Ruppin Academic Center
School of Engineering
Department of Electrical and Computer Engineering

HUMAN ACTIVITY RECOGNITION

Submitted in fulfilment of BSc Computer Engineering Final Project, 2023
Supervised by Dr. Benjamin Gur Salomon

ABSTRACT
Human Activity Recognition refers to the classification of temporal data, sampled by sensors, to different human activities. In this project, different types of neural networks are applied to classify a dataset of time-series samples of human activities. Each model is thoroughly tested, and compared against the rest of the models and against state-of-the-art models and algorithms in the literature.

Roei Shchory
BSc Computer Engineering

Contents

Background

Machine Learning and Deep Learning

In the recent decades, the increasing accessibility of consumer electronics, and in particular smart devices (such as smart phones, smart watches, etc.), created a large magnitude of data availability, collected from the different sensors in the aforementioned devices. Vast amounts of data allowed for statistical analysis of different behaviors, and in a more advanced form, a prediction, or a classification of data, based on previously seen data samples.

Prediction, as a problem, is trying to predict what the future behavior of some data will look like, based on previous samples. One such example is trying to predict the behavior of stocks' prices, based on previously collected data, potentially considering different factors (features) such as related events in the field, advertisements, holidays, etc.

Classification, as a problem, is trying to classify a given data sample as a "is a" or "belonging to" a specific group, activity, being, or a thing in general. For example, given a message (email, SMS, etc.), classify it as spam or not. Another example could be classifying in image as containing a dog or a cat.

These approaches to statistical analysis of data are termed as machine learning, where the developer decides which features of the data will be the deciding factors for the prediction/classification. This, however, creates multiple problems: which features are the most efficient to be the deciding factors? What if the developer has no background in the tested field? To overcome these difficulties, a newer approach was developed, deep learning. In essence, this approach allows the program to extract the relevant features without requiring the developer to explicitly state what they are.

Deep learning has multiple methods, algorithms, and approaches to achieve this goal. One such approach is Neural Networks. This method aims to mimic the neural network in a biological brain, allowing the data to "pass through" different layers, where each layer has a different responsibility in prediction/classification. Inside of these layers, different "basic" calculations are taking place, such as matrix multiplication, convolution, etc.

The method that uses the convolution in its layers is named after it – Convolutional Neural Network (CNN). Instead of the classic operation of matrix multiplication (essentially summation of weights) in traditional neural networks, CNNs use “filters” (matrices) to convolve through the input data, creating “feature maps”. These feature maps are “automatically generated” features created by the convolutional layer, containing the likelihood (probability) that a specific filter recognized its pattern in a specific section of the input, creating the powerful locality-continuation that CNNs utilize. These feature maps then go through a “pooling” layer. Pooling layers, in essence, reduce the size of the data being propagated in the network. The idea is that if the filter recognized its pattern in an area in the input (either local area such as in an image, or a temporal “area” such as in a time-series sampling), it isn’t quite as important to remember where exactly the pattern was found, just that it **was** found.
The last stage of a CNN (before training the classifier that comes after it) is the “flatten” layer. The flatten layer takes the feature map(s) that came before it (which are essentially

matrices of probabilities) and, as its name suggests, flattens them one after the other, from first row, first column, through the columns and then the rows. This layer effectively acts as the features generated by the convolutional layer(s) that preceded it, and connects to dense layer(s) (fully connected) for the classification.

By “automatic feature extracting”, CNNs essentially eliminate the need for domain-specific knowledge/expertise for problems that have **continuity** in their nature, such as images (spatial continuity) or time-series (temporal continuity).

Another type of neural networks, Recurrent Neural Network (RNN), are a type of artificial neural network designed to effectively process sequential data by maintaining an internal memory. Unlike traditional feedforward neural networks, RNNs can retain information from previous inputs, allowing them to consider context and temporal dependencies within the data. This memory mechanism enables RNNs to make decisions based on both current and past inputs, making them well-suited for time-series analysis. RNNs are structured in a way that allows the network to take in sequential inputs, process them while retaining information about past inputs, and generate corresponding outputs, making them a powerful tool for modeling sequential data.

Traditional (“vanilla”) RNNs have multiple limitations, rendering them unusable in most real-world applications. These limitations include:

- Vanishing/exploding gradients.
- Short-term memory limitations: difficulties in capturing long-term dependencies, by “forgetting” information from earlier time steps in the time-series samples.
- Lack of parallelization: the data processing is done sequentially, which completely eliminates any option for parallel computing the different stages of the model training.

To combat some of these limitations, an advanced variant of RNN has been developed. Long Short-Term Memory Neural Networks (LSTM) were designed to overcome both the vanishing/exploding gradients problem, and the short-term only limitation. LSTMs have 3 gates: the input gate, the forget gate, and the output gate. The forget gate is responsible for holding the long-term memory (with contradiction to its terminology name).

In this project, I will learn about the different kinds of neural networks described above, and experiment with them and with the Human Activity Recognition data set.

Data Set Information

The experiments have been carried out with a group of 30 volunteers within an age bracket of 19-48 years. Each person performed six activities (walking, walking upstairs, walking downstairs, sitting, standing, laying) wearing a smartphone (Samsung Galaxy S II) on the waist. Using its embedded accelerometer and gyroscope, 3-axial linear acceleration and 3-axial angular velocity were captured at a constant rate of 50Hz. The experiments have been video-recorded to label the data manually. The obtained dataset has been randomly partitioned into two sets, where 70% of the volunteers were selected for generating the training data and 30% the test data.

The sensor signals (accelerometer and gyroscope) were pre-processed by applying noise filters and then sampled in fixed-width sliding windows of 2.56 sec and 50% overlap (128 readings/window). The sensor acceleration signal, which has gravitational and body motion components, was separated using a Butterworth low-pass filter into body acceleration and gravity. The gravitational force is assumed to have only low frequency components, therefore a filter with 0.3 Hz cutoff frequency was used. From each window, a vector of features was obtained by calculating variables from the time and frequency domain.

For each record in the dataset, it is provided:

- Triaxial acceleration from the accelerometer (total acceleration) and the estimated body acceleration.
- Triaxial Angular velocity from the gyroscope.
- A 561-feature vector with time and frequency domain variables (**not used in this project**).
- Its activity label.
- An identifier of the subject who carried out the experiment.

From Human Activity Recognition Using Smartphones Data Set

Project's Goal (taken from the project's proposition):

- A suite of Convolutional Neural Network models for human activity recognition from accelerometer data.
- A suite of Long Short-Term Memory Neural Network models for human activity recognition from accelerometer data.
- Comparison with state-of-the-art methods on Activity Recognition Using Smartphones dataset.

Motivation

By utilizing the methods listed above, a model can try to classify a person's activity either in real-time or in retrospect. These could prove beneficial in areas such as healthcare and senior care, or in recreational activities such as life-logging and fitness. Another example could be broadening the range of activities recognized, and including more daily activities such as driving or riding a bicycle. This, in turn, may allow a better tailored experience using smart devices (like a smart phone recognizing when the user is driving), or allow detection of emergencies such as car crashes.

In addition to real-world applications, the project allows me to experience and learn about the world of data science, machine learning, and deep learning, as an academic project, as part of my BSc. In particular, learn to use Convolutional Neural Networks and Long Short-Term Memory Neural Networks for time-series classification problems.

Literature Review

Integrated Development Environment and Programming Language

Machine Learning and Deep Learning projects are written in different programming languages such as C, R, Python, Matlab. For this project, Python was chosen. Any required information and know-how was acquired by: previous knowledge in coding and in Python, Jason Brownlee's books, and online resources such as guides, videos, forums, etc.

The are multiple popular IDEs for Python development, and specifically for Machine Learning and/or Deep Learning development, such as Jupyter Notebook.

The IDE used in this project is Google Colaboratory. Specifically designed for Data Science, Machine Learning, and Deep Learning, it allows using Python and Jupyter Notebooks on the web browser, accessing CPUs, GPUs, and TPUs provided by Google. It does not require any installations of libraries locally.

Code Libraries

There are libraries designed for Machine Learning and Deep Learning development in Python, meant for "black-box implementation" of concepts and models, such as Neural Network layers. This project utilizes the following libraries and packages:

Numpy – adding support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays.

Pandas – data manipulation and analysis. In particular, it offers data structures and operations for manipulating numerical tables and time series.

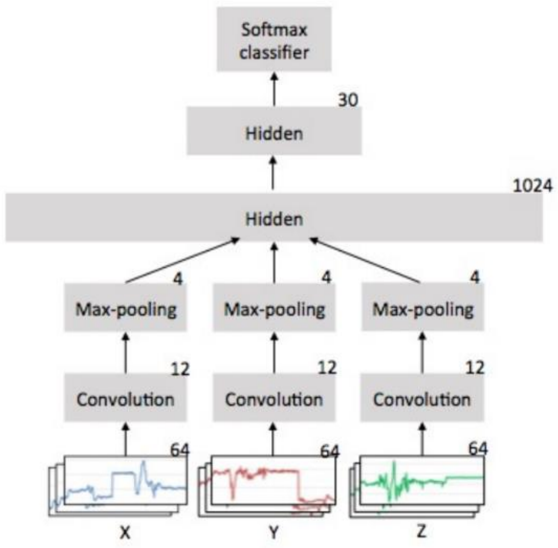
TensorFlow – free and open-source software library for machine learning and artificial intelligence. It can be used across a range of tasks but has a particular focus on training and inference of deep neural networks.

