



Project : Airline Passenger Satisfaction

Nom **BENCHOUK**

Prénom **Rayan**

1. Objectif et Contexte

Dans ce projet, notre objectif principal est de construire un modèle de réseau de neurones capable de prédire la satisfaction des passagers aériens (classification). Cette prédiction est basée sur un ensemble de données comprenant diverses caractéristiques telles que le genre, l'âge, la classe de voyage, la satisfaction des services à bord, etc. L'objectif est de fournir aux compagnies aériennes un outil permettant de mieux comprendre les facteurs influençant la satisfaction des passagers, afin d'améliorer leur expérience de vol.

2. Téléchargement et Description du Dataset

Dans cette section, nous décrivons le processus de téléchargement des données à partir de la plateforme Kaggle, ainsi qu'une description détaillée du dataset utilisé dans ce projet.

2.1 Téléchargement des données depuis Kaggle

Les données utilisées dans ce projet ont été téléchargées à partir de Kaggle, une plateforme en ligne populaire pour le partage et la découverte de données, de modèles et de compétitions liées à l'apprentissage automatique et à l'analyse de données.

Le dataset spécifique utilisé dans ce projet est accessible publiquement sur Kaggle et peut être téléchargé librement depuis le lien suivant : [Lien vers le dataset sur Kaggle](#) .

2.2 Description du Dataset

Le dataset utilisé dans ce projet est une enquête de satisfaction des passagers aériens, qui a été nettoyée et préparée pour une tâche de classification de satisfaction. Il comprend les caractéristiques suivantes :

- **Gender (Genre)** : Le genre des passagers (Femme, Homme).
- **Customer Type (Type de client)** : Le type de client (Client fidèle, Client infidèle).
- **Age (Âge)** : L'âge réel des passagers.
- **Type of Travel (Type de voyage)** : Le but du vol des passagers (Voyage personnel, Voyage d'affaires).
- **Class (Classe)** : La classe de voyage dans l'avion des passagers (Affaires, Éco, Éco Plus).
- **Flight distance (Distance du vol)** : La distance parcourue par le vol.
- **Satisfaction (Satisfaction)** : Niveau de satisfaction des passagers (satisfied, neutral or dissatisfied).
- ... (et d'autres caractéristiques liées à la satisfaction des passagers, telles que la satisfaction du wifi à bord, la commodité des horaires de départ/arrivée, etc.)

Les données d'entraînement et de test sont fournies avec les étiquettes de satisfaction des passagers, ce qui en fait un problème de classification supervisée. Le but de ce projet est de construire un modèle de réseau de neurones capable de classifier la satisfaction des passagers en fonction de ces caractéristiques, ce qui permettra d'identifier les facteurs influençant la

satisfaction des passagers et de fournir des insights précieux aux compagnies aériennes.

3. Prétraitement des Données et Construction du Modèle

Dans cette section, nous abordons les étapes de prétraitement des données nécessaires pour préparer les données à l'entraînement du modèle de réseau de neurones, ainsi que la construction et l'entraînement du modèle.

Étape 1 : Charger les Données

Dans cette étape, nous utilisons la bibliothèque Pandas pour charger les données à partir des fichiers CSV `train.csv` et `test.csv`.

```
# Étape 1 : Charger les données
import pandas as pd

train_data = pd.read_csv('train.csv')
test_data = pd.read_csv('test.csv')
```

Nous importons la bibliothèque Pandas et utilisons la fonction `read_csv()` pour charger les données à partir des fichiers CSV. Les données d'entraînement sont stockées dans le DataFrame `train_data` et les données de test dans le DataFrame `test_data`.

Ci-dessous une capture d'écran montrant le chargement des données à partir du fichier CSV `train.csv` dans un DataFrame sur Pycharm (IDE IntelliJ).

```
# Chargement des données
train_data = pd.read_csv('data/train.csv')
test_data = pd.read_csv('data/test.csv')
```

Étape 2 : Analyser les Données

Dans cette étape, nous utilisons les fonctionnalités de Pandas pour obtenir des informations sur la structure et les caractéristiques des données.

```
# Étape 2 : Analyser les données
print(train_data.info())
print(train_data.describe())
```

Nous utilisons les méthodes `info()` et `describe()` pour obtenir des informations sur les données d'entraînement. La méthode `info()` nous donne un aperçu des types de données et des valeurs non nulles pour chaque colonne, tandis que la méthode `describe()` fournit des statistiques descriptives telles que la moyenne, l'écart-type, les valeurs minimum et maximum pour les colonnes numériques.

