Date remise: Lundi 14 novembre, 23h

Instructions

- *Ce devoir est difficile – commencez le bien en avance.*
- *Montrez votre démarche pour toutes les questions!*
- *votre rapport (PDF) et votre code électroniquement via la page Gradescope du cours. Votre rapport doit contenir des réponses au problème 2.5 et problème 3 (toutes les questions).*
- *Pour les expériences ouvertes (c.-à-d. les expériences qui n'ont pas de cas de test associés – Problème 2.5, Problème 3), vous n'avez pas besoin de soumettre de code - un rapport suffira.*
- *Les TAs pour ce devoir sont* ***Arian Khorasani et Nanda Harishankar Krishna***

Lien (Notebook): cliquez ici

Dans ce devoir, vous effectuerez **neural machine translation** sur le Multi30k dataset, entre Anglais et Français. L'ensemble de données contient des paires de phrases en français et en anglais. Les modèles de langage séquentiels *next-word prediction*: ils prédisent des jetons dans une séquence un à la fois, avec chaque prédiction basée sur tous les éléments précédents de la séquence. Un modèle de langage séquentiel peut ensuite être utilisé pour générer de nouvelles séquences de texte, en conditionnant chaque prédiction sur le passé *predictions*. Vous développerez des modèles d'encodeur-décodeur qui traitent la phrase à partir de la langue source et la traduire dans la langue cible.

Vous allez implémenter un modèle Seq2Seq (encodeur-décodeur) utilisant **GRUs**, et un encoder-decoder **Transformer** model. Dans le problème 1, vous utiliserez des modules PyTorch intégrés pour implémenter un encoder-decoder GRU-based model. Dans le problème 2, vous allez implémenter différents blocs de construction d'un transformateur, y compris **LayerNorm** (layer normalization) et le **Attention** mechanism.

Nous avons déjà prétraité l'ensemble de données Multi30k et fourni à vous sous forme de chargeurs de données qui fournissent séquences de longueur fixe. Tout au long de cette mission, **toutes les séquences auront une longueur de 60**, et nous utiliserons zéro-padding pour tamponner des séquences plus courtes. Chaque échantillon de l'ensemble de données est une ligne de la `source` (the input sentence in English) et le `target` (the target sentence en Anglais). Ils ont déjà été encodés, Vous n'avez donc pas besoin d'effectuer un autre prétraitement. Chaque minilot du chargeur de données contient 128 séquences à traduire.

Tests: Nous avons fourni des cas types publics pour certaines fonctions pour que vous puissiez vérifier votre code. Par exemple, nous avons fourni des échantillons d'entrées aléatoires pour vérifier votre `forward` méthods. Notez que ces tests ne sont pas complets et vous serez noté sur un ensemble séparé de tests sur Gradescope. Si les tests sur Gradescope échouent, en règle générale `x` correspond à la valeur de votre mission (e.g. la valeur retournée par votre fonction), et `y` est la valeur attendue.

Embeddings: Dans le modèle GRU Seq2Seq et le transformateur, la couche d'enrobage est parmi les couches qui contiennent le plus de paramètres. En effet, il se compose d'une matrice de taille `(vocabulary_size, embedding_size)` (noter qu'ici `vocabulary_size` est égal à $N + 1$, pour tenir compte du zéro-padding). Dans ce devoir, nous apprenons les noyaux, c'est-à-dire que nous n'utilisons pas les noyaux pré-formés.

Instructions de codage: Vous devrez utiliser PyTorch pour répondre à toutes les questions. De plus, ce devoir **nécessite d'exécuter les modèles sur GPU** (autrement cela prendra un temps incroyablement long); si vous n'avez pas accès à vos propres ressources (e.g. votre propre machine, un cluster), veuillez utiliser Google Colab (the notebook `solution.ipynb` est ici pour vous aider). Pour certaines questions, il vous sera demandé de ne pas utiliser certaines fonctions dans PyTorch et implémenter ces vous-même en utilisant des fonctions primitives de `torch`.

