Artificial intelligence: Simple Activity Recognition

Sharareh Sayyad[1]

[1]*Department of Mathematics and Statistics, Washington State University, Pullman, Washington, USA*

(Dated: May 4, 2025)

CONTENTS

I. INTRODUCTION

One of the active research areas in computer vision is the ability to recognize human actions from sensor-based data. Collecting data from various portable devices, e.g., mobile phones, and analyzing their underlying information offers numerous applications in healthcare systems and also in improving interactions between humans and robots/computers. The activity recognition tasks may vary depending on the sensors and the complexity of their collected data. Reported attempts at tackling such problems range from deep learning approaches [3] to statistical learning methods [1], each with its shortcomings and strengths.

This project addresses a simple human activity recognition (HAR) problem: classifying human activities given time-dependent sensor-based data. For this purpose, we implemented a Python package and examined its performance on an open-access dataset known as the "WISDM Smartphone and Smartwatch Activity and Biometrics Dataset". As activity labels were available in this dataset, we implemented various supervised deep-learning approaches to achieve our goal.

This report is organized as follows. We present a concise summary of the statistical properties of our dataset, followed by how we preprocess samples for training input in Sec. II. Section III provides details of the implemented models in our `DL_ActRec` package available on Github with installation steps presented at GoogleColab notebook. We discuss further implementation details for training and testing our models in Sec. IV. Our results and related discussions can be found in Section V.

II. DATA ANALYSIS AND DATA PREPROCESSING

a. Statistical properties of the dataset The WISDM dataset contains 1098207 samples whose features are user, timestep, x-accelerometer, y-accelerometer, z-accelerometer, and activities. The activities that we will use as labels for our supervised classification tasks are Walking, Jogging, Upstairs, Downstairs, Sitting, Standing. Figure 1 presents the activities of four random users chosen from the WISDM data set. Clearly, one notices that certain actions for different users are not collected for all time instances, and in some cases, particular activities are not available for every user. This results in an imbalance in the number of instances for each activity.

To better understand the distributions and the number of samples in each activity, we present the pie chart (right panel) and density plot (left) of each activity in Fig. 2. The results show that the

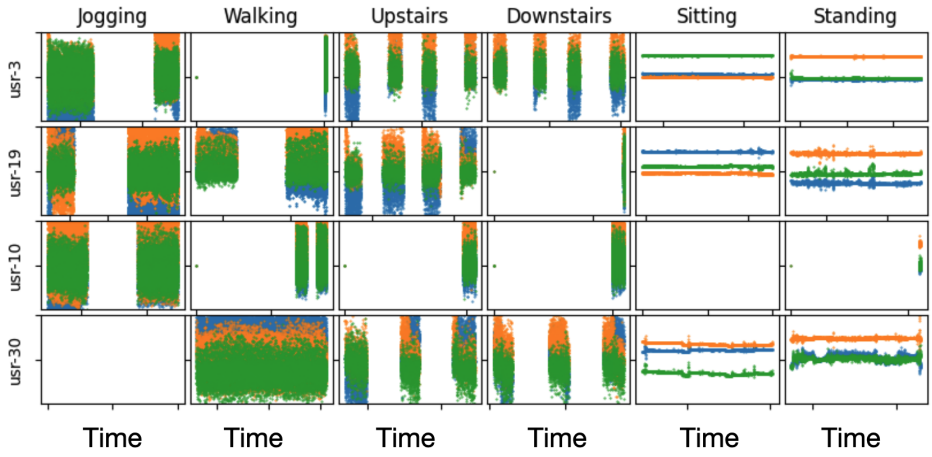

Figure 1. Time-dependent activities of four different users from the WISDM dataset.

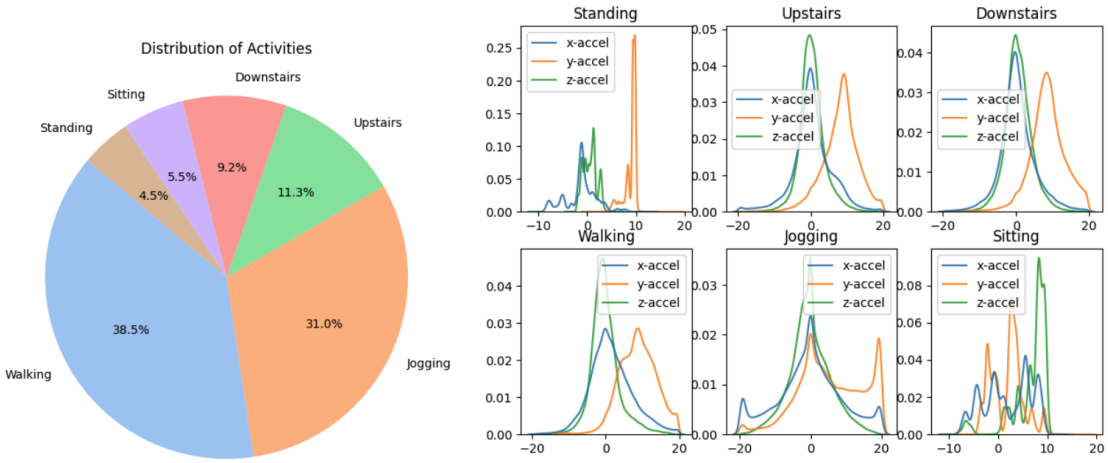

Figure 2. The pie chart (left) and density distributions (right) for different activities.

majority group is { walking, jogging }, adding to almost 70% of the samples. The minority group comprises {Standing, Sitting} accounts for almost 10% of the total dataset. We further notice that the density distributions of $x/y/z$-accelometers for most activities are quite distinct, except upstairs and downstairs. For these two activities, we have nearly identical distributions with close values of mean and standard deviation; see Table I.

