Notebook

November 20, 2024

1 Partie 1:

1.1 Importation et installation des packages

[ ]:
```
# Importation des bibliothèques nécessaires
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import tensorflow as tf
from tensorflow.keras import layers, models
import matplotlib.pyplot as plt
from tabulate import tabulate
import seaborn as sns

!pip install ucimlrepo
```

```
Collecting ucimlrepo
  Downloading ucimlrepo-0.0.3-py3-none-any.whl (7.0 kB)
Installing collected packages: ucimlrepo
Successfully installed ucimlrepo-0.0.3
```

1.2 Importation de la base de donnée

[ ]:
```
# Étape 1: Chargement et préparation des données
# ----------------------

# Chargement des données

red_url = "https://archive.ics.uci.edu/ml/machine-learning-databases/
 ↪wine-quality/winequality-red.csv"
white_url = "https://archive.ics.uci.edu/ml/machine-learning-databases/
 ↪wine-quality/winequality-white.csv"

red_wine_data = pd.read_csv(red_url, sep=';')
white_wine_data = pd.read_csv(white_url, sep=';')
```

```
#Deux ensembles de données sont inclus, ils sont liés aux variantes rouges et␣
 ↪blanches du vin portugais « Vinho Verde ».
#L'objectif est de modéliser la qualité du vin à partir de tests␣
 ↪physico-chimiques.
#Par la suite, nous allons focaliser notre étude sur le vin blanc.
```

1.3 Visualisation et analyse des données

[ ]:
```
## Visualisation et analyse des données
```

[ ]:
```
print("Dimension de la dataset wine_data : ",white_wine_data.shape)

#Cette fonction nous montre les dimensions du wine_data, il y a 4898 lignes et␣
 ↪12 colonnes
```

```
Dimension de la dataset wine_data :  (4898, 12)
```

[ ]:
```
print(white_wine_data.head())

#Cette ligne de code nous montre les cinq première lignes de notre dataset. On␣
 ↪obtient un aperçu rapide des données, les noms des colonnes, et la nature␣
 ↪des informations contenues dans la dataset.
#Nous avons 12 variables qui sont: fixed_acidity, volatile acidity, citric␣
 ↪acid, residual sugar, chlorides, free sulfure, dioxide, total sulfur␣
 ↪dioxide, density, pH, sulphates, alcohol, quality
#La variable quality est la variable que l'on cherche à expliquer
```

```
   fixed acidity  volatile acidity  citric acid  residual sugar  chlorides  \
0            7.0              0.27         0.36            20.7      0.045
1            6.3              0.30         0.34             1.6      0.049
2            8.1              0.28         0.40             6.9      0.050
3            7.2              0.23         0.32             8.5      0.058
4            7.2              0.23         0.32             8.5      0.058

   free sulfur dioxide  total sulfur dioxide  density    pH  sulphates  \
0                 45.0                 170.0   1.0010  3.00       0.45
1                 14.0                 132.0   0.9940  3.30       0.49
2                 30.0                  97.0   0.9951  3.26       0.44
3                 47.0                 186.0   0.9956  3.19       0.40
4                 47.0                 186.0   0.9956  3.19       0.40

   alcohol  quality
0      8.8        6
1      9.5        6
2     10.1        6
3      9.9        6
4      9.9        6
```

[ ]:
```
### Vérification de la présence de valeurs nulles dans les différentes dataset␣
↪et variables

white_wine_data.isna().any()

#On remarque que pour chaque variables est indiqué False, ce qui veut dire␣
↪qu'il n'y a pas de valeurs manquantes
```

[ ]:
```
fixed acidity           False
volatile acidity        False
citric acid             False
residual sugar          False
chlorides               False
free sulfur dioxide     False
total sulfur dioxide    False
density                 False
pH                      False
sulphates               False
alcohol                 False
quality                 False
dtype: bool
```

Nous faisons un tableau des statistiques descriptives, fournissant ainsi une vue plus organisée et structurée des différentes mesures statistiques pour chaque variable dans la dataset white_wine_data On observe que 50 % des vins ont une qualité comprise entre un score 5 et un score 6 Ce qui veut dire qu'il y a beaucoup plus de vins normaux que d'excellents ou de mauvais

[ ]:
```
# Statistiques descriptives

descriptive_stats = white_wine_data.describe()
print("Statistiques descriptives :")
table = tabulate(descriptive_stats, headers='keys', tablefmt='pretty')
print(table)
```

```
Statistiques descriptives :
+-------+------------------+--------------------+--------------------+-------
------------+----------------------+--------------------+----------------------
+-----------------------+--------------------+--------------------+-----------
--------+--------------------+
|       |   fixed acidity    |  volatile acidity    |     citric acid     |
residual sugar    |      chlorides       | free sulfur dioxide | total sulfur
dioxide |        density        |         pH         |      sulphates      |
alcohol       |      quality       |
+-------+------------------+--------------------+--------------------+-------
------------+----------------------+--------------------+----------------------
+-----------------------+--------------------+--------------------+-----------
--------+--------------------+
| count |      4898.0       |       4898.0        |       4898.0        |
```

