NATIONAL INSTITUTE OF TECHNOLOGY

- WARANGAL

DEPARTMENT OF ELECTRONICS & COMMUNICATION ENGINEERING

Summer Internship May-June 2024

SPECIALIZATION: Deep Learning

"sEMG-based Lower Limb Activities Prediction with Various Deep Learning Models"

Presented by

- Sohith Pamu (N190830)

- Sirisha Bollineni (N190641)

Under Guidance of

Prof. Mohammad Farukh Hashmi

Project documentation

Abstract

In this project, we address the challenge of predicting lower limb activity by analyzing two distinct signals: Channel 3 ('VM') with 5681 values in millivolts (mV), and Channel 5 ('FX') with 285 values in degrees (deg). Leveraging the power of deep learning, we develop a comprehensive model that integrates various advanced techniques to enhance prediction accuracy. Our approach employs Convolutional Neural Networks (CNN), attention blocks, residual blocks, and capsule networks to capture intricate patterns within the data. Additionally, the model architecture includes fully connected layers, dropout layers, global pooling, batch normalization, and early stopping mechanisms to prevent overfitting and improve generalization.

A crucial preprocessing step involves converting the signals into spectrogram images, facilitating effective feature extraction. Wavelet denoising and outlier removal are applied to enhance signal quality before feeding it into the model. The Nadam optimizer is utilized to optimize the learning process, striking a balance between the adaptive momentum estimation of Adam and the accelerated convergence properties of Nesterov momentum.

Our model achieves an impressive accuracy of 99.2%, effectively predicting lower limb activity and showcasing the efficacy of combining multiple advanced deep learning techniques. This approach has significant implications for improving predictive models in biomedical engineering and related fields.

Dataset Description

Original EMG Datasets

This dataset, posted on July 10, 2017, by Yi Zhang, Peiyang Li, Xuyang Zhu, Steven W. Su, Qing Guo, Peng Xu, and Dezhong Yao, comprises electromyography (EMG) recordings from the vastus medialis muscle of the leg in 14 healthy subjects. The data were collected during the performance of three different exercise programs: leg extension from a sitting position (sitting), leg flexion while standing (standing), and gait (walking).

Data Collection

For each subject, the dataset includes three distinct files corresponding to the three exercise programs. This results in a total of 42 files (14 subjects * 3 exercises per subject). Each file captures two channels of data:

1. **Channel 3: 'VM'**
    - Represents the EMG signals from the vastus medialis muscle.

    - Measured in millivolts (mV).
    - No filters applied to this channel.
2. **Channel 5: 'FX'**
    - Represents the angle data, capturing the movement of the leg.
    - Contains 285 values.
    - Measured in degrees (deg).
    - No filters applied to this channel.
    - The original sampling rate was 50 samples per second, extrapolated to 1000 samples per second.

The first column represents the values from Channel 3 ('VM'), while the second column represents the values from Channel 5 ('FX'). These values change across different files, reflecting the variations in muscle activity and leg angles during different exercises and across different subjects.

Data Characteristics

- **Vastus Medialis EMG ('VM') Channel**:
    - Captures electrical activity of the vastus medialis muscle.
    - The data is in millivolts, indicative of muscle activation levels.
    - This channel has a high density of values (5681 per exercise).
- **Angle ('FX') Channel**:
    - Captures the angular movement of the leg.
    - The data is in degrees, representing the range of motion.
    - This channel has a lower density of values (285 per exercise), extrapolated from a lower original sampling rate.

One notable characteristic of the dataset is the significantly larger range of values in the angle channel compared to the EMG channel. This difference in range necessitates careful preprocessing to ensure the signals are appropriately scaled and normalized for analysis.

Dataset Link

The dataset can be accessed via the following link: EMG Lower Limb Dataset.

This dataset provides a valuable resource for studying the relationship between muscle activity and movement, enabling the development and testing of predictive models for lower limb activity. The comprehensive data collection across multiple subjects and exercise programs enhances the generalizability of any findings derived from this dataset.

