applied sciences

MDPI

*Article*

# Comparing Human Activity Recognition Models Based on Complexity and Resource Usage

**Simon Angerbauer, Alexander Palmanshofer *, Stephan Selinger and Marc Kurz**

Department of Mobility and Energy, University of Applied Sciences Upper Austria, 4232 Hagenberg, Austria; simon.angerbauer@gmail.com (S.A.); stephan.selinger@fh-hagenberg.at (S.S.); marc.kurz@fh-hagenberg.at (M.K.)
* Correspondence: alexander.palmanshofer@fh-hagenberg.at

**Abstract:** Human Activity Recognition (HAR) is a field with many contrasting application domains, from medical applications to ambient assisted living and sports applications. With ever-changing use cases and devices also comes a need for newer and better HAR approaches. Machine learning has long been one of the predominant techniques to recognize activities from extracted features. With the advent of deep learning techniques that push state of the art results in many different domains like natural language processing or computer vision, researchers have also started to build deep neural nets for HAR. With this increase in complexity, there also comes a necessity to compare the newer approaches to the previous state of the art algorithms. Not everything that is new is also better. Therefore, this paper aims to compare typical machine learning models like a Random Forest (RF) or a Support Vector Machine (SVM) to two commonly used deep neural net architectures, Convolutional Neural Nets (CNNs) and Recurrent Neural Nets (RNNs). Not only in regards to performance but also in regards to the complexity of the models. We measure complexity as the memory consumption, the mean prediction time and the number of trainable parameters of the models. To achieve comparable results, the models are all tested on the same publicly available dataset, the UCI HAR Smartphone dataset. With this combination of prediction performance and model complexity, we look for the models achieving the best possible performance/complexity tradeoff and therefore being the most favourable to be used in an application. According to our findings, the best model for a strictly memory limited use case is the Random Forest with an F1-Score of 88.34%, memory consumption of only 0.1 MB and mean prediction time of 0.22 ms. The overall best model in terms of complexity and performance is the SVM with a linear kernel with an F1-Score of 95.62%, memory consumption of 2 MB and a mean prediction time of 0.47 ms. The two deep neural nets are on par in terms of performance, but their increased complexity makes them less favourable to be used.

**Keywords:** human activity recognition; machine learning; deep learning; CNN; RNN; model complexity

check for updates

**Citation:** Angerbauer, S.; Palmanshofer, A.; Selinger, S.; Kurz, M. Comparing Human Activity Recognition Models Based on Complexity and Resource Usage. *Appl. Sci.* **2021**, *11*, 8473. https://doi.org/10.3390/app11188473

Academic Editor: Giancarlo Mauri

Received: 13 August 2021
Accepted: 7 September 2021
Published: 13 September 2021

**Publisher's Note:** MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.

**Copyright:** © 2021 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

## 1. Introduction

### *1.1. Motivation*

Due to the amount of possibilities where Human Activity Recognition (HAR) can be applied, it is a heavily researched field, with application scenarios ranging from medical applications, ambient assisted living, sports and leisure, tele-immersion to security surveillance. With these contrasting use cases also come very specific requirements that introduce the need for very specific approaches. For example, the use case of a security surveillance system used in a public place to recognize criminal activities comes with its inherent need for a vision-based approach as there is no possibility to equip any of the subjects with sensors. Sports applications on the other hand could range from the typical wearable fitness tracker that automatically starts tracking the activity you are doing to a supportive fitness mirror that recognizes specific body-weight training exercises and counts your repetitions automatically. With some of the applications evolving, new use

cases being created and new devices being introduced, the need for ever evolving HAR approaches arises.

As typical machine learning models have long been performing well for many different fields, recent years have shown that deep neural nets, with their ability to model highly complex data, have greatly improved the state of the art performances in fields like natural language processing or computer vision. This is why many researchers also started to apply the techniques to the domain of HAR. Not only for the most apparent vision-based approaches but also for sensor-based approaches. With this interest comes very contrasting ideas, from different deep architectures to varying ways to represent and feed data into the models. With this push towards more sophisticated and complex approaches, it becomes more and more important to not lose sight of the previously existing methods and models and relate to them. The complex approaches only being sensible when the complexity of the use case and the data warrants them, which is where our work comes in.

### 1.2. Goal

With this motivation, the goal is to use a publicly available dataset and develop a system to train and evaluate a set of typically used machine learning models and two of the mainly used deep neural net architectures. With this system we then want to put the prediction performance and the model complexity in perspective and find out which models exhibit the best performance/complexity tradeoff. As representation for the typical machine learning models, we use a K-Nearest Neighbour (KNN) classifier, a Random Forest (RF) classifier and a Support Vector Machine (SVM). Put into relation to the most commonly used deep neural net architectures, the Convolutional Neural Net (CNN) and the Recurrent Neural Net (RNN). To quantify the complexity of the models, we want to use three different metrics, the memory consumption and the mean prediction time of a deployed model and the number of trainable parameters of the models. As the number of trainable parameters is a metric typically only used to compare deep neural nets, we aim to introduce formulas for the approximation of the trainable parameters for the used machine learning models and therefore find a metric that can give an initial indication on the relative complexity across model types.

Contribution

This paper adds the following insights to the existing pool of research in this domain:

(i) A comparison of machine learning and deep learning approaches using not only performance but also three complexity metrics,
(ii) New comparable and reproducible insight into models trained on the UCI HAR Smartphone dataset,
(iii) Approximations for the number of trainable parameters in the used machine learning models.

### 1.3. Outline

This paper aims to reach these goals by first looking at important related work (see Section 2), starting with the applications of HAR before coming to the most closely related work with sensor-based and data-based approaches. In Section 3 we then explain the chosen dataset and machine learning models before going to the definition of the criteria used to evaluate the models. After that, the processing chains used to develop the machine learning and deep learning models are explained. To make the results of this paper clearly reproducible, the used and evaluated hyperparameters of all models are stated and for the deep neural nets, the chosen architectures are discussed. All of the combinations of these hyperparameters are then evaluated and the results are presented in Section 4 with discussions on the effects of certain parameters and the best performing models of each of the chosen model types. Using these results, Section 5 then compares the prediction performance and the complexity of the models and explains some of the differences. Finally, two models are chosen as the most favourable to be used if we would go forward and

deploy them in an application. One of them for a strictly memory-limited use case and one as the overall best tradeoff between complexity and performance. In Section 6 we then summarize this work and the key findings before outlining the limitations and the potential sections for future work.

## 2. Related Work

Recognizing the activity a human is doing at any given moment is a technique that has a wide range of areas of application. From health and care applications [1–3] to automation and context-awareness [4–6], sports [7,8], social media [9] and even security surveillance [10]. Dependent on the use case, the applicability of a certain approach in regards to sensors, attributes or models can vary.

### 2.1. Machine Learning

Machine learning has long become one of the predominant tools for HAR. Already in 2004, before the mass adoption of smartphones and wearables which provide unobtrusive access to typically used sensors like accelerometers or gyroscopes, Bao and Intille [11] used a setup of biaxial accelerometers attached to each of the limbs and the hip to gather and annotate sensor data. Using simple machine learning models like C4.5 decision trees, Naive Bayes and K-Nearest Neighbours (KNN), they achieved activity recognition accuracies of around 84% for the C4.5 and the KNN models. Most probably helped by their obtrusive setup with multiple sensors in different positions capturing motion in several important areas at once [11].

He and Jin [12] use a triaxial accelerometer placed in the typical place for a smartphone, the trouser pocket of the test subjects, to record their data set. Using the DCT, they represent the acceleration signals as a set of the $N$ most important underlying cosine signals. Using DCT instead of DFT, they do not have to deal with the underlying complex components but instead only get the real-valued cosine components which still provides them with a very good representation of the data and a reduction in complexity, as their best results come from using only the first 48 low-frequency components instead of 512 sensor readings per axis. As a next step, they reduce the feature dimension even further by using PCA. When using 20 principal components they lose no recognition performance even though this is a big reduction from 144 features (48 times 3 axes). Using these feature extraction methods and using a SVM as a classifier, He and Jin report a recognition accuracy of 97.51% for their four activities, running, standing still, walking and jumping [12].

Krishnan and Panchanathan [13] also use a SVM as a classifier in their approach. They compare it to a boosting based classifier with AdaBoost and to a regularized logistic regression classifier. The used data set was recorded by Bao and Intille [11] and contains seven activities (walking, sitting, standing, running, bicycling, lying down and climbing stairs). From this data, they extract several statistical and spectral features and combine them with statistical features from the first derivative (the rate of change) of the acceleration, without going into more detail on the exact features, unfortunately. They achieve their best results of 92.81% accuracy for subject independent training using the AdaBoost classifier. The AdaBoost classifier works by using a base classifier, typically a weak learner like a simple decision tree, to train lots of them one after the other on the same dataset whilst modifying weights in between to focus on misclassified samples, that normally might be harder to classify. They also try to use temporal relations of sequences of activities (e.g., changing from walking to running) to increase their recognition performance. Including three of the previous frames to influence the recognition of the current frame results in a 2.5–3% increase in performance [13].

Bayat et al. [14] in addition to evaluating multiple classifiers available in the Weka toolkit with Multilayer Perceptron, SVM, Random Forest, LMT, Simple Logistic and Logit Boost classifiers, also propose to split acceleration signals of smartphones into gravitational and body acceleration signals. They do so by using a digital low-pass filter, as the high-frequency components of the acceleration signals represent the body movements and

the low-frequency components represent the gravitational part. This results in multiple, more specialized, acceleration signals from which statistical features can be extracted. To calculate those descriptive features, they use consecutive windows with a 50 percent overlap on the time-series signals. For each window they extract features like the mean, the number of periodic peaks, the time between each of the periodic peaks, the root mean square (RMS), the standard deviation, the difference between the minimum and maximum values and the correlation between different axes. As the best result for accelerometer data of a smartphone placed in the pocket, they report 89.72%, achieved by a Multilayer Perceptron. They also try to combine multiple classifiers, which boosts their reported performance up to 91.15% accuracy [14].

As those summaries above show, besides evaluating and choosing a classifier, quite a lot of focus goes towards engineering features that represent the data in the best way possible and that can be modelled well by the classifiers.

### 2.2. Deep Learning

Compared to typical machine learning methods, deep learning methods can train more complex models to model more complex data. With ML models, the practitioner needs to help it understand the data by extracting more representable features. Deep models on the other hand can have so many trainable parameters through their different layers and various amounts of units to be able to understand raw data. The more trainable parameters a deep model has, the more complex data it can represent. The tradeoff between overfitting and underfitting data with a model is present for both approaches, but for deep models, the focus can shift from applying domain knowledge and extracting the most useful features to balancing that tradeoff by using different architectures, learning parameters and regularization methods. Given enough data, deep models outperform typical models like support vector machines in many problem domains. This section will highlight approaches for the domain of HAR using various architectures for deep models.

As mentioned in the introduction, finding the best performing architecture and hyperparameters is key to modeling complex data and still achieving good recognition performances. Hammerla et al. [15] focus on this exact problem by evaluating several architectures with ranges of hyperparameters regarding learning, regularization and architectural constraints. Their chosen architectures consist of a deep feed forward neural net (DNN), a CNN and three different RNNs. For the RNNs, they use Long Short Term Memory (LSTM) setups denoted LSTM-F, LSTM-S and b-LSTM-S. The LSTM-F and LSTM-S are deep feed forward LSTMs with recurrent cells only connected forwards in time whilst the b-LSTM-S is connected forwards and backwards using two parallel recurrent layers that are followed by a layer to combine them. The second distinguishing feature of those LSTMs is the way the input is fed into them. With the LSTM-F using predefined timespans of sensor readings and the LSTM-S and the b-LSTM-S only using one single sensor reading at a time [15]. Hammerla et al. [15] use fANOVA analysis to estimate the impact each of architectural, learning, regularization parameters and beneficial interactions between those three on the performance of the model. The hyperparameters for each trained model were chosen at random. On average, architectural parameters and learning parameters are equally important whilst the regularization, to reduce overfitting, is less important. But a very significant portion of the variance comes from beneficial interactions of certain parameters in combination with each other, which indicates that focusing on one single type is not as important as finding the best combination of different kinds of parameters [15].