Keras – open-source library that provides a Python interface for artificial neural networks. Keras acts as an interface for the TensorFlow library.

Academic Papers

The first iterations of human activity recognition relied on machine learning principles, requiring explicitly defining the features of the data. Some of these heuristically-defined features perform well in recognizing one activity, but perform poorly for others[1]. Thus, using Convolutional Neural Networks (CNN) allows extraction of human activity features without any domain knowledge, and may lead to greater results in terms of accuracy.

In the paper *Convolutional Neural Networks for Human Activity Recognition using Mobile Sensors*[1], the authors utilize this very principle. They suggest a 1D CNN architecture that provides each axis with its own convolutional and pooling layer, extracting features that rely on each axis independently.

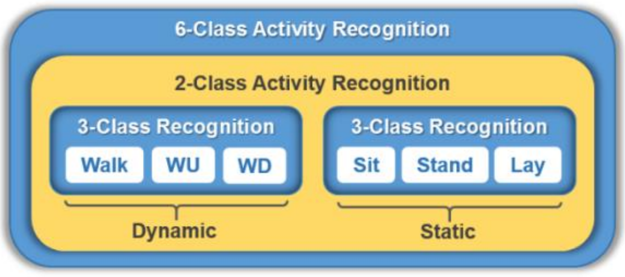


In addition, they describe a principle of “partial weight-sharing”, referring to a more relaxed constraint of weight sharing.

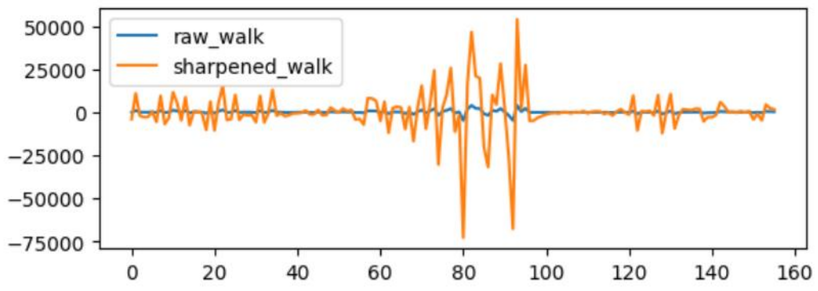
> *In image processing task, full weight sharing is suitable because the same image pattern can appear at any position in an image. However, in AR, because different patterns appear in different frame, the signal appearing at different units may behave quite differently. Therefore, it may be better to relax the weight sharing constraint.*

Another example of trying to better the results is applying a "divide and conquer" approach – instead of straightforwardly recognizing the individual activities using a single 6-class classifier, the authors apply a divide and conquer approach and build a two-stage activity recognition process, where abstract activities, i.e., dynamic and static, are first recognized using a 2-class (binary) classifier, and then individual activities are recognized using 3-class classifiers.

[1] Convolutional Neural Networks for Human Activity Recognition using Mobile Sensors, Ming Zeng, Le T. Nguyen, Bo Yu, Ole J. Mengshoel, published November 2014.



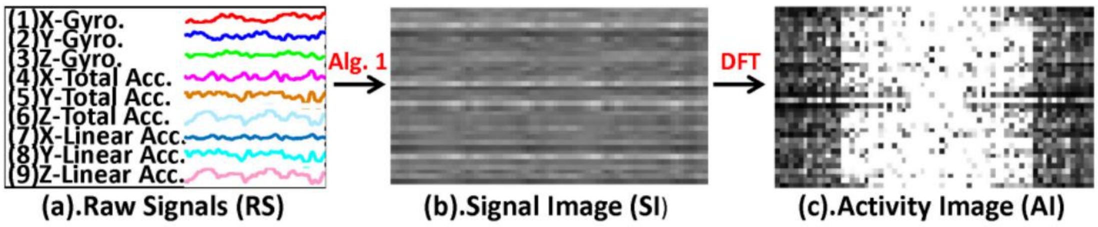
In addition to this approach, the authors performed data sharpening on test data alone for improving the recognition, instead of on the train data as well, as have been done in prior works.[2]

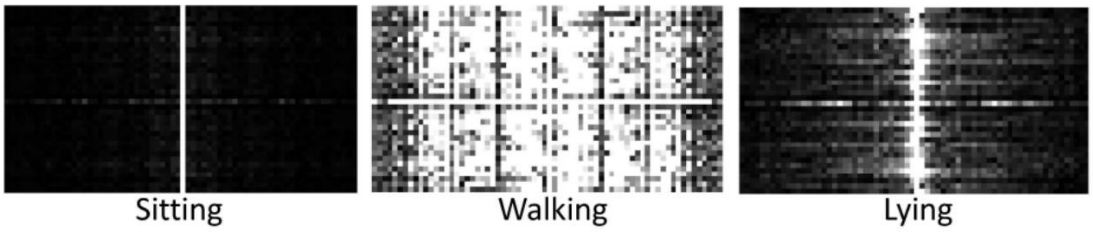


Our approach leverages a two-stage learning of multiple 1D CNN models; we first build a binary classifier for recognizing abstract activities, and then build two multi-class 1D CNN models for recognizing individual activities. We then introduce test data sharpening during prediction phase to further improve the activity recognition accuracy. While there have been numerous researches exploring the benefits of activity signal denoising for HAR, few researches have examined the effect of test data sharpening for HAR. We evaluate the effectiveness of our approach on two popular HAR benchmark datasets, and show that our approach outperforms both the two-stage 1D CNN-only method and other state of the art approaches.

[2] Divide and Conquer-Based 1D CNN Human Activity Recognition Using Test Data Sharpening, Heeryon Cho, Sang Min Yoo, published April 2018.

In the paper *Human Activity Recognition using Wearable Sensors by Deep Convolutional Neural Networks*, the authors describe a method for altering the data samples from the sensors in a way to create images from the data (algorithm and then Discrete Fourier Transform). They claim to better utilize the power of CNNs of finding patterns in locality-continuous data[6].





With the images above, the authors create a 2D CNN architecture to classify activities. In addition, in cases of uncertainty, they apply an ML model to aid in classification, SVM. This proved to improve the test accuracy on the UCI data set from 95.18% (DCNN) to 97.59% (DCNN+ as they termed, DCNN + SVM).

Since human activities are made of complex sequences of motor movements, and based on recent success of recurrent neural networks (RNN) for time series domains, another approach emerged – using Long Short-Term Memory (LSTM) recurrent units.[3] The results of this approach outperform competing deep non-recurrent networks on the same dataset by 4% on average, and outperform some of the previous reported results by up to 9%. In general, deep recurrent neural networks (DRNN) are capable of capturing long-range dependencies, which non-recurrent neural networks lack.[4]

In the paper *Deep Recurrent Neural Networks for Human Activity Recognition*, the authors suggest using Long Short-Term Memory (LSTM) Neural Networks.

An RNN is neural network architecture that contains cyclic connections, which enable it to learn the temporal dynamics of sequential data.

In addition, they describe how a regular RNN may experience the vanishing or exploding gradient problems (the phenomenon of gradients getting too small or too large during back propagation, restricting the learning of the model). To combat these issues, LSTM-based RNNs can be used.

LSTM-based RNNs can model temporal sequences and their wide-range dependencies by replacing the traditional nodes with memory cells that have internal and outer recurrence.

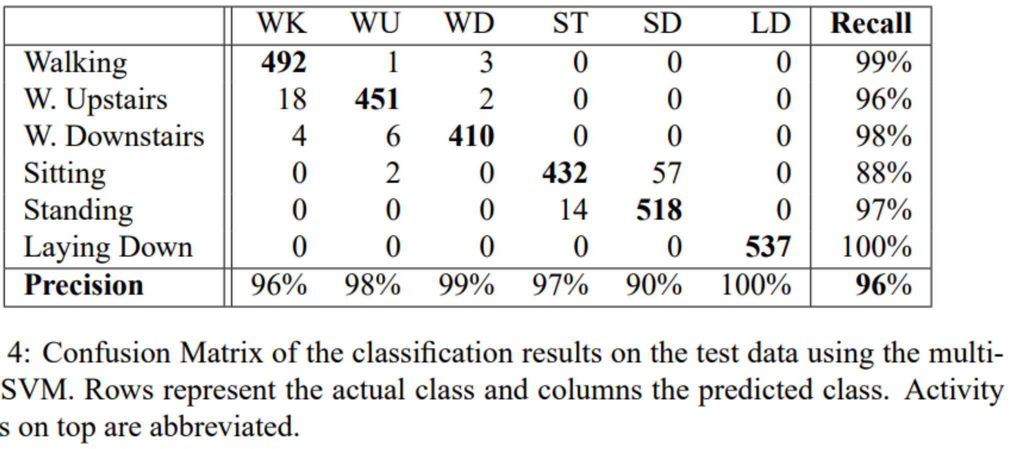
[3] Deep Convolutional and LSTM Recurrent Neural Networks for Multimodal Wearable Activity Recognition, Francisco Javier Ordonez, Daniel Roggen, published January 2016.
[4] Deep Recurrent Neural Networks for Human Activity Recognition, Abdulmajid Murad, Jae-Young Pyun, November 2017.