Data Preprocessing and Spectrogram Image Generation Techniques

Data Loading

The initial step in the data preprocessing pipeline involves loading the raw EMG data from text files. Each file contains two columns: 'EMG' representing electromyography signals and 'Angle' representing the angular movement of the leg. These files are read into a pandas

DataFrame, skipping the first few lines to eliminate any metadata or headers and focusing on the numeric data.

Removing Outliers

Outliers in the data can skew the results and negatively impact the performance of the model. The Z-score method is used for outlier removal. This statistical technique calculates the number of standard deviations a data point is from the mean of the dataset. Data points with Z-scores above a certain threshold (commonly 3) are considered outliers and are removed from the dataset. This ensures that the dataset contains only the most representative and reliable data points.

Signal Denoising

Noise in the signals can introduce unwanted variability and reduce the accuracy of the model. Gaussian filtering, a type of low-pass filter, is used to denoise the signals. This filter smooths the data by averaging the values of neighboring points, effectively reducing high-frequency noise while preserving the essential features of the signal. The sigma parameter controls the extent of smoothing, with higher values resulting in more aggressive noise reduction.

Short-Time Fourier Transform (STFT)

The core of the preprocessing pipeline involves converting the time-domain signals into time-frequency domain representations using the Short-Time Fourier Transform (STFT). STFT is a mathematical technique that divides a longer time signal into shorter segments of equal length and then computes the Fourier transform for each segment. This results in a two-dimensional representation where one axis represents time, the other represents frequency, and the intensity of colors represents the magnitude of the signal at each frequency and time point. This transformation is essential for capturing both the temporal and spectral characteristics of the EMG and angle signals, providing a more informative input for the deep learning model.

Sampling and Image Creation

To ensure the model receives sufficient and varied training data, the signals are segmented into smaller, overlapping samples. Each sample is transformed into an STFT spectrogram image. This involves:

1. **Segmentation**: The data is divided into samples of a fixed size (750 data points in this case). Multiple samples are generated from each signal by sliding a window across the entire length of the signal, creating overlapping segments.
2. **STFT Computation**: For each sample, STFT is applied to both 'EMG' and 'Angle' signals. This generates two spectrograms per sample.
3. **Image Generation**: The spectrograms of 'EMG' and 'Angle' are plotted side by side to create a single combined image. Consistent frequency and time axis limits are maintained across all images to ensure uniformity. The color intensity in these images corresponds to the magnitude of the STFT, with careful scaling to capture the full range of magnitudes across different samples.

Saving and Formatting Images

The generated images are saved in a specified directory, organized by exercise type (sitting, standing, walking). Each image is saved at a high resolution to ensure the clarity of the spectrograms. Post-saving, each image is resized to a standard dimension (400x400 pixels) to maintain consistency when used as input to the deep learning model. This resizing step is crucial for ensuring that all images have the same dimensions, which is a requirement for most convolutional neural networks (CNNs).

Summary

By meticulously following these preprocessing steps, we ensure that the dataset is clean, noise-free, and well-structured for deep learning. The conversion of signals into spectrogram images allows the model to leverage both time and frequency domain information, enhancing its ability to predict lower limb activity accurately. The entire process results in the generation of 4200 images (1400 per exercise), providing a rich and diverse dataset for training robust and reliable predictive models.

