Bachelor's Programme in Electrical Engineering

Supervised learning for sensor data classification

Supervised learning applied to human activity recognition

Eemil Nyyssönen

Aalto University

BACHELOR'S THESIS

Aalto University
BACHELOR'S THESIS 2021

Supervised learning for sensor data classification

Supervised machine learning algorithms applied to human activity recognition

Eemil Nyyssönen

Thesis submitted in partial fulfillment of the requirements for the degree of Bachelor of Science in Technology.
Otaniemi, 29 Apr 2022

Supervisor:	senior university lecturer, docent Samuli Aalto
Advisor:	associate professor Esa Ollila

Aalto University
School of Electrical Engineering
Bachelor's Programme in Electrical Engineering

Aalto University

Abstract

Aalto University, P.O. Box 11000, FI-00076 Aalto www.aalto.fi

Author
Eemil Nyyssönen

Title
Supervised learning for sensor data classification

School School of Electrical Engineering

Degree programme Bachelor's Programme in Electrical Engineering

Major Information Technology **Code** ELEC3015

Supervisor senior university lecturer, docent Samuli Aalto

Advisor associate professor Esa Ollila

Level Bachelor's thesis **Date** 29 Apr 2022 **Pages** 30 **Language** English

Abstract

The first part of this thesis consists of a literature survey on supervised learning, focusing on neural network- and support vector machine classifiers and their applications for human activity recognition based on smart phone sensor data. In the second part, these two learning algorithms are evaluated on smartphone sensor data in a case study. The main source of information was text-books on statistical machine learning along with conference papers in the area of human activity recognition i.e. performing classification from sensor data.

Supervised learning aim to learn the underlying mapping between input data (e.g. sensor readings) and corresponding output (e.g. test subject walking). Critical part in supervised learning is evaluating the performance, i.e., the ability to successfully classify the input to its true output class. In the thesis we also explain the basic performance measure, the confusion matrix, and related metrics such as precision and f1-score.

Classification algorithms build an estimate of the output by using the available training data. Neural network and support vector machine are such algorithms. Neural networks use nonlinear statistical modeling to classify samples. Important factors in neural network model training are the choice of the loss function and activation functions. The optimization problem in neural networks is handled by stochastic gradient descent method which uses backpropagation method for computing the gradient. Support vector machine aims to find an optimal separating hyperplane that minimizes its objective function.

The last part of the thesis contains a case study. The case study demonstrates the power of the mentioned classification algorithms by using multi-dimensional sensor readings as data. In this thesis two different models was presented. Neural network classifier achieved 94.67% accuracy and F1-score of 0.9478. Support vector machine achieved 95.05% accuracy and 0.9504 F1-score. Results suggest the models could be further adopted for use in real-time mobile app for example.

Keywords signal processing, supervised learning, neural networks, support vector machine, human activity recognition

urn https://aaltodoc.aalto.fi

Tekijä
Eemil Nyyssönen

Työn nimi
Ohjattu koneoppiminen sensoridatan luokitteluun

Korkeakoulu Sähkötekniikan korkeakoulu

Koulutusohjelma Sähkötekniikan kandidaattiohjelma

Pääaine Informaatioteknologia **Koodi** ELEC3015

Vastuuopettaja Vanhempi yliopistonlehtori, Dosentti Samuli Aalto

Ohjaaja Professori (Associate Professor) Esa Ollila

Työn laji Kandidaatintyö **Päiväys** 29.04.2022 **Sivuja** 30 **Kieli** englanti

Tiivistelmä

Ihmisen suorittaman aktiviteetin tunnistuksesta on tullut monelle arkipäiväistä aktiivisuusrannekkeiden ja älypuhelinten suosion kasvaessa. Aktiviteettien tunnistus perustuu laitteesta kerättyyn sensoridataan, jota tutkimalla voidaan muodostaa matemaattinen funktio, jonka avulla aktiviteetit voidaan luokitella aktiviteettia vastaaviin kategorioihin. Älypuhelinten suoritinpiirit ovat kehittyneet hyvin tehokkaiksi, ja ne mahdollistavat reaaliaikaisen aktiviteetin tunnistuksen sisäänrakennettujen sensoreiden avulla. Näin puhelimella voidaan seurata helposti esimerkiksi askelten määrää tai tunnistaa vaaralliset kaatumiset ja hälyttää apua. Luokittelija perustuu sen kykyyn oppia malli opetusdatan perusteella, jonka jälkeen oikean aktiviteetin ennustaminen tapahtuu testidatalla. Opetusdata sisältää selittävät muuttujat ts. piirteet, sekä ennustettavat muuttujat tai kohteet (aktiviteetit). Testidata saattaa koostua esimerkiksi reaaliaikaisista sensorilukemista, joille kohteita ei ole tiedossa. Luokittelualgortimeilla on oleellinen rooli aktiviteetin tunnistuksessa, sillä suorituskyvyn optimoinnin lisäksi täytyy pitää mielessä puettavien laitteiden usein rajallinen suorituskyky, ja toisaalta akunkeston optimointi.

Tämän kandidaatintyön tavoitteena on tehdä kirjallisuuskatsaus kahdesta nykyaikaisesta luokittelualgoritmista, joita käytetään laajasti ihmisen aktiviteetin tunnistuksen kirjallisuudessa. Aineisto koostuu alan tutkimuspapereiden lisäksi tilastollista oppimista käsittelevistä perusteoksista. Neuroverkko ja tukivektorikone luokittelijoiden toimintaperiaatteet käydään läpi, mutta sitä ennen kerrataan ohjatun oppimisen perusperiaatteet. Luokittelualgoritmejä esiteltäessä keskitytään keskeiseen optimointiongelmaan, jonka kukin luokittelija pyrkii ratkaisemaan. Lisäksi, luokittimien perusrakenne ja toiminta käydään läpi, jolloin syntyy perusymmärrys kuinka luokitin oppii datasta. Käsitellyt luokittelijat soveltuvat hyvin sensoridatan luokitteluun, perustuen niiden kykyyn oppia hyvin moniulotteisesta datasta. Sensori datasta muodostetaan opetusdata useista erilaisista piirteistä, minkä johdosta opetusdata voi olla hyvin moniulotteinen. Lopuksi esitettyjen luokittelualgoritmien suorituskykyä kokeillaan käytännössä älypuhelimen avulla kerättyyn sensoridataan. Täten käydään samalla läpi kuinka tyypillinen ihmisen aktiviteetin tunnistus tapahtuu käytännössä, ja mitkä tekijät datassa vaikuttavat tuloksiin, sekä selvennetään ja hyödynnetään aiemmin määriteltyjä konsepteja.

Data-analyysissä käytetään julkisesti saatavilla olevaa dataa, johon on kerätty dataa kuudesta eri aktiviteetistä. Sensoreiden lukemat on kerätty älypuhelimella, joka kiinnitettiin testihenkilöiden vyötärölle aktiviteettien suoritusten ajaksi. Datajoukkoon kerättiin dataa yhteensä kolmeltakymmeneltä henkilöltä. Käsitelty data sisältää yhteensä 561 piirrettä, jotka koostuvat raakadatasta lasketuista aika-ja taajuustason tilastollisista määreistä. Kokeellisessa osuudessa osoitetaan, että esitetyt luokittelualgoritmit onnistuvat aktiviteetin tunnistuksessa hyvin. Data-analyysi toteutetaan käyttäen Python ohjelmointikieltä, ja algoritmien toteutuksessa hyödynnetään kirjastoja kuten Tensorflow/Keras ja Scikit-learn. Neuroverkko-luokittelija tunnisti aktiviteetit tarkkuudella 94.67%, kun taas tukivektorikone tarkkuudella 95.05%. Luokittelijoiden arkkitehtuuri ja ennustamiseen tarvittava suoritinteho on resurssitehokas, ja se soveltuisi hyvin reaaliaikaiseen aktiivisuuden tunnistukseen laajennettuna esimerkiksi mobiilisovellukseen.

Avainsanat signaalinkäsittely, ohjattu oppiminen, neuroverkko, tukivektorikone, ihmisen aktiviteetin tunnistus

urn https://aaltodoc.aalto.fi

Contents

1. Introduction

Human activity recognition (HAR) has emerged to be a key research area in mobile and ubiquitous computing [1]. The recent rise in ubiquitous devices such as smartphones and activity trackers, have made monitoring one's health and activity related data a daily function for many people [1]. One goal of HAR is to proactively assist user with their tasks based on the information gathered from the device.