Problem 1

Implémentation de GRU-based Seq2Seq model (17 pts): Dans ce problème, vous utiliserez les modules existants de PyTorch pour implémenter un GRU-based Seq2Seq model. The architecture you will be asked to implement is the following:

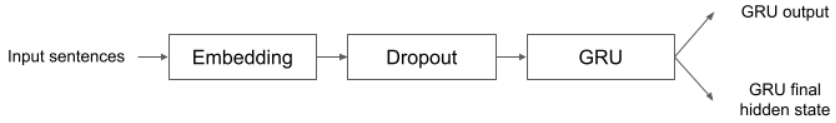

Figure 1: Encoder model

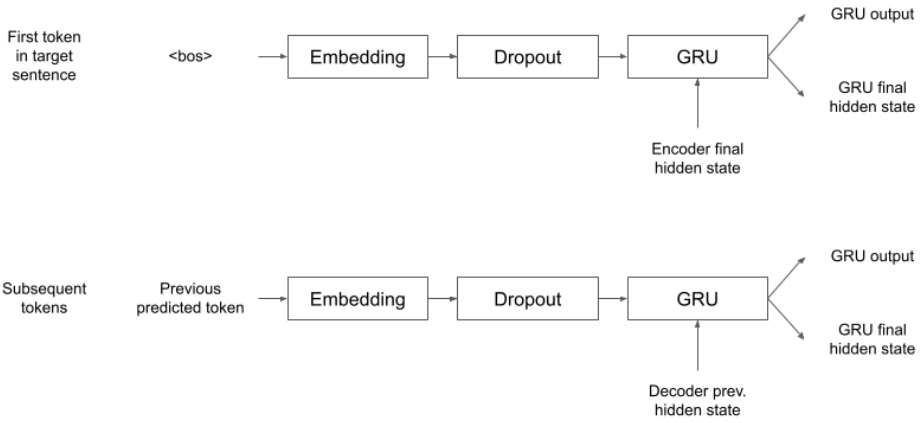

Figure 2: Decoder model

vous sont indiqués:

- `Encoder` class contenant tous les blocs nécessaires à la création du modèle d'encodeur. En particulier, `self.embedding` est une `nn.Embedding` module qui convertit des séquences d'indices de jetons en intégration, `self.dropout` est une instance de `nn.Dropout` and `self.gru` est un `nn.GRU` module qui exécute un GRU sur une séquence de vecteurs.
- `Decoder` class contenant tous les blocs nécessaires à la création du modèle décodeur. En particulier, `self.embedding` est une `nn.Embedding` module qui convertit un index de jeton (le précédent jeton prédit) en une intégration, `self.dropout` est une instance de `nn.Dropout` and `self.gru` est une `nn.GRU` module qui exécute un GRU qui prend le token intégré et l'état caché précédent (pour le premier jeton cible, il s'agit de l'état caché final de l'encodeur) en entrée, et retourne l'état caché mis à jour et la sortie.
- `Seq2Seq` class qui rassemble le modèle. Il s'agit des modèles d'encodeur et de décodeur et effectue l'ensemble du processus d'encodage et de décodage.

1. (5 pts) pour `Encoder`, complétez des `__init__()` et `forward()` fonctions. The `forward()` fonction doit retourner les sorties et l'état caché final du GRU.
2. (5 pts) pour `Decoder`, complétez des `__init__()` et `forward()` fonctions. le `forward()` fonction prend en entrée le token précédent et l'état caché précédent et doit retourner les sorties et l'état caché final du GRU.
3. (5 pts) pour `Seq2Seq`, complétez des `__init__()` et `forward()` fonctions. le `forward()` fonction prend en charge les phrases source et cible. Vous mettrez également en œuvre le décodage gourmand et l'échantillonnage aléatoire avec la température pour le décodage.
   - pour de décodage glouton, à chaque étape de décodage, vous devez choisir le jeton avec le score le plus élevé avec avidité.
   - Pour échantillonnage aléatoire avec température, vous calculerez les probabilités de jetons à chaque étape de décodage comme

     $$p(x_i|x_{1:i-1}) = \frac{\exp\left(\frac{o_i}{t}\right)}{\sum_{j=0}^{n_{\text{tokens}}-1} \exp\left(\frac{o_j}{t}\right)}$$

     où x_i sont des jetons au pas de temps i, o_i représentent les valeurs prédites par le décodeur au pas de temps i pour chaque jeton dans le vocabulaire et t représente le facteur de température entre 0 et 1.

