Deep Learning for NLP

Assignment 2 : Neural Language Modeling

CAUDARD Joris

Pour le 13/12/2024

Résumé

Ce rapport compare les performances de différents modèles de génération de texte, entraîné sur un dataset issu de Google Research. On comparera dans ce rapport la structure des modèles ainsi que les structures des datasets utilisés. On s'intéressera enfin plus particulièrement aux méthodes d'entraînement des modèles ainsi qu'à la perplexité obtenue sur un jeu de données test.

Dépôt Github lié

Dépôt GitHub du projet : https ://github.com/JorisCaudard/M2_DL4NLP

Table des matières

1 Introduction

Dans cette étude, on cherche à construire un modèle de Deep Learning permettant la génération de texte. Ce problème de Text Generation est courant en NLP, et on portera une attention particulière ici sur l'architecture des modèles utilisés (décrits en partie 2).

On se basera pour cela sur un jeu de données public issus des équipes de Google Research. Initialement développé dans un but de problème de Sentiment Analysis, on en extraira uniquement les parties "text" correspondant uniquement aux textes originaux. En se basant sur cette liste de phrases, on implémentera deux des principaux modèles de générations de textes (N-gram model et modèle LSTM).

On étudiera également leurs performances en se basant sur différentes métriques : Perte sur les jeux d'entraînement et de développement, Perplexité sur un jeu de données test, temps d'entraînement. On observera également quelques exemples de génération de phrase par chacun des modèles, de façon déterministe d'abord puis de façon aléatoire.

2 Modèles utilisés

2.1 N-gram Model

Le premier modèle qu'on étudiera sera basé sur le principe de n-grams. Pour cela, il est nécessaire de préalablement traité le dataset d'entraînement, afin de le séparer en n-grams.

On commencera tout d'abord (et ce également pour le modèle LSTM) par compléter les phrases d'entraînement par des tokens dénotant les débuts et fins de phrases, en utilisant des tokens non présent dans le vocabulaire initial du jeu d'entraînement. On utilisera dans notre cas des tokens *<bos>* et *<eos>*. De plus, dans le cas du modèle N-gram, et afin de prévoir les premiers mots des phrases, il est nécessaire de "padder" le début de chaque phrase du dataset de n tokens *<bos>*. De ce fait, on aura dans le dataset d'entraînement des couples **(context, target)** de la forme **(<bos>, ..., <bos>, premier_mot)**, afin de permettre au modèle d'apprendre la distribution des mots débutants chaque phrase du dataset.

Le dataset d'entraînement contiendra donc des éléments de la forme **(context, target)**, ou context sera une liste de mot de longueur n, et target sera le prochain mot de la phrase. Chaque mot sera bien évidemment encodé par sa position dans le vocabulaire, afin de permettre l'entraînement du modèle.

Le modèle présentera alors une structure classique de modèle neuronaux n-gram, en étant constitué d'une couche d'*embeddings*, puis de couches linéaires *fully connected*, en produisant en sortie un vecteur de longueur "nombre de mots du dictionnaire". En effet, on peut voir le problème de génération de phrase comme un problème de classification, ou on prédit le mot le plus probable parmi l'ensemble des mots connus, comprenant

également le token *<eos>* si la phrase se termine après le n-gram vu en entrée.

De ce fait, le modèle produira en sortie en vecteur de longueur "nombre de mots du dictionnaire" consistant en l'ensemble des logits en sortie du modèle. On tirera alors parti des optimisations de pytorch lors du calcul de la fonction de perte, qui sera dans notre cas la fonction de perte *CrossEntropyLoss*, complètement équivalente à la fonction de log-vraisemblance négative NLL, mais qui présente l'avantage d'être optimisé en interne par la bibliothèque pytorch.

Bien que simple, le modèle présente l'inconvénient d'être entraîné sur un plus grand nombre d'items. En effet, une phrase de longueur l présentera $l - n$ n-grams, ce qui correspond donc à $k * (l - n)$ n-grams d'entraînement sur un corpus de k phrases. En raison de ce décuplement exponentiel des données d'entraînement, et en raison de limites computationnelles, on n'entraînera le modèle que sur un corpus restreint de phrases.

2.2 LSTM model

Pour le modèle LSTM, l'approche est légèrement différente. En effet, plutôt que se focaliser sur un ensemble de n-gram, le modèle se focalise sur l'ensemble de la phrase pour prédire le mot suivant. Plutôt que d'utiliser uniquement le dernier état caché comme prédiction du modèle, on utilisera l'ensemble de la sortie de la couche LSTM du modèle, afin de prévoir une phrase de la même longueur que la phrase d'entrée. On utilisera pour cela un paddding dynamique des phrases dans chaque batch du dataset d'entraînement, afin de garantir la même longueur des phrases au sein d'un même batch.

