TP3 - Modèle de langage neuronal

IAAA - TLNL

1 Objectif

L'objectif de ce projet est de programmer un modèle de langage neuronal à l'aide d'un perceptron multi-couches. Ce modèle de langage prend en entrée les plongements de k mots consécutifs et produit en sortie une distribution de probabilité sur l'ensemble du vocabulaire. Contrairement au projet précédent, nous ne ferons pas le calcul du gradient à la main, mais utiliserons la librairie `pytorch` et les outils de dérivation automatique qu'elle propose.

2 Le classifieur

Le cœur du modèle est constitué d'un perceptron multicouche C composé d'une couche d'entrée $\mathbf{x}$, d'une couche cachée $\mathbf{h}$ et d'une couche de sortie $\mathbf{y}$. Les paramètres du modèle sont regroupés dans deux matrices : $\mathbf{W}$ et $\mathbf{U}$ et deux vecteurs : $\mathbf{b}_1$ et $\mathbf{b}_2$. $\mathbf{W}$ et $\mathbf{b}_1$ permettent de calculer les valeurs de la couche cachée à partir de la couche d'entrée tandis que $\mathbf{U}$ et $\mathbf{b}_2$ permettent de calculer les valeurs de la couche de sortie à partir de la couche cachée.

Étant donné une séquence de mots $m_1 \dots m_n$, le classifieur C prend en entrée les plongements des k mots $\mathbf{m}_{i-k} \dots \mathbf{m}_{i-1}$ et produit en sortie une distribution de probabilité sur l'ensemble des mots du vocabulaire V.

On considère que les plongements de chaque mot sont de dimension d. La taille de la couche d'entrée, notée $\mathbf{x}$ vaut par conséquent $d_x = k \times d$ et la couche de sortie a pour dimension $|V|$, la taille du vocabulaire ! La dimension de la couche cachée est arbitraire, on la notera d_h. Les dimensions des deux matrices $\mathbf{W}$ et $\mathbf{U}$ sont donc $d_h \times d_h$ pour $\mathbf{W}$ et $d_h \times |V|$ pour $\mathbf{U}$.

Le calcul de la couche caché peut être décomposée en deux étapes :

— une étape linéaire : $\mathbf{h}' = \mathbf{x}\mathbf{W} + \mathbf{b}_1$

— suivie d'une étape non linéaire : $\mathbf{h} = \mathit{ReLU}(\mathbf{h}')$

De même, le calcul de la couche de sortie peut aussi être décomposée en deux étapes :

— une étape linéaire : $\mathbf{y}' = \mathbf{h}\mathbf{U} + \mathbf{b}_2$

— suivie d'une étape non linéaire : $\mathbf{y} = \mathit{Softmax}(\mathbf{y}')$

La fonction $\mathbf{y} = \mathit{softmax}(\mathbf{y}')$ permet de transformer un vecteur de valeurs réelles $\mathbf{y}'$ en un vecteur de probabilités $\mathbf{y}$. Chaque composante $\mathbf{y}_i$ de $\mathbf{y}$ est calculée de la manière suivante :

$$\mathbf{y}_i = \frac{\exp(\mathbf{y'}_i)}{\sum_{j=1}^{|V|} \exp(\mathbf{y'}_j)}$$

Autrement dit, la fonction $\exp(\cdot)$ rend la valeur de $\mathbf{y}'_i$ positive, et ensuite chaque valeur est normalisée par la somme de toutes les valeurs du vecteur $\mathbf{y}'_j$. Cela garantit que $\mathbf{y}_i$ est bien une probabilité car $\sum_{i=1}^{|V|} \mathbf{y}_i = 1$.

2.1 Fonction de perte

La fonction de perte utilisée pour mettre à jour les matrices de paramètres $\mathbf{W}$ et $\mathbf{U}$ est l'entropie croisée. Étant donné deux distributions de probabilité discrètes r et y, définies sur le même ensemble fondamental $\Omega = \{1, \ldots, N\}$, l'entropie croisée $H(r, y)$ se calcule de la manière suivante :

$$H(r, y) = -\sum_{i=1}^{N} P_r(i) \log_2(P_y(i))$$

Si on considère que la distribution r est la distribution de référence (la distribution que l'on aimerait que notre modèle calcule) et que y est la distribution calculée par le modèle, $H(r, y)$ mesure d'une certaine façon l'erreur que l'on commet en utilisant la distribution y à la place de la distribution r.

Dans notre cas r est une distribution bien particulière, il s'agit de la représentation *one-hot* d'un mot du vocabulaire. Etant donné un vocabulaire V, où chaque mot correspond à un indice compris entre 1 et $|V|$, la représentation one-hot du mot d'indice i est un vecteur de dimension $|V|$ dont toutes les composantes sont à 0 sauf la composante i qui vaut 1. Un tel vecteur binaire peut être vu comme la représentation (pas très économique) du i-ème mot du vocabulaire. Il peut aussi être vu comme une distribution de probabilité où toute la masse de probabilité est concentrée sur un seul mot : celui d'indice i.

