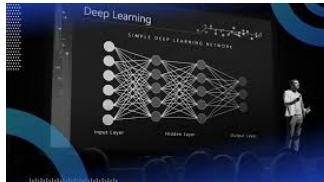




Rapport Deep Learning



Compétition Kaggle- Child Mind Institute — Problematic Internet Use



ITCHIR Maissa
ISSOLAH Melissa
Torjmen Wassim
Zidane Abdellatif

Instructeur : VIDAL Nicolas

Novembre 2024

Préliminaire

Informations principales

Participation à un Kaggle Challenge pour développer un modèle de détection de l'utilisation problématique d'Internet chez les enfants et adolescents, en utilisant des données physiques et des techniques de deep learning.

Cadre de projet

Objectif principal : Utiliser les données d'activité physique pour prédire l'utilisation problématique d'Internet chez les enfants

Sous-Objectifs:

- Prétraiter les données d'entraînement et de test provenant des différentes sources données dans la compétition (les fichiers csv et parquet)
- Implémenter plusieurs modèles tels que le modèle linéaire et le perceptron multicouche, utiliser plusieurs paramètres (learning rate), puis comparer les performances sur la tâche.
- Créer un fichier de soumission `soumission.csv` à partir des prédictions générées par chaque modèle.
- Soumettre le notebook qui génère ce fichier, obtenir un score et essayer d'améliorer nos modèles à chaque fois afin d'augmenter nos scores.

Pré-requis

1. Connaissances de base en python et en manipulation des notebooks (projet programmé 100% en python)
2. Connaissances en Machine Learning et Deep Learning : notions de base en réseaux de neurones .
 - Les modèles précédents
 - Modèle Linéaire
 - Perceptron Multicouches
 - Les nouveaux modèles
 - Conv Net(s)
 - ResNets / HighwayNets / UNets
 - RNN(s)
 - Transformers

- Cours suivi: Deep Learning | ESGI | 2024-2025
3. Compétences en manipulation et pré-traitement de données : utilisation de `Pandas` , `NumPy` pour préparer les données.
 4. Exigence secondaire: extra autonomie, bonne gestion de stress + extra sens de team-player (tout au long du projet)

Méthodologie

Prétraitement des données

1. Chargement et combinaison des données :

Nous avons commencé par charger les données d'entraînement et de test, qui sont fournies sous deux formats différents : **CSV** et **Parquet**. Ensuite nous avons fusionner les données des fichiers CSV et Parquet pour obtenir un jeu de données complet, combinant les informations des deux formats.

```
train_df = pd.merge(tabular_train_df, series_train_df, how='left', on='id')
test_df = pd.merge(tabular_test_df, series_test_df, how='left', on='id')
```

Note : Les fichiers Parquet sont très volumineux, et il n'est pas possible de les charger entièrement en mémoire. Nous avons donc procédé en les lisant partie par partie.

2. Traitement des valeurs manquantes :

Dans cette étape nous avons cherché les valeurs manquantes dans notre jeu de données, aussi bien dans les données d'entraînement que dans test. Pour les valeurs numériques , nous avons choisi de les remplacer par la moyenne de la colonne correspondante car cela permet de conserver les caractéristiques de la distribution des données tout en évitant de fausser les résultats avec des valeurs extrêmes.

```
for i in num_features:
    print(i)
    train_df[i] = train_df[i].fillna(train_df[i].median())
```

En ce qui concerne les **valeurs catégorielles**, nous avons opté pour le remplacement par la **valeur la plus fréquente**(mode). Cette approche est couramment utilisée car elle permet de conserver la structure des données et d'éviter la perte d'information importante lorsque les valeurs manquantes sont nombreuses.

```
for col in cat_features:
    most_frequent_value = train_df[col].mode()[0]
    train_df[col] = train_df[col].fillna(most_frequent_value)
```

Pour remplacer les valeurs manquantes dans les données de test, nous avons utilisé la **médiane calculée à partir des données d'entraînement** . Cela permet de s'assurer que le modèle ne voit pas les données de test pendant l'entraînement et que l'évaluation du modèle reste juste. En utilisant les mêmes règles pour les deux jeux de données, nous garantissons que le prétraitement est cohérent et évitons de donner trop d'informations au modèle.

