

Câu 1:

1. Chạy code trong CODE_1

a. Các bước thực hiện:

B1 :Thực hiện thêm các thư viện

```
[4]: import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from sklearn.model_selection import train_test_split
from sklearn.datasets import make_classification
import numpy as np
```

B2: Tạo và chia tập dữ liệu

```
[5]: X, y = make_classification(n_samples=1000, n_features=20, n_classes=2, random_state=42)
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2, random_state=42)
```

B3: Định nghĩa mô hình Deeplearning

```
[6]: model = Sequential()
model.add(Dense(64, activation='relu', input_shape=(X_train.shape[1],))) # Lớp đầu vào
model.add(Dense(32, activation='relu')) # Lớp ẩn
model.add(Dense(1, activation='sigmoid')) # Lớp đầu ra cho phân Loại nhị phân
```

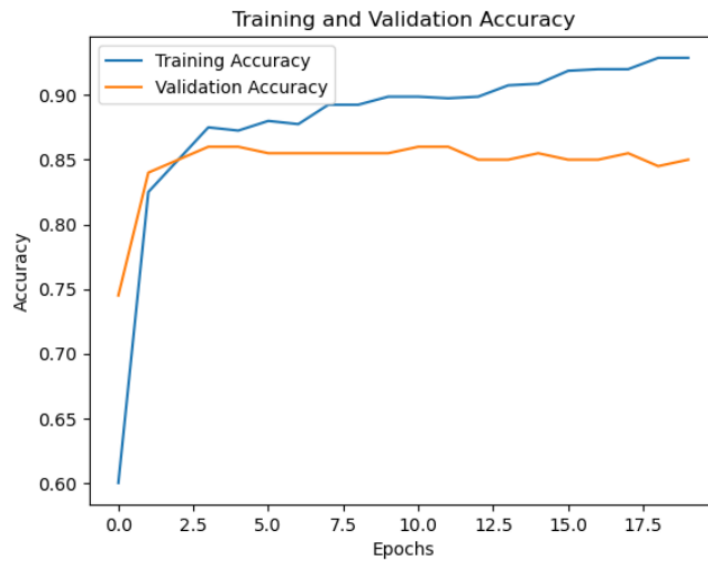
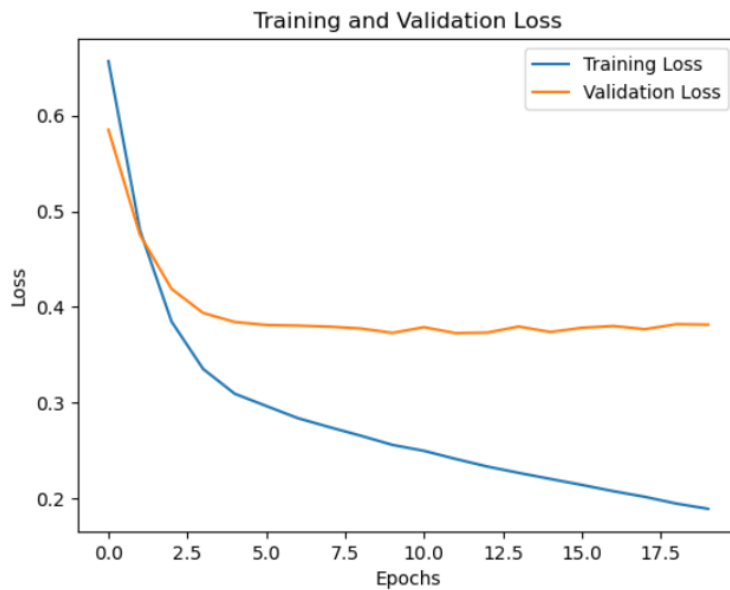
B4: Thực hiện huấn luyện mô hình

```
[7]: # 4. Compile mô hình
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
# 5. Huấn luyện mô hình với dữ liệu train và validate đồng thời
history = model.fit(
    X_train, y_train,
    epochs=20, # Số lượng epoch
    batch_size=32, # Kích thước batch
    validation_data=(X_val, y_val), # Sử dụng tập validation
    verbose=1 # Hiển thị chi tiết quá trình huấn luyện
)
# 6. Đánh giá mô hình trên tập validation sau huấn luyện
val_loss, val_accuracy = model.evaluate(X_val, y_val)
print(f"Validation Loss: {val_loss:.4f}")
print(f"Validation Accuracy: {val_accuracy:.4f}")

Epoch 1/20
25/25 — 3s 21ms/step - accuracy: 0.4995 - loss: 0.7154 - val_accuracy: 0.7450 - val_loss: 0.5852
Epoch 2/20
25/25 — 0s 6ms/step - accuracy: 0.8153 - loss: 0.5123 - val_accuracy: 0.8400 - val_loss: 0.4752
Epoch 3/20
25/25 — 0s 6ms/step - accuracy: 0.8541 - loss: 0.3912 - val_accuracy: 0.8500 - val_loss: 0.4188
Epoch 4/20
25/25 — 0s 6ms/step - accuracy: 0.8870 - loss: 0.3187 - val_accuracy: 0.8600 - val_loss: 0.3936
Epoch 5/20
25/25 — 0s 7ms/step - accuracy: 0.8921 - loss: 0.2845 - val_accuracy: 0.8600 - val_loss: 0.3841
Epoch 6/20
25/25 — 0s 8ms/step - accuracy: 0.8596 - loss: 0.3222 - val_accuracy: 0.8550 - val_loss: 0.3810
Epoch 7/20
25/25 — 0s 7ms/step - accuracy: 0.8834 - loss: 0.2779 - val_accuracy: 0.8550 - val_loss: 0.3804
Epoch 8/20
25/25 — 0s 6ms/step - accuracy: 0.8931 - loss: 0.2694 - val_accuracy: 0.8550 - val_loss: 0.3792
Epoch 9/20
25/25 — 0s 6ms/step - accuracy: 0.8968 - loss: 0.2548 - val_accuracy: 0.8550 - val_loss: 0.3773
Epoch 10/20
```

B5: Thực hiện vẽ biểu đồ thể hiện sự tương quan giữa các tham số

```
[8]: import matplotlib.pyplot as plt
# Vẽ đồ thị Loss (mất mát) giữa train và validation
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Training and Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
# Vẽ đồ thị Accuracy (độ chính xác) giữa train và validation
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Training and Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```



b. Giải thích

+ Giải thích về epoch:

epoch là một thuật ngữ chỉ một vòng lặp mà mô hình sẽ duyệt toàn bộ tập dữ liệu huấn luyện để cập nhật trọng số của nó.

Đối với ví dụ ở trên, epochs = 20 tức là mô hình sẽ duyệt toàn bộ tập dữ liệu huấn luyện 20 lần

+ Giải thích về batch

Batch là số mẫu mà mô hình sẽ xử lý trước khi cập nhật trọng số. Trong ví dụ ở trên, batch_size = 32 nghĩa là mô hình sẽ tính toán dựa trên 32 mẫu dữ liệu đầu vào, sau đó cập nhật trọng số rồi tiếp tục với 32 mẫu dữ liệu tiếp theo cho đến khi duyệt hết tập dữ liệu

+ Giải thích về train set

Train set là tập dữ liệu sử dụng để huấn luyện mô hình, để mô hình học từ các dữ liệu trong tập này. Mô hình sẽ điều chỉnh các trọng số và cập nhật các tham số để giảm thiểu lỗi trên tập huấn luyện

+ Giải thích về validation set

Validation là tập dữ liệu không tham gia vào quá trình huấn luyện trực tiếp mà được dùng để đánh giá mô hình sau mỗi epoch (một lần lặp qua toàn bộ tập huấn luyện). Qua đó, validation set giúp theo dõi hiệu suất của mô hình khi nó chưa "nhìn thấy" dữ liệu này trước đó. Điều này quan trọng để nhận biết nếu mô hình overfitting

+ Giải thích ý nghĩa về đồ thị loss



Training loss là mức độ sai lệch của mô hình trên tập huấn luyện.

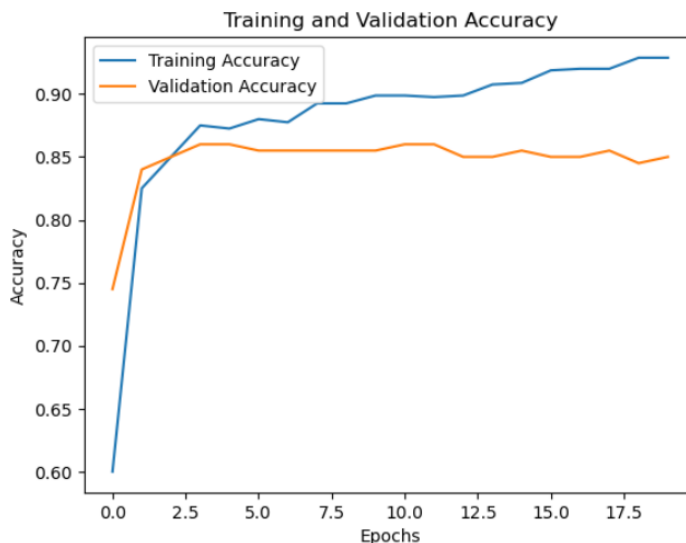
Validation loss là mức độ sai lệch của mô hình trên tập validation.

Từ đồ thị ở trên, ta có thể thấy:

- + Giá trị của training loss đang giảm dần qua các epoch, cho thấy mô hình đang học tốt hơn và cải thiện độ chính xác sau mỗi epoch từ tập huấn luyện

- + Ban đầu, validation loss cũng giảm, nhưng sau một số epoch, nó ổn định và có xu hướng không giảm nữa, trong khi training loss vẫn giảm. Điều này có thể là dấu hiệu của overfitting, khi mô hình bắt đầu học quá kỹ vào chi tiết của tập huấn luyện và không còn hiệu quả trên dữ liệu mới.

- + Giải thích về đồ thị accuracy



Training accuracy là độ chính xác của mô hình trên tập huấn luyện

Validation Accuracy là độ chính xác trên tập kiểm định(validation set)

Đối với đồ thị trên:

- + Training accuracy: Độ chính xác này tăng dần qua các epoch, đồng nghĩa với việc mô hình ngày càng dự đoán đúng hơn trên dữ liệu huấn luyện.

- + Validation accuracy: Ban đầu, nó tăng theo training accuracy, nhưng sau đó đạt một mức ổn định, thậm chí có dấu hiệu giảm nhẹ, cho thấy mô hình đã bắt đầu overfit.

+ Thêm tập test và đánh giá sử dụng MAE, MSE, RMSE

+ Khái niệm về MAE, MSE, RMSE

B1: Thực hiện chia lại tập dữ liệu. Thêm tập test với tỉ lệ tập train 70%, validation 15%, test 15%.

```
X, y = make_classification(n_samples=1000, n_features=20, n_classes=2, random_state=42)

X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.3, random_state=42)
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5, random_state=42)
```

B2: Thực hiện tính toán và đánh giá sử dụng MAE, MSE, RMSE

```
y_pred = model.predict(X_test).flatten()
y_pred_class = (y_pred > 0.5).astype(int)

# Tính MAE, MSE, RMSE cho tập test
mae = mean_absolute_error(y_test, y_pred)
mse = mean_squared_error(y_test, y_pred)
rmse = sqrt(mse)

print(f"Test MAE: {mae:.4f}")
print(f"Test MSE: {mse:.4f}")
print(f"Test RMSE: {rmse:.4f}")
```

5/5 ————— 0s 14ms/step

Test MAE: 0.2049

Test MSE: 0.1255

Test RMSE: 0.3542

Từ kết quả trên, với giá trị $MAE = 0.2049$, $MSE = 0.1255$, $RMSE = 0.3542$ ta có thể nhận xét rằng mô hình đang hoạt động tốt và sai lệch nhỏ. Nói chung, với các chỉ số MAE, MSE và RMSE như trên, mô hình này có thể xem là khá hiệu quả trong việc dự đoán trên tập dữ liệu hiện tại.

2. Tạo mô hình 7_layer bằng cách thêm 7 layer vào code, thêm 100 neuron và dropout và giải thích như Câu 1

B1: Thực hiện thêm 7 layer, mỗi layer 100 neuron và dropout

```
[14]: from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout

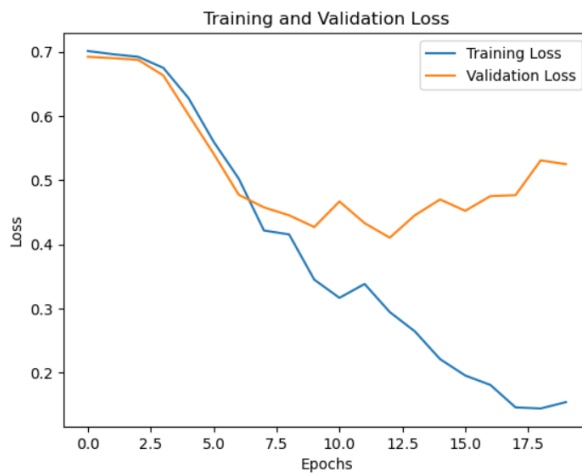
model_7_layer = Sequential()
model_7_layer.add(Dense(64, activation='relu', input_shape=(X_train.shape[1],))) # Lớp đầu vào
model_7_layer.add(Dense(32, activation='relu')) # Lớp ẩn đầu tiên
for _ in range(7):
    model_7_layer.add(Dense(100, activation='relu'))
    model_7_layer.add(Dropout(0.5))

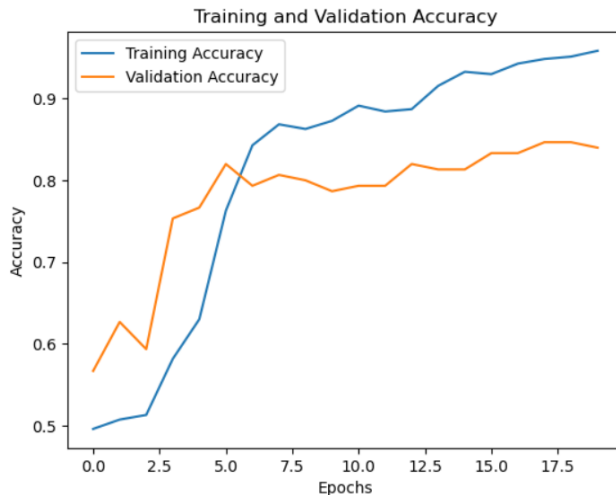
model_7_layer.add(Dense(1, activation='sigmoid')) # Lớp đầu ra cho phân loại nhị phân
model_7_layer.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
history_7_layer = model_7_layer.fit(
    X_train, y_train,
    epochs=20, # Số lượng epoch
    batch_size=32, # Kích thước batch
    validation_data=(X_val, y_val), # Sử dụng tập validation
    verbose=1 # Hiển thị chi tiết quá trình huấn luyện
)
```

B2: Thực hiện vẽ các đồ thị thể hiện sự tương quan và thay đổi của mô hình

```
[7]: import matplotlib.pyplot as plt
# Vẽ đồ thị Loss (mất mát) giữa train và validation
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Training and Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()

# Vẽ đồ thị Accuracy (độ chính xác) giữa train và validation
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Training and Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```





+ Giải thích về epoch:

epoch là một thuật ngữ chỉ một vòng lặp mà mô hình sẽ duyệt toàn bộ tập dữ liệu huấn luyện để cập nhật trọng số của nó.

Đối với ví dụ ở trên, epochs = 20 tức là mô hình sẽ duyệt toàn bộ tập dữ liệu huấn luyện 20 lần

+ Giải thích về batch

Batch là số mẫu mà mô hình sẽ xử lý trước khi cập nhật trọng số. Trong ví dụ ở trên, `batch_size = 32` nghĩa là mô hình sẽ tính toán dựa trên 32 mẫu dữ liệu đầu vào, sau đó cập nhật trọng số rồi tiếp tục với 32 mẫu dữ liệu tiếp theo cho đến khi duyệt hết tập dữ liệu

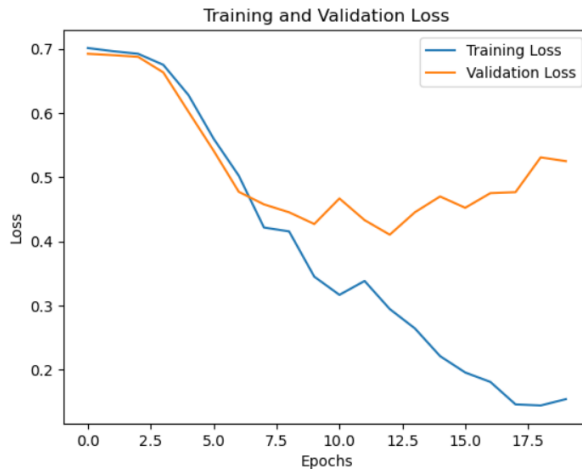
+ Giải thích về train set

Train set là tập dữ liệu sử dụng để huấn luyện mô hình, để mô hình học từ các dữ liệu trong tập này. Mô hình sẽ điều chỉnh các trọng số và cập nhật các tham số để giảm thiểu lỗi trên tập huấn luyện

+ Giải thích về validation set

Validation là tập dữ liệu không tham gia vào quá trình huấn luyện trực tiếp mà được dùng để đánh giá mô hình sau mỗi epoch (một lần lặp qua toàn bộ tập huấn luyện). Qua đó, validation set giúp theo dõi hiệu suất của mô hình khi nó chưa "nhìn thấy" dữ liệu này trước đó. Điều này quan trọng để nhận biết nếu mô hình overfitting

+Giải thích về đồ thị loss



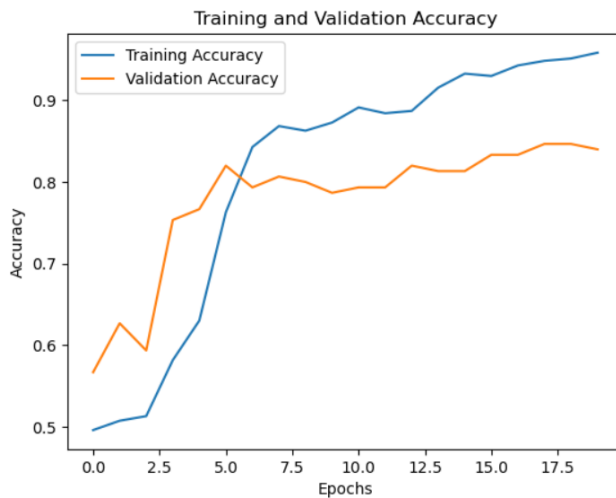
Training loss là mức độ sai lệch của mô hình trên tập huấn luyện.