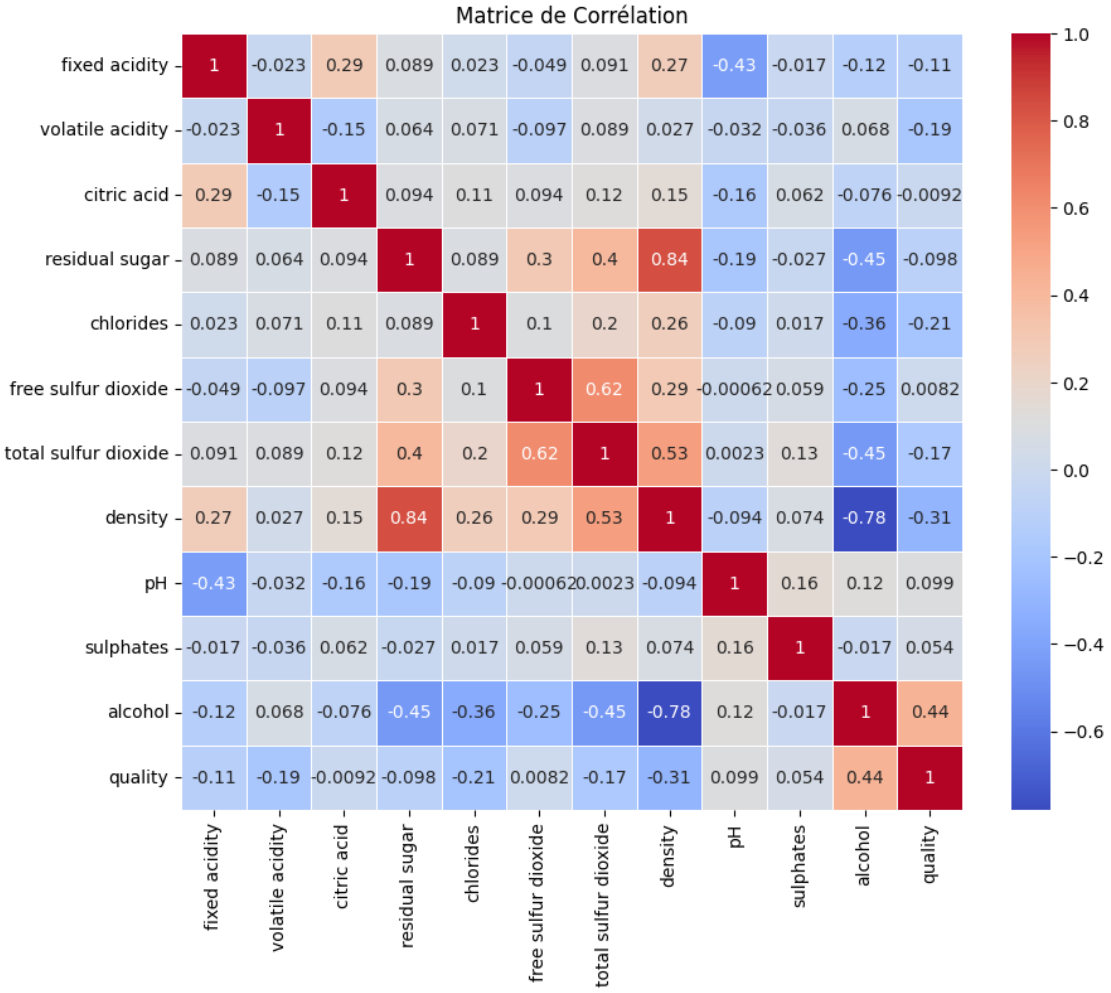
```
4898.0       |        4898.0        |       4898.0        |        4898.0
|        4898.0         |       4898.0       |       4898.0        |
4898.0       |       4898.0       |
| mean  | 6.854787668436097 | 0.27824111882400976 | 0.33419150673744386 |
6.391414863209474 | 0.04577235606369946  |  35.30808493262556  |
138.36065741118824  |  0.9940273764801959   | 3.1882666394446715 |
0.48984687627603113 | 10.514267047774602 |  5.87790935075541  |
|  std  | 0.843868227687513 | 0.10079454842486534 | 0.12101980420298249 |
5.072057784014881 | 0.021847968093728798 |  17.00713732523259  |
42.49806455414291   | 0.0029909069169369337 | 0.1510005996150668 |
0.1141258339488323  | 1.230620567757318  | 0.8856385749678312 |
|  min  |        3.8        |        0.08         |         0.0         |
0.6        |        0.009         |         2.0         |         9.0          |
0.98711        |        2.72         |        0.22         |        8.0         |
3.0         |
|  25%  |        6.3        |        0.21         |        0.27         |
1.7        |        0.036         |        23.0         |        108.0         |
0.9917225000000001   |        3.09         |        0.41         |        9.5
|        5.0         |
|  50%  |        6.8        |        0.26         |        0.32         |
5.2        |        0.043         |        34.0         |        134.0         |
0.99374        |        3.18         |        0.47         |        10.4        |
6.0         |
|  75%  |        7.3        |        0.32         |        0.39         |
9.9        |         0.05         |        46.0         |        167.0         |
0.9961         |        3.28         |        0.55         |        11.4        |
6.0         |
|  max  |       14.2        |         1.1         |        1.66         |
65.8        |        0.346         |        289.0        |        440.0
|        1.03898        |        3.82         |        1.08         |        14.2
|        9.0         |
+-------+------------------+--------------------+--------------------+-------
------------+----------------------+--------------------+----------------------
+----------------------+--------------------+--------------------+------------
--------+--------------------+
```

1.3.1 Tableau de corrélation

Ce graphe permet de visualiser les relations de corrélation entre les différentes variables de la dataset, et d'identifier les relations importantes. Les valeurs de correlation varient de -1 à 1, où -1 indique une corrélation négative parfaite, 1 une corrélation positive parfaite et 0 une absence de corrélation linéaire. Alcohol a une corrélation positive de 0,44 avec quality

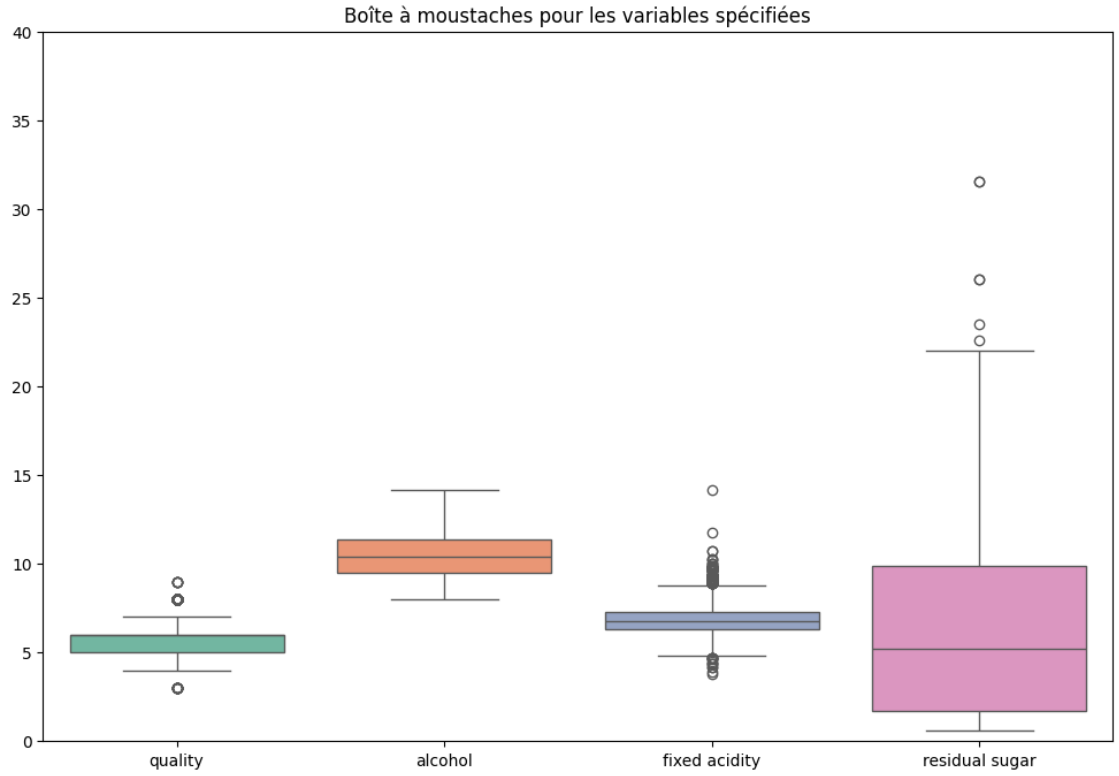
[ ]:
```
# Tableau de corrélations

correlation_matrix = white_wine_data.corr()

# Graphe de corrélation
```

```
plt.figure(figsize=(10, 8))
sns.heatmap(correlation_matrix, annot=True, cmap="coolwarm", linewidths=.5)
plt.title("Matrice de Corrélation")
plt.show()
```



Pour notre étude nous avons décidé de sélectionner 4 variables qui sont: quality, alcohol, fixed acidity, residual sugar La ligne centrale de la boîte à moustache représente la médiane qui définit la tendance de la distribution La longueur de la boîte indique la dispersion des valeurs centrales, c'est la variable residual sugar qui possède la plus grande dispersion des valeurs centrales L'étendue des moustaches donne une indication de la variabilité globale des données, c'est la variable sugar qui possède la plus forte variabilité globale des données Les points individuels en dehors des moustaches sont des outliers, on en remarque pour la variable fixed acidity principalement, residual sugar également, et quelques uns pour la variable quality On observe la présence de valeurs extrêmes, nous avons cependant décidé de les conserver puisqu'elles semblent être expliquées par l'hétérogénéité et ne pose pas de problème à notre étude