Research and Testing

Specifications

The expected product of this project is a software package that implements different models, resulting in different trained models and comparison measures for each method.

The original paper published with the data set used SVM for classification, and achieved the following results (taken from the paper):

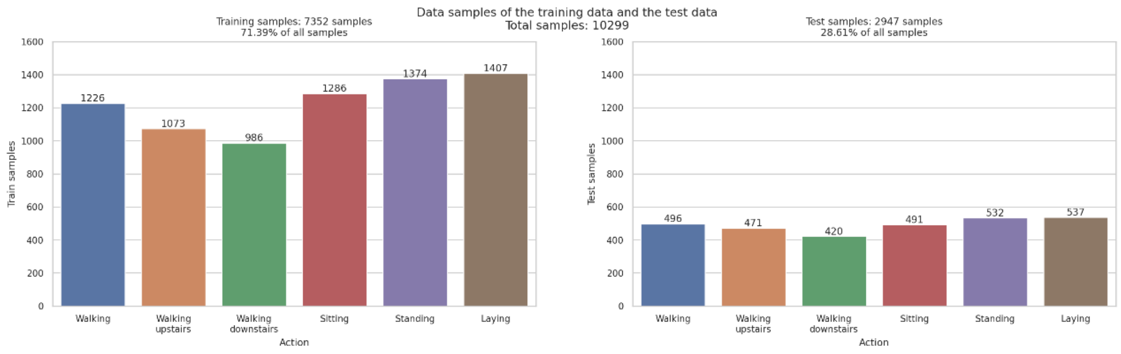


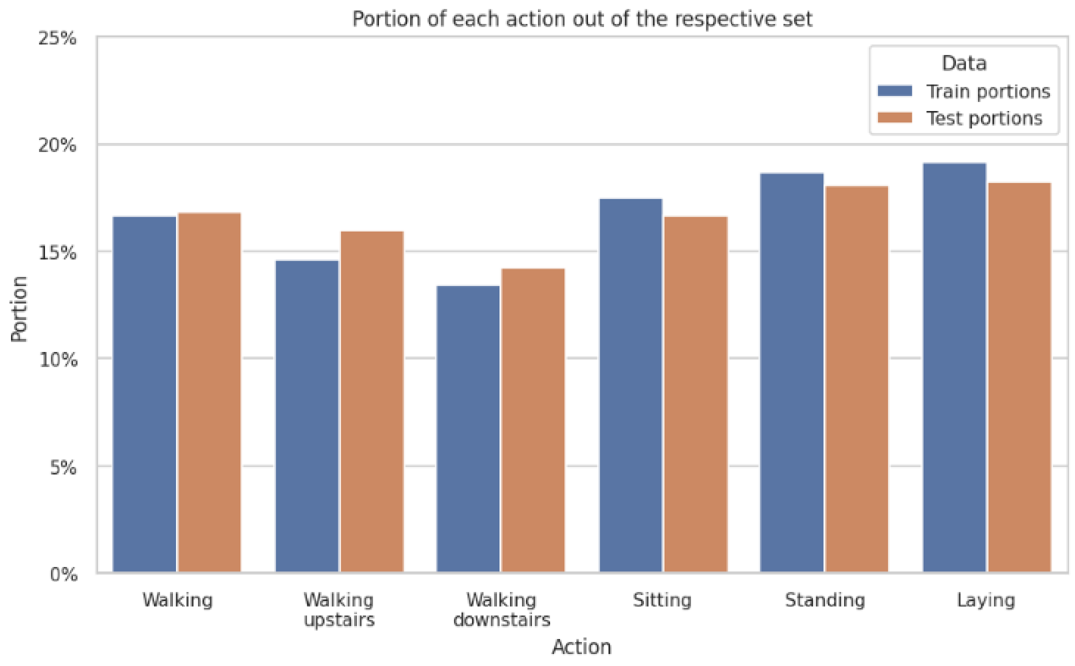
	WK	WU	WD	ST	SD	LD	**Recall**
Walking	**492**	1	3	0	0	0	99%
W. Upstairs	18	**451**	2	0	0	0	96%
W. Downstairs	4	6	**410**	0	0	0	98%
Sitting	0	2	0	**432**	57	0	88%
Standing	0	0	0	14	**518**	0	97%
Laying Down	0	0	0	0	0	**537**	100%
Precision	96%	98%	99%	97%	90%	100%	**96%**

Table 4: Confusion Matrix of the classification results on the test data using the multi-class SVM. Rows represent the actual class and columns the predicted class. Activity names on top are abbreviated.

A Public Domain Dataset for Human Activity Recognition Using Smartphones[5]

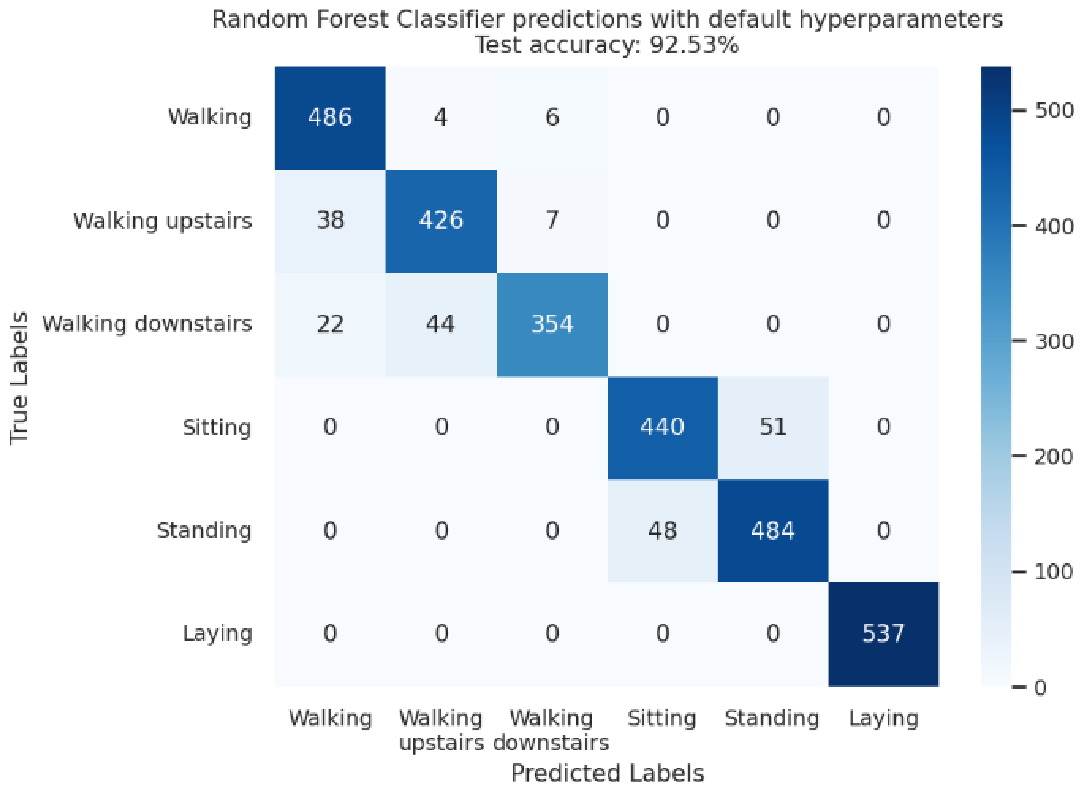
Attached is a Jupyter notebook titled "EDA" (Exploratory Data Analysis) that explores data samples in the data set (EDA.ipynb).





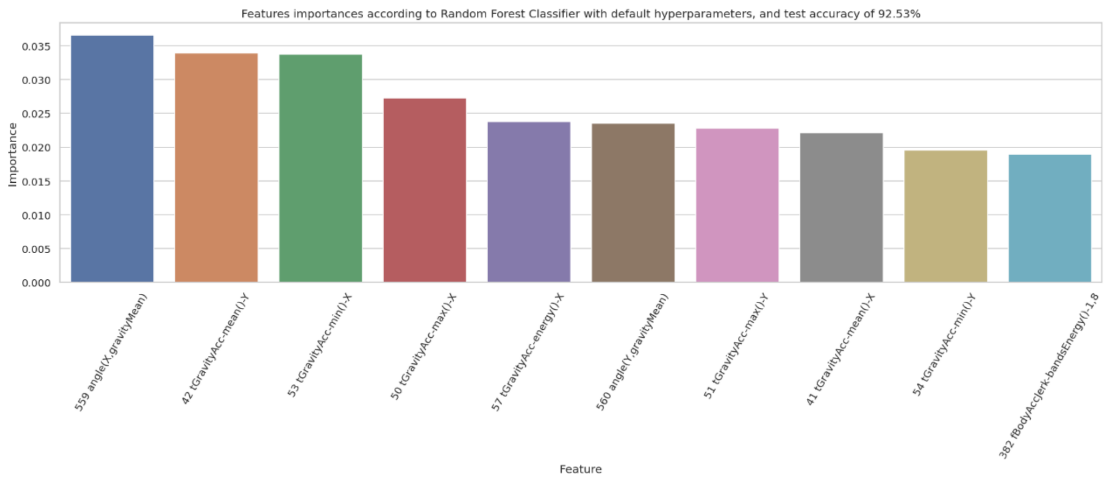
With a 70% to 30% training-set to test-set ratio, we can see a relatively balanced data set. Each class has a similar representation (proportionally) in both train and test sets.

