Information Systems xxx (xxxx) xxx

Contents lists available at ScienceDirect

Information Systems

journal homepage: www.elsevier.com/locate/is

DeepMatch2: A comprehensive deep learning-based approach for in-vehicle presence detection

Magnus Oplenskedal [a,b,*], Peter Herrmann [a], Amir Taherkordi [b,c]

[a] Norwegian University of Science and Technology (NTNU), Trondheim, Norway
[b] Forkbeard Technologies, Oslo, Norway
[c] University of Oslo, Oslo, Norway

ARTICLE INFO

Article history:
Received 10 April 2021
Received in revised form 6 October 2021
Accepted 13 October 2021
Available online xxxx
Recommended by Gottfried Vossen

Keywords:
Mobile context
In-vehicle presence detection
Sensor event streams analysis
Deep learning
Event matching
Intelligent transportation

ABSTRACT

The accurate detection of the *mobile context* information of public transportation vehicles and their passengers is a key feature to realize intelligent transportation systems. A topical example is *in-vehicle presence detection* that can, *e.g.*, be used to ticket passengers automatically. Unfortunately, most existing solutions in this field suffer from low spatiotemporal accuracy which impedes their use in practice. In previous work, we addressed this challenge through a deep learning-based framework, called DeepMatch, that allows us to detect in-vehicle presence with a high degree of accuracy. DeepMatch utilizes the smartphone of a passenger to analyse and match the event streams of its own sensors with the event streams of counterpart sensors provided by a reference unit that is installed inside the vehicle. This is achieved through a new learning model architecture using Stacked Convolutional Autoencoders to compress sensor input streams by feature extraction and dimensionality reduction as well as a deep convolutional neural network to match the streams of the user phone and the reference device. The sensor stream compression is offloaded to the smartphone, while the matching is performed in a server. In this paper, we introduce DeepMatch2. It is an amended version of DeepMatch that reduces the amount of data to be transferred from the user and reference devices to the server by the factor of four. Further, DeepMatch2 improves the already good accuracy of DeepMatch from 97.81% to 98.51%. Moreover, we propose a travel inference algorithm, based on DeepMatch2, to detect the duration of whole passenger trips in public transport vehicles with a high degree of precision. This is needed to create intelligent and highly reliable auto-ticketing systems. Thanks to the high accuracy of 98.51% by DeepMatch2, the inferences can be carried out with a negligible error rate.

1. Introduction

In recent years, the rapid development of mobile technologies, IoT and cellular network infrastructures has led to new unprecedented opportunities for making public transportation a very environment-friendly mode of travelling more attractive. The fact that more than 3.8 billion people in the world own smartphones [1], provides a very worthwhile research and development area already utilized by public transportation providers in many areas of the world, *e.g.*, in Northern Europe. For instance, smartphone applications, that provide passengers the option to buy tickets and offer them other context-aware services such as path-finding and travel planning, are quite common nowadays.

However, the next generation of context-aware service within public transportation will require data sources providing an extremely high degree of precision and sophistication. In particular, one may consider the *mobile context*, *i.e.*, all kinds of spatiotemporal properties of the participating passengers and vehicles [2]. For example, if we know whether a person is inside a vehicle or not at a certain time and place, services such as dynamic vehicle-route planners based on passenger load and route optimization can be realized. We can detect this kind of mobile context by precise *in-vehicle detection* systems. These systems can also make the ticketing of passengers considerably simpler. In today's smartphone applications, the passengers have to remember buying tickets before starting a ride. Further, they often need in-depth knowledge about the ticketing system to buy the correct ticket for the planned trip. In contrast, using a highly accurate in-vehicle presence detection solution, a so-called Be-In/Be-Out (BIBO) system [3], tickets can be issued automatically to the passengers based on the exact duration of their journey. This way, the passengers can conveniently enter and leave public transport

* Corresponding author at: Norwegian University of Science and Technology (NTNU), Trondheim, Norway.
E-mail addresses: magnukop@ntnu.no (M. Oplenskedal), herrmann@ntnu.no (P. Herrmann), amirhost@ifi.uio.no (A. Taherkordi).

https://doi.org/10.1016/j.is.2021.101927

Please cite this article as: M. Oplenskedal, P. Herrmann and A. Taherkordi, DeepMatch2: A comprehensive deep learning-based approach for in-vehicle presence detection, Information Systems (2021) 101927, https://doi.org/10.1016/j.is.2021.101927.

vehicles without having to deal with planning and purchasing tickets in advance.

In-vehicle presence detection has attracted the attention of the research community and industry. Early approaches like [4,5] utilize communication systems such as Radio Frequency Identification (RFID) or Bluetooth Low Energy (BLE). While travelling, temporary connections are built up between the user's mobile device and certain fixed vehicle equipment to detect the passengers presence inside the vehicle. Other approaches, for instance [6, 7], analyse event streams from smartphone sensors for certain properties. Modern smartphones are equipped with a variety of sensors such as magnetometers, accelerometers, gyroscopes, GPS, and barometers which offer unprecedented opportunities to analyse mobile context information from the user's environment. Finally, machine learning techniques have recently been leveraged to analyse sensor events for detecting mobile contexts, *e.g.*, [8]. We will argue later that the accuracy of works within the above categories is still not good enough to make them suitable for auto-ticketing in practice.

To realize in-vehicle presence detection with a high degree of accuracy, in *our previous work*, we proposed a deep learning-based framework, called DeepMatch [9]. Each vehicle is equipped with a stationary *Reference Device (RefDev)* (*e.g.*, an Android phone). We record the streams of sensor events gauged in both, the RefDev and the smartphones of potential passengers. In a so-called *in-vehicle presence detection process*, the stream generated in a smartphone is then compared with the one from the RefDev to find out if both devices are in the same vehicle. If that is the case, the owner of the smartphone is necessarily a passenger in the vehicle containing the stationary RefDev and can, *e.g.*, be billed for the journey. The in-vehicle presence detection process is realized by *data compression* using Stacked Convolutional Autoencoders as well as a deep neural network *matching* component that matches compressed sensor samples to find out if they were taken from within the same vehicle. The data compression is offloaded to the users' smartphones and reference devices, while the matching process is performed in a server that can be external, *e.g.*, in a cloud, or within the vehicle realizing an Edge computing solution [10]. By training both parts of our model together, *i.e.*, the compression and matching parts, we achieve that the matching process does not need the full smartphone and reference data for its comparison but can rely on the compressed versions.

Since the design and development of DeepMatch, we continuously iterated and improved our deep learning model to improve its *efficiency* and *accuracy*. Moreover, we enhanced the original framework with inference algorithms that allow us to deduce the *period of time that passengers travel* in public transportation with a very high accuracy. DeepMatch lacks this feature which is highly needed, *e.g.*, in automatic ticketing. The result of the improvements is a new version of our deep learning-based framework that we call DeepMatch2. It is introduced in this paper. In contrast to the original framework, DeepMatch2 incorporates the following amendments:

- The efficiency was enhanced by reducing the amount of data necessary for in-presence detection by a factor of four, *i.e.*, from previously 512 float values in DeepMatch to just 128 float values in DeepMatch2.
- Considering accuracy, we gradually amended the original layer structure, and for each change, trained and evaluated the results using the designated performance metrics. In spite of the concomitant reduction of the size of the input parameters, we further managed to increase the accuracy of DeepMatch2 to 98.51% in comparison to the accuracy value of 97.81% in DeepMatch.
- In [9], we provided only a short sketch about how one can use the results of DeepMatch to detect whole trips of passengers in public transport vehicles with a high degree of precision. In this paper, we go much deeper into this topic and discuss travelling user inference systems that are based on DeepMatch2 and can infer if and for which period of time a passenger makes a trip in a public transportation vehicle with a very low error rate.

The rest of this paper is organized as follows. In Section 2, we discuss existing solutions followed by the presentation of the original method DeepMatch in Section 3. In Section 4, we elaborate on the improvements made in DeepMatch2. Thereafter, we report the experimental evaluation results for the variants of our deep learning model and some baseline methods in Section 5. The travelling user inference algorithms that allow us to detect whole passenger trips, are introduced in Section 6. Finally, we conclude our paper in Section 7 with a discussion on the results gained so far and a look at our future plan.

2. Related work

In-vehicle presence detection solutions can be classified into three different categories. The first category is focused purely on utilizing communication technologies, while the second one is based on analysing mobile sensor events to detect in-vehicle presence. The third category consists of some recent works that leverage deep learning to analyse mobile contexts. In the following, we discuss each category in detail.

2.1. Communication technology-based solutions

Early in-vehicle presence detection systems were implemented using Radio Frequency Identification (RFID) with active tags carried by the passengers, and a single communication unit in the centre of a vehicle. To track the RFID devices, contactless, mid-range radio-based identification and communication protocols were used. One of the first solutions was EasyRide [4], developed by the Swiss Railways Association. Allfa [11] is another RFID-based system, tested in busses, trams and trains in Dresden, Germany, for half a year. In total, the system covered about 120,000 trips carried out by 2000 users. Unfortunately, testing these systems proved that they were too unreliable to be used for in-vehicle presence detection in practice. The main reason for that is the weak transmitter strengths of the active RFID-tags, which makes it difficult to detect them reliably in all areas of the vehicle. As discussed in [4], this affords not only one but a vast number of readers in the vehicle, at least one at each door. But even that does not seem to be sufficient to make the passenger assignment sufficiently predictable. For instance, Allfa has an accuracy rate of just 68% making it unsuitable for practical use.

Other approaches focus on Bluetooth Low Energy (BLE)-based automated in-vehicle detection. Compared with active RFID approaches with battery-powered tags, BLE-based BIBO systems can utilize smartphones with additional monitoring options, the possibility to measure signal strengths for proximity determination, larger distribution channels, etc. One of the early works on Bluetooth-based public transport ticketing system was carried out by the authors of [12]. Their system was in charge of collecting only the source and destination of each passenger journey. The first BLE-based solution is proposed in [3], where the authors are cautiously optimistic that BLE might work for BIBO systems. Nevertheless, the chassis of a vehicle does not limit the accessibility of a BLE transmitter which makes it possible that somebody close to it, *e.g.*, a person in another vehicle, is wrongly detected. On the other hand, objects in a vehicle may inhibit a BLE connection such that devices in the vehicle may not be detected.

This is confirmed by the authors of [13] who found out that BLE is not well suited for indoor localization. As reasons preventing connections, they name the position of a device as well as human body obstacles like the hand carrying the device. The authors of [14] suggest a ticketing system adding a custom profile on top of the BLE specification to fulfil the payment procedure. In SEAT [5], a BLE-enabled smartphone communicates with devices installed in the vehicles to track the journey for automatic pricing. The main focus of the authors, however, is on security, performance, and battery friendliness but not on the accuracy of the in-vehicle presence detection. To conclude, BLE-based solutions are also suffering from low accuracy values making them less suited for in-vehicle presence detection scenarios.

2.2. Mobile sensor data analytics-based solutions

Works in this category analyse the data of the sensors in user smartphones to detect mobile contexts in transportation. The authors of [15] focus on context detection using only the smartphone barometer as it is independent of the phone's position and orientation. They demonstrate that the barometer can be applied to detect user activities of IDLE, WALKING, and VEHICLE at low-power. Likewise, in [16], user activities are classified using the barometer sensor on smartphones. This approach leverages Bayesian networks, decision trees, and RNN as inference models to predict user action, *e.g.*, riding or leaving a cable-car. The authors of [17] demonstrate how the pressure data collected from a smartphone barometer can be utilized to accurately track driving patterns based on the pressure data collected from the smart phone's barometer. By correlating pressure time-series data against topographic elevation data and road maps for a given region, a centralized server can estimate the possible routes through which users have travelled. Another barometer-based mobile system is HybridBaro [7] that features a hybrid algorithm to adaptively utilize GPS data to increase the detection accuracy in flat areas. RoadSphygmo [18] uses the barometer in smartphones to detect traffic congestion. RideSense [6] is aimed to match a passenger's sensor trace against the traces of busses to determine the riding and leaving times. The authors of [19] present a vertical location system for vehicles in metropolises. In particular, they utilize the barometers and gravity sensors of smartphones to remedy the deficiency of vertical localization such as GPS. To achieve that, several novel algorithms are used (*e.g.*, height and angle detection, relative height measurement, and tracking) to build a highly accurate detection system.

