Rapport - Stage Cardiff

Antao Amory

Lundi 10-06 :

Mardi 11-06 :

Mercredi 12-06 :

Jeudi 13-06 :

Vendredi 14-06 :

Matin ça clique. Meeting avec Steven Schockaert, il a recentré le truc sur les transformers.
Prochaine étape : écrire un transformer from scratch, les chemins dans le graphe de connaissance vont être sous la forme de séquences (quoi que ça veuille dire), et après il faudras voir pour les marches aléatoires (où c'est nécessaire, comment l'implémenter, à quoi ça sert ? ?).

Bon pour le transformer, il va falloir le faire from scratch (car c'est tricher si on en utilise un pré-entrainé). Attention, il existe deux principales tâches pour ces transformers : soit on a une phrase et on masque un mot et il faut le retrouver (ça, ça ne nous intéresse pas) ou soit il faut déterminer comment la va être la suite d'une phrase (ça ça nous intéresse !).

Pour m'introduire au sujet des transformers, je vais regarder :
https ://jalammar.github.io/illustrated-transformer/
(et en particulier la traduction française.
On va résumer ça ici :

Le seq2seq et le processus d'attention Un modèle seq2seq est, comme son nom l'indique, un modèle qui prend une séquence (de mots, de lettres, ou de plus ou moins tout ce que l'on veut) et qui en resort une autre.

Le modèle est décomposé en deux parties : un encoder et un décoder.
L'encoder prend la séquence en entrée, traite chaque élément et met le résultat (à priori un vecteur de flottants) de ce traitement dans un vecteur (appelé context), qu'il fait passer au décoder, qui lui décode le context item par item pour réobtenir une séquence.

En pratique, l'encoder et le décoder sont souvent des réseaux de neuronnes récurrents (RNN). Un RNN prend deux entrées à chaque pas de temps, une entrée classique (ex : un élément de notre séquence) et un état caché. De même, il produit une sortie classique et l'état caché suivant (celui que l'on réutilise, d'où le récurrent).
Dans notre cas, le dernier état caché produit par l'encoder va être le context que l'on fourni au décoder.
Dans le contexte des traductions automatique, le problème était qu'avec de longues phrases, les différentes étapes du décoder ne pouvait pas bien prendre en compte les données importantes.

C'est pourquoi le concept d'attention a été créé. Il a permit d'amplifier le signal des parties importantes de la séquence d'entré suivant à quelle étape du décoder on était.
Le modèle nouveau modèle ainsi créé, modèle d'attention, a deux principales différences avec le modèle seq2seq.
1 - L'encoder ne passe plus uniquement que le dernier état caché mais tous les états cachés intermédiaire (beaucoup plus de données).
2 - Le décoder donne un score (un peu mystérieux pour le moment) à chaque état caché puis applique un softmax (pour si le score est s_i alors $\frac{e^{s_i}}{\sum e^{s_j}}$), et prend ce nouveau vecteur comme context.

Ainsi au final, le RNN du décoder fait comme précédemment, sauf que sa sortie n'est pas la sortie définitive. En effet on applique l'étape de l'attention, et au colle le vecteur que l'on obtient à la précédente sortie. On fait passer le tout dans un réseau neuronal feed-forward (entraîné en même temps que le modèle), et cette sortie là est la bonne.

Illustration du word embedding et de word2vec

Illustration du transformer Le transformer est est composé d'une pile d'encoders et d'une pile de décoders de même nombre.
Tous les encoders ont une structure identique : ils sont divisés en deux sous-couches, d'abord la self-attention puis le feed forward.
Les décoders ont la même structure à laquelle on a ajouté une couche d'attention (comme dans le seq2seq) qui est entre la self-attention et le feed forward.

Tout d'abord, chaque mot d'entrée est transformé en un vecteur (embedding).

Tous les mots d'entrée avances selon leurs propre chemin, et sans dépandance lors de la couche de feed forward (on peut donc exécuter les différents chemins en parallèle dans ces couches).

