Fall Detection System Using IMU Sensors - Technical Report

Executive Summary

This report presents a comprehensive machine learning solution for fall detection using Inertial Measurement Unit (IMU) sensor data. The system achieved a peak performance of **94.34% F1-Score** using a Transformer architecture on the general dataset without PCA, demonstrating robust capability in distinguishing between Activities of Daily Living (ADLs), Falls, and Near-Falls. The project implemented three distinct deep learning architectures (LSTM, GRU, Transformer), evaluated dimensionality reduction techniques, and analyzed subject-specific versus general model approaches.

1. Methodology Overview

1.1 Problem Definition

The challenge required developing a classification system for time-series data collected from 7 body-worn IMU sensors, categorizing movements into three classes:

- **Class 0**: Activities of Daily Living (ADLs)
- **Class 1**: Falls
- **Class 2**: Near-Falls

1.2 Data Pipeline Architecture

```
python

Raw Data → Sequence Creation → Optional PCA → Train/Val/Test Split → Model Training → Evaluation
```

2. Data Preprocessing and Feature Engineering

2.1 Data Loading Strategy

The system implements a controlled data loading approach to balance computational efficiency with model performance:

```python
def load_subject_data(self, subject_folder: str, max_files_per_category: int = 4):
    categories = {
        'ADLs': 0,
        'Falls': 1,
        'Near_Falls': 2
    }
    # Load 4 files per category × 3 categories = 12 files per subject
    # Total: 96 files across 8 subjects for general model
```

Rationale: Loading 4 files per category ensures sufficient data diversity while maintaining manageable training times. This approach captures approximately 20% of available data, which proved sufficient for achieving >90% accuracy.

2.2 Sequence Generation with Sliding Windows

The system transforms continuous sensor readings into fixed-length sequences suitable for time-series models:

```python
def create_sequences(self, df: pd.DataFrame, sequence_length: int = 50):
    # Create sliding windows with 50% overlap
    for i in range(0, len(data) - sequence_length + 1, sequence_length // 2):
        sequences.append(data[i:i + sequence_length])
```

Key Parameters:

- **Sequence Length**: 50 timesteps (~0.39 seconds at 128Hz sampling rate)
- **Overlap**: 50% to increase training samples and capture transitional movements
- **Result**: 7,408 sequences from 96 files for the general model

Rationale: The 50-timestep window captures sufficient temporal context for fall detection while maintaining computational efficiency. The 50% overlap ensures critical transition moments aren't missed between windows.

2.3 Feature Composition

Each timestep contains 63 features from 7 sensor locations:

```
7 locations × 3 sensor types × 3 axes = 63 features

Sensor Locations: [r.ankle, l.ankle, r.thigh, l.thigh, head, sternum, waist]
Sensor Types: [Acceleration (m/s²), Angular Velocity (rad/s), Magnetic Field (μT)]
Axes: [X, Y, Z]
```

2.4 Data Normalization

All sensor data undergoes standardization before PCA or model training:

```python
scaler = StandardScaler()
data_scaled = scaler.fit_transform(data)
```

This ensures all features contribute equally to the learning process, preventing features with larger scales from dominating.

3. Dimensionality Reduction Analysis

3.1 PCA Implementation

Principal Component Analysis (PCA) was applied to reduce computational complexity while preserving information:

```python
pca = PCA(n_components=30)
data_transformed = pca.fit_transform(data_scaled)
# Reduced from 63 → 30 features
# Explained variance: 99.5%
```

3.2 PCA Impact Analysis

Metric	Without PCA	With PCA	Impact
Features	63	30	-52.4%
Training Time (LSTM)	916.83s	560.69s	**-38.8%**
Training Time (GRU)	1043.03s	702.94s	**-32.6%**
Training Time (Transformer)	377.22s	211.57s	**-43.9%**
Average F1-Score	0.9302	0.9207	**-1.0%**

Key Finding: PCA provides a highly favorable trade-off, reducing training time by approximately **38%** while maintaining accuracy above **92%** with only a **1% performance loss**.

4. Model Architecture and Selection

4.1 LSTM (Long Short-Term Memory)

```python
```

```
def create_lstm_model(self, input_shape, num_classes=3):
    model = Sequential([
        LSTM(128, return_sequences=True),
        BatchNormalization(),
        Dropout(0.3),
        LSTM(64, return_sequences=True),
        BatchNormalization(),
        Dropout(0.3),
        LSTM(32),
        Dense(64, activation='relu'),
        Dropout(0.2),
        Dense(num_classes, activation='softmax')
    ])
```

Architecture Rationale:

- Three LSTM layers (128→64→32) for hierarchical feature extraction
- Batch normalization for training stability
- Dropout (0.3, 0.2) for regularization
- Progressive dimension reduction captures both high and low-level temporal patterns

4.2 GRU (Gated Recurrent Unit)

Similar architecture to LSTM but with fewer parameters:

- **Advantage**: 25% fewer parameters than LSTM
- **Performance**: Comparable to LSTM (93.23% vs 91.49% F1-Score)
- **Training**: Slightly slower than expected due to implementation overhead

4.3 Transformer

python

```
def create_transformer_model(self, input_shape, num_classes=3):
    # Multi-head attention with 4 heads
    attn_output = MultiHeadAttention(num_heads=4, key_dim=input_shape[1]//4)
    # 2 transformer blocks for deep feature extraction
    # Global average pooling for sequence aggregation
```

Architecture Advantages:

- Parallel processing of sequences
- Global context awareness through self-attention
- **Best Performance**: 94.34% F1-Score

- **Fastest Training**: 377.22s (2.4× faster than LSTM)

5. Training Strategy and Optimization

5.1 Data Splitting

```
python

# Stratified split maintaining class distribution
Train: 64% (4,740 samples)
Validation: 16% (1,186 samples)
Test: 20% (1,482 samples)

# Class distribution maintained:
ADLs: ~35%, Falls: ~32%, Near-Falls: ~32%
```

5.2 Class Imbalance Handling

```
python

def calculate_class_weights(self, y_train):
    class_weights = compute_class_weight('balanced',
                                         classes=unique_classes,
                                         y=y_train_labels)
```

Automated class weighting ensures the model doesn't bias toward majority classes.

5.3 Hyperparameter Optimization

Learning Rate Scheduling:

```
python

ReduceLROnPlateau(monitor='val_loss', factor=0.5, patience=5)
# Reduces learning rate by 50% when validation loss plateaus
```

Early Stopping:

```
python

EarlyStopping(monitor='val_loss', patience=15, restore_best_weights=True)
# Prevents overfitting by stopping when no improvement for 15 epochs
```

5.4 Label Encoding

One-hot encoding for multi-class classification:

```python
y_sequences_cat = to_categorical(y_sequences, num_classes=3)
# Converts [0,1,2] → [[1,0,0], [0,1,0], [0,0,1]]
```

6. Evaluation Metrics Selection

6.1 Why F1-Score as Primary Metric

The F1-Score was chosen as the primary evaluation metric for several critical reasons:

1. **Balanced Metric**: F1-Score is the harmonic mean of precision and recall, providing a balanced measure that considers both false positives and false negatives.
2. **Medical Context Relevance**: In fall detection:
   - **False Negatives** (missed falls) are dangerous - could result in delayed medical attention
   - **False Positives** (false alarms) reduce system trust and cause alert fatigue
   - F1-Score penalizes both types of errors equally
3. **Class Imbalance Robustness**: Unlike accuracy, F1-Score remains informative even with slight class imbalances.
4. **Mathematical Foundation**:

```
F1 = 2 × (Precision × Recall) / (Precision + Recall)
```

7. Subject-Specific Analysis

7.1 Performance Variability

Comparing Subject 1 vs Subject 2 reveals individual movement pattern impacts:

Model	Subject 1 F1	Subject 2 F1	Difference	Implication
LSTM	0.9833	0.9497	3.4%	Moderate variability
GRU	0.9944	0.9610	3.3%	Consistent with LSTM
Transformer	0.9553	0.9722	1.7%	More robust to individual differences

Key Insights:

- Individual movement patterns affect model accuracy by 1.7-3.4%
- Transformer shows better generalization across subjects
- Subject-specific models achieve near-perfect accuracy (>95%) but lack generalization

7.2 General vs Subject-Specific Models

Approach	Pros	Cons	F1-Score Range
General Model	Better generalization, Works for new users, More robust	Lower individual accuracy	91-94%
Subject-Specific	Higher accuracy, Personalized detection	Requires individual training, Poor generalization	95-99%

Recommendation: Deploy general model initially, then fine-tune with user-specific data after 1-2 weeks of usage.

8. Results Summary

8.1 Best Overall Model

Winner: Transformer (General, No PCA)

- F1-Score: 94.34%
- Training Time: 377.22s
- Inference Time: 1.39s per batch
- Accuracy: 94.33%

8.2 Complete Performance Comparison

Model	Configuration	F1-Score	Training Time	Rank
Transformer	General, No PCA	**0.9434**	377.22s	**1st**
GRU	General, No PCA	0.9323	1043.03s	2nd
GRU	General, PCA	0.9253	702.94s	3rd
LSTM	General, PCA	0.9197	560.69s	4th
Transformer	General, PCA	0.9172	211.57s	5th
LSTM	General, No PCA	0.9149	916.83s	6th

9. Confusion Matrix Analysis - Best Model

9.1 Transformer Model Confusion Matrix (General, No PCA)

```
        Predicted
        ADL  Fall  Near-Fall
Actual:
ADL     512   6     4     (98.1% correct)
```

```
Fall       1  459    20     (95.6% correct)
Near-Fall  2   51    427     (89.0% correct)
```

9.2 Error Pattern Analysis

False Positives by Class:

1. **ADLs misclassified as Falls (6 cases, 1.1%)**
   - Likely cause: Rapid movements during daily activities (sitting quickly, bending)
   - Impact: Minor - causes false alarms but no safety risk
2. **Falls misclassified as Near-Falls (20 cases, 4.2%)**
   - Likely cause: Slower falls or falls with partial recovery attempts
   - Impact: Moderate - both require attention, severity misclassification
3. **Near-Falls misclassified as Falls (51 cases, 10.6%)**
   - **Most common error**
   - Likely cause: Similar acceleration patterns during recovery phase
   - Impact: Acceptable - triggers unnecessary but precautionary alerts

Root Causes of Misclassification:

1. **Transition Ambiguity**: Near-falls and falls share initial acceleration patterns
2. **Individual Variations**: Different falling speeds across subjects
3. **Sensor Noise**: Magnetic field sensors (sternum, waist) introduce noise
4. **Edge Cases**: Activities like quick squatting resemble fall initiation

10. AI-Powered Explainability Integration

10.1 Gemini Integration Benefits

The system integrates Google Gemini AI for business-friendly explanations:

```python
def explain_prediction_with_gemini(self, sample_data, predicted_label, confidence):
    # Converts technical metrics to clinical insights
    # Example output: "High acceleration spike (15.2 m/s²) combined with
    # sudden orientation change suggests backward fall with 87% confidence"
```

Business Value:

- Translates technical metrics to actionable insights
- Provides clinical context for healthcare professionals

- Enables non-technical stakeholders to understand model decisions
- Supports audit trails for medical compliance

11. Computational Efficiency Analysis

11.1 Resource Utilization

Configuration	Memory Usage	Training Time	GPU Utilization
No PCA (63 features)	~4.2 GB	916.83s (LSTM)	85%
With PCA (30 features)	~2.1 GB	560.69s (LSTM)	72%
Improvement	**-50%**	**-38.8%**	Better efficiency

11.2 Deployment Considerations

For production deployment:

- **Cloud**: Use PCA version for cost optimization (38% less compute time)
- **Edge Devices**: PCA essential for memory-constrained environments
- **Real-time**: Transformer with PCA offers best latency (1.09s inference)

12. Recommendations and Conclusions

12.1 Final Recommendations

1. **Model Selection**: Deploy **Transformer architecture** for production
   - Best F1-Score (94.34%)
   - Fastest training (377.22s)
   - Most robust across subjects
2. **PCA Strategy**:
   - **Development/Testing**: Use without PCA for maximum accuracy
   - **Production**: Enable PCA for 38% cost reduction with minimal accuracy loss
   - **Edge Deployment**: PCA mandatory for resource constraints
3. **Deployment Approach**:
   - Start with general model for all users
   - Collect user-specific data for 1-2 weeks
   - Fine-tune personalized models for >95% accuracy
4. **Monitoring Strategy**:
   - Track false negative rate closely (missed falls are critical)
   - Implement alert fatigue monitoring for false positives

- Regular model updates with new data

12.2 Future Improvements

1. **Ensemble Methods**: Combine Transformer + GRU for potentially >95% general accuracy
2. **Adaptive Thresholds**: User-specific decision boundaries
3. **Temporal Context**: Extend sequence length for activities with longer patterns
4. **Multi-Task Learning**: Simultaneously predict fall severity and type

12.3 Conclusion

The developed fall detection system successfully achieves the challenge objectives with **94.34% F1-Score**, demonstrating that modern deep learning architectures can effectively distinguish between normal activities, falls, and near-falls using IMU sensor data. The Transformer architecture's superior performance, combined with optional PCA for resource optimization, provides a production-ready solution suitable for both cloud and edge deployments. The integration of AI explainability through Gemini ensures the system remains interpretable and actionable for healthcare professionals, bridging the gap between technical metrics and clinical decision-making.

Appendix A: Key Code Components

A.1 Complete Pipeline Flow

python

```python
# 1. Data Loading
data = load_subject_data(subject, max_files=4)

# 2. Sequence Creation
sequences = create_sequences(data, sequence_length=50)

# 3. Optional PCA
if use_pca:
    sequences = PCA(n_components=30).fit_transform(sequences)

# 4. Train/Test Split (64/16/20)
X_train, X_val, X_test = stratified_split(sequences)

# 5. Model Training with Early Stopping
model.fit(X_train, callbacks=[EarlyStopping, ReduceLROnPlateau])

# 6. Evaluation
metrics = evaluate_model(model, X_test)
```