Générateur de Texte de Critical Role

Implémentation d'un réseau de neurones à base d'un Transformer

0. Introduction

Le but de ce projet est double:

- En termes de NLP, la tâche visée est d'entraîner un model de machine learning, basé sur un Transfomer, à générer du texte de l'émission Critical Role.
- En termes pédagogiques, l'objectif est d'apprendre à maîtriser un transformer et tous les aspects liés à l'entraînement d'un réseau de neurones

I. Données

Critical Role est une émission américaine ayant lieu tous les jeudis en live sur la chaîne Twitch.tv de Critical Role, et dans lequel des comédiens de doublage professionnels jouent au jeu de rôle sur table (ou 'TTRPG' pour 'Tabletop Role-Playing Game) *Donjons et Dragons*. Les premiers épisodes de la première saison (ou 'campagne', pour employer le vocabulaire des TTRPG) eurent lieu en mars 2015, et la seconde saison est encore en cours aujourd'hui. (Voir la section **Données** pour plus de détails). L'émission n'est pas scriptée, et repose uniquement sur les plans du maître du jeu, Matthew Mercer, et sur l'improvisation des différents joueurs.

Le jeu de données utilisé ici provient du projet bénévole Critical Role Transcript (abrégé en crT), qui a, à des fins originelles de sous-titrage, retranscrit la totalité des épisodes de la première campagne et les épisodes 1 à 50 de la seconde campagne, ainsi que divers épisodes spéciaux (tels que les 'one-shots' par exemple).

L'ensemble des transcripts est disponible sur le site de crT : https://crtranscript.tumblr.com/

De l'ensemble des transcripts original trouvable sur le site de crT, on retirera les transcriptions qui ne proviennent pas d'épisodes de Critical Role, telles que

celles des épisodes de l'émission Talks Machina, ou celles des interviews live à diverses conventions.

Cela nous amène à un dataset ayant notamment les quelques propriétés données à titre d'information ci-dessous, mis en comparaison à côté des datasets Wikitext-2 et Wikitext-103, utilisés pour le language modeling eux aussi.

	CRTranscripts	Wikitext-2	Wikitext-103
Langue	Anglais		
Nombre de fichiers	180	720	28 595
Taille totale	31 Mo	4,3 Mo	181 Mo
Taille du vocabulaire	45 504 tokens	33 278 tokens	267 735 tokens
Longueur moyenne d'une phrase	~10 tokens	~20 tokens	
Score d'entropie croisée atteint	?	5,29	4,036

Il est notable que notre vocabulaire est relativement restreint: il n'est pas beaucoup plus fourni que celui de Wikitext-2, alors que notre dataset est plus de sept fois plus large. Notons également que nos phrases sont en général très courtes, ce qui s'explique par la nature orale de Critical Role.

Voyons à quoi ressemblent nos données :

[...]
MATT: *She puts her hand on the table and Mertin comes over and hands you what looks like a small leather chest. It's made of sculpted leather. Rolls back, and pushes it forward. You can hear the clinking of glass as he does so, and on the inside you can see there are four glass vials, prepared with some sort of a swirling red, silver-ish liquid inside. You recognize these as healing-based potions.*

SAM: *Like Moderate Healing?*
LAURA: *Superior? Perhaps.*
MATT: *These would constitute as Greater Healing Potions, and there are four in total. You can distribute as you see fit.*
SAM: *I think the members of Vox Machina should get them.*
LAURA: *That doesn't seem really--*
MARY: *Is that what you think?*
SAM: *Well, just because I-- I defer to--*
FELICIA: *As the leader, I enjoy being hospitable, and I agree with this small person.*
MARY: *Well Scanlan, if you feel you need it.*
TALIESIN: *I already have a couple, so I will gladly give one to somebody who is going to need it.*
[...]

II. Preprocessing - `cr_proc.py`

Nettoyage

Les transcriptions étant originellement pensées pour du sous-titrage, il nous faut retirer les retours à la ligne superflus que contient le dataset.