Human activity recognition is an important area of research especially in health technology domain. Since lifestyle in modern world has shifted towards more sedentary, wearable sensors can offer valuable insights to research the links with physical activity and common diseases [1]. Furthermore these insights can be used to aid users to a more healthy lifestyle by suggesting daily activity targets for instance.

The recognition of human activity is accomplished using *classification algorithms*. Data collected from body-worn inertial sensors is used to classify activities in correct categories. Such categories can be for example, the subject performing a certain exercise, walking or any other daily task such as watering the plants.

Activity sensing devices tend to reduce in size, beneficial to the end-user. However the reduction in size applies likewise to battery capacity. Thus, the design of the classification algorithm is obligated to optimize accuracy in favor of elongated battery life. The choice of suitable classification algorithm is therefore essential when performing HAR.

The goal of this thesis is to review two fundamental classification algorithms, and the main concepts in supervised learning to understand the process of classification, from sensor data to correct activity. In Chapter 2, basic concepts in supervised learning are introduced. Furthermore introduction to how to properly assess the performance of a classification algorithm is presented. In Chapter 3, two classification algorithms are examined: neural networks and support vector machine. In Chapter 4, a data-analysis is performed to smartphone sensor data to investigate the performance of classification algorithms. Finally, in Chapter 5, the results and conclusion of this thesis are presented.

2. Background

This chapter presents the fundamental principles used in the typical classification problem. We start by defining the three main concepts that make a machine learning problem: data, model and loss. Next, classification algorithm is described in further detail, and we go over the basics of feature engineering, which is a important part in classification of sensor data. Finally, we review few of the methods used to evaluate the performance of a classification problem.

2.1 Classification: basic principles

In machine learning problems there are set of variables, *inputs* or *features*, which are usually measured values or observed values. These inputs influence the outcome, *outputs* or *labels*. The main task in classification is to predict the output using the input. There is some underlying function f that captures the input-output relationship which we would like to estimate. We do not know f, but we get to observe example input-output pairs (training data) from which f can be estimated. This is called *supervised learning* which is topic of this thesis.

2.1.1 Data

Data point, input and output variables

Data point is a unit of information, usually a design choice. Data point can represent e.g. a pixel in an image, random variables, signal samples of time series generated by sensors or time series generated by a collection of sensors [2]. A data point can be an *input variable* or an *output variable*. The input variable is denoted by $X \in \mathbb{R}^d$, where d is the dimensionality of the vector (equal to number of input features). If X is a vector, components can be accessed by subscripts X_j. The output variable is denoted as Y, where subscripts are accessed by Y_j. The ith observation of input X is denoted by x_i and y_i denotes the ith observed output variable. To make accurate predictions, we often need a great number of data for constructing the prediction rules. When output of the system is categorical or qualitative, the prediction task is called *classification*, and for continuous output variables the task is called *regression*. The classification task can be formulated as follows: for some input vector X learn a satisfactory predictor of output Y, denoted as $\hat{Y} = \hat{f}(X)$, where $\hat{f}$ is the estimated classification rule (classifier or predictor function) based on the training data [3].

Training set

Training set is a known set of measurements for which the prediction rule is based on. Training data is denoted by a set of N (N is called the *sample size*) observed measurements $\mathcal{T} = \{(x_i, y_i), i = 1, \ldots, N\}$, for which we know the ground truth output values y_i. Training data is used in order to train a *model* or *hypothesis* $\hat{f}(x, \boldsymbol{\theta}) : X \to Y$, where $\boldsymbol{\theta}$ is a vector of *hyperparameters* we can use to tune our classifier algorithm. Multiple models form the *hypothesis space* $\mathcal{H}$.

Test set

Test set is independent data that our model has not seen in the model training phase, and is to be used only when testing the model. When training the model, usually an assumption is made, that no future test data is available [4]. Typically when considerable amount of data is available, the data used to determine model complexity can be formed by taking a fraction of data into reserve. This is called the *validation set*. The model is fit into the training data and the performance can then be tested with validation. Usually the final estimation of model accuracy and prediction performance is done on the test data.

2.1.2 Generalization and Cross-validation

In classification, an important aspect of model's performance is *generalization*. This refers to real world performance in cases we have independent test data. Generalization performance guides the process of choosing the model or learning method.

For a way to measure errors between the actual output vector Y and the predicted output from the prediction model $\hat{f}(X)$ a *loss function* is used, which is denoted by $\mathcal{L}(Y, \hat{f}(X))$. There are numerous different loss functions to choose from, but the most common is the *squared error loss*: $\mathcal{L}(Y, \hat{f}(X)) = (Y - \hat{f}(X))^2$ [3].

For measuring the generalization performance, the *test error* or the *generalization error* for a classifier $\hat{f}$ is defined as:

$$R(\hat{f}) = E[\mathcal{L}(Y, \hat{f}(X))|\mathcal{T}], \tag{2.1}$$

where X and Y are drawn randomly from their joint distribution and the training set $\mathcal{T}$ is fixed. Test error refers to the error specific for the training data $\mathcal{T}$. Moreover, this is known as the *population risk* [5]. The population risk is not directly accessible to assess, since both the underlying probability distribution and f (function we wish to estimate) is unknown. However, we can measure the *training error* or *empirical risk*, which captures the average loss over a training sample:

$$\hat{R}(\hat{f}) = \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}(y_i, \hat{f}(x_i)). \tag{2.2}$$

Note that, by the linearity of the expectation, for a fixed $\hat{f}$, the expectation of empirical risk for training data independently drawn from a identical distribution (i.i.d) equals to the population risk:

$$E[\hat{R}(\hat{f})] = R(\hat{f}). \tag{2.3}$$

To choose a predictor $\hat{f}$, we wish to select a model so that we minimize the generalization error. A natural way to choose such predictor is to approximate 2.1 by empirical risk. Thus we wish to minimize the training error

$$\hat{f}_{ERM} = \arg\min_{\hat{f} \in \mathcal{H}} \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}(y_i, \hat{f}(x_i)), \tag{2.4}$$

where hypothesis that minimizes the error yields a classifier $\hat{f}_{ERM}$, and the minimization is done over a hypothesis space $\mathcal{H}$ of functions. This procedure is called *empirical risk minimization* (ERM) [5]. ERM does not always yield a unique solution indicating there can be several functions in $\mathcal{H}$ that minimize the loss.

Generally, the training error is not a favorable estimate of the generalization error since with complex enough model the training error can be driven to zero. In this case the model simply memorizes the correct output for each data point in the training set $\mathcal{T}$. This lead to poor generalization and the model is said to suffer from *overfitting*. In contrast, if model is not complex enough the generalization error is less than training error, and the model has large bias. Hence, the generalization is again poor and the model is said to *underfit*.

For more attractive method of estimating the prediction error the dataset is partitioned into training set used for model training, and validation set, used to approximate the population risk. According to Murphy [4], an usual split into validation and training sets is 80% for the training and the rest 20% for validation. However, if the total data available is limited in size, the number of training instances become insufficient resulting in a poor estimate of future model performance.

Cross-validation (CV) is a simple and popular solution to combat this problem. Cross validation estimates the expected extra-sample error $Err = E[\mathcal{L}(Y, \hat{f}(X))]$, which is the average generalization error when applying the prediction model $\hat{f}(X)$ to an independent test sample from the dataset [3]. The basic idea of K-fold cross validation is that the data $\mathcal{T}$ is split into K *folds*, that are of roughly the same size. For all $k \in \{1, \dots, K\}$, the model is fit on all folds but the k^{th}, and the prediction error is calculated when using k^{th} fold as validation set. In applications, typically K is set to be 5 or 10, the appropriate choice depends on the data available [6].