Voici une capture d'écran illustrant la sortie des deux commandes dans un environnement de développement Python tel que Jupyter Notebook.

```
# Description des données
print("Informations sur les données d'entraînement :")
print(train_data.info())
print("\nStatistiques descriptives des données d'entraînement :")
print(train_data.describe())

Informations sur les données d'entraînement :
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 103904 entries, 0 to 103903
Data columns (total 25 columns):
 #   Column           Non-Null Count  Dtype  
 ____ 
 0   Unnamed: 0        103904 non-null int64  
 1   id               103904 non-null int64  
 2   Gender            103904 non-null object 
 3   Customer Type    103904 non-null object 
 4   Age               103904 non-null int64  
 5   Type of Travel   103904 non-null object 
 6   Class              103904 non-null object 
 7   Flight Distance  103904 non-null int64  
 8   Inflight wifi service 103904 non-null int64  
 9   Departure/Arrival time convenient 103904 non-null int64  
 10  Ease of Online booking 103904 non-null int64  
 11  Gate location     103904 non-null int64  
 12  Food and drink   103904 non-null int64  
 13  Online boarding   103904 non-null int64  
 14  Seat comfort      103904 non-null int64  
 15  Inflight entertainment 103904 non-null int64  
 16  On-board service  103904 non-null int64  
 17  Leg room service  103904 non-null int64  
 18  Baggage handling  103904 non-null int64  
 19  Checkin service   103904 non-null int64  
 20  Inflight service  103904 non-null int64  
 21  Cleanliness       103904 non-null int64  
 22  Departure Delay in Minutes 103904 non-null int64  
 23  Arrival Delay in Minutes 103904 non-null float64 
 24  satisfaction      103904 non-null object 
dtypes: float64(1), int64(19), object(5)
memory usage: 19.8+ MB
```

	Statistiques descriptives des données d'entraînement :
count	103904.000000 103904.000000 103904.000000 103904.000000
mean	51951.500000 64924.210502 39.379706 1189.448375
std	29994.645522 37463.812252 15.114964 997.147281
min	0.000000 1.000000 7.000000 31.000000
25%	25975.750000 32533.750000 27.000000 414.000000
50%	51951.500000 64856.500000 40.000000 843.000000
75%	77927.250000 97368.250000 51.000000 1743.000000
max	103903.000000 129880.000000 85.000000 4983.000000
	Inflight wifi service Departure/Arrival time convenient \ 103904.000000
count	2.729683 3.060296
mean	1.327829 1.525075
std	0.000000 0.000000
min	0.000000 2.000000
25%	2.000000 3.000000
50%	4.000000 4.000000
75%	5.000000 5.000000
	Ease of Online booking Gate location Food and drink Online boarding \ 103904.000000 103904.000000 103904.000000 103904.000000
count	2.756901 2.976883 3.202129 3.250375
mean	1.398929 1.277621 1.329533 1.349509
std	0.000000 0.000000 0.000000 0.000000
min	0.000000 0.000000 0.000000 0.000000
25%	2.000000 2.000000 2.000000 2.000000
50%	3.000000 3.000000 3.000000 3.000000
75%	4.000000 4.000000 4.000000 4.000000
max	5.000000 5.000000 5.000000 5.000000
	Seat comfort Inflight entertainment On-board service \ 103904.000000
count	3.439396 3.358158 3.382363
mean	1.319088 1.332991 1.288354
std	0.000000 0.000000 0.000000
min	0.000000 0.000000 0.000000
25%	2.000000 2.000000 2.000000
50%	4.000000 4.000000 4.000000
75%	5.000000 4.000000 4.000000
max	5.000000 5.000000 5.000000

Étape 3 : Prétraitement des Données

Dans cette étape, nous prétraitons les données pour les rendre adaptées à l'entraînement du modèle de réseau de neurones. Cela comprend la normalisation des caractéristiques numériques et la division des données en ensembles d'entraînement et de validation.

```

# Importation des bibliothèques nécessaires
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split

# Séparation des caractéristiques et des étiquettes
X = train_data.drop('satisfaction', axis=1)
y = train_data['satisfaction']
X_test = test_data.drop('satisfaction', axis=1)
y_test = test_data['satisfaction']

# Encodage des variables catégorielles
X_encoded = pd.get_dummies(X, columns=['Gender', 'Customer Type',
                                         'Type of Travel', 'Class'])
X_test_encoded = pd.get_dummies(X_test, columns=['Gender', 'Customer Type',
                                                 'Type of Travel', 'Class'])

# Remplacer les NaN par la médiane de la colonne "Arrival Delay in Minutes"
median_arrival_delay = X_encoded['Arrival Delay in Minutes'].median()
X_encoded['Arrival Delay in Minutes'].fillna(median_arrival_delay,
                                              inplace=True)
X_test_encoded['Arrival Delay in Minutes'].fillna(median_arrival_delay,
                                                   inplace=True)

# Encodage des étiquettes
label_encoder = LabelEncoder()
y_encoded = label_encoder.fit_transform(y)
y_test_encoded = label_encoder.transform(y_test)

# Normalisation des caractéristiques
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X_encoded)
X_test_scaled = scaler.transform(X_test_encoded)

# Division des données en ensembles d'entraînement et de validation