Fonction correspondante dans le code: `CRprocessing.CRcleaning`

Division et rassemblement

On voit que si l'on ne fait rien au dataset, non seulement nous n'avons que 180 données, mais chaque épisode est également bien trop long pour être raisonnablement fourni directement au modèle (cf. `batchify` et `max_len`). Enfin et surtout, on aimerait séparer notre jeu de données en trois jeux (entraînement, évaluation et test) sans risquer de couper au milieu d'un mot ou d'une phrase.

On décide donc de rassembler ('bundle') les phrases par groupe pour obtenir un jeu de données bien plus fourni, avec chaque donnée d'une taille traitable par la suite.

Fonction correspondante dans le code: `CRprocessing.CRbundling`

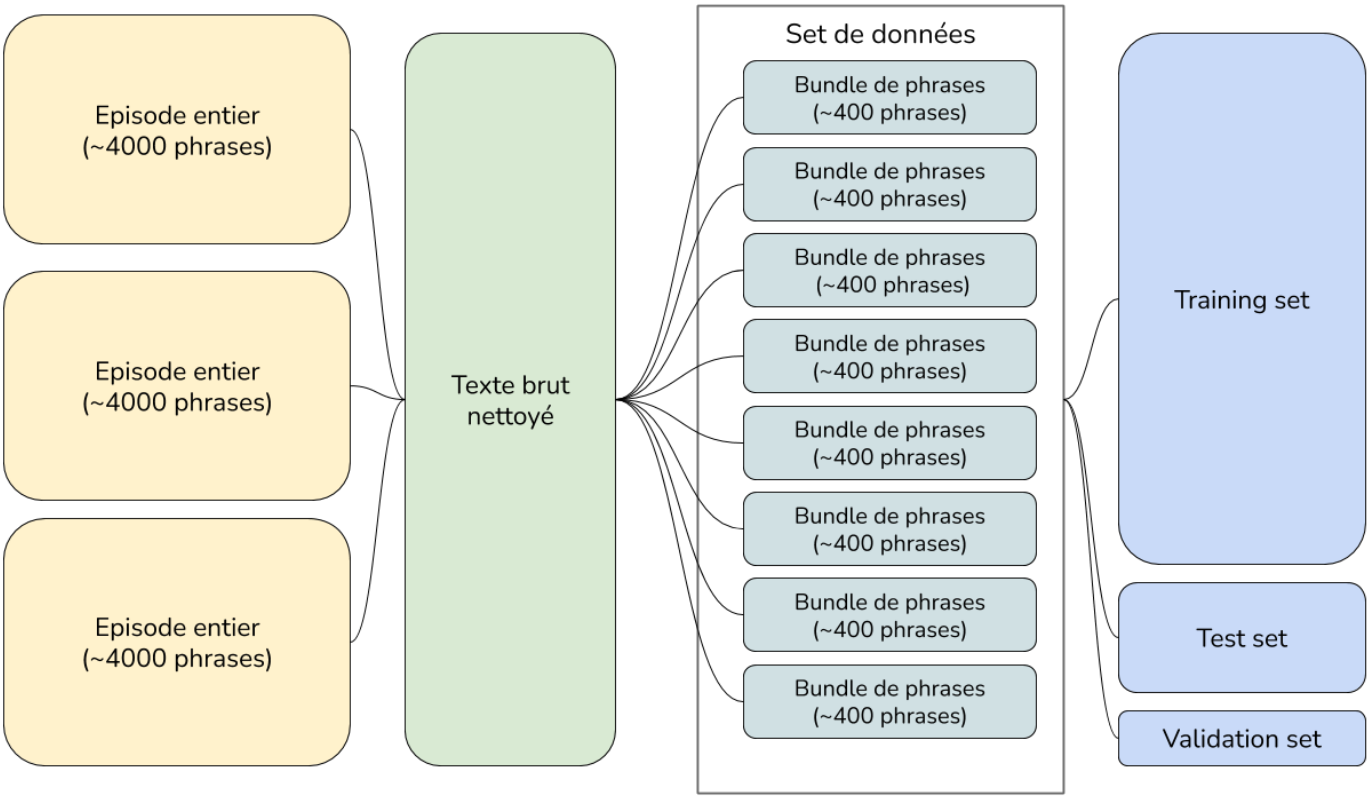
Données d'entraînement, de validation et de test

Nous avons besoin d'un jeu d'entraînement sur lequel faire apprendre notre modèle, un (relativement 'petit') jeu de validation sur lequel mesurer la performance de notre modèle durant l'entraînement, et un jeu de test (plus grand que le jeu de validation) sur lequel évaluer la meilleure itération de notre modèle après l'apprentissage.