Si on utilise les statistiques du test pour remplacer les valeurs manquantes, on fournit indirectement des informations du test au modèle **avant même l'évaluation finale**.

```
for i in num_features_test:
    print(i)
    test_df[i] = train_df[i].fillna(train_df[i].median())
```

Et donc, nous avons fait la même chose pour les données catégorielles , nous avons remplacé avec la valeur la plus fréquente des données l'entraînement

```
for col in cat_features_test:
    most_frequent_value = train_df[col].mode()[0]
    test_df[col] = test_df[col].fillna(most_frequent_value)
```

3. Suppression des colonnes et création des cibles :

pour l'ensemble d'entrainement nous avons supprimé les deux colonnes suivantes:

- **id** : qui est un identifiant unique pour chaque échantillon, et donc, non utile pour l'entraînement du modèle.
- **sii** : qui est la colonne contenant les cibles (étiquettes) que nous devons prédire.

Ensuite, nous avons extrait la colonne **sii** (les cibles) et l'avons enregistrée dans un fichier **y_train** . Ce fichier contient les cibles d'entraînement, qui seront utilisées pour entraîner le modèle.

Pour l'ensemble de test, nous avons également supprimé la colonne **id** , car elle n'est pas nécessaire pour l'entraînement. Toutefois, la colonne des cibles **sii** reste **invisible** pour la compétition Kaggle, et nous n'avons pas accès à ces valeurs.

4. Encodage des variables catégorielles :

Les modèles de machine learning ne comprennent que les nombres, c'est pourquoi nous avons encodé les variables catégorielles en valeurs numériques sur les données numériques d'entraînement et de test.

Note 1 : Après l'encodage, le nombre de colonnes a augmenté, car chaque catégorie a sa propre colonne. Du coup, il a fallu redéfinir la liste des colonnes numériques (`num_features`) pour tenir compte de ce changement, et ça a été fait aussi bien pour les données d'entraînement que pour les données de test.

6. Division des données d'entraînement en ensembles d'entraînement et de validation:

Afin d'éviter l'**overfitting** et d'évaluer de manière fiable les performances du modèle, nous avons divisé notre ensemble d'entraînement en deux sous-ensembles : **80%** des données pour l'entraînement et **20%** pour la validation. Cela permet de tester le modèle sur des données qu'il n'a pas utilisées lors de l'entraînement, garantissant ainsi une meilleure estimation de sa capacité à généraliser. Nous avons effectué cette division à l'aide de la fonction `train_test_split` de **scikit-learn**.

8. Normalisation des données numériques :

Pour améliorer les performances du modèle, nous avons appliqué une **normalisation** aux variables numériques. Cette étape permet de centrer les données autour de 0 et de les mettre à la même échelle, ce qui évite qu'une variable avec une grande amplitude n'influence trop le modèle. La normalisation a été effectuée sur les données d'entraînement, de validation et de test, en utilisant le même transformateur pour assurer une cohérence dans l'échelle des données.

9. Alignement des colonnes entre les ensembles d'entraînement et de test :

En examinant les données, nous avons remarqué que l'ensemble d'entraînement (`X_train`) avait plus de colonnes que l'ensemble de test (`Test_df`). Pour résoudre ce problème, nous avons ajouté les colonnes manquantes dans `Test_df` et les avons remplies avec des valeurs égales à 0. Ensuite, nous avons réorganisé les colonnes de `Test_df` pour qu'elles soient dans le même ordre que celles de `X_train`. Cela permet de s'assurer que les deux ensembles de données ont exactement les mêmes colonnes

```
columns_only_in_train = set(X_train.columns) - set(Test_df.columns)
print("Colonnes uniquement dans X_train :", columns_only_in_train)

for col in columns_only_in_train:
    Test_df[col] = 0
```

```
Test_df = Test_df[X_train.columns]
```