Un item d'entraînement consistera donc en un couple de phrases de même longueur $(input\_sentence, target\_sentence)$, où la phrase cible correspond à la phrase d'entrée mais décalé d'un rang. On paddera la phrase cible avec un token *<pad>* de plus à droite, afin de garantir la même longueur des phrases d'entrée et cible. De ce fait, le modèle apprendra sur l'ensemble de la phrase, ce qui permet d'avoir un dataset d'entraînement de la même longueur que le corpus initial.

On accompagnera le modèle d'une couche d'*embeddings*, ainsi que d'une couche finale linéaire *fully connected* afin de projeter la sortie de la couche lstm sur l'ensemble du vocabulaire. La sortie finale du modèle aura donc pour forme *(batch_size, sequence_length, vocab_siz)*, où la longueur sequence_length peut varier d'un batch à l'autre, celui-ci dépendant de la phrase de longueur maximale dans le batch.

De la même manière que le modèle N-gram, on utilisera une fonction de perte de type *CrossEntropyLoss*, cette fois ci sur chaque mot des phrases prédites en sortie. De cette manière, le dataset d'entraînement a la même longueur que le corpus initial, et cela permet également au modèle de s' entraîner uniformément sur toute la phrase.

Contrairement au modèle N-gram, il n'est pas nécessaire ici de padder la phrase d'entrée deplusieurs tokens *<bos>*. On effectuera le même padding avec uniquement un token *<bos>* en début de phrase ainsi qu'un token *<eos>* en fin de phrase pour permettre au

modèle d'apprendre la distribution des mots en commençant ou terminant les phrases. On ajoutera également au vocabulaire un token *<pad>* afin de garantir une même longueur des phrases au sein d'un batch comme évoqué précédemment.

On ajoutera également au modèle LSTM une couche de *Variational Dropout* lors de l'entraînement uniquement, cela octroyant au modèle une couche de régularisation similaire au dropout implémenté dans le modèle n-gram, tout en en étant plus adapté au cas spécifique des LSTM. Cette startégie permet de réduire le risque de sut-apprentissage, en apportant une stabilité lors de l'activation des couches récurrentes du modèle.

3 Méthodologie

3.1 Prétraitement & Entraînement

Afin de préparer le dataset, une légère étape de prétraitement sera appliqué au corpus d'entraînement initial. En effet, plutôt que simplement tokenizé chaque mot tel qu'il se présente dans le corpus, on appliquera une légère étape de nettoyage sur l'ensemble des phrases du dataset.

On appliquera alors les étapes suivantes à l'ensemble des phrases du corpus. Une première étape de traitement sera la mise en minuscules complète de chaque mot du dataset, afin d'éviter de mettre dans le dictionnaire le même mot avec et sans majuscule en tant que deux mots différents. On appliquera également un simple split des phrases en mots, ainsi qu'un split de certains mots selon des règles spécifiques à la langue anglaise (négation en n't, contraction you're séparé en you + re ...). Cette simple tokenisation, bien que moins efficace qu'un tokenisation plus complexe de type Word2Vec ou équivalent permettra néanmoins d'obtenir des premiers résultats.

Comme discuté précédemment, on utilisera une fonction de perte de type *CrossEntropyLoss*, en prenant en entrée non pas les probabilités de chaque mot, mais directement les logits obtenus en sortie du modèle. De cette manière, on tirera pleinement parti des optimisations en amont du module pytorch lors du calcul des différents gradient.

On entraînera chacun des deux modèles sur le même corpus initial - bien que les formes des datasets d'entraînement varient comme discuté en partie 2 selon les spécificités du modèle. De cette manière, on comparera les performances des modèles sur le même jeu de données, en comparant leur fonction de perte sur les jeux d'entraînement et de validation, ainsi que leurs perplexités calculées sur le même jeu de données test.

3.2 Hyper-paramètres

Les modèles étant relativement complexes, nombre d'hyper-paramètres doivent être pris en compte lors de l'évaluation des modèles. En raison de soucis computationnels lors de l'entraînement des modèles, on restreindra l'optimisation des hyper-paramètres à une recherche expérimentale manuelle de différentes structures de réseaux.

On utilisera les mêmes paramètres pour les couches d'embeddings de chaque modèle, ainsi que les mêmes paramètres d'entraînement afin de pouvoir comparer leurs performances sur des données équivalentes.

On utilisera néanmoins différentes taille de contexte lors de l'élaboration des modèles n-grams, afin de comparer leurs performances entre eux.