Activity	Mean [x-accel]	Std[x-accel]	Mean [y-accel]	Std[y-accel]	Mean [z-accel]	Std[z-accel]
Jogging	-0.219	9.168	5.434	9.216	-0.150	5.847
Walking	1.546	5.826	8.695	5.022	-0.112	4.018
Upstairs	0.381	5.495	8.111	4.890	0.323	3.568
Downstairs	0.471	4.956	8.600	4.905	0.684	3.706
Sitting	1.856	4.759	1.853	3.258	6.560	3.736
Standing	-1.178	3.235	8.987	1.2648	0.580	1.377

Table I. Summary of the mean and standard deviation values of x-accelerometer, y-accelerometer, z-accelerometer for each activity.

b. input preparation for neural network model training Having the WISDM dataset, we merely use x-acelometer, y-accelometer and z-accelometer as the data and their associated activities as labels to train our models. Before model training, we prepare the sample in three different ways, namely a) treating all pairs equally without enforcing the sequential properties, b) building up a time series out of related instances with a fixed maximum duration, and c) introducing a non-zero, non-overlapping

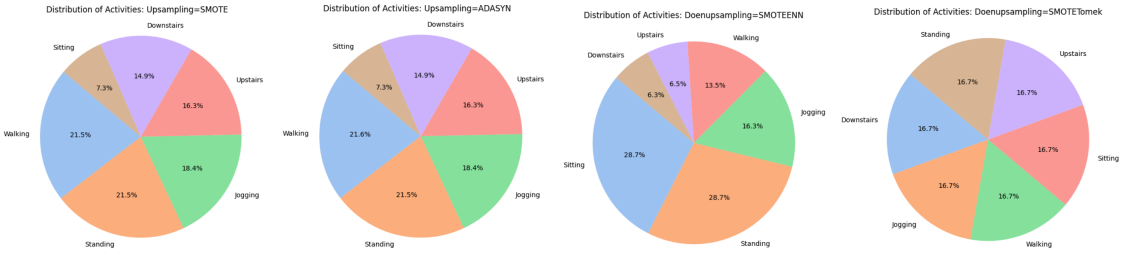

Figure 3. The distribution of activities after performing upsampling (left two panels) and downupsapling techniques (right two panels).

sequence of time series prepared in b). The size of the input dataset in a) is the same as the number of instances in the whole dataset. In b), we merely get 179 samples each with a length of 4000. Setting the time window to $d = 100$, we obtain the sample size of 5725 for training. The value of d is set by the users, and we usually choose values from $\{25, 50, 100\}$.

Up to now, we have not touched upon the challenge of dealing with an imbalanced dataset, as shown in Fig. 2. A popular approach is to upsample minority classes to reduce the imbalance rate between different activity classes. To upsample our dataset, we imported `SMOTE, ADASYN` modules from the Imbalanced-learn (imported as imblearn) open-source library. Another possibility to deal with the uneven distribution of samples in classes is to downsample from the majority classes and copy the minorities to reduce the imbalance. The approach is known as the down-sampling technique, with implementation available in the imblearn library. We employed `SMOTEENN, SMOTETomek` modules to carry out downupsampling tasks. The outcome of employing such upsampling and downupsampling techniques results in redistributing samples for different activities, as shown in Fig. 3 [2]. We witness an increase in the number of samples in the minority classes and a reduction in the size of the majority classes. Notably, employing `SMOTETomek` brings the samples into a balanced dataset with uniform distribution.

III. ARCHITECTURE OF NEURAL NETWORK MODELS

In this section, we present a summary of model architectures to classify activities within our dataset. We implement all of these models using `PyTorch` library. For all models, two parameters `num_input, num_output` are passed to the code, which is set to be 3 (number of features) and 6 (number of classes), respectively.

a. Multi-layer perceptron neural network models. One of our deep learning models is a Multi-layer perceptron (MLP). Our implementation is stored in `MLP/nn_mlp_model.py`. Our code contains two different MLP models dubbed `MLPI, MLPII`. The MLP1 model with a rectangular shape requires `num_hidden, num_layers` to determine the depth and the width of hidden linear layers. The model further allows 20% of dropout of weights between input and hidden layers as well as hidden and output layers. For the MLP2 model, we pass `hidden_lyrs` to the code, which is a list of the size of hidden units in the network; we typically set the order of elements in `hidden_lyrs` to first increase and then decrease in value. This list is generated by `create_increasing_decreasing_list` function available in `MLP/run_training_testing_mpl.py`.

b. Convolution neural network models. The second family of models that we included in our package is the convolutional neural network (CNN) model. Our implementation is available in `CNN_LSTM/nn_CNN_model.py`, consisting of three different models `CNNI, CNNwithBatchNorm, CNNSkipConnections`. The parameters that should be set for the CNNI and CNNwithBatchNorm models are as follows. `max_length_series`, which is the maximum length of the input series, `num_conv_layers`, which determines the number of convolutional layers, the size of the linear layer `size_linear_lyr`, and `initial_channels`, which is the number of initial channels. An extra parameter `num_blocks_per_layer`, which fixes the number of residual blocks in our convolutional layers, should be provided for the CNNSkipConnections model. In all one-dimensional convolutional layers in these models, we set the kernel size to be 3 with padding 1. Max pooling and batch normalization layers are also incorporated into the architecture, usually called after the convolutional layers.

c. Long Short-Term Memory neural network models. Our package also contains Long Short-Term Memory (LSTM) models. Our implementation is available in `CNN_LSTM/nn_LSTM_model.py`. Here, we

only have one class of LSTM model, dubbed `LSTMI`. For using this mode, we should set the maximum length of the time series (`max_length_series`), the number of LSTM layers (`num_lstm_layers`), the size of the hidden state in LSTM (hidden_size), and the size of the linear layer (`size_linear_lyr`).

d. CNN-LSTM models We also provide combinations of CNN and LSTM models in our package. Our implementation is available in `CNN_LSTM/nn_CNN_LSTM_model.py`. We combine three classes of CNN models with the LSTM model such that inputs go through convolutional layers, followed by LSTM layers, and at the end, pass two linear classification layers. This results in introducing three CNN-LSTM models dubbed `CNN_LSTMI, CNNwithBatchNormLSTM, CNNSkipConnectionsLSTM`. The parameters needed to make these models are a combination of those discussed for CNN and LSTM models.