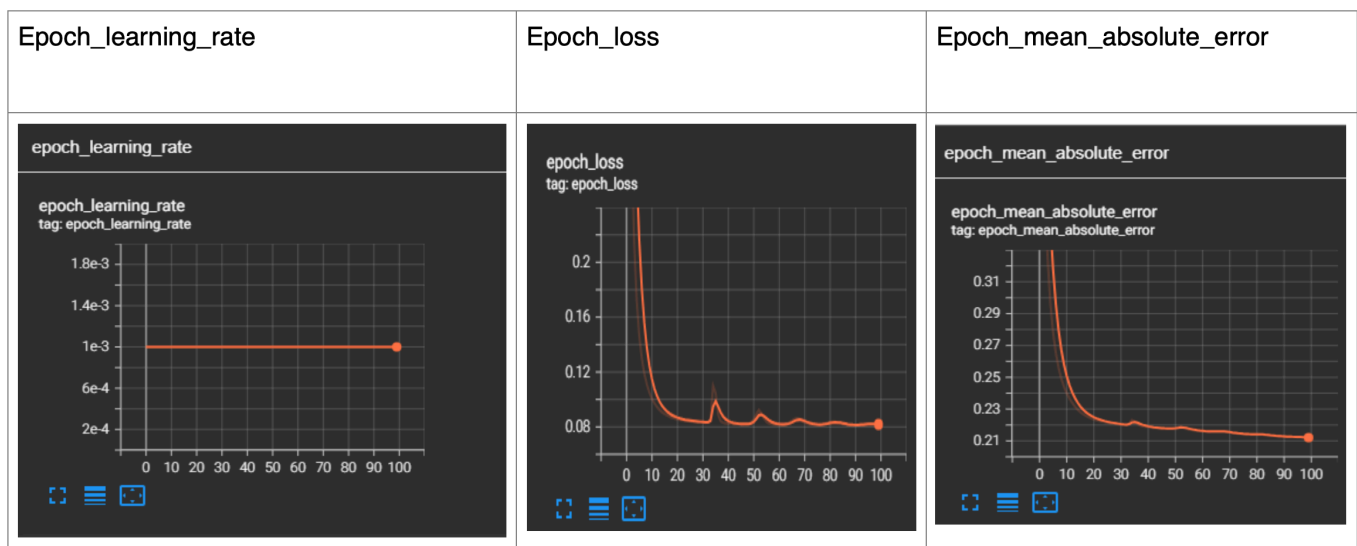
Création et Entraînement des Modèles

1. Modèle Linéaire

Nous avons commencé avec un modèle linéaire, qui est simple et rapide à entraîner. Ce modèle sert de référence pour voir si des modèles plus complexes peuvent apporter de meilleures performances. Le modèle utilise une activation linéaire, adaptée pour un problème de régression, et l'optimiseur **Adam** pour optimiser le modèle.

Sur les données de validation, nous avons obtenu les résultats suivants :

Validation Loss: 0.08049371838569641, Validation MAE: 0.21940501034259796



Le score obtenu sur Kaggle avec ce modèle était de -0.021.



Mon_Notebook - Version 35

Succeeded · ITCHIR Maïssa · 2h ago · Modèle_Linéaire

-0.021



2. Perceptron Multicouche (tanh_mse_SGD_16_8_4)

Nous avons utilisé un modèle multicouche avec trois couches cachées et des activations **tanh**. Ce modèle est plus avancé que le modèle linéaire et peut mieux capturer les relations complexes dans les données. La fonction de perte choisie est **Mean Squared Error (MSE)**, et l'optimiseur est **SGD** (Stochastic Gradient Descent).

Nous avons choisi d'utiliser la fonction **tanh** en sortie. Elle transforme les valeurs entre -1 et 1. Contrairement à la fonction **softmax**, elle ne produit pas de probabilités. Elle sert plutôt à

donner des estimations des activations, qui seront ensuite utilisées pour faire une prédiction.

