Nguyễn Quang Thắng – B21DCCN669

Bài 1:

1.1.

Phần 1: Import các thư viện cần thiết

```
# Import các thư viện cần thiết
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from sklearn.model_selection import train_test_split
from sklearn.datasets import make_classification
import numpy as np
import matplotlib.pyplot as plt
```

Phần 2: Tạo dataset giả lập (phân loại nhị phân)

```
# 1. Tạo dataset giả lập (phân loại nhị phân)
X, y = make_classification(n_samples=1000, n_features=20, n_classes=2, random_state=42)
```

- make_classification: Tạo một tập dữ liệu giả để thử nghiệm mô hình phân loại nhị phân.
- n_samples=1000: Tạo 1000 mẫu dữ liệu.
- n_features=20: Mỗi mẫu có 20 đặc trưng (feature).
- n_classes=2: Phân loại thành 2 lớp.
- random_state=42: Đảm bảo tính ngẫu nhiên ổn định để kết quả có thể tái hiện lại.

Phần 3: Chia tập dữ liệu thành tập huấn luyện (train) và tập kiểm tra (validation)

```
# 2. Chia tập dữ liệu thành train (80%) và validation (20%)
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2, random_state=42)
```

- train_test_split: Chia dữ liệu thành hai phần:
- X_train, y_train: Tập huấn luyện chiếm 80% dữ liệu.
- X_val, y_val: Tập kiểm tra (validation) chiếm 20% dữ liệu.

- test_size=0.2: Chỉ định tỷ lệ dữ liệu dành cho tập validation là 20%.
- random_state=42: Đảm bảo dữ liệu chia ra sẽ cố định cho mỗi lần chạy.

Phần 4: Xây dựng mô hình MLP đơn giản

```
# 3. Xây dựng mô hình MLP đơn giản (Perceptron đa lớp)
model = Sequential()
model.add(Dense(64, activation='relu', input_shape=(X_train.shape[1],))) # Lớp đầu vào
model.add(Dense(32, activation='relu')) # Lớp ẩn
model.add(Dense(1, activation='sigmoid')) # Lớp đầu ra cho phân loại nhị phân
```

 Sequential(): Khởi tạo mô hình dạng tuần tự (các lớp sẽ được thêm vào một cách tuần tự).
 Dense(64, activation='relu', input_shape=(X_train.shape[1],)): Thêm một lớp đầu vào với:

- 64 neurons: Số lượng đơn vị trong lớp.
- activation='relu': Sử dụng hàm kích hoạt ReLU, giúp mô hình học được các phi tuyến tính.
- input_shape=(X_train.shape[1],): Định dạng đầu vào có 20 đặc trưng (bằng số đặc trưng trong tập dữ liệu).

 Dense(32, activation='relu'): Thêm một lớp ẩn với:

- 32 neurons và hàm kích hoạt ReLU.

 Dense(1, activation='sigmoid'): Thêm lớp đầu ra với:

- 1 neuron: Cho phân loại nhị phân.
- activation='sigmoid': Hàm sigmoid biến đầu ra thành giá trị giữa 0 và 1 (xác suất).

Phần 5: Compile mô hình

```
# 4. Compile mô hình
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
```

- optimizer='adam': Sử dụng thuật toán Adam để tối ưu hóa mô hình.
- loss='binary_crossentropy': Hàm mất mát (loss function) dùng cho bài toán phân loại nhị phân.
- metrics=['accuracy']: Đo lường độ chính xác của mô hình trong quá trình huấn luyện.

Phần 6: Huấn luyện mô hình

```
# 5. Huấn luyện mô hình với dữ liệu train và validate đồng thời
history = model.fit(
    X_train, y_train,
    epochs=20,
    batch_size=32,
    validation_data=(X_val, y_val),
    verbose=1
)
```

- X_train, y_train: Dữ liệu huấn luyện.
- epochs=20: Mô hình sẽ lặp lại toàn bộ dữ liệu huấn luyện 20 lần.
- batch_size=32: Sử dụng kích thước batch là 32, có nghĩa là mô hình sẽ cập nhật trọng số sau khi huấn luyện trên 32 mẫu.
- validation_data=(X_val, y_val): Sử dụng tập validation để đánh giá mô hình sau mỗi epoch.
- verbose=1: Hiển thị chi tiết quá trình huấn luyện (mỗi epoch sẽ hiển thị thông tin loss và accuracy).

```
Epoch 1/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 2s 18ms/step - accuracy: 0.5648 - loss: 0.6852 - val_accuracy: 0.7400 - val_loss: 0.5446
Epoch 2/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.8220 - loss: 0.4816 - val_accuracy: 0.8000 - val_loss: 0.4517
Epoch 3/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.8616 - loss: 0.3853 - val_accuracy: 0.8200 - val_loss: 0.4091
Epoch 4/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.8888 - loss: 0.3319 - val_accuracy: 0.8400 - val_loss: 0.3838
Epoch 5/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.8943 - loss: 0.2964 - val_accuracy: 0.8500 - val_loss: 0.3717
Epoch 6/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.8882 - loss: 0.2916 - val_accuracy: 0.8450 - val_loss: 0.3660
Epoch 7/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.8941 - loss: 0.2898 - val_accuracy: 0.8400 - val_loss: 0.3653
Epoch 8/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9050 - loss: 0.2689 - val_accuracy: 0.8400 - val_loss: 0.3639
Epoch 9/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9090 - loss: 0.2694 - val_accuracy: 0.8400 - val_loss: 0.3648
Epoch 10/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9037 - loss: 0.2542 - val_accuracy: 0.8400 - val_loss: 0.3623
Epoch 11/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9145 - loss: 0.2334 - val_accuracy: 0.8450 - val_loss: 0.3665
Epoch 12/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9078 - loss: 0.2477 - val_accuracy: 0.8400 - val_loss: 0.3649
Epoch 13/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9227 - loss: 0.2073 - val_accuracy: 0.8350 - val_loss: 0.3659
Epoch 14/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9316 - loss: 0.1977 - val_accuracy: 0.8450 - val_loss: 0.3641
Epoch 15/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9072 - loss: 0.2258 - val_accuracy: 0.8500 - val_loss: 0.3732
Epoch 16/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9371 - loss: 0.1946 - val_accuracy: 0.8450 - val_loss: 0.3747
Epoch 17/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9034 - loss: 0.2257 - val_accuracy: 0.8350 - val_loss: 0.3754
Epoch 18/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9348 - loss: 0.1883 - val_accuracy: 0.8450 - val_loss: 0.3819
Epoch 19/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9170 - loss: 0.1903 - val_accuracy: 0.8400 - val_loss: 0.3870
Epoch 20/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9195 - loss: 0.2177 - val_accuracy: 0.8250 - val_loss: 0.3807
```

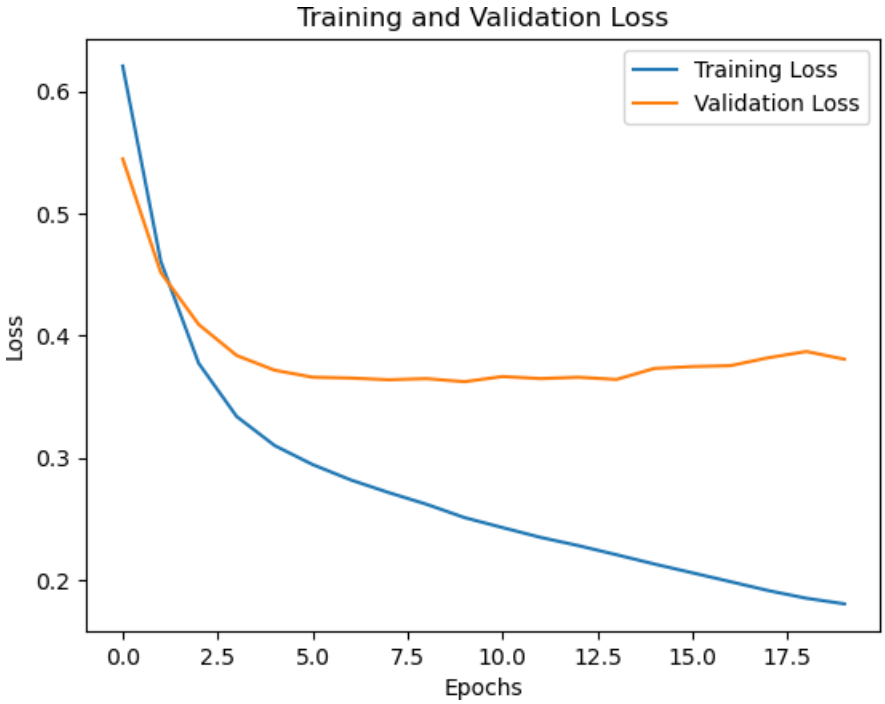
Phần 7: Đánh giá mô hình trên tập validation

- model.evaluate(X_val, y_val): Đánh giá mô hình trên tập validation.
- val_loss: Giá trị mất mát (loss) trên tập validation.
- val_accuracy: Độ chính xác (accuracy) trên tập validation.

- print: Hiển thị kết quả của loss và accuracy trên tập validation.

