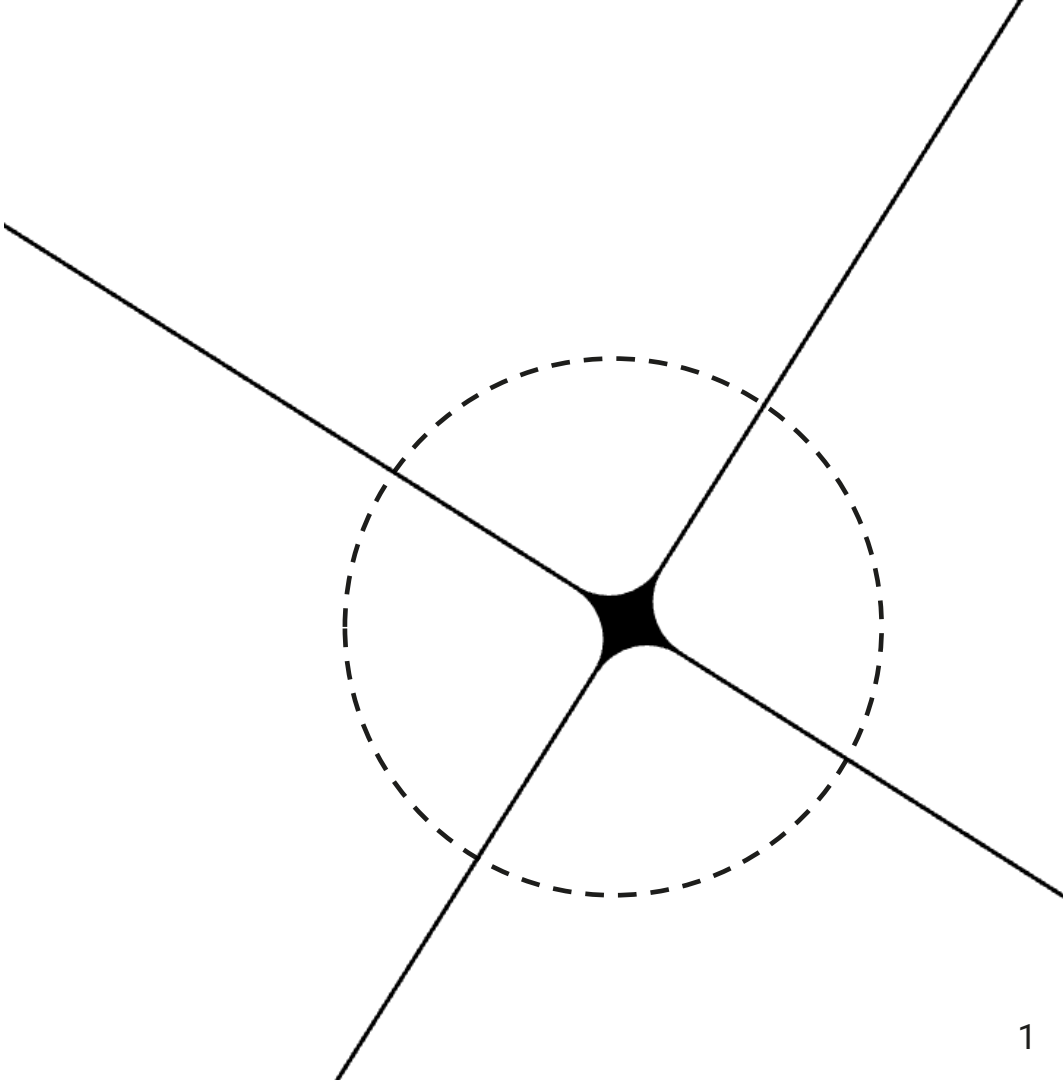
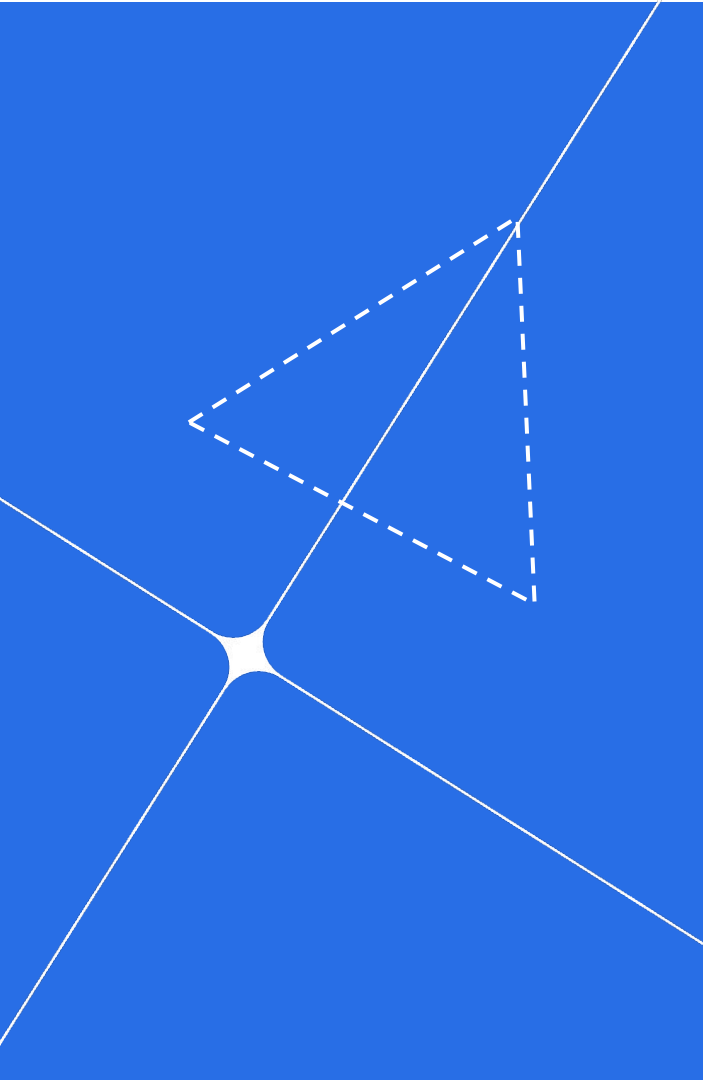
INSTITUT POLYTECHNIQUE DE PARIS

TELECOM SudParis

Graph Neural Networks

Julien Romero

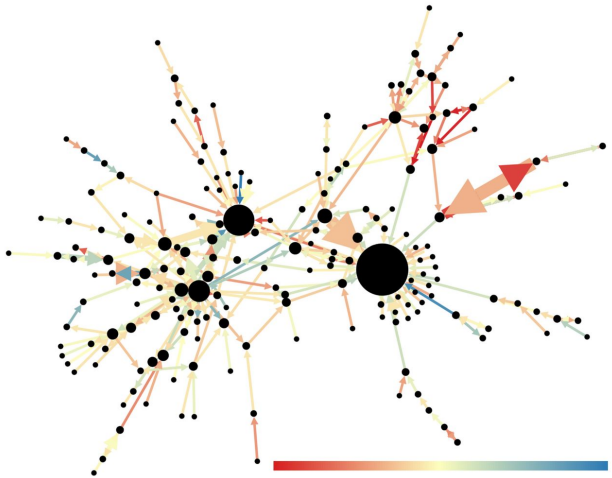




Motivation : pourquoi des graphes ?

Pourquoi modéliser en graphe ?

- Beaucoup de données “réelles” = **entités + relations** (pas juste des lignes indépendantes)
- Exemples typiques :
  - comptes ↔ transactions, utilisateurs ↔ items, services ↔ appels réseau, molécules (atomes ↔ liaisons)
- Les liens portent de l’info : **qui interagit avec qui**, **dans quel contexte**, **à quelle fréquence**
- Objectif : apprendre des représentations (embeddings) qui exploitent **la structure + les features**
- Deux tâches phares (dans ce cours) : **node classification + link prediction**
- Enjeu industrie : précision **et** passage à l’échelle (latence, mémoire, mises à jour)

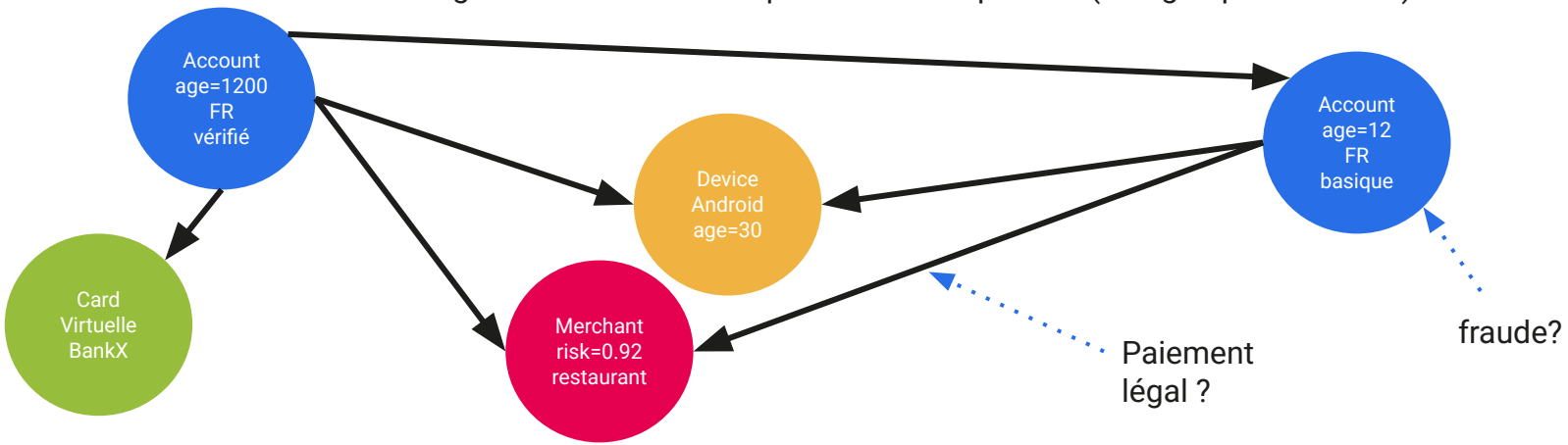


Le “biais relationnel” : ce que les modèles classiques perdent

- Hypothèse classique : données **IID** (Independent and Identically Distributed) → souvent fausse en pratique
- En graphe : les labels dépendent du voisinage (**dépendances**, contagion, homophilie/structure)
- Aplatissement tabulaire :
  - force du **feature engineering** (comptage de voisins, motifs, chemins)
  - mais perd la flexibilité (nouveaux liens, nouveaux nœuds, graphes dynamiques)
- **GNN (Graph Neural Network)** : apprend à combiner features locales + structure via **message passing**
- Propriété clé : indépendance à l’ordre des nœuds (**permutation invariance/equivariance**)
- Résultat attendu : embeddings utiles pour prédire, détecter, recommander, compléter un graphe