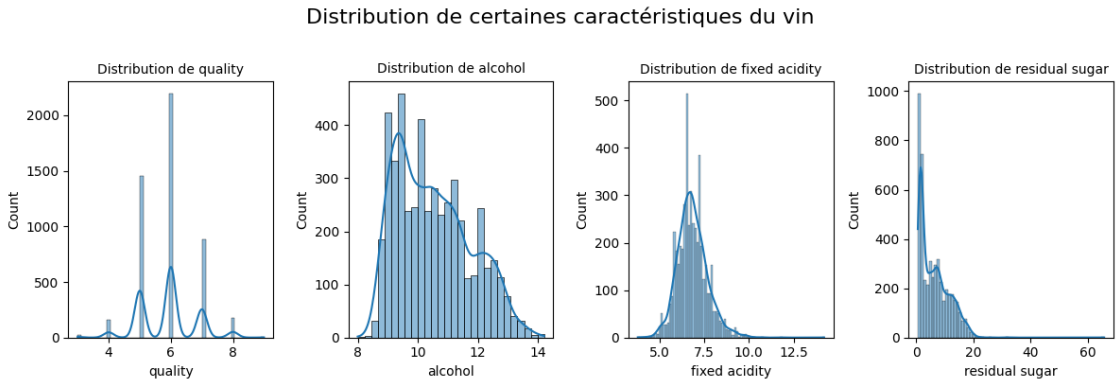
[ ]:
```
# Sélectionner les colonnes spécifiques
boxplot_columns = ['quality', 'alcohol', 'fixed acidity', 'residual sugar']

selected_data = white_wine_data[boxplot_columns]

# Boîte à moustaches pour les variables spécifiées
plt.figure(figsize=(12, 8))
sns.boxplot(data=selected_data, palette="Set2")
plt.title("Boîte à moustaches pour les variables spécifiées")
plt.ylim(0, 40)
plt.show()
```



Les histogrammes obtenus dévoilent la structure des données et examinent la distribution des variables représentées Pour la distribution de la variable quality, on voit que les scores s'étendent de 3 à 9 La distribution pour cette variable correspond à une distribution multimodale Les queues de l'histogramme représentent la fréquence des valeurs extrêmes, une queue plus longue indique des valeurs plus fréquentes Pour cette variable c'est le score 6 qui est le plus fréquent, suivit du score 5, puis le score 4, et au même niveau le score 4 et 8 L'épaisseur des barres de l'histogramme peut également fournir des informations. Des barres plus fines peuvent indiquer une dispersion plus fine des valeurs. Pour la variable quality la dispersion des valeurs semble plus fine que pour la variable alcohol, tandis qu'en comparaison à la variable fixed acidity et residual sugar, la distribution de quality est plus importante Les courbes superposés aux barres montrent la forme générale de la distribution Pour la variable alcohol, les valeurs vont de 8 à 14, l'histogramme s'étale, mais le pic se

situe principalement aux alentours de 9,5 L'étalement de l'histogramme pour cette variable alcohol explique la forte dispersion des valeurs Pour la variable fixed acidity la distribution est normale mais légérement décallée vers la gauche, les valeurs s'étendent d'environ 4 à 14, les barres sont fines, de plus la dispersion des valeurs est faible, avec un pic aux alentours de 6,5 Pour la variable residual sugar, l'histogramme est concentré sur la gauche et dévoile une très faible dispersion des valeurs, les valeurs vont tout de même de 0,5 à 66 environ

[ ]:
```
fig, ax = plt.subplots(ncols=4, nrows=1, figsize=(12, 4))  # Réduction du␣
 ↪nombre de colonnes
ax = ax.flatten()

columns = ['quality', 'alcohol', 'fixed acidity', 'residual sugar']

for i, col in enumerate(columns):
    sns.histplot(white_wine_data[col], ax=ax[i], kde=True)
    ax[i].set_title(f'Distribution de {col}', fontsize=10)  # Ajout de titres

# Ajustements de la mise en page
plt.suptitle('Distribution de certaines caractéristiques du vin', y=1.02,␣
 ↪fontsize=16)
plt.tight_layout()
plt.show()
```



1.4 Création d'une variable binaire pour la qualité du vin

Cette ligne de code permet l'analyse et la classification binaire de la qualité du vin blanc Cela permet de créer une variable binaire d'intérêt ('quality_binary') qui indique si la qualité du vin est supérieure ou égale à 5 (1) ou non (0). Ainsi, la nouvelle colonne 'quality_binary' contiendra des valeurs binaires : 1 si la qualité est supérieure ou égale à 5, et 0 sinon. (bon vin"grade<5"/mauvais vin"grade>=5)

[ ]:
```
white_wine_data['quality_binary'] = (white_wine_data['quality'] >= 5).
 ↪astype(int)
```

1.5 Prétraitement des données

Ce code effectue les étapes préliminaires de prétraitement des données nécessaires avant d'entraîner un modèle d'apprentissage automatique.

[ ]:
```
# Séparation des caractéristiques et de la variable cible
X = white_wine_data.drop(['quality', 'quality_binary'], axis=1)
y = white_wine_data['quality_binary']

# Division des données en ensembles d'entraînement et de test
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
↪random_state=42)

# Normalisation des caractéristiques
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

1.6 Conception d'un FFNN simple

Ce code définit un modèle de réseau de neurones pour la classification binaire et le compile avec l'optimiseur SGD et la fonction de perte binary_crossentropy.

[ ]:
```
# Création du modèle
model = models.Sequential()
model.add(layers.Dense(10, activation='relu', input_shape=(X_train_scaled.
↪shape[1],)))
model.add(layers.Dense(1, activation='sigmoid'))

# Compilation du modèle
model.compile(optimizer='sgd', loss='binary_crossentropy', metrics=['accuracy'])
```

1.7 Entraînement du modèle

Ce code montre les résultats de l'entraînement d'un modèle d'apprentissage automatique sur les données d'entraînement. Pour la première époque, la perte sur les données d'entraînement est de 0.4415 avec une précision de 92.21%. Et, la perte sur les données de validation est 0.3582 avec une précision de 96.30%. Ce processus est répété pour chaque époque, on comptabilise 10 époques.

[ ]:
```
# Entraînement du modèle
history = model.fit(X_train_scaled, y_train, epochs=10, batch_size=32,␣
↪validation_split=0.2)
```

```
Epoch 1/10
98/98 [==============================] - 1s 5ms/step - loss: 0.4415 - accuracy:
0.9221 - val_loss: 0.3582 - val_accuracy: 0.9630
Epoch 2/10
98/98 [==============================] - 0s 3ms/step - loss: 0.3155 - accuracy:
0.9588 - val_loss: 0.2827 - val_accuracy: 0.9643
```

```
Epoch 3/10
98/98 [==============================] - 0s 2ms/step - loss: 0.2623 - accuracy:
0.9601 - val_loss: 0.2457 - val_accuracy: 0.9643
Epoch 4/10
98/98 [==============================] - 0s 3ms/step - loss: 0.2338 - accuracy:
0.9601 - val_loss: 0.2241 - val_accuracy: 0.9643
Epoch 5/10
98/98 [==============================] - 0s 2ms/step - loss: 0.2162 - accuracy:
0.9601 - val_loss: 0.2103 - val_accuracy: 0.9643
Epoch 6/10
98/98 [==============================] - 0s 3ms/step - loss: 0.2044 - accuracy:
0.9601 - val_loss: 0.2010 - val_accuracy: 0.9643
Epoch 7/10
98/98 [==============================] - 0s 3ms/step - loss: 0.1959 - accuracy:
0.9601 - val_loss: 0.1942 - val_accuracy: 0.9643
Epoch 8/10
98/98 [==============================] - 0s 3ms/step - loss: 0.1894 - accuracy:
0.9601 - val_loss: 0.1891 - val_accuracy: 0.9643
Epoch 9/10
98/98 [==============================] - 0s 2ms/step - loss: 0.1843 - accuracy:
0.9601 - val_loss: 0.1852 - val_accuracy: 0.9643
Epoch 10/10
98/98 [==============================] - 0s 2ms/step - loss: 0.1800 - accuracy:
0.9601 - val_loss: 0.1821 - val_accuracy: 0.9643
```

1.8 Évaluation du modèle

Ce code évalue les performances du modèle sur l'ensemble de test, il mesure la performance réelle du modèle sur des données qu'il n'a pas vues lors de l'entraînement. Le modèle a évalué 31 lots de données de test. La perte moyenne sur l'ensemble de test est de 0.1758, et la précision moyenne est de 0.9694. Une précision élevée et une perte faible indiquent que le modèle généralise bien aux nouvelles données et est efficace dans la résolution du problème de classification.

[ ]:
```python
# Évaluation du modèle sur l'ensemble de test