In contrast to the work by Hammerla et al. [15], most of the research focuses on one single deep learning approach and one or more datasets to evaluate its performance. Ronao and Cho [16] for instance use a CNN on the dataset also used in our work. They propose CNNs as a good solution for HAR due to their ability to exploit local temporal relations within time-series signals and cancel out smaller translations in the sensor readings. They report a peak performance of 94.79% accuracy for a model consisting of three convolutional and pooling layers followed by a fully connected layer and a softmax layer. They also

report their evaluation results of the performance increase received by changing the number of convolutional layers, the kernel size of the convolutional layers and the pooling size of the pooling layers. While the pooling size had no effect on performance, the kernel size shows major performance increases up to a kernel size of 9. For the input of the complex raw sensor data, the amount of convolutional layers is quite important. Going from one to two convolutional layers, the performance increases drastically, with a small improvement available when adding a third layer. The fourth layer on the other hand already introduces too much complexity and probably overfitting which results in a slight decrease in performance [16].

Jiang and Yin [17] also use a CNN, as it seems to be the most used deep architecture. Compared to other works using CNNs, they use a novel approach regarding their use of input data. They use all three axes of the gyroscope signal, all three axes of the total acceleration signal (body motion and gravity) and all three axes of the linear acceleration signal (only body motion) to append them as rows of an image. This allows the two dimensional CNN to also learn from relations between those signals [17].

Their algorithm to calculate this image includes all of these signals multiple times per image in differently ordered sequences to make sure the relations of one signal with multiple others can be captured. The magnitude of the Discrete Fourier Transform (DFT) is then used to get the activity image. These activity images ($36 \times 68$ in this case) are fed into two different two-dimensional convolutional layers with subsampling layers in between followed by a fully connected layer and a softmax layer to arrive at the activity predictions. Similar to the findings of Ronao and Cho [16], they achieve the best results using two convolutional layers, compared to their other evaluated architectures with only one or three or more convolutional layers. For their best performing model, they report an accuracy of 95.18% for the UCI HAR Smartphone dataset [17].

In their work, Guan and Plötz [18] iterate that most of the activity recognition setups treat individual frames of sensor data statistically independent but this actually gets rid of one important component of sensor readings in an activity recognition context, which is the temporal relation of one of these frames with for example its predecessors. This is why they view RNN approaches as a better alternative for the mostly used CNNs. Most of the RNN approaches focus on models with Long Short Term Memory (LSTM) layers, while Guan and Plötz focus on an ensemble of such models in order to better deal with their identified issues of current HAR solutions. The first of these identified issues is the data used for HAR, which is mostly recorded in mobile or wearable context, as this data tends to be very noisy with high variations even for one and the same activity. Besides data quality, the quantity of annotated data is an issue, as the recording of sensor data on mobile devices is trivial, but the experiments to record and annotate very big data sets, fitting for deep learning approaches, are challenging to do with good enough class balancing. This can also be seen in available public datasets for HAR that mostly suffer from bad class distribution. Additionally, most approaches are based on sliding window and frame-based predictions that skip the challenge of finding and locating certain relevant sequences of sensor readings in a sequential stream. Guan and Plötz want to address these issues by employing their ensemble-based approach. For each training epoch, they train multiple models on randomly sampled sequences of random length of the training data. The best models of these are determined by the validation data and the predictions of those best models are then fused to a single prediction for a specific sample from the test data at a specific point in time. This is essentially bagging, which is normally done with very simple classifiers like trees, with highly complex models. In their results, they observe that this ensemble method mostly has a positive impact on very challenging and similar classes and therefore could indicate better robustness against the previously mentioned issues with real-world data [18].

Additionally to the above mentioned DNNs, RNNs and CNNs, Stacked Autoencoders and Deep Belief Networks (DBNs) are also used for HAR. Almasklukh et al. [19] for example use a Stacked Autoencoder to train their model in two stages with the first

stage being about learning to reconstruct the data in an unsupervised manner before refining the model by learning on the labelled data in a supervised manner. Autoencoders consist of a decoder and an encoder with the encoder producing the code and the decoder reconstructing the input from this code. Autoencoders are typically restricted in the way they can represent the data to not be able to do it perfectly and therefore force it to find good representations, that for example reduce the dimensionality of the input data, or extract more representative features. Different types of autoencoders are used for different applications. One of those types is the Sparse Autoencoder that serves the purpose of adding sparsity (many inactive hidden units) to the input data by applying a regularization function. This forces the units that are left active to learn very latent and powerful features in the input data to then have a representation that generalizes better on unseen data. For the stacked autoencoder employed by Almasklukh et al., they use two of those sparse autoencoders to be able to find good features in an unsupervised way in phase one and then align this with the labeled data to predict activities accurately in phase two. They report an accuracy of 97.5% on the UCI HAR Smartphone data set [19].

DBNs as used by Zhang et al. [20] consist of multiple stacked Restricted Boltzmann Machines (RBMs). An RBM is a variant of a Boltzmann machine with the constraint that connections between units of different groups (hidden and visible units) are allowed, but no connections between units of the same group. These RBMs are then stacked with each of the hidden layers being the visible layer of the following. The hidden units of the RBMs act as feature detectors with one RBM being trained after the other in an unsupervised manner. This training phase learns a generative model that is not yet able to classify the activities. To enable the model to do that, back-propagation is used to adapt the trained units to the labeled data [20].

As shown by these highlighted papers, many different deep learning approaches are possible. The mostly used architectures are CNNs [15–17,21–25] followed by RNNs [18,26–28] and the occasional outlier with DNNs [15], Autoencoders [19], Deep Belief Networks [20] or even Generative Adversarial Networks (GANs) for specific use cases [29]. There seems to be not one single approach working especially well, as all of them achieve good results when tuned to a certain extent.

## 3. Materials and Methods

As already highlighted in the introduction, the goal of this work is to train typical machine learning models like support vector machines and compare them to two deep neural nets with recurrent and convolutional architecture for the domain of HAR. This section will show which criteria are used to evaluate those approaches, the dataset and its recording and preprocessing steps and the evaluated models with their respective hyperparameters.

### *3.1. Evaluation Criteria*

To compare the different model types with each other, the two evaluation criteria explained in this section are used. For each model type the models are first tuned to achieve the best performance and then the best performing models of each type are compared using the performance and the model complexity.

#### 3.1.1. Performance

For our dataset and problem, we use the macro averaged F1-Score as a robust metric based on recall and precision and the accuracy to be able to compare to the related work also using accuracy. We use macro averaging, because for HAR, each of the classes is equally important and should contribute in equal amounts. The dataset is balanced enough to allow that without being unrepresentative.

#### 3.1.2. Model Complexity

As HAR is often applied in the mobile or wearable sector, model complexity becomes an important factor to consider when deciding which approach to go with. Hardware

limitations like memory usage or power consumption can be critical to the applicability of the approach. However, it is also important to consider the effects of the model complexity on the usability, because if the model works perfectly but a user needs to wait a long time to actually get the result, it will not be beneficial for the application. This is why we use memory consumption, prediction time and number of trainable parameters as metrics to capture the complexity of the models. The memory usage and prediction time act as indicators for the strain on the hardware, while prediction time also is an important factor for the usability. The number of trainable parameters is used as a comparable measure of the size and complexity of a model that can be related to the two other hardware and usability relevant measures.

To record the memory usage and prediction time, the models are deployed on an iPhone XR using the Core ML framework for the machine learning models and Tensorflow Lite for the deep learning models. This shows the two metrics in a real world usage scenario on a mobile device that is more representative than gathering the data on the development machine. Using a debug session within Xcode, the memory consumption for each of the models can be determined. To get the mean prediction time, a set of samples from the test data are fed into the Core ML and Tensorflow Lite models and then averaged. The samples are not recorded from the sensors of the device because for these two metrics, reproducing the exact processing pipeline and getting the result of the prediction in a full real world scenario is not of interest. However, to evaluate the performance of these models on real world data would definitely be interesting for future work (see Section 6), even though it is not included in this work.

The number of trainable parameters is a metric that is typically used for deep neural nets, where each weight that can be learned during model training represents one such trainable parameter. For deep neural nets, the higher the number of parameters, the more complex the data it can represent, but also the more likely it is to overfit the training data if it is not that complex. As it is quite difficult to compare the complexity of models of different types to each other and there is no perfect solution for this problem available, we try to apply the metric of trainable parameters to all of our evaluated models. For the deep learning models, this number can be obtained through the respective deep learning libraries used to train them. For the machine learning models, we use the following formulas to approximate the value:

$$Parameters_{KNN} = num_samples \tag{1}$$

$$Parameters_{RF} = \sum_{n=0}^{num_estimators-1} (node_count_n) \tag{2}$$

$$Parameters_{SVM} = num_support_vectors \cdot num_features \tag{3}$$

### 3.2. Technologies

Python 3.7 is the language used to develop and run the experiments. A set of Python libraries are used throughout the system. Numpy and Pandas are used for data handling when reading, using and visualizing data. Matplotlib and Seaborn are used for plotting. Scikit-learn (Sklearn) is used as a library for feature selection and machine learning model evaluation. For the RNN and CNN evaluation, Tensorflow, Keras and TensorBoard are used. To convert the scikit-learn models to Core ML, the coremltools library is used. For the Xcode deployment project, Core ML (for the machine learning models) and Tensorflow Lite (for the deep learning models) are used to deploy the converted models on an iPhone XR.

### 3.3. Dataset

Fortunately, as HAR is a well researched topic, there are many public datasets available [30]. While several of the available datasets may be applicable for our experiments, there are a few major factors that influence whether a dataset fits a given use case or not. For our work, we were concerned with the type of activities that were recorded and labelled, the way these activities were recorded, the number of recorded subjects as well

as the total count of samples. Concerning the count of samples in a dataset, the way a dataset reports its sample count should be investigated closely. For example, the UCI HAR Smartphone dataset [31] reports 10,299 samples, while the WISDM dataset [32] reports 1,098,207 samples. However, in contrast to the UCI HAR Smartphone dataset, where each sample is a sequence of 128 sensor readings, the WISDM dataset reports one sample as exactly one sensor reading. The UCI HAR Smartphone dataset was selected as the basis for this work, because it was recorded on a high number of subjects (30), has a good class distribution, has enough samples to allow deep learning (10,299 samples with 128 sensor readings each) and an easily reproducible recording setup with pocket-worn smartphones as sensors. This section will explain the recording setup and preprocessing steps of the experiment (see Section 3.3.1), the extracted features (see Section 3.3.2) and analyze some features and samples to get a better understanding of the dataset (see Section 3.3.3).

#### 3.3.1. Recording and Preprocessing

Experiments to record six exercises (standing, sitting, laying, walking, walking upstairs, walking downstairs) were conducted by 30 participants (age 18–49). Each subject performed each activity twice in a laboratory condition. One time with a smartphone (Samsung Galaxy SII) placed on the left side of the waist and one time with the smartphone placed in the preferred position of the subject. The accelerometer and gyroscope sensors of the smartphone were used to record three-axial linear acceleration and three-axial angular velocity. The sensor values were recorded at a rate of 50 Hz. The experiments were recorded on video and the samples were labelled manually. To reduce noise, a median filter and a third-order low-pass Butterworth filter with 20 Hz cutoff frequency were applied. To separate the low-frequency gravitational components from higher frequency body motion components, another Butterworth filter, this time with a very low frequency of 0.3 Hz, was applied. The euclidean magnitude and time derivatives were calculated to get more time signals (jerk and angular acceleration) and Fast Fourier Transform (FFT) was used to get the frequency space of most of the signals, resulting in 17 signals. Each of those signals was windowed by a fixed-width sliding window with 50% overlap, resulting in 2.56 s long samples (128 values). The resulting dataset has a balanced class distribution albeit walking upstairs and walking downstairs having somewhat fewer samples [31].