     En utilisant ces probabilités, vous allez échantillonner un jeton à chaque étape du décodeur.
4. (2 pts) Complétez le `get_criterion()` fonction, qui retourne un `nn.CrossEntropyLoss` objet qui ignore l'index de remplissage (comme spécifié dans Notebook).

Problem 2

Implémentation un transformateur encodeur-décodeur (27pts) Alors que les RNNs vont typiquement "retenir" l'information passée en prenant l'état caché à chaque temps, ces dernières années ont vues l'émergences de méthodes qui utilisent l'information du passée de manières différentes. Les transformers[1] est une relativement nouvelle architecture qui utilise plusieurs têtes auto-attentives (*self-attention heads* en anglais) en parallèle, en plus de spécificités architecturales. Implémenter un transformer est un processus assez laborieux – nous vous donnons donc la structure du code, et votre tâche est d'implémenter uniquement le mécanisme d'attention à multi-têtes par produit scalaire pondéré (*multi-head scaled dot-product attention mechanism* en anglais), ainsi que l'opération de layernorm.

Implémenter Layer Normalization (5pts): Vous allez dans un premier temps implémenter la technique de layer normalization (LayerNorm) que vous avez vu en classe. Pour ce devoir, **vous n'êtes pas autorisés** à utiliser le module PyTorch `nn.LayerNorm` (ou toute fonction appelant `torch.layer_norm`).

Tel que défini dans l'article sur la layer normalization, l'opération layernorm sur un minibatch d'entrées x est définie par

$$\text{layernorm}(x) = \frac{x - \mathbb{E}[x]}{\sqrt{\text{Var}[x] + \epsilon}} * \texttt{weight} + \texttt{bias}$$

où $\mathbb{E}[x]$ est l'espérance de x, $\text{Var}[x]$ est la variance sur x, ici toutes les deux prises sur la dernière dimension du tenseur x seulement. `weight` et `bias` sont des paramètres affines apprenables.

1. Dans le fichier `gpt1_solution_template.py`, implémentez la fonction `forward()` de la classe `LayerNorm`. Faîtes attention à utiliser les slides de cours pour connaître les détails exacts sur la manière dont $\mathbb{E}[x]$ et $\text{Var}[x]$ sont implémentées. En particulier, la fonction `torch.var` en PyTorch utilise par défaut une estimation non biaisée de la variance, définie par la formule de gauche

   $$\overline{\text{Var}}(X)_{\text{unbiased}} = \frac{1}{n-1}\sum_{i=1}^{n}(X_i - \overline{X})^2 \qquad \overline{\text{Var}}(X)_{\text{biased}} = \frac{1}{n}\sum_{i=1}^{n}(X_i - \overline{X})^2$$

   alors que LayerNorm utilise l'estimateur biaisé de droite (où $\overline{X}$ ici est l'estimateur de la moyenne). Veuillez vous référer aux docstrings de cette fonction pour plus d'informations concernant les signatures d'entrée/sortie.

