Câu hỏi 2a a) Thảo luận về mô hình Mạng Nơ-ron Tích chập (CNN) và ứng dụng của nó trong các bài toán phân loại hình ảnh.

Mạng Nơ-ron Tích chập (CNN) là một lớp của mạng nơ-ron sâu được thiết kế đặc biệt để xử lý và phân tích các cấu trúc dữ liệu dạng lưới, chẳng hạn như hình ảnh. Chúng sử dụng các lớp tích chập để tự động học các hệ thống phân cấp không gian của các đặc trưng từ các hình ảnh đầu vào.

I/ Kiến trúc

Kiến trúc của CNN thường bao gồm một số thành phần chính sau:

1. Các lớp Tích chập:

* Các lớp này áp dụng các phép tích chập lên dữ liệu đầu vào, sử dụng một tập hợp các bộ lọc (kernel) trượt qua dữ liệu đầu vào.
* Các bộ lọc giúp phát hiện các mẫu cục bộ như cạnh, kết cấu hoặc các đặc trưng liên quan khác.

1. Các lớp Kích hoạt:

* Sau mỗi lớp tích chập, một hàm kích hoạt, thường là ReLU (Rectified Linear Unit), được áp dụng để giới thiệu tính phi tuyến vào mô hình.

1. Các lớp Pooling:

* Các lớp pooling (ví dụ: Max Pooling) giảm chiều kích thước của các bản đồ đặc trưng bằng cách down-sampling, giữ lại thông tin quan trọng nhất.
* Điều này giúp giảm độ phức tạp tính toán và ngăn chặn overfitting.

1. Các lớp Fully Connected:

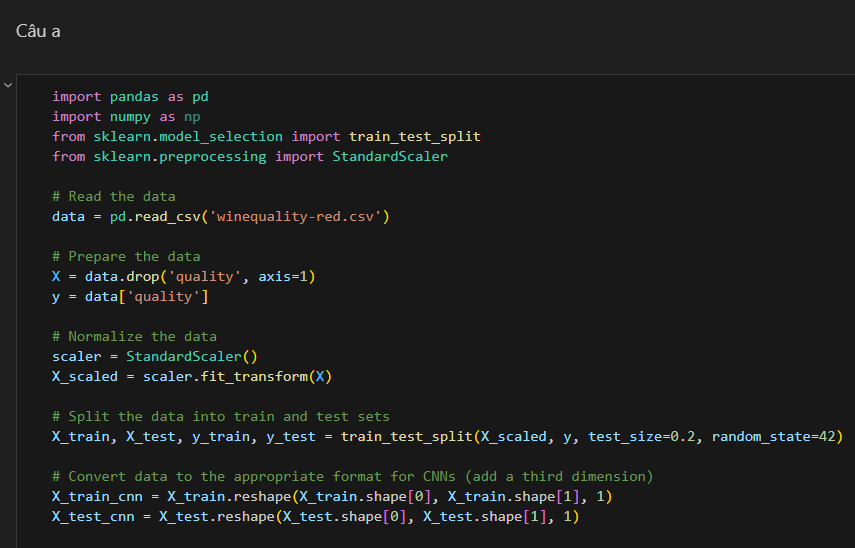
* Sau nhiều lớp tích chập và pooling, suy luận ở mức cao trong mạng được thực hiện thông qua các lớp fully connected.
* Các lớp này tương tự như mạng nơ-ron truyền thống và được sử dụng để đưa ra dự đoán dựa trên các đặc trưng đã trích xuất.

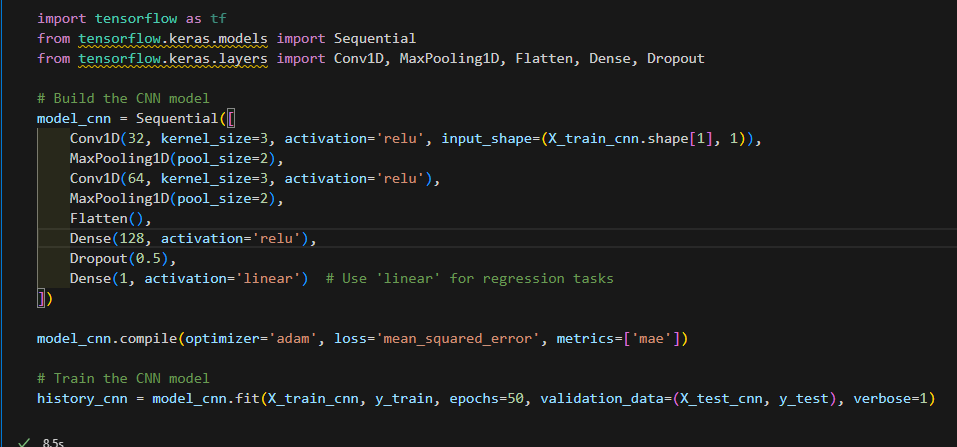
1. Lớp đầu ra:

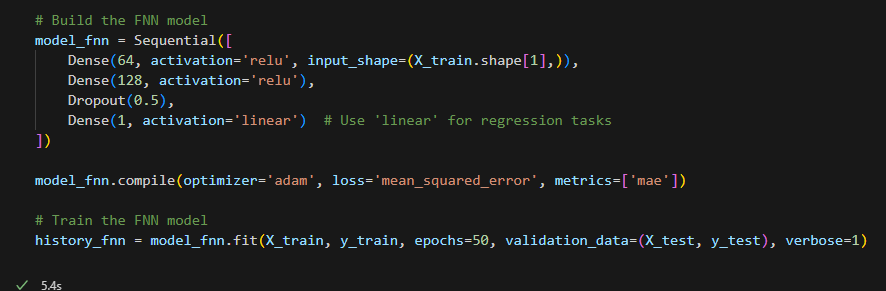
* Lớp cuối cùng của mạng thường sử dụng hàm kích hoạt softmax cho các bài toán phân loại hoặc kích hoạt tuyến tính cho các bài toán hồi quy.