#### 3.3.2. Feature Extraction

For each of those 17 windowed time domain and frequency domain signals, a set of features was calculated, as can be seen in Table 1, resulting in 561 extracted features that are ready to be used for machine learning as a better separable representation of the raw sensor data. The calculated features range from mean values and standard deviations to measures like the interquartile range or the correlation coefficient of two correlated signals [31].

**Table 1.** List of calculated measures of the windowed signals of the UCI HAR Smartphone dataset [31].

| Function | Description |
|---|---|
| mean | Mean value |
| std | Standard deviation |
| mad | Median absolute deviation |
| max | Maximum value |
| min | Minimum value |
| sma | Signal magnitude area |

**Table 1.** *Cont.*

| Function | Description |
|---|---|
| energy | Average sum of squares |
| iqr | Interquartile range |
| entropy | Signal entropy |
| arCoeff | Autoregression coefficients |
| correlation | Correlation coefficient |
| maxFreqInd | Largest frequency component |
| meanFreq | Frequency signal weighted average |
| skewness | Frequency signal skewness |
| kurtosis | Frequency signal kurtosis |
| energyBand | Energy of a frequency interval |
| angle | Angle between to vectors |

#### 3.3.3. Analysis

When analyzing the extracted features, they do not always look perfectly separable, but with the combination of multiple different features, a machine learning model should be able to classify the activities correctly. The boxplots grouped by activity in Figure 1a,b for example show a clear difference in the distributions for activities with little to no movement compared to activities with more movement. With those two boxplots both showing features based on the jerk signals, it makes sense that this separation is clearly visible, as the change of acceleration for activities with little movement is quite different to the change of acceleration for more active movements. Another example in Figure 1c shows a clear separation of especially laying and also sitting from the other four activities. Other features on the other hand are probably less fitting to decide between activities. Figure 1d for example shows the correlation between $x$- and $y$-axis signals of the body acceleration, where besides the fact that the distributions of the values for walking, walking upstairs and walking downstairs are more narrow, they are all over the same value range and therefore hard to discriminate.

As indicated by those boxplots, the separation of the low movement activities from the high movement activities looks simple, while the difference between the activities of those two categories seems marginal. However, when looking at the scatter plot of some of the features with each other (see Figure 2), they look a bit easier to discriminate. For example, the plot on the left side in the middle, which shows the scatter plot of the mean of the gravitational acceleration ($y$-axis) and the entropy of the jerk signal of the body acceleration ($x$-axis), shows a clearer segregation of the laying, walking and standing activities. In contrast to the plot on the left side on the bottom, which shows the scatter plot of the standard deviation of the frequency domain signal of the body acceleration ($x$-axis) and the entropy of the jerk signal of the body acceleration ($x$-axis). In this plot, the walking, walking upstairs and walking downstairs activities are easier to discriminate. Given this improvement, it bodes well for a machine learning model, that should be able to perform well when combining some of the best features and learning from them.

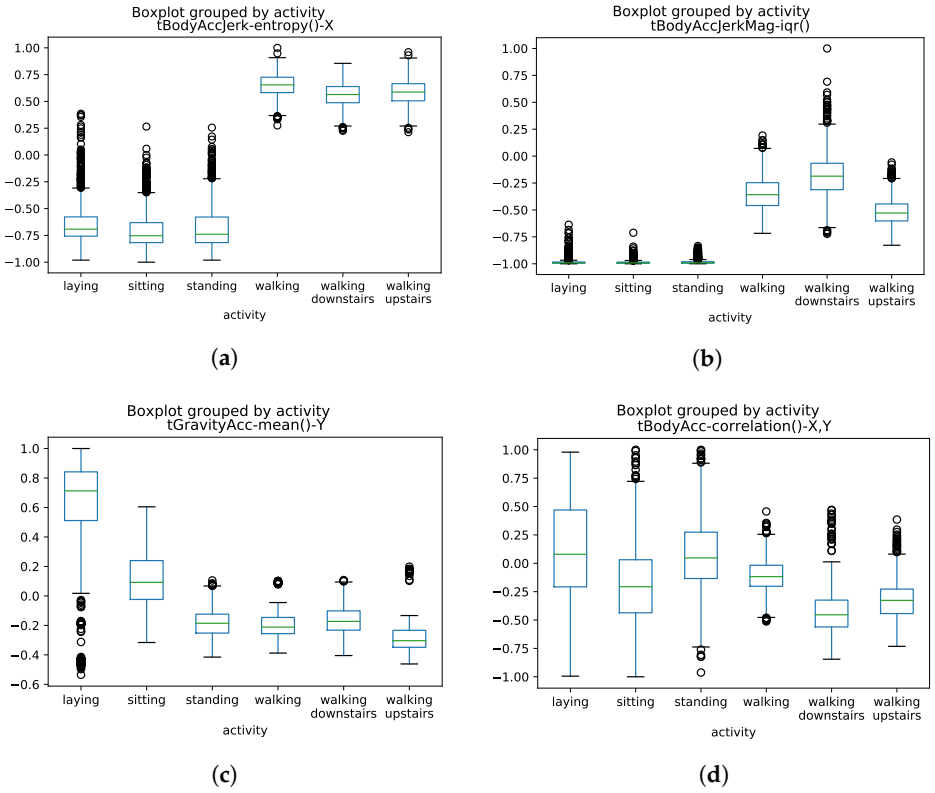



**Figure 1.** Boxplots for four features of the dataset grouped by activity. (**a**) Entropy of the *x*-axis jerk signal of the body acceleration. (**b**) Interquartile range of the jerk signal magnitude of the body acceleration. (**c**) Mean of the *y*-axis signal of the gravity acceleration. (**d**) Correlation between *x*- and *y*-axis signals of the body acceleration.

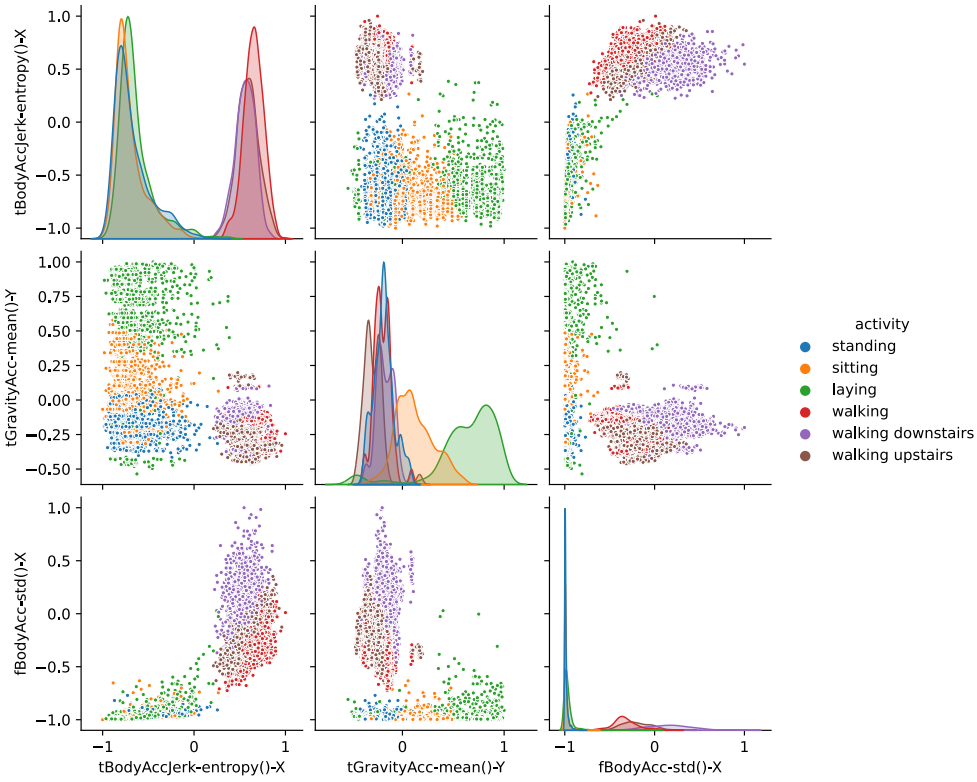



**Figure 2.** Feature pair plot for a few selected features.

### *3.4. Machine Learning*

The data recording, preprocessing and feature extraction necessary to get reasonable results from machine learning is already done by the UCI HAR Smartphone dataset [31] (see Section 3.3). As a first step, to understand the dataset, different plots like bar charts to indicate class distribution or boxplots per activity to understand distribution of features for different activities are used (see Section 3.3.3). Given the dataset comes with 561 extracted features, the next step is to find an optimal amount of features that can represent the data in the best way. For this, we use recursive feature elimination (RFECV) with stratified 10 fold cross validation and a baseline classifier. Figure 3 shows the performance of the baseline classifier depending on the amount of features used. Ninety-one is the amount of features achieving the best performance and those features are therefore used to train the models. This figure also shows that there is potential to decrease the amount of features, and by that also the complexity, even further. To find a well performing model, several classifiers (see Section 3.4) are used and evaluated over a certain range of hyperparameters. To find the leading model with its best hyperparameters, a grid search with 10 fold cross validation is employed. For each of the evaluated classifiers, the best configuration is then tested against the held back test set and the performances can be compared to each other. The best of those typical machine learning models will then be compared to the deep neural nets.

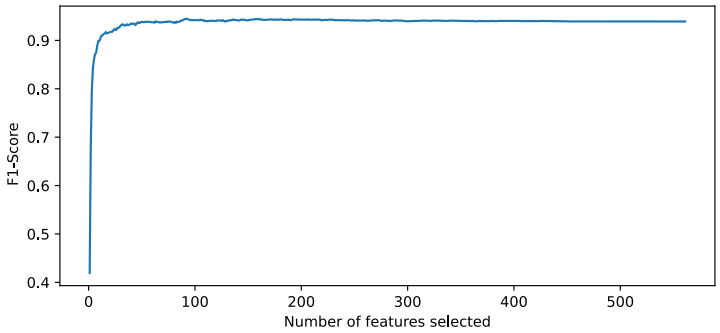



**Figure 3.** Performance of baseline SVC for number of selected features using RFECV with StratifiedKFold cross validation.

Models

Three different classifiers, Support Vector Classifier (SVC), K-Nearest Neighbors (KNN) and Random Forest (RF) are evaluated regarding their performance and complexity. For each of those models, the hyperparameters are tuned in order to improve the performance. For the SVC, there are two hyperparameter ranges, as the different kernels (RBF and linear) have different available hyperparameters. For each of the hyperparameters, there is either a set of values or a range that is indicated by the min/max values.

For example the cost (C) of the support vector classifier with the RBF kernel seen in Table 2 is evaluated in a range of $2^{-7}$ to $2^{7}$, which means $2^{-7}$, $2^{-6}$, $2^{-5}$ and so on are evaluated. The hyperparameter selection including their ranges for the linear kernel SVC can be found in Table 3. Contrary to the SVC models, the criterion of the random forest model seen in Table 4 is evaluated as both gini and entropy. Finally, the hyperparameter definitions for the KNN model can be inspected in Table 5.

**Table 2.** Evaluated hyperparameters for the SVC with RBF kernel.

| **Hyperparameter** | **min** | **max** |
|---|---|---|
| kernel | rbf | |
| C | $2^{-7}$ | $2^{7}$ |
| gamma | $2^{-7}$ | $2^{0}$ |
| class-weight | balanced | |

**Table 3.** Evaluated hyperparameters for the SVC with linear kernel.

| Hyperparameter | min | max |
|---|---|---|
| kernel | linear | |
| C | $2^{-10}$ | $2^4$ |
| class-weight | balanced | |

**Table 4.** Evaluated hyperparameters for the Random Forest.

| Hyperparameter | min | max |
|---|---|---|
| n-estimators | $3^1$ | $3^6$ |
| criterion | gini, entropy | |
| max-depth | $3^1$ | $3^7$ |
| class-weight | balanced | |
| max-features | auto, sqrt, log2 | |
| min-samples-split | $2^1$ | $2^2$ |

**Table 5.** Evaluated hyperparameters for the KNN.

| Hyperparameter | min | max |
|---|---|---|
| n-neighbors | $3^1$ | $3^4$ |
| weights | uniform, distance | |
| leaf-size | $3^1$ | $3^4$ |
| algorithm | brute, kd-tree, ball-tree | |
| p | 1 | 2 |

### *3.5. Deep Learning*

Similar to the machine learning process, the model training process starts with the data recording and preprocessing part, that is already done by the UCI HAR Smartphone dataset [31], but this time without the feature extraction part, as we do want to compare the machine learning models using extracted features to deep neural nets which are not using those extracted features but instead model the intricacies of the preprocessed sensor data directly. Another major difference to the machine learning process is that prior to tuning the hyperparameters, a fitting architecture for the RNN and CNN needs to be found. This prior architectural evaluation results in well performing architectures with for example specific amounts of convolutional layers and max pooling layers. Additionally, only in the next step, each of the best architectures of the RNNs and CNNs will undergo finer tuning of for example the number of units of each of the convolutional layers or the factor of regularization on a another layer. For this fine tuning a grid search is used to explore the hyperparameter space and find the best performing RNN and CNN configuration for the preprocessed sensor data by selecting the hyperparameters performing best on the test data on average.