For data $\mathcal{T}$ of N observations, we randomly split it into k subsamples or folds of equal size $n = \frac{N}{K}$. The k^{th} fold is thus a labeled sample $((x_{k1}, y_{k1}), \dots, (x_{kn}, y_{kn}))$ of size n. Then for each $k \in [K] = \{1, \dots, K\}$, the learning algorithm is trained on all but the k^{th} fold, and we denote the model fitted on the k^{th} fold as $\hat{f}_k$. The performance of the model is tested on the k^{th} fold [6]. Thus we can define the *cross-validation error* for classifier $\hat{f}$ by

$$\hat{R}_{CV}(\hat{f}) = \frac{1}{K} \sum_{k=1}^{n} \frac{1}{n} \sum_{j=1}^{n} \mathcal{L}(y_{kj}, \hat{f}_k(x_{kj})), \tag{2.5}$$

which provides an estimate of the expected prediction error for model $\hat{f}$.

2.2 Feature engineering

Feature engineering is an important part in machine learning, especially when the number of features p is large. *Feature selection* and *feature extraction* are important parts of feature engineering, and they allow for better training times as well as improved generalization performance of our machine learning model [7].

Feature selection aim to select relevant features from the feature-space, while preserving the model performance. Feature selection allows for reduced overfitting and improved model interpretability [5]. Feature extraction aim to perform inter-feature transformations in order to reduce the dimensionality. It is important to not sacrifice large amount of significant information from the original set, if done so it can again lead to poor generalization [7].

For example a popular approach in signal processing is to adopt wavelet transformation $x^* = \mathbf{H}^T x$ to the raw features before using the features x^* as inputs for a *neural network* (see Chapter 3.1) classifier. Wavelets are robust at capturing discrete jumps or edges which may lie in the signal features, thus making separation of activities more clear and improving classification performance [3, 1]. The features required for each application is variable, and usually requires excellent domain knowledge to construct the relevant features.

Feature exctraction is important part of human activity recognition. From the raw inertial sensor data, features are typically calculated by using signal characteristics in both time-domain, and frequency-domain [8]. Some time-domain features include, but are not limited to, mean, median, variance, skewness, kurtosis and range. For frequency-domain typical features are peak frequency, peak power, spectral power, and spectral entropy. From the computed features the feature space can be reduced by feature selection methods such as filters, wrappers, or exhaustive search. Furthermore, feature extraction can be performed to reduce the dimensionality even more. A popular technique is to use *Principal Component Analysis* (PCA), which is a linear technique that transforms the original featured into new mutually uncorrelated features [8].

2.3 Classification performance evaluation

2.3.1 Confusion matrix

Assume classification scheme where we have data with two possible classes: 0 and 1. Thus we have some binary classifier $\hat{f}$, for which we can define the confusion matrix. Let $P(Y = 1|X = x)$ denote the probability that the an observation with input x belongs to class 1. For a fixed threshold parameter τ, we can then consider the following decision rule:

$$\hat{y}_\tau(x) = I(\hat{f}(x) > 1 - \tau), \tag{2.6}$$

where I is the indicator function, that evaluates to 1 if the inequality holds and 0 otherwise. In other words 2.6 assigns $\hat{y}_\tau$ to a class (0 or 1) following the decision threshold [5]. From this decision rule for each τ, the number of *true positives (TP), false positives (FP), true negatives (TN)* and *false negatives (FN)* can be calculated. From these values we can form a table of computed errors, which is called a *confusion matrix* [4]. Now, the empirical number of FP can be calculated as follows:

$$FP_\tau = \sum_{n=1}^{N} I(\hat{y_\tau}(x_n) = 1 \wedge y_n = 0), \tag{2.7}$$

where N is the number of labeled examples. In a similar fashion the empirical number of FN, TP and TN can be calculated [5].

These results can then be stored in a $m \times m$ confusion matrix C, assuming there are m classes. In the matrix C, rows represent the ground truth labels, and columns the predicted ones. Thus, C_{ij} denotes the number of items with true label i being classified as j [7]. Values in the diagonal are correct predictions and other subscripts are errors. Simple confusion matrix for a binary classifier is visualized in Figure 2.1.

From the confusion matrix, multiple summary statistics can be derived. In the following definitions we omit the threshold parameter τ to preserve readability. The *true positive rate* (TPR), also called *recall* or *sensitivity* is a measure of classifiers ability to make correct predictions of actual positive samples [7]. It is defined as

$$TPR = \frac{TP}{TP + FN}, \tag{2.8}$$

		Predicted	
		Class 0	**Class 1**
Actual	**Class 0**	True Positive (TP)	False Negative (FN)
	Class 1	False Positive (FP)	True Negative (TN)

Figure 2.1. Example of a 2x2 confusion matrix for binary classifier.

and the *false positive rate* (FPR) is defined as

$$FPR = \frac{FP}{FP+TN}. \tag{2.9}$$

Other important measures are *accuracy*, defined as

$$accuracy = \frac{TP+TN}{TP+FP+FN+TN}, \tag{2.10}$$

and *precision* which captures the fraction of detections that are actually positive:

$$precision = \frac{TP}{TP+FP}. \tag{2.11}$$

For a general measure of classifier's accuracy the *F1-score*, which is the harmonic mean of precision and recall,is defined as

$$F_1 = 2 \cdot \frac{precision \cdot recall}{recall + precision}. \tag{2.12}$$

In case of an unbalanced dataset, the overall accuracy is a poor metric to describe the actual classifier performance. An example of an unbalanced dataset is one having distinct amount of ground truth items per class. In cases where dataset is clearly unbalanced, normalized confusion matrices as well as *precision-recall* (PR) curves should be used to analyze the class confusion matrix [5, 1].

2.3.2 Receiver Operating Characteristic- and Area Under Curve

Using decision threshold parameter τ, TPR against FPR can be plotted as a implicit function of τ. Often setting optimal value of τ beforehand for the classifier's score is strenuous. By plotting the TPR vs FPR all the cases are covered. This analyzing method is called *Receiving Operating Characteristic* (ROC) [1]. The higher the ROC-curve goes, the better the classification algorithm performs, in other words we thus have higher TPR for the same FPR, which is desirable. We can plot different ROC-curves for different hypotheses to same plot to analyse model performance. If two ROC curves cross it means neither hypothesis performs better globally.

The nature of a ROC-curve is sometimes summarized by *area under curve* (AUC). For AUC the higher the score, the better, while maximum value is 1.

3. Classification methods

In this chapter two popular classification methods, the neural networks and support vector machines, and their basic principles are covered briefly.

3.1 Neural Networks

Neural networks [9, 10, 11] can be thought of as nonlinear statistical models, with an aim to approximate some function f^*. Neural networks are typically represented as a network diagram, which is a directed acyclic graph describing the composition of functions bound together. Typically a classifier, $y = f^*(x)$ maps an input variable x to a category y, whereas a *feedforward* network defines a mapping $y = f(x; \theta)$ and learns the parameter θ to minimize the error. Here the term feedforward is used since there is no feedback connections from which the outputs of the model are fed back in the model itself. Thus, information flows through the function which is evaluated from x, through the computations used to define f, to the output y [12].

3.1.1 Structure

Feedforward neural networks' basic architecture can be described by a *directed acyclic graph* (DAG) $G = (L, E)$, where L is the number of layers, and E denotes the graph's edges. The DAG is associated with a weight function $w : E \to \mathbb{R}$ that is a mapping over the graph edges E. Each node or *neuron* in the graph is modeled as a scalar function $\sigma : \mathbb{R} \to \mathbb{R}$, and is called the *activation function* of the node. Each edge in the graph connects the output of some node in layer l to the input of another node in layer $l + 1$. The input value obtained is a weighted sum of all outputs connected to the node, where the weighting is achieved using a weight function w. We also introduce the bias b, that measures how easy it is for a neuron to fire, or how easily it outputs 1. To further demonstrate the structure, we can present the network as DAG visually, as seen in Figure 3.1.