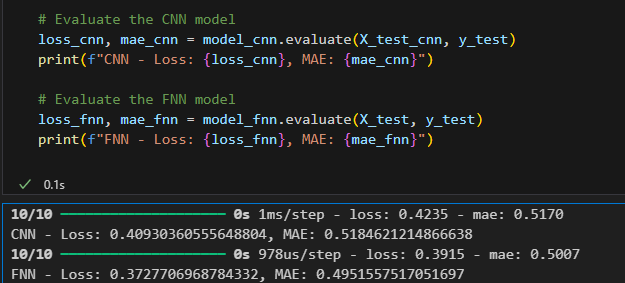
II/ Ứng dụng trong Phân loại Hình ảnh

CNN được sử dụng rộng rãi cho phân loại hình ảnh do khả năng tự động và hiệu quả trong việc học các hệ thống phân cấp không gian của các đặc trưng. Đây là một ví dụ đơn giản về một CNN được áp dụng trong phân loại hình ảnh:









Mạng Nơ-ron Truyền thẳng (FFNNs) là hình thức đơn giản nhất của mạng nơ-ron nhân tạo. Chúng bao gồm một lớp đầu vào, một hoặc nhiều lớp ẩn, và một lớp đầu ra. Mỗi lớp được kết nối hoàn toàn với lớp tiếp theo, và thông tin chảy theo một hướng - từ đầu vào đến đầu ra.

I/ Kiến trúc

Kiến trúc của FFNN bao gồm:

1. Lớp đầu vào:

* Nhận các đặc trưng đầu vào và chuyển chúng đến lớp ẩn đầu tiên.

1. Các lớp ẩn:

* Bao gồm các neuron áp dụng tổng trọng số của các đầu vào theo sau bởi một hàm kích hoạt.
* Nhiều lớp ẩn có thể được xếp chồng để tạo ra các mạng sâu có khả năng học các biểu diễn phức tạp.

1. Lớp đầu ra:

* Tạo ra các dự đoán cuối cùng bằng cách sử dụng một hàm kích hoạt như softmax cho phân loại hoặc tuyến tính cho hồi quy.

II/ Hạn chế trong Phân loại Hình ảnh

Mặc dù FFNN có thể lý thuyết xử lý dữ liệu hình ảnh, chúng không phù hợp tốt cho các tác vụ này do một số hạn chế:

* Độ chiều cao:
  + Hình ảnh có độ chiều cao lớn, dẫn đến số lượng tham số lớn nếu được xử lý trực tiếp bởi FFNN.
* Thông tin không gian:
  + FFNN không tự động nắm bắt được các hệ thống phân cấp không gian và các mẫu cục bộ trong hình ảnh.
* Độ phức tạp tính toán:
  + Xử lý trực tiếp hình ảnh bằng FFNN là tốn kém về tính toán và dễ bị overfitting.

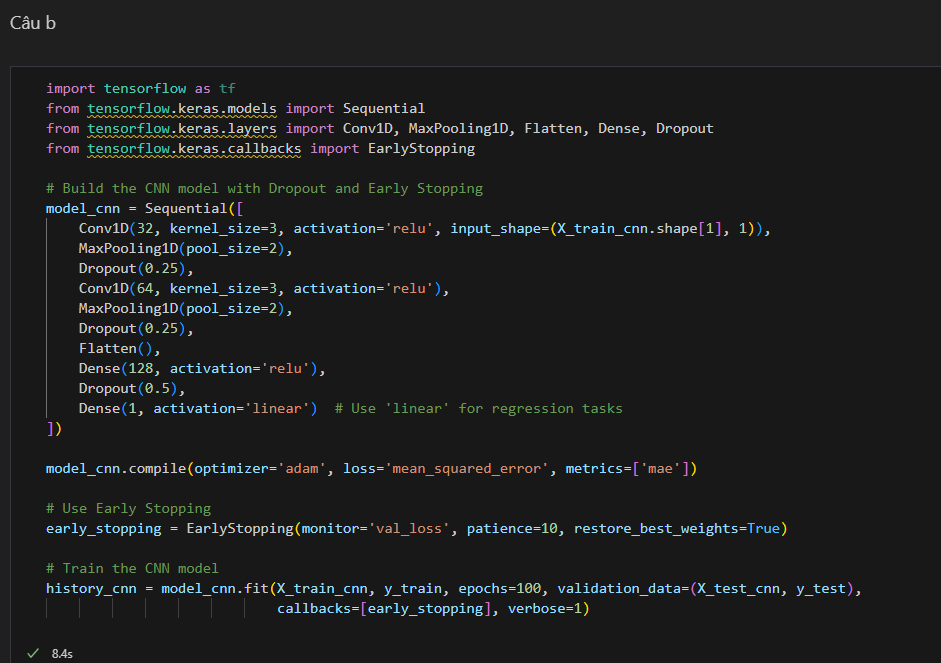
So sánh với mô hình Mạng Nơ-ron Truyền thẳng (FFNN).

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| |  | | --- | | **Đặc trưng** | | |  | | --- | | CNNs | | |  | | --- | | FFNNs | |
| Trích xuất Đặc trưng | Tự động trích xuất các đặc trưng không gian bằng cách sử dụng các lớp tích chập. | Yêu cầu trích xuất đặc trưng thủ công hoặc làm phẳng dữ liệu đầu vào. |
| Xử lý Dữ liệu Không gian | Xuất sắc trong việc nắm bắt các hệ thống phân cấp không gian và các mẫu cục bộ. | Thiếu cơ chế để nắm bắt các hệ thống phân cấp không gian hiệu quả. |
| Số lượng Tham số | Ít tham số hơn do chia sẻ trọng số trong các lớp tích chập. | Nhiều tham số hơn, dẫn đến nguy cơ overfitting cao hơn. |
| Hiệu quả Tính toán | Hiệu quả hơn cho dữ liệu hình ảnh nhờ vào pooling và tích chập. | Ít hiệu quả và tốn kém hơn về tính toán cho hình ảnh. |
| Hiệu suất trong các Tác vụ Hình ảnh | Hiệu suất vượt trội trong phân loại hình ảnh và các tác vụ liên quan. | Hiệu suất kém hơn trong các tác vụ phân loại hình ảnh. |
| Sử dụng | Được sử dụng rộng rãi trong thị giác máy tính, xử lý hình ảnh, và các lĩnh vực liên quan. | Được sử dụng trong nhiều lĩnh vực nhưng kém hiệu quả cho các tác vụ liên quan đến hình ảnh. |