Out of curiosity, I trained a model of a Random Forest Classifier (RFC) from scikit-learn library with the human-engineered features that the data set provides. The default parameters reached ~92% test data accuracy (model at EDA.ipynb):

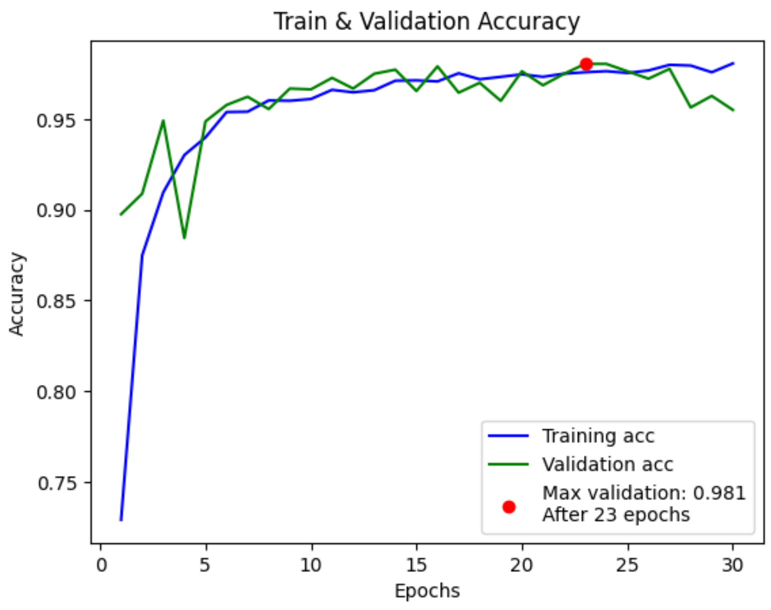


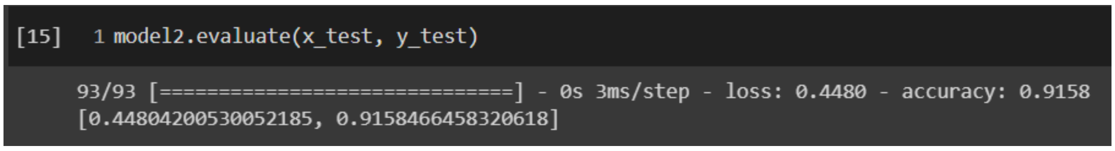
True \ Predicted	Walking	Walking upstairs	Walking downstairs	Sitting	Standing	Laying
Walking	486	4	6	0	0	0
Walking upstairs	38	426	7	0	0	0
Walking downstairs	22	44	354	0	0	0
Sitting	0	0	0	440	51	0
Standing	0	0	0	48	484	0
Laying	0	0	0	0	0	537

In addition, I was curious what were the most influential features, which is why I chose the RFC model. The model has an attribute of "feature_importances_" that we can easily access to visualize the different features influence in the classification.



Another curious attempt was training a simple neural network model with the human-engineered features. The architecture was deliberately not a deep neural network architecture, consisting of 1 input layer (561 nodes, 1 for each input), 1 hidden layer (100 nodes), and 1 output layer (softmax, 6 nodes). The test accuracy was 91.58% (model at ann_features.ipynb).





```
[15]   1 model2.evaluate(x_test, y_test)

     93/93 [==============================] - 0s 3ms/step - loss: 0.4480 - accuracy: 0.9158
     [0.44804200530052185, 0.9158466458320618]
```

It will be interesting to see how a simple CNN model performs, and if it's anything comparable, CNNs will show their strength in "automatic" feature extraction.

Convolutional Neural Network Models

Background

When applied to time-series data, Convolutional Neural Networks (CNNs) exhibit a different approach compared to their usage with image data. CNNs for time-series data apply a similar concept of convolution but on the temporal dimension rather than the spatial dimension. These networks use filters to extract temporal features and patterns, sliding across the time-series data to capture relevant information. By doing so, CNNs can identify and analyze sequential patterns within the time-series data, making them effective in tasks such as time-series forecasting, anomaly detection, and signal processing.

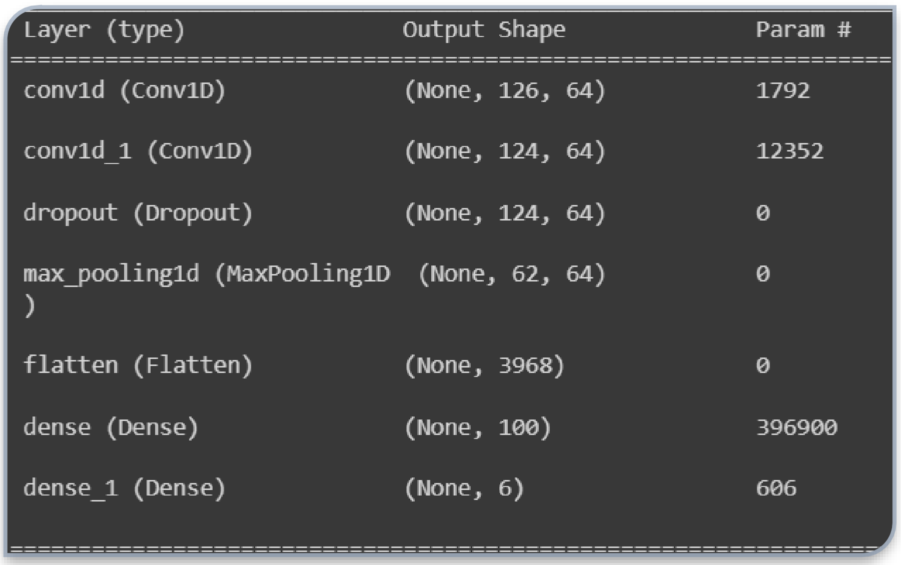
Moreover, in the context of time-series data, CNNs rely on matrix multiplications to process the temporal relationships within the data, enabling them to learn complex temporal patterns and dependencies. This matrix-based approach enhances the network's capability to comprehend sequential dependencies and capture long-range dependencies within the time-series, thus improving the accuracy of predictions and analyses.

In contrast, traditional Artificial Neural Networks (ANNs) lack the ability to effectively capture sequential dependencies in time-series data, as they primarily focus on processing individual data points without considering their temporal relationships. As a result, CNNs are particularly advantageous in time-series analysis, as they can effectively learn from sequential data, model temporal dependencies, and make accurate predictions, thereby facilitating better insights and decision-making in various time-series-related applications.

Perhaps the most advantageous capability of CNNs is their “automatic” feature extraction, based on the concept of filters sliding across the time-series data. This capability enables capturing features that cannot be easily described by humans (such as the question “what are the best features to describe a dog compared to a cat?”), or when the researcher has limited or no domain-specific knowledge for manual feature extraction (such as in Human Activity Recognition, having knowledge of signal processing).

CNN Model I

(attached in cnn_1.ipynb)



```
Layer (type)                 Output Shape              Param #
=================================================================
conv1d (Conv1D)              (None, 126, 64)           1792

conv1d_1 (Conv1D)            (None, 124, 64)           12352

dropout (Dropout)            (None, 124, 64)           0

max_pooling1d (MaxPooling1D  (None, 62, 64)            0
)

flatten (Flatten)            (None, 3968)              0

dense (Dense)                (None, 100)               396900

dense_1 (Dense)              (None, 6)                 606

=================================================================
```

- 2 consecutive convolutional layers, both with “relu” activation, and the latter having “dropout” regularization of 50%.
- 1 max pooling layer with pool size 2.
- 3 fully connected layers for classification:
  - Flatten ($62 \cdot 64 = 3968$).
  - Dense with 100 nodes with relu activation function.
  - Dense with softmax for classification (6 classes) – the output layer.
- Optimization algorithm: Adam (Adaptive Moment Estimation), with a learning rate of 0.001.
- Loss function: Categorical Cross-Entropy (aka Log-Loss), multi-class classification, for one-hot encoded labels.
- Metrics: accuracy (training).

With each of the following parameters (listed in the tables below), the model was evaluated over the test data 10 times. Below are the mean test accuracy and the standard deviation of test accuracy over the 10 evaluations (in the format of $mean \pm std$).

