In [1]:
```python
import numpy as np
import matplotlib.pyplot as plt
import random
```

In [2]:
```python
import tensorflow as tf
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Dense, Dropout, LSTM, Attention, Concatenate
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras import layers, models
from tensorflow.keras.optimizers import Adam
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, roc_curve, auc
from sklearn.metrics import classification_report, confusion_matrix, ConfusionMatrixDisplay
```

In [3]:
```python
def set_seeds(seed):
    np.random.seed(seed)
    random.seed(seed)
    tf.random.set_seed(seed)

# Call this function before creating and training your model
set_seeds(42)
```

In [4]:
```python
# CONSTANT
LEARNING_RATE_VAL = 0.0001
BATCH_SIZE = 32
VALIDATION_SPLIT_VAL = 0.2
EPOCH_NUMBER = 50
```

In [5]:
```python
early_stopping = EarlyStopping(
    monitor='val_loss',        # Metric to monitor
    patience=1,               # Number of epochs to wait for improvement
    restore_best_weights=True # Restore model weights from the epoch with the best value
)
```

load dataset

In [6]:
```python
def load_np_array(file_name):
    X_array = np.load('data/X_' + file_name + '_array.npy')
    y_array = np.load('data/y_' + file_name + '_array.npy')
```

```python
    return X_array, y_array

X_train_fall, y_train_fall = load_np_array("train_fall")
X_train_notfall, y_train_notfall = load_np_array("train_notfall")
X_test_fall, y_test_fall = load_np_array("test_fall")
X_test_notfall, y_test_notfall = load_np_array("test_notfall")

# Combine fall and non-fall data for training
X_train = np.concatenate((X_train_fall, X_train_notfall), axis=0)
y_train = np.concatenate((y_train_fall, y_train_notfall), axis=0)

# Combine fall and non-fall data for testing
X_test = np.concatenate((X_test_fall, X_test_notfall), axis=0)
y_test = np.concatenate((y_test_fall, y_test_notfall), axis=0)

# Shuffle the training data
train_indices = np.arange(X_train.shape[0])
np.random.shuffle(train_indices)
X_train = X_train[train_indices]
y_train = y_train[train_indices]

# Shuffle the testing data
test_indices = np.arange(X_test.shape[0])
np.random.shuffle(test_indices)
X_test = X_test[test_indices]
y_test = y_test[test_indices]

print("Combined training data shape:", X_train.shape, y_train.shape)
print("Combined testing data shape:", X_test.shape, y_test.shape)
```

```
Combined training data shape: (5824, 40, 3) (5824,)
Combined testing data shape: (2912, 40, 3) (2912,)
```

In [7]:

```python
class TFTModel(tf.keras.Model):
    def __init__(self, input_shape, num_features, num_heads, num_units):
        super(TFTModel, self).__init__()
        # LSTM Layer
        self.lstm = layers.LSTM(num_units, return_sequences=True, input_shape=input_shape)
        # Multi-Head Attention Layer
        self.attention = layers.MultiHeadAttention(num_heads=num_heads, key_dim=num_units)
        # Gated Mechanism
        self.gate = layers.Dense(num_units, activation='sigmoid')
```

```
        # Dense Layers
        self.dense1 = layers.Dense(32, activation='relu')
        self.dropout_dense1 = layers.Dropout(0.5)
        # Output Layer
        self.output_layer = layers.Dense(1, activation='sigmoid')  # Binary classification

    def call(self, inputs, training=False):
        # LSTM layer
        x = self.lstm(inputs)
        # Attention layer
        attn_output = self.attention(x, x)
        # Gated residual connections
        gated_output = self.gate(attn_output) * attn_output
        # Dense Layers
        x = tf.reduce_mean(gated_output, axis=1)  # Global average pooling
        x = self.dense1(x)
        x = self.dropout_dense1(x, training=training)
        # Output layer
        return self.output_layer(x)

# Define model parameters
input_shape = (X_train.shape[1], X_train.shape[2]) # (40, 3)
num_features = X_train.shape[2]
num_heads = 4
num_units = 256  # Number of LSTM units

# Instantiate and compile the model
model = TFTModel(input_shape=input_shape, num_features=num_features, num_heads=num_heads, num_units=num_uni
model.compile(optimizer=Adam(learning_rate=LEARNING_RATE_VAL), loss='binary_crossentropy', metrics=['accura
```

```
2024-08-05 00:03:52.578245: I metal_plugin/src/device/metal_device.cc:1154] Metal device set to: Apple M1 P
ro
2024-08-05 00:03:52.578266: I metal_plugin/src/device/metal_device.cc:296] systemMemory: 16.00 GB
2024-08-05 00:03:52.578272: I metal_plugin/src/device/metal_device.cc:313] maxCacheSize: 5.33 GB
2024-08-05 00:03:52.578286: I tensorflow/core/common_runtime/pluggable_device/pluggable_device_factory.cc:3
05] Could not identify NUMA node of platform GPU ID 0, defaulting to 0. Your kernel may not have been built
with NUMA support.
2024-08-05 00:03:52.578298: I tensorflow/core/common_runtime/pluggable_device/pluggable_device_factory.cc:2
71] Created TensorFlow device (/job:localhost/replica:0/task:0/device:GPU:0 with 0 MB memory) -> physical P
luggableDevice (device: 0, name: METAL, pci bus id: <undefined>)
/opt/miniconda3/envs/tensorflow/lib/python3.10/site-packages/keras/src/layers/rnn/rnn.py:204: UserWarning:
Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an
`Input(shape)` object as the first layer in the model instead.
  super().__init__(**kwargs)
```

In [8]:

```
# Train the model
history = model.fit(X_train, y_train, epochs=EPOCH_NUMBER, batch_size=BATCH_SIZE, validation_split=VALIDATI
```

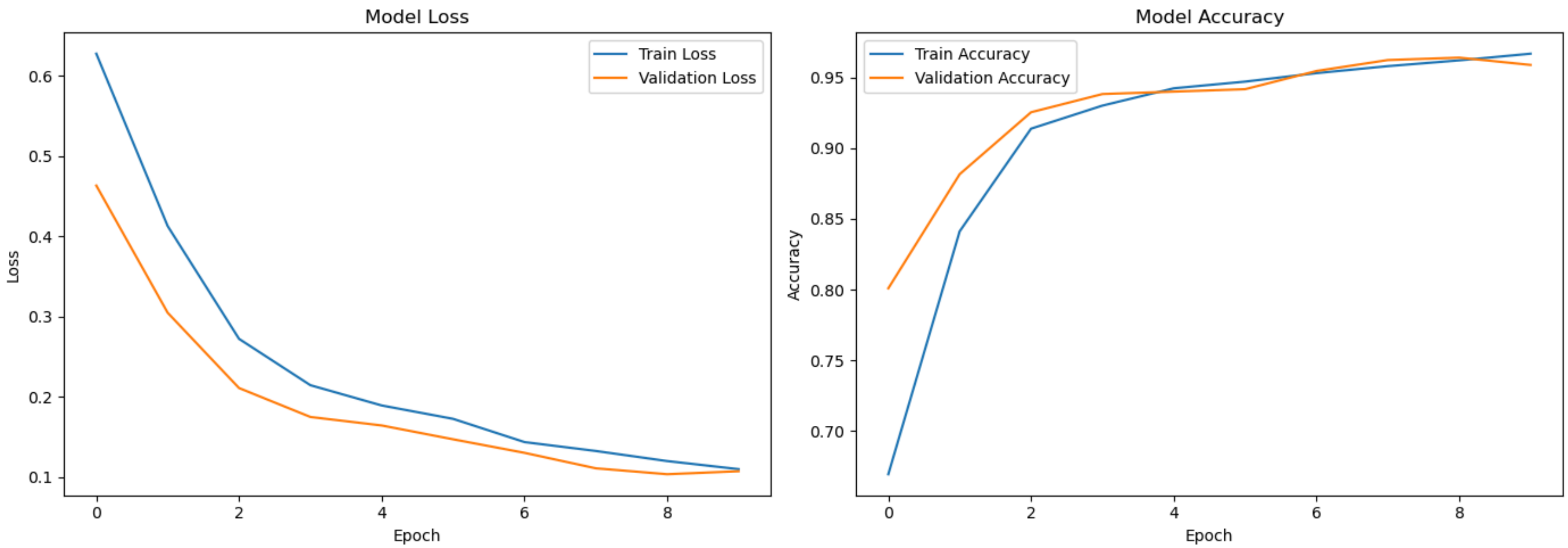
```
Epoch 1/50
```

```
2024-08-05 00:03:53.464315: I tensorflow/core/grappler/optimizers/custom_graph_optimizer_registry.cc:117] P
lugin optimizer for device_type GPU is enabled.
```

```
146/146 ━━━━━━━━━━━━━━━━━━━━ 7s 30ms/step - accuracy: 0.6205 - loss: 0.6688 - val_accuracy: 0.8009 - val_lo
ss: 0.4632
Epoch 2/50
146/146 ━━━━━━━━━━━━━━━━━━━━ 4s 26ms/step - accuracy: 0.8175 - loss: 0.4564 - val_accuracy: 0.8815 - val_lo
ss: 0.3048
Epoch 3/50
146/146 ━━━━━━━━━━━━━━━━━━━━ 4s 26ms/step - accuracy: 0.9052 - loss: 0.2958 - val_accuracy: 0.9253 - val_lo
ss: 0.2110
Epoch 4/50
146/146 ━━━━━━━━━━━━━━━━━━━━ 4s 27ms/step - accuracy: 0.9288 - loss: 0.2217 - val_accuracy: 0.9382 - val_lo
ss: 0.1750
Epoch 5/50
146/146 ━━━━━━━━━━━━━━━━━━━━ 4s 27ms/step - accuracy: 0.9392 - loss: 0.1988 - val_accuracy: 0.9399 - val_lo
ss: 0.1644
Epoch 6/50
146/146 ━━━━━━━━━━━━━━━━━━━━ 4s 27ms/step - accuracy: 0.9448 - loss: 0.1826 - val_accuracy: 0.9416 - val_lo
ss: 0.1472
Epoch 7/50
146/146 ━━━━━━━━━━━━━━━━━━━━ 4s 29ms/step - accuracy: 0.9496 - loss: 0.1498 - val_accuracy: 0.9545 - val_lo
ss: 0.1303
Epoch 8/50
146/146 ━━━━━━━━━━━━━━━━━━━━ 4s 30ms/step - accuracy: 0.9577 - loss: 0.1315 - val_accuracy: 0.9622 - val_lo
ss: 0.1110
Epoch 9/50
146/146 ━━━━━━━━━━━━━━━━━━━━ 6s 38ms/step - accuracy: 0.9630 - loss: 0.1174 - val_accuracy: 0.9639 - val_lo
ss: 0.1037
Epoch 10/50
146/146 ━━━━━━━━━━━━━━━━━━━━ 6s 40ms/step - accuracy: 0.9676 - loss: 0.1145 - val_accuracy: 0.9588 - val_lo
ss: 0.1074
```

In [9]:
```python
# Plot training & validation loss values
plt.figure(figsize=(14, 5))

plt.subplot(1, 2, 1)
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Model Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend(loc='upper right')

# Plot training & validation accuracy values
```

```python
plt.subplot(1, 2, 2)
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Model Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(loc='upper left')

plt.tight_layout()
plt.show()
```



In [10]:

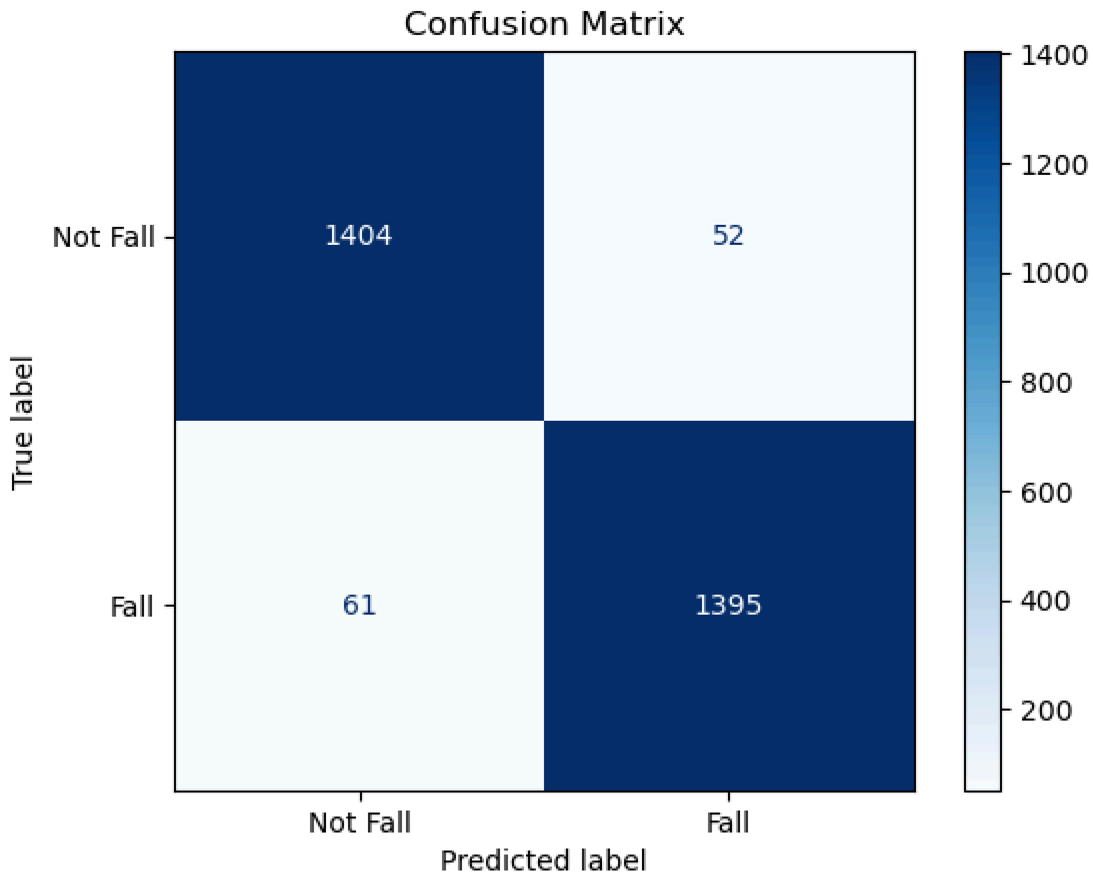
```python
# Generate predictions
y_pred_proba = model.predict(X_test)
y_pred = (y_pred_proba > 0.5).astype(int).flatten()

# Calculate evaluation metrics
accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)

print(f"Accuracy: {accuracy:.4f}")
print(f"Precision: {precision:.4f}")
```

```
print(f"Recall: {recall:.4f}")
print(f"F1 Score: {f1:.4f}")

# Generate classification report
print("\nClassification Report:\n", classification_report(y_test, y_pred, target_names=['Not Fall', 'Fall']
```

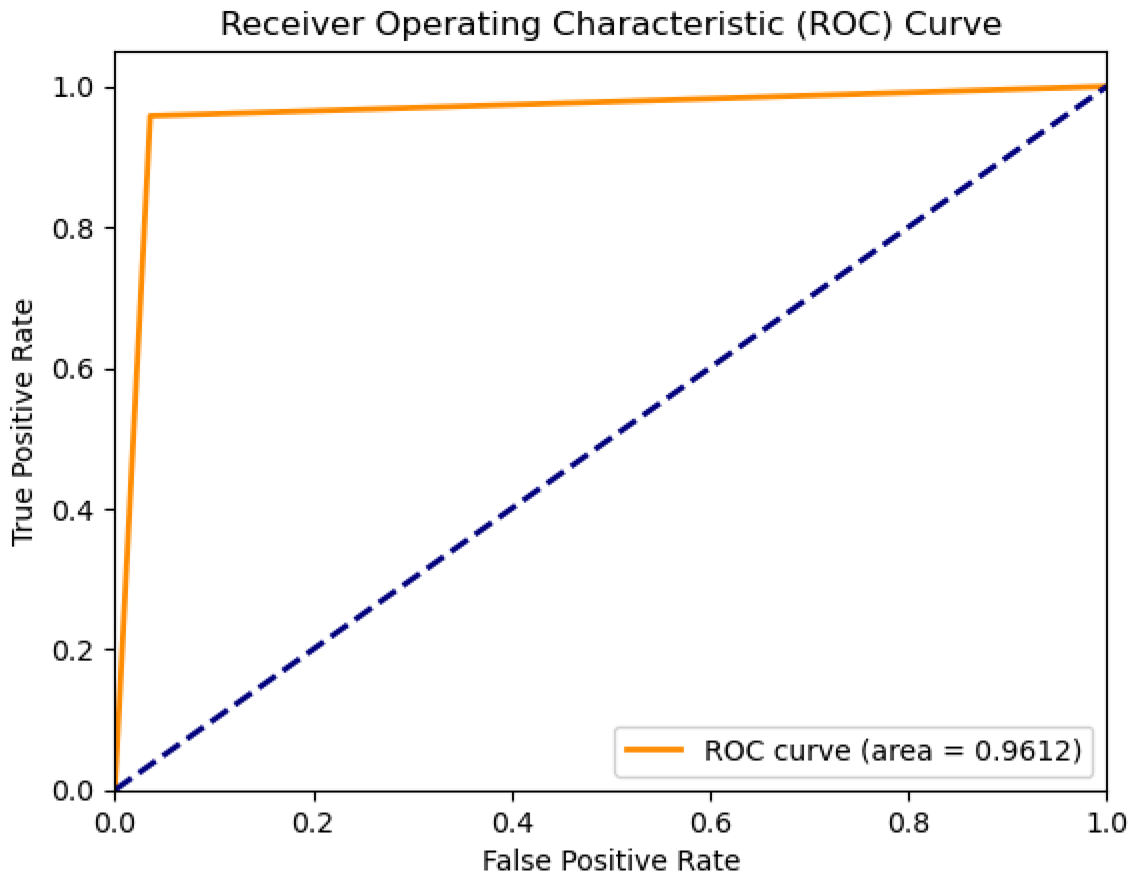
```
91/91 ━━━━━━━━━━━━━━━━━━━━ 1s 8ms/step
Accuracy: 0.9612
Precision: 0.9641
Recall: 0.9581
F1 Score: 0.9611

Classification Report:
               precision    recall  f1-score   support

    Not Fall       0.96      0.96      0.96      1456
        Fall       0.96      0.96      0.96      1456

    accuracy                           0.96      2912
   macro avg       0.96      0.96      0.96      2912
weighted avg       0.96      0.96      0.96      2912
```

In [11]:

```
# Compute and plot the confusion matrix
cm = confusion_matrix(y_test, y_pred)
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=['Not Fall', 'Fall'])
disp.plot(cmap=plt.cm.Blues)
plt.title('Confusion Matrix')
plt.show()
```



In [12]:
```python
# Compute ROC curve and AUC
fpr, tpr, thresholds = roc_curve(y_test, y_pred)
roc_auc = auc(fpr, tpr)

plt.figure()
plt.plot(fpr, tpr, color='darkorange', lw=2, label=f'ROC curve (area = {roc_auc:.4f})')
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve')
plt.legend(loc="lower right")
plt.show()
```


Receiver Operating Characteristic (ROC) Curve
True Positive Rate
False Positive Rate
ROC curve (area = 0.9612)
0.0
0.2
0.4
0.6
0.8
1.0

```
In [ ]:
```