test_loss, test_accuracy = model.evaluate(X_test_scaled, y_test)
print(f"Test Accuracy: {test_accuracy}")
```

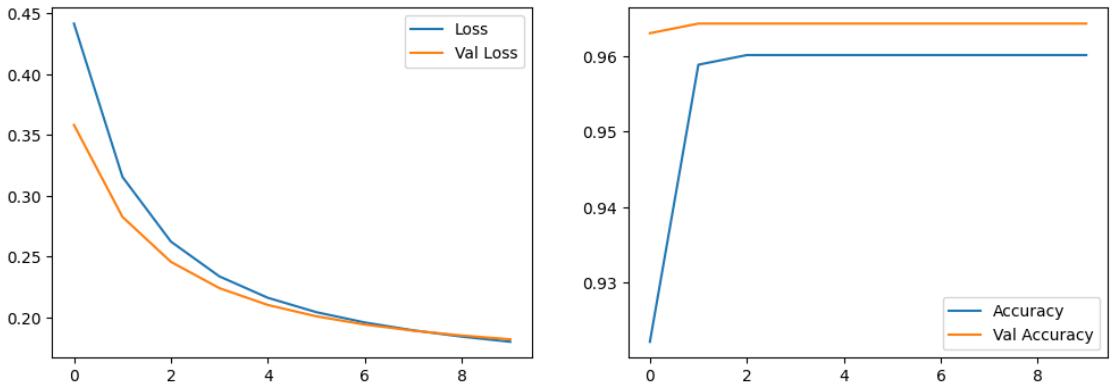
```
31/31 [==============================] - 0s 2ms/step - loss: 0.1758 - accuracy:
0.9694
Test Accuracy: 0.9693877696990967
```

1.9 Visualisation et quantification

Ce code permet de visualiser graphiquement l'évolution de la perte et de la précision du modèle sur les ensembles d'entraînement et de validation au fil des epochs. La courbe intitulée "Loss" représente l'évolution de la perte d'entraînement au fil des epochs. Elle mesure l'erreur du modèle sur les données d'entraînement pendant le processus d'entraînement. Sur cette courbe, la perte diminue

progressivement au fil des epochs. Plus la valeur de la perte est basse, meilleure est la performance du modèle. Cela indique donc que le modèle s'ajuste correctement aux données d'entraînement et apprend à minimiser son erreur. La courbe intitulée "Val Loss" représente l'évolution de la perte de validation au fil des epochs. Elle permet d'évaluer la capacité du modèle à généraliser à de nouvelles données qu'il n'a pas vu pendant l'entraînement. On remarque que la perte diminue au début, puis finit par se stabiliser, c'est synonyme d'un bon modèle. En effet, car si cette perte commence à augmenter après un certain nombre d'epochs, cela peut indiquer un surapprentissage (overfitting). La courbe intitulée "Accuracy" représente l'évolution de la précision d'entraînement au fil des epochs. On constate une augmentation régulière de la précision, cela signifie que le modèle apprend à bien classer les données d'entraînement. La courbe intitulée "Val Accuracy" représente l'évolution de la précision de validation au fil des epochs. On remarque qu'elle suit une tendance similaire à celle de la précision d'entraînement. L'analyse de ces courbres témoignent d'une bonne performance du modèle.

[ ]:
```
# Visualisation de la perte et de la précision
plt.figure(figsize=(12, 4))
plt.subplot(1, 2, 1)
plt.plot(history.history['loss'], label='Loss')
plt.plot(history.history['val_loss'], label='Val Loss')
plt.legend()
plt.subplot(1, 2, 2)
plt.plot(history.history['accuracy'], label='Accuracy')
plt.plot(history.history['val_accuracy'], label='Val Accuracy')
plt.legend()
plt.show()
```



2 Partie 2 :

Ce code crée un modèle de réseau neuronal avec une architecture composée de deux couches cachées, où chaque couche contient 50 neurones. Ensuite, il ajoute une couche de sortie avec une seule sortie binaire. La fonction d'activation ReLU est employée dans les couches cachées pour introduire des aspects non linéaires dans le modèle. La fonction sigmoïde est utilisée dans la couche de sortie afin de générer des prédictions qui se situent dans l'intervalle [0, 1].

```
[ ]: # Construction du deuxième modèle

model2 = models.Sequential()
model2.add(layers.Dense(50, activation='relu', input_shape=(X_train_scaled.
 ↪shape[1],)))
model2.add(layers.Dense(50, activation='relu'))
model2.add(layers.Dense(1, activation='sigmoid'))
```

L'argument sgd est utilisé pour minimiser la fonction de perte du modèle. La fonction de perte binary_crossentropy est utilisée, car elle est adaptée aux problèmes de classification binaire. L'argument accuracy est utilisé car le modèle sera évalué en fonction de sa capacité à prédire correctement les classes des échantillons de données.

```
[ ]: # Compilation du modèle

model2.compile(optimizer='sgd', loss='binary_crossentropy',␣
 ↪metrics=['accuracy'])
```

Ce code entraîne le modèle de réseau neuronal étudié précedemment sur des données d'entraînement normalisées. L'entraînement se fait sur 150 époques avec une taille de lot de 32 échantillons, et 20% des données d'entraînement sont réservées pour la validation.

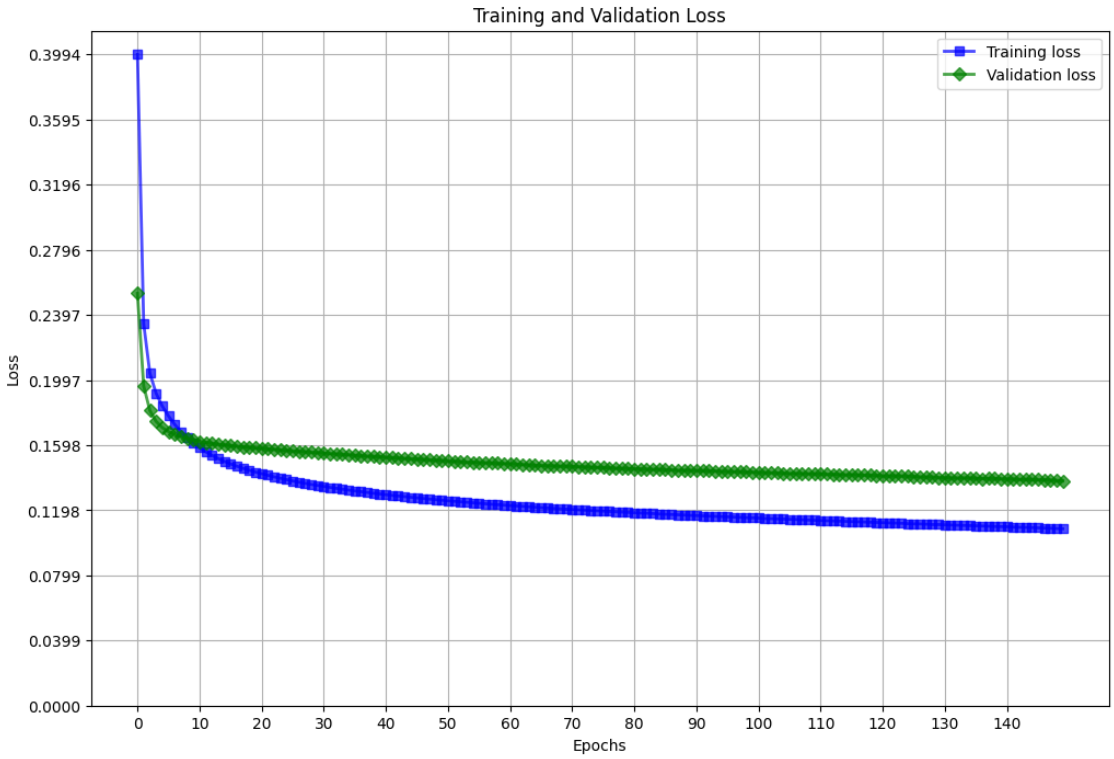
```
[ ]: # Entraînement du modèle