Fil rouge industrie : fraude sur graphe de transactions

- Nœuds : **Compte**, **Carte**, **Device**, **Marchand**, **Transaction** (graph potentiellement hétérogène)
- Arêtes : “utilise”, “paye”, “appartient”, “partage device”, “transfère vers”
- Fraude = motifs **multi-sauts** (ex : compte → device partagé → autre compte → marchand à risque)
- Deux formulations directes :
  - **Node classification** : “ce compte est-il frauduleux ?”
  - **Link prediction** : “ce lien/transaction est-il suspect(e) ?”
- Contraintes production : forte **rareté** (class imbalance), **évolution temporelle** (drift), besoin de **règles + modèle**
- Valeur métier : réduire faux négatifs coûteux **sans** exploser les faux positifs (charge opérationnelle)

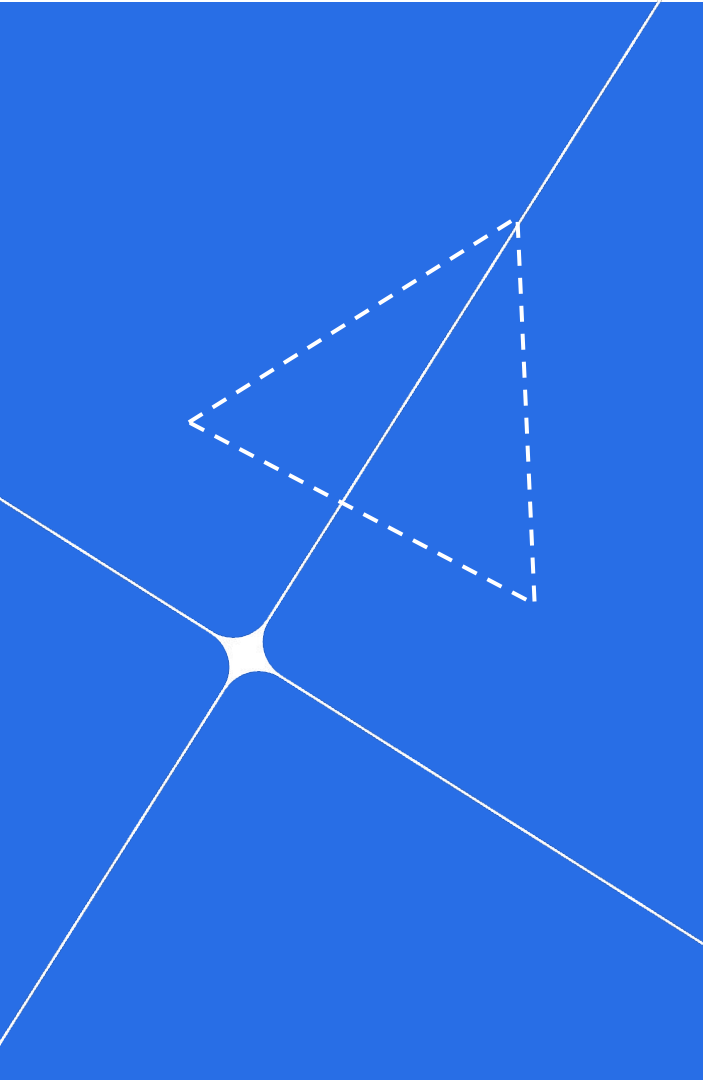


Pourquoi pas juste un modèle tabulaire “classique” ?

- Pipeline tabulaire typique : agrégations → features de voisinage → modèle (XGBoost/MLP)
- Limites récurrentes :
  - features de voisinage **rigides** (1-hop, 2-hop) et difficiles à maintenir
  - motifs complexes (chemins, cycles) → explosion combinatoire
  - “cold start” : nouveaux nœuds/liens → features pauvres
- GNN : apprend une représentation **end-to-end** (structure + features) *sans* coder chaque agrégat à la main
- En pratique industrie : GNN souvent **complémentaire** (stack) :
  - embeddings GNN → features pour un modèle final (XGBoost / MLP / règles)
- Message clé : on n’abandonne pas les baselines ; on les **dépasse** quand le signal relationnel est fort

Quand un GNN est (ou n'est pas) un bon choix

- Bon candidat si :
  - relations riches, **multi-hop** informatif, dépendances entre exemples
  - données incomplètes : la structure "compense" des features manquantes
  - besoin de généraliser à de nouveaux nœuds (setting **inductif**)
- Mauvais candidat si :
  - relations quasi aléatoires / bruitées, graphe trop clairsemé (sparse extrême)
  - le problème est déjà résolu par un baseline simple (coût/complexité injustifiés)
  - contraintes strictes sans possibilité de sampling / caching (latence, mémoire)
- Règle d'ingénierie : commencer par **baseline** (tabulaire) puis justifier le graphe par gain mesuré
- Ce qu'on va faire ensuite : formaliser "comment on apprend sur un graphe" (spectral + message passing)



Modéliser un problème en graphe

Du monde réel à G = (V, E)

- Graphe G=(V,E) : **nœuds** V, **arêtes** E (orientées ou non)
- Features :
  - nœud : $x_v \in R^d$ (ex : âge compte, pays, embedding texte)
  - arête : e_{uv} (ex : montant, timestamp, type de relation)
- Labels possibles :
  - nœud : y_v (fraude, catégorie, communauté)
  - arête : y_{uv} (lien valide/suspect)
  - graphe : y_G (propriété d'une molécule, type de document)
- Choix de modélisation = impact direct sur la tâche et l'évaluation
- En industrie : le graphe est souvent **partiel**, bruité, et **évolue**

Homogène vs hétérogène

- **Graphe homogène** : 1 type de nœud, 1 type d'arête (ex : citations entre papiers)
- **Graphe hétérogène** (*heterogeneous graph*) :
  - plusieurs types de nœuds : Account, Device, Merchant, Transaction…
  - plusieurs relations : uses_device, paid, transfers_to, owns_card…
- Pourquoi c'est fréquent en industrie : données multi-tables (SQL) + clés étrangères → naturellement multi-types
- 2 stratégies courantes :
  - projeter en homogène (perte d'info / simplification)
  - garder l'hétéro (meilleure fidélité / plus complexe)
- Dans ce cours : on **travaille surtout homogène**, hétéro = **positionnement** + intuition
- Terme utile : *metapath* (chemin typé)

Représentations de données

- Edge list : liste (u,v) + attributs (simple, scalable, proche base de données)
- Matrice d'adjacence $A \in \{0,1\}^{|V| \times |V|}$:
  - utile pour la théorie, mais trop grosse en pratique (ou alors représentations spéciales)
- En deep learning graphe : stockage **sparse** (COO/CSR) :
  - indices des arêtes + features (compatible GPU si bien fait)
- "Message passing" = itérer sur les arêtes : coût typique $O(|E|)$
- Attention : orientation (directed) vs non-orienté (souvent symétrisé)
- Pré-processing minimal : self-loops, normalisation, features manquantes

Panorama des tâches

- **Node classification** : prédire une classe pour chaque nœud
  - ex : “compte frauduleux ?”, “topic d’un papier ?”
- **Link prediction** : prédire existence/type d’un lien
  - ex : “transaction suspecte ?”, “ami probable ?”, “achat probable ?”
- **Edge classification** : classer une arête observée (lien déjà là)
  - ex : “transaction = fraude / legit”
- **Graph classification / regression** : label au niveau graphe
  - ex : molécule → toxicité, logP, efficacité
- **Node/graph clustering** : structure latente (communautés)
- Intuition : GNN apprend des embeddings ; la tête de sortie change selon la tâche

Ce qu'on observe (et ce qu'on doit prédire)

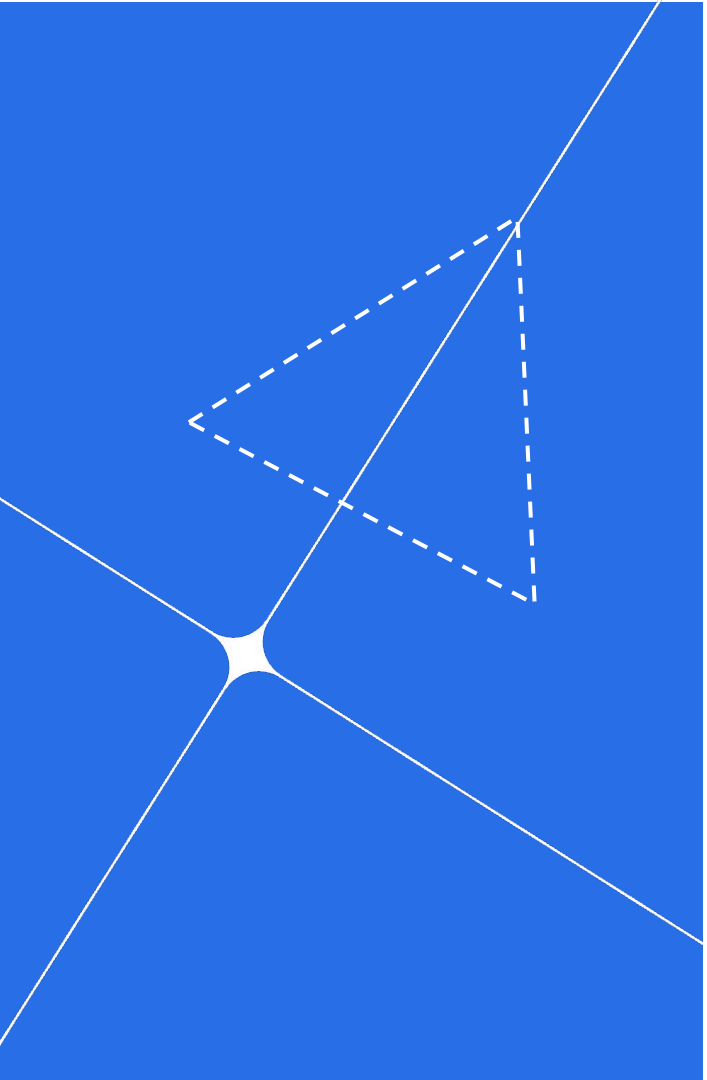
- Jeu de données graphe = souvent un **snapshot** (ou plusieurs snapshots)
- Ensembles usuels :
  - features X, adjacency E, labels partiels Y (semi-supervisé fréquent)
- En node classification : labels disponibles sur une fraction des nœuds
- En link prediction :
  - on masque un sous-ensemble d'arêtes (positives)
  - on crée des **négatives** artificiels (liens inexistants) via *negative sampling*
- Danger : *data leakage* (fuites) si on utilise une arête "future" pour prédire le passé
- Bon réflexe : définir ce qui est "connu au moment T" (industrie)