#### Models

As a first step before fine tuning the models, the goal is to find architectures that tend to perform well and have the potential for further optimization and tuning by changing hyperparameters. Table 6 shows the CNN architecture, that shows the best promise and will be used going forward. The input layer has an output shape of $128 \times 9$, where 128 represents the number of sensor readings and 9 represents the number of signals used

(total acceleration x/y/z, body acceleration x/y/z and body gyro x/y/z). The output of this input layer is then fed into the first of two sets of a convolutional layer, a dropout layer and a max pooling layer. The convolutional layer is one-dimensional, as we only want to exploit the local relations within signals and not across signals, in contrast to, for example, Jiang and Yin [17]. In order to reduce overfitting a dropout layer, that deactivates a certain percentage of neurons, and a one dimensional max pooling layer, that only uses the maximum value in the pool and therefore reduces the dimension, is used. As mentioned, this combination is used twice, due to the fact that when only using it once, it tends to underfit the data similar to the findings by Ronao et al. [16]. To arrive at the output shape of the six possible activities, the output of the last max pooling layer is flattened down into one dimension and then reduced by two dense layers.

Table 7 shows the second architecture, the RNN, that is used going forward. The output of the input layer has the same shape as for the CNN with $128 \times 9$ and is fed into this time two LSTM layers with an accompanying dropout layer for each to reduce overfitting. Again, one layer of this combination tends to underfit which is why there are two layers again. The second LSTM already has a one dimensional output, which is why in this case no flattening is needed, but just the two dense layers to arrive at the output shape of the six possible activities.

**Table 6.** Chosen CNN architecture.

| Layer |
|---|
| Sensor Data (Input) |
| Convolutional Layer 1 (1D) |
| Dropout Layer 1 |
| Max Pooling Layer 1 (1D) |
| Convolutional Layer 2 (1D) |
| Dropout Layer 2 |
| Max Pooling Layer 2 (1D) |
| Flattening Layer |
| Dense Layer 1 |
| Dense Layer 2 (Output) |

**Table 7.** Chosen RNN architecture.

| Layer |
|---|
| Sensor Data (Input) |
| LSTM Layer 1 |
| Dropout Layer 1 |
| LSTM Layer 2 |
| Dropout Layer 2 |
| Dense Layer 1 |
| Dense Layer 2 (Output) |

In order to fine-tune the CNN architecture depicted in Table 6, we evaluate the hyperparameter ranges that can be seen in Table 8. Same as in the machine learning section, the ranges are indicated through minimum and maximum with each different exponent in between those ranges being evaluated. For the dropout and the kernel regularization, no values in between are evaluated, only the specified minimum and maximum. To find the

best performing combination of those hyperparameters, we use a grid search over all the specified ranges.

Exactly the same as for the CNN, Table 9 shows the evaluated hyperparameter ranges for the RNN architecture. A grid search is used to find the best performing combination over all the specified ranges.

**Table 8.** Evaluated hyperparameters of the CNN architecture.

| Hyperparameter | Min | Max |
|---|---|---|
| Number of Epochs | 10 | 10 |
| Batch Size | $2^6$ | $2^7$ |
| Optimizer | RMSprop | |
| Loss Function | Categorical Crossentropy | |
| Number of Units (Convolutional Layer 1) | $2^5$ | $2^9$ |
| Number of Units (Convolutional Layer 2) | $2^5$ | $2^9$ |
| Kernel Size (Convolutional Layer 1) | $3^1$ | $3^2$ |
| Kernel Size (Convolutional Layer 2) | $3^1$ | $3^2$ |
| Activation Function (Convolutional Layer 1) | ReLU | |
| Activation Function (Convolutional Layer 2) | ReLU | |
| Dropout 1 | 0.0 | 0.5 |
| Dropout 2 | 0.0 | 0.5 |
| Pooling Size (Max Pooling Layer 1) | $2^1$ | $2^2$ |
| Pooling Size (Max Pooling Layer 2) | $2^1$ | $2^2$ |
| Number of Units (Dense Layer 1) | $2^7$ | $2^8$ |
| Kernel Regularization (Dense Layer 1) | 0 | 0.01 |
| Activation Function (Dense Layer 1) | ReLU | |
| Activation Function (Dense Layer 2) | Softmax | |

**Table 9.** Evaluated hyperparameters of the RNN architecture.

| Hyperparameter | Min | Max |
|---|---|---|
| Number of Epochs | 10 | 10 |
| Batch Size | $2^6$ | $2^7$ |
| Optimizer | RMSprop | |
| Loss Function | Categorical Crossentropy | |
| Number of Units (LSTM Layer 1) | $2^5$ | $2^9$ |
| Number of Units (LSTM Layer 2) | $2^5$ | $2^9$ |
| Dropout 1 | 0.0 | 0.5 |
| Dropout 2 | 0.0 | 0.5 |
| Number of Units (Dense Layer 1) | $2^7$ | $2^9$ |
| Activation Function (Dense Layer 1) | ReLU | |
| Activation Function (Dense Layer 2) | Softmax | |

## 4. Results

After explaining the materials and methods (see Section 3), this section will present the results of the conducted experiments to find the best performing machine learning (see Section 4.1) and deep learning models (see Section 4.2) of the selected model types and architectures.

### *4.1. Machine Learning*

To represent the group of the typical machine learning approaches, three model types (SVM, RF and KNN) have been chosen. Each of those model types has been tuned to extract the best possible performance for the given dataset using an exhaustive grid search with cross-validation over a defined hyperparameter space. Section 4.1.1 shows the results of these hyperparameter evaluations while Section 4.1.2 shows the best performing configuration for each of the model types.

#### 4.1.1. Hyperparameter Evaluation

For the Random Forest Classifier, the hyperparameter ranges depicted in Table 4 were evaluated. Figure 4 shows the results of this evaluation. As the evaluation was done by running a grid search over the defined hyperparameters, many different combinations are trained and therefore for each of the hyperparameters, multiple models are trained. The mean performances of the models with a certain value of the hyperparameter are displayed in a subplot for each of them. Intuitively, the most important factors to influence the performance of a Random Forest would be the number of trees it consists of and how deep these individual trees can go. This is also partially in line with our results, where the most performance is gained by finding the best amount of trees (n_estimators) with 81. The other evaluated parameters, the maximum allowed depth, the maximum number of features per tree, the minimum needed samples to split a tree and the criterion function to measure the quality of a split did result only in smaller gains. Across all of the models and parameters, there is a very big spread of performances visible by looking at the error bars on the test performances showing the standard deviation. This could indicate that finding a certain combination of hyperparameters that work well together is very important.

For the Support Vector Classifier, there were two separate evaluation runs as one of them uses an radial basis function (RBF) kernel while the other uses a linear kernel. With these two having different available hyperparameters, a single evaluation was not possible. Figure 5 shows the results for the SVC with RBF kernel and Figure 6 shows the results for the SVC with linear kernel. For both of them, the determining factor is the regularization parameter $C$, with the RBF version performing best with a value of 64 and the linear version performing best with a value of 3. For the RBF kernel SVC, another important parameter is gamma, the kernel coefficient, which leads to the best performance with a value of around 0.03.

The KNN is a somewhat different classifier from the other two in the sense that the predictions always depend on all the trained samples instead of a parameterized model to discriminate classes. Figure 7 shows the evaluated hyperparameters for the KNN. The plots show that the most important hyperparameter, in this case, is the number of neighbouring samples that are used to predict the class of another sample. A value of 9 for n_neighbors performs the best with everything below overfitting too much due to too few neighbours being considered and everything above probably being to coarse to take advantage of local dependencies for a specific class. According to these results, the algorithm used to compute the nearest neighbours, the leaf size of the trees (only applicable, if kd_tree or ball_tree is used as the algorithm) and the weighting of the neighbours (uniform or based on actual distance) do provide no positive gains at all. The one hyperparameter that does still influence the performance is the power parameter $p$ that influences how the Minkowski metric calculates the distance to the neighbours. With a value of 1, the Minkowski metric is equivalent to the Manhatten distance, while a value of 2 represents

the Euclidean distance. With the Manhatten distance looking to be a slightly better fit in this case.

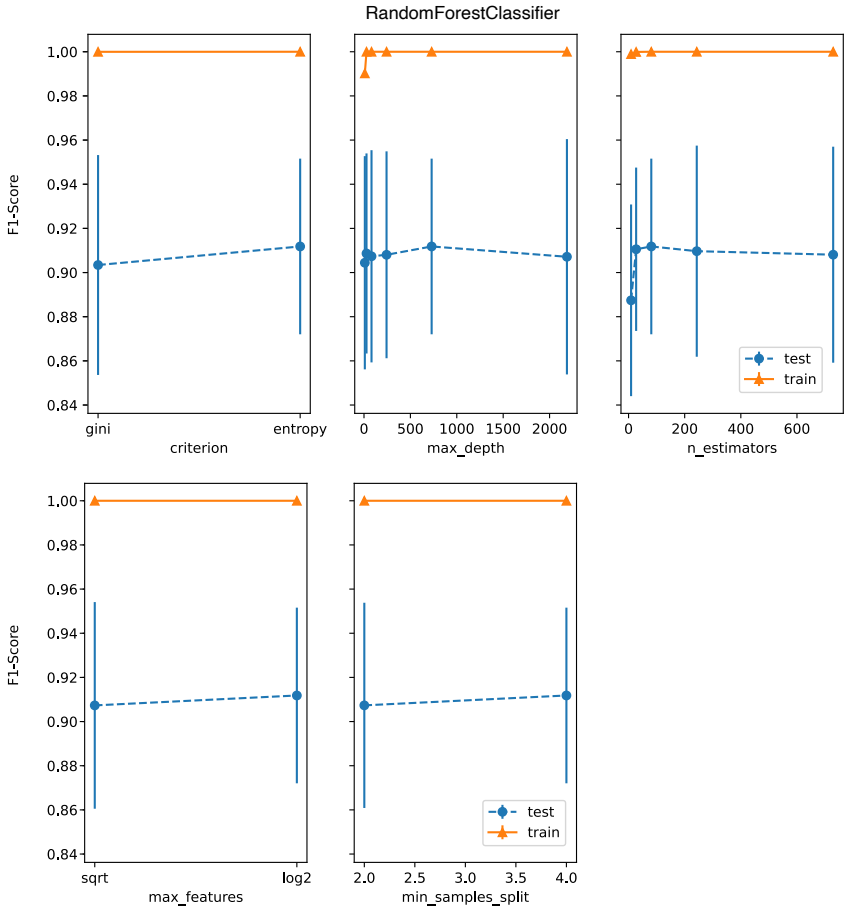



**Figure 4.** Mean performance of Random Forest models for respective evaluated hyperparameters with standard deviation.

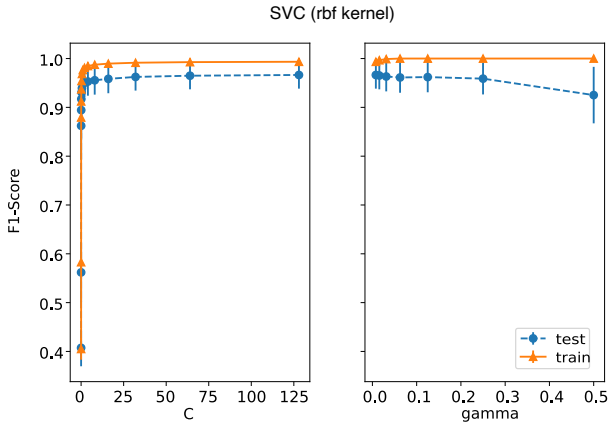



**Figure 5.** Mean performance of SVC (RBF kernel) models for respective evaluated hyperparameters with standard deviation.

