update rule is:

$$h_v^{(l+1)} = \sigma\left(\sum_{u \in \mathcal{N}(v)} W^{(l)} h_u^{(l)}\right)$$

This fails in heterogeneous graphs because a "User" node feature vector might have dimension $d = 16$, while a "Movie" node has dimension $d = 32$. We cannot multiply them by a single matrix W.

The Heterogeneous Update Rule (RGCN Style): To solve this, we use a specific weight matrix W_r for each relation type r.

$$h_v^{(l+1)} = \sigma\left(\sum_{r \in \mathcal{R}} \sum_{u \in \mathcal{N}_r(v)} \frac{1}{c_{v,r}} W_r^{(l)} h_u^{(l)} + W_0^{(l)} h_v^{(l)}\right)$$

- $\mathcal{N}_r(v)$: Neighbors of v under relation r.
- $W_r^{(l)}$: Learnable weight matrix specific to relation r (transforms neighbor features).
- $W_0^{(l)}$: Self-loop weight (preserves node's own features).

Part 2: Implementation - Data Structures

We will use `HeteroData` in PyG. This is a dictionary-like object where keys are tuples representing node or edge types.

Step 2.1: Constructing a Synthetic Hetero Graph

We will build a graph representing a simplified **Academic Network**:

- **Nodes:** `Paper` ($d = 128$), `Author` ($d = 64$)
- **Edges:** `Author` $\rightarrow$ `writes` $\rightarrow$ `Paper`

```
import torch
from torch_geometric.data import HeteroData

# Initialize container
data = HeteroData()

# --- 1. Define Node Features ---
# Let's say we have 100 Authors and 200 Papers.
num_authors = 100
num_papers = 200

# Author features: [100, 64]
data['author'].x = torch.randn(num_authors, 64)
# Paper features: [200, 128]
data['paper'].x = torch.randn(num_papers, 128)

# --- 2. Define Edge Connectivity (COO Format) ---
# Edge type: ('author', 'writes', 'paper')
# We randomly create 500 connections
author_ids = torch.randint(0, num_authors, (500,))
paper_ids = torch.randint(0, num_papers, (500,))

# edge_index shape must be [2, num_edges]
data['author', 'writes', 'paper'].edge_index =
torch.stack([author_ids, paper_ids], dim=0)

print("Original Data Structure:")
print(data)
```

Step 2.2: The Necessity of Undirected Graphs

Concept: In PyTorch Geometric, message passing follows the direction of edges. If we only have `Author -> writes -> Paper`, a GNN on the `Author` nodes will update based on `Paper` info, but `Paper` nodes will **not** learn anything from `Authors` (no incoming edges).

To allow information flow in both directions, we must add **Reverse Edges**.

```
import torch_geometric.transforms as T
```

```
# T.ToUndirected() automatically finds edge types and creates
their reverse
# e.g., ('paper', 'rev_writes', 'author')
data = T.ToUndirected()(data)

print("\nAfter Adding Reverse Edges:")
print(data.edge_types)
# You should see: [('author', 'writes', 'paper'), ('paper',
'rev_writes', 'author')]
```

Part 3: Node Classification with `to_hetero`

Task: Predict the "Research Topic" of a paper (Multiclass Classification).

Concept: Instead of writing the complex math of RGCN manually (looping over dictionaries), PyG provides a compiler called `to_hetero`. It takes a standard GNN (like GraphSAGE or GAT) and **duplicates it** for every relation type in the graph.

Step 3.1: Loading a Real Dataset

We use the **DBLP** dataset (Authors, Papers, Terms, Conferences).

```
from torch_geometric.datasets import DBLP
import torch.nn.functional as F
from torch_geometric.nn import SAGEConv, to_hetero

# Load dataset
dataset = DBLP(root='./data/DBLP')
data = dataset[0]

print(f"Target: Classify 'author' nodes into {dataset.num_classes}
classes.")
print(f"Node Types: {data.node_types}")
```

Step 3.2: Model Architecture

Mathematical Logic of `to_hetero`: If you define a `SAGEConv(in, out)`, `to_hetero` converts it into a dictionary of convolutions:

1. `SAGEConv_writes(feature_author, feature_paper)`
2. `SAGEConv_cites(feature_paper, feature_paper)` ... and then sums the results for the target node.

```
class HeteroGNN(torch.nn.Module):
    def __init__(self, hidden_channels, out_channels, num_layers):
        super().__init__()
        self.convs = torch.nn.ModuleList()

        # Layer 1: Input to Hidden
        # SAGEConv((-1, -1)) allows handling different input
feature sizes
        # for source and target nodes automatically (Lazy
Initialization)
        self.convs.append(SAGEConv((-1, -1), hidden_channels))

        # Hidden Layers
        for _ in range(num_layers - 1):
            self.convs.append(SAGEConv((-1, -1), hidden_channels))

        # Post-processing linear layer (optional, but good for
classification)
        self.lin = torch.nn.Linear(hidden_channels, out_channels)

    def forward(self, x, edge_index):
        # Standard GNN flow
        for conv in self.convs:
            x = conv(x, edge_index).relu()
        return x

# 1. Initialize the Homogeneous Model
model = HeteroGNN(hidden_channels=64, out_channels=4,
num_layers=2)

# 2. Convert to Heterogeneous Model
# 'aggr="sum"' defines how we combine messages from different
```

```
relation types
# (e.g., messages from Papers + messages from Conferences)
model = to_hetero(model, data.metadata(), aggr='sum')

# Visualization of the generated architecture
print(model)
```

Step 3.3: Training Loop

Note how we pass `x_dict` and `edge_index_dict` instead of standard tensors.

```
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)

def train():
    model.train()
    optimizer.zero_grad()

    # Forward pass: returns a Dictionary of embeddings {node_type:
embedding}
    out_dict = model(data.x_dict, data.edge_index_dict)

    # We only compute loss on 'author' nodes with ground truth
labels
    mask = data['author'].train_mask
    pred = out_dict['author'][mask]
    target = data['author'].y[mask]

    loss = F.cross_entropy(pred, target)
    loss.backward()
    optimizer.step()
    return loss.item()

# Run training
for epoch in range(51):
    loss = train()
    if epoch % 10 == 0:
        print(f"Epoch {epoch:03d}: Loss {loss:.4f}")
```

Part 4: Link Prediction (Knowledge Graph Focus)

Task: Predict missing citations (`Author` -> `Paper`).

Math: Encoder-Decoder Architecture

1. **Encoder:** A GNN produces embeddings Z.
2. **Decoder:** A scoring function $s(u, v)$.
   - For simple links: Dot product $s(u, v) = z_u^\top z_v$.
   - For Knowledge Graphs (TransE): $s(h, r, t) = -||z_h + z_r - z_t||$.

We will use a **Dot Product Decoder** here, which is standard for recommender-style link prediction.

Step 4.1: Rigorous Data Splitting

We cannot just mask nodes. We must hide *edges*.

- **Message Passing Edges:** Used to calculate embeddings.
- **Supervision Edges:** Used to calculate Loss (Positive samples).
- **Negative Edges:** Random pairs that do not exist (Negative samples).

PyG's `RandomLinkSplit` handles this complex logic.

```
# We want to predict the ('author', 'to', 'paper') relation
transform = T.RandomLinkSplit(
    num_val=0.1,
    num_test=0.1,
    disjoint_train_ratio=0.3,  # 30% of train edges used for
supervision, 70% for message passing
    neg_sampling_ratio=2.0,    # 2 negative edges for every 1
positive edge
    edge_types=[('author', 'to', 'paper')], # The relation to
predict
    rev_edge_types=[('paper', 'to', 'author')] # The reverse
relation to update
)

train_data, val_data, test_data = transform(data)
```

```
# Inspect the supervision edges
print("Supervision Labels:", train_data['author', 'to',
'paper'].edge_label.shape)
print("Supervision Indices:", train_data['author', 'to',
'paper'].edge_label_index.shape)
```

Step 4.2: Defining the Link Predictor

This replaces the standard classifier head.

```
class LinkPredictor(torch.nn.Module):
    def __init__(self):
        super().__init__()

    def forward(self, z_source, z_target, edge_label_index):
        # z_source: Embeddings of all source nodes (e.g., authors)
        # z_target: Embeddings of all target nodes (e.g., papers)
        # edge_label_index: [2, num_pairs] indices to score

        row, col = edge_label_index

        # Fetch the specific embeddings for the pairs we are
evaluating
        feat_src = z_source[row]
        feat_dst = z_target[col]

        # Dot Product score
        return (feat_src * feat_dst).sum(dim=-1)
```