La couche de sefl-attention a pour objectif d'améliorer la compréhension, le sens du mot qu'il traite (comment il est relié sémantiquement aux autres mots), par exemple dans "The animal didn't cross the street because it was too tired", ça a pour vocation de relier le mot "it" à "The animal" (et non à "street" ou a autre chose).
En pratique, la self-attention crée trois vecteurs pour chaque vecteur d'entrée, la query, la key et la value, en multipliant le vecteur d'entrée par trois matrices obtenue par le processus d'entraînement.
Une fois ces trois vecteurs obtenues, on calcule un score pour chaque mot (représentant le degré de concentration à placer sur les autres mots) en prenant le produit scalaire de la query de l'entrée et de la key du mot dont on calcule le score.
On divise ces scores par la racine carré de la dimension des vecteurs, puis on applique une la fonction softmax (ce que l'on obtient correspond à l'importance (entre 0 et 1) d'un mot vis à vis du mot d'entrée).
Pour finir, on somme le produit de la value et du softmax, pour que la value des mots important soit plus pris en compte que celle des autres mots.
Une fois fait cela pour toute les entrées, on a terminé la couche de self-attention.

Cependant, les calculs avec les vecteurs sont long, donc en pratique on traite avec des matrices pour gagner en efficacité.
Pour ce faire, on concatène toutes les entrées en une matrice, que l'on multiplie par les matrices que l'on a entraîné pour calculer la value, la key et la query (qui sont donc maintenant des matrices, et non plus des vecteurs). On peut alors rassembler toutes les étapes restantes en une seule.
Si, si Q est la query, K la key, V la value et d la dimension des vecteurs, on calcule la sortie Z par $Z = \text{softmax}(\frac{Q \times K^T}{\sqrt{d}})V$.
De plus, on peut encore améliorer cela en considérer plusieurs têtes (8). Le principe est que l'on crée dans chaque tête une matrice Q, une matrice K et une matrice V (qui sont différente, puisque on considère que les matrices permettant des les obtenirs sont choisie aléatoirement au départ). Il ne suffit plus que de les refusionner en une seule matrice (pour la sortie), ce que l'on fait en concaténant toutes les sorties puis en les multipliant par une matrice W^0 entraîné parallèlement.

D'autre choses en vrac, important en pratique mais pas pour comprendre. Il est utile de rajouté en entrée un encodage positionnel. Il y a une formule, c'est assez bizarre. De plus, entre chaque couche il y a une partie de normalisation.

Pour le décoder, tout est assez similaire même si rien n'est pareil...

Le décoder renvoie un vecteur de flottant. Il faut donc le traiter et pour se faire on a deux couches finales : linéaire et softmax. La couche linéaire est un réseau neuronal qui projette le vecteur de sortie dans un vecteur de logits, aussi large que tout le vocabulaire que l'on connait, et qui va lui donner un score, que l'on transforme en probabilité par la couche softmax (et le mot obtenu est celui de probabilité la plus grande).

Pour entraîner le modèle, l'on doit déjà connaître l'ensemble de notre vocabulaire. Une fois connu, on peut créer un vecteur correspondant à chaque mot (le vecteur est le one-hot, équivalent de l'indicatrice).
On peut alors faire tourner le modèle pour obtenir un vecteur de sortie, et faire du rétro-pédalage dans le but de modifier les poids pour être plus proche de la réalité.

Lundi 17-06 :

Implémentation d'un transformer : Je suis un tuto youtube.
Les mots que je veux c'est une suite de relation formant un chemin dans notre KG.
L'objectif est de trouver le mot suivant dans la séquence.
J'implémente, ça ne marche pas hyper bien.
Je regarde un autre tuto, sur une page, pour un translater d'anglais vers kannada.
Je suis pas à pas pour comprendre les étapes.
Ça marche quasiment directement (seulement quelques debug, je suis content.

