Casamian Marius, Projet de Traduction automatique, Pierre Pereira, 13/01/2025

Introduction:

L'objectif de ce TP est d'entraîner des modèles de traduction automatique de l'Anglais vers le Français.
Les grandes étapes seront l'élaboration d'un tokenizers avec sentencepiece, un RNN, un Transformer en Pytorch, et une méthode de sampling plus avancée (top-p sampling).
Nous expliquerons étape par étape, notre progression dans le TP tout en respectant les consignes, vous trouverez aussi des exemples de code (entier dans le dossier), des graphiques, des observations, des remarques ainsi que les ressources utilisées pour ce projet. Dans mon cas j'ai utilisé VsCode, mais pour cette présentation un fichier notebook intégrale sera disponible, pour plus de lisibilité et de facilité à exécuter.

Partie 1: Character-Level RNN

On construit les bases en utilisant un RNN simple et un tokenizer caractère par caractère.

Pipeline:
-Lire le CSV (en-fr),
-Faire un **char-level tokenizer**,
-Dataset préparé
-Entraîner un petit **RNN seq2seq(**sans pousser l'optimisation) caractère un à un
-Teacher forcing
-Calculer la **loss**, la **top-1 accuracy**, la **top-5 accuracy**,
-Tester la génération avec une fonction,

La classe ***SimpleTokenizer*** permet de transformer chaque phrase en une séquence de caractères, avec des tokens spéciaux <bos> (begin of sentence) et <eos>(end of sentence) qui marquent le début et la fin d'une phrase, ensuite la méthode **fit** construit le dictionnaire associant chaque caractère à un ID unique ex: {0: <bos>, 1: 'H', 2:'E', 3:'L', 4:'L', 5: 'O', 6:<eos> }.
encode et **decode** vont ensuite chacun transformer une phrase en une séquence d'ID et revenir d'une séquence d'IDs à une phrase.

```
Appareil utilisé : cuda
Nombre de paires chargées : 50000
exemple voc généré: {'<bos>': 0, '<eos>': 1}
Taille du vocabulaire : 192
Source tensor: tensor([0, 2, 3, 4, 5, 6, 7, 5, 6, 8, 9, 1])
Target tensor: tensor([ 0, 21, 26,  8,  4,  8, 17, 23,  4,  5, 12,  1])
Source (decoded): Changing L
Target (decoded): Il a trans
```

On peut voir que cette sortie est correcte.

*Premier points et limite*s: Par sa simplicité le tokenizer découpe au sens strict caractère par caractère, sans pré processing avancé (pas de gestion de la casse, des accents, etc.). **Limites** : Des séquences longues, ce qui peut poser problème en mémoire ou en temps de calcul.

La classe **SimpleTextPairDataset** a pour objectif de gérer les paires, (en, fr) en coupant chaque phrase en ID. Voici la trame:
-Charge des paires (source, cible) depuis un fichier CSV.
-Tronque les phrases à une longueur maximale (par défaut max_len=50) pour limiter l'explosion en mémoire.
-Retourne les phrases sous forme de tenseurs PyTorch.

On peut voir la source tensor [0,2,3,4,5,6,7,5,6,8,9,1], décodé donne: "Changing L", cela correspond à la phrase tronquée à la limite données de 10 + <bos> et <eos> les tokens, idem pour target tensor.

Deuxième points:
Troncature : Explication de l'importance de couper les séquences trop longues comme vous me l'avez dit par mail.
Ajout des tokens spéciaux : Justification de l'ajout de <bos> et <eos> pour aider le modèle à différencier le début et la fin des phrases.

Nous allons entraîner un modèle de traduction simple où le RNN prédit chaque caractère un par un dans la séquence cible, conditionné sur la séquence source.