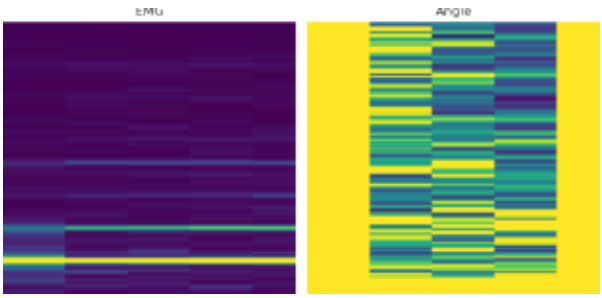


Figure 1 Activity-Gait

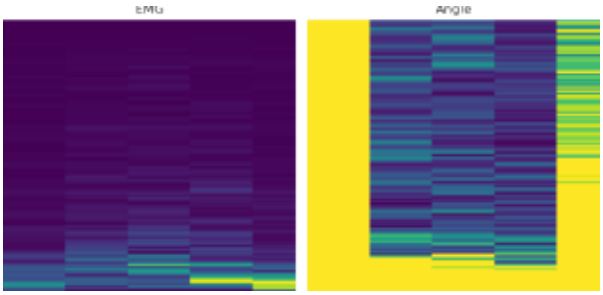


Figure 2 Activity-Sitting

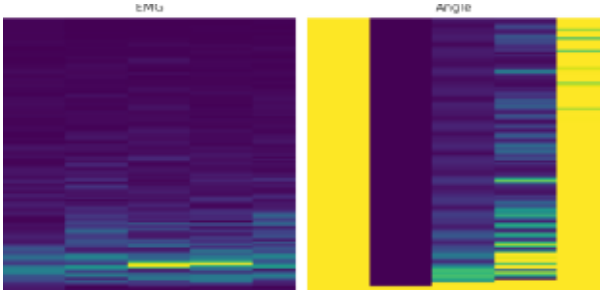


Figure 3 Activity-Standing

Data Loading and Augmentation Process for Training the Model

Overview

To efficiently train a deep learning model on a large dataset of spectrogram images, we employ data generators to load and preprocess the data in batches. This approach optimizes memory usage and speeds up the training process. The following steps outline the process of preparing the data for model training, including data scaling, batching, and splitting into training and test sets.

Data Scaling and Normalization

The first step involves scaling the pixel values of the images. Spectrogram images typically have pixel values ranging from 0 to 255. To normalize these values, we use an instance of `ImageDataGenerator` with a rescale parameter set to `1./255`. This normalization step ensures that the pixel values are within the range [0, 1], which is crucial for the stability and performance of many deep learning models.

```
Python code
datagen = ImageDataGenerator(rescale=1./255)
```

Data Loading with ImageDataGenerator

The `ImageDataGenerator` is used to create an efficient pipeline for loading images from the directory structure. This generator reads images in batches directly from the disk, reducing memory usage by avoiding loading the entire dataset into memory at once.

```
python
Copy code
generator = datagen.flow_from_directory(
    dataset_path,
    target_size=(200, 200),
    color_mode='rgb',
    class_mode='categorical',
    batch_size=32,
    shuffle=False
)
```

- **dataset_path**: Path to the directory containing the spectrogram images.
- **target_size**: Resize all images to 200x200 pixels.
- **color_mode**: Load images in RGB format.
- **class_mode**: Use categorical labels for multi-class classification.
- **batch_size**: Number of images to load in each batch.
- **shuffle**: Disabled to maintain consistent order of images across epochs.

Collecting All Images and Labels

To prepare the data for splitting into training and test sets, all images and labels are collected from the generator. This step involves iterating over all batches produced by the generator and concatenating them into single arrays.

```
python
Copy code
for _ in range(generator.samples // generator.batch_size + 1):
    imgs, lbls = next(generator)
    all_images.append(imgs)
    all_labels.append(lbls)

all_images = np.concatenate(all_images, axis=0)
all_labels = np.concatenate(all_labels, axis=0)
```

- **all_images**: Array containing all spectrogram images.
- **all_labels**: Array containing the corresponding labels for the images.

Splitting Data into Training and Test Sets

To evaluate the model's performance, the dataset is split into training and test sets using `train_test_split` from `sklearn.model_selection`. This function randomly partitions the dataset, ensuring that a portion (20% in this case) is set aside for testing.

```
python
Copy code
X_train, X_test, Y_train, Y_test = train_test_split(all_images, all_labels,
test_size=0.2, random_state=42)
```

- **X_train, Y_train**: Training data and labels.
- **X_test, Y_test**: Test data and labels.
- **test_size**: Fraction of the dataset to reserve for testing.
- **random_state**: Seed for reproducibility.