history2 = model2.fit(X_train_scaled, y_train, validation_split=0.2,␣
 ↪epochs=150, batch_size=32, verbose=0)
```

2.1 Ajustement Insuffisant et Ajustement Excessif

Les pertes d'entraînement et de validation servent généralement à évaluer l'efficacité du modèle en détectant l'overfitting. Le graphique montre une diminution des pertes d'entraînement et de validation au fil des epochs, ce qui confirme que le processus d'entraînement progresse dans la bonne direction. Les résultats du graphique révelent donc que le modèle apprend correctement et s'améliore progressivement. On constate que la perte pour l'entraînement est plus forte que la perte de validation. Cependant, l'écart n'est pas important, les pertes diminuent de manière cohérente et reste proches l'une de l'autre. Ces analyses dévoilent donc que nous ne sommes donc pas dans une situation d'overfitting. Le modèle ne se contente pas de mémoriser les données d'entraînement mais parvient à généraliser efficacement à de nouvelles données.

```
[ ]: # Visualisation des pertes d'entraînement et de validation

plt.figure(figsize=(12, 8))

plt.plot(history2.history['loss'], 'b-s', label='Training loss', markersize=6,␣
 ↪linewidth=2, alpha=0.7)
plt.plot(history2.history['val_loss'], 'g-D', label='Validation loss',␣
 ↪markersize=6, linewidth=2, alpha=0.7)
plt.title('Training and Validation Loss')
```

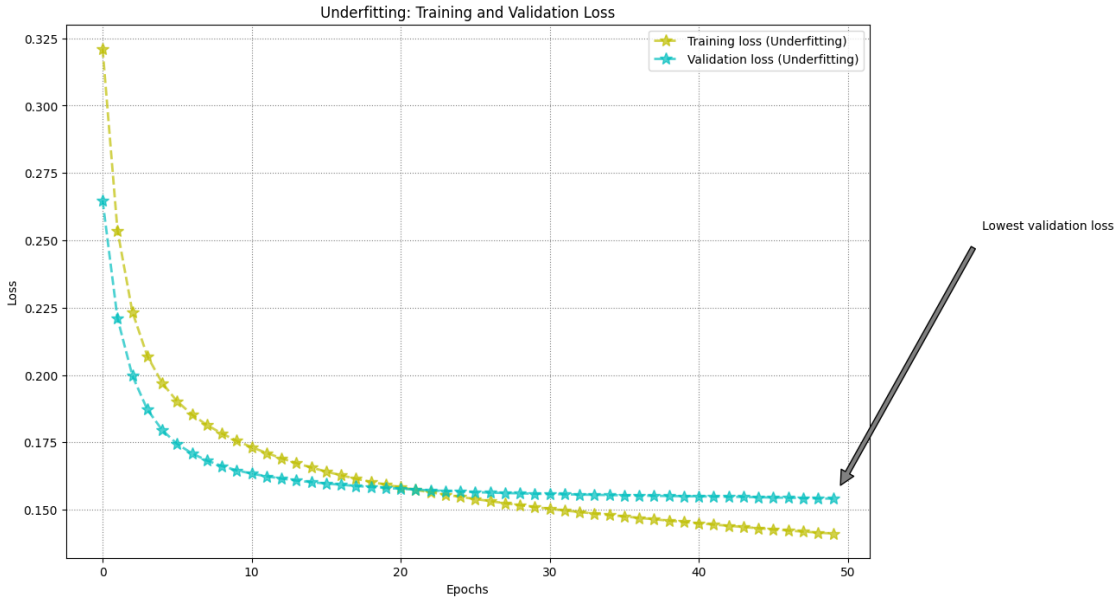
```
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.grid(True)
plt.legend()
plt.xticks(ticks=range(0, len(history2.history['loss']), 10))
plt.yticks(ticks=np.linspace(0, max(history2.history['loss']), num=11))
plt.show()
```



Ce code est destiné à illustrer une situation d'insuffisance d'ajustement (underfitting) d'un modèle d'apprentissage automatique. On remarque que les pertes d'entraînement et de validation restent élevées et ne convergent pas. De 0 à 20 epochs, les perte de validation sont plus fortes que les pertes d'entraînement. En conclusion, le modèle n'est pas suffisamment complexe pour capturer la structure des données et sous-apprend.

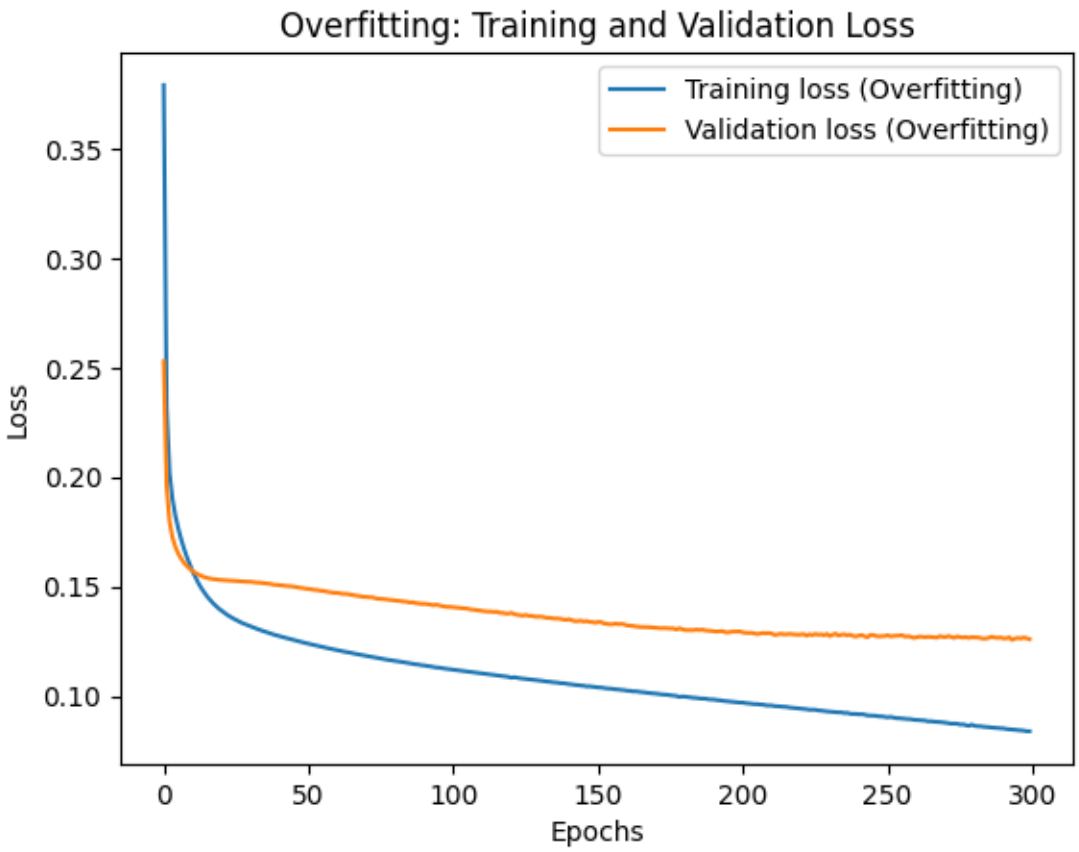
[ ]:
```
model_underfit = models.Sequential()
model_underfit.add(layers.Dense(10, activation='relu',␣
 ↪input_shape=(X_train_scaled.shape[1],))) # Moins de neurones
model_underfit.add(layers.Dense(1, activation='sigmoid'))
model_underfit.compile(optimizer='sgd', loss='binary_crossentropy',␣
 ↪metrics=['accuracy'])
history_underfit = model_underfit.fit(X_train_scaled, y_train,␣
 ↪validation_split=0.2, epochs=50, batch_size=32, verbose=0) # Moins d'époques
```