The last CNN-LSTM model, `CNNwithBatchNormLSTMParrallel`, consists of two independent training of `CNNwithBatchNorm` and `LSTMI`, each with only one final linear layer. The output of these two linear layers is next concatenated and passed through the final linear layer for classification purposes. The parameters for this model are the combination of parameters for `CNNwithBatchNorm` and `LSTMI` models.

e. Transformer neural network models. The last model architecture that we provide in our package is the transformer neural network model. The implementation details are stored in `Transformer/nn_Transformer_model.py`. The transformer model is called `TransformerI` and is set by parameters `seq_len, embed_size, nhead, dim_feedforward, dropout, conv1d_emb, conv1d_kernel_size, size_linear_layers, num_encoderlayers, device` For the input embedding, the code decides to use a 1-dim convolutional layer with kernel size `conv1d_kernel_size` or a linear layer with output size `embed_size` using the boolean parameter `conv1d_emb`. The positional embedding is implemented borrowing the code from the PyTorch Library using the following equations

$$PE_{pos,2i} = sin\left(\frac{pos}{10000^{2i/d_{model}}}\right), \tag{1}$$

$$PE_{pos,2i+1} = \cos\left(\frac{pos}{10000^{2i/d_{model}}}\right). \tag{2}$$

Inside each of the `num_encoderlayers` encode layers, the attention units are set by `embed_size, nhead, dim_feedforward, dropout` parameters. We usually set `dropout` to be zero in the transformer models. The outputs of the encoder layer are passed to three linear layers, with the input size of the last linear layer being `size_linear_layers`, for the final classification purpose.

So far, we have concisely explained the model architectures explored in his project. The code provides both "Kaiming" and "Xavier" initializations for model weights. We only used the Kaiming initialization to generate our results.

IV. FURTHER CONSIDERATIONS FOR TRAINING/TESTING DATA

Aside from model architectures to best tackle the classification of our dataset, we have taken further steps to improve the training of our models and assess their performance. These details comprise details of training based on the weights of activity classes and additional information on regularization techniques. In the following, we briefly present these pieces of information. Related implementation can be found in the directory for each model and in files `nn_(...)_train.py`, `run_training_testing_(...).py`, `run_training_testing_...multiplit.py`.

The pre-training steps with the input dataset in Sec. II 0 b start with splitting the train, validation, and test sets with a ratio of 60%, 20%, 20%. This step provides the class weights for the code to ensure stratified sampling. The generated set will then be normalized based on the mean and std of the train set for two types of normalization, namely `"per-channel", "per-timestep"` (Note that the normalization for the MLP training is merely by standardizing based on the train set as no time-ordering is available for the input of this model.). The rescaled sets are then ready for our training/testing steps. As an optimization, we implement both Adam and AdamW in the code, which can be decided by the user which one to use; all results of this project are obtained using AdamW. To prevent overfitting, we introduced an early stopping and learning rate scheduler, specifically `ReduceLROnPlateau`, in our code. After passing these steps, the training, validation, and test sets are ready to go through the training/training steps. See functions `run_training, testing_step` in files `nn_(...)_train.py` stored the folder assigned for each model.

To assess the performance of our models at each epoch and on the test dataset, we benefit from various metrics, including loss, accuracy, F1-score, Geometric-mean (Gmean), recall, precision, and

Batch size	Model type	Learning rate	# hidden layers	size hidden layers	Accuracy
16	mlp2	0.0001	8	26	0.5679
16	mlp2	0.0001	8	24	0.5669
16	mlp2	0.0001	16	20	0.5621
16	mlp1	0.0001	8	26	0.5512

Table II. Top-4 best MLP performances.

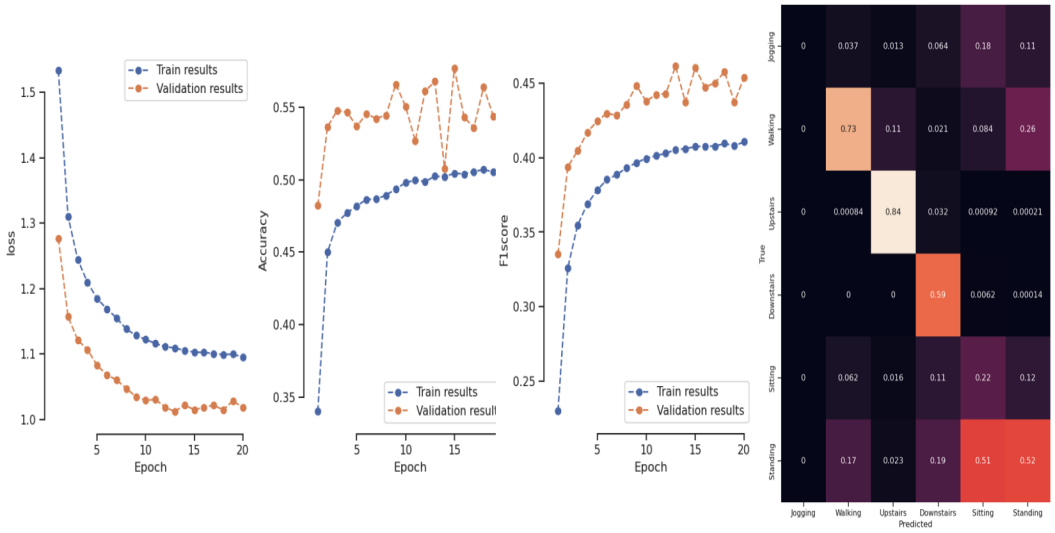

Figure 4. Evolution of metrics for training the MLP model on the first row of TableII The sample distribution for (train,val, test) datasets is (695340,173836,217295).