Phần 8: Vẽ đồ thị của Loss và Accuracy giữa tập huấn luyện và tập validation

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



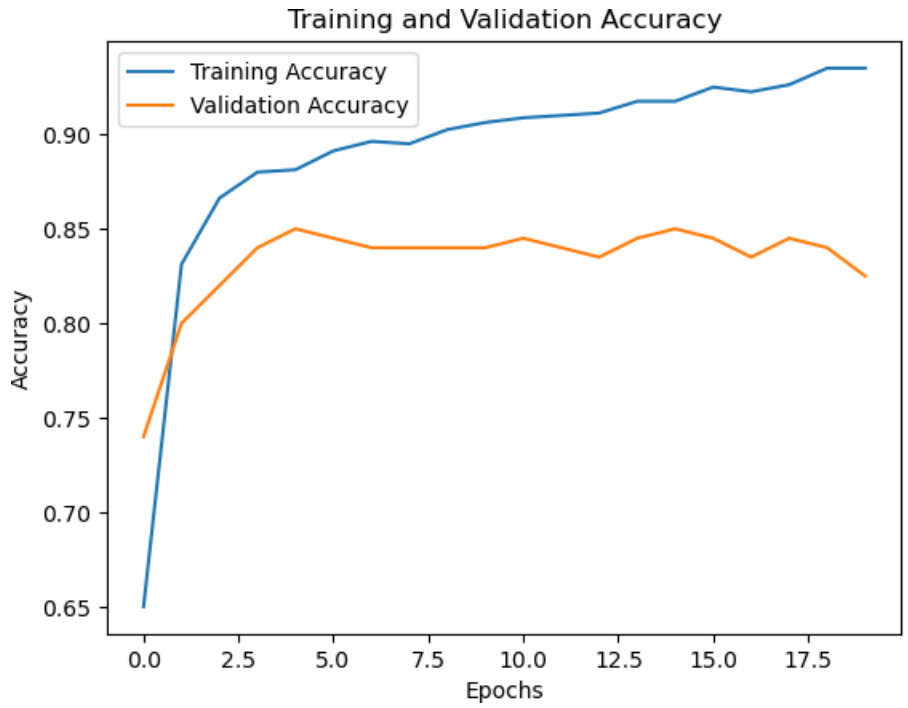
- Training Loss (Đường màu xanh): Đại diện cho độ mất mát (loss) của mô hình trên tập huấn luyện qua các epoch. Độ mất mát giảm dần cho thấy mô hình đang học từ dữ liệu, dần dần tối ưu hóa để giảm thiểu sai số trên tập huấn luyện.
- Validation Loss (Đường màu cam): Đại diện cho độ mất mát của mô hình trên tập kiểm tra (validation) qua các epoch. Độ mất mát này giúp kiểm tra khả năng khái quát hóa của mô hình đối với dữ liệu mà nó chưa được huấn luyện.

Nhận xét:

- Độ mất mát giảm dần: Khi số epoch tăng, cả training loss và validation loss đều giảm, chứng tỏ mô hình học tốt trên tập huấn luyện và cũng đang cải thiện độ chính xác trên tập kiểm tra.

- Khoảng cách giữa training loss và validation loss: Độ mất mát trên tập validation có xu hướng ổn định sau một số epoch, trong khi training loss tiếp tục giảm. Điều này có thể là dấu hiệu của sự overfitting nếu validation loss ngừng giảm hoặc bắt đầu tăng, nhưng trong trường hợp này, validation loss vẫn ổn định, cho thấy mô hình hoạt động khá tốt.

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



1. Training Accuracy (màu xanh): Đường này thể hiện độ chính xác của mô hình trên tập huấn luyện. Ta thấy đường này tăng đều, gần như không giảm, và đạt độ chính xác cao nhất ở cuối quá trình.
2. Validation Accuracy (màu cam): Đường này thể hiện độ chính xác của mô hình trên tập kiểm tra. Độ chính xác của validation tăng nhanh trong các epoch đầu tiên, nhưng sau đó dao động quanh một mức nhất định và có dấu hiệu giảm nhẹ ở cuối.

Nhận xét:

- Mô hình có dấu hiệu overfitting: Độ chính xác trên tập huấn luyện tiếp tục tăng trong khi độ chính xác trên tập kiểm tra giảm dần sau một số epoch. Điều này có nghĩa là mô hình đang học quá kỹ các mẫu trong tập huấn luyện, dẫn đến khả năng khái quát hóa kém trên dữ liệu mới.
- Để cải thiện mô hình, bạn có thể thử các kỹ thuật như regularization, dropout, hoặc điều chỉnh các tham số hyperparameters để giảm hiện tượng overfitting.

Giải thích chi tiết:

1. Epoch, Batch

- Epoch: Trong học máy, một epoch là một lần duyệt qua toàn bộ dữ liệu huấn luyện. Ví dụ, nếu dữ liệu huấn luyện có 1000 mẫu và chúng ta chọn số lượng epoch là 20, thì mô hình sẽ học từ dữ liệu 20 lần, giúp tối ưu hóa trọng số của mô hình từng chút một qua mỗi lần duyệt qua.
- Batch: Batch size là số lượng mẫu dữ liệu được sử dụng trong một lần cập nhật trọng số. Nếu batch size là 32, điều này có nghĩa là sau mỗi 32 mẫu, mô

hình sẽ tính toán lại trọng số và cập nhật chúng. Kích thước batch ảnh hưởng đến hiệu suất và tốc độ hội tụ của mô hình.

2. Train và Validation Set

- Train Set: Là tập dữ liệu dùng để huấn luyện mô hình. Dữ liệu này giúp mô hình học các mẫu và quan hệ giữa đầu vào và đầu ra, nhằm tối ưu hóa trọng số của nó.
- Validation Set: Là tập dữ liệu dùng để đánh giá hiệu suất của mô hình trong quá trình huấn luyện. Tập này giúp theo dõi xem mô hình có đang học quá nhiều (overfitting) hoặc học quá ít (underfitting) hay không. Dữ liệu validation không được sử dụng để cập nhật trọng số của mô hình mà chỉ dùng để đánh giá mô hình sau mỗi epoch.

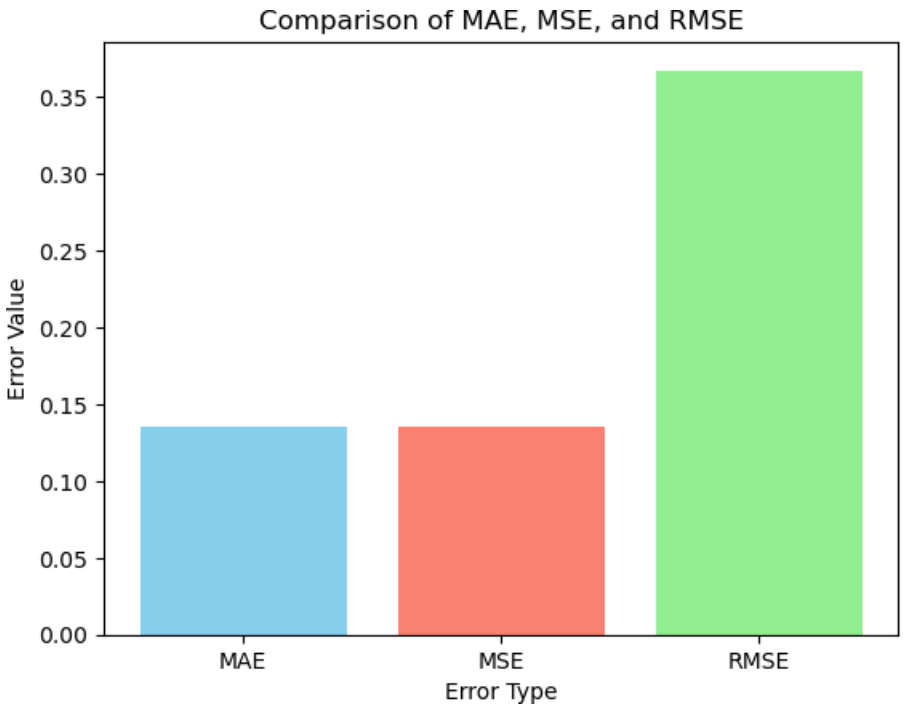
3. Thêm tập test vào và đánh giá sử dụng MAE, MSE, RMSE

```python
# 1. Chia tập dữ liệu thành train (60%), validation (20%), và test (20%)
X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.4, random_state=42)
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5, random_state=42)

# 2. Xây dựng và huấn luyện mô hình với tập train và validation
model = Sequential()
model.add(Dense(64, activation='relu', input_shape=(X_train.shape[1],)))
model.add(Dense(32, activation='relu'))
model.add(Dense(1, activation='sigmoid'))

model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
history = model.fit(
    X_train, y_train,
    epochs=20,
    batch_size=32,
    validation_data=(X_val, y_val),
    verbose=1
)

# 3. Đánh giá mô hình trên tập test
y_pred_proba = model.predict(X_test).flatten()
y_pred = np.round(y_pred_proba)  # Làm tròn thành nhãn 0 hoặc 1

# Tính các chỉ số MAE, MSE, RMSE
mae = mean_absolute_error(y_test, y_pred)
mse = mean_squared_error(y_test, y_pred)
rmse = np.sqrt(mse)

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