```
# Visualisation pour l'Ajustement Insuffisant

plt.figure(figsize=(12, 8))

# Utilisation des lignes pointillées avec des marqueurs en forme d'étoiles
plt.plot(history_underfit.history['loss'], 'y--*', label='Training loss␣
 ↪(Underfitting)', markersize=10, linewidth=2, alpha=0.7)
plt.plot(history_underfit.history['val_loss'], 'c--*', label='Validation loss␣
 ↪(Underfitting)', markersize=10, linewidth=2, alpha=0.7)

# Titre et étiquettes avec des couleurs standards
plt.title('Underfitting: Training and Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')

# Utilisation des couleurs de texte standard pour les axes
plt.xticks()
plt.yticks()

# Ajout d'une légende
plt.legend()

# Ajout d'une grille
plt.grid(True, linestyle=':', color='grey')

# Ajout d'une annotation pour indiquer le point de la plus petite perte de␣
 ↪validation
min_val_loss_epoch = np.argmin(history_underfit.history['val_loss'])  # Trouver␣
 ↪l'époque de la plus petite perte de validation
min_val_loss = min(history_underfit.history['val_loss'])
plt.annotate('Lowest validation loss',
             xy=(min_val_loss_epoch, min_val_loss),
             xytext=(min_val_loss_epoch+10, min_val_loss+0.1),
             arrowprops=dict(facecolor='grey', shrink=0.05))

# Affichage du graphique
plt.show()
```



Ce code est destiné à illustrer un cas de surajustement (overfitting) dans un modèle d'apprentissage automatique. On constate que les pertes d'entraînement continuent de diminuer tandis que les pertes de validation commencent à augmenter après un certain point. Cette situation suggère que le modèle s'adapte excessivement aux données d'entraînement et ne généralise pas conveablement aux nouvelles données.

[ ]:
```
model_overfit = models.Sequential()
model_overfit.add(layers.Dense(100, activation='relu',␣
 ↪input_shape=(X_train_scaled.shape[1],))) # Plus de neurones
model_overfit.add(layers.Dense(100, activation='relu')) # Couche supplémentaire
model_overfit.add(layers.Dense(1, activation='sigmoid'))
model_overfit.compile(optimizer='sgd', loss='binary_crossentropy',␣
 ↪metrics=['accuracy'])
history_overfit = model_overfit.fit(X_train_scaled, y_train, validation_split=0.
 ↪2, epochs=300, batch_size=32, verbose=0) # Plus d'époques

# Visualisation pour l'Ajustement Excessif

plt.plot(history_overfit.history['loss'], label='Training loss (Overfitting)')
plt.plot(history_overfit.history['val_loss'], label='Validation loss␣
 ↪(Overfitting)')
plt.title('Overfitting: Training and Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```