Pour éviter, par exemple, que le modèle ne soit entraîné que sur des épisodes de la saison 1 (le but n'étant pas de prédire le token suivant différemment selon la saison de la séquence en entrée), on mélange le set de données avant de le séparer selon les proportions voulues. Note: l'étape de bundling rend le mélange encore plus efficace.

Fonctions correspondantes dans le code: `CRprocessing.shuffle_split` *et* `CRprocessing.CRtrain_test_eval`

Note: dans le code, 70% des données constituent le training set, et 80% des données restantes donnent le test set.



Tokenizing et Encoding

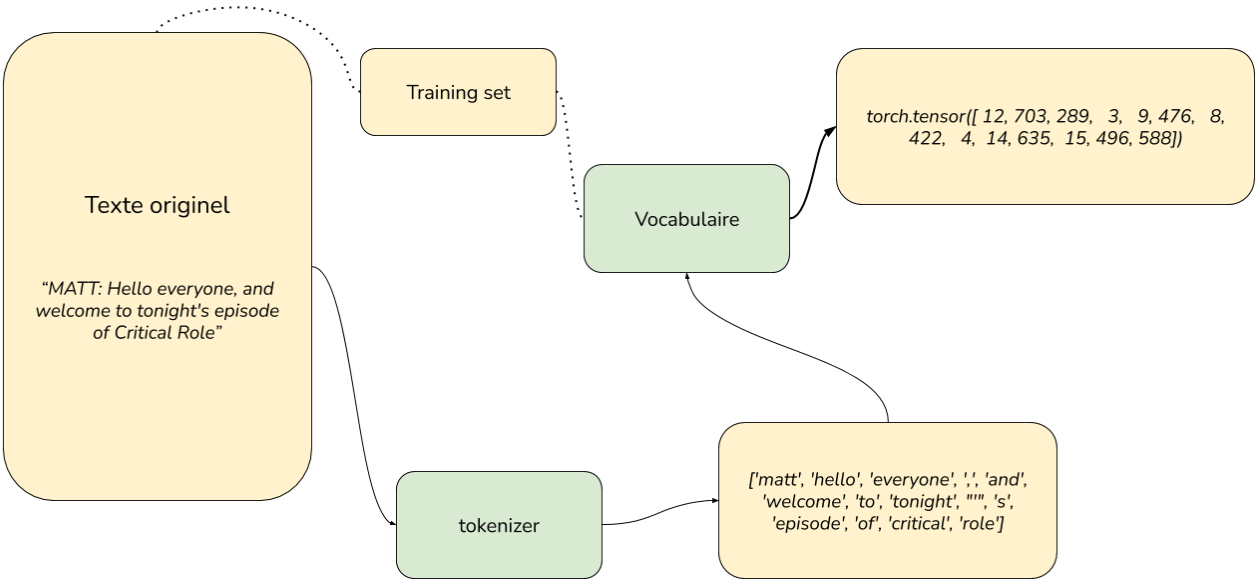
On utilisera pour la tokenization le tokenizer basique de pytorch, qui normalise la séquence avant de la séparer par mot entier (voir exemple).

Pour pouvoir encoder nos tokens, on construit un vocabulaire à partir du jeu d'entraînement. Dans ce vocabulaire, chaque token est associé à un identifiant numérique (entier), et l'objet vocab peut ensuite être utilisé dans les deux sens. Il est à noter que cet encoding ne porte aucune sémantique.

Fonction correspondante dans le code: `CRprocessing.CRcreate_vocab`

Il ne reste plus qu'à encoder les versions tokenizées de chacune de nos entrées (i.e. nos bundles). On en profite pour convertir nos données en tensors pytorch, car c'est ce que l'on va utiliser par la suite.