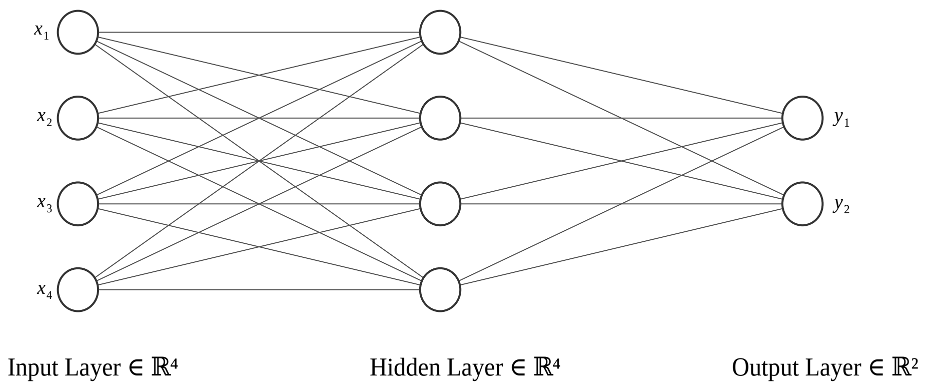
x_1
x_2
x_3
x_4
y_1
y_2
Input Layer $\in \mathbb{R}^4$
Hidden Layer $\in \mathbb{R}^4$
Output Layer $\in \mathbb{R}^2$

Figure 3.1. Example of neural network architecture with four input vectors, one hidden layer and two output units.

The input layer's l_0 values are the observed values of our input vector X. These values are the fed to the next layer.

Finally one can understand neural networks simply as computational graphs, that aims to learn complex functions by the means of recursive composition of simpler functions.

Furthermore, having these graphs parameterized one needs to learn optimal parameters in order to obtain satisfying generalization. That is, minimize the loss function by optimizing the parameters values. These parameters of interest in the context of neural networks are the weights w and the biases b. The optimal weights and biases are learned by *backpropagation*. Backpropagation will be discussed more below.

When training a neural network, the objective is to learn such values for the parameters (w, b) that the model achieves small training error. Thus, we need to define a loss function $\mathcal{L}$ for the model and gain an insight on how changes in weights and biases affect the loss. This means we need to compute

$$\frac{\partial \mathcal{L}}{\partial w_{jk}^l} \text{ and } \frac{\partial \mathcal{L}}{\partial b_j^l}, \tag{3.1}$$

where w_{jk}^l is the weight from k^{th} neuron in the $(l-1)^{th}$ layer to the j^{th} neuron in the l^{th} layer, and similarly b_j^l, as bias of the j^{th} neuron in layer l. This expression captures the rate of change for the loss function when we change the weights and biases. Commonly used loss functions are for example, *cross-entropy loss* and *error loss*, defined as

$$\mathcal{L} = -y_i \cdot \log \hat{y}_i,$$

for cross-entropy and

$$\mathcal{L} = \frac{1}{2}(\hat{y}_i - y_i)^2,$$

for the squared error loss. The losses above are for single instance, and where y_i is the ground truth class and $\hat{y}_i$ is the predicted class.

The relation of successive neurons is seen in the activation function of each node

$$a_j^l = \sigma(\sum_k w_{jk}^l a_k^{l-1} + b_j^l), \tag{3.2}$$

where we take the weighted sum over allkan k nodes in layer $l-1$ connected to the j^{th} neuron in layer l, and add the corresponding bias, to get the activation (output) a_j^l of the j^{th} neuron in layer l. We can present this relation also in vectorized form, if we denote W^l as weight matrix of layer l, that includes all weights that connect to the layer l. Similarly denote b^l as the bias vector that holds the bias for all neurons in layer l, and a^l as vector containing activations from layer l. Thus the vectorized form for activations of a layer is

$$a^l = \sigma(W^l a^{l-1} + b^l). \tag{3.3}$$

In short, the main idea in vectorization is that we wish to apply the activation function σ to every element to its input vector, i.e., function σ is applied element-wise in equation 3.3 [10]. By using this form of view allows for easier understanding of the relations in layers and relieves the index burden. In practice this form allows for fast computation by using matrix multiplication, vector addition and other vector operations, from readily available libraries such as *NumPy* [13] for the python programming language.

In Equation 3.3 the term fed in to the activation function $W^l a^{l-1} + b^l$ is called *weighted input* to the neurons in layer l. It is denoted by $z^l = W^l a^{l-1} + b^l$. Thus we can write $a^l = \sigma(z^l)$, as the output vector [10]. To demonstrate the computations done in a single node, Figure 3.2 visualize the relation of weighted sum z^l to the output a^l.

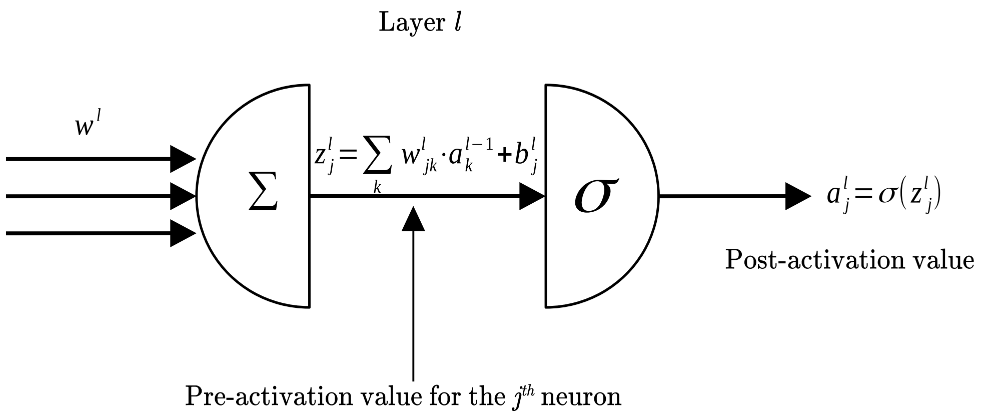
Layer l
w^l
Σ
$z_j^l = \sum_k w_{jk}^l \cdot a_k^{l-1} + b_j^l$
σ
$a_j^l = \sigma(z_j^l)$
Post-activation value
Pre-activation value for the j^{th} neuron

Figure 3.2. The computations done in j^{th} neuron of the l^{th} layer of a neural network, exploded in to two parts to clarify the operations done in neurons. Adapted from [9].

Finally the output layer L outputs K-values for K-class classification scheme, usually transforming the outputs into a probability distribution using the *softmax* (Eq. 3.7) activation function, that scales the outputs a^L to be positive estimates that sum to one. If we denote by a^L the output of the output layer L for a given input x, then the obtained prediction is

$$\hat{f}(x) = \arg\max_k a_k^L, \tag{3.4}$$

where the predicted output belongs to class k, and is determined by taking the maximum likelihood of the final transformation of the output neurons.

Activation functions

The activation function we are going to use in the examples of this section is the *sigmoid* activation function

$$\sigma(v) = \frac{1}{1+\exp(-v)}, \tag{3.5}$$

for an argument $v \in \mathbb{R}$. The derivative expressed in terms of output of the sigmoid can then be written as:

$$\begin{aligned} \sigma'(v) &= \frac{\exp(-v)}{(1+\exp(-v))^2} \\ &= \sigma(v)(1-\sigma(v)). \end{aligned} \tag{3.6}$$

We can see that it is computationally easy to obtain the derivative of the sigmoid function. Furthermore it is usual choice for activation function in classification [3].

Activation function usually used in output layer in multiclass classification, softmax is defined as

$$\sigma(v) = \frac{\exp v}{1 + \exp v}. \tag{3.7}$$

The output of the softmax function is probabilistic, and is commonly used with the cross-entropy loss [9].

Other activation functions typically used are *hard tanh*, and *Rectified Linear Unit* (ReLU), defined as:

$$\sigma(v) = \max\{v, 0\} \ (ReLU), \tag{3.8}$$

$$\sigma(v) = \max\{\min[v, 1], -1\} \ (hard\ tanh). \tag{3.9}$$

In particular, hard tanh and ReLU have mostly replaced the use of sigmoid and soft tanh in modern neural networks to allow for easier computation, especially important in deep multilayer networks [9].

3.1.2 Optimization with backpropagation

Next, the optimization problem in neural networks is described, following Chapter 1 of [9]. Multilayer network evaluates a composition of functions, which are computed in individual nodes. Denote functions computed in m layers as $g_1(\cdot), g_2(\cdot), \ldots, g_m(\cdot)$, and node in layer $m+1$ computing $f(\cdot)$. Thus the composition function for this layer is $f(g_1(\cdot), \ldots, g_m(\cdot))$. If all the layers would use the identity activation function $\sigma(v) = v$, the model would simplify to linear regression. Therefore the loss becomes complicated composition function as described above, thus making the computation of gradient more complicated. For making the computations more efficient, backpropagation algorithm is introduced in order to compute the gradient.