```
19/19 ━━━━━━━━━━━━━━━━━━━━ 1s 15ms/step - accuracy: 0.5572 - loss: 0.6870 - val_accuracy: 0.7200 - val_loss: 0.6001
Epoch 2/20
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.7939 - loss: 0.5554 - val_accuracy: 0.7950 - val_loss: 0.5084
Epoch 3/20
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.8708 - loss: 0.4404 - val_accuracy: 0.8050 - val_loss: 0.4389
Epoch 4/20
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.8796 - loss: 0.3653 - val_accuracy: 0.8250 - val_loss: 0.4043
Epoch 5/20
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.8893 - loss: 0.3075 - val_accuracy: 0.8350 - val_loss: 0.3921
Epoch 6/20
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9007 - loss: 0.2907 - val_accuracy: 0.8300 - val_loss: 0.3936
Epoch 7/20
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.8913 - loss: 0.2566 - val_accuracy: 0.8400 - val_loss: 0.3989
Epoch 8/20
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.8904 - loss: 0.2728 - val_accuracy: 0.8400 - val_loss: 0.4000
Epoch 9/20
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9016 - loss: 0.2484 - val_accuracy: 0.8400 - val_loss: 0.4035
Epoch 10/20
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9088 - loss: 0.2456 - val_accuracy: 0.8350 - val_loss: 0.4087
Epoch 11/20
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9044 - loss: 0.2423 - val_accuracy: 0.8400 - val_loss: 0.4086
Epoch 12/20
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9106 - loss: 0.2384 - val_accuracy: 0.8400 - val_loss: 0.4145
Epoch 13/20
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9134 - loss: 0.2140 - val_accuracy: 0.8400 - val_loss: 0.4160
Epoch 14/20
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9220 - loss: 0.2121 - val_accuracy: 0.8350 - val_loss: 0.4194
Epoch 15/20
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9155 - loss: 0.2104 - val_accuracy: 0.8350 - val_loss: 0.4167
Epoch 16/20
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9262 - loss: 0.2016 - val_accuracy: 0.8350 - val_loss: 0.4213
Epoch 17/20
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9348 - loss: 0.1730 - val_accuracy: 0.8350 - val_loss: 0.4287
Epoch 18/20
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9236 - loss: 0.1952 - val_accuracy: 0.8350 - val_loss: 0.4277
Epoch 19/20
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9262 - loss: 0.1859 - val_accuracy: 0.8300 - val_loss: 0.4348
Epoch 20/20
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9505 - loss: 0.1525 - val_accuracy: 0.8350 - val_loss: 0.4304
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step
```

```
Test MAE: 0.1350
Test MSE: 0.1350
Test RMSE: 0.3674
```

```python
# Tạo dữ liệu cho biểu đồ
errors = [mae, mse, rmse]
labels = ['MAE', 'MSE', 'RMSE']
colors = ['skyblue', 'salmon', 'lightgreen']

# Vẽ biểu đồ
plt.bar(labels, errors, color=colors)
plt.title('Comparison of MAE, MSE, and RMSE')
plt.xlabel('Error Type')
plt.ylabel('Error Value')
plt.show()
```



 MAE và MSE gần tương đương cho thấy rằng sai số trung bình là nhỏ, và mô hình dự đoán khá chính xác với phần lớn các điểm dữ liệu.

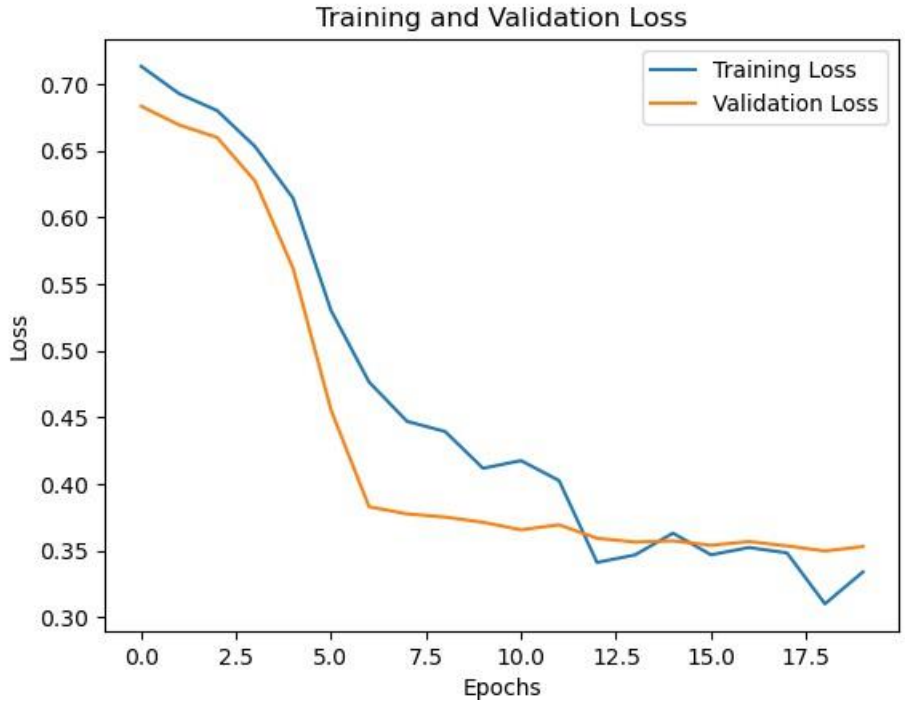
 RMSE cao hơn đáng kể cho thấy rằng mô hình có một số sai lệch lớn hơn ở một số mẫu (outliers). Điều này có thể là do các đặc điểm của dữ liệu hoặc mô hình chưa đủ phức tạp để nắm bắt đầy đủ các mẫu khó.

1.2.

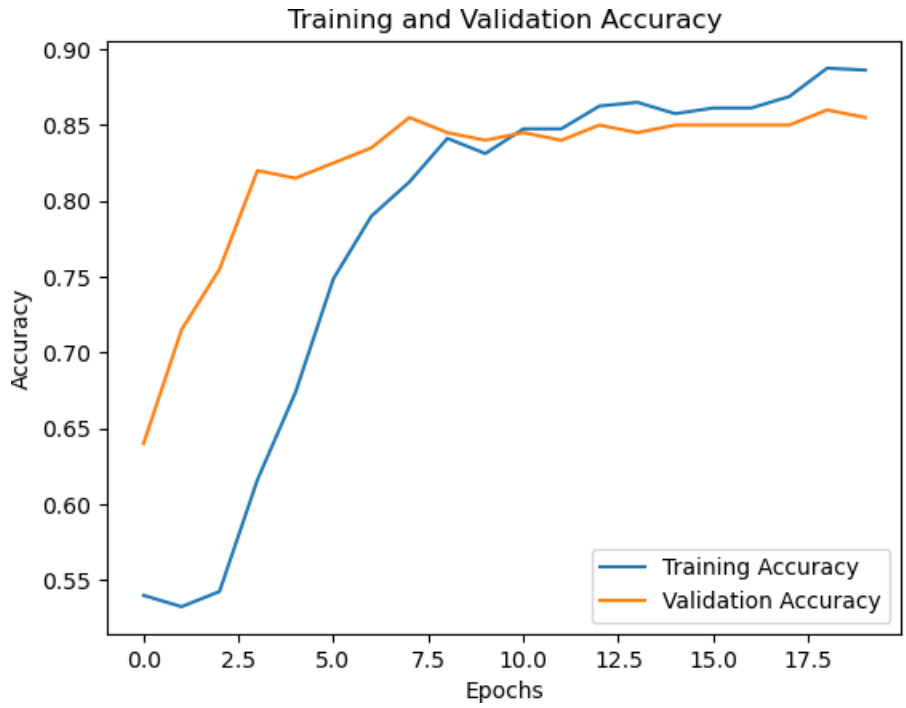
```python
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from sklearn.model_selection import train_test_split
from sklearn.datasets import make_classification
import numpy as np

# 1. Tạo dataset giả lập (phân loại nhị phân)
X, y = make_classification(n_samples=1000, n_features=20, n_classes=2, random_state=42)

# 2. Chia tập dữ liệu thành train (80%) và validation (20%)
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2, random_state=42)

# 3. Xây dựng mô hình 7 lớp với Dropout và thêm 100 neuron
model = Sequential()
model.add(Dense(128, activation='relu', input_shape=(X_train.shape[1],))) # Lớp đầu vào
model.add(Dropout(0.5))
model.add(Dense(100, activation='relu')) # Lớp ẩn với 100 neuron
model.add(Dropout(0.5))
model.add(Dense(64, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(32, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(16, activation='relu'))
model.add(Dense(8, activation='relu'))
model.add(Dense(1, activation='sigmoid')) # Lớp đầu ra cho phân loại nhị phân

# 4. Compile mô hình
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

# 5. Huấn luyện mô hình với dữ liệu train và validate đồng thời
history = model.fit(
    X_train, y_train,
    epochs=20, # Số lượng epoch
    batch_size=32, # Kích thước batch
    validation_data=(X_val, y_val), # Sử dụng tập validation
    verbose=1 # Hiển thị chi tiết quá trình huấn luyện
)
```

```
25/25 ━━━━━━━━━━━━━━━━━━━━ 2s 12ms/step - accuracy: 0.5262 - loss: 0.7278 - val_accuracy: 0.6400 - val_loss: 0.6835
Epoch 2/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.5330 - loss: 0.6883 - val_accuracy: 0.7150 - val_loss: 0.6694
Epoch 3/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.5319 - loss: 0.6825 - val_accuracy: 0.7550 - val_loss: 0.6600
Epoch 4/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.5961 - loss: 0.6642 - val_accuracy: 0.8200 - val_loss: 0.6272
Epoch 5/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.6638 - loss: 0.6210 - val_accuracy: 0.8150 - val_loss: 0.5618
Epoch 6/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.7301 - loss: 0.5528 - val_accuracy: 0.8250 - val_loss: 0.4552
Epoch 7/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.7888 - loss: 0.4997 - val_accuracy: 0.8350 - val_loss: 0.3828
Epoch 8/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.8076 - loss: 0.4408 - val_accuracy: 0.8550 - val_loss: 0.3774
Epoch 9/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.8498 - loss: 0.4073 - val_accuracy: 0.8450 - val_loss: 0.3751
Epoch 10/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.8468 - loss: 0.4065 - val_accuracy: 0.8400 - val_loss: 0.3711
Epoch 11/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.8424 - loss: 0.4121 - val_accuracy: 0.8450 - val_loss: 0.3655
Epoch 12/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.8338 - loss: 0.4420 - val_accuracy: 0.8400 - val_loss: 0.3692
Epoch 13/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.8546 - loss: 0.3434 - val_accuracy: 0.8500 - val_loss: 0.3592
Epoch 14/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.8603 - loss: 0.3538 - val_accuracy: 0.8450 - val_loss: 0.3563
Epoch 15/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.8622 - loss: 0.3574 - val_accuracy: 0.8500 - val_loss: 0.3572
Epoch 16/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.8461 - loss: 0.3577 - val_accuracy: 0.8500 - val_loss: 0.3539
Epoch 17/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.8697 - loss: 0.3419 - val_accuracy: 0.8500 - val_loss: 0.3567
Epoch 18/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.8790 - loss: 0.3444 - val_accuracy: 0.8500 - val_loss: 0.3534
Epoch 19/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.8696 - loss: 0.3433 - val_accuracy: 0.8600 - val_loss: 0.3496
Epoch 20/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.8759 - loss: 0.3061 - val_accuracy: 0.8550 - val_loss: 0.3529
```

```
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8502 - loss: 0.3750
Validation Loss: 0.3529
Validation Accuracy: 0.8550
```



- Đường màu xanh biểu diễn mất mát trên tập huấn luyện, trong khi đường màu cam biểu diễn mất mát trên tập kiểm tra.
- Mất mát của cả hai tập dữ liệu giảm dần trong các epoch đầu, cho thấy mô hình đang học để giảm thiểu lỗi.
- Tương tự như biểu đồ độ chính xác, sau khoảng epoch thứ 10, mất mát trên tập huấn luyện tiếp tục giảm nhưng mất mát trên tập kiểm tra không giảm nhiều nữa và có xu hướng giữ nguyên hoặc dao động nhẹ. Điều này củng cố dấu hiệu của overfitting khi mô hình tiếp tục giảm lỗi trên dữ liệu huấn luyện nhưng không cải thiện đáng kể trên dữ liệu kiểm tra.



 Đường màu xanh biểu diễn độ chính xác trên tập huấn luyện, trong khi đường màu cam biểu diễn độ chính xác trên tập kiểm tra.

 Nhìn chung, độ chính xác trên cả hai tập dữ liệu đều tăng lên qua các epoch, điều này cho thấy mô hình đang học và cải thiện hiệu suất.

 Tuy nhiên, sau khoảng epoch thứ 10, độ chính xác trên tập huấn luyện tiếp tục tăng nhưng độ chính xác trên tập kiểm tra bắt đầu dao động nhẹ quanh một giá trị cố định. Điều này có thể là dấu hiệu của hiện tượng overfitting, khi mô hình bắt đầu học quá kỹ các đặc điểm của dữ liệu huấn luyện và không tổng quát tốt trên dữ liệu kiểm tra.

Giải thích mô hình 7 layer:

```
model.add(Dense(100, activation='relu'))
model.add(Dropout(0.5))
```

 Dense(100, activation='relu'): Đây là lớp ẩn thứ hai trong mô hình, chứa 100 neuron và sử dụng hàm kích hoạt ReLU. Việc tăng số lượng neuron có thể giúp mô hình học được các đặc trưng phức tạp hơn từ dữ liệu.

 Dropout(0.5): Lớp Dropout được thêm vào để giảm hiện tượng overfitting. Tỷ lệ 0.5 có nghĩa là trong mỗi lần huấn luyện, 50% số neuron ở lớp trước sẽ bị "tắt" ngẫu nhiên. Điều này giúp mô hình tránh học quá mức (overfitting) bằng cách không quá phụ thuộc vào một nhóm neuron cố định nào.

```
model.add(Dense(64, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(32, activation='relu'))
model.add(Dropout(0.5))
```

 Dense(64, activation='relu') và Dense(32, activation='relu'): Đây là các lớp ẩn tiếp theo, với lần lượt 64 và 32 neuron. Các lớp này giúp mô hình có thêm độ sâu, từ đó có khả năng học các biểu diễn phức tạp hơn của dữ liệu.

 Dropout(0.5): Các lớp Dropout được thêm sau mỗi lớp Dense để tiếp tục giảm nguy cơ overfitting bằng cách tắt ngẫu nhiên 50% số neuron ở lớp trước trong mỗi lần huấn luyện.

```
model.add(Dense(16, activation='relu'))
model.add(Dense(8, activation='relu'))
```

 Dense(16, activation='relu') và Dense(8, activation='relu'): Đây là các lớp ẩn bổ sung, với 16 và 8 neuron tương ứng. Các lớp này giúp hoàn thiện mô hình với tổng cộng 7 lớp (kể cả lớp đầu vào và lớp đầu ra). Số lượng neuron giảm dần từ các lớp trước đến các lớp sau, tạo ra một cấu trúc dạng "phễu", giúp mô hình trích xuất dần dần các đặc trưng quan trọng.

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

- Dense(1, activation='sigmoid'): Đây là lớp đầu ra của mô hình, với 1 neuron và hàm kích hoạt sigmoid. Sigmoid được sử dụng để đưa ra xác suất cho phân loại nhị phân, trong đó đầu ra là giá trị trong khoảng từ 0 đến 1.

1.2.3. Thêm tập test vào và đánh giá sử dụng MAE, MSE, RMSE

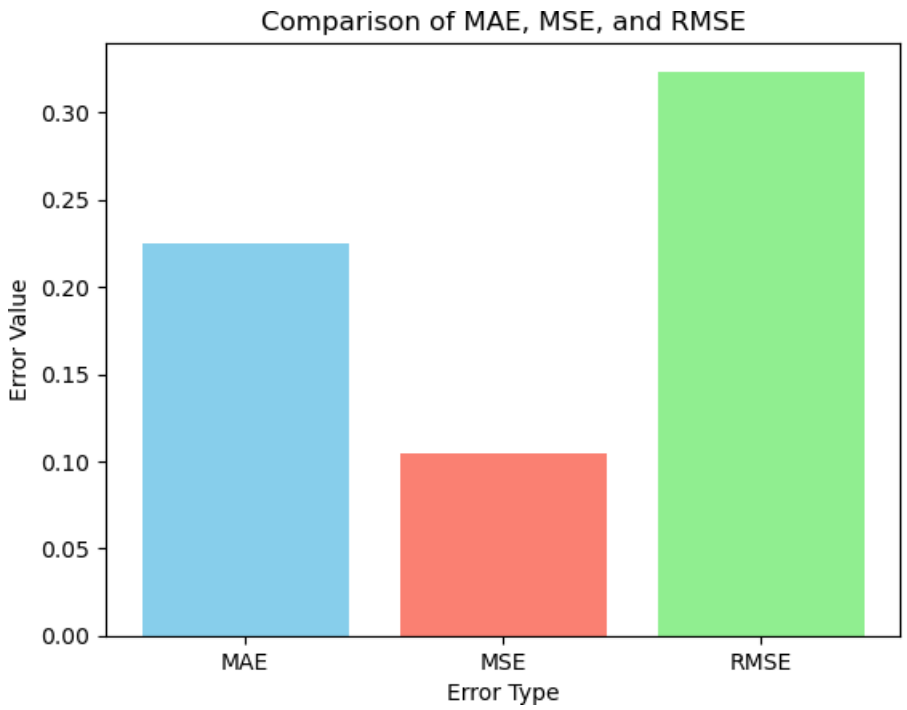
Để thêm tập kiểm tra (test set) và đánh giá mô hình sử dụng các chỉ số MAE (Mean Absolute Error), MSE (Mean Squared Error), và RMSE (Root Mean Squared Error), ta có thể thực hiện theo các bước dưới đây:

1. Chia dữ liệu thành ba tập: train (80%), validation (10%), và test (10%).
2. Huấn luyện mô hình trên tập train và validation.
3. Đánh giá mô hình trên tập test với các chỉ số MAE, MSE và RMSE.

```python
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_absolute_error, mean_squared_error
import numpy as np

# 1. Tạo dataset giả lập (phân loại nhị phân)
X, y = make_classification(n_samples=1000, n_features=20, n_classes=2, random_state=42)

# 2. Chia tập dữ liệu thành train (80%), validation (10%), và test (10%)
X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.2, random_state=42)
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5, random_state=42)

# 3. Xây dựng mô hình 7 lớp với Dropout và thêm 100 neuron
model = Sequential()
model.add(Dense(128, activation='relu', input_shape=(X_train.shape[1],)))  # Lớp đầu vào
model.add(Dropout(0.5))
model.add(Dense(100, activation='relu'))  # Lớp ẩn với 100 neuron
model.add(Dropout(0.5))
model.add(Dense(64, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(32, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(16, activation='relu'))
model.add(Dense(8, activation='relu'))
model.add(Dense(1, activation='sigmoid'))  # Lớp đầu ra cho phân loại nhị phân
```

```python
# 4. Compile mô hình
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

# 5. Huấn luyện mô hình với dữ liệu train và validate đồng thời
history = model.fit(
    X_train, y_train,
    epochs=20,  # Số lượng epoch
    batch_size=32,  # Kích thước batch
    validation_data=(X_val, y_val),  # Sử dụng tập validation
    verbose=1  # Hiển thị chi tiết quá trình huấn luyện
)

# 6. Đánh giá mô hình trên tập test và tính các chỉ số MAE, MSE, RMSE
y_pred = model.predict(X_test)
y_pred = y_pred.flatten()  # Chuyển đổi về mảng 1D

# Tính MAE, MSE, RMSE
mae = mean_absolute_error(y_test, y_pred)
mse = mean_squared_error(y_test, y_pred)
rmse = np.sqrt(mse)

print(f"Mean Absolute Error (MAE): {mae}")
print(f"Mean Squared Error (MSE): {mse}")
print(f"Root Mean Squared Error (RMSE): {rmse}")
```

```
Epoch 1/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 2s 11ms/step - accuracy: 0.5029 - loss: 0.7144 - val_accuracy: 0.6600 - val_loss: 0.6757
Epoch 2/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.5334 - loss: 0.6975 - val_accuracy: 0.7100 - val_loss: 0.6653
Epoch 3/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.5670 - loss: 0.6859 - val_accuracy: 0.7400 - val_loss: 0.6601
Epoch 4/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.6220 - loss: 0.6549 - val_accuracy: 0.7700 - val_loss: 0.6324
Epoch 5/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.6007 - loss: 0.6437 - val_accuracy: 0.8200 - val_loss: 0.5820
Epoch 6/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.7009 - loss: 0.5930 - val_accuracy: 0.8200 - val_loss: 0.4895
Epoch 7/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.7385 - loss: 0.5670 - val_accuracy: 0.8200 - val_loss: 0.4145
Epoch 8/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.7694 - loss: 0.5049 - val_accuracy: 0.8300 - val_loss: 0.3800
Epoch 9/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.8124 - loss: 0.4391 - val_accuracy: 0.8300 - val_loss: 0.3763
Epoch 10/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.8403 - loss: 0.3891 - val_accuracy: 0.8400 - val_loss: 0.3660
Epoch 11/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.8054 - loss: 0.4471 - val_accuracy: 0.8400 - val_loss: 0.3663
Epoch 12/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.8498 - loss: 0.4186 - val_accuracy: 0.8400 - val_loss: 0.3653
Epoch 13/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.8226 - loss: 0.4445 - val_accuracy: 0.8300 - val_loss: 0.3704
Epoch 14/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.8600 - loss: 0.3477 - val_accuracy: 0.8200 - val_loss: 0.3707
Epoch 15/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.8498 - loss: 0.3677 - val_accuracy: 0.8400 - val_loss: 0.3706
Epoch 16/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.8652 - loss: 0.3419 - val_accuracy: 0.8400 - val_loss: 0.3667
Epoch 17/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.8542 - loss: 0.3849 - val_accuracy: 0.8400 - val_loss: 0.3668
Epoch 18/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.8474 - loss: 0.3530 - val_accuracy: 0.8300 - val_loss: 0.3669
Epoch 19/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.8712 - loss: 0.3477 - val_accuracy: 0.8200 - val_loss: 0.3615
Epoch 20/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.8662 - loss: 0.4124 - val_accuracy: 0.8400 - val_loss: 0.3656
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 23ms/step
Mean Absolute Error (MAE): 0.22482746215071528
Mean Squared Error (MSE): 0.104671755327408
Root Mean Squared Error (RMSE): 0.32353014593296864
```



1.3. Giải thích CNN, RNN, LSTM

1. CNN (Convolutional Neural Network) - Mạng nơ-ron tích chập

CNN là một loại mạng nơ-ron chuyên dụng, được thiết kế để xử lý dữ liệu dạng hình ảnh và chuỗi có cấu trúc không gian (như ảnh, video). Đặc điểm nổi bật của CNN là sử dụng các lớp tích chập (convolutional layers) và lớp gộp (pooling layers).

- Lớp tích chập (Convolutional Layer):
    - Lớp này sử dụng các kernel (hoặc filter) - một ma trận có kích thước nhỏ hơn ảnh đầu vào. Filter này quét qua ảnh và trích xuất các đặc điểm (feature) quan trọng của ảnh, như cạnh, góc, hoặc các mẫu phức tạp hơn.
    - Sau mỗi lần quét (convolution), một bản đồ đặc trưng (feature map) được tạo ra, thể hiện thông tin về đặc trưng đã được trích xuất.
- Lớp gộp (Pooling Layer):

    - Lớp này giảm kích thước của các feature map mà không làm mất đi các đặc trưng quan trọng. Quá trình này giúp giảm số lượng tham số và tính toán trong mạng, từ đó làm cho mô hình nhanh hơn và ít overfitting hơn.
    - Phổ biến nhất là max pooling - chọn giá trị lớn nhất trong vùng được quét, giúp tập trung vào các đặc trưng mạnh nhất.
- Ứng dụng của CNN:
    - Phân loại ảnh, nhận diện đối tượng, phân đoạn ảnh, nhận diện khuôn mặt.
    - CNN cũng có thể được áp dụng cho các chuỗi thời gian (như xử lý tín hiệu âm thanh, phân tích chuỗi số liệu) bằng cách thay đổi các lớp tích chập để hoạt động trên một chiều dữ liệu.

2. RNN (Recurrent Neural Network) - Mạng nơ-ron hồi quy

RNN là một loại mạng nơ-ron chuyên dụng cho dữ liệu dạng chuỗi (sequential data), như văn bản, âm thanh hoặc chuỗi thời gian. Điểm khác biệt của RNN là khả năng ghi nhớ thông tin từ các bước trước đó trong chuỗi và sử dụng nó để xử lý bước tiếp theo.

- Hoạt động của RNN:
    - Tại mỗi bước thời gian ttt, RNN nhận đầu vào (input) xtx_txt và trạng thái ẩn (hidden state) từ bước trước đó ht−1h_{t-1}ht−1.
    - RNN sẽ tính toán trạng thái ẩn mới hth_tht dựa trên đầu vào hiện tại xtx_txt và trạng thái ẩn trước đó ht−1h_{t-1}ht−1. Do đó, RNN có thể giữ thông tin từ bước trước và lan truyền nó qua toàn bộ chuỗi.
- Vấn đề vanishing gradient (độ dốc biến mất):

    - Khi chuỗi dữ liệu dài, RNN gặp khó khăn trong việc ghi nhớ các thông tin từ quá khứ xa, do các gradient trở nên rất nhỏ và khó lan truyền ngược qua nhiều bước thời gian.
    - Điều này làm RNN gặp hạn chế khi làm việc với chuỗi dài hoặc các dữ liệu cần "nhớ" các thông tin từ lâu.
- Ứng dụng của RNN:
    - Dự đoán chuỗi thời gian, xử lý ngôn ngữ tự nhiên (NLP) như dịch máy, phân loại cảm xúc, tạo văn bản, và nhận diện giọng nói.

3. LSTM (Long Short-Term Memory) - Mạng nơ-ron bộ nhớ dài-ngắn

LSTM là một biến thể của RNN, được thiết kế để khắc phục vấn đề vanishing gradient và cải thiện khả năng ghi nhớ các thông tin từ quá khứ xa.

- Kiến trúc của LSTM:
    - Mỗi LSTM có ba cổng (gate): cổng vào (input gate), cổng quên (forget gate), và cổng đầu ra (output gate).
        - Cổng vào quyết định bao nhiêu thông tin từ đầu vào hiện tại sẽ được lưu vào trạng thái ô nhớ.
        - Cổng quên quyết định phần nào trong trạng thái ô nhớ sẽ bị "quên" hoặc bỏ qua.
        - Cổng đầu ra quyết định phần nào của trạng thái ô nhớ sẽ được đưa ra làm đầu ra.
    - Nhờ các cổng này, LSTM có thể kiểm soát dòng chảy của thông tin tốt hơn so với RNN truyền thống, cho phép nó lưu giữ và loại bỏ thông tin theo cách phù hợp với mỗi bước thời gian.
- Khả năng ghi nhớ dài hạn và ngắn hạn:

    - LSTM sử dụng hai trạng thái khác nhau: trạng thái ô nhớ (cell state), lưu giữ thông tin trong thời gian dài, và trạng thái ẩn (hidden state), lưu giữ thông tin ngắn hạn.
    - Điều này cho phép LSTM lưu trữ các thông tin dài hạn mà vẫn có thể xử lý các thông tin ngắn hạn, nhờ đó giải quyết tốt vấn đề vanishing gradient.
- Ứng dụng của LSTM:
    - Tương tự như RNN, nhưng đặc biệt tốt hơn khi xử lý chuỗi dài hoặc dữ liệu cần ghi nhớ lâu dài. LSTM thường được dùng trong dịch máy, phân tích chuỗi thời gian dài, dự đoán hành vi người dùng, và tạo nhạc.

1.4. Thay bởi mô hình CNN, RNN, LSTM với >=5 layer và 100 neuron

1.4.1. CNN

1.4.1: CNN

```
from tensorflow.keras.layers import Conv1D, MaxPooling1D, Flatten

# Mô hình CNN
cnn_model = Sequential()
cnn_model.add(Conv1D(64, kernel_size=3, activation='relu', input_shape=(X_train.shape[1], 1)))
cnn_model.add(MaxPooling1D(pool_size=2))
cnn_model.add(Conv1D(128, kernel_size=3, activation='relu'))
cnn_model.add(MaxPooling1D(pool_size=2))
cnn_model.add(Flatten())
cnn_model.add(Dense(100, activation='relu'))
cnn_model.add(Dense(50, activation='relu'))
cnn_model.add(Dense(1, activation='sigmoid'))

# Compile và huấn luyện
cnn_model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
cnn_history = cnn_model.fit(X_train.reshape(-1, X_train.shape[1], 1), y_train,
                            epochs=20, batch_size=32, validation_data=(X_val.reshape(-1, X_val.shape[1], 1), y_val))

# Đánh giá trên tập test
y_cnn_pred = cnn_model.predict(X_test.reshape(-1, X_test.shape[1], 1)).flatten()
cnn_mae = mean_absolute_error(y_test, y_cnn_pred)
cnn_mse = mean_squared_error(y_test, y_cnn_pred)
cnn_rmse = np.sqrt(cnn_mse)
```

```
25/25 ━━━━━━━━━━━━━━━━━━━━ 2s 12ms/step - accuracy: 0.6508 - loss: 0.6348 - val_accuracy: 0.8100 - val_loss: 0.4261
Epoch 2/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.8102 - loss: 0.4348 - val_accuracy: 0.8300 - val_loss: 0.3779
Epoch 3/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.8528 - loss: 0.3588 - val_accuracy: 0.8300 - val_loss: 0.3794
Epoch 4/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.8800 - loss: 0.3195 - val_accuracy: 0.8600 - val_loss: 0.3730
Epoch 5/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.8980 - loss: 0.2663 - val_accuracy: 0.8100 - val_loss: 0.3778
Epoch 6/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.8820 - loss: 0.2783 - val_accuracy: 0.8200 - val_loss: 0.4292
Epoch 7/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9012 - loss: 0.2513 - val_accuracy: 0.8500 - val_loss: 0.3986
Epoch 8/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9233 - loss: 0.2201 - val_accuracy: 0.8400 - val_loss: 0.3911
Epoch 9/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9210 - loss: 0.2104 - val_accuracy: 0.8100 - val_loss: 0.4468
Epoch 10/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9171 - loss: 0.2015 - val_accuracy: 0.8400 - val_loss: 0.4034
Epoch 11/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9383 - loss: 0.1699 - val_accuracy: 0.8300 - val_loss: 0.4418
Epoch 12/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9428 - loss: 0.1656 - val_accuracy: 0.8400 - val_loss: 0.4111
Epoch 13/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9474 - loss: 0.1520 - val_accuracy: 0.8000 - val_loss: 0.5400
Epoch 14/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9402 - loss: 0.1569 - val_accuracy: 0.8500 - val_loss: 0.4849
Epoch 15/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9633 - loss: 0.1125 - val_accuracy: 0.8300 - val_loss: 0.4979
Epoch 16/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9572 - loss: 0.1283 - val_accuracy: 0.8100 - val_loss: 0.5457
Epoch 17/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9604 - loss: 0.1051 - val_accuracy: 0.8500 - val_loss: 0.5336
Epoch 18/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9758 - loss: 0.0817 - val_accuracy: 0.8100 - val_loss: 0.5913
Epoch 19/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9821 - loss: 0.0715 - val_accuracy: 0.8100 - val_loss: 0.6376
Epoch 20/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9948 - loss: 0.0453 - val_accuracy: 0.8400 - val_loss: 0.6010
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 26ms/step
```

 Conv1D(64, kernel_size=3, activation='relu', input_shape=(X_train.shape[1], 1)):

- Đây là lớp tích chập 1 chiều đầu tiên với 64 bộ lọc (filter), kích thước bộ lọc là 3.
- Hàm kích hoạt là ReLU, giúp thêm tính phi tuyến vào mô hình.
- input_shape=(X_train.shape[1], 1) xác định kích thước đầu vào (số đặc trưng trong tập huấn luyện, kèm theo một kênh).

 MaxPooling1D(pool_size=2):

- Đây là lớp pooling giúp giảm kích thước đầu ra của lớp tích chập trước đó bằng cách chọn giá trị lớn nhất từ mỗi cửa sổ có kích thước pool_size=2.
- Pooling giúp giảm số lượng tham số và tính toán, giảm khả năng overfitting.

 Conv1D(128, kernel_size=3, activation='relu'):

- Thêm một lớp tích chập với 128 bộ lọc và kích thước bộ lọc là 3.
- Việc tăng số lượng bộ lọc giúp mô hình có khả năng học được nhiều đặc trưng phức tạp hơn từ dữ liệu.

 MaxPooling1D(pool_size=2):

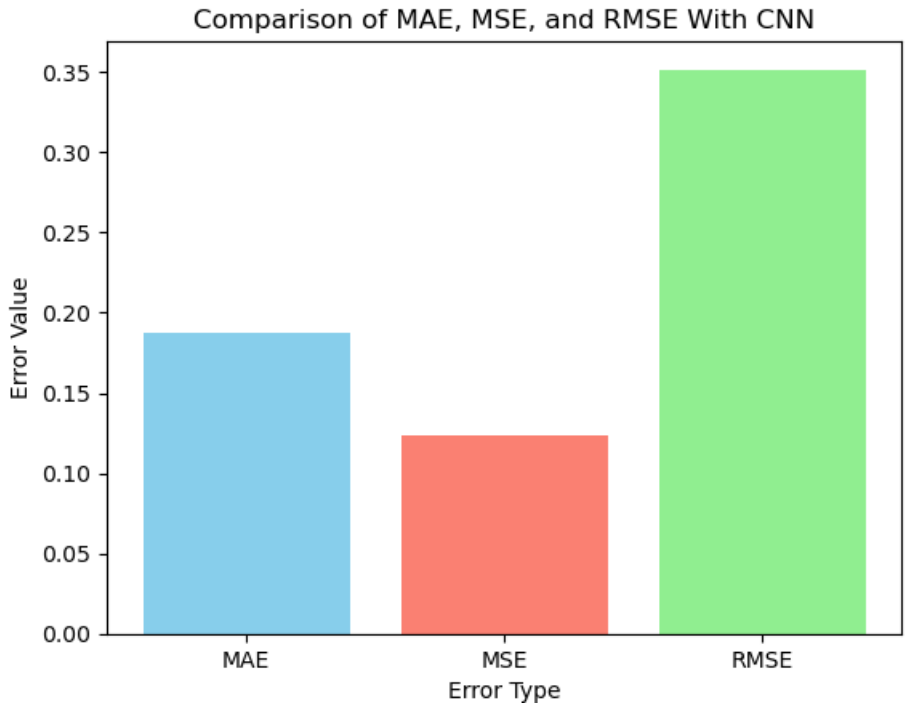
- Thêm một lớp pooling khác để tiếp tục giảm kích thước không gian của dữ liệu.

 Flatten():

- Chuyển đổi dữ liệu từ dạng 2D thành dạng 1D để đưa vào các lớp fully-connected tiếp theo.

- Dense(100, activation='relu'):
  - Thêm một lớp fully-connected với 100 neuron và hàm kích hoạt ReLU.
  - Đây là lớp đầu tiên trong số các lớp fully-connected có số lượng neuron lớn.
- Dense(50, activation='relu'):
  - Thêm một lớp fully-connected với 50 neuron và hàm kích hoạt ReLU.
- Dense(1, activation='sigmoid'):
  - Lớp đầu ra với 1 neuron và hàm kích hoạt sigmoid, vì đây là bài toán phân loại nhị phân.

```python
# Tạo dữ liệu cho biểu đồ
errors = [cnn_mae, cnn_mse, cnn_rmse]
labels = ['MAE', 'MSE', 'RMSE']
colors = ['skyblue', 'salmon', 'lightgreen']

# Vẽ biểu đồ
plt.bar(labels, errors, color=colors)
plt.title('Comparison of MAE, MSE, and RMSE With CNN')
plt.xlabel('Error Type')
plt.ylabel('Error Value')
plt.show()
```



1.4.2. RNN

1.4.2. RNN

```
from tensorflow.keras.layers import SimpleRNN

# Mô hình RNN
rnn_model = Sequential()
rnn_model.add(SimpleRNN(64, activation='relu', input_shape=(X_train.shape[1], 1), return_sequences=True))
rnn_model.add(SimpleRNN(64, activation='relu', return_sequences=True))
rnn_model.add(SimpleRNN(64, activation='relu'))
rnn_model.add(Dense(100, activation='relu'))
rnn_model.add(Dense(50, activation='relu'))
rnn_model.add(Dense(1, activation='sigmoid'))

# Compile và huấn luyện
rnn_model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
rnn_history = rnn_model.fit(X_train.reshape(-1, X_train.shape[1], 1), y_train,
                            epochs=20, batch_size=32, validation_data=(X_val.reshape(-1, X_val.shape[1], 1), y_val))

# Đánh giá trên tập test
y_rnn_pred = rnn_model.predict(X_test.reshape(-1, X_test.shape[1], 1)).flatten()
rnn_mae = mean_absolute_error(y_test, y_rnn_pred)
rnn_mse = mean_squared_error(y_test, y_rnn_pred)
rnn_rmse = np.sqrt(rnn_mse)
```

```
25/25 ━━━━━━━━━━━━━━━━━━━━ 4s 44ms/step - accuracy: 0.6641 - loss: 0.6440 - val_accuracy: 0.8400 - val_loss: 0.4127
Epoch 2/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.8529 - loss: 0.3719 - val_accuracy: 0.8200 - val_loss: 0.3723
Epoch 3/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step - accuracy: 0.8580 - loss: 0.3625 - val_accuracy: 0.8100 - val_loss: 0.3576
Epoch 4/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.8832 - loss: 0.2839 - val_accuracy: 0.8300 - val_loss: 0.3400
Epoch 5/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - accuracy: 0.9080 - loss: 0.2402 - val_accuracy: 0.8400 - val_loss: 0.3746
Epoch 6/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - accuracy: 0.8963 - loss: 0.2338 - val_accuracy: 0.8800 - val_loss: 0.3341
Epoch 7/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - accuracy: 0.9356 - loss: 0.1795 - val_accuracy: 0.8600 - val_loss: 0.3774
Epoch 8/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - accuracy: 0.9312 - loss: 0.1757 - val_accuracy: 0.8600 - val_loss: 0.3759
Epoch 9/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 15ms/step - accuracy: 0.9569 - loss: 0.1533 - val_accuracy: 0.8800 - val_loss: 0.3439
Epoch 10/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step - accuracy: 0.9626 - loss: 0.1089 - val_accuracy: 0.8700 - val_loss: 0.3428
Epoch 11/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step - accuracy: 0.9623 - loss: 0.1091 - val_accuracy: 0.8500 - val_loss: 0.3201
Epoch 12/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.9427 - loss: 0.1313 - val_accuracy: 0.8500 - val_loss: 0.4833
Epoch 13/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 13ms/step - accuracy: 0.9528 - loss: 0.1148 - val_accuracy: 0.8600 - val_loss: 0.3970
Epoch 14/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - accuracy: 0.9850 - loss: 0.0602 - val_accuracy: 0.8700 - val_loss: 0.5414
Epoch 15/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 15ms/step - accuracy: 0.9888 - loss: 0.0413 - val_accuracy: 0.8400 - val_loss: 0.4738
Epoch 16/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - accuracy: 0.9879 - loss: 0.0349 - val_accuracy: 0.8500 - val_loss: 0.4821
Epoch 17/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - accuracy: 0.9788 - loss: 0.0616 - val_accuracy: 0.8000 - val_loss: 0.8049
Epoch 18/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 15ms/step - accuracy: 0.9516 - loss: 0.0999 - val_accuracy: 0.8300 - val_loss: 0.6529
Epoch 19/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.9782 - loss: 0.0639 - val_accuracy: 0.8500 - val_loss: 0.6369
Epoch 20/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step - accuracy: 0.9895 - loss: 0.0353 - val_accuracy: 0.8100 - val_loss: 0.7740
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 112ms/step
```

 SimpleRNN(64, activation='relu', input_shape=(X_train.shape[1], 1), return_sequences=True):

- Đây là lớp RNN với 64 đơn vị RNN và hàm kích hoạt ReLU.
- input_shape=(X_train.shape[1], 1) chỉ định kích thước đầu vào.
- return_sequences=True cho phép lớp RNN này trả về toàn bộ chuỗi đầu ra, cần thiết khi có nhiều hơn một lớp RNN trong mô hình.

 SimpleRNN(64, activation='relu', return_sequences=True):

- Thêm một lớp RNN khác với 64 đơn vị RNN và trả về toàn bộ chuỗi đầu ra (return_sequences=True).
- Việc xếp chồng nhiều lớp RNN giúp mô hình học được các mối liên hệ phức tạp hơn trong dữ liệu tuần tự.

 SimpleRNN(64, activation='relu'):

- Thêm một lớp RNN cuối cùng trong mô hình với 64 đơn vị RNN, nhưng không cần return_sequences=True vì đây là lớp RNN cuối.

 Dense(100, activation='relu'):

- Lớp fully-connected với 100 neuron, giúp tăng tính phi tuyến của mô hình.

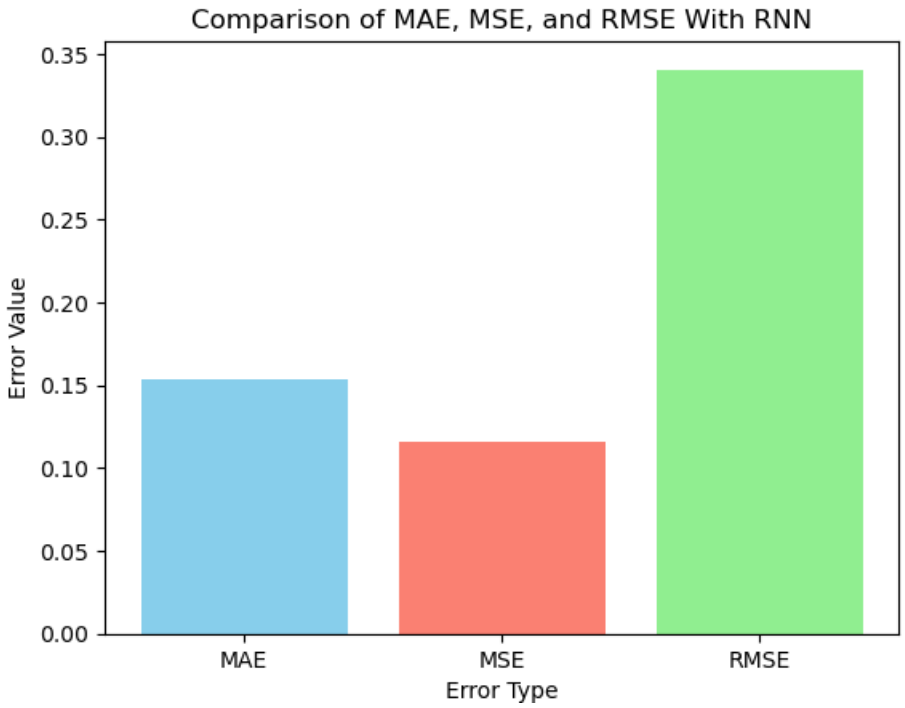
 Dense(50, activation='relu'):

- Lớp fully-connected khác với 50 neuron.

 Dense(1, activation='sigmoid'):

- Lớp đầu ra cho bài toán phân loại nhị phân.

```
: # Tạo dữ liệu cho biểu đồ
errors = [rnn_mae, rnn_mse, rnn_rmse]
labels = ['MAE', 'MSE', 'RMSE']
colors = ['skyblue', 'salmon', 'lightgreen']

# Vẽ biểu đồ
plt.bar(labels, errors, color=colors)
plt.title('Comparison of MAE, MSE, and RMSE With RNN')
plt.xlabel('Error Type')
plt.ylabel('Error Value')
plt.show()
```



1.4.3. LSTM

```python
from tensorflow.keras.layers import LSTM

# Mô hình LSTM
lstm_model = Sequential()
lstm_model.add(LSTM(64, activation='relu', input_shape=(X_train.shape[1], 1), return_sequences=True))
lstm_model.add(LSTM(64, activation='relu', return_sequences=True))
lstm_model.add(LSTM(64, activation='relu'))
lstm_model.add(Dense(100, activation='relu'))
lstm_model.add(Dense(50, activation='relu'))
lstm_model.add(Dense(1, activation='sigmoid'))

# Compile và huấn luyện
lstm_model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
lstm_history = lstm_model.fit(X_train.reshape(-1, X_train.shape[1], 1), y_train,
                              epochs=20, batch_size=32, validation_data=(X_val.reshape(-1, X_val.shape[1], 1), y_val))

# Đánh giá trên tập test
y_lstm_pred = lstm_model.predict(X_test.reshape(-1, X_test.shape[1], 1)).flatten()
lstm_mae = mean_absolute_error(y_test, y_lstm_pred)
lstm_mse = mean_squared_error(y_test, y_lstm_pred)
lstm_rmse = np.sqrt(lstm_mse)
```

```
25/25 ━━━━━━━━━━━━━━━━━━━━ 5s 43ms/step - accuracy: 0.4892 - loss: 0.6934 - val_accuracy: 0.4700 - val_loss: 0.6939
Epoch 2/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 22ms/step - accuracy: 0.5371 - loss: 0.6919 - val_accuracy: 0.4700 - val_loss: 0.6922
Epoch 3/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 25ms/step - accuracy: 0.5716 - loss: 0.6877 - val_accuracy: 0.6100 - val_loss: 0.6649
Epoch 4/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.5692 - loss: 0.6691 - val_accuracy: 0.6000 - val_loss: 0.6535
Epoch 5/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.5608 - loss: 0.6657 - val_accuracy: 0.6300 - val_loss: 0.6501
Epoch 6/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.6320 - loss: 0.6481 - val_accuracy: 0.5900 - val_loss: 0.6532
Epoch 7/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.6242 - loss: 0.6436 - val_accuracy: 0.6400 - val_loss: 0.6268
Epoch 8/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 18ms/step - accuracy: 0.5884 - loss: 0.6528 - val_accuracy: 0.6300 - val_loss: 0.6370
Epoch 9/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 20ms/step - accuracy: 0.6417 - loss: 0.6342 - val_accuracy: 0.6500 - val_loss: 0.6318
Epoch 10/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 18ms/step - accuracy: 0.6096 - loss: 0.6350 - val_accuracy: 0.6400 - val_loss: 0.6320
Epoch 11/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 26ms/step - accuracy: 0.6204 - loss: 0.6318 - val_accuracy: 0.6800 - val_loss: 0.6258
Epoch 12/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 17ms/step - accuracy: 0.6481 - loss: 0.6114 - val_accuracy: 0.6500 - val_loss: 0.6118
Epoch 13/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 17ms/step - accuracy: 0.6483 - loss: 0.6036 - val_accuracy: 0.6800 - val_loss: 0.6193
Epoch 14/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 24ms/step - accuracy: 0.6928 - loss: 0.5732 - val_accuracy: 0.6000 - val_loss: 0.6021
Epoch 15/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 32ms/step - accuracy: 0.6793 - loss: 0.5884 - val_accuracy: 0.6800 - val_loss: 0.6350
Epoch 16/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 31ms/step - accuracy: 0.6888 - loss: 0.5914 - val_accuracy: 0.6700 - val_loss: 0.5971
Epoch 17/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 29ms/step - accuracy: 0.6897 - loss: 0.5940 - val_accuracy: 0.6800 - val_loss: 0.5913
Epoch 18/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 31ms/step - accuracy: 0.6620 - loss: 0.5863 - val_accuracy: 0.6800 - val_loss: 0.5854
Epoch 19/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 27ms/step - accuracy: 0.7138 - loss: 0.5548 - val_accuracy: 0.6500 - val_loss: 0.5878
Epoch 20/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 24ms/step - accuracy: 0.7005 - loss: 0.5633 - val_accuracy: 0.7100 - val_loss: 0.5886
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 132ms/step
```

 LSTM(64, activation='relu', input_shape=(X_train.shape[1], 1), return_sequences=True):

- Đây là lớp LSTM đầu tiên với 64 đơn vị và hàm kích hoạt ReLU.
- input_shape=(X_train.shape[1], 1) chỉ định kích thước đầu vào.
- return_sequences=True để trả về toàn bộ chuỗi đầu ra, cần thiết khi có nhiều lớp LSTM.

 LSTM(64, activation='relu', return_sequences=True):

- Thêm một lớp LSTM với 64 đơn vị và trả về toàn bộ chuỗi đầu ra (return_sequences=True).
- Việc xếp chồng nhiều lớp LSTM giúp mô hình học được các mối liên hệ dài hạn trong dữ liệu.

 LSTM(64, activation='relu'):

- Thêm một lớp LSTM cuối cùng với 64 đơn vị, không cần return_sequences=True vì đây là lớp LSTM cuối trong mô hình.

 Dense(100, activation='relu'):

- Thêm một lớp fully-connected với 100 neuron, giúp tăng khả năng học của mô hình.

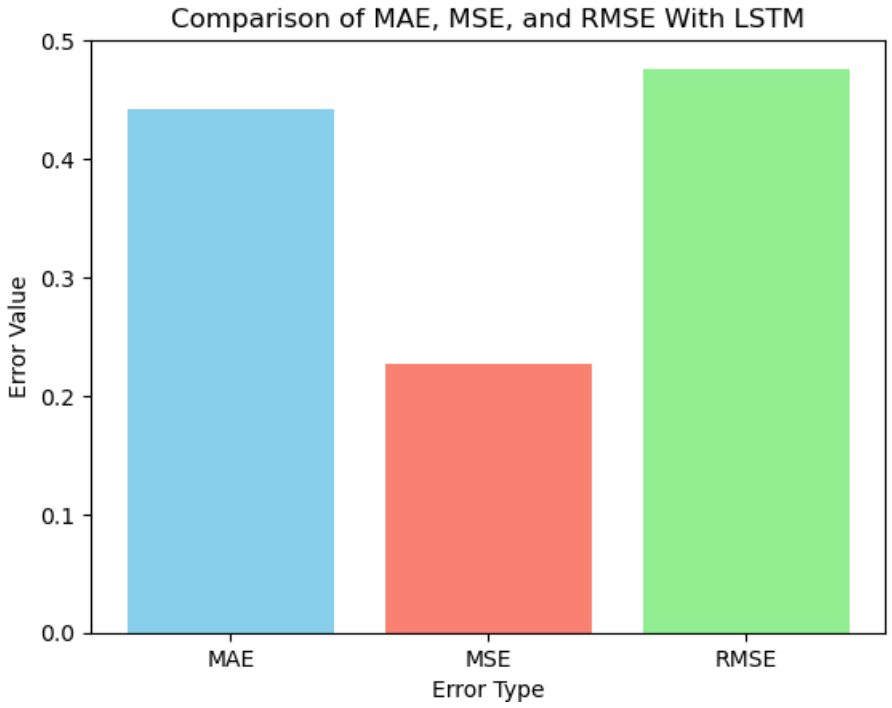
 Dense(50, activation='relu'):

- Lớp fully-connected khác với 50 neuron.

 Dense(1, activation='sigmoid'):

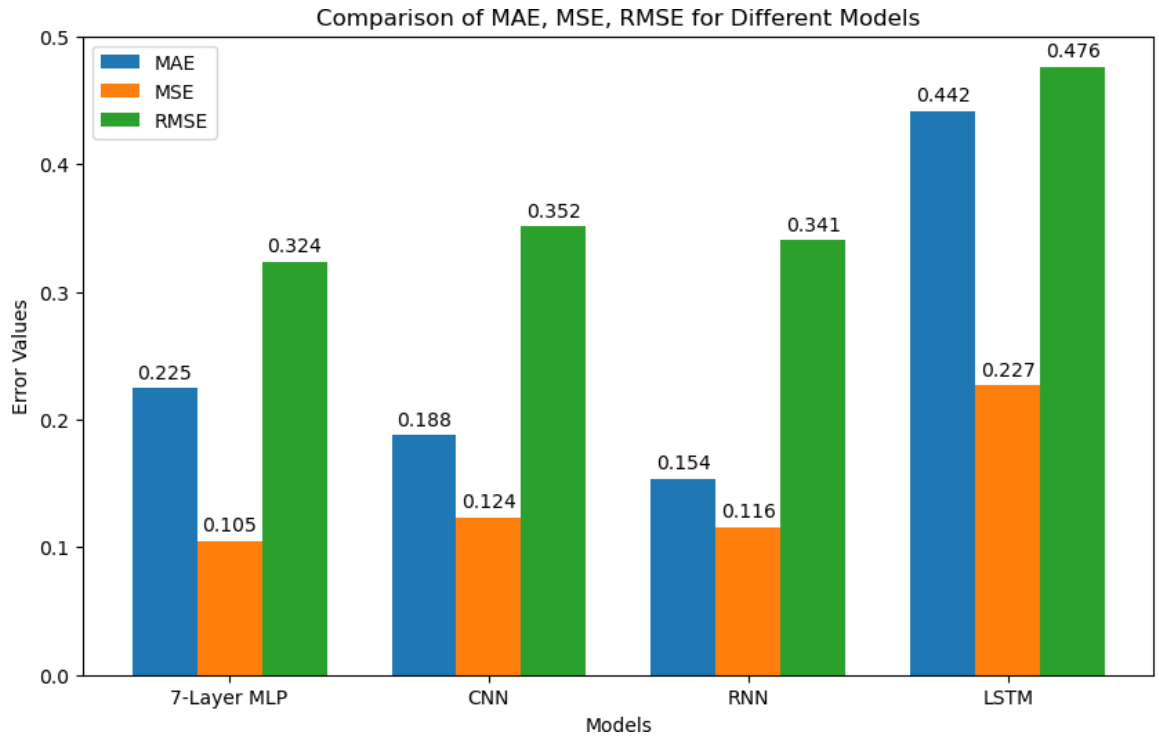
- Lớp đầu ra cho bài toán phân loại nhị phân.

```
# Tạo dữ liệu cho biểu đồ
errors = [lstm_mae, lstm_mse, lstm_rmse]
labels = ['MAE', 'MSE', 'RMSE']
colors = ['skyblue', 'salmon', 'lightgreen']

# Vẽ biểu đồ
plt.bar(labels, errors, color=colors)
plt.title('Comparison of MAE, MSE, and RMSE With LSTM')
plt.xlabel('Error Type')
plt.ylabel('Error Value')
plt.show()
```



1.4.4. So sánh

```python
import matplotlib.pyplot as plt
import numpy as np

# Giá trị MAE, MSE, RMSE của từng mô hình (giả định các giá trị này đã tính ở trên)
mae_values = [mae, cnn_mae, rnn_mae, lstm_mae]
mse_values = [mse, cnn_mse, rnn_mse, lstm_mse]
rmse_values = [rmse, cnn_rmse, rnn_rmse, lstm_rmse]

# Tên các mô hình
models = ['7-Layer MLP', 'CNN', 'RNN', 'LSTM']

# Thiết lập biểu đồ
x = np.arange(len(models))  # vị trí các cột
width = 0.25  # độ rộng của mỗi cột

# Tạo biểu đồ
fig, ax = plt.subplots(figsize=(10, 6))
rects1 = ax.bar(x - width, mae_values, width, label='MAE')
rects2 = ax.bar(x, mse_values, width, label='MSE')
rects3 = ax.bar(x + width, rmse_values, width, label='RMSE')

# Thêm nhãn và tiêu đề
ax.set_xlabel('Models')
ax.set_ylabel('Error Values')
ax.set_title('Comparison of MAE, MSE, RMSE for Different Models')
ax.set_xticks(x)
ax.set_xticklabels(models)
ax.legend()

# Hiển thị giá trị trên cột
def add_labels(rects):
    for rect in rects:
        height = rect.get_height()
        ax.annotate(f'{height:.3f}',
                    xy=(rect.get_x() + rect.get_width() / 2, height),
                    xytext=(0, 3),  # khoảng cách văn bản so với đỉnh cột
                    textcoords="offset points",
                    ha='center', va='bottom')

add_labels(rects1)
add_labels(rects2)
add_labels(rects3)

plt.show()
```



7-layer MLP:

- MAE: 0.225, MSE: 0.105, RMSE: 0.324.
- Mô hình MLP (Multi-Layer Perceptron) có độ lỗi khá thấp, cho thấy khả năng dự đoán tương đối tốt trên dữ liệu này.
- Tuy nhiên, vì MLP không có các cơ chế nắm bắt mối quan hệ tuần tự, nó có thể không tốt như CNN hoặc RNN/LSTM đối với dữ liệu có tính chất thời gian hoặc không gian.

CNN:

- MAE: 0.188, MSE: 0.124, RMSE: 0.352.
- CNN (Convolutional Neural Network) có hiệu suất tốt hơn MLP, với MAE và RMSE đều thấp hơn so với MLP.

- Mô hình CNN hoạt động tốt trong việc nắm bắt các đặc trưng cục bộ, thường phù hợp với dữ liệu có tính chất không gian. Điều này có thể giải thích lý do CNN cho kết quả tốt hơn MLP trên dữ liệu này.

RNN:

- MAE: 0.154, MSE: 0.116, RMSE: 0.341.
- Mô hình RNN (Recurrent Neural Network) cho kết quả tốt hơn MLP và CNN, với MAE thấp nhất trong ba mô hình đầu tiên. Điều này cho thấy RNN có khả năng nắm bắt mối quan hệ tuần tự trong dữ liệu tốt hơn.
- RNN thường được sử dụng cho dữ liệu tuần tự, như chuỗi thời gian hoặc dữ liệu có tính liên kết về mặt thứ tự. Do đó, nếu dữ liệu có đặc trưng thời gian, RNN có thể tận dụng tốt hơn so với MLP và CNN.

LSTM:

- MAE: 0.227, MSE: 0.442, RMSE: 0.476.
- LSTM (Long Short-Term Memory) có hiệu suất thấp nhất trong bốn mô hình với MAE, MSE, và RMSE cao nhất. Điều này có thể do cấu trúc của LSTM có xu hướng phức tạp hơn và thường yêu cầu nhiều dữ liệu để huấn luyện hiệu quả.
- Tuy nhiên, nếu dữ liệu có mối liên hệ dài hạn, LSTM có thể sẽ hoạt động tốt hơn RNN và các mô hình khác. Trong trường hợp này, dữ liệu có thể không đủ dài hạn hoặc phức tạp để phát huy sức mạnh của LSTM, dẫn đến hiệu suất thấp hơn.