Note: since 10 evaluations may still have fluctuations (because of the randomness in the model, such as batch size, weights initialization, etc.), the results may vary between evaluations with the same hyperparameters (a simple solution would be to run the evaluations more times, but at the cost of overall evaluation time).

	Base parameters	Lower learning rate (1e-3 → 1e-2)	Increase epochs #1 (10 → 50)	Increase epochs #2 (10 → 300) (Runtime)	Increase batch size (32 → 512)	Increase nodes in dense layer (100 → 1024)
Test Accuracy	$90.780\% \pm 1.028\%$	$89.491\% \pm 1.669\%$	$91.727\% \pm 0.609\%$	$92.243\% \pm 0.560\%$	$89.515\% \pm 0.707\%$	$90.618\% \pm 1.344\%$

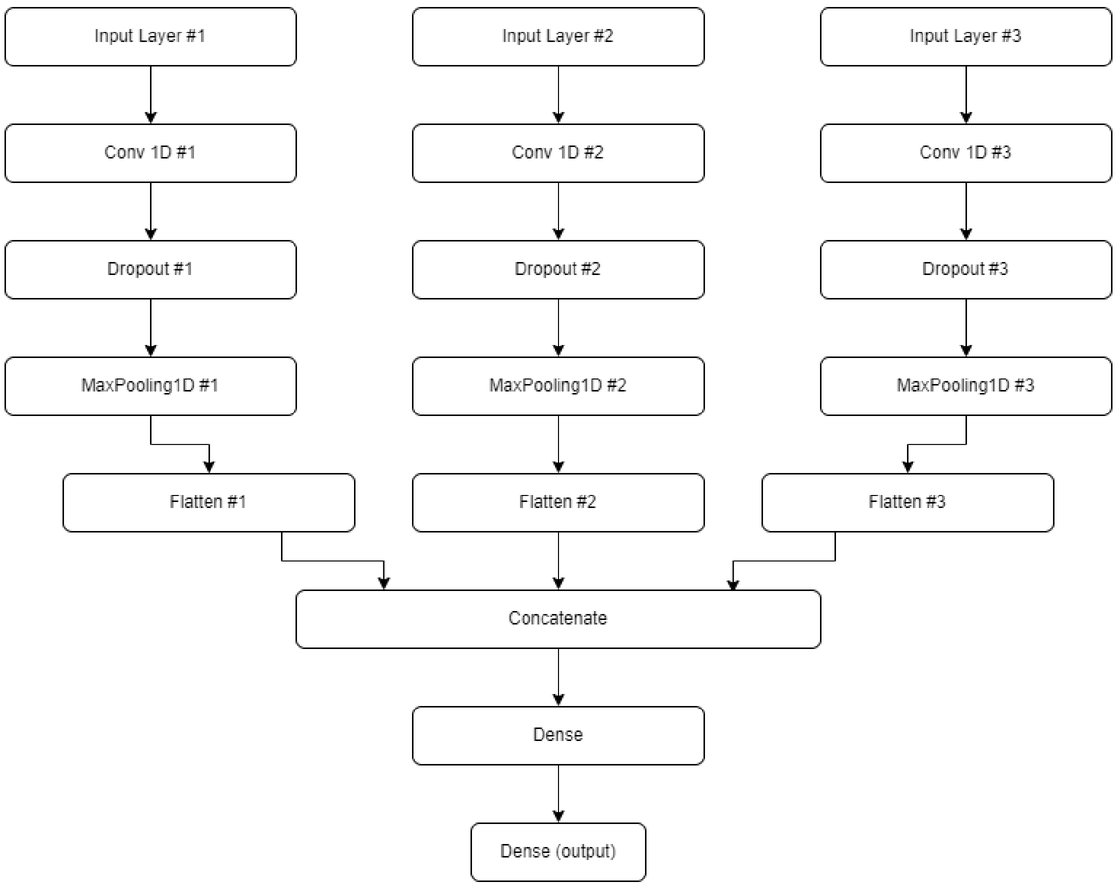
	Add dense layers (1024, 512, 256, 128, 64)	Add 1 conv + 1 max pool layers	Increase filter size (1x3 → 1x7)	Increase pooling size (2 → 4)	Increase dropout in 2nd conv layer (0.5 → 0.8)	Decrease dropout in 2nd conv layer (0.5 → 0.3)
Test Accuracy	$90.818\% \pm 1.499\%$	$90.665\% \pm 0.897\%$	$90.740\% \pm 1.870\%$	$90.638\% \pm 1.072\%$	$90.638\% \pm 0.690\%$	$90.098\% \pm 1.739\%$

- Loss function Mean Squared Error (MSE): $90.828\% \pm 0.844\%$
- Loss function Mean Absolute Error (MAE): $71.944\% \pm 6.957\%$

We can see that the base model achieved an impressive $90.780\% \pm 1.028\%$ test data accuracy, comparable to the simple neural network (that used the human-engineered features) described earlier. This architecture does not require any human-engineered features, and does not require any domain-specific knowledge. In addition, increasing the epochs to 300 resulted in a significant increase in the data accuracy, at the cost of training time (approximately 40 minutes using Google Colab GPU).

CNN Model II – Multi-headed CNN

(attached in cnn_2.ipynb)



- 3 different heads, each with:
    - a conv layer
    - a dropout layer
    - a max pooling layer
    - a flatten layer
- All 3 flattened feature maps are then concatenated into one dense layer.
- 2 dense layers:
    - Dense layer(s) to act as the classifier with relu activation function.
    - Dense with softmax for classification (6 classes) – the output layer.
- Optimization algorithm: Adam (Adaptive Moment Estimation), with a base learning rate of 0.001.
- Loss function: Categorical Cross-Entropy (aka Log-Loss), multi-class classification, for one-hot encoded labels.
- Metrics: accuracy (training).

Multi-headed CNN – axis grouping: group according to axis:

- one head gets x-axis data
- another gets y-axis data
- the last gets z-axis data

The base results with the above-listed hyperparameters are $88.653 \pm 0.928\%$, which are clearly worse than the previous architecture ($90.78\% \pm 1.028\%$).

	Base parameters	**Increase learning rate (1e-3 → 1e-2)**	**Decrease learning rate (1e-3 → 1e-4)**	**Increase filter size (3 → 5)**	**Increase filter size (3 → 7)**	**Increase nodes in dense layer (100 → 1024)**
Test Accuracy	$88.653 \pm 0.928\%$	$88.205 \pm 0.682\%$	$87.431\% \pm 0.760\%$	$88.968\% \pm 0.714\%$	$90.061\% \pm 0.582\%$	$89.898\% \pm 0.889$
	Add 1 conv + 1 max pool layers to each head	**Increase pooling size (1x2 → 1x5)**	**Increase pooling size (1x2 → 1x7)**	**Increase dropout in conv layer (0.5 → 0.75)**	**Decrease dropout in conv layer (0.5 → 0.25)**	
Test Accuracy	$89.410\% \pm 0.902\%$	$89.338\% \pm 0.638\%$	$89.827\% \pm 0.806\%$	$89.813\% \pm 0.950$	$88.269\% \pm 1.335\%$	
	Increase epochs #1 (10 → 50)	**Increase epochs #2 (10 → 300)**	**Decrease filters (64 → 32)**	**Increase filters (64 → 128)**		
Test Accuracy	$89.698\% \pm 0.839\%$	$90.136\% \pm 0.747\%$	$88.836\% \pm 0.923\%$	$89.002\% \pm 0.742\%$		

Multi-headed CNN – logical grouping: group according to sensor type:

- one head gets total_acc_x/y/z
- another gets body_acc_x/y/z
- the last gets body_gyro_x/y/z

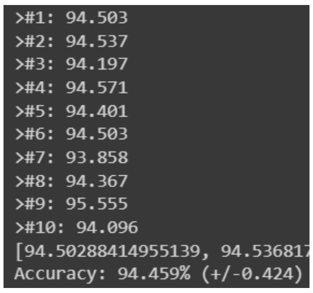
The base results with the above-listed hyperparameters are $90.543 \pm 1.110\%$, which are slightly worse than the previous architecture ($90.78\% \pm 1.028\%$).