Mini-exemple “prêt ML” : node classification sur graphe homogène

- Graphe : nœuds = comptes, arêtes = transferts (directed)
- Features nœuds : stats (solde, âge), pays, device_count, merchant_risk_mean…
- Label : fraud ∈ {0,1} sur un sous-ensemble de comptes investigués
- Split :
  - train/val/test sur **nœuds**, en gardant un graphe commun (cas classique)
  - ou split temporel (plus réaliste, mais hors focus)
- Baseline : MLP sur features seules (sans structure) → référence de gain
- Objectif : montrer la valeur du voisinage (2-hop) sans feature engineering lourd

Outils

- **PyTorch Geometric (PyG)** : objets Data / HeteroData, edge_index, mini-batch loaders
- Datasets standards pour se calibrer : OGB (Open Graph Benchmark) *optionnel*
- Graphe réel = souvent construit depuis SQL :
  - tables entités + tables relations → edge list
- GPU : accélère l'agrégation si les données sont bien "batchées"
- Point clé : les GNN ne sont pas "magiques" : qualité du graphe + features = déterminant



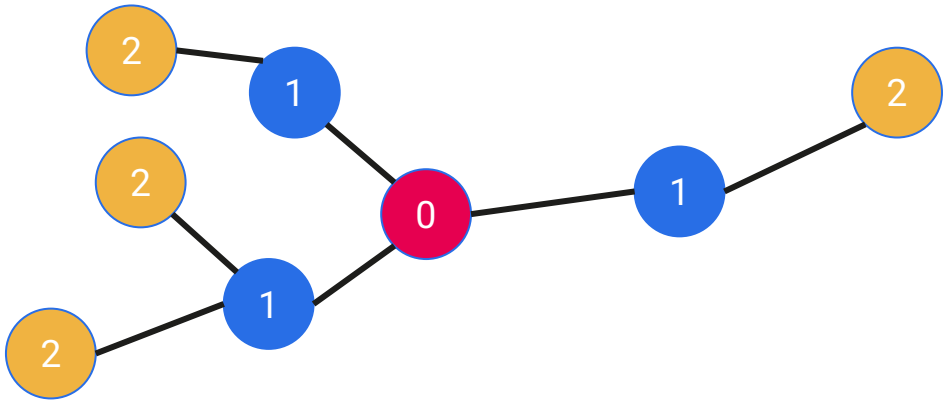
Fondations : contraintes du graphe & Message Passing

Propriété clé : l'ordre des nœuds ne doit pas compter

- Un graphe n'a **pas d'ordre canonique** des nœuds (renommer les IDs ne change rien)
- Donc un modèle doit être :
  - **Permutation equivariant** au niveau nœud : réordonner les nœuds réordonne les sorties
  - **Permutation invariant** au niveau graphe : sortie globale identique malgré permutation
- Conséquence : pas de "convolution 2D" naïve (pas de grille régulière)
- Les opérations valides : agrégations symétriques sur ensembles (sum/mean/max)
- Même message : un GNN doit "respecter" la structure combinatoire
- C'est le cœur du design des GNN modernes

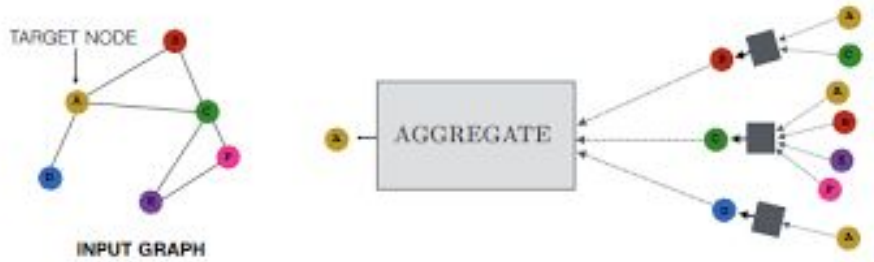
Localité & voisinage : le biais inductif des GNN

- On suppose que l'info pertinente est souvent **locale** (1-hop, 2-hop, …)
- Une couche GNN = "mélanger" l'info d'un nœud avec celle de ses voisins
- Empiler k couches ≈ exploiter un voisinage à k sauts (*k-hop receptive field*)
- Intuition fraude : signaux multi-sauts (device partagé, chaîne de transferts)
- Intuition recommandation : utilisateurs proches via items partagés (ou inverse)
- Compromis : plus profond = plus loin, mais risques (oversmoothing / coût / bruit)



Cadre unificateur : MPNN (Message Passing Neural Networks)

- **MPNN** = vue “spatiale” qui unifie GCN, GraphSAGE, GAT, …
- À chaque couche ℓ, on calcule des messages le long des arêtes
- Forme générique (nœud v) :
  - message : $m_{u\to v}^{(\ell)} = \phi^{(\ell)}(h_u^{(\ell)}, h_v^{(\ell)}, e_{uv})$ ($h_u^{(\ell)}$ est le vecteur représentant le nœud u après ℓ couches, message calculé à partir des états de la couche précédente et de l’arête)
  - agrégation : $M_v^{(\ell)} = \mathrm{AGG}_{u\in N(v)} m_{u\to v}^{(\ell)}$ (on agrège tous les messages venant des voisins)
  - update : $h_v^{(\ell+1)} = \psi^{(\ell)}(h_v^{(\ell)}, M_v^{(\ell)})$ (on met à jour en fonction de la couche précédente et de l’agrégation)
- $h_v^{(0)} = x_v$ (features initiales, parfois on apprend aussi un embedding initial), sortie = logits/embedding
- AGG doit être **symétrique** (sum/mean/max) pour respecter la permutation



“Pseudo-code” générique d’une couche MPNN

```
# h: [num_nodes, d], edge_index: [2, num_edges]
for (u, v) in edges:
    m[u, v] = phi(h[u], h[v], e[u, v])   # message
for v in nodes:
    M[v] = AGG( m[u, v] for u in N(v) )  # permutation-invariant
    h_new[v] = psi(h[v], M[v])           # update (MLP, GRU, ...)
return h_new
```

- En pratique : vectorisé + sparse ops (sinon trop lent)
- Variations : inclure self-loop, normaliser, residual connections
- Output : embeddings nœuds, puis tête (classification / score de lien)

Readout : passer de nœuds à un embedding “graphe”

- Pour **graph classification/regression**, il faut une sortie globale
- Readout invariant (exemples) :
  - $h_G = \sum_{v \in V} h_v$ (sum pooling)
  - $h_G = \text{mean}(h_v)$, max pooling, attention pooling
- Analogie : “bag of node embeddings” + pooling symétrique
- Point d’attention : graphes de tailles différentes → normalisation utile
- Même exigence de permutation invariance
- Dans ce cours : surtout node/link, mais ce pattern revient souvent

Limitation 1 : oversmoothing (intuition + symptôme)

- Empiler des couches mélange fortement les voisins
- Risque : embeddings deviennent **trop similaires** (perte de séparabilité)
- Symptômes : performance qui plafonne puis chute avec la profondeur
- Causes (intuition) : propagation + normalisation ≈ diffusion sur graphe
- Parades courantes :
  - residual/skip connections, normalization, dropout, Jumping Knowledge
  - limiter la profondeur, enrichir features, utiliser attention/sampling
- Message : profondeur ≠ toujours mieux sur graphe

Limitation 2 : oversquashing (intuition “information bottleneck”)

