# Devoir 6
## Entraînement & Fine-tuning

Etienne Collin
20237904
contact@etiennecollin.com

Sciences des données - IFT3700

Section A
Professeur Gauthier Gidel

Université de Montréal
Automne 2024
20 Décembre 2024

# Table des matières

# 1. Partie 1

## 1.1. *Baseline*

Voici les arguments d'entraînement utilisés:

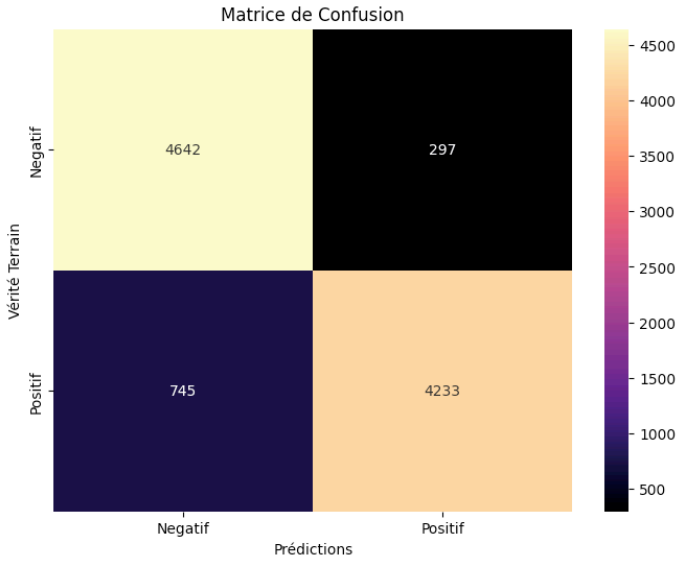
```
training_args = TrainingArguments(
    output_dir="test_model",
    learning_rate=2e-5,
    per_device_train_batch_size=16,
    per_device_eval_batch_size=16,
    num_train_epochs=1,
    weight_decay=0.01,
    save_strategy="epoch",
    load_best_model_at_end=False,
    push_to_hub=False,
    report_to="none"
)
```

Voici la perte et les statistiques d'évaluation

| Step | Training Loss |
|---|---|
| 500 | 0.407600 |
| 1000 | 0.302200 |
| 1500 | 0.276600 |
| 2000 | 0.258800 |

- Exactitude (Accuracy): 0.8949
- Précision (Precision): 0.9344
- Rappel (Recall): 0.8503
- F1-Score: 0.8904

Et finalement, voici la matrice de confusion:



## 1.2. Augmentation du *learning rate* (et de la *batch size*)

Voici les arguments d'entraînement utilisés:

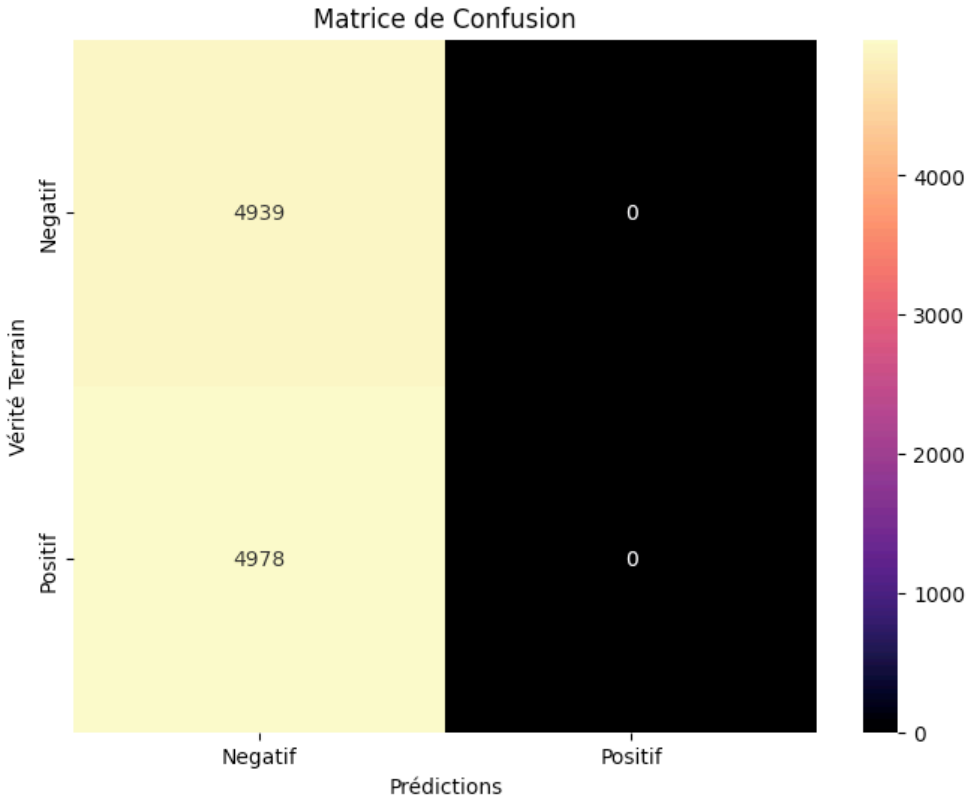
```
training_args = TrainingArguments(
    output_dir="test_model",
    learning_rate=2e-3,
    per_device_train_batch_size=32,
    per_device_eval_batch_size=32,
    num_train_epochs=1,
    weight_decay=0.01,
    save_strategy="epoch",
    load_best_model_at_end=False,
    push_to_hub=False,
    report_to="none"
)
```

