Nguyễn Quang Thắng – B21DCCN669 – intsyst02

CÂU 1:

A. CNN

1. Lịch sử phát triển của Convolutional Neural Network (CNN)

Bối cảnh ra đời: CNN là một loại mô hình mạng nơ-ron được thiết kế để xử lý các dữ liệu có dạng lưới, chẳng hạn như hình ảnh, và đã trở thành nền tảng trong lĩnh vực xử lý ảnh và nhận dạng ảnh. CNN phát triển từ ý tưởng về cách thức mà não người nhận diện hình ảnh và phản hồi với các đặc điểm khác nhau trong môi trường.

Những bước đầu tiên: Vào những năm 1980, **Kunihiko Fukushima** đã tạo ra một mạng nơ-ron dựa trên cấu trúc tự tổ chức gọi là "Neocognitron". Neocognitron được thiết kế với khả năng tự học để nhận dạng các mẫu hình ảnh bằng cách tạo ra các mức độ trừu tượng từ dữ liệu ảnh. Mặc dù không phải là CNN hiện đại như hiện nay, Neocognitron đã hình thành nền tảng cho CNN bằng cách sử dụng cơ chế kết nối cục bộ và phát triển ý tưởng về các lớp liên kết không đầy đủ.

CNN hiện đại: Vào cuối thập niên 1980 và đầu 1990, **Yann LeCun** và các cộng sự đã phát triển một trong những ứng dụng đầu tiên của CNN trong việc nhận dạng chữ số viết tay. Mô hình LeNet (phiên bản đầu tiên là LeNet-5) của LeCun sử dụng kiến trúc CNN cơ bản bao gồm các lớp tích chập và các lớp pooling, từ đó thành công trong việc phân loại chữ số viết tay. Đây là bước đột phá quan trọng giúp chứng minh hiệu quả của CNN.

Bùng nổ với sức mạnh tính toán và dữ liệu lớn: Đến năm 2012, với sự ra đời của mô hình **AlexNet** do Alex Krizhevsky và các cộng sự phát triển, CNN thực sự nổi bật trong lĩnh vực trí tuệ nhân tạo. Mô hình AlexNet đã đạt được kết quả vượt trội trong cuộc thi ImageNet Large Scale Visual Recognition Challenge (ILSVRC), với độ chính xác vượt trội. Điều này đã giúp chứng minh sức mạnh của CNN trong việc xử lý hình ảnh và mở ra kỷ nguyên của các ứng dụng CNN trên quy mô lớn.

2. Kiến trúc của Convolutional Neural Network (CNN)

CNN có kiến trúc đặc biệt để xử lý dữ liệu dạng lưới, chủ yếu gồm ba thành phần cơ bản: lớp tích chập (convolutional layer), lớp pooling (pooling layer), và lớp fully connected (FC layer).

Các lớp cơ bản trong CNN

1. **Lớp tích chập (Convolutional Layer):**
    - Lớp này áp dụng các bộ lọc (filter) lên ảnh đầu vào. Các bộ lọc di chuyển qua từng phần của ảnh, thực hiện phép nhân chập giữa giá trị ảnh với giá trị trong bộ lọc.
    - Kết quả là một ma trận đầu ra chứa các đặc điểm trích xuất từ ảnh.
    - Bộ lọc sẽ học các đặc trưng đơn giản như cạnh và góc ở các lớp đầu, sau đó sẽ học các đặc trưng phức tạp hơn ở các lớp sâu hơn.
2. **Lớp pooling (Pooling Layer):**

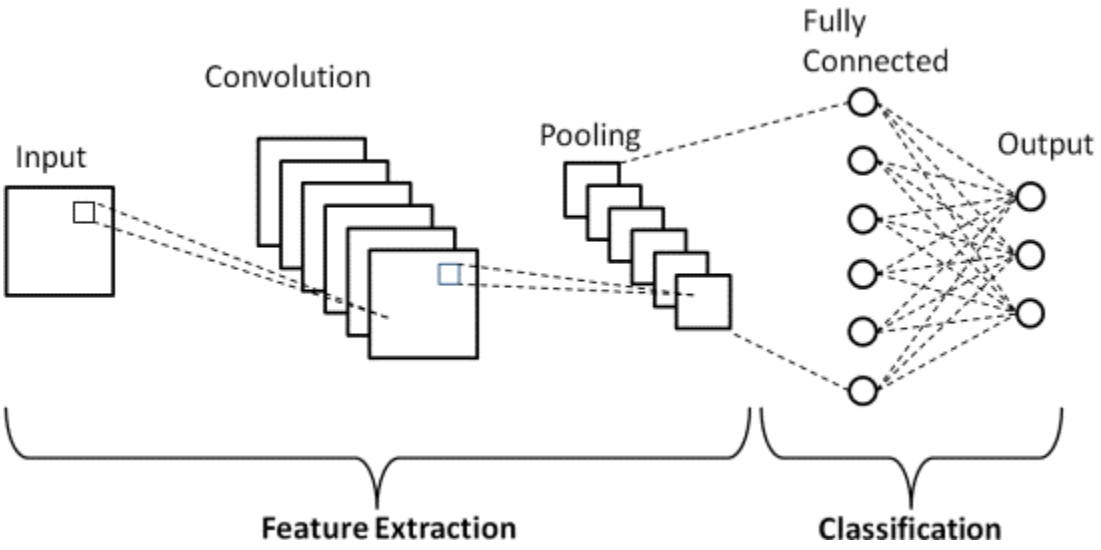
- Pooling giúp giảm kích thước đầu vào, giảm thiểu tính toán và giúp trích xuất các đặc trưng mạnh nhất.
- Các phương pháp pooling phổ biến bao gồm max pooling (chọn giá trị lớn nhất) và average pooling (chọn giá trị trung bình).

3. **Lớp fully connected (FC Layer):**
   - Lớp này nhận đầu vào là một vector và liên kết với tất cả các neuron trong lớp tiếp theo, từ đó thực hiện phân loại hoặc hồi quy.
   - Thông thường, lớp fully connected được đặt ở phần cuối của CNN để kết hợp các đặc trưng đã được trích xuất từ các lớp tích chập và pooling.

Các thành phần khác trong CNN

- **Activation Function (Hàm kích hoạt):** Thường sử dụng hàm ReLU (Rectified Linear Unit) để tạo tính phi tuyến cho mạng, giúp học được các đặc trưng phức tạp hơn.
- **Dropout Layer:** Giúp tránh hiện tượng overfitting bằng cách loại bỏ ngẫu nhiên một số neuron trong quá trình huấn luyện.
- **Batch Normalization Layer:** Giúp tăng tốc độ huấn luyện và cải thiện độ ổn định của mạng.