While better than RFID and BLE, the accuracy promised by the approaches mentioned above is still not good enough to fulfil the demands of transportation systems. For example, the accuracy of RideSense [6] collected from five bus lines over more than 20 h, is between 84 to 98%. As pointed out in [9], only the uppermost value of 98% would be sufficient for using this technology in practice.

2.3. Mobile sensor events and deep learning

Recently, deep learning has been leveraged to analyse sensor events for detecting mobile contexts. In [20], the authors report on the accuracy of models such as RNN, CNN, various Hybrid models, Restricted Boltzman Machines, and Autoencoders with respect to their ability to classify human activities from body-worn sensors. They conclude that, compared to traditional pattern recognition methods, deep learning reduces the dependency on human-crafted feature extraction and achieves better performance by automatically learning high-level representations of the sensor events. The authors also state that, from a technical viewpoint, there is no model outperforming all the others in general. Thus, they recommend to choose the models based on the requirements of specific scenarios. *DeepSense* [8] uses CNN and RNN to provide an estimation and classification framework for car tracking with motion sensors and human activity recognition. In [21], *DeepSleepNet*, a deep learning framework for automatic sleep stage scoring based on electroencephalogram data, is proposed. The authors show that the model automatically learns features for different datasets without utilizing any hand-engineered features. The model achieves an accuracy that is similar to the state-of-the-art methods using hand-engineering. Some works in this category focus on detecting the transportation mode using machine learning techniques and sensors data on smartphones such as [22,23]. From the ML-based stream matching perspective, *StreamLearner* [24] is a distributed Complex Event Processing (CEP) system proposed for scalable and low-latency event detection on streaming data that uses neural networks. It is mainly designed for systems with multiple event sources causing diverse patterns in the event streams. As a case study, the authors discuss anomaly detection (*i.e.*, finding abnormal sequences of sensor events) in smart factories.

The important finding of most works in this category is that deep learning can outperform hand-crafted feature extraction methods when applied to mobile sensor event streams. This can be used to deduce valuable information about the mobile context. We aim to exploit this power of deep learning in DeepMatch2 to build a model capable of highly accurate in-vehicle presence prediction solely based on sensor event streams. On the other side, the limited number of works, yet carried out on machine learning-based sensor stream matching, focus mainly on the quality aspects of stream matching, *e.g.*, to provide high throughput. In contrast to this paper, the efficiency and accuracy of these approaches are only superficially discussed in most cases.

3. DeepMatch

As we will discuss later, our deep learning method DeepMatch2 enhances its predecessor DeepMatch in both, in-vehicle presence detection accuracy and in the amount of data transfer needed. Nevertheless, the basic structures of DeepMatch and DeepMatch2 are very similar. Therefore, we decided to introduce the fundamentals for the architecture of both DeepMatch and DeepMatch2 in this section. Thereafter, we discuss the changes leading to DeepMatch2 in Section 4.

In the following, we start with a general overview of the model architecture of our deep learning model, followed by a discussion of the hardware and software settings, on which our approach has been built. Thereafter, we describe how DeepMatch conducts the analysis of the mobile data sensed, followed by the design considerations and architecture of our learning model, and a discussion on how the learning model is trained. Finally, we present the design rationale and experimental settings behind the DeepMatch deep learning model.

3.1. Overview

Fig. 1 depicts the equipment needed in a bus[1] to realize our approach. The bus is equipped with a Bluetooth Low Energy (BLE) transmitter as well as a so-called *Reference Device (RefDev)*. The RefDev can be a smartphone mounted onto the bus or any other kind of hardware providing the same type of sensors, that can be found in a modern smartphone. These sensors include but are not limited to *accelerometers*, *magnetometers*, *gyroscopes*, *barometers*,

[1] The realization of DeepMatch in other types of public transport like subways, trams, or trains is similar.

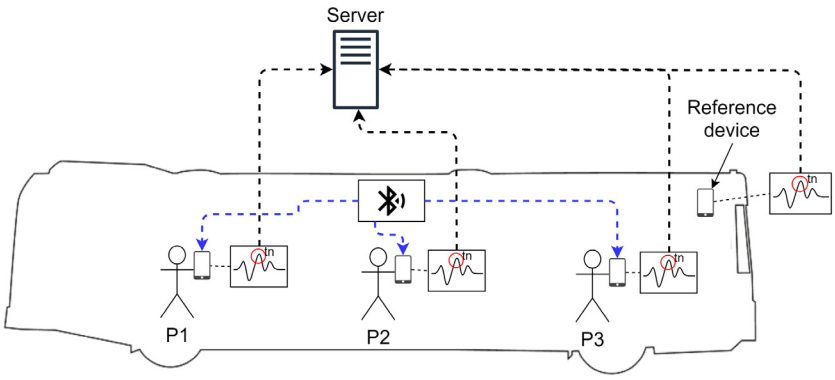

Fig. 1. DeepMatch bus scenario.

and *GPS receivers*. The passengers travelling with the bus carry smartphones in which a special application is installed realizing parts of the DeepMatch deep learning model.

The BLE-transmitter continuously transmits a special ID that is unique to the bus it is installed in. Due to the low signal strength, this signal can be only detected by devices that are either inside the bus or nearby. When a passenger's phone picks up the BLE-transmitted signal for the first time, its operating system starts the DeepMatch application. From that moment, the sensors of the phone sample values that are forwarded to the deep learning model of DeepMatch running in the smartphone application. DeepMatch extracts relevant features from the sensed events and compresses them through dimensionality reduction. Finally, the compressed data are timestamped and tagged with the IDs of the BLE-transmitters, the phone is currently receiving.

In a similar way, the RevDef is used to continuously stream events from its own sensors and compresses them through DeepMatch. The compressed data is also timestamped and tagged but, in contrast to the user phones, only the tag of the BLE transmitter installed in the same vehicle is used.

Both, the RevDef and the user phones send the compressed sensor data to a server. Following the wishes of the public transport operators, we may realize DeepMatch using different hardware configurations. For instance, the server functionality can be realized using a cloud provider. Alternatively, following the principle of fog computing [10], it can be a unit locally installed in the vehicle, *e.g.*, together with the RevDef.

The server matches the data of the RevDef and the user phones, that carry the same BLE-transmitter IDs and timestamps, against each other by a special module of DeepMatch. If the module reports a match, we assume that both datasets were sensed within the same vehicle. Since the RevDef can be unambiguously allocated to a particular bus, we can then assume that the smartphone and its carrier are in the same bus.

3.2. Hardware requirements and system settings

As mentioned above, all vehicles using DeepMatch to provide automated in-vehicle presence prediction, require both, a BLE-transmitter and a Reference Device (RevDef). In contrast to the communication technology-based approaches discussed in Section 2, the BLE-transmitter is not directly used for in-vehicle detection. Instead, we apply it to perform a coarse-grained guess in which vehicle a passenger might be inside. In this way, the server only needs to match the user data with those from RefDevs, that are related to the sensed BLE ID received by the user phone, and not with the data of all RefDevs in the transport network. An additional advantage of this approach is that we can reduce the time DeepMatch is required to run on a user phone. Both Android and iOS provide the ability to awaken applications in smartphones when detecting a BLE-signal with a pre-defined ID. This provides us with the ability to run the application only if the user is either very close to a vehicle, or inside it. Thus, both computation overhead and battery consumption is at a minimum.

The authors of [25] show that BLE offers a good reliability also in noisy in-house environments like those we might come across in public transport vehicles. In their tests, at least 99.45% of all packets were transmitted within the expected delay bounds. Based on these numbers we expect that the phone of a passenger receives a fair number of the packets broadcasted by the BLE-transmitter within in the first seconds after entering the vehicle. Therefore, DeepMatch will almost certainly be started timely.

As we discuss to greater detail in Section 5.3, we found out through experiments, that using only the barometric sensor provides by far the best matching accuracy. Using DeepMatch alone with the barometric sensor provides an accuracy of 97.81% while no other combination of sensor data exceeds 80.82%. Moreover, performance tests show that registering barometer events with a frequency of 10 Hz incurs a very low battery consumption. The battery drain on the phones we tried in our tests is between 15 and 25 mAh while continuously registering events from the barometer. This equals a drain of between 0.6% and 0.8% of the total battery capacity per hour. That is described more closely in Section 5.8.

In the case that a vehicle enters a *dead spot*, *i.e.*, an area with no cellular network coverage, we temporarily store the compressed data locally until connectivity is regained and the data can be transmitted to the server for a delayed matching.

3.3. Mobile data analysis

The deep learning model performing the in-vehicle presence prediction, *i.e.*, the matching of sensor events, is trained on real sensor events that were collected from Android-based smartphones in the public transport systems of the Norwegian cities Oslo and Trondheim. In the following, we sketch the process of collecting the data and converting it to training and evaluation sets.

The datasets used to train the deep learning model were built from sensor events gathered by the means of an Android application, called *Datacollector*, that we developed for this purpose. *Datacollector* registers events from all sensors available in the phone, timestamps them, and stores them locally as *data points*. Further, we can use the application to upload the data points to our *Data Analysis* centre. There, the data points can be processed further into training and testing samples that are used to train and evaluate our deep learning network.

To allow the parallel collection of sensor data by several phones, multiple devices running the *Datacollector* can be connected using a simple client–server communication protocol. This

allows us to synchronize the clocks of the various phones. Further, we can tag all events registered by the connected devices with a unique *trip ID*. When a data collection session is initiated, the *trip ID* is generated by the initiating device and propagated to all devices taking part in the collection.

Android provides developers with a sensor framework where the sampling rate of each available sensor can be separately defined. The effective sampling rate, however, comes usually with a standard deviation of one to two milliseconds. In addition, even though each sensor is collecting events at the provided sampling rate, there is often a shift of the exact sensing times (*e.g.*, while both, the barometer and accelerometer sensors collect data every 20 ms, the exact points of time, the samplings take place, deviate from each other by a few milliseconds). On the other hand, two data streams can be matched best when the sensors in both devices carry out their sampling steps at exactly the same points of time t. To dissolve this contradiction between precise sampling times and the aforementioned shortcomings of the Android framework, we implemented an interpolation technique in our *Data Analysis* tool which is described in detail in [9].

The deep learning model in DeepMatch was created to support travel times of varying lengths, and to reduce the amount of data to be transmitted from the devices in the public transport vehicles to the server, as well as the number of operations required by the server. To fulfil these requirements, we train our model to perform predictions on smaller *segments* of the collected events. In Section 5.4, we report the results from training the model on segment sizes of five, ten and 15 s. As elaborated in Section 5.4, our tests showed that the model being trained on segments consisting of ten seconds of barometer sensor events provides the best results.

3.4. Design and architecture of the learning model

The goal of the deep learning model of DeepMatch is to predict the in-vehicle presence of a device by matching its sensor events against the sensor events generated by the on-board Reference Device (RefDev). The deep learning model consists of three modules, an *encoder*, a *decoder*, and a *matching module*. All three modules are trained jointly as one large neural network. In practice, however, the in-vehicle presence predictions can be achieved by utilizing only the encoder and the matching module. Therefore, we use the full model that also includes the decoder, *only* during the training phase. When our deep learning model is sufficiently trained, we extract the encoder and matching modules from it. Copies of the encoder are then used in the RefDev and the passenger devices, while the matching module is executed in the server.