Voici le model:

```
Seq2SeqRNN(
  (embedding): Embedding(192, 128)
  (rnn): RNN(128, 256, batch_first=True)
  (fc): Linear(in_features=256, out_features=192, bias=True)
)
```

D'abord l'embedding layer, convertit chaque ID de caractère en un vecteur de taille 128 et permet d'avoir une représentation dense et continue des caractères, facilitant leur traitement par le RNN, exemple si la phrase source est hello, elle est convertie en une séquence de vecteurs de taille (seq_len, 128).
Ensuite, le RNN layer apprend les relations séquentielles entre les vecteurs de caractères.

Encodeur : Lit la phrase source et produit un état caché final résumant son contenu.
Décodeur : Génère les caractères de la phrase cible, conditionnés sur l'état caché initial
Enfin, le Fully connected layer projette les états cachés du RNN dans un espace de probabilité correspondant aux tokens du vocabulaire, les scores sont ensuite transformés en probabilités, permettant de sampler ou d'identifier le caractère le plus probable suivant la méthode choisie.

Points forts: Ce modèle simple est bien adapté pour une première étape d'expérimentation. Cette première approche pourra être étendue et modifiée pour des modèles plus complexes (Transformer) par la suite. Les limites sont qu'ils à du mal à générer les dépendances long terme, par exemple entre des tokens éloignés dans une séquence, caractère par caractère n'est pas la meilleure approche pour des langues complexes

```
Appareil utilisé : cuda
Nombre de paires chargées : 50000
Taille du vocabulaire : 192
Source tensor: tensor([0, 2, 3, 4, 5, 6, 7, 5, 6, 8, 9, 1])
Target tensor: tensor([ 0, 21, 26,  8,  4,  8, 17, 23,  4,  5, 12,  1])
Source (decoded): Changing L
Target (decoded): Il a trans
Epoch 1, Loss: 1.8084, Top1: 0.5076, Top5: 0.7936
Epoch 2, Loss: 1.6915, Top1: 0.5415, Top5: 0.8146
Epoch 3, Loss: 1.7287, Top1: 0.5319, Top5: 0.8088
Epoch 4, Loss: 1.7956, Top1: 0.5149, Top5: 0.7988
Epoch 5, Loss: 1.8352, Top1: 0.5034, Top5: 0.7933
Epoch 6, Loss: 1.9056, Top1: 0.4838, Top5: 0.7830
Epoch 7, Loss: 1.9903, Top1: 0.4605, Top5: 0.7706
Epoch 8, Loss: 2.0219, Top1: 0.4517, Top5: 0.7644
Epoch 9, Loss: 2.0470, Top1: 0.4441, Top5: 0.7611
Epoch 10, Loss: 2.0971, Top1: 0.4270, Top5: 0.7525
Token généré : 113, Probabilité : 0.0006
Token généré : 8, Probabilité : 0.0758
Token généré : 64, Probabilité : 0.0097
Token généré : 11, Probabilité : 0.0628
Token généré : 12, Probabilité : 0.2122
Token généré : 17, Probabilité : 0.6002
Token généré : 4, Probabilité : 0.5715
Token généré : 26, Probabilité : 0.0614
Token généré : 11, Probabilité : 0.5598
Token généré : 8, Probabilité : 0.0999
Token généré : 71, Probabilité : 0.0042
Token généré : 1, Probabilité : 0.4420

Traduction de 'hello' = '* Nestale 4'
(env) PS C:\Users\Marius\Desktop\Projet_Methodes_Apprentissages>
```

Je n'ai pas passé trop de temps à entraîner car comme indiqué dans la consigne et le code va changer donc pas la peine, par contre on peut quand même tenter d'interpréter les métriques: ***loss, top-1 et top-5***.