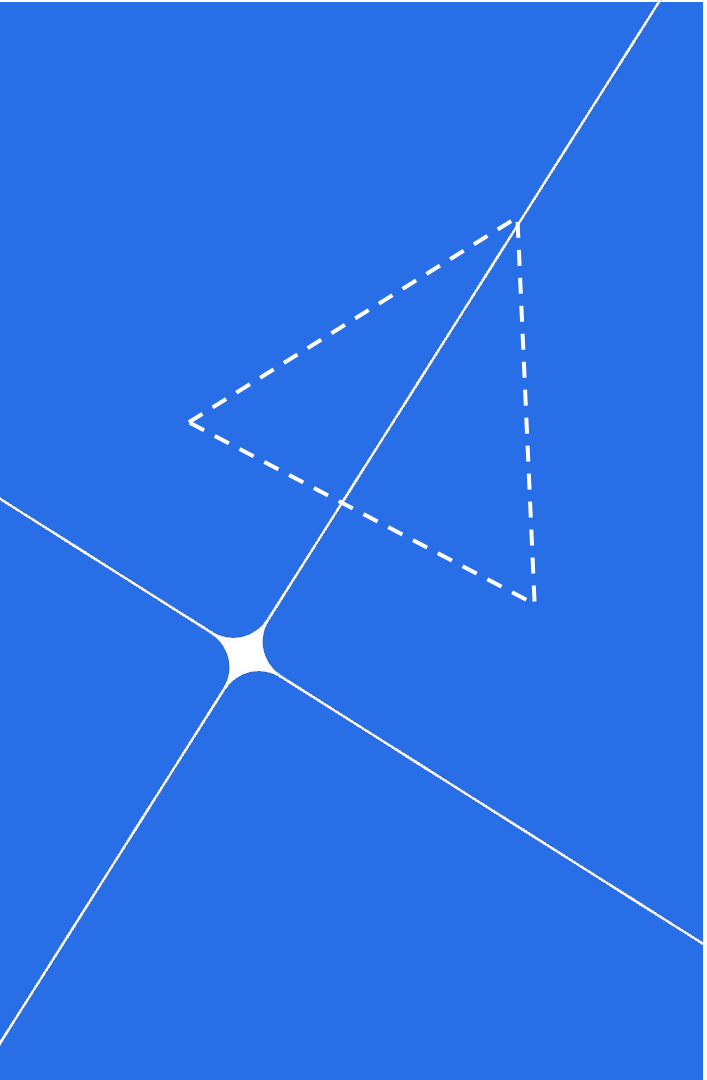
- Quand beaucoup de nœuds lointains doivent influencer v, l’info se “compresse”
- Le nombre de nœuds à distance k peut croître très vite (expansion)
- Mais $h_v^{(\ell)}$ a dimension fixe : goulot d’étranglement
  - Similaire avec les RNN
- Résultat : signaux longs chemins mal transmis (même sans oversmoothing)
- Indices : tâches nécessitant dépendances longues (chemins) difficiles
- Parades (mention) :
  - attention/biais structurels, rewiring, positional encodings, architectures plus expressives
- Transition : différentes architectures = différents choix de ϕ, AGG, ψ

Expressivité : ce que “voit” une GNN

- Un GNN MPNN agrège des voisins : elle distingue surtout via **multisets** de voisinage
- Certaines structures non isomorphes (différentes) peuvent devenir indiscernables (limites)
- Lien classique : test WL (*Weisfeiler-Lehman*) comme référence d’expressivité (culture)
- Implication pratique : si la structure fine compte, il faut :
    - bonnes features initiales, positional encodings, architectures plus puissantes
- Pour l’industrie : souvent OK, car features riches + signal statistique
- On garde en tête : “plus expressif” implique souvent “plus coûteux”

Transition : du cadre MPNN aux architectures (GCN, GraphSAGE, GAT)

- GCN : ϕ simple + normalisation (vue diffusion / spectral-lite)
- GraphSAGE : agrégation + **sampling** pour passer à l'échelle (inductif)
- GAT : ϕ pondéré par attention (voisins plus/moins importants)
- Même squelette : messages → agrégation → update
- Prochaine section : relier MPNN à la vue "convolution" via le Laplacien (spectral)
- Puis dériver les intuitions pratiques pour entraîner à grande échelle



Convolution de graphe : vue spectrale & GCN

Pourquoi une “convolution” sur un graphe ?

- En vision : convolution = filtre local + partage de poids sur une grille régulière
- Sur un graphe : pas de grille, mais une notion de **voisinage** et de **diffusion**
- Objectif : définir un opérateur “convolution-like” qui respecte la structure
- Deux lectures complémentaires :
  - **spectrale** : filtre dans une base de Fourier de graphe
  - **spatiale / message passing** : agrégation de voisins (MPNN)
- Intuition industrie : “lisser” le bruit local, propager un signal relationnel utile
- GCN (Graph Convolutional Network) : pont pratique entre ces vues

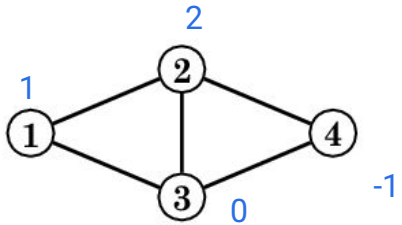
Laplacien de graphe : l'objet central (Graph Signal Processing)

- Graphe (non orienté) : matrice d'adjacence A, degré D avec $D_{vv} = \sum_u A_{vu}$
- **Laplacien combinatoire** : $L = D - A$
- **Laplacien normalisé** : $L_{sym} = D^{-1/2} L D^{-1/2} = I - D^{-1/2} A D^{-1/2}$
- Propriété clé : L mesure une "variation" du signal sur le graphe
- Signal de graphe : $x \in \mathbb{R}^{|V|}$ (ou $X \in \mathbb{R}^{|V| \times d}$)
- $x^\top L x = \frac{1}{2} \sum_{(u,v) \in E} A_{uv} (x_u - x_v)^2$: pénalise les différences sur arêtes

$$L_{i,j} := \begin{cases} \deg(v_i) & \text{if } i = j \\ -1 & \text{if } i \neq j \text{ and } v_i \text{ is adjacent to } v_j \\ 0 & \text{otherwise,} \end{cases}$$

$$L^{\text{sym}}_{i,j} := \begin{cases} 1 & \text{if } i = j \text{ and } \deg(v_i) \neq 0 \\ -\frac{1}{\sqrt{\deg(v_i)\deg(v_j)}} & \text{if } i \neq j \text{ and } v_i \text{ is adjacent to } v_j \\ 0 & \text{otherwise.} \end{cases}$$

$$D = \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 2 \end{bmatrix}$$



Si x = [1, 2, 0, -1], $A^T x$ = [2, 0, 2, 2] -> somme brute des voisins
$L^T x$ = [0, 6, -2, 0] -> on garder valeur * degré et on soustrait la somme des voisins
$L_{sim}^T x$ = [0.18, 2, -0.66, -1.82] -> on garde valeur et on soustrait les voisins normalisés

$$A = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}$$

$$L = \begin{bmatrix} 2 & -1 & -1 & 0 \\ -1 & 3 & -1 & -1 \\ -1 & -1 & 3 & -1 \\ 0 & -1 & -1 & 2 \end{bmatrix}$$

$$L_{sim} = \begin{bmatrix} 1 & -0.41 & -0.41 & 0 \\ -0.41 & 1 & -0.33 & -0.41 \\ -0.41 & -0.33 & 1 & -0.41 \\ 0 & -0.41 & -0.41 & 1 \end{bmatrix}$$

Fourier sur graphe : fréquences = modes propres du Laplacien

- Décomposition : $L=U\Lambda U^{\top}$
  - U : vecteurs propres (base de Fourier de graphe)
  - Λ : valeurs propres (fréquences, $0=\lambda_0 \leq \lambda_1 \leq \dots$)
- Transformée de Fourier : $\hat{x} = U^{\top}x$, inverse : $x = U\hat{x}$
- “Basses fréquences” : variations lentes sur le graphe (signal lisse)
- “Hautes fréquences” : variations rapides (fort contraste entre voisins)
- Intuition : filtrer = amplifier/atténuer certaines fréquences sur le graphe
- Problème pratique : calculer U coûte $O(|V|^3)$ (impossible à grande échelle)

Convolution spectrale : définition (et pourquoi c'est dur)

- Filtre spectral $g_\theta(\Lambda)$ (fonction des valeurs propres)
- Convolution sur graphe :
  - $y = g_\theta(L)x = Ug_\theta(\Lambda)U^\top x$
- Lecture : on transforme en Fourier, on filtre, on revient dans l'espace des nœuds
- Problèmes :
  - dépend de U → pas "transférable" entre graphes différents
  - multiplication par U coûte cher
- Solution : approximer $g_\theta(L)$ par un polynôme de L (localité + calcul sparse)
- Idée clé : polynôme en L ⇒ dépend seulement des voisinages à k sauts

De l'approximation polynomiale à une couche locale

- Approche Chebyshev (Defferrard et al.) : $g_\theta(L) \approx \sum_{k=0..K} \theta_k T_k(\tilde{L})$
  - T_k = polynômes de Chebyshev ($T_0 = 1$, $T_1 = X$, $T_{k+2} = 2XT_{k+1} - T_k$)
  - $\tilde{L}$ = Laplacien "rescalé" entre -1 et 1
- K contrôle la portée : K=1 ≈ très local (1-hop)
- Simplification "GCN-style" :
  - on vise un filtre lissant (low-pass) simple
  - on supprime la dépendance explicite à la base spectrale
- Résultat : une couche qui ressemble à un **mélange normalisé des voisins**
- Pont avec MPNN : ϕ simple, AGG = somme/ moyenne pondérée
- Conséquence : calcul basé sur A sparse ⇒ GPU-friendly

GCN (Kipf & Welling) : équation de couche

- Ajout de self-loops : $\tilde{A}=A+I$, $D_{vv}=\sum_u \tilde{A}_{vu}$
- Normalisation symétrique : $\hat{A}=D^{-1/2}\tilde{A}D^{-1/2}$
- Couche GCN :
  - $H^{(\ell+1)}=\sigma(\hat{A}\,H^{(\ell)}\,W^{(\ell)})$
- $H^{(0)}=X$ (features), $W^{(\ell)}$ poids appris, σ activation (ReLU, GELU…)
- Interprétation : agrégation normalisée + projection linéaire partagée
- Setting classique : **semi-supervised node classification** (labels sur une partie des nœuds)