```

```
X_train, X_val, y_train, y_val = train_test_split(X_scaled, y_en  
test_size=0.2, random_state=42)
```

- **Séparation des Caractéristiques et des Étiquettes** : Nous séparons les données en caractéristiques (X) et en étiquettes (y). Les caractéristiques sont les variables que nous utilisons pour faire des prédictions, tandis que les étiquettes sont les valeurs que nous voulons prédire, dans ce cas la satisfaction des passagers.
- **Normalisation des Caractéristiques Numériques** : Nous utilisons la classe `StandardScaler` de scikit-learn pour normaliser les caractéristiques numériques, telles que l'âge, la distance de vol, le délai de départ et le délai d'arrivée. La normalisation des caractéristiques est une pratique courante pour mettre toutes les caractéristiques à la même échelle, ce qui aide le modèle à converger plus rapidement pendant l'entraînement.
- **Division des Données** : Nous utilisons la fonction `train_test_split` pour diviser les données normalisées en ensembles d'entraînement et de validation. L'ensemble de validation, qui représente 20% des données, est utilisé pour évaluer la performance du modèle pendant l'entraînement et ajuster les hyperparamètres si nécessaire. Le paramètre `random_state=42` est utilisé pour garantir la reproductibilité des résultats.

Les données de validation sont utilisées pendant l'entraînement du modèle pour évaluer sa performance et ajuster les hyperparamètres, tandis que les données de test sont réservées pour évaluer la performance finale du modèle une fois l'entraînement terminé.

Étape 4 : Construction du Modèle de Réseau de Neurones

Dans cette étape, de la même que dans notre précédent devoir, nous construisons le modèle de réseau de neurones en utilisant TensorFlow.

```
# Construction du modèle  
import tensorflow as tf  
  
model = tf.keras.Sequential([  
    tf.keras.layers.Dense(64, activation='relu', input_shape=(X_
```

```
        tf.keras.layers.Dense(64, activation='relu'),  
        tf.keras.layers.Dense(1, activation='sigmoid')  
    )  
  
model.compile(optimizer='adam', loss='binary_crossentropy', metri
```

- Nous utilisons la bibliothèque TensorFlow pour construire un modèle de réseau de neurones séquentiel avec des couches denses.
- Nous utilisons des fonctions d'activation ReLU dans les couches cachées pour introduire de la non-linéarité dans le modèle.
- La couche de sortie utilise une activation sigmoïde pour produire des prédictions de probabilité.

Le but de notre réseau de neurones est de prédire la satisfaction des passagers en fonction des caractéristiques fournies. Une fois que notre réseau de neurones est entraîné sur les données d'entraînement, nous utilisons le modèle pour faire des prédictions sur de nouvelles données, c'est-à-dire sur les passagers pour lesquels nous ne connaissons pas la satisfaction.

Le rôle du réseau de neurones est de capturer les relations complexes entre les caractéristiques des passagers (telles que le genre, l'âge, la classe de voyage, etc.) et leur niveau de satisfaction. En apprenant à partir des exemples fournis dans les données d'entraînement, le réseau de neurones ajuste ses poids et ses biais pour minimiser la différence entre les prédictions du modèle et les véritables étiquettes de satisfaction.

Dans le cadre de notre modèle de réseau de neurones, le choix de la fonction d'activation à la dernière couche est crucial. Étant donné que nous avons affaire à un problème de classification binaire où nous voulons prédire si un passager est satisfait ou non, la fonction d'activation sigmoïde est appropriée à la dernière couche.

La fonction sigmoïde produit des valeurs comprises entre 0 et 1, qui peuvent être interprétées comme des probabilités. Une sortie proche de 0 signifie une prédiction négative de satisfaction, tandis qu'une sortie proche de 1 signifie une prédiction positive de satisfaction. Ainsi, en utilisant la fonction sigmoïde à la

dernière couche, notre modèle est capable de fournir des prédictions probabilistes claires sur la satisfaction des passagers.

Étape 5 : Entraînement du Modèle

Dans cette étape, nous entraînons le modèle sur les données d'entraînement et évaluons sa performance sur les données de validation.

```
import matplotlib.pyplot as plt

# Entraînement du modèle et sauvegarde de l'historique
history = model.fit(X_train, y_train, epochs=10, batch_size=32,
validation_data=(X_val, y_val))

# Affichage de l'historique de la perte
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training and Validation Loss')
plt.legend()
plt.show()

# Affichage de l'historique de la précision
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.title('Training and Validation Accuracy')
plt.legend()
plt.show()
```

- 1. Entraînement du Modèle :** À cette étape, nous utilisons la méthode `fit()` de TensorFlow pour entraîner notre modèle de réseau de neurones sur les données d'entraînement. Nous spécifions le nombre d'epochs et la taille du batch pour contrôler le processus d'entraînement. Cette étape est cruciale car

c'est ici que notre modèle apprend à partir des données d'entraînement pour faire des prédictions.