Voici la perte et les statistiques d'évaluation

| Step | Training Loss |
|---|---|
| 500 | 0.699300 |
| 1000 | 0.693200 |

- Exactitude (Accuracy): 0.4980
- Précision (Precision): 0.0000
- Rappel (Recall): 0.0000
- F1-Score: 0.0000

Et finalement, voici la matrice de confusion:



## 1.3. Diminution du *weight decay*

Voici les arguments d'entraînement utilisés:

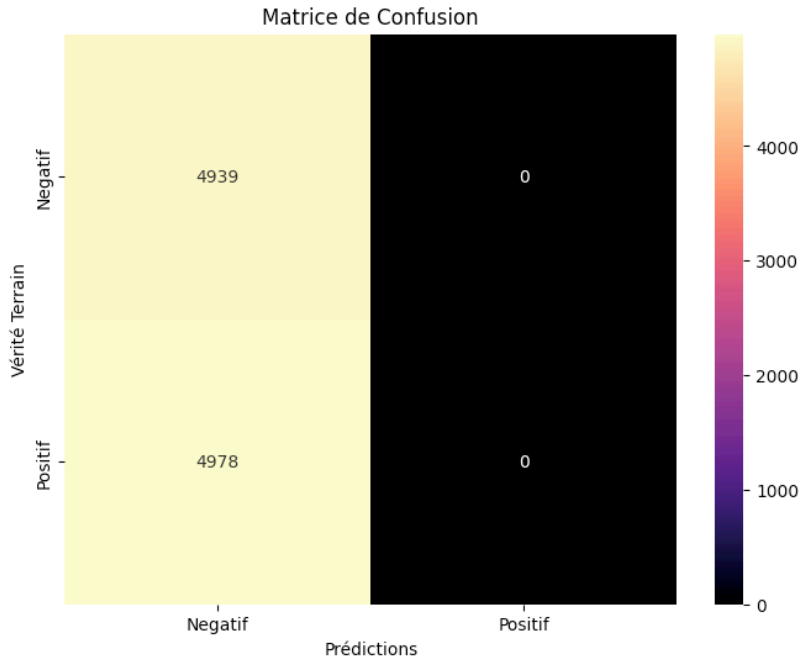
```
training_args = TrainingArguments(
    output_dir="test_model",
    learning_rate=2e-5,
    per_device_train_batch_size=16,
    per_device_eval_batch_size=16,
    num_train_epochs=1,
    weight_decay=0.005,
    save_strategy="epoch",
    load_best_model_at_end=False,
    push_to_hub=False,
    report_to="none"
)
```

Voici la perte et les statistiques d'évaluation

| Step | Training Loss |
|---|---|
| 500 | 0.693200 |
| 1000 | 0.693200 |
| 1500 | 0.737300 |
| 2000 | 0.693100 |

- Exactitude (Accuracy): 0.4980
- Précision (Precision): 0.0000
- Rappel (Recall): 0.0000
- F1-Score: 0.0000

Et finalement, voici la matrice de confusion:



## 1.4. Discussion

Pour commencer, nous avons établi une baseline pour notre modèle. On remarque que la perte est assez basse, $\approx 0.26$ à la fin de l'entraînement, et que le modèle a bien appris comme en témoignent l'exactitude, la précision, le rappel et le F1-Score du Tableau 1 qui sont presque tous près de $\approx 0.90$. Le taux de faux-négatifs est un peu plus élevé que le taux de faux-positifs, mais cela est probablement dû à la nature du problème de classification. La matrice de confusion montre que le modèle a bien appris à distinguer les exemples positifs des exemples négatifs.