	Base parameters	**Increase learning rate (1e-3 → 1e-2)**	**Decrease learning rate (1e-3 → 1e-4)**	**Increase filter size (3 → 5)**	**Increase filter size (3 → 7)**	**Increase nodes in dense layer (100 → 1024)**
Test Accuracy	$90.543\% \pm 1.110\%$	$89.002\% \pm 0.851\%$	$89.206\% \pm 0.637\%$	$90.618\% \pm 1.047\%$	$90.665\% \pm 1.769\%$	$90.546\% \pm 1.306\%$
	Add 1 conv + 1 max pool layers to each head	**Increase pooling size (1x2 → 1x5)**	**Increase pooling size (1x2 → 1x7)**	**Increase dropout in conv layer (0.5 → 0.75)**	**Decrease dropout in conv layer (0.5 → 0.25)**	
Test Accuracy	$91.476\% \pm 1.166\%$	$90.475\% \pm 1.268\%$	$90.675\% \pm 0.762\%$	$90.000\% \pm 1.055\%$	$90.964\% \pm 0.732\%$	

	Increase epochs #1 (10 → 50)	Increase epochs #2 (10 → 300)	Decrease filters (64 → 32)	Increase filters (64 → 128)		
Test Accuracy	$91.856\% \pm 0.579\%$	$92.226\% \pm 0.367\%$	$90.241\% \pm 1.067\%$	$91.418\% \pm 0.836\%$		

After applying all the tweaks that provided improvements over the base parameters: increase filter size, add 1 conv & 1 max-pool layers to each head, decrease dropout, and increase the number of filters, we get a test accuracy of (still with "only" 10 epochs) $92.681\% \pm 1.441\%$, the highest so far.

After running the same parameters with 500 epochs instead of 10, the test accuracy over 10 attempts was $\mathbf{94.459}\% \pm \mathbf{0.424}\%$. This model required a total of 113 minutes of training time for all 10 attempts (11.3 minutes on average for training the model over 500 epochs).


```
>#1: 94.503
>#2: 94.537
>#3: 94.197
>#4: 94.571
>#5: 94.401
>#6: 94.503
>#7: 93.858
>#8: 94.367
>#9: 95.555
>#10: 94.096
[94.50288414955139, 94.536817
Accuracy: 94.459% (+/-0.424)
```

Multi-headed CNN – deep network (multiple layers):

After finding superior results in the model above with the adjusted parameters, I tried increasing the number of fully-connected layers for classification (dense layers), from 1 layer with 100 nodes, to 5 layers, with 1024, 512, 256, 128, 64 nodes each one, before the final softmax layer. The test results were $91.686\% \pm 1.578\%$. This suggests that model complexity may have a bad effect on the test results, like over-fitting to the training set.

Long Short-Term Memory Models

Background

Long Short-Term Memory (LSTM) Neural Networks are a specialized form of Recurrent Neural Networks (RNNs) designed to handle long-range dependencies in sequential data like time-series, and to avoid the exploding/vanishing gradient problem that vanilla RNNs face. Compared to Artificial Neural Networks (ANNs), LSTMs excel at capturing temporal patterns by using a more complex memory cell structure, including input, forget, and output gates. These gates allow LSTMs to selectively retain or discard information over varying time intervals, making them highly effective in modeling sequential data that involves long-term dependencies.

LSTMs may be a good fit for the Human Activity Recognition problem since the model can support multiple parallel sequences of input data, such as each axis of the accelerometer and gyroscope data.

According to Brownlee: *the benefit of using LSTMs for sequence classification is that they can learn from the raw time series data directly, and in turn do not require domain expertise to manually engineer input features. The model can learn an internal representation of the time series data and ideally achieve comparable performance to models fit on a version of the dataset with engineered features.*

LSTM Model I

A basic LSTM model with the following architecture:

- 1 LSTM layer with 100 units.
- 1 dropout layer with a rate of 50%.
- 2 dense layers:
    - 1 dense layer with 100 nodes to act as the classifier with identity activation function.
    - 1 dense layer with softmax for classification (6 classes) – the output layer.
- Optimization algorithm: Adam (Adaptive Moment Estimation), with a base learning rate of 0.001.
- Loss function: Categorical Cross-Entropy (aka Log-Loss), multi-class classification, for one-hot encoded labels.
- Metrics: accuracy (training).

	Base parameters	**Increase learning rate (1e-3 → 1e-2)**	**Decrease learning rate (1e-3 → 1e-4)**	**Decrease units (100 → 30)**	**Increase units (100 → 300)**	**Increase nodes in dense layer (100 → 512)**
Test Accuracy	$89.186\% \pm 2.228\%$	$71.049\% \pm 13.113\%$	$73.773\% \pm 1.914\%$	$83.848 \pm 4.098\%$	$74.384\% \pm 16.446\%$	$90.071\% \pm 1.784\%$
	Activation function (identity → tanh)	**Decrease dropout (0.5 → 0.25)**	**Increase dropout (0.5 → 0.75)**	**Increase epochs #1 (10 → 50)**		
Test Accuracy	$90.129\% \pm 1.770\%$	$88.663\% \pm 4.701\%$	$78.799\% \pm 7.182\%$	$91.418\% \pm 0.862\%$		

This is, perhaps, the most basic form of an LSTM network we can implement, and it acts as a starting point for additional networks that rely on LSTM layers.

Considering that using a simple ANN for classification when utilizing the human-engineered features resulted in 91.5% test accuracy, LSTM shows its strength in not-requiring doman-specific expertise for creating features for the model.

LSTM Model II – Bidirectional LSTM

The key difference between a bidirectional LSTM and a regular LSTM lies in how they processs input sequences. A regular LSTM processes the sequence from the beginning to the end, taking into account the previous information to predict the next steps. Conversely, a bidirectional LSTM processes the sequence in both directions, from the beginning to the end and from the end to the beginning, essentially capturing information from both past and future contexts.

By capturing information from past and future contexts (which is not possible in real-time classification, but does fit our need for retrospect classification of data samples), a bidirectional LSTM can better understand the temporal dynamics of the data.

The disadvantage to using bidirectional LSTMs lies in their training time. Since both directions are used for calculations, the training time is noticeably longer when compared to regular LSTMs, which are already relatively longer being a type of RNN.

Thanks to the "extra information" bidirectional LSTMs process, we may expect better results than a regular LSTM, even if marginally.

Below are results from the following bidirectional LSTM model:

- 2 bidirectional LSTM layers, one with 64 units and 50% dropout for each unit, and the next with 32 units.
- 2 dense layers:
  - 1 dense layer to act as the classifier with relu activation function.
  - 1 dense with softmax for classification (6 classes) – the output layer.
- Optimization algorithm: Adam (Adaptive Moment Estimation), with a base learning rate of 0.001.
- Loss function: Categorical Cross-Entropy (aka Log-Loss), multi-class classification, for one-hot encoded labels.
- Metrics: accuracy (training).

	Base parameters	**Increase learning rate (1e-3 → 1e-2)**	**Decrease learning rate (1e-3 → 1e-4)**	**Decrease units (100 → 30)**	**Increase units (100 → 300)**	**Increase nodes in dense layer (100 → 512)**
Test Accuracy	$90.322\% \pm 1.063\%$	$86.403\% \pm 6.094\%$	$76.379\% \pm 2.965\%$	$89.498\% \pm 2.438\%$	$90.227\% \pm 1.119\%$	$90.560\% \pm 1.042\%$
	Activation function (relu → tanh)	**Decrease dropout (0.5 → 0.25)**	**Increase dropout (0.5 → 0.75)**	**Increase epochs #1 (10 → 50)**		
Test Accuracy	$89.403\% \pm 2.041\%$	$89.946\% \pm 1.012\%$	$84.140\% \pm 5.815\%$			

Each time the model is trained takes ~15-20 minutes (any of the cells in the above table).

The bidirectional LSTM provides slightly better results when compared to the regular LSTM model. This may suggest that there are "backward temporal dependencies" in the time-series samples of the dataset.

LSTM Model III – Stacked LSTMs

Stacked LSTMs refer to a configuration where multiple LSTM layers are stacked on top of each other within a neural network architecture. Each LSTM layer in the stack processes the entire sequence and passes its output to the next LSTM layer. This stacking allows the model to learn complex temporal representations at different levels of abstraction, potentially leading to improved performance in capturing intricate temporal dependencies and patterns in the data.

While stacked LSTMs can improve the model's ability to capture complex temporal dependencies, they may have increased computational complexity, and longer training times for having multiple LSTM layers.

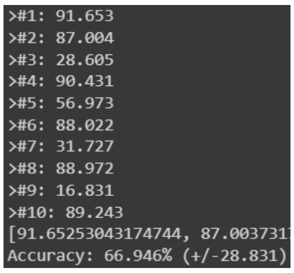
“building a deep RNN by stacking multiple recurrent hidden states on top of each other. This approach potentially allows the hidden state at each level to operate at different timescale”

-How to Construct Deep Recurrent Neural Networks, 2013

Below are results from the following stacked LSTM model:

- 4 layers of LSTM, each with 100 units, and each unit with a dropout rate of 50%.
- 2 dense layers:
    - 1 dense layer to act as the classifier with relu activation function.
    - 1 dense with softmax for classification (6 classes) – the output layer.
- Optimization algorithm: Adam (Adaptive Moment Estimation), with a base learning rate of 0.001.
- Loss function: Categorical Cross-Entropy (aka Log-Loss), multi-class classification, for one-hot encoded labels.
- Metrics: accuracy (training).