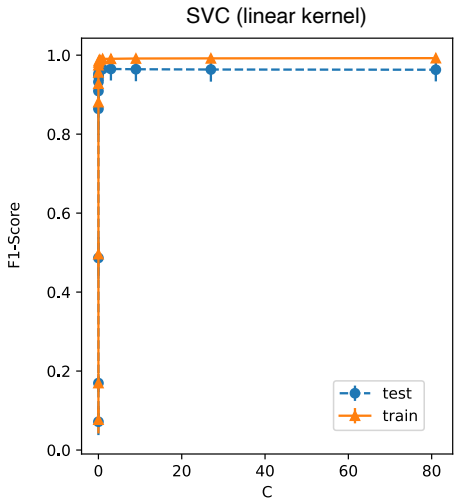



**Figure 6.** Mean performance of SVC (linear kernel) models for respective evaluated hyperparameters with standard deviation.

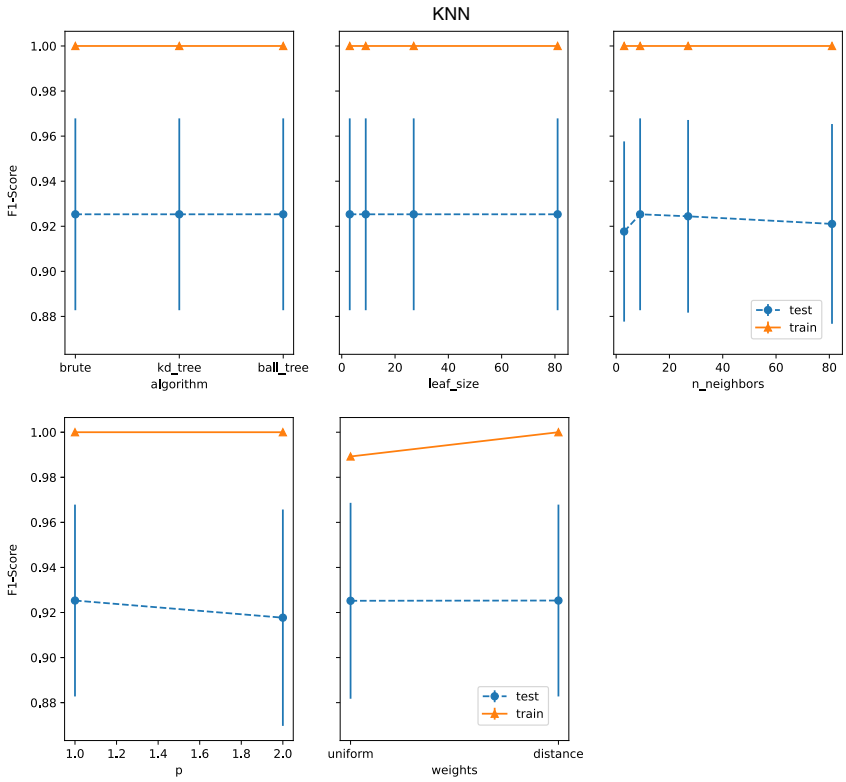
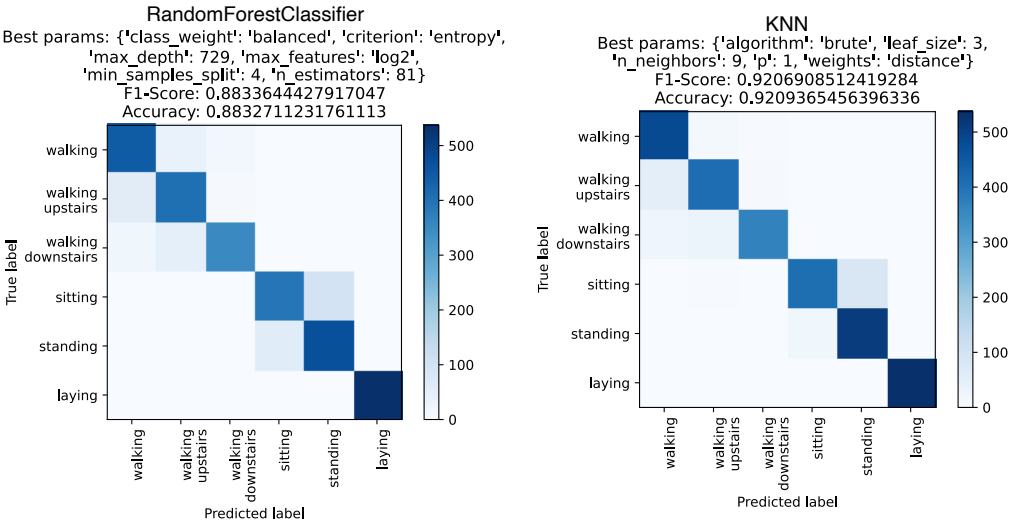



**Figure 7.** Mean performance of KNN models for respective evaluated hyperparameters with standard deviation.

#### 4.1.2. Best Performing Models

Having seen the evaluated hyperparameters of all the model types, we can now look at the best performing configurations of each type and compare their performance on the 30% held-back test set. In Figure 8a you can see the performance of the best Random Forest model with its hyperparameters achieving an F1-Score of 0.88. In contrast to that, the KNN shown in Figure 8b exhibits a better performance with an F1-Score of 0.92. The narrow characteristic of specific splits of trees for certain feature ranges seems to not be able to generalize as well as the KNN that always can rely on other local samples instead of depending on exactly learned values. One thing both models have in common is that they struggle to discriminate between standing and sitting and also walking upstairs and

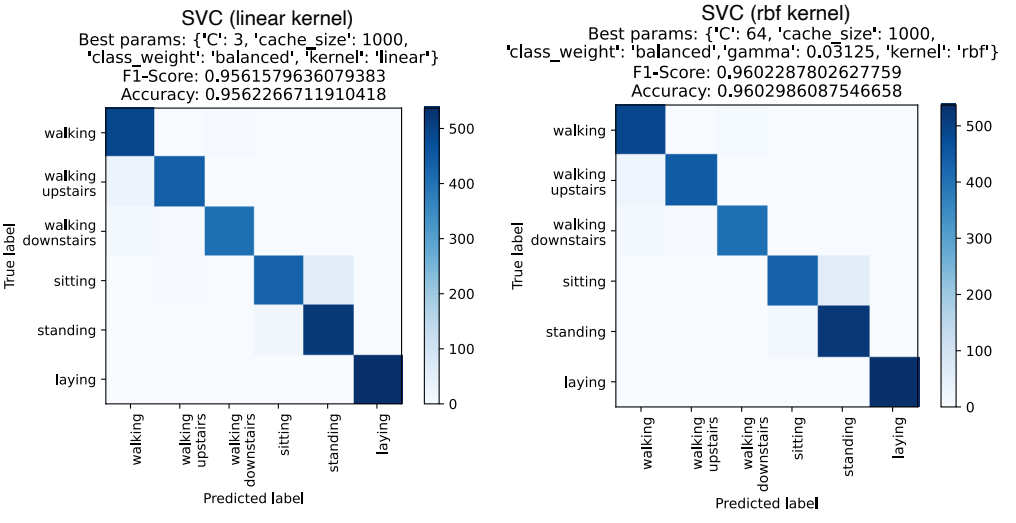
walking downstairs. Even though for the KNN the problem is not as severe as for the Random Forest.




(**a**) (**b**)

**Figure 8.** Confusion matrices of the best performing RF and KNN models with its hyperparameters. (**a**) Random Forest. (**b**) KNN.

In reference to the two results of the Random Forest and KNN models, the results visible in Figure 9a for the SVM with linear kernel and Figure 9b for the SVM with RBF kernel show even better performances. The nature of the support vector machine with its many dimensions in which hyperplanes are able to separate classes seems to generalize very well for the chosen features of the data set. The difference in performance between the linear and RBF kernel versions are only marginal. From the initial data analysis, one would probably assume that a non-linear model would perform better, but as we can see the linear SVM is able to find many good hyperplanes to separate the feature space. Looking at the cost $C$, which influences how much wrong classifications are penalized, the linear version could actually be better at generalizing due to the smaller $C$, which makes it less likely to overfit. Similar to the last two models, the KNN and RF, sitting and standing is still a difficult discrimination but the difference between walking upstairs and walking downstairs is not as big of an issue.




(**a**) (**b**)

**Figure 9.** Confusion matrices of the best performing SVM models with its hyperparameters. (**a**) Support Vector Machine (linear kernel). (**b**) Support Vector Machine (RBF kernel).

### 4.2. Deep Learning

Now that we have seen the results of the machine learning models, it is time to look at the two deep learning architectures, the RNNs and CNNs. Same as the previous section, we will start with the results of the hyperparameter evaluation using an exhaustive grid search in Section 4.2.1 before continuing with the configuration and results of the best models in Section 4.2.2.

#### 4.2.1. Hyperparameter Evaluation

As a short recap, the architecture of the CNN we saw in Table 6, consists of two one dimensional convolutional layers each followed by a dropout layer and a max pooling layer. In the end, a flattening layer puts everything in one dimension and two dense layers reduce to the output shape of our six labels. Figure 10 shows the mean performances of models, trained with a certain hyperparameter, with its standard deviation on the test data. Starting with the plot for num_units_conv1, the number of units of the first convolutional layer, it can be seen that there is a trend towards more units providing better peak performance. Interestingly, for the value of 64, the models tend to have a lower spread of performances visible in the lower standard deviation bars. Leading to the conclusion that for this number of units, the tradeoff between underfitting and overfitting is the best as there are fewer outlier performances and the models seem to consistently perform well. For the number of the units of the second convolutional layer num_units_conv2 this general trend of increasing peak performances is also visible. In this case, the value of 128 seems like the best available tradeoff as it performs better than the lower values but does not bring too much more complexity to the model that could make it overfit more. The same value also seems favourable for the number of units of the dense layer, num_units_dense1.

The two dropout layers that should reduce the possibility for the CNN to overfit by randomly switching off a certain part of the neurons also display interesting results. The two parameters controlling the rate at which each of the layers switches off the neurons, dropout1 and dropout2, clearly show that adding for example a dropout of 0.2 or 20% allows models to be able to generalize better in some cases but unfortunately also to perform worse in some other cases. In order to be sure not to overfit the data, we see the value of 0.2 for both dropouts as the most favourable. Another parameter that tries to limit overfitting is the kernel regularizer for the first dense layer, *kernel_regularizer_dense*1. In contrast to the dropout, in this case, the potential of the regularizer does not seem to warrant the use as the models tend to get more worse than they get better, which is why it is not used going forward. For the kernel size of the two convolutional layers, kernel_size_conv1 and kernel_size_conv2, that affect how many values are considered for each step of the convolution, the bigger value of 9 is favourable even though it does not have that much impact. Lastly, the pool sizes of the max-pooling layers, pool_size_maxpooling1 and pool_size_maxpooling2. They do not have any effect and therefore 2 is used going forward.