Lorsque r est une représentation one-hot, tous les termes s'annulent sauf un. Dans ce cas, l'entropie croisée se calcule de manière très économique, elle vaut :

$$H(r, c) = -\log_2(\mathbf{y}_i)$$

où i est l'indice du mot attendu (le mot correspondant à la valeur 1 dans la représentation one-hot r) et est $\mathbf{y}_i$ la i-ème composante de la couche de sortie du classifieur (la probabilité qu'associe le modèle au mot que l'on souhaite prédire via la fonction softmax).

2.2 Mots hors vocabulaire

Les données d'apprentissage sont constituées de texte brut T : une séquence de mots divisée en phrases, et d'une matrice de plongements E. Avant d'être fournis en entrée au classifieur, les mots doivent être transformés en plongements, par un simple accès à la matrice E.

Le modèle de langage doit prévoir le cas des mots inconnus, c'est-à-dire, les mots vus lors d'utilisation du modèle qui ne se trouvaient pas dans son corpus d'entraînement (appelés aussi OOV, pour *out of vocabulary*). Par exemple, il est possible que, lors du calcul de la perplexité d'un texte T', certains mots de T' n'apparaissent pas dans le corpus d'apprentissage T.

Pour résoudre le problème, on remplacera, dans T, les mots dont la fréquence d'occurrence est inférieure à un seuil γ, par un mot particulier, par exemple `<unk>`. Ainsi lors de l'apprentissage, le classifieur aura appris à traiter des occurrences de `<unk>` comme si c'était un mot du vocabulaire.

Lors de la prédiction, si un mot m n'appartient pas au vocabulaire de T, il sera remplacé par `<unk>` (vous n'avez pas besoin de le remplacer en dur : il suffit, lors de la préparation des données, d'utiliser pour m le plongement de `<unk>`).

Cela suppose que `<unk>` ait un plongement vu par le classifieur lors de l'entraînement. Pour cela, nous utiliserons la même procédure que pour l'apprentissage du modèle de langage : lors de l'apprentissage des plongements, on remplacera les mots dont la fréquence d'occurrence est inférieure à γ, par le mot `<unk>`, cela permettra donc de calculer un plongement pour le mot `<unk>`.

2.3 Données

Pour réaliser vos expériences, nous vous fournissons les données suivantes :

- `Le_comte_de_Monte_Cristo.train.unk5.tok` un corpus d'apprentissage dans lequel les mots dont le nombre d'occurrence est strictement inférieur à $\gamma = 5$ ont été remplacés par `<unk>`.
- `Le_comte_de_Monte_Cristo.train.100.unk5.tok` un petit corpus d'apprentissage à utiliser lors de la phase de mise au point de votre programme.
- `embeddings-word2vecofficial.train.unk5.txt` un fichier de plongements appris sur le corpus `Le_comte_de_Monte_Cristo.train.unk5.tok` (dimension $d = 100$).

Afin de ne pas passer trop de temps à écrire le code pour la lecture du fichier de plongements, nous fournissons aussi une classe `Vocab.py`. Cette classe prend le nom du fichier de plongements en paramètre de son constructeur et charge tous les plongements dans une matrice `pytorch`. Des fonctions permettent de convertir facilement un mot au format textuel vers son indice, son plongement, ou sa représentation one-hot.

3 Ce qu'il faut faire

1. Ecrire la fonction de préparation des données qui prend en entrée un fichier de texte T ainsi qu'un dictionnaire associant à tout mot du vocabulaire, un indice [1]. Cette fonction produit en sortie une liste dont les

1. Vous pourrez utiliser la classe `Vocab` qui permet de lire un fichier de plongements et d'accéder à l'indice d'un mot ainsi qu'à son plongement.

éléments sont des listes de $k+1$ mots consécutifs de T représentés par leur indices.

Par exemple, si le corpus se présente de la façon suivante `a b c d e f` et que le mot `a` correspond à l'indice 1, le mot `b` correspond à l'indice `2` ...et que k vaut 3, les listes `X` et `Y` se présenteront sous la forme suivante :

```
X = [[1,2,3,4],[2,3,4,5],[3,4,5,6]]
```

2. Ecrire la boucle d'apprentissage qui parcours un certain nombre de fois la liste des exemples et, pour chaque exemple, construit le graphe de calcul de la fonction de perte, calcule son gradient et effectue un pas d'optimisation. Il est intéressant, pour chaque itération (epoch) d'afficher la valeur moyenne de la fonction de perte sur les données d'entraînement. Cette valeur doit décroître si l'apprentissage se passe bien. N'oubliez pas de mélanger les exemples aléatoirement entre chaque itération.
3. Sauvegarder les paramètres dont les valeurs ont été optimisées lors de l'apprentissage. Vous utiliserez pour cela la fonction `torch.save`.
4. Ecrire le programme `perplexite.py` qui prend en entrée un modèle M (le fichier sauvegardé à l'issue de l'apprentissage) et un texte T et calcule la perplexité de T étant donné les probabilités données par M.
5. Ecrire le programme `genere.py` qui prend en entrée un modèle M et génère du texte par tirage aléatoire dans les distributions de probabilités calculées par le classifieur.