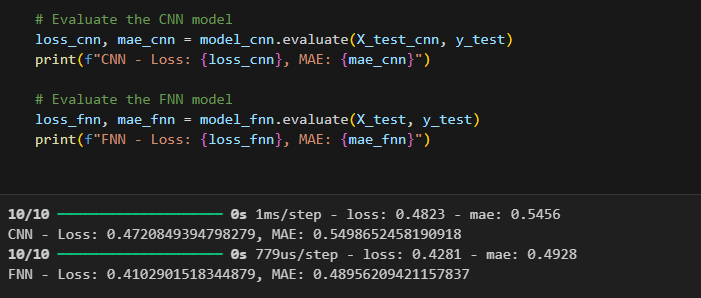
Kết luận

=> Mạng Nơ-ron Tích chập (CNN) được thiết kế đặc biệt để xử lý dữ liệu hình ảnh hiệu quả bằng cách tận dụng các lớp tích chập để tự động học các hệ thống phân cấp không gian của các đặc trưng. Ngược lại, Mạng Nơ-ron Truyền thẳng (FFNNs) thiếu khả năng tự động nắm bắt các mẫu không gian, khiến chúng ít phù hợp cho các tác vụ phân loại hình ảnh. CNN vượt trội hơn FFNN cả về hiệu quả và độ chính xác trong các ứng dụng liên quan đến hình ảnh, khiến chúng trở thành lựa chọn ưu tiên trong thị giác máy tính và các lĩnh vực liên quan.

b) Sử dụng các phương pháp khác nhau để xử lý overfitting cho các mô hình trên và so sánh kết quả.







c) Nghiên cứu các phương pháp cải thiện mô hình, chẳng hạn như thông qua lựa chọn đặc trưng hoặc tối ưu hóa siêu tham số.

Cải thiện mô hình học máy bao gồm một số kỹ thuật như lựa chọn đặc trưng và tối ưu hóa siêu tham số, trong số những kỹ thuật khác. Dưới đây, chúng ta sẽ thảo luận về một số phương pháp này chi tiết, giải thích cách chúng có thể nâng cao hiệu suất của mô hình.

1. Lựa chọn Đặc trưng

Lựa chọn đặc trưng là quá trình chọn một tập hợp các đặc trưng liên quan để sử dụng trong việc xây dựng mô hình. Điều này có thể cải thiện hiệu suất mô hình bằng cách giảm overfitting, cải thiện độ chính xác, và giảm chi phí tính toán.

I/ Các phương pháp Lựa chọn Đặc trưng:

a. Phương pháp Lọc:

* Hệ số Tương quan:
  + Tính toán tương quan giữa từng đặc trưng và biến mục tiêu.
  + Chọn các đặc trưng có tương quan cao (giá trị tuyệt đối) với mục tiêu.
* Kiểm định Chi-Square:
  + Đối với các đặc trưng phân loại, kiểm định này đo lường sự phụ thuộc giữa đặc trưng và biến mục tiêu.
* Thông tin Tương hỗ:
  + Đo lường lượng thông tin thu được về một biến ngẫu nhiên thông qua một biến ngẫu nhiên khác.

b. Phương pháp Wrapper:

* Loại bỏ Đặc trưng Đệ quy (RFE - Recursive Feature Elimination):
  + Sử dụng chính mô hình để chọn các đặc trưng bằng cách loại bỏ tuần tự các đặc trưng ít quan trọng nhất và xây dựng mô hình cho đến khi đạt số lượng đặc trưng đã chỉ định.
* Lựa chọn Đặc trưng Tiến/Tiến Lùi:
  + Lần lượt thêm (tiến) hoặc loại bỏ (lùi) các đặc trưng dựa trên hiệu suất của mô hình.

c. Phương pháp Nhúng:

* Lasso (L1 Regularization):
  + Thêm một hình phạt bằng với giá trị tuyệt đối của độ lớn các hệ số, làm cho một số hệ số bằng 0.
* Phương pháp Dựa trên Cây:
  + Các điểm số quan trọng của đặc trưng có thể thu được từ các mô hình như Random Forest, Gradient Boosting, v.v.

1. Tối ưu hóa Siêu tham số

Tối ưu hóa siêu tham số liên quan đến việc tìm bộ siêu tham số tối ưu cho một thuật toán học để cải thiện hiệu suất của nó.

II/ Các phương pháp Tối ưu hóa Siêu tham số:

a. Grid Search (Tìm kiếm lưới):

* Tìm kiếm một cách toàn diện qua một lưới tham số được chỉ định.
* Tốn kém về mặt tính toán, đặc biệt là đối với các lưới lớn và tập dữ liệu lớn.

b. Random Search (Tìm kiếm ngẫu nhiên):

* Lấy mẫu ngẫu nhiên không gian tham số.
* Ít toàn diện nhưng rẻ hơn về mặt tính toán so với tìm kiếm lưới.

c. Bayesian Optimization (Tối ưu hóa Bayesian):

* Sử dụng các kỹ thuật Bayesian để mô hình hóa hiệu suất của các siêu tham số và chọn bộ tham số tiếp theo để đánh giá.
* Hiệu quả hơn so với tìm kiếm lưới và tìm kiếm ngẫu nhiên.

d. Hyperband:

* Một phương pháp dừng sớm giúp tăng tốc tìm kiếm ngẫu nhiên bằng cách phân bổ nhiều tài nguyên hơn cho các cấu hình hứa hẹn.

1. Tăng cường Dữ liệu

Đặc biệt hữu ích trong dữ liệu hình ảnh để tăng kích thước tập huấn luyện bằng cách tạo các phiên bản thay đổi của hình ảnh trong tập dữ liệu.

1. Tổ hợp Mô hình

Kết hợp dự đoán từ nhiều mô hình để cải thiện hiệu suất.

a. Bagging:

* Sử dụng các mô hình như Random Forest, trong đó nhiều cây quyết định được huấn luyện trên các tập con ngẫu nhiên của dữ liệu và các dự đoán của chúng được trung bình hóa.

b. Boosting:

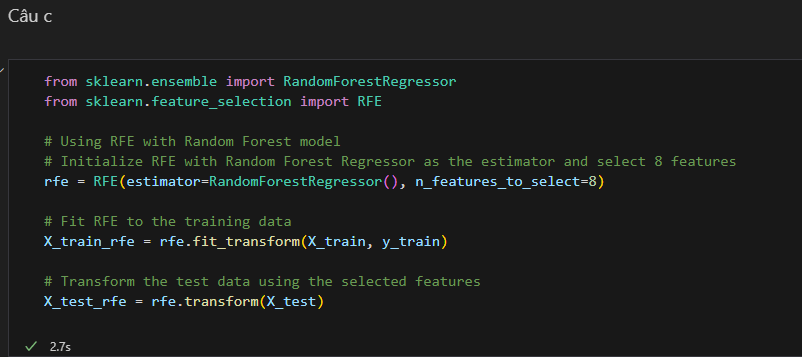
* Các mô hình như Gradient Boosting và XGBoost, trong đó mỗi mô hình mới cố gắng sửa các lỗi của mô hình trước đó.