Fonctions correspondantes dans le code: `CRprocessing.CRencoding_to_torch` et `CRprocessing.CRtext_to_torch`



Découpage en batchs

Pour profiter de la haute parallélisabilité du Transformer, on divise les données équitablement en batchs que l'on traitera tous en même temps (dans un seul tensor). Notons que l'on se débarasse des données de fin lorsqu'elles feraient un batch incomplet.

Il est à noter que la dimension des batches est la première (la dimension 0), car c'est ainsi qu'est codé le Transformer de pytorch. (cf. https://pytorch.org/docs/1.7.1/generated/torch.nn.Transformer.html#torch.nn.Transformer.forward).

Fonction correspondante dans le code: `batchify` *de* `training.py`

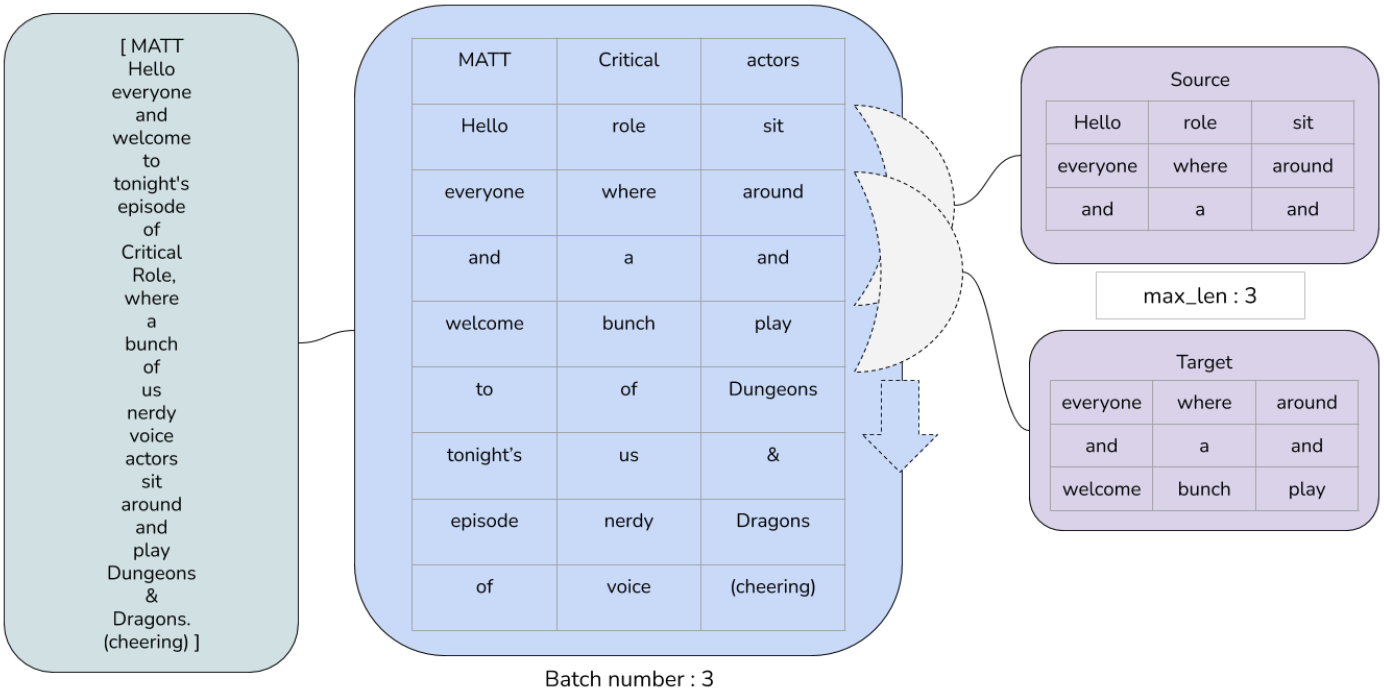
On choisit le nombre de batches empiriquement et à l'équilibre entre la rapidité d'entraînement et la qualité de convergence. Dans le code, nous avons choisi une largeur des tenseurs (nb_batches) de 30 pour le training set (et 10 pour les sets de validation et de test).