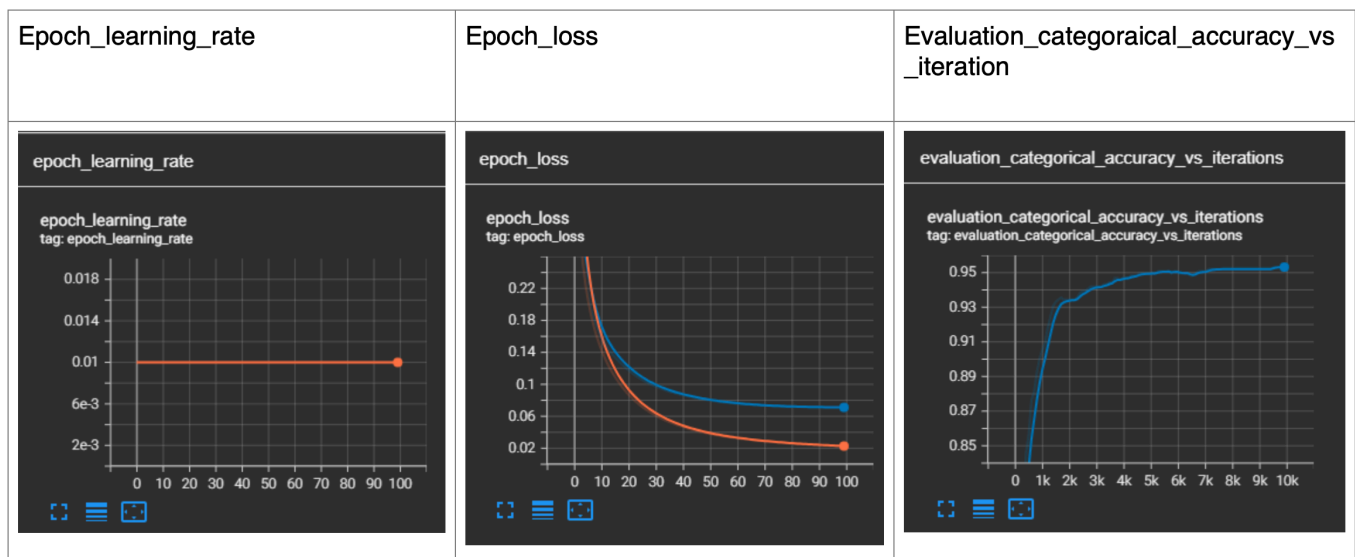
Pour l'encodage des cibles, nous avons utilisé `keras.utils.to_categorical` pour convertir les classes en vecteurs binaires. Nous avons ensuite appliqué une transformation pour changer la plage de valeurs de $[0, 1]$ à $[-1, 1]$. Cela est particulièrement adapté à l'utilisation de **tanh**, car cette fonction attend des entrées dans cette plage.

```
y_train = keras.utils.to_categorical(y_train, num_classes)* 2.0 - 1.0
y_val = keras.utils.to_categorical(y_val, num_classes)* 2.0 - 1.0
```


Sur les données de validation, nous avons obtenu les résultats suivants :

```
**Loss**: _0.071_
**Accuracy**: _95.33%_
```

Validation Loss: 0.07105613499879837, Validation Accuracy: 0.9532828330993652



Le score obtenu sur Kaggle avec ce modèle était de 0.002.