confusion matrix. The value of loss is obtained from the output of the cross-entropy loss. The other measures read

$$Accuracy = \frac{TP+TN}{TP+TN+FP+FN}, \quad Pecision = \frac{TP}{TP+FP}, \quad Recall = \frac{TP}{TP+FN}, \tag{3}$$

$$F1-score = \frac{2\times Precision \times Recall}{Precision + Recall}, \quad G-mean = \sqrt{\frac{TN\times TP}{(TP+FN)}(TN+FP)}. \tag{4}$$

Here, TP, FP, TN, FN denote true positive, false positive, true negative, and false negative, respectively.

V. RESULTS AND DISCUSSION

To collect results for each model and taking into account different types of input data, we implemented the following Python codes `(mlp/cnn/transformer)_wisdm_nodevice(_segmented).py` for inputs with or without time-ordering and segmented time-ordering, `(mlp/cnn/transformer)_wisdm_nodevice_withupsampling(_segmented).py` which includes upsampling technique and `(mlp/cnn/transformer)_wisdm_nodevice_withdownupsampling(_segmented).py` which incorporates downupsampling technique. These codes accept various command-line arguments; see the code for further details. They are stored in the directories for each family of models. For a quick check of how the algorithm works, Jupyter notebooks `test_(mlp/cnnlstm/transformer).ipynb` are available in each folder. However, we did not use these notebooks to collect data for numerical (time/space) efficiency reasons.

The four best outcomes of the MLP model trained on a dataset with no time-ordering are presented in Table II, and the calculated of metrics as a function of the epoch are plotted in Fig. 4. Figure 4 shows that the MLP model performs poorly and is confused in classifying most classes. This is not surprising as the time-ordering of our features was neglected in training the MLP model.

model	lr	#convlyrs	sizelinearlyr	#blocksperlyr	initchannels	accuracy	f1-score	gmean	precision	recall
cnn1	1E-05	4	32	2	8	0.8056	0.8125	0.8783	0.8135	0.8241
cnnbatchnorm	1E-06	4	32	2	8	0.8056	0.8107	0.8752	0.8095	0.8143
cnnbatchnorm	1E-06	8	32	2	8	0.8056	0.8082	0.8776	0.8079	0.8304
cnn1	1E-05	4	32	4	8	0.7778	0.7576	0.8480	0.7817	0.7794

Table III. Top-4 CNN performances on time-series data with batch size = 1000, normtype= 'per-channel', and max-length-series=4000.

model	lr	#convlyrs	sizelinearlyr	#blocksperlyr	initchannels	accuracy	f1-score	gmean	precision	recall
cnn1	1E-03	6	8	2	8	0.9345	0.9392	0.9635	0.9418	0.9373
cnn1	1E-03	6	8	4	8	0.9293	0.9364	0.9622	0.9401	0.9347
cnnbatchnorm	1E-03	6	8	2	8	0.9275	0.9320	0.9590	0.9343	0.9309
cnnskip	1E-03	2	16	2	8	0.9275	0.9318	0.9587	0.9337	0.9311

Table IV. Top-4 CNN performances on segmented time-series data with batch size =32, normtype= 'per-timetep', max-length-series=4000 and segmentaton durtion=100.

The best outcomes of training various CNN models are in Tables. III, IV, V, VI, VII, VIII. It is evident that training with a segmented dataset results in higher values of measures due to having a larger number of samples available for training. The learning curves of the best model with upsampling techniques are shown in Fig. 5. From the confusion matrix, we notice that the most challenging activities for the classification tasks are upstairs and downstairs.

The best outputs of the LSTM models on the test sets are provided in Tables IX, X. The evolution of measures for the best model is plotted in Fig. 6. Comparing the number of epochs with the CNN models, we realize that training the LSTM model required a larger number of epochs despite the smaller number of parameters in the LSTM models. This results in a larger time complexity for training LSTM models than the CNN ones.

Tables XI, XII summarize the CNN-LSTM model architectures with the best performances. The overall behavior of learning curves for the best model here is similar to the CNN models as shown in Fig. 7. We again witness the confusion in classifying upstairs and downstairs activities, which can be attributed to the same density distribution in the x,y, and z-accelerometer shown in Fig. 2.

The collection of best transformer models are preseneted in Tables XIII, XIV and XV. Note that transformer models are known to require large sample sizes for their training. This is the reason that we merely employ them in training segmented time series with a batch size of 128. The learning curves of the best model for classifying the dataset with the test accuracy of 99% are shown in Fig. 8. Note that downstairs and upstairs are not fully distinguished by our model.