The distribution of the modules is depicted in Fig. 2.Here, the green networks in the passenger and reference devices illustrate that the encoders are residing on these devices. In contrast, the blue network illustrates the matching module that runs on the server.

Fig. 3 provides a sketch of the neural network used in DeepMatch. The coloured boxes represent layers of the neural network that can be trained while the grey boxes refer to model layers that do not contain trainable parameters. The green boxes describe trainable layers of the encoder, the orange ones trainable layers of the decoder, and the blue boxes trainable layers of the matching module. The hyperparameters of each layer in the neural network are represented as numbers next to the description of the layer. More specifically, in the boxes representing the conv layers, the size of each filter in the layer is represented as *width* x *height*, whilst the number on the right side of a box refers to the number of filters used. For instance, in the uppermost layers of the Stacked Convolutional Autoencoders in Fig. 3, there are

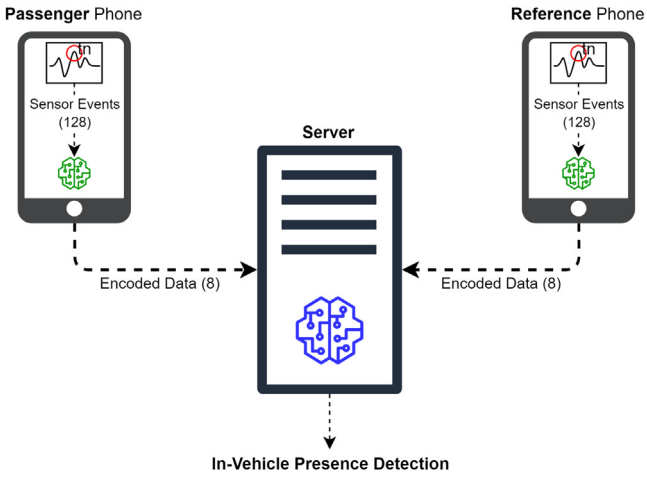

Fig. 2. Overview of the DeepMatch distributed framework. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

128 filters of size 8 × 1. Furthermore, for the dense layers, the number of neurons in the layer is described by the number at the end of the layer description, *e.g.* in the first dense layer in the matching module the number of neurons is 256. The properties and utilization of the various layers are discussed below.

Since the matching module has to compare the sensor data from two devices, the RefDev and a passenger phone, we show two copies of the encoder and decoder in Fig. 3. This type of neural network topology is generally known as a *Siamese Architecture* and has been successfully used to solve other matching problems such as face recognition [26], gait recognition for person identification [27], and signature verification [28].

Configuring the deep learning model as a Siamese architecture provides the model with the ability to receive two simultaneous inputs, *e.g.* sensor data segments X_a and X_b. Since the two Convolutional Autoencoders share the same weights, the mapping performed by the encoders on the two inputs are identical. In consequence, two matching input segments, *i.e.*, samples of *Class 1*, result in latent representations e_a and e_b that are also matching. The same is true for not matching samples belonging to *Class 0*. Here, the two latent representations are dissimilar as well.

3.5. Encoder and decoder

We use an architecture called *autoencoder* [29]. This kind of neural network consists of two parts, that directly reflect our encoder and decoder. The encoder transforms the input of the autoencoder into an internal representation often referred to as the *latent representation* in latent space, whilst the decoder aims to reconstruct the original input from its latent representation.

The loss, *i.e.*, the error of encoding and later decoding data, is calculated by comparing the input with the output of the autoencoder. The goal of training the autoencoder with a large set of samples is to keep this error minimal. As depicted in Fig. 3, autoencoders usually consist of several encoder and decoder layers through which the input data is sequentially forwarded. The quality of an autoencoder often depends on the arrangement of these layers. Usually, the length of the data forwarded between two layers is restricted such that the neural network needs to learn to prioritize the most important characteristics of its input, *i.e.*, the encoder must learn which features of its input are most relevant to perform a correct matching.

This *feature extraction*, can be seen as a compression algorithm. Then, the latent representation corresponds to the compressed

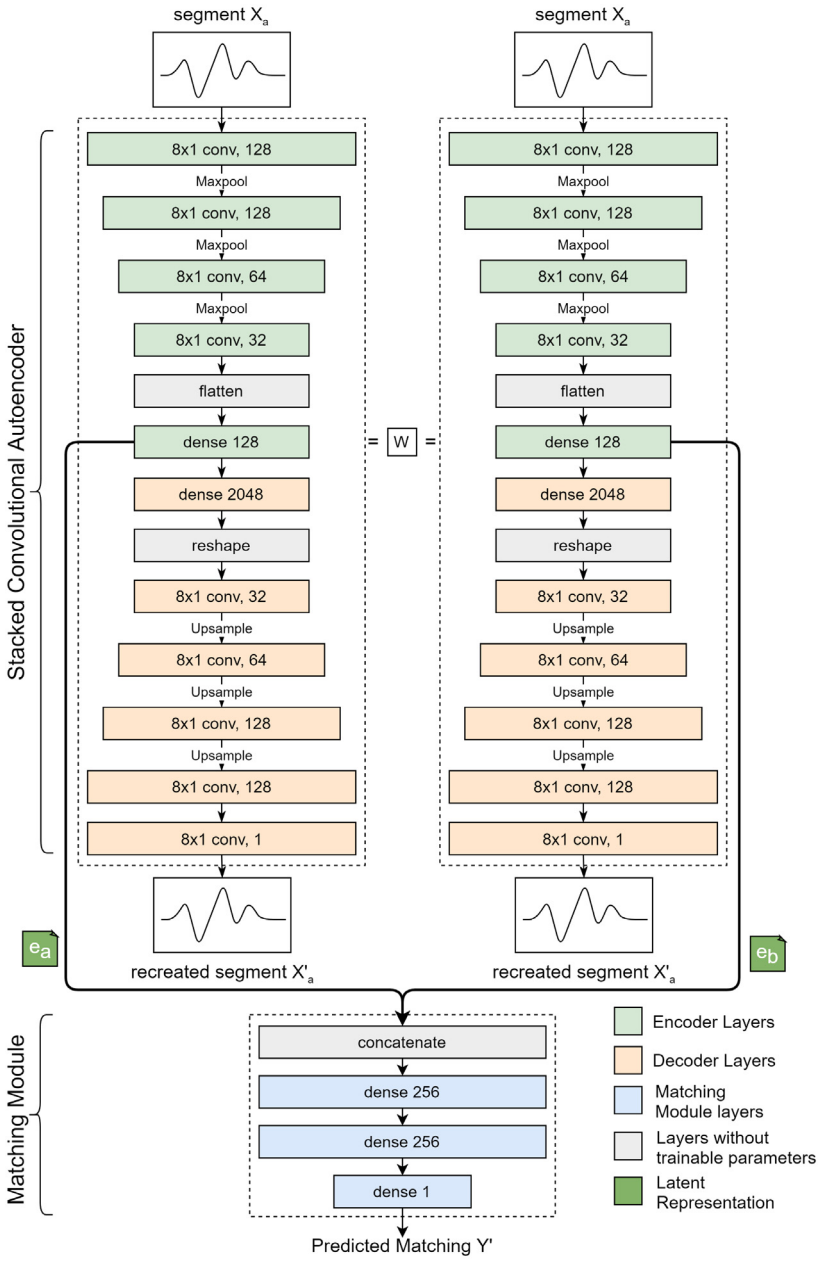

Fig. 3. Original architecture of DeepMatch. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

data while the decoder is the corresponding decompression algorithm. In addition, the encoder provides *input noise reduction* since the compression forces it to learn the most important features of its input and to discard irrelevant features.

In DeepMatch, the ability to compress data is utilized to reduce the amount of transmitted information from the user phones and the RefDev to the external server. To let the autoencoder learn to prioritize those parts of the input data, that are most relevant, we train it together with the matching module in a single neural network. In this way, it learns to discard only those parts of the sensor events that are less important for the in-vehicle presence detection, but to keep all relevant data in the latent representation. This allows us to run the matching module based on the latent representation of the sensor such that the server does not need to decode them first.

As shown in Fig. 3, we created our autoencoder using alternating *convolutional* (*conv*) and *maxpooling* layers in the encoder part. Here, the conv layers are responsible for the feature extraction while the maxpool layers reduce the size resp. dimensions of the input. In the decoder, the conv layers alternate with *upsample* layers that are responsible for reverting the maxpool operation in the decoder.

We use the convolutional layers since they are especially suitable to detect and extract time-invariant features in sequences, see [20,30–32]. Of course, this time-invariance is very important for our in-vehicle presence detection problem since we want to find out whether two devices are at the same place, *i.e.*, the same vehicle, independent from temporal influences like those caused by the distance between the phones in the vehicle. The maxpool layers reduce the size of their input data by a factor of two using the *max* operator. The upsample layers make it possible to reverse this process by duplicating each value in its input sequence, *e.g.*, $upsample(x, y, z) = x, x, y, y, z, z$. An autoencoder

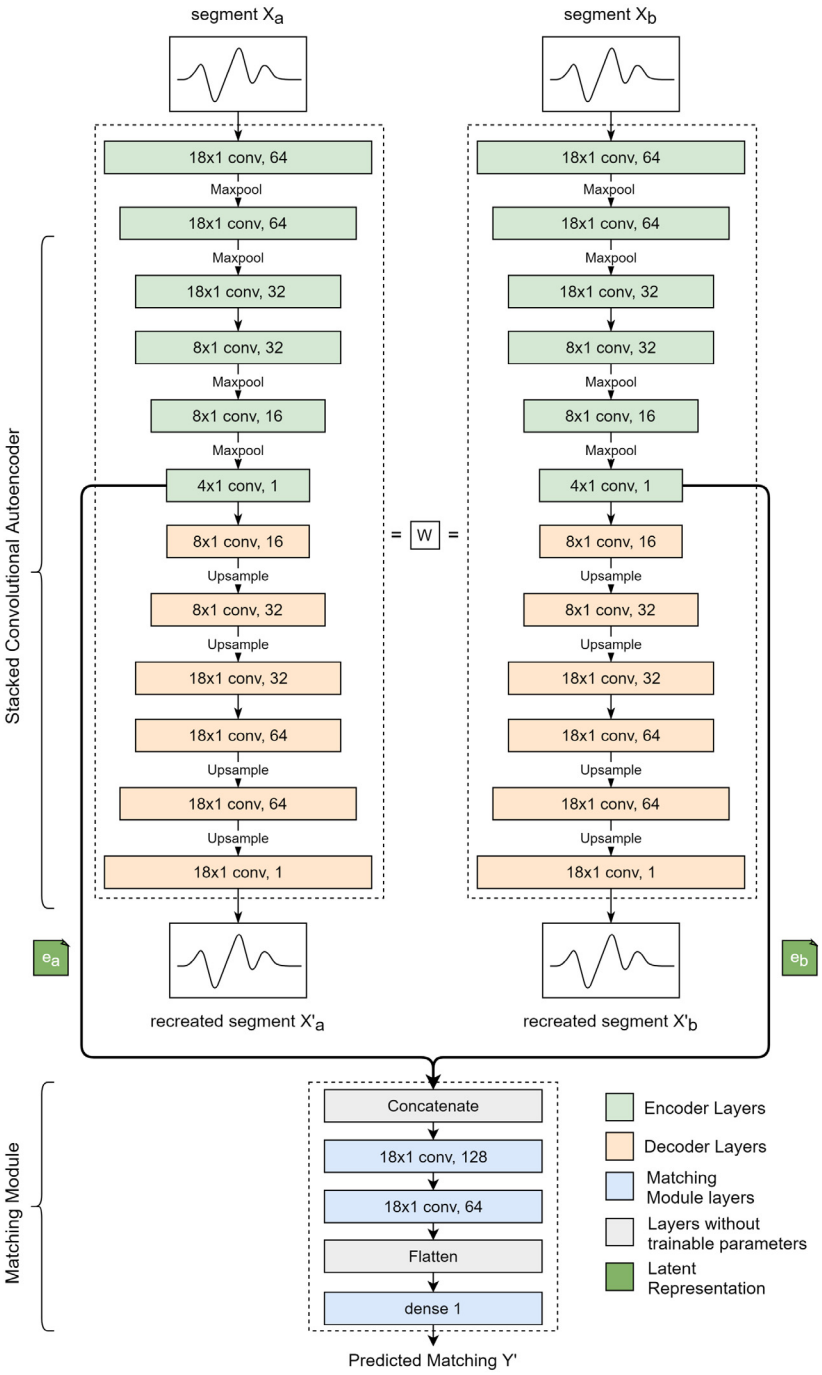

Fig. 4. Architecture of DeepMatch2. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

consisting of conv, maxpool, and sample layers, is called a *Stacked Convolutional AutoEncoder* (CAE) [31].