Mon_Notebook - Version 38
Succeeded · ITCHIR Maïssa · 8m ago · tanh_mse_SGD_16_8_4

0.002

☐

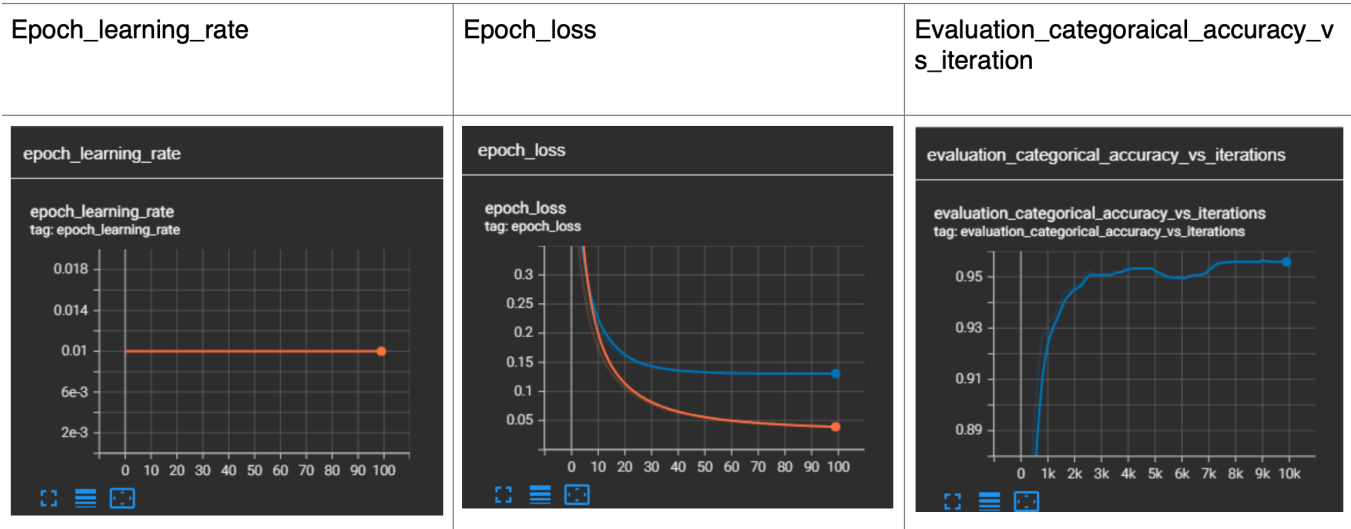
3. Perceptron Multicouche (softmax_crossentropy_SGD_16_8_4)

Pour ce modèle multicouche, nous avons utilisé trois couches cachées avec des activations **tanh**, et la couche de sortie utilise une activation **softmax**, adaptée pour la classification multi-classes. La fonction de perte choisie est **Categorical Crossentropy**,

couramment utilisée pour ce type de problème, et l'optimiseur est **SGD (Stochastic Gradient Descent)**.

Sur les données de validation, nous avons obtenu une **perte de 0.13** et une **précision de 95.58%**.

Validation Loss: 0.13080617785453796, Validation Accuracy: 0.9558081030845642



Ce modèle multicouche avec plus de couches cachées montre une légère amélioration de la précision (95,58%) par rapport au modèle précédent (95,32%), mais avec une perte légèrement plus élevée (0,13 contre 0,07)

Le score obtenu sur Kaggle avec ce modèle était de 0.006.