Kiến trúc tổng thể của CNN



Kiến trúc CNN thường bao gồm một chuỗi các lớp tích chập và pooling để trích xuất đặc trưng từ dữ liệu, sau đó là một hoặc nhiều lớp fully connected để thực hiện phân loại. Một kiến trúc CNN cơ bản có thể bao gồm các lớp theo thứ tự sau:

1. Convolutional Layer + Activation Layer (ReLU)
2. Pooling Layer
3. Convolutional Layer + Activation Layer (ReLU)
4. Pooling Layer
5. Fully Connected Layer + Activation Layer (ReLU)

6. Output Layer

3. Code minh họa về CNN

Dưới đây là một ví dụ đơn giản sử dụng thư viện Keras (trong TensorFlow) để xây dựng một mô hình CNN cho bài toán phân loại ảnh trên tập dữ liệu MNIST (nhận diện chữ số viết tay).

```
import tensorflow as tf
from tensorflow.keras import layers, models

# 1. CNN Model
def create_cnn_model(input_shape):
   model = models.Sequential([
      layers.Conv2D(32, (3, 3), activation='relu', input_shape=input_shape),
      layers.MaxPooling2D((2, 2)),
      layers.Conv2D(64, (3, 3), activation='relu'),
      layers.MaxPooling2D((2, 2)),
      layers.Conv2D(64, (3, 3), activation='relu'),
      layers.Flatten(),
      layers.Dense(64, activation='relu'),
      layers.Dense(10, activation='softmax')
   ])
   model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',
metrics=['accuracy'])
   return model

input_shape = (28, 28, 1)  # Nếu là hình ảnh 28x28 đen trắng (như trong MNIST)
cnn_model = create_cnn_model(input_shape)
cnn_model.summary()
```

Giải thích các bước trong code:

1. Hàm create_cnn_model
Hàm này định nghĩa một kiến trúc CNN với các lớp tích chập (convolutional layers), lớp pooling (max-pooling layers), và các lớp dày đặc (dense layers). Hàm nhận vào:

- input_shape: Kích thước đầu vào của hình ảnh, ví dụ: (28, 28, 1) cho ảnh 28x28 pixels với một kênh màu (ảnh đen trắng).

2. Các lớp trong mô hình
Các lớp trong mô hình được khai báo lần lượt như sau:

- **Conv2D(32, (3, 3), activation='relu', input_shape=input_shape)**:
  - Lớp tích chập đầu tiên với 32 bộ lọc (filters) kích thước 3x3 và hàm kích hoạt ReLU (Rectified Linear Unit).
  - Lớp này phát hiện các đặc trưng đơn giản trong ảnh đầu vào như đường viền và góc cạnh.
  - input_shape chỉ định kích thước đầu vào cho lớp đầu tiên (28x28x1).
- **MaxPooling2D((2, 2))**:
  - Lớp pooling giúp giảm kích thước ảnh, chọn giá trị lớn nhất trong mỗi vùng 2x2.
  - Giảm số lượng tham số, ngăn chặn quá khớp (overfitting) và tăng tốc độ tính toán.
- **Conv2D(64, (3, 3), activation='relu') và MaxPooling2D((2, 2))**:
  - Lặp lại với 64 bộ lọc 3x3, giúp mạng học các đặc trưng phức tạp hơn từ ảnh, tiếp

tục giảm kích thước ảnh đầu vào.

- **Conv2D(64, (3, 3), activation='relu')**:
  - Lớp tích chập thứ ba với 64 bộ lọc, nhưng không có pooling sau đó.
  - Mạng tiếp tục học các đặc trưng sâu hơn từ đầu vào đã qua các lớp pooling trước đó.
- **Flatten()**:
  - Chuyển đầu ra 2D thành dạng 1D để có thể đưa vào các lớp dense sau này.
- **Dense(64, activation='relu')**:
  - Lớp dense với 64 nút (neurons) và hàm kích hoạt ReLU, giúp mạng học các đặc trưng tổng quát.
- **Dense(10, activation='softmax')**:
  - Lớp đầu ra với 10 nút và hàm kích hoạt softmax, tạo ra xác suất dự đoán cho mỗi trong 10 lớp (ví dụ: 10 chữ số từ 0 đến 9).

3. model.compile

Phương thức này thiết lập các tham số học:

- **optimizer='adam'**: Sử dụng trình tối ưu Adam, giúp mạng học và điều chỉnh trọng số một cách hiệu quả.
- **loss='sparse_categorical_crossentropy'**: Hàm mất mát cho bài toán phân loại nhiều lớp, sử dụng khi nhãn là số nguyên.
- **metrics=['accuracy']**: Đo lường độ chính xác (accuracy) trong quá trình huấn luyện.

4. Kết quả

Cuối cùng, cnn_model.summary() sẽ hiển thị cấu trúc và số lượng tham số của mô hình để bạn có thể kiểm tra tổng quan về kiến trúc của mạng.

Layer (type)	Output Shape	Param #
conv2d_2 (Conv2D)	(None, 26, 26, 32)	320
max_pooling2d_2 (MaxPooling2D)	(None, 13, 13, 32)	0
conv2d_3 (Conv2D)	(None, 11, 11, 64)	18,496
max_pooling2d_3 (MaxPooling2D)	(None, 5, 5, 64)	0
conv2d_4 (Conv2D)	(None, 3, 3, 64)	36,928
flatten_1 (Flatten)	(None, 576)	0
dense_2 (Dense)	(None, 64)	36,928
dense_3 (Dense)	(None, 10)	650

Total params: 93,322 (364.54 KB)

Trainable params: 93,322 (364.54 KB)

Non-trainable params: 0 (0.00 B)

CNN là một công cụ mạnh mẽ và hiệu quả trong xử lý dữ liệu hình ảnh, với khả năng học các đặc trưng từ ảnh thông qua kiến trúc nhiều lớp tích chập, giúp mô hình tự động trích xuất các đặc trưng quan trọng nhất từ dữ liệu đầu vào.

B. RNN

1. Lịch sử phát triển của Recurrent Neural Network (RNN)

Bối cảnh ra đời: RNN ra đời nhằm xử lý các dữ liệu dạng chuỗi (sequence) như âm thanh, văn bản, và dữ liệu thời gian, nơi mà thứ tự và mối liên kết giữa các phần tử là rất quan trọng. Trước khi có RNN, các mô hình mạng nơ-ron truyền thẳng (feedforward neural networks) gặp nhiều hạn chế trong việc xử lý các chuỗi dài, do chúng thiếu khả năng "ghi nhớ" các thông tin trong quá khứ.

Các bước phát triển:

- **Những năm 1980:** Các nhà khoa học như **David Rumelhart** và **Geoffrey Hinton** đặt nền móng cho mạng RNN với ý tưởng rằng mạng nơ-ron có thể sử dụng thông tin từ quá khứ để đưa ra dự đoán. Tuy nhiên, vấn đề của RNN ban đầu là gặp phải hiện tượng **biến mất hoặc bùng nổ gradient** (vanishing/exploding gradient) khi độ dài chuỗi tăng.
- **LSTM (Long Short-Term Memory):** Để khắc phục vấn đề biến mất gradient, vào năm 1997, **Sepp Hochreiter** và **Jürgen Schmidhuber** đã giới thiệu mô hình LSTM. LSTM có cơ chế "bộ nhớ" đặc biệt với các cổng vào (input gate), cổng quên (forget gate), và cổng ra (output gate), giúp ghi nhớ các thông tin trong chuỗi dài hơn mà không gặp vấn đề về gradient.
- **GRU (Gated Recurrent Unit):** Một biến thể khác là GRU, được giới thiệu bởi **Kyunghyun Cho** và các đồng nghiệp vào năm 2014. GRU đơn giản hóa các cơ chế cổng của LSTM, làm giảm độ phức tạp của mô hình nhưng vẫn duy trì hiệu quả trong các bài toán xử lý chuỗi.

Ứng dụng phổ biến: RNN và các biến thể như LSTM và GRU được ứng dụng rộng rãi trong dịch máy, nhận diện giọng nói, dự đoán chuỗi thời gian và nhiều lĩnh vực khác. Với sự phát triển của RNN, việc xử lý các chuỗi dữ liệu phức tạp đã trở nên khả thi và hiệu quả hơn nhiều.