Step 4.3: The Hetero Link Prediction Loop

```
# Re-initialize model (Encoder)
model = HeteroGNN(hidden_channels=64, out_channels=64,
num_layers=2)
model = to_hetero(model, data.metadata(), aggr='sum')

predictor = LinkPredictor()
optimizer = torch.optim.Adam(list(model.parameters()) +
list(predictor.parameters()), lr=0.01)
```

```
def train_link_prediction(split_data):
    model.train()
    optimizer.zero_grad()

    # 1. Encode: Get embeddings for ALL nodes using Message
Passing edges
    # Note: split_data.edge_index_dict contains ONLY message
passing edges
    # (The supervision edges were hidden by RandomLinkSplit)
    node_embeddings = model(split_data.x_dict,
split_data.edge_index_dict)

    # 2. Decode: Score the specific supervision pairs
    # These indices include both POSITIVE edges (real) and
NEGATIVE edges (sampled)
    edge_label_index = split_data['author', 'to',
'paper'].edge_label_index
    edge_label = split_data['author', 'to', 'paper'].edge_label #
1.0 or 0.0

    # Get scores
    preds = predictor(
        node_embeddings['author'],
        node_embeddings['paper'],
        edge_label_index
    )

    # 3. Loss (BCEWithLogits combines Sigmoid + BCE for stability)
    loss = F.binary_cross_entropy_with_logits(preds, edge_label)
    loss.backward()
    optimizer.step()
    return loss.item()

print("Training Link Prediction...")
for epoch in range(1, 51):
    loss = train_link_prediction(train_data)
    if epoch % 10 == 0:
        print(f"Epoch {epoch:03d}: Loss {loss:.4f}")
```

- **The Core Problem: Data Leakage (Cheating)**

  In standard Machine Learning (like Image Classification), your data points (images) are independent. You can split images into Train/Test easily.

  In Graphs, data points (nodes) are **interconnected**.

  - **Scenario:** You want to predict if **Alice** is friends with **Bob**.
  - **The Mistake:** You leave the edge `(Alice, Bob)` in the graph while training.
  - **The GNN's Action:** When the GNN computes the embedding for **Alice**, it aggregates information from her neighbors. If **Bob** is in her neighbor list, **Alice's embedding will contain Bob's information directly.**
  - **The Result:** The model doesn't learn to *predict* friendship based on shared interests; it simply learns to look up if the connection already exists in the input. It memorizes the graph.

  Solution: We must **physically remove** the specific edge we want to predict from the graph structure used for Message Passing. We "hide" it.

- **The Three Sets of Edges**

  To train a Link Predictor properly, we actually manage three distinct sets of edges simultaneously during a single training step.

  A. Message Passing Edges (The "Context")

  - **What are they?** The edges that remain in the graph structure (`edge_index`).
  - **Purpose:** They allow the GNN to pass messages and calculate node embeddings.
  - **Analogy:** These are the "clues" available to solve the mystery.
  - **Math:** These define the neighborhood $\mathcal{N}(v)$ in the GNN equation:

$$h_v = \sigma\left(\sum_{u \in \mathcal{N}_{MP}(v)} W h_u\right)$$

B. Supervision Edges (The "Positive" Targets)

- **What are they?** The real edges we removed/hid from the Message Passing set.
- **Label:** $y = 1$ (True connection).
- **Purpose:** These are the "questions" on the exam. We take the embeddings generated using the MP edges and ask: *"Based on the context, should these two nodes be connected?"*
- **Math:** We maximize the score $s(u, v)$ for these pairs.

C. Negative Edges (The "Negative" Targets)

- **What are they?** Randomly sampled pairs of nodes (u, v_{rand}) that **do not** have an edge in the original graph.
- **Label:** $y = 0$ (False connection).
- **Purpose:** To prevent the "Trivial All-One Solution."
- **Math:** We minimize the score $s(u, v_{rand})$ for these pairs.

- **Why do we need Negative Edges?**

  Imagine we only trained on Positive Edges (Supervision Edges).

  - **The Task:** "Maximize the similarity score for all these pairs."
  - **The Model's Cheat:** "Okay, I will just make **every** embedding identical, or set all weights to infinity."
  - **Result:** The model predicts "Yes" (100% probability) for *every possible pair of nodes in the universe*.

  To force the model to discriminate, we must teach it what a **bad** link looks like.

  - We tell the model: "Alice and Bob are connected (Score ↑), but Alice and **Random_Stranger** are NOT connected (Score ↓)."
  - This forces the embeddings to arrange themselves geometrically where friends are close, and non-friends are far apart.