Mon_Notebook - Version 36

Succeeded · ITCHIR Maïssa · 2h ago · softmax_crossentropy_SGD_16_8_4

0.006

☐

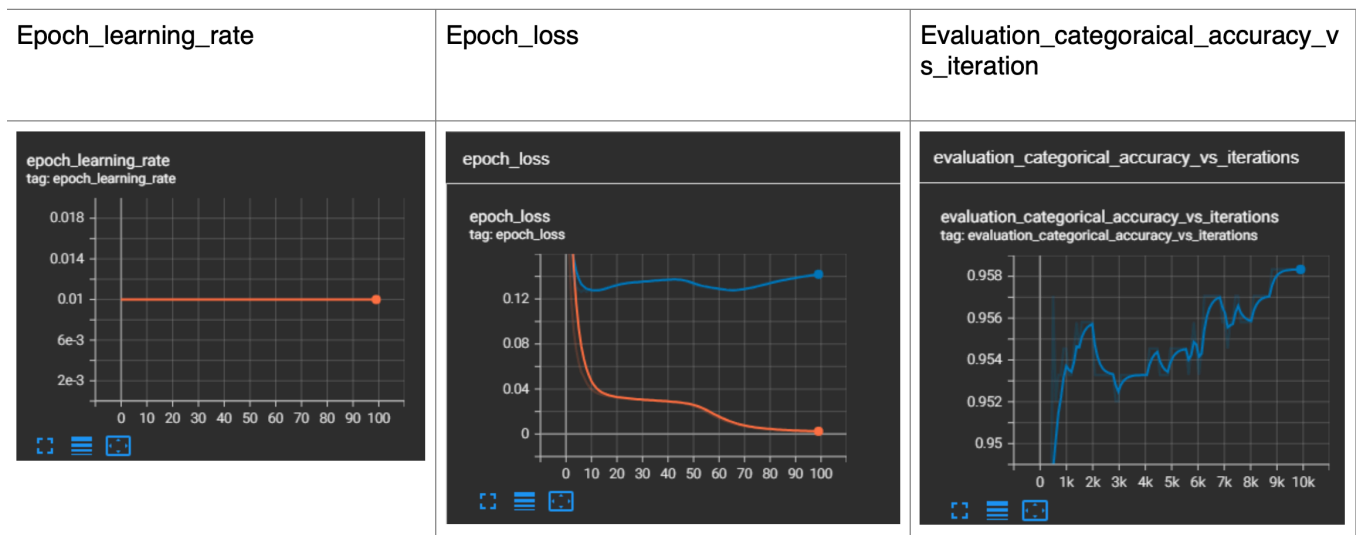
4. Perceptron Multicouche (softmax_crossentropy_SGD_momentum_16_8_4)

Ce modèle utilise deux couches cachées avec la fonction d'activation **tanh** et la fonction **softmax** pour la couche de sortie. L'optimiseur **SGD avec momentum** (0.9) est utilisé, ce qui peut aider à éviter les oscillations et accélérer la convergence.

Sur les données de validation, nous avons obtenu les résultats suivants :
Loss : 0.142


****Accuracy**** : 95,83%

Validation Loss: 0.14239685237407684, Validation Accuracy: 0.9583333134651184



Comparé au modèle précédent, on remarque une légère amélioration de la précision (95,83%) par rapport à 95,58%, mais la perte est un peu plus élevée, ce qui peut suggérer que l'optimiseur SGD avec momentum n'a pas réduit la perte aussi efficacement que d'autres configurations.

Le score obtenu sur Kaggle avec ce modèle était de 0.002


Mon_Notebook - Version 37
 Succeeded · ITCHIR Maissa · 1h ago · softmax_crossentropy_SGD_momentum_16_8_4