	Base parameters	**Increase learning rate (1e-3 → 1e-2)**	**Decrease learning rate (1e-3 → 1e-4)**	**Decrease units (100 → 30)**	**Increase units (100 → 300)***	**Increase nodes in dense layer (100 → 512)**
Test Accuracy	89.209% ± 3.658%	22.806% ± 6.018%	78.633% ± 4.803%	87.981% ± 2.466%	66.946% ± 28.831%	89.620% ± 1.299%
	Activation function (identity → relu)	**No dropout**	**Decrease dropout (0.5 → 0.25)**	**Increase dropout (0.5 → 0.75)**	**Decrease LSTM layers (4 → 2)**	**Increase LSTM layers (4 → 6)**
Test Accuracy	86.291% ± 8.250%	89.644% ± 3.070%	89.301% ± 3.710%	89.148% ± 0.979%	90.594% ± 1.700%	86.342% ± 10.251%



```
>#1: 91.653
>#2: 87.004
>#3: 28.605
>#4: 90.431
>#5: 56.973
>#6: 88.022
>#7: 31.727
>#8: 88.972
>#9: 16.831
>#10: 89.243
[91.65253043174744, 87.003731
Accuracy: 66.946% (+/-28.831)
```

Each time the model is trained takes ~10-15 minutes (any of the cells in the above table).

*When the number of units was 300, some models were good (87% +) and some were exceptionally low (16%) (the picture to the right). One possible reason could be that the long-term memory (forget gate) may have been "zeroed-out" during training in the low test accuracy models, and effectively being completely left out of the model.

The Stacked LSTM model did not show better results when compared to previous models. Rather, it displayed a more random nature in the test accuracy results, as seen in some of the above tweaked hyperparameters in the table, such as awfully low test accuracy when the learning rate increased (other models did not plummet similarly with the same changes), or very polarized results when the number of units increased in each layer.

CNN-LSTM Model

CNN-LSTM neural networks combine the strengths of Convolutional Neural Networks (CNNs) and Long Short-Term Memory (LSTM) networks to (hopefully) effectively address the activities classification. In this architecture, the CNN layers serve as feature extractors that operate on the input data, allowing the network to capture local patterns and temporal dependencies. The output from the CNN layers is then fed into the LSTM layers, enabling the model to understand temporal dependencies and long-range sequential information within the data.

According to Brownlee: the CNN-LSTM model will read subsequences of the main sequence in as blocks, extract features from each block, then allow the LSTM to interpret the features extracted from each block. The model will split each window of 128 time steps into subsequences for the CNN model to process. The 128 time steps in each window will be split into four subsequences of 32 time steps.

Below are results from the following CNN-LSTM model:

- 2 layers of CNN, each with 64 filters, filter size 3, activation function relu.
- Dropout layer with rate of 50%.
- MaxPooling layer with a pool size of 2.
- Flatten layer.
- LSTM layer with 100 units.
- 2 dense layers:
  - 1 dense layer to act as the classifier with relu activation function.
  - 1 dense with softmax for classification (6 classes) – the output layer.
- Optimization algorithm: Adam (Adaptive Moment Estimation), with a base learning rate of 0.001.
- Loss function: Categorical Cross-Entropy (aka Log-Loss), multi-class classification, for one-hot encoded labels.
- Metrics: accuracy (training).

	Base parameters	**Increase learning rate (1e-3 → 1e-2)**	**Decrease learning rate (1e-3 → 1e-4)**	**Decrease units in LSTM layer (100 → 30)**	**Increase units in LSTM layer (100 → 300)**	**Increase nodes in dense layer (100 → 512)**
Test Accuracy	90.407% ± 0.956%	87.214% ± 1.159%	88.432% ± 0.582%	89.284% ± 0.953%	90.149% ± 0.983%	90.855% ± 0.676%
	Dense layer activation function (relu → tanh)	**No dropout in LSTM layer**	**Decrease dropout (0.5 → 0.25)**	**Increase dropout (0.5 → 0.75)**	**Increase filter size (3 → 5)**	**Increase filter size (3 → 7)**
Test Accuracy	90.366% ± 0.932%	90.254% ± 1.166%	90.621% ± 1.101%	90.098% ± 0.825%	90.584% ± 0.732%	91.401% ± 0.691%

	Increase pool size (2 → 5)	**Increase pool size (2 → 7)**	**Decrease filters (64 → 32)**	**Increase filters (64 → 128)**		
Test Accuracy	$89.912\% \pm 0.873\%$	$90.275\% \pm 0.627\%$	$89.796\% \pm 1.660\%$	$90.750\% \pm 0.547\%$		

Taking into account all the hyperparameters' values that contributed to an increase of the test accuracy from the base values: decrease dropout, increase filter size, increase filters, increase nodes in dense layers. We get a test accuracy of $91.388\% \pm 0.801\%$. The lesser increase may suggest an overly-complex model, maybe over-fitting, etc. This result is lower than the best multi-headed CNN result (over 10 epochs) of $92.681\% \pm 1.441\%$.

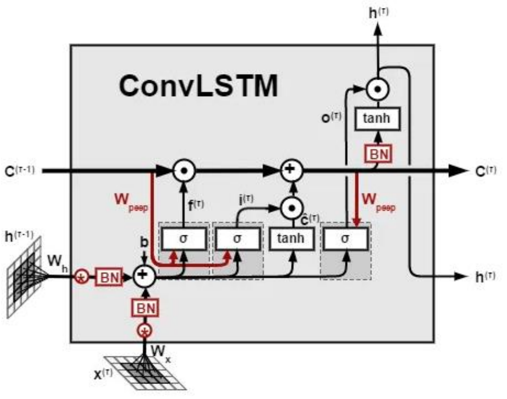
ConvLSTM Model

ConvLSTM is an extension to the previous CNN-LSTM model. It is done by performing the convolution operations as part of the LSTM: the internal matrix multiplication that are in the LSTM are exchanged with convolution operations.

A possible benefit to using ConvLSTM is simultaneously capturing spatial and temporal dependencies within the data samples. Potentially effective for tasks that require analysis of both time and space such as video processing.

Below are results from the following Convolutional LSTM model:

- 1 Convolutional-LSTM layer with 64 filters, with filter-size 3, and 50% dropout for the units in the LSTM.
- 1 flatten layer.
- 1 dense layer to act as the classifier with relu activation function.
- 1 dropout layer with 50% dropout rate.
- 1 dense with softmax for classification (6 classes) – the output layer.
- Optimization algorithm: Adam (Adaptive Moment Estimation), with a base learning rate of 0.001.
- Loss function: Categorical Cross-Entropy (aka Log-Loss), multi-class classification, for one-hot encoded labels.
- Metrics: accuracy (training).



A ConvLSTM cell

	Base parameters	**Increase learning rate (1e-3 → 1e-2)**	**Decrease learning rate (1e-3 → 1e-4)**	**No dropout in LSTM layer**	**Decrease dropout (0.5 → 0.25)**	**Increase dropout (0.5 → 0.75)**
Test Accuracy	90.251% ± 0.842%	89.087% ± 0.982%	81.588% ± 1.921%	90.115% ± 0.708%	89.355% ± 0.729%	84.126% ± 3.837%
	Increase nodes in dense layer (100 → 512)	**Increase filter size (3 → 5)**	**Increase filter size (3 → 7)**	**Decrease filters (64 → 32)**	**Increase filters (64 → 128)**	
Test Accuracy	90.238% ± 0.763%	90.892% ± 1.034%	91.120% ± 0.694%	89.291% ± 0.470%	90.421% ± 1.124%	

Each time the model is trained takes ~10-15 minutes (any of the cells in the above table).