To tune the RNN architecture, as previously explained in Table 7, the five hyperparameters visible in Figure 11 have been evaluated as they are expected to change most of the behaviour. The leading two hyperparameters in terms of influence on the performance of the models are num_units_lstm1 and num_units_lstm2, the amounts of units of the two LSTM layers. For both of these parameters, there is a very interesting trend visible at the highest evaluated value of 512, where the trained models have the potential for very high highs but also for very low lows as can be seen by the comparably lower mean values and the comparably high standard deviations. This indicates that the models using these higher values tend to be too complex to generalize well enough as they can accidentally perform very well for the test set or just the opposite. Given this fact, we view 128 as the most promising value for the number of units of both LSTM layers. In order for the model to be less likely to overfit, two dropout layers that disable a certain rate of the neurons of the previous layers have been added after each of the LSTM layers. The performances of models with different dropout rates for dropout1 and dropout2 show that they do not have as much effect on the RNN as they have on the CNN. The second dropout layer has its best

effect when using a dropout rate of 0.2 or 20%, even though small. For the last evaluated parameter num_units_dense1, the number of units of the first dense layer the results do not differ that much. To be in line with the number of units of the LSTM layers, we choose the value of 128 going forward.

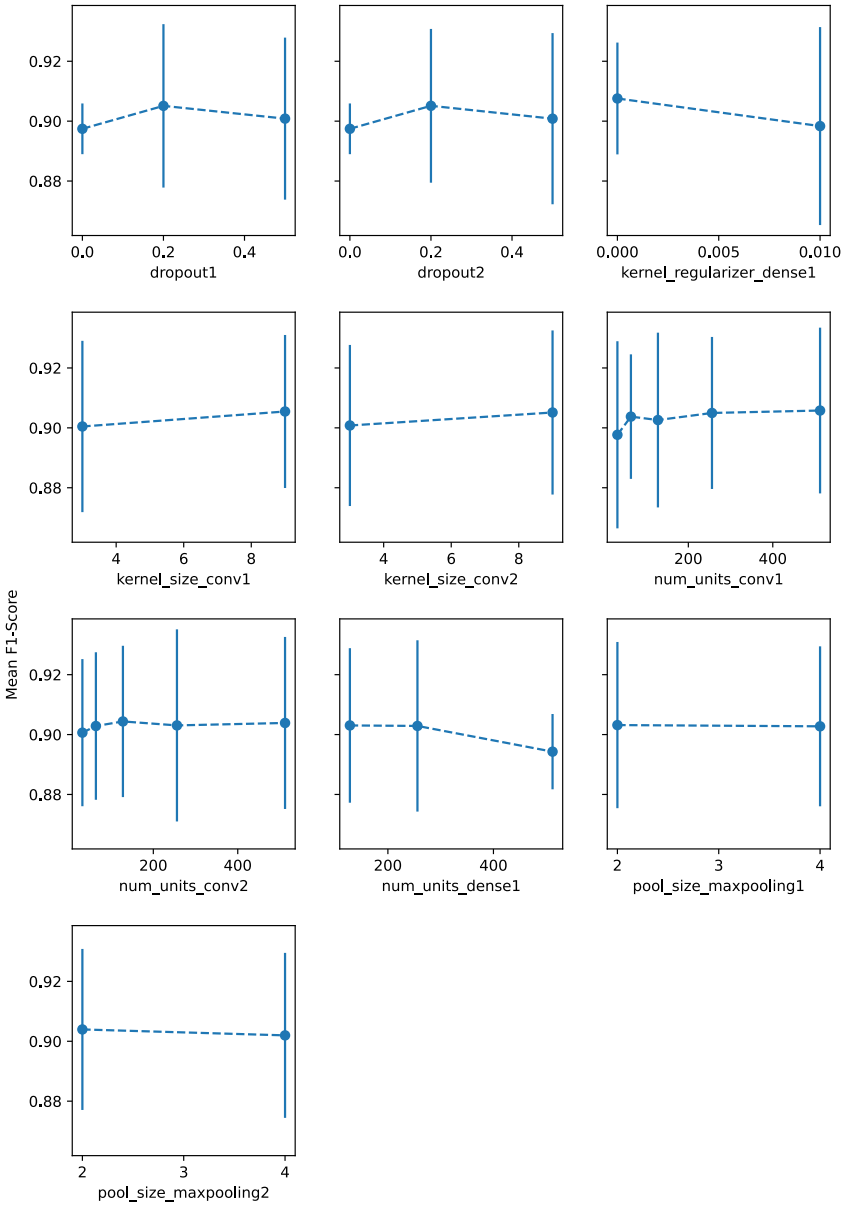



**Figure 10.** Mean test performance of CNN models for respective evaluated hyperparameters with standard deviation.

#### 4.2.2. Best Performing Models

Using the insights gained from the hyperparameter evaluation in the previous section, we use the hyperparameters with the best potential to train a final RNN and CNN and get a comparison of their performance and characteristic by testing on the 30% test set. Table 10 shows the set of parameters that have been used to train the final CNN. While many of them have been evaluated, some of them were left constant and were therefore not mentioned in the previous hyperparameter evaluation section. The model achieves an F1-Score of 0.919. It performs well given we chose a few parameters that decrease peak performance but should also decrease the risk of overfitting. An area where it does not perform that well is the discrimination of the sitting and standing activities which can also be seen in the performances of most of the machine learning models. However, on

the other hand, the difference between walking upstairs and walking downstairs, that many other models struggle with to a varying extent, seems to be less of a problem for the CNN. In contrast to the CNN, the RNN performs a little bit worse overall with an F1-Score of 0.903. Table 11 shows the used hyperparameters for the final trained RNN. The performance is probably not as high as someone would expect having seen the results of the hyperparameter evaluation in Section 4.2.1. This is due to the fact that we intentionally chose certain parameters to not extract peak performance but do more to combat the risk of overfitting instead.

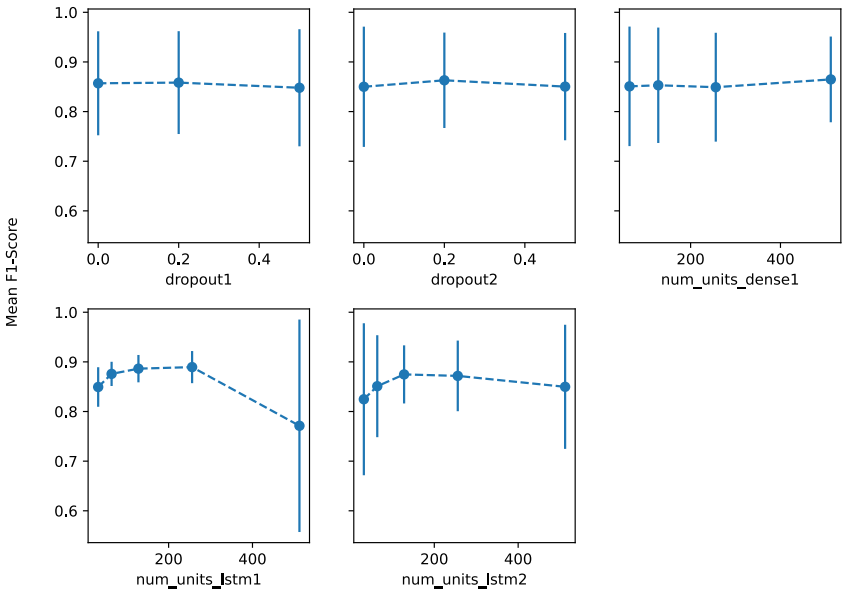



**Figure 11.** Mean test performance of RNN models for respective evaluated hyperparameters with standard deviation.

**Table 10.** Used hyperparameters for the training of the best CNN.

| Hyperparameter | Value |
|---|---|
| Number of Epochs | 10 |
| Batch Size | $2^7$ |
| Optimizer | RMSprop |
| Loss Function | Categorical Crossentropy |
| Number of Units (Convolutional Layer 1) | $2^6$ |
| Number of Units (Convolutional Layer 2) | $2^7$ |
| Kernel Size (Convolutional Layer 1) | $3^2$ |
| Kernel Size (Convolutional Layer 2) | $3^2$ |
| Activation Function (Convolutional Layer 1) | ReLU |
| Activation Function (Convolutional Layer 2) | ReLU |
| Dropout 1 | 0.2 |
| Dropout 2 | 0.2 |
| Pooling Size (Max Pooling Layer 1) | $2^1$ |
| Pooling Size (Max Pooling Layer 2) | $2^1$ |
| Number of Units (Dense Layer 1) | $2^7$ |
| Kernel Regularization (Dense Layer 1) | 0 |
| Activation Function (Dense Layer 1) | ReLU |
| Activation Function (Dense Layer 2) | Softmax |

**Table 11.** Used hyperparameters for the training of the best RNN.

| Hyperparameter | Value |
|---|---|
| Number of Epochs | 10 |
| Batch Size | $2^7$ |
| Optimizer | RMSprop |
| Loss Function | Categorical Crossentropy |
| Number of Units (LSTM Layer 1) | $2^7$ |
| Number of Units (LSTM Layer 2) | $2^7$ |
| Dropout 1 | 0.0 |
| Dropout 2 | 0.2 |
| Number of Units (Dense Layer 1) | $2^7$ |
| Activation Function (Dense Layer 1) | ReLU |
| Activation Function (Dense Layer 2) | Softmax |

## 5. Discussion

The previous section highlighted the results of the trained models. This section will now compare and evaluate those models and results according to the evaluation criteria defined in Section 3.1. The first being the prediction performance of the models in Section 5.1 and the complexity of the models in Section 5.2.

### *5.1. Performance*

Looking at the performance of the best models of each model type depicted in Table 12, a few interesting points become apparent. The first is that the best performing model in terms of F1-Score or accuracy is not one of the deep neural nets, but one of the machine learning models, the SVM with RBF kernel with an F1-Score of 96.02% followed closely by the SVM with linear kernel with 95.62%. Whereas the deep neural nets only achieve an F1-Score of 91.98% for the CNN and 90.34% for the RNN. Of course, considering that the machine learning models need highly engineered features in contrast to the deep neural nets, they can still achieve very good performances with just the preprocessed sensor data. What also can be seen is that the KNN works very well (F1-Score of 92.07%) considering its simple approach of using the closest available samples to predict a new sample. Unfortunately, with this simple approach also comes a very big drawback with its high memory usage (see Section 3.1.2). Interestingly, the Random Forest is not really able to keep up with the performances of the other models. We attribute this to the way trees can only split very specific ranges in a node which in turn makes it hard for the different trees to split the very unstructured data we have also seen in the data analysis section (see Section 3.3.3). Of course, trees could perfectly learn to model the training data in this way and overfit, but generalizing well for this kind of data seems difficult.

Coming back to the results of the deep neural nets seen in Table 12, where our CNN (F1-Score of 91.98%) performs better than our RNN (F1-Score of 90.34%) on this particular dataset. But the difference is small and some of it could also be attributed to the fact that the training of RNNs takes much longer than the training of CNNs, which in turn limited the amount of hyperparameter tuning due to the limited resources in Google Colab. Another reason could be that the relevant relations and patterns of values within a sample are better identified using convolutions with different kernels of size 9 than what the LSTM layers are able to find with their long and short term relations.

Would we now need to decide on one model to use in production purely based on these performances, then we would choose one of the SVMs as they show the best performance. However, instead of just going with the best model in terms of performance,

the SVM with RBF kernel, we would choose the SVM with linear kernel. This is due to the fact that the linear SVM only shows a marginal decrease in the performance, but when looking at the hyperparameters, the SVM with RBF kernel has a much higher cost (64 compared to 3 for the linear SVM) and also has a very low gamma, which should make it more likely to overfit than the linear SVM.