Implémenter le mécanisme d'attention (17pts): Vous allez maintenant implémenter le coeur du transformer – le mécanisme d'attention à multi-têtes. En supposant que vous ayez m têtes

[1] Voir https://arxiv.org/abs/1706.03762 pour plus de détails.

d'attention, le vecteur d'attention pour la tête à l'indice i est donné par:

$$[\boldsymbol{q}_1, \ldots, \boldsymbol{q}_m] = \boldsymbol{Q}\boldsymbol{W}_Q + \boldsymbol{b}_Q \qquad [\boldsymbol{k}_1, \ldots, \boldsymbol{k}_m] = \boldsymbol{K}\boldsymbol{W}_K + \boldsymbol{b}_K \qquad [\boldsymbol{v}_1, \ldots, \boldsymbol{v}_m] = \boldsymbol{V}\boldsymbol{W}_V + \boldsymbol{b}_V$$
$$\boldsymbol{A}_i = \text{softmax}\left(\frac{\boldsymbol{q}_i \boldsymbol{k}_i^\top}{\sqrt{d}}\right)$$
$$\boldsymbol{h}_i = \boldsymbol{A}_i \boldsymbol{v}_i$$
$$A(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = \text{concat}(\boldsymbol{h}_1, \ldots, \boldsymbol{h}_m)\boldsymbol{W}_O + \boldsymbol{b}_O$$

Ici $\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}$ sont les requêtes, clés et valeurs respectivement, où toutes les têtes ont été concaténées en un seul vecteur (e.g. ici $\boldsymbol{K} \in \mathbb{R}^{T \times md}$, où d est la dimension d'un seul vecteur de clé, et T est la longueur de la séquence). $\boldsymbol{W}_Q, \boldsymbol{W}_K, \boldsymbol{W}_V$ sont les matrices de projections correspondantes (avec biais $\boldsymbol{b}$), et $\boldsymbol{W}_O$ est la projection de sortie (avec biais $\boldsymbol{b}_O$). $\boldsymbol{Q}, \boldsymbol{K}$, et $\boldsymbol{V}$ sont donnés par la sortie de la couche précédente du réseau principal. $\boldsymbol{A}_i$ sont les valeurs d'attention, qui spécifient sur quels éléments de la séquence d'entrée chaque tête doit porter son attention. Dans cette question, **vous n'êtes pas autorisés**) utiliser le module `nn.MultiheadAttention` (ou tout fonction appelant `torch.nn.functional.multi_head_attention_forward`). Veuillez vous référer aux docstrings de chaque fonction pour un description précise de chaque fonction, ainsi que les tailles des tenseurs d'entrée/sortie.

2. Les équation ci-dessus demandent plusieurs manipulations vectorielles pour découper et combiner les vecteurs de têtes ensemble. Par exemple, les requêtes concaténées $\boldsymbol{Q}$ sont découpées en m vecteurs $[\boldsymbol{q}_1, \ldots, \boldsymbol{q}_m]$ (un pour chaque tête) après une projection affine par $\boldsymbol{W}_Q$, et les h_i's sont ensuite concaténés à nouveau pour la projection affine par $\boldsymbol{W}_O$. Dans la classe `MultiHeadedAttention`, implémentez les fonctions d'utilité `split_heads()` et `merge_heads()` pour effectuer ces deux opérations, ainsi qu'une transposition par commodité pour plus tard. Par exemple, pour la 1ère séquence du mini-batch:

   ```
   y = split_heads(x)  →  y[0, 1, 2, 3] = x[0, 2, num_heads * 1 + 3]
   x = merge_heads(y)  →  x[0, 1, num_heads * 2 + 3] = y[0, 2, 1, 3]
   ```

   Ces deux fonctions sont exactement inverses l'une de l'autre. Notez que dans le code, le nombre de têtes m est appelé `self.num_heads`, et la dimension d'une tête d est `self.head_size`. Vos fonctions doivent traiter des mini-batch de séquences de vecteurs, voir les docstrings pour les détails sur les signatures d'entrée/sortie.

3. Dans la classe `MultiHeadedAttention`, implémentez la fonction `get_attention_weights()`, qui retourne les $\boldsymbol{A}_i$'s (pour toutes les têtes d'un coup) à partir des $\boldsymbol{q}_i$'s et des $\boldsymbol{k}_i$'s. Rappelez vous que le modèle de langage ici est *auto-régressif* (parfois également appelé *causal*), ce qui signifie que l'attention doit être calculée seulement sur les entrées passées, et jamais sur le futur.