Each layer of our CAE consists of three or four more elemental machine learning operations. At first, a convolution is carried out followed by a Rectified Linear Unit (ReLU) activation. If the layer has maxpooling or upsampling functionality, this is executed after the ReLU. Finally, in each layer a batch normalization is carried out.

3.6. Matching module

As previously mentioned, the matching predictions to find out if a smartphone is in the same vehicle as a RefDev, are performed by the matching module residing in a server. To match the sensor data previously compressed by the encoders, without having first to decompress them, the matching module needs to be able to compare the latent representations. To achieve this, we trained it to learn an accurate spatiotemporal threshold for separating instances of *Class 1*, *i.e.*, sensor events gathered by two devices in the same vehicle at the same time, from instances of *Class 0*, *i.e.*, a pair of sensor event sequences collected during different trips or at different locations.

The functionality of the matching module is represented by the blue boxes in Fig. 3. It consists of two *Dense* layers using the ReLU activation function, and finally another dense layer using the *Sigmoid* activation function. The Sigmoid function converts any real number into a value between zero and one. It is used in

DeepMatch to describe whether the deep learning model believes that its input pairs belong to Class 1 or Class 0.

3.7. Model training

As discussed above, the three modules of the DeepMatch deep learning model are jointly trained using the configuration shown in Fig. 3. In our Siamese architecture, we observe two separate copies of the CAE that compress the sensor data segments X_a and X_b. Both CAEs share the same set W of trainable parameters causing the model to perform identical mappings for its two inputs. The CAEs produce the latent representations e_a and e_b that we illustrate as dark green squares in the figure. In the training phase, the latent representations are propagated through the layers of the decoders mapping them to the recreated segments X'_a and X'_b. In parallel, e_a and e_b are also sent to the matching module that is depicted by the blue boxes in Fig. 3. Here, the latent representations are being matched and the class prediction Y' is being produced. It assigns the value $Y' = 1$ if the model predicts that e_a and e_b are matches, and $Y' = 0$ if they are not. The values of Y' correspond to the ground truth labels Y of our sample pairs since we assign $Y = 1$ to a pair X_a and X_b if the samples are from *Class 1*. Likewise, for a sample pair of *Class 0*, we use $Y = 0$.

The goal of the training regime is now to train the layers of the matching module such that their computed values Y' are mostly identical to the ground truth labels Y of the training sample pairs. This kind of training utilizing external information about the training samples, *i.e.*, the ground truths, is usually called *Supervised Learning*. Formally, we describe the deviation between the ground truth Y and its prediction using the *Binary Cross Entropy L*:

$$L = -Y \cdot log(Y') + (1 - Y) \cdot log(1 - Y') \quad (1)$$

The goal of the training is to find weights for the layers of the matching module such that the prediction renders values of L close to zero.

In the same step, the layers of the encoder and decoder are trained by reducing the disagreements between the original segment pairs X_a and X_b and the recreated ones X'_a and X'_b. This improves the quality of the sample compression and decompression steps. Since neither the ground truths of the samples nor other external information is used, this training regime is referred to as *Unsupervised Learning*. We quantify the disagreements between the original and recreated segments using the *Mean Squared Error MSE*:

$$MSE = \frac{1}{n}\sum_{t=1}^{n}(X'_a[t] - X_a[t])^2 \quad (2)$$

In formula (2), n is the duration of the sensor event sequence X_a while $X'_a[t]$ is the recreation of the value $X_a[t] \in X_a$ at time t.

The disagreements between original and recreated samples as well as those between the ground truths and the values predicted by the matching module are used to update the weights of the neural network through the machine learning technique *Stochastic Gradient Descent*. In this technique, both the gradients from the mean squared error calculations and the Binary Cross Entropy loss functions are backpropagated to the neurons of the encoders. This enables our encoder to extract both, the most important features of the input segments for a good recreation and the features that are relevant for an accurate matching prediction. In consequence, it is sufficient to use the latent representations e_a and e_b instead of the original sample pairs X_a and X_b to conduct the matchings. This is the reason that, when executing DeepMatch to detect real in-vehicle presence of passengers, we only need to use the encoder of the CAE in passenger smartphones and in the RefDev as well as the matching module in the server, while the functionality of the decoder module is not needed.

3.8. Design rationale and experimental settings behind the DeepMatch model

In our quest to find the best model, we conducted hundreds of experiments on various model design and hyperparameter configurations. Our approach relied on starting with smaller, shallower neural networks, before expanding them by adding layers, filters within the CONV layers, and by increasing the size of these filters. To keep track of the various experiments, every configuration and design of the network was evaluated using the performance metrics described in Section 5.2. During this work, we also experimented with various activation functions for both CONV and dense layers. Moreover, we tried swapping the CONV layers in the Autoencoders with dense layers and exchanging the Matching Module with a function calculating the Euclidean Distance between the latent representations, respectively. Furthermore, we experimented on how we trained the modules of the network, *e.g.* we tried training the autoencoders separately from the matching module. This was done by first training the autoencoders, and thereafter using the autoencoders to create datasets consisting of encoded samples. These encoded samples where then used to train the matching module. In addition to design, architectural, and training experiments, we tested a large number of hyperparameter settings, *e.g.* the number of neurons in each dense layer, the size of batches used during training, the number of epochs for each training session and so on. From the experiments conducted for our previous work in [9], the model architecture shown in Fig. 3, using the hyperparameter settings described in Section 3.4, gave us the pest performance. All experiments were conducted on a desktop PC with an Intel i7 4.00 GHz CPU, 16 GB memory, and a Nvidia GTX 1080 GPU. The models were created, trained and evaluated using Google tensorflow 2.0, version 2.0.0-rc0 [33].

4. DeepMatch2

Since the original publication of DeepMatch in [9], we continuously iterated and improved our deep learning models. That has led to several improvements that we could incorporate into the new version DeepMatch2 of our in-vehicle presence detection system. In particular, we amended the layer structure of the architecture. In the following, we will discuss the improvements in greater detail.

4.1. Design rationale and experimental settings of the DeepMatch2 model

We started the work on our new model basing it on the original architecture of DeepMatch, depicted in Fig. 3. We followed the approach described in Section 3.8 by gradually making incremental changes to the model architecture, and for each change, training and evaluating the results using the performance metrics described in Section 5.2. Moreover, we investigated adapting the numbers of convolutional layers and filters within the CONV layers as well as changing the sizes of the individual filters. Thereafter, we experimented with the ratio between the numbers of CONV and maxpool layers used by the CAEs. In particular, the structure of the matching module was modified. DeepMatch2 uses another method to concatenate the two latent representations, the module receives from the autoencoders. This is described in detail in Section 4.3. Our experiments revealed that the model architecture and the model parameters depicted in Fig. 4 yielded the best results. Also for these experiments, we used the desktop PC and Google tensorflow version described in Section 3.8.

4.2. Dimensionality reduction

During the initial development of our original deep learning model, we were not aware that the best in-vehicle detection accuracy could be achieved using events from the barometer sensor alone. We learned this fact through the empirical studies described in Section 5.3, for which a first version of DeepMatch was needed.

After gaining the insight that just barometer events would be sufficient for inference, we started to refine and restructure parts of the model in order to better accommodate samples containing only this type of data. Our aim hereby was to utilize the limitation to barometer inputs for a reduction of the size of data transmitted between the distributed computational units. Of course, this should happen without worsening the accuracy of the in-vehicle prediction, but rather with an improvement.

Our first major change was to reduce the size of the model inputs, from previously 512 float values in DeepMatch to just 128 float values in DeepMatch2. The model of the previous version was laid out to accept events from multiple smartphone sensors that publish events at fixed rates but possibly with different maximum frequencies and varying reading points in time. In each layer of our model, the maxpool operator reduces the size of its input by the factor of two such that the input length needs to be a multiple of two. However, the output of a layer must also be a multiple of two since it is directly used as the input for the maxpool operator of the next layer. To guarantee this property for a number of subsequent layers, we therefore need the size of the initial encoder input to be an exponent of two. The highest frequency of the sensors tested in DeepMatch was 50 Hz, and the first sample size, we aimed to create, was 10 s worth of sensor data resulting in 500 events. Considering the aforementioned requirement that the input must be an exponent of two, the closest sample size to 500 was 512 events, resulting in an actual sample size length of 10.24 s.

On the other hand, the barometer sensor conducts its sensing with a maximum frequency of 10 Hz such that a sample of 10 s only needs 100 events. To fulfil the demand of using a number of events that is an exponent of two, we decided to increase the sample period to 12.8 s such that a sample processed in DeepMatch2 contains 128 barometer events. That is just a quarter of the original sample size.

4.3. Accuracy improvement

While the accuracy of 97.81% of the original deep learning model is quite good, we aimed to make it even better in DeepMatch2. The most impactful change, we made to improve the accuracy of our deep learning model architecture, was altering the way the matching module concatenates its two inputs. These changes are illustrated in Fig. 5. As shown at the top of the figure, in DeepMatch, the two input samples were simply concatenated along the first axis. That means that two float vectors that both had the length x, were transformed into one vector of the length $2x$. This concatenation, however, usually creates a large spatial distance between the pairs of values that need to be compared by the matching module.

To reduce this distance, we altered the concatenation by transforming two input vectors of length x into a matrix with the size $(x, 2)$ which is shown at the bottom of Fig. 5. This adaptation to the input of the matching module allowed us to replace some of the dense layers by convolutional layers. As discussed in Section 3.5, convolutional layers are well suited to handle time-invariant features in different sequences such that they promise to be a better fit than the dense layers.

As a result, we achieved the layer structure that is illustrated by the blue boxes in Fig. 4. Its uppermost layer is a concatenate layer that concatenates the inputs from the two encoders by transferring them into a matrix of the size $(x, 2)$. This matrix then forms the input of the first of two convolutional layers. Thereafter, a *Flatten* layer flattens its n-dimensional input into a one-dimensional vector. Finally, a dense layer uses the Sigmoid function explained in Section 3.6. This amended layout led to an improved accuracy of 98.51% which we discuss in greater detail in Section 5.6.

Table 1
Smartphones used by volunteers to collect data.

Brand	Model
LG	Nexus 5X
Huawei	Nexus P6
Samsung	Galaxy S8
Sony	Z3 Compact
Google	Pixel XL
Google	Pixel 3a

5. Evaluating the deep learning models

Our evaluation effort includes three main steps. *First*, we explain how we created our training data, and present the performance metrics used to evaluate different models. Thereafter, we report on the experiments carried out to find which sensor data are best for in-vehicle presence detection and the best segment size (cf. Sections 5.1—5.4). *Second*, we evaluate the prediction performance of both DeepMatch and DeepMatch2. To this end, we compare several versions of the original deep learning model DeepMatch with varying sample lengths. This is followed by comparison of DeepMatch with two well-known baseline methods and, of course, with our updated version DeepMatch2 (cf. Sections 5.5 and 5.6). *Third*, we investigate and discuss the execution time overhead of the matching module on the server, and the battery consumption as well as the CPU run-time overhead for the smartphones of the passengers (cf. Sections 5.7—5.9). Note that the evaluation results, in this step, *apply to both versions of our learning models*, namely, DeepMatch and DeepMatch2.