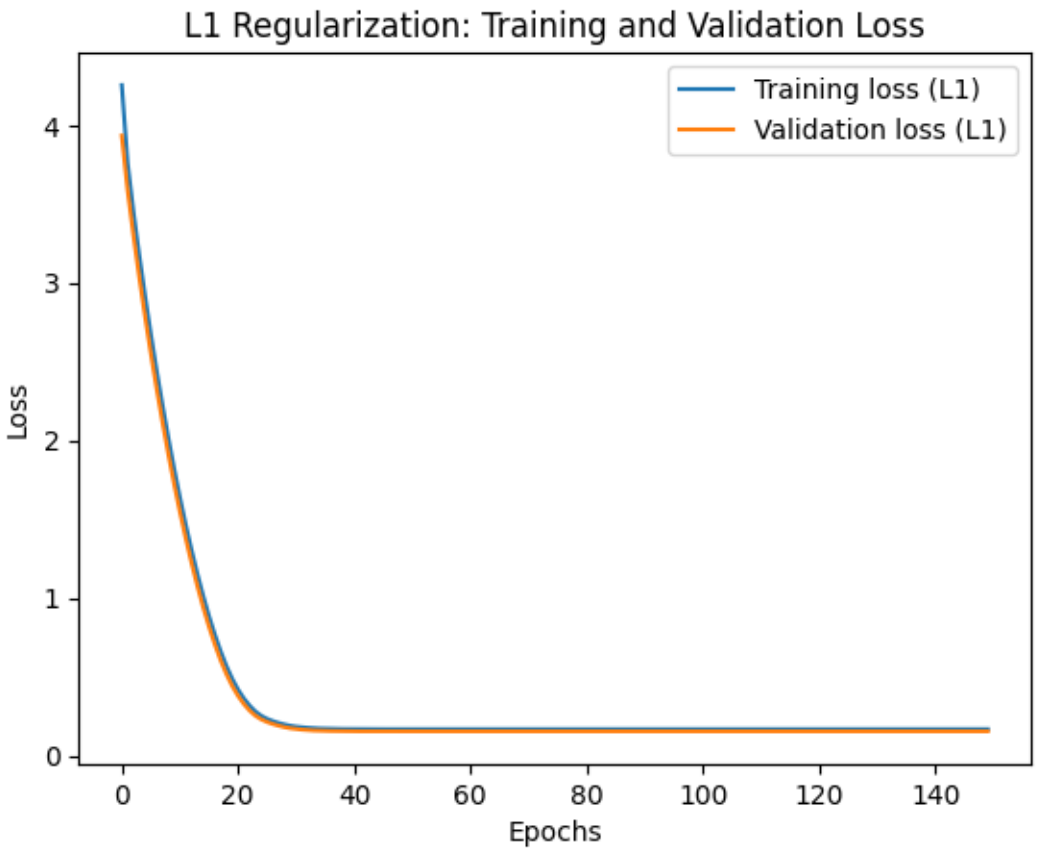


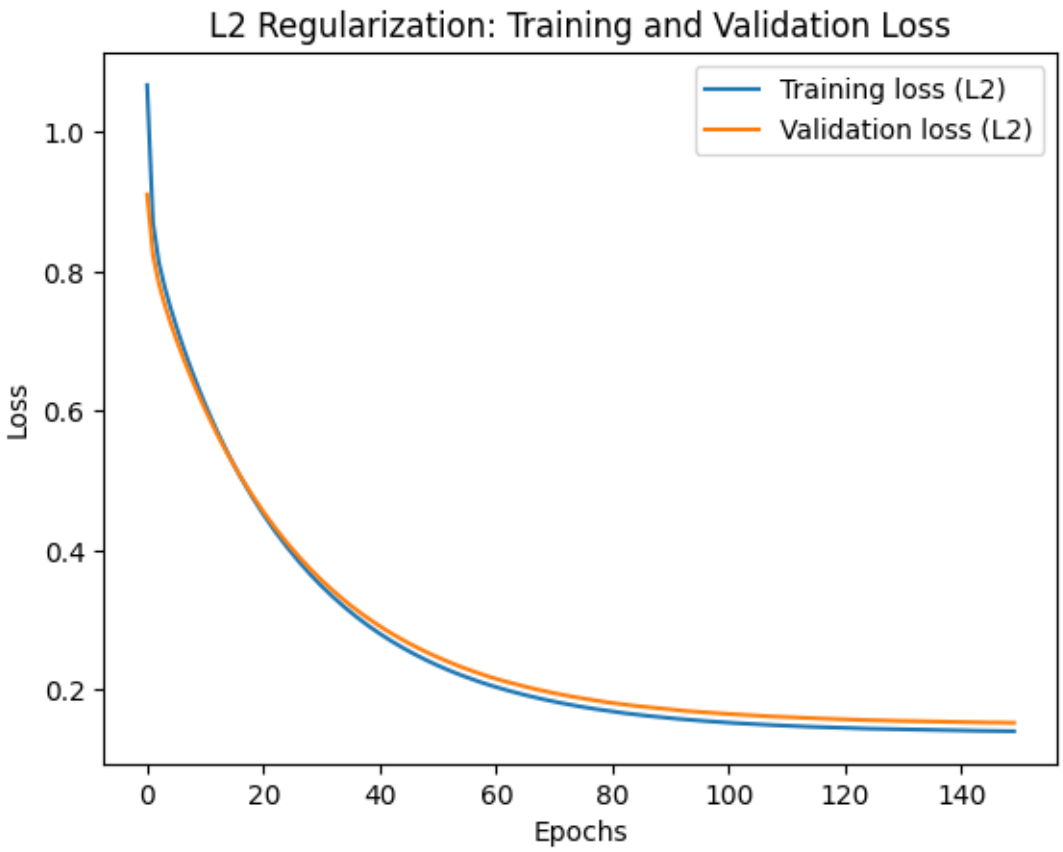
2.2 Régularisation L1 et L2

Ce code illustre l'utilisation de la régularisation L1 et L2 dans l'entraînement de réseaux de neurones. Les courbes représentent l'évolution de la perte d'entraînement et de validation au fil des époques pendant l'entraînement des modèles utilisant la régularisation L1 et L2. La perte d'entraînement est minimisé durant les époques, le modèle apprend à mieux ajuster ses paramètres pour obtenir des prédictions plus précises sur les données d'entraînement. La perte de validation diminue également durant les époques, le modèle généralise bien et ne surapprend pas les données d'entraînement. Les deux courbes diminuent et restent proches l'une de l'autre, on peut alors dire que le modèle ne surapprend pas et généralise bien. En conclusion, la régularisation L1 et la régularisation L2 sont efficaces.

