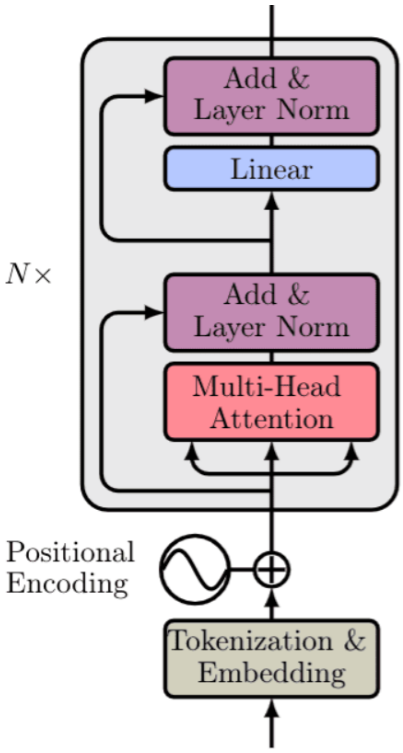
UNIVERSITÉ DE TOULON

UE73
TP 3
Le transformer encoder



Enseignant : Monsieur Chareyre

Master 1 Informatique | Université de Toulon | 25 décembre 2025

Résumé

L'objectif de ce TP est double. Dans une première partie, vous améliorerez l'instrumentation de votre entraînement en suivant correctement les métriques, la loss et la dynamique des gradients, et en mettant en place une stratégie de sauvegarde du meilleur modèle. Dans une seconde partie, vous implémenterez puis entraînerez un modèle basé sur un *Transformer Encoder* (d'abord via le module PyTorch, puis en réécrivant les blocs principaux), afin de comparer ses performances à votre MLP, ainsi que les compromis en termes de complexité et de nombre de paramètres.

Question 1

Suivi avancé des métriques et sauvegarde du meilleur modèle

Dans le TP précédent, vous avez mis en place une boucle d'entraînement et calculé des métriques (accuracy, recall, F1, etc.). Dans les prochaines questions, vous allez améliorer la traçabilité de l'apprentissage et mettre en place un mécanisme robuste de sélection du meilleur modèle au cours d'un entraînement.

1.1 Courbes de métriques par époque

À partir de votre code existant, écrivez une fonction `show_metrics_per_epoch` qui :
- prend en entrée l'historique des métriques (au minimum `F1`, `accuracy`, `recall`) calculées à chaque époque ;
- produit un graphique `matplotlib` montrant l'évolution de ces métriques en fonction du numéro d'époque ;
- sauvegarde le graphique dans un fichier (par exemple `metrics.png`).

1.2 Courbe de loss par époque

Écrivez une fonction `show_loss_per_epoch` qui :
- prend en entrée l'historique de la loss (`train_loss` et `valid_loss`) ;
- trace la courbe de loss en fonction des époques ;
- sauvegarde le graphique dans un fichier (par exemple `loss.png`).

1.3 Suivi de la norme des gradients

Écrivez une fonction `show_gradient_per_epoch` qui suit la dynamique des gradients durant l'entraînement.

Consigne : on souhaite suivre une grandeur représentative, par exemple la *norme* ℓ_2 des gradients (globale ou moyenne) sur l'ensemble des paramètres du modèle.
- Pendant l'entraînement, à chaque époque, mesurez une norme de gradient (par exemple après un `backward()` sur un batch ou bien moyennée sur tous les batches).
- Stockez cette valeur et tracez son évolution en fonction des époques.
- Sauvegardez le graphique (par exemple `grads.png`).

1.4 Sauvegarde du meilleur modèle

Adaptez votre boucle d'entraînement afin de sauvegarder automatiquement le modèle dès que sa performance sur le *validset* est meilleure que toutes les performances précédentes, selon le critère suivant de valeur sur un métrique. Vous privilégierez le score f1.

— Sauvegardez le modèle entraîné
— Justifiez pourquoi le *validset* est utilisé pour sélectionner le meilleur modèle, plutôt que le *trainset* ou le *testset*.

Question 2

Utilisation d'un Transformer Encoder PyTorch

Dans cette question, vous allez entraîner un modèle basé sur un *Transformer Encoder* fourni par PyTorch, puis étudier ses contraintes d'entrée/sortie et comparer ses performances à votre MLP.