**Table 12.** Performances of the best performing models for each evaluated model type on the UCI HAR dataset.

| Model Type | F1-Score | Accuracy |
|---|---|---|
| KNN | 92.07% | 92.09% |
| Random Forest | 88.34% | 88.33% |
| SVM (Linear Kernel) | 95.62% | 95.62% |
| SVM (RBF Kernel) | 96.02% | 96.03% |
| CNN | 91.98% | 91.99% |
| RNN | 90.34% | 90.30% |

To put the results of this work in perspective, Table 13 shows a set of results of related work on the exact same dataset. Looking at the SVM by Anguita et al. [31], the performance is nearly the same with them reporting an accuracy of 96% in comparison to our achieved performance of the two SVMs of around 95.6% to 96%. Unfortunately, there is no mention of the used kernel or any of the other available hyperparameters and therefore only the performance without further context can be related. For our best performing deep neural net architecture, the CNN, there are two data points to relate to. Ronao and Cho [16] achieve an accuracy of 94.79%, while Jiang and Yin [17] report 95.18% accuracy, which are both more than our accuracy of 91.99%. Looking at the finer details of the experiments of Ronao and Cho [16], they report that their CNN performs best when using either two or three layers of one-dimensional convolution and pooling. With the number of units being best at around 120 for the two-layer version and 200 for the three-layer version. This is higher than the number of units we used for the final CNN (64) because we actively chose to reduce the complexity and therefore the chance of overfitting with only marginal drawbacks in terms of performance. Similar to our findings in regards to the best kernel size of the convolutional layers (9 in our case), Ronao and Cho [16] found that higher kernel sizes from 9 to 14 tend to work better than lower kernel sizes. Representing 0.18 s to 0.28 s worth of sensor data being considered for each calculated value in the feature maps.

The performance of Jiang and Yin [17] with an accuracy of 95.18% on the other hand is not as easy to compare in terms of hyperparameters as they use a very different CNN approach. As explained in Section 2.2, they use a so-called activity image as input that contains all the relevant signals multiple times in different order. This allows their two-dimensional convolutional layers to also try to find patterns across multiple neighbouring signals. Additionally, due to the varying order of the repeated signals, one signal has many different neighbouring signals.

**Table 13.** Performances of a set of models of related work on the UCI HAR dataset.

| Paper | Model Type | Accuracy |
|---|---|---|
| Anguita et al. [31] | SVM | 96% |
| Ronao and Cho [16] | CNN | 94.79% |
| Jiang and Yin [17] | CNN | 95.18% |

### *5.2. Model Complexity*

In addition to the prediction performance of the developed models, one factor that influences its value for the desired use case is the model complexity. As defined in Section 3.1, we report the model complexity as three metrics, memory consumption, mean prediction time and number of trainable parameters. This allows us to draw conclusions on the usability of the classifiers, looking at which models can be run on mobile or wearable devices and what their response time is. Using the number of trainable parameters, a typical deep learning metric to compare the complexity of models to put the other two metrics in perspective. For the machine learning models, where the number of trainable parameters is not commonly used and therefore not readily available, we define a set of formulas to approximate it depending on the model type (see Section 3.1).

#### 5.2.1. Memory Consumption

Looking at current smartwatches, memory consumption is not as much of an issue anymore. For example, the Apple Watch Series 5 and onwards already comes with 1 GB of RAM and the Samsung Galaxy Watch 3 already has 1.5 GB of RAM. Therefore developing a mobile app or smartwatch app to deploy HAR machine learning should be no problem. Of course, if you want to create your own wearable device, the memory consumption might matter a bit more if you want to shrink the hardware or to increase battery life. Using the setup explained in Section 3.1, we record the memory usage and arrive at the results visible in Figure 12.

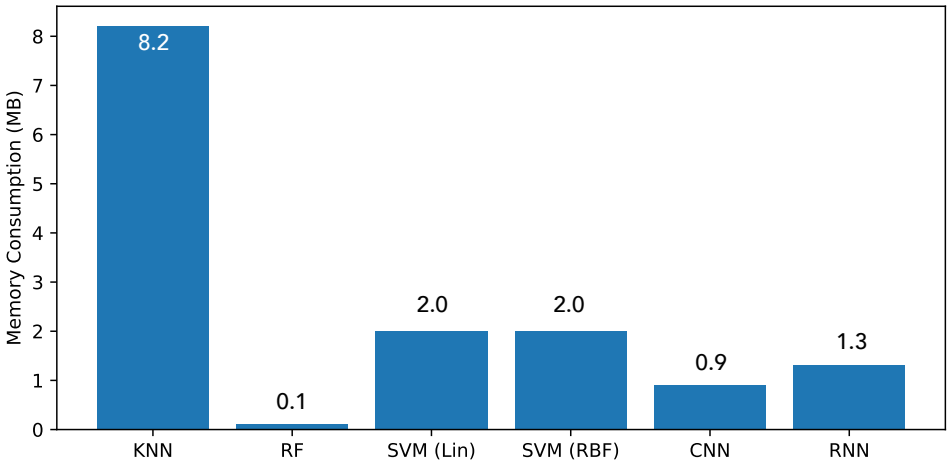



**Figure 12.** Memory consumption in MB for each of the best models.

At first sight, the memory usage of the KNN jumps out with 8.2 MB as it is by far the most of all models. Considering the architecture of the KNN, this definitely makes sense as the training data is basically the model and therefore the whole training data needs to be stored to make predictions on new samples. Especially as our KNN uses the brute algorithm to compute the nearest neighbour, which could definitely need more memory than the other tree-based algorithms. The RF needs the least amount of RAM with only 0.1 MB. The structure of the trees of the RF seems to be the most efficient way to make predictions. Depending on how the trees are loaded into memory, this could also be loaded lazily, where each decision narrows the tree by quite a lot and therefore the number of nodes that need to be loaded into memory could be reduced. For both the support vector machines, the RAM consumption is identical with 2 MB. The different kernel does not change anything in this regard. Interestingly, the deep neural nets use less memory than the support vector machines. The CNN only needs 0.9 MB and the RNN only needs 1.3 MB. Of course, the reason for this could also be the different model format and framework implementation as they are the only ones using Tensorflow Lite instead of Core ML.

#### 5.2.2. Prediction Time

Another metric we capture using the same method (see Section 3.1) as for the memory consumption is the mean prediction time. Figure 13 shows the results of this experiment with the mean prediction time in milliseconds for each of the best models on a random collection of samples. Clearly, the RNN takes the longest to arrive at a prediction with 16.8 ms. Of course, 16.8 ms would still be acceptable in terms of user experience, but as the other models show, it also can be done quite a lot quicker. Another factor to consider here is that these times are achieved on an iPhone XR, which has more processing power than a typical smartwatch or wearable, which might mean that when running on a smartwatch, the RNN could already be too slow for a given use case. The CNN for example does not even need a quarter of the time with 3.1 ms. When training RNNs, it quickly becomes apparent that they take lots more time to train than for example CNNs. Convolutions can be done in parallel, because to calculate the result of one kernel, only a few neighbouring values are needed, but for an RNN, much of the work needs to happen sequentially, as outputs depend on previous outputs. This is why it also makes sense that this applies when predicting a sample. When looking at the machine learning models, one of the advantages of simpler models becomes quite clear. With the KNN and the RF being the quickest models with 0.26 ms and 0.22 ms respectively and the SVMs being a little bit slower with 0.47 ms for the linear version and 0.62 ms for the RBF version.

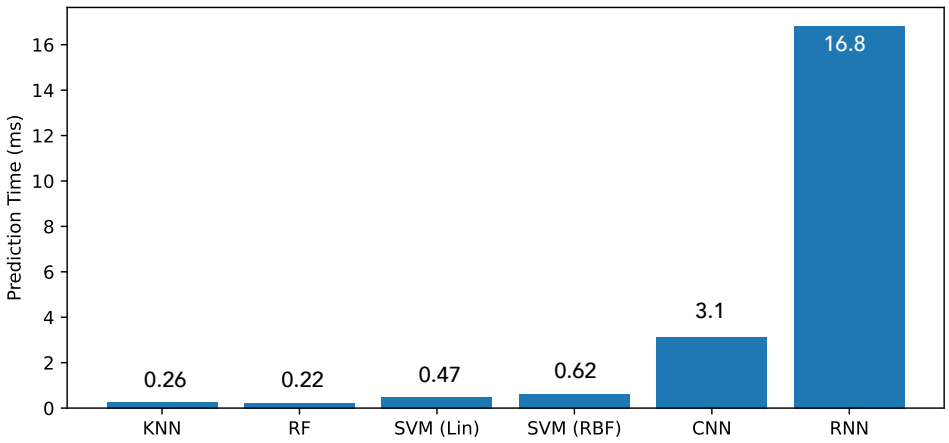



**Figure 13.** Mean prediction time in ms for each of the best models.

#### 5.2.3. Trainable Parameters

With the memory consumption and mean prediction time in mind, we can look at our third metric for model complexity, the number of trainable parameters. As mentioned, for the machine learning models, we approximate this number by using the formulas defined in Section 3.1. For the KNN, it represents the number of samples used as training data (see Equation (1)). To calculate the value for the Random Forest, Equation (2) is used and it is based on the sum of nodes of the trees. Finally, for the SVM, the number of trainable parameters is estimated using Equation (3) and it is based on the number of trained support vectors and the number of features. Having defined these formulas, Figure 14 shows the results of the calculations for the best machine learning models and the values for the best CNN and RNN that can be easily obtained from the model summary. The results are in line with what we would intuitively expect when thinking of the complexity difference of these models, with the deep neural nets being much more complex with 505,990 parameters for the CNN and 219,526 parameters for the RNN. In contrast to that, the RF and SVMs are similar with the RF at 53,063 parameters, the linear SVM at 48,230 parameters and the RBF SVM at 63,700 parameters. The simplest model according to this approximation is the KNN with 7352 parameters. With these and the previous results in mind, we can clearly see that the effect seen in the previous results that are mostly determined by the architecture of the model are not as visible here. For example, the KNN is the simplest model and it is also very fast in terms of prediction time, but it still has the highest memory footprint. The

RNN on the other hand looks to be less complex in this figure, but as we have seen, it is the slowest by quite some margin. Nevertheless, it gives a good initial indication of how these models relate to each other. However, when it comes to adhering to specific requirements of a use case it makes sense to use a metric closer to what is actually the most important factor to determine the applicability, be it memory consumption or prediction time or any other metric.

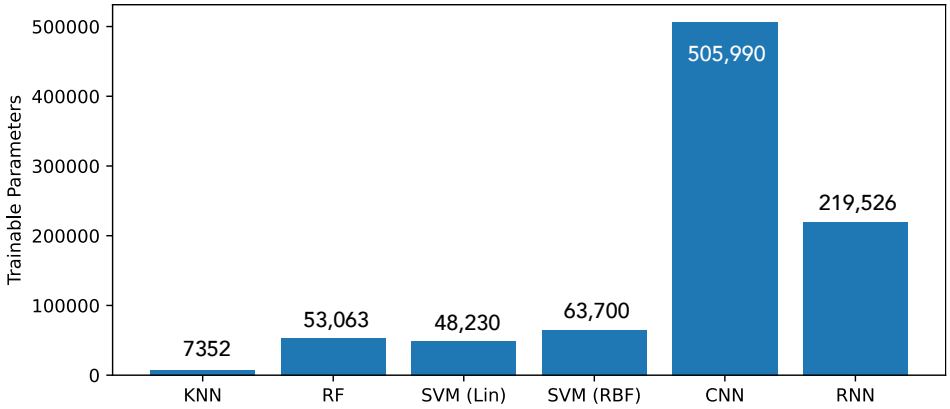



**Figure 14.** Number of trainable paramaters for each of the best models.

### *5.3. Final Models*

Considering all the results presented in the previous sections we now have the ability to determine which model would be most favourable going forward when to goal is to deploy it on a smartwatch for example. Of course when the requirements change, so could the preferred model. For example, for a strictly memory limited use case, the Random Forest does seem like a very good choice. With a good F1-Score of 88.34%, a very fast mean prediction time of 0.22 ms and the very low RAM usage of 0.1 MB, it makes for the perfect candidate for such scenarios. However, of course there are a few other models that only need up to 2 MB of memory but show better prediction performance. For example, the SVM with the linear kernel which needs 2 MB of RAM has a quick mean prediction time of 0.47 ms and an F1-Score of 95.62%. Trading a little more memory usage and prediction time against a better performance and probably also a better ability to generalize for unseen data, as the model is not as limited to the decisions of trees. Additionally, the linear SVM is also a little less complex than the RBF SVM and especially the deep neural nets, giving a good tradeoff between complexity and performance.