   Concrètement, cela veut dire qu'au lieu de prendre le softmax sur l'ensemble de la séquence, on doit introduire un masque binaire $\boldsymbol{s}_t$ (qui est différent de la clé `mask` du dataloader), où $s_t(\tau)$ vaut 1 si l'élément courrant peut porter son attention sur la position τ de la séquence (i.e. si $\tau \leq t$), et 0 sinon. Le softmax est ainsi modifié en

$$[\text{softmax}(\boldsymbol{x}, \boldsymbol{s}_t)]_\tau = \frac{\exp(x_\tau) s_t(\tau)}{\sum_i \exp(x_i) s_t(i)}.$$

En pratique, pour éviter les potentielles instabilités numériques, nous vous recommendons d'utiliser une implémentation différentes:

$$[\text{softmax}(\boldsymbol{x}, \boldsymbol{s}_t)]_\tau = \frac{\exp(\widetilde{x}_\tau)}{\sum_i \exp(\widetilde{x}_i)} \qquad \text{where} \qquad \widetilde{x}_\tau = x_\tau s_t(\tau) - 10^4(1 - s_t(\tau))$$

La deuxième version est presque équivalente à la première (à erreur numérique près), tant que $x \gg -10^4$, ce qui est le cas en pratique. Nous vous recommendons fortement de vectoriser vous opérations le plus possible pour accélérer l'apprentissage en Problème 3.

4. En utilisant les fonction que vous avez implémentées, complétez la fonction `apply_attention()` de la classe `MultiHeadedAttention`, qui calcule les vecteurs $\boldsymbol{h}_i$'s en fonction des $\boldsymbol{q}_i$'s, $\boldsymbol{k}_i$'s et $\boldsymbol{v}_i$'s, et concatène les vecteurs de têtes.

   $$\texttt{apply\_attention}(\{\boldsymbol{q}_i\}_{i=1}^m, \{\boldsymbol{k}_i\}_{i=1}^m, \{\boldsymbol{v}_i\}_{i=1}^m) \texttt{ = } \text{concat}(\boldsymbol{h}_1, \ldots, \boldsymbol{h}_m).$$

5. En utilisant les fonction que vous avez implémentées, complétez la fonction `forward()` de la classe `MultiHeadedAttention`. Vous pouvez implémenter les différentes projections affines de n'importe quelle manière vous le souhaiter (n'oubliez pas les biais), et vous pouvez ajouter des modules dans la fonction `__init__()`. Combien de paramètres apprenables a votre module, en fonction de `num_heads` et `head_size`?

La passe avant et décodage (5pts): Vous avez maintenant tous les éléments pour implémenter la passe avant (*forward pass* en anglais) d'une version miniature de Encoder Decoder Transformer. Vous avez également un module appelé `EncoderLayer` et `DecoderLayer`, qui correspond à un block entier d'auto-attention (et l'attention encodeur-décodeur dans DecoderLayer) en utilisant les modules `LayerNorm` et `MultiHeadAttention` que vous avez implémentés auparavant. Vous allez ensuite implémenter 2 stratégies de décodage dans la fonction compute BLEU.

6. Dans cette partie de l'exercice, vous allez compléter la classe `TEncoder` et `TDecoder`. Ce module contient tous les éléments nécessaires pour créer ce modèle. En particulier, `self.embedding` est un module responsable de convertir des séquences de tokens entiers en embeddings (avec les embeddings d'entrée et positionnels), `self.layers` est un `nn.ModuleList` contenant les différentes couches `EncoderLayer` ou `DecoderLayer`, et `self.classifier` est une couche linéaire.

7. Implémentez le greedy decoding et random sampling avec température dans la fonction de `compute bleu`. La mise en œuvre est presque identique à celle du problème 1.

