Transfer Learning: Transfer Learning est une technique en machine learning où les informations apprises pour une tâche est réutilisée pour booster la performance sur une autre tâche. Ici, on a pris le modèle BERT de base et on l'a amélioré pour faire de la classification d'invite.

Classe Prompt_Dataset(Dataset) :

Attention Mask : retourne pour chaque token 1 si c'est un vrai token et 0 si c'est un padding token.

Padding Token : Token utilisé pour normaliser tous les vecteurs de tokens à la même longueur.

Fonction tokenizer.encode_plus() renvoie plus d'information qu'un encode normal à noter l'attention mask.

Fonction FloatTensor() transforme liste en Tensor de réels 32-bit.

Classe prompt_Data_Module(pl.LightningDataModule) :

Classe pl.LightningDataModule: Un datamodule encapsule les 5 étapes inclues dans le traitement de données dans PyTorch:

- Téléchargement/tokenization/traitement
- Nettoyer & sauver sur disque (optionel)
- Charger dans un **dataset**
- Appliquer transformations (rotation, tokenization, etc..)
- Mettre dans un **DataLoader**

Le **LightningDataModule** est une conventionpour traiter les données dans PyTorch Lightning. Il est censé encapsuler les **DataLoaders** d'entrainement, de test et de prediction ainsi que toute étape pour le traitement de données, téléchargement ou transformation.

En utilisant un LightningDataModule, on peut facileemnt developper une classe pour le traitement des datasets, changer les datasets rapidement et partager la séparation des données (pour le train/test) ou les transformations.

Il est donc **très important** de créer une classe fille de pytorch_lightning.LightningDataModule comme prompt_Data_Module ici car elle facilite les choses énormément.

Fonction DataLoader() : Pour accélérer l'entrainement du modèle et éviter l'overfitting, on va vouloir séparer notre dataset en différents batch de taille batch_size et mélanger les données au début de chaque époque ainsi que l'utilisation du multiprocessing de python. C'est donc ça que permet l'objet DataLoader().

Multiprocessing: utilisation de deux ou plus CPU sur une seule machine, permettant le traitement simultané de différentes parties d'un même programme

Classe Prompt_Classifier(pl.LightningModule) :

Hyperparamètre pretrained_model.config.hidden_size : réfère au hyperparamètre qui définit la taille des vecteurs dans les couches cachées du modèle

Couche self.hidden : Couche linéaire cachée qui sert de couche de transformation intermédiaire entre la sortie du modèle BERT et la couche de classification finale. Elle prend les vecteurs denses générés par BERT de taille pretrained_model.config.hidden_size et mes transforme à nouveau dans un espace de même dimension permettant donc au modèle d'apprendre des représentations plus spécifiques à la tâche (ici, classification de prompt) après l'encodage généraliste de BERT

Couche classifier : Couche qui va produire la sortie finale du modèle : un vecteur de taille n_labels, où chaque valeur représente un logit pour une classe. C'est la seule couche supervisée du modèle : elle relie les représentations apprises à la prédiction concrète d'un label. C'est sur ses sorties que la fonction de perte est calculée.

Fonction d'activation ReLU (F.relu()) : ReLu signifie Rectified Linear Unit. C'est une fonction d'activation non-linéaire avec moi formule mathématique : ReLU(x) = max(0,x). La non-linéarité qu'elle introduit permet d'éviter que plusieurs couches empilées se comportent comme une seule transformation linéaire ce qui limiterai l'apprentissage. C'est essentiel pour que le modèle puisse apprendre des relations complexes et on la place après les couches linéaires

Création du pipeline avec forward() :

1. Passage dans le modèle BERT

output = self.pretrained_model(inpute_ids = input_ids, attention_mask = attention_mask)

On passe le tesste tokenisé à BERT renvoyans un matrics de taille (batch_size, seq_length, hidden_size) soit une représentaion par token.

2. Moyenne des représentations des tokens

Pooled_output = torch.mean(output.last_hidden_state, 1)

On prend la moyenne sur la dimension des tokens (dim=1) pour obtenir une seule représentation par phrase. Pooled_output a maintenant une forme de (batch_size, hidden_size). La moyenne est une alternative au token CLS (classification) qui est toujours en première position dans les tokens de sortie de BERT, output.last_hidden_state[:, 0,:] fait donc référence à ce token.

3. Couche cachée (hidden layer)

pooled_output = self.hidden(pooled_output)

On passe la représentation du prompt dans une couche linéaire (Linear(hidden_size, hidden_size)). Elle permet au modèle de réapprendre ou affiner les représentations produites pr BERT, de3 façon spécifique à notre tâche de classification.

4. Dropout

Pooled_output = self.droupout(pooled_output)

Le dropout est une technique de régularisation utilisée pour réduire l'overfitting. Pendant l'entraînement, il met aléatoirement à 0 certains neurones de l'entrée avec une certaine probabilité (par défaut 0.5 si aucune n'est donnée). Cela force le modèle à ne pas de3venir trop dépendant de certaines activations spécifiques et à apprendre des représentations plus robustes. Placée ici, la couche empêche la couche finale (classification) contre le sur-apprentissage.

5. Activation ReLU

Pooled_output = F.relo(pooled_output)

On applique une non-linéarité après le dropout ce qui améliore la capacité d'expression du réseau.

6. Couche de classification

Pooled_output = self.classifier(pooled_output)

On passe donc les tenseurs finaux dans cette fonction et on obtient un logit par classe (0 ou 1)

7. Fonction de perte

BCEWithLogitsLoss calcule la fonction de perte pour des cas multi-label.

Weight decay loss : weight de3cay aussi connu comme régularisation L2 est une méthode communément utilisée pour régulariser un réseau neuronal. Il aide le modèle à apprendre des fonctions plus simples qui fonctionnent mi3eux la plupart du temps comparée à une fonction avec beaucoup de bruit. Le régularisateur weight_de3cay pousse les poids vers 0 petit à petit à chaque étape.

Optimisation des poids et calendrier d'apprentissage (learning rate schedule):

1. Définir l'optimiseur : AdamW (Adam Weights)

Optimizer= = AdamW(self.parameters(), lr=..., weight_decay=...)

Version améliorée d'Adam qui sépare le weight decay du calcul du gradient (comportement erroné dans l'algorithme d'Adam de base). Self.parameters() est l'ensemble des poids du modèle qu'on va chercher à optimiser. Avec la méthode de weight decay, on cherche à diminuer la capacité du modèle pour qu'un modèle qui over-fit ne le fait plus autant et est poussé automatiquement vers le "sweet spot". Cependant, des recherches récentes révèlent un phénomène nommé double descente profonde comme montrée dans l'image ci-dessous, montrant donc l'existence d'une troisième région. La théorie classique dit qu'en augmentant la complexité du modèle, on augmente l'erreur de test alors que plus récemment, on suppose qu'en augmentent la complexité du modèle,

on peut dépasser la zone d’over-fitting et passer en région de ”sur-paramétrisation”.