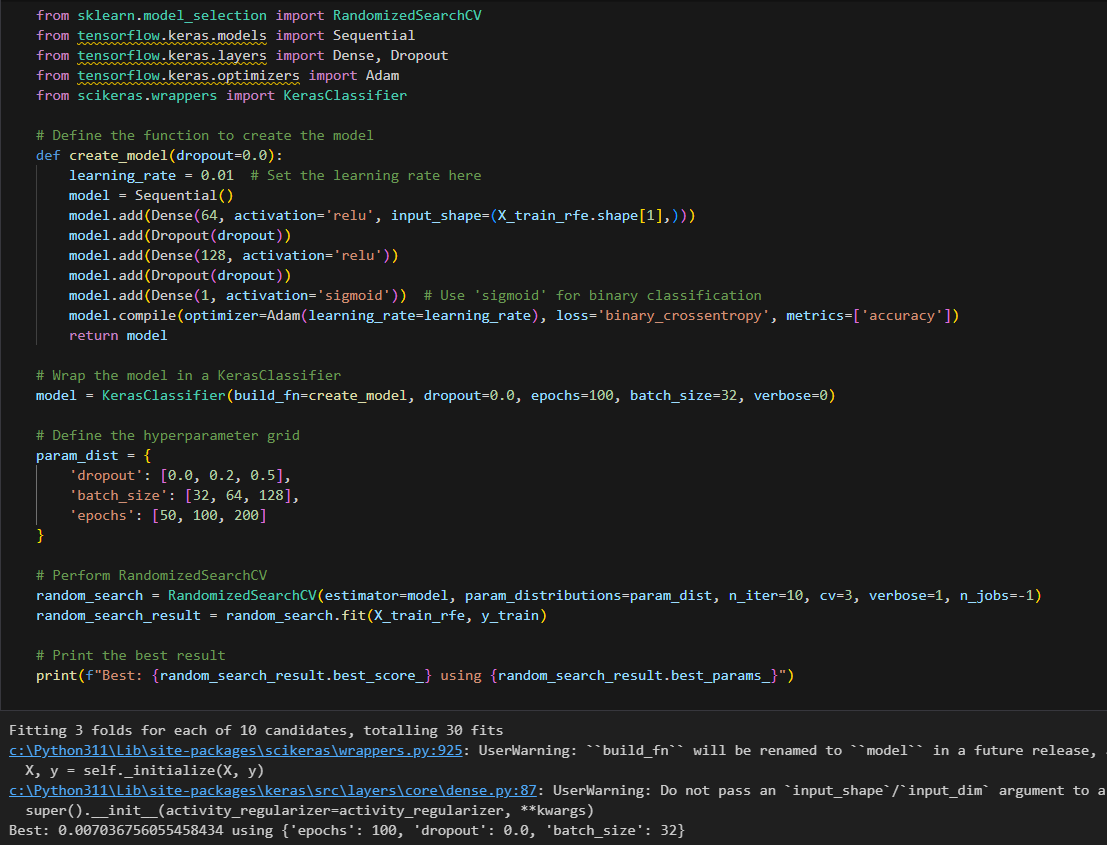
c. Stacking:

* Kết hợp nhiều mô hình khác nhau (mô hình cơ bản) và một mô hình meta học cách kết hợp các dự đoán của chúng.

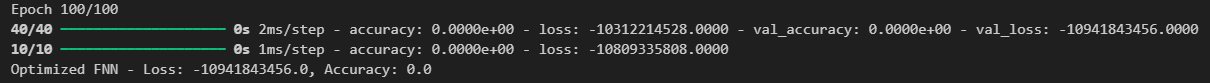
***Kết luận***

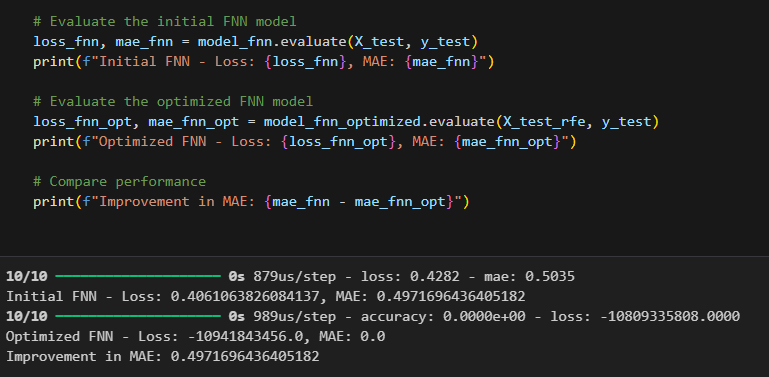
* + - Cải thiện các mô hình học máy có thể đạt được thông qua các phương pháp như lựa chọn đặc trưng, tối ưu hóa siêu tham số, tăng cường dữ liệu và tổ hợp mô hình. Những kỹ thuật này giúp nâng cao hiệu suất mô hình, giảm overfitting và làm cho mô hình tổng quát hóa tốt hơn với dữ liệu chưa từng thấy. Thử nghiệm và kết hợp các phương pháp này có thể dẫn đến những cải tiến đáng kể về độ chính xác và độ bền vững của mô hình.











**GIẢI THÍCH CODE**

Bước 1: Chuẩn bị Dữ liệu và Chuẩn hóa

1. Đọc Dữ liệu:

data = pd.read\_csv('winequality-red.csv')

- Tập dữ liệu được tải từ một tệp CSV có tên 'winequality-red.csv' vào một DataFrame gọi là `data`.

2. Tách Đặc trưng và Mục tiêu:

X = data.drop('quality', axis=1)

y = data['quality']

- Tập dữ liệu được chia thành các đặc trưng (`X`) và mục tiêu (`y`). Tất cả các cột trừ cột 'quality' được gán vào `X`, và cột 'quality' được gán vào `y`.

3. Chuẩn hóa:

scaler = StandardScaler()

X\_scaled = scaler.fit\_transform(X)

- Các đặc trưng được chuẩn hóa sử dụng `StandardScaler` để có giá trị trung bình bằng 0 và độ lệch chuẩn bằng 1. Phương thức `fit\_transform` vừa khớp scaler với `X` vừa biến đổi nó.