Problem 3

Modèles linguistiques de formation et comparaison de modèles (25pts): Pour la traduction automatique, il est normal de déclarer la note de l'BLEU comme mesure de la performance d'un modèle. l est défini comme "la moyenne géométrique pondérée de toutes les précisions n-gram précisions, multiplié par la peine de brièveté". vous pouvez en lire plus à ce sujet ici. Nous fournissons un code pour calculer la note BLEU. (après avoir terminé les stratégies de décodage dans la fonction `compute bleu`). Nous vous demandons de rapporter seulement. BLEU-1 et BLEU-2, c.-à-d. pour une précision de 1 gramme et la moyenne géométrique des précisions de 1 et 2 grammes respectivement. Dans la recherche, il est standard de rapporter la moyenne géométrique des précisions de 1 à 4 grammes.

Pendant l'entraînement, nous examinerons aussi la perplexité, qui est l'exponentielle de la perte d'entropie croisée. Bien qu'il soit courant de surveiller l'BLEU pendant la formation, Ne faites pas cela parce qu'il augmente considérablement. le temps de calcul (Comme vous devez exécuter le décodage).

Le but de cette question est d'effectuer l'exploration et l'évaluation de modèles. Vous allez former chacune des configurations spécifiées et enregistrer les métriques que nous demandons dans votre le fichier de rapport.

Note: Pour chaque expérience, observer de près les courbes de formation et signaler la validation la plus faible loss/perplexité sur epochs (pas nécessairement le score de validation pour la dernière epoch).

Configurations pour éxecuter:

Pour le modèle GRU et le transformateur, exécuter les configurations suivantes:

1. Adam optimizer, `use_dropout=False`, greedy decoding
2. SGD optimizer, `use_dropout=True`, greedy decoding
3. Adam optimizer, `use_dropout=True`, greedy decoding
4. Adam optimizer, `use_dropout=True`, random sampling with temperature

1. On vous demande d'exécuter 8 expériences avec différentes architectures, optimisateurs et stratégies de décodage. Ces paramètres vous sont donnés sous forme de cellules distinctes dans notebook. Pour chacune de ces 8 expériences, tracer des courbes d'apprentissage (train et validation) de la perte et perplexité sur les deux **epochs** et **time**. Les figures doivent comporter des axes marqués, une légende et une légende explicative.
2. Faire un tableau des résultats résumant le train et le rendement de validation pour chaque expérience, indiquant la stratégie d'architecture, d'optimisation, d'abandon et de décodage.

Trier par architecture, puis `use_dropout`, puis optimiseur et enfin par stratégie de décodage. Gras le meilleur résultat pour chaque architecture.[2] Le tableau devrait avoir une légende explicative et des en-têtes de colonne et/ou de ligne appropriés. Tout raccourci ou symbole dans le tableau doit être expliqué dans la légende.

3. Quel modèle et quelle configuration utiliseriez-vous si vous étiez le plus préoccupé par le temps? avec performances de généralisation? Quel a été l'impact du Dropout?

4. Comparer le GRU et le transformateur en fonction de la configuration de l'expérience: Adam optimizer, `use_dropout=False`, greedy decoding. Selon vous, quel modèle a donné de meilleurs résultats? Pourquoi?

5. Vous avez formé un transformateur avec quelques configurations différentes dans cette expérience. Compte tenu des récents modèles de langage basés sur les transformateurs, Les résultats sont-ils conformes à vos attentes? Spéculez sur les raisons.

6. Pour chaque des configurations d'expérience ci-dessus, mesurer l'utilisation moyenne de la mémoire GPU à l'état stationnaire (`nvidia-smi` est votre ami!). Commentaire sur les empreintes mémoire GPU de chaque modèle, discuter des raisons de l'augmentation ou de la diminution de la consommation de mémoire, le cas échéant.

7. commenter le comportement excessif des différents modèles que vous avez formés. Est-ce qu'une classe particulière de modèles a dépassé les autres plus facilement ? Pouvez-vous faire une estimation des différentes mesures qu'un praticien peut prendre pour éviter le surenchérissement dans ce cas?

[2] Vous pouvez également faire le tableau dans LaTeX; pour plus de commodité, vous pouvez utiliser des outils comme LaTeX table generator pour générer des tables en ligne et obtenir le code LaTeX correspondant.