Backpropagation uses chain rule of differential calculus to compute the error gradients in terms of summations of local-gradient products over the path from node to output. This summation has an exponential number of computations, but can be solved efficiently using dynamic programming. Dynamic programming refers to breaking down a complicated problem into sub-problems and then recursively obtaining the solutions to the sub-problems. The backpropagation algorithm can be viewed as an application of dynamic programming [9].

Backpropagation is divided into two subroutines the forward phase or forward pass and the backward phase or backward pass. The forward phase feeds the inputs into the network and computes the following loss (and derivation), while backward phase learns the gradient of the loss function by exploiting the chain rule of differential calculus.

Forward phase

In the forward phase the inputs x_i of the training sample are fed into the neural network. This results in a cascade of computations over all layers, using the current set of weights, until we reach the output layer L. The goal is to compute all the intermediate hidden and output variables, and thus obtain the predicted output $\hat{y} = \arg\max_k a_k^L$. This output value is then used to compute the loss and to calculate the derivative of the loss with respect to the output. Next, the derivative of the loss with respect to weights needs to be computed in the backward phase.

Backward phase

In the backward phase the goal is to learn the gradient of the loss function respecting the different weights. Computation of the gradients allows for updating the weights. Backward phase is initiated from output layer L, and the gradients are learned by propagating in the backward direction. This section follows formulation of backpropagation functions presented in Chapter 2 of [10].

To compute the partial derivative in 3.1 an intermediate quantity δ^l is introduced. It is the error term for the l^{th} layer and is defined by

$$\delta^l = \frac{\partial \mathcal{L}}{\partial z^l}. \tag{3.10}$$

Now, the equation for the error in output layer L can be defined as

$$\delta^L = \frac{\partial \mathcal{L}}{\partial a^L} \cdot \sigma'(z^L), \tag{3.11}$$

that can be derived from 3.10 by using the chain rule. Define $\nabla\mathcal{L}$ as a vector which holds the partial derivatives $\partial\mathcal{L}/\partial a^L$. This allows for writing 3.11 in matrix based form:

$$\delta^L = \nabla\mathcal{L} \odot \sigma'(z^L), \tag{3.12}$$

where $\odot$ is the Hadamard product operator, referring to component wise matrix multiplication of the same dimension.

Finally an equation for error in layer l in terms of the error in the layer

$l+1$ is defined as:

$$\delta^l = (((W^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l), \tag{3.13}$$

where $(W^{l+1})^T$ is the transpose of the weight matrix for layer $l+1$.

Combining 3.11 and 3.13 the error term δ^l is available to compute for any layer l. First, using 3.11, δ^L is computed. Next δ^{L-1} is computed by exploiting 3.13. By propagating backwards to the first layer and continue using 3.13, the error term is computed for each layer l.

An equation for the rate of change of the cost with respect to any bias in the network is thus defined as

$$\frac{\partial \mathcal{L}}{\partial b_j^l} = \delta_j^l. \tag{3.14}$$

Similarly an equation for the rate of change of the cost with respect to any weight in the network is computed by

$$\frac{\partial \mathcal{L}}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l. \tag{3.15}$$

Therefore 3.1 can be calculated. The process of updating weights and biases in nodes is repeated to convergence, cycling through the training data in *epochs*. One cycle through the data is referred as epoch [9].

3.2 Support Vector Machines

In this section we cover the optimization problem, that generalizes linear decision boundaries for classification, called *support vector machine* (SVM). SVM produces nonlinear classification boundaries, by composing a linear boundary in a modified feature space [3]. This allows for finding optimal separating hyperplanes also in the case where the class-conditional distributions overlap, meaning that classes are not linearly separable.

SVM can be used for both for linearly separable training data, and for non-separable data. However in most practical settings the training data in hand is not linearly separable, so we therefore focus on the more general setting which applies to the non-separable data as well.

3.2.1 Formulation

Assume we have linearly separable training data sample $S = (x_1, \ldots, x_N)$, with values belonging into two classes $y_i \in \{-1, 1\}$. This means we assume existence of a hyperplane that separates the sample perfectly into two classes. That is, there exists a non-zero weight vector (normal to the hyperplane) $w \in \mathbb{R}^N$ and scalar $b \in \mathbb{R}$, so that

$$\forall i \in m,\ y_i(w \cdot x_i + b) \geq 0. \tag{3.16}$$

For a linear classifier $f(x) = \text{sgn}(w \cdot x + b)$, typically there exists several hyperplanes that are consistent estimates for the data. Hence we need to have a rule for which classifier our learning algorithm should select.

To determine the best classifier, we wish to find the optimal separating hyperplanes by finding the maximal *geometric margin* over the data. Following definition 5.1 in [6], the geometric margin ρ_f for a linear classifier f is defined as

$$\rho_f(x) = \frac{|w \cdot x + b|}{\|w\|_2}, \tag{3.17}$$

where $\|w\|_2$ is the L2-norm of the weight vector w. Now the *maximum-margin* hyperplane ρ can be computed by choosing the hyperplane $w \cdot x = 0$ that is furthest away from training data, and is given by

$$\rho = \max_{w,b} \min_{i \in [N]} \frac{y_i(w \cdot x_i + b)}{\|w\|}. \tag{3.18}$$

Support vector machines are based on this principle [14]. However, the optimization in 3.18 does not produce unique optimal optimal pair (w^*, b^*) vector, since (w^*, b^*) is invariant to multiplication $(cw^*, cb^*), c > 0$, where c is a scalar. By adding a constraint $\min_{i \in [N]} y_i(w \cdot x_i + b) = 1$, we get:

$$\rho = \max_{\substack{w,b: \\ \min_{i \in [N]} y_i(w \cdot x_i + b) = 1}} \frac{1}{\|w\|} = \max_{\substack{w,b: \\ \forall i \in [N], y_i(w \cdot x_i + b) \geq 1}} \frac{1}{\|w\|}, \tag{3.19}$$

where the second equality follows from the fact that for the maximizing pair (w, b), the minimum of $y_i(w \cdot x_i + b)$ is equal to 1 [6]. Therefore, since we can maximize $1/\|w\|$ by minimizing $\frac{1}{2}\|w\|^2$, the margin maximization problem can now be solved as follows:

$$\min_{\mathbf{w},b} \frac{1}{2}\|w\|^2 \tag{3.20}$$

$$\text{subject to: } y_i(w \cdot x_i + b) \geq 1, \forall i \in [N].$$

The optimization problem in 3.20 is the so called *hard margin* SVM [15]. It returns the optimal weights and bias (w, b) in the separable case, to obtain the maximum margin hyperplane. The maximum margin hyperplane separates the classes with a minimum functional margin of 1. The sample instances that have margin $y(w \cdot x + b = 1)$, are called the support vectors, and they exclusively define the position of the hyperplane.

3.2.2 Non-separable case

Following the definitions given in [6] (Ch. 5.3), we can formulate the support vector machine classifier for non-separable classes. As discussed earlier, in practice we seldom have linearly separable data, meaning that for any hyperplane there exists $x_i \in S$, such that

$$y_i(w \cdot x_i + b) \ngeq 1. \tag{3.21}$$

Therefore the constraint introduced for linearly separable case in 3.20 does not hold. To extend the model for non-separable data, the constraints can be relaxed, that is for each $i \in [m]$ exists a *slack variable* $\xi_i \geq 0$, so that

$$y_i(w \cdot x_i + b) \geq 1 - \xi_i, \xi_i \geq 0. \tag{3.22}$$

The slack variables are used commonly in optimization problems to define constraints in a relaxed manner, and in this context a slack variable ξ_i captures the distance by which the feature vector x_i violates the constraint (inequality) in 3.20. To put simply, by using slack variables we allow the functional margin of some data points to be less than 1 by the value of ξ_i.