- **Step-by-Step Visualization of a Single Training Step**

  Let's trace the data flow for a single edge between **User A** and **Item B**.

  1. **Splitting:**
     - We select `(A, B)` to be a **Supervision Edge**.
     - We **delete** `(A, B)` from the graph structure.
     - We randomly select `(A, C)` as a **Negative Edge** (assuming A never bought C).
  2. **Forward Pass (Message Passing):**
     - The GNN looks at **User A**. It sees their other purchases (Item D, Item E) but **NOT** Item B.
     - It calculates `Embedding_A` based on D and E.
     - It calculates `Embedding_B` and `Embedding_C`.
  3. **Prediction (Decoding):**
     - It calculates Score 1: `DotProduct(Embedding_A, Embedding_B)`. Target: **1**.
     - It calculates Score 2: `DotProduct(Embedding_A, Embedding_C)`. Target: **0**.
  4. **Loss Calculation:**
     - It compares Score 1 to 1.0 and Score 2 to 0.0.
     - It backpropagates the error to update the weights.

Summary Table

Edge Type	In `edge_index`?	Used for Convolution?	Label (y)	Goal
Message Passing	Yes	**Yes**	N/A	Build Node Embeddings
Supervision (Positives)	**No** (Hidden)	No	**1**	Teach model to recognize real links
Negative Samples	**No** (Never existed)	No	**0**	Teach model to recognize fake links

---

Knowledge Graph Completion & PyKEEN

- **Theoretical Deep Dive: The Geometry of Knowledge**

  In standard Graph ML (like Part 3), we often used a **Dot Product** decoder: $Score = z_u \cdot z_v$. This measures "similarity."

  However, in Knowledge Graphs, relations have **semantics**.

  - *Symmetric:* `(Alice, is_married_to, Bob)` $\iff$ `(Bob, is_married_to, Alice)`
  - *Anti-symmetric:* `(Alice, mother_of, Bob)` $\neq$ `(Bob, mother_of, Alice)`
  - *Inversion:* `(Alice, buys, Item)` $\iff$ `(Item, bought_by, Alice)`

  A simple dot product cannot capture these nuances. We need **Geometric Scoring Functions**.

  A. TransE (Translational Embedding)

  Inspired by word2vec ($King - Man + Woman \approx Queen$).

  - **Intuition:** If (h, r, t) holds, the embedding of the tail t should be close to the embedding of the head h plus the relation vector r.
  - **Equation:** $h + r \approx t$
  - **Scoring Function:** $f(h, r, t) = -||\mathbf{h} + \mathbf{r} - \mathbf{t}||$ (L1 or L2 norm).
  - **Goal:** Minimize distance for true triples, maximize it for false ones.

  B. DistMult (Factorization)

  - **Intuition:** Each relation r is a diagonal matrix M_r that scales the dimensions of the interaction.
  - **Equation:** $f(h, r, t) = \mathbf{h}^\top \mathbf{M}_r \mathbf{t}$
  - **Limitation:** Since $\mathbf{h}^\top \mathbf{M}_r \mathbf{t} = \mathbf{t}^\top \mathbf{M}_r \mathbf{h}$, DistMult is strictly symmetric. It cannot model "father_of".

  C. RotatE (Rotation in Complex Space)

  - **Intuition:** Models relations as **rotations** in the complex plane.
  - **Equation:** $t = h \circ r$ (Element-wise Hadamard product in complex space).
  - **Power:** Can model symmetry, anti-symmetry, inversion, and composition.

- **The Training Objective: Ranking Loss**

  In KGs, we don't just classify "Yes/No." We want the true answer to be **ranked** higher than false answers.

  Margin Ranking Loss:

$$\mathcal{L} = \sum_{(h,r,t) \in \mathcal{D}^+} \sum_{(h',r,t') \in \mathcal{D}^-} \max(0, \gamma + f(h', r, t') - f(h, r, t))$$

  - $\mathcal{D}^+$: Real triples.
  - $\mathcal{D}^-$: Corrupted triples (e.g., replace Head with random entity).
  - γ: Margin. We want the Positive score to be higher than the Negative score by at least γ.

- **PyKEEN Tutorial (The "Scikit-Learn" of KGs)**

  While PyG is excellent for **GNNs** (passing messages using features), **PyKEEN** (Python Knowledge Embeddings) is the industry standard for **Shallow Embeddings** (TransE, RotatE) where nodes don't necessarily have features, just identities.