Séquences source et cible

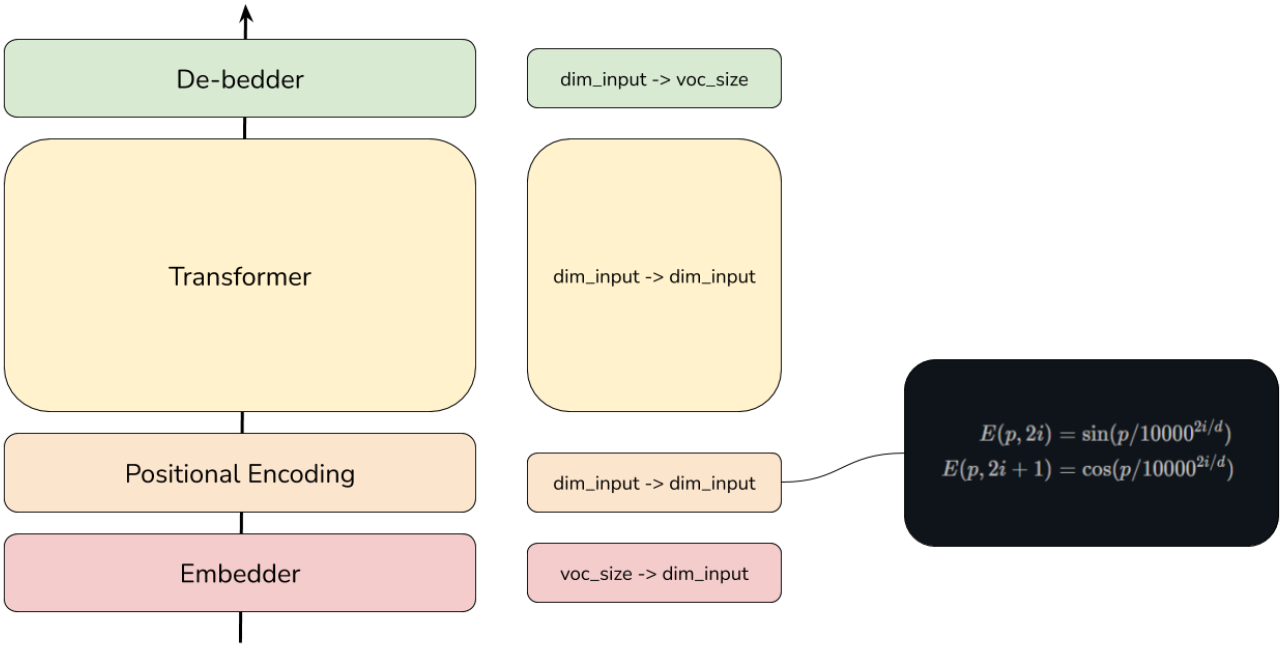
Il ne nous reste plus qu'à coder une fonction qui fournit au modèle la partie des données qu'il doit process à chaque pas de longueur `max_len` (choisie empiriquement à 45 dans le code). Pour pouvoir fournir à la fonction *loss* la cible à prédire, notre fonction renvoie une deuxième fois la séquence visée, mais décaléés d'un token vers la droite. (note: la partie self-attention, ou 'mémoire' du stack des

decoders (qui regarde la sortie du stack d'encoders) est gérée automatiquement dans le transformer)

Fonction correspondante dans le code: `get_batch` *de* `training.py`



III. Modèle - `cr_gen.py`



Information de position

Pour le moment, nos encodings n'ont aucune notion de leur position dans la phrase dont ils viennent. Afin de remédier à ce problème, et avant de pouvoir définir notre modèle, on programme donc un encoding de la position d'un mot dans sa séquence, que l'on ajoutera à son encoding de base.

On utilise l'approche relativement répandue proposée par *Vasmari et al*, avec deux fonctions cosinus et sinus. Avec p la position dans la séquence, et i la position selon la dimension de l'encoding : [cf équations dans l'image ci-dessus].

Notons que l'on calcule le tensor des positional encodings une seule fois à l'initialisation, et que l'on en prend que la partie nécessaire dans la passe avant (cf. `PositionalEncoding.forward`), où l'on effectue une simple somme pour intégrer cette information de position dans l'entrée du transformer.