Sur 10 epochs, la loss diminue légèrement puis ré augmente ce qui peut être dû à la capacité limité du modèle ou que le dataset est trop complexe pour un RNN *simple*. La *top-1 accuracy* est en baisse constante, donc le modèle devient moins performant pour prédire exactement le bon caractère au fil des époques, c'est un signe d'une mauvaise convergence ou qu' il sur-apprend, top-5 ici n'est pas très fiable. On voit que la traduction attendue n'est pas du tout correcte et certains tokens ont une probabilité très faible. Donc le modèle doit sampler des caractère improbables (*,4).Je pense que le

sampling naïf (prenant des caractères selon leur probabilité sans contrainte) conduit à une génération incohérente en plus de l'approche caractère par caractère du tokenizer.

Teacher Forcing est utilisé dans le modèle au moment de l'entraînement, il me permet de fournir au modèle la séquence correcte (ground truth) à chaque étape du décodage, plutôt que de se fier aux prédictions précédentes. Le modèle converge mieux, l'accélère et réduit ainsi les erreurs.

```
#teacher forcing
# On sépare la cible en (tgt_input, tgt_target)
tgt_input = tgt[:, :-1]   # tout sauf le dernier token
tgt_target = tgt[:, 1:]   # tout sauf le premier token

# Passage modèle
logits = model(src, tgt_input)
```

Donc les prochaines étapes seront de réaliser un tokenizer avancé avec SentencePiece pour pouvoir travailler avec des sous mots au lieu du simple caractères, et ensuite passer à l'architecture Transformer et implémenter un top-p sampling pour générer des traductions plus cohérentes (enfin…)

Partie 2: Tokenizers:

Voici les ajustements que je vais devoir faire pour implémenter deux tokenizers avec SentencePiece:

• Entraîner un tokenizer par langue.
• Remplacer le tokenizer par nos deux nouveaux tokenizers entraînés.
• Modifier la fonction qui génère la traduction pour prendre en compte les nouveaux tokenizers.

1) Je dois d'abord créer un sous-ensemble pour entraîner les tokenizers.
   On extrait aléatoirement 5 000 paires anglais/français du dataset, parmis les 100000 lignes extraites du df initial, comme ça on optimise la mémoire, le temps de calcul tout en capturant des relations dans les langues.

2) Ensuite les tokenizers (anglais et français) sont entraînés séparément, avec les paramètres suivants:

```
#Entraînement du tokenizer anglais
spm.SentencePieceTrainer.train(
    input='en_sub.txt',
    model_prefix='spm_en',
    vocab_size=5000,
    model_type='bpe', #BPE plutôt que le default cf. compte-rendu
    user_defined_symbols=['<pad>','<bos>', '<eos>']
)
#Entraînement du tokenizer français
spm.SentencePieceTrainer.train(
    input='fr_sub.txt',
    model_prefix='spm_fr',
    vocab_size=5000,
    model_type='bpe',
    user_defined_symbols=['<pad>','<bos>', '<eos>']
)
```

BPE est un algorithme de segmentation assez flexible.

3) La classe SentencePieceTokenize vient améliorer le simple Tokenizer de base, dans le but d'encapsuler l'encodage et le décodage des phrases.
On encode en convertissant la phrase en une liste d'ID et on décode en ignorant le token <pad> (voir notebook).

Phrase source : "This is a test." Phrase encodée : [1,2, 13, 42, 7, 5, 28, 2] # (avec <bos> et <eos>) Phrase décodée : "This is a test."

Ici par exemple on observe un résultat (on suppose <bos> = 1 et <eos> = 2).
<pad>, <bos> et <eos> sont respectivement les tokens qui permettent le padding (remplissage des séquences dans un batch), le début et la fin d'une séquence.