Creating Data Generators for Training and Testing

New instances of `ImageDataGenerator` are created for training and testing to provide batches of data during model training. These generators yield batches of images and labels indefinitely, which is useful for training in multiple epochs.

```
python
Copy code
train_datagen = ImageDataGenerator()
train_generator = train_datagen.flow(X_train, Y_train, batch_size=32)

test_datagen = ImageDataGenerator()
test_generator = test_datagen.flow(X_test, Y_test, batch_size=32)
```

- **train_generator**: Generates batches of training data.
- **test_generator**: Generates batches of test data.

Determining Steps per Epoch and Validation Steps

The number of steps per epoch and validation steps are calculated based on the length of the training and test generators. These values determine how many batches the generator should produce in each epoch during training.

```
python
Copy code
```

```
steps_per_epoch = len(train_generator)
validation_steps = len(test_generator)
```

- **steps_per_epoch**: Number of batches of training data to use in each epoch.
- **validation_steps**: Number of batches of test data to use for validation after each epoch.

Summary

This process ensures that the data is efficiently loaded, scaled, and split into training and test sets, ready for the model training phase. Using `ImageDataGenerator` for loading and augmenting data helps in optimizing memory usage and speeding up the training process, especially for large datasets like spectrogram images.

Technologies and Techniques Used in Model Building

TensorFlow and Keras

TensorFlow: TensorFlow is an open-source deep learning framework developed by Google for building and training machine learning models.

Keras: Keras is an API designed for human beings, not machines. It puts user experience front and center. Keras is a high-level neural networks API, capable of running on top of TensorFlow.

Layers and Techniques

1. **Convolutional Layers (`Conv2D`)**:
    - Used for spatial convolution over images.
    - Applies a specified number of filters to the image, each filter performing convolution to produce a set of activations.
2. **Batch Normalization (`BatchNormalization`)**:
    - Normalizes the activations of a previous layer at each batch, i.e., applies a transformation that maintains the mean activation close to 0 and the activation standard deviation close to 1.
3. **MaxPooling2D (`MaxPooling2D`)**:
    - Downsamples the input representation by taking the maximum value over the window defined by `pool_size` for each dimension along the features axis.
4. **Dropout (`Dropout`)**:
    - Regularization technique where a proportion of nodes in the layer are randomly ignored (dropped out) during training.
    - Helps prevent overfitting by forcing the model to learn redundant representations.
5. **Residual Blocks**:
    - **Residual Connection**: A skip connection that allows gradients to flow through a shortcut, alleviating the vanishing gradient problem.
    - **Residual Learning**: Learning residual functions with reference to the layer inputs, which makes training deeper networks easier.
6. **Attention Mechanism**:
    - Adds attention weights to the feature map to highlight important regions.

    - Consists of `Conv2D` layers with `ReLU` and `Sigmoid` activations to compute attention weights.
7. **Capsule Layer (`Capsule`)**:
    - A group of neurons whose activity vector represents the instantiation parameters of a specific type of entity.
    - Uses a custom `Capsule` layer implementation with dynamic routing for better representation of hierarchical relationships within data.
8. **Global Average Pooling (`GlobalAveragePooling2D`)**:
    - Computes the average of all values in the feature maps across spatial dimensions.
    - Reduces each feature map to a single value, preserving spatial information.
9. **Custom Activation Functions**:
    - **Squash Activation (`squash`)**: Scales input vectors along their length to a value less than 1.
    - **Softmax Activation (`softmax`)**: Converts logits to probabilities, ensuring they sum to 1 across classes.

Optimizer

- **Nadam Optimizer (`Nadam`)**:
    - Adam optimizer variant that incorporates Nesterov momentum.
    - Adaptive learning rate optimization algorithm that combines momentum techniques with adaptive learning rates.

Metrics

- **Custom F1 Score Metric (`F1Score`)**:
    - Calculates the F1 score, a measure of a model's accuracy, combining precision and recall.
    - Custom implementation using TensorFlow operations for binary or multi-class classification tasks.

Callbacks

- **EarlyStopping (`EarlyStopping`)**:
    - Monitors a specified metric (e.g., validation loss) and stops training when it stops improving, thereby preventing overfitting.
- **ReduceLROnPlateau (`ReduceLROnPlateau`)**:
    - Reduces the learning rate when a monitored metric (e.g., validation loss) has stopped improving.
    - Helps in fine-tuning the model during training.

