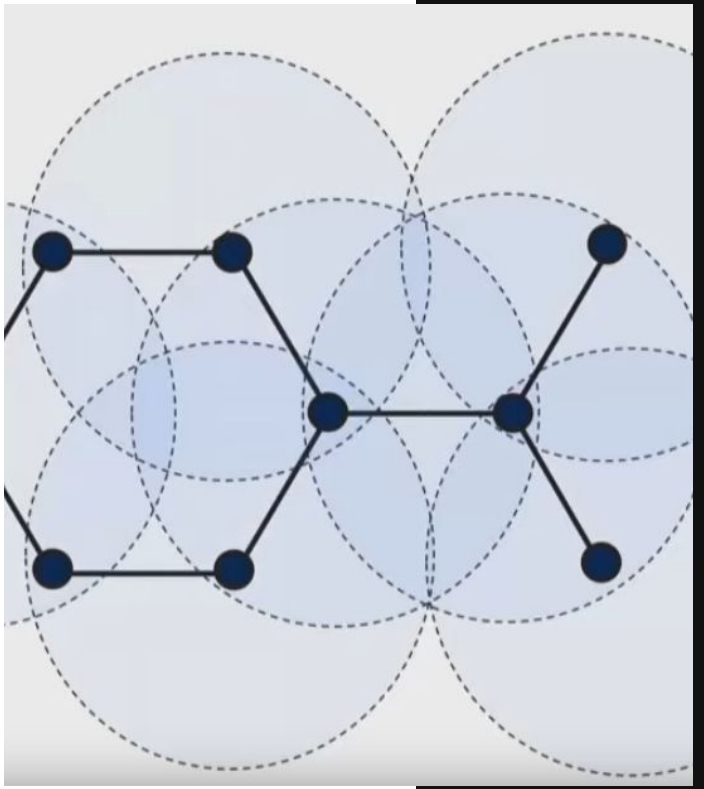
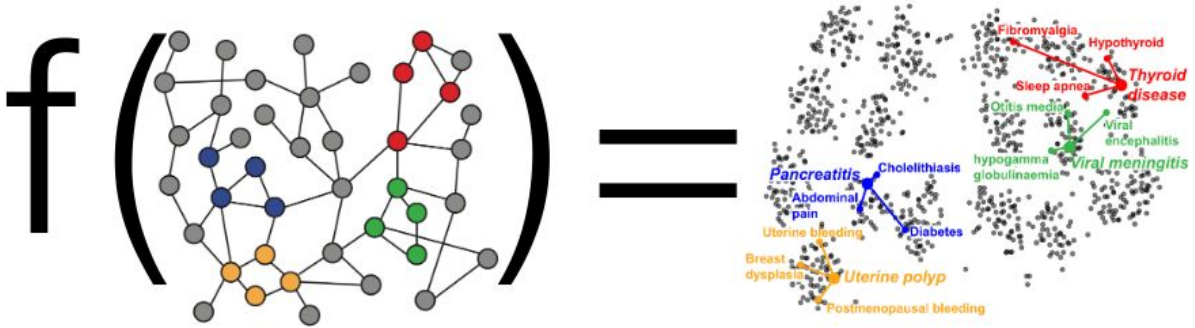
Graph Mining & network science

Graph Convolutional Networks



Résumé des épisodes précédents

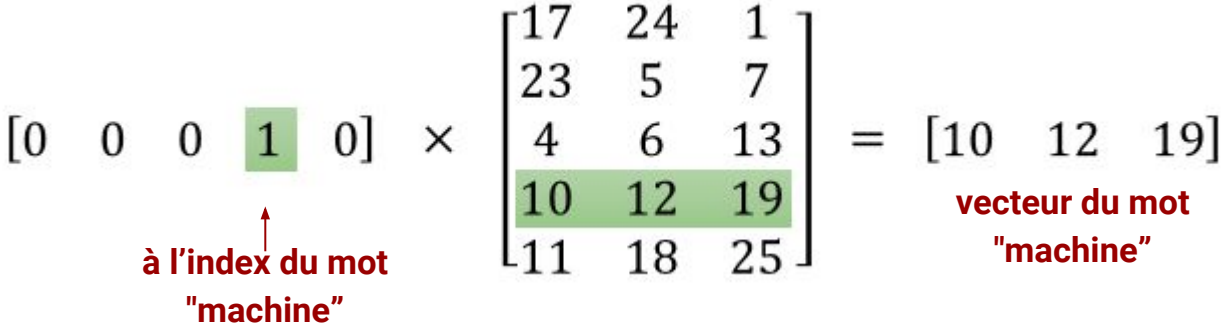
Nous cherchions à apprendre une représentation **d-dimensionnelle** des noeuds d'un graphe telle que des noeuds similaires soient proches dans l'espace :



Résumé des épisodes précédents

Dans le cas de Node2Vec, l'encoder était peu profond:

création d'embeddings ex-nihilo, accessibles par **lookups**



puis optimisation de ces embeddings en exploitant certaines métriques (produit scalaire, distance euclidienne, similarité cosinus, etc.)

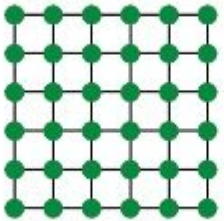
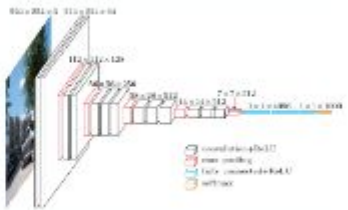
Résumé des épisodes précédents

Problèmes avec les méthodes d'embedding (hors FEATHER, plus récent) :

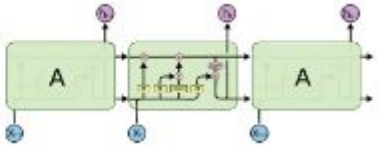
- n'incorporent pas de donnée extérieure à la structure du graphe
- apprentissage des embeddings massif : **|V| x d paramètres**
- apprentissage transductif : pas de façon simple de générer les embeddings de nouveaux noeuds dans le graphe

Résumé des épisodes précédents

- Les CNNs exploitent des images de taille fixe

- Les RNNs exploitent des séquences



Définition du problème

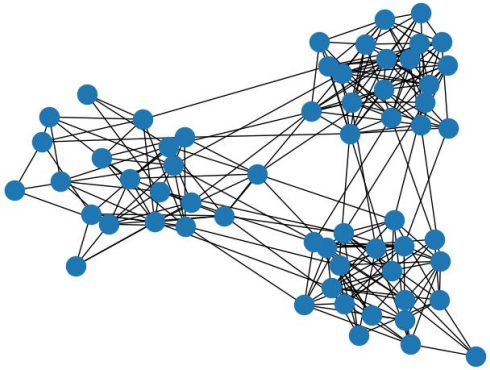
Soit un graphe **G** :

- **V** est l'ensemble des sommets
- **A** est la matrice d'adjacence
- **X** est une matrice de taille **(n, d)** contenant les features des sommets
    - en cas d'absence de features : one hot encoding d'un noeud, ou vecteur constant = 1, ou caractéristiques d'un noeud (degré, cc, …)
- **Y** une matrice de labels pour classification

Exemples :