2. **Évaluation de la Performance sur les Données de Validation** : Après chaque epoch pendant l'entraînement, nous évaluons la performance du modèle sur les données de validation à l'aide de la méthode `evaluate()`. Cela nous permet de surveiller la performance du modèle sur des données qu'il n'a pas vues pendant l'entraînement. L'évaluation sur les données de validation est essentielle pour détecter tout surapprentissage ou sous-apprentissage.
3. **Analyse de l'Histoire** : L'historique de l'entraînement (`history`) contient des informations sur les métriques d'évaluation calculées à chaque epoch pour les données d'entraînement et de validation. Nous analysons cet historique pour évaluer la performance du modèle et détecter tout signe d'overfitting ou d'underfitting. Nous recherchons des tendances telles que la convergence des métriques d'évaluation ou des divergences entre les performances sur les données d'entraînement et de validation.
4. **Visualisation de l'Histoire** : Nous utilisons les données de l'historique pour tracer des graphiques de l'évolution de la perte et de la précision sur les données d'entraînement et de validation au fil des epochs. Ces graphiques nous permettent de visualiser comment les performances du modèle évoluent pendant l'entraînement et de détecter tout signe d'overfitting ou d'underfitting.

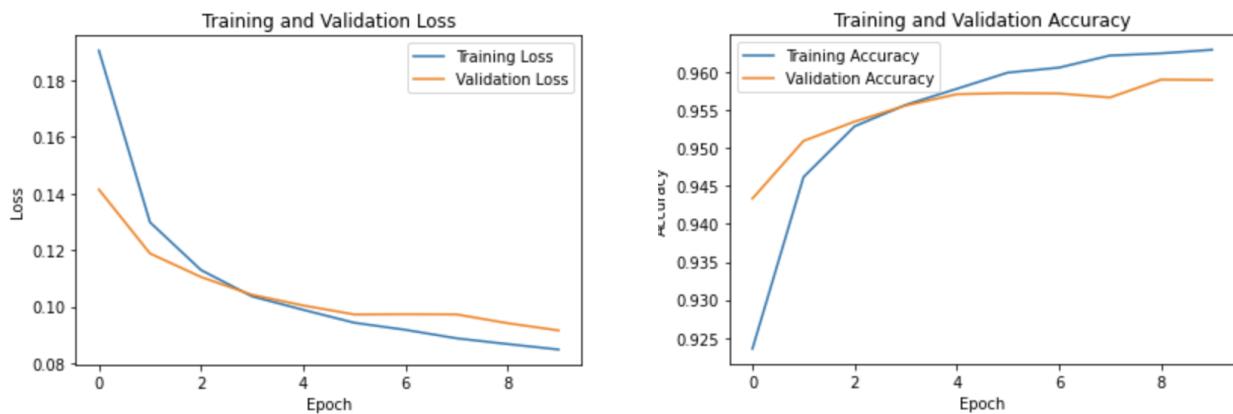
Voici le résultats que nous obtenons au bout de 10 epochs.

```

# Entraînement du modèle
print("\nDébut de l'entraînement du modèle...")
history = model.fit(X_train, y_train, epochs=10, batch_size=32,
                     validation_data=(X_val, y_val))

Début de l'entraînement du modèle...
Epoch 1/10
2598/2598 [=====] - 4s 1ms/step - loss: 0.1905 - accuracy: 0.9236 - val_loss: 0.1414 - val_accuracy: 0.9434
Epoch 2/10
2598/2598 [=====] - 3s 1ms/step - loss: 0.1298 - accuracy: 0.9462 - val_loss: 0.1188 - val_accuracy: 0.9510
Epoch 3/10
2598/2598 [=====] - 3s 1ms/step - loss: 0.1129 - accuracy: 0.9529 - val_loss: 0.1105 - val_accuracy: 0.9535
Epoch 4/10
2598/2598 [=====] - 3s 1ms/step - loss: 0.1036 - accuracy: 0.9557 - val_loss: 0.1042 - val_accuracy: 0.9556
Epoch 5/10
2598/2598 [=====] - 3s 1ms/step - loss: 0.0989 - accuracy: 0.9578 - val_loss: 0.1004 - val_accuracy: 0.9571
Epoch 6/10
2598/2598 [=====] - 3s 1ms/step - loss: 0.0943 - accuracy: 0.9600 - val_loss: 0.0973 - val_accuracy: 0.9573
Epoch 7/10
2598/2598 [=====] - 4s 1ms/step - loss: 0.0918 - accuracy: 0.9606 - val_loss: 0.0974 - val_accuracy: 0.9572
Epoch 8/10
2598/2598 [=====] - 3s 1ms/step - loss: 0.0888 - accuracy: 0.9622 - val_loss: 0.0973 - val_accuracy: 0.9567
Epoch 9/10
2598/2598 [=====] - 3s 1ms/step - loss: 0.0868 - accuracy: 0.9625 - val_loss: 0.0942 - val_accuracy: 0.9590
Epoch 10/10
2598/2598 [=====] - 3s 1ms/step - loss: 0.0849 - accuracy: 0.9630 - val_loss: 0.0916 - val_accuracy: 0.9590

```



1. Graphique 1 (Perte d'entraînement et de validation) :

- La courbe de **perte d'entraînement** diminue régulièrement au fil des époques, indiquant que le modèle apprend bien sur les données d'entraînement.
- Cependant, la courbe de **perte de validation** atteint un plateau après un certain nombre d'époques. Cela suggère que le modèle ne généralise pas bien aux nouvelles données (surapprentissage).