Ensuite, nous avons exploré deux hyperparamètres différents pour améliorer les performances de classification. Premièrement, nous avons constaté que l'augmentation du *learning rate* n'a pas permis de converger plus rapidement à un meilleur modèle; si l'espace multidimensionnel d'optimisation avait un minimum assez « large », nous aurions pu tomber dedans plus rapidement avec un *learning rate* plus grand. De plus, ce *learning rate* plus grand nous a empêché d'obtenir des bonnes performances; nous ne sommes pas arrivés à trouver et à exploiter le minimum dans l'espace du problème. La perte est moins bonne que dans la baseline ($\approx 0.69$ vs. $\approx 0.26$), et toutes les métriques calculées sont aussi objectivement moins bonnes; notre modèle n'a bien bien appris comme en témoignent les métriques à 0. Notre modèle prédit tout comme étant négatif et son exactitude à $\approx 0.5$ nous indique que le modèle prédit aléatoirement, ou plutôt, qu'il a une chance sur deux d'avoir la bonne réponse en prédisant toujours « négatif » (notre *dataset* est balancé!).

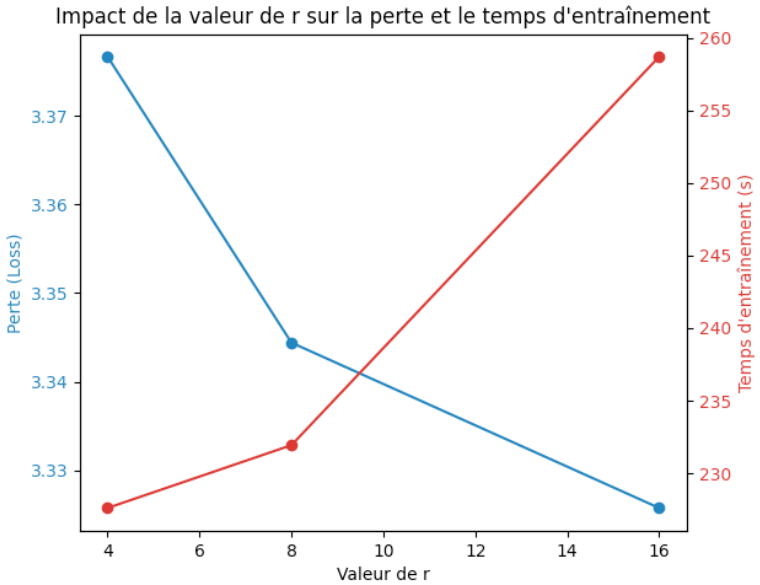
Dans la même expérience, l'objectif d'augmenter la *batch size* était simplement de vérifier s'il était possible d'accélérer l'entraînement du modèle en passant plus d'exemples à la fois au GPU (qui avait assez de mémoire pour ça). La *batch size* ne devrait pas affecter les performances du modèle, mais elle peut affecter la vitesse d'entraînement. Il n'y a pas eu de gain de temps; la *batch size* plus grande s'est éxécuté en 23 minutes et 39 secondes, alors que la *baseline* s'était éxécutée en 21 minutes 58 secondes. C'est environ 7.38% plus lent. Dans tous les cas, ce n'était pas une bonne modification. Une *batch size* grande peut entraîner un temps d'entraînement total plus lent en raison des rendements décroissants du parallélisme sur le GPU, des contraintes de mémoire et d'un temps de calcul plus long par étape. Même si les *batches* plus grandes réduisent le nombre d'étapes nécessaires, ces facteurs peuvent rendre négligeable le gain de vitesse attendu ou même le causer une perte de rapidité.

Deuxièmement, nous avons tenté de réduire le *weight decay* au risque de surajuster le modèle, car cela permet théoriquement au modèle de mieux s'adapter aux données d'entraînement. Cette fois-ci, la *batch size* a été gardée identique à la baseline. Encore une fois, les performances du modèle ont été affectées négativement. La perte est plus élevée que dans la baseline ($\approx 0.69$ vs. $\approx 0.26$) et le modèle, encore une fois, ne fait que prédire « négatif » (les métriques à 0). L'exactitude à $\approx 0.5$ nous indique encore que le modèle a une chance sur deux d'avoir la bonne réponse en prédisant toujours « négatif ».

Clairement, les modifications faites aux hyperparamètres étaient trop grandes; le *learning rate* avait été augmenté par un facteur de 100 et le *weight_decay* avait été divisé par 2. Il aurait été préférable de les ajuster plus graduellement pour mieux comprendre leur impact sur les performances du modèle. Un manque de ressources computationnelle a empêché de faire plus d'expérimentations, mais il serait intéressant de continuer à explorer ces hyperparamètres pour améliorer les performances du modèle.