We have provided the values of measures on the test sets after the training model. However, without reporting the uncertainties, it is difficult to understand the properties of specific architectures in terms of the relations between the model parameters and the range of metric values. For this reason, taking the best architecture from the above-reported results, excluding the MLP models, we trained each model 20 times for the segmented datasets, and when upsampling or downupsampling is present/absent. We plot the mean values of measures on the test dataset and standard deviations(error bars) as a function of model parameters in Fig. 9. We notice that LSTM (diamond) models have the least number of parameters, and their uncertainty is most significant. Moving to the CNN model (square) or the CNN-LSTM (triangle), we deal with larger models with smaller std values and better performances. The transformer models (circles) possess the largest number of parameters, but their uncertainty is the smallest; this is what we usually witness in the modern era of deep learning models. We further notice that incorporating downupsampling or upsampling techniques to reduce the imbalance ratio of

model	lr	maxT	#convlyrs	sizelinearlyr	#blocksperlayer	norm-type	accuracy	f1score	gmean	precision	recall
cnnskip	1E-03	2000	2	16	4	'per-timestep'	0.8605	0.8583	0.9116	0.8601	0.8796
cnnskip	1E-05	2000	4	32	2	'per-channel'	0.8372	0.8342	0.8922	0.8333	0.8769
cnnbatchnorm	1E-05	2000	4	32	2	'per-channel'	0.8140	0.8031	0.8771	0.8095	0.8093
cnnbatchnorm	1E-06	2000	8	32	4	'per-channel'	0.8140	0.8110	0.8794	0.8095	0.8218

Table V. Top-4 CNN performances on time-series data after performing the downupsampling SMOTETomek technique. The batch size is 1000. The initial channel is set to 8 in all of these trainings. We show the max-length-series parameter by maxT.

model	lr	#convlyrs	sizelinearlyr	#blocks-per-layer	norm-type	accuracy	f1score	gmean	precision	recall
cnnbatchnorm	1E-03	4	16	2	8	0.9890	0.9812	0.9897	0.9817	0.9809
cnnskip	1E-03	2	16	4	8	0.9860	0.9738	0.9858	0.9747	0.9738
cnn1	1E-03	4	8	2	8	0.9850	0.9713	0.9857	0.9746	0.9703
cnnbatchnorm	0.0001	4	8	2	8	0.9840	0.9717	0.9833	0.9705	0.9744

Table VI. Top-4 CNN performances on segmented time-series data after performing the downupsampling SMOTEENN technique. The bach size is 32 and normalzation tpe is 'per-timestep'. The maximum length of the time series is 4000, and the segmentation duration is 100. The initial channel is set to 8 in all of these trainings.

model	Upsampling	lr	maxT	#convlyrs	linearlyr	#blocks	norm-type	accuracy	f1score	gmean	precision	recall
cnnskip	ADASYN	1E-03	4000	2	16	4	'per-timestep'	0.8718	0.8644	0.9146	0.8651	0.8725
cnnbatchnorm	ADASYN	1E-03	4000	6	2	8	'per-timestep'	0.8462	0.8418	0.8988	0.8413	0.8611
cnnskip	SMOTE	1E-03	2000	2	8	2	'per-timestep'	0.8462	0.8448	0.9014	0.8437	0.8667
cnnskip	SMOTE	1E-03	4000	2	16	2	'per-timestep'	0.8462	0.8418	0.8990	0.8397	0.8572

Table VII. Top-4 CNN performances on time-series data after performing the upsampling technique. The batch size is 1000. The initial channel is set to 8 in all of these trainings. We show the max-length-series parameter by maxT, #blocks-per-layer by #blocks and sizelinearlyr by linearlyr.

model	Upsampling	lr	#convlyrs	linearlyr	#blocks	norm-type	accuracy	f1score	gmean	precision	recall
cnnbatchnorm	ADASYN	1E-03	6	16	2	'per-timestep'	0.9491	0.9471	0.9669	0.9450	0.9497
cnnbatchnorm	ADASYN	1E-03	6	8	2	'per-timestep'	0.9461	0.9432	0.9654	0.9429	0.9439
cnn1	SMOTE	1E-03	6	8	2	'per-timestep'	0.9447	0.9415	0.9635	0.9405	0.9462
cnnbatchnorm	ADASYN	1E-03	6	8	4	'per-timestep'	0.9432	0.9408	0.9635	0.9393	0.9426

Table VIII. Top-4 CNN performances on segmented time-series data after performing the upsampling technique. The batch size is 32. The initial channel is set to 8 in all of these trainings. The max-length-series parameter is 4000, and the segmentation duration is 100. We show #blocks-per-layer by #blocks and sizelinearlyr by linearlyr.

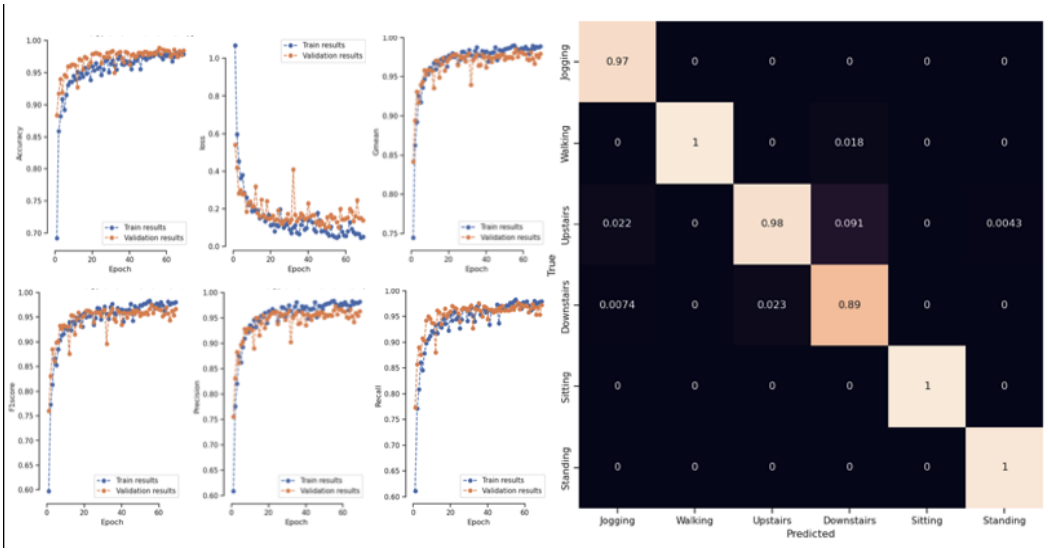

Figure 5. Evolution of metrics for training the CNN model on the first row of TableVIII The sample distribution for (train,val, test) datasets is (3204,802,1002).