2.1 Encapsulation du Transformer Encoder

Écrivez une classe `PyTorchTransformerEncoder` héritant de `torch.nn.Module` qui contient un bloc `torch.nn.TransformerEncoder`. Votre classe devra :

— définir les hyperparamètres principaux (dimension d'embedding `d_model`, nombre de têtes `num_heads`, nombre de couches, dimension du feed-forward, dropout) ;
— implémenter la méthode `forward`.

2.2 Adapter la sortie du Transformer à une prédiction binaire

Le Transformer Encoder renvoie une sortie de dimension $k \times d$ (où k est une longueur de séquence et d la dimension d'embedding). Or, notre tâche de détection binaire nécessite une sortie scalaire (logit) pour que la loss puisse s'appliquer.

1. Proposez une stratégie pour passer de $k \times d$ à un scalaire.
2. Implémentez cette stratégie dans votre modèle.

2.3 Contrainte `d_model` divisible par `num_heads`

Comme vu en cours, il faut que :

$$d\_model \equiv 0 \pmod{\texttt{num\_heads}}.$$

1. Adaptez vos paramètres de spectrogramme (`n_fft`, `window_size`, `hop_length` et/ou découpage fréquentiel et/ou temporel) afin que la dimension de vos entrées soit compatible avec d_model.
2. Régénérez le dataset si nécessaire.

2.4 Transposée du spectrogramme et interprétation séquentielle

Le Transformer Encoder attend une entrée de dimension $k \times d$: k est la longueur de séquence (nombre de "tokens") et d la dimension des tokens.

1. Expliquez comment votre spectrogramme est interprété par le Transformer dans votre version actuelle (qu'est-ce qui joue le rôle de k ? qu'est-ce qui joue le rôle de d ?).
2. Testez l'ajout d'une transposée du spectrogramme avant l'entrée dans le modèle.
3. Expliquez l'effet de cette transposition sur l'interprétation des données (tokens temporels vs tokens fréquentiels) et discutez son impact potentiel sur les performances.

2.5 Recherche du meilleur seuil de classification

Jusqu'ici, vous utilisiez un seuil fixe à 0.5 pour binariser les probabilités et calculer les métriques. Or, ce seuil ne joue aucun rôle pendant l'entraînement (qui optimise une loss), mais influence fortement les métriques finales.

1. Écrivez une fonction `get_best_threshold` qui :
   — prend en entrée des logits ou probabilités et les labels associés ;
   — balaye plusieurs seuils (par exemple 0.0 à 1.0 avec un pas) ;
   — renvoie le seuil maximisant une métrique cible (par exemple le score F1).
2. Choisissez sur quel ensemble (*trainset*, *validset*, *testset*) il est le plus pertinent de déterminer ce seuil, et justifiez.

2.6 Comparaison MLP vs Transformer

Réalisez un bilan comparatif entre :
— votre modèle `MyMLP` ;
— votre modèle `PyTorchTransformerEncoder`.
Vous comparerez au minimum :

1. les performances (au moins sur *valid* puis sur *test* une seule fois en fin d'étude) ;
2. la vitesse d'entraînement (ordre de grandeur) ;
3. la quantité de paramètres (utilisez `sum(p.numel() for p in model.parameters())`).

Concluez : dans quel contexte l'un ou l'autre modèle semble-t-il préférable ?

Question 3

Réécriture d'un Transformer Encoder

Dans cette question, vous allez réécrire (simplifiée mais fonctionnelle) la partie *encodeur* de l'architecture Transformer vue en cours.

Vous devrez implémenter les blocs suivants :

- `PositionalEncoding`
- `EncoderModel`
  - `EncoderBlock`
    - `MultiHeadAttentionBlock` (utilisation de `torch.nn.MultiheadAttention` et `torch.nn.Layer`
    - `FeedForwardBlock` (utilisation de `torch.nn.Linear`, `torch.nn.ReLU`, `torch.nn.LayerNorm`)

Consignes :

1. Votre implémentation doit respecter le schéma global vu en cours.
2. Votre modèle final doit pouvoir être entraîné sur le même dataset que le modèle PyTorch (Question 2).

Vous êtes encouragés à consulter l'article fondateur : https://arxiv.org/abs/1706.03762

Attention : le papier original décrit une architecture encodeur + décodeur. Dans ce TP, seule la partie *encodeur* est requise.