Primal optimization problem

The optimization problem in non-separable case, also known as *soft margin SVM*, is defined as

$$\min_{w,b,\boldsymbol{\xi}} \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{N} \xi_i^p \tag{3.23}$$

$$\text{subject to: } y_i(w \cdot x_i + b) \geq 1 - \xi_i \wedge \xi_i \geq 0, i \in [N],$$

where $\boldsymbol{\xi} = (\xi_1, \ldots, \xi_N)^T$ are the *slack variables*, and $C \geq 0$ determines the tradeoff between minimization of $\|w\|^2$, and minimization of the slack penalty $\sum_{i=1}^{N} \xi_i$. The choice of p affects how aggressively the slack terms are penalized. Typical choice is $p = 1$ or $p = 2$, since they lead to most straightforward solutions [6]. Loss functions associated with these values are called *hinge loss* and the *quadratic hinge loss*. However the hinge loss ($p = 1$) is the most widely used loss function with SVMs [6]. Problem 3.23 is quadratic, with linear inequality constraints, hence being a convex optimization problem [3].

4. Exploratory data-analysis

In this section, we test the algorithms covered in this thesis using mobile accelerometer sensor data. Analysis of data is done using *Python* [16] programming language. Popular deep-learning library *Keras* with *TensorFlow* [17] backend was used to train the multilayer neural network model. For visualization *Seaborn* [18] and *Matplotlib* [19] was used.

4.1 Introduction

To conduct the data analysis *Human Activity Recognition Using Smartphones* dataset [20] (HAR dataset) was used. The dataset was built by collecting triaxial linear acceleration and angular velocity signals from test subjects performing Activities of Daily Living (ADL). The sensor data was acquired by mounting Samsung Galaxy S II smartphone to test subject's waist in order to capture data from accelerometer and gyroscope at a sampling rate of 50Hz. The test subject was then instructed to perform six different ADL (*standing, sitting, laying down, walking, walking downstairs, walking upstairs*), in two different occasions to capture also non-clinical placement of the device.

Raw sensor data is noisy in nature and usually need signal processing before usage. *Activity Recognition Chain* (ARC) describes the typical workflow in recognition of human activity. Bulling et al. [1] defines ARC as follows: "An Activity Recognition Chain (ARC) is a sequence of signal processing, pattern recognition, and machine learning techniques that implements a specific activity recognition system behavior". Typically ARC consists of data acquisition, preprocessing, data segmentation, feature extraction and classification as demonstrated in Figure 4.1. Problems using supervised classification algorithms also need model training before classification.

In case of our dataset the raw sensor data was already heavily preprocessed and contained feature vectors in time and frequency domain. For the HAR dataset a median filter and a 3rd order low-pass Butterworth filter with 20Hz cutoff were used in favor of reducing the signal noise. By using another low-pass Butterworth filter the acceleration signal was separated into it's respective components; gravity and body acceleration. Finally before feature extraction, the signals were sampled in 2.56s sliding window, with 50% overlap. From each segmented window feature vector was constructed using different statistical measures and applying them to both time and frequency domain signals.

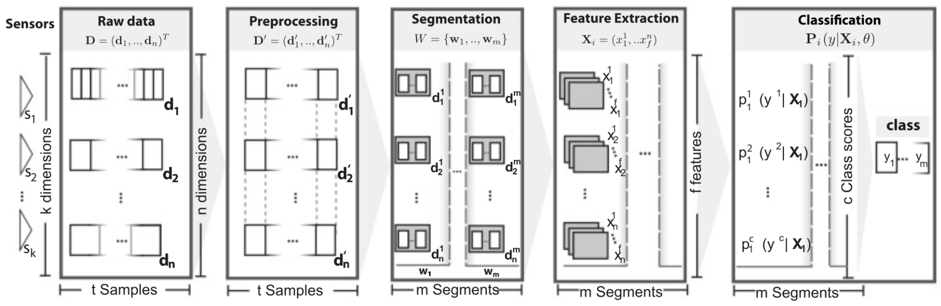
Sensors
Raw data
$\mathbf{D} = (\mathbf{d}_1, .., \mathbf{d}_n)^T$
Preprocessing
$\mathbf{D}' = (\mathbf{d}'_1, .., \mathbf{d}'_n)^T$
Segmentation
$W = \{\mathbf{w}_1, .., \mathbf{w}_m\}$
Feature Extraction
$\mathbf{X}_i = (x_1^1, .. x_f^n)$
Classification
$\mathbf{P}_i(y|\mathbf{X}_i, \theta)$
k dimensions
n dimensions
f features
c Class scores
class
t Samples
t Samples
m Segments
m Segments
m Segments

Figure 4.1. Demonstration of the typical activity recognition chain (ARC) [1].

4.2 Methods

4.2.1 Data exploration

The dataset had already been preprocessed and randomly partitioned into training and testing sets with 70/30 split accordingly. Thus leaving 7352 samples for training and 2947 samples for testing. The total data included both the raw sensor readings and a computed 561-dimensional feature vector. Features are normalized and bounded within $[-1, 1]$. For the sake of this analysis we want to address the suitability of classification algorithm, so we focus on the preprocessed data with features extracted, rather than follow the full pipeline, or ARC, from raw data.

First data was read into a *pandas* [21, 22] DataFrame from .txt files using a python script. Consisting of numerical variables, data was read to floating point numbers with 16-bit precision. As expected due to the preprocessing done, missing values or outliers were both absent. However, class imbalance was present with passive actions having more datapoints with regard to the active ones (see Table 4.1).

Class	Test	Train
WALKING	496	1226
WALKING_UPSTAIRS	471	1073
WALKING_DOWNSTAIRS	420	986
SITTING	491	1286
STANDING	532	1374
LAYING	537	1407

Table 4.1. Table demonstrating the class distributions in both training and test data. Numerical values in columns indicates the count each class is present in the data.

Raw captured acceleration data is in signal form as demonstrated in Fig. 4.2. There are two different activities performed, laying and walking. For subject walking the variation in acceleration naturally follows a sequential pattern, the cadence of subject. As anticipated, when test subject is laying, the signal is monotonous, with low variation.

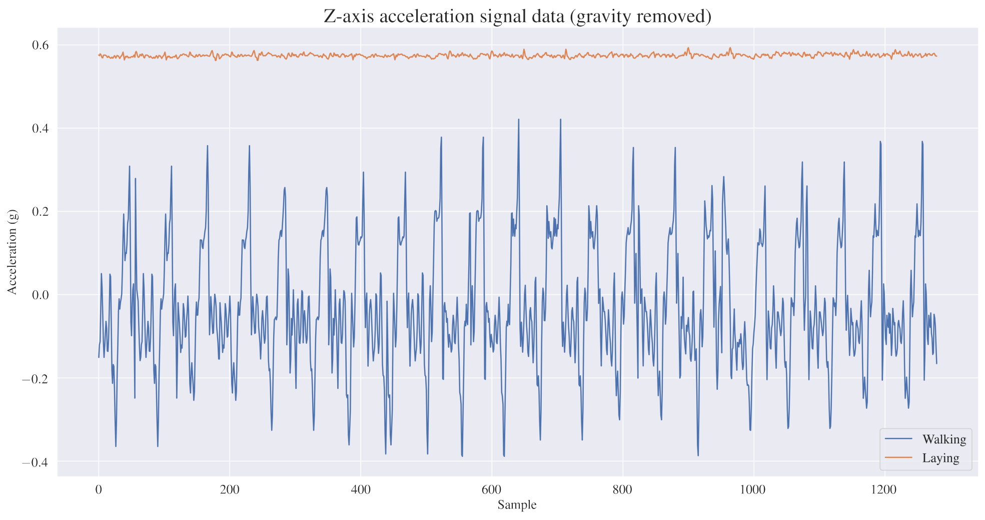

Figure 4.2. Raw acceleration sensor reading, with earth's gravity component subtracted from the signal. Ten samples of 2.56s windows at 50Hz sampling rate, accumulating to 1280 samples and duration of 25.6s.

By inspecting the distributions between classes using mean total body acceleration in frequency domain (*fBodyAccMag-mean()*), one can observe that the distributions between active classes and passive classes are easily distinguishable. The probability density functions (PDF) are estimated by *kernel density estimation* (KDE). KDE is a nonparametric method for estimating the continuous density function from data. It is based on using a Gaussian kernel K to smooth the discrete observations.

Thus one could classify activities between active and passive classes using only simple if-else statements, and still obtain feasible result. Distributions of active classes differ noticeably, while passive ones have less variation in masses since obviously the of acceleration is far inferior, thus resulting in skewness towards the local minimum (-1.0). The probability density functions are demonstrated in Figure 4.3.