Validation loss là mức độ sai lệch của mô hình trên tập validation.

Từ đồ thị ở trên, ta có thể thấy:

- + Giá trị của training loss đang giảm dần qua các epoch, cho thấy mô hình đang học tốt hơn và cải thiện độ chính xác sau mỗi epoch từ tập huấn luyện
- + Validation loss cũng giảm dần qua các epochs, nhưng đến một ngưỡng nhất định thì nó không giảm mà ngược lại có xu hướng tăng. Điều này thể hiện việc mô hình đang có dấu hiệu overfitting

+ Giải thích về đồ thị accuracy



Training accuracy là độ chính xác của mô hình trên tập huấn luyện

Validation Accuracy là độ chính xác trên tập kiểm định(validation set)

Đối với đồ thị trên:

+ Training accuracy: Độ chính xác này tăng dần qua các epoch, đồng nghĩa với việc mô hình ngày càng dự đoán đúng hơn trên dữ liệu huấn luyện.

+ Validation accuracy: độ chính xác được cải thiện và tăng qua các epoch, nhưng đến một ngưỡng nhất định ví dụ trong hình là khoảng 0.83 thì mô hình bắt đầu chững lại mà không tăng. Điều này chứng tỏ việc mô hình đang bị overfitting.

+ Thêm tập test và đánh giá các tham số MAE, MSE, RMSE

```
cpulab

[20]: y_pred_7_layer = model_7_layer.predict(X_test).flatten()

# Tính MAE, MSE, RMSE cho tập test
mae = mean_absolute_error(y_test, y_pred_7_layer)
mse = mean_squared_error(y_test, y_pred_7_layer)
rmse = sqrt(mse)

print(f"Test MAE: {mae:.4f}")
print(f"Test MSE: {mse:.4f}")
print(f"Test RMSE: {rmse:.4f}")

5/5 ————— 0s 6ms/step
Test MAE: 0.2119
Test MSE: 0.1662
Test RMSE: 0.4077
```

Sau khi thêm vào 7 layer mỗi layer 100 neuron và lớp dropout thì mô hình có dấu hiệu của sự gia tăng việc overfitting. Bằng chứng là các tham số MAE, MSE, RMSE đều có dấu hiệu tăng. Từ 0.2115 tăng lên 0.2119 đối với MAE, tương tự với MSE và RMSE là 0.1268 -> 0.1662 và 0.3561 -> 0.4077.

3. Giải thích CNN, RNN, LSTM

a. CNN

Bắt đầu từ những năm 1980, CNN được phát triển bởi Yann LeCun để nhận diện ký tự viết tay trong hệ thống đọc zip-code. Kiến trúc của CNN giúp xử lý tốt các dữ liệu hình ảnh, và đã được cải tiến thành nhiều loại CNN phức tạp như AlexNet (2012), VGGNet (2014) và ResNet (2015), đạt được thành công lớn trong các bài toán nhận diện hình ảnh.

Lịch sử phát triển:

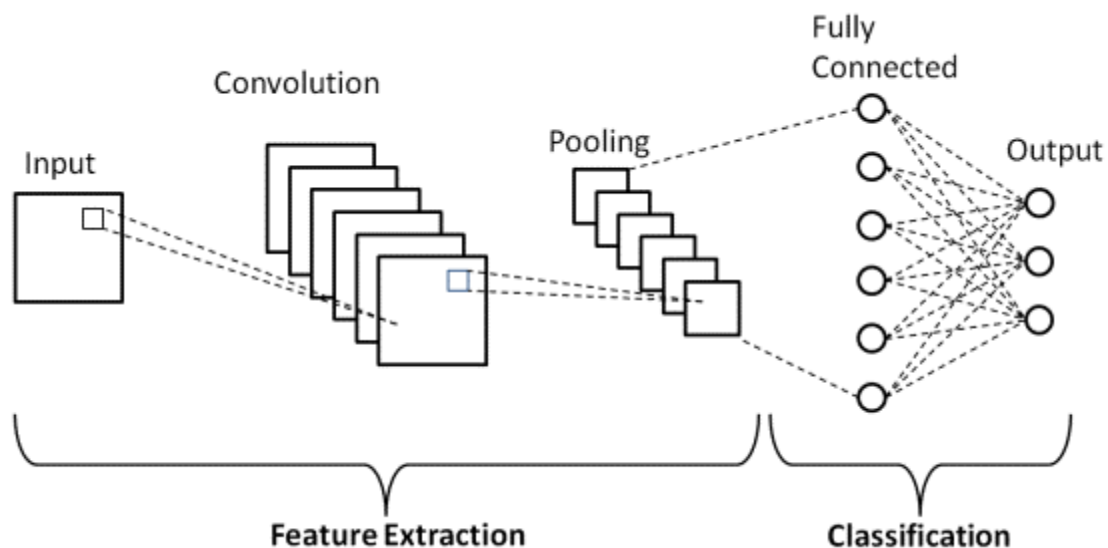
+ 1980s - Neocognitron: Kunihiko Fukushima đã phát triển Neocognitron, một trong những mô hình CNN đầu tiên, mô phỏng cơ chế xử lý hình ảnh của thị giác người. Neocognitron có cấu trúc phân cấp nhiều lớp, giúp phát hiện các mẫu đặc trưng như cạnh, góc và hình dạng trong hình ảnh.

+ 1998 - LeNet-5: Yann LeCun và các cộng sự phát triển LeNet-5, mô hình CNN đầu tiên được áp dụng rộng rãi trong nhận dạng chữ số viết tay (ví dụ như tập dữ liệu MNIST). LeNet-5 có các lớp tích chập và pooling, cho phép nhận diện các đặc trưng thị giác mà không cần trích xuất đặc trưng thủ công. LeNet-5 đã đặt nền móng cho việc sử dụng CNN trong nhận dạng hình ảnh.

+ 2012 - AlexNet: Alex Krizhevsky, Ilya Sutskever và Geoffrey Hinton phát triển AlexNet, kiến trúc CNN tiên phong áp dụng vào phân loại hình ảnh trên tập dữ liệu ImageNet, đánh bại các phương pháp truyền thống và giành chiến thắng trong cuộc thi ImageNet Large Scale Visual Recognition Challenge (ILSVRC). AlexNet sử dụng nhiều lớp tích chập và cải tiến như dropout và ReLU (Rectified Linear Unit) để tăng độ chính xác và giảm hiện tượng overfitting.

+ 2014 - VGGNet và GoogLeNet: VGGNet do Karen Simonyan và Andrew Zisserman của Đại học Oxford phát triển với kiến trúc sâu hơn, sử dụng các bộ lọc nhỏ hơn (3x3) và tạo ra kiến trúc VGG16, VGG19. GoogLeNet được phát triển bởi Google với kiến trúc Inception, kết hợp nhiều kích thước bộ lọc trong một lớp để giảm số lượng tham số và tăng tính hiệu quả.

+ 2015 trở đi - ResNet, DenseNet và các cải tiến khác: ResNet (Residual Networks) do Kaiming He và các cộng sự phát triển, sử dụng các kết nối tắt (skip connections) để giúp đào tạo các mô hình cực sâu. DenseNet giới thiệu thêm các kết nối dày đặc giữa các lớp. Các kiến trúc này tiếp tục đẩy giới hạn của CNN trong nhận dạng và phân loại hình ảnh phức tạp



Mạng CNN là một tập hợp các lớp Convolution chồng lên nhau và sử dụng các hàm nonlinear activation như ReLU và tanh để kích hoạt các trọng số trong các node. Mỗi một lớp sau khi thông qua các hàm kích hoạt sẽ tạo ra các thông tin trừu tượng hơn cho các lớp tiếp theo.

Các layer trong mô hình CNN liên kết được với nhau thông qua cơ chế convolution. Layer tiếp theo là kết quả convolution từ layer trước đó, nhờ vậy mà ta có được các kết nối cục bộ. Như vậy mỗi neuron ở lớp kế tiếp sinh ra từ kết quả của filter áp đặt lên một vùng ảnh cục bộ của neuron trước đó. Mỗi một lớp được sử dụng các filter khác nhau thông thường có hàng trăm hàng nghìn filter như vậy và kết hợp kết quả của chúng lại. Ngoài ra có một số layer khác như pooling/subsampling layer dùng để chắt lọc lại các thông tin hữu ích hơn (loại bỏ các thông tin nhiễu). Trong quá trình huấn luyện mạng (training) CNN tự động học các giá trị qua các lớp filter dựa vào cách thức mà bạn thực hiện. Ví dụ trong tác vụ phân lớp ảnh, CNNs sẽ cố gắng tìm ra thông số tối ưu cho các filter tương ứng theo thứ tự raw pixel > edges > shapes > facial > high-level features. Layer cuối cùng được dùng để phân lớp ảnh.

Code mẫu :

```

}

[ ]: from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense

# Khởi tạo mô hình
model_cnn = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)), # Lớp tích chập
    MaxPooling2D((2, 2)), # Lớp gộp (pooling)
    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    Flatten(), # Biến đổi thành vector phẳng
    Dense(128, activation='relu'), # Fully connected Layer
    Dense(10, activation='softmax') # Lớp đầu ra với softmax
])

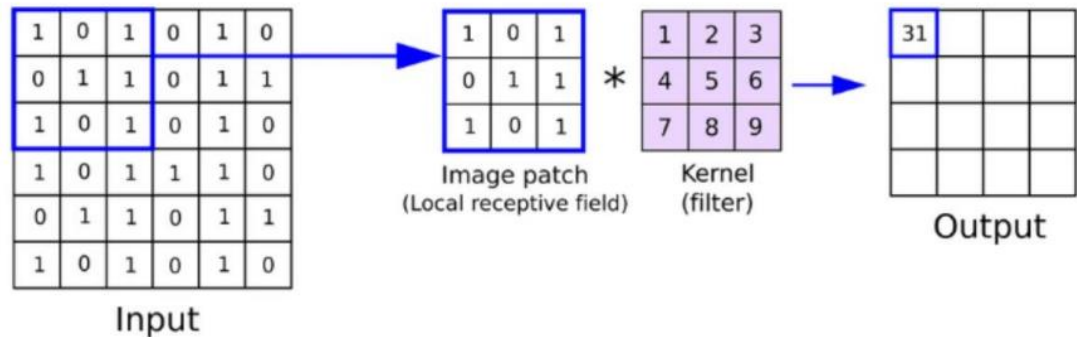
# Biên dịch mô hình
model_cnn.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
model_cnn.summary()

```

Các lớp trong mô hình CNN:

+

Conv2D



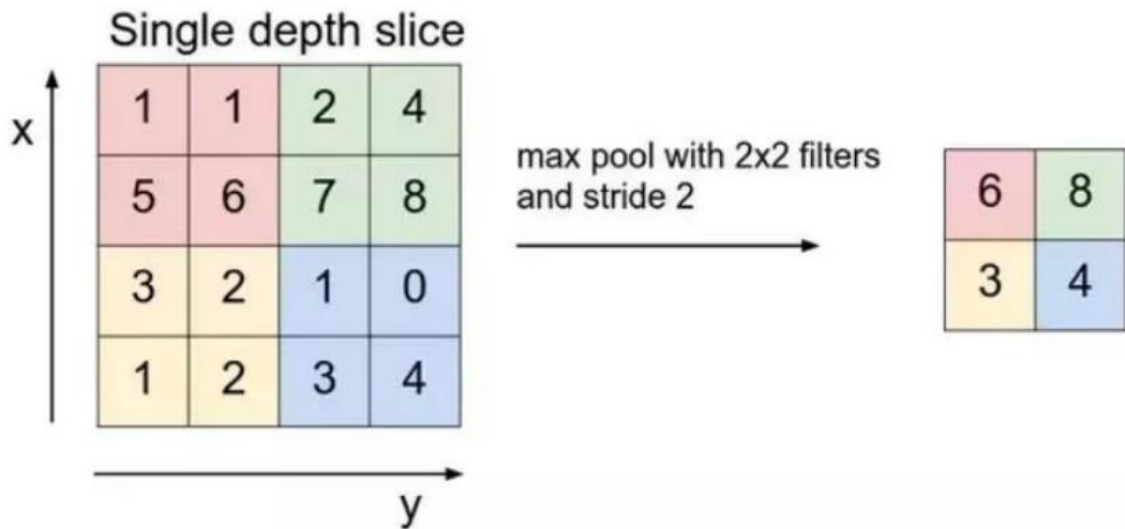
Mô tả: Đây là lớp tích chập (Convolutional Layer) 2D, áp dụng các filter (bộ lọc) lên các phần của ảnh đầu vào để trích xuất đặc trưng (feature). Conv2D là thành phần chính trong kiến trúc CNN, giúp mô hình học được các mẫu cục bộ trong hình ảnh.

Tham số chính: filters: Số lượng bộ lọc (số lượng đặc trưng cần trích xuất), trong đoạn code là 32 và 64. kernel_size: Kích thước của filter (ở đây là 3x3).

Activation='relu': Sử dụng hàm kích hoạt ReLU để loại bỏ các giá trị âm, chỉ giữ lại giá trị dương.

+

MaxPooling2D



Mô tả: Lớp gộp (Pooling Layer) 2D này được dùng để giảm kích thước của đặc trưng trích xuất, giúp giảm thiểu số lượng tham số và ngăn chặn hiện tượng quá khớp (overfitting).

Tham số chính: `pool_size=(2, 2)`: Kích thước của filter dùng để lấy giá trị lớn nhất trong vùng này. Kích thước (2, 2) giúp giảm dữ liệu thành một nửa chiều dài và chiều rộng.

+ Flatten: Mô tả: Lớp này làm phẳng dữ liệu từ dạng ma trận 2D (hoặc 3D) thành vector 1D. Điều này giúp chuyển đổi dữ liệu từ các lớp tích chập thành dạng đầu vào phù hợp cho các lớp fully connected (dense layers).

+ Dense: Mô tả: Đây là lớp fully connected, kết nối tất cả các neuron từ lớp trước với mỗi neuron trong lớp này. Tham số chính: `units`: Số lượng neuron trong lớp. Ví dụ, lớp đầu tiên có 128 neuron và lớp cuối cùng có 10 neuron cho 10 nhãn phân loại.
activation: Sử dụng relu trong lớp hidden và softmax trong lớp cuối để tính xác suất cho mỗi nhãn trong bài toán phân loại.

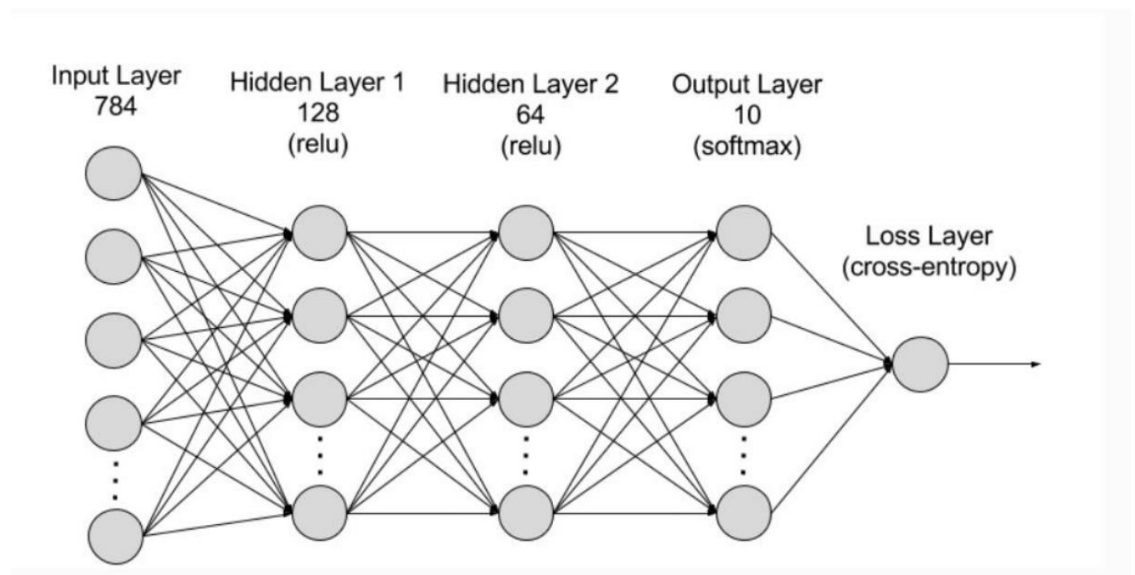
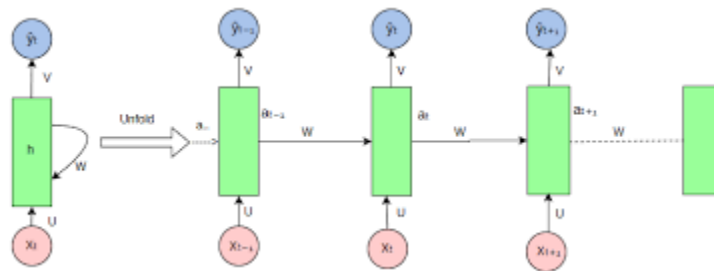
b. RNN

RNN được phát triển từ những năm 1980 và được ứng dụng nhiều vào các năm 1990. Các ý tưởng ban đầu về RNN đã được đề xuất bởi các nhà khoa học như David Rumelhart và Ronald J. Williams. RNN được phát triển để xử lý dữ liệu tuần tự (sequence data), đặc biệt hữu ích cho các ứng dụng ngôn ngữ tự nhiên (NLP). Tuy nhiên, RNN gặp khó khăn với các chuỗi dài do hiện tượng “vanishing gradient” (độ dốc mất dần) khiến mô hình khó học tốt các phụ thuộc xa. Ứng dụng phổ biến nhất hiện nay của RNN là xử lý ngôn ngữ tự nhiên, ví dụ như dịch máy, phân loại văn bản, và phân tích cảm xúc.

Lịch sử phát triển:

+ 1986 - RNN cơ bản: David Rumelhart, Geoffrey Hinton và Ronald J. Williams phát triển mô hình RNN cơ bản, một loại mạng có khả năng duy trì trạng thái dựa trên các đầu vào trước đó. RNN có khả năng học từ dữ liệu tuần tự nhưng gặp phải vấn đề biến mất hoặc bùng nổ độ dốc khi chuỗi dữ liệu quá dài, dẫn đến khó khăn trong học các phụ thuộc dài hạn.