Mardi 18-06 :

Je continu sur le traducteur. En réalité ça ne marche pas tout à fait bien.
Je l'ai fait tourner sur le supercalculateur la nuit et j'ai eu des erreurs.
Je cherche et je trouve un problème avec les tokens (ils étaient tous codé par "").
Après je fais tourner et ça marche bien. Le problème c'est que je ne connais pas le kannada.
Je vais donc chercher des datas pour la traduction anglais-français (kaggle).
Je traite les données pour avoir ce que je veux, et je modifie légèrement le code.
Je lance et ça marche plutôt bien.
Là je suis content :).
Ok mais ce que je veux moi c'est pas un traducteur mais un truc sur les graphes de connaissance.
Je reprend le transformer-traducteur comme base.
En vocabulaire je met la liste des relations et la liste des noeuds (important).
Pour se faire je doit traiter les données.

Après j'importe mes données de trainning et je fais une marche aléatoire pour obtenir une liste de chemin entre des points.
Première version : complètement aléatoire mais ça n'a aucune chance de donner quoi que ce soit.
Second version : j'utilise networkx et pour mettre une distance entre deux points et pouvoir pondérer mes probabilités.
Je fais en fonction du plus court chemin jusqu'au noeud d'arrivé (avec la fonction nx.shortest_path_length).
Je normalise après avec un sorfmax (où le poids est multiplié par deux car desfois il y a vraiment beaucoup de voisins).
Le random walk marche très bien, mais en réalité c'est tout bonnement inenvisageable car beaucoup trop long à calculer sur de grandes entrées.
Après je fais une fonction pour obtenir des random walk entre point aléatoire.
Et je vais dormir.

Mercredi 19-06 :

Je continu de faire le transformer pour le KG.
J'ai créer le dataset, et je suis en train de coriger les erreurs dans le transformer pour pouvoir lancer les epoch comme il faut. Pour le moment tout le reste marche comme il faut.
Pas grand chose à dire car c'est pas mal de débuggage mais ça avance bien donc c'est cool.
Il faut notamment bien faire gaffe à la taille des input et des output.
En effet je veux que les input soit de taille 2 (noeud de départ et noeud d'arrivé) tandis que les output sont des séquences de relations, de taille variable (normalisé à 200 car c'est bien mieux si elles ont toutes les même tailles).
Autre chose tricky à géré c'est lorsqu'il y a plusieurs relations différentes entre les mêmes noeuds. Ça m'a pris pas mal de temps.
J'ai géré en choisissant aléatoirement quel relation je prennais. Peut être d'autre manière où les probas sont pondéré (ex : si une relation est beaucoup plus présente dans le graphe que l'autre) mais on verra plus tard si jamais.
Autre problème d'avoir les inputs et les outputs de taille différente c'est pour la MultiHeadCrossAttention

Jeudi 20-06 :

J'ai tout mis au propre dans un github, c'est mieux !
J'ai séparer mes fichier pour avoir un module_transformer et un truc avec les données, c'est mieux.
Je trouves quelques bug, c'est chiant, ça prend du temps, mais ils se corrigent.
J'ai rajouter une fonction de prédiction.
Meeting avec Akash : Je dois rajouter les relations inverse.

En entrée j'ai pas d'entitéx, mais juste une relation car le but c'est de trouver une séquence de relation qui donne la même chose. Exemple :
relation : Parle_langue.
prédiction : Habite, Langue_officielle.
Ça permet de sortir la règle :
Habite(X,Y) $\wedge$ Langue_officielle(Y,Z) $\rightarrow$ Parle_langue(X,Z).x

Je suppose donc que nos random walk prennent deux noeuds en relations et fond des walks un peut autour avant de revenir. Va falloir que je regarde comment faire ça...