model	lr	maxT	seg-duration	sizelinearlyr	lstm-hidden-size	#lstm-lyrs	accuracy	f1-score	gmean	precision	recall
lstm	1E-03	4000	50	32	8	4	0.9316	0.933	0.9592	0.9338	0.9329
lstm	1E-03	4000	50	16	8	4	0.9224	0.9284	0.9559	0.9297	0.9277
lstm	1E-03	4000	50	16	8	2	0.9106	0.9167	0.9494	0.9191	0.9146
lstm	1E-03	4000	50	32	8	2	0.9084	0.9155	0.9481	0.9181	0.9146

Table IX. Top-4 LSTM performances on segmented time-series data with batch size = 32 and normtype= 'per-timestep'. We show the max-length-series parameter by maxT.

model	Upsampling	lr	sizelinearlyr	lstm-hidden-size	#lstm-lyrs	accuracy	f1-score	gmean	precision	recall
lstm	SMOTE	1E-03	32	8	4	0.9554	0.9533	0.9722	0.9541	0.9527
lstm	ADASYN	1E-03	32	8	4	0.9408	0.9397	0.9628	0.9391	0.9408
lstm	SMOTE	1E-03	16	8	4	0.9408	0.9367	0.9625	0.9384	0.9353
lstm	SMOTE	1E-03	8	8	4	0.9295	0.9258	0.9559	0.9287	0.9254

Table X. Top-4 LSTM performances on segmented time-series data after performing the upsampling technique. The batch size is 32, normtype= 'per-timestep', max-length-series=4000 and segmentation duration is 50.

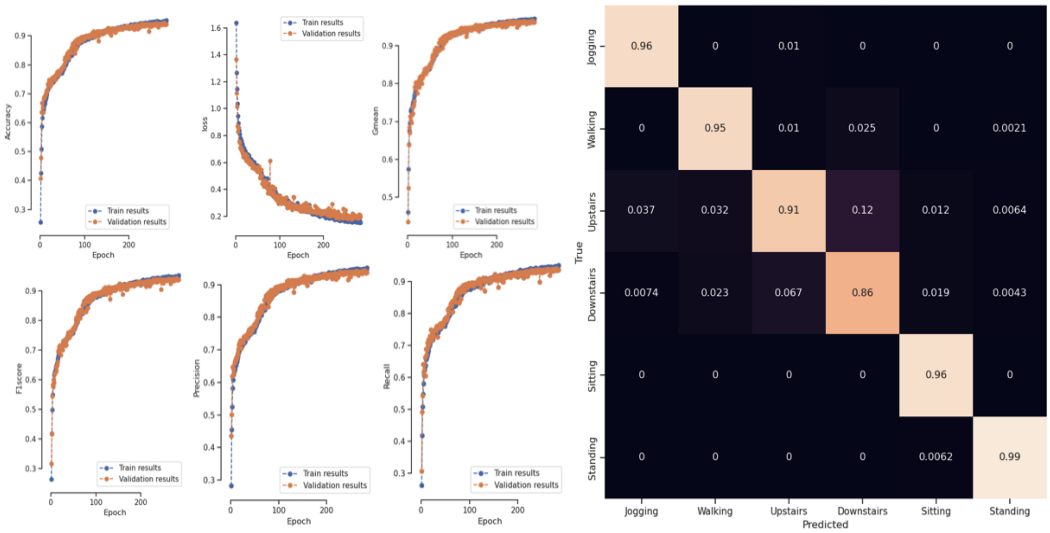

Figure 6. Evolution of metrics for training the LSTM model on the first row of TableX The sample distribution for (train,val, test) datasets is (8535,2134,2668).

model	lr	#convlyrs	sizelinearlyr	#blocks	lstmsize	#lstmlyrs	accuracy	f1-score	gmean	precision	recall
cnnlstmbatchnorm	1E-03	2	16	4	8	2	0.9290	0.9347	0.9598	0.9354	0.9340
cnnlstmbatchnorm	1E-03	2	16	4	6	2	0.9264	0.9344	0.9591	0.9350	0.9341
cnnlstmbatchnorm	1E-03	2	16	2	6	1	0.9251	0.9318	0.9579	0.9331	0.9312
cnnlstmskipparallel	1E-03	8	16	4	8	1	0.9238	0.9294	0.9574	0.9321	0.9277

Table XI. Top-4 CNN-LSTM performances on segmented time-series data with batch size = 32, initialchanels=8, and normtype= 'per-timestep'. We show #blocksperlayer parameter by #blocks and lstmhiddensize by lstmsize.