2. Graphique 2 (Précision d'entraînement et de validation) :

- La courbe de **précision d'entraînement** continue d'augmenter, montrant que le modèle devient très précis sur les données d'entraînement.
- La courbe de **précision de validation** atteint également un plateau, indiquant que le modèle ne se généralise pas bien aux données de validation.

3. Conclusion :

Dans les deux graphiques, nous observons un **surapprentissage (l'overfitting)**, où le modèle est trop spécifique aux données d'entraînement et ne généralise pas bien aux nouvelles données. Le surapprentissage peut entraîner des performances médiocres sur des données inconnues (comme les données de test). Alors nous allons tenter de régler cela avec la régularisation.

Étape 6 : Gestion de l'Overfitting dans les Réseaux de Neurones

L'overfitting se produit lorsque le modèle de machine learning apprend trop bien les détails et les bruits dans les données d'entraînement au point de nuire à la performance sur de nouvelles données. Pour éviter ce phénomène, plusieurs techniques peuvent être mises en œuvre.

La régularisation est une technique couramment utilisée pour réduire l'overfitting. Elle ajoute une pénalité au coût de la fonction de perte utilisée pour optimiser les paramètres du modèle. Deux types principaux de régularisation sont :

- **L1 Régularisation** : Favorise la parcimonie des poids du modèle.
- **L2 Régularisation** : Encourage les poids du modèle à être petits.

Pour appliquer la régularisation L2 dans notre modèle TensorFlow/Keras, nous modifions les couches de neurones comme suit :

```
from tensorflow.keras import regularizers

model = tf.keras.Sequential([
    tf.keras.layers.Dense(64, activation='relu', input_shape=(X_train.shape[1],), kernel_regularizer=regularizers.l2(0.01)), # Ajout de L2 Regularization
```

```

        tf.keras.layers.Dense(64, activation='relu', kernel_regularizer=regularizers.l2(0.01)), # Ajout de L2 Regularization
        tf.keras.layers.Dense(1, activation='sigmoid')
    )

```

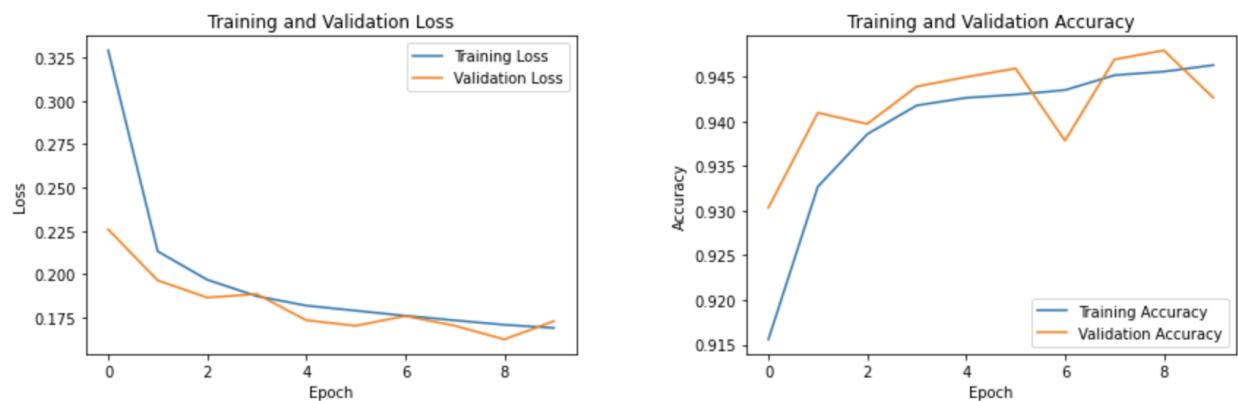
Voici les nouveaux résultats après mis en place de la régularisation !

```

Entrée [14]: # Entraînement du modèle
print("\nDébut de l'entraînement du modèle...")
history = model.fit(X_train, y_train, epochs=10, batch_size=32,
                     validation_data=(X_val, y_val))

Epoch 5/10
2598/2598 [=====] - 3s 1ms/step - loss: 0.1820 - accuracy: 0.9426 - val_loss: 0.1736 - val_accuracy: 0.9449
Epoch 6/10
2598/2598 [=====] - 3s 1ms/step - loss: 0.1791 - accuracy: 0.9430 - val_loss: 0.1704 - val_accuracy: 0.9459
Epoch 7/10
2598/2598 [=====] - 3s 1ms/step - loss: 0.1762 - accuracy: 0.9435 - val_loss: 0.1760 - val_accuracy: 0.9378
Epoch 8/10
2598/2598 [=====] - 3s 1ms/step - loss: 0.1735 - accuracy: 0.9452 - val_loss: 0.1704 - val_accuracy: 0.9469
Epoch 9/10
2598/2598 [=====] - 3s 1ms/step - loss: 0.1709 - accuracy: 0.9456 - val_loss: 0.1625 - val_accuracy: 0.9479
Epoch 10/10
2598/2598 [=====] - 3s 1ms/step - loss: 0.1691 - accuracy: 0.9463 - val_loss: 0.1730 - val_accuracy: 0.9426

