

**ĐẠI HỌC HUẾ**  
**TRƯỜNG ĐẠI HỌC KINH TẾ**



**HỌC PHẦN: TÀI CHÍNH VÀ HỌC MÁY**  
**BÁO CÁO TỔNG KẾT HỌC PHẦN**

**GIẢNG VIÊN HƯỚNG DẪN : TS. HOÀNG HỮU TRUNG**  
**SINH VIÊN THỰC HIỆN : PHẠM TUẤN LINH**  
**MÃ SINH VIÊN : 21A5020701**

Thành phố Huế - Năm 2025

## MỤC LỤC

<b>PHẦN I: MẠNG THẦN KINH HỒI QUY RNN .....</b>	<b>3</b>
1.1. Giới thiệu bộ dữ liệu .....	3
1.2. Sử dụng các mô hình để dự đoán tỷ giá USD/JPY .....	5
1.2.1. Sử dụng SimpleRNN và LSTM dự đoán tỷ giá USD/JPY .....	5
1.2.2. Sử dụng ARIMA dự đoán tỷ giá USD/JPY .....	11
1.3. Sử dụng các mô hình để dự đoán tỷ giá GBP/USD .....	13
1.3.1. Sử dụng SimpleRNN và LSTM để dự đoán tỷ giá GBP/USD .....	13
1.3.2. Sử dụng ARIMA dự đoán tỷ giá GBP/USD .....	18
1.4. Sử dụng các mô hình để dự đoán tỷ giá EUR/USD .....	20
1.4.1. Sử dụng SimpleRNN và LSTM dự đoán tỷ giá EUR/USD .....	20
1.4.2. Sử dụng ARIMA dự đoán tỷ giá EUR/USD .....	25
1.5. Sử dụng các mô hình để dự đoán giá dầu .....	27
1.5.1. Sử dụng SimpleRNN và LSTM để dự đoán giá dầu .....	27
1.5.2. Sử dụng ARIMA dự đoán giá dầu .....	32
1.6. Sử dụng các mô hình để dự đoán giá vàng .....	33
1.6.1. Sử dụng SimpleRNN và LSTM để dự đoán giá vàng .....	33
1.6.2. Sử dụng ARIMA để dự đoán giá vàng .....	39
1.7. So sánh hiệu quả mô hình và kết luận .....	41
<b>PHẦN II: MẠNG THẦN KINH TÍCH CHẬP CNN .....</b>	<b>42</b>
2.1. Giới thiệu bộ dữ liệu .....	42
2.2. Sử dụng MobileNet để phân loại .....	43
2.2.1. Nhập các thư viện và tiền xử lý dữ liệu .....	43
2.2.2. Khởi tạo và huấn luyện mô hình .....	45
2.2.3. Đánh giá hiệu quả mô hình .....	51
2.3. Sử dụng AlexNet để phân loại .....	53
2.3.1. Nhập các thư viện và tiền xử lý dữ liệu .....	53
2.3.2. Khởi tạo và huấn luyện mô hình .....	54
2.3.3. Đánh giá hiệu quả mô hình .....	56
2.4. Sử dụng ResNet để phân loại .....	61
2.4.1. Nhập các thư viện và tiền xử lý dữ liệu .....	61
2.4.2. Khởi tạo và huấn luyện mô hình .....	62
2.4.3. Đánh giá hiệu quả mô hình .....	65
2.5. So sánh hiệu quả mô hình và kết luận .....	67

## PHẦN I: MẠNG THẦN KINH HỒI QUY RNN

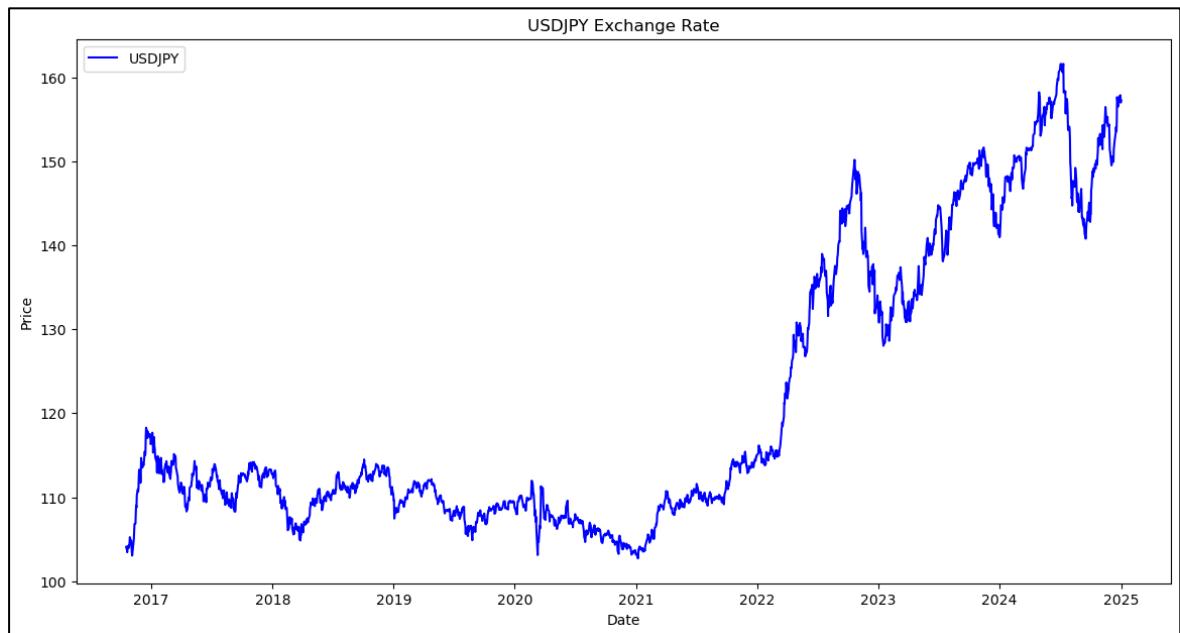
### 1.1. Giới thiệu bộ dữ liệu

Bộ dữ liệu gồm những mã ký tự (mã tiền tệ) để định nghĩa cho tên của tiền tệ do Tổ chức Tiêu chuẩn hóa quốc tế (ISO) ban hành. Danh sách mã ISO 4217 là chuẩn hiện hành trong niêm yết ngân hàng và kinh doanh trên toàn thế giới để xác định những loại tiền tệ khác nhau. Đây là những tỷ giá hối đoái và ký hiệu được giao dịch nhiều nhất trên các sàn giao dịch ngoại hối, được cung cấp bởi thư viện yfinance.

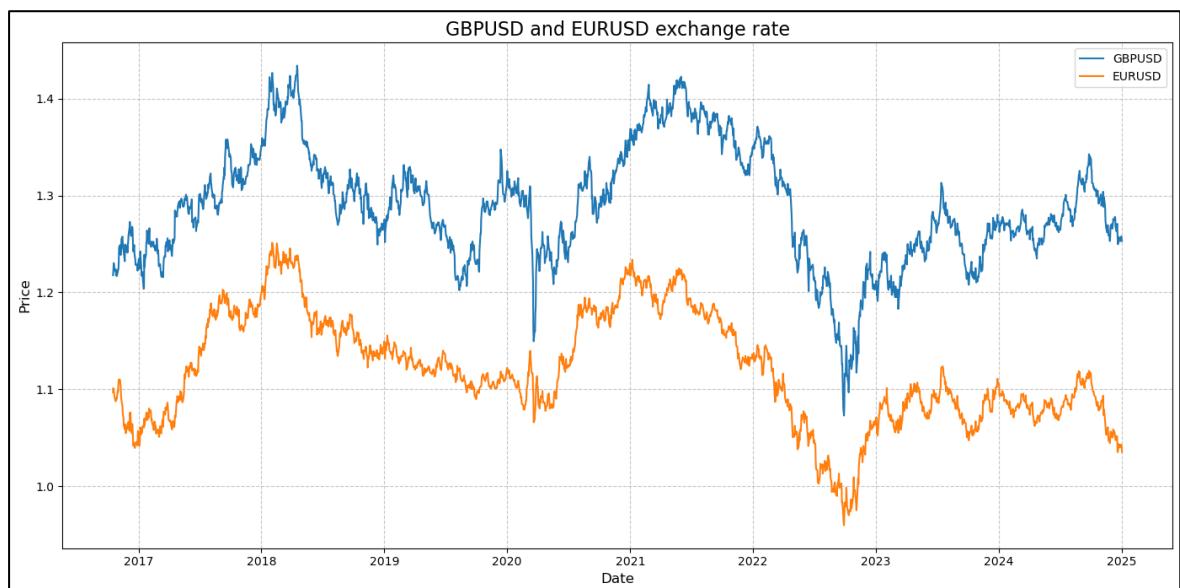
STT	Ký hiệu	Chú thích
1	USDPY (USDPY=X)	Cặp tiền tệ US Dollar/Japanese Yen, phản ánh tỷ giá chéo giữa đồng Đô la Mỹ và Yên Nhật.
2	GBPUSD (GBPUSD=X)	Cặp tiền tệ Great British Pound/US Dollar, phản ánh tỷ giá chéo giữa đồng Bảng Anh và Đô la Mỹ.
3	EURUSD (EURUSD=X)	Cặp tiền tệ Euro/US Dollar, phản ánh tỷ giá chéo giữa đồng Euro và Đô la Mỹ.
4	USOIL (CL=F)	Ký hiệu US/OIL, phản ánh giá dầu thô WTI (West Texas Intermediate) tính bằng đồng Đô la Mỹ trên mỗi thùng.
5	XAUUSD (GC=F)	Ký hiệu XAU/USD, phản ánh giá trị của vàng, tính bằng đồng Đô la Mỹ trên 1 ounce vàng.

Việc sử dụng mạng thần kinh hồi quy (Recurrent Neural Network) dự đoán những giá trị này giúp những nhà đầu tư nhỏ lẻ đưa ra quyết định giao dịch nhằm tối đa hóa lợi nhuận hoặc giảm thiểu rủi ro. Doanh nghiệp hoạt động xuất nhập khẩu có thể dựa vào dự đoán tỷ giá để bảo vệ mình trước biến động tiền tệ, giúp quản lý chi phí và lợi nhuận. Ngoài ra, việc dự đoán giá dầu và vàng giúp các công ty năng lượng, sản xuất và tổ chức tài chính lập kế hoạch chi tiêu và đầu tư dài hạn. Các tổ chức tài chính và ngân hàng trung ương cũng sử dụng thông tin này để đưa ra các chính sách tiền tệ phù hợp với tình hình kinh tế. Đồng thời, nó hỗ trợ các nhà phân tích hiểu rõ hơn về xu hướng kinh tế toàn cầu và ảnh hưởng của các sự kiện địa chính trị đối với thị trường.

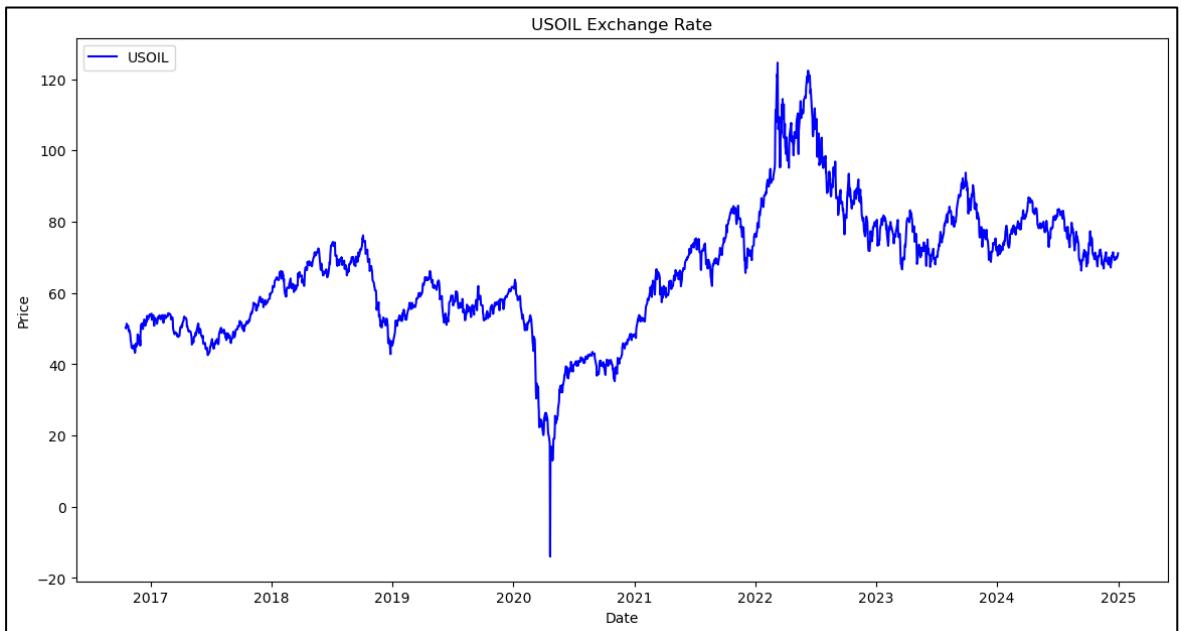
Biểu đồ tỷ giá của USD/JPY



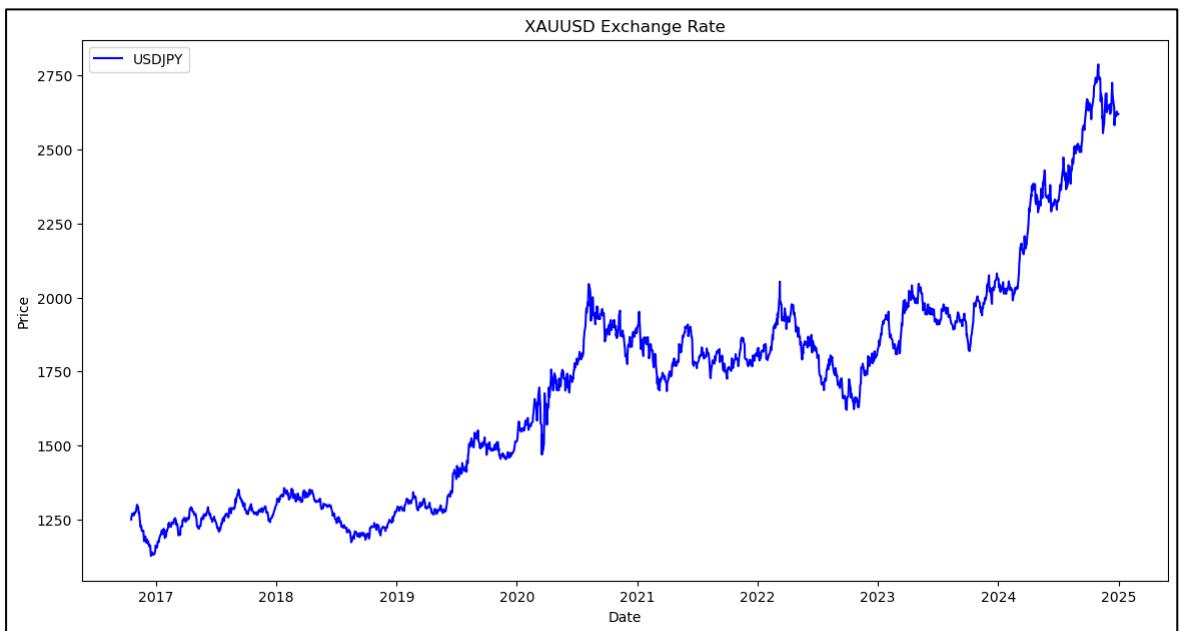
Biểu đồ tỷ giá của GBP/USD và EUR/USD



Biểu đồ giá dầu US/OIL



Biểu đồ giá vàng



## 1.2. Sử dụng các mô hình để dự đoán tỷ giá USD/JPY

### 1.2.1. Sử dụng SimpleRNN và LSTM dự đoán tỷ giá USD/JPY

```
import numpy as np
import pandas as pd
import yfinance as yf
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import SimpleRNN, LSTM, Dense, Dropout
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
```

```

stock = 'USDJPY=X'
start_date = '2019-01-01'
end_date = '2024-12-31'

data = yf.download(stock, start=start_date, end=end_date)
data = data[['Close']]

```

```

scaler = MinMaxScaler(feature_range=(0, 1))
scaled_data = scaler.fit_transform(data)

```

### 1.2.1.1. Khởi tạo và huấn luyện mô hình

```

def create_dataset(data, time_step=60):
    X, y = [], []
    for i in range(len(data) - time_step):
        X.append(data[i:i + time_step, 0])
        y.append(data[i + time_step, 0])
    return np.array(X), np.array(y)

time_step = 365
X, y = create_dataset(scaled_data, time_step)
X = X.reshape(X.shape[0], X.shape[1], 1)

train_size = int(len(X) * 0.8)
X_train, X_test = X[:train_size], X[train_size:]
y_train, y_test = y[:train_size], y[train_size:]

rnn_model = Sequential([
    SimpleRNN(50, activation='relu', return_sequences=True,
    input_shape=(time_step, 1)),
    Dropout(0.2),
    SimpleRNN(50, activation='relu'),
    Dropout(0.2),
    Dense(1)
])

rnn_model.compile(optimizer='adam', loss='mean_squared_error')
rnn_model.summary()

rnn_history = rnn_model.fit(X_train, y_train, epochs=50, batch_size=32,
validation_data=(X_test, y_test))

lstm_model = Sequential([
    LSTM(50, activation='relu', return_sequences=True, input_shape=(time_step,
1)),
    Dropout(0.2),

```

```

        LSTM(50, activation='relu'),
        Dropout(0.2),
        Dense(1)
    )

lstm_model.compile(optimizer='adam', loss='mean_squared_error')
lstm_model.summary()

lstm_history = lstm_model.fit(X_train, y_train, epochs=50, batch_size=32,
validation_data=(X_test, y_test))

rnn_predicted_prices = rnn_model.predict(X_test)
rnn_predicted_prices = scaler.inverse_transform(rnn_predicted_prices)

lstm_predicted_prices = lstm_model.predict(X_test)
lstm_predicted_prices = scaler.inverse_transform(lstm_predicted_prices)

actual_prices = scaler.inverse_transform(y_test.reshape(-1, 1))
test_dates = data.index[-len(actual_prices):]

```

Layer (type)	Output Shape	Param #
simple_rnn_20 (SimpleRNN)	(None, 365, 50)	2,600
dropout_40 (Dropout)	(None, 365, 50)	0
simple_rnn_21 (SimpleRNN)	(None, 50)	5,050
dropout_41 (Dropout)	(None, 50)	0
dense_20 (Dense)	(None, 1)	51

Total params: 7,701 (30.08 KB)

Trainable params: 7,701 (30.08 KB)

Non-trainable params: 0 (0.00 B)

Epoch 1/50

30/30 —————— 2s 52ms/step - loss: 0.0452 - val\_loss: 0.0040

Epoch 2/50

30/30 —————— 1s 50ms/step - loss: 0.0095 - val\_loss: 0.0013

Epoch 3/50

30/30 —————— 1s 50ms/step - loss: 0.0106 - val\_loss: 0.0049

Epoch 4/50

30/30 —————— 1s 50ms/step - loss: 0.0072 - val\_loss: 0.0010

Epoch 5/50

30/30 —————— 1s 50ms/step - loss: 0.0061 - val\_loss: 0.0045

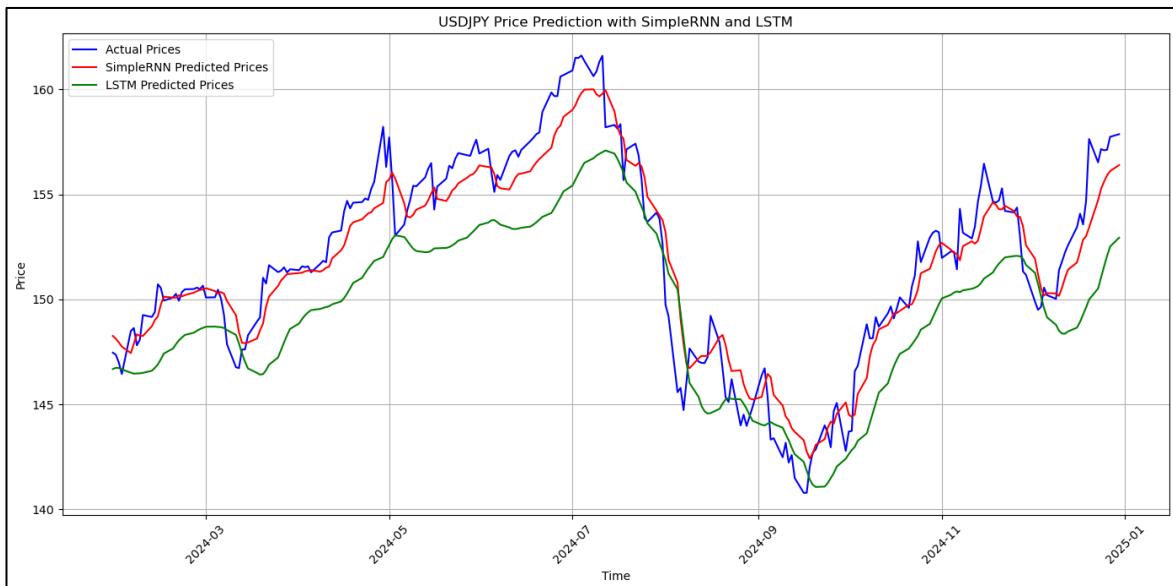
Epoch 6/50  
**30/30** ————— 1s 49ms/step - loss: 0.0061 - val\_loss: 0.0048  
 Epoch 7/50  
**30/30** ————— 2s 52ms/step - loss: 0.0056 - val\_loss: 0.0022  
 Epoch 8/50  
**30/30** ————— 2s 50ms/step - loss: 0.0045 - val\_loss: 0.0031  
 Epoch 9/50  
**30/30** ————— 2s 50ms/step - loss: 0.0040 - val\_loss: 8.8079e-04  
 Epoch 10/50  
**30/30** ————— 2s 50ms/step - loss: 0.0040 - val\_loss: 0.0099  
 Epoch 11/50  
**30/30** ————— 1s 49ms/step - loss: 0.0035 - val\_loss: 0.0011  
 Epoch 12/50  
**30/30** ————— 2s 50ms/step - loss: 0.0042 - val\_loss: 0.0065  
 Epoch 13/50  
 ...  
 Epoch 49/50  
**30/30** ————— 2s 50ms/step - loss: 0.0018 - val\_loss: 7.4947e-04  
 Epoch 50/50  
**30/30** ————— 2s 51ms/step - loss: 0.0019 - val\_loss: 5.4846e-04

Layer (type)	Output Shape	Param #
lstm_20 (LSTM)	(None, 365, 50)	10,400
dropout_42 (Dropout)	(None, 365, 50)	0
lstm_21 (LSTM)	(None, 50)	20,200
dropout_43 (Dropout)	(None, 50)	0
dense_21 (Dense)	(None, 1)	51

Total params: 30,651 (119.73 KB)  
 Trainable params: 30,651 (119.73 KB)  
 Non-trainable params: 0 (0.00 B)

Epoch 1/50  
**30/30** ————— 4s 116ms/step - loss: 0.1130 - val\_loss: 0.0629  
 Epoch 2/50  
**30/30** ————— 3s 114ms/step - loss: 0.0112 - val\_loss: 0.0035  
 Epoch 3/50  
**30/30** ————— 3s 113ms/step - loss: 0.0057 - val\_loss: 0.0025  
 Epoch 4/50  
**30/30** ————— 3s 114ms/step - loss: 0.0046 - val\_loss: 0.0020  
 Epoch 5/50