model	Upsampling	lr	#convlyrs	sizelinearlyr	#blocks	lstmsize	accuracy	f1-score	gmean	precision	recall
cnnlstmbatchnorm	ADASYN	1E-03	6	8	2	8	0.9589	0.9560	0.9728	0.9544	0.9577
cnnlstmparallel	SMOTE	1E-03	6	16	2	4	0.9559	0.9533	0.9721	0.9541	0.9539
cnnlstmparallel	SMOTE	1E-03	6	8	2	8	0.9536	0.9503	0.9694	0.9495	0.9531
cnnlstmbatchnorm	ADASYN	1E-03	6	16	2	6	0.9529	0.9504	0.9687	0.9481	0.9535

Table XII. Top-4 CNN-LSTM performances on segmented time-series data after performing the upsampling technique. The batch size is 32, initialchanels=8, normtype='per-timestep', #lstmlyrs=1, max-length-series=4000 and segmentation duration is 50. We show #blocksperlayer parameter by #blocks and lstmhiddensize by lstmsize. The model "cnnlstmparallel" stands for the "cnnlstmbatchnormparallel" model.

model	lr	boolconv1demb	embedsize	#encoderlyrs	sizelinearlyr	accuracy	f1-score	gmean	precision	recall
trans1	1E-03	FALSE	64	6	32	0.9500	0.9528	0.9714	0.9536	0.9522
trans1	1E-03	FALSE	64	6	64	0.9483	0.9505	0.9695	0.9504	0.9507
trans1	1E-03	TRUE	32	4	32	0.9404	0.9455	0.9662	0.9454	0.9458
trans1	1E-03	TRUE	64	6	32	0.9378	0.9371	0.9629	0.9396	0.9356

Table XIII. Top-4 transformer performances on segmented time-series data with batch size = 128, normtype= 'per-channel', dim-feed-forward=16, conv1d-kernelsize=7, max-length-series=4000 and segmentation duration is 50.

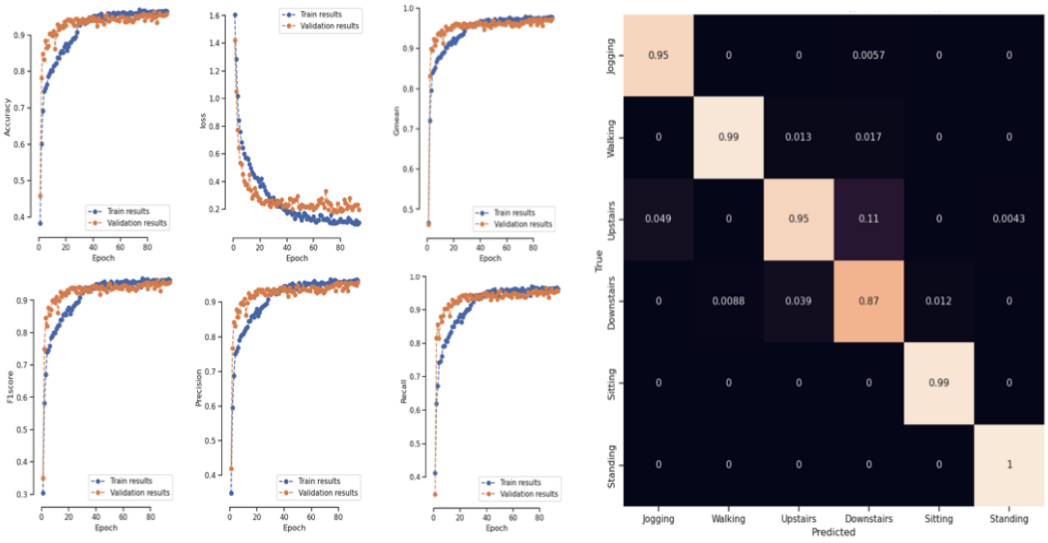

Figure 7. Evolution of metrics for training the CNN-LSTM model on the first row of TableXII The sample distribution for (train,val, test) datasets is (4276,1064,1337).

Upsampling	lr	nhead	boolconv1demb	embedsize	dimfeedfwd	#enc	sizelinearlyr	accuracy	f1-score	gmean	precision	recall
SMOTE	1E-03	16	FALSE	64	16	6	32	0.9633	0.9621	0.9771	0.9621	0.9622
ADASYN	1E-03	16	TRUE	64	16	6	64	0.9629	0.9618	0.9761	0.9605	0.9635
ADASYN	1E-03	16	FALSE	64	16	6	32	0.9599	0.9579	0.9753	0.9593	0.9567
SMOTE	1E-03	8	TRUE	32	32	4	32	0.9558	0.9534	0.9712	0.9521	0.9556

Table XIV. Top-4 transformer performances on segmented time-series data after performing the upsampling technique. The batch size is 128, normtype= 'per-channel', conv1d-kernelsize=7, max-length-series=4000 and segmentation duration is 50. Here, #enc represents the number of encoder layers.

classes improves the overall performance of our models.

Our models' estimates for time complexity are given in Table XVI. Regarding the training duration of our models, CNN families were the fastest.

VI. CONCLUSION

We have extensively explored the classification problem of human activity recognition on an imbalanced dataset. We have trained nearly ten different model architectures within MLP, CNN, LSTM, CNN-LSTM, and Transformer models. We have explored various techniques to deal with the imbalance in our models' input. Our best model reached the test accuracy of 99% using downupsampling techniques. Our best model exhibits small confusion between the upstairs and downstairs activities,

Downupsampling	lr	nhead	boolconv1demb	dimfeedfwd	#enc	sizelinearlyr	accuracy	f1-score	gmean	precision	recall
SMOTEENN	1E-03	16	TRUE	32	4	16	0.9904	0.9860	0.9922	0.9862	0.9858
SMOTEENN	1E-03	32	TRUE	32	1	32	0.9861	0.9808	0.9888	0.9805	0.9812
SMOTEENN	1E-03	8	TRUE	16	4	32	0.9857	0.9809	0.9886	0.9802	0.9816
SMOTEENN	1E-03	16	TRUE	16	4	32	0.9849	0.9792	0.9884	0.9799	0.9787

Table XV. Top-4 transformer performances on segmented time-series data after performing the upsampling technique. The batch size is 128, normtype= 'per-channel', conv1d-kernelsize=7, embedsize=32, max-length-series=4000 and segmentation duration is 50. Here, #enc stands for the number of encoder layers.