4 Résultats et analyse

On récapitule les résultats obtenus par chacun des modèles dans le tableau suivant :

Modèle	CrossEntropyLoss (Train)	CrossEntropyLoss (val)	Perplexity (test)
Unigram Model	4.87	7.51	92.22
Bigram Model	**4.79**	8.16	**76.77**
Trigram Model	4.89	8.44	82.56
4-gram Model	4.99	9.12	90.42
5-gram Model	5.07	9.40	96.34
LSTM Model	5.23	**6.10**	NaN

Table 1 – Résultats obtenus par les différents modèles

On peut alors déduire de ces résultats plusieurs conclusions. Premièrement, on peut remarquer que les différents modèle N-grams présentent des caractéristiques relativement proches. On peut également remarquer que ces différents modèles ont tendance à perdre en performances si on augmente trop la longueur de contexte. En effet, il est alors plus complexe pour le modèle d'extraire des structures significatives dans ces n-grams. Dans notre cas, la longueur de contexte optimale semble être 2, indiquant qu'un modèle entraîné sur les bigrams du corpus initial est plus efficace que les autres modèles du même type.

Néanmoins, le meilleur compromis reste le modèle LSTM. En effet, ce modèle est celui présentant les meileurs résultats sur le jeu de test, indiquant que sa capacité de généralisation est plus grande que les modèles précédents. Il présente également des performances similaires sur le jeu de données d'entraînement. De plus, ce modèle s'entraînant directement sur les phrases du dataset, il est également plus rapide d'entraîner ce modèle. On peut alors augmenter le nombres d'epochs d'entraînement, augmentant encore plus les performances du modèle.

On peut également s'intéresser aux phrases générées par chacun des modèles. Pour cela, on distingue deux manières de générer des phrases : l'une déterministe, l'autre aléatoire. Pour les modèles n-grams, on peut initialiser une phrase en passant en entrée du modèle le contexte [**<bos>, ..., <bos>**] (bien évidemment tokenizé). Ensuite, il suffit de passer ce contexte via le modèle pour générer le premier mot de la phrase, avant de recommencer avec un nouveau contexte incluant le mot généré. Pour choisir le prochain mot, on peut alors soit prendre le mot maximisant la vraisemblance (déterministe), soit tirer aléatoirement en suivant la distribution de ces prbabilités (aléatoire). De même pour

le modèle LSTm, en utilisant cette fois ci comme contexte initial [**<bos>**]. on peut alors générer des phrases, jusqu'à obtenir un token <eos>, ou jusqu'à atteindre une longueur maximale fixée afin d'éviter de tomber dans un cycle de mots se répétant.

En s'intéressant à ces phrases générées, on remarque que les différents modèles arrivent remarquabement bien à capter des structures simples : Si le dernier token est un point d'interrogation ou autre signe de ponctuation, le token suivant est probablement <eos>, Un mot commençant par "ca" ou "do" est probablement suivi d'un token "n't", etc …Néanmoins, les modèles semblent peiner à généraliser des structures plus diffusent. Les modèles N-grams en particulier semblent fortement prédire les mêmes structures, en utilisant souvent les tokens "I", "you" …

5 Conclusion

Les résultats montrent alors bien que ces modèles sont capables de capter des structures sous-jacentes au sein d'un corpus de texte, même si celui-ci consiste en des phrases écrites par des natifs, avec ces abréviations, approximations …De plus, cette étude met en avant la supériorité des modèles LSTMs façe aux modèles plus traditionnels. On met en avant aussi que la simplicité des modèles N-grams en fait néanmoins un point de départ intéressant de tout modèle de languages, si le problème sous-jacent reste simple.

Pour aller plus loin, on pourrait s'intéresser à l'optimisation des autres hyper-paramètres des modèles, comme la taille des réseaux *fully connected* de projection, ou les paramètres de *dropout*. On pourrait également s'intéresser à d'autres méthodes d'embeddings que celle simple utilisée dans le cadre de ce rapport, en se rapprochant d'une tokenisation Word2Vec par exemple. De plus, un meilleur prétraitement des mots du corpus pourraient permettre de meilleurs performances pour les modèles entraînés ici.

A Annexes

A.1 Tables et figures

Liste des tableaux

Table des figures

A.2 Hyper-paramètres

A.2.1 Modèle N-Gram

```
NeuralNGramModel(
  (embeddings): Embedding(8159, 128)
  (fc1): Linear(in_features=640, out_features=64, bias=True)
  (fc2): Linear(in_features=64, out_features=8159, bias=True)
  (dropout): Dropout(p=0.3, inplace=False)
)
```

Listing 1 – Structure du modèle N-gram

- `Vocab size` : 8159
- `Embedding size` : 128
- `Hidden size` : 64

A.2.2 Modèle LSTM

```
LSTMModel(
  (embeddings): Embedding(8159, 128)
  (lstm): LSTM(128, 64, batch_first=True)
  (fc): Linear(in_features=64, out_features=8159, bias=True)
)
```

Listing 2 – Structure du modèle LSTM

- `Vocab size` : 8159
- `Embedding size` : 128
- `Hidden size` : 64

A.2.3 Hyper-paramètres d'Entraînement

- `Batch size` : 32

— `Training Dataset Size` : 5000
— `Nombre maximal d'epochs` : 10
— `Optimizer` : Adam