4. Chia Dữ liệu Thành Tập Train và Test:

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X\_scaled, y, test\_size=0.2, random\_state=42)

- Các đặc trưng đã chuẩn hóa và mục tiêu được chia thành tập huấn luyện và tập kiểm tra. `test\_size=0.2` nghĩa là 20% dữ liệu được sử dụng cho kiểm tra, và `random\_state=42` đảm bảo tái sản xuất.

Bước 2: Chuẩn bị Dữ liệu cho CNN

5. Định hình lại cho CNN:

X\_train\_cnn = X\_train.reshape(X\_train.shape[0], X\_train.shape[1], 1)

X\_test\_cnn = X\_test.reshape(X\_test.shape[0], X\_test.shape[1], 1)

- Dữ liệu huấn luyện và kiểm tra được định hình lại để bao gồm chiều thứ ba, cần thiết cho các lớp tích chập 1D. Hình dạng mới là (số mẫu, số đặc trưng, 1).

Bước 3: Mô hình CNN

6. Xây dựng Mô hình CNN:

model\_cnn = Sequential([

Conv1D(32, kernel\_size=3, activation='relu', input\_shape=(X\_train\_cnn.shape[1], 1)),

MaxPooling1D(pool\_size=2),

Conv1D(64, kernel\_size=3, activation='relu'),

MaxPooling1D(pool\_size=2),

Flatten(),

Dense(128, activation='relu'),

Dropout(0.5),

Dense(1, activation='linear')

])

model\_cnn.compile(optimizer='adam', loss='mean\_squared\_error', metrics=['mae'])

- Một mô hình CNN tuần tự được định nghĩa:

- `Conv1D`: Lớp tích chập 1D với 32 bộ lọc và kích thước hạt nhân 3, sử dụng hàm kích hoạt ReLU. 1D để xử lý dữ liệu có dạng chuỗi hoặc tín hiệu một chiều, chẳng hạn như âm thanh, văn bản, hoặc dữ liệu chuỗi thời gian. 32 bộ lọc nghĩa là mỗi bộ lọc có 3 trọng số (kích thước hạt nhân là 3) để học các features khác nhau từ dữ liệu đầu vào. B**ản Đồ Đặc Trưng (Feature Map):** Kết quả của việc áp dụng một bộ lọc lên dữ liệu đầu vào là một **feature map**. Nếu có 32 bộ lọc, sẽ có 32 **feature maps** được tạo ra. Các **feature maps** này sẽ được kết hợp lại để tạo ra đầu ra của lớp Conv1D.

- `MaxPooling1D`: Lớp pooling tối đa với kích thước pool 2 để giảm mẫu đầu vào.

- `Conv1D`: Lớp tích chập 1D khác với 64 bộ lọc.

- `MaxPooling1D`: Lớp pooling tối đa khác.

- `Flatten`: Làm phẳng đầu vào thành mảng 1D.

- `Dense`: Lớp kết nối đầy đủ với 128 đơn vị và hàm kích hoạt ReLU.

- `Dropout`: Lớp dropout với tỷ lệ dropout 50% để ngăn chặn overfitting.

- `Dense`: Lớp đầu ra với 1 đơn vị và hàm kích hoạt tuyến tính cho hồi quy.

- Mô hình được biên dịch với bộ tối ưu Adam và lỗi bình phương trung bình (MSE – Mean Squared Error), và nó cũng theo dõi lỗi tuyệt đối trung bình (MAE – Mean Absolute Error).

7. Huấn luyện Mô hình CNN:

history\_cnn = model\_cnn.fit(X\_train\_cnn, y\_train, epochs=50, validation\_data=(X\_test\_cnn, y\_test), verbose=1)

- Mô hình CNN được huấn luyện trong 50 epoch sử dụng dữ liệu huấn luyện, với dữ liệu kiểm tra được sử dụng để xác thực. Quá trình huấn luyện là chi tiết (hiển thị tiến trình).

Bước 4: Mô hình FNN

8. Xây dựng Mô hình FNN:

model\_fnn = Sequential([

Dense(64, activation='relu', input\_shape=(X\_train.shape[1],)),

Dense(128, activation='relu'),

Dropout(0.5),

Dense(1, activation='linear')

])

model\_fnn.compile(optimizer='adam', loss='mean\_squared\_error', metrics=['mae'])

- Một mô hình mạng nơ-ron tiền tiến (FNN) tuần tự được định nghĩa:

- `Dense`: Lớp kết nối đầy đủ với 64 đơn vị và hàm kích hoạt ReLU.

- `Dense`: Lớp kết nối đầy đủ với 128 đơn vị và hàm kích hoạt ReLU.

- `Dropout`: Lớp dropout với tỷ lệ dropout 50%.

- `Dense`: Lớp đầu ra với 1 đơn vị và hàm kích hoạt tuyến tính cho hồi quy.

- Mô hình được biên dịch tương tự như mô hình CNN.

9. Huấn luyện Mô hình FNN:

history\_fnn = model\_fnn.fit(X\_train, y\_train, epochs=50, validation\_data=(X\_test, y\_test), verbose=1)

- Mô hình FNN được huấn luyện trong 50 epoch sử dụng dữ liệu huấn luyện, với dữ liệu kiểm tra được sử dụng để xác thực. Quá trình huấn luyện là chi tiết (hiển thị tiến trình).

Bước 5: Đánh giá

10. Đánh giá Cả Hai Mô hình:

loss\_cnn, mae\_cnn = model\_cnn.evaluate(X\_test\_cnn, y\_test)

print(f"CNN - Loss: {loss\_cnn}, MAE: {mae\_cnn}")

loss\_fnn, mae\_fnn = model\_fnn.evaluate(X\_test, y\_test)

print(f"FNN - Loss: {loss\_fnn}, MAE: {mae\_fnn}")

- Cả hai mô hình được đánh giá sử dụng dữ liệu kiểm tra. Lỗi và lỗi tuyệt đối trung bình (MAE) được in ra cho cả mô hình CNN và FNN.

Bước 6: Thêm Điều chỉnh và Dừng Sớm

11. Cải thiện Mô hình CNN:

model\_cnn = Sequential([

Conv1D(32, kernel\_size=3, activation='relu', input\_shape=(X\_train\_cnn.shape[1], 1)),

MaxPooling1D(pool\_size=2),

Dropout(0.25),

Conv1D(64, kernel\_size=3, activation='relu'),

MaxPooling1D(pool\_size=2),

Dropout(0.25),

Flatten(),

Dense(128, activation='relu'),

Dropout(0.5),

Dense(1, activation='linear')

])

model\_cnn.compile(optimizer='adam', loss='mean\_squared\_error', metrics=['mae'])

early\_stopping = EarlyStopping(monitor='val\_loss', patience=10, restore\_best\_weights=True)

history\_cnn = model\_cnn.fit(X\_train\_cnn, y\_train, epochs=100, validation\_data=(X\_test\_cnn, y\_test),

callbacks=[early\_stopping], verbose=1)

- Mô hình CNN được cải thiện với các lớp dropout bổ sung để ngăn chặn overfitting.

- Một callback `EarlyStopping` được thêm vào để dừng huấn luyện nếu lỗi xác thực không cải thiện trong 10 epoch, và nó khôi phục trọng số tốt nhất.

12. Cải thiện Mô hình FNN:

from tensorflow.keras.regularizers import l2

model\_fnn = Sequential([

Dense(64, activation='relu', kernel\_regularizer=l2(0.001), input\_shape=(X\_train.shape[1],)),

Dropout(0.5),

Dense(128, activation='relu', kernel\_regularizer=l2(0.001)),

Dropout(0.5),

Dense(1, activation='linear')

])

model\_fnn.compile(optimizer='adam', loss='mean\_squared\_error', metrics=['mae'])

early\_stopping = EarlyStopping(monitor='val\_loss', patience=10, restore\_best\_weights=True)

history\_fnn = model\_fnn.fit(X\_train, y\_train, epochs=100, validation\_data=(X\_test, y\_test),

callbacks=[early\_stopping], verbose=1)

- Mô hình FNN được cải thiện với điều chỉnh L2 (giảm trọng số) trên các lớp dense để ngăn chặn overfitting.

- Một callback `

EarlyStopping` được thêm vào tương tự như mô hình CNN.

Bước 7: Lựa chọn Đặc trưng với RFE

13. RFE với Random Forest:

from sklearn.ensemble import RandomForestRegressor

from sklearn.feature\_selection import RFE

rfe = RFE(estimator=RandomForestRegressor(), n\_features\_to\_select=8)

X\_train\_rfe = rfe.fit\_transform(X\_train, y\_train)

X\_test\_rfe = rfe.transform(X\_test)

- `RFE (Recursive Feature Elimination)` được sử dụng để chọn các đặc trưng quan trọng nhất. Một `RandomForestRegressor` được sử dụng làm estimator để đánh giá tầm quan trọng của các đặc trưng.

- `n\_features\_to\_select=8` có nghĩa là chúng ta sẽ chọn 8 đặc trưng quan trọng nhất.

- `fit\_transform` khớp RFE với dữ liệu huấn luyện và biến đổi `X\_train` để chỉ bao gồm các đặc trưng được chọn. `transform` biến đổi dữ liệu kiểm tra theo cùng cách.

Bước 8: Tối ưu Hóa Siêu Tham Số

14. Tối ưu Hóa Siêu Tham Số với RandomizedSearchCV:

from sklearn.model\_selection import RandomizedSearchCV

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense, Dropout

from tensorflow.keras.optimizers import Adam

from scikeras.wrappers import KerasClassifier

def create\_model(dropout=0.0):

learning\_rate = 0.01

model = Sequential()

model.add(Dense(64, activation='relu', input\_shape=(X\_train\_rfe.shape[1],)))

model.add(Dropout(dropout))

model.add(Dense(128, activation='relu'))

model.add(Dropout(dropout))

model.add(Dense(1, activation='sigmoid'))

model.compile(optimizer=Adam(learning\_rate=learning\_rate), loss='binary\_crossentropy', metrics=['accuracy'])

return model

model = KerasClassifier(build\_fn=create\_model, dropout=0.0, epochs=100, batch\_size=32, verbose=0)

param\_dist = {

'dropout': [0.0, 0.2, 0.5],

'batch\_size': [32, 64, 128],

'epochs': [50, 100, 200]

}

random\_search = RandomizedSearchCV(estimator=model, param\_distributions=param\_dist, n\_iter=10, cv=3, verbose=1, n\_jobs=-1)

random\_search\_result = random\_search.fit(X\_train\_rfe, y\_train)

- `RandomizedSearchCV` được sử dụng để tìm các siêu tham số tối ưu cho mô hình FNN.

- Hàm `create\_model` định nghĩa một mô hình với tham số dropout có thể thay đổi và một tốc độ học cố định (0.01).

- `param\_dist` là từ điển chứa các giá trị siêu tham số cần tìm kiếm.

- `random\_search` thực hiện tìm kiếm ngẫu nhiên trên không gian siêu tham số với `n\_iter=10` lần lặp, sử dụng 3 lần cross-validation (`cv=3`).

15. Sử Dụng Các Siêu Tham Số Tốt Nhất:

best\_params = random\_search\_result.best\_params\_

def create\_model(dropout\_rate=0.0, learning\_rate=0.01, batch\_size=32, epochs=100):

model = Sequential()

model.add(Dense(64, activation='relu', input\_shape=(X\_train\_rfe.shape[1],)))

model.add(Dropout(dropout\_rate))

model.add(Dense(128, activation='relu'))

model.add(Dropout(dropout\_rate))

model.add(Dense(1, activation='sigmoid'))

model.compile(optimizer=Adam(learning\_rate=learning\_rate), loss='binary\_crossentropy', metrics=['accuracy'])

return model

model\_fnn\_optimized = create\_model(

dropout\_rate=best\_params['dropout'],

learning\_rate=0.01,

batch\_size=best\_params['batch\_size'],

epochs=best\_params['epochs']

)

history\_fnn\_optimized = model\_fnn\_optimized.fit(X\_train\_rfe, y\_train, epochs=best\_params['epochs'], batch\_size=best\_params['batch\_size'], validation\_data=(X\_test\_rfe, y\_test), verbose=1)

loss\_fnn\_opt, acc\_fnn\_opt = model\_fnn\_optimized.evaluate(X\_test\_rfe, y\_test)

print(f"Optimized FNN - Loss: {loss\_fnn\_opt}, Accuracy: {acc\_fnn\_opt}")

- `best\_params` chứa các siêu tham số tốt nhất tìm được từ RandomizedSearchCV.