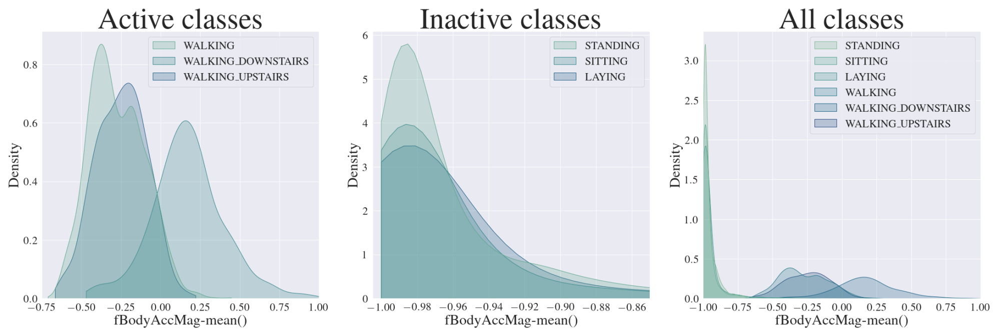

Figure 4.3. Probability density function of mean total body acceleration in frequency domain. The figure demonstrates the differences between active and inactive classes.

4.2.2 Model selection and classification

To begin with, we started with a DNN with depth of N=5 layers. To preserve test data to final evaluation, a fraction (20%) of training data was used for validation during model selection. The validation was done with categorical cross entropy loss, using precision and f1-score as evaluation criteria. For SVM classifier, library scikit-learn was used to obtain an already composed model, with default parameters and a gaussian kernel.

The neural network model was constructed following the architecture proposed in [23]. The model consist of four hidden dense layers and adopts the rectified linear unit (ReLU) as the activation function for the hidden layers. Utilization of dropout with probability 0.2 is used in first three of the hidden layers to prevent the model from overfitting. Finally the output layer outputs a log-probability vector for classifying the sample into it's most probable class. Total trainable parameters thus accumulate to 187355. The architecture is presented in Fig. 4.4.

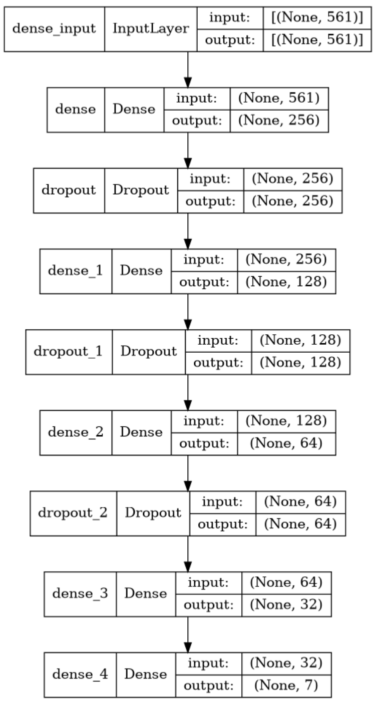
dense_input | InputLayer | input: [(None, 561)] | output: [(None, 561)]
dense | Dense | input: (None, 561) | output: (None, 256)
dropout | Dropout | input: (None, 256) | output: (None, 256)
dense_1 | Dense | input: (None, 256) | output: (None, 128)
dropout_1 | Dropout | input: (None, 128) | output: (None, 128)
dense_2 | Dense | input: (None, 128) | output: (None, 64)
dropout_2 | Dropout | input: (None, 64) | output: (None, 64)
dense_3 | Dense | input: (None, 64) | output: (None, 32)
dense_4 | Dense | input: (None, 32) | output: (None, 7)

Figure 4.4. Neural network architecture used in classification.

4.3 Results

The NN model was fit with the full training data, consisting of 561 feature-vectors. Training was done for 100 epochs, with batch sizes $32, 64$ and 128, from which 128 was selected for the final model as it achieved the lowest validation loss. As stated earlier the validation data was 20% of the training data, allowing for preserving the test data separate from the training process. The model was trained with Adam(adaptive moment estimation)-optimizer with learning rate 0.001, using categorical cross-entropy as the loss function.

Loss, precision, recall and precision-recall-curve as a function of epoch count is visualized in Fig. 4.5. As seen from the figure, both training and

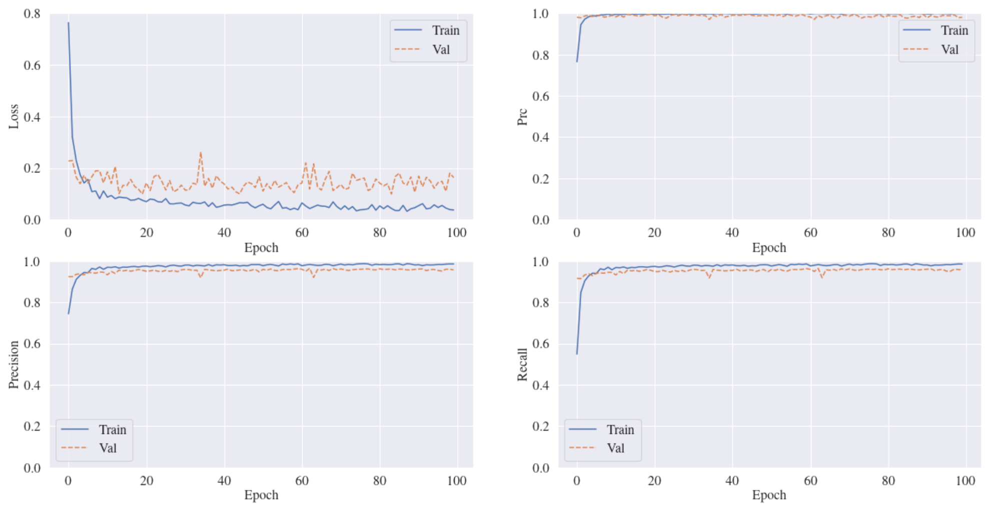

Figure 4.5. Neural network model performance.

validation loss quickly approaches low values, and respectively precision and recall approaches 1. Overfitting was not present since the validation loss remained low throughout the training.

The final results were evaluated on the test data. The NN model achieved 94.67% accuracy, with F1-score 0.9478. The performance is demonstrated by confusion matrix in Fig. 4.6. It can be seen that overall the model performance is acceptable, and for laying activity, all instances were classified correctly. The model confused sitting and standing the most. With sensor placement on the subject's waist sitting and standing are similar activities, with distributions resembling each other.

SVM model performed slightly better with 95.05% accuracy and F1-score of 0.950. Confusion matrix is presented in Figure 4.7. Similiar confusion is present in comparison to NN confusion matrix. However, SVM confused walking downstairs to walking upstairs more compared to NN.

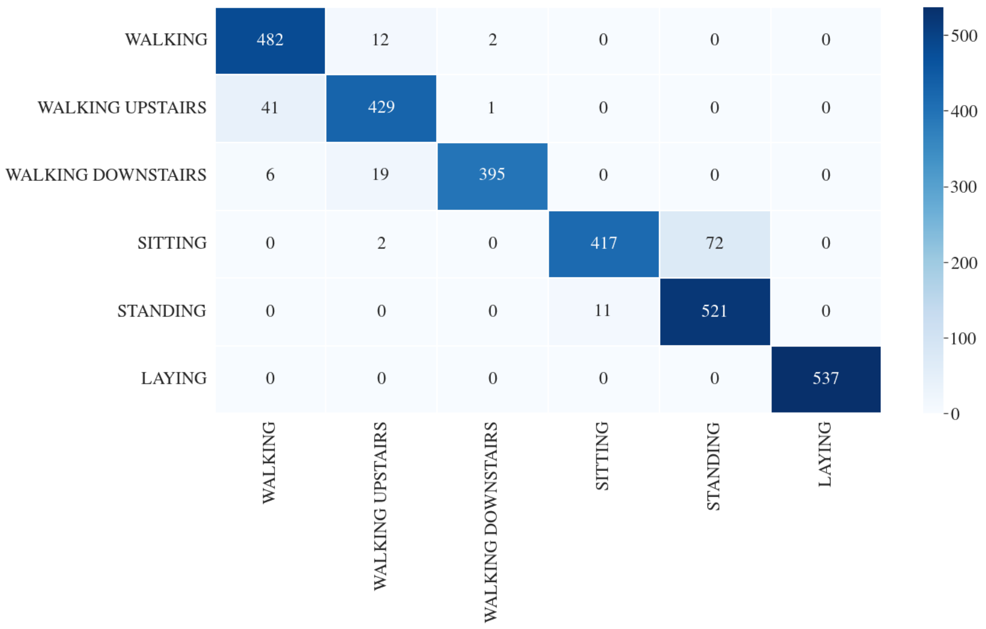

Figure 4.6. Confusion matrix for neural network model, evaluated on test data. Y-axis represents the ground truth class and x-axis the predicted class.