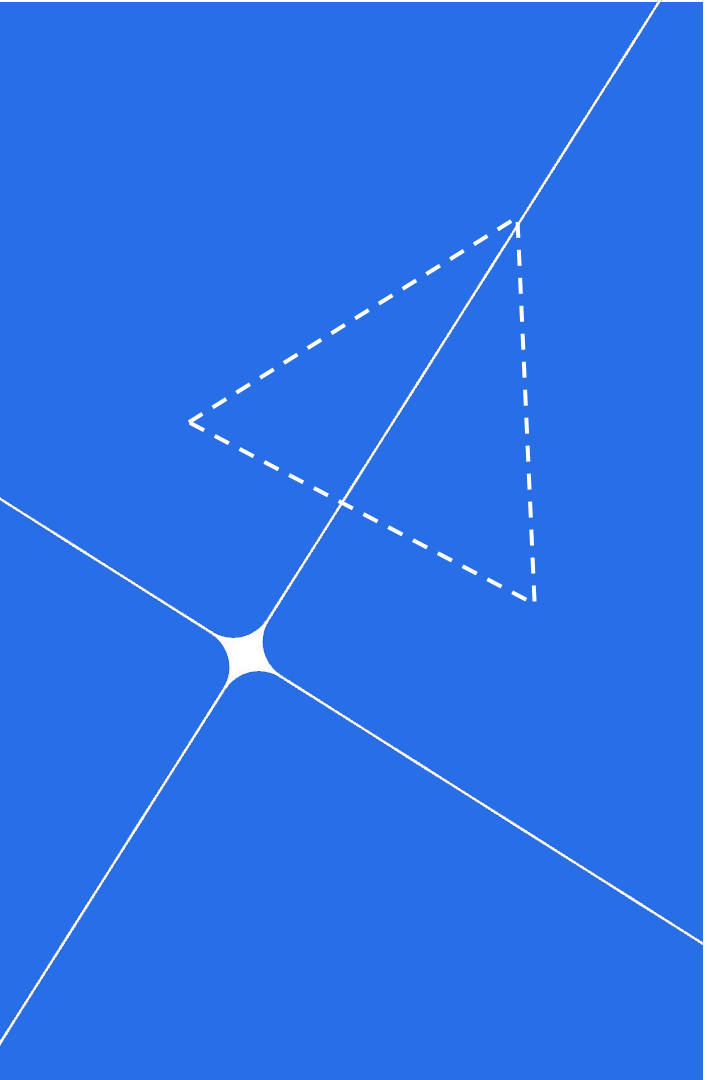
Lecture “diffusion / lissage” et implications pratiques

- Â agit comme un opérateur de **diffusion** (propagation) du signal
- Effet : rend voisins plus similaires (utile si homophilie, dangereux si trop profond)
- Points forts :
  - simple, stable, très bonne baseline GNN
  - bon point de départ pour comprendre le reste (GraphSAGE, GAT)
- Limites :
  - **full-batch** sur gros graphes (si on multiplie ÂH sur tout le graphe)
  - oversmoothing si trop de couches
  - sur graphes très hétérophiles (voisins de classes différentes), peut dégrader
- Heuristiques pratiques : 2-3 couches souvent, residual + dropout si besoin

Exemple code “générique” : propagation normalisée

```
# X: [N, d], edge_index: [2, E]
A_hat = normalized_adjacency_with_self_loops(edge_index, N)  # sparse
H = X
for _ in range(L):
    H = relu( A_hat @ (H @ W) )  # sparse-dense matmul
logits = H @ W_out
```

- Coût dominant : A_hat @ (...) (sparse × dense)
- En pratique : caching possible (A_hat fixe) + mini-batch requis à grande échelle
- Transition : GraphSAGE va remplacer ce full-batch par neighbor sampling



Architectures cœur : GCN vs GraphSAGE vs GAT

Pourquoi 3 architectures (et pas 15) ?

- Objectif : couvrir 3 “axes de design” qui reviennent partout
- **GCN** : propagation normalisée simple (baseline robuste)
- **GraphSAGE** : **inductif** + **sampling** pour gros graphes (industrie)
- **GAT** : pondération adaptative des voisins (attention)
- Toutes rentrent dans le cadre **MPNN** : message → agrégation → update
- On compare : hypothèses, coût, scalabilité, quand choisir quoi
- Mot-clé : la différence est souvent dans **AGG** + normalisation + accès au graphe

GCN en pratique

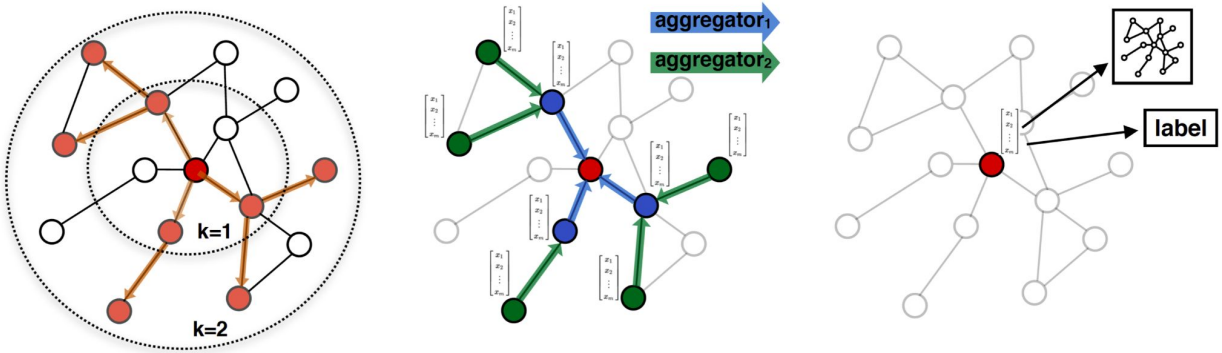
- Couche : $H^{\ell+1} = \sigma(\hat{A}H^{\ell}W^{\ell})$ avec $\hat{A} = D^{-1/2}\tilde{A}D^{-1/2}$
- AGG implicite : somme pondérée par degrés (normalisation symétrique)
- Points forts :
  - simple à entraîner, bonne baseline, peu d'hyperparamètres
- Points faibles (industrie) :
  - full-batch sur $|E|$ si on fait passer tout le graphe
  - pas idéal si graph très grand ou très dynamique
- Bon usage : graph "moyen", features riches, homophilie correcte
- Astuce : 2-3 couches + residual/dropout si besoin

GraphSAGE : setting inductif (nouveaux nœuds)

- Problème réel : nouveaux comptes / nouveaux utilisateurs arrivent tous les jours
- **Inductif** : le modèle doit traiter un nœud jamais vu à l'entraînement
- Idée GraphSAGE :
  - générer h_v à partir de x_v + agrégation des voisins
  - sans dépendre d'une matrice Â fixée sur un graphe spécifique
- Forme générique :
  - $h_v^{\ell+1} = \sigma(W^{\ell} \cdot \mathrm{CONCAT}(h_v^{\ell}, \mathrm{AGG}(\{h_u^{\ell}, u \in N(v)\})))$
- AGG typiques : mean, max-pool (via MLP), LSTM-aggregator (plus rare)
- Message : même logique MPNN, mais conçu pour la **généralisation inductive**

GraphSAGE : neighbor sampling (scalabilité)

- Problème : N(v) peut être énorme (hubs), et 2-3 couches explosent en taille
- Solution : échantillonner un sous-ensemble de voisins à chaque couche
- Fanout typique : ex 15 voisins au 1-hop, 10 au 2-hop, etc.
- Mini-batch : on entraîne sur un batch de nœuds cibles + leur “computation graph” échantillonné
- Coût approximatif : $O(\text{batch} \times \prod_{\ell} \text{fanout}_{\ell})$
- Trade-off : vitesse/mémoire vs variance du gradient
- Très industrie : compatible avec graphes très grands + mises à jour fréquentes



1. Sample neighborhood

2. Aggregate feature information from neighbors

3. Predict graph context and label using aggregated information

Code générique : GraphSAGE avec sampling (style PyTorch)

```
# seed_nodes: nodes we want to train on (mini-batch)
subgraph = sample_k_hop_neighbors(edge_index, seed_nodes, fanouts=[15, 10])
h = X[subgraph.nodes]
for layer in layers:
    h = layer(h, subgraph.edge_index)  # aggregation only on sampled edges
logits = head(h[subgraph.seed_mask])
loss = CE(logits, y[seed_nodes])
```

- Concept clé : on entraîne sur un sous-graphe échantillonné, pas le graphe entier
- Compatible GPU : sous-graphe “dense enough” pour amortir le coût
- Attention : sampling doit être reproductible (seeds) pour debug/benchmark

GAT : attention sur voisins (Graph Attention Network)

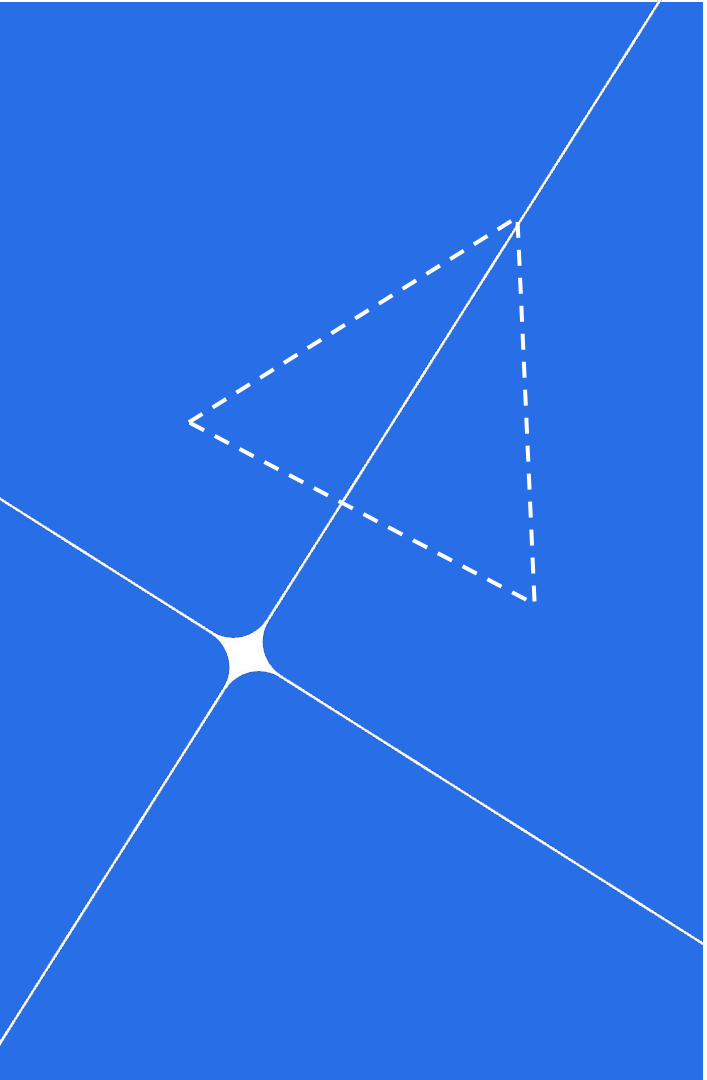
- Motivation : tous les voisins ne se valent pas (bruit, hubs, relations faibles)
- Idée : apprendre un poids α_{uv} pour chaque arête $(u \to v)$
- Forme (simplifiée) :
  - $e_{uv} = a^{\top}[Wh_u \,//\, Wh_v]$
  - $\alpha_{uv} = \mathrm{softmax}_{u \in N(v)}(\mathrm{LeakyReLU}(e_{uv}))$
  - $h_v' = \sigma(\sum_{u \in N(v)} \alpha_{uv} Wh_u)$
- Multi-head attention : stabilise + augmente capacité (concat/mean des têtes)
- Point pratique : GAT = MPNN où AGG est une somme **pondérée** apprise
- Gain attendu : robustesse au voisinage bruité + focus sur signaux pertinents

GAT : forces, limites, et coût

- Forces :
  - pondération adaptative (utile si hétérophilie locale / bruit)
  - interprétation partielle via α_{uv} (attention ≠ explication complète)
- Limites :
  - coût plus élevé : calcul α par arête + softmax par voisinage
  - sur gros graphes : nécessite sampling / approximation (sinon trop cher)
  - peut sur-apprendre sur structures de degré (attention sur hubs)
- En industrie : GAT fonctionne bien si on contrôle fanout et régularisation
- Heuristiques : dropout sur attention, heads raisonnables, residual + norm
- Message : GAT = “plus flexible”, pas toujours “meilleur”

Comparaison synthétique : quand choisir quoi ?