5.1. Data collection and dataset creation

The data used to develop and evaluate our deep learning model was collected in various public transportation vehicles (*i.e.*, trains, subways, busses and trams) in the Norwegian cities of Oslo and Trondheim by volunteers. They used different Android phones that are listed in Table 1. All these phones were provided with the *Datacollector* application introduced in Section 3.3 such that their clocks could be synchronized and common *trip IDs* assigned.

In total, our volunteers collected 212,520 s of unique sensor data events from the magnetometer, accelerometer, gyroscope and barometer sensors. Segments of sensor events from two different sources were paired in each dataset sample to model that DeepMatch and DeepMatch2 match pairs of sensor data segments from a user's phone and a RefDev. Moreover, the pairs of sensor event segments were classified as illustrated in Fig. 6. Thus, sensor event segments with identical trip IDs and beacon identifiers, *i.e.*, segments computed by different devices in the same vehicle at the same time, were labelled as *positive samples*, *i.e.*, *Class 1*. In contrast, event segment pairs with differing trip IDs or beacon identifiers were fetched at different times or in different vehicles. Therefore, they are labelled as *negative samples*, *i.e.*, *Class 0*.

Following common practice in machine learning, we shuffled the samples of our dataset. Thereafter, we normalized the data using minmax and, finally, we split our samples into training

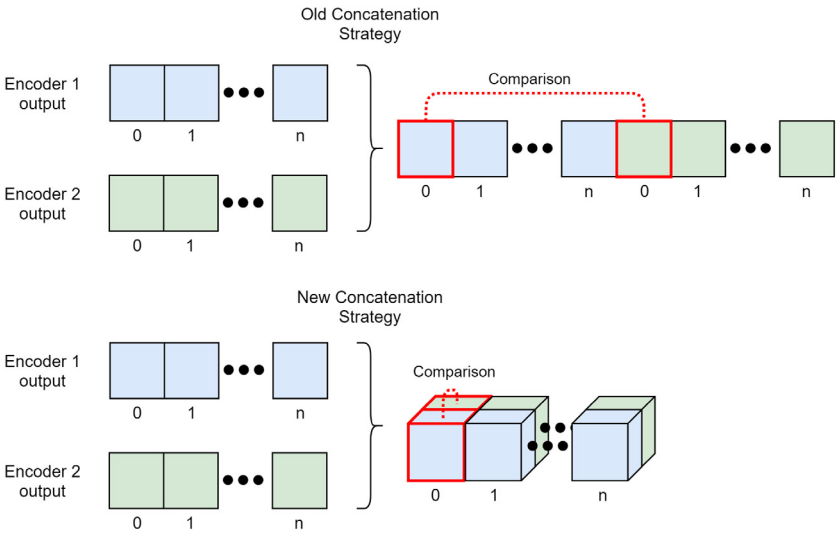

Fig. 5. Old and new matching module input concatenation strategy.

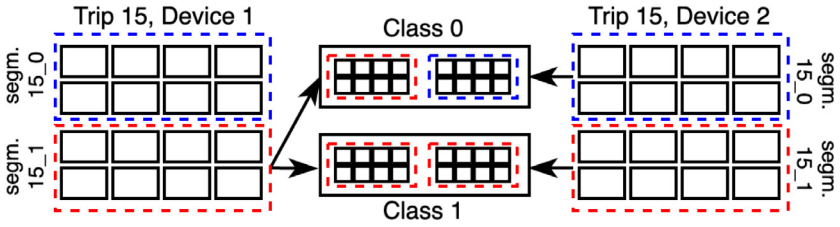

Fig. 6. Matching samples created from trip segments.

and testing ones. The training set consisted of around 70% of the overall data. The other 30% were used to create testing sets for the purpose of model evaluation. By completely separating the training from the testing samples, we avoided to use the same sensor events in both phases. Further, we made sure that the distribution of *Class 1* and *Class 0*, were about 50/50 in both datasets.

5.2. Metrics to evaluate learning models

To evaluate the different versions of our deep learning model with each other and with other methods introduced later, we use the four metrics *precision*, *recall*, *accuracy*, and *F1-score* that all are popular means to evaluate machine learning models. In order to define these metrics, we use four binary classifiers that describe if a sample is positive or negative in reality, and if it is correctly classified:

- *True Positive(TP)*: A correctly classified positive sample,
- *True Negative(TN)*: A correctly classified negative sample,
- *False Positive(FP)*: A negative sample that is falsely classified as a positive one,
- *False Negative(FN)*: A positive sample that is wrongly classified as a negative one.

With the help of these classifiers, we can now introduce the four metrics used to evaluate the models:

- *Precision (PR)*: The ratio of correct positive predictions to the total number of predicted positive samples, *i.e.*, out of all samples classified as positive, how many belong to *Class 1*:

$$PR = \frac{TP}{TP + FP} \tag{3}$$

- *Recall (RE)*: The ratio of correct positive predictions to the total number of positive samples, *i.e.*, out of all samples in the dataset that are indeed positive, how many were correctly classified as such by the model:

$$RE = \frac{TP}{TP + FN} \tag{4}$$

- *Accuracy (ACC)*: This metric states how good the model classifies samples from all classes, *i.e.*, it describes how many of all predictions are correct:

$$ACC = \frac{TP + TN}{TP + FP + TN + FN} \tag{5}$$

  The accuracy results are usually only reliable if the number of members from *Class 0* and *Class 1* are about equal. That is the reason for using equal representation of *Class 1* and *Class 0* in our training and testing sets (see Section 5.1).
- *F1-score (F1)*: The harmonic mean between precision and recall.

$$F1 = 2 \cdot \frac{PR \cdot RE}{PR + RE} \tag{6}$$

In the various experiments introduced in the next subsections, we compare the different methods using these four metrics.

5.3. Sensor modality experiments

In a first set of experiments, we wanted to find out which combinations of smartphone sensors are most suited to be used in our deep learning model. To be able to compare various sensor mixes, we created the following seven sensor modality combinations: Accelerometer (A), Magnetometer (M), Barometer (B), Barometer and Accelerometer (BA), Barometer and Magnetometer (BM), Accelerometer, Magnetometer, and Gyroscope (AMG),

Table 2
Performance comparison between various sensor combinations.

Model	PR	RE	ACC	F1
DeepMatch 10 A	0.5065	0.9531	0.5122	0.6615
DeepMatch 10 M	0.5064	0.9280	0.5118	0.6553
DeepMatch 10 B	**0.9751**	**0.9812**	**0.9781**	**0.9781**
DeepMatch 10 BA	0.7332	0.9697	0.8082	0.8350
DeepMatch 10 BM	0.7081	0.9708	0.7853	0.8189
DeepMatch 10 AMG	0.5011	0.9646	0.5020	0.6595
DeepMatch 10 AMGB	0.7079	**0.9892**	0.7905	0.8253

Table 3
Performance comparison of the barometer-based DeepMatch with various sample lengths as well as with baseline methods and DeepMatch2.

Model	PR	RE	ACC	F1
DeepMatch 5	0.9408	0.9765	0.9574	0.9583
DeepMatch 10	0.9751	0.9812	0.9781	0.9781
DeepMatch 15	0.9348	0.9816	0.9566	0.9576
NORM_CORR	0.9174	0.9595	0.9393	0.9380
DTW	**0.9810**	0.7350	0.8136	0.8404
DeepMatch2	0.9769	**0.9935**	**0.9851**	**0.9851**

and Accelerometer, Magnetometer, Gyroscope, and Barometer (AMGB).

We trained and tested each of these modality combinations with our original deep learning method DeepMatch using sample lengths of 10 s. The test results of these experiments are depicted in Table 2. From the table, we can see that the models trained on datasets of type DeepMatch 10 B containing sensor events only from the barometer, have a significantly higher precision, accuracy, and F1-scores than the other models. The recall values are much closer for all models but also here DeepMatch 10 B finishes in the top two, albeit a little behind DeepMatch 10 AMGB.

We believe, the reason for the good result of just using the barometer is that this sensor particularly suited to capture the movements of the vehicle rather than those of the smartphone user. For instance, it is position independent, *i.e.*, the precision of the barometer events is not affected by the location of the sensor. This is especially important for the use in underground transportation, *e.g.*, in subways, where the GPS performs poorly. Further, the barometer is highly resistant to vibrations as well as movements of the smartphone user. This in stark contrast to the accelerometer and gyroscope which are more strongly affected by the movements of the user than by those of the vehicle. Unlike the barometer, the magnetometer is highly sensitive to magnetic objects in the environment like the power unit of the vehicle. All these traits make the barometer perfectly suited to capture just the movements of the vehicle and to ignore the movements and immediate surroundings of the carrier of the smart device.

The absence of the mentioned weaknesses of the other sensors makes it easier for our deep learning model to learn how to separate instances of *Class 1* and *Class 0* only using the barometer events. This is confirmed by the overall good values for DeepMatch 10 B in comparison to the other sensor combinations. The fact, that DeepMatch 10 AMGB has a little better recall value, results probably from a tendency to classify samples as positive even if they are negative in reality. This is not punished by the recall value but by the precision and accuracy values that are meager for DeepMatch 10 AMGB.

Altogether, our sensor modality experiments led to the decision to consider only the barometer sensor data for our deep learning method. This is in accordance with most of the other works introduced in Section 2.2 which also claim that the barometer data are best for classifying in-vehicle detection, *e.g.*, [15, 16].

5.4. Segment size experiments

After deciding to base the deep learning model just on the barometer input, we wanted to find the sample length for which DeepMatch renders the best results. Therefore, we trained and tested the model with different sample lengths of five, ten, and 15 s. The results are shown in the first three lines of Table 3.

We see that variant DeepMatch 10 with its ten seconds long samples outperforms DeepMatch 5, the model trained on matching samples of five seconds. The cause is most likely the greater number of sensor events contained in a DeepMatch 10 sample. This provides a better foundation for training the neural network in DeepMatch 10 than in DeepMatch 5.

Following this logic, however, we should expect that DeepMatch 15, the model trained on 15 s long matching data, outperforms DeepMatch 10 since it has an even higher number of sensor events available in a sample. Yet, this is not the case for the precision, accuracy, and F1-score performance metrics, while the recall values are basically even. Like with DeepMatch AMGB in the tests discussed in Section 5.3, it seems that DeepMatch 15 is biased towards classifying samples as positive since it produced good recall but bad precision values. The most likely reason for this surprising effect is that we have fewer 15 s long samples available than shorter ones in our training set. In consequence, there might be simply too few sample pairs available to train the neural network well.

Due to the ongoing Covid-19 pandemic, our volunteers were not able to collect more data from public transport vehicles. When the pandemic is over, however, we intend to collect a larger number of longer samples expecting that DeepMatch 15 will outperform DeepMatch 10 when the new samples can also be used for training and testing.

5.5. Comparing DeepMatch with two baseline methods

In order to get a better comparison of our deep learning method with other possible approaches, we also employed two well-known baseline methods that seem to be suited to perform sensor event matching for in-vehicle presence prediction. One of the selected baseline methods is *Normalized Correlation (NORM_CORR)*. It calculates the correlation between two vectors by comparing the values in the same position. The other baseline method is *Dynamic Time Warping (DTW)*. In DTW, all values in the two vectors are compared by warping the temporal dimension until the best correlation for any data point is found. In consequence, DTW does not inherently describe the correlation between two vectors but the distance between them, and a large distance equals a small correlation. Thus, to use DTW as a measure of correlation on par with NORM_CORR, we had to inverse the results produced by it.