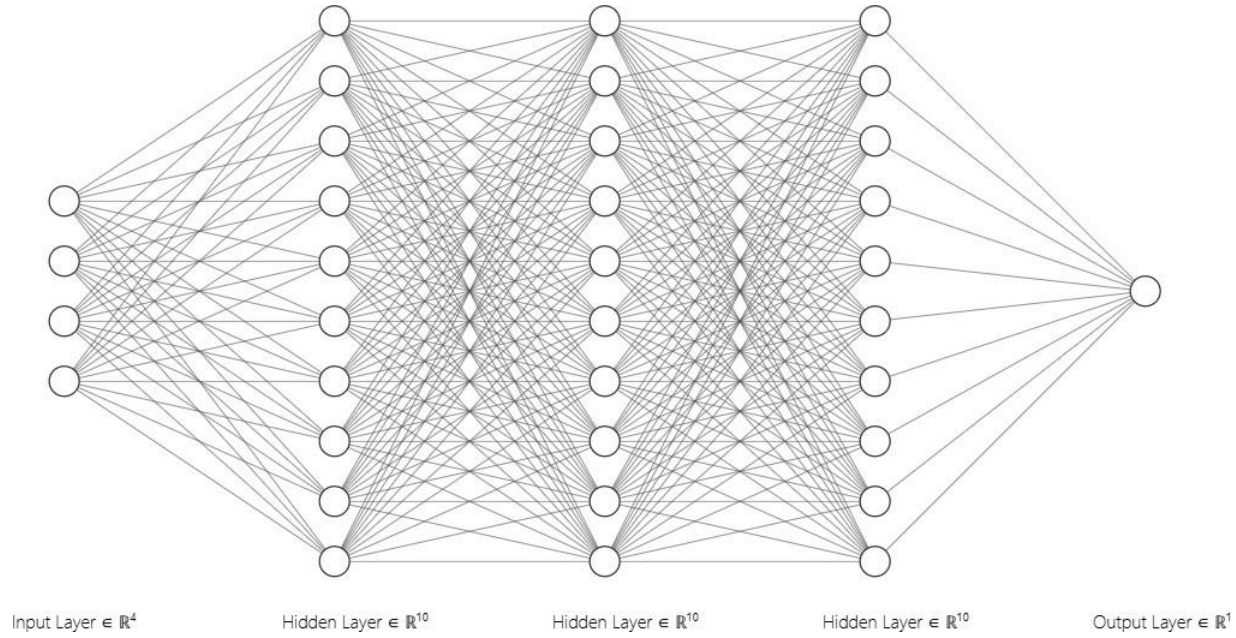
2. Kiến trúc của Recurrent Neural Network (RNN)

Thành phần chính trong RNN

1. **Lớp tái hồi (Recurrent Layer):**
    - RNN duy trì một **trạng thái ẩn (hidden state)** để lưu trữ thông tin từ các bước thời gian trước đó. Trạng thái ẩn này được cập nhật tại mỗi bước thời gian và giúp ghi nhớ ngữ cảnh của chuỗi.
    - Tại mỗi bước thời gian, đầu vào hiện tại và trạng thái ẩn của bước trước được kết hợp để tạo ra đầu ra và cập nhật trạng thái ẩn mới.
2. **Các biến thể của RNN (LSTM, GRU):**
    - **LSTM:** Mô hình LSTM sử dụng các cổng để quyết định khi nào ghi nhớ hoặc quên thông tin. Cấu trúc của LSTM giúp kiểm soát cách thông tin được duy trì và loại bỏ theo chuỗi, rất phù hợp với các chuỗi dài.

- **GRU:** GRU là một biến thể đơn giản hơn của LSTM, chỉ có hai cổng chính là cổng cập nhật (update gate) và cổng xoá (reset gate), giúp xử lý chuỗi với độ dài vừa phải và ít tiêu tốn tài nguyên hơn.

Kiến trúc tổng thể của RNN



Kiến trúc của RNN bao gồm một chuỗi các lớp tái hồi (recurrent layer) được kết nối tuần tự để truyền tải trạng thái giữa các bước thời gian. Sau lớp tái hồi, ta thường thêm một lớp fully connected để tạo ra đầu ra cuối cùng.

1. Input Layer (đầu vào chuỗi dữ liệu)
2. Recurrent Layer (lớp tái hồi) như LSTM hoặc GRU
3. Fully Connected Layer (lớp fully connected) cho mục tiêu đầu ra

```
# 2. RNN Model
def create_rnn_model(input_shape):
    model = models.Sequential([
        layers.SimpleRNN(50, activation='relu', input_shape=input_shape),
        layers.Dense(1)
    ])
    model.compile(optimizer='adam', loss='mean_squared_error')
    return model

# Sử dụng ví dụ:
input_shape = (10, 1)  # Như (10, 1) nếu có 10 bước thời gian và 1 đặc trưng mỗi bước
rnn_model = create_rnn_model(input_shape)
rnn_model.summary()
```

Layer (type)	Output Shape	Param #
simple_rnn (SimpleRNN)	(None, 50)	2,600
dense_4 (Dense)	(None, 1)	51

Total params: 2,651 (10.36 KB)

Trainable params: 2,651 (10.36 KB)

Non-trainable params: 0 (0.00 B)

3. Code minh họa về RNN

Dưới đây là ví dụ về RNN với LSTM để dự đoán chuỗi số liệu trong bài toán dự đoán giá trị tiếp theo (time series prediction).

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense

# Tạo dữ liệu giả lập cho bài toán chuỗi thời gian
def create_dataset(sequence, n_steps):
```

```
    X, y = [], []
    for i in range(len(sequence)):
        end_ix = i + n_steps
        if end_ix > len(sequence) - 1:
            break
        seq_x, seq_y = sequence[i:end_ix], sequence[end_ix]
        X.append(seq_x)
        y.append(seq_y)
    return np.array(X), np.array(y)

# Giả lập một chuỗi thời gian
sequence = np.array([i for i in range(100)])
n_steps = 3  # Độ dài chuỗi đầu vào
X, y = create_dataset(sequence, n_steps)

# Thay đổi kích thước đầu vào cho phù hợp với LSTM (samples, timesteps,
features)
X = X.reshape((X.shape[0], X.shape[1], 1))

# 1. Xây dựng mô hình RNN với LSTM
model = Sequential([
    LSTM(50, activation="relu", input_shape=(n_steps, 1)),
    Dense(1)
])