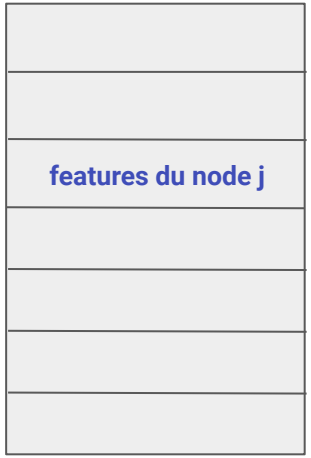
- réseaux sociaux : profil de l'utilisateur, photo de profil
- réseaux biologiques : profils d'expression de gènes, fonctions des gènes

Définition du problème



+

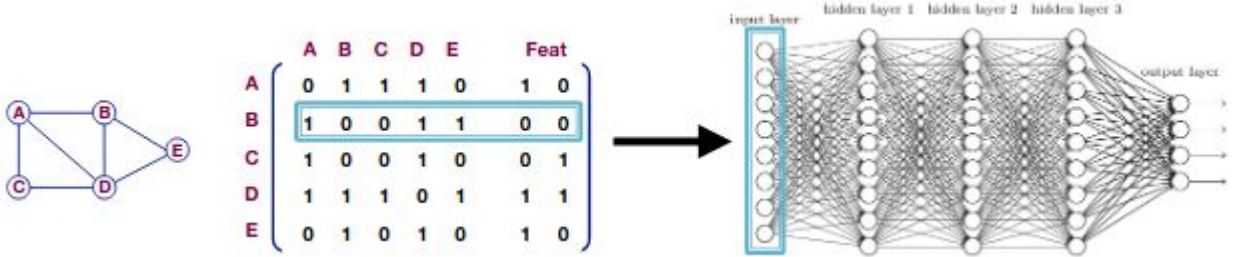
j



→ classification

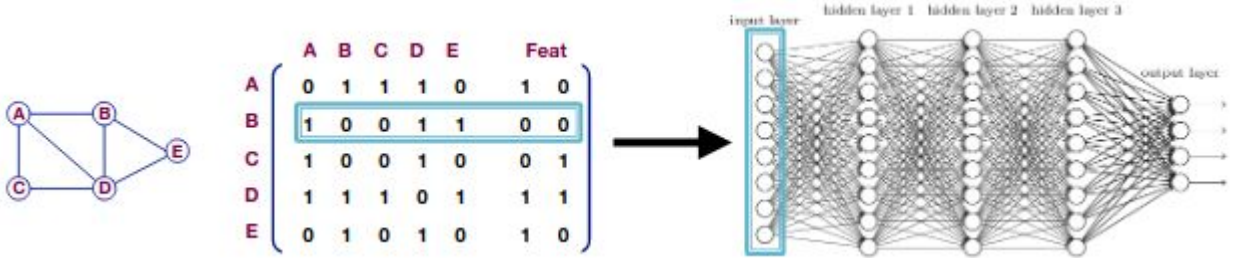
Une approche naïve

- on concatène la matrice d’adjacence et les features
- on envoie chaque sample dans un réseau de neurones profond :



Une approche naïve

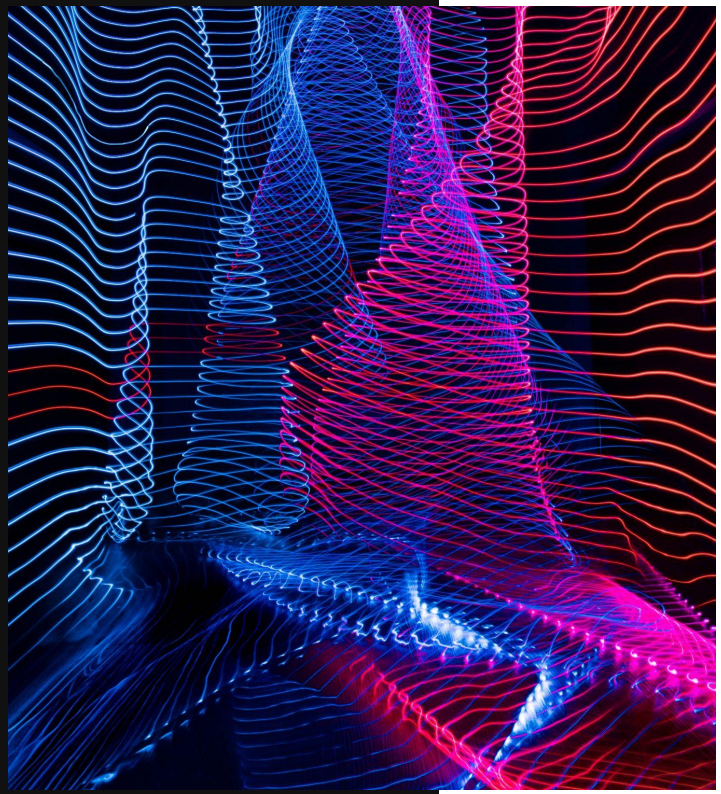
- on concatène la matrice d'adjacence et les features
- on envoie chaque sample dans un réseau de neurones profond :



Problèmes :

- **O(n)** paramètres
- pas de généralisation à des graphes de tailles supérieures
- et surtout : pas d'invariance par permutation des noeuds

Deep learning : rappels utiles



Machine learning comme problème optimisation

- Apprentissage supervisé :
  - en entrée **x**
  - notre but : prédire le label **y**
- L'entrée **x** peut être :
  - vecteurs de nombres réels
  - séquences (NLP, speech to text)
  - matrices/tenseurs (images)
  - graphes (potentiellement avec des features de nodes ou d'edges)

Nous formalisons cette tâche comme un problème d'optimisation

Machine learning comme problème optimisation

- **Θ** : un ensemble de paramètre à optimiser
  - peut contenir des scalaires, vecteurs, matrices, …
  - exemple, **Θ = {Z}** avec **Z** notre matrice d'embedding (cf. lookup)
- Fonction **loss** :
  - une fonction qui prend en entrée notre prédiction et les valeurs observées
  - et nous renvoie un scalaire (un nombre réel) nous informant sur la qualité de la prédiction : plus la valeur est basse, plus notre problème est résolu
  - exemples : MSE, MAE, cross entropy, etc.

Exemple de fonction loss : cross entropy

Classification : directement entraîner le modèle en choisissant une fonction loss comme l'entropie croisée. (**y** = ground truth, **ŷ** = prédiction)

Dans le cas d'une classification binaire (un médicament est-il toxique ?) :

$$\mathcal{L} = -y \cdot log(\hat{y}) - (1 - y) \cdot log(1 - \hat{y})$$

(dans Keras : binary_crossentropy)

Dans le cas d'une classification multi-class (à quelle catégorie appartient tel utilisateur ?) :

$$\mathcal{L} = -\sum_{i} y_i log(\hat{y}_i)$$

(dans Keras : categorical_crossentropy)

Gradient d'une fonction

Comment résoudre notre problème d'optimisation ?

- le gradient d'une fonction est un vecteur qui pointe dans la direction de la plus forte montée
- avec Θ_1, Θ_2, … les composants de Θ:

$$\nabla_\Theta \mathcal{L} = \left(\frac{\partial \mathcal{L}}{\partial \Theta_1}, \frac{\partial \mathcal{L}}{\partial \Theta_2}, \ldots\right)$$

- **Gradient descent :** mise à jour de Θ itérativement dans la direction opposée du gradient afin de minimiser L (+back-prop)

Implémentation du gradient

Comment calculer le gradient avec nos machines ?

- le calcul manuel d'un gradient est dans la plupart des cas prohibitivement long
- un réseau de neurones peut avoir une structure très complexe et peu régulière
- il nous faut une solution capable de calculer le gradient automatiquement de façon dynamique

Nombres duaux

Une première approche simple : les nombres duaux

- un nombre dual est un nombre de la forme :

  $a + b\epsilon$

- avec : $\epsilon^2 = 0$
- epsilon peut être interprétée comme une valeur infinitésimale non nulle

Avantages :

- une structure de donnée facile à coder (similaire à un nombre complexe)
- interprétation géométrique plaisante

Inconvénients :

- pas de généralisation aux fonctions irrationnelles (log, exp, ...)
- calculs prohibitifs pour de larges réseaux

Graphe de calcul

L'approche moderne : le graphe de calcul et la dérivation automatique

- un graphe de calcul est un graphe orienté où les noeuds correspondent soit à des opérations ou des variables
  - les variables envoient leurs valeurs aux opérations
  - les opérations envoient leurs résultats à d'autres opérations
- le gradient est calculé en appliquant les règles de dérivation classiques et la chain-rule, à chaque opération récursivement
  (de bas en haut ou de haut en bas)

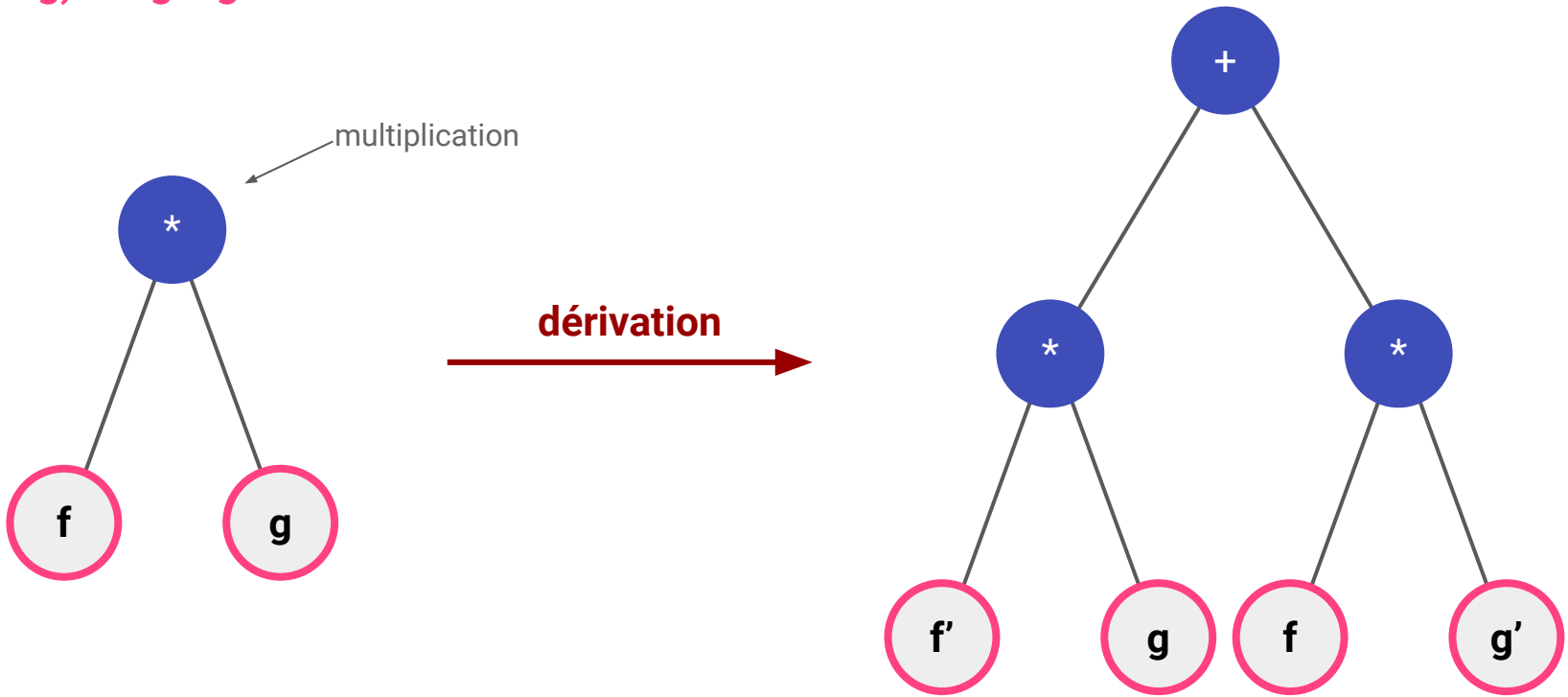
Question : à quoi ressemble le graphe de calcul de 3x + y ?

Illustration de la dérivation automatique

$(f + g)' = f' + g'$

Illustration de la dérivation automatique

$(f * g)' = f'g + g'f$

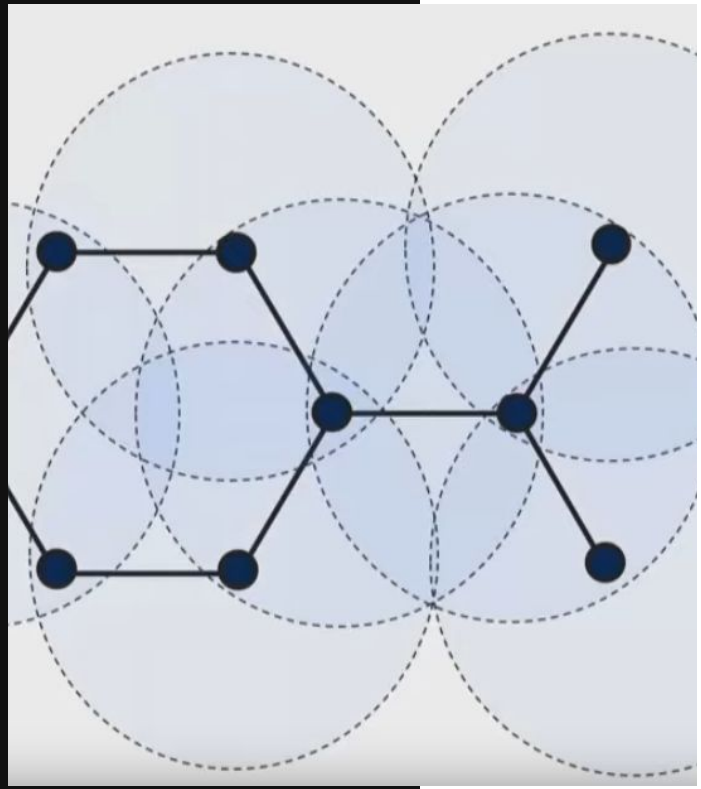


Graphe de calcul et dérivation automatique

Avantages :

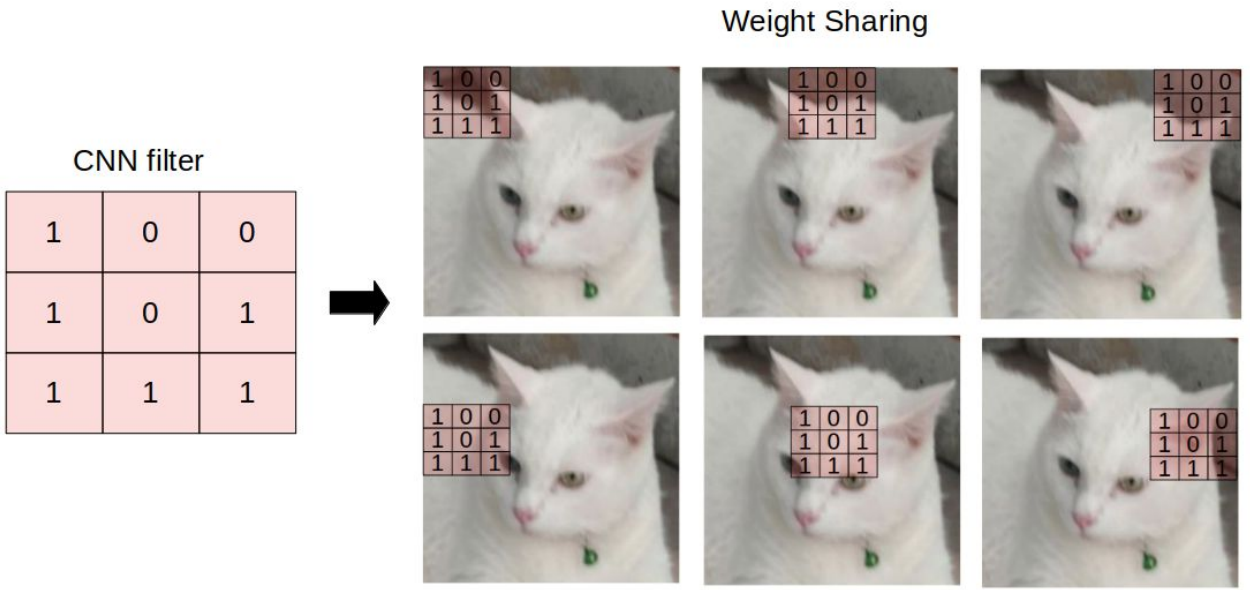
- permet le calcul de toutes les dérivées partielles en une passe
- le calcul peut être massivement optimisé
- une fois le graphe de calcul construit, tout calcul devient très rapide (Python vs C++)

Convolution sur les graphes : intuition

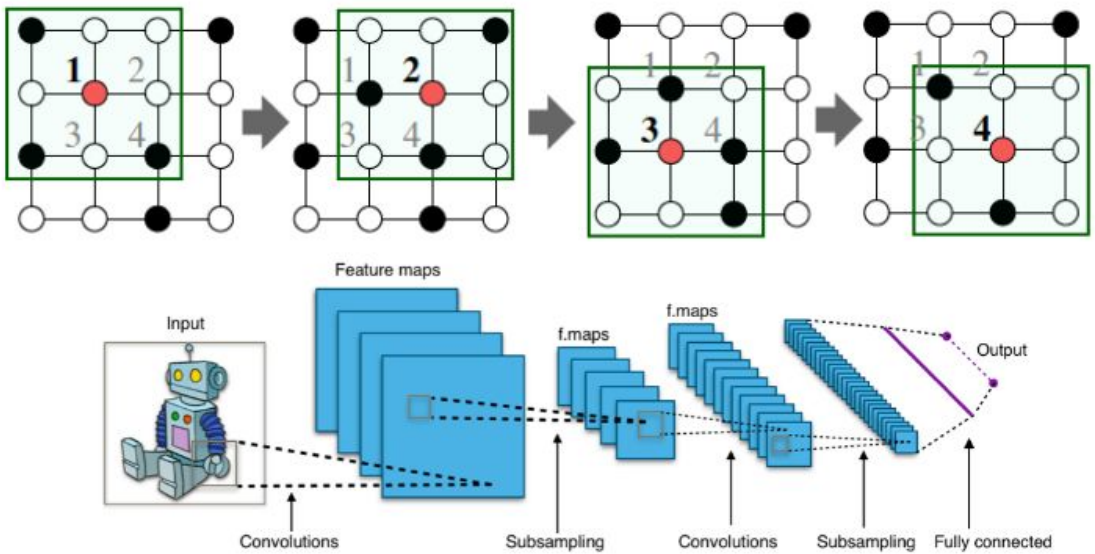


Revisiter les convolutional networks

Convolution classique : un filtre agit comme une fenêtre glissante sur l'ensemble de l'image et permet au CNN **d'incorporer les features des cellules voisines**



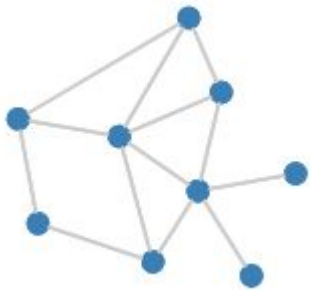
Revisiter les convolutional networks

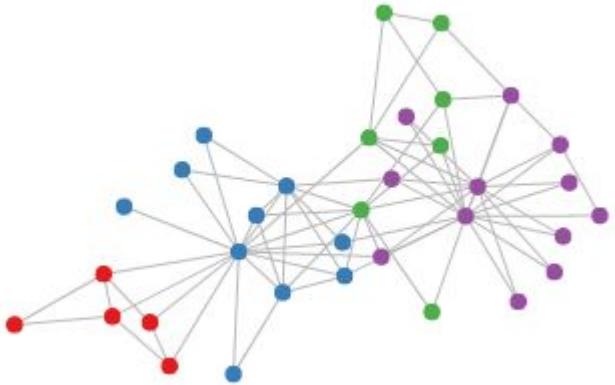


Notre objectif : généraliser les convolutions au delà d'une simple grille régulière, tout en exploitant les features/attributs de chaque noeud (texte, images, ...)

Revisiter les convolutional networks

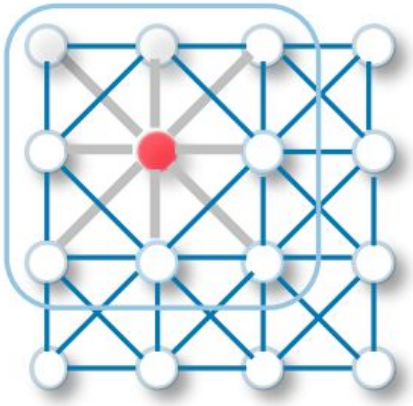
Comment faire cette convolution si notre graphe ressemble à ceux ci-dessous ?

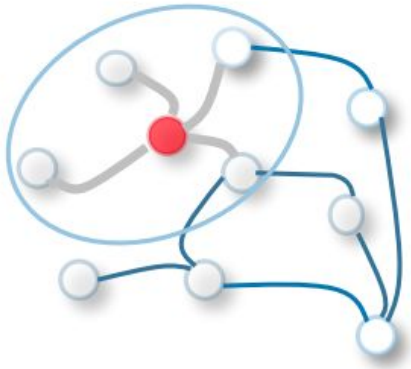




Revisiter les convolutional networks

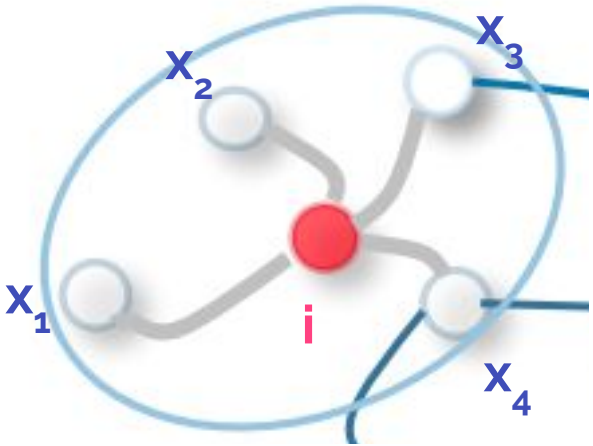
- La convolution classique peut être vue comme un cas particulier d'un graphe en grille.
- La convolution sur un graphe correspond donc à une somme pondérée des features des voisins d'un noeud.





Rappel : le voisinage est "encodé" dans la matrice d'adjacence

Intuition

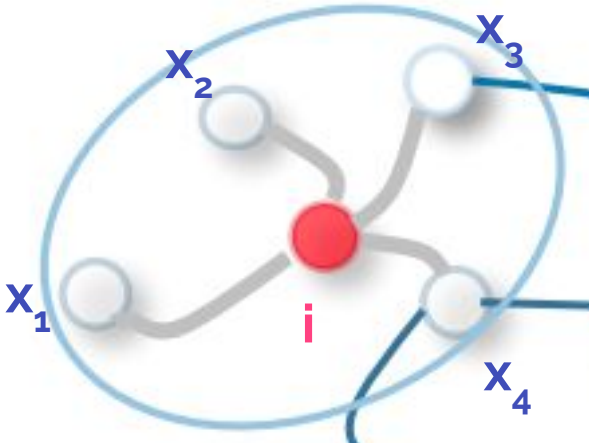


- Soit (x_j) les features des noeuds.
- On cherche à inclure dans les features de **i**
- les features de ses voisins.
- Une solution évidente, faire la somme des features : $x'_i = \sum_{j \in N_i} x_j$

Problèmes :

- n'inclut pas les features de **i**
- la somme des features implique un risque "d'explosion" des valeurs

Intuition



Problèmes :

- n'inclut pas les features de i
- la somme des features implique un risque "d'explosion" des valeurs

Solutions :

- on ajoute un lien de i vers i
- somme -> moyenne

Formulation matricielle

Ne pas oublier d'ajouter les boucles à notre graphe :

$$A \leftarrow A + I$$

Les features agrégées sont donc :

$$X' = D^{-1}AX$$

avec **D** la matrice des degrés du graphe avec boucles

X' est donc une nouvelle matrice de features, où chaque noeud contient une certaine information de ses voisins directs.

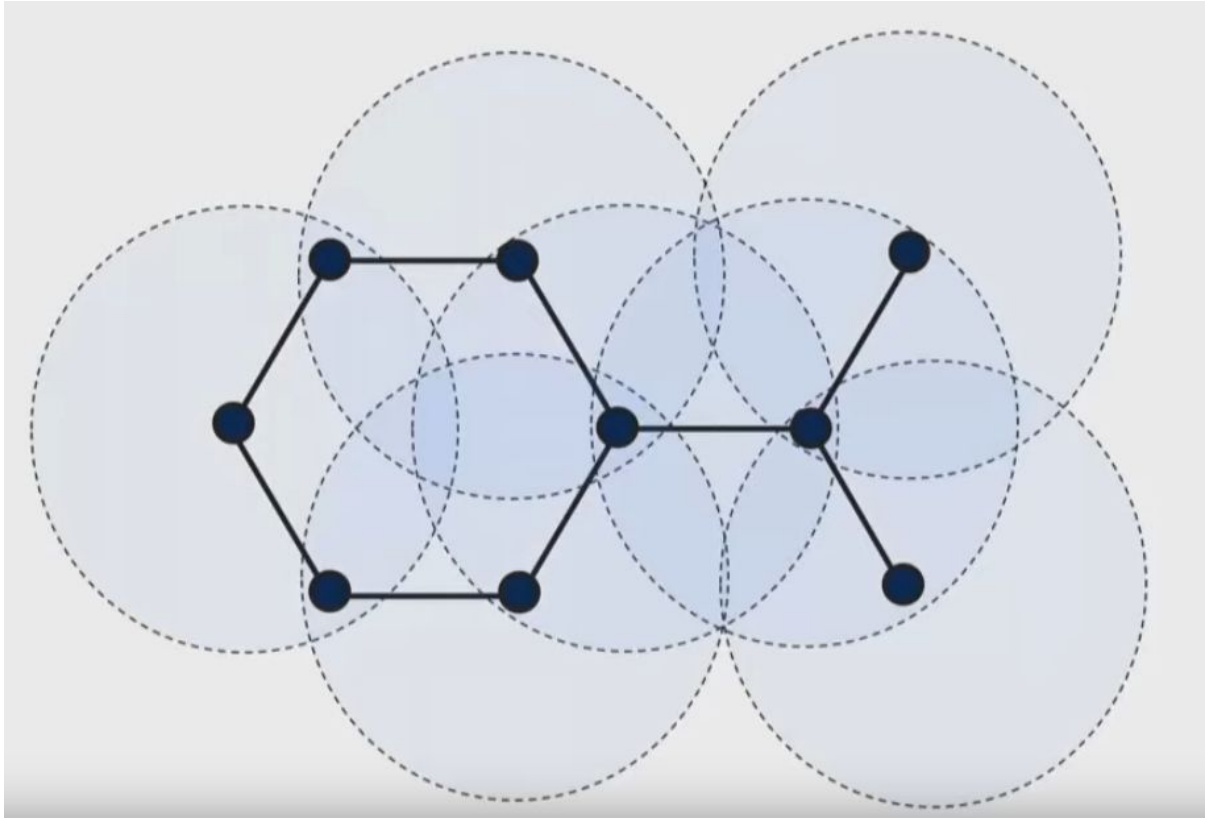
Formulation matricielle

- idéalement, nous cherchons à inclure les features des voisins indirects : ceux étant à une distance 2, 3, etc.
- nous pourrions appliquer le calcul précédent plusieurs fois, afin que le noeud **i** contienne l'information de ses voisins de voisins de voisins …
- nous aurions alors :

$$H^{(1)} = D^{-1}AX$$
$$H^{(2)} = D^{-1}AH^{(1)}$$
$$\dots$$
$$H^{(L)} = D^{-1}AH^{(L-1)}$$

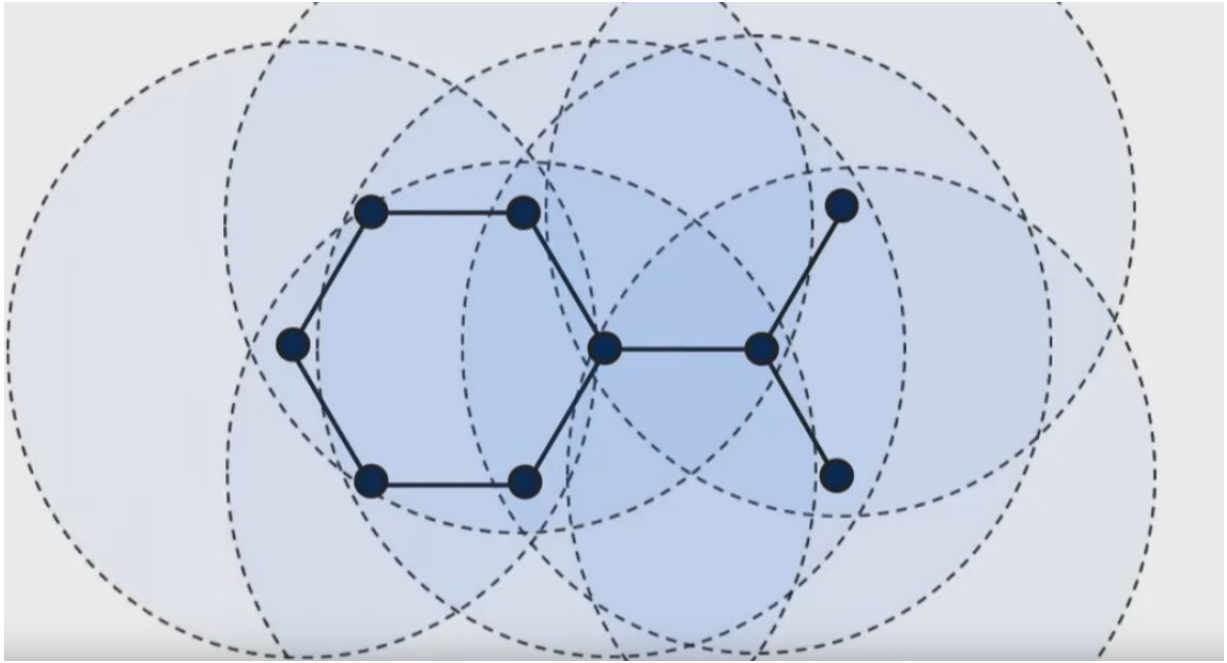
Cela rappelle-t-il un réseau de neurones à plusieurs couches ?

Formulation matricielle



Après une étape, chaque noeud “perçoit” les features de ses voisins directs.

Formulation matricielle



Après 2 étapes, chaque noeud “perçoit” les features des voisins de ses voisins.

En introduisant une fonction d'activation non-linéaire **σ** et une matrice de poids $\mathbf{W^{(k)}}$ pour chaque layer, nous obtenons :

$$H^{(1)} = \sigma(D^{-1}AXW^{(0)})$$
$$H^{(2)} = \sigma(D^{-1}AH^{(1)}W^{(1)})$$
$$\dots$$
$$H^{(L)} = \sigma(D^{-1}AH^{(L-1)}W^{(L-1)})$$

En pratique, $\mathbf{D^{-1}A}$ est remplacé par $\mathbf{D^{-½}\,AD^{-½}}$

- Et si **σ** est la fonction **ReLU**, cette formulation correspond à la méthode :

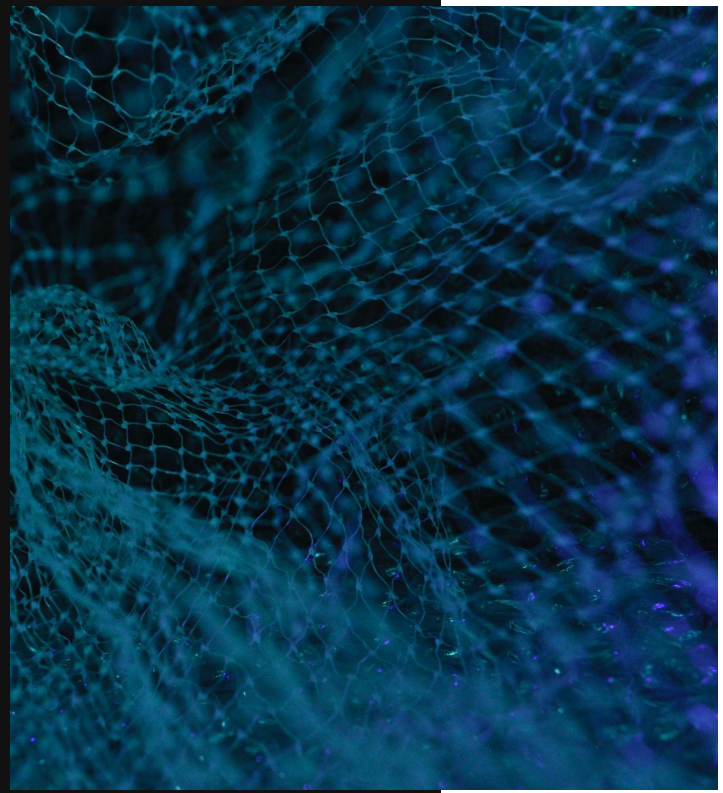
  Semi-supervised classification with GCN, de Kipf & Welling - 2017

La représentation de chaque layer **k + 1** est donnée par la formule suivante :

$$H^{(k+1)} = \sigma\left(D^{-\frac{1}{2}} A D^{-\frac{1}{2}} H^{(k)} W^{(k)}\right)$$

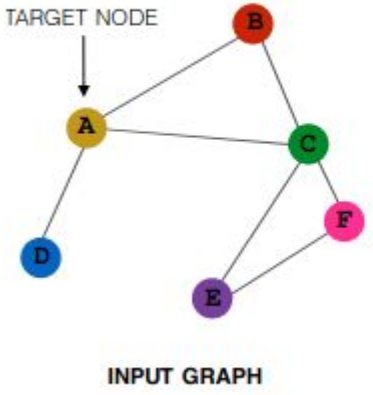
avec $H^{(0)} = X$, la matrice initiale des features.

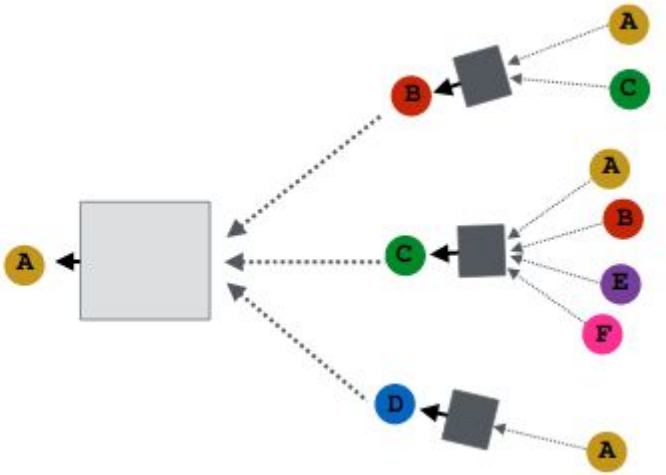
Graph neural networks



Idée générale : agrégation des voisins

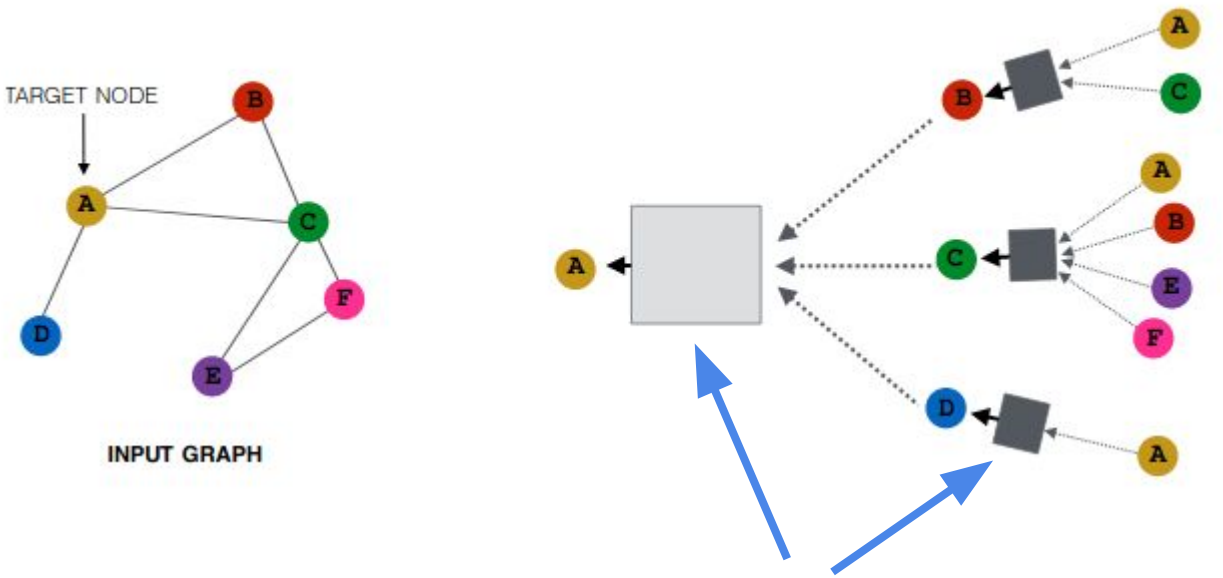
Génération d'embeddings des noeuds à partir de leurs voisinages





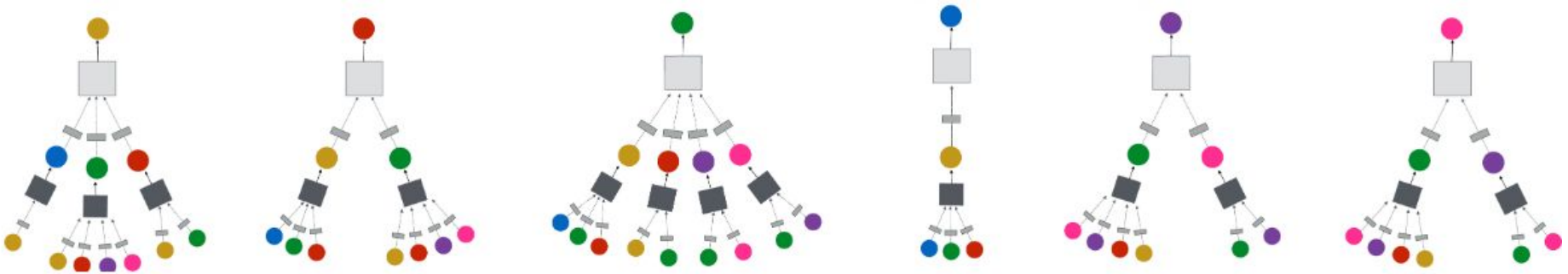
Idée générale : agrégation des voisins

Intuition : les noeuds agrègent l'information contenue chez leurs voisins grâce à des réseaux de neurones (message passing)



Idée générale : agrégation des voisins

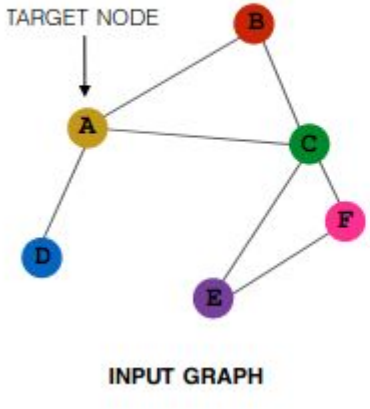
Chaque noeud dispose de son propre graphe de calcul

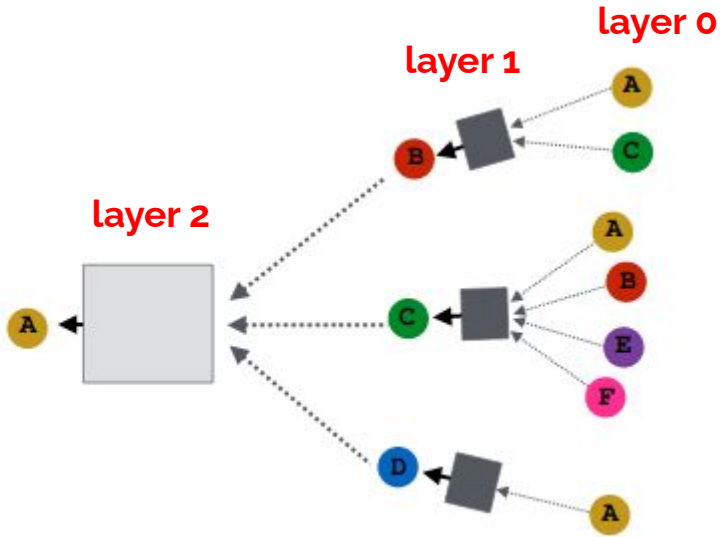


Modèle profond : plusieurs couches

Les modèles peuvent être de profondeur arbitraire :

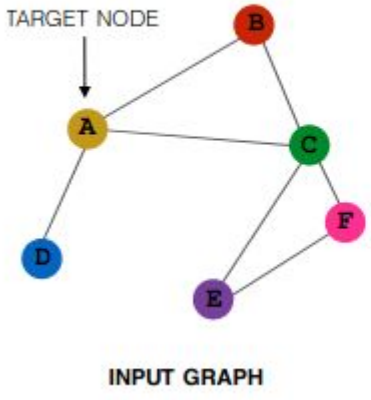
- les noeuds ont un embedding à chaque layer
- au layer **0**, les embeddings correspondent aux features **X**
- au layer **k**, les embeddings “reçoivent” l’information des noeuds qui sont à une distance de **k** sauts

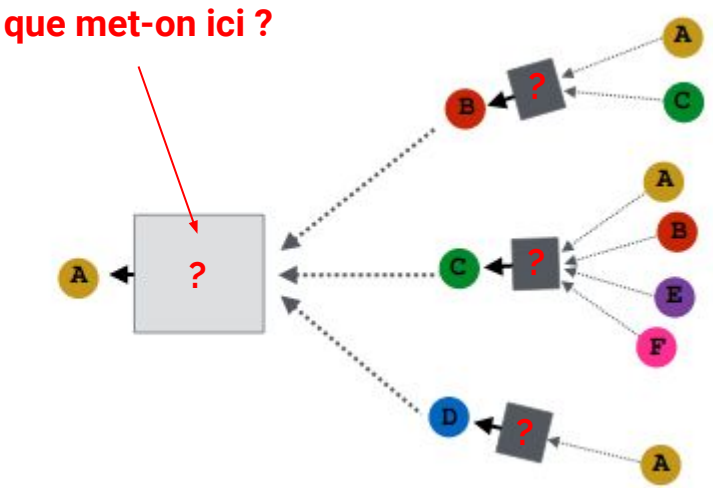




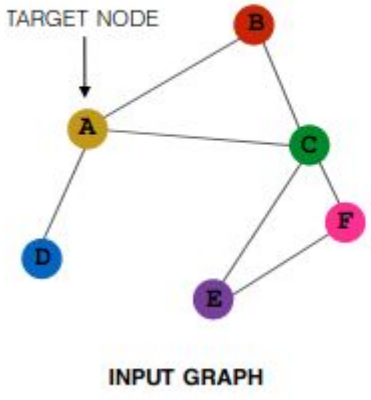
Modèle profond : plusieurs couches

Agrégation du voisinage : ce qui différencie chaque implémentation, c'est l'approche utilisée pour **agréger les voisins à travers chaque couche**.

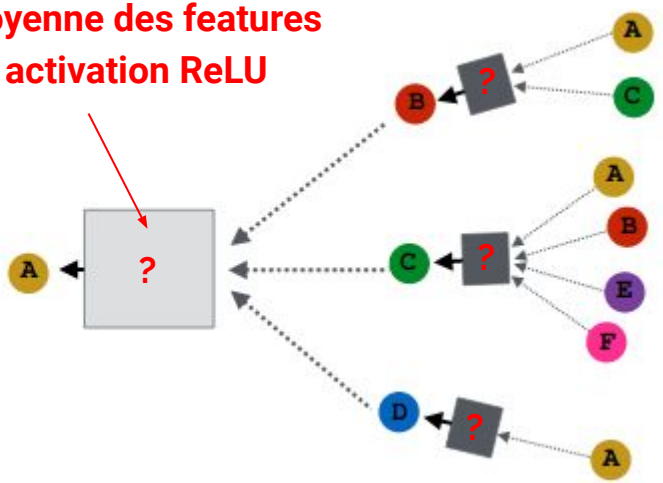




Approche vue précédemment, par Kipf & Welling



1. moyenne des features
2. activation ReLU



La moyenne des embeddings des voisins peut en général être calculée efficacement en exploitant le fait que la matrice d'adjacence est **creuse** :

$$\sum_{u \in N(v)} \frac{h_u^{(l-1)}}{|N(v)|} = D^{-1} A H^{(l-1)}$$

Ce calcul est permis notamment par l'apparition du produit matriciel entre une matrice creuse et une matrice dense sur **Tensorflow** (*tf.sparse.sparse_dense_matmul*).

GraphSage, Hamilton et al. - 2018

GraphSage distingue le noeud considéré de ses voisins,

puis concatène les deux embeddings.

Attention : cet algorithme ne considère pas v comme voisin de lui-même.

privé de v

$$h_v^l = \sigma([W_l \cdot agg(\{h_u^{(l-1)}, \forall u \in N(v)\}), B_l h_v^{(l-1)}])$$

agrégation des voisins

embedding du noeud lui-même

concaténation des deux embeddings

Plusieurs agrégations possibles

Moyenne : on agrège en faisant une moyenne pondérée des features des voisins.

$$agg = \sum_{u \in N(v)} \frac{h_u^{(l-1)}}{|N(v)|}$$

Pool : on transforme les embeddings des voisins, et on applique une fonction "élément par élément"

$$agg = \underbrace{\gamma}_{\text{max, min, avg, etc.}}(\{Qh_u^{(l-1)}, \forall u \in N(v)\})$$

LSTM : on applique un LSTM aux embeddings aléatoirement permutés des voisins

$$agg = LSTM([h_u^{(l-1)}, \forall u \in \pi(N(v))])$$

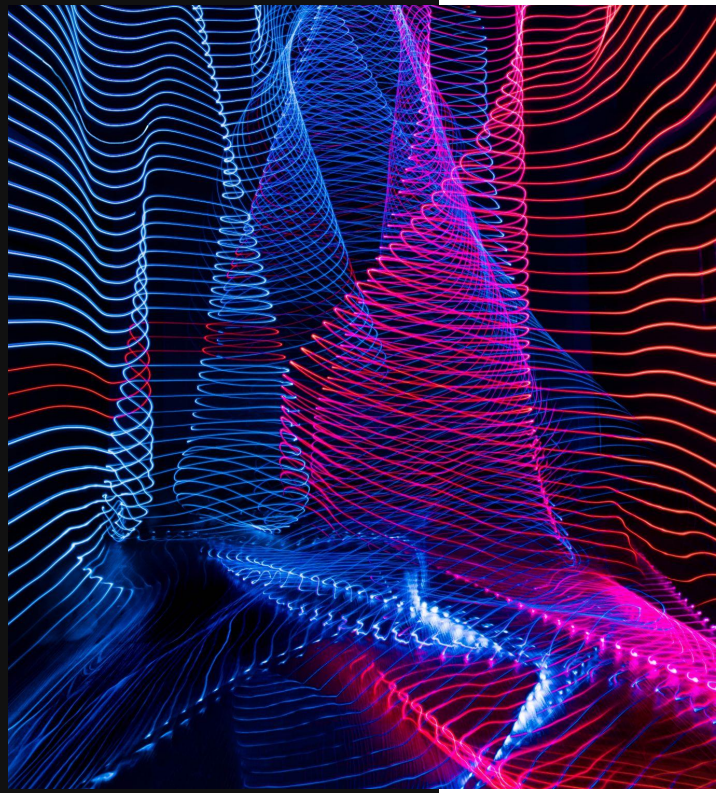
Récapitulons… Qu’est-ce qu’un GNN ?

- un GNN génère des embeddings à partir des features du voisinage d’un noeud
- les noeuds agrègent l’information contenue chez leurs voisins grâce à des réseaux de neurones
  - le GNN de Kipf & Welling agrège les features en faisant une moyenne
  - GraphSage agrège les features en concaténant l’agrégation des embeddings des voisins et l’embedding du noeud considéré
- un GCN incorpore plusieurs couches permettant à un noeud d’incorporer les features de voisins de voisins ...

Attention ! Deux notions de profondeur :

- la profondeur des réseaux de neurones d’agrégation
- la profondeur des graphes de calcul du GCN

Graph attention networks



Moyenne des embeddings

Récapitulons l'agrégation basique des embeddings :

$$h_v^{(l)} = \sigma(W_l \cdot \sum_{u \in N(v)} \frac{h_u^{(l-1)}}{|N(v)|})$$

- l'agrégation se fait sur tous les voisins de **v**
  (rappel : dans le cas général, **v** est voisin de lui-même).
- nous pouvons interpréter **1 / |N(v)|** comme un facteur de pondération d'un voisin **u**
  → Dans ce cas, tous les voisins sont de **v** sont également importants !

Peut-on faire mieux ? Peut-on imaginer un facteur de pondération défini implicitement, c'est-à-dire géré par le GNN lui-même ? (oui)

Notre objectif :

- assigner une pondération différente à chaque voisin d'un noeud d'un graphe
- on appelle ici cette pondération : α_{vu} (importance du voisin **u** de **v**).

Cela revient à calculer l'embedding de chaque noeud en suivant une stratégie "d'attention"

α_{vu} est calculé et directement optimisé par le GNN :

- calcul des coefficients d'attention (ou scores) e_{vu} de **(v, u)** basé sur leurs messages :

$$e_{vu} = \alpha(W_l \cdot h_u^{(l-1)}, W_l \cdot h_v^{(l-1)})$$

- e_{vu} indique l'importance du message du noeud **u** pour le noeud **v**. La fonction **α** est appelée mécanisme d'attention.
- on normalise les coefficients d'attention en utilisant la fonction softmax afin d'obtenir α_{vu} :

$$\alpha_{vu} = \sum_{u \in N(v)} \frac{exp(e_{vu})}{\sum_{k \in N(v)} exp(e_{vk})}$$

Graph Attention Network

- l'approche générale est agnostique de la fonction **α**

  ex : on peut utiliser un réseau de neurones pour la calculer
- **α** a des paramètres à estimer :
  - nous apprenons conjointement les matrices $\mathbf{W}_k$ et les paramètres de **α** en "end-to-end" (de bout en bout)
- à la manière des **transformers** (NLP), il est possible d'implémenter une "multi-head attention" :
  - plusieurs systèmes d'attention implémentés en parallèle et indépendamment
  - leurs résultats sont agrégés (concaténation, somme, moyenne)

Graph Attention Network

- computationnellement efficace :
  - le calcul des coefficients peut être parallélisé
  - l'agrégation peut être parallélisée
- économe en stockage :
  - les opérations sur des matrices creuses ne requièrent jamais plus de **O(|V| + |E|)** éléments à stocker
  - nombre de paramètres fixe, indépendant de la taille du graphe
- capacité **inductive** :
  - le calcul ne dépend pas de la structure globale du graphe et peut être appliqué à de nouveaux noeuds