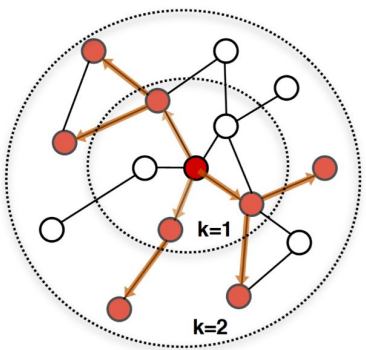
- **GCN** :
  - simple, baseline, stable
  - difficile à scaler en full-batch, lissage fort
- **GraphSAGE** :
  - inductif, scalable via sampling, bon pour production
  - variance sampling, dépend qualité du voisinage échantillonné
- **GAT** :
  - pondération fine des voisins, utile si voisinage hétérogène/bruité
  - plus coûteux, parfois instable sans régularisation
- Règle pratique :
  - commencer GCN/GraphSAGE baseline
  - passer GAT si besoin de sélectivité locale et budget compute OK
- Prochaine étape : relier ces embeddings aux **tâches** (node & link), pertes, negative sampling



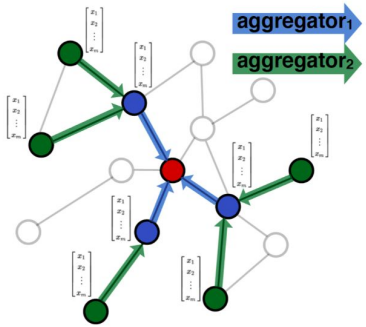
Objectifs d’entraînement & tâches

De l'embedding à la prédiction : "encoder" + "head"

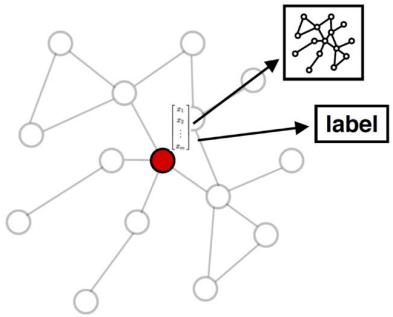
- GNN = **encodeur** : X, E→H (embeddings nœuds h_v)
- On ajoute une **tête** (*prediction head*) selon la tâche
- Node classification : $h_v \rightarrow \text{logits}_v \rightarrow \hat{y}_v$
- Link prediction : $(h_u, h_v) \rightarrow s(u,v) \rightarrow \hat{p}(u,v)$
- Graph classification : $\{h_v\} \rightarrow h_G$ (pooling) $\rightarrow \hat{y}_G$
- Supervision souvent **partielle** (labels sur un sous-ensemble)
- Objectif : apprendre H "utile" via une loss cohérente avec la tâche



1. Sample neighborhood



2. Aggregate feature information from neighbors



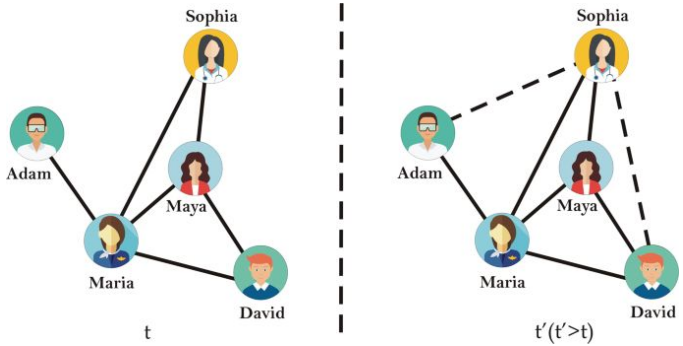
3. Predict graph context and label using aggregated information

Node classification : setup, loss, métriques

- Données : X, E labels y_v sur un sous-ensemble $V_L \subset V$
- Modèle : $logits_v = MLP(h,)$, $\hat{y}_v = argmax(softmax(logits_v))$
- Loss standard : **cross-entropy** sur nœuds labellisés $L = \sum_{v \in VL} CE(logits_v, y_v)$
- Cas fraude : **class imbalance** (fraude rare)
  - pondération, focal loss, sampling de classes
- Métriques usuelles :
  - **F1** (harmonique précision/rappel)
  - **ROC-AUC** (Area Under ROC Curve)
  - **PR-AUC** (Area Under Precision-Recall Curve, souvent plus parlante si rareté)
- Bon réflexe : baseline MLP (sans graphe) pour mesurer le gain structurel

Link prediction : score de lien et “decoder”

- But : prédire un lien (u,v) ou son type (existence, fraude, relation)
- Encodeur : calcule h_u,h_v à partir du graphe “observé”
- **Decoder / scoring** (exemples) :
  - dot product : $s(u, v)=h_u^{\top}h_v$
  - bilinear : $s(u, v)=h_u^{\top}Wh_v$
  - MLP : $s(u, v)=\text{MLP}([h_u \,//\, h_v \,//\, |h_u - h_v|])$
- Probabilité : $\hat{p}(u, v) = \sigma(s(u,v))$ (sigmoid)
- Loss typique (binaire) : **binary cross-entropy** sur liens positifs + négatifs
- En pratique : choix du split d’arêtes = critique (éviter fuite d’info)



Negative sampling : pourquoi et comment (link prediction)

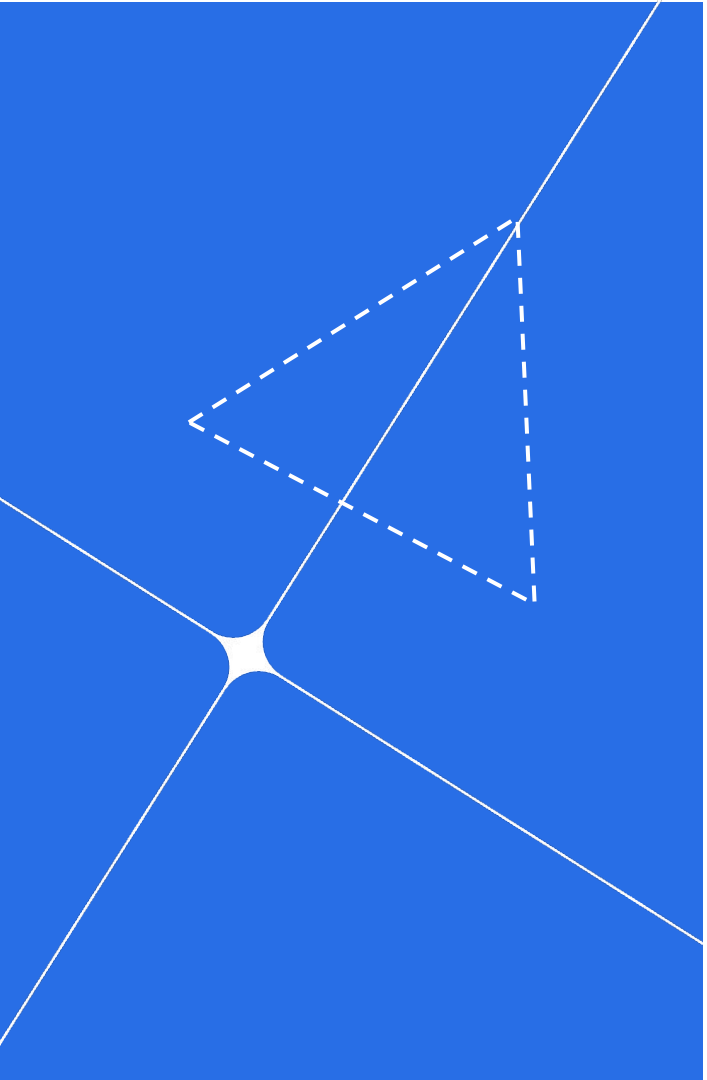
- Problème : on observe surtout des **positifs** (liens existants), pas de “non-liens” labellisés
- Solution : générer des **négatifs** (paires (u,v) supposées non connectées)
- Sampling simple : uniforme sur paires de nœuds (rapide, mais souvent trop facile)
- Variantes utiles :
  - **degree-based** : éviter de ne tirer que des non-liens triviaux
  - **hard negatives** : négatifs “plausibles” (mêmes communautés, mêmes types)
  - type-aware (si hétéro) : respecter les types (Account-Merchant vs Account-Device)
- Ratio négatifs/positifs : hyperparamètre (ex : 1:1, 1:5, 1:10)
- Risque : “faux négatifs” (lien réel mais non observé) qui bruitent l’apprentissage

Panorama rapide des tâches

- **Node classification** : compte frauduleux ? topic d'un papier ?
- **Edge classification** : classer une arête observée (transaction legit/fraude)
- **Link prediction** : futur achat ? transaction suspecte ? relation manquante ?
- **Graph classification** : molécule toxique ? document = catégorie ?
- **Graph regression** : prédire une valeur (propriété chimique, KPI agrégé)
- **Clustering / community detection** : segmentation (marketing, fraude en groupes)
- Message : même embeddings, mais objectif et protocole changent

Évaluation : pièges fréquents

- **Leakage** : une arête tenue “secrète” ne doit pas influencer l’embedding au train
- Splits possibles :
  - node split (labels), edge split (liens), parfois par temps (plus réaliste)
- En link prediction : séparer clairement
  - edges pour message passing (train graph)
  - edges à prédire (val/test)
- Attention aux hubs : un score peut tricher via degré (baseline “popularité”)
- Calibration utile en production : probas exploitables, seuils stables (precision/recall trade-off)
- Toujours rapporter : baseline non-graph + ablation (sans sampling, sans attention, etc.)