Classe correspondante dans le code: `PositionalEncoding`

Programmation du modèle

On peut maintenant définir notre modèle. On utilise un transformer complet (encoder *et* decoder layers), car ses performances dans des tâches similaires à la nôtre ont été prouvées à de nombreuses reprises. Son dernier niveau est un niveau linéaire (une convolution de taille de kernel égale à 1) qui permet de retourner dans la dimension du vocabulaire.

Son premier niveau est un embedding, qui transforme à chaque identifiant du vocabulaire associe un vecteur unique de dimension `dim_input`.

On renvoie ici au code du modèle, exhaustivement commenté.

Classe correspondante dans le code: `CRTransformer`

IV. Apprentissage - `training.py`

Hyperparamètres

Nous initialisons le modèle avec les hyperparamètres ci-dessous, choisis arbitrairement pour permettre un compromis entre la rapidité d'entraînement et la qualité de convergence.

Nom du paramètre	Valeur
`input_dim`	`256`
`nb_heads`	`2`
`dim_ff`	`512`
`n_enc_layers`	`5`
`n_dec_layers`	`5`
`dropout`	`0.1`

Boucle d'apprentissage

On utilisera:

- le critère `CrossEntropyLoss` pour calculer la loss, puisque ce critère intègre tout seul la fonction LogSoftmax dont on aurait besoin pour passer de la sortie du modèle aux encodings effectivement prédits.
- la descente de gradient stochastique comme algorithme d'optimisation, avec une learning rate de 5 pour commencer, et que l'on ajuste avec le scheduler `StepLR` à chaque passe sur tout le training set. `StepLR` multiplie la learning rate par un facteur inférieur à 1 (ici, 0.3) à chaque epoch.

- `clip_grad_norm_` pour empêcher l'explosion des gradients, en limitant leurs normes à 0.5.

On imprimera la progression du modèle périodiquement.

Fonction correspondante dans le code: `train` de `training.py`

Evaluation

Entre chaque epoch, on utilise le set d'évaluation pour étudier la performance de notre modèle. On garde le modèle si cette dernière est la meilleure que l'on aie rencontré jusqu'ici.

Une fois l'entraînement terminé, on utilise le test set pour obtenir la performance finale du meilleur modèle.

Fonction correspondante dans le code: `evaluate` de `training.py`

On voit (cf training logs) que le modèle s'améliore rapidement, et que l'on obtient des scores d'entropie croisée comparables à ceux présentés en début de rapport.

	CRTranscripts	Wikitext-2	Wikitext-103
Score d'entropie croisée atteint	6,01	5,29	4,036

Faible influence des paramètres suivants:
- Nombre d'epochs
- Dimension d'entrée
- Nombre de têtes (très couteux par rapport au temps de calcul)
- Longueur traitée par une passe (max_len)

V. Commentaires finaux

Notre modèle obtient déjà des résultats satisfaisants (bien qu'il nous reste à l'utiliser dans la pratique pour générer du texte concrètement), mais notons quelques points et pistes sur lesquels nous n'avons pas joué et qui pourraient le rendre plus optimal:

- Modifier les hyperparamètres (notamment `input_dim`, `dim_ff`, et surtout `n_layers`) à la hausse : Entraîner une version BIG d'un modèle est pratique courante, et peut mener à des meilleures performances moyennant un temps d'entraînement (bien) plus long.

- Modifier les hyperparamètres à la baisse : Il existe également des versions 'légères' des modèles les plus classiques (on pense à distilbert par exemple), qui donnent des résultats tout à fait corrects. Il serait intéressant de voir si l'on peut en faire autant.

- Augmenter les données: Nous n'avons ici utilisé aucune technique de Data Augmentation, alors que notre dataset est relativement petit.

- Utiliser un meilleur tokenizer (le nôtre sépare "tonight's episode" en "tonight", "'", "s", "episode" par exemple), et un meilleur encoding (le nôtre ne porte aucune information sémantique). On pourrait, comme cela se fait souvent, utiliser les encoders/embedders de modèles déjà entraînés.

- Effectuer du transfer learning. Pratique courante, qui consiste en l'utilisation d'un modèle déjà entraîné, que l'on fine-tune sur notre dataset.