[ ]:
```
from tensorflow.keras import regularizers

# Régularisation L1
model_l1 = models.Sequential()
model_l1.add(layers.Dense(50, activation='relu',␣
 ↪kernel_regularizer=regularizers.l1(0.01), input_shape=(X_train_scaled.
 ↪shape[1],)))
```

```
model_l1.add(layers.Dense(50, activation='relu',␣
 ↪kernel_regularizer=regularizers.l1(0.01)))
model_l1.add(layers.Dense(1, activation='sigmoid'))
model_l1.compile(optimizer='sgd', loss='binary_crossentropy',␣
 ↪metrics=['accuracy'])
history_l1 = model_l1.fit(X_train_scaled, y_train, validation_split=0.2,␣
 ↪epochs=150, batch_size=32, verbose=0)

# Visualisation pour la Régularisation L1
plt.plot(history_l1.history['loss'], label='Training loss (L1)')
plt.plot(history_l1.history['val_loss'], label='Validation loss (L1)')
plt.title('L1 Regularization: Training and Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()

# Régularisation L2
model_l2 = models.Sequential()
model_l2.add(layers.Dense(50, activation='relu',␣
 ↪kernel_regularizer=regularizers.l2(0.01), input_shape=(X_train_scaled.
 ↪shape[1],)))
model_l2.add(layers.Dense(50, activation='relu',␣
 ↪kernel_regularizer=regularizers.l2(0.01)))
model_l2.add(layers.Dense(1, activation='sigmoid'))
model_l2.compile(optimizer='sgd', loss='binary_crossentropy',␣
 ↪metrics=['accuracy'])
history_l2 = model_l2.fit(X_train_scaled, y_train, validation_split=0.2,␣
 ↪epochs=150, batch_size=32, verbose=0)

# Visualisation pour la Régularisation L2
plt.plot(history_l2.history['loss'], label='Training loss (L2)')
plt.plot(history_l2.history['val_loss'], label='Validation loss (L2)')
plt.title('L2 Regularization: Training and Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```


L1 Regularization: Training and Validation Loss
Training loss (L1)
Validation loss (L1)
Loss
Epochs
0
1
2
3
4
0
20
40
60
80
100
120
140



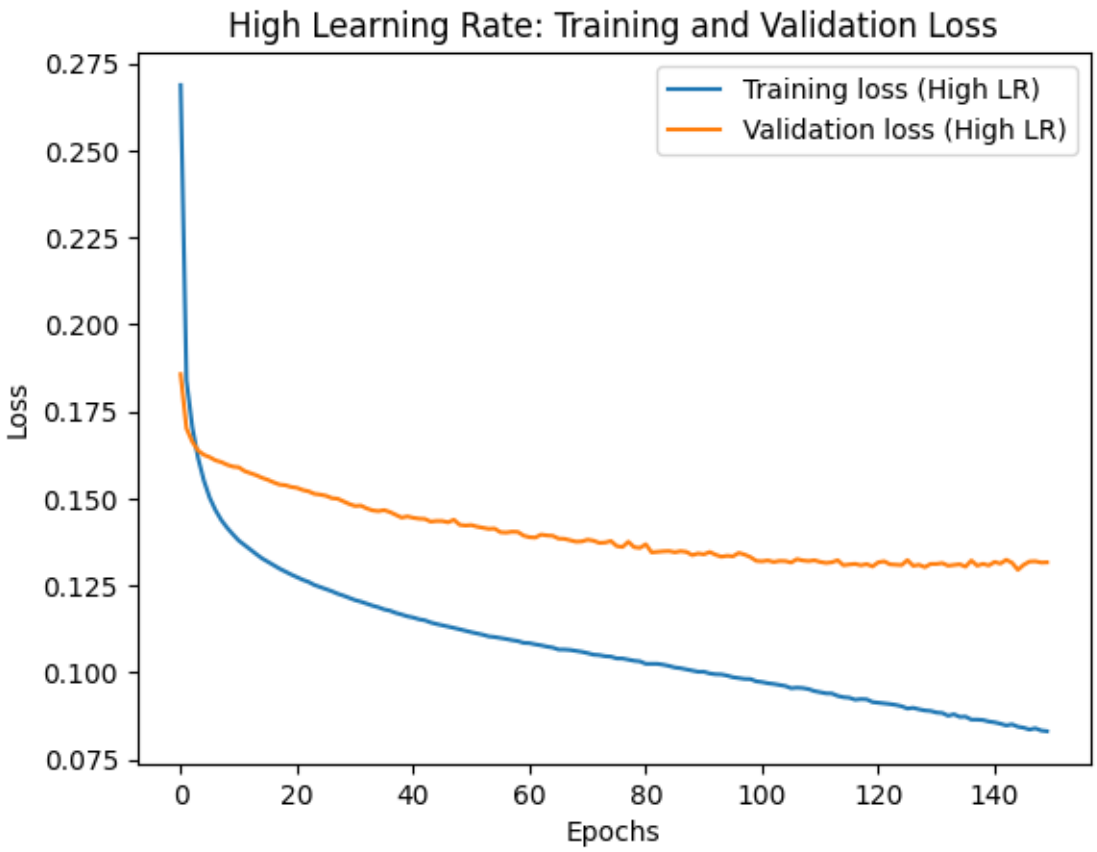
2.3 Différents Taux d'Apprentissage et Momentum pour le SGD

Ce code illustre l'effet du taux d'apprentissage sur l'entraînement des réseaux de neurones en utilisant l'optimiseur SGD avec différents taux d'apprentissage. D'une part, nous allons étudier la première courbe où le taux d'apprentissage est elevé. On peut voir que la perte d'entraînement diminue rapidement et de manière stable mais qu'elle nécessite plus d'époques pour converger. La perte de validation tend à suivre une trajectoire plus stable et à diminuer progressivement au fil des époques, elle reste proche de la perte d'entraînement. D'autre part, nous allons nous focaliser sur le cas où le taux d'apprentissage est faible. On peut voir que la perte d'entraînement diminue rapidement, ce qui peut indiquer une convergence plus rapide du modèle. La perte de validation s'éloigne considerablement de la perte d'entraînement durant les époques, elle augmente de manière instable. Cela peut laisser penser que dans cette situation le modèle surapprend les données.

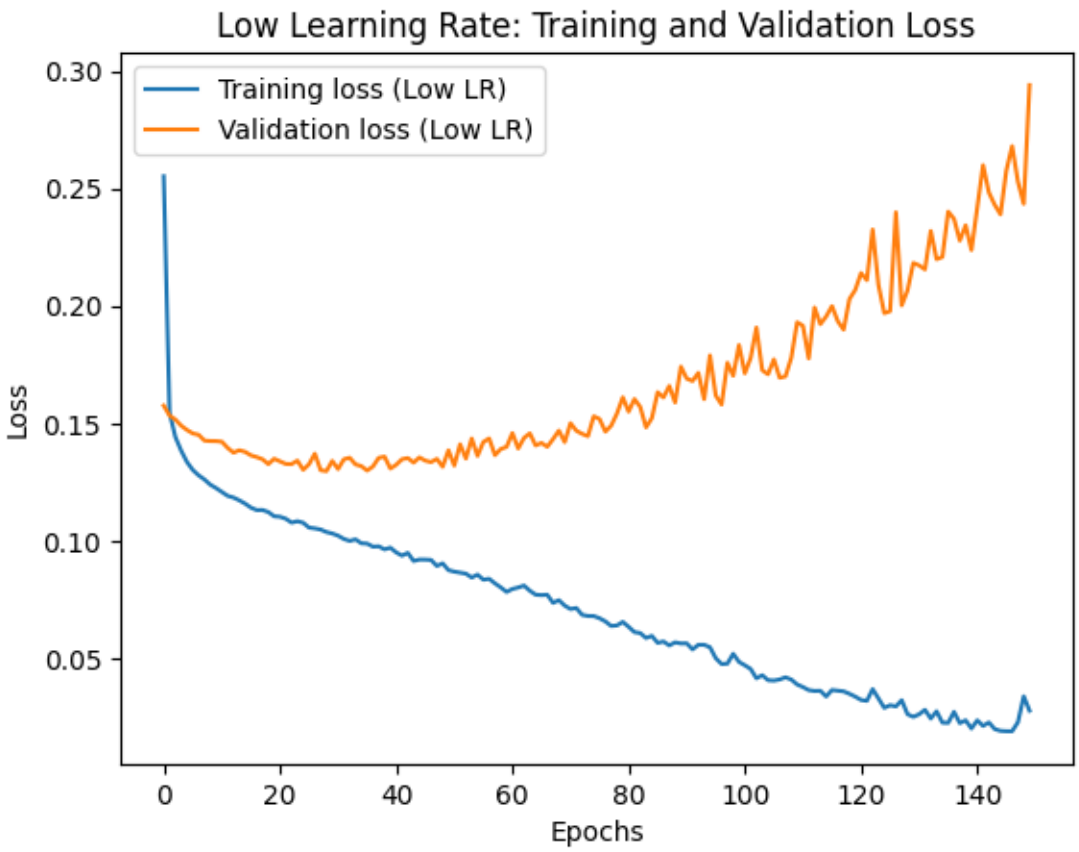
[ ]:
```
from tensorflow.keras import optimizers


# SGD avec un taux d'apprentissage élevé
sgd_high_lr = optimizers.SGD(lr=0.4, momentum=0.6)
model_high_lr = models.Sequential()
```

```
model_high_lr.add(layers.Dense(50, activation='relu',␣
 ↪input_shape=(X_train_scaled.shape[1],)))
model_high_lr.add(layers.Dense(50, activation='relu'))
model_high_lr.add(layers.Dense(1, activation='sigmoid'))
model_high_lr.compile(optimizer=sgd_high_lr, loss='binary_crossentropy',␣
 ↪metrics=['accuracy'])
history_high_lr = model_high_lr.fit(X_train_scaled, y_train, validation_split=0.
 ↪2, epochs=150, batch_size=32, verbose=0)

# Visualisation pour un taux d'apprentissage élevé
plt.plot(history_high_lr.history['loss'], label='Training loss (High LR)')
plt.plot(history_high_lr.history['val_loss'], label='Validation loss (High LR)')
plt.title('High Learning Rate: Training and Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()

# SGD avec un faible taux d'apprentissage
sgd_low_lr = optimizers.SGD(lr=0.1, momentum=0.9)
model_low_lr = models.Sequential()
model_low_lr.add(layers.Dense(50, activation='relu',␣
 ↪input_shape=(X_train_scaled.shape[1],)))
model_low_lr.add(layers.Dense(50, activation='relu'))
model_low_lr.add(layers.Dense(1, activation='sigmoid'))
model_low_lr.compile(optimizer=sgd_low_lr, loss='binary_crossentropy',␣
 ↪metrics=['accuracy'])
history_low_lr = model_low_lr.fit(X_train_scaled, y_train, validation_split=0.
 ↪2, epochs=150, batch_size=32, verbose=0)

# Visualisation pour un faible taux d'apprentissage
plt.plot(history_low_lr.history['loss'], label='Training loss (Low LR)')
plt.plot(history_low_lr.history['val_loss'], label='Validation loss (Low LR)')
plt.title('Low Learning Rate: Training and Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```

```
WARNING:absl:`lr` is deprecated in Keras optimizer, please use `learning_rate`
or use the legacy optimizer, e.g.,tf.keras.optimizers.legacy.SGD.
```




```
WARNING:absl:`lr` is deprecated in Keras optimizer, please use `learning_rate`
or use the legacy optimizer, e.g.,tf.keras.optimizers.legacy.SGD.
```


Low Learning Rate: Training and Validation Loss
Training loss (Low LR)
Validation loss (Low LR)
Loss
Epochs
0.30
0.25
0.20
0.15
0.10
0.05
0
20
40
60
80
100
120
140