Entraîner à l’échelle : mini-batch, sampling, perf

Pourquoi le full-batch casse en industrie

- Sur gros graphes : $|V|$ et $|E|$ trop grands pour un passage complet GPU
- Coût d'une couche "message passing" ≈ $O(|E| \cdot d)$ (sparse × dense)
- Mémoire : stocker H (embeddings) + activations + gradients explose vite
- Graphes réels : **skew** (degrés très hétérogènes) → hubs dominent le coût
- Données évolutives : recalculer tout le graphe à chaque update = impraticable
- Donc : on entraîne sur **sous-graphes / mini-batch** (comme en vision, mais plus subtil)

Mini-batch sur graphe : 2 stratégies principales

- **Neighbor sampling** (GraphSAGE-style) :
  - batch de nœuds cibles + échantillonnage de voisins par couche
  - bon compromis perf, très utilisé en prod
- **Subgraph sampling** :
  - on tire un sous-graphe “cohérent” (cluster/partition) puis on entraîne dedans
  - réduit les coupures de voisinage, meilleure locality mémoire
- Différence clé vs vision :
  - un batch n’est pas un ensemble indépendant, c’est un **computation graph**
- Propriété à garder : agrégation sur voisins = opération sur un ensemble (pas d’ordre)
- Objectif : limiter l’explosion du k-hop tout en gardant du signal

Neighbor explosion : comprendre le coût

- Sans sampling : taille du voisinage croît ≈ produit des degrés (explosion)
- Avec sampling : fanout fixé par couche, ex [15, 10] pour 2 couches
- Taille du computation graph ≈ batch×$(1+f_1+f_1f_2+\dots)$
- Trade-offs :
  - fanout faible : plus rapide, mais perte de signal (variance)
  - fanout élevé : mieux statistiquement, mais coût GPU/CPU monte vite
- Cas hubs : sampling évite d'agréger 50k voisins, mais introduit du bruit
- Bon réflexe : profiler (temps data loading vs temps GPU)

Pipeline perf : CPU↔GPU, loaders, et goulets d'étranglement

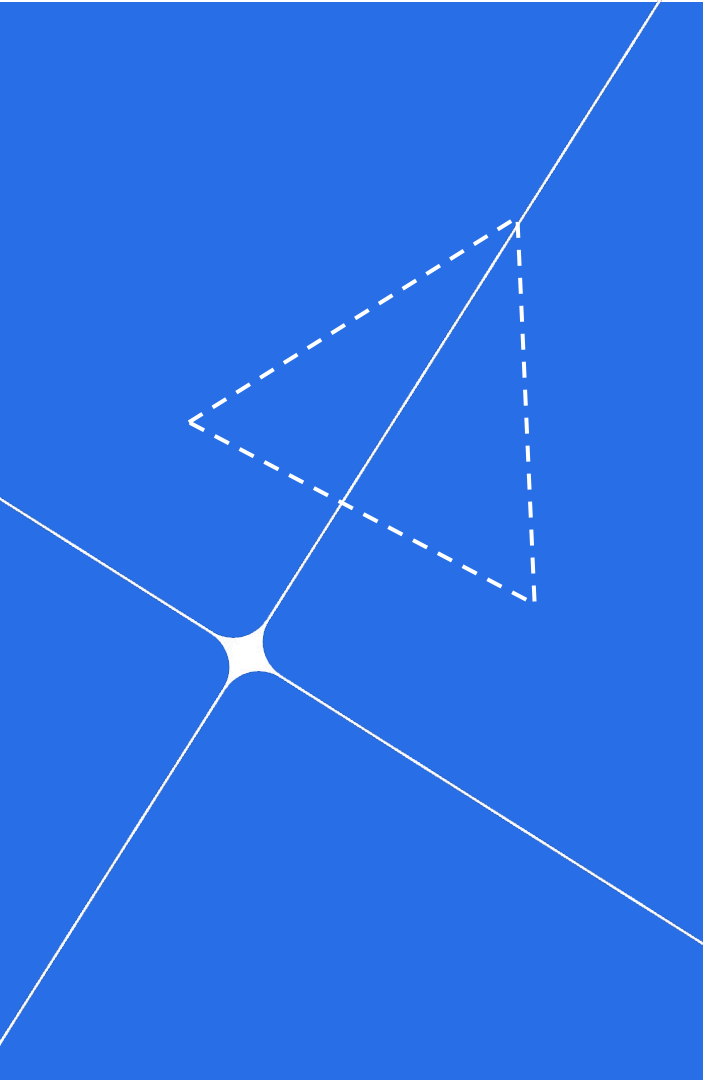
- En pratique, le bottleneck n'est pas toujours le modèle mais :
  - sampling CPU, transfert vers GPU, construction de mini-batch
- Points critiques :
  - **pin_memory**, préfetch, workers DataLoader (selon infra)
  - format sparse + coalescing (éviter fragmentation)
  - limiter les copies : garder edge_index compact, features contiguës
- Stratégies :
  - cache des features "chaudes" (hubs / nœuds fréquents)
  - mixed precision si stable (gain mémoire / débit)
  - gradient accumulation si batch GPU limité
- Observabilité : mesurer "data time" vs "compute time" à chaque epoch

Inference en production : embeddings offline vs online

- Deux modes classiques :
  - **offline** : pré-calcul embeddings périodiquement (batch), puis servir un modèle léger
  - **online** : calcul à la volée sur un sous-graphe local (latence + complexité)
- Fraude typique :
  - scoring temps réel d'une transaction → besoin d'un voisinage récent
  - compromis : cache + features agrégées + embeddings mis à jour régulièrement
- Déploiement :
  - séparer "feature store" graphe et "model serving"
  - attention aux changements de graphe (drift structurel)
- Évaluation continue : recalibrer seuils (PR/ROC) quand le base-rate change
- Message : le choix d'archi (SAGE/GAT) dépend aussi de la stratégie d'inférence

Distribué : quand un GPU ne suffit plus

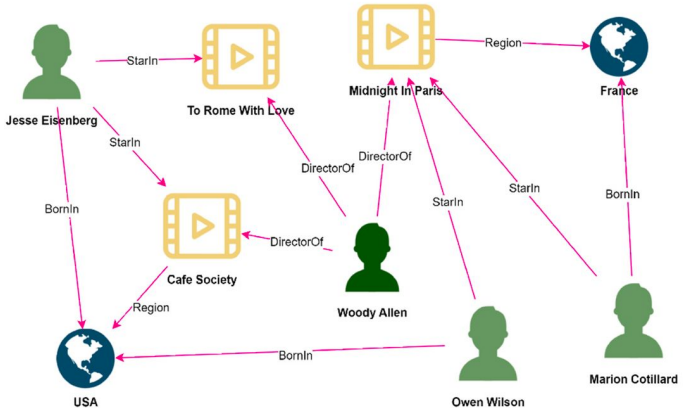
- Motif : graphes trop gros → partitionnement / multi-GPU / multi-nœuds
- Idées clés :
  - partitionner le graphe (réduire les arêtes coupées)
  - sampler localement par partition, synchroniser gradients
- Coûts cachés :
  - communication inter-nœuds, déséquilibres (partitions avec hubs)
- Approche pragmatique :
  - commencer single-GPU + sampling
  - scaler ensuite (data parallel, partitions) si gain clair
- Sur cluster : planifier jobs reproductibles (seeds, versions, artefacts)



Au-delà du cœur : graphes hétérogènes & Graph Transformers

Pourquoi l'hétérogène arrive "naturellement" en industrie

- Données réelles = plusieurs tables SQL, clés étrangères, logs multi-sources
- Types de nœuds typiques (fraude) : Account, Device, Merchant, Card, Transaction
- Types d'arêtes : uses_device, paid, transfers_to, owns_card, shares_ip, …
- Gains : structure plus fidèle, signaux relationnels plus riches (type-aware)
- Coût : plus de paramètres, plus de complexité de modélisation / debug
- Stratégie pragmatique : commencer homogène, passer hétéro si gain clair
- Terme utile : **schema** du graphe (types + relations autorisées)



Hétéro en pratique : l'idée "relation-specific"

- Problème : la relation "paid" n'a pas le même sens que "uses_device"
- Idée : messages/poids **dépendent du type d'arête** (relation r)
- Schéma conceptuel :
  - $m_{u \to v}^{(\ell)} = W_r^{(\ell)} h_u^{(\ell)}$ (poids par relation)
  - $h_v^{(\ell+1)} = \sigma(\sum_r \sum_{u \in N_r(v)} \text{norm} \cdot m_{u \to v}^{(\ell)})$
- Avantages : meilleure expressivité sur multi-relations (graphes de connaissances, fraude)
- Risques : explosion de paramètres si beaucoup de relations (régulariser / factoriser)
- Alternative simple : encoder les types en features + utiliser une archi homogène (baseline)
- Message : hétéro = **modélisation** + **ingénierie** (pas juste une "option")

Graph Transformers : pourquoi ça existe (et pourquoi c'est cher)

