AMAL - TP 11
Graph Neural Networks

Nicolas Baskiotis - Edouard Oyallon - Benjamin Piwowarski - Laure Soulier

2022-2023

Les graphes sont des structures particulières qui ne peuvent pas être traités en séquence (comme le texte, les séries temporelles, ...) et qui ne sont pas forcément régulières (comme les images, les grilles, ...). Ces structures sont utilisées dans beaucoup de domaines d'applications comme la bio-informatique ou les réseaux sociaux. Les tâches d'apprentissage les plus courantes sont la classification de nœuds, la prédiction de lien, la détection de communauté et l'étude de similarité entre sous-graphes.

Les approches classiques de Machine Learning utilisaient principalement du *feature engineering* pour réussir à représenter les informations d'un graphe dans un espace de représentation. Depuis l'avènement des approches de *Representation Learning*, un grand nombre de travaux s'est concentré à l'adaptation des techniques éprouvées en texte ou en image pour ces données structurées. L'objectif est bien sûr d'apprendre une représentation des nœuds de manière *end-to-end*, guidée par la tâche. Nous aborderons dans ce TP trois approches : la première qui regarde uniquement localement le voisinage d'un nœud pour apprendre la représentation, la seconde *node2vec* qui s'inspire de *word2vec* pour explorer plus en profondeur la topologie du graphe et enfin *Graph Convolutional Network* qui construit de manière itérative une séquence de représentations en fonction de la longueur des chemins considérés.

Nous utiliserons les données de MovieLens pour illustrer les algorithmes : il s'agit d'une liste de films notés par des utilisateurs. Nous allons modéliser de façon très simple un graphe entre les films en prenant en compte les notations des utilisateurs. Des modélisations beaucoup plus informatives sont possibles, vous êtes libres d'adapter le code fourni pour complexifier le traitement du problème et aboutir à un vrai problème de système de recommandation (en modélisant par exemple un graphe bipartite qui fait intervenir les utilisateurs et les films).

Le code pour la construction du graphe vous est fourni dans le fichier `utils.py` : chaque nœud du graphe est un film, et le poids de l'arête entre deux nœuds représente la *Pointwise Mutual Information* pondérée entre les deux films : $pmi(x, y) = log(\frac{x}{D}\frac{y}{D}) - log(\frac{xy}{D})$ avec

- x le nombre de fois qu'un utilisateur a noté le film x à un score supérieur à un seuil (ici pris à 5, note maximale)
- y le nombre de fois qu'un utilisateur a noté le film y à un score supérieur au seuil

— xy le nombre de fois qu'un utilisateur a noté les films x et y à un score supérieur au seuil
— D le nombre de notes supérieures au seuil

Pour éviter les effets des films rarement notés, la pmi est multipliée par xy. L'arête entre x et y aura donc un poids de $xy \times pmi(x, y)$.

Le module `networkx` nous sera utile dans ce TME pour implémenter les graphes. Vous aurez en particulier besoin des fonctions :
— `graph.neighbors(idx)` qui permet d'obtenir le voisinage du nœud `idx`
— `graph.nodes` qui permet d'obtenir la liste des nœuds d'un graphe

1 Triplet loss et représentation de la topologie locale

Dans cette section, nous nous intéressons pas aux poids du graphe mais uniquement à sa topologie : un film est jugé similaire à un autre si une arête les relie. L'objectif est de trouver une représentation telle que les films similaires aient des projections très proches et en même temps éloignées des films dissimilaires.

Pour construire cette représentation, on peut utiliser par exemple le coût *Triplet Loss* (utilisé en particulier pour les *siamese networks*). Ce coût utilise un triplet de nœuds a, p, n : a l'ancre, p un nœud positif i.e. dans le voisinage de a, et n un nœud négatif, i.e. qui n'est pas relié à a. La fonction de coût s'écrit

$$L(a, p, n) = max(0, d(f_\theta(a), f_\theta(p)) - d(f_\theta(a), f_\theta(n)) + m)$$

avec f_θ la représentation apprise, d une distance (euclidienne ou cosine par exemple) et m un scalaire positif représentant une marge. Il vise à écarter les nœuds non connectés et à rapprocher les nœuds connectés.

Question 1

Ecrire une classe `TripletDataset` qui renvoie un triplet de nœuds `ancre, positif, négatif` (ou le positif et le négatif sont tirés au hasard dans le voisinage et hors du voisinage de l'ancre).
Implémenter la boucle d'apprentissage pour la triplet loss et apprendre une représentation des films. Vous pouvez visualiser la représentation avec l'algorithme t-sne (en particulier dans `tensorboard` avec la fonction `add_embedding`).
Pour quelques films, observer quels sont les films ayant les représentations les plus proches d'eux.

Un des gros défaut de la triplet loss est qu'elle nécessite que les négatifs soient "bien" tirés : si que des points éloignés sont choisis, alors le coût n'évolue plus. Il est possible de biaiser la sélection des négatifs pour ne sélectionner que des négatifs "utiles" dans le voisinage de l'ancre.