4 Traitement par lots

Quand vous aurez implémenté le modèle tel qu'il est décrit ci-dessus, vous allez voir que l'apprentissage est extrêmement long sur le texte d'entraînement complet. Cela est dû au fait que chaque exemple est fourni individuellement, avec une mise à jour coûteuse des paramètres à chaque fois.

Une fois votre modèle testé sur un petit texte, vous allez le modifier pour traiter les entrées par lots (*batches*). Vous devez modifier la lecture des données pour décomposer la liste d'exemples en une liste de batches contenant un nombre fixe b d'exemples. Les fonctions `torch.stack` et `torch.split` peuvent vous aider.

Ensuite, le calcul *forward* de la sortie ne se fera plus sur un vecteur d'entrée $\mathbf{x}$, mais sur un batch (matrice) de dimension $b \times d_x$. La sortie du modèle sera également une matrice de dimension $b \times |V|$.

`pytorch` prend en charge la parallélisation des opérations sur un batch de manière complètement transparente. Par conséquent, les opérations de produit et somme sur les `tensor`, ainsi que la fonction de perte, prennent en compte des batches sans qu'aucune modification du code du classifieur ne soit nécessaire.

Notez que, lors de la somme des vecteurs de biais $\mathbf{b}_1$ et $\mathbf{b}_2$, ils sont automatiquement convertis vers des matrices de dimension $b \times d_h$ et $b \times |V|$. Cela rend leurs dimensions compatibles avec le calcul par batch (on parle de *broadcast*).

Ici, aussi, vous n'avez rien à changer dans le modèle, tout est automatique tant que les dimensions des sorties sont inférables.

Vous devrez trouver la valeur optimale de b (des valeurs typiques sont les puissances de 2 entre $[4, 64]$) : des batches trop petits rendent l'apprentissage trop long, des batches trop gros demandent beaucoup d'itérations pour converger. N'oubliez pas de mélanger les batches aléatoirement entre chaque itération. Il est inutile de mélanger les exemples à l'intérieur d'un batch, la mise à jour ne s'éffectuant qu'une fois tout le batch traité.

5 Quelques pistes à creuser

Le modèle de langage décrit ci-dessus utilise des plongements qui ont été appris à l'avance. Les pistes ci-dessous visent à explorer d'autres manières d'utiliser des plongements. Trois pistes sont proposées, parmi lesquelles vous en choisirez deux. La première est obligatoire (les deux autres en dépendent). Vous devez donc choisir (1,2) ou bien (1,3).

1. Il est possible de faire en sorte que le modèle de langage apprenne ses propres plongements. Pour cela, les plongements doivent devenir des paramètres du modèle, qui seront mis à jour lors de l'apprentissage. Pour cela, il faut enrichir les paramètres du modèle d'une matrice $\mathbf{E}$ apprenable de dimension $V \times d$ où V est la taille du lexique et d la dimension des plongements. La création du vecteur d'entrée consiste à concaténer les lignes de $\mathbf{E}$ correspondant aux indices des k mots du prefixe (on pourra utiliser la fonction `torch.concat`).
2. Nous avons maintenant deux manière de construire des plongements, soit à l'aide de l'algorithm SGNS vu dans le projet précédent, soit à l'aide du modèle de langage programmé dans ce projet. L'idée est de comparer ces plongements. Vous pourrez commencer par évaluer leurs performances respective sur la tâche de calcul de similarité du projet précédent. Vous analyserez ensuite la répartition des mots dans les deux espaces d'embeddings : est ce que les mots les plus proche d'un mot cible t dans le premier espace sont aussi les mots les plus proches de t dans le second ? Vous pourrez aussi utiliser la projection t-sne pour visualiser en deux dimensions les deux espaces.
3. Lorsque le modèle de langage apprend ses propres plongements (la matrice $\mathbf{E}$), on peut partir d'une matrice aléatoire, comme on fait pour les autres matrices de paramètres, mais on peut aussi partir de de plongements pré-entrainés, par exemple par l'algorithme SGNS. L'objectif de cette piste est de comparer ces deux façon de faire par rapport à la performance des modèles de langage (est ce que les perplexités des modèles de langage sont différentes), la qualité des plongements (est ce que les plongements appris avec initialisation obtiennent de meilleurs résultats sur la tâche de similarité) et à la vitesse d'apprentissage (est ce que l'apprentissage est plus rapide si on part de plongements pré-entrainés ?).