# 2. Biên dịch mô hình
model.compile(optimizer="adam", loss="mse")

# 3. Huấn luyện mô hình
model.fit(X, y, epochs=200, verbose=0)

# 4. Dự đoán giá trị tiếp theo
x_input = np.array([97, 98, 99]).reshape((1, n_steps, 1))
yhat = model.predict(x_input, verbose=0)
print("Dự đoán giá trị tiếp theo:", yhat[0][0])
```

Giải thích từng bước trong code:

- **Bước 1:** Tạo dữ liệu giả lập cho chuỗi thời gian bằng cách lấy các chuỗi con có độ dài `n_steps` từ chuỗi chính.
- **Bước 2:** Xây dựng mô hình RNN với LSTM với một lớp LSTM có 50 đơn vị ẩn, theo sau là một lớp fully connected để dự đoán giá trị tiếp theo.
- **Bước 3:** Biên dịch và huấn luyện mô hình với hàm mất mát `mse` (Mean Squared Error) cho bài toán hồi quy.
- **Bước 4:** Dự đoán giá trị tiếp theo của chuỗi dựa trên dữ liệu đầu vào `[97, 98, 99]`.

Tổng kết

RNN là một kiến trúc mạnh mẽ cho các bài toán xử lý chuỗi, với các biến thể như LSTM và GRU giúp giải quyết vấn đề về biến mất gradient và ghi nhớ được các mối liên kết dài hạn trong chuỗi. Những mô hình này được ứng dụng rộng rãi trong các lĩnh vực như dịch máy, nhận dạng giọng nói, và dự đoán chuỗi thời gian, nhờ khả năng ghi nhớ và xử lý dữ liệu chuỗi hiệu quả.

C. LSTM

1. Lịch sử phát triển của Long Short-Term Memory (LSTM)

Bối cảnh ra đời: Mặc dù Recurrent Neural Network (RNN) đã được phát triển để xử lý các dữ liệu chuỗi, nhưng khi làm việc với chuỗi dài, RNN gặp phải vấn đề **biến mất gradient** (vanishing gradient), khiến nó khó học các mối quan hệ xa trong chuỗi. Điều này làm RNN gặp khó khăn trong việc duy trì và xử lý thông tin ở các bước thời gian dài.

Phát minh LSTM: Để khắc phục vấn đề này, vào năm 1997, **Sepp Hochreiter** và **Jürgen Schmidhuber** đã giới thiệu **Long Short-Term Memory (LSTM)**. LSTM là một loại RNN đặc biệt với khả năng kiểm soát cách thông tin được lưu trữ và quên trong quá trình huấn luyện, nhờ vào một hệ thống các **cổng (gates)**. Nhờ vậy, LSTM có thể học và ghi nhớ thông tin trong các chuỗi dài hơn so với RNN thông thường.

Các cải tiến sau này: Nhiều biến thể và cải tiến của LSTM đã ra đời để tăng hiệu quả và độ chính xác trong quá trình xử lý chuỗi, chẳng hạn như **Peephole LSTM**, cho phép các cổng tiếp cận trực tiếp trạng thái của ô nhớ để ra quyết định chính xác hơn.

Ứng dụng thực tiễn: LSTM đã được ứng dụng rộng rãi trong các lĩnh vực như dịch máy, nhận dạng giọng nói, và dự đoán chuỗi thời gian, mang lại kết quả đáng kể trong các bài toán liên quan đến dữ liệu chuỗi dài và có mối quan hệ phức tạp.

2. Kiến trúc của Long Short-Term Memory (LSTM)

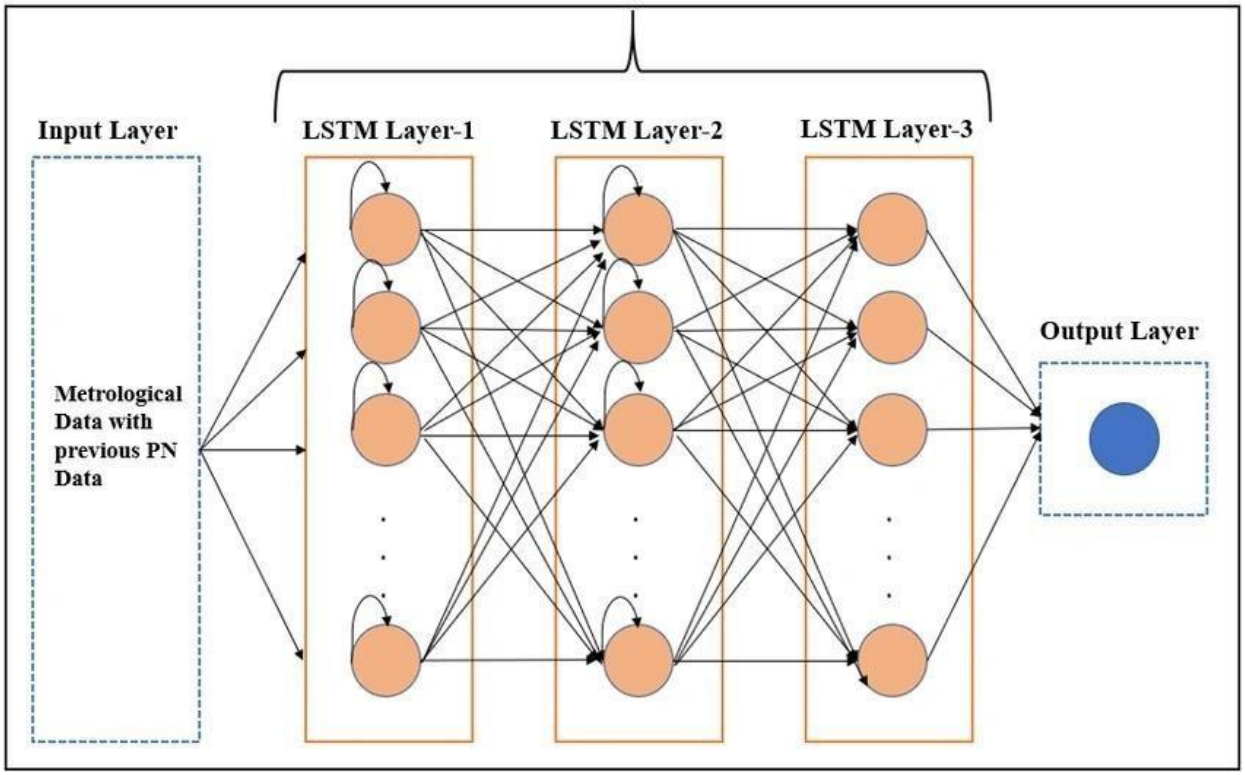
LSTM là một loại mạng RNN có cấu trúc đặc biệt gồm các **cổng** để kiểm soát thông tin nào được duy trì và thông tin nào sẽ bị loại bỏ. Các cổng này bao gồm:

1. **Forget Gate (Cổng quên):** Quyết định phần nào của trạng thái ô nhớ cần được quên đi.
2. **Input Gate (Cổng vào):** Quyết định thông tin nào từ đầu vào mới sẽ được lưu vào trạng thái ô nhớ.
3. **Cell State (Trạng thái ô nhớ):** Là nơi lưu trữ thông tin chính, có khả năng lưu giữ thông tin trong khoảng thời gian dài.
4. **Output Gate (Cổng ra):** Quyết định phần nào của trạng thái ô nhớ sẽ được sử dụng làm đầu ra của LSTM ở bước hiện tại.

Quá trình hoạt động của LSTM qua các cổng:

- **Cổng quên** quyết định loại bỏ những thông tin nào không còn cần thiết từ trạng thái ô nhớ.
- **Cổng vào** nhận thông tin từ đầu vào hiện tại và kết hợp với trạng thái ẩn trước đó để thêm thông tin vào trạng thái ô nhớ.
- **Cổng ra** quyết định đầu ra nào sẽ được sử dụng tại bước hiện tại dựa trên trạng thái ô nhớ.

Kiến trúc tổng thể của LSTM



Kiến trúc LSTM thường bao gồm một chuỗi các lớp LSTM để duy trì mối liên kết giữa các bước thời gian trong chuỗi, theo sau là một hoặc nhiều lớp fully connected để thực hiện nhiệm vụ phân loại hoặc hồi quy.

1. Input Layer (đầu vào chuỗi dữ liệu)
2. LSTM Layer (lớp LSTM) với các cổng quản lý trạng thái ô nhớ
3. Fully Connected Layer (lớp fully connected) cho mục tiêu đầu ra

```
# 3. LSTM Model
def create_lstm_model(input_shape):
    model = models.Sequential([
        layers.LSTM(50, activation='relu', input_shape=input_shape),
        layers.Dense(1)
    ])
    model.compile(optimizer='adam', loss='mean_squared_error')
    return model