## 6. Conclusions

In this paper, based on other research, we trained and evaluated our deep neural nets (CNNs and RNNs) and classic machine learning models (SVM, RF and KNN). The employed techniques and architectures were chosen and defined to be representative concerning common state of the art approaches. We chose a comprehensive public dataset, the UCI HAR dataset [31] to train them using preprocessed sensor data for the deep neural nets and extracted features for the machine learning models. We used metrics for performance (macro-averaged F1-Score and Accuracy) and model complexity (mean prediction time, memory usage and the number of trainable parameters) to compare the models and chose the most favourable to be used in an application. For a strictly memory-limited use case, the Random Forest that achieved an F1-Score of 88.34%, a mean prediction time of 0.22 ms and memory usage of only 0.1 MB, performed best. For a non-memory-limited use case, the SVM with linear kernel performed best with an F1-Score of 95.62%, a mean prediction time of 0.47 ms and memory usage of 2 MB.

As our results show, it does not always have to be the most complex model or the newest of architecture to perform the best. The SVMs clearly perform better than the CNN and especially the RNN. Not only is the performance better in terms of prediction accuracy, prediction speed and memory usage, but normally more complex models also tend to run

a higher risk of overfitting. This is why in many cases it makes sense to start with simpler approaches to get a baseline and then try out more complex ones to then make informed decisions whether an increased performance is worth the added complexity.

Even though in our case the deep neural nets were not on par with some of the machine learning models when it comes to either prediction performance, memory consumption or mean prediction time, they still performed very well. Additionally, it needs to be mentioned again that the deep neural nets were able to do this with just the preprocessed sensor data instead of the extracted features. However, the complexity of the activities and the data also needs to be taken into account. Our dataset only features six very simple activities and only one simple data source with the smartphone. Having more and more complex activities may be combined with data recorded from multiple sources like ambient sensors or even cameras could drastically increase the need for such complex models.

*Limitations and Future Work*

As this paper builds on a dataset recorded in laboratory conditions and the experiments carried out by us also only use random samples of the test data on a mobile device, there are no results on the real-world performance of those models. This could be remedied by extending the developed iOS application that already deploys the models with the full data preprocessing pipeline and feature extraction steps that are used for the dataset and use it to test performances in varying conditions. Furthermore, the dataset and the developed models are limited to six simple activities and the results with datasets with more and also more complex activities could favour complex models. Therefore, it would be interesting to train the model configurations of this work on different and more complex datasets as well. Especially for the deep neural nets, this could give more insight into their possibilities and it could also show if the performance is very specific to this dataset or if the same architecture actually can handle diverse datasets. Conclusively, a guideline could be developed to aid researchers in the process of selecting a suitable model for a given dataset. In order to formulate such a guideline, additional tests and analyses are required.

**Author Contributions:** Conceptualization, S.A. and A.P.; methodology, S.A. and A.P.; software, S.A.; validation, S.A., A.P. and S.S.; formal analysis, S.A.; investigation, S.A.; resources, A.P. and S.S.; data curation, S.A.; writing—original draft preparation, S.A.; writing—review and editing, A.P., S.A., S.S. and M.K.; visualization, S.A.; supervision, A.P., S.S. and M.K.; project administration, M.K. All authors have read and agreed to the published version of the manuscript.

**Funding:** Open Access Funding by the University for Continuing Education Krems, the University of Applied Sciences BFI Vienna and the University of Applied Sciences Upper Austria.

**Institutional Review Board Statement:** Not applicable.

**Informed Consent Statement:** Not applicable.

**Data Availability Statement:** The dataset used in this work is the publicly available UCI HAR Smartphone dataset by Anguita et al. [31].

**Conflicts of Interest:** The authors declare no conflict of interest.

## References

1. Weiss, G.M.; Lockhart, J.W.; Pulickal, T.T.; McHugh, P.T.; Ronan, I.H.; Timko, J.L. Actitracker: A Smartphone-Based Activity Recognition System for Improving Health and Well-Being. In Proceedings of the 2016 IEEE International Conference on Data Science and Advanced Analytics (DSAA), Montreal, QC, Canada, 17–19 October 2016; pp. 682–688.
2. Chen, M.; Li, Y.; Luo, X.; Wang, W.; Wang, L.; Zhao, W. A novel human activity recognition scheme for smart health using multilayer extreme learning machine. *IEEE Internet Things J.* **2018**, *6*, 1410–1418. [CrossRef]
3. Rashidi, P.; Mihailidis, A. A Survey on Ambient-Assisted Living Tools for Older Adults. *IEEE J. Biomed. Health Inform.* **2013**, *17*, 579–590. [CrossRef]
4. Mashita, T.; Shimatani, K.; Iwata, M.; Miyamoto, H.; Komaki, D.; Hara, T.; Kiyokawa, K.; Takemura, H.; Nishio, S. Human activity recognition for a content search system considering situations of smartphone users. In Proceedings of the 2012 IEEE Virtual Reality Workshops, Costa Mesa, CA, USA, 4–8 March 2012; IEEE: Piscataway, NJ, USA, 2012; pp. 1–2.

5. Ho, J.; Intille, S.S. Using context-aware computing to reduce the perceived burden of interruptions from mobile devices. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, Portland, OR, USA, 2–7 April 2005; pp. 909–918.
6. Mehr, H.D.; Polat, H.; Cetin, A. Resident activity recognition in smart homes by using artificial neural networks. In Proceedings of the 2016 4th International Istanbul Smart Grid Congress and Fair (ICSG), Istanbul, Turkey, 20–21 April 2016; IEEE: Piscataway, NJ, USA, 2016; pp. 1–5.
7. Pärkkä, J.; Cluitmans, L.; Ermes, M. Personalization Algorithm for Real-Time Activity Recognition Using PDA, Wireless Motion Bands, and Binary Decision Tree. *IEEE Trans. Inf. Technol. Biomed.* **2010**, *14*, 1211–1215. [CrossRef] [PubMed]
8. Yamato, J.; Ohya, J.; Ishii, K. Recognizing human action in time-sequential images using hidden markov model. In Proceedings of the 1992 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, Champaign, IL, USA, 15–18 June 1992; IEEE: Piscataway, NJ, USA, 1992; pp. 379–385.
9. Lockhart, J.W.; Pulickal, T.; Weiss, G.M. Applications of mobile activity recognition. In Proceedings of the 2012 ACM Conference on Ubiquitous Computing, Pennsylvania, PA, USA, 5–8 September 2012; ACM: New York, NY, USA, 2012; pp. 1054–1058.
10. Brémond, F.; Thonnat, M.; Zúniga, M. Video-understanding framework for automatic behavior recognition. *Behav. Res. Methods* **2006**, *38*, 416–426. [CrossRef] [PubMed]
11. Bao, L.; Intille, S.S. Activity recognition from user-annotated acceleration data. In *International Conference on Pervasive Computing*; Springer: Berlin/Heidelberg, Germany, 2004; pp. 1–17.
12. He, Z.; Jin, L. Activity recognition from acceleration data based on discrete consine transform and SVM. In Proceedings of the 2009 IEEE International Conference on Systems, Man and Cybernetics, San Antonio, TX, USA, 11–14 October 2009; pp. 5041–5044. [CrossRef]
13. Krishnan, N.C.; Panchanathan, S. Analysis of low resolution accelerometer data for continuous human activity recognition. In Proceedings of the 2008 IEEE International Conference on Acoustics, Speech and Signal Processing, Las Vegas, NV, USA, 30 March–4 April 2008; IEEE: Piscataway, NJ, USA, 2008; pp. 3337–3340. [CrossRef]
14. Bayat, A.; Pomplun, M.; Tran, D.A. A study on human activity recognition using accelerometer data from smartphones. *Procedia Comput. Sci.* **2014**, *34*, 450–457. [CrossRef]
15. Hammerla, N.Y.; Halloran, S.; Plötz, T. Deep, convolutional, and recurrent models for human activity recognition using wearables. *arXiv* **2016**, arXiv:1604.08880.
16. Ronao, C.A.; Cho, S.B. Human activity recognition with smartphone sensors using deep learning neural networks. *Expert Syst. Appl.* **2016**, *59*, 235–244. [CrossRef]
17. Jiang, W.; Yin, Z. Human activity recognition using wearable sensors by deep convolutional neural networks. In Proceedings of the 23rd ACM International Conference on Multimedia, Brisbane, Australia, 26–30 October 2015; ACM: New York, NY, USA, 2015; pp. 1307–1310.
18. Guan, Y.; Plötz, T. Ensembles of deep lstm learners for activity recognition using wearables. In *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*; ACM: New York, NY, USA, 2017; Volume 1, pp. 1–28.
19. Almaslukh, B.; AlMuhtadi, J.; Artoli, A. An effective deep autoencoder approach for online smartphone-based human activity recognition. *Int. J. Comput. Sci. Netw. Secur.* **2017**, *17*, 160–165.
20. Zhang, L.; Wu, X.; Luo, D. Recognizing human activities from raw accelerometer data using deep neural networks. In Proceedings of the 2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA), Miami, FL, USA, 9–11 December 2015; IEEE: Piscataway, NJ, USA, 2015; pp. 865–870.
21. Ignatov, A. Real-time human activity recognition from accelerometer data using Convolutional Neural Networks. *Appl. Soft Comput.* **2018**, *62*, 915 – 922. [CrossRef]
22. Lee, S.M.; Yoon, S.M.; Cho, H. Human activity recognition from accelerometer data using Convolutional Neural Network. In Proceedings of the 2017 IEEE International Conference on Big Data and Smart Computing (BigComp), Jeju, Korea, 13–16 February 2017; pp. 131–134. [CrossRef]
23. Gao, W.; Zhang, L.; Huang, W.; Min, F.; He, J.; Song, A. Deep neural networks for sensor based human activity recognition using selective kernel convolution. *IEEE Trans. Instrum. Meas.* **2021**, *2021*, 2512313.
24. Qin, Z.; Zhang, Y.; Meng, S.; Qin, Z.; Choo, K.K.R. Imaging and fusing time series for wearable sensor-based human activity recognition. *Inf. Fusion* **2020**, *53*, 80–87. [CrossRef]
25. Teng, Q.; Wang, K.; Zhang, L.; He, J. The layer-wise training convolutional neural networks using local loss for sensor-based human activity recognition. *IEEE Sens. J.* **2020**, *20*, 7265–7274. [CrossRef]
26. Pienaar, S.W.; Malekian, R. Human activity recognition using LSTM-RNN deep neural network architecture. In Proceedings of the 2019 IEEE 2nd Wireless Africa Conference (WAC), Pretoria, South Africa, 18–20 August 2019; IEEE: Piscataway, NJ, USA, 2019; pp. 1–5.
27. Inoue, M.; Inoue, S.; Nishida, T. Deep recurrent neural network for mobile human activity recognition with high throughput. *Artif. Life Robot.* **2018**, *23*, 173–185. [CrossRef]
28. Mekruksavanich, S.; Jitpattanakul, A. Lstm networks using smartphone data for sensor-based human activity recognition in smart homes. *Sensors* **2021**, *21*, 1636. [CrossRef] [PubMed]

29. Li, X.; Luo, J.; Younes, R. ActivityGAN: generative adversarial networks for data augmentation in sensor-based human activity recognition. In Proceedings of the 2020 ACM International Joint Conference on Pervasive and Ubiquitous Computing and Proceedings of the 2020 ACM International Symposium on Wearable Computers, Virtual Event, 12–17 September 2020; ACM: New York, NY, USA, 2020; pp. 249–254.
30. Wang, J.; Chen, Y.; Hao, S.; Peng, X.; Hu, L. Deep learning for sensor-based activity recognition: A survey. *Pattern Recognit. Lett.* **2019**, *119*, 3–11. [CrossRef]
31. Anguita, D.; Ghio, A.; Oneto, L.; Parra, X.; Reyes-Ortiz, J.L. A public domain dataset for human activity recognition using smartphones. In Proceedings of the 21th European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning, ESANN 2013, Bruges, Belgium, 24–26 April 2013
32. Kwapisz, J.R.; Weiss, G.M.; Moore, S.A. Activity recognition using cell phone accelerometers. *ACM SigKDD Explor. Newsl.* **2011**, *12*, 74–82. [CrossRef]