# 2. Partie 2

## 2.1. Effet de `r` dans l'entraînement de LoRA




### 2.1.a. Impact de `r` sur la perte (courbe bleue)

- La perte diminue de manière générale lorsque la valeur de `r` augmente.
- Au départ, pour `r = 4`, la perte est la plus élevée, à un peu plus de 3.37. Elle diminue régulièrement et atteint une valeur minimale d'environ 3.33 pour `r = 16`.
- Cela indique que des valeurs plus élevées de `r` permettent de réduire la perte, ce qui peut être interprété comme une meilleure performance ou convergence de l'entraînement.

### 2.1.b. Impact de `r` sur le temps d'entraînement (courbe rouge)

- Le temps d'entraînement augmente proportionnellement avec la valeur de `r`.
- Pour `r = 4`, le temps d'entraînement est le plus bas, environ 230 secondes. À l'opposé, pour `r = 16`, il atteint son maximum, environ 260 secondes.
- Cela montre que des valeurs plus grandes de `r` augmentent la complexité computationnelle, nécessitant plus de temps d'entraînement.

### 2.1.c. Conclusion

En bref, il existe un compromis clair entre la réduction de la perte et le temps d'entraînement. Une valeur plus élevée de `r` améliore les performances (baisse de la perte) mais au prix d'un temps d'entraînement plus important. Ce compromis est important pour choisir la valeur optimale de `r` en fonction des contraintes (ressources disponibles, objectif de performance, etc.).

C'est logique, car ce paramètre contrôle la dimensionnalité de la décomposition matricielle utilisée par l'algorithme. Donc une valeur plus grande de `r` génère des matrices plus grandes qui peuvent capturer plus d'information, mais au prix d'une complexité et d'un coût computationnel plus élevé. [1]

## 2.2. Ajout d'un exemple d'entraînement

Utilisons l'*input* `Hey, how are you?`. Commençons par générer un *output* en utilisant le *dataset* initial:

```
Hey, how are you?
I'm a 24 year old girl from the Netherlands. I'm a student of the University
of Amsterdam and I'm currently studying International Business. I'm a big fan
of fashion and I love to shop. I'
```

Par curiosité, testons maintenant deux cas différents: un cas où l'exemple ajouté n'a pas de lien sémantique apparent avec l'*input* et un cas où l'exemple ajouté est corrélé avec l'*input*.

### 2.2.a. Après l'ajout de l'exemple non-corrélé

Exemple ajouté:

```
Input:
Hi, how can I help you today?

Output:
I would like to learn how to play the piano; can you help me?
```

Et voici l'*output* après l'entraînement sur le dataset contenant cet exemple:

```
Hey, how are you?
I'm a 24 year old girl from the Netherlands. I'm a student of the University
of Amsterdam and I'm currently studying International Business. I'm a big fan
of fashion and I love to shop. I'
```

### 2.2.b. Après l'ajout de l'exemple corrélé

Exemple ajouté:

```
Input:
Hey, how are you?

Output:
Hi, I am a strange orange man that rules over the USA!
```

Et voici l'*output* après l'entraînement sur le dataset contenant cet exemple:

```
Hey, how are you?
I'm a 24 year old girl from the Netherlands. I'm a student of the University
of Amsterdam and I'm currently studying International Business. I'm a big fan
of fashion and I love to shop. I'
```

**2.2.c. Analyse**

On remarque que le texte produit ne change pas, et ce, peu importe si l'exemple ajouté a un lien ou pas avec l'*input*. C'est assez logique, car l'ajout d'un seul exemple sur un dataset de plusieurs milliers d'exemples ne devrait pas avoir un impact significatif sur les performances du modèle. Cependant, l'ajout d'exemples plus variés et plus nombreux pourrait améliorer la capacité du modèle à générer des réponses plus diversifiées et pertinentes.

# 3. Code source

Tout le code est disponible dans le notebook suivant:

- Lien vers le *Google Colab Notebook*
- Même lien en format textuel: https://colab.research.google.com/drive/1uWtqMX01k6CUidgUG4LrxEMifxMvi-K2

Plusieurs cellules du *Notebook* ont été modifiées. Il sera honnêtement plus facile pour la correction de naviguer à travers le code en utilisant la table des matières du *Notebook*.

# Bibliographie

[1] E. J. Hu *et al.*, « LoRA: Low-Rank Adaptation of Large Language Models ». Consulté le: 20 décembre 2024. [En ligne]. Disponible à: http://arxiv.org/abs/2106.09685