4) On ajoute quelques fonctionnalités un peu plus complexe à notre RNN, je rajoute un dropout, la nonlinearity avec relu, plusieurs couches

```
Seq2SeqRNN(
  (embedding_src): Embedding(5000, 512)
  (embedding_tgt): Embedding(5000, 512)
  (dropout): Dropout(p=0.2, inplace=False)
  (encoder): RNN(512, 1024, num_layers=4, batch_first=True, dropout=0.2)
  (decoder): RNN(512, 1024, num_layers=4, batch_first=True, dropout=0.2)
  (fc_out): Linear(in_features=1024, out_features=5000, bias=True)
)
```

5) J'ajoute ensuite plus loin une fonction collate_fn() qui est vraiment utile et m'a permis de corriger certains problèmes, car par défaut, le DataLoader de PyTorch tente de créer un batch en empilant directement les données brutes. Si les séquences ont des longueurs variables, cela entraînera une erreur. La fonction collate_fn() est donc essentielle pour aligner les séquences suivant le padding, elle prend en entrée un batch et renvoie deux tenseurs contenant les séquences cibles et sources paddées. Le padding n'est rien d'autre qu'un 0 qui "remplit" les listes afin qu'elles aient des longueurs identiques.

6) Voici un tableau qui compare la fonction qui génère la traduction:

Particularités	Fonction de base	Fonction modifiée
Gestion des tokenizers	Un seul tokenizer caractère par caractère	deux tokenizers : sp_en et sp_fr avec Sentencepiece

Encodeur/ Décodeur	Encodeur et Décodeur partagés	Encodeur et Décodeur séparés
Longueur max	max_len = 50	max_len = 80
Fin	tokenizer.char2id["<eos >"]	sp_fr_eos.id

7) Enfin SentencePiece est un tokenizer efficace et flexible qui malgré son score assez faible sur le top1 et le top5 reste en augmentation par rapport au tokenizer simple dont les métriques ne faisaient que décroitre d'autant plus qu'il possède un vocabulaire plus important, il est censé mieux généralisé, peut gérer les mots inconnus.

Conclusion:

Dans cette partie, nous avons enrichi notre pipeline en introduisant SentencePiece Tokenizers, un vocabulaire basé sur les sous-mots, pour remplacer la tokenization caractère par caractère. Ce changement a permis de réduire la longueur des séquences et d'améliorer la représentativité du vocabulaire. Parallèlement, un modèle RNN plus complexe a été utilisé, avec plusieurs couches et des embeddings de plus grande dimension, pour capturer des relations plus riches dans les séquences. Ces ajustements structurent une base solide pour des optimisations futures, bien que les performances et la cohérence des traductions nécessitent encore des ajustements pour exploiter pleinement cette architecture plus avancée.

Partie 3: Transformer

Cette partie consiste à remplacer le modèle RNN précédent par une architecture Transformer. Ce changement permet d'exploiter les mécanismes d'attention pour mieux capturer les relations à longue portée dans les séquences, tout en permettant un traitement plus efficace grâce à la parallélisation (lecture des séquences en même temps).

On commence par le positional encoding car le Transformer contrairmeent au RNN ne possède pas de mécanisme de traitement d'ordre des séquences
Chaque token est ensuite associé à un vecteur pour être utilisé par le Transformer, l'architecture se compose de 6 couches, 8 têtes d'attention par couches, et une projection linéaire.

Utilisation du tokenizer SentencePiece pour encoder les paires anglais/français.
Limitation des séquences à une longueur maximale de 50 tokens.