+ 1990 - BPTT (Backpropagation Through Time): Thuật toán BPTT được phát triển nhằm cải thiện khả năng học của RNN bằng cách lan truyền ngược qua thời gian. Dù vậy, do vấn đề độ dốc biến mất, RNN cơ bản vẫn không hiệu quả với các chuỗi dài.



RNN được tạo thành từ các nơ-ron: các nút xử lý dữ liệu kết hợp cùng nhau để thực hiện các tác vụ phức tạp. Các nơ-ron được tổ chức dưới dạng lớp đầu vào, đầu ra và ẩn. Lớp đầu vào nhận thông tin để xử lý và lớp đầu ra cung cấp kết quả. Quá trình xử lý dữ liệu, phân tích và dự đoán diễn ra trong lớp ẩn.

Lớp ẩn RNN hoạt động bằng cách lần lượt truyền dữ liệu tuần tự nhận được đến các lớp ẩn. Tuy nhiên, RNN cũng có quy trình làm việc tự lặp lại hay hồi quy: lớp ẩn có

thể ghi nhớ và sử dụng các đầu vào trước đó cho các dự đoán trong tương lai trong một thành phần bộ nhớ ngắn hạn. Quy trình này sử dụng đầu vào hiện tại và bộ nhớ đã lưu trữ để dự đoán chuỗi tiếp theo.

Ví dụ: hãy xem xét chuỗi: Apple is red (Táo màu đỏ). Bạn muốn RNN dự đoán red (màu đỏ) khi nhận được chuỗi đầu vào Apple is (Táo màu). Khi xử lý từ Apple (Táo), lớp ẩn sẽ lưu trữ một bản sao trong bộ nhớ. Tiếp theo, khi thấy từ is (màu), lớp ẩn gọi lại Apple (Táo) từ bộ nhớ của mình và hiểu toàn bộ chuỗi: Apple is (Táo màu) là ngữ cảnh. Sau đó, lớp ẩn có thể dự đoán red (màu đỏ) để cải thiện độ chính xác. Do đó, RNN trở nên hữu ích trong nhận dạng giọng nói, dịch máy và các tác vụ lập mô hình ngôn ngữ khác

Code mẫu

```
[ ]: from tensorflow.keras.layers import SimpleRNN

# Khởi tạo mô hình
model_rnn = Sequential([
    SimpleRNN(50, activation='tanh', input_shape=(100, 1)), # Lớp RNN
    Dense(1) # Lớp đầu ra
])

# Biên dịch mô hình
model_rnn.compile(optimizer='adam', loss='mse')
model_rnn.summary()
```

Mô hình RNN này bao gồm ba lớp, mỗi lớp đảm nhiệm một vai trò cụ thể trong quá trình xử lý và dự đoán dữ liệu:

`layers.SimpleRNN(50, activation='relu', input_shape=(X_train.shape[1], 1))`

Đây là lớp mạng nơ-ron hồi quy đơn giản (Simple Recurrent Neural Network - RNN) với 50 đơn vị ẩn (neurons).

`input_shape=(X_train.shape[1], 1)`: Cấu trúc đầu vào là (số bước thời gian, số đặc trưng đầu vào). Ở đây, đầu vào được thiết lập với một đặc trưng trên mỗi bước thời gian.

`activation='relu'`: Chức năng kích hoạt ReLU (Rectified Linear Unit) được áp dụng tại mỗi bước thời gian. ReLU thường giúp mô hình học nhanh hơn và hạn chế vấn đề vanishing gradient. Lớp này sẽ xử lý đầu vào tuần tự, giúp mô hình nắm bắt thông tin theo trình tự thời gian và học cách giữ thông tin từ các bước trước đó.

Lớp Dense 64 đơn vị ẩn: `layers.Dense(64, activation='relu')` Đây là lớp kết nối dày đặc (fully connected) với 64 neurons.

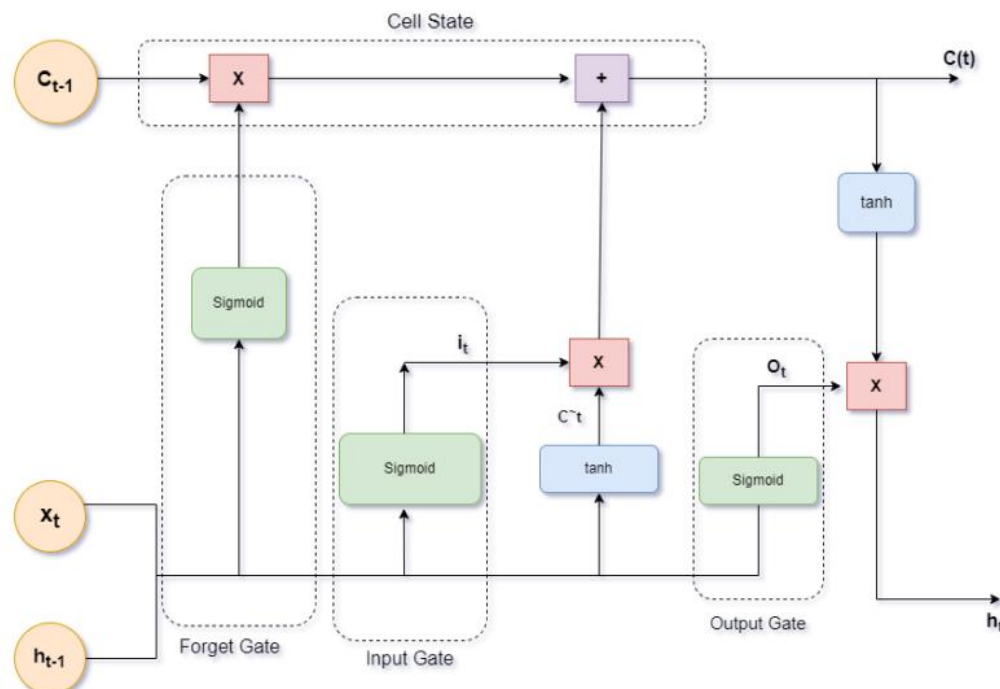
`activation='relu'`: Kích hoạt ReLU được dùng để tạo phi tuyến cho mô hình, giúp học các mẫu phức tạp hơn trong dữ liệu. Lớp này hoạt động như một tầng ẩn thêm để tăng độ sâu của mô hình, học các đặc trưng từ đầu ra của lớp RNN phía trước.

Lớp Dense đầu ra: `layers.Dense(1)` Lớp cuối cùng này chứa một neuron, dùng để dự đoán một giá trị đầu ra duy nhất (giá trị hồi quy). Không sử dụng hàm kích hoạt ở lớp này vì giá trị đầu ra là một số thực liên tục, phù hợp cho các bài toán dự đoán giá trị số như hồi quy.

Mô hình này thực hiện tốt cho bài toán dự đoán chuỗi thời gian hoặc các bài toán hồi quy cần xử lý dữ liệu tuần tự và lấy thông tin từ những bước trước đó trong chuỗi.

c. LSTM

LSTM được giới thiệu vào năm 1997. Được phát triển bởi Sepp Hochreiter và Jürgen Schmidhuber, hai nhà nghiên cứu Đức nổi tiếng trong lĩnh vực học sâu. LSTM được thiết kế để khắc phục vấn đề vanishing gradient của RNN. Mô hình này sử dụng các “cổng” (input gate, forget gate, output gate) để kiểm soát cách thông tin được lưu trữ, ghi đè, và xuất ra. Nhờ vậy, LSTM có thể duy trì trạng thái ổn định và ghi nhớ thông tin quan trọng trong thời gian dài. LSTM rất phổ biến trong các bài toán yêu cầu ghi nhớ thông tin dài hạn như dịch máy, nhận diện giọng nói, và các hệ thống chatbot. Đặc biệt, LSTM đã cải thiện đáng kể độ chính xác của các mô hình xử lý ngôn ngữ tự nhiên.



Code mẫu

```
[ ]: from tensorflow.keras.layers import LSTM

# Khởi tạo mô hình
model_lstm = Sequential([
    LSTM(50, activation='tanh', input_shape=(100, 1)), # Lớp LSTM
    Dense(1) # Lớp đầu ra
])

# Biên dịch mô hình
model_lstm.compile(optimizer='adam', loss='mse')
model_lstm.summary()
```

Lớp LSTM layers.

LSTM(50, activation='relu', input_shape=(X_train.shape[1], 1)):

LSTM (Long Short-Term Memory): Đây là lớp chính trong mô hình, chuyên xử lý dữ liệu tuần tự. LSTM giúp giải quyết vấn đề vanishing gradient mà các mô hình RNN truyền thống gặp phải, cho phép mô hình học từ các mối quan hệ dài hạn trong dữ liệu.

50: Số lượng đơn vị ẩn (neurons) trong lớp LSTM. Điều này quyết định kích thước của trạng thái ẩn và số lượng thông tin mà mô hình có thể lưu trữ và xử lý.

activation='relu': Hàm kích hoạt ReLU (Rectified Linear Unit) được sử dụng tại mỗi đơn vị. Hàm ReLU giúp tăng tốc độ huấn luyện và cung cấp tính phi tuyến cho mô hình.

input_shape=(X_train.shape[1], 1):

Định nghĩa kích thước đầu vào của mô hình. X_train.shape[1] đại diện cho số bước thời gian (timesteps) trong mỗi chuỗi dữ liệu. 1 là số đặc trưng đầu vào tại mỗi bước thời gian (có thể có nhiều hơn nếu dữ liệu có nhiều đặc trưng).

Lớp Dense (kết nối dày đặc)

layers.Dense(64, activation='relu'): Đây là một lớp kết nối dày đặc với 64 neurons. Lớp này được sử dụng để tăng cường khả năng học của mô hình, cho phép nó học các đặc trưng phức tạp hơn từ đầu ra của lớp LSTM.

activation='relu': Hàm kích hoạt ReLU cũng được sử dụng ở lớp này để cung cấp tính phi tuyến và giúp mô hình học nhanh hơn.

Lớp Dense đầu ra

layers.Dense(1): Đây là lớp đầu ra của mô hình, chứa một neuron. Lớp này dùng để dự đoán giá trị đầu ra duy nhất (trong trường hợp này là giá trị hồi quy). Không sử dụng hàm kích hoạt tại lớp này vì đầu ra là một số thực liên tục, phù hợp với các bài toán hồi quy.

Kết luận

Mô hình LSTM này bao gồm một lớp LSTM để xử lý dữ liệu tuần tự và học từ các mối quan hệ dài hạn, sau đó là một lớp Dense với 64 neurons để tăng cường khả năng học, và cuối cùng là một lớp Dense đầu ra với một neuron để dự đoán giá trị hồi quy. Cấu trúc này giúp mô hình có khả năng nắm bắt các thông tin quan trọng trong dữ liệu chuỗi và tạo ra dự đoán chính xác hơn

- Thay bởi mô hình CNN, RNN, LSTM với ≥ 5 layer và 100 neuron. So sánh, đánh giá 3 mô hình với 7_layer với các độ đo MAE, MSE, RMSE. Giải thích dựa trên các plot sinh ra được

Việc thay đổi mô hình 7_layer thành 3 mô hình CNN, RNN và LSTM cần một số điểm lưu ý:

+ Kích thước dữ liệu huấn luyện ban đầu có cấu trúc 2D

```
[7]: X, y = make_classification(n_samples=1000, n_features=20, n_classes=2, random_state=42)
      X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2, random_state=42)
      print(X.shape)
      print(X)

(1000, 20)
[[-0.6693561 -1.49577819 -0.87076638 ... -1.26733697 -1.2763343
  1.01664321]
 [ 0.09337237  0.78584826  0.10575379 ... -0.12270893  0.6934308
  0.91136272]
 [-0.90579721 -0.60834121  0.29514098 ...  0.83049813 -0.73733198
 -0.5782121 ]
 ...
 [-0.20013455 -1.46108168  1.79701652 ... -1.50280171 -1.27473745
  1.60111869]
 [ 0.03935575  0.24868361 -0.47532342 ...  0.09912579  0.54269228
  1.20827474]
 [ 0.76921528  0.47076539  0.16994471 ...  0.6561162  0.64333186
 -2.02100232]]
```

Để các mô hình có trên có thể sử dụng tập dữ liệu để huấn luyện, ta cần phải thay đổi số chiều của dữ liệu đầu vào.

1. Đối với hai mô hình RNN & LSTM

RNN và LSTM là các mô hình được thiết kế để xử lý dữ liệu chuỗi (sequential data). Chúng có khả năng ghi nhớ thông tin từ các bước thời gian trước đó trong chuỗi để dự đoán kết quả hiện tại. Hai mô hình này yêu cầu dữ liệu đầu vào có cấu trúc đặc biệt: Dữ liệu đầu vào cho RNN và LSTM cần phải có dạng 3D: (số mẫu, số bước thời gian, số tính năng).

Số mẫu (samples): Đây là số lượng chuỗi dữ liệu mà bạn có. Mỗi mẫu là một chuỗi riêng biệt.

Số bước thời gian (timesteps): Đây là số lượng các bước trong mỗi chuỗi (tức là chiều dài của chuỗi).

Số tính năng (features): Đây là số lượng các đặc trưng hoặc giá trị mà bạn có tại mỗi bước thời gian. Nếu mỗi bước thời gian chỉ có một giá trị (ví dụ: một giá trị số duy nhất như nhiệt độ), số tính năng sẽ là 1. Nếu tại mỗi bước thời gian có nhiều giá trị (ví dụ: các giá trị đo lường khác nhau cùng một lúc), số tính năng sẽ lớn hơn 1.

2. CNN

Trong khi RNN và LSTM chủ yếu xử lý dữ liệu chuỗi (sequence data), CNN thường được sử dụng cho dữ liệu có cấu trúc dạng lưới (ví dụ: hình ảnh, âm thanh, v.v.). Khi áp dụng CNN cho dữ liệu chuỗi thời gian, dữ liệu cũng cần được chuyển đổi thành dạng mà CNN có thể xử lý.

Đối với dữ liệu chuỗi thời gian:

CNN yêu cầu đầu vào có dạng 3D (cho từng mẫu): (số mẫu, số bước thời gian, số tính năng). Tương tự như RNN và LSTM, nhưng CNN sẽ sử dụng kernel (filter) để trượt qua các bước thời gian để học các đặc trưng.

Dữ liệu ảnh: Đối với ảnh, cấu trúc đầu vào sẽ là (số mẫu, chiều cao, chiều rộng, số kênh) (ví dụ: (batch_size, 64, 64, 3) đối với ảnh màu kích thước 64x64). Trong trường hợp bạn sử dụng CNN cho chuỗi thời gian, chuỗi dữ liệu có thể được coi như một "hình ảnh" 1D, nơi mỗi bước thời gian là một "pixel" và mỗi tính năng là một "kênh" trong ảnh.

+ Thực hiện thay đổi tập dữ liệu

```
# Chuyển đổi dữ liệu cho các mô hình RNN, CNN, LSTM
X_train_rnn = X_train.reshape((X_train.shape[0], X_train.shape[1], 1))
X_val_rnn = X_val.reshape((X_val.shape[0], X_val.shape[1], 1))
X_test_rnn = X_test.reshape((X_test.shape[0], X_test.shape[1], 1))
```

+ Định nghĩa hàm để thực hiện việc vẽ và huấn luyện mô hình

```
# Hàm để đánh giá và vẽ đồ thị
def evaluate_and_plot(model, model_name):
    history = model.fit(
        X_train_rnn, y_train,
        epochs=20,
        batch_size=32,
        validation_data=(X_val_rnn, y_val),
        verbose=0
    )
    # Dự đoán
    y_pred = model.predict(X_test_rnn).flatten()
    y_pred_class = (y_pred > 0.5).astype(int)
    mae = mean_absolute_error(y_test, y_pred)
    mse = mean_squared_error(y_test, y_pred)
    rmse = sqrt(mse)

    # Vẽ đồ thị
    plt.plot(history.history['loss'], label='train_loss')
    plt.plot(history.history['val_loss'], label='val_loss')
    plt.title(f'{model_name} Loss')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.legend()
    plt.show()

    return mae, mse, rmse
```

+ Giải thích code

```
history = model.fit(
    X_train_rnn, y_train,
    epochs=20,
    batch_size=32,
    validation_data=(X_val_rnn, y_val),
    verbose=0
)
```

X_train_rnn, y_train: Đây là dữ liệu đầu vào và nhãn tương ứng cho mô hình khi huấn luyện. X_train_rnn là dữ liệu đầu vào đã được chuyển đổi thành dạng 3D cho các mô hình RNN hoặc LSTM.

epochs=20: Đây là số lần mô hình sẽ học từ toàn bộ dữ liệu huấn luyện. Trong trường hợp này, mô hình sẽ được huấn luyện qua 20 epochs.

batch_size=32: Đây là số mẫu mà mô hình sẽ xử lý trong một lần cập nhật trọng số. Mỗi lần huấn luyện sẽ được chia thành các batch nhỏ, và mỗi batch có kích thước 32 mẫu.

validation_data=(X_val_rnn, y_val): Dữ liệu xác thực (validation data) được sử dụng để đánh giá mô hình sau mỗi epoch. Điều này giúp bạn theo dõi sự tiến bộ của mô hình trong suốt quá trình huấn luyện và tránh overfitting.

verbose=0: Tham số này kiểm soát mức độ hiển thị trong quá trình huấn luyện. verbose=0 có nghĩa là không hiển thị thông tin huấn luyện trong suốt quá trình. Bạn có thể thay đổi nó thành 1 để hiển thị tiến trình hoặc 2 để chỉ hiển thị thông tin mỗi epoch.

```

)
# Dự đoán
y_pred = model.predict(X_test_rnn).flatten()
y_pred_class = (y_pred > 0.5).astype(int)
mae = mean_absolute_error(y_test, y_pred)
mse = mean_squared_error(y_test, y_pred)
rmse = sqrt(mse)

```

`model.predict(X_test_rnn)`: Sau khi huấn luyện xong, mô hình sẽ được dùng để dự đoán kết quả trên dữ liệu kiểm tra (`X_test_rnn`). Dữ liệu kiểm tra đã được chuyển thành dạng 3D (cho phù hợp với yêu cầu của mô hình RNN hoặc LSTM).