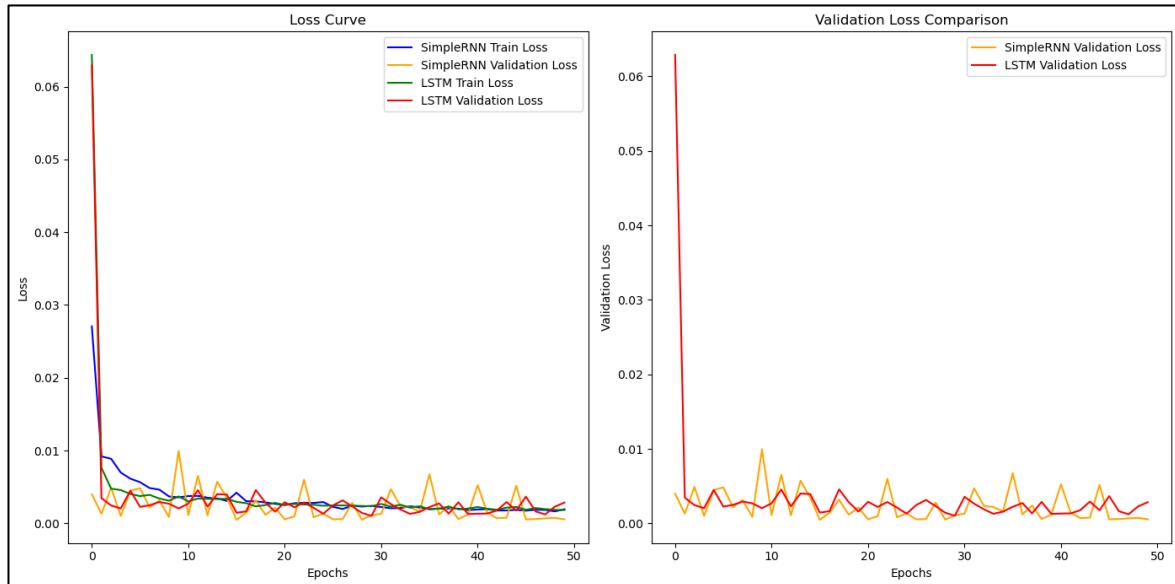
**30/30** ————— **3s** 115ms/step - loss: 0.0043 - val\_loss: 0.0045  
 Epoch 6/50  
**30/30** ————— **3s** 115ms/step - loss: 0.0037 - val\_loss: 0.0023  
 Epoch 7/50  
**30/30** ————— **3s** 115ms/step - loss: 0.0036 - val\_loss: 0.0025  
 Epoch 8/50  
**30/30** ————— **3s** 114ms/step - loss: 0.0032 - val\_loss: 0.0029  
 Epoch 9/50  
**30/30** ————— **4s** 119ms/step - loss: 0.0034 - val\_loss: 0.0027  
 Epoch 10/50  
**30/30** ————— **3s** 113ms/step - loss: 0.0036 - val\_loss: 0.0020  
 Epoch 11/50  
**30/30** ————— **3s** 115ms/step - loss: 0.0028 - val\_loss: 0.0027  
 Epoch 12/50  
**30/30** ————— **3s** 115ms/step - loss: 0.0035 - val\_loss: 0.0046  
 Epoch 13/50  
 ...  
 Epoch 50/50  
**30/30** ————— **3s** 116ms/step - loss: 0.0020 - val\_loss: 0.0029  
**8/8** ————— **0s** 20ms/step  
**8/8** ————— **0s** 42ms/step



SimpleRNN MAE: 1.1002084096272786  
 SimpleRNN MSE: 1.90551095228487  
 SimpleRNN RMSE: 1.3804024602574678  
 SimpleRNN R<sup>2</sup>: 0.9197644695128224  
 SimpleRNN MAPE: 0.7242856509821266%  
 LSTM MAE: 2.751445198059082  
 LSTM MSE: 9.907729371176295  
 LSTM RMSE: 3.147654582570377

LSTM  $R^2$ : 0.5828142991954439

LSTM MAPE: 1.793673832957366%



### 1.2.1.2. Đánh giá kết quả SimpleRNN và LSTM dự đoán tỷ giá USD/JPY

SimpleRNN:

MAE (1.10): Sai số trung bình tương ứng với 1.1 yên, mức sai số nhỏ so với biến động thông thường của cặp tỷ giá USD/JPY.

MSE (1.91): Sai số bình phương trung bình thấp, phản ánh độ chính xác cao.

RMSE (1.38): Sai số chuẩn khoảng 1.38 yên, chấp nhận được trong phân tích tỷ giá.

$R^2$  (0.9198): SimpleRNN giải thích được 91.98% biến động của tỷ giá, rất tốt.

MAPE (0.7243%): Sai số phần trăm nhỏ hơn 1%, cho thấy mô hình rất phù hợp để dự đoán cặp USD/JPY.

LSTM:

MAE (2.75): Sai số trung bình cao hơn SimpleRNN, gần 2.75 yên, thể hiện LSTM kém chính xác hơn.

MSE (9.91): Sai số bình phương trung bình lớn hơn nhiều so với SimpleRNN, cho thấy dự đoán có độ chênh lệch lớn.

RMSE (3.15): Sai số chuẩn khoảng 3.15 yên, cao hơn SimpleRNN, không tối ưu trong việc dự đoán tỷ giá USD/JPY.

$R^2$  (0.5828): Chỉ giải thích được 58.28% biến động của tỷ giá, thấp hơn đáng kể so với SimpleRNN.

MAPE (1.79%): Sai số tương đối cao hơn SimpleRNN, cho thấy mô hình LSTM không hiệu quả trong trường hợp này.

Tỷ giá thực USD/JPY thường dao động trong khoảng 110–150 yên trong các năm gần đây. Sai số của SimpleRNN (1.10 yên) là rất nhỏ so với biên độ dao động hàng ngày, trong khi sai số của LSTM (2.75 yên) là đáng kể hơn. MAPE của SimpleRNN (0.72%) cho thấy mô hình dự đoán chính xác, trong khi MAPE của LSTM (1.79%) phản ánh LSTM chưa xử lý tốt dữ liệu tỷ giá này.

Vì vậy, SimpleRNN vượt trội hơn LSTM ở tất cả các chỉ số: MAE, MSE, RMSE thấp hơn đáng kể.  $R^2$  cao hơn, giải thích tốt hơn biến động tỷ giá. MAPE thấp hơn, thể hiện khả năng dự đoán chính xác hơn.

## 1.2.2. Sử dụng ARIMA dự đoán tỷ giá USD/JPY

### 1.2.2.1. Khởi tạo mô hình

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from statsmodels.tsa.arima.model import ARIMA
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
from sklearn.preprocessing import MinMaxScaler
import yfinance as yf
from datetime import datetime
stock_symbol = 'USDJPY=X'
data = yf.download(stock_symbol, start='2024-12-01',
end=datetime.now()]['Close']
data_forecast = yf.download(stock_symbol, start='2024-12-12',
end=datetime.now().strftime('%Y-%m-%d'))['Close']
data_combined = pd.concat([data, data_forecast])

scaler = MinMaxScaler()
data_scaled = scaler.fit_transform(data_combined.values.reshape(-1, 1))
data_train = data_scaled[:len(data)]
data_test = data_scaled[len(data):]

order = (60, 2, 3)
model = ARIMA(data_train, order=order)
model_fit = model.fit()

future_steps = len(data_test)
forecast_scaled = model_fit.forecast(steps=future_steps)
forecast = scaler.inverse_transform(forecast_scaled.reshape(-1, 1))
```

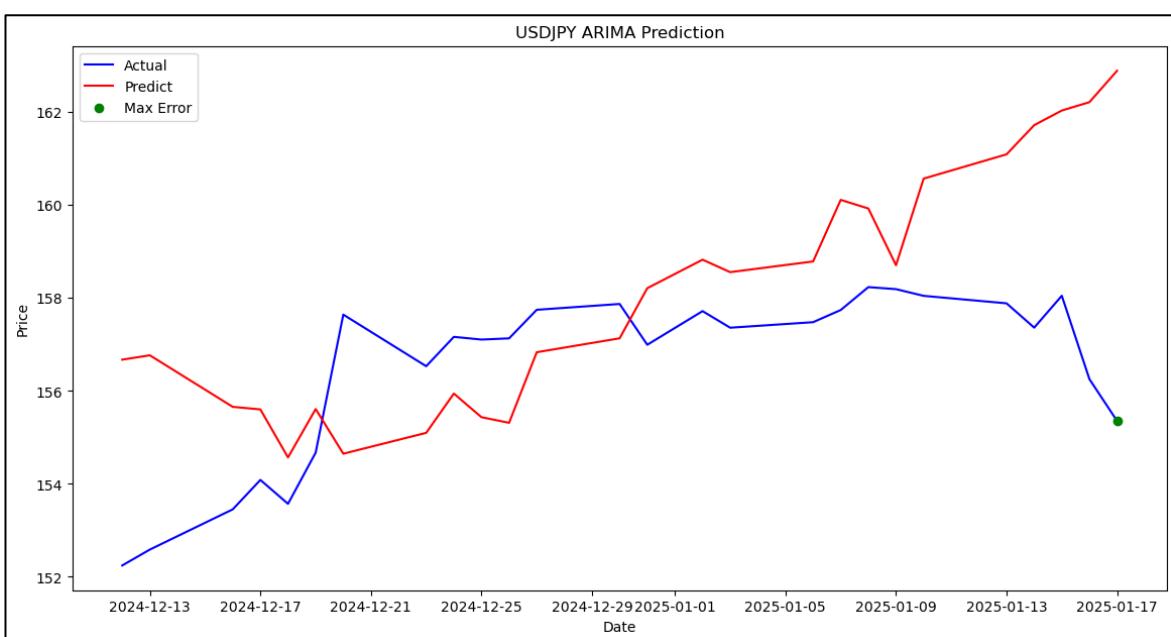
```
data_test_actual = scaler.inverse_transform(data_test.reshape(-1, 1))

errors = np.abs(data_test_actual - forecast)
max_error_index = np.argmax(errors)
max_error_actual = data_test_actual[max_error_index]
max_error_predicted = forecast[max_error_index]

print(f"Max Error Index: {max_error_index}")
print(f"Max Error Actual Value: {max_error_actual}")
print(f"Max Error Predicted Value: {max_error_predicted}")
```

[\*\*\*\*\*100%\*\*\*\*\*] 1 of 1 completed  
[\*\*\*\*\*100%\*\*\*\*\*] 1 of 1 completed

```
Max Error Index: 25  
Max Error Actual Value: [155.35600281]  
Max Error Predicted Value: [162.89242514]
```



#### **1.2.2.2. Đánh giá kết quả ARIMA dự đoán tỷ giá USD/JPY**

Giá trị thực tế (Actual Value) tại điểm có sai số lớn nhất là 155.3560. Giá trị dự đoán (Predicted Value) tại cùng thời điểm là 162.8924. Sai số lớn nhất (Max Error) là  $162.8924 - 155.3560 = 7.5364$  (tương đương 4,85%). Sai số này phản ánh mô hình ARIMA không nắm bắt được các biến động mạnh trong dữ liệu thực tế.

### 1.3. Sử dụng các mô hình để dự đoán tỷ giá GBP/USD

#### 1.3.1. Sử dụng SimpleRNN và LSTM để dự đoán tỷ giá GBP/USD

```
stock = 'GBPUUSD=X'  
start_date = '2019-01-01'  
end_date = '2024-12-31'  
  
data = yf.download(stock, start=start_date, end=end_date)  
data = data[['Close']]  
  
scaler = MinMaxScaler(feature_range=(0, 1))  
scaled_data = scaler.fit_transform(data)
```

##### 1.3.1.1. Khởi tạo và huấn luyện mô hình

```
def create_dataset(data, time_step=60):  
    X, y = [], []  
    for i in range(len(data) - time_step):  
        X.append(data[i:i + time_step, 0])  
        y.append(data[i + time_step, 0])  
    return np.array(X), np.array(y)  
  
time_step = 365  
X, y = create_dataset(scaled_data, time_step)  
X = X.reshape(X.shape[0], X.shape[1], 1)  
  
train_size = int(len(X) * 0.8)  
X_train, X_test = X[:train_size], X[train_size:]  
y_train, y_test = y[:train_size], y[train_size:]  
  
rnn_model = Sequential([  
    SimpleRNN(50, activation='relu', return_sequences=True,  
    input_shape=(time_step, 1)),  
    Dropout(0.2),  
    SimpleRNN(50, activation='relu'),  
    Dropout(0.2),  
    Dense(1)  
])  
  
rnn_model.compile(optimizer='adam', loss='mean_squared_error')  
rnn_model.summary()  
  
rnn_history = rnn_model.fit(X_train, y_train, epochs=50, batch_size=32,  
validation_data=(X_test, y_test))  
  
lstm_model = Sequential([
```

```

        LSTM(50, activation='relu', return_sequences=True, input_shape=(time_step,
1)),
        Dropout(0.2),
        LSTM(50, activation='relu'),
        Dropout(0.2),
        Dense(1)
    )

lstm_model.compile(optimizer='adam', loss='mean_squared_error')
lstm_model.summary()

lstm_history = lstm_model.fit(X_train, y_train, epochs=50, batch_size=32,
validation_data=(X_test, y_test))

rnn_predicted_prices = rnn_model.predict(X_test)
rnn_predicted_prices = scaler.inverse_transform(rnn_predicted_prices)

lstm_predicted_prices = lstm_model.predict(X_test)
lstm_predicted_prices = scaler.inverse_transform(lstm_predicted_prices)

actual_prices = scaler.inverse_transform(y_test.reshape(-1, 1))
test_dates = data.index[-len(actual_prices):]

```

Layer (type)	Output Shape	Param #
simple_rnn_14 (SimpleRNN)	(None, 365, 50)	2,600
dropout_28 (Dropout)	(None, 365, 50)	0
simple_rnn_15 (SimpleRNN)	(None, 50)	5,050
dropout_29 (Dropout)	(None, 50)	0
dense_14 (Dense)	(None, 1)	51

Total params: 7,701 (30.08 KB)  
Trainable params: 7,701 (30.08 KB)  
Non-trainable params: 0 (0.00 B)

Epoch 1/50  
**30/30** ————— 6s 150ms/step - loss: 0.2230 - val\_loss: 0.0102  
Epoch 2/50  
**30/30** ————— 3s 93ms/step - loss: 0.0310 - val\_loss: 0.0021  
Epoch 3/50  
**30/30** ————— 3s 84ms/step - loss: 0.0197 - val\_loss: 0.0015

Epoch 4/50  
**30/30** —————— 2s 75ms/step - loss: 0.0179 - val\_loss: 5.2474e-04

Epoch 5/50  
**30/30** —————— 3s 88ms/step - loss: 0.0133 - val\_loss: 0.0025

Epoch 6/50  
**30/30** —————— 2s 58ms/step - loss: 0.0129 - val\_loss: 4.3301e-04

Epoch 7/50  
**30/30** —————— 2s 59ms/step - loss: 0.0100 - val\_loss: 0.0010

Epoch 8/50  
**30/30** —————— 2s 77ms/step - loss: 0.0086 - val\_loss: 0.0011

Epoch 9/50  
**30/30** —————— 2s 62ms/step - loss: 0.0089 - val\_loss: 0.0011

Epoch 10/50  
**30/30** —————— 2s 59ms/step - loss: 0.0087 - val\_loss: 0.0011

Epoch 11/50  
**30/30** —————— 2s 53ms/step - loss: 0.0092 - val\_loss: 4.4031e-04

Epoch 12/50  
**30/30** —————— 2s 51ms/step - loss: 0.0084 - val\_loss: 0.0013

Epoch 13/50  
 ...  
 Epoch 49/50  
**30/30** —————— 2s 52ms/step - loss: 0.0044 - val\_loss: 6.4743e-04

Epoch 50/50  
**30/30** —————— 2s 52ms/step - loss: 0.0034 - val\_loss: 4.7953e-04

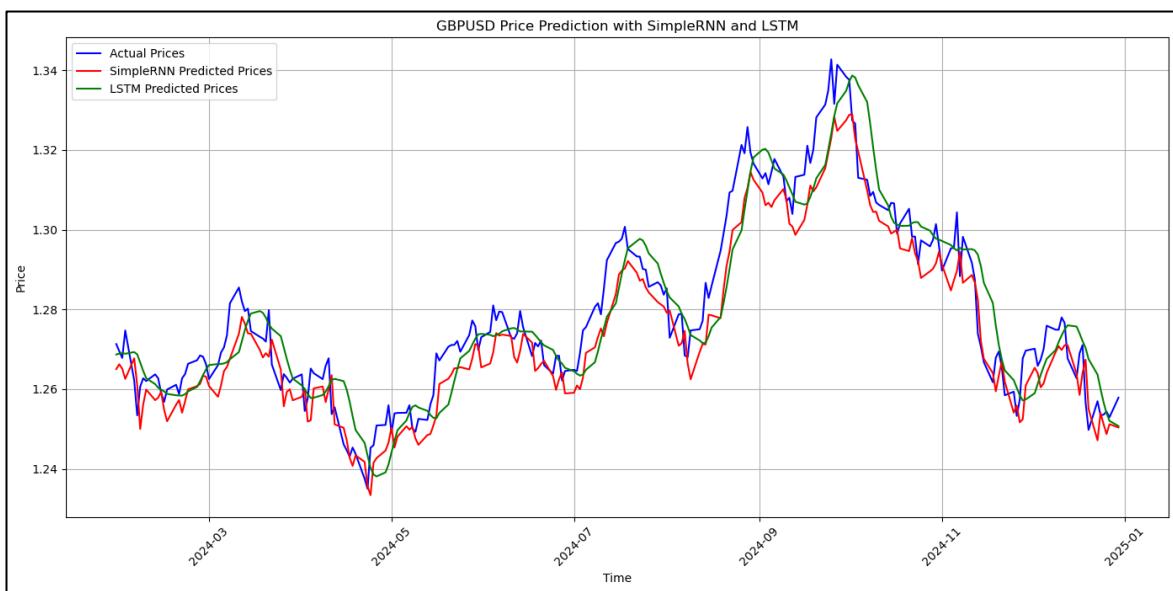
Layer (type)	Output Shape	Param #
lstm_14 (LSTM)	(None, 365, 50)	10,400
dropout_30 (Dropout)	(None, 365, 50)	0
lstm_15 (LSTM)	(None, 50)	20,200
dropout_31 (Dropout)	(None, 50)	0
dense_15 (Dense)	(None, 1)	51

**Total params:** 30,651 (119.73 KB)  
**Trainable params:** 30,651 (119.73 KB)  
**Non-trainable params:** 0 (0.00 B)

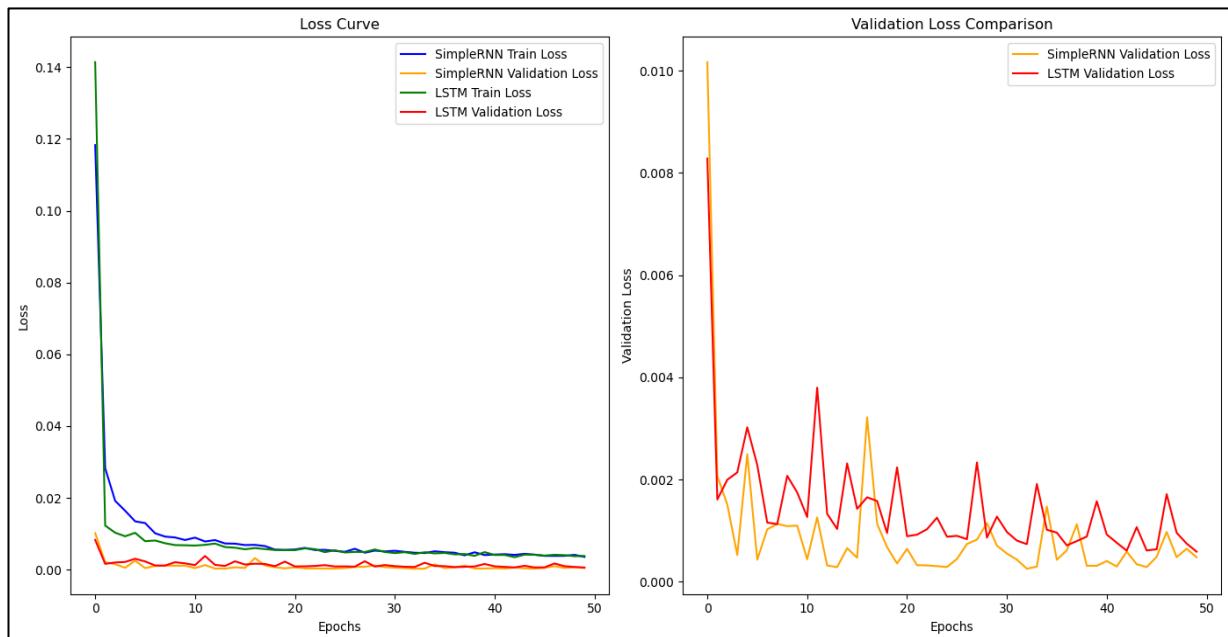
Epoch 1/50  
**30/30** —————— 4s 119ms/step - loss: 0.2578 - val\_loss: 0.0083

Epoch 2/50  
**30/30** —————— 4s 118ms/step - loss: 0.0130 - val\_loss: 0.0016

Epoch 3/50  
**30/30** ————— 3s 115ms/step - loss: 0.0107 - val\_loss: 0.0020  
 Epoch 4/50  
**30/30** ————— 4s 128ms/step - loss: 0.0094 - val\_loss: 0.0021  
 Epoch 5/50  
**30/30** ————— 4s 129ms/step - loss: 0.0100 - val\_loss: 0.0030  
 Epoch 6/50  
**30/30** ————— 4s 116ms/step - loss: 0.0078 - val\_loss: 0.0023  
 Epoch 7/50  
**30/30** ————— 4s 118ms/step - loss: 0.0076 - val\_loss: 0.0012  
 Epoch 8/50  
**30/30** ————— 4s 127ms/step - loss: 0.0076 - val\_loss: 0.0011  
 Epoch 9/50  
**30/30** ————— 4s 135ms/step - loss: 0.0068 - val\_loss: 0.0021  
 Epoch 10/50  
**30/30** ————— 4s 133ms/step - loss: 0.0070 - val\_loss: 0.0017  
 Epoch 11/50  
**30/30** ————— 4s 126ms/step - loss: 0.0073 - val\_loss: 0.0013  
 Epoch 12/50  
**30/30** ————— 4s 131ms/step - loss: 0.0070 - val\_loss: 0.0038  
 Epoch 13/50  
 ...  
 Epoch 50/50  
**30/30** ————— 4s 121ms/step - loss: 0.0038 - val\_loss: 5.8967e-04  
**8/8** ————— 0s 20ms/step  
**8/8** ————— 0s 44ms/step



SimpleRNN MAE: 0.006480283538500468  
 SimpleRNN MSE: 5.871650282180951e-05  
 SimpleRNN RMSE: 0.007662669431850073  
 SimpleRNN R<sup>2</sup>: 0.8837798466121317  
 SimpleRNN MAPE: 0.5047139781941262%  
 LSTM MAE: 0.00671647290388743  
 LSTM MSE: 7.220266443267084e-05  
 LSTM RMSE: 0.008497215098646782  
 LSTM R<sup>2</sup>: 0.857086094503215  
 LSTM MAPE: 0.5239471723343855%



### 1.3.1.2. Đánh giá kết quả SimpleRNN và LSTM dự đoán tỷ giá GBP/USD

SimpleRNN:

MAE (0.00648): Sai số trung bình tương ứng với 64.8 pip (1 pip = 0.0001 điểm).

Đây là mức chấp nhận được đối với dữ liệu tỷ giá hối đoái.

MSE (5.87e-05): Sai số bình phương trung bình thấp, cho thấy các dự đoán gần sát với thực tế.

RMSE (0.00766): Sai số chuẩn khoảng 76.6 pip, phản ánh mức độ chênh lệch giữa giá dự đoán và giá thực.

R<sup>2</sup> (0.8838): SimpleRNN giải thích được 88.38% biến động của tỷ giá, một kết quả tốt.

MAPE (0.5047%): Sai số phần trăm rất nhỏ, chỉ khoảng 0.5%, cho thấy SimpleRNN có khả năng dự đoán chính xác.

LSTM:

MAE (0.00672): Sai số trung bình khoảng 67.2 pip, cao hơn SimpleRNN nhưng vẫn ở mức chấp nhận được.

MSE (7.22e-05): Sai số bình phương trung bình lớn hơn SimpleRNN, chứng tỏ LSTM dự đoán kém chính xác hơn trong trường hợp này.

RMSE (0.00850): Sai số chuẩn khoảng 85.0 pip, cao hơn SimpleRNN, cho thấy mức độ dự đoán kém hơn.

R<sup>2</sup> (0.8571): Mô hình giải thích được 85.71% biến động của tỷ giá, thấp hơn SimpleRNN.

MAPE (0.5239%): Sai số tương đối cũng cao hơn SimpleRNN, thể hiện rằng LSTM không tối ưu bằng SimpleRNN.

Tỷ giá thực GBP/USD dao động trong khoảng 1.20–1.40 trong những năm gần đây. Sai số của SimpleRNN (0.00648) và LSTM (0.00672) tương ứng với mức biến động rất nhỏ (64.8–67.2 pip) so với biến độ dao động hàng ngày của cặp tiền này. MAPE dưới 1% cho cả hai mô hình cho thấy độ chính xác cao và có thể áp dụng cho dự đoán thực tế.

Trong trường hợp này, SimpleRNN vượt trội hơn LSTM về hầu hết các chỉ số: MAE, MSE, RMSE và MAPE đều thấp hơn. R<sup>2</sup> cao hơn, chứng tỏ SimpleRNN giải thích tốt hơn các biến động trong dữ liệu tỷ giá GBP/USD.

### 1.3.2. Sử dụng ARIMA dự đoán tỷ giá GBP/USD

#### 1.3.2.1. Khởi tạo mô hình