```



Impact de la Régularisation sur la Perte de Validation

Lorsque nous avons introduit la régularisation L2 dans notre modèle, nous nous attendions à ce qu'elle améliore la généralisation et réduise le surapprentissage. Cependant, les résultats obtenus ont montré une augmentation de la perte de validation (val_loss) de (0.0916 à 0.1730). Voici quelques raisons possibles pour cet effet inattendu :

1. Coefficient de Régularisation :

- Le coefficient de régularisation (dans notre cas, la valeur de 0.01 pour la régularisation L2) peut jouer un rôle crucial.
- Si le coefficient est trop élevé, il peut entraîner une forte pénalisation des poids du modèle, ce qui affecte négativement les performances.

2. Complexité du Modèle :

- La régularisation vise à réduire la complexité du modèle en limitant les valeurs des poids.
- Cependant, si le modèle était déjà relativement simple, l'ajout de régularisation peut entraîner une baisse de performance.

3. Interaction avec les Données d'Entraînement :

- La régularisation peut réagir différemment selon les données d'entraînement spécifiques.
- Si les données d'entraînement ont des caractéristiques particulières, la régularisation peut avoir un impact inattendu.

En réduisant le coefficient de régularisation de 0.1 à 0.001, nous prenons une mesure pour réduire la pénalisation des poids du modèle. Cela peut aider à améliorer la généralisation et à réduire le surapprentissage.

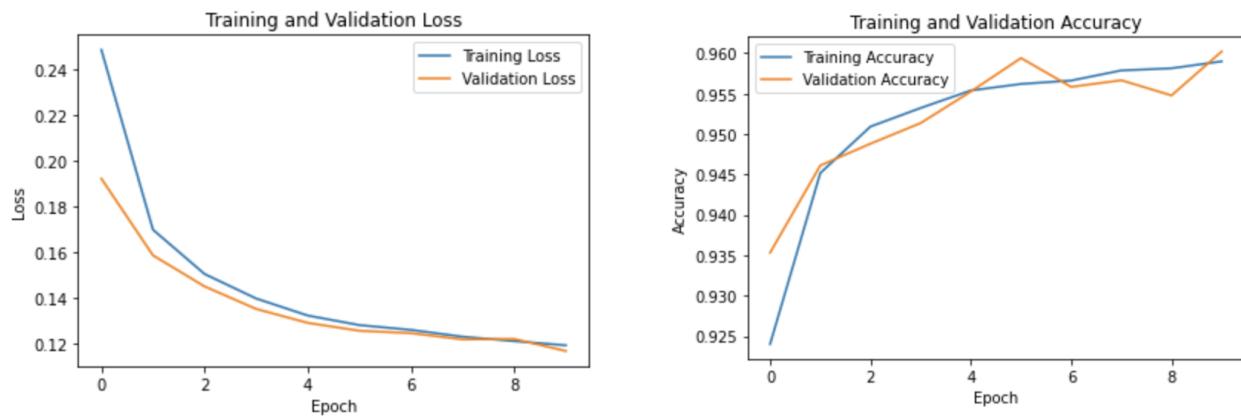
```
Entrée [11]: from tensorflow.keras import regularizers
model = tf.keras.Sequential([
    tf.keras.layers.Dense(64, activation='relu', input_shape=(X_train.shape[1],),
                          kernel_regularizer=regularizers.l2(0.001)), # Ajout de L2 Regularization
    tf.keras.layers.Dense(64, activation='relu',
                          kernel_regularizer=regularizers.l2(0.001)), # Ajout de L2 Regularization
    tf.keras.layers.Dense(1, activation='sigmoid')
])
2024-04-14 22:40:53.531563: I tensorflow/core/platform/cpu_feature_guard.cc:193] This TensorFlow binary is optimized with oneAPI Deep Neural Network Library (oneDNN) to use the following CPU instructions in performance-critical operations: AVX2 FMA
To enable them in other operations, rebuild TensorFlow with the appropriate compiler flags.
```

```

Entrée [14]: # Entraînement du modèle
print("\nDébut de l'entraînement du modèle...")
history = model.fit(X_train, y_train, epochs=10, batch_size=32,
                     validation_data=(X_val, y_val))

Epoch 0/10
2598/2598 [=====] - 3s 1ms/step - loss: 0.1324 - accuracy: 0.9513 - val_loss: 0.1292 - val_accuracy: 0.9552
Epoch 1/10
2598/2598 [=====] - 3s 1ms/step - loss: 0.1282 - accuracy: 0.9562 - val_loss: 0.1258 - val_accuracy: 0.9594
Epoch 2/10
2598/2598 [=====] - 3s 1ms/step - loss: 0.1261 - accuracy: 0.9566 - val_loss: 0.1247 - val_accuracy: 0.9558
Epoch 3/10
2598/2598 [=====] - 3s 1ms/step - loss: 0.1232 - accuracy: 0.9579 - val_loss: 0.1220 - val_accuracy: 0.9566
Epoch 4/10
2598/2598 [=====] - 3s 1ms/step - loss: 0.1212 - accuracy: 0.9581 - val_loss: 0.1222 - val_accuracy: 0.9548
Epoch 5/10
2598/2598 [=====] - 3s 1ms/step - loss: 0.1194 - accuracy: 0.9590 - val_loss: 0.1169 - val_accuracy: 0.9602