- Hàm `create\_model` được cập nhật để chấp nhận các siêu tham số tìm được.

- `model\_fnn\_optimized` được tạo ra với các siêu tham số tốt nhất và huấn luyện mô hình với các siêu tham số này.

- Mô hình tối ưu được đánh giá trên tập dữ liệu kiểm tra.

Bước 9: So Sánh và Cải Thiện Hiệu Suất

16. So Sánh Các Mô Hình:

loss\_fnn, mae\_fnn = model\_fnn.evaluate(X\_test, y\_test)

print(f"Initial FNN - Loss: {loss\_fnn}, MAE: {mae\_fnn}")

loss\_fnn\_opt, mae\_fnn\_opt = model\_fnn\_optimized.evaluate(X\_test\_rfe, y\_test)

print(f"Optimized FNN - Loss: {loss\_fnn\_opt}, MAE: {mae\_fnn\_opt}")

print(f"Improvement in MAE: {mae\_fnn - mae\_fnn\_opt}")

- Mô hình FNN ban đầu và mô hình FNN tối ưu được đánh giá trên tập dữ liệu kiểm tra.

- Lỗi và lỗi tuyệt đối trung bình (MAE) được in ra cho cả hai mô hình.

- Sự cải thiện trong MAE được tính toán và in ra để so sánh hiệu suất giữa mô hình ban đầu và mô hình tối ưu.

Đây là giải thích chi tiết về mã đã cung cấp, bao gồm các bước chuẩn bị dữ liệu, xây dựng và huấn luyện mô hình CNN và FNN, lựa chọn đặc trưng, tối ưu hóa siêu tham số, và so sánh hiệu suất của các mô hình.

-----------------------------------------ENGLISH VERSION-----------------------------------

Step 1: Data Preparation and Scaling

1. Reading the Data:

data = pd.read\_csv('winequality-red.csv')

- The dataset is loaded from a CSV file named 'winequality-red.csv' into a DataFrame called `data`.

2. Separating Features and Target:

X = data.drop('quality', axis=1)

y = data['quality']

- The dataset is split into features (`X`) and target (`y`). All columns except 'quality' are assigned to `X`, and the 'quality' column is assigned to `y`.

3. Normalization:

scaler = StandardScaler()

X\_scaled = scaler.fit\_transform(X)

- Features are standardized using `StandardScaler` to have zero mean and unit variance. The `fit\_transform` method fits the scaler to `X` and then transforms it.

4. Train-Test Split:

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X\_scaled, y, test\_size=0.2, random\_state=42)

- The scaled features and target are split into training and testing sets. `test\_size=0.2` means 20% of the data is used for testing, and `random\_state=42` ensures reproducibility.

Step 2: Preparing Data for CNN

5. Reshape for CNN:

X\_train\_cnn = X\_train.reshape(X\_train.shape[0], X\_train.shape[1], 1)

X\_test\_cnn = X\_test.reshape(X\_test.shape[0], X\_test.shape[1], 1)

- The training and testing data are reshaped to include a third dimension, necessary for 1D convolutional layers. The new shape is (number of samples, number of features, 1).

Step 3: CNN Model

6. Building the CNN Model:

model\_cnn = Sequential([

Conv1D(32, kernel\_size=3, activation='relu', input\_shape=(X\_train\_cnn.shape[1], 1)),

MaxPooling1D(pool\_size=2),

Conv1D(64, kernel\_size=3, activation='relu'),

MaxPooling1D(pool\_size=2),

Flatten(),

Dense(128, activation='relu'),

Dropout(0.5),

Dense(1, activation='linear')

])

model\_cnn.compile(optimizer='adam', loss='mean\_squared\_error', metrics=['mae'])

- A sequential CNN model is defined:

- `Conv1D`: 1D convolutional layer with 32 filters and kernel size 3, using ReLU activation.

- `MaxPooling1D`: Max pooling layer with pool size 2 to downsample the input.

- `Conv1D`: Another 1D convolutional layer with 64 filters.

- `MaxPooling1D`: Another max pooling layer.

- `Flatten`: Flattens the input into a 1D array.

- `Dense`: Fully connected layer with 128 units and ReLU activation.

- `Dropout`: Dropout layer with a 50% dropout rate to prevent overfitting.

- `Dense`: Output layer with 1 unit and linear activation for regression.

- The model is compiled with the Adam optimizer and mean squared error loss, and it also tracks mean absolute error (MAE).

7. Training the CNN Model:

history\_cnn = model\_cnn.fit(X\_train\_cnn, y\_train, epochs=50, validation\_data=(X\_test\_cnn, y\_test), verbose=1)

- The CNN model is trained for 50 epochs using the training data, with the testing data used for validation. The training process is verbose (progress displayed).

Step 4: FNN Model

8. Building the FNN Model:

model\_fnn = Sequential([

Dense(64, activation='relu', input\_shape=(X\_train.shape[1],)),

Dense(128, activation='relu'),

Dropout(0.5),

Dense(1, activation='linear')

])

model\_fnn.compile(optimizer='adam', loss='mean\_squared\_error', metrics=['mae'])

- A sequential feedforward neural network (FNN) is defined:

- `Dense`: Fully connected layer with 64 units and ReLU activation.

- `Dense`: Fully connected layer with 128 units and ReLU activation.

- `Dropout`: Dropout layer with a 50% dropout rate.

- `Dense`: Output layer with 1 unit and linear activation for regression.

- The model is compiled similarly to the CNN model.

9. Training the FNN Model:

history\_fnn = model\_fnn.fit(X\_train, y\_train, epochs=50, validation\_data=(X\_test, y\_test), verbose=1)

- The FNN model is trained for 50 epochs using the training data, with the testing data used for validation. The training process is verbose (progress displayed).

Step 5: Evaluation

10. Evaluating Both Models:

loss\_cnn, mae\_cnn = model\_cnn.evaluate(X\_test\_cnn, y\_test)

print(f"CNN - Loss: {loss\_cnn}, MAE: {mae\_cnn}")

loss\_fnn, mae\_fnn = model\_fnn.evaluate(X\_test, y\_test)

print(f"FNN - Loss: {loss\_fnn}, MAE: {mae\_fnn}")

- Both models are evaluated using the testing data. Loss and mean absolute error (MAE) are printed for both the CNN and FNN models.