In both baseline methods, we need to find a threshold value α such that all sample pairs with a correlation c larger than α are from *Class 1* and all others from *Class 0*. This corresponds to the following equation:

$$c = f(X_a, X_b) \quad Y' = \begin{cases} 1 & \text{if } c > \alpha \\ 0 & \text{else} \end{cases} \tag{7}$$

Here f represents the baseline method used. To find a good threshold α, we first applied f to all samples of our training set and added the resulting c-values to a sorted array. Thereafter, we searched the sorted array for an optimal delimiting value that minimizes the number of falsely grouped sample pairs. In the final step, α was set to this delimiter.

The results of our baseline methods tests are shown in the fourth and fifth lines in Table 3. We see that, except for the

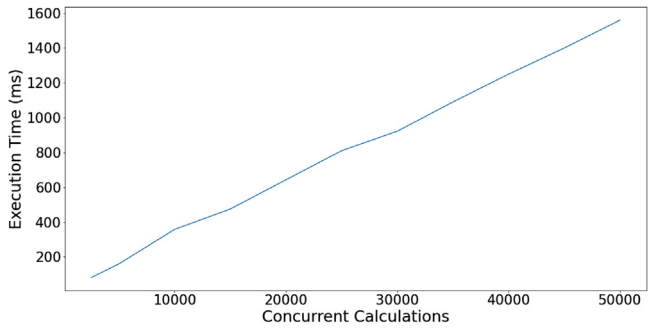

Fig. 7. Execution time to execute matchings in parallel.

precision of DTW, the two baselines are clearly outperformed by DeepMatch with the different sample sizes.

The reason for the poor performance of NORM_CORR is most likely its sensitivity to potential time-lags between its input sequences. This is due to the fact that two passengers, who are at different locations in a public transportation vehicle, register changes in altitude with a time-lag that corresponds with the quotient of the spatial distance between them and the speed of the vehicle. This time-lag produces a lower correlation value for NORM_CORR even though the passengers are in the same vehicle.

On the other hand, the poor results of DTW are most likely caused by its insensitivity to the temporal dimension. Warping the temporal dimension can cause the function to achieve a very high correlation value for some negative samples which increases the number of false positives. While this error is rewarded by the precision metric, it makes it very hard to find a good delimiter value α. This is the likely reason that the other three metrics are particularly bad for DTW.

Altogether, it seems that the baseline methods are less suited to perform in-vehicle presence detection on barometer data than DeepMatch.

5.6. Prediction performance of DeepMatch2

We also trained our updated version DeepMatch2 with the available data. The performance results of our tests are listed in the last row of Table 3. We can see that, except for the precision value of DTW that was already discussed above, DeepMatch2 outperforms all other tested models for all four performance metrics used. If we take a closer look at the results, we see that the largest change between DeepMatch 10 and DeepMatch2 is the increase of the recall value by more than a percentage point. The likely reason is that the improvements we made to the model (see Section 4), increased the ability of DeepMatch2 to correctly classify positive samples of the dataset. This could be achieved without classifying too many samples as positive, as can be seen by its precision value that is also slightly better than the one of DeepMatch 10.

Achieving good results for both the precision and recall means that DeepMatch2 is good at separating the samples in our evaluation dataset. That is also proved by the very high accuracy and F1-scores of DeepMatch2 which are both around 0.7% better than their counterparts in DeepMatch 10. Thus, in spite of increasing the compression of the sensor data by the factor of four, we managed to improve the performance metrics of DeepMatch2.

5.7. Execution time in the server

Since usually a lot of passengers use public transportation throughout the day, particularly during the rush hour, the server of a public transport authority performing the matching calculations of DeepMatch or DeepMatch2, needs to be able to handle large amounts of concurrent data simultaneously. To test the expected load for such a central server, we exploited a feature of Tensorflow that allows the matching module to accept multiple inputs. Further, we used the powerful parallel computational capabilities of Tensorflow that make it possible to calculate the matchings for all received inputs simultaneously.

Fig. 7 depicts the execution time of the matching module performing its matches based on latent data, after it has been extracted from the overall deep learning model of DeepMatch2. It reveals that 50,000 matching calculations can be executed in parallel in 1560 ms all running on a single five years old GTX 1080 GPU. Due to the fact that the matching calculation is only performed once for every 12.8 s of collected data per passenger, a data centre consisting of only three such GPUs could serve a city like Oslo with its 960,000 daily passenger trips even if all passengers travel at the same time. Running these calculation on a newer GPU with improved concurrency and computational capabilities, would improve these results even further.

Table 4
Android phones used in battery tests.

Brand	Age	Battery capacity
LG nexus 5X	5 years	2700 mAh
Huawei nexus P6	4 years	3450 mAh
Samsung galaxy S8	3 years	3000 mAh
Sony Z3 compact	6 years	2600 mAh
Google pixel 3a	0 years	3000 mAh

Table 5
Battery consumption per hour.

Brand	Data collection	Learning	Complete
Samsung	25 mA	26 mA	31 mA
LG	23 mA	24 mA	26 mA
Huawei	22 mA	23 mA	25 mA
Google	16 mA	17 mA	18 mA
Sony	15 mA	18 mA	21 mA

5.8. Battery consumption on smartphones

Power consumption is an important issue when using deep learning models on mobile devices that run on rechargeable batteries since the models often require a large number of calculations at high speed, which usually demands a lot of power. Fortunately, we do not require the *encoders* executed on the smartphones to run continuously in our approach. Even if the phone is in close proximity to a BLE-transmitter (see Section 3.1), we only carry out the encoder in certain intervals corresponding to the lengths of the samples to be matched, *e.g.*, every 12.8 s when using DeepMatch2.

In addition to running the deep learning model, the continuous sensor event generation and the transmission of the compressed data to the server are power consuming tasks the smartphones will have to perform. To ensure that our approach does not cause an excessive drain on the batteries of the smartphones, we constructed three test scenarios that reflect the battery consumption factors mentioned above:

- *Complete scenario*: All three factors of battery consumption, *i.e.*, the barometer data collection, data processing by the encoder, and data transmission,
- *Learning scenario*: Barometer data collection and data processing,
- *Data collection scenario*: Only barometer data collection.

Table 6
Run time and CPU overhead.

Brand	CPU	Mean run time	Overhead
Samsung	2.3 GHz + 1.7 GHz, Cortex-A53	49 ms	1–2%
LG	1.4 GHz + 1.8 GHz, 64-Bit Hexa-Core	46 ms	1–2%
Huawei	2.0 GHz + 1.55 GHz, 64-Bit Octa-Core	52 ms	1–2%
Google	2.0 GHz + 1.7 GHz, 64-Bit Octa-Core	19 ms	0–1%
Sony	2.5 GHz Quad-Core, 400 Krait	73 ms	3–4%

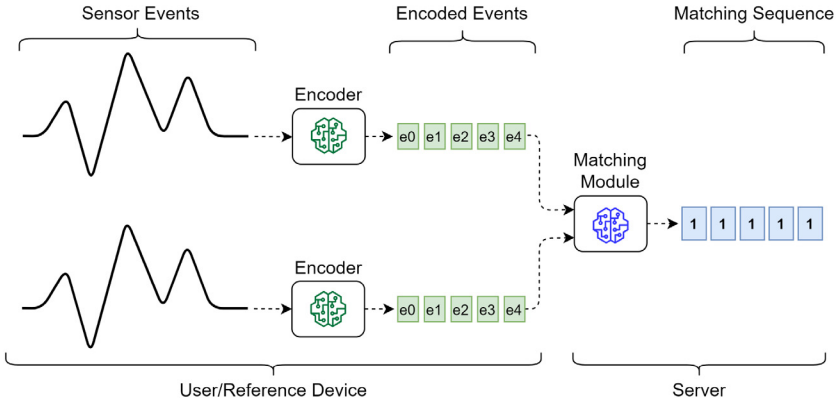

Fig. 8. Matching sequence output from the matching module.

To ensure diversity in our testing devices, we used five Android smartphones from five different manufacturers. To ensure age diversity, these phones are between zero and six years old, as can be seen in Table 4. To make sure that the test results were not influenced by the environments in which the tests were run, we performed all tests indoors with a constant temperature of 19 °C, representing the indoor temperature of a typical public transportation vehicle in Norway.

To measure the battery power usage, we used the tools *Batterystats* and *Battery Historian* provided by Google to log the battery consumption of all processes running on an Android device [34]. The tests were run in the background with the *wake lock* parameter enabled. This allowed us to simulate an environment in which our application retrieving sensor inputs, encoding them, and forwarding the encoded data to the server, runs in the background of a user phone.

The results of our tests are depicted in Table 5. They clearly show that for all devices used in the tests, our learning models influence the battery consumption on a smart device only marginally. Considering a passenger travelling with a public transportation vehicle for over two hours, measuring, encoding and transmitting barometer sensor events, only around 62 mAh will be used for the Samsung Galaxy S8, the phone with the highest power usage. With a battery capacity of 3000 mAh, this equals to the use of 2.1% of the overall battery capacity. Compared to the numbers reported in [35], this is substantially lower than most smartphone applications.

As a result of our tests, we consider that our approach has no significant negative impact on the overall battery consumption of the user smartphones. For the reference devices, we expect that they have a power connection with the battery of the transport vehicle.

5.9. Computational overhead on smartphones

As mentioned above, deep learning models are often large complex computational units. Thus, in addition to affecting the battery consumption of the devices running the model, the computations might influence their CPU usage such that a smartphone is not able to support other applications, that the user likes to run in parallel to our learning models DeepMatch or DeepMatch2. Therefore, we evaluated also the computational overhead of the models executed on smartphones. For these experiments, we used the same five devices listed in Table 4. We analysed both, the CPU usage and the time, the encoder module needed to process one sample of input. Table 6 shows the results of these tests. We can clearly see from the mean run time and CPU overhead produced for the devices, that the overhead of our models are barely noticeable and should not impact other functions executed on the phone in parallel.

6. Travelling user inference

Up to now, DeepMatch and DeepMatch2 can determine with a high accuracy whether a person's smartphone is in the same public transport vehicle as a RefDev over a fixed time interval of, *e.g.*, 12.8 s. To make our approach useable for, *e.g.*, automatic ticketing, however, a solution is required to find out, over which time period the owner of the phone effectively travels in the vehicle.

To infer the duration of being in the vehicle, we can utilize that a normal trip in a city bus may be up to an hour such that DeepMatch2 can conduct hundreds of samples. The samples of the user's phone taken on a trip and the corresponding ones produced by the reference device are paired and merged to a so-called *matching sequence*. This is illustrated in Fig. 8.

As an example, let us assume that a passenger travels with a bus for 20 min. That gives DeepMatch2 the necessary time to generate a matching sequence containing 93 successive matches. Due to the accuracy of 98.51% for DeepMatch2, the likelihood that all these matches are correctly detected as being in-vehicle (*i.e.*, *Class 1*), however, is only 24.76%. Thus, in more than three quarters of trips with a 20 min duration, at least one matching will be falsely declared as being out of the vehicle (*i.e.*, *Class 0*). Similarly, if a person rides in a car next to a bus for 20 min, e.g., due to slow moving traffic, the chance that all matches are *Class 0*, is also only around 25%. Thus, we need an algorithm that can infer passenger trips from matching sequences with a high degree of precision in spite of occasional matching errors. In this section, we describe how one can develop and evaluate such an inference algorithm.

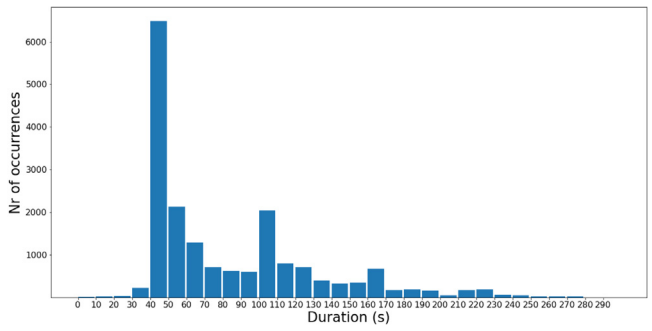

Fig. 9. Number of times any vehicle travelled a route segment in x seconds.