```



En effet, le modèle avec un coefficient de régularisation plus faible (0.001) semble mieux généraliser aux nouvelles données. La perte de validation (val_loss) est plus basse qu'avec le coefficient de régularisation initial (0.01). Cela signifie que le modèle est moins sujet au surapprentissage et qu'il devrait mieux se comporter sur des exemples inconnus.

On peut se demander maintenant pourquoi lors de notre premier graphique sans régularisation on avait un plus petit score au niveau de la val_loss (0.0916) mais du surapprentissage et au contraire avec ce dernier graphique on a beaucoup moins de surapprentissage mais la valeur de val_loss est plus élevée avec (0.1169).

1. Complexité du Modèle Initial :

- **Dans le premier graphique sans régularisation, le modèle était plus complexe (peut-être surajusté) aux données d'entraînement.**

- Cela a conduit à une perte de validation plus basse (0.0916), car le modèle s'ajustait très bien aux exemples d'entraînement spécifiques.

2. Effet de la Régularisation :

- Lorsque nous avons ajouté la régularisation L2, nous avons introduit une pénalisation sur les poids du modèle.
- Cela a réduit la complexité du modèle et l'a empêché de trop s'adapter aux données d'entraînement.
- En conséquence, la perte de validation a augmenté (0.1169), mais le modèle généralise mieux aux nouvelles données.

3. Objectif de Généralisation :

- L'objectif principal est d'obtenir un modèle qui généralise bien aux exemples inconnus (comme les données de validation ou de test).
- Parfois, une perte de validation légèrement plus élevée peut être préférable si elle signifie que le modèle est moins sujet au surapprentissage.

Évaluation du modèle sur les données de test

```
Entrée [17]: # Évaluation du modèle sur les données de test
test_loss, test_accuracy = model.evaluate(X_test_scaled, y_test_encoded)
print(f"Test Loss: {test_loss}")
print(f"Test Accuracy: {test_accuracy}")

812/812 [=====] - 1s 841us/step - loss: 0.1081 - accuracy: 0.9617
Test Loss: 0.1081
Test Accuracy: 0.9617
```

Le modèle a été évalué sur un ensemble de données de test distinct pour évaluer sa performance dans des conditions réelles. Les résultats de l'évaluation sont les suivants :

- **Perte de test (Test Loss) : 0.1081**
- **Précision de test (Test Accuracy) : 96.17%**

La précision de test de 96.17% indique que le modèle est capable de prédire correctement la satisfaction des clients dans la grande majorité des cas.