Summary

This model incorporates advanced techniques like residual connections, attention mechanisms, and capsule networks to improve feature representation and capture hierarchical relationships in spectrogram images. Custom metrics and optimizers enhance model evaluation and training efficiency, while callbacks ensure effective training and prevent overfitting. Overall, this setup demonstrates a sophisticated approach to deep learning model building using TensorFlow and Keras for image classification tasks.

Model Architecture

This document describes the architecture of the deep learning model designed for spectrogram image classification. The model integrates several advanced techniques, including residual blocks, attention mechanisms, capsule layers, and regularization strategies to achieve high performance and robustness.

Input Layer

- **Input Shape**: `(200, 200, 3)`
- The model expects input images of size 200x200 with 3 color channels (RGB).

Initial Convolutional Layer

- **Layer Type**: Convolutional Layer
- **Number of Filters**: 32
- **Filter Size**: (3, 3)
- **Activation Function**: ReLU
- **Padding**: Same
- **Additional Layers**:
    - **Batch Normalization**: Normalizes the activations of the previous layer to speed up training.
    - **MaxPooling**: (2, 2) - Reduces the spatial dimensions of the feature maps.
    - **Dropout**: 0.25 - Randomly drops 25% of the neurons to prevent overfitting.

Residual Blocks with Attention

- The model includes multiple residual blocks with integrated attention mechanisms to improve feature extraction and model performance.
- **Number of Residual Blocks per Group**: 2
- **Residual Block**:
    - **Convolutional Layers**: 2 convolutional layers with the same number of filters as the input.
    - **Batch Normalization**: Applied after each convolutional layer.
    - **ReLU Activation**: Applied after the first convolutional layer and at the end of the residual block.
    - **Skip Connection**: Adds the input of the block to the output of the second convolutional layer.
- **Attention Block**:
    - **Conv2D Layers**: Two (1, 1) convolutional layers, the first with ReLU activation and the second with Sigmoid activation.
    - **Element-wise Multiplication**: Multiplies the output of the attention block with its input to emphasize important features.

First Group

- **Filters**: 32
- **Layers**:
    - 2 Residual Blocks with Attention

    - Convolutional Layer: 64 filters, (3, 3), ReLU, same padding
    - Batch Normalization
    - MaxPooling: (2, 2)
    - Dropout: 0.25

Second Group

- **Filters**: 64
- **Layers**:
    - 2 Residual Blocks with Attention
    - Convolutional Layer: 128 filters, (3, 3), ReLU, same padding
    - Batch Normalization
    - MaxPooling: (2, 2)
    - Dropout: 0.25

Third Group

- **Filters**: 128
- **Layers**:
    - 2 Residual Blocks with Attention
    - Convolutional Layer: 256 filters, (3, 3), ReLU, same padding
    - Batch Normalization
    - MaxPooling: (2, 2)
    - Dropout: 0.25

Fourth Group

- **Filters**: 256
- **Layers**:
    - 2 Residual Blocks with Attention

Global Average Pooling

- Reduces each feature map to a single value by computing the average of all values in the feature map, significantly reducing the number of parameters and helping to prevent overfitting.

Capsule Layer

- **Reshape**: Reshapes the output from global average pooling to be compatible with the capsule layer.
- **Capsule Layer**:
    - **Number of Capsules**: 10
    - **Dimension of Each Capsule**: 16
    - **Routings**: 3
    - **Share Weights**: True
- **Flatten**: Flattens the output from the capsule layer to prepare it for the dense layers.

Fully Connected Layer

- **Dense Layer**: 128 neurons, ReLU activation

- **Dropout**: 0.5 - Randomly drops 50% of the neurons to prevent overfitting.

Output Layer

- **Dense Layer**: 3 neurons, Softmax activation
- The output layer produces a probability distribution over the 3 classes.