  Installation:

```
pip install pykeen
```

  Step 3.1: The High-Level Pipeline

  We will train a **TransE** model on the **Nations** dataset (a small graph of political interactions).

```
from pykeen.pipeline import pipeline
from pykeen.datasets import Nations

# 1. Run the pipeline
```

```
# This handles: Data splitting, Negative Sampling, Training Loop,
and Evaluation
result = pipeline(
    dataset='Nations',        # Built-in dataset
    model='TransE',           # The model architecture
    training_loop='slcwa',    # SLCWA = Stochastic Local Closed
World Assumption (Negative Sampling)

    # Hyperparameters
    model_kwargs={'embedding_dim': 50},
    training_kwargs={'num_epochs': 100, 'batch_size': 32},
    evaluator_kwargs={'filtered': True}, # Filter out known
positives during eval
    random_seed=42,
)

# 2. Check Results
print(f"Mean Reciprocal Rank (MRR):
{result.get_metric('mrr'):.4f}")
print(f"Hits@10: {result.get_metric('hits@10'):.4f}")

# Visualize the loss curve
result.plot_losses()
```

Step 3.2: Making Predictions (Inference)

Once trained, how do we answer questions like *"Which country does Brazil have a conference with?"*?

```
import torch
from pykeen.models import Model

# Get the trained model
model = result.model

# Get the entity and relation mappings (ID to String)
entity_to_id = result.training.entity_to_id
relation_to_id = result.training.relation_to_id

# Helper function to predict tails
def predict_tail(head_name, relation_name):
```

```
    # 1. Convert names to IDs
    h_id = torch.as_tensor([entity_to_id[head_name]])
    r_id = torch.as_tensor([relation_to_id[relation_name]])

    # 2. Predict scores for ALL possible tails against this (h, r)
    # predict_t scores (h, r, t) for all t
    scores = model.predict_t(h_id, r_id)

    # 3. Rank them (higher score = more likely)
    # Note: TransE uses distance, so PyKEEN automatically converts
this
    # so that larger values are better for consistency.
    top_k = torch.topk(scores, k=5)

    # 4. Decode back to names
    id_to_entity = {v: k for k, v in entity_to_id.items()}

    print(f"Query: ({head_name}, {relation_name}, ?)")
    for score, t_id in zip(top_k.values[0], top_k.indices[0]):
        print(f"  -> {id_to_entity[t_id.item()]} (Score:
{score:.4f})")

# Run a query (check available names in entity_to_id.keys() if
'brazil' is missing)
# Example from Nations dataset:
predict_tail('brazil', 'conferences')
```

- **Advanced: Hybridizing PyG and PyKEEN**

  In a graduate class, you should explain the distinction:

  1. **PyKEEN (Shallow):** Learns a lookup table $E \in \mathbb{R}^{|V| \times d}$. Good when graph structure is the only signal.
  2. **PyG (Deep):** Learns a function $f(X, A)$. Good when nodes have rich features (text, images) and we want to be inductive (handle new nodes).

The Hybrid Approach (Encoder-Decoder): You can use a PyG GNN as the **Encoder** to generate node embeddings, and then use a PyKEEN scoring function (like TransE or DistMult) as the **Decoder** or **Loss Function**.

Conceptual Code Snippet:

```
# PyG Encoder
class GNNEncoder(torch.nn.Module):
    def forward(self, x, edge_index):
        return self.sage_conv(x, edge_index)

# Hybrid Model
class HybridModel(torch.nn.Module):
    def __init__(self):
        self.encoder = GNNEncoder(...)
        self.decoder = pykeen.nn.DistMultInteraction() # Used
purely for scoring

    def forward(self, x, edge_index, h_idx, r_idx, t_idx):
        # 1. Get contextualized embeddings via GNN
        node_emb = self.encoder(x, edge_index)

        # 2. Select embeddings for the triples batch
        h = node_emb[h_idx]
        t = node_emb[t_idx]
        r = self.relation_embeddings[r_idx] # Relations usually
don't have GNN features

        # 3. Score using KG geometry
        return self.decoder(h, r, t)
```

- Use **PyG** when you have **Node Features** or need to classify nodes.
- Use **PyKEEN** when you have a pure **Knowledge Graph** (Subject-Predicate-Object) and need to predict missing links based on structure.
- **Link Prediction** requires negative sampling and ranking metrics (MRR, Hits@K), not just accuracy.