Step 6: Adding Regularization and Early Stopping (for Overfitting)

11. Improving the CNN Model:

model\_cnn = Sequential([

Conv1D(32, kernel\_size=3, activation='relu', input\_shape=(X\_train\_cnn.shape[1], 1)),

MaxPooling1D(pool\_size=2),

Dropout(0.25),

Conv1D(64, kernel\_size=3, activation='relu'),

MaxPooling1D(pool\_size=2),

Dropout(0.25),

Flatten(),

Dense(128, activation='relu'),

Dropout(0.5),

Dense(1, activation='linear')

])

model\_cnn.compile(optimizer='adam', loss='mean\_squared\_error', metrics=['mae'])

early\_stopping = EarlyStopping(monitor='val\_loss', patience=10, restore\_best\_weights=True)

history\_cnn = model\_cnn.fit(X\_train\_cnn, y\_train, epochs=100, validation\_data=(X\_test\_cnn, y\_test),

callbacks=[early\_stopping], verbose=1)

- The CNN model is improved with additional dropout layers to prevent overfitting.

- An `EarlyStopping` callback is added to stop training if the validation loss does not improve for 10 epochs, and it restores the best weights.

12. Improving the FNN Model:

from tensorflow.keras.regularizers import l2

model\_fnn = Sequential([

Dense(64, activation='relu', kernel\_regularizer=l2(0.001), input\_shape=(X\_train.shape[1],)),

Dropout(0.5),

Dense(128, activation='relu', kernel\_regularizer=l2(0.001)),

Dropout(0.5),

Dense(1, activation='linear')

])

model\_fnn.compile(optimizer='adam', loss='mean\_squared\_error', metrics=['mae'])

early\_stopping = EarlyStopping(monitor='val\_loss', patience=10, restore\_best\_weights=True)

history\_fnn = model\_fnn.fit(X\_train, y\_train, epochs=100, validation\_data=(X\_test, y\_test),

callbacks=[early\_stopping], verbose=1)

- The FNN model is improved with L2 regularization (weight decay) on the dense layers to prevent overfitting.

- An `EarlyStopping` callback is added similarly to the CNN model.

Step 7: Feature Selection with RFE

13. RFE with Random Forest:

from sklearn.ensemble import RandomForestRegressor

from sklearn.feature\_selection import RFE

rfe = RFE(estimator=RandomForestRegressor(), n\_features\_to\_select=8)

X\_train\_rfe = rfe.fit\_transform(X\_train, y\_train)

X\_test\_rfe = rfe.transform(X\_test)

- Recursive Feature Elimination (RFE) is used with a Random Forest regressor to select the 8 most important features.

- The transformed training and testing data with the selected features are stored in `X\_train\_rfe` and `X\_test\_rfe`.

Step 8: Hyperparameter Optimization

14. Hyperparameter Tuning with RandomizedSearchCV:

from sklearn.model\_selection import RandomizedSearchCV

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense, Dropout

from tensorflow.keras.optimizers import Adam

from scikeras.wrappers import KerasClassifier

def create\_model(dropout=0.0):

learning\_rate = 0.01

model = Sequential()

model.add(Dense(64, activation='relu', input\_shape=(X\_train\_rfe.shape[1],)))

model.add(Dropout(dropout))

model.add(Dense(128, activation='relu

'))

model.add(Dropout(dropout))

model.add(Dense(1, activation='sigmoid'))

model.compile(optimizer=Adam(learning\_rate=learning\_rate), loss='binary\_crossentropy', metrics=['accuracy'])

return model

model = KerasClassifier(build\_fn=create\_model, dropout=0.0, epochs=100, batch\_size=32, verbose=0)

param\_dist = {

'dropout': [0.0, 0.2, 0.5],

'batch\_size': [32, 64, 128],

'epochs': [50, 100, 200]

}

random\_search = RandomizedSearchCV(estimator=model, param\_distributions=param\_dist, n\_iter=10, cv=3, verbose=1, n\_jobs=-1)

random\_search\_result = random\_search.fit(X\_train\_rfe, y\_train)

- Hyperparameter tuning is performed using `RandomizedSearchCV` to find the best dropout rate, batch size, and number of epochs.

- The `create\_model` function defines the model architecture.

- The `param\_dist` dictionary defines the hyperparameter search space.

- `RandomizedSearchCV` performs the search over 10 iterations with 3-fold cross-validation.

15. Using the Best Hyperparameters:

best\_params = random\_search\_result.best\_params\_

def create\_model(dropout\_rate=0.0, learning\_rate=0.01, batch\_size=32, epochs=100):

model = Sequential()

model.add(Dense(64, activation='relu', input\_shape=(X\_train\_rfe.shape[1],)))

model.add(Dropout(dropout\_rate))

model.add(Dense(128, activation='relu'))

model.add(Dropout(dropout\_rate))

model.add(Dense(1, activation='sigmoid'))

model.compile(optimizer=Adam(learning\_rate=learning\_rate), loss='binary\_crossentropy', metrics=['accuracy'])

return model

model\_fnn\_optimized = create\_model(

dropout\_rate=best\_params['dropout'],

learning\_rate=0.01,

batch\_size=best\_params['batch\_size'],

epochs=best\_params['epochs']

)

history\_fnn\_optimized = model\_fnn\_optimized.fit(X\_train\_rfe, y\_train, epochs=best\_params['epochs'], batch\_size=best\_params['batch\_size'], validation\_data=(X\_test\_rfe, y\_test), verbose=1)

loss\_fnn\_opt, acc\_fnn\_opt = model\_fnn\_optimized.evaluate(X\_test\_rfe, y\_test)

print(f"Optimized FNN - Loss: {loss\_fnn\_opt}, Accuracy: {acc\_fnn\_opt}")

- The best hyperparameters from the random search are used to create and train an optimized FNN model.

- The optimized model is trained with the selected hyperparameters and evaluated on the testing data.

Step 9: Comparison and Performance Improvement

16. Comparing Models:

loss\_fnn, mae\_fnn = model\_fnn.evaluate(X\_test, y\_test)

print(f"Initial FNN - Loss: {loss\_fnn}, MAE: {mae\_fnn}")

loss\_fnn\_opt, mae\_fnn\_opt = model\_fnn\_optimized.evaluate(X\_test\_rfe, y\_test)

print(f"Optimized FNN - Loss: {loss\_fnn\_opt}, MAE: {mae\_fnn\_opt}")

print(f"Improvement in MAE: {mae\_fnn - mae\_fnn\_opt}")

- The initial and optimized FNN models are evaluated and their performance is compared to measure the improvement in mean absolute error (MAE).