Ok j'ai essayer de me rebaser sur nn.Transformer mais c'est vraiment bizarre.
J'ai viré les noeuds en entré pour prendre juste des relation.
Je modifie en conséquence tout mes trucs.
J'accroche pas mal au niveau des masques.
À priori un masque est de taille batch $\times$ tailleTxt $\times$ tailleTxt.
Pour le cross, je sais pas si c'est le plus petit ou le plus gros texte.
Pour les autres déjà :
Après embedding, les données sont de taille : batch $\times$ tailleTxt $\times$ d_model
Notre masque est lui de taille : batch $\times$ tailleTxt $\times$ tailleTxt.
Ok j'ai la ref le masque n'est utilisé que dans le calcul avec les tenseur key, query et value.

Ok à priori j'ai de bon masques. Je me suis pas mal aidé visuellement de cette question dans un forum :
https ://datascience.stackexchange.com/questions/126187/cross-attention-mask-in-transformers

Maintenant il faut que je modifie encore deux trois trucs dans le modèle de transformer et après ça sera OK je pense.
Ah et la marche aléatoire aussi.

Vendredi 21-06 :

J'ai commencé par terminer le modèle du transformer pour prendre en entrée une relation au lieu de deux noeuds (potentiellement éloignées).
J'ai aussi rajouter les relations inverse dans la liste des relations.
Ça marche plutôt bien, je suis pas mal content, je vois assez bien comment ça marche maintenant.
Maintenant il faudrait peu être que je m'attaque à l'histoire des random walk. Je ne suis pas exactement comme faire encore.
L'ojectif est de rentrer une relation (prise aléatoirement), de choisir (toujours aléatoirement) deux entités liés par cette relation, puis trouver un chemin reliant ces deux entités.

Je vais déjà tester l'implémentation hyper basique. Full aléatoire mais chemins limité à genre 4 de longueur (en vérifiant bien que je ne mette pas la relation de base (et potentiellement éviter aussi, si la relation est r, d'avoir un chemin x - x-1 - r)).
Ok ça ne marchait pas vraiment. Je garde les probababilités en fonction de la distance à l'arrivé, avec un softmax, mais je fais attention à empêcher la relation d'être prise en premier.
De plus j'ai stocker mes grosses données, le graphe de train, les index, dans des fichier.pickel.
Ça semble fonctioner mais c'est vraiment hyper lent, surtout en mode débuggage.
Ok j'ai trouver comment faire avec networkx. Je peut créer un générateur avec nx.all_simpls_paths puis prendre un chemin aléatoire (jepeux même limiter la taille).
Réunion avec Stephan Schockaert.
C'est pas trop mal, ça avance. Là je fini avec mon transformer.
Si je teste sur (h,r,t) il faut que j'empêche de prendre cette relation dans le chemin.
De plus, il faut que j'empêche d'avoir un truc du style (n, r, n') suivit de (n', r-1, n) (mais si c'est une autre relation c'est ok et si les entités sont différentes c'est ok aussi).
Il faudra ensuite que je crée un autre transformer pour classifier mes chemins et voir les quels modélisent réellement une relation et lesquels sont beaucoup mais sûr.
J'ai aussi changé de data, ça devrait être bien plus rapide.
Mon job pour la prochaine semaine est de finir ça (le random walk qui marche bien, avec les contraintes souhaités) puis de voir quels doivent être les bons hyperparamètres pour le transformer (nombre de têtes, de layers, ...).
Ça risque d'être un poil frustrant mais ça avante.
Je devrais ensuite tester mon transformer (sur une partie des chemins aléatoire), puis créé le second comme dit précédemment.
Si j'ai du temps / de la motivation, je pourrais aussi regarder les méthodes LSTM.
Et puis il sera l'heure de s'attaquer aux règles de productions. Potentiellement avec AnyBURL, potentiellement avec autre chose.

On j'ai trouver une méthode pour le random walk. Je n'utilise pas d'algo compliqué de netorkx, mais juste les trucs basique. C'est surement pas ouf mais ça semble marche plus ou moins et ça va assez vite.
J'ai tout bon maintenant.
Je lance.
Et pouf, mes predictions sont toutes à zero, je ne sais pas trop pourquoi.
Bon il est quasi 19h.
J'arête, j'aurais peu être la motivation de continuer ce weekend.