`flatten()`: Lý do gọi `flatten()` là vì hàm `predict()` trả về một mảng 2D (tương ứng với các mẫu và các dự đoán). Hàm `flatten()` chuyển mảng này thành một mảng 1D.

`(y_pred > 0.5).astype(int)`: Đối với bài toán phân loại nhị phân, mô hình sẽ trả về xác suất (dự đoán thuộc về lớp 1). Ta sẽ so sánh xác suất này với 0.5, và nếu lớn hơn 0.5, ta gán nhãn là 1 (chuyển sang lớp 1), nếu nhỏ hơn hoặc bằng 0.5, nhãn sẽ là 0 (chuyển sang lớp 0).

+ Định nghĩa mô hình RNN

```

# 1. Mô hình RNN với 7 Layers
model_rnn = Sequential([
    SimpleRNN(100, activation='relu', input_shape=(X_train.shape[1], 1), return_sequences=True),
    SimpleRNN(100, activation='relu', return_sequences=True),
    SimpleRNN(100, activation='relu', return_sequences=True),
    SimpleRNN(100, activation='relu', return_sequences=True),
    SimpleRNN(100, activation='relu'),
    Dense(100, activation='relu'),
    Dense(1, activation='sigmoid')
])
model_rnn.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
mae_rnn, mse_rnn, rmse_rnn = evaluate_and_plot(model_rnn, "RNN")

```

`SimpleRNN(100, activation='relu', input_shape=(X_train.shape[1], 1), return_sequences=True)`:

- Đây là lớp SimpleRNN đầu tiên với 100 neuron.
- `activation='relu'`: Sử dụng hàm kích hoạt ReLU cho lớp này.
- `input_shape=(X_train.shape[1], 1)`: Kích thước đầu vào là (số bước thời gian, số tính năng). Dữ liệu đầu vào cho mô hình này là 3D, với số bước thời gian là `X_train.shape[1]` và 1 tính năng (feature) tại mỗi bước thời gian.
- `return_sequences=True`: Chỉ định rằng lớp này sẽ trả về một chuỗi các trạng thái ẩn (hidden states) cho mỗi bước thời gian, thay vì chỉ trả về trạng thái ẩn cuối cùng. Điều này là cần thiết để truyền thông tin qua các lớp RNN tiếp theo.

Các lớp SimpleRNN(100, activation='relu', return_sequences=True):

- Có 4 lớp SimpleRNN, mỗi lớp đều có 100 neuron và sử dụng hàm kích hoạt ReLU.
- Các lớp này đều có return_sequences=True, nghĩa là mỗi lớp sẽ trả về chuỗi các trạng thái ẩn (hidden states) cho các bước thời gian, cho phép các lớp sau có thể tiếp tục xử lý dữ liệu chuỗi.

SimpleRNN(100, activation='relu'): Đây là lớp SimpleRNN cuối cùng không có return_sequences=True, nghĩa là nó sẽ trả về trạng thái ẩn cuối cùng cho toàn bộ chuỗi. Lớp này sẽ cho thông tin tóm tắt từ các bước thời gian trước đó, sau đó được truyền đến các lớp tiếp theo.

Dense(100, activation='relu'): Đây là lớp dense có 100 neuron và sử dụng hàm kích hoạt ReLU, giúp mô hình học được các đặc trưng phức tạp từ dữ liệu.

Dense(1, activation='sigmoid'): Đây là lớp cuối cùng với một neuron và sử dụng hàm kích hoạt sigmoid. Nó được sử dụng trong bài toán phân loại nhị phân, trả về xác suất của lớp 1 (ví dụ: xác suất của đối tượng thuộc lớp 1).

+ Định nghĩa mô hình CNN

```
# 2. Mô hình CNN với 7 Layers
model_cnn = Sequential([
    Conv1D(100, kernel_size=3, activation='relu', input_shape=(X_train.shape[1], 1)),
    Conv1D(100, kernel_size=3, activation='relu'),
    Conv1D(100, kernel_size=3, activation='relu'),
    Conv1D(100, kernel_size=3, activation='relu'),
    Conv1D(100, kernel_size=3, activation='relu'),
    Flatten(),
    Dense(1, activation='sigmoid')
])
model_cnn.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
mae_cnn, mse_cnn, rmse_cnn = evaluate_and_plot(model_cnn, "CNN")
```

Conv1D(100, kernel_size=3, activation='relu', input_shape=(X_train.shape[1], 1)):

- Conv1D là lớp convolutional 1D, thường được dùng để xử lý dữ liệu chuỗi hoặc tín hiệu 1 chiều (như chuỗi thời gian, văn bản, v.v.).
- 100 là số lượng bộ lọc (filters) trong lớp này. Mỗi bộ lọc học được một đặc trưng trong chuỗi đầu vào.
- kernel_size=3: Kích thước của bộ lọc, trong trường hợp này, mỗi bộ lọc sẽ "quét" qua 3 bước thời gian tại mỗi lần di chuyển.
- activation='relu': Sử dụng hàm kích hoạt ReLU (Rectified Linear Unit), một hàm kích hoạt phổ biến giúp mô hình học nhanh hơn và tránh hiện tượng vanishing gradient.

- `input_shape=(X_train.shape[1], 1)`: Cấu trúc đầu vào của mô hình, với `X_train.shape[1]` là số bước thời gian (time steps) trong mỗi chuỗi đầu vào và 1 là số lượng tính năng tại mỗi bước thời gian.

Các lớp Conv1D tiếp theo: Các lớp này tiếp tục sử dụng 100 bộ lọc và `kernel_size=3`, giúp mô hình học thêm các đặc trưng phức tạp hơn từ dữ liệu. Tất cả các lớp này đều sử dụng ReLU làm hàm kích hoạt.

`Flatten()`:

- Lớp này "phẳng hóa" đầu ra từ các lớp convolutional, chuyển nó thành một mảng 1D, để có thể đưa vào các lớp Dense.
- Nếu đầu ra của các lớp convolutional là một mảng 3D (số lượng mẫu, số lượng bước thời gian, số lượng bộ lọc), lớp Flatten sẽ chuyển nó thành một mảng 2D (số lượng mẫu, số lượng đặc trưng) để có thể đưa vào lớp dense.

`Dense(1, activation='sigmoid')`:

- Đây là lớp fully connected (Dense) cuối cùng, với 1 neuron và hàm kích hoạt sigmoid. Vì đây là bài toán phân loại nhị phân, lớp này sẽ trả về xác suất của lớp 1 (tức là xác suất mẫu thuộc lớp 1).

+ Định nghĩa mô hình LSTM

```
# 3. Mô hình LSTM với 7 Layers
model_lstm = Sequential([
    LSTM(100, activation='relu', input_shape=(X_train.shape[1], 1), return_sequences=True),
    LSTM(100, activation='relu', return_sequences=True),
    LSTM(100, activation='relu', return_sequences=True),
    LSTM(100, activation='relu', return_sequences=True),
    LSTM(100, activation='relu'),
    Dense(100, activation='relu'),
    Dense(1, activation='sigmoid')
])
model_lstm.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
mae_lstm, mse_lstm, rmse_lstm = evaluate_and_plot(model_lstm, "LSTM")
```

`LSTM(100, activation='relu', input_shape=(X_train.shape[1], 1), return_sequences=True)`:

- LSTM là một lớp Long Short-Term Memory. Đây là một loại mạng nơ-ron tái hồi đặc biệt, giúp mạng học và ghi nhớ thông tin trong một khoảng thời gian dài, rất hữu ích cho các bài toán chuỗi thời gian.
- 100: Số lượng neuron trong lớp LSTM, tương tự như số lượng bộ lọc trong lớp Conv1D. Mỗi LSTM unit có thể học được một số lượng đặc trưng (features) từ dữ liệu.

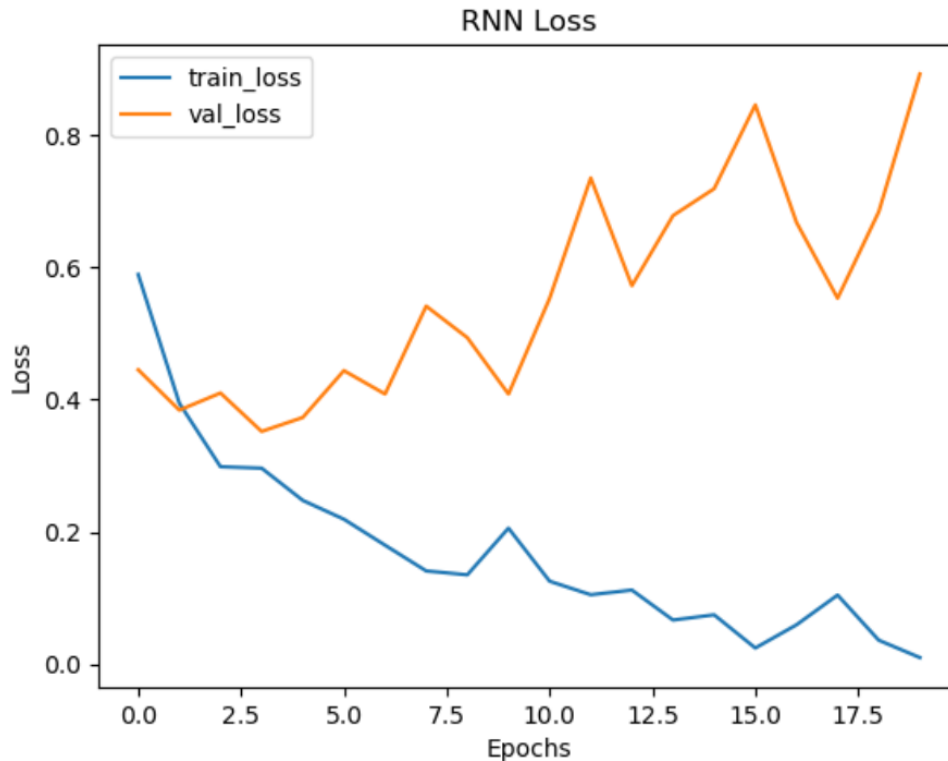
- `activation='relu'`: Sử dụng hàm kích hoạt ReLU. Đây là một lựa chọn phổ biến, giúp tăng tốc độ huấn luyện và tránh vấn đề vanishing gradient.
- `input_shape=(X_train.shape[1], 1)`: Đây là hình dạng (shape) đầu vào của mô hình, nơi `X_train.shape[1]` là số bước thời gian (timesteps) và 1 là số lượng tính năng (features) tại mỗi bước thời gian.
- `return_sequences=True`: Lớp LSTM này sẽ trả về toàn bộ chuỗi đầu ra cho mỗi bước thời gian (chứ không phải chỉ trả về đầu ra cuối cùng). Điều này giúp truyền thông tin qua các lớp LSTM tiếp theo.

Các lớp LSTM tiếp theo: Có tổng cộng 5 lớp LSTM trong mô hình này. Các lớp LSTM tiếp theo đều có `return_sequences=True`, có nghĩa là mỗi lớp sẽ trả về một chuỗi các trạng thái ẩn (hidden states) thay vì chỉ trả về trạng thái ẩn cuối cùng, giúp mô hình có thể tiếp tục học từ các đặc trưng ở các bước thời gian trước đó. Các lớp này giúp mô hình học thêm các đặc trưng phức tạp hơn từ dữ liệu theo các bước thời gian.

`Dense(100, activation='relu')`: Đây là một lớp fully connected (dense) với 100 neuron và sử dụng hàm kích hoạt ReLU. Lớp này giúp mô hình học các đặc trưng cao cấp và các kết nối phi tuyến tính giữa các đặc trưng từ các lớp LSTM.

`Dense(1, activation='sigmoid')`: Đây là lớp đầu ra của mô hình, với 1 neuron và hàm kích hoạt sigmoid. Do đây là bài toán phân loại nhị phân (2 lớp), hàm sigmoid sẽ giúp mô hình đưa ra một xác suất cho lớp 1. Nếu xác suất lớn hơn 0.5, mô hình sẽ dự đoán là lớp 1, nếu nhỏ hơn 0.5, dự đoán là lớp 0.

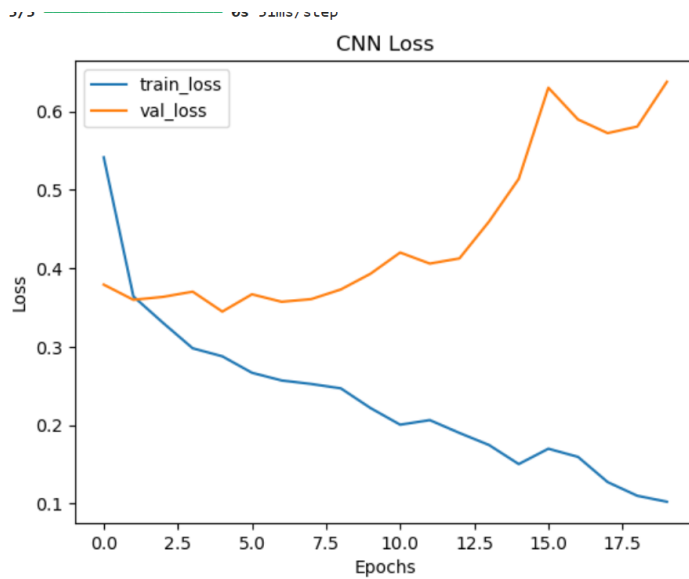
+Thực hiện vẽ biểu đồ trực quan và so sánh tham số MAE, MSE, RMSE



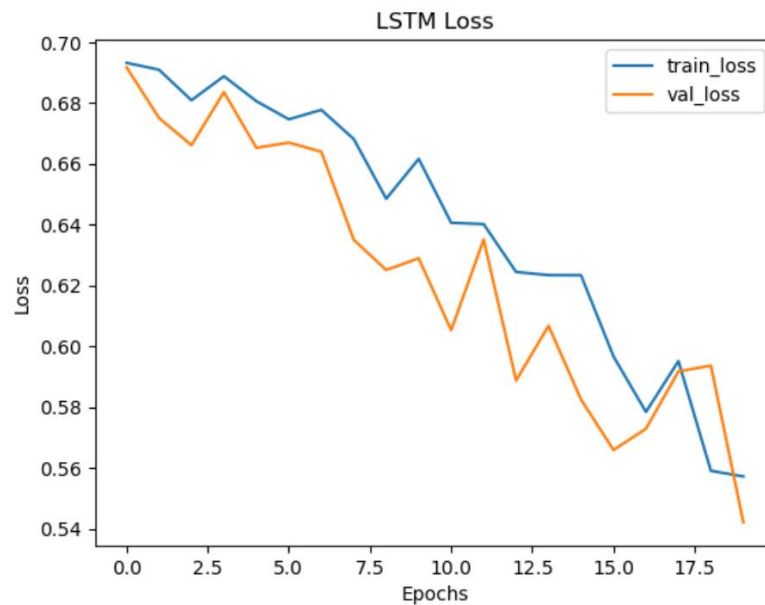
Đối với mô hình RNN, có thể thấy một dấu hiệu rõ ràng của việc overfitting.

Train_loss giảm dần đều qua các epoch, cho thấy mô hình ngày càng học tốt trên dữ liệu huấn luyện. Tuy nhiên, val_loss lại tăng lên theo thời gian, nghĩa là mô hình đang hoạt động kém dần trên tập validation. Điều này cho thấy mô hình đã bắt đầu ghi nhớ quá mức các mẫu huấn luyện mà không khái quát hóa tốt cho dữ liệu mới.

Chênh lệch giữa train_loss và val_loss: Sự chênh lệch lớn giữa train_loss và val_loss (đặc biệt là ở các epoch cuối cùng) cũng là một dấu hiệu của việc overfitting. Lý tưởng là val_loss nên giảm hoặc dao động quanh một giá trị nhất định, thay vì liên tục tăng lên.



Tương tự với mô hình RNN, mô hình CNN có dấu hiệu overfitting. Train_loss có xu hướng giảm nhưng val_loss thì ngược lại có xu hướng giảm xong tiếp tục tăng.



Đối với mô hình LSTM:

Tốt hơn về khả năng tổng quát hóa: Biểu đồ cho thấy train_loss và val_loss đều giảm dần theo thời gian, đồng thời hai đường này không có sự chênh lệch lớn. Điều này là dấu hiệu tốt cho thấy mô hình LSTM không bị overfitting và có khả năng tổng quát hóa tốt hơn so với mô hình RNN và CNN trước đó.

Hội tụ ổn định: Cả train_loss và val_loss giảm đều đặn qua các epoch, không có dấu hiệu dao động lớn. Điều này cho thấy quá trình học của mô hình diễn ra ổn định và không gặp vấn đề về việc học quá chậm hoặc quá nhanh.

Hiệu quả mô hình: Khi hai giá trị train_loss và val_loss giảm đồng đều, mô hình LSTM có khả năng tiếp tục cải thiện nếu cho chạy thêm epoch. Việc val_loss thấp hơn train_loss vào một số epoch cũng là bình thường, cho thấy mô hình học được các mẫu trong dữ liệu validation mà không ghi nhớ chúng một cách máy móc.

+ So sánh tham số MAE, MSE, RMSE

Epochs

So sánh kết quả của các mô hình với 7 layer:

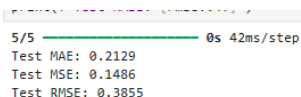
RNN	- Test MAE: 0.1784,	Test MSE: 0.1550,	Test RMSE: 0.3937
CNN	- Test MAE: 0.1998,	Test MSE: 0.1465,	Test RMSE: 0.3828
LSTM	- Test MAE: 0.3489,	Test MSE: 0.1654,	Test RMSE: 0.4067

Mô hình CNN có các giá trị MAE, MSE và RMSE thấp nhất, cho thấy hiệu suất tốt nhất trong ba mô hình. Điều này có thể là do CNN tận dụng tốt các bộ lọc để tìm ra các đặc trưng quan trọng từ dữ liệu, ngay cả khi dữ liệu không có tính tuần tự. CNN thường xử lý tốt dữ liệu có tính cục bộ, giúp mô hình có độ tổng quát hóa tốt hơn trong bài toán này.

Mô hình RNN có kết quả tốt hơn so với LSTM nhưng kém hơn so với CNN. Mặc dù RNN có cấu trúc đơn giản hơn và không có khả năng ghi nhớ dài hạn như LSTM, nhưng nó vẫn cho kết quả khá tốt, có thể là do dữ liệu trong bài toán này không yêu cầu ghi nhớ dài hạn, làm cho RNN dễ dàng học được các mẫu hơn.