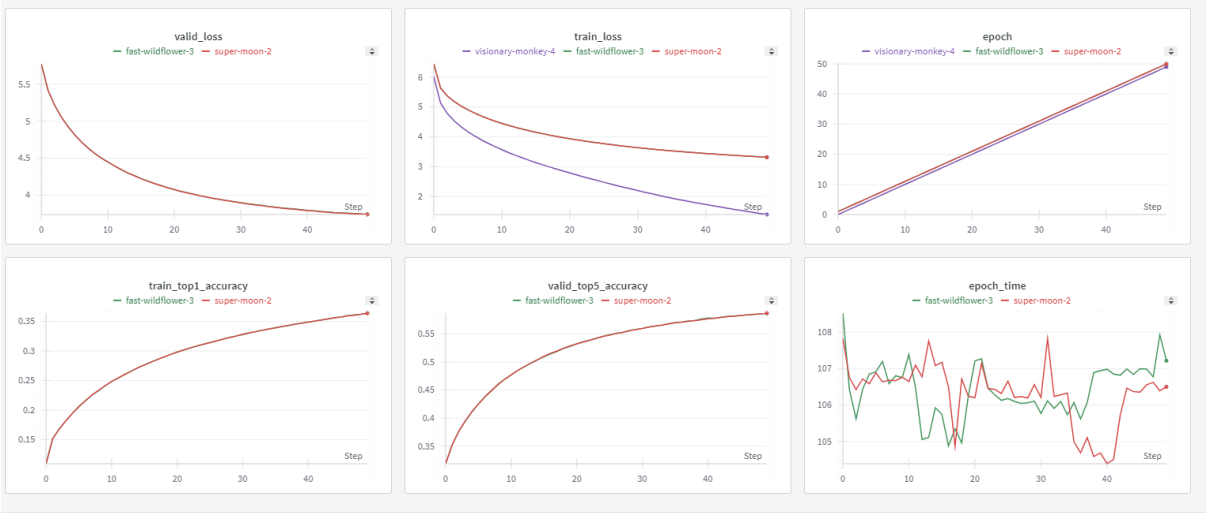
Les données sont préparées via un DataLoader avec un batch de 32 exemples.
Perte : CrossEntropyLoss, on entraîne sur 50 époques puis visualisation sur wandb pour suivres les métriques de chaques runs, et ainsi voir si la loss converge bien

Au début j'avais fait en train test split ce qui me permettait de voir s'il le modèle overfitter ou pas grâce au validation set, puis j'ai supprimer pour garder que la train loss.

```
71,806,544 total parameters.
71,806,544 training parameters.
Seq2SeqTransformer(
  (transformer): Transformer(
    (encoder): TransformerEncoder(
      (layers): ModuleList(
        (0-5): 6 x TransformerEncoderLayer(
          (self_attn): MultiheadAttention(
            (out_proj): NonDynamicallyQuantizableLinear(in_features=512, out_features=512, bias=True)
          )
          (linear1): Linear(in_features=512, out_features=2048, bias=True)
          (dropout): Dropout(p=0.2, inplace=False)
          (linear2): Linear(in_features=2048, out_features=512, bias=True)
          (norm1): LayerNorm((512,), eps=1e-05, elementwise_affine=True)
          (norm2): LayerNorm((512,), eps=1e-05, elementwise_affine=True)
          (dropout1): Dropout(p=0.2, inplace=False)
          (dropout2): Dropout(p=0.2, inplace=False)
        )
      )
      (norm): LayerNorm((512,), eps=1e-05, elementwise_affine=True)
    )
    (decoder): TransformerDecoder(
      (layers): ModuleList(
        (0-5): 6 x TransformerDecoderLayer(
          (self_attn): MultiheadAttention(
            (out_proj): NonDynamicallyQuantizableLinear(in_features=512, out_features=512, bias=True)
          )
          (multihead_attn): MultiheadAttention(
            (out_proj): NonDynamicallyQuantizableLinear(in_features=512, out_features=512, bias=True)
          )
          (linear1): Linear(in_features=512, out_features=2048, bias=True)
          (dropout): Dropout(p=0.2, inplace=False)
          (linear2): Linear(in_features=2048, out_features=512, bias=True)
          (norm1): LayerNorm((512,), eps=1e-05, elementwise_affine=True)
          (norm2): LayerNorm((512,), eps=1e-05, elementwise_affine=True)
          (norm3): LayerNorm((512,), eps=1e-05, elementwise_affine=True)
          (dropout1): Dropout(p=0.2, inplace=False)
          (dropout2): Dropout(p=0.2, inplace=False)
          (dropout3): Dropout(p=0.2, inplace=False)
        )
      )
      (norm): LayerNorm((512,), eps=1e-05, elementwise_affine=True)
    )
  )
  (generator): Linear(in_features=512, out_features=18000, bias=True)
  (src_tok_emb): TokenEmbedding(
    (embedding): Embedding(18000, 512)
  )
  (tgt_tok_emb): TokenEmbedding(
    (embedding): Embedding(18000, 512)
  )
  (positional_encoding): PositionalEncoding(
    (dropout): Dropout(p=0.2, inplace=False)
  )
)
```