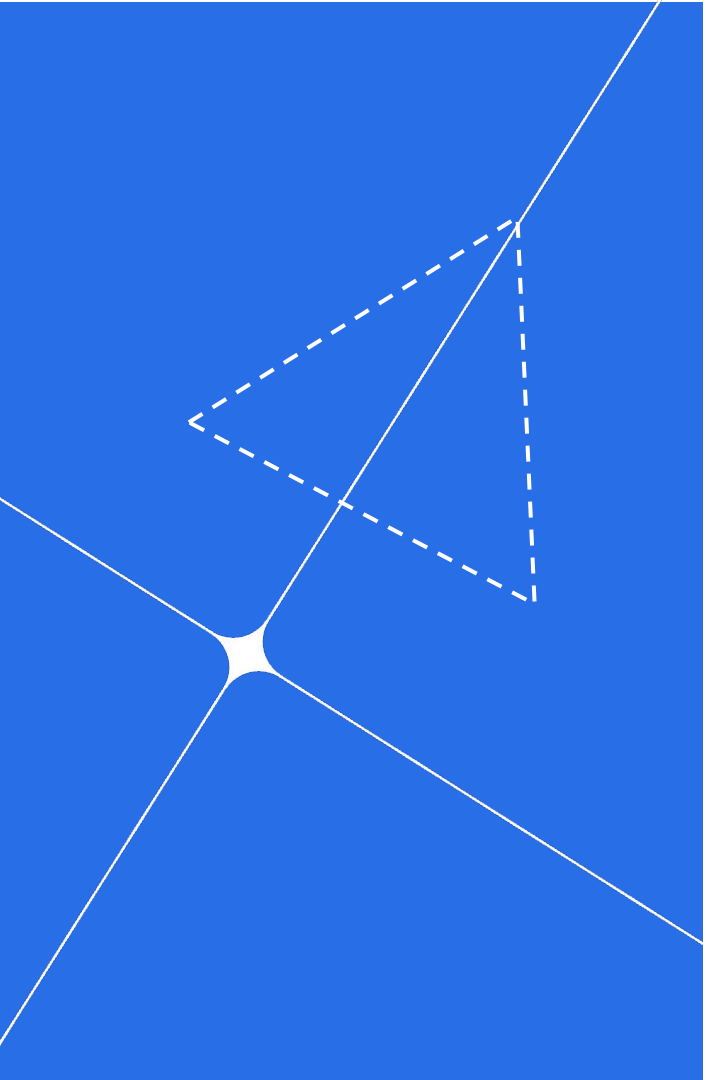
- Motivation : message passing local peut manquer de dépendances longues / globales
- Transformer "pur" sur nœuds : attention globale $O(|V|^2) \rightarrow$ prohibitif
- Idée clé : injecter la structure dans l'attention via des **bias** et/ou sparsification
- Exemples de signaux structurels utilisés :
  - distance de plus court chemin (shortest path), appartenance au même sous-graphe
  - degrés, centralités, types d'arêtes (si hétéro)
- Avantage : plus expressif, interactions globales possibles
- Inconvénient : coût, mémoire, et sensibilité au choix des signaux structurels
- Positionnement : utile si structure fine + budget compute + besoin global

Positional Encodings (PE) sur graphes : Laplacian PE

- Sur séquences : position = index
  - sur graphes : pas d'ordre, donc PE "non trivial"
- **Laplacian PE** : utiliser des vecteurs propres du Laplacien comme coordonnées
- Intuition : ces vecteurs capturent la géométrie du graphe (communautés, variations)
- Utilisation typique : concaténer PE aux features nœuds avant l'encodeur
- Attention pratique : signe/rotation ambiguë des eigenvectors (traitements robustes)
- Alternative PE : random walk, shortest-path encodings, degree encodings
- Message : sans PE, un Transformer de graphe "voit" mal la structure

Synthèse : quand rester sur GCN/SAGE/GAT vs aller plus loin

- Rester sur **GCN** : baseline stable, graph modéré, homophilie OK
- Choisir **GraphSAGE** : gros graphe + besoin inductif + training scalable (sampling)
- Choisir **GAT** : voisinage bruité / hétérogène local, besoin de pondération fine
- Passer **hétéro** (R-GCN / type-aware) : multi-relations fortement sémantiques
- Envisager **Graph Transformers** : dépendances globales + structure fine + compute
- Règle industrie : complexifier uniquement si baseline + ablations justifient le gain



Synthèse & stack outillée

Résumé : 10 idées à retenir

- Les graphes modélisent entités + relations : signal souvent **non-IID**
- Une GNN respecte la permutation : agrégation **symétrique** sur voisins
- Cadre unificateur : **MPNN** (messages → AGG → update)
- Vue spectrale : Laplacien → “fréquences” → intuition de filtrage/lissage
- **GCN** : diffusion normalisée simple, baseline solide
- **GraphSAGE** : inductif + **neighbor sampling** pour l’échelle (industrie)
- **GAT** : pondération apprise des voisins, coût plus élevé
- Tâches : node / link / edge / graph
  - on a focalisé node + link
- Link prediction requiert souvent **negative sampling** + protocoles anti-leakage
- Scalabilité = autant data/loader/profiling que “architecture”

Checklist ingénierie : ce qui fait échouer un projet GNN

- Graphe mal défini (relations non pertinentes) → le modèle “apprend du bruit”
- Features pauvres → la structure seule ne suffit pas (souvent)
- Leakage (edges futures / splits incohérents) → métriques irréalistes
- Hubs non gérés → coûts explosifs, sampling mal calibré
- Mauvaise métrique (ROC-AUC seule en fraude rare) → décision produit mauvaise
- Non-reproductibilité (sampling aléatoire sans seeds) → impossible à debug
- Pas de baseline tabulaire → on ne sait pas si le graphe apporte réellement
- Message : rigueur data + protocole d’éval = moitié du succès

Outils : PyTorch Geometric (PyG) — ce qu'il faut savoir

- Objets clés :
  - Data(x, edge_index, y, ...) pour homogène
  - HeteroData si multi-types (mention)
- Conventions : edge_index = indices COO [2,E], compatible sparse ops
- Modules typiques : GCNConv, SAGEConv, GATConv (et variantes)
- Loaders : neighbor sampling / mini-batch (conceptuellement, pour scaler)
- Bon réflexe : garder un "graph builder" propre (depuis SQL/ETL) + versionné
- Profiling : identifier data bottleneck vs compute bottleneck avant d'optimiser le modèle

Benchmarks & calibration

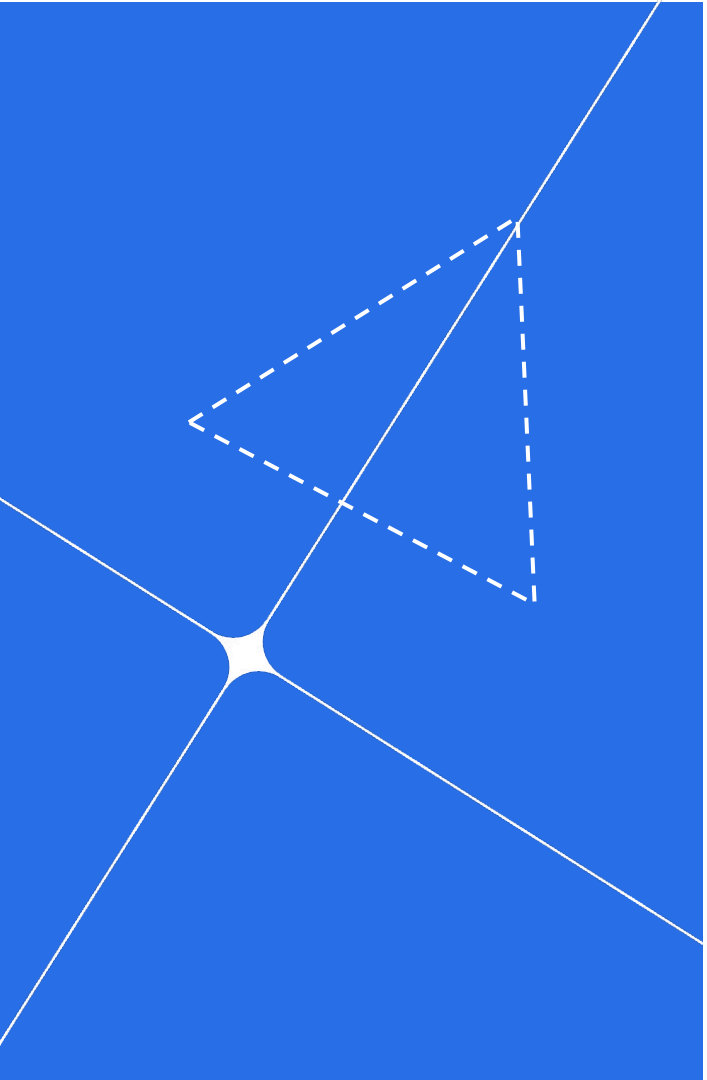
- Se comparer à l’existant : jeux publics (ex : citations, produits, interactions)
- **OGB (Open Graph Benchmark)** : standard pour node/link/graph tasks (référence)
- Pourquoi c’est utile :
  - sanity check de pipeline (splits, metrics, training loop)
  - ordre de grandeur sur ce que GCN/SAGE/GAT peuvent atteindre
- En industrie : vos données ≠ benchmark, mais le benchmark détecte les bugs
- Attention : ne pas sur-optimiser pour un benchmark au détriment du cas réel
- Message : benchmark = outil de robustesse, pas objectif final

Encadré “cluster GPU / Slurm”

- Entraînement GNN = souvent data-heavy (sampling) + compute GPU
- Sur cluster : exécutions reproductibles (configs, seeds, versions, artefacts)
- Points pratiques :
  - réserver GPU + CPU suffisants pour le sampling
  - surveiller IO (features) et temps DataLoader
- Bon pattern : logs structurés (métriques + throughput) pour comparer runs
- On vise une approche “engineering-grade” : on mesure avant d’optimiser

Conclusion : où placer les GNN dans votre boîte à outils deep learning

- Utiliser GNN si la relation porte de l'information exploitable (multi-hop)
- Commencer simple : baseline tabulaire + GCN/SAGE
- Complexifier si nécessaire : GAT, hétéro, puis graph transformers
- Toujours valider protocole (splits, leakage, negatives) avant d'interpréter les gains
- Objectif final : embeddings utiles + système entraînable et maintenable



En route vers le TP