# Sử dụng ví dụ:
input_shape = (10, 2)  # Như (10, 1) nếu có 10 bước thời gian và 1 đặc trưng mỗi bước
lstm_model = create_lstm_model(input_shape)
lstm_model.summary()
```

Layer (type)	Output Shape	Param #
lstm (LSTM)	(None, 50)	10,600
dense_5 (Dense)	(None, 1)	51

Total params: 10,651 (41.61 KB)

Trainable params: 10,651 (41.61 KB)

Non-trainable params: 0 (0.00 B)

3. Code minh họa về LSTM

Dưới đây là một ví dụ đơn giản sử dụng LSTM cho bài toán dự đoán chuỗi thời gian với thư viện Keras (trong TensorFlow).

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense

# Tạo dữ liệu giả lập cho bài toán chuỗi thời gian
def create_dataset(sequence, n_steps):
    X, y = [], []
    for i in range(len(sequence)):
        end_ix = i + n_steps
        if end_ix > len(sequence) - 1:
            break
        seq_x, seq_y = sequence[i:end_ix], sequence[end_ix]
        X.append(seq_x)
```

```
        y.append(seq_y)
    return np.array(X), np.array(y)

# Giả lập một chuỗi thời gian
sequence = np.array([i for i in range(100)])
n_steps = 3  # Độ dài chuỗi đầu vào
X, y = create_dataset(sequence, n_steps)

# Thay đổi kích thước đầu vào cho phù hợp với LSTM (samples, timesteps,
features)
X = X.reshape((X.shape[0], X.shape[1], 1))

# 1. Xây dựng mô hình LSTM
model = Sequential([
    LSTM(50, activation="relu", input_shape=(n_steps, 1)),
    Dense(1)
])

# 2. Biên dịch mô hình
model.compile(optimizer="adam", loss="mse")

# 3. Huấn luyện mô hình
model.fit(X, y, epochs=200, verbose=0)

# 4. Dự đoán giá trị tiếp theo
x_input = np.array([97, 98, 99]).reshape((1, n_steps, 1))
yhat = model.predict(x_input, verbose=0)
print("Dự đoán giá trị tiếp theo:", yhat[0][0])
```

Giải thích các bước trong code:

- **Bước 1:** Tạo dữ liệu giả lập cho chuỗi thời gian bằng cách lấy các chuỗi con có độ dài `n_steps` từ chuỗi chính.
- **Bước 2:** Xây dựng mô hình LSTM với một lớp LSTM có 50 đơn vị ẩn, theo sau là một lớp fully connected để dự đoán giá trị tiếp theo.
- **Bước 3:** Biên dịch và huấn luyện mô hình với hàm mất mát `mse` (Mean Squared Error) cho bài toán hồi quy.
- **Bước 4:** Dự đoán giá trị tiếp theo của chuỗi dựa trên dữ liệu đầu vào `[97, 98, 99]`.

Tổng kết

LSTM là một biến thể của RNN được thiết kế để khắc phục vấn đề về biến mất gradient, cho phép ghi nhớ thông tin từ các chuỗi dài. Nhờ vào hệ thống các cổng thông minh, LSTM giúp các mô hình mạng nơ-ron xử lý dữ liệu chuỗi hiệu quả hơn, đem lại kết quả xuất sắc trong các lĩnh vực như nhận dạng giọng nói, dịch máy, và phân tích chuỗi thời gian.

CÂU 2:
Bước 1: Tiền xử lý dữ liệu

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.metrics import mean_absolute_error, mean_squared_error
import tensorflow as tf
```

- **`pandas`**: Thư viện để thao tác và xử lý dữ liệu trong dạng bảng (DataFrame).
- **`numpy`**: Thư viện hỗ trợ các phép toán với mảng và ma trận.
- **`sklearn`**: Thư viện cho các công cụ học máy, bao gồm chia tách dữ liệu, chuẩn hóa và đánh giá mô hình.
- **`tensorflow`**: Thư viện phổ biến để xây dựng và huấn luyện các mô hình học sâu.

Tải dữ liệu và xử lý

```
# Load data
data = pd.read_csv("bds_hadong_refined.csv")

# Encode 'Property Type' using One-Hot Encoding
data = pd.get_dummies(data, columns=['Property Type'], drop_first=True)

# Separate features and target variable
X = data.drop("Price (VND)", axis=1)
y = data["Price (VND)"]

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

# Scale features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

- **Tải dữ liệu**: Dữ liệu từ file CSV được tải vào DataFrame.
- **One-Hot Encoding**: Chuyển đổi thuộc tính phân loại `Property Type` thành các biến nhị phân để mô hình có thể hiểu được.
- **Tách dữ liệu**: Tách dữ liệu thành hai phần: `X` (các đặc trưng) và `y` (giá mục tiêu).
- **Chia tách dữ liệu**: Sử dụng `train_test_split` để chia dữ liệu thành tập huấn luyện và tập kiểm tra, với 80% cho huấn luyện và 20% cho kiểm tra.
- **Chuẩn hóa dữ liệu**: Sử dụng `StandardScaler` để chuẩn hóa dữ liệu, giúp cải thiện khả năng hội tụ của các mô hình.

Bước 2: Định nghĩa các mô hình

1. Mô hình CNN

```
def create_cnn_model(input_shape):
    model = tf.keras.models.Sequential([
        tf.keras.layers.Reshape((input_shape[0], 1),
input_shape=input_shape),
        tf.keras.layers.Conv1D(64, kernel_size=2, activation="relu"),
        tf.keras.layers.Conv1D(32, kernel_size=2, activation="relu"),
        tf.keras.layers.Flatten(),
        tf.keras.layers.Dense(64, activation="relu"),
        tf.keras.layers.Dense(1)
    ])
    model.compile(optimizer="adam", loss="mse", metrics=["mae", "mse",
tf.keras.metrics.RootMeanSquaredError()])
    return model
```

- **`Sequential`**: Kiến trúc mô hình tuần tự.
- **`Reshape`**: Chuyển đổi dữ liệu đầu vào thành dạng mà CNN có thể xử lý.
- **`Conv1D`**: Tầng tích chập 1 chiều, thường dùng trong xử lý chuỗi (như dữ liệu thời gian).
- **`Flatten`**: Làm phẳng dữ liệu từ dạng đa chiều sang một chiều trước khi đưa vào các tầng Dense.
- **`Dense`**: Tầng hoàn toàn kết nối, với tầng đầu ra có một nút (để dự đoán giá).

2. Mô hình RNN

```
def create_rnn_model(input_shape):
    model = tf.keras.models.Sequential([
        tf.keras.layers.Reshape((input_shape[0], 1),
input_shape=input_shape),
        tf.keras.layers.SimpleRNN(64, activation="relu"),
        tf.keras.layers.Dense(64, activation="relu"),
        tf.keras.layers.Dense(1)
    ])
    model.compile(optimizer="adam", loss="mse", metrics=["mae", "mse",
tf.keras.metrics.RootMeanSquaredError()])
    return model
```

- **`SimpleRNN`**: Tầng RNN đơn giản, được sử dụng để xử lý dữ liệu theo chuỗi (thời gian).

3. Mô hình LSTM

```
def create_lstm_model(input_shape):
    model = tf.keras.models.Sequential([
        tf.keras.layers.Reshape((input_shape[0], 1),
input_shape=input_shape),
        tf.keras.layers.LSTM(64, activation="relu"),
        tf.keras.layers.Dense(64, activation="relu"),
```

```
        tf.keras.layers.Dense(1)
    ])
    model.compile(optimizer="adam", loss="mse", metrics=["mae", "mse",
tf.keras.metrics.RootMeanSquaredError()])
    return model
```

- **`LSTM`**: Tầng LSTM (Long Short-Term Memory), một loại mạng RNN, có khả năng ghi nhớ thông tin lâu hơn và rất hiệu quả trong các bài toán dự đoán chuỗi thời gian.

Bước 3: Huấn luyện các mô hình

```
# Define input shape
input_shape = X_train.shape[1:]

# Initialize models
cnn_model = create_cnn_model(input_shape)
rnn_model = create_rnn_model(input_shape)
lstm_model = create_lstm_model(input_shape)

# Train models
epochs = 50
cnn_history = cnn_model.fit(X_train, y_train, epochs=epochs,
validation_data=(X_test, y_test), verbose=1)
rnn_history = rnn_model.fit(X_train, y_train, epochs=epochs,
validation_data=(X_test, y_test), verbose=1)
lstm_history = lstm_model.fit(X_train, y_train, epochs=epochs,
validation_data=(X_test, y_test), verbose=1)
```

- **Khởi tạo các mô hình**: Tạo các mô hình CNN, RNN và LSTM với các hình dạng đầu vào.
- **Huấn luyện mô hình**: Sử dụng phương thức `fit` để huấn luyện các mô hình trong 50 epochs, với dữ liệu kiểm tra được cung cấp để theo dõi hiệu suất.