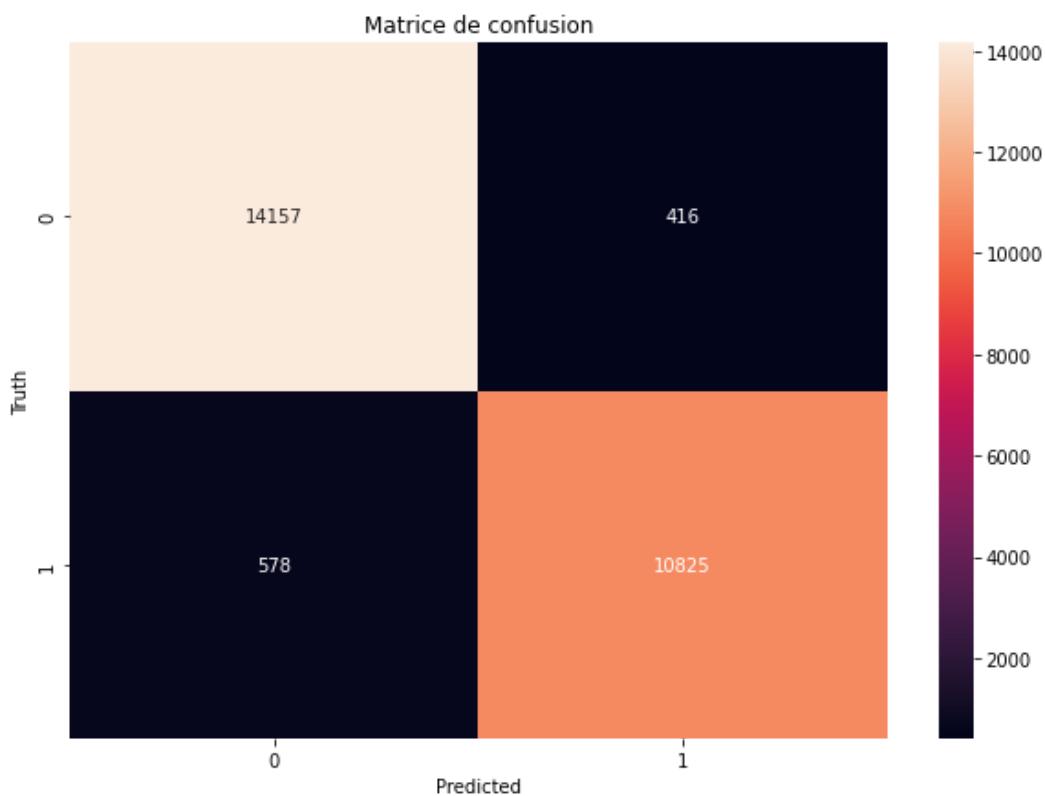
Cependant, il est également important d'examiner d'autres mesures telles que la

matrice de confusion et le rapport de classification pour obtenir une image plus complète de la performance du modèle.

Prédictions sur les données de test

Le modèle a été utilisé pour faire des prédictions sur l'ensemble de données de test. Les prédictions ont été comparées aux étiquettes réelles pour évaluer la performance du modèle.

- **Matrice de confusion :**



La matrice de confusion montre que le modèle a bien performé dans la prédiction des deux classes. Cependant, il a tendance à faire plus d'erreurs de prédiction pour la classe "Satisfied" (satisfait) que pour la classe "Neutral or Dissatisfied" (neutre ou insatisfait). Cela peut indiquer un déséquilibre dans les données d'entraînement ou des défis intrinsèques dans la prédiction de la satisfaction par rapport à l'insatisfaction.

Rapport de classification

```
Entrée [20]: # Affichage du rapport de classification
print(classification_report(y_test_encoded, predicted_classes))

precision    recall   f1-score   support
          0       0.96      0.97      0.97     14573
          1       0.96      0.95      0.96     11403
avg / total       0.96      0.96      0.96     25976
```

Le rapport de classification fournit une analyse détaillée de la performance du modèle pour chaque classe. Voici les mesures de performance pour chaque classe :

Classe 0 (Satisfied - Satisfait) :

- **Précision** : 96%
- **Rappel** : 97%
- **F1-score** : 97%

Ces résultats indiquent que le modèle a une précision élevée de 96% pour prédire les clients satisfaits. De plus, il a un rappel élevé de 97%, ce qui signifie qu'il identifie correctement la grande majorité des clients satisfaits. Le F1-score, qui combine à la fois la précision et le rappel, est également élevé à 97%.

Globalement, le modèle a une performance solide pour prédire la satisfaction des clients.

Classe 1 (Neutral or Dissatisfied - Neutre ou insatisfait) :

- **Précision** : 96%
- **Rappel** : 95%
- **F1-score** : 96%

Pour la classe des clients neutres ou insatisfaits, le modèle affiche une précision de 96%, ce qui signifie qu'il prédit correctement la grande majorité des clients comme étant neutres ou insatisfaits. Le rappel, cependant, est légèrement inférieur à 95%, indiquant que le modèle a tendance à manquer certains des clients neutres ou insatisfaits. Malgré cela, le F1-score reste élevé à 96%, ce qui suggère une bonne capacité du modèle à généraliser sur cette classe.

Conclusion

En conclusion, ce travail pratique a permis de construire et d'évaluer un modèle de classification visant à prédire la satisfaction des clients. À travers différentes étapes, nous avons traité les données, construit un modèle de réseau neuronal avec des couches de régularisation, et évalué sa performance sur des données de test. Le modèle de classification de satisfaction des clients présente une précision élevée d'environ 96% pour prédire la satisfaction ou l'insatisfaction des clients. Ces résultats solides suggèrent que le modèle peut être un outil efficace pour évaluer le niveau de satisfaction des clients dans divers contextes commerciaux. Cependant, des améliorations pourraient être apportées pour mieux identifier les clients insatisfaits. Dans l'ensemble, le modèle offre une perspective précieuse pour comprendre et améliorer l'expérience client.