Model Compilation

- **Loss Function**: Categorical Crossentropy - Suitable for multi-class classification problems.
- **Optimizer**: Nadam (Nesterov-accelerated Adaptive Moment Estimation) - Combines the benefits of Nesterov momentum and Adam optimizer.
- **Metrics**:
    - Accuracy: Measures the percentage of correct predictions.
    - Custom F1 Score: Combines precision and recall to give a single performance metric, particularly useful for imbalanced datasets.

Model Summary

- The model architecture includes detailed layer configurations, parameter counts, and shapes of the data as it flows through the network. This summary helps in understanding the complexity and size of the model.

This model leverages the power of advanced neural network techniques to effectively classify spectrogram images, ensuring robust performance and generalization.

Evaluation Metrics & Results

In the context of evaluating machine learning models, various metrics are employed to assess their performance. These metrics provide insights into how well the model generalizes to unseen data and how effectively it classifies different classes within the dataset. Here's an overview of the evaluation metrics used in the classification report and confusion matrix for a multi-class classification task:

Precision

Precision quantifies the number of true positive predictions made by the model over the total number of positive predictions (both true positives and false positives). It measures the accuracy of positive predictions.

- **Macro Precision**: Computes precision for each class and takes the unweighted average.
- **Weighted Precision**: Computes precision for each class, weighted by the number of true instances for each class.

Recall

Recall calculates the ratio of true positive predictions to the sum of true positives and false negatives. It measures the model's ability to correctly identify all positive instances.

- **Macro Recall**: Calculates recall for each class and takes the unweighted average.
- **Weighted Recall**: Computes recall for each class, weighted by the number of true instances for each class.

F1-Score

The F1-score is the harmonic mean of precision and recall. It provides a single metric that balances both precision and recall, offering a more comprehensive view of model performance.

- **Macro F1-Score**: Computes the F1-score for each class and takes the unweighted average.
- **Weighted F1-Score**: Calculates the F1-score for each class, weighted by the number of true instances for each class.

Accuracy

Accuracy measures the proportion of correctly predicted instances out of the total instances in the dataset. It is a straightforward metric but can be misleading in imbalanced datasets.

Classification Report

The classification report provides a detailed summary of evaluation metrics for each class in the dataset. It includes precision, recall, and F1-score for each class, along with support (the number of true instances for each class).

Confusion Matrix

A confusion matrix is a table that visualizes the performance of a classification model. It shows the number of true positive, true negative, false positive, and false negative predictions made by the model across all classes.

Interpretation of Results

- **Precision, Recall, and F1-Score**: In the provided example, the precision, recall, and F1-score are all high across the classes ('gait', 'standing', 'sitting'), indicating that the model performs well in correctly identifying each class.
- **Support**: This refers to the number of actual occurrences of each class in the dataset.
- **Accuracy**: The overall accuracy of 99% indicates that the model correctly predicts the class label for 99% of instances in the test set.
- **Macro and Weighted Averages**: Macro averages give equal weight to each class, while weighted averages consider the support (number of instances) for each class, providing a more realistic measure of performance in imbalanced datasets.

The classification report for the model is as follows:

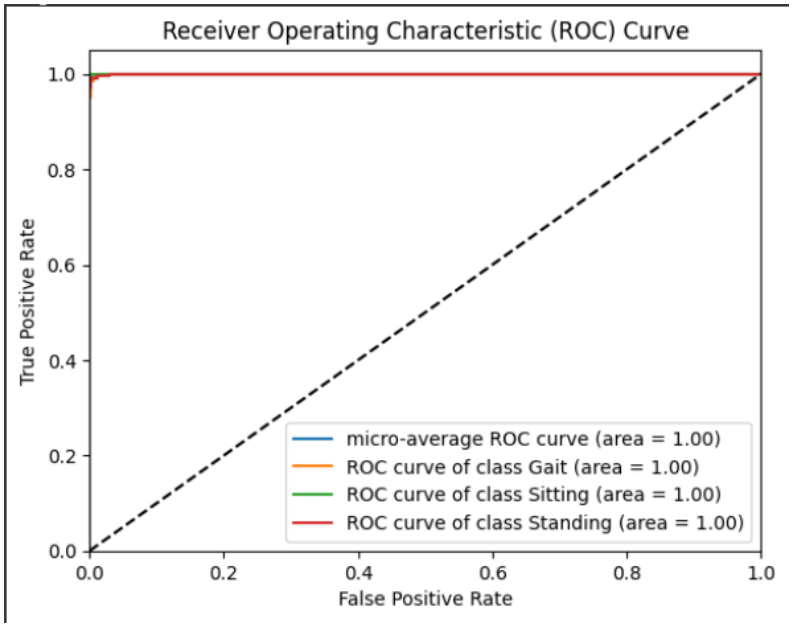
	precision	recall	f1-score	support
gait	0.99	0.99	0.99	246
standing	1.00	1.00	1.00	214
sitting	0.99	0.99	0.99	202
accuracy			0.99	662
macro avg	0.99	0.99	0.99	662
weighted avg	0.99	0.99	0.99	662