Bước 4: Tính toán và đánh giá các chỉ số

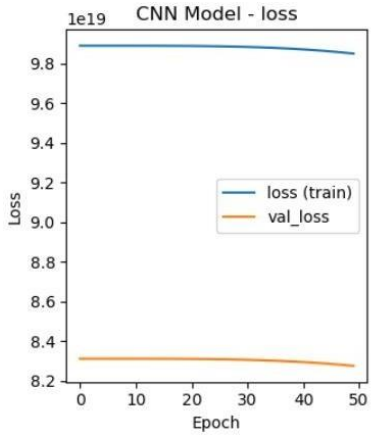
```
# Define function to evaluate model
def evaluate_model(model, X_test, y_test):
    y_pred = model.predict(X_test)
    mae = mean_absolute_error(y_test, y_pred)
    mse = mean_squared_error(y_test, y_pred)
    rmse = np.sqrt(mse)
    return mae, mse, rmse

# Evaluate models
cnn_mae, cnn_mse, cnn_rmse = evaluate_model(cnn_model, X_test, y_test)
rnn_mae, rnn_mse, rnn_rmse = evaluate_model(rnn_model, X_test, y_test)
lstm_mae, lstm_mse, lstm_rmse = evaluate_model(lstm_model, X_test, y_test)
```

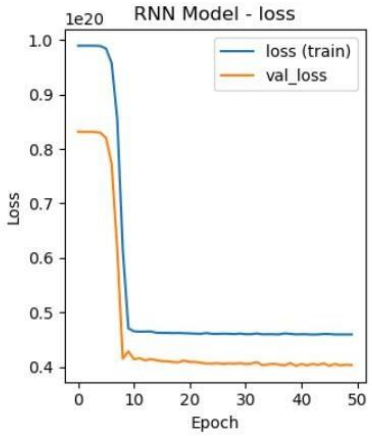
- **`evaluate_model`**: Hàm này tính toán các chỉ số đánh giá MAE (Mean Absolute Error), MSE (Mean Squared Error) và RMSE (Root Mean Squared Error) cho từng mô hình dựa trên dữ liệu kiểm tra.
- Các chỉ số này giúp đánh giá độ chính xác của mô hình trong việc dự đoán giá bất động sản.

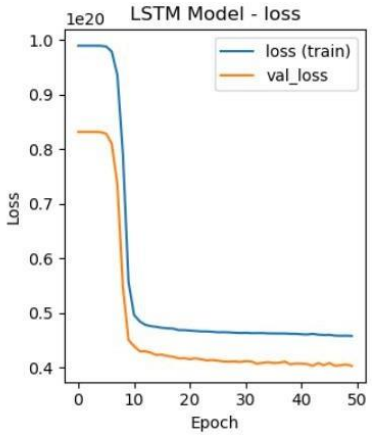
Bước 5: Vẽ biểu đồ chỉ số trong các subplot

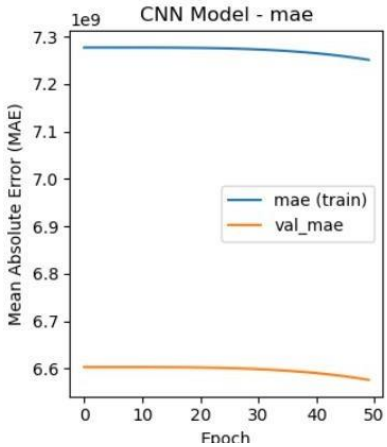
```
import matplotlib.pyplot as plt

# Create a function to plot metrics for all models in subplots
def plot_metrics_subplots(histories, model_names, metric, ylabel):
    plt.figure(figsize=(14, 8))
    for i, (history, model_name) in enumerate(zip(histories, model_names)):
        plt.subplot(1, 3, i+1)
        plt.plot(history.history[metric], label=f'{metric} (train)')
        plt.plot(history.history[f'val_{metric}'], label=f'val_{metric}')
        plt.title(f'{model_name} - {metric}')
        plt.xlabel('Epoch')
        plt.ylabel(ylabel)
        plt.legend()
    plt.tight_layout()
    plt.show()

# Collect all histories and model names
histories = [cnn_history, rnn_history, lstm_history]
model_names = ["CNN Model", "RNN Model", "LSTM Model"]

# Plot loss for all models
plot_metrics_subplots(histories, model_names, "loss", "Loss")

# Plot MAE for all models
plot_metrics_subplots(histories, model_names, "mae", "Mean Absolute Error
(MAE)")

# Plot MSE for all models
plot_metrics_subplots(histories, model_names, "mse", "Mean Squared Error
(MSE)")

# Plot RMSE for all models
plot_metrics_subplots(histories, model_names, "root_mean_squared_error",
"Root Mean Squared Error (RMSE)")
```

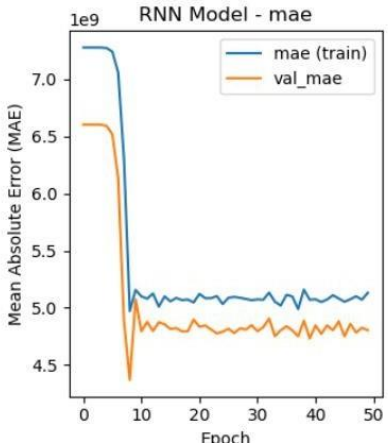
- **Vẽ biểu đồ**: Hàm `plot_metrics_subplots` được sử dụng để vẽ các biểu đồ cho từng chỉ số của các mô hình trong các subplot.
- Mỗi subplot hiển thị độ giảm của các chỉ số qua từng epoch, giúp so sánh trực quan hiệu suất của các mô hình.