```
stock_symbol = 'GBPUSD=X'
data = yf.download(stock_symbol, start='2024-12-01',
end=datetime.now()['Close']
data_forecast = yf.download(stock_symbol, start='2024-12-12',
end=datetime.now().strftime('%Y-%m-%d'))['Close']
data_combined = pd.concat([data, data_forecast])

scaler = MinMaxScaler()
data_scaled = scaler.fit_transform(data_combined.values.reshape(-1, 1))
data_train = data_scaled[:len(data)]
data_test = data_scaled[len(data):]
```

```

order = (45, 2, 9)
model = ARIMA(data_train, order=order)
model_fit = model.fit()

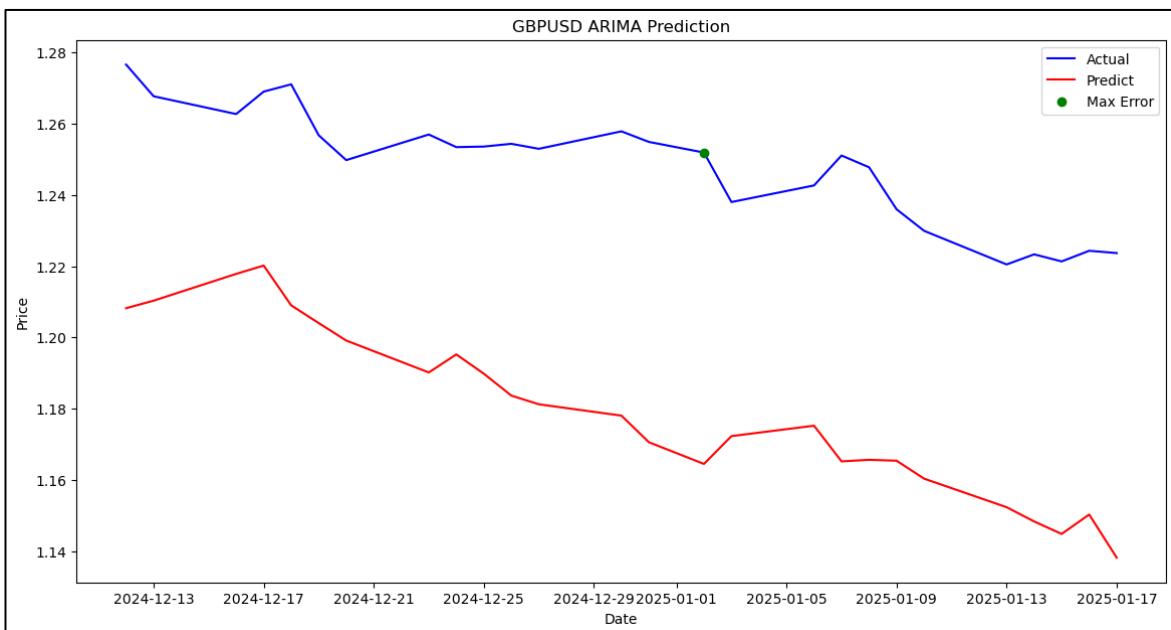
future_steps = len(data_test)
forecast_scaled = model_fit.forecast(steps=future_steps)
forecast = scaler.inverse_transform(forecast_scaled.reshape(-1, 1))
data_test_actual = scaler.inverse_transform(data_test.reshape(-1, 1))

errors = np.abs(data_test_actual - forecast)
max_error_index = np.argmax(errors)
max_error_actual = data_test_actual[max_error_index]
max_error_predicted = forecast[max_error_index]

```

[\*\*\*\*\*100%\*\*\*\*\*] 1 of 1 completed  
[\*\*\*\*\*100%\*\*\*\*\*] 1 of 1 completed

Max Error Index: 14  
Max Error Actual Value: [1.25192487]  
Max Error Predicted Value: [1.16446659]



### 1.3.2.2. Đánh giá kết quả ARIMA dự đoán tỷ giá GBP/USD

Sai số lớn nhất =  $1.25192487 - 1.16446659 = 0.874528$ , tương đương 6,98% giá thị trường tại thời điểm xảy ra sai số, điều này cho thấy mô hình ARIMA không hoàn toàn nắm bắt được các biến động mạnh của tỷ giá GBP/USD. Tỷ giá này thường

có mức độ biến động lớn hơn so với EUR/USD do ảnh hưởng từ cả nền kinh tế Anh và sự nhạy cảm với các sự kiện kinh tế toàn cầu, khiến ARIMA khó dự đoán chính xác hơn. Mặc dù sai số vẫn ở mức chấp nhận được, việc sử dụng ARIMA để dự đoán cặp tiền tệ này vào thực tế là không đáng tin cậy.

## 1.4. Sử dụng các mô hình để dự đoán tỷ giá EUR/USD

### 1.4.1. Sử dụng SimpleRNN và LSTM dự đoán tỷ giá EUR/USD

```
stock = 'EURUSD=X'  
start_date = '2019-01-01'  
end_date = '2024-12-31'  
  
data = yf.download(stock, start=start_date, end=end_date)  
data = data[['Close']]  
  
scaler = MinMaxScaler(feature_range=(0, 1))  
scaled_data = scaler.fit_transform(data)
```

#### 1.4.1.1. Khởi tạo và huấn luyện mô hình

```
def create_dataset(data, time_step=60):  
    X, y = [], []  
    for i in range(len(data) - time_step):  
        X.append(data[i:i + time_step, 0])  
        y.append(data[i + time_step, 0])  
    return np.array(X), np.array(y)  
  
time_step = 365  
X, y = create_dataset(scaled_data, time_step)  
X = X.reshape(X.shape[0], X.shape[1], 1)  
  
train_size = int(len(X) * 0.8)  
X_train, X_test = X[:train_size], X[train_size:]  
y_train, y_test = y[:train_size], y[train_size:]  
  
rnn_model = Sequential([  
    SimpleRNN(50, activation='relu', return_sequences=True,  
    input_shape=(time_step, 1)),  
    Dropout(0.2),  
    SimpleRNN(50, activation='relu'),  
    Dropout(0.2),  
    Dense(1)  
])  
  
rnn_model.compile(optimizer='adam', loss='mean_squared_error')
```

```

rnn_model.summary()

rnn_history = rnn_model.fit(X_train, y_train, epochs=50, batch_size=32,
validation_data=(X_test, y_test))

lstm_model = Sequential([
    LSTM(50, activation='relu', return_sequences=True, input_shape=(time_step,
1)),
    Dropout(0.2),
    LSTM(50, activation='relu'),
    Dropout(0.2),
    Dense(1)
])

lstm_model.compile(optimizer='adam', loss='mean_squared_error')
lstm_model.summary()

lstm_history = lstm_model.fit(X_train, y_train, epochs=50, batch_size=32,
validation_data=(X_test, y_test))

rnn_predicted_prices = rnn_model.predict(X_test)
rnn_predicted_prices = scaler.inverse_transform(rnn_predicted_prices)

lstm_predicted_prices = lstm_model.predict(X_test)
lstm_predicted_prices = scaler.inverse_transform(lstm_predicted_prices)

actual_prices = scaler.inverse_transform(y_test.reshape(-1, 1))
test_dates = data.index[-len(actual_prices):]

```

Layer (type)	Output Shape	Param #
simple_rnn_12 (SimpleRNN)	(None, 365, 50)	2,600
dropout_24 (Dropout)	(None, 365, 50)	0
simple_rnn_13 (SimpleRNN)	(None, 50)	5,050
dropout_25 (Dropout)	(None, 50)	0
dense_12 (Dense)	(None, 1)	51

Total params: 7,701 (30.08 KB)  
Trainable params: 7,701 (30.08 KB)  
Non-trainable params: 0 (0.00 B)

Epoch 1/50

```

30/30 ━━━━━━━━━━ 2s 53ms/step - loss: 0.2480 - val_loss: 0.0040
Epoch 2/50
30/30 ━━━━━━━━ 1s 48ms/step - loss: 0.0274 - val_loss: 3.8700e-04
Epoch 3/50
30/30 ━━━━━━ 1s 49ms/step - loss: 0.0176 - val_loss: 0.0016
Epoch 4/50
30/30 ━━━━ 1s 48ms/step - loss: 0.0169 - val_loss: 5.3825e-04
Epoch 5/50
30/30 ━━ 1s 49ms/step - loss: 0.0132 - val_loss: 3.3498e-04
Epoch 6/50
30/30 ━ 1s 48ms/step - loss: 0.0135 - val_loss: 0.0019
Epoch 7/50
30/30 1s 49ms/step - loss: 0.0111 - val_loss: 3.4278e-04
Epoch 8/50
30/30 1s 48ms/step - loss: 0.0091 - val_loss: 4.4672e-04
Epoch 9/50
30/30 1s 48ms/step - loss: 0.0096 - val_loss: 9.6291e-04
Epoch 10/50
30/30 1s 49ms/step - loss: 0.0071 - val_loss: 4.3621e-04
Epoch 11/50
30/30 1s 48ms/step - loss: 0.0066 - val_loss: 7.6311e-04
Epoch 12/50
30/30 1s 49ms/step - loss: 0.0079 - val_loss: 8.6673e-04
Epoch 13/50
...
Epoch 49/50
30/30 ━━━━━━ 2s 52ms/step - loss: 0.0034 - val_loss: 2.8478e-04
Epoch 50/50
30/30 ━━━━ 2s 52ms/step - loss: 0.0036 - val_loss: 2.9626e-04

```

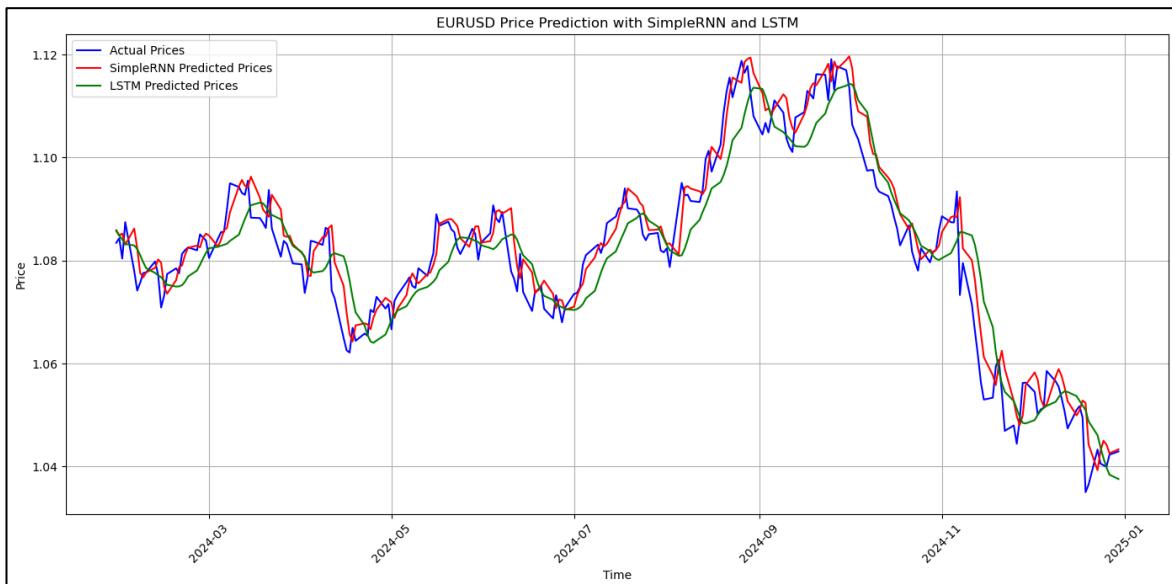
Layer (type)	Output Shape	Param #
lstm_12 (LSTM)	(None, 365, 50)	10,400
dropout_26 (Dropout)	(None, 365, 50)	0
lstm_13 (LSTM)	(None, 50)	20,200
dropout_27 (Dropout)	(None, 50)	0
dense_13 (Dense)	(None, 1)	51

Total params: 30,651 (119.73 KB)

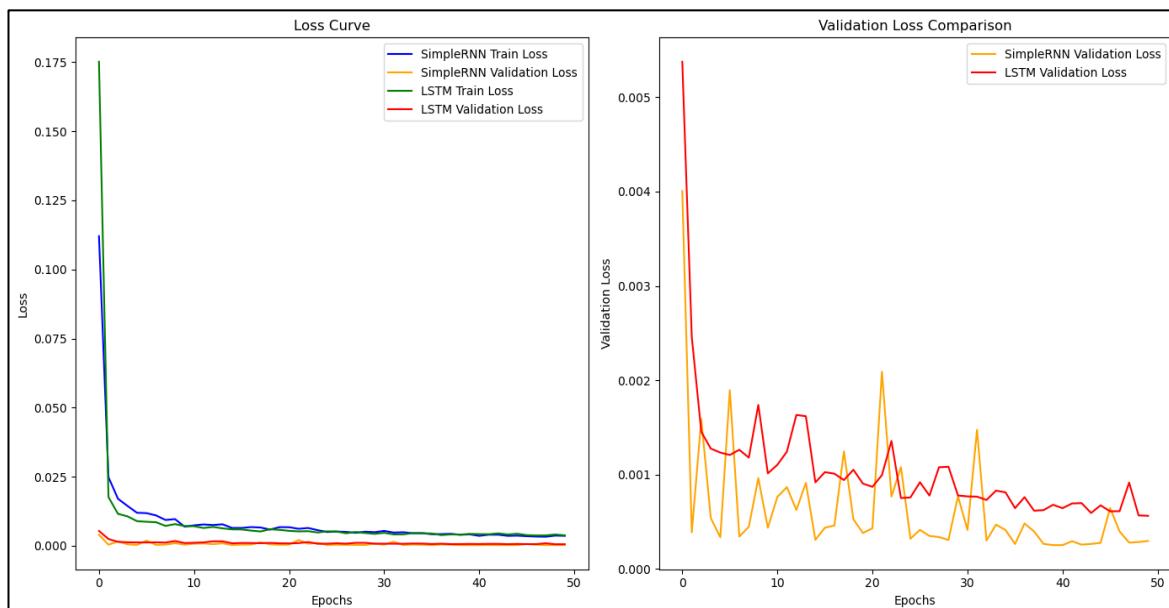
Trainable params: 30,651 (119.73 KB)

Non-trainable params: 0 (0.00 B)

Epoch 1/50  
**30/30** ————— **5s** 122ms/step - loss: 0.2859 - val\_loss: 0.0054  
 Epoch 2/50  
**30/30** ————— **4s** 121ms/step - loss: 0.0223 - val\_loss: 0.0025  
 Epoch 3/50  
**30/30** ————— **4s** 116ms/step - loss: 0.0122 - val\_loss: 0.0015  
 Epoch 4/50  
**30/30** ————— **4s** 118ms/step - loss: 0.0108 - val\_loss: 0.0013  
 Epoch 5/50  
**30/30** ————— **4s** 121ms/step - loss: 0.0094 - val\_loss: 0.0012  
 Epoch 6/50  
**30/30** ————— **4s** 127ms/step - loss: 0.0091 - val\_loss: 0.0012  
 Epoch 7/50  
**30/30** ————— **3s** 115ms/step - loss: 0.0088 - val\_loss: 0.0013  
 Epoch 8/50  
**30/30** ————— **4s** 120ms/step - loss: 0.0073 - val\_loss: 0.0012  
 Epoch 9/50  
**30/30** ————— **4s** 129ms/step - loss: 0.0084 - val\_loss: 0.0017  
 Epoch 10/50  
**30/30** ————— **4s** 134ms/step - loss: 0.0074 - val\_loss: 0.0010  
 Epoch 11/50  
**30/30** ————— **4s** 141ms/step - loss: 0.0072 - val\_loss: 0.0011  
 Epoch 12/50  
**30/30** ————— **4s** 120ms/step - loss: 0.0064 - val\_loss: 0.0012  
 Epoch 13/50  
 ...  
 Epoch 50/50  
**30/30** ————— **4s** 118ms/step - loss: 0.0040 - val\_loss: 5.6286e-04  
**8/8** ————— **0s** 22ms/step  
**8/8** ————— **0s** 42ms/step



SimpleRNN MAE: 0.0036404152711232503  
 SimpleRNN MSE: 2.232166794750393e-05  
 SimpleRNN RMSE: 0.004724581245730031  
 SimpleRNN R<sup>2</sup>: 0.930457907268083  
 SimpleRNN MAPE: 0.3372004028393359%  
 LSTM MAE: 0.005236815909544627  
 LSTM MSE: 4.2408885365515424e-05  
 LSTM RMSE: 0.006512210482279839  
 LSTM R<sup>2</sup>: 0.8678771386761133  
 LSTM MAPE: 0.4839991592485328%



#### 1.4.1.2. Đánh giá kết quả mô hình dự đoán tỷ giá EUR/USD

SimpleRNN:

MAE (0.00364): Sai số trung bình rất nhỏ (chỉ 0.00364), tương ứng với 36.4 pip (1 pip = 0.0001 đổi với EUR/USD). Đây là sai số chấp nhận được trong giao dịch ngoại hối.

MSE (2.23e-05): Sai số bình phương trung bình rất thấp, cho thấy các dự đoán gần sát với giá trị thực tế.

RMSE (0.00472): Sai số chuẩn chỉ khoảng 47.2 pip, phù hợp với mức biến động thông thường của EUR/USD.

R<sup>2</sup> (0.93): Mô hình SimpleRNN giải thích được 93% biến động của tỷ giá, mô hình hoạt động rất tốt.

MAPE (0.3372%): Sai số phần trăm tương đối rất nhỏ, chỉ 0.34%, cho thấy SimpleRNN dự đoán hiệu quả và đáng tin cậy.

LSTM:

MAE (0.00524): Sai số trung bình khoảng 52.4 pip, lớn hơn SimpleRNN nhưng vẫn chấp nhận được.

MSE (4.24e-05): Sai số bình phương trung bình cao hơn SimpleRNN, thể hiện rằng mô hình LSTM ít hiệu quả hơn trong việc học dữ liệu EUR/USD.

RMSE (0.00651): Sai số chuẩn khoảng 65.1 pip, lớn hơn SimpleRNN, nhưng vẫn trong ngưỡng chấp nhận được.

$R^2$  (0.87): Mô hình giải thích được 87% biến động của tỷ giá, thấp hơn SimpleRNN.

MAPE (0.4840%): Sai số tương đối nhỏ, nhưng cao hơn SimpleRNN (0.34%), cho thấy LSTM không hoạt động tốt bằng SimpleRNN trong trường hợp này.

Đối chiếu với tỷ giá thực EUR/USD dao động trong khoảng 1.05–1.15 trong những năm gần đây. Sai số của SimpleRNN (0.00364) và LSTM (0.00524) tương ứng với mức biến động rất nhỏ (chỉ vài pip đến vài chục pip), nghĩa là cả hai mô hình đều dự đoán khá tốt. MAPE thấp (SimpleRNN: 0.34%, LSTM: 0.48%) cho thấy khả năng dự đoán tỷ giá EUR/USD với độ chính xác cao, đặc biệt là SimpleRNN.

Do đó, SimpleRNN vượt trội hơn LSTM trong trường hợp này, với mọi chỉ số lỗi (MAE, MSE, RMSE, MAPE) thấp hơn và hệ số  $R^2$  cao hơn.

#### 1.4.2. Sử dụng ARIMA dự đoán tỷ giá EUR/USD

##### 1.4.2.1. Khởi tạo mô hình

```
stock_symbol = 'EURUSD=X'
data = yf.download(stock_symbol, start='2024-12-01',
end=datetime.now()['Close']
data_forecast = yf.download(stock_symbol, start='2024-12-12',
end=datetime.now().strftime('%Y-%m-%d'))['Close']
data_combined = pd.concat([data, data_forecast])

scaler = MinMaxScaler()
data_scaled = scaler.fit_transform(data_combined.values.reshape(-1, 1))
data_train = data_scaled[:len(data)]
data_test = data_scaled[len(data):]
order = (58, 2, 5)
model = ARIMA(data_train, order=order)
```

```

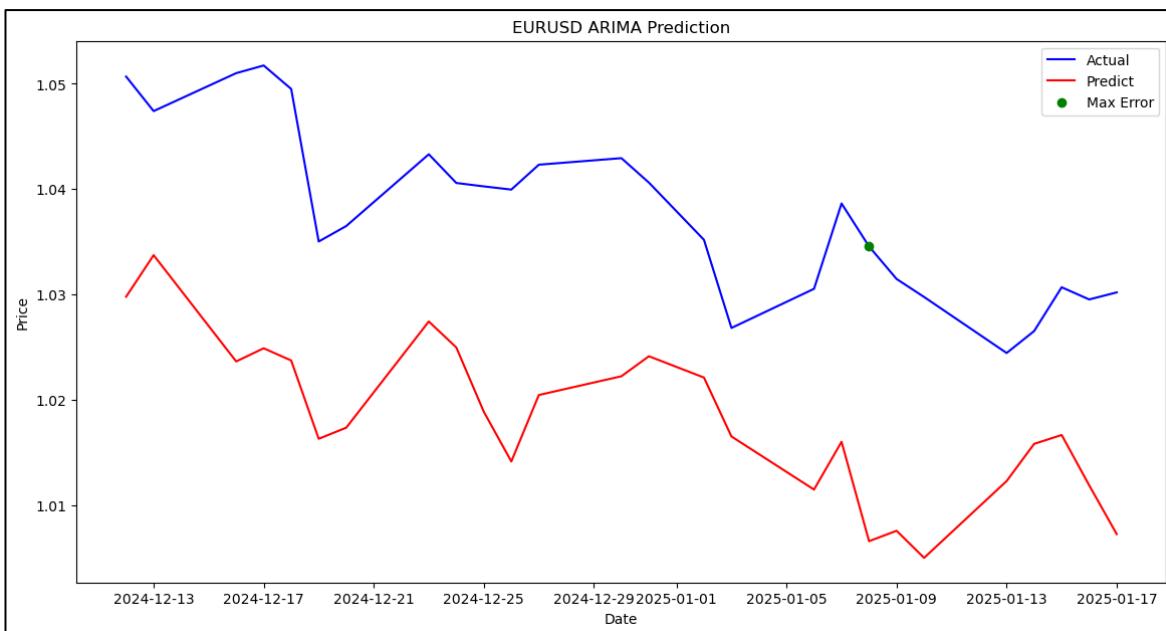
model_fit = model.fit()

future_steps = len(data_test)
forecast_scaled = model_fit.forecast(steps=future_steps)
forecast = scaler.inverse_transform(forecast_scaled.reshape(-1, 1))
data_test_actual = scaler.inverse_transform(data_test.reshape(-1, 1))
errors = np.abs(data_test_actual - forecast)
max_error_index = np.argmax(errors)
max_error_actual = data_test_actual[max_error_index]
max_error_predicted = forecast[max_error_index]

```

[\*\*\*\*\*100%\*\*\*\*\*] 1 of 1 completed  
[\*\*\*\*\*100%\*\*\*\*\*] 1 of 1 completed

Max Error Index: 18  
Max Error Actual Value: [1.03455412]  
Max Error Predicted Value: [1.00658161]



#### 1.4.2.2. Đánh giá kết quả ARIMA dự đoán tỷ giá EUR/USD

Sai số lớn nhất =  $1.03455412 - 1.00658161 = 0.02797251$ , tương đương 2.70% giá thị trường tại thời điểm sai số, mức này tương đối nhỏ, cho thấy mô hình ARIMA đã dự đoán khá sát với giá trị thực. Kết quả này chứng minh ARIMA phù hợp hơn khi dự đoán các cặp tỷ giá như EUR/USD, nơi dao động thường có tính ổn định hơn so với các tài sản có mức biến động cao như dầu hoặc vàng.

## 1.5. Sử dụng các mô hình để dự đoán giá dầu

### 1.5.1. Sử dụng SimpleRNN và LSTM để dự đoán giá dầu

```
stock = 'CL=F'
start_date = '2019-01-01'
end_date = '2024-12-31'
data = yf.download(stock, start=start_date, end=end_date)
data = data[['Close']]

scaler = MinMaxScaler(feature_range=(0, 1))
scaled_data = scaler.fit_transform(data)
```

#### 1.5.1.1. Khởi tạo và huấn luyện mô hình

```
def create_dataset(data, time_step=60):
    X, y = [], []
    for i in range(len(data) - time_step):
        X.append(data[i:i + time_step, 0])
        y.append(data[i + time_step, 0])
    return np.array(X), np.array(y)

time_step = 365
X, y = create_dataset(scaled_data, time_step)
X = X.reshape(X.shape[0], X.shape[1], 1)

train_size = int(len(X) * 0.8)
X_train, X_test = X[:train_size], X[train_size:]
y_train, y_test = y[:train_size], y[train_size:]

rnn_model = Sequential([
    SimpleRNN(50, activation='relu', return_sequences=True,
              input_shape=(time_step, 1)),
    Dropout(0.2),
    SimpleRNN(50, activation='relu'),
    Dropout(0.2),
    Dense(1)
])

rnn_model.compile(optimizer='adam', loss='mean_squared_error')
rnn_model.summary()

rnn_history = rnn_model.fit(X_train, y_train, epochs=50, batch_size=32,
                            validation_data=(X_test, y_test))

lstm_model = Sequential([
```

```

        LSTM(50, activation='relu', return_sequences=True, input_shape=(time_step,
1)),
        Dropout(0.2),
        LSTM(50, activation='relu'),
        Dropout(0.2),
        Dense(1)
    )

lstm_model.compile(optimizer='adam', loss='mean_squared_error')
lstm_model.summary()

lstm_history = lstm_model.fit(X_train, y_train, epochs=50, batch_size=32,
validation_data=(X_test, y_test))

rnn_predicted_prices = rnn_model.predict(X_test)
rnn_predicted_prices = scaler.inverse_transform(rnn_predicted_prices)

lstm_predicted_prices = lstm_model.predict(X_test)
lstm_predicted_prices = scaler.inverse_transform(lstm_predicted_prices)

actual_prices = scaler.inverse_transform(y_test.reshape(-1, 1))
test_dates = data.index[-len(actual_prices):]

```

Layer (type)	Output Shape	Param #
simple_rnn_10 (SimpleRNN)	(None, 365, 50)	2,600
dropout_20 (Dropout)	(None, 365, 50)	0
simple_rnn_11 (SimpleRNN)	(None, 50)	5,050
dropout_21 (Dropout)	(None, 50)	0
dense_10 (Dense)	(None, 1)	51

Total params: 7,701 (30.08 KB)  
Trainable params: 7,701 (30.08 KB)  
Non-trainable params: 0 (0.00 B)

Epoch 1/50  
**29/29** ————— 2s 53ms/step - loss: 0.1344 - val\_loss: 1.5745e-04  
Epoch 2/50  
**29/29** ————— 1s 49ms/step - loss: 0.0267 - val\_loss: 0.0037  
Epoch 3/50  
**29/29** ————— 1s 48ms/step - loss: 0.0198 - val\_loss: 0.0065  
Epoch 4/50

```

29/29 ━━━━━━━━━━ 1s 49ms/step - loss: 0.0171 - val_loss: 2.6593e-04
Epoch 5/50
29/29 ━━━━━━━━━━ 1s 49ms/step - loss: 0.0146 - val_loss: 1.3160e-04
Epoch 6/50
29/29 ━━━━━━━━━━ 1s 50ms/step - loss: 0.0112 - val_loss: 1.1482e-04
Epoch 7/50
29/29 ━━━━━━━━━━ 1s 48ms/step - loss: 0.0110 - val_loss: 1.3253e-04
Epoch 8/50
29/29 ━━━━━━━━━━ 1s 50ms/step - loss: 0.0109 - val_loss: 0.0049
Epoch 9/50
29/29 ━━━━━━━━━━ 1s 50ms/step - loss: 0.0100 - val_loss: 0.0050
Epoch 10/50
29/29 ━━━━━━━━━━ 1s 49ms/step - loss: 0.0092 - val_loss: 0.0019
Epoch 11/50
29/29 ━━━━━━━━━━ 1s 49ms/step - loss: 0.0078 - val_loss: 0.0014
Epoch 12/50
29/29 ━━━━━━━━━━ 1s 49ms/step - loss: 0.0075 - val_loss: 0.0030
Epoch 13/50
...
Epoch 49/50
29/29 ━━━━━━━━━━ 1s 49ms/step - loss: 0.0036 - val_loss: 2.3917e-04
Epoch 50/50
29/29 ━━━━━━━━━━ 1s 49ms/step - loss: 0.0034 - val_loss: 0.0015

```

Layer (type)	Output Shape	Param #
lstm_10 (LSTM)	(None, 365, 50)	10,400
dropout_22 (Dropout)	(None, 365, 50)	0
lstm_11 (LSTM)	(None, 50)	20,200
dropout_23 (Dropout)	(None, 50)	0
dense_11 (Dense)	(None, 1)	51

Total params: 30,651 (119.73 KB)

Trainable params: 30,651 (119.73 KB)

Non-trainable params: 0 (0.00 B)

```

Epoch 1/50
29/29 ━━━━━━━━━━ 4s 116ms/step - loss: 0.2632 - val_loss: 0.0127
Epoch 2/50
29/29 ━━━━━━━━━━ 3s 118ms/step - loss: 0.0148 - val_loss: 7.1355e-04
Epoch 3/50
29/29 ━━━━━━━━━━ 3s 114ms/step - loss: 0.0096 - val_loss: 0.0011

```

Epoch 4/50

**29/29** ————— 3s 113ms/step - loss: 0.0093 - val\_loss: 5.2541e-04

Epoch 5/50

**29/29** ————— 3s 115ms/step - loss: 0.0080 - val\_loss: 6.7517e-04

Epoch 6/50

**29/29** ————— 3s 113ms/step - loss: 0.0094 - val\_loss: 5.0084e-04

Epoch 7/50

**29/29** ————— 3s 114ms/step - loss: 0.0079 - val\_loss: 3.0571e-04

Epoch 8/50

**29/29** ————— 3s 115ms/step - loss: 0.0084 - val\_loss: 7.4966e-04

Epoch 9/50

**29/29** ————— 3s 113ms/step - loss: 0.0074 - val\_loss: 6.5767e-04

Epoch 10/50

**29/29** ————— 3s 115ms/step - loss: 0.0074 - val\_loss: 5.4372e-04

Epoch 11/50

**29/29** ————— 3s 116ms/step - loss: 0.0071 - val\_loss: 7.2708e-04

Epoch 12/50

**29/29** ————— 3s 113ms/step - loss: 0.0075 - val\_loss: 7.1075e-04

Epoch 13/50

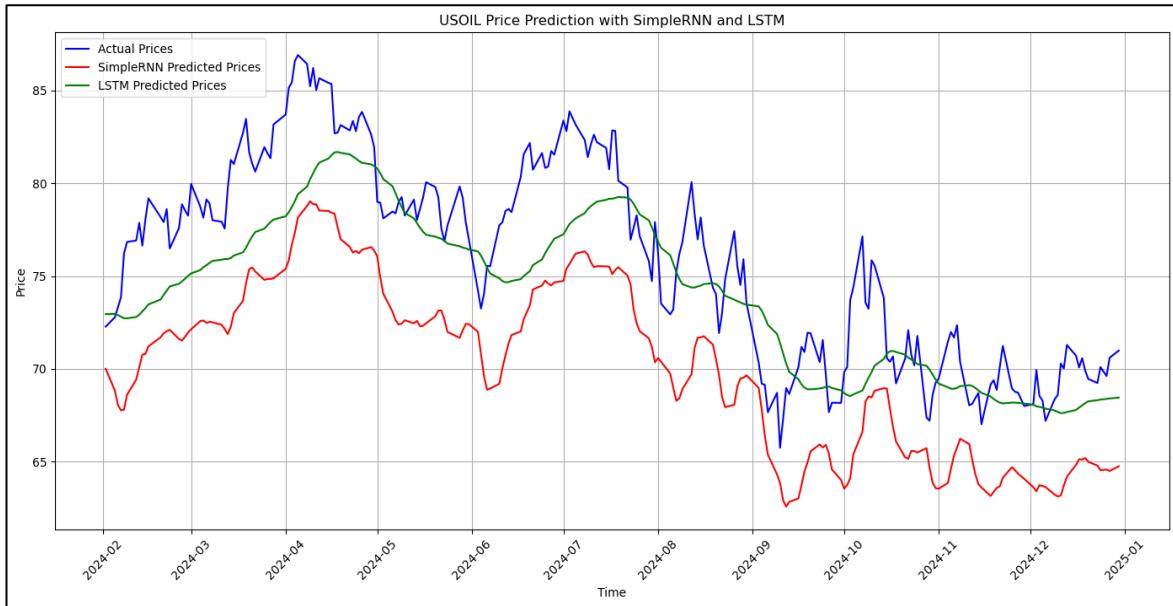
...

Epoch 50/50

**29/29** ————— 3s 113ms/step - loss: 0.0039 - val\_loss: 3.9859e-04

**8/8** ————— 0s 25ms/step

**8/8** ————— 0s 37ms/step

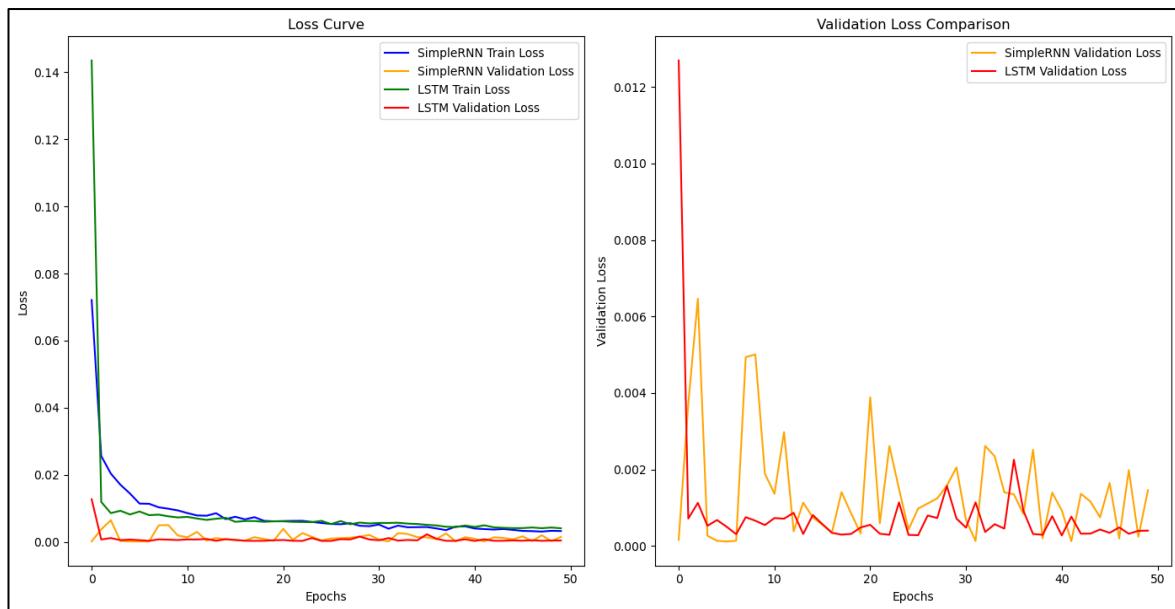


SimpleRNN MAE: 5.883242811178012

SimpleRNN MSE: 37.87482194714326

SimpleRNN RMSE: 6.154252346722814

SimpleRNN  $R^2$ : -0.32136434589310947  
 SimpleRNN MAPE: 7.686473527296571%  
 LSTM MAE: 2.63374738401721  
 LSTM MSE: 10.374199181108569  
 LSTM RMSE: 3.2209003680816592  
 LSTM  $R^2$ : 0.6380683469815149  
 LSTM MAPE: 3.3950928764925066%



### 1.5.1.2. Đánh giá kết quả SimpleRNN và LSTM dự đoán giá dầu

SimpleRNN:

MAE (5.88): Sai số trung bình mỗi lần dự đoán là 5.88 USD, khá lớn khi so sánh với dao động giá dầu (khoảng 60–90 USD/thùng trong các năm gần đây).

MSE (37.87): Sai số bình phương trung bình cho thấy độ lệch lớn trong các lần dự đoán.

RMSE (6.15): Sai số chuẩn lớn, gần tương đương 6 USD, cho thấy SimpleRNN không phù hợp với dữ liệu giá dầu trong trường hợp này.

$R^2$  (-0.32): Kết quả âm, nghĩa là mô hình dự đoán tệ hơn so với giá trị trung bình của tập dữ liệu.

MAPE (7.69%): Sai số tương đối cao, cho thấy SimpleRNN không đạt hiệu quả khi áp dụng vào dự đoán giá dầu.

LSTM:

MAE (2.63): Sai số trung bình chỉ khoảng 2.63 USD, cải thiện rõ rệt so với SimpleRNN.

MSE (10.37): Sai số bình phương trung bình nhỏ hơn đáng kể so với SimpleRNN.

RMSE (3.22): Sai số chuẩn chỉ khoảng 3.22 USD, phù hợp hơn với dao động giá dầu.

R<sup>2</sup> (0.64): LSTM giải thích được khoảng 64% biến động của giá dầu thực tế, khá tốt trong trường hợp dữ liệu phức tạp.

MAPE (3.39%): Sai số tương đối nhỏ, chỉ 3.39%, phù hợp để dự đoán giá dầu.

Đối chiếu với giá dầu thực dao động mạnh trong khoảng 60–90 USD/thùng. Sai số trung bình của LSTM (2.63 USD) là chấp nhận được, trong khi SimpleRNN có sai số lớn hơn (5.88 USD), không thực sự phù hợp với dữ liệu. MAPE của LSTM thấp (3.39%) cho thấy mô hình này dự đoán khá tốt ngay cả khi giá dầu biến động mạnh, trong khi MAPE của SimpleRNN (7.69%) cho thấy mức sai lệch lớn hơn và không phù hợp với dữ liệu giá dầu.

## 1.5.2. Sử dụng ARIMA dự đoán giá dầu

### 1.5.2.1. Khởi tạo mô hình

```
stock_symbol = 'CL=F'

data = yf.download(stock_symbol, start='2024-12-01',
end=datetime.now()]['Close']

data_forecast = yf.download(stock_symbol, start='2024-12-12',
end=datetime.now().strftime('%Y-%m-%d'))['Close']

data_combined = pd.concat([data, data_forecast])

scaler = MinMaxScaler()

data_scaled = scaler.fit_transform(data_combined.values.reshape(-1, 1))

data_train = data_scaled[:len(data)]
data_test = data_scaled[len(data):]

order = (60, 2, 10)
model = ARIMA(data_train, order=order)
model_fit = model.fit()

future_steps = len(data_test)
forecast_scaled = model_fit.forecast(steps=future_steps)

forecast = scaler.inverse_transform(forecast_scaled.reshape(-1, 1))
data_test_actual = scaler.inverse_transform(data_test.reshape(-1, 1))
```

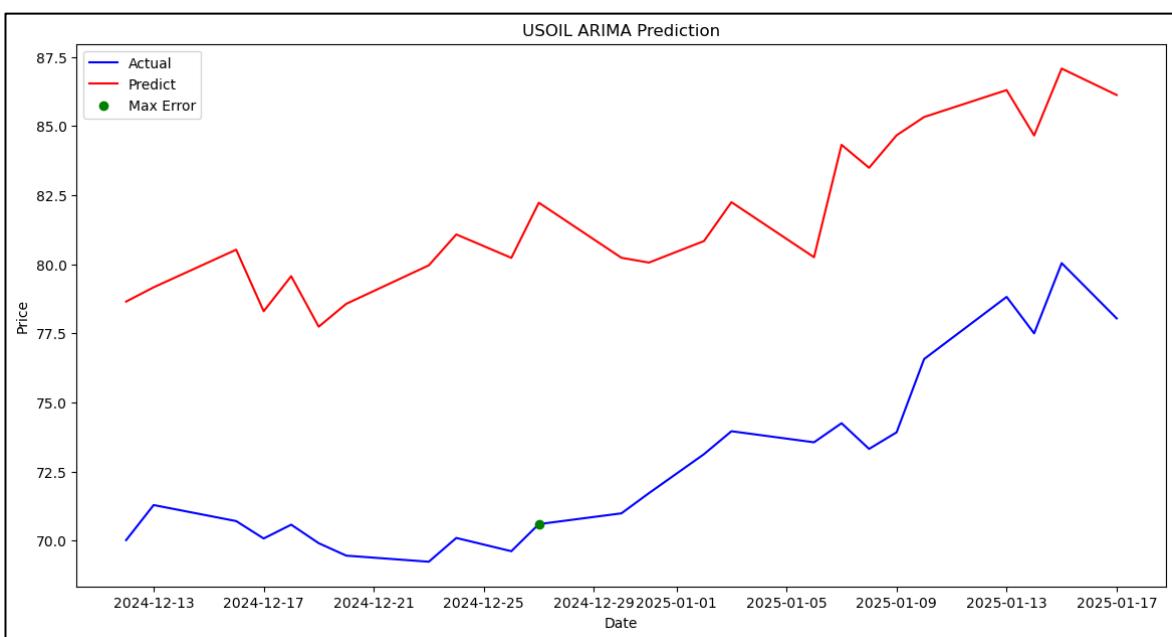
```

errors = np.abs(data_test_actual - forecast)
max_error_index = np.argmax(errors)
max_error_actual = data_test_actual[max_error_index]
max_error_predicted = forecast[max_error_index]

```

[\*\*\*\*\*100%\*\*\*\*\*] 1 of 1 completed  
[\*\*\*\*\*100%\*\*\*\*\*] 1 of 1 completed

Max Error Index: 10  
Max Error Actual Value: [70.59999847]  
Max Error Predicted Value: [82.22530436]



### 1.5.2.2. Đánh giá kết quả ARIMA dự đoán giá dầu

Sai số lớn nhất =  $82.2253 - 70.6 = 11.6253$ , tương đương 16,46% giá thị trường của dầu tại thời điểm sai số. Mức sai số này khá cao, cho thấy mô hình ARIMA không theo kịp biến động mạnh của giá dầu. Điều này còn phản ánh tính không tuyến tính và sự biến động bất thường trong dữ liệu giá dầu, dẫn đến việc ARIMA gặp khó khăn trong việc mô phỏng.

## 1.6. Sử dụng các mô hình để dự đoán giá vàng

### 1.6.1. Sử dụng SimpleRNN và LSTM để dự đoán giá vàng

```

stock = 'GC=F'
start_date = '2019-01-01'
end_date = '2024-12-31'

```

```
data = yf.download(stock, start=start_date, end=end_date)
data = data[['Close']]
```

```
scaler = MinMaxScaler(feature_range=(0, 1))
scaled_data = scaler.fit_transform(data)
```

### 1.6.1.1. Khởi tạo và huấn luyện mô hình

```
def create_dataset(data, time_step=60):
    X, y = [], []
    for i in range(len(data) - time_step):
        X.append(data[i:i + time_step, 0])
        y.append(data[i + time_step, 0])
    return np.array(X), np.array(y)

time_step = 365
X, y = create_dataset(scaled_data, time_step)
X = X.reshape(X.shape[0], X.shape[1], 1)

train_size = int(len(X) * 0.8)
X_train, X_test = X[:train_size], X[train_size:]
y_train, y_test = y[:train_size], y[train_size:]

rnn_model = Sequential([
    SimpleRNN(50, activation='relu', return_sequences=True,
    input_shape=(time_step, 1)),
    Dropout(0.2),
    SimpleRNN(50, activation='relu'),
    Dropout(0.2),
    Dense(1)
])

rnn_model.compile(optimizer='adam', loss='mean_squared_error')
rnn_model.summary()

rnn_history = rnn_model.fit(X_train, y_train, epochs=50, batch_size=32,
validation_data=(X_test, y_test))

lstm_model = Sequential([
    LSTM(50, activation='relu', return_sequences=True, input_shape=(time_step,
1)),
    Dropout(0.2),
    LSTM(50, activation='relu'),
    Dropout(0.2),
    Dense(1)
```

```

] )

lstm_model.compile(optimizer='adam', loss='mean_squared_error')
lstm_model.summary()

lstm_history = lstm_model.fit(X_train, y_train, epochs=50, batch_size=32,
validation_data=(X_test, y_test))

rnn_predicted_prices = rnn_model.predict(X_test)
rnn_predicted_prices = scaler.inverse_transform(rnn_predicted_prices)

lstm_predicted_prices = lstm_model.predict(X_test)
lstm_predicted_prices = scaler.inverse_transform(lstm_predicted_prices)

actual_prices = scaler.inverse_transform(y_test.reshape(-1, 1))
test_dates = data.index[-len(actual_prices):]

```

Layer (type)	Output Shape	Param #
simple_rnn_2 (SimpleRNN)	(None, 365, 50)	2,600
dropout_4 (Dropout)	(None, 365, 50)	0
simple_rnn_3 (SimpleRNN)	(None, 50)	5,050
dropout_5 (Dropout)	(None, 50)	0
dense_2 (Dense)	(None, 1)	51

Total params: 7,701 (30.08 KB)  
Trainable params: 7,701 (30.08 KB)  
Non-trainable params: 0 (0.00 B)

Epoch 1/50  
**29/29** ————— 2s 52ms/step - loss: 0.0189 - val\_loss: 0.0078  
Epoch 2/50  
**29/29** ————— 1s 47ms/step - loss: 0.0065 - val\_loss: 0.0146  
Epoch 3/50  
**29/29** ————— 1s 48ms/step - loss: 0.0052 - val\_loss: 0.0056  
Epoch 4/50  
**29/29** ————— 1s 49ms/step - loss: 0.0042 - val\_loss: 0.0088  
Epoch 5/50  
**29/29** ————— 1s 47ms/step - loss: 0.0037 - val\_loss: 0.0050  
Epoch 6/50  
**29/29** ————— 1s 48ms/step - loss: 0.0029 - val\_loss: 0.0051  
Epoch 7/50

```

29/29 ━━━━━━━━━━ 1s 47ms/step - loss: 0.0029 - val_loss: 0.0069
Epoch 8/50
29/29 ━━━━━━━━━━ 1s 48ms/step - loss: 0.0024 - val_loss: 0.0088
Epoch 9/50
29/29 ━━━━━━━━━━ 1s 48ms/step - loss: 0.0027 - val_loss: 0.0060
Epoch 10/50
29/29 ━━━━━━━━━━ 1s 49ms/step - loss: 0.0024 - val_loss: 0.0079
Epoch 11/50
29/29 ━━━━━━━━━━ 1s 48ms/step - loss: 0.0021 - val_loss: 0.0018
Epoch 12/50
29/29 ━━━━━━━━━━ 1s 48ms/step - loss: 0.0020 - val_loss: 0.0079
Epoch 13/50
...
Epoch 49/50
29/29 ━━━━━━━━━━ 1s 48ms/step - loss: 5.8605e-04 - val_loss: 0.0025
Epoch 50/50
29/29 ━━━━━━━━━━ 1s 49ms/step - loss: 5.8328e-04 - val_loss: 9.8688e-04

```

Layer (type)	Output Shape	Param #
lstm_2 (LSTM)	(None, 365, 50)	10,400
dropout_6 (Dropout)	(None, 365, 50)	0
lstm_3 (LSTM)	(None, 50)	20,200
dropout_7 (Dropout)	(None, 50)	0
dense_3 (Dense)	(None, 1)	51

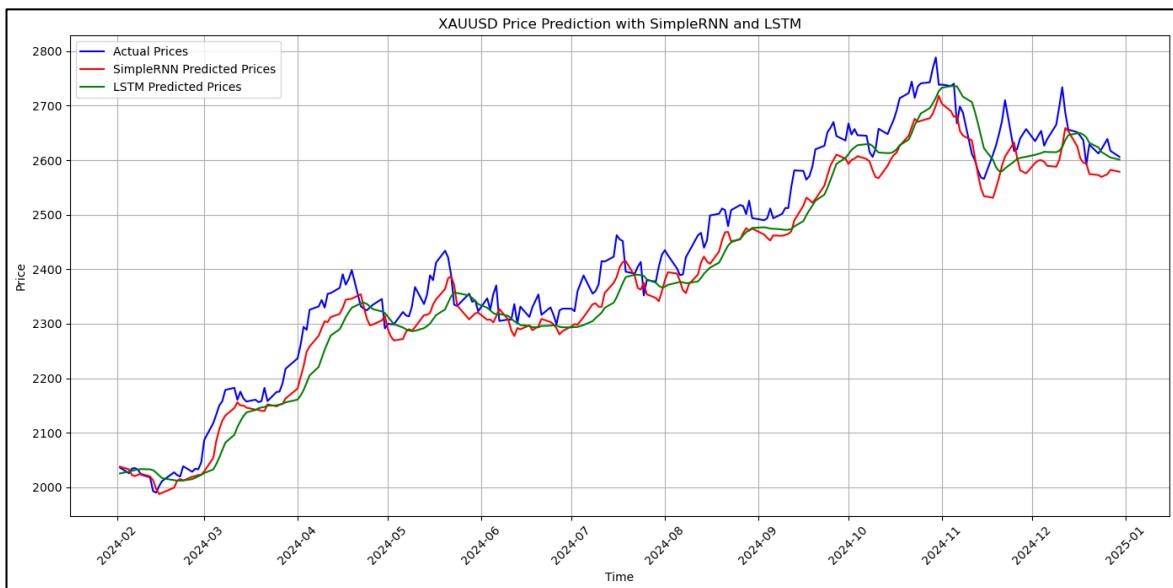
**Total params:** 30,651 (119.73 KB)  
**Trainable params:** 30,651 (119.73 KB)  
**Non-trainable params:** 0 (0.00 B)

```

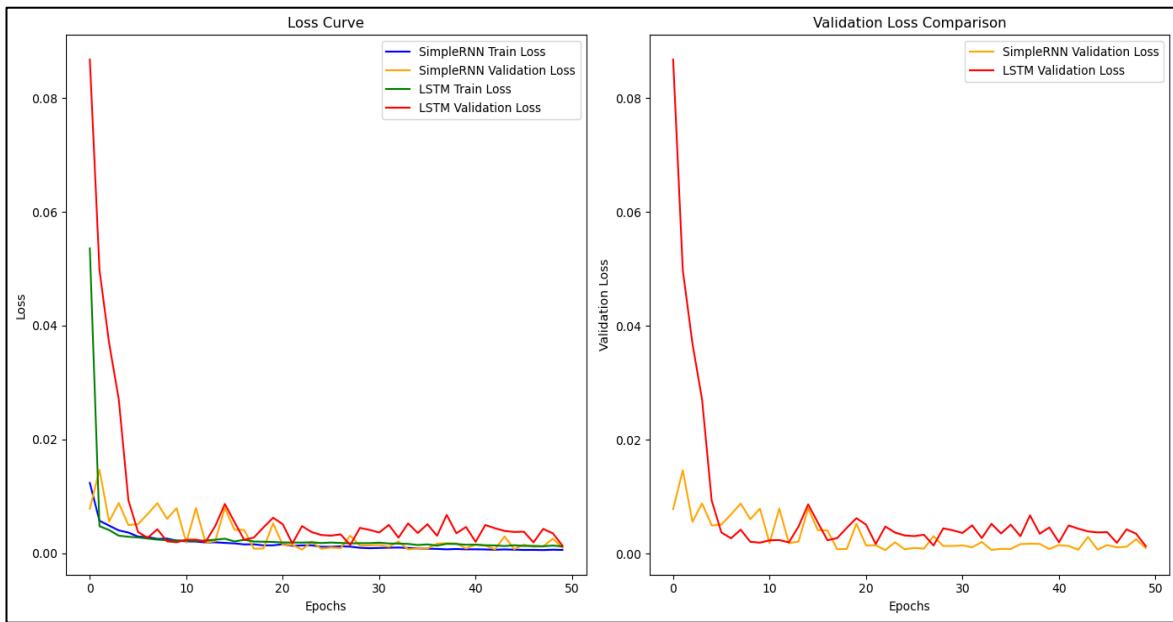
Epoch 1/50
29/29 ━━━━━━━━━━ 4s 117ms/step - loss: 0.0985 - val_loss: 0.0868
Epoch 2/50
29/29 ━━━━━━━━━━ 3s 114ms/step - loss: 0.0057 - val_loss: 0.0498
Epoch 3/50
29/29 ━━━━━━━━━━ 3s 115ms/step - loss: 0.0040 - val_loss: 0.0369
Epoch 4/50
29/29 ━━━━━━━━━━ 3s 113ms/step - loss: 0.0032 - val_loss: 0.0270
Epoch 5/50
29/29 ━━━━━━━━━━ 3s 113ms/step - loss: 0.0029 - val_loss: 0.0093
Epoch 6/50
29/29 ━━━━━━━━━━ 3s 114ms/step - loss: 0.0030 - val_loss: 0.0037

```

Epoch 7/50  
**29/29** ————— 3s 113ms/step - loss: 0.0027 - val\_loss: 0.0027  
 Epoch 8/50  
**29/29** ————— 3s 113ms/step - loss: 0.0023 - val\_loss: 0.0042  
 Epoch 9/50  
**29/29** ————— 3s 114ms/step - loss: 0.0026 - val\_loss: 0.0021  
 Epoch 10/50  
**29/29** ————— 3s 115ms/step - loss: 0.0022 - val\_loss: 0.0019  
 Epoch 11/50  
**29/29** ————— 4s 130ms/step - loss: 0.0024 - val\_loss: 0.0023  
 Epoch 12/50  
**29/29** ————— 3s 115ms/step - loss: 0.0021 - val\_loss: 0.0024  
 Epoch 13/50  
 ...  
**8/8** ————— 0s 20ms/step  
**8/8** ————— 0s 37ms/step



SimpleRNN MAE: 41.0633640872339  
 SimpleRNN MSE: 2277.6961246453648  
 SimpleRNN RMSE: 47.725214767933366  
 SimpleRNN R<sup>2</sup>: 0.9444655655279066  
 SimpleRNN MAPE: 1.661509631786148%  
 LSTM MAE: 44.51820713359716  
 LSTM MSE: 3048.7264748049097  
 LSTM RMSE: 55.215273926739776  
 LSTM R<sup>2</sup>: 0.9256664228355962  
 LSTM MAPE: 1.8233967205127286%



### 1.6.1.2. Đánh giá kết quả SimpleRNN và LSTM dự đoán giá vàng

SimpleRNN:

MAE (41.06): Trung bình mỗi lần dự đoán lệch khoảng 41.06 USD so với giá thực tế.

MSE (2277.70): Mức độ sai lệch bình phương trung bình ở mức vừa phải.

RMSE (47.73): Sai số chuẩn khoảng 47.73 USD, cho thấy sai số không quá lớn so với dao động giá thực tế của vàng.

$R^2$  (0.944): Mô hình giải thích được khoảng 94.4% biến động trong dữ liệu thực tế, đây là một kết quả rất tốt.

MAPE (1.66%): Sai số tương đối nhỏ, chỉ khoảng 1.66%, cho thấy dự đoán của SimpleRNN khá chính xác.

LSTM:

MAE (44.52): Sai số trung bình mỗi lần dự đoán cao hơn một chút so với SimpleRNN (+3.46 USD).

MSE (3048.73): Sai số bình phương lớn hơn so với SimpleRNN, cho thấy LSTM có độ biến thiên dự đoán lớn hơn.

RMSE (55.22): Sai số chuẩn lớn hơn khoảng 7.49 USD so với SimpleRNN.

$R^2$  (0.926): LSTM giải thích được khoảng 92.6% biến động trong dữ liệu thực tế, thấp hơn SimpleRNN.

MAPE (1.82%): Sai số tương đối lớn hơn SimpleRNN một chút, khoảng 1.82%, vẫn là kết quả tốt.

Đối chiếu với giá vàng thực cho thấy dao động trong khoảng từ 1500 USD đến 2000 USD/oz trong các giai đoạn gần đây (2019–2024). Sai số trung bình MAE khoảng 40–44 USD chỉ chiếm xấp xỉ 2% giá trị của vàng, cho thấy mô hình có khả năng dự đoán hợp lý. Sai số chuẩn 47–55 USD nằm trong giới hạn dao động thường ngày của giá vàng, đặc biệt trong các giai đoạn biến động lớn, nên mức độ chấp nhận được. MAPE thấp (<2%) cho thấy cả SimpleRNN và LSTM đều hoạt động hiệu quả trong bối cảnh giá vàng có xu hướng ổn định.

So sánh hai mô hình SimpleRNN cho kết quả tốt hơn LSTM về các chỉ số sai số (MAE, MSE, RMSE, R<sup>2</sup> và MAPE). Điều này có thể do tập dữ liệu tương đối ổn định và không quá phức tạp, nên SimpleRNN dễ dàng học được các quy luật từ dữ liệu hơn. LSTM thường hiệu quả hơn trên dữ liệu phức tạp hoặc có quan hệ dài hạn, nhưng trong trường hợp này, độ phức tạp của LSTM không phát huy được lợi thế so với SimpleRNN.

## 1.6.2. Sử dụng ARIMA để dự đoán giá vàng

### 1.6.2.1. Khởi tạo mô hình

```
stock_symbol = 'GC=F'
data = yf.download(stock_symbol, start='2024-12-01',
end=datetime.now()['Close']
data_forecast = yf.download(stock_symbol, start='2024-12-12',
end=datetime.now().strftime('%Y-%m-%d'))['Close']
data_combined = pd.concat([data, data_forecast])

scaler = MinMaxScaler()
data_scaled = scaler.fit_transform(data_combined.values.reshape(-1, 1))
data_train = data_scaled[:len(data)]
data_test = data_scaled[len(data):]

order = (30, 2, 5)
model = ARIMA(data_train, order=order)
model_fit = model.fit()

future_steps = len(data_test)
forecast_scaled = model_fit.forecast(steps=future_steps)
forecast = scaler.inverse_transform(forecast_scaled.reshape(-1, 1))
data_test_actual = scaler.inverse_transform(data_test.reshape(-1, 1))
```

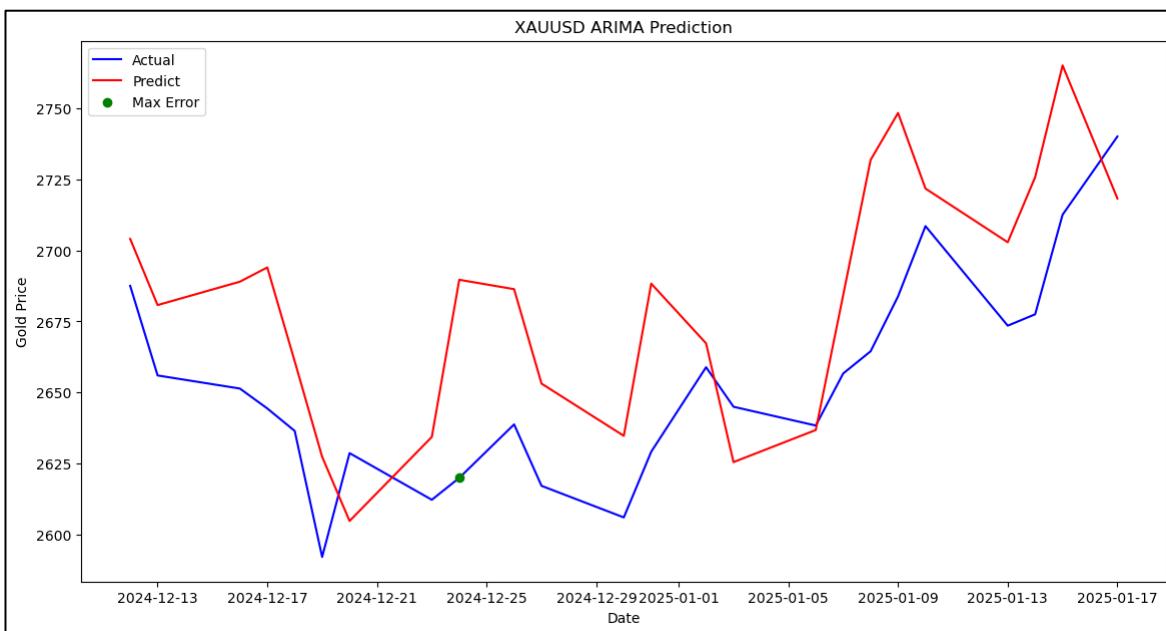
```

errors = np.abs(data_test_actual - forecast)
max_error_index = np.argmax(errors)
max_error_actual = data_test_actual[max_error_index]
max_error_predicted = forecast[max_error_index]

```

[\*\*\*\*\*100%\*\*\*\*\*] 1 of 1 completed  
[\*\*\*\*\*100%\*\*\*\*\*] 1 of 1 completed

Max Error Index: 8  
Max Error Actual Value: [2620.]  
Max Error Predicted Value: [2689.63968853]



### 1.6.2.2. Đánh giá kết quả ARIMA dự đoán giá vàng

Sai số lớn nhất =  $2689.63968853 - 2620 = 69.63968853$ , tương đương 2.66% giá trị thị trường, cho thấy mức độ sai lệch của mô hình ARIMA trong việc dự đoán giá vàng tại thời điểm biến động mạnh. Với mức giá cao của vàng, sai số tuyệt đối 69.64 tuy có vẻ lớn nhưng sai số phần trăm lại tương đối nhỏ, cho thấy mô hình ARIMA vẫn có khả năng dự đoán xu hướng tương đối tốt trong bối cảnh chung. Tuy nhiên, giá vàng thường nhạy cảm với các yếu tố toàn cầu như lãi suất, lạm phát, và các sự kiện kinh tế bất ngờ, khiến việc sử dụng mô hình ARIMA truyền thống bị hạn chế trong việc nắm bắt các đợt biến động mạnh. Do đó, tác giả khuyến nghị không nên sử dụng ARIMA để dự đoán giá vàng.

## **1.7. So sánh hiệu quả mô hình và kết luận**

SimpleRNN hoạt động tốt hơn so với LSTM trong hầu hết các trường hợp: Sai số (MAE, MSE, RMSE) thấp hơn.  $R^2$  cao hơn, thể hiện khả năng giải thích biến động tốt hơn. MAPE nhỏ hơn, phù hợp cho các dự đoán cần độ chính xác cao. Đặc biệt, đối với các dữ liệu ít biến động (tỷ giá) hoặc có xu hướng ổn định (giá dầu), SimpleRNN thể hiện rõ sự vượt trội. LSTM chỉ vượt SimpleRNN trong trường hợp dữ liệu có xu hướng phi tuyến tính phức tạp hơn, hoặc yêu cầu khả năng nắm bắt dài hạn. Qua đây bộc lộ rõ nhược điểm của mô hình LSTM: Đối với dữ liệu rất dài, mô hình dễ bị quá tải nếu không tối ưu. Do đó, tác giả khuyến nghị đối với dữ liệu là các cặp tỷ giá biến động chậm thì nên sử dụng SimpleRNN, ngược lại với các mã có biến động mạnh như giá vàng và giá dầu thì có thể sử dụng LSTM. Tuy nhiên cần điều chỉnh các tham số đầu vào của dữ liệu khi khởi tạo để mô hình đạt hiệu suất tốt nhất.

ARIMA (AutoRegressive Integrated Moving Average) là một mô hình thống kê sử dụng các quan hệ tuyến tính trong dữ liệu thời gian (Time Series) để dự đoán giá trị tương lai. ARIMA không phải là mô hình học sâu, ARIMA không học từ các mẫu phức tạp mà chỉ mô hình hóa dựa trên các quan hệ tuyến tính trong dữ liệu. Vì vậy tác giả khuyến nghị ARIMA chỉ thích hợp cho dữ liệu tĩnh hay các cặp tiền tệ có xu hướng dao động tương đối ổn định trong điều kiện thị trường bình thường. ARIMA có thể dự đoán khá chính xác EUR/USD, GBP/USD. Đối với cặp tiền tệ có dữ liệu biến động bất thường như USD/JPY, hay giá dầu thô USOIL và giá vàng XAU/USD, ARIMA bộc lộ rõ điểm yếu hoạt động kém với dữ liệu phi tuyến tính.

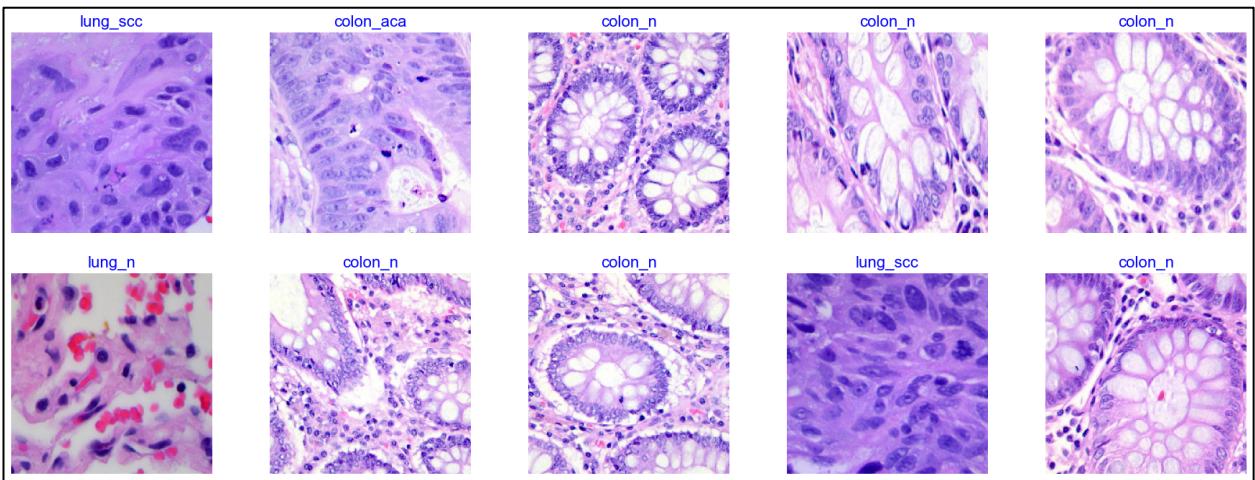
## PHẦN II: MẠNG THẦN KINH TÍCH CHẬP CNN

### 2.1. Giới thiệu bộ dữ liệu

Bộ dữ liệu mô bệnh ung thư phổi và ung thư đại tràng chứa 1.250 hình ảnh, bao gồm 750 hình ảnh về mô phổi và 500 hình ảnh về mô đại tràng, đại diện cho các loại tế bào bình thường hoặc bất thường (ung thư) từ các mẫu mô phổi và đại tràng. Bộ dữ liệu đã được sử dụng Data Augmentor để tăng số lượng mỗi lớp lên 5.000 hình ảnh, tổng cộng bộ dữ liệu có 25.000 hình ảnh định dạng jpeg, kích thước 300x300.

STT	Ký hiệu	Chú thích
1	lung_aca	Ung thư biểu mô tuyến phổi (Lung adenocarcinoma)
2	lung_n	Mô lành tính ở phổi (Lung benign tissue)
3	lung_scc	Ung thư biểu mô tế bào vảy phổi (Lung squamous cell carcinoma)
4	colon_n	Mô lành tính đại tràng (Colon benign tissue)
5	colon_aca	Ung thư biểu mô tuyến đại tràng (Colon adenocarcinoma)

Việc sử dụng mạng thần kinh tích chập (Convolutional Neural Network) để phân loại ảnh mô ung thư phổi và đại tràng là một nhiệm vụ quan trọng trong y học nhằm hỗ trợ các bác sĩ chẩn đoán nhanh chóng và chính xác tình trạng tế bào ung thư. Điều này không chỉ giúp phát hiện ung thư ở giai đoạn sớm, cải thiện cơ hội sống sót của bệnh nhân, mà còn cung cấp cơ sở để xây dựng các phác đồ điều trị cá nhân hóa phù hợp. Sử dụng các mô hình học sâu như MobileNet, AlexNet và ResNet cho phép tự động hóa quá trình phân tích ảnh y khoa, giảm tải công việc cho bác sĩ và tăng độ chính xác trong phân loại. Bên cạnh đó, việc phân tích từ bộ dữ liệu lớn và đa dạng như 25.000 ảnh giúp nâng cao chất lượng nghiên cứu, tìm hiểu sâu hơn về đặc điểm sinh học của ung thư, đồng thời mở ra tiềm năng ứng dụng cho nhiều loại bệnh khác. Đây là bước tiến quan trọng để cải thiện hệ thống chăm sóc sức khỏe hiện đại, đặc biệt ở các khu vực thiếu nhân lực y tế chuyên môn.



## 2.2. Sử dụng MobileNet để phân loại

### 2.2.1. Nhập các thư viện và tiền xử lý dữ liệu

```

import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import backend as K
from tensorflow.keras.layers import Dense, Activation, Dropout, Conv2D,
MaxPooling2D, BatchNormalization, Flatten, Add, AveragePooling2D, Input,
ZeroPadding2D
from tensorflow.keras.layers import Activation, BatchNormalization, Add,
Reshape, DepthwiseConv2D, GlobalAveragePooling2D
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.metrics import categorical_crossentropy
from tensorflow.keras import regularizers
from tensorflow.keras.initializers import glorot_uniform
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.models import Model, load_model, Sequential
import numpy as np
import pandas as pd
import shutil
import time
from tensorflow.keras import layers
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
from matplotlib.pyplot import imshow
import os
import seaborn as sns
sns.set_style('darkgrid')
from PIL import Image
from sklearn.metrics import confusion_matrix, classification_report
data_dir = '/Users/linhtuanpham/Downloads/lung_colon_image_set'

```

```

filepaths = []
labels = []
Fseries = pd.Series(filepaths, name='filepaths')
Lseries = pd.Series(labels, name='labels')
data_df = pd.concat([Fseries, Lseries], axis=1)
df = data_df.reset_index(drop=True)
print(df['labels'].value_counts())
labels
lung_aca      5000
lung_n        5000
lung_scc      5000
colon_n       5000
colon_aca     5000
Name: count, dtype: int64

```

Có 5 nhãn tương ứng với 5 ký hiệu đã liệt kê ở phần giới thiệu, mỗi nhãn chứa 5.000 hình ảnh. Nhiệm vụ được đặt ra là nhận diện và giảm thiểu sai số với những nhãn có chứa “aca”, “scc”.

```

train_split=.8
test_split=.1
dummy_split=test_split/(1-train_split)
train_df, dummy_df=train_test_split(df, train_size=train_split, shuffle=True,
random_state=123)
test_df, valid_df=train_test_split(dummy_df, train_size=dummy_split,
shuffle=True, random_state=123)
print ('train_df length: ', len(train_df), ' test_df length: ', len(test_df), '
valid_df length: ', len(valid_df))
train_df length: 20000  test_df length: 2500  valid_df length: 2500

```

Thực hiện chia bộ dữ liệu với tỷ lệ 80/10/10 với mục đích tương ứng là huấn luyện (train), kiểm thử (validation) và kiểm tra (test). Tập huấn luyện có 20.000 ảnh, tập kiểm thử có 2.500 ảnh, tập kiểm tra có 2.500 ảnh.

```

from tensorflow.keras.applications.inception_v3 import InceptionV3
from tensorflow.keras.layers import Input
from tensorflow.keras.applications import MobileNet
from tensorflow.keras.layers import Input, GlobalAveragePooling2D, Concatenate
mobilenet = MobileNet(weights='imagenet', include_top=False)

model = Model(inputs=input_layer, outputs=predictions)

```

[/var/folders/0w/n8j1dj1s13d5mf73xzjt1f980000gn/T/ipykernel\\_66328/2830154816.py:6](#)  
: UserWarning: Weights for input shape (224, 224) will be loaded as the default.  
mobilenet = MobileNet(weights='imagenet', include\_top=False)

Tải về mô hình MobileNet đã được huấn luyện sẵn trên tập dữ liệu ImageNet. Đây là một tập dữ liệu lớn được sử dụng phổ biến trong lĩnh vực thị giác máy tính (Computer Vision). Nó được thiết kế để hỗ trợ phát triển các thuật toán học sâu (Deep Learning) và học máy (Machine Learning), đặc biệt là trong các bài toán phân loại và nhận diện hình ảnh.

Chọn đầu vào của ảnh là 224x224 với 3 kênh màu, vì đây là kích thước đầu vào cố định của MobileNet. Việc này còn giúp mô hình tập trung vào các đặc trưng chính (features) thay vì các chi tiết không quan trọng để khai quát tốt hơn, đặc biệt là khi tập dữ liệu không quá lớn.

### 2.2.2. Khởi tạo và huấn luyện mô hình

```
for layer in mobilenet.layers:  
    layer.trainable = False  
input_layer = Input(shape=(224,224,3))  
x1 = mobilenet(input_layer)  
concatenated = Concatenate()([x1])  
flat = Flatten()(concatenated)  
dense = Dense(units=1024, activation='relu')(flat)  
dense = Dense(512, activation='relu')(dense)  
dense = Dense(units=256, activation='relu')(dense)  
predictions = Dense(num_classes, activation='softmax')(dense)  
  
model.compile(Adam(learning_rate=.001), loss='categorical_crossentropy',  
             metrics=['accuracy', Precision(), Recall(), AUC(), 'mae', 'mse',  
RootMeanSquaredError()])  
model.summary()
```

Layer (type)	Output Shape	Param #
conv2d_5 (Conv2D)	(None, 54, 54, 96)	34,944
max_pooling2d_3 (MaxPooling2D)	(None, 26, 26, 96)	0
conv2d_6 (Conv2D)	(None, 26, 26, 256)	614,656
max_pooling2d_4 (MaxPooling2D)	(None, 12, 12, 256)	0
conv2d_7 (Conv2D)	(None, 12, 12, 384)	885,120
conv2d_8 (Conv2D)	(None, 12, 12, 384)	1,327,488
conv2d_9 (Conv2D)	(None, 12, 12, 256)	884,992
max_pooling2d_5 (MaxPooling2D)	(None, 5, 5, 256)	0
flatten_1 (Flatten)	(None, 6400)	0
dense_3 (Dense)	(None, 4096)	26,218,496
dropout_2 (Dropout)	(None, 4096)	0
dense_4 (Dense)	(None, 4096)	16,781,312
dropout_3 (Dropout)	(None, 4096)	0
dense_5 (Dense)	(None, 5)	20,485

**Total params:** 55,267,525 (210.83 MB)

**Trainable params:** 52,038,661 (198.51 MB)

**Non-trainable params:** 3,228,864 (12.32 MB)

InputLayer: Dữ liệu đầu vào có kích thước (224, 224, 3).

MobileNet: Kích thước đầu ra (None, 7, 7, 1024).

Flatten: Chuyển đổi tensor 3D (7, 7, 1024) thành vector 1D (50176).

Các tầng Dense giảm dần kích thước đầu ra: 1024 → 512 → 256 → 5.

Hiển thị tổng số tham số của mỗi tầng:

Trainable params: Các tham số được tối ưu trong quá trình huấn luyện (hầu hết các tham số Dense).

Non-trainable params: Tham số không được cập nhật, thường là tham số từ mô hình pre-trained (MobileNet).

```
model_name='MobileNet'
from tensorflow.python.keras.callbacks import EarlyStopping
early_stop = EarlyStopping(monitor= 'val_loss', patience=5, mode = 'min'
,verbose=1)
def print_in_color(txt_msg,fore_tupple,back_tupple,):
    rf, gf, bf=fore_tupple
    rb, gb, bb=back_tupple
    msg='{0}' + txt_msg
```

```

        mat='\33[38;2;' + str(rf) + ';' + str(gf) + ';' + str(bf) + ';\48;2;' +
str(rb) + ';' +str(gb) + ';' + str(bb) +'m'
print(msg .format(mat), flush=True)
print('\33[0m', flush=True)
return

class LRA(keras.callbacks.Callback):
    reset = False
    count = 0
    stop_count = 0
    tepochs = 0
    best_weights = None

    def __init__(self, model, patience, stop_patience, threshold, factor, dwell,
model_name, freeze, initial_epoch):
        super(LRA, self).__init__()
        self._model = model
        self.patience = patience
        self.stop_patience = stop_patience
        self.threshold = threshold
        self.factor = factor
        self.dwell = dwell
        self.lr =
float(tf.keras.backend.get_value(model.optimizer.learning_rate))
        self.highest_tracc = 0.0
        self.lowest_vloss = np.inf
        self.initial_epoch = initial_epoch
        LRA.best_weights = model.get_weights()

        msgs = (f' Starting training using base model {model_name} with weights
frozen to imagenet weights initializing LRA callback'
        print_in_color(msgs, (244, 252, 3), (55,65,80))

@property
def model(self):
    return self._model
@model.setter
def model(self, value):
    self._model = value
def on_epoch_begin(self, epoch, logs=None):
    self.now = time.time()
def on_epoch_end(self, epoch, logs=None):
    later = time.time()
    duration = later - self.now

```

```

if epoch == self.initial_epoch or LRA.reset:
    LRA.reset = False
    msg = '{0:^8s}{1:^10s}{2:^9s}{3:^9s}{4:^9s}{5:^9s}{6:^9s}'.format(
        'Epoch', 'Loss', 'Acc', 'V_loss', 'V_acc', 'LR', 'Next LR')
    print_in_color(msg, (244,252,3), (55,65,80))
    lr = float(tf.keras.backend.get_value(self.model.optimizer.learning_rate))
    current_lr = lr
    v_loss = logs.get('val_loss')
    acc = logs.get('accuracy')
    v_acc = logs.get('val_accuracy')
    loss = logs.get('loss')

if acc < self.threshold:
    if acc > self.highest_tracc:
        self.highest_tracc = acc
        LRA.best_weights = self.model.get_weights()
        self.count = 0
        self.stop_count = 0
    if v_loss < self.lowest_vloss:
        self.lowest_vloss = v_loss
        self.lr = lr
    else:
        if self.count >= self.patience - 1:
            self.lr = lr * self.factor
            tf.keras.backend.set_value(self.model.optimizer.learning_rate,
self.lr)
            self.count = 0
            self.stop_count += 1
        if self.dwell:
            self.model.set_weights(LRA.best_weights)
        else:
            self.count += 1
    else:
        if v_loss < self.lowest_vloss:
            self.lowest_vloss = v_loss
            LRA.best_weights = self.model.get_weights()
            self.count = 0
            self.stop_count = 0
            self.lr = lr
    else:
        if self.count >= self.patience - 1:
            self.lr = self.lr * self.factor
            self.stop_count += 1
            self.count = 0

```

```

        tf.keras.backend.set_value(self.model.optimizer.learning_rate,
self.lr)
        if self.dwell:
            self.model.set_weights(LRA.best_weights)
        else:
            self.count += 1
msg =
f'{str(epoch+1):^3s}/{str(LRA.tePOCHS):4s}{loss:^9.3f}{acc:^9.3f}{v_loss:^9.3f}
{v_acc:^9.3f}{current_lr:^9.5f}{self.lr:^9.5f}'

epochs =15
patience=9
stop_patience =3
threshold=.9
factor=.5
dwell=True
freeze=False
callbacks=[LRA(model=model,patience=patience,stop_patience=stop_patience,
threshold=threshold, factor=factor,dwell=dwell,model_name= model_name,
freeze=freeze, initial_epoch=0 )]
LRA.tePOCHS=epochs
history=model.fit(x=train_gen, epochs=epochs, callbacks=callbacks, verbose=1,
validation_data=valid_gen, validation_steps=None, shuffle=False,
initial_epoch=0)

```

Quy trình hoạt động của mô hình này như sau

Khởi tạo callback:

Lưu trạng thái ban đầu của mô hình, learning rate, trọng số tốt nhất (best\_weights).

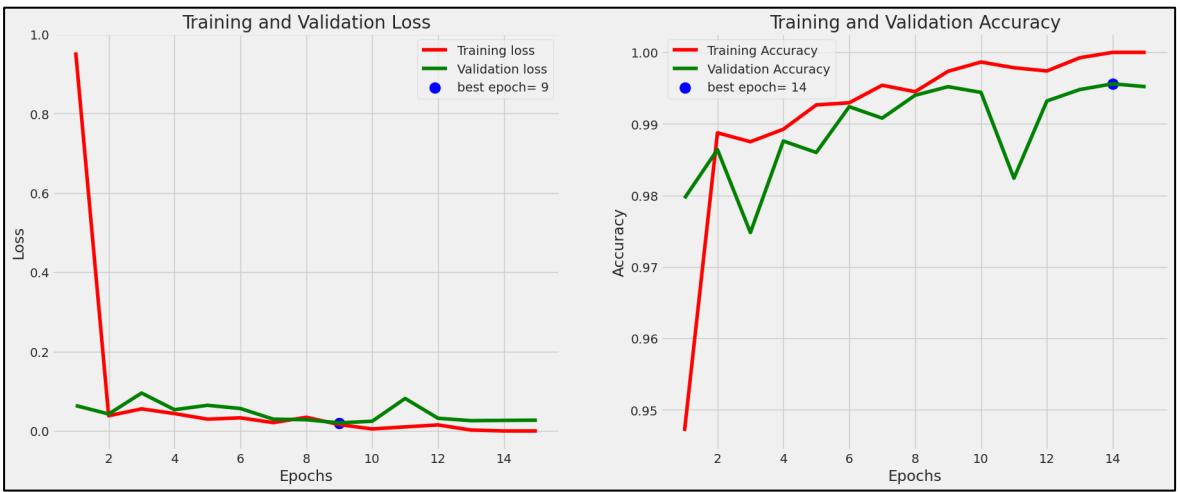
In thông báo bắt đầu huấn luyện (dựa trên việc đóng băng hay không các layer của mô hình).

Trong mỗi epoch, có các tác vụ theo dõi loss và accuracy, kiểm tra cải thiện: (Nếu có, lưu lại trạng thái mô hình. Nếu không, giảm learning rate khi vượt ngưỡng patience và sử dụng trọng số tốt nhất nếu dwell=True), lập tức dừng huấn luyện nếu giảm learning rate quá số lần quy định (stop\_patience), callback dừng huấn luyện và in thông báo.

```

Epoch 1/15
625/625 [=====] - ETA: 0s - loss: 0.9543 - accuracy: 0.9470 - precision: 0.9482 - recall: 0.9468 - auc: 0.9821 - mae: 0.0222 - mse: 0.0187 - root_mean_squared_error: 0.1969 - f1_score: 0.9470 - val_loss: 0.9543 - val_accuracy: 0.9470 - val_precision: 0.9482 - val_recall: 0.9468 - val_auc: 0.9821 - val_mae: 0.0222 - val_mse: 0.0187 - val_root_mean_squared_error: 0.1969 - val_f1_score: 0.9470
625/625 [=====] - 285s 432ms/step - loss: 0.9543 - accuracy: 0.9470 - precision: 0.9482 - recall: 0.9468 - auc: 0.9821 - mae: 0.0222 - mse: 0.0187 - root_mean_squared_error: 0.1969 - f1_score: 0.9470 - val_loss: 0.9543 - val_accuracy: 0.9470 - val_precision: 0.9482 - val_recall: 0.9468 - val_auc: 0.9821 - val_mae: 0.0222 - val_mse: 0.0187 - val_root_mean_squared_error: 0.1969 - val_f1_score: 0.9470
625/625 [=====] - 138s 221ms/step - loss: 0.0383 - accuracy: 0.9887 - precision: 0.9887 - recall: 0.9887 - auc: 0.9988 - mae: 0.0055 - mse: 0.0036 - root_mean_squared_error: 0.0087 - f1_score: 0.9887 - val_loss: 0.0383 - val_accuracy: 0.9887 - val_precision: 0.9887 - val_recall: 0.9887 - val_auc: 0.9988 - val_mae: 0.0055 - val_mse: 0.0036 - val_root_mean_squared_error: 0.0087 - val_f1_score: 0.9887
625/625 [=====] - ETA: 0s - loss: 0.0557 - accuracy: 0.9875 - precision: 0.9875 - recall: 0.9873 - auc: 0.9981 - mae: 0.0057 - mse: 0.0041 - root_mean_squared_error: 0.0115 - f1_score: 0.9875 - val_loss: 0.0557 - val_accuracy: 0.9875 - val_precision: 0.9875 - val_recall: 0.9873 - val_auc: 0.9981 - val_mae: 0.0057 - val_mse: 0.0041 - val_root_mean_squared_error: 0.0115 - val_f1_score: 0.9875
625/625 [=====] - 137s 219ms/step - loss: 0.0557 - accuracy: 0.9875 - precision: 0.9875 - recall: 0.9873 - auc: 0.9981 - mae: 0.0057 - mse: 0.0041 - root_mean_squared_error: 0.0115 - f1_score: 0.9875 - val_loss: 0.0557 - val_accuracy: 0.9875 - val_precision: 0.9875 - val_recall: 0.9873 - val_auc: 0.9981 - val_mae: 0.0057 - val_mse: 0.0041 - val_root_mean_squared_error: 0.0115 - val_f1_score: 0.9875
625/625 [=====] - ETA: 0s - loss: 0.0437 - accuracy: 0.9893 - precision: 0.9893 - recall: 0.9892 - auc: 0.9984 - mae: 0.0046 - mse: 0.0034 - root_mean_squared_error: 0.0087 - f1_score: 0.9893 - val_loss: 0.0437 - val_accuracy: 0.9893 - val_precision: 0.9893 - val_recall: 0.9892 - val_auc: 0.9984 - val_mae: 0.0046 - val_mse: 0.0034 - val_root_mean_squared_error: 0.0087 - val_f1_score: 0.9893
625/625 [=====] - 141s 225ms/step - loss: 0.0437 - accuracy: 0.9893 - precision: 0.9893 - recall: 0.9892 - auc: 0.9984 - mae: 0.0046 - mse: 0.0034 - root_mean_squared_error: 0.0087 - f1_score: 0.9893 - val_loss: 0.0437 - val_accuracy: 0.9893 - val_precision: 0.9893 - val_recall: 0.9892 - val_auc: 0.9984 - val_mae: 0.0046 - val_mse: 0.0034 - val_root_mean_squared_error: 0.0087 - val_f1_score: 0.9893
625/625 [=====] - ETA: 0s - loss: 0.0298 - accuracy: 0.9926 - precision: 0.9926 - recall: 0.9926 - auc: 0.9989 - mae: 0.0034 - mse: 0.0024 - root_mean_squared_error: 0.0041 - f1_score: 0.9926 - val_loss: 0.0298 - val_accuracy: 0.9926 - val_precision: 0.9926 - val_recall: 0.9926 - val_auc: 0.9989 - val_mae: 0.0034 - val_mse: 0.0024 - val_root_mean_squared_error: 0.0041 - val_f1_score: 0.9926
625/625 [=====] - 139s 222ms/step - loss: 0.0298 - accuracy: 0.9926 - precision: 0.9926 - recall: 0.9926 - auc: 0.9989 - mae: 0.0034 - mse: 0.0024 - root_mean_squared_error: 0.0041 - f1_score: 0.9926 - val_loss: 0.0298 - val_accuracy: 0.9926 - val_precision: 0.9926 - val_recall: 0.9926 - val_auc: 0.9989 - val_mae: 0.0034 - val_mse: 0.0024 - val_root_mean_squared_error: 0.0041 - val_f1_score: 0.9926
625/625 [=====] - ETA: 0s - loss: 0.0330 - accuracy: 0.9930 - precision: 0.9930 - recall: 0.9929 - auc: 0.9988 - mae: 0.0031 - mse: 0.0023 - root_mean_squared_error: 0.0041 - f1_score: 0.9930 - val_loss: 0.0330 - val_accuracy: 0.9930 - val_precision: 0.9930 - val_recall: 0.9929 - val_auc: 0.9988 - val_mae: 0.0031 - val_mse: 0.0023 - val_root_mean_squared_error: 0.0041 - val_f1_score: 0.9930
625/625 [=====] - 140s 223ms/step - loss: 0.0330 - accuracy: 0.9930 - precision: 0.9930 - recall: 0.9929 - auc: 0.9988 - mae: 0.0031 - mse: 0.0023 - root_mean_squared_error: 0.0041 - f1_score: 0.9930 - val_loss: 0.0330 - val_accuracy: 0.9930 - val_precision: 0.9930 - val_recall: 0.9929 - val_auc: 0.9988 - val_mae: 0.0031 - val_mse: 0.0023 - val_root_mean_squared_error: 0.0041 - val_f1_score: 0.9930
...
625/625 [=====] - 137s 218ms/step - loss: 6.2076e-06 - accuracy: 1.0000 - precision: 1.0000 - recall: 1.0000 - auc: 1.0000 - mae: 2.4708e-06 - mse: 1.0000 - root_mean_squared_error: 0.0000 - f1_score: 1.0000 - val_loss: 6.2076e-06 - val_accuracy: 1.0000 - val_precision: 1.0000 - val_recall: 1.0000 - val_auc: 1.0000 - val_mae: 6.3026e-07 - val_mse: 1.0000 - val_root_mean_squared_error: 0.0000 - val_f1_score: 1.0000
Epoch 15/15
625/625 [=====] - 145s 232ms/step - loss: 1.5762e-06 - accuracy: 1.0000 - precision: 1.0000 - recall: 1.0000 - auc: 1.0000 - mae: 6.3026e-07 - mse: 1.0000 - root_mean_squared_error: 0.0000 - f1_score: 1.0000 - val_loss: 1.5762e-06 - val_accuracy: 1.0000 - val_precision: 1.0000 - val_recall: 1.0000 - val_auc: 1.0000 - val_mae: 6.3026e-07 - val_mse: 1.0000 - val_root_mean_squared_error: 0.0000 - val_f1_score: 1.0000

```



Nhận xét: Mô hình đạt hiệu suất rất cao, cả training và validation đều đạt trên 99%. Không có dấu hiệu overfitting nghiêm trọng khi cả loss và accuracy của training/validation khá sát nhau.

### 2.2.3. Đánh giá hiệu quả mô hình

		Confusion Matrix				
		colon_aca	colon_n	lung_aca	lung_n	lung_scc
Actual	colon_aca	479	0	0	0	0
	colon_n	0	519	0	0	0
lung_aca	0	0	478	0	10	
lung_n	0	0	0	516	0	
lung_scc	0	0	3	0	495	
Predicted		colon_aca	colon_n	lung_aca	lung_n	lung_scc

Classification Report:

	precision	recall	f1-score	support
colon_aca	1.00	1.00	1.00	479
colon_n	1.00	1.00	1.00	519
lung_aca	0.99	0.98	0.99	488
lung_n	1.00	1.00	1.00	516
lung_scc	0.98	0.99	0.99	498
accuracy			0.99	2500
macro avg	0.99	0.99	0.99	2500
weighted avg	0.99	0.99	0.99	2500

Hiệu suất hoàn hảo (đạt 1.00 ở tất cả các chỉ số): Ung thư biểu mô tuyến đại tràng (colon\_aca), Mô đại tràng bình thường (colon\_n), Mô phổi bình thường (lung\_n)

Hiệu suất gần như hoàn hảo: Ung thư biểu mô tuyến phổi (lung\_aca): độ chính xác 0.99, Ung thư biểu mô tế bào vảy phổi (lung\_scc): độ chính xác 0.98.

Mô hình thể hiện hiệu suất ổn định trong cả việc phát hiện ung thư và mô bình thường. Những biến động nhỏ trong phát hiện ung thư phổi là không đáng kể và có lẽ không ảnh hưởng đến ý nghĩa lâm sàng

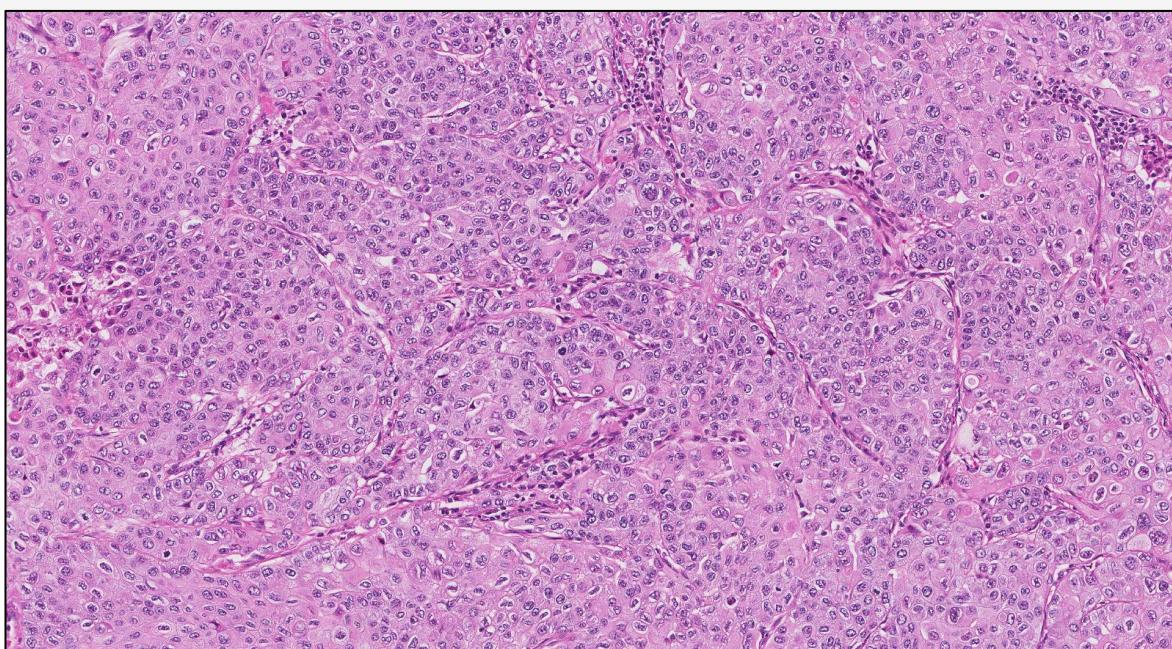
```
import tensorflow as tf
import numpy as np
from tensorflow.keras.preprocessing.image import load_img, img_to_array

image_path = '/Users/linhtuanpham/Downloads/lungscce.jpg'

img = load_img(image_path, target_size=(224, 224))
img_array = img_to_array(img)
img_array = np.expand_dims(img_array, axis=0)
img_array = img_array / 255.0

predictions = model.predict(img_array)

classes = ['colon_aca', 'colon_n', 'lung_aca', 'lung_n', 'lung_scc']
print(f"Dự đoán: {classes[predicted_class[0]]}")
```



1/1 ————— 0s 65ms/step

Dự đoán: lung\_scc

Thử đưa vào hình ảnh từ bên ngoài và có nguồn khác với bộ dữ liệu gốc để mô hình MobileNet phân loại. Ảnh được cung cấp bởi MyPathologyReport. Đây là một nguồn tài nguyên giáo dục do các bác sĩ tạo ra để giúp mọi người có thể đọc và hiểu bệnh án của mình.

Mô hình đã dự đoán đúng kết quả là lung\_scc, tức Ung thư biểu mô tế bào vảy phổi.

### 2.3. Sử dụng AlexNet để phân loại

#### 2.3.1. Nhập các thư viện và tiền xử lý dữ liệu

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense,
Dropout
from tensorflow.keras.preprocessing.image import ImageDataGenerator
import os
image_set = '/Users/linhtuanpham/Downloads/lung_colon_image_set2'
SIZE_X = SIZE_Y = 224
BATCH_SIZE = 64

datagen = ImageDataGenerator(rescale=1./255, validation_split=0.2)
train_set = datagen.flow_from_directory(
    image_set,
    target_size=(SIZE_X, SIZE_Y),
    batch_size=BATCH_SIZE,
    class_mode="categorical",
    subset="training",
    seed=42
)
validate_set = datagen.flow_from_directory(
    image_set,
    target_size=(SIZE_X, SIZE_Y),
    batch_size=BATCH_SIZE,
    class_mode="categorical",
    subset="validation",
    seed=42
)
num_classes = len(train_set.class_indices)
Found 20000 images belonging to 5 classes.
Found 5000 images belonging to 5 classes.
```

Lần này, tác giả chia bộ dữ liệu tỷ lệ 80/20, với 80% bộ dữ liệu để huấn luyện và 20% bộ dữ liệu để kiểm thử. Tập huấn luyện có 20.000 ảnh, tập kiểm thử có 5.000 ảnh, bao gồm 5 lớp để phân loại.

### 2.3.2. Khởi tạo và huấn luyện mô hình

```
model = Sequential([
    Conv2D(96, (11, 11), strides=(4, 4), activation='relu', input_shape=(SIZE_X,
SIZE_Y, 3)),
    MaxPooling2D((3, 3), strides=(2, 2)),
    Conv2D(256, (5, 5), padding='same', activation='relu'),
    MaxPooling2D((3, 3), strides=(2, 2)),
    Conv2D(384, (3, 3), padding='same', activation='relu'),
    Conv2D(384, (3, 3), padding='same', activation='relu'),
    Conv2D(256, (3, 3), padding='same', activation='relu'),
    MaxPooling2D((3, 3), strides=(2, 2)),
    Flatten(),
    Dense(4096, activation='relu'),
    Dropout(0.5),
    Dense(4096, activation='relu'),
    Dropout(0.5),
    Dense(num_classes, activation='softmax')
])
model.compile(
    optimizer=tf.keras.optimizers.Adam(learning_rate=0.0001),
    loss='categorical_crossentropy',
    metrics=['accuracy']
)
model.summary()
```

Layer (type)	Output Shape	Param #
input_layer_9 (InputLayer)	(None, 224, 224, 3)	0
mobilenet_1.00_224 (Functional)	(None, 7, 7, 1024)	3,228,864
concatenate_1 (Concatenate)	(None, 7, 7, 1024)	0
flatten_2 (Flatten)	(None, 50176)	0
dense_9 (Dense)	(None, 1024)	51,381,248
dense_10 (Dense)	(None, 512)	524,800
dense_11 (Dense)	(None, 256)	131,328
dense_12 (Dense)	(None, 5)	1,285

```
Total params: 46,767,493 (178.40 MB)
Trainable params: 46,767,493 (178.40 MB)
Non-trainable params: 0 (0.00 B)
```

Tổng quan mô hình

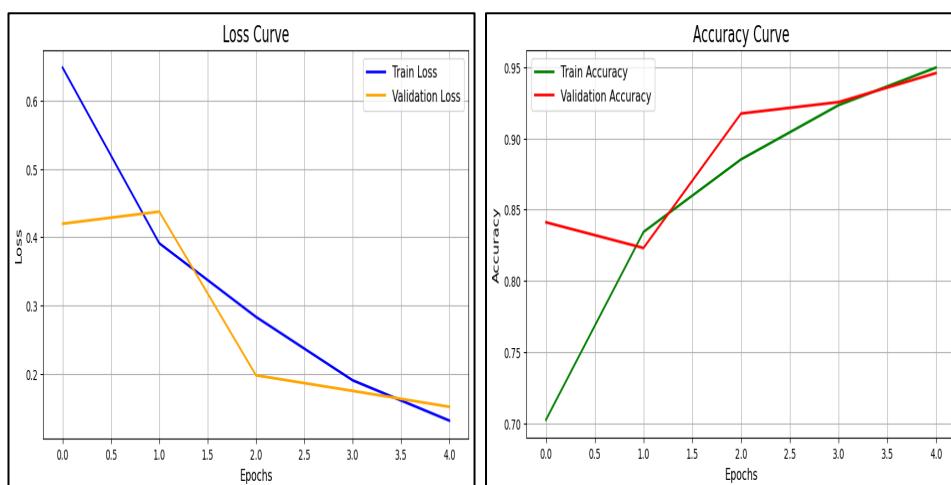
Đầu vào: Hình ảnh RGB (224x224x3).

Tiến trình: Qua các lớp tích chập và pooling để trích xuất đặc trưng → Flatten để chuyển đổi dữ liệu → Fully Connected để phân loại.

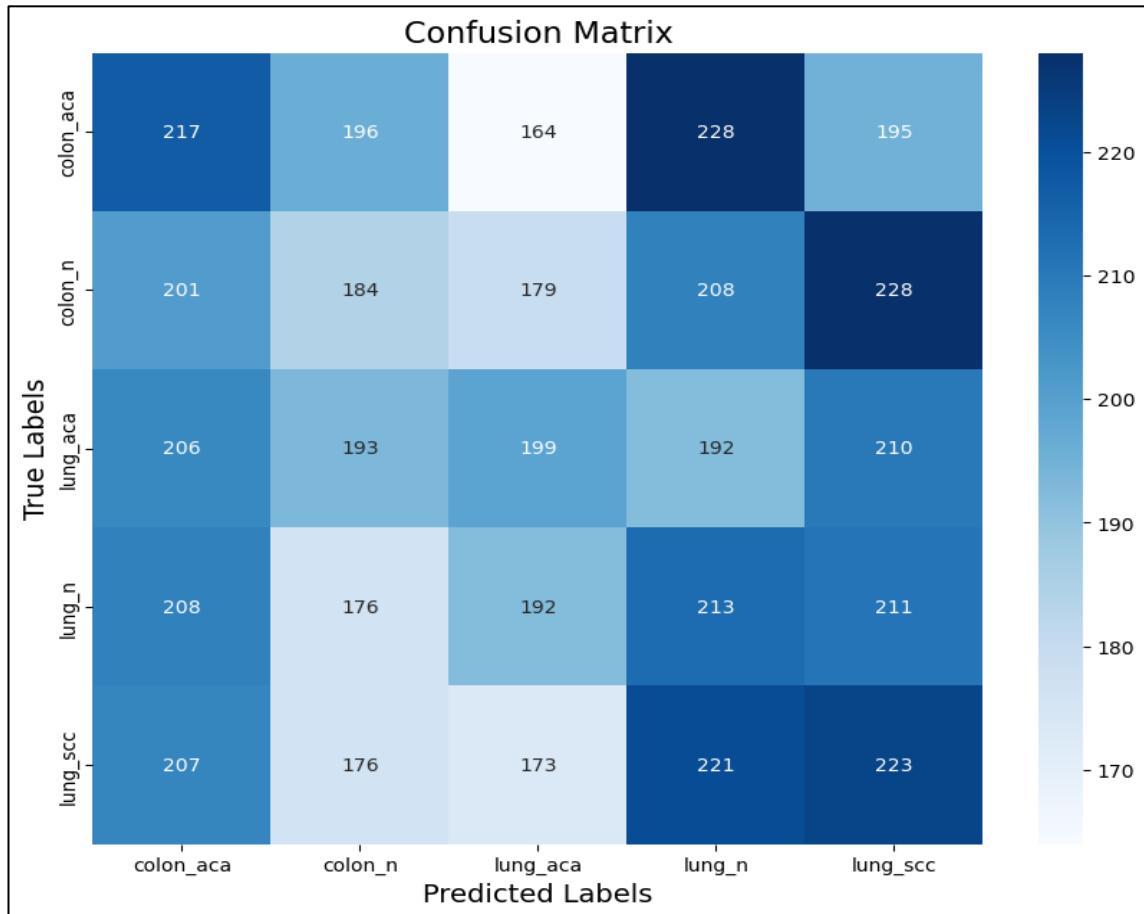
Đầu ra: Xác suất thuộc về 1 trong 5 lớp (softmax).

```
history = model.fit(
    train_set,
    validation_data=validate_set,
    epochs=5,
    verbose=1)
```

```
Epoch 1/5
313/313 ━━━━━━━━ 759s 2s/step - accuracy: 0.5874 - loss: 0.8593 -
val_accuracy: 0.8410 - val_loss: 0.4198
Epoch 2/5
313/313 ━━━━━━━━ 764s 2s/step - accuracy: 0.8228 - loss: 0.4176 -
val_accuracy: 0.8230 - val_loss: 0.4375
Epoch 3/5
313/313 ━━━━━━━━ 762s 2s/step - accuracy: 0.8601 - loss: 0.3415 -
val_accuracy: 0.9174 - val_loss: 0.1978
Epoch 4/5
313/313 ━━━━━━━━ 768s 2s/step - accuracy: 0.9067 - loss: 0.2319 -
val_accuracy: 0.9254 - val_loss: 0.1750
Epoch 5/5
313/313 ━━━━━━━━ 763s 2s/step - accuracy: 0.9469 - loss: 0.1385 -
val_accuracy: 0.9458 - val_loss: 0.1517
```



### 2.3.3. Đánh giá hiệu quả mô hình



Classification Report

	precision	recall	f1-score	support
colon_aca	0.21	0.22	0.21	1000
colon_n	0.20	0.18	0.19	1000
lung_aca	0.22	0.20	0.21	1000
lung_n	0.20	0.21	0.21	1000
lung_scc	0.21	0.22	0.22	1000
accuracy			0.21	5000
macro avg	0.21	0.21	0.21	5000
weighted avg	0.21	0.21	0.21	5000

Hai biểu đồ ở trên cho thấy mô hình được huấn luyện tốt, với Accuracy cao nhất đạt tới 94,58%. Tuy nhiên, khi sử dụng mô hình đã được huấn luyện để kiểm thử thì kết quả cho thấy mô hình đạt hiệu suất rất kém. Độ chính xác chỉ đạt 21%, gần bằng với kết quả dự đoán ngẫu nhiên (khi Random 5 kết quả). Tất cả các chỉ số

(precision, recall, f1-score) đều ở mức 0.20-0.22. Điều này cho thấy mô hình AlexNet này không học được các đặc trưng phân biệt giữa các loại mô. Do đó cần sử dụng một số biện pháp để giảm overfit.

```
datagen = ImageDataGenerator(  
    rescale=1./255,  
    rotation_range=20,  
    width_shift_range=0.2,  
    height_shift_range=0.2,  
    shear_range=0.2,  
    zoom_range=0.2,  
    horizontal_flip=True,  
    validation_split=0.2  
)  
  
train_set = datagen.flow_from_directory(  
    image_set,  
    class_mode="categorical",  
    target_size=(SIZE_X, SIZE_Y),  
    batch_size=64,  
    subset='training',  
    seed=42  
)  
  
validate_set = datagen.flow_from_directory(  
    image_set,  
    class_mode="categorical",  
    target_size=(SIZE_X, SIZE_Y),  
    batch_size=64,  
    subset='validation',  
    seed=42  
)  
  
model = Sequential([  
    Conv2D(96, (11, 11), strides=4, activation='relu', input_shape=(SIZE_X,  
SIZE_Y, 3), kernel_regularizer=l2(0.001)),  
    BatchNormalization(),  
    MaxPooling2D(pool_size=(3, 3), strides=2),  
  
    Conv2D(256, (5, 5), activation='relu', padding='same',  
kernel_regularizer=l2(0.001)),  
    BatchNormalization(),  
    MaxPooling2D(pool_size=(3, 3), strides=2),  
  
    Conv2D(384, (3, 3), activation='relu', padding='same',  
kernel_regularizer=l2(0.001)),
```

```

        Conv2D(384, (3, 3), activation='relu', padding='same',
kernel_regularizer=l2(0.001)),
        Conv2D(256, (3, 3), activation='relu', padding='same',
kernel_regularizer=l2(0.001)),
        MaxPooling2D(pool_size=(3, 3), strides=2),

        Flatten(),
        Dense(4096, activation='relu', kernel_regularizer=l2(0.001)),
        Dropout(0.5),
        Dense(4096, activation='relu', kernel_regularizer=l2(0.001)),
        Dropout(0.5),
        Dense(5, activation='softmax')

    )
model.compile(
    optimizer=tf.keras.optimizers.Adam(learning_rate=0.0001),
    loss='categorical_crossentropy',
    metrics=['accuracy']
)
history = model.fit(
    train_set,
    validation_data=validate_set,
    epochs=5,
    verbose=1
)
loss, accuracy = model.evaluate(validate_set, verbose=1)
print(f"Validation Accuracy: {accuracy * 100:.2f}%, Validation Loss:
{loss:.4f}")

```

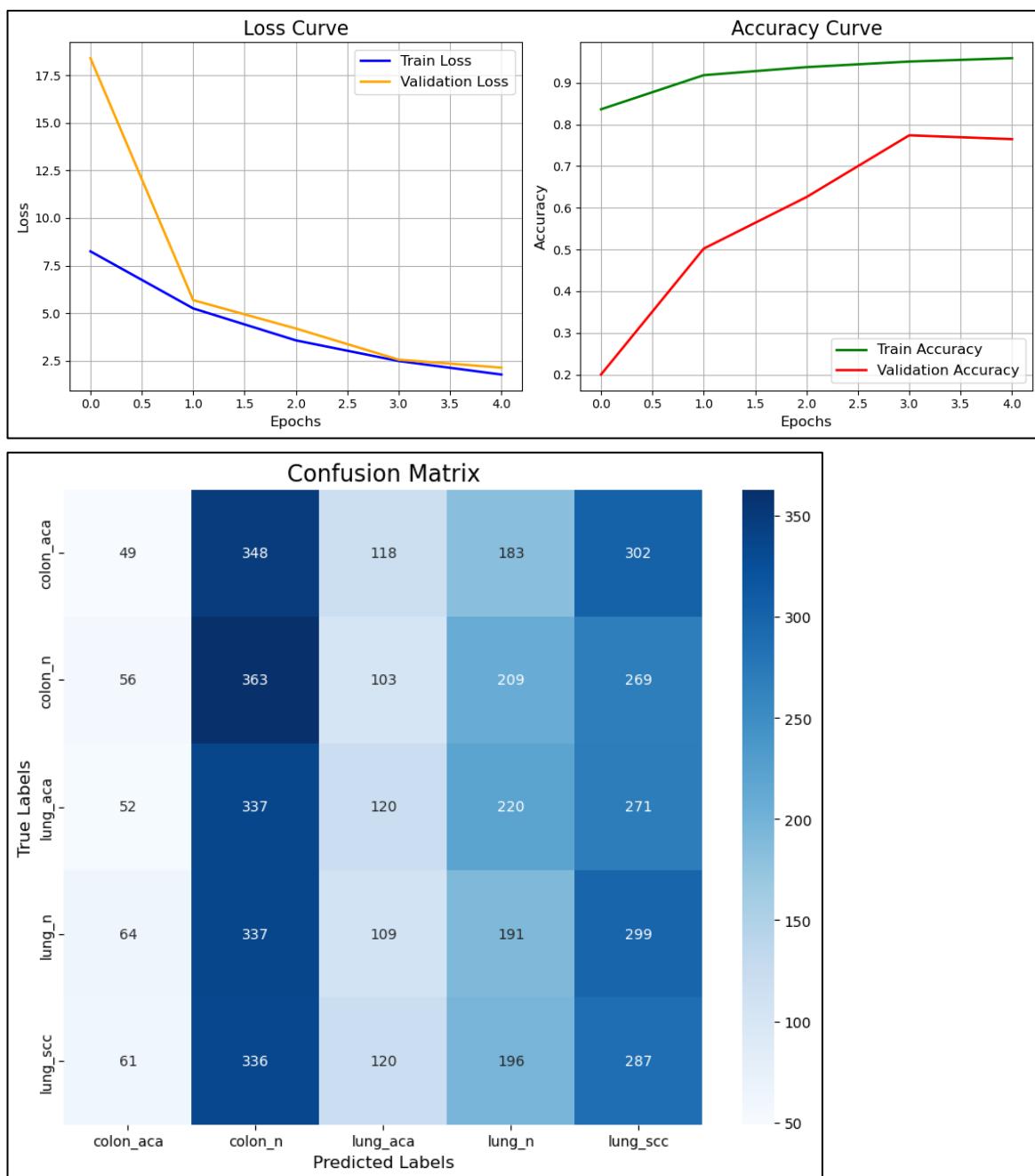
**313/313** ————— **886s** 3s/step – accuracy: 0.7502 – loss: 9.5017 –  
val\_accuracy: 0.2000 – val\_loss: 18.3934  
Epoch 2/5  
**313/313** ————— **906s** 3s/step – accuracy: 0.9050 – loss: 5.8099 –  
val\_accuracy: 0.5022 – val\_loss: 5.6917  
Epoch 3/5  
**313/313** ————— **907s** 3s/step – accuracy: 0.9361 – loss: 3.9081 –  
val\_accuracy: 0.6258 – val\_loss: 4.2028  
Epoch 4/5  
**313/313** ————— **906s** 3s/step – accuracy: 0.9506 – loss: 2.7112 –  
val\_accuracy: 0.7740 – val\_loss: 2.5683  
Epoch 5/5  
**313/313** ————— **902s** 3s/step – accuracy: 0.9587 – loss: 1.9275 –  
val\_accuracy: 0.7648 – val\_loss: 2.1481  
**79/79** ————— **83s** 1s/step – accuracy: 0.7578 – loss: 2.1457  
Validation Accuracy: 76.02%, Validation Loss: 2.1415

Sử dụng ImageDataGenerator để tăng cường dữ liệu (rotation, shift, shear, zoom, flip) nhằm tạo thêm các mẫu dữ liệu đa dạng từ tập dữ liệu ban đầu. Điều này giúp giảm overfitting, đặc biệt khi bạn không có nhiều dữ liệu.

Thêm L2 regularization (kernel\_regularizer=l2(0.001)) vào các lớp Conv2D và Dense để giảm overfitting.

Thêm Dropout (0.5) vào các lớp Dense để ngẫu nhiên bỏ qua các kết nối trong quá trình huấn luyện, giúp giảm bớt sự phụ thuộc quá mức vào các nút cụ thể.

Sử dụng Optimizer Adam với learning rate nhỏ (learning\_rate=0.0001), phù hợp để đảm bảo quá trình huấn luyện ổn định và không bị dao động mạnh.



#### Classification Report:

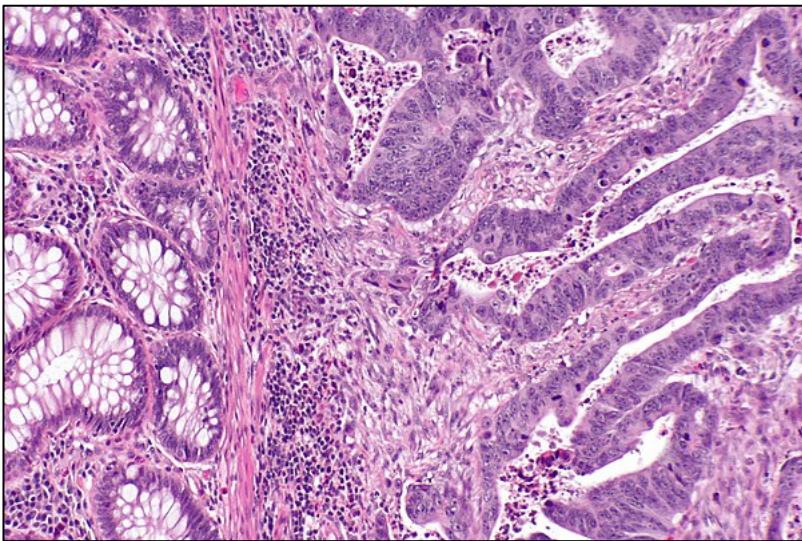
	precision	recall	f1-score	support
colon_aca	0.17	0.05	0.08	1000
colon_n	0.21	0.36	0.27	1000
lung_aca	0.21	0.12	0.15	1000
lung_n	0.19	0.19	0.19	1000
lung_scc	0.20	0.29	0.24	1000
accuracy			0.20	5000
macro avg	0.20	0.20	0.18	5000
weighted avg	0.20	0.20	0.18	5000

Nhận xét:

Độ chính xác tăng đều qua các epoch, đặc biệt là bước nhảy lớn từ epoch 1 sang epoch 2. Mô hình có dấu hiệu overfitting, khi độ chính xác của tập huấn luyện (95.87%) cao hơn nhiều so với tập kiểm thử (76.02%). Điều này chỉ ra rằng mô hình đã học quá tốt trên tập huấn luyện nhưng không tổng quát hóa được cho tập kiểm tra.

Kết quả phân loại không được cải thiện, thậm chí kém hơn so với trước khi dùng các biện pháp giảm overfit. Tuy nhiên, việc độ chính xác của tập kiểm thử khi vẽ biểu đồ đường quá lệch so với độ chính xác của tập kiểm thử khi vẽ ma trận nhầm lẫn. Có thể đã xảy ra lỗi khi khởi tạo và vẽ ma trận nhầm lẫn này.

```
import tensorflow as tf
import numpy as np
from tensorflow.keras.preprocessing.image import load_img, img_to_array
image_path = '/Users/linhtuanpham/Downloads/lungaca.jpg'
img = load_img(image_path, target_size=(224, 224))
img_array = img_to_array(img)
img_array = np.expand_dims(img_array, axis=0)
img_array = img_array / 255.0
predictions = model.predict(img_array)
predicted_class = np.argmax(predictions, axis=1)
classes = ['colon_aca', 'colon_n', 'lung_aca', 'lung_n', 'lung_scc']
print(f"Dự đoán: {classes[predicted_class[0]]}")
```



1/1 ————— 0s 118ms/step

Dự đoán: lung\_n

Thử đưa vào hình ảnh từ bên ngoài và có nguồn khác với bộ dữ liệu gốc để mô hình AlexNet phân loại (được cung cấp bởi MyPathologyReport).

Mô hình AlexNet đã dự đoán sai kết quả là lung\_n (Mô phổi bình thường), trong khi kết quả đúng là lung\_aca (Ung thư biểu mô tuyến đại tràng). Đây là lỗi sai nghiêm trọng, do đó không nên sử dụng mô hình AlexNet với bộ dữ liệu này.

## 2.4. Sử dụng ResNet để phân loại

### 2.4.1. Nhập các thư viện và tiền xử lý dữ liệu

```
import tensorflow as tf
from tensorflow.keras.layers import Input, Lambda, Dense, Flatten
from tensorflow.keras.models import Model
from tensorflow.keras.applications.resnet50 import ResNet50
from tensorflow.keras.applications.resnet50 import preprocess_input
from tensorflow.keras.preprocessing import image
from tensorflow.keras.preprocessing.image import ImageDataGenerator, load_img
from tensorflow.keras.models import Sequential
import numpy as np
from glob import glob
import matplotlib.pyplot as plt
image_set = '/Users/linhtuanpham/Downloads/lung_colon_image_set2'
SIZE_X = SIZE_Y = 224

datagen = tf.keras.preprocessing.image.ImageDataGenerator(validation_split =
0.2)

train_set = datagen.flow_from_directory(image_set,
                                         class_mode = "categorical",
```

```

        target_size = (SIZE_X,SIZE_Y),
        color_mode="rgb",
        batch_size = 128,
        shuffle = False,
        subset='training',
        seed = 42)

validate_set = datagen.flow_from_directory(image_set,
                                            class_mode = "categorical",
                                            target_size = (SIZE_X, SIZE_Y),
                                            color_mode="rgb",
                                            batch_size = 128,
                                            shuffle = False,
                                            subset='validation',
                                            seed = 42)

```

Found 20000 images belonging to 5 classes.

Found 5000 images belonging to 5 classes.

Nhập vào các thư viện và chọn kích thước ảnh đầu vào là 224x224. Đây là kích thước đầu vào tiêu chuẩn của mô hình ResNet.

Tương tự với AlexNet, chia bộ dữ liệu tỷ lệ 80/20, với 80% bộ dữ liệu để huấn luyện và 20% bộ dữ liệu để kiểm thử. Tập huấn luyện có 20.000 ảnh, tập kiểm thử có 5.000 ảnh, bao gồm 5 lớp để phân loại.

#### **2.4.2. Khởi tạo và huấn luyện mô hình**

```

IMAGE_SIZE = [224, 224]
resnet = ResNet50(input_shape=IMAGE_SIZE + [3], weights='imagenet',
include_top=False) for layer in resnet.layers:
    layer.trainable = False
flatten = Flatten()(resnet.output)
dense = Dense(256, activation = 'relu')(flatten)
dense = Dense(128, activation = 'relu')(dense)
prediction = Dense(5, activation = 'softmax')(dense)
model = Model(inputs = resnet.input, outputs = prediction)
model.summary()

```

Layer (type)	Output Shape	Param #	Connected to				
input_layer_16 (InputLayer)	(None, 224, 224, 3)	0	-	conv5_block1_3_conv (Conv2D)	(None, 7, 7, 2048)	1,050,624	conv5_block1_2_r...
conv1_pad (ZeroPadding2D)	(None, 230, 230, 3)	0	input_layer_16[0]	conv5_block1_0_bn (BatchNormalization)	(None, 7, 7, 2048)	8,192	conv5_block1_0_c...
conv1_conv (Conv2D)	(None, 112, 112, 64)	9,472	conv1_pad[0][0]	conv5_block1_3_bn (BatchNormalization)	(None, 7, 7, 2048)	8,192	conv5_block1_3_c...
conv1_bn (BatchNormalizatio...	(None, 112, 112, 64)	256	conv1_conv[0][0]	conv5_block1_add (Add)	(None, 7, 7, 2048)	0	conv5_block1_0_b...
conv1_relu (Activation)	(None, 112, 112, 64)	0	conv1_bn[0][0]	conv5_block1_out (Activation)	(None, 7, 7, 2048)	0	conv5_block1_add...
pool1_pad (ZeroPadding2D)	(None, 114, 114, 64)	0	conv1_relu[0][0]	conv5_block2_1_conv (Conv2D)	(None, 7, 7, 512)	1,049,088	conv5_block1_out...
pool1_pool (MaxPooling2D)	(None, 56, 56, 64)	0	pool1_pad[0][0]	conv5_block2_1_bn (BatchNormalization)	(None, 7, 7, 512)	2,048	conv5_block2_1_c...
conv2_block1_1_conv (Conv2D)	(None, 56, 56, 64)	4,160	pool1_pool[0][0]	conv5_block2_1_relu (Activation)	(None, 7, 7, 512)	0	conv5_block2_1_b...
conv2_block1_1_bn (BatchNormalizatio...	(None, 56, 56, 64)	256	conv2_block1_1_c...	conv5_block2_2_conv (Conv2D)	(None, 7, 7, 512)	2,359,808	conv5_block2_1_r...
conv2_block1_1_relu (Activation)	(None, 56, 56, 64)	0	conv2_block1_1_b...	conv5_block2_2_bn (BatchNormalization)	(None, 7, 7, 512)	2,048	conv5_block2_2_c...
conv2_block1_2_conv (Conv2D)	(None, 56, 56, 64)	36,928	conv2_block1_1_r...	conv5_block2_2_relu (Activation)	(None, 7, 7, 512)	0	conv5_block2_2_b...
conv2_block1_2_bn (BatchNormalizatio...	(None, 56, 56, 64)	256	conv2_block1_2_c...	conv5_block2_3_conv (Conv2D)	(None, 7, 7, 2048)	1,050,624	conv5_block2_2_r...
conv2_block1_2_relu (Activation)	(None, 56, 56, 64)	0	conv2_block1_2_b...	conv5_block2_3_bn (BatchNormalization)	(None, 7, 7, 2048)	8,192	conv5_block2_3_c...
conv2_block1_0_conv (Conv2D)	(None, 56, 56, 256)	16,640	pool1_pool[0][0]	conv5_block2_3_add (Add)	(None, 7, 7, 2048)	0	conv5_block1_out...
conv2_block1_3_conv (Conv2D)	(None, 56, 56, 256)	16,640	conv2_block1_2_r...	conv5_block2_out (Activation)	(None, 7, 7, 2048)	0	conv5_block2_add...
conv2_block1_0_bn (BatchNormalizatio...	(None, 56, 56, 256)	1,024	conv2_block1_0_c...	conv5_block3_1_conv (Conv2D)	(None, 7, 7, 512)	1,049,088	conv5_block2_out...
conv2_block1_3_bn (BatchNormalizatio...	(None, 56, 56, 256)	1,024	conv2_block1_3_c...	conv5_block3_1_bn (BatchNormalization)	(None, 7, 7, 512)	2,048	conv5_block3_1_c...
conv2_block1_0_add (Add)	(None, 56, 56, 256)	0	conv2_block1_0_b...	conv5_block3_1_relu (Activation)	(None, 7, 7, 512)	0	conv5_block3_1_b...
conv2_block1_out (Activation)	(None, 56, 56, 256)	0	conv2_block1_add...	conv5_block3_2_conv (Conv2D)	(None, 7, 7, 512)	2,359,808	conv5_block3_1_r...
conv2_block2_1_conv (Conv2D)	(None, 56, 56, 64)	16,448	conv2_block1_out[0]	conv5_block3_2_bn (BatchNormalization)	(None, 7, 7, 512)	2,048	conv5_block3_2_c...
conv2_block2_1_bn (BatchNormalizatio...	(None, 56, 56, 64)	256	conv2_block2_1_c...	conv5_block3_2_relu (Activation)	(None, 7, 7, 512)	0	conv5_block3_2_b...
conv2_block2_1_relu (Activation)	(None, 56, 56, 64)	0	conv2_block2_1_b...	conv5_block3_3_conv (Conv2D)	(None, 7, 7, 2048)	1,050,624	conv5_block3_2_r...
conv2_block2_2_conv (Conv2D)	(None, 56, 56, 64)	36,928	conv2_block2_1_r...	conv5_block3_3_bn (BatchNormalization)	(None, 7, 7, 2048)	8,192	conv5_block3_3_c...
conv2_block2_2_bn (BatchNormalizatio...	(None, 56, 56, 64)	256	conv2_block2_2_c...	conv5_block3_3_add (Add)	(None, 7, 7, 2048)	0	conv5_block2_out...
conv2_block2_2_relu (Activation)	(None, 56, 56, 64)	0	conv2_block2_2_b...	conv5_block3_out (Activation)	(None, 7, 7, 2048)	0	conv5_block3_add...
conv2_block2_3_conv (Conv2D)	(None, 56, 56, 256)	16,640	conv2_block2_2_r...	flatten_7 (Flatten)	(None, 100352)	0	conv5_block3_out...
				dense_27 (Dense)	(None, 256)	25,690,368	flatten_7[0][0]
				dense_28 (Dense)	(None, 128)	32,896	dense_27[0][0]
				dense_29 (Dense)	(None, 5)	645	dense_28[0][0]

**Total params: 49,311,621 (188.11 MB)**

**Trainable params: 25,723,909 (98.13 MB)**

**Non-trainable params: 23,587,712 (89.98 MB)**

```
model.compile(loss = 'categorical_crossentropy', optimizer = 'adam', metrics = ['accuracy'])

history = model.fit(train_set, validation_data = (validate_set), epochs = 5, verbose = 1)
```

Epoch 1/5

**157/157 ————— 1089s 7s/step - accuracy: 0.4943 - loss: 23.0923 - val\_accuracy: 0.9580 - val\_loss: 0.1947**

Epoch 2/5

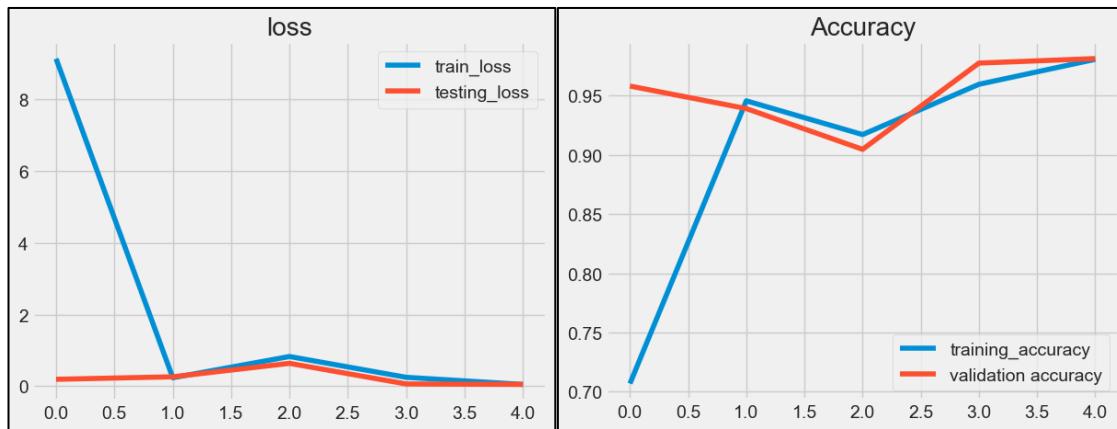
**157/157 ————— 1090s 7s/step - accuracy: 0.9477 - loss: 0.2210 - val\_accuracy: 0.9390 - val\_loss: 0.2637**

Epoch 3/5

```

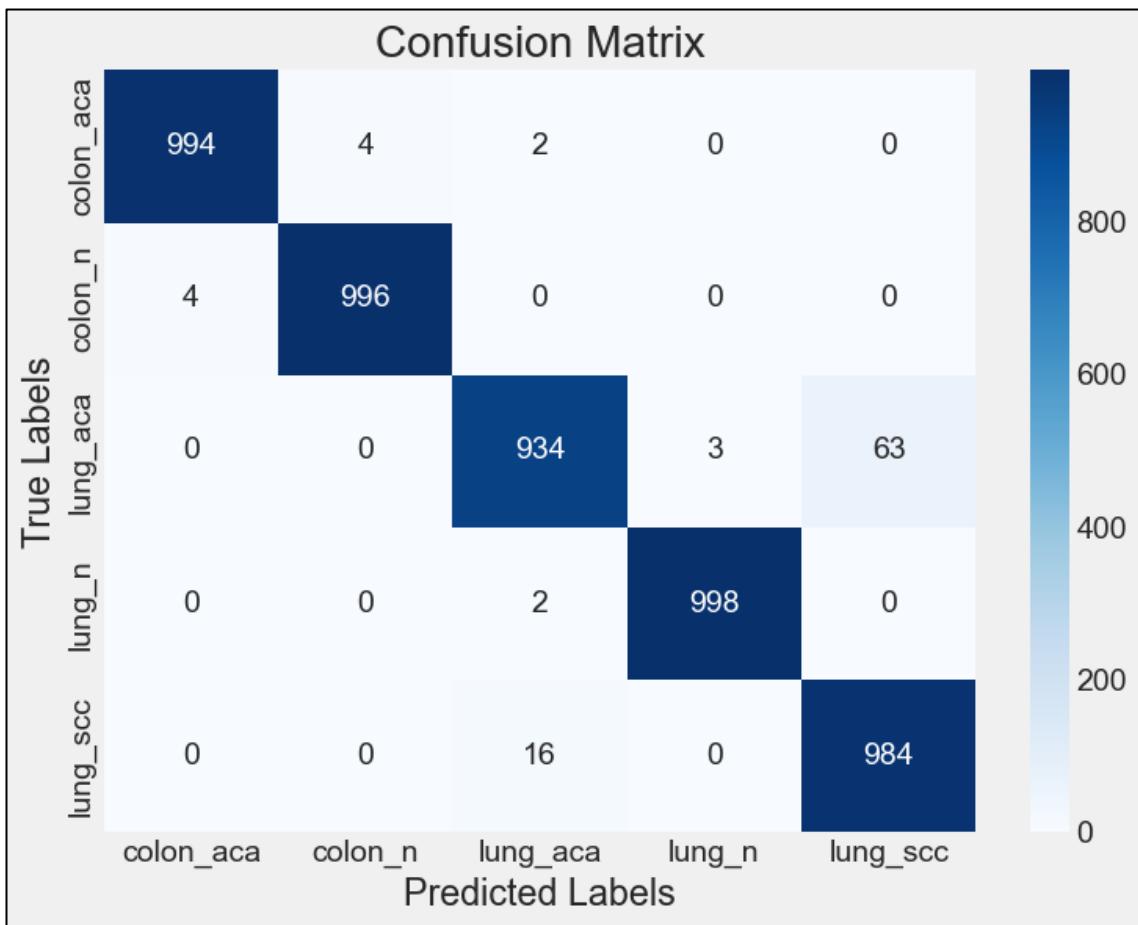
157/157 ━━━━━━━━━━ 1089s 7s/step - accuracy: 0.9247 - loss: 0.6660 -
val_accuracy: 0.9046 - val_loss: 0.6397
Epoch 4/5
157/157 ━━━━━━━━━━ 1086s 7s/step - accuracy: 0.9405 - loss: 0.4373 -
val_accuracy: 0.9774 - val_loss: 0.0645
Epoch 5/5
157/157 ━━━━━━━━━━ 1095s 7s/step - accuracy: 0.9810 - loss: 0.0515 -
val_accuracy: 0.9812 - val_loss: 0.0528

```



Mô hình có hiệu suất xuất sắc: Độ chính xác cao nhất đạt 98,12% trên tập kiểm thử. Không có dấu hiệu overfitting rõ ràng trong quá trình huấn luyện do có sự cân bằng tốt giữa training và validation ở những epoch cuối. Việc này cho thấy mô hình có khả năng tổng quát tốt.

### 2.4.3. Đánh giá hiệu quả mô hình



Classification Report

	precision	recall	f1-score	support
colon_aca	1.00	0.99	0.99	1000
colon_n	1.00	1.00	1.00	1000
lung_aca	0.98	0.93	0.96	1000
lung_n	1.00	1.00	1.00	1000
lung_scc	0.94	0.98	0.96	1000
accuracy			0.98	5000
macro avg	0.98	0.98	0.98	5000
weighted avg	0.98	0.98	0.98	5000

Mô hình đặc biệt mạnh trong việc phân loại mô bình thường (Độ chính xác 100%). Mô hình đạt độ chính xác cao trong phát hiện ung thư đại tràng (colon\_aca). Ung thư phổi (lung\_aca) có hiệu suất thấp hơn một chút nhưng vẫn rất tốt (F1-score 0.96). Đây là một mô hình có hiệu suất xuất sắc, đáp ứng được yêu cầu về độ chính xác và đáng tin cậy trong chẩn đoán y tế.

```

import tensorflow as tf
import numpy as np
from tensorflow.keras.preprocessing.image import load_img, img_to_array

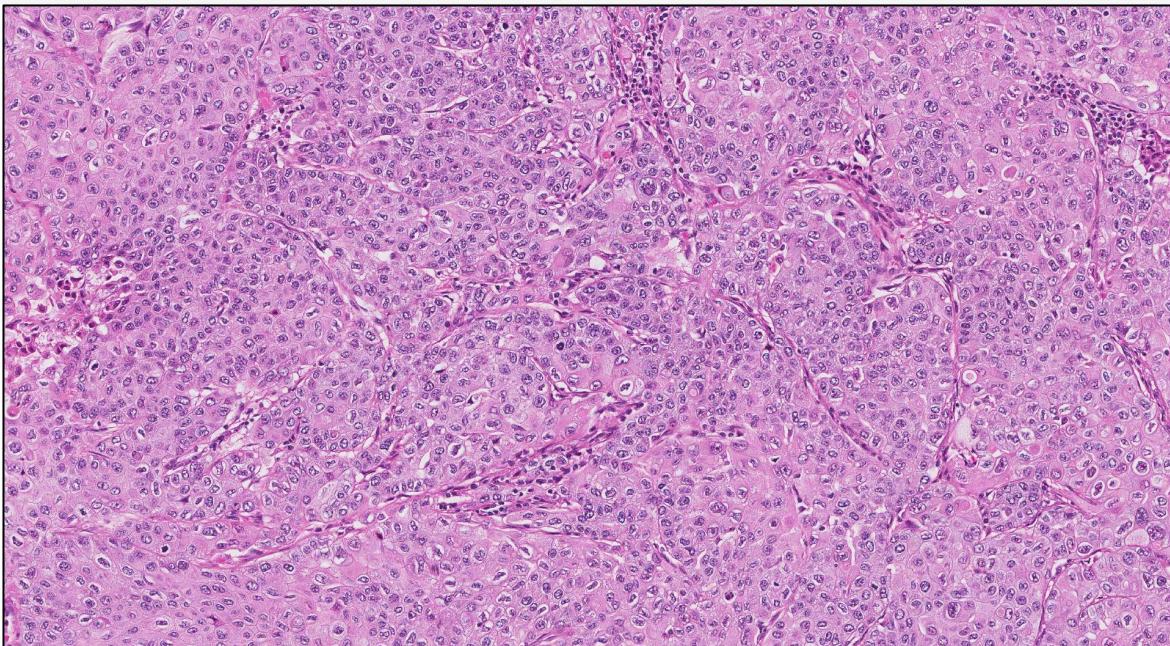
image_path = '/Users/linhtuanpham/Downloads/lungaca.jpg'

img = load_img(image_path, target_size=(224, 224))
img_array = img_to_array(img)
img_array = np.expand_dims(img_array, axis=0)
img_array = img_array / 255.0

predictions = model.predict(img_array)

classes = ['colon_aca', 'colon_n', 'lung_aca', 'lung_n', 'lung_scc']
print(f"Dự đoán: {classes[predicted_class[0]]}")

```



1/1 ————— 0s 74ms/step

Dự đoán: lung\_scc

Thử đưa vào hình ảnh từ bên ngoài và có nguồn khác với bộ dữ liệu gốc để mô hình ResNet phân loại (được cung cấp bởi MyPathologyReport).

Mô hình ResNet đã dự đoán sai kết quả là lung\_scc (Ung thư biểu mô tế bào vảy phổi), trong khi kết quả đúng là lung\_aca (Ung thư biểu mô tuyến đại tràng). Đây là lỗi sai không quá nghiêm trọng, do cả hai kết quả đều là mô bệnh ung thư ở phổi. Có thể chấp nhận sử dụng mô hình ResNet này để chẩn đoán y tế.

## **2.5. So sánh hiệu quả mô hình và kết luận**

Mô hình MobileNet hoạt động tốt nhất trong cả 3 mô hình, với Accuracy 99%, kết quả dự đoán khi đưa hình ảnh từ ngoài vào cũng chính xác. Việc này chứng tỏ mô hình MobileNet đáng tin cậy khi áp dụng vào thực tế.

Mô hình AlexNet được kỳ vọng phù hợp nhất bởi AlexNet có Dropout giúp giảm overfitting, đặc biệt hữu ích với bộ dữ liệu hạn chế (1250 ảnh x 20 lần bằng DataAugmentor). Tuy nhiên trong quá trình xử lý đã xảy ra lỗi khiến mô hình không chạy được như kết quả huấn luyện. Accuracy khi kiểm thử là 95%, nhưng chỉ đạt 20% khi kiểm tra. Khi đưa ảnh từ bên ngoài vào để mô hình phân loại cũng nhận kết quả sai nghiêm trọng, do đó mô hình AlexNet không đáng tin cậy để sử dụng trong bộ dữ liệu này.

Mô hình ResNet tuy dự đoán sai kết quả khi đưa hình ảnh từ bên ngoài vào nhưng kết quả này là chấp nhận được. Kết quả trên tập kiểm thử cũng rất khả quan (Accuracy 98%). Điều này cho thấy hoàn toàn có thể sử dụng mô hình ResNet để chẩn đoán mô bệnh ung thư trong thực tế.