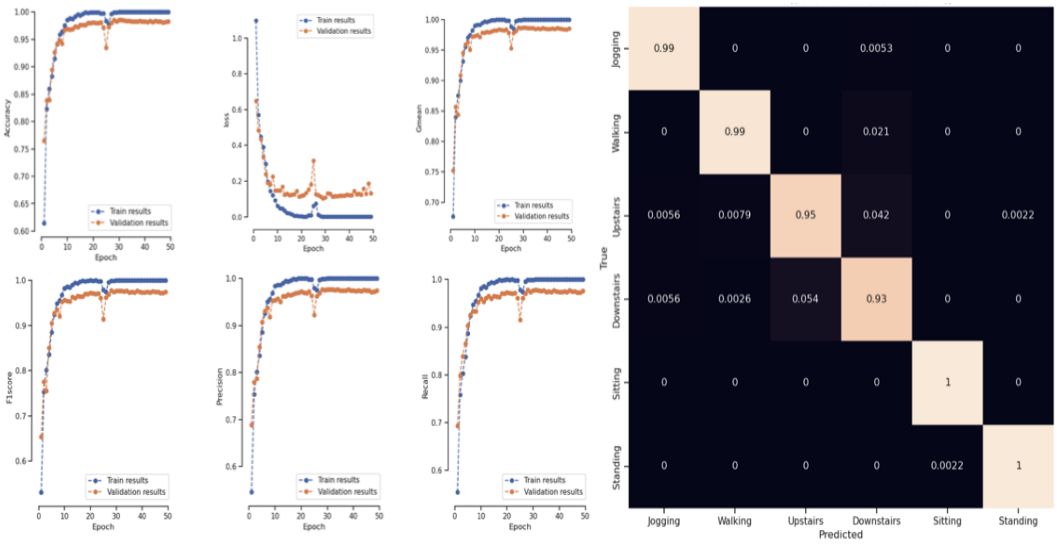

Figure 8. Evolution of metrics for training the transformer model on the first row of TableXV. The sample distribution for (train,val, test) datasets is (8588,2073,2591).

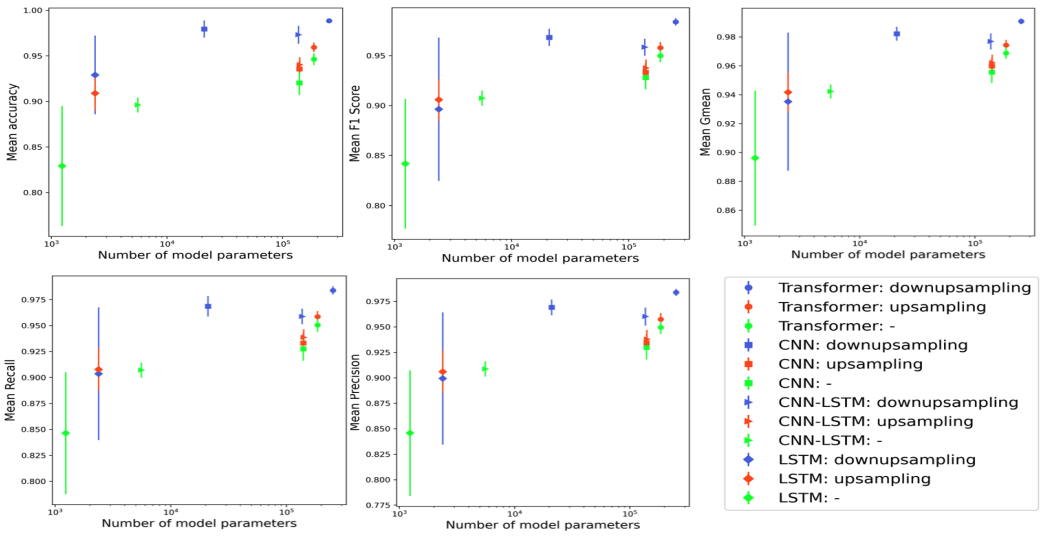

Figure 9. Averaging the measures of best models for each architecture trained 20 times on segmented time-series data.

model	Time-complexity
LSTM	$\mathcal{O}(Nd^2)$ per layer
CNN	$\mathcal{O}(kNd^2)$ per layer
CNN-LSTM	$\mathcal{O}(Nd^2) + \mathcal{O}(kNd^2)$ per layer
Transformer	$\mathcal{O}(N^2d + Nd^2)$ per layer

Table XVI. Time complexity for different models. Here, N denotes the sequence length, d is the hidden dimension, and k represents the kernel size.

which we attribute to the similar underlying density distributions for these two activities.

---

[1] Ranganath Krishnan, Mahesh Subedar, and Omesh Tickoo. Bar: Bayesian activity recognition using variational inference, 2018.
[2] The notebook for plotting these figures is stored in `data_analsis/test_upsampling_downupsampling.ipynb`.
[3] Yi Zhu, Xinyu Li, Chunhui Liu, Mohammadreza Zolfaghari, Yuanjun Xiong, Chongruo Wu, Zhi Zhang, Joseph Tighe, R Manmatha, and Mu Li. A comprehensive study of deep video action recognition. *arXiv preprint arXiv:2012.06567*, 2020.