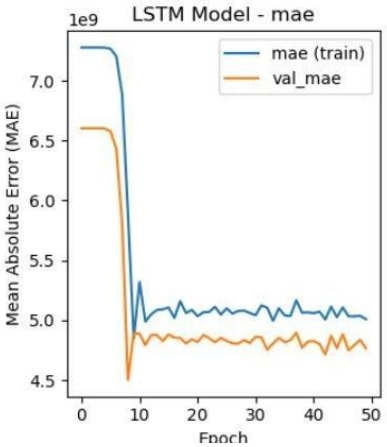

CNN Model - loss
1e19
Loss
Epoch
loss (train)
val_loss

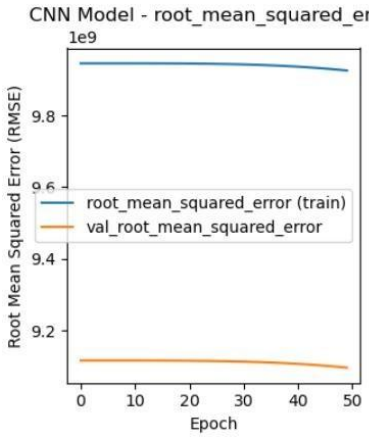

RNN Model - loss
1e20
Loss
Epoch
loss (train)
val_loss

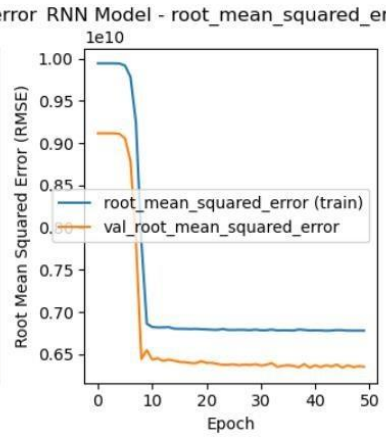

LSTM Model - loss
1e20
Loss
Epoch
loss (train)
val_loss

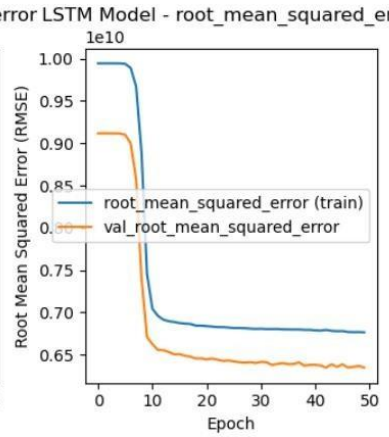

CNN Model - mae
1e9
Mean Absolute Error (MAE)
Epoch
mae (train)
val_mae


RNN Model - mae
1e9
Mean Absolute Error (MAE)
Epoch
mae (train)
val_mae


LSTM Model - mae
1e9
Mean Absolute Error (MAE)
Epoch
mae (train)
val_mae


CNN Model - root_mean_squared_error
1e9
Root Mean Squared Error (RMSE)
Epoch
root_mean_squared_error (train)
val_root_mean_squared_error


RNN Model - root_mean_squared_error
1e10
Root Mean Squared Error (RMSE)
Epoch
root_mean_squared_error (train)
val_root_mean_squared_error


LSTM Model - root_mean_squared_error
1e10
Root Mean Squared Error (RMSE)
Epoch
root_mean_squared_error (train)
val_root_mean_squared_error

CÂU 3:

Bước 1: Nhập Thư Viện Cần Thiết

```
import pandas as pd
import tensorflow as tf
from sklearn.model_selection import train_test_split
from transformers import BertTokenizer, TFBertForSequenceClassification
import matplotlib.pyplot as plt
```

- **pandas**: Thư viện này được sử dụng để xử lý dữ liệu, giúp đọc, viết và thao tác với dữ liệu dạng bảng (DataFrame).
- **tensorflow**: Thư viện mã nguồn mở cho học sâu. Nó cho phép xây dựng và huấn luyện các mô hình học máy.
- **sklearn.model_selection**: Cung cấp công cụ cho việc chia tách dữ liệu thành các tập huấn luyện và kiểm thử.
- **transformers**: Thư viện từ Hugging Face, chứa nhiều mô hình học sâu bao gồm BERT và các công cụ liên quan đến xử lý ngôn ngữ tự nhiên (NLP).
- **matplotlib.pyplot**: Thư viện để vẽ biểu đồ, sử dụng để hiển thị kết quả của quá trình huấn luyện mô hình.

Bước 2: Đọc Dữ Liệu từ CSV

```
data = pd.read_csv("comments_data.csv
```

- Sử dụng hàm `read_csv` của pandas để đọc dữ liệu từ file CSV chứa bình luận sản phẩm và các nhãn tương ứng (đánh giá).
- `data` sẽ là một DataFrame chứa tất cả các thông tin trong file CSV.

Bước 3: Chuẩn Bị Dữ Liệu

```
X = data['comment'].values
y = data['rating'].values - 1  # Chuyển đổi nhãn từ 1-5 thành 0-4

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

- **X**: Lưu trữ các bình luận (text data) từ cột `'comment'`.
- **y**: Lưu trữ nhãn đánh giá, được trừ 1 để biến đổi từ khoảng [1-5] thành [0-4]. Việc này cần thiết vì trong mô hình phân loại, nhãn bắt đầu từ 0.
- **train_test_split**: Chia dữ liệu thành hai phần: 80% cho huấn luyện và 20% cho kiểm thử. `random_state=42` đảm bảo rằng việc chia dữ liệu là có thể lặp lại.

Bước 4: Tokenization và Chuyển Đổi Dữ Liệu

```
tokenizer = BertTokenizer.from_pretrained("bert-base-uncased")
max_len = 128  # Giới hạn độ dài câu để tránh quá tải mô hình

# Tokenize và encode
train_encodings = tokenizer(X_train.tolist(), truncation=True, padding=True,
max_length=max_len)
test_encodings = tokenizer(X_test.tolist(), truncation=True, padding=True,
max_length=max_len)

# Chuyển đổi dữ liệu vào TensorFlow dataset
train_dataset = tf.data.Dataset.from_tensor_slices((
    dict(train_encodings),
    y_train
)).shuffle(len(X_train)).batch(16)

test_dataset = tf.data.Dataset.from_tensor_slices((
    dict(test_encodings),
    y_test
)).batch(16)
```

- **BertTokenizer**: Sử dụng tokenizer từ mô hình BERT để mã hóa các bình luận thành định dạng mà mô hình có thể hiểu.
    - `truncation=True`: Cắt ngắn câu nếu độ dài vượt quá giới hạn.
    - `padding=True`: Thêm padding vào các câu ngắn hơn để chúng có cùng độ dài.
    - `max_length`: Giới hạn độ dài tối đa cho mỗi bình luận (128 từ).
- **tf.data.Dataset**: Chuyển đổi dữ liệu đã mã hóa thành tập dữ liệu TensorFlow.
    - `shuffle(len(X_train))`: Xáo trộn dữ liệu để cải thiện độ chính xác của mô hình.
    - `batch(16)`: Chia dữ liệu thành các nhóm (batch) gồm 16 bình luận cho mỗi lần huấn luyện.

Bước 5: Tạo Mô Hình BERT

```
model = TFBertForSequenceClassification.from_pretrained("bert-base-uncased",
num_labels=5)  # 5 cho đánh giá 1-5 sao

# Compile model với các chỉ số cần thiết
model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=2e-5),

loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=["accuracy", "mae"])
```

- **TFBertForSequenceClassification**: Tạo mô hình BERT cho nhiệm vụ phân loại chuỗi với 5 nhãn (đánh giá từ 1 đến 5).
- **compile**: Biên dịch mô hình với:

- `optimizer`: Adam là một trong những thuật toán tối ưu phổ biến nhất trong học sâu.
- `loss`: Sử dụng `SparseCategoricalCrossentropy` cho bài toán phân loại đa lớp, với `from_logits=True` để xử lý đầu ra của mô hình.
- `metrics`: Theo dõi độ chính xác và MAE (Mean Absolute Error) trong quá trình huấn luyện.

Bước 6: Huấn Luyện Mô Hình

```
epochs = 3
history = model.fit(train_dataset, validation_data=test_dataset,
epochs=epochs)
```

- **fit**: Huấn luyện mô hình trên tập dữ liệu huấn luyện (`train_dataset`) và kiểm thử (`test_dataset`).
- `epochs`: Số lần huấn luyện toàn bộ tập dữ liệu.

Bước 7: Đánh Giá Mô Hình