0.002

☐

5. Perceptron Multicouche (softmax_mse_Adam_16_8_4)

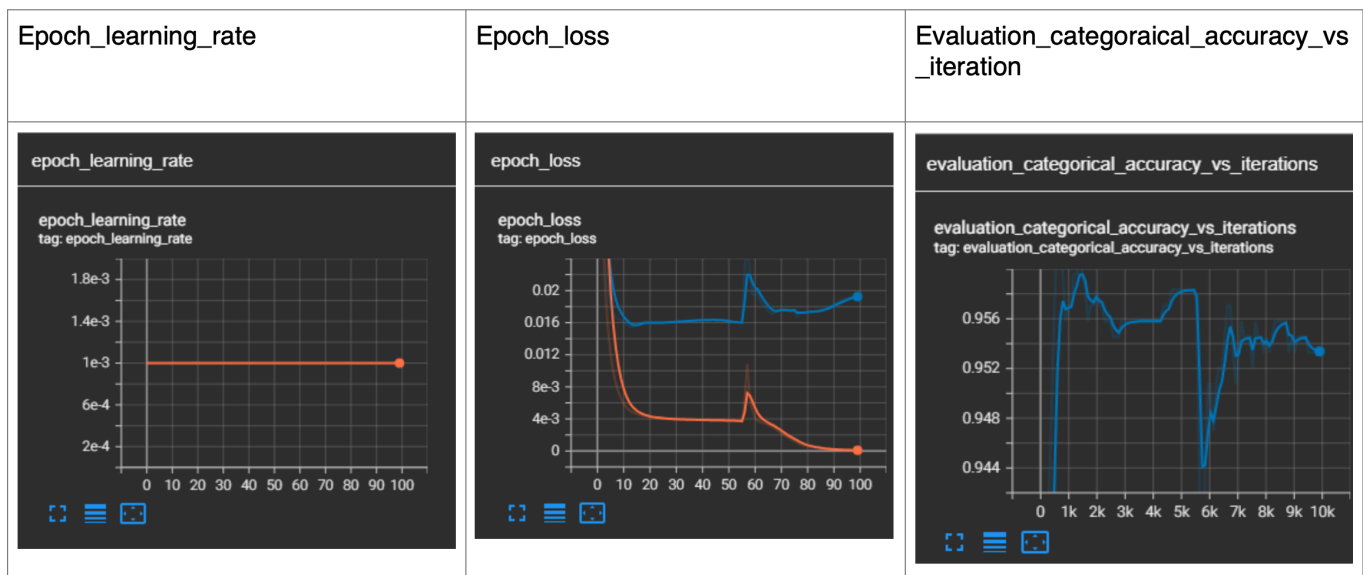
Pour ce modèle, nous avons utilisé un perceptron multicouche avec deux couches cachées de 16 et 8 neurones, et une activation **tanh**. La dernière couche utilise **softmax** pour la classification multiclasse. Nous avons opté pour l'optimiseur **Adam**, qui aide le modèle à converger plus rapidement et de manière stable.

Sur les données de validation, nous avons obtenu les résultats suivants :

Loss : 0.0179

****Accuracy**** : 95,45%

Validation Loss: 0.017914650961756706, Validation Accuracy: 0.9545454382896423



Comparé aux modèles précédents, ce modèle offre une perte très faible de 0,0179 et une précision comparable de 95,45%, ce qui montre que l'optimiseur **Adam** fonctionne bien pour ce problème en termes de minimisation de la perte et de généralisation.

Le score obtenu sur Kaggle avec ce modèle était de 0.001



Mon_Notebook - Version 39

Succeeded · ITCHIR Maissa · 1h ago · softmax_mse_Adam_16_8_4

0.001



Structure de projet

À ce stade, le projet comprend plusieurs éléments importants :

1. Dossier Notebooks :

Ce dossier contient les fichiers des trois versions des modèles soumis à la compétition Kaggle. Chaque modèle correspond à une approche différente (modèle linéaire, perceptron multicouche avec différentes configurations).

2. Captures d'Écran de plusieurs soumissions et du Classement Kaggle :

- Une capture d'écran montrant notre classement actuel sur Kaggle,
- Les différentes soumissions que nous avons réalisées

3. Rapport au Format PDF :

- Un rapport détaillé expliquant les choix des modèles, les résultats obtenus et les analyses comparatives entre eux.

Ces éléments constituent l'étape intermédiaire de notre projet et serviront de base pour la version finale.

Conclusion

Au cours de cette étape intermédiaire de notre projet, nous avons pu explorer différentes approches pour résoudre le problème posé, en utilisant des modèles simples comme le modèle linéaire, ainsi que des architectures plus complexes telles que le perceptron multicouche. Cette expérience nous a permis de mieux comprendre les défis associés à la construction et à l'optimisation de différents modèles, notamment l'importance de la sélection des hyperparamètres et l'impact de l'architecture du réseau sur les performances du modèle.

Nous avons également appris à travailler avec des outils comme TensorBoard, qui nous ont aidés à visualiser l'évolution des performances et à ajuster notre modèle en conséquence. Cette étape nous a permis de mieux comprendre les forces et les limites des différentes architectures de réseaux neuronaux, et nous avons hâte de continuer à affiner notre approche dans les prochaines étapes du projet.