Mô hình LSTM có MAE cao nhất, cho thấy nó hoạt động kém nhất trong ba mô hình. Tuy nhiên, MSE và RMSE của LSTM gần với RNN, cho thấy rằng lỗi bình phương vẫn tương đối thấp. LSTM thường hiệu quả cho dữ liệu có tính tuần tự rõ rệt, nhưng với dữ liệu này (giả sử không có tính tuần tự phức tạp), LSTM có thể dư thừa, dẫn đến kết quả không tốt như CNN và RNN. Điều này cũng có thể do LSTM yêu cầu thời gian huấn luyện lâu hơn để đạt hiệu quả tối đa.

+ So sánh với mô hình 7_layer



```
5/5 0s 42ms/step
Test MAE: 0.2129
Test MSE: 0.1486
Test RMSE: 0.3855
```

Từ các thông số MAE, MSE, RMSE ta có thể đưa ra kết luận CNN và RNN có hiệu suất tốt hơn mô hình 7_layer và LSTM có hiệu suất kém hơn 7_layer.

Câu 2:) Dự đoán điểm và phân loại sinh viên A+, A, B+....

1. Tự sinh tập DATA với 1000 sinh viên trong đó 500 Môn 1 (10%, 10%, 20%, 60%), 500 Môn 2(10%, 20%, 20%, 50%)

2. Sử dụng mô hình 7_layer, CNN, RNN, LSTM với ≥ 5 Layer và 100 neuron, cơ chế dropout...để dự đoán điểm và phân loại 3. Giải thích các mô hình 4. Đánh giá và xây dựng ứng dụng mobile, web

Bài làm

Import các thư viện cần thiết

```
[31]: import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from tensorflow.keras.utils import to_categorical
```

Thực hiện sinh tập dữ liệu

```
[1]: # Tạo điểm thành phần cho Môn 1 và Môn 2 với 4 thành phần và trọng số khác nhau
def generate_scores(num_students, weights):
    scores = np.random.uniform(0, 10, (num_students, 4)) # Điểm ngẫu nhiên từ 0 đến 10
    total_scores = np.dot(scores, weights) / 100
    return scores, total_scores

# Trọng số cho Môn 1 và Môn 2
weights_mon1 = [10, 10, 20, 60]
weights_mon2 = [10, 20, 20, 50]

# Sinh dữ liệu
scores_mon1, total_scores_mon1 = generate_scores(500, weights_mon1)
scores_mon2, total_scores_mon2 = generate_scores(500, weights_mon2)

# Gộp dữ liệu của cả hai môn học
scores = np.vstack([scores_mon1, scores_mon2])
# print(scores)
total_scores = np.concatenate([total_scores_mon1, total_scores_mon2])
```

Phân loại điểm theo thang điểm yêu cầu:

```
# Phân Loại điểm theo thang điểm yêu cầu
def classify_grade(score):
    if score < 4:
        return 'F'
    elif 4 <= score < 6:
        return 'D'
    elif 6 <= score < 6.5:
        return 'C'
    elif 6.5 <= score < 7:
        return 'C+'
    elif 7 <= score < 8:
        return 'B'
    elif 8 <= score < 8.5:
        return 'B+'
    elif 8.5 <= score < 9:
        return 'A'
    else:
        return 'A+'

grades = np.array([classify_grade(score) for score in total_scores])
# print(grades)

# Lấy 3 cột đầu làm X và cột cuối làm y
scores = np.round(scores, 2)

X = scores[:, :3]
print(X.shape)
y = scores[:, -1]
print(y.shape)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

(1000, 3)
(1000,)
```

Xây dựng mô hình 7_layer:

```
[2]: from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout

def create_dnn_model(input_shape):
    model = Sequential()
    model.add(Dense(100, input_shape=input_shape, activation='relu'))
    model.add(Dropout(0.1))
    model.add(Dense(100, activation='relu'))
    model.add(Dropout(0.1))
    model.add(Dense(100, activation='relu'))
    model.add(Dropout(0.1))
    model.add(Dense(100, activation='relu'))
    model.add(Dropout(0.1))
    model.add(Dense(100, activation='relu'))
    model.add(Dropout(0.1))
    model.add(Dense(1)) # Một neuron đầu ra cho dự đoán điểm số liên tục
    return model

# Tạo và biên dịch mô hình
model = create_dnn_model(X_train.shape[1,])
model.compile(optimizer='adam', loss='mean_squared_error', metrics=['mae'])
```

- **Input Layer:** Lớp đầu vào có số đầu vào bằng với số đặc trưng của dữ liệu (input_shape = (X_train.shape[1,])).
- **Hidden Layers:** Mô hình bao gồm **5 lớp ẩn**, mỗi lớp có:
 - **100 neuron**, kích hoạt bởi hàm **ReLU (Rectified Linear Unit)**. Hàm ReLU giúp mô hình học được các quan hệ phi tuyến trong dữ liệu.
 - **Dropout layer** với tỷ lệ **0.1** sau mỗi lớp ẩn. Dropout là một kỹ thuật regularization để ngăn chặn overfitting. Nó hoạt động bằng cách ngẫu nhiên bỏ bớt 10% các neuron trong mỗi lớp trong quá trình huấn luyện, giúp mô hình trở nên linh hoạt hơn và giảm độ phù hợp quá mức với dữ liệu huấn luyện.
- **Output Layer:** Lớp đầu ra có **1 neuron** không có hàm kích hoạt, phù hợp cho bài toán dự đoán giá trị liên tục. Điều này gợi ý rằng mô hình này đang được thiết kế cho bài toán hồi quy (regression).

- **Biên dịch (Compile) mô hình:**

- **Optimizer:** adam - một thuật toán tối ưu hóa phổ biến, hiệu quả trong việc điều chỉnh trọng số, giúp mô hình hội tụ nhanh hơn.
- **Loss function:** mean_squared_error (MSE) - hàm lỗi trung bình bình phương, phù hợp cho bài toán hồi quy vì nó giúp giảm thiểu sai số giữa giá trị dự đoán và giá trị thực.
- **Metrics:** mae (Mean Absolute Error) - một độ đo khác để đánh giá độ chính xác của mô hình, giúp quan sát sai số tuyệt đối trung bình giữa các dự đoán và giá trị thực.

Huấn luyện và đánh giá mô hình:

```
[3]: # Huấn luyện mô hình
model.fit(X_train, y_train, epochs=30, batch_size=32, validation_data=(X_test, y_test))

# Đánh giá mô hình
loss, mae = model.evaluate(X_test, y_test)
print(f"Test loss: {loss}, Test MAE: {mae}")

Epoch 1/30
25/25 — 5s 27ms/step - loss: 19.2315 - mae: 3.5867 - val_loss: 10.6688 - val_mae: 2.7094
Epoch 2/30
25/25 — 0s 7ms/step - loss: 9.9762 - mae: 2.6070 - val_loss: 9.0514 - val_mae: 2.5698
Epoch 3/30
25/25 — 0s 6ms/step - loss: 9.8601 - mae: 2.6233 - val_loss: 9.0965 - val_mae: 2.5689
Epoch 4/30
25/25 — 0s 8ms/step - loss: 9.9626 - mae: 2.6245 - val_loss: 8.9409 - val_mae: 2.5494
Epoch 5/30
25/25 — 0s 6ms/step - loss: 10.5189 - mae: 2.7193 - val_loss: 9.0797 - val_mae: 2.5579
Epoch 6/30
25/25 — 0s 7ms/step - loss: 9.6783 - mae: 2.6142 - val_loss: 8.9100 - val_mae: 2.5436
Epoch 7/30
25/25 — 0s 6ms/step - loss: 9.9459 - mae: 2.6500 - val_loss: 8.7567 - val_mae: 2.5329
Epoch 8/30
25/25 — 0s 9ms/step - loss: 9.5550 - mae: 2.6002 - val_loss: 8.5963 - val_mae: 2.5114
Epoch 9/30
25/25 — 0s 6ms/step - loss: 9.3236 - mae: 2.5842 - val_loss: 8.7043 - val_mae: 2.5204
Epoch 10/30
25/25 — 0s 6ms/step - loss: 9.2006 - mae: 2.5366 - val_loss: 9.1489 - val_mae: 2.5477
Epoch 11/30
25/25 — 0s 6ms/step - loss: 9.3490 - mae: 2.5937 - val_loss: 8.6092 - val_mae: 2.5006
Epoch 12/30
25/25 — 0s 6ms/step - loss: 9.3367 - mae: 2.5732 - val_loss: 8.3848 - val_mae: 2.4913
Epoch 13/30
25/25 — 0s 6ms/step - loss: 9.1473 - mae: 2.5519 - val_loss: 8.6683 - val_mae: 2.5033
Epoch 14/30
25/25 — 0s 6ms/step - loss: 8.6557 - mae: 2.4967 - val_loss: 8.6580 - val_mae: 2.4993
Epoch 15/30
25/25 — 0s 6ms/step - loss: 8.8742 - mae: 2.5234 - val_loss: 8.2920 - val_mae: 2.4726
Epoch 16/30
25/25 — 0s 6ms/step - loss: 8.7337 - mae: 2.5540 - val_loss: 8.2045 - val_mae: 2.4630
```

Dự đoán điểm số:

```
[4]: # Dự đoán điểm số
prediction = model.predict(np.array([[8,7,9]]))
print("Predictions: ", prediction)

1/1 — 0s 253ms/step
Predictions: [[4.573632]]
```

Lưu mô hình dưới dạng HDF5:

```
Predictions: [[4.573632]]

[5]: import pickle
#---save the model to disk---
filename = 'cuoicungdiemthi.sav'
#---write to the file using write and binary mode---
pickle.dump(model, open(filename, 'wb'))
```

Xây dựng mô hình CNN:

```
[6]: from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv1D, MaxPooling1D, Flatten, Dense, Dropout

# Reshape dữ liệu để phù hợp với CNN (samples, features, 1)
X_train_cnn = X_train.reshape(X_train.shape[0], X_train.shape[1], 1) # (1000, 3, 1)
X_test_cnn = X_test.reshape(X_test.shape[0], X_test.shape[1], 1) # (200, 3, 1)

# Xây dựng mô hình CNN
def create_cnn_model(input_shape):
    model = Sequential()

    # Lớp Conv1D đầu tiên với 64 bộ lọc và kích thước bộ lọc là 1 (thử giảm kích thước bộ lọc)
    model.add(Conv1D(64, 1, activation='relu', input_shape=input_shape))

    # Lớp MaxPooling1D để giảm kích thước
    model.add(MaxPooling1D(1)) # Thử dùng kích thước maxpooling = 1

    # Lớp Conv1D thứ hai với 128 bộ lọc và kích thước bộ lọc là 1
    model.add(Conv1D(128, 1, activation='relu'))

    # Lớp MaxPooling1D thứ hai để giảm kích thước
    model.add(MaxPooling1D(1)) # Thử dùng kích thước maxpooling = 1

    # Lớp Flatten để chuyển đổi dữ liệu từ 2D thành 1D
    model.add(Flatten())

    # Lớp Dense để thêm các lớp hoàn thiện, với 100 neuron và hàm kích hoạt ReLU
    model.add(Dense(100, activation='relu'))

    # Dropout để tránh overfitting
    model.add(Dropout(0.1))

    # Lớp Dense cuối cùng để dự đoán một giá trị liên tục (dùng activation='linear' cho dự đoán số thực)
    model.add(Dense(1)) # Dự đoán một giá trị liên tục

    return model

# Định nghĩa input_shape phù hợp với dữ liệu đầu vào (3 đặc trưng, 1 chiều cho mỗi đặc trưng)
input_shape = (3, 1) # Dữ liệu có 3 đặc trưng

# Tạo mô hình CNN
model_cnn = create_cnn_model(input_shape)
```

Input Layer:

- **Input shape:** (3, 1) - dữ liệu đầu vào có 3 đặc trưng và 1 kênh, tức là mỗi đặc trưng sẽ được coi như một kênh 1 chiều.

Lớp Convolutional đầu tiên (Conv1D):

- **Số bộ lọc:** 64 - lớp này có 64 bộ lọc, nghĩa là nó sẽ tạo ra 64 đặc trưng sau khi tích chập.
- **Kích thước bộ lọc:** 1 - bộ lọc có kích thước 1, cho phép học đặc trưng từ từng phần tử trong chuỗi dữ liệu.
- **Activation:** ReLU - hàm kích hoạt ReLU để tăng tính phi tuyến của mô hình và cho phép mô hình học được các mẫu phức tạp hơn.

Lớp MaxPooling đầu tiên (MaxPooling1D):

- **Kích thước:** 1 - giảm kích thước của các đặc trưng bằng cách lấy giá trị lớn nhất ở mỗi vùng con.
- Mục đích của lớp MaxPooling là giảm độ phức tạp của mô hình, giảm số lượng tham số và tránh overfitting.

Lớp Convolutional thứ hai (Conv1D):

- **Số bộ lọc:** 128 - lớp này có 128 bộ lọc, giúp mô hình học được nhiều đặc trưng phức tạp hơn.
- **Kích thước bộ lọc:** 1, với hàm kích hoạt ReLU tương tự lớp tích chập đầu tiên.

Lớp MaxPooling thứ hai (MaxPooling1D):

- **Kích thước:** 1 - tiếp tục giảm độ phức tạp của dữ liệu đầu ra từ lớp tích chập trước đó. **Lớp**

Flatten:

- Lớp này chuyển đổi dữ liệu từ 2D (các đặc trưng từ các lớp tích chập) sang 1D để có thể đưa vào các lớp Dense ở phía sau. Nó giúp kết nối các đặc trưng trích xuất từ các lớp tích chập với các lớp fully connected (kết nối đầy đủ) tiếp theo.

Dense (Fully Connected Layer):

- **100 neuron** với hàm kích hoạt ReLU - lớp này có nhiệm vụ học các đặc trưng từ các lớp tích chập đã được chuyển đổi thành 1D. Với 100 neuron, lớp này sẽ giúp tăng cường khả năng biểu diễn của mô hình. **Lớp Dropout:**

- **Dropout rate:** 0.1 - có tác dụng ngẫu nhiên bỏ 10% các kết nối giữa các neuron trong quá trình huấn luyện, giúp giảm thiểu overfitting và làm cho mô hình trở nên mạnh mẽ hơn.

Output Layer:

- **1 neuron** - lớp đầu ra chỉ có một neuron, vì bài toán yêu cầu dự đoán một giá trị liên tục.
- Không có hàm kích hoạt (mặc định là hàm tuyến tính linear), phù hợp cho bài toán hồi quy.

Biên dịch và huấn luyện mô hình:

```
# Biên dịch mô hình
model_cnn.compile(optimizer='adam', loss='mean_squared_error', metrics=['mae'])

# Huấn Luyện mô hình
model_cnn.fit(X_train_cnn, y_train, epochs=30, batch_size=32, validation_data=(X_test_cnn, y_test))

# Đánh giá mô hình CNN
loss, mae = model_cnn.evaluate(X_test_cnn, y_test)
print(f"CNN Test loss: {loss}, Test MAE: {mae}")
```

D:\App\Anaconda\Lib\site-packages\keras\src\layers\convolutional\base_conv.py:107: UserWarning: Do not pass an `input_shape`/`input_dim` argument to layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.

```
super().__init__(activity_regularizer=activity_regularizer, **kwargs)

Epoch 1/30
25/25 — 4s 25ms/step - loss: 18.0946 - mae: 3.4771 - val_loss: 9.3227 - val_mae: 2.6055
Epoch 2/30
25/25 — 0s 10ms/step - loss: 10.9245 - mae: 2.7791 - val_loss: 9.0510 - val_mae: 2.5827
Epoch 3/30
25/25 — 0s 9ms/step - loss: 10.2165 - mae: 2.6839 - val_loss: 8.8060 - val_mae: 2.5574
Epoch 4/30
25/25 — 0s 10ms/step - loss: 9.6951 - mae: 2.6134 - val_loss: 8.5919 - val_mae: 2.5398
Epoch 5/30
25/25 — 0s 12ms/step - loss: 9.0692 - mae: 2.5608 - val_loss: 8.4986 - val_mae: 2.6235
Epoch 6/30
25/25 — 0s 13ms/step - loss: 9.0785 - mae: 2.5641 - val_loss: 8.3420 - val_mae: 2.4933
Epoch 7/30
25/25 — 0s 13ms/step - loss: 8.1771 - mae: 2.4261 - val_loss: 8.3461 - val_mae: 2.4807
Epoch 8/30
25/25 — 0s 13ms/step - loss: 8.2572 - mae: 2.4568 - val_loss: 8.3801 - val_mae: 2.4834
Epoch 9/30
25/25 — 0s 12ms/step - loss: 8.4993 - mae: 2.4887 - val_loss: 8.5532 - val_mae: 2.5009
Epoch 10/30
25/25 — 0s 12ms/step - loss: 8.9857 - mae: 2.5481 - val_loss: 8.4116 - val_mae: 2.5007
Epoch 11/30
```

Xây dựng mô hình RNN:


```
[9]: from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import SimpleRNN, Dense, Dropout

# Reshape dữ liệu để phù hợp với RNN (samples, time_steps, features)
X_train_rnn = X_train.reshape(X_train.shape[0], X_train.shape[1], 1) # (1000, 3, 1)
X_test_rnn = X_test.reshape(X_test.shape[0], X_test.shape[1], 1) # (200, 3, 1)

# Xây dựng mô hình RNN với 7 Lớp
def create_rnn_model(input_shape):
    model = Sequential()

    # Thêm 5 lớp SimpleRNN với 64 đơn vị và hàm kích hoạt 'relu'
    for _ in range(5):
        model.add(SimpleRNN(64, activation='relu', return_sequences=True, input_shape=input_shape))

    # Thêm 1 lớp Dropout để tránh overfitting
    model.add(Dropout(0.1))

    # Lớp Dense để dự đoán một giá trị liên tục
    model.add(Dense(1)) # Dự đoán một giá trị liên tục

    return model

# Định nghĩa input_shape cho RNN (3 đặc trưng)
input_shape = (3, 1)

# Tạo mô hình RNN
model_rnn = create_rnn_model(input_shape)
```

X_train.reshape(): Dữ liệu huấn luyện X_train được thay đổi kích thước sao cho phù hợp với đầu vào của mô hình RNN. Cụ thể, dữ liệu sẽ có dạng (số mẫu, số bước thời gian, số đặc trưng).