In the following, we first describe how we gathered the data that explains the travel time between two adjacent stops for busses in Norway's capital Oslo. Thereafter, we introduce some concepts based on which the algorithm is designed and presented. A relevant parameter of this algorithm is the degree of fault tolerance it supports, which is investigated at the end of this section.

6.1. Travelling times between adjacent bus stops

To get the travelling times between two adjacent stops for busses in Oslo, we gathered all real-time data of the bus network of Oslo over twelve hours on a normal Monday. ENTUR, a government-funded organization, gathers and openly shares traffic data from all public transportation operators in Norway. In particular, it offers an API [36] which is based on the SIRI 2.0 standard [37].

The collected data allows us to understand the distribution of the travelling times between two adjacent stops. To this end, we aggregated the arrival and departure times for all buses at all stops throughout the recorded 12-hour slot. From this, we calculated for all bus trips the times needed to travel between two adjacent stops. The results were summed up, and we obtained the distribution depicted in Fig. 9. The *x* axis refers to travelling times in seconds while the *y* axis shows how often busses needed a particular time segment between two neighbouring stops during the 12-h slot. As can be seen, the most prevalent travelling time between two adjacent stops is between 40 s and 50 s. Another interesting fact is that in less than 2% of the gathered cases, the travelling time between two stops is less than 40 s. We will show how these facts can be utilized to provide good results.

6.2. Matching sequences and travel inference algorithms

As introduced above and illustrated in Fig. 8, a *matching sequence* is the sequence of outputs generated by the matching module of DeepMatch2. It is basically a sequence of *ones* and *zeros* that forms the input of the *inference algorithm*. Based on this input, the inference algorithm decides whether the user travelled in the vehicle for the duration of the matching sequence, or not.

To better understand the possible errors that the inference algorithm could make, we classify its results (in analogy to the true and false positives and negatives introduced in Section 5.2) as follows:

- *True Travelling user* (T_T) denotes an actual passenger who is correctly inferred as a traveller.
- *False Travelling user* (T_F) is a person not using the public transport vehicle, but who was falsely inferred to be travelling in it.

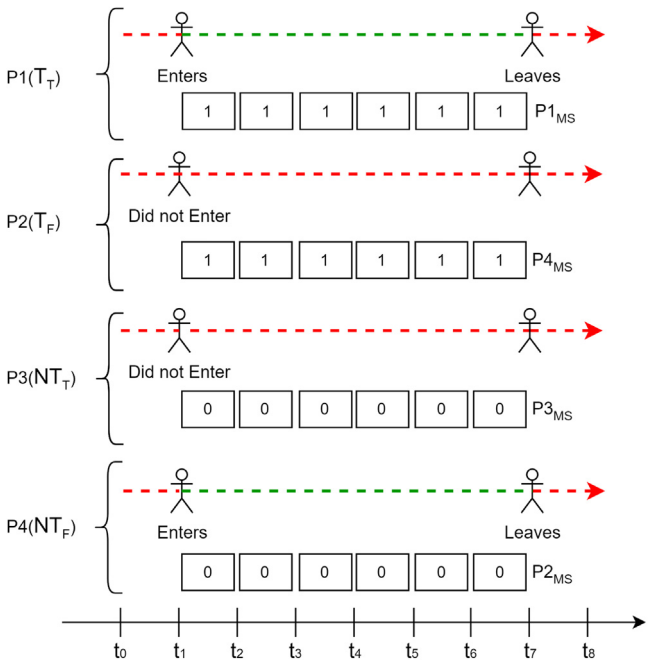

Fig. 10. Example travellers and expected matching sequence. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

- *True Non-Travelling user* (NT_T) is a user not travelling with the vehicle in question who is correctly inferred as being not in the vehicle.
- *False Non-Travelling user* (NT_F) is an actual travelling user who was, however, falsely inferred as not travelling with the vehicle.

In Fig. 10, the four types of travellers are illustrated, where green dashed lines indicate the time period that the passengers are inside the vehicle, while red dashed lines refer to the phases they are outside. The orders of white boxes beneath the dotted lines represent the matching sequence for each passenger. The goal of the algorithm is to detect true travelling and true non-travelling users with a high accuracy. In particular, it should keep the number of false travelling users very low since billing people who do not use public transport, may easily lead to complaints, lawsuits, and bad press for the operator. Moreover, false non-travelling users shall be avoided since they end up with travelling for free.

6.3. User travel inference algorithm

In this section, we describe our methodology to design and evaluate the inference algorithm. It shall take the accuracy of DeepMatch2 of 98.51% into account. While this accuracy is relatively high, it is not perfect as matching sequences can occasionally contain a mix of *ones* and *zeros*, and the inference algorithm should be able to perform inference with a high accuracy even from such mixed sequences. Overall, our goal is to find an inference strategy that keeps the number of false non-travellers and false travellers low.

We first need to calculate the accuracy of an inference algorithm based on the accuracy of DeepMatch2. We use the *binomial experiment* to calculate the likelihood of occurring a certain ratio of *ones* and *zeros* in a matching sequence:

$$P(k) = \binom{n}{k} p^k (1-p)^{n-k} \tag{8}$$

Here, n denotes the number of trials, *i.e.*, the length of the matching sequence, while k refers to the number of successful trials, *i.e.*, the number of correct matchings in the sequence. Finally, p states the probability of a successful trial which, in our case, corresponds to the accuracy of DeepMatch2, *i.e.*, 0.9851.

With the binomial experiment, we can compute the two probabilities that a certain proportion of *ones* and *zeros* in a matching sequence is rightfully inferred to be in or out of the vehicle. From that, we can easily determine the likelihoods for the four user types introduced in Section 6.2.

As an example, let us assume that, in fear of inferring false travelling users, the public transport authority decides to bill a passenger when all the six entries in the matching sequence are *ones*. Using formula (8) leads to the following likelihoods for the four inference results:

$$P_{T_T}(6) = 0.9851^6 = 0.9139$$
$$P_{NT_F}(6) = 1 - P_{T_T}(6) = 0.0861$$
$$P_{T_F}(0) = (1 - 0.9851)^6 = 1.09 \cdot 10^{-11}$$
$$P_{NT_T}(0) = 1 - P_{NT_F}(0) = 0.999999999989$$

The likelihood, that DeepMatch2 produced all six *Class 1* predictions wrongly and with that infers a false travelling user, is $P_{T_F}(0) = 1.09 \cdot 10^{-11}$. In correspondence, the likelihood to detect a true non-traveller is very high since our strategy declares all passengers who produced at least one *zero* in their matching sequences, as not travelling. The corresponding probability is $1 - P_{T_{NF}}(0) = 0.999999999989$.

The price of this very rigid approach is the relatively high rate of false non-travelling users which is calculated as $1 - P_{T_T}(6) = 0.0861$. Thus, nearly every twelfth passenger gets a free ride which might be unacceptable for most operators. In consequence, the likelihood for a true travelling user is only $P_{T_T}(6) = 0.9139$. To avoid such a large rate of false non-travelling users, we need to bring some *tolerance* into our inference algorithm. To achieve that, we extend Eq. (8) for the binomial probability to the so-called cumulative binomial probability, that is described by the following formulas:

$$P(k \geq M) = \sum_{k=M}^{n} \binom{n}{k} p^k (1-p)^{n-k} \tag{9}$$

$$P(k \leq M) = \sum_{k=0}^{M} \binom{n}{k} p^k (1-p)^{n-k} \tag{10}$$

In formula (9), M refers to the *minimum* and in (10) to the *maximum* number of trials that have to be successful in order to accept a trip as in-vehicle. The other symbols used in these formulas are identical to those introduced for formula (8).

The cumulative binomial probability provides the means to calculate the probabilities for more fault-tolerant inference algorithms, *e.g.*, accepting matching sequences of the length six as in-vehicle, when they contain at least five *ones*. This algorithm leads to the following results for the four inference results:

$$P_{T_T}(k \geq 5) = 0.9851^6 + 6 \cdot 0.9851^5(1 - 0.9851) = 0.9968$$
$$P_{NT_F} = (1 - P_{T_T}(k \geq 5)) = 0.0032$$
$$P_{T_F}(k \leq 1) = (1 - 0.9851)^6 + 6 \cdot 0.9851(1 - 0.9851)^5$$
$$= 4.35 \cdot 10^{-9}$$
$$P_{NT_T} = (1 - P_{T_F}(k \leq 1)) = 0.9999999957$$

This more fault-tolerant algorithm reduces the likelihood of false non-travelling users to just around 0.3%. While the number of false travelling users is increased by two digits compared to the more rigid algorithm described above, it is still very low. Therefore, this more tolerant inference algorithm seems to be a better fit for a billing system than the stricter one discussed above.

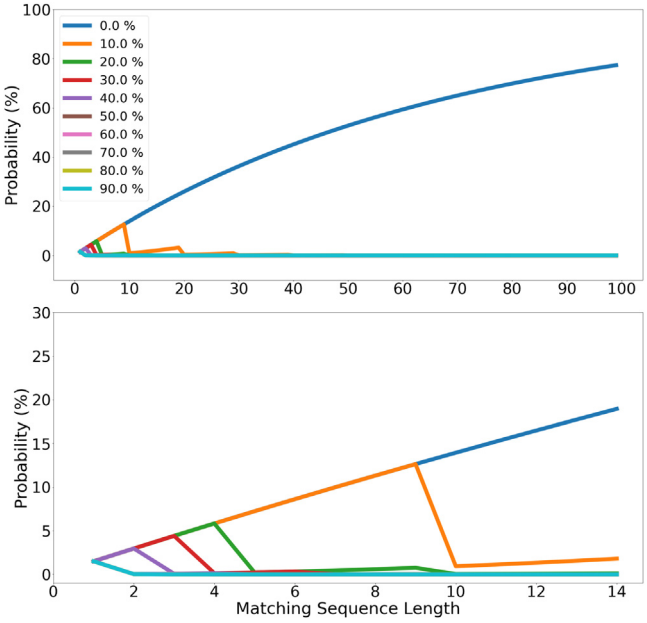

Fig. 11. Probability of predicting a False Non-Travelling user (P_{NT_F}) over varying matching sequence lengths. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

The fact is that we still do not know if this algorithm is the best, or if more fault-tolerant policies render even better results. Moreover, for most passenger trips, DeepMatch2 produces much longer matching sequences than ones with just six matching pairs. Utilizing the data from the public transportation behaviour analysis discussed in Section 6.1, we conducted several experiments to find out the best inference algorithm solutions, which are discussed below.

6.4. Considering different forms of fault tolerance

The aforementioned comparison of two algorithms for matching sequences of length six shows that there is a trade-off between false travellers and false non-travellers based on the degree of fault tolerance—the percentage of *zero* values that a matching sequence may contain whilst still being inferred as in-vehicle. In the rigid example in Section 6.3, the fault tolerance was 0, while for the second case it was $\frac{1}{6} = 16.\bar{6}\%$. The value M in the cumulative binomial probability formulas (9) and (10) can be calculated from the fault tolerance as follows:

$$M = \left\lfloor (1 - fault\_tolerance) \cdot n + \frac{1}{2} \right\rfloor \tag{11}$$

Thus, the higher the fault tolerance is, the more occasional *zeros* are allowed in the matching sequence when the inference algorithm infers that the user is inside the vehicle. In consequence, by using a higher fault tolerance, we reduce the number of false non-travellers, albeit at the cost of more false travellers.