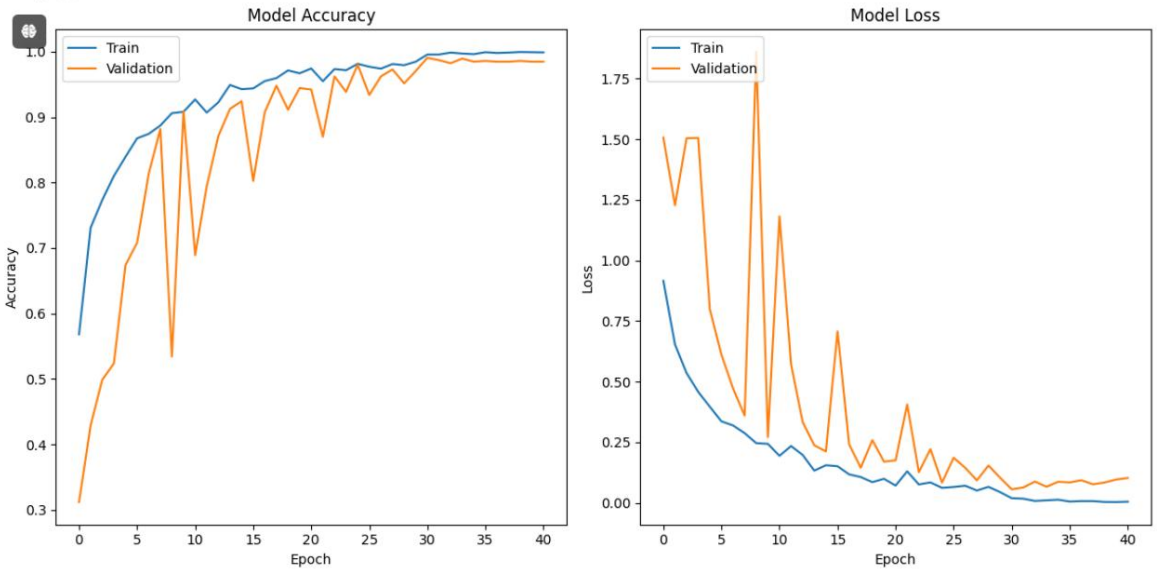
- **Gait**: Precision = 0.99, Recall = 0.99, F1-score = 0.99
- **Standing**: Precision = 1.00, Recall = 1.00, F1-score = 1.00
- **Sitting**: Precision = 0.99, Recall = 0.99, F1-score = 0.99
- **Overall Accuracy**: 0.99
- **Macro Average**: Precision = 0.99, Recall = 0.99, F1-score = 0.99
- **Weighted Average**: Precision = 0.99, Recall = 0.99, F1-score = 0.99

Confusion Matrix:

```
[[244   0   2]
 [  0 213   1]
 [  2   0 200]]
```

- **Gait**: 244 true positives, 0 false positives, 2 false negatives
- **Standing**: 213 true positives, 0 false positives, 1 false negative
- **Sitting**: 200 true positives, 0 false positives, 2 false negatives






```
27/27 - 2s - loss: 0.0559 - accuracy: 0.9905 - f1_score: 0.9905 - 2s/epoch - 58ms/step
Test Accuracy: 0.9905437231063843
Test F1 Score: 0.9905436635017395
```