- X_train.shape[0]: Số lượng mẫu trong tập huấn luyện.
- X_train.shape[1]: Số bước thời gian, tức là số lượng các giá trị trong một chuỗi thời gian.
- 1: Số đặc trưng (features) tại mỗi bước thời gian (ở đây chỉ có 1 đặc trưng).
- Hàm create_rnn_model(input_shape) định nghĩa một mô hình RNN mới sử dụng Sequential (mô hình tuần tự) trong Keras

Vòng lặp 5 lần: Thêm 5 lớp SimpleRNN, mỗi lớp có 64 đơn vị (neuron).

- activation='relu': Sử dụng hàm kích hoạt ReLU, giúp tăng cường khả năng học của mô hình.
- return_sequences=True: Đảm bảo mỗi lớp RNN trả về chuỗi giá trị (sequence) thay vì chỉ trả về giá trị cuối cùng của chuỗi, điều này cần thiết để chồng các lớp RNN lên nhau.
- input_shape=input_shape: Lớp đầu tiên nhận đầu vào có dạng (3, 1), tức là 3 bước thời gian và 1 đặc trưng.

Lớp Dropout: Thêm một lớp dropout với tỷ lệ 0.1, có nghĩa là trong quá trình huấn luyện, 10% các đơn vị trong lớp này sẽ bị bỏ qua ngẫu nhiên, nhằm tránh hiện tượng overfitting (quá khớp mô hình với dữ liệu huấn luyện).

Lớp Dense: Thêm một lớp Dense với 1 đơn vị, dùng để dự đoán một giá trị liên tục (do bài toán của bạn là hồi quy).

Xây dựng mô hình LSTM:

```
[10]: from tensorflow.keras.models import Sequential
      from tensorflow.keras.layers import LSTM, Dense, Dropout

      # Reshape dữ liệu để phù hợp với LSTM (samples, time_steps, features)
      X_train_lstm = X_train.reshape(X_train.shape[0], X_train.shape[1], 1) # (1000, 3, 1)
      X_test_lstm = X_test.reshape(X_test.shape[0], X_test.shape[1], 1) # (200, 3, 1)

      # Xây dựng mô hình LSTM với 7 Lớp
      def create_lstm_model(input_shape):
          model = Sequential()

          # Thêm 5 Lớp LSTM với 64 đơn vị và hàm kích hoạt 'relu'
          for _ in range(5):
              model.add(LSTM(64, activation='relu', return_sequences=True, input_shape=input_shape))

          # Thêm 1 Lớp Dropout để tránh overfitting
          model.add(Dropout(0.1))

          # Lớp Dense để dự đoán một giá trị liên tục
          model.add(Dense(1)) # Dự đoán một giá trị liên tục

          return model

      # Định nghĩa input_shape cho LSTM (3 đặc trưng)
      input_shape = (3, 1)

      # Tạo mô hình LSTM
      model_lstm = create_lstm_model(input_shape)

      # Đánh giá mô hình
```

LSTM Layer:

- Mô hình sử dụng 5 lớp **LSTM** (Long Short-Term Memory). Mỗi lớp LSTM có **64 đơn vị** (neuron). LSTM đặc biệt hiệu quả trong việc học và dự đoán các chuỗi thời gian dài, vì nó có khả năng ghi nhớ và sử dụng thông tin từ các bước thời gian trước đó.
- **activation='relu'**: Mỗi đơn vị trong LSTM sử dụng hàm kích hoạt **ReLU** (Rectified Linear Unit), một hàm kích hoạt phổ biến, giúp cải thiện tốc độ huấn luyện và giảm vấn đề vanishing gradient.
- **return_sequences=True**: Đảm bảo mỗi lớp LSTM trả về toàn bộ chuỗi kết quả thay vì chỉ trả về giá trị của bước thời gian cuối cùng. Điều này là cần thiết vì mô hình có nhiều lớp LSTM chồng lên nhau.

Dropout:

- Lớp **Dropout** với tỷ lệ **10%** (0.1) được thêm vào sau các lớp LSTM. Dropout giúp giảm hiện tượng **overfitting** (quá khớp mô hình với dữ liệu huấn luyện) bằng cách "bỏ qua" ngẫu nhiên một số đơn vị trong quá trình huấn luyện.

Dense Layer:

- Lớp cuối cùng của mô hình là một lớp **Dense** (mật độ), với 1 đơn vị, dùng để dự đoán giá trị liên tục (vì đây là bài toán hồi quy). Lớp này không sử dụng hàm kích hoạt, vì bạn đang thực hiện một bài toán hồi quy và muốn dự đoán một giá trị liên tục.

Xây dựng web dự đoán điểm:

Rest_Api.py

```
1 import pickle
2 from flask import Flask, request, jsonify, send_file
3 import numpy as np
4 from flask_cors import CORS
5
6 app = Flask(__name__)
7 CORS(app)
8 # ---the filename of the saved model---
9 filename = 'cuoicungdiemthi.sav'
10 filename_cnn = 'cuoicungdiemthi_cnn.sav'
11 filename_rnn = 'cuoicungdiemthi_rnn.sav'
12 filename_lstm = 'cuoicungdiemthi_lstm.sav'
13
14 # ---load the saved model---
15 loaded_model = pickle.load(open(filename, 'rb'))
16 loaded_model_cnn = pickle.load(open(filename_cnn, 'rb'))
17 loaded_model_rnn = pickle.load(open(filename_rnn, 'rb'))
18 loaded_model_lstm = pickle.load(open(filename_lstm, 'rb'))
19
20 @app.route('/')
21 def home():
22     return send_file('predict.html')
23
24 @app.route('/diabetes/v1/predict', methods=['POST'])
25 def predict():
26     # ---get the features to predict---
27     features = request.json
28     # ---create the features list for prediction---
29     features_list = [
30         float(features["1"]),
31         float(features["2"]),
32         float(features["3"])
33     ]
34
35 @app.route('/')
36 def home():
37     return send_file('predict.html')
38
39 @app.route('/diabetes/v1/predict', methods=['POST'])
40 def predict():
41     # ---get the features to predict---
42     features = request.json
43     # ---create the features list for prediction---
44     features_list = [
45         float(features["1"]),
46         float(features["2"]),
47         float(features["3"])
48     ]
49
50     # ---get the prediction class---
51     prediction = loaded_model.predict(np.array([features_list]))
52     prediction_cnn = loaded_model_cnn.predict(np.array([features_list]))
53     prediction_rnn = loaded_model_rnn.predict(np.array([features_list]))
54     prediction_lstm = loaded_model_lstm.predict(np.array([features_list]))
55     prediction = float(prediction[0][0])
56     prediction_cnn = float(prediction_cnn[0][0])
57     prediction_rnn = float(prediction_rnn[0][2][0])
58     prediction_lstm = float(prediction_lstm[0][2][0])
59     response = {
60         'prediction': prediction,
61         'prediction_cnn': prediction_cnn,
62         'prediction_rnn': prediction_rnn,
63         'prediction_lstm': prediction_lstm
64     }
65     # 'confidence': str(round(np.amax(confidence[0]) * 100, 2))
66
67     return jsonify(response)
68
69 if __name__ == '__main__':
70     app.run(host='0.0.0.0', port=5000)
```

Predict.html:

Diabetes Prediction

Select Subject:

Subject 1

Field 1:

5

Field 2:

9

Field 3:

10

Select Model:

RNN Model

Predict

Prediction Result:

C

Bài 3: Xây dựng Hệ dự đoán giá nhà Hà Đông với Data cho trên trang face:

Đọc dữ liệu nhadathadong.csv:

```
import pandas as pd

data = pd.read_csv('nhadathadong.csv')
display(data)
```

0	92 m ²	92.0	NaN	3 PN	3 phòng	NaN	NaN	NaN	20.979528	Hợp đồng mua bán	105.787567	3,4 tỷ	3.40	~36,96 triệu/m ²
1	100 m ²	100.0	NaN	NaN	NaN	NaN	NaN	5 m	20.979240	NaN	105.741783	20 tỷ	20.00	~200 triệu/m ²
2	80 m ²	80.0	NaN	NaN	NaN	Tây - Bắc	NaN	4 m	20.979240	Sổ đỏ	105.741783	16,3 tỷ	16.30	~203,75 triệu/m ²
3	60 m ²	60.0	NaN	NaN	NaN	Tây - Bắc	NaN	5 m	20.968100	Sổ đỏ/ Sổ hồng	105.754989	137,5 triệu/m ²	NaN	~8,25 tỷ
4	75,3 m ²	NaN	NaN	NaN	NaN	NaN	NaN	NaN	20.984566	NaN	105.785019	35 tỷ	35.00	~464,81 triệu/m ²
...
66	82,5 m ²	NaN	NaN	NaN	NaN	Đông - Nam	4 tầng	NaN	20.973156	SĐCC	105.757645	20 tỷ	20.00	~242,42 triệu/m ²
67	79,4 m ²	NaN	NaN	2 PN	2 phòng	NaN	NaN	NaN	20.974472	Sổ đỏ/ Sổ hồng	105.760880	3,9 tỷ	3.90	~49,12 triệu/m ²
68	144 m ²	144.0	NaN	NaN	NaN	Tây - Bắc	4 tầng	6 m	20.979240	Sổ đỏ/ Sổ hồng	105.741783	23 tỷ	23.00	~159,72 triệu/m ²
69	75 m ²	75.0	NaN	5 PN	5 phòng	NaN	5 tầng	NaN	20.959990	Sổ đỏ/ Sổ hồng	105.735924	10 tỷ	10.00	~133,33 triệu/m ²
70	67 m ²	67.0	Đông - Nam	2 PN	2 phòng	Tây - Bắc	NaN	NaN	20.939703	Đang chờ sổ	105.787254	2,39 tỷ	2.39	~35,67 triệu/m ²

71 rows × 19 columns

Làm sạch dữ liệu:

Bước 1:

```
columns_to_drop = ['bedroomCount', 'priceExt', 'priceBil', 'area', 'balconyDirection', 'priceVnd', 'road', 'floorCount']
data_cleaned = data.drop(columns=columns_to_drop)
display(data_cleaned)
```

	areaM2	bedroom	direction	frontage	lat	legal	long	price	priceMil	pricePerM2	toiletCount
0	92.0	3 PN	NaN	NaN	20.979528	Hợp đồng mua bán	105.787567	3,4 tỷ	3400	36.956522	2 phòng
1	100.0	NaN	NaN	5 m	20.979240	NaN	105.741783	20 tỷ	20000	200.000000	NaN
2	80.0	NaN	Tây - Bắc	4 m	20.979240	Sổ đỏ	105.741783	16,3 tỷ	16300	203.750000	NaN
3	60.0	NaN	Tây - Bắc	5 m	20.968100	Sổ đỏ/ Sổ hồng	105.754989	137,5 triệu/m ²	-1	-0.016667	NaN
4	NaN	NaN	NaN	NaN	20.984566	NaN	105.785019	35 tỷ	35000	NaN	NaN
...
66	NaN	NaN	Đông - Nam	NaN	20.973156	SĐCC	105.757645	20 tỷ	20000	NaN	NaN
67	NaN	2 PN	NaN	NaN	20.974472	Sổ đỏ/ Sổ hồng	105.760880	3,9 tỷ	3900	NaN	2 phòng
68	144.0	NaN	Tây - Bắc	6 m	20.979240	Sổ đỏ/ Sổ hồng	105.741783	23 tỷ	23000	159.722222	NaN
69	75.0	5 PN	NaN	NaN	20.959990	Sổ đỏ/ Sổ hồng	105.735924	10 tỷ	10000	133.333333	5 phòng
70	67.0	2 PN	Tây - Bắc	NaN	20.939703	Đang chờ sổ	105.787254	2,39 tỷ	2390	35.671642	2 phòng

71 rows × 11 columns

1. Xác định các cột cần loại bỏ:

columns_to_drop = ['bedroomCount', 'priceExt', 'priceBil', 'area', 'balconyDirection', 'priceVnd', 'road', 'floorCount']

- **columns_to_drop:** Đây là một danh sách chứa tên các cột mà bạn muốn loại bỏ khỏi DataFrame data. Các cột này bao gồm thông tin như số lượng phòng ngủ (bedroomCount), giá trị căn hộ ở dạng khác nhau (priceExt, priceBil, priceVnd), diện tích (area), hướng ban công (balconyDirection), thông tin về đường (road), và số tầng (floorCount).
- **Lý do loại bỏ:** Các cột này có thể không cần thiết cho quá trình phân tích hoặc mô hình hóa của bạn, hoặc có thể chứa các giá trị không hợp lý, không có ý nghĩa, hoặc dữ liệu không đầy đủ.

2. Loại bỏ các cột khỏi DataFrame:

`data_cleaned = data.drop(columns=columns_to_drop)`

- **data.drop(columns=columns_to_drop):** Đây là lệnh để loại bỏ các cột có tên trong danh sách `columns_to_drop` từ DataFrame `data`.
 - **data:** DataFrame ban đầu, chứa dữ liệu bạn đang làm việc.
 - **.drop():** Phương thức của DataFrame dùng để loại bỏ các cột hoặc hàng.
 - **columns=columns_to_drop:** Chỉ định các cột cần loại bỏ, trong trường hợp này là các cột trong danh sách `columns_to_drop`.
- **data_cleaned:** Sau khi loại bỏ các cột, kết quả được lưu vào một DataFrame mới gọi là `data_cleaned`. DataFrame này chứa dữ liệu đã được làm sạch, với các cột không cần thiết đã bị loại bỏ.

3. Hiển thị DataFrame đã làm sạch:

`display(data_cleaned)`

- **display(data_cleaned):** Đây là lệnh để hiển thị DataFrame `data_cleaned` (dữ liệu sau khi đã loại bỏ các cột không cần thiết). Hàm `display()` thường được sử dụng trong các môi trường như Jupyter Notebooks để hiển thị dữ liệu dưới dạng bảng dễ đọc.

Bước 2:

```

import pandas as pd
import numpy as np

df = data_cleaned

# Xóa các ký tự không cần thiết và chuyển đổi sang kiểu số cho các trường thích hợp
def clean_numeric_column(value, remove_units):
    """ Loại bỏ ký tự không cần thiết và chuyển đổi giá trị thành số """
    if pd.isna(value):
        return np.nan
    for unit in remove_units:
        value = str(value).replace(unit, '').strip() # Loại bỏ đơn vị
    return float(value.replace(',', '.')) # Chuyển thành float

df['toiletCount'] = df['toiletCount'].apply(lambda x: clean_numeric_column(x, ['phòng']))

# Làm sạch trường mặt tiền 'frontage' và chuyển thành số
df['frontage'] = df['frontage'].apply(lambda x: clean_numeric_column(x, ['m']))

# Điền giá trị thiếu cho các trường nếu cần thiết (tùy chọn)
df.fillna({
    'balconyDirection': 'Không xác định', # Thay giá trị thiếu bằng giá trị mặc định
    'direction': 'Không xác định',
    'legal': 'Không có thông tin',
}, inplace=True)

# Kiểm tra lại dữ liệu đã làm sạch
display(df)

```

	areaM2	bedroom	direction	frontage	lat	legal	long	price	priceMil	pricePerM2	toiletCount
0	92.0	3 PN	Không xác định	NaN	20.979528	Hợp đồng mua bán	105.787567	3,4 tỷ	3400	36.956522	2.0
1	100.0	NaN	Không xác định	5.0	20.979240	Không có thông tin	105.741783	20 tỷ	20000	200.000000	NaN
2	80.0	NaN	Tây - Bắc	4.0	20.979240	Sổ đỏ	105.741783	16,3 tỷ	16300	203.750000	NaN
3	60.0	NaN	Tây - Bắc	5.0	20.968100	Sổ đỏ/ Sổ hồng	105.754989	137,5 triệu/m²	-1	-0.016667	NaN
4	NaN	NaN	Không xác định	NaN	20.984566	Không có thông tin	105.785019	35 tỷ	35000	NaN	NaN
...
66	NaN	NaN	Đông - Nam	NaN	20.973156	SDCC	105.757645	20 tỷ	20000	NaN	NaN
67	NaN	2 PN	Không xác định	NaN	20.974472	Sổ đỏ/ Sổ hồng	105.760880	3,9 tỷ	3900	NaN	2.0
68	144.0	NaN	Tây - Bắc	6.0	20.979240	Sổ đỏ/ Sổ hồng	105.741783	23 tỷ	23000	159.722222	NaN
69	75.0	5 PN	Không xác định	NaN	20.959990	Sổ đỏ/ Sổ hồng	105.735924	10 tỷ	10000	133.333333	5.0
70	67.0	2 PN	Tây - Bắc	NaN	20.939703	Đang chờ sổ	105.787254	2,39 tỷ	2390	35.671642	2.0

71 rows × 11 columns

1. Khởi tạo DataFrame và thư viện:

```
import pandas as pd
```

```
import numpy as np
```

```
df = data_cleaned
```

- **import pandas as pd** và **import numpy as np**: Nhập các thư viện **Pandas** và **Numpy**, hai thư viện phổ biến để xử lý và phân tích dữ liệu.
- **df = data_cleaned**: Gán DataFrame `data_cleaned` (đã được làm sạch ở phần trước) cho biến `df` để tiếp tục xử lý và làm sạch thêm.

2. Hàm `clean_numeric_column`:

```
def clean_numeric_column(value, remove_units):
```

```
    """ Loại bỏ ký tự không cần thiết và chuyển đổi giá trị thành số """
```

```
    if pd.isna(value):
```

```

return np.nan
for unit in remove_units:
    value = str(value).replace(unit, "").strip() # Loại bỏ đơn vị
return float(value.replace(',', '.')) # Chuyển thành float

```

- **clean_numeric_column(value, remove_units):** Hàm này dùng để làm sạch và chuyển đổi một giá trị thành kiểu số (float). Các bước trong hàm này:
 - **pd.isna(value):** Kiểm tra nếu giá trị là NaN (không có dữ liệu). Nếu có, trả về NaN.
 - **for unit in remove_units::** Duyệt qua danh sách các đơn vị (như 'phòng', 'm', ...) và loại bỏ chúng khỏi giá trị bằng cách thay thế chúng bằng chuỗi rỗng.
 - **value.replace(',', '.):** Thay thế dấu phẩy (,) bằng dấu chấm (.) để chuẩn hóa định dạng số (nhất là khi dữ liệu có dấu phẩy thay cho dấu chấm thập phân).
 - **float(value):** Chuyển giá trị sau khi đã làm sạch thành kiểu số thực (float).

3. Làm sạch và chuyển đổi các cột:

```
df['toiletCount'] = df['toiletCount'].apply(lambda x: clean_numeric_column(x, ['phòng']))
```

- **Làm sạch cột toiletCount:**
 - Cột toiletCount có thể chứa các giá trị kiểu chuỗi với đơn vị, ví dụ như '2 phòng'. Hàm clean_numeric_column sẽ loại bỏ đơn vị 'phòng' và chuyển giá trị còn lại thành số thực.
 - **apply(lambda x: clean_numeric_column(x, ['phòng'])):** Áp dụng hàm clean_numeric_column cho mỗi giá trị trong cột toiletCount.

python

Sao chép mã

```
df['frontage'] = df['frontage'].apply(lambda x: clean_numeric_column(x, ['m']))
```

- **Làm sạch cột frontage:** Tương tự, cột frontage có thể chứa đơn vị 'm' (mét). Hàm clean_numeric_column sẽ loại bỏ đơn vị 'm' và chuyển đổi giá trị còn lại thành kiểu số.

4. Điền giá trị thiếu (Missing Values):

```

df.fillna({
    'balconyDirection': 'Không xác định', # Thay giá trị thiếu bằng giá trị mặc định
    'direction': 'Không xác định',
    'legal': 'Không có thông tin',
}, inplace=True)

```

- **df.fillna({...}, inplace=True):** Phương thức này được sử dụng để thay thế các giá trị thiếu (NaN) trong các cột nhất định bằng các giá trị mặc định.
 - **'balconyDirection': 'Không xác định':** Nếu cột balconyDirection có giá trị thiếu, nó sẽ được thay thế bằng 'Không xác định'.
 - **'direction': 'Không xác định':** Tương tự cho cột direction.
 - **'legal': 'Không có thông tin':** Và thay thế giá trị thiếu trong cột legal bằng 'Không có thông tin'.

- **inplace=True**: Điều này có nghĩa là việc thay đổi sẽ được thực hiện trực tiếp trên DataFrame df mà không tạo ra một bản sao mới.

5. Hiển thị dữ liệu đã làm sạch:

display(df)

- **display(df)**: Cuối cùng, lệnh này hiển thị DataFrame df sau khi đã được làm sạch, giúp bạn kiểm tra lại dữ liệu sau khi đã xử lý các giá trị thiếu và làm sạch các cột.

Bước 3:

```
# Hàm làm sạch
def clean_numeric(value, units):
    """ Loại bỏ các đơn vị và dấu thập phân để chuyển sang kiểu số """
    if pd.isna(value):
        return np.nan
    for unit in units:
        value = value.replace(unit, '').strip()
    try:
        return float(value.replace('.', ''))
    except ValueError:
        return np.nan # Trả về NaN nếu không chuyển đổi được thành số

# Làm sạch cột 'price', 'priceMil', và 'pricePerM2'
df['price'] = df['price'].apply(lambda x: clean_numeric(str(x), ['tỷ', 'triệu/m²']))
df['priceMil'] = df['priceMil'].replace(-1, np.nan) # Thay -1 bằng NaN
df['pricePerM2'] = df['pricePerM2'].replace(-0.016667, np.nan) # Thay -0.016667 bằng NaN

# Làm sạch cột 'frontage'
df['frontage'] = df['frontage'].apply(lambda x: clean_numeric(str(x), ['m']))

# Làm sạch số phòng ngủ 'bedroom' và 'toiletCount'
df['bedroom'] = df['bedroom'].apply(lambda x: clean_numeric(str(x), ['PN', 'phòng']))
df['toiletCount'] = df['toiletCount'].apply(lambda x: clean_numeric(str(x), ['phòng']))

# Điền giá trị mặc định cho các trường nếu cần thiết
df.fillna({
    'direction': 'Không xác định',
    'legal': 'Không có thông tin',
}, inplace=True)

df['direction'].fillna('Không xác định', inplace=True)
df['legal'].fillna('Không có thông tin', inplace=True)

# 2. Điền giá trị trung bình cho các cột dạng số
df['areaM2'].fillna(df['areaM2'].mean(), inplace=True)
df['frontage'].fillna(df['frontage'].mean(), inplace=True)
df['price'].fillna(df['price'].mean(), inplace=True)
df['priceMil'].fillna(df['priceMil'].mean(), inplace=True)
df['pricePerM2'].fillna(df['pricePerM2'].mean(), inplace=True)

# 3. Điền số 0 cho các cột như số phòng ngủ và số phòng vệ sinh
df['bedroom'].fillna(0, inplace=True)
df['toiletCount'].fillna(0, inplace=True)

# Kết quả làm sạch
display(df)
```

	areaM2	bedroom	direction	frontage	lat	legal	long	price	priceMil	pricePerM2	toiletCount
0	92.000000	3.0	Không xác định	6.3405	20.979528	Hợp đồng mua bán	105.787567	3.40	3400.000000	36.956522	2.0
1	100.000000	0.0	Không xác định	5.0000	20.979240	Không có thông tin	105.741783	20.00	20000.000000	200.000000	0.0
2	80.000000	0.0	Tây - Bắc	4.0000	20.979240	Sổ đỏ	105.741783	16.30	16300.000000	203.750000	0.0
3	60.000000	0.0	Tây - Bắc	5.0000	20.968100	Sổ đỏ/ Sổ hồng	105.754989	137.50	15865.862069	-0.016667	0.0
4	100.065574	0.0	Không xác định	6.3405	20.984566	Không có thông tin	105.785019	35.00	35000.000000	134.998558	0.0
...
66	100.065574	0.0	Đông - Nam	6.3405	20.973156	ĐCCC	105.757645	20.00	20000.000000	134.998558	0.0
67	100.065574	2.0	Không xác định	6.3405	20.974472	Sổ đỏ/ Sổ hồng	105.760880	3.90	3900.000000	134.998558	2.0
68	144.000000	0.0	Tây - Bắc	6.0000	20.979240	Sổ đỏ/ Sổ hồng	105.741783	23.00	23000.000000	159.722222	0.0
69	75.000000	5.0	Không xác định	6.3405	20.959990	Sổ đỏ/ Sổ hồng	105.735924	10.00	10000.000000	133.333333	5.0
70	67.000000	2.0	Tây - Bắc	6.3405	20.939703	Đang chờ sổ	105.787254	2.39	2390.000000	35.671642	2.0

71 rows × 11 columns

Hàm clean_numeric: Loại bỏ các đơn vị không cần thiết và dấu thập phân, sau đó chuyển giá trị thành kiểu số (float). Nếu không thể chuyển đổi, trả về NaN.

Làm sạch các cột có dữ liệu kiểu chuỗi:

- Cột price, priceMil, pricePerM2 được làm sạch bằng cách loại bỏ các đơn vị như 'tỷ', 'triệu/m²', và thay giá trị không hợp lệ (-1, -0.016667) bằng NaN.
- Cột frontage loại bỏ đơn vị 'm'.
- Cột số phòng ngủ (bedroom) và số phòng vệ sinh (toiletCount) loại bỏ đơn vị 'PN', 'phòng'.

Điền giá trị thiếu:

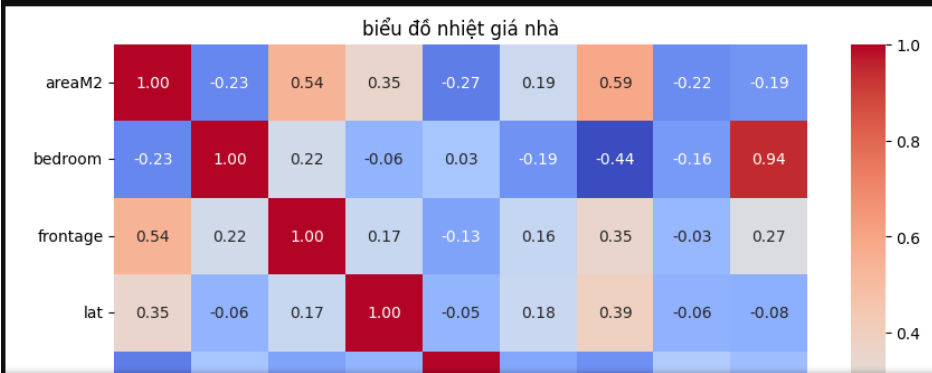
- Điền giá trị mặc định 'Không xác định' cho cột direction và 'Không có thông tin' cho cột legal.
- Điền giá trị trung bình vào các cột dạng số (areaM2, frontage, price, priceMil, pricePerM2).
- Điền giá trị 0 cho các cột số phòng ngủ và phòng vệ sinh (bedroom, toiletCount).

Hiển thị kết quả: Cuối cùng, hiển thị DataFrame đã được làm sạch.

Hiện thị biểu đồ nhiệt giá nhà:

```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

num_df = df.select_dtypes(include=[np.number])
plt.figure(figsize=(10,8))
sns.heatmap(num_df.corr(), annot=True, cmap='coolwarm', fmt='.2f')
plt.title('biểu đồ nhiệt giá nhà')
plt.show()
```



Xây dựng các biểu đồ liên quan với dữ liệu đã được làm sạch:

```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error

# Giả sử df là DataFrame của bạn và đã được làm sạch
# Kiểm tra tên cột
print(df.columns)

# Giả sử bạn muốn sử dụng các cột sau:
# Sửa lại tên cột cho đúng
correlation_matrix = df[['areaM2', 'lat', 'bedroom', 'price']].corr()

# Vẽ biểu đồ phân tán
sns.pairplot(df[['areaM2', 'lat', 'bedroom', 'price']])
plt.show()

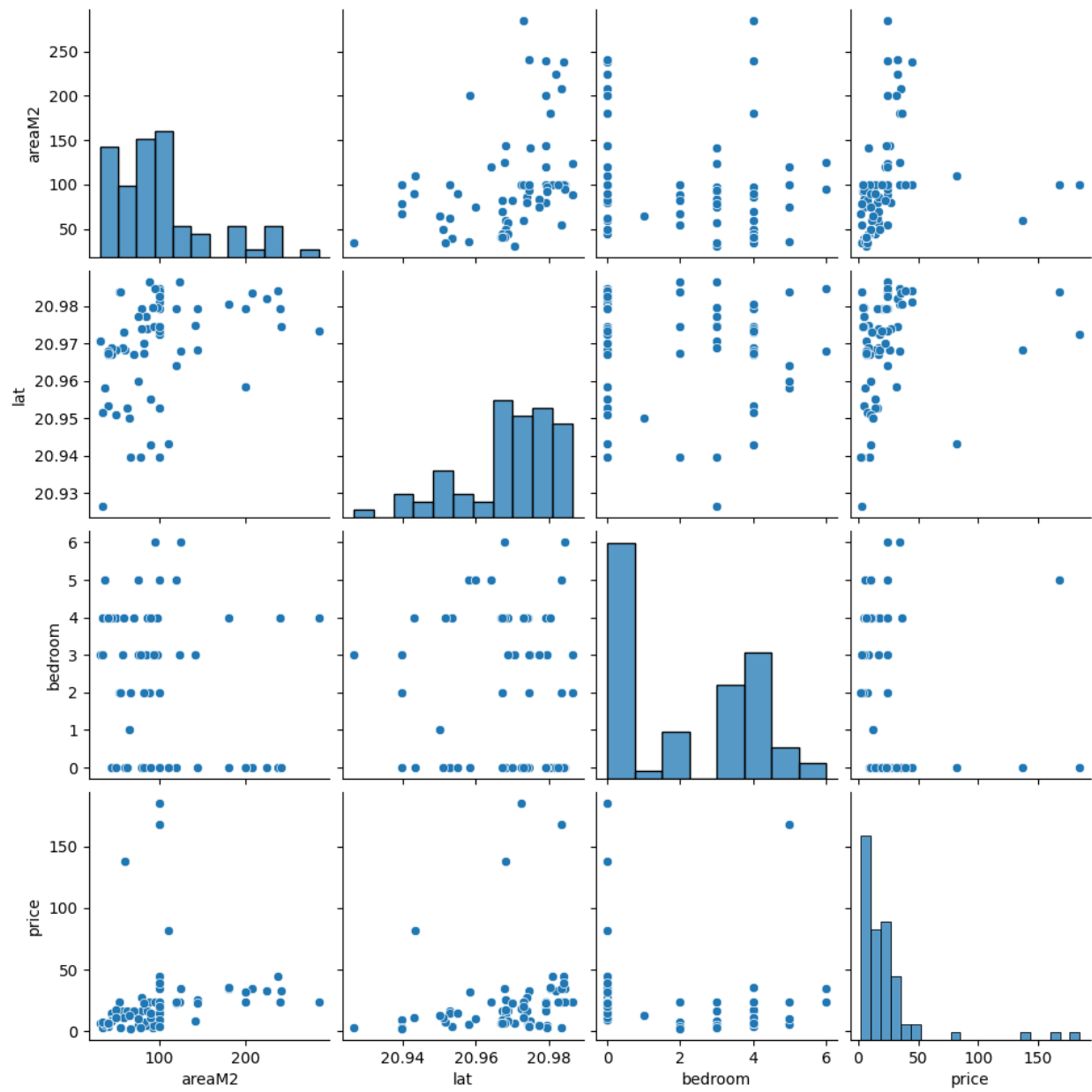
# Tạo biểu đồ nhiệt độ
plt.figure(figsize=(8, 6))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', fmt='.2f')
plt.title('Ma trận Tương Quan')
plt.show()

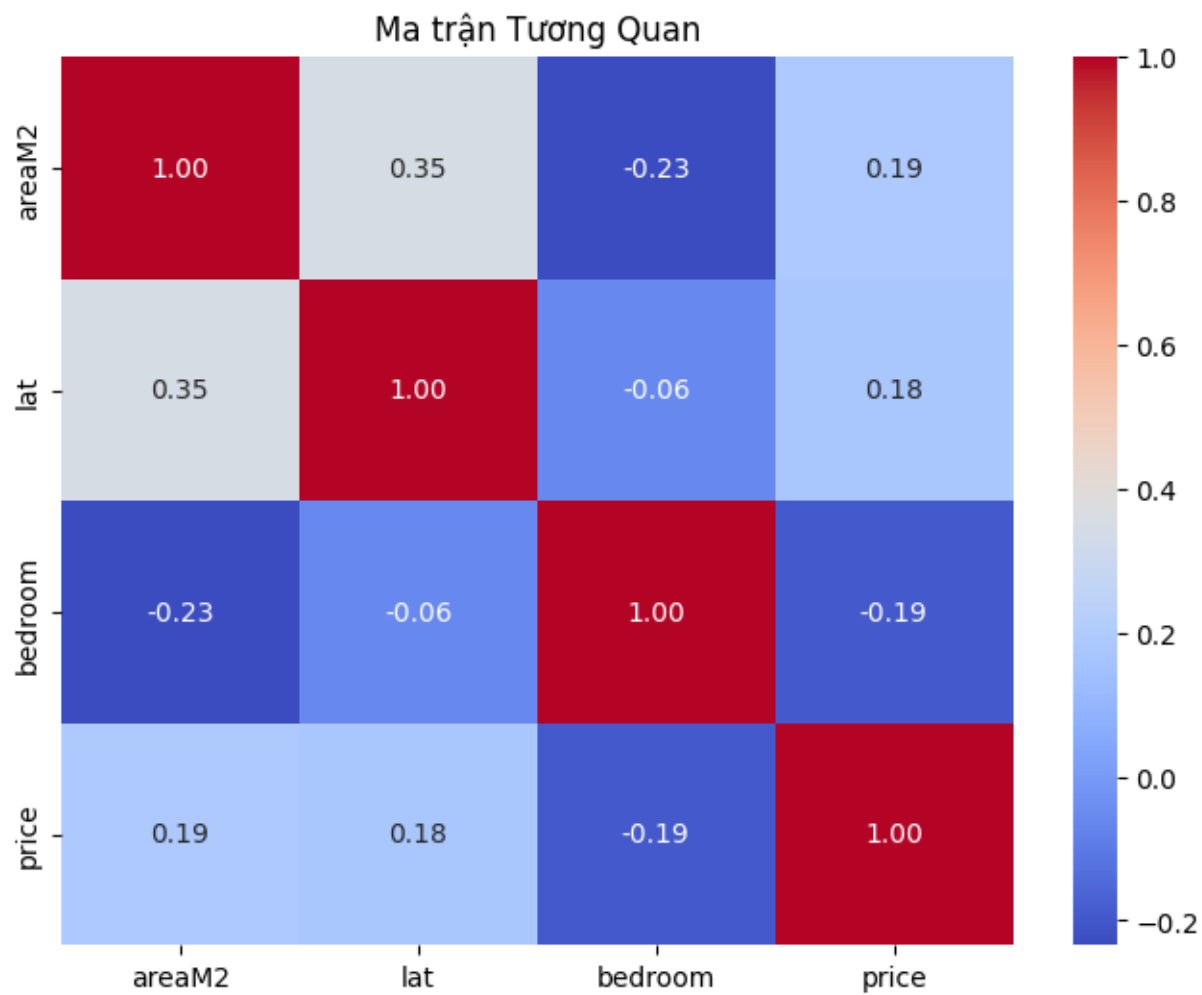
# X và y cho mô hình dự đoán
X = df[['lat', 'bedroom', 'areaM2']]
y = df['price']

# Chia tập dữ liệu thành tập huấn luyện và tập kiểm tra
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Nếu bạn muốn kiểm tra độ chính xác, bạn có thể thêm một mô hình ở đây
# Ví dụ: sử dụng hồi quy tuyến tính
from sklearn.linear_model import LinearRegression

model = LinearRegression()
model.fit(X_train, y_train)
```





Sử dụng CNN, RNN, LSTM với 7_layer để đánh giá mô hình:

```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_absolute_error, mean_squared_error
import tensorflow as tf
from tensorflow.keras import layers, models

# Giả sử df là DataFrame của bạn đã được làm sạch
# Thay thế 'areaM2' bằng tên cột mục tiêu của bạn
X = df[['lat', 'bedroom', 'areaM2']]
y = df['price']

# Chia dữ liệu thành tập huấn luyện và kiểm tra
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# --- Mô hình CNN ---
def create_cnn_model(input_shape):
    model = models.Sequential()
    model.add(layers.Conv1D(32, kernel_size=2, activation='relu', input_shape=input_shape))
    model.add(layers.MaxPooling1D(pool_size=2))
    model.add(layers.Flatten())
    model.add(layers.Dense(64, activation='relu'))
    model.add(layers.Dense(1)) # Đầu ra cho dự đoán số liệu
    return model

# Chuyển đổi X_train, X_test thành dạng phù hợp cho CNN
X_train_cnn = np.expand_dims(X_train.values, axis=2)
X_test_cnn = np.expand_dims(X_test.values, axis=2)

cnn_model = create_cnn_model((X_train_cnn.shape[1], 1))
cnn_model.compile(optimizer='adam', loss='mean_squared_error')
cnn_model.fit(X_train_cnn, y_train, epochs=50, batch_size=32, verbose=0)

# Dự đoán và đánh giá
y_pred_cnn = cnn_model.predict(X_test_cnn)
cnn_mae = mean_absolute_error(y_test, y_pred_cnn)
cnn_mse = mean_squared_error(y_test, y_pred_cnn)
cnn_rmse = np.sqrt(cnn_mse)

# --- Mô hình RNN ---
def create_rnn_model(input_shape):
    model = models.Sequential()
    model.add(layers.SimpleRNN(32, input_shape=input_shape))
    model.add(layers.Dense(1))
    return model

X_train_rnn = np.expand_dims(X_train.values, axis=2)
X_test_rnn = np.expand_dims(X_test.values, axis=2)

rnn_model = create_rnn_model((X_train_rnn.shape[1], 1))
rnn_model.compile(optimizer='adam', loss='mean_squared_error')
rnn_model.fit(X_train_rnn, y_train, epochs=50, batch_size=32, verbose=0)

# Dự đoán và đánh giá
y_pred_rnn = rnn_model.predict(X_test_rnn)
rnn_mae = mean_absolute_error(y_test, y_pred_rnn)
rnn_mse = mean_squared_error(y_test, y_pred_rnn)
rnn_rmse = np.sqrt(rnn_mse)

# --- Mô hình LSTM ---
def create_lstm_model(input_shape):
    model = models.Sequential()
    model.add(layers.LSTM(32, input_shape=input_shape))
    model.add(layers.Dense(1))
    return model

X_train_lstm = np.expand_dims(X_train.values, axis=2)
X_test_lstm = np.expand_dims(X_test.values, axis=2)

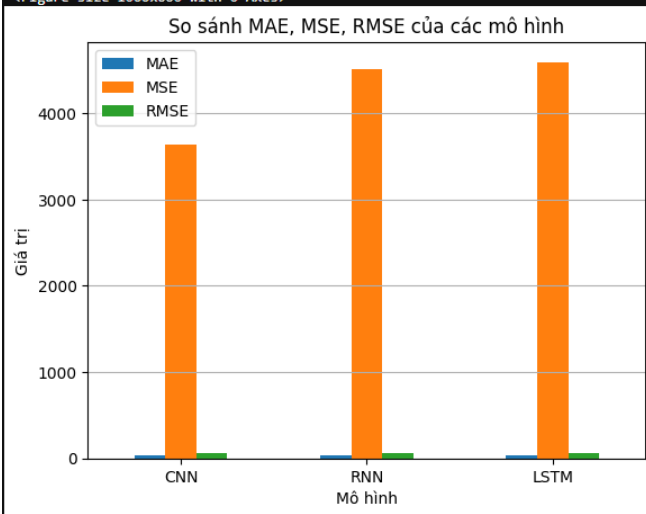
lstm_model = create_lstm_model((X_train_lstm.shape[1], 1))
lstm_model.compile(optimizer='adam', loss='mean_squared_error')
lstm_model.fit(X_train_lstm, y_train, epochs=50, batch_size=32, verbose=0)
```

```
# --- So sánh kết quả ---
results = {
    'Model': ['CNN', 'RNN', 'LSTM'],
    'MAE': [cnn_mae, rnn_mae, lstm_mae],
    'MSE': [cnn_mse, rnn_mse, lstm_mse],
    'RMSE': [cnn_rmse, rnn_rmse, lstm_rmse]
}

results_df = pd.DataFrame(results)

# Vẽ biểu đồ so sánh
plt.figure(figsize=(10, 6))
results_df.plot(x='Model', kind='bar', legend=True)
plt.title('So sánh MAE, MSE, RMSE của các mô hình')
plt.xlabel('Mô hình')
plt.ylabel('Giá trị')
plt.xticks(rotation=0)
plt.grid(axis='y')
plt.show()
```

```
C:\Users\Admin\AppData\Local\Programs\Python\Python311\Lib\site-packages\keras\src\layers\convolutional\base_conv.py:99: UserWarning: Do not pass an `input_shape`/'input_dim' argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(
1/1 — 0s 57ms/step
C:\Users\Admin\AppData\Local\Programs\Python\Python311\Lib\site-packages\keras\src\layers\rnn\rnn.py:204: UserWarning: Do not pass an `input_shape`/'input_dim' argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(**kwargs)
1/1 — 0s 109ms/step
C:\Users\Admin\AppData\Local\Programs\Python\Python311\Lib\site-packages\keras\src\layers\rnn\rnn.py:204: UserWarning: Do not pass an `input_shape`/'input_dim' argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(**kwargs)
1/1 — 0s 111ms/step
<Figure size 1000x600 with 0 Axes>
```



1. Chuẩn bị dữ liệu:

```
X = df[['lat', 'bedroom', 'areaM2']]
```

```
y = df['price']
```

- **X**: Chọn ba đặc trưng (lat, bedroom, areaM2) làm đầu vào cho mô hình.
- **y**: Cột mục tiêu là price (giá trị cần dự đoán).

python

Sao chép mã

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

- Dữ liệu được chia thành **tập huấn luyện** (80%) và **tập kiểm tra** (20%) bằng train_test_split.

2. Mô hình CNN (Convolutional Neural Network):

```
def create_cnn_model(input_shape):
```

```

model = models.Sequential()
model.add(layers.Conv1D(32, kernel_size=2, activation='relu', input_shape=input_shape))
model.add(layers.MaxPooling1D(pool_size=2))
model.add(layers.Flatten())
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(1)) # Đầu ra cho dự đoán số liệu
return model

```

- **Mô hình CNN** sử dụng **Conv1D** để trích xuất đặc trưng từ dữ liệu, tiếp theo là **MaxPooling1D** để giảm kích thước và **Flatten** để chuyển dữ liệu sang dạng 1 chiều, sau đó là **Dense layers** để dự đoán giá trị cuối cùng.
- Dữ liệu X_{train} và X_{test} được chuyển đổi thành dạng phù hợp với CNN bằng cách thêm một chiều ($axis=2$).

python

Sao chép mã

```

cnn_model.fit(X_train_cnn, y_train, epochs=50, batch_size=32, verbose=0)

```

- Mô hình CNN được huấn luyện trong 50 epochs.

3. Mô hình RNN (Recurrent Neural Network):

```

def create_rnn_model(input_shape):

```

```

    model = models.Sequential()
    model.add(layers.SimpleRNN(32, input_shape=input_shape))
    model.add(layers.Dense(1))
    return model

```

- **Mô hình RNN** sử dụng **SimpleRNN** để xử lý dữ liệu chuỗi, phù hợp với các bài toán có dữ liệu theo thời gian. Sau đó là **Dense layer** để dự đoán giá trị.
- Dữ liệu X_{train} và X_{test} cũng được chuyển đổi thành dạng phù hợp cho RNN.

python

Sao chép mã

```

rnn_model.fit(X_train_rnn, y_train, epochs=50, batch_size=32, verbose=0)

```

- Mô hình RNN được huấn luyện trong 50 epochs.

4. Mô hình LSTM (Long Short-Term Memory):

```

def create_lstm_model(input_shape):

```

```

    model = models.Sequential()
    model.add(layers.LSTM(32, input_shape=input_shape))
    model.add(layers.Dense(1))
    return model

```

- **Mô hình LSTM** sử dụng **LSTM layer** để xử lý dữ liệu chuỗi, tiếp theo là **Dense layer** để dự đoán giá trị.
- Dữ liệu X_{train} và X_{test} được chuyển đổi thành dạng phù hợp cho LSTM.

python

Sao chép mã


```
lstm_model.fit(X_train_lstm, y_train, epochs=50, batch_size=32, verbose=0)
```

- Mô hình LSTM được huấn luyện trong 50 epochs.

5. Đánh giá các mô hình:

```
y_pred_cnn = cnn_model.predict(X_test_cnn)
```

```
cnn_mae = mean_absolute_error(y_test, y_pred_cnn)
```

```
cnn_mse = mean_squared_error(y_test, y_pred_cnn)
```

```
cnn_rmse = np.sqrt(cnn_mse)
```

- Dự đoán và tính toán các chỉ số lỗi **MAE**, **MSE**, **RMSE** cho mô hình CNN.

Cách tương tự được thực hiện cho **RNN** và **LSTM**.

6. So sánh kết quả:

```
results = {  
    'Model': ['CNN', 'RNN', 'LSTM'],  
    'MAE': [cnn_mae, rnn_mae, lstm_mae],  
    'MSE': [cnn_mse, rnn_mse, lstm_mse],  
    'RMSE': [cnn_rmse, rnn_rmse, lstm_rmse]  
}
```

```
results_df = pd.DataFrame(results)
```

- Kết quả của các mô hình được lưu vào DataFrame results_df để tiện so sánh.

7. Vẽ biểu đồ so sánh:

```
plt.figure(figsize=(10, 6))
```

```
results_df.plot(x='Model', kind='bar', legend=True)
```

```
plt.title('So sánh MAE, MSE, RMSE của các mô hình')
```

```
plt.xlabel('Mô hình')
```

```
plt.ylabel('Giá trị')
```

```
plt.xticks(rotation=0)
```

```
plt.grid(axis='y')
```

```
plt.show()
```

- Biểu đồ **bar chart** được vẽ để so sánh các chỉ số lỗi MAE, MSE, RMSE của ba mô hình CNN, RNN, và LSTM.

Huấn luyện mô hình và vẽ biểu đồ đường để so sánh loss giữa các mô hình:

```
# --- Huấn luyện mô hình CNN và Lưu lịch sử ---
cnn_history = cnn_model.fit(X_train_cnn, y_train, epochs=50, batch_size=32, verbose=0, validation_split=0.2)

# --- Huấn luyện mô hình RNN và Lưu lịch sử ---
rnn_history = rnn_model.fit(X_train_rnn, y_train, epochs=50, batch_size=32, verbose=0, validation_split=0.2)

# --- Huấn luyện mô hình LSTM và Lưu lịch sử ---
lstm_history = lstm_model.fit(X_train_lstm, y_train, epochs=50, batch_size=32, verbose=0, validation_split=0.2)

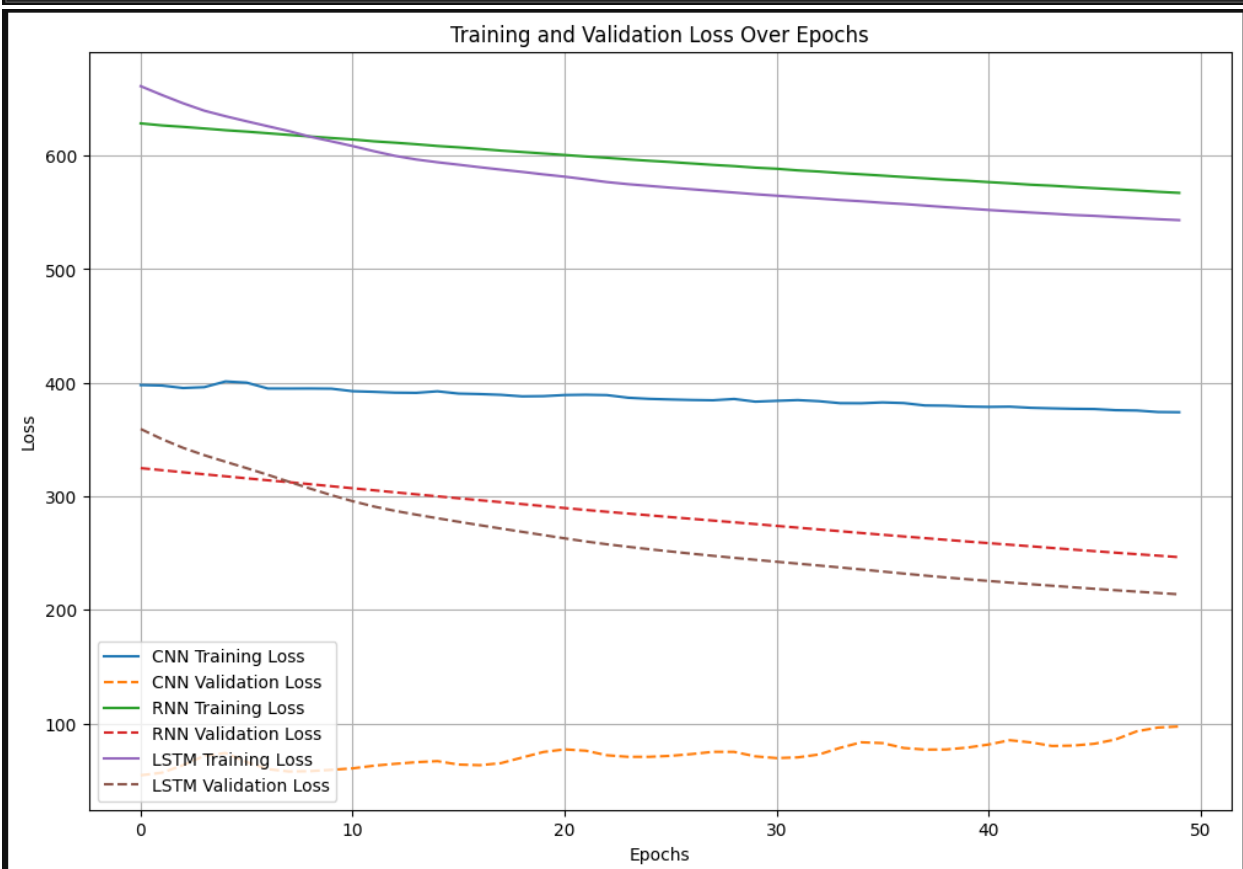
# Vẽ biểu đồ đường để so sánh loss giữa các mô hình
plt.figure(figsize=(12, 8))

# Vẽ Loss của mô hình CNN
plt.plot(cnn_history.history['loss'], label='CNN Training Loss')
plt.plot(cnn_history.history['val_loss'], label='CNN Validation Loss', linestyle='--')

# Vẽ Loss của mô hình RNN
plt.plot(rnn_history.history['loss'], label='RNN Training Loss')
plt.plot(rnn_history.history['val_loss'], label='RNN Validation Loss', linestyle='--')

# Vẽ Loss của mô hình LSTM
plt.plot(lstm_history.history['loss'], label='LSTM Training Loss')
plt.plot(lstm_history.history['val_loss'], label='LSTM Validation Loss', linestyle='--')

# Thiết lập tiêu đề và các thông số cho biểu đồ
plt.title('Training and Validation Loss Over Epochs')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.grid(True)
plt.show()
```



1. Huấn luyện các mô hình và lưu lịch sử:

```
cnn_history = cnn_model.fit(X_train_cnn, y_train, epochs=50, batch_size=32, verbose=0, validation_split=0.2)
```

```
rnn_history = rnn_model.fit(X_train_rnn, y_train, epochs=50, batch_size=32, verbose=0,
validation_split=0.2)
lstm_history = lstm_model.fit(X_train_lstm, y_train, epochs=50, batch_size=32, verbose=0,
validation_split=0.2)
```

- Ba mô hình CNN, RNN và LSTM được huấn luyện trong 50 epochs với kích thước batch là 32.
- **validation_split=0.2**: 20% dữ liệu huấn luyện sẽ được tách ra làm tập validation, giúp theo dõi hiệu suất mô hình trên dữ liệu chưa thấy trong quá trình huấn luyện.
- **verbose=0**: Tắt các thông báo tiến trình trong quá trình huấn luyện.
- Lịch sử huấn luyện của mỗi mô hình được lưu trong các biến `cnn_history`, `rnn_history`, và `lstm_history`. Lịch sử này chứa thông tin về loss, accuracy (nếu có), và `val_loss` trong mỗi epoch.

2. Vẽ biểu đồ so sánh loss giữa các mô hình:

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

- Thiết lập kích thước của biểu đồ với chiều rộng là 12 inch và chiều cao là 8 inch.

```
plt.plot(cnn_history.history['loss'], label='CNN Training Loss')
```

```
plt.plot(cnn_history.history['val_loss'], label='CNN Validation Loss', linestyle='--')
```

- Vẽ đường biểu diễn loss (mất mát) của mô hình CNN trên tập huấn luyện (đường liên tục) và `val_loss` trên tập validation (đường đứt đoạn).

```
plt.plot(rnn_history.history['loss'], label='RNN Training Loss')
```

```
plt.plot(rnn_history.history['val_loss'], label='RNN Validation Loss', linestyle='--')
```

- Vẽ tương tự cho mô hình RNN.

```
plt.plot(lstm_history.history['loss'], label='LSTM Training Loss')
```

```
plt.plot(lstm_history.history['val_loss'], label='LSTM Validation Loss', linestyle='--')
```

- Vẽ tương tự cho mô hình LSTM.

3. Thiết lập tiêu đề và các thông số cho biểu đồ:

```
plt.title('Training and Validation Loss Over Epochs')
```

```
plt.xlabel('Epochs')
```

```
plt.ylabel('Loss')
```

```
plt.legend()
```

```
plt.grid(True)
```

- **plt.title**: Thiết lập tiêu đề cho biểu đồ.
- **plt.xlabel** và **plt.ylabel**: Thiết lập nhãn cho trục x và trục y.
- **plt.legend()**: Hiển thị chú thích cho các đường trong biểu đồ.
- **plt.grid(True)**: Hiển thị lưới trên biểu đồ để dễ dàng so sánh các giá trị.

4. Hiển thị biểu đồ:

```
plt.show()
```

- Hiển thị biểu đồ đã vẽ với các đường mất mát huấn luyện và mất mát validation của ba mô hình CNN, RNN và LSTM.