Assez content d'avoir fait ce modèle, c'est une première, il comporte sans doute trop de paramètres, j'ai vu sur internet des modèles avec 14 mo de paramètres qui arrivaient à des bons résultats pour des tâches de traductions mais j'ai voulu essayer quand même avec plus de couches et d'autres fonctionnalités pour faire des vrais runs sur wandb.

Exemple (tous les runs seront dans le dossier de rendu):



Pour mon dernier run, la loss passe de 6.0148 à 1.4019 ce qui est censé montré que le modèle à appris sur 50 époques, concernant ces questions:

<u>Pour conclure sur cette partie:</u>
Comparez votre Transformer avec votre RNN. Quel est le meilleur modèle ? Lequel est le plus rapide ? Lequel consomme le moins de mémoire ?

Personnellement, je préfère le Transformer pour ses performances et sa rapidité. Il est clairement mieux adapté aux besoins actuels de traduction automatique, surtout pour des phrases complexes. Cela dit, il est plus gourmand en mémoire, même si j'ai été agréablement surpris de constater, suite à nos discussions, qu'en réglant correctement la fonction collate, sa consommation devient constante, il reste plus rapide car il ne traite pas séquence par séquence comme dit précédemment .

Finalement, avec une bonne optimisation, le Transformer est vraiment efficace.

Je me suis également renseigné sur sa complexité, qui est $O(n^2)$ (quadratique), ce qui en fait un sujet de recherche actif pour réduire cette contrainte (notamment avec des approches comme Mamba). Malgré cela, je pense que le RNN reste une excellente alternative si les ressources matérielles sont limitées.

Pour le choix du meilleur modèle je dirais Transformer, pour ces raisons (parallélisme, rapidité, attentions)

Partie 4: Top-p sampling:

Le top-p sampling est une amélioration par rapport au sampling naïf initial.

Dans mon implémentation précédente, les logits étaient transformés en probabilités avec softmax, et un token était choisi aléatoirement via torch.multinomial, ce qui entraînait des incohérences car tous les tokens, même les moins probables, peuvent être sélectionnés.

Avec le top-p sampling, seuls les tokens les plus significatifs, dont la probabilité cumulée atteint un seuil pp, sont considérés. Cela permet aussi de filtrer les tokens improbables.

Cette méthode est censée améliorer la cohérence de mes traductions tout en pouvant jouer sur la diversité grâce au paramètre p à la température. Par rapport au sampling naïf, le top-p sampling offre des traductions plus logiques, réduit les erreurs et apporte un meilleur équilibre entre diversité et robustesse.

La temp contrôle la créativité : une température basse favorise les mots sûrs, une haute ajoute de la variété. Le p limite les choix aux mots les plus probables : avec p=0.85, seuls ceux qui cumulent 85 % des probabilités sont pris en compte. Ces deux paramètres équilibrent diversité et cohérence.

Voici un premier compte-rendu, on va continuer à optimiser et essayer d'autres architectures pour mieux traduire car pour le moment c'est encore approximatif même s'il capte et sais reconnaître les langues, il faudra aussi sans doute passer à une architecture GRU/RNN et ne pas transformer les types en bool pour le transformer.

A suivre pour le prochain compte rendu plus détaillé et avec des traductions plus justes ...

source:
https://codefinity.com/blog/Understanding-Temperature%2C-Top-k%2C-and-Top-p-Sampling-in-Generative-Models

FIN