Results, Conclusions, and Recommendations

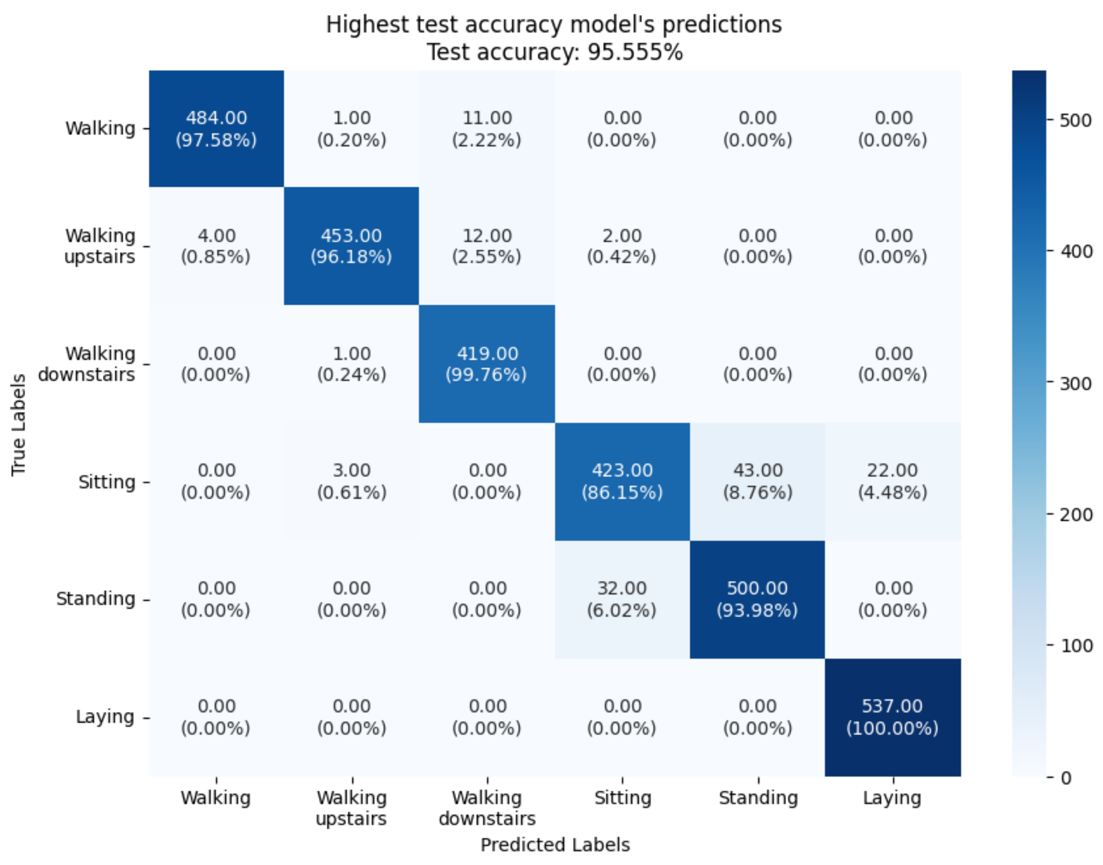
Results

The original paper that published the data set used the machine learning algorithm multi-class Support Vector Machine (SVM). With the human-engineered features that require domain-specific expertise and knowledge (signal processing), the model achieved 96.37% test accuracy. During the following years, multiple papers with different, novel approaches were published. Below is a summary of the different models and methods, and their performance with the same dataset, for comparison:

Paper	Model	Test Accuracy
Human activity recognition with smartphone sensors using deep learning neural networks	CNN with FFT applied to the dataset	95.75%[7]
A Public Domain Dataset for Human Activity Recognition Using Smartphones	Multiclass Support Vector Machine (original paper)	96.37%[5]
Deep Recurrent Neural Networks for Human Activity Recognition	Deep RNN	96.7%[4]
Human Activity Recognition using Wearable Sensors by Deep Convolutional Neural Networks	DCNN+ (creating an "activity image" from the samples and applying deep CNN)	97.59%[6]
Divide and Conquer-Based 1D CNN Human Activity Recognition Using Test Data Sharpening	Two-stage model (decision tree for static/dynamic activities, and two 3-class CNN models for the second stage individual activity) with test data sharpening	97.62%[2]

In this project, I started with a simple artificial neural network, that used the human-engineered features provided in the dataset, to hopefully show the strength of CNN and LSTM in automatic feature extraction, and to set a benchmark for this project. The result was 91.58%.

In most models in this project, the base-hyperparameters model had a test accuracy of around 90%. Specifically, the multi-headed CNN model with the logical-grouping in each head achieved the highest average test accuracy (over 10 attempts of model training) with $\mathbf{94.459}\% \pm \mathbf{0.424}\%$ over 500 epochs. Moreover, there was a singular model (from the total 10 attempts) that had a $\mathbf{95.555}\%$ test accuracy.



The classification of “Sitting” resulted in noticeably lower test accuracy when compared to the other actions’ classifications, failing to correctly classify 13.85% of the test samples, most of which were “stationary” actions (Standing and Laying), and even some “dynamic” actions (Walking upstairs). The high failure rate of misclassifying “Sitting” appears in the other articles reviewed in this project as well.

By increasing the maximum number of epochs to a few thousands, it could be possible to achieve even higher results with this model (given enough computational power and time). In addition, given greater computational power, more complex mode may achieve greater results as well.

Comparing the results of the Multiclass Support Vector Machine (96.7%) with this paper’s most successful model (Multi-headed CNN, 95.5%), it’s clear that a non-deep-learning model proved more accurate. However, such a model (as well as the Random Forest Classifier shown in Research and Testing – Specifications, 92.5%) requires domain-specific knowledge and expertise for feature engineering. These models used over 500 human-engineered features that require signal processing, data manipulation, etc., while a CNN/LSTM uses the raw data, eliminating the need for domain-specific knowledge and expertise, and allowing researchers to concentrate on fine-tuning the model instead of preparing features for the problem.

Conclusions

Throughout this study, various deep learning architectures were explored and applied to the problem of Human Activity Recognition. The project delved into the intricate dynamics of different CNN and LSTM variants, each exhibiting unique strengths and limitations in the context of time-series classification as a whole, and activity recognition in particular.

Both Convolutional Neural Networks and Long Short-Term Memory Neural Networks provide a powerful tool: automatic feature extraction. This ability not only saves time, but allows a larger audience to attempt and tackle problems that require knowledge or expertise in their domain. In addition, this enables multiple layers of abstraction in feature extraction. For example, in face recognition, the first abstraction layers could theoretically recognize edges, lines, etc., while deeper layers may recognize elements such as lips, eyes, nose, etc. These abstraction levels may not always be clear to us as humans in other domains. For example, it is hard to divide time-series classification problems to different layers of abstraction.

All types of machine learning and deep learning models and algorithms require hyperparameters. These hyperparameters can be tweaked and refined to achieve higher results with the model. Specifically in neural networks, there are many hyperparameters to refine, and since training a model takes time, it requires significant amounts of time and compute power to efficiently and accurately find the most optimized values for each hyperparameter. For example, to optimize a simple CNN, one may need to run multiple tests for each of the following hyperparameters: number of filters, filter size, pooling type (max/average/etc.), activation functions, learning rate, loss functions, optimizers (RMSProp, Adam, etc.), epochs, batch size, etc. Each of these need to be tested over a range of values to find the most optimized value for the specific given problem.

Lastly, every single problem in the world of machine learning is unique, and there is no “one size fits all” solution to most problems and issues that one faces during the development and learning processes. It requires time, effort, and dedication to tailor a solution to a given problem.

Recommendations

It is likely that the best-performance model can be refined even further, by modifying any or all of the following hyperparameters (given sufficiently powerful machines for testing in a reasonable time for model training):

- Different types of pooling: AveragePooling/GlobalMaxPooling/GlobalAveragePooling.
- Tweaking the number of deep fully-connected layers and the number of nodes in each of them.
- Modifying the number of filters each head gets: instead of the same number of filters for all 3 heads, each head can have a different number of filters.
- Using different types of optimizers. The optimizer used in all models was “Adam” – Adaptive Moment Estimation, which is an “extended version” of the Stochastic Gradient Descent. Possible other optimizers:
    - Stochastic Gradient Descent (SGD).
    - Root Mean Squared Propagation (RMSprop).
    - AdamW – modifies the typical implementation of weight decay in Adam.

  and others. For each of these, the learning rate can be tweaked and tested both linearly and exponentially to cover a decent range of values, and to find the most optimized value.
- Applying dropout optimization in other/additional layers, including the input layer, and testing additional values for the dropout rate.
- L1/L2 optimization can be used if the model becomes too complex.
- Testing different activation fuctions in all dense layers: linear, ReLU, logistic, tanh, leaky ReLU, etc. Most models were tested with ReLU.
- Test different batch size values. All the models were tested with batch size of 32.
- Tweaking the weights/bias/filters initializers. The default bias initializer in Keras is “zeros”, and the default kernel initializer in Keras is “glorot uniform”.

In addition to testing different values for the hyperparameters, it’s possible to modify the samples themselves, before even training a model. For example: standardization, normalization, etc.

Bibliography

[1] *Convolutional Neural Networks for Human Activity Recognition using Mobile Sensors,* Ming Zeng, Le T. Nguyen, Bo Yu, Ole J. Mengshoel, November 2014

[2] *Divide and Conquer-Based 1D CNN Human Activity Recognition Using Test Data Sharpening*, Heeryon Cho, Sang Min Yoo, April 2018

[3] *Deep Convolutional and LSTM Recurrent Neural Networks for Multimodal Wearable Activity Recognition*, Francisco Javier Ordonez, Daniel Roggen, January 2016

[4] *Deep Recurrent Neural Networks for Human Activity Recognition*, Abdulmajid Murad, Jae-Young Pyun, November 2017

[5] *A Public Domain Dataset for Human Activity Recognition Using Smartphones,* Davide Anguita, Alessandro Ghio, Luca Oneto, Xavier Parra, Jorge L. Reyes-Ortiz, April 2013

[6] *Human Activity Recognition using Wearable Sensors by Deep Convolutional Neural Networks*, Wenchao Jiang, Zhaozheng Yin, October 2015

[7] *Human activity recognition with smartphone sensors using deep learning neural networks,* Charissa Ann Ronao, Sung-Bae Cho, April 2016

[8] *Human Activity Recognition on Smartphones for Mobile Context Awareness*, Davide Anguita, Alessandro Ghio, Luca Oneto, Xavier Parra, Jorge L. Reyes-Ortiz∗, January 2012

[9] *An introduction to ConvLSTM*, Alexandre Xavier, Blog, March 2019