2 Node2Vec

L'approche *Node2Vec* est inspirée de *word2vec* en texte et plus exactement du modèle *Skip-Gram*. Très succinctement, ce modèle vise à prédire à partir d'un mot son contexte (les mots autour de lui que l'on retrouve dans les phrases du corpus). L'apprentissage se déroule de la manière suivante : soit une phrase $t_1 t_2 t_3 \dots t_L$, une fonction d 'embedding du mot cible f_{in} et une fonction d'embedding d'un mot du contexte f_{cont}, et une fenêtre w, pour une position i donnée les couples (t_i, t_{i+j}) sont construits avec $-w \leq j \leq w$ (et $j \neq 0$). L'objectif est de maximiser $p(t_{i-w}, \dots t_{i+w}|t_i) = \prod_{-w \leq j \leq w, j \neq 0} p(t_{i+j}|t_i)$, modélisé par

$$p(t_{i+j}|t_i) = \frac{e^{f_{in}(t_i) \cdot f_{cont}(t_{i+j})}}{\sum_t e^{f_{in}(t_i) \cdot f_{cont}(t)}}$$

Comme optimiser cette fonction de coût est impossible du fait du softmax sur tout le vocabulaire, on utilise le *negative sampling* pour approximer le coût. Dans cette approximation, la fonction de coût est

$$\sum_{-w \leq j \leq w, j \neq 0} \left[log\left(\sigma\left(f_{in}(t_i) \cdot f_{cont}(t_{i+j})\right)\right) + \sum_{k=1}^{N} log\left(\sigma\left(-f_{in}(t_i) \cdot f_{cont}(\tilde{t}_k)\right)\right) \right]$$

, avec $\tilde{t}_k$ N mots tirés au hasard dans le vocabulaire (supposés négatifs).

Pour pouvoir utiliser le modèle Skip-gram sur un graphe où les nœuds représenteraient les tokens, il faut pouvoir le représenter sous forme d'une séquence. L'idée est d'échantillonner à partir de tous les nœuds du graphe des chemins de manière aléatoire et que cet ensemble de chemins constituent les "phrases" du corpus : la fréquence d'apparition d'un nœud dans le contexte d'un autre renseigne ainsi sur la connectivité entre ces deux nœuds et la topologie est ainsi prise en compte partiellement.

La marche aléatoire uniforme dans un graphe souffre cependant de certaines limitations en fonction de la topologie du graphe (nœuds centraux trop privilégies et redondants en particulier), c'est pour cela que l'approche privilégie une marche aléatoire biaisée permettant d'alterner entre un régime de parcours en largeur où les nœuds voisins sont parcourus et un régime de parcours en profondeur qui permet de s'écarter plus du nœud initial. Un paramètre p permet de contrôler la probabilité de revenir sur le nœud précédemment visité (une faible valeur va engendrer un chemin qui ne fait que cycler entre deux nœuds alors qu'une grosse valeur empêchera tout retour en arrière), et un paramètre q permet d'explorer plutôt des nœuds voisins de nœuds déjà visités ou au contraire de s'éloigner des nœuds connus (pour une valeur faible du paramètre). Le code de cet algorithme vous est fourni dans `utils.py`.

Question 2

Implémenter l'algorithme `Node2Vec` et analyser les représentations apprises.

3 Graph Neural Networks (GNN)

L'objectif des GNNs est d'apprendre une représentation des nœuds itérativement : à partir d'une représentation initiale (qui peut être un one-hot encoding ou une représentation vectorielle du nœud, tel que des features le décrivant - ce que ne permettaient pas les approches précédentes), une première couche permet d'apprendre une nouvelle représentation en aggrégeant les informations des nœuds adjacents et de lui-même. Cette représentation est à son tour transformée par une deuxième couche, puis une troisième et ainsi de suite. Ce processus permet de prendre en compte au fur et à mesure des informations qui viennent de nœuds de plus en plus éloignés du nœud initial : la première couche n'intègre que les informations du voisinage locale, la seconde cependant peut prendre en compte le voisinage des voisins du nœud et ainsi de suite. C'est une approche inspirée des réseaux de convolutions, l'aggregation de l'information que fait une couche pour un nœud peut s'apparenter à un filtre centré en ce nœud.

Soit h_i^{l-1} la représentation du nœud i à la couche $l-1$, la forme générique pour obtenir la représentation de la couche $l+1$ est la suivante :

$$h_i^l = g\left(W_l \sum_{j \in V_i} \frac{h_j^{l-1}}{|V_i|} + B_l h_i^{l-1}\right)$$

avec V_i le voisinage du nœud i et g une non-linéarité telle qu'une ReLU par exemple. L'opération sur les voisins est équivalent à un average-pooling de leur représentation.

Cette équation définie le modèle du réseau. Pour son entraînement, tout type de méthode et de fonction de coût peut être utilisé, par exemple dans notre cas la même marche aléatoire qu'à la section précédente.

Une version un peu plus stable - le Graph Convolutional Network - normalise l'opération de pooling en fonction du voisinage :

$$h_i^l = g\left(W_l \sum_{j \in V_i \bigcup \{i\}} \frac{h_j^{l-1}}{\sqrt{|V_i||V_j|}}\right)$$

et à l'avantage d'être plus efficiente à calculer en utilisant la matrice d'adjacence A et la matrice diagonales des degrés D tel que $D_{ii} = \sum_j A_{ij}$:

$$H^l = g\left(D^{-\frac{1}{2}}(A+I)D^{-\frac{1}{2}}H^{l-1}W_l\right)$$

Question 3

Implémenter un GNN et comparer les représentations obtenues.