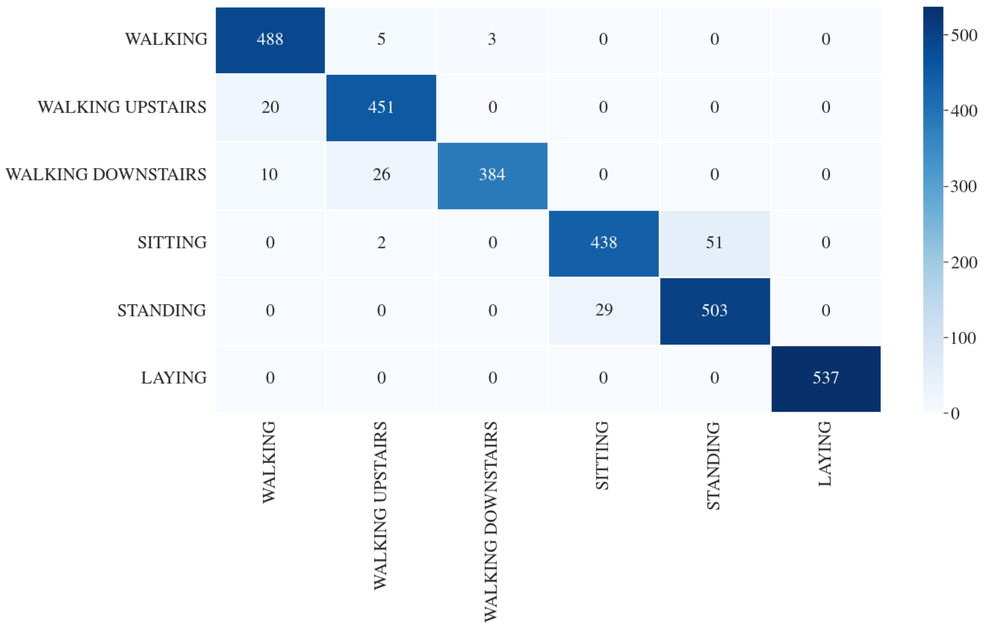

Figure 4.7. Confusion matrix for SVM model, evaluated on test data. Y-axis represents the ground truth class and x-axis the predicted class.

In comparison to the results in [23], our models performed slightly worse than the one proposed in the paper. The NN model proposed in the paper achieved 95.7% accuracy, while the NN model constructed in this thesis achieved 94.67%. Similarly SVM achieved accuracy of 95.05%.

The difference could be present from the absence of feature selection done in this study, whereas in [23] uses non-parametric feature selection method called NCA. Not only could feature selection improve the accuracy, it would significantly reduce the amount of data needed for training.

The results underlines the neural networks capability of fitting high dimensional data, by having a vast amount of trainable parameters. Even with a simple model like this we achieve feasible accuracy, indicating that constructing a real time activity recognition system could be implemented based on this model. However the results were not tested on lower processing capacity computers.

5. Conclusion

In this thesis the main concepts regarding supervised learning and human activity recognition were covered. In particular covering two classification algorithms, neural networks and support vector machine, in regard to the statistical optimization problem present in each algorithm. Principles of supervised learning was described, including how to evaluate performance of different classifiers or the technical background for understanding the classification pipeline. Finally the knowledge was applied to practice by conducting a case study to demonstrate the typical human activity recognition procedure.

The need of human activity recognition in the future is still growing. While mobile computing becomes more inexpensive and more powerful, the classification process from sensor data to activity becomes even more feasible on mobile systems. As shown in Chapter 4 constructing a good model can be done even with relatively shallow networks, even without further parameter optimization. By utilizing feature selection and hyperparameter tuning one could further improve the model performance.

When limitations in processing power becomes less of a burden, the future work may rely on more complex classification models to further optimize the accuracy. Therefore, more sophisticated human actions could be recognized with high performance, for example capturing certain micro-movements could allow for more insight in neurological rehabilitation of patients.

Bibliography

[1] A. Bulling, U. Blanke, and B. Schiele, “A tutorial on human activity recognition using body-worn inertial sensors,” *ACM computing surveys*, vol. 46, no. 3, 2014-01.

[2] A. Jung, *Machine Learning: The Basics*, 2021, https://alexjungaalto.github.io/MLBasicsBook.pdf.

[3] T. Hastie, R. Tibshirani, and J. Friedman, *The elements of statistical learning: data mining, inference and prediction*, 2nd ed. Springer, 2009. [Online]. Available: http://www-stat.stanford.edu/~tibs/ElemStatLearn/

[4] K. P. Murphy, *Machine learning : a probabilistic perspective*, ser. Adaptive computation and machine learning series. Cambridge, MA: MIT Press, 2012.

[5] ——, *Probabilistic Machine Learning: An introduction*. MIT Press, 2022. [Online]. Available: probml.ai

[6] M. Mohri, *Foundations of machine learning*, ser. Adaptive computation and machine learning series. Cambridge, MA: MIT Press, 2012.

[7] J. L. R. Ortiz, *Smartphone-based human activity recognition*, ser. Springer theses. Cham: Springer, 2015 - 2015.

[8] F. Attal, S. Mohammed, M. Dedabrishvili, F. Chamroukhi, L. Oukhellou, and Y. Amirat, “Physical human activity recognition using wearable sensors,” *Sensors.*, vol. 15, no. 12, pp. 31 314–31 338, 2015-12-11.

[9] C. C. Aggarwal, *Neural networks and deep learning : a textbook*. Cham, Switzerland: Springer, 2018.

[10] M. Nielsen, *Neural Networks and Deep Learning*. Determination Press, 2015. [Online]. Available: https://books.google.fi/books?id=STDBswEACAAJ

[11] S. Shalev-Shwartz and S. Ben-David, *Understanding Machine Learning: From Theory to Algorithms*. Cambridge University Press, 2014.

[12] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[13] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau *et al.*, “Array programming with NumPy,” *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020. [Online]. Available: https://doi.org/10.1038/s41586-020-2649-2

[14] J. Rousu, “Lecture 6: Support vector machines,” CS-E4710 Machine Learning: Supervised Methods, Department of Computer Science, Aalto University, Oct 2021.

[15] C. Cortes and V. Vapnik, “Support-vector networks,” *Machine learning*, vol. 20, no. 3, pp. 273–297, 1995.

[16] G. Van Rossum and F. L. Drake, *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace, 2009.

[17] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro *et al.*, “TensorFlow: Large-scale machine learning on heterogeneous systems,” 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[18] M. L. Waskom, “seaborn: statistical data visualization,” *Journal of Open Source Software*, vol. 6, no. 60, p. 3021, 2021. [Online]. Available: https://doi.org/10.21105/joss.03021

[19] J. D. Hunter, “Matplotlib: A 2d graphics environment,” *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007.

[20] D. Anguita, A. Ghio, L. Oneto, X. Parra, and J. L. Reyes-Ortiz, “A public domain dataset for human activity recognition using smartphones,” in *ESANN*, 2013.

[21] T. pandas development team, “pandas-dev/pandas: Pandas,” Feb. 2020. [Online]. Available: https://doi.org/10.5281/zenodo.3509134

[22] Wes McKinney, “Data Structures for Statistical Computing in Python,” in *Proceedings of the 9th Python in Science Conference*, Stéfan van der Walt and Jarrod Millman, Eds., 2010, pp. 56 – 61.

[23] S. K. Bashar, A. Al Fahim, and K. H. Chon, “Smartphone based human activity recognition with feature selection and dense neural network,” in *2020 42nd Annual International Conference of the IEEE Engineering in Medicine Biology Society (EMBC)*, 2020, pp. 5888–5891.