```
results = model.evaluate(test_dataset)
print("Test Loss, Test Accuracy, Test MAE:", results)
```

- **evaluate**: Đánh giá mô hình trên tập kiểm thử và trả về các chỉ số (loss, accuracy, MAE).
- In ra các chỉ số để xem hiệu suất của mô hình sau khi huấn luyện.

Bước 8: Hàm Vẽ Biểu Đồ

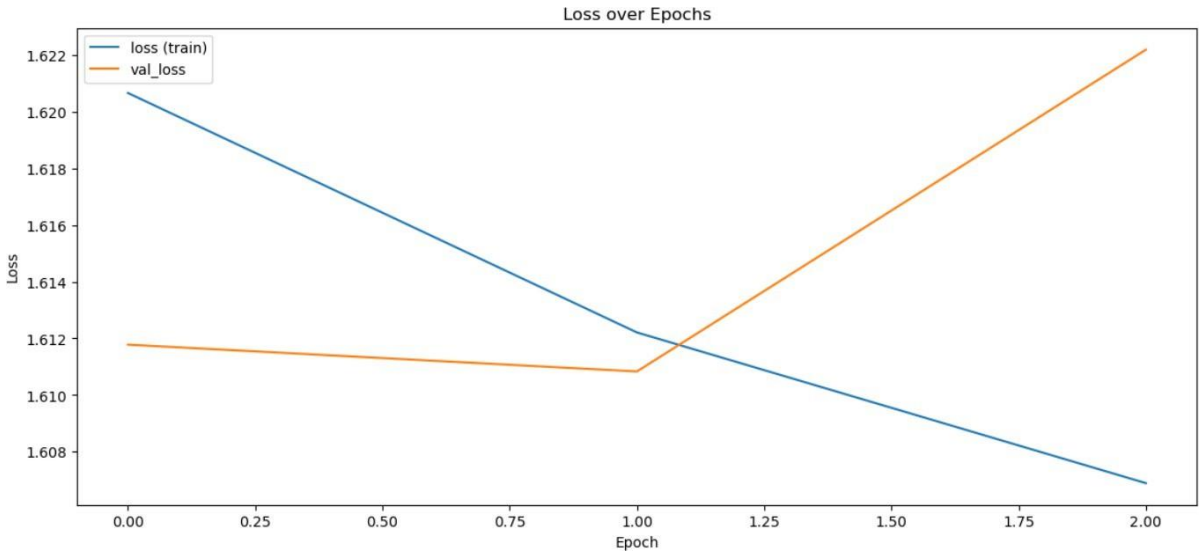
```
def plot_metrics(history, metrics, ylabel):
    plt.figure(figsize=(14, 6))
    for metric in metrics:
        if metric in history.history:
            plt.plot(history.history[metric], label=f'{metric} (train)')
        if f'val_{metric}' in history.history:
            plt.plot(history.history[f'val_{metric}'], label=f'val_{metric}')
    plt.xlabel('Epoch')
    plt.ylabel(ylabel)
    plt.title(f'{ylabel} over Epochs')
    plt.legend()
    plt.show()
```

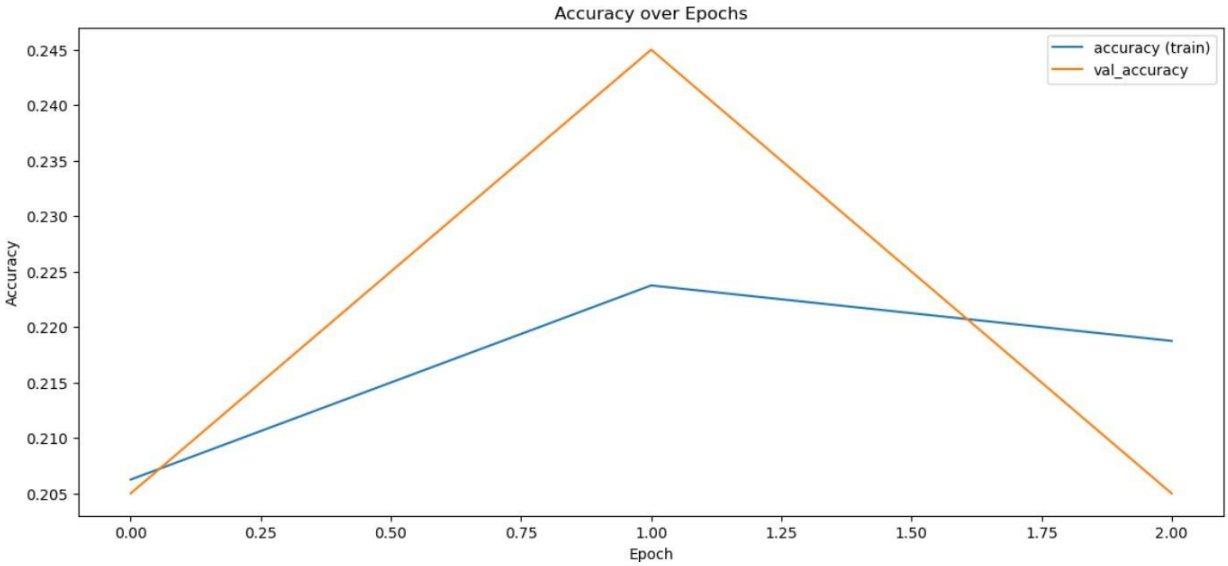
- **plot_metrics**: Hàm này vẽ biểu đồ cho các chỉ số như loss, accuracy và MAE.
- **history.history**: Chứa các giá trị của các chỉ số trong quá trình huấn luyện.
- Biểu đồ sẽ hiển thị các chỉ số qua các epoch để theo dõi sự cải thiện hoặc xấu đi của mô hình.

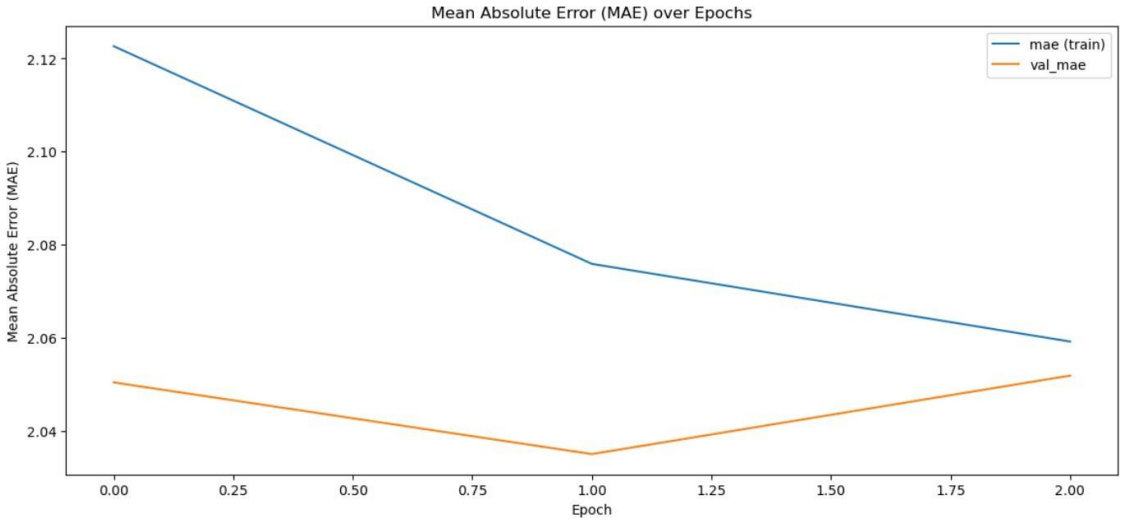
Bước 9: Vẽ Biểu Đồ

```
# Bước 9: Vẽ biểu đồ loss
plot_metrics(history, ["loss"], "Loss")

# Vẽ biểu đồ accuracy
plot_metrics(history, ["accuracy"], "Accuracy")

# Vẽ biểu đồ MAE
plot_metrics(history, ["mae"], "Mean Absolute Error (MAE)")
```

- Gọi hàm `plot_metrics` với các chỉ số cụ thể để vẽ biểu đồ tương ứng.
- Mỗi biểu đồ sẽ cung cấp cái nhìn tổng quan về hiệu suất của mô hình trong suốt quá trình huấn luyện.




Accuracy over Epochs
accuracy (train)
val_accuracy
Accuracy
Epoch
0.245
0.240
0.235
0.230
0.225
0.220
0.215
0.210
0.205
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00


Mean Absolute Error (MAE) over Epochs
mae (train)
val_mae
Mean Absolute Error (MAE)
Epoch
2.12
2.10
2.08
2.06
2.04
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00