To analyse the influence, we calculated the likelihoods of false non-travellers and false travellers for different fault tolerances ranging from 0% to 90% in steps of 10% and for all matching sequence lengths from 1 to 100. Fig. 11 depicts the probabilities for non-travelling users depending on the lengths of the matching sequences. The dark blue curve, showing no fault tolerance at all, rises towards 1 while all other trajectories converge towards 0. The interesting observation is how early the approximation

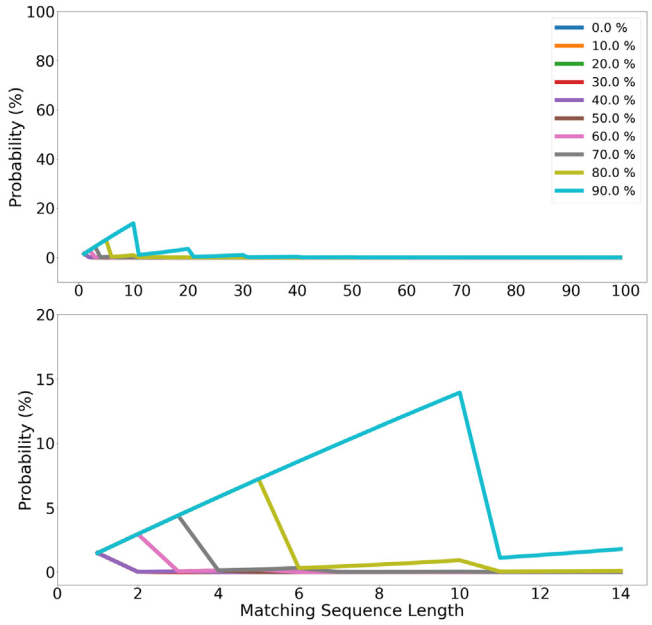

Fig. 12. Probability of predicting a False Travelling user (P_{T_F}) over matching sequence lengths.

towards 0 starts: Even with a fault tolerance of just 10%, the likelihood for producing non-travelling users is next to nothing for trips that produce more than 40 matchings. Thus, for trip lengths longer than nine minutes, this low fault tolerance would be sufficient.

The curves show that the selection of a good fault tolerance value is only relevant for short trips for which relatively short matching sequences are produced by DeepMatch2. To make the differences of the curves for short journeys more legible, we show a zoomed-in version in the lower part of Fig. 11. In Section 6.1, we claimed that less than 2% of all time periods between adjacent bus stops is lower than 40 s. Since passengers can enter and leave the vehicle only at bus stops, we can therefore assume that for nearly every trip, at least three matchings are produced. From our curves, one can see that with a fault tolerance of 40%, it is enough to produce nearly no false non-travellers.

Fig. 12 shows the same curves for false travelling users. Similarly to the false non-travelling users, the likelihood of creating false travelling users is negligible if a smartphone user is close to a vehicle for a longer time period. Thus, even with a fault tolerance of 90%, the inference is enough if the length of the matching sequence is at least 30, which corresponds to 6.4 min.

Yet, a non-travelling user will usually be close to a vehicle only for a relatively short time period, *e.g.*, less than a minute. Therefore, it is particularly important that the inference algorithm handles those cases correctly. The zoomed-in curves in the lower part of Fig. 12 show that the fault tolerance of 40%, that we already mentioned as sufficient for false non-travellers, is very close to zero for all matching sequences except for those consisting of just one matching.

Our findings about false non-travelling and false travelling users provide useful hints for the configuration of the inference algorithm. Since nearly no distances between two stops are less than 30 s (cf. Fig. 9), if our algorithm declares all matching sequences of lengths one and two as out-of-vehicle, it will hardly affect the overall accuracy of the algorithm. For all matching sequences of lengths three or larger, the policy may use a fault tolerance of 40%, which should lead to excellent results with very few false non-travellers and false travellers.

Table 7
Weighted averages for all matching sequences with lengths between one and 21.

Fault tolerance	P_{NT_F}	P_{T_F}	$0.5 \cdot P_{NT_F} + 0.5 \cdot P_{T_F}$	$0.1 \cdot P_{NT_F} + 0.9 \cdot P_{T_F}$
0%	31.22957%	**0.15125%**	15.69041%	3.25908%
10%	8.49190%	0.15125%	4.32158%	0.98532%
20%	1.73878%	0.15126%	0.94502%	0.31001%
30%	0.95806%	0.15139%	0.55473%	0.23206%
40%	0.46660%	0.15828%	**0.31244%**	**0.18911%**
50%	0.15829%	0.46545%	**0.31187%**	0.43473%
60%	0.15796%	0.48792%	0.32294%	0.45492%
70%	0.15139%	0.96164%	0.55652%	0.88062%
80%	0.15125%	2.54341%	1.34733%	2.30419%
90%	**0.15125%**	10.11554%	5.13340%	9.11911%

6.5. A more formal look at the trade off between NT_F and T_F

The discussion above to determine a good fault tolerance seems coherent, however, it is not very formal. A public transport authority that has resilient statistics about the travelling times of its passengers, can therefore go a step further and determine the trade offs between NT_F and T_F for different fault tolerance percentages more formally.

To be able to assess false travellers differently than the potentially less problematic false non-travellers, we use a weight factor $w \in [0, 1]$ that describes the weight that T_F shall have in comparison to NT_F. With $P^{ft}_{NT_F}(ft, msl)$ and $P^{ft}_{T_F}(ft, msl)$, we state the probabilities for NT_F resp. T_F for a certain fault tolerance ft and message sequence length msl that can be calculated using the formulas (9), (10), and (11). Moreover, p_l refers to the likelihood that the travelling time of a passenger has a certain length such that the corresponding matching sequence has l-many entries. Finally, Max_{msl} describes the maximum length of matching sequences that can occur in practice. These values can be computed from the operator's travelling time statistics. The *weighted average* Av between NT_F and T_F can be computed as follows:

$$Av(ft, w) = \sum_{l=1}^{Max_{msl}} p_l \cdot (w \cdot P^{ft}_{T_F}(ft, l) + (1 - w) \cdot P^{ft}_{NT_F}(ft, l)) \qquad (12)$$

With formula (12), one can compare different fault tolerances and select the one ft that gives the lowest value $Av(ft, w)$ for the desired weight w.

Unfortunately, in spite of an in-depth search and requests to some public transportation authorities, we could not get access to meaningful statistics about usual travel durations of passengers. Thus, we based our calculations on the worst case scenario in which every passenger travels only between a stop and the adjacent one. We also assume that the number of people travelling are evenly distributed among the operator's buses. Then, we can use the distribution presented in Fig. 9 to calculate the likelihoods p_l for the lengths of the matching sequences.

The result of applying formula (12) to these values and the 10 different fault tolerant values 0% to 90% is listed in Table 7. The columns show the likelihoods for false non-travellers and false travellers as well as the weighted averages for the weights $w = 0.5$ and $w = 0.9$. Non-surprisingly, the likelihood for NT_F is lower if a higher degree of fault tolerance is allowed, while it is the opposite for T_F. If we weight both error cases equally (*i.e.*, $w = 50\%$), fault tolerances of 40% and 50% render the best trade off results closely followed by 60%. When we consider false travellers as more important and set the weight to $w = 90\%$, the result is much clearer. Here, 40% is the winner followed by 30%. These results foster our assumption from Section 6.4 that 40% seems to be a efficient fault tolerance value.

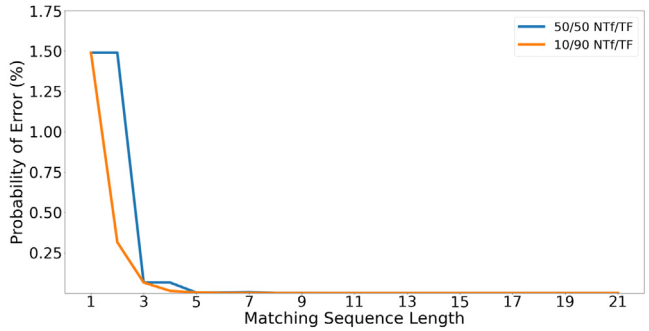

Fig. 13. Total number of erroneous travel predictions over MS lengths using a Fault Tolerance of 40% and the average weight w set to 50% (blue) and 90% (orange). (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

In a final step, we checked the probability of errors for certain matching sequence lengths l, *i.e.*, $w \cdot P^{ft}_{T_F}(ft, l) + (1-w) \cdot P^{ft}_{NT_F}(ft, l)$. The results for the two weights and a fault tolerance of 40% are shown in Fig. 13. We see in both curves that the two shortest matching sequence lengths 1 and 2 are responsible for nearly all of the errors, while all the others give good results. This confirms the suggestion of our inference algorithm to bill only passengers for which at least three matchings were generated. The only losses are for passengers that travel only between two adjacent stops with less than 40 s distance. Assuming an average bus speed of 27 km/h between these stops, they are just 300 metres apart. In reality, people would walk this distance instead of waiting for a bus.

It would be nice to repeat our trade off calculations based on real travelling time statistics, and we will do so as soon as we get our hands on such data. However, we do not expect very different results since, in reality, the average travelling times will be much longer than in our thought experiment, and the graphs in Figs. 11 and 12 show that the inference is becoming better with longer matching sequences. Therefore, the likelihoods for the errors would probably be lower than in Table 7. However, based on the considerations from Section 6.4 and in this subsection, we still expect that 40% will be the winning fault tolerance value in most cases.

7. Conclusions and future work

To address the challenge of in-vehicle presence detection as an important aspect of mobile context analysis, we introduced our proposed deep learning-based approach, called DeepMatch and its improved version DeepMatch2. Our approach utilizes the sensor event streams of smartphones to predict their presence inside public transportation vehicles with an accuracy of 98.51% in the case of DeepMatch2. The deep learning model consists of custom made Stacked Convolutional Autoencoders for feature extraction and dimensionality reduction configured in a Siamese architecture, and a matching module consisting of several layers of stacked Convolutional Layers for event stream matching. The deep learning model is distributed among the smartphones carried by passengers, a reference device installed in public transport vehicles, and a central server. The Stacked Convolutional Autoencoders allow for compressing the sensor events through feature extraction and dimensionality reduction on the smartphone and the reference device, while the event matching is performed on the server. Through dimensionality reduction, the input data is reduced by the factor eight such that the bandwidth of the data transferred to the server is considerably reduced without losing the information of the data necessary to perform the matching.

Furthermore, we discussed how the deep learning-based approach can be used to create inference algorithms that deduce the travel time duration for passengers travelling in public transportation with a very high accuracy. In particular, we presented a theoretical framework that can be used to configure the inference algorithms to weight the various types of potential erroneous inferences, and thus accommodate the needs of the public transportation providers.

As our future plan, we intend to implement a pilot of DeepMatch2 together with a public transportation provider in Norway. Moreover, we intend to research on the optimum length of the data segments and the frequency of data gathering (from the reference devices and the smartphones) in order to minimize the amount of data needed for in-vehicle presence detection.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was supported by the Norwegian Research Council within the Industrial Ph.D. scheme under Grant 276259 (MobiTrack project).

References

[1] BankMyCell, How many smartphones are in the world? 2021, https://www.bankmycell.com/blog/how-many-phones-are-in-the-world. Accessed: 2021-04-09.
[2] Seungwoo Kang, Youngki Lee, Chulhong Min, Younghyun Ju, Taiwoo Park, Jinwon Lee, Yunseok Rhee, Junehwa Song, Orchestrator: An active resource orchestration framework for mobile context monitoring in sensor-rich mobile environments, in: IEEE International Conference on Pervasive Computing and Communications (PerCom), IEEE Computer, Mannheim, Germany, 2010, pp. 135–144.
[3] W. Narzt, S. Mayerhofer, O. Weichselbaum, S. Haselböck, N. Höfler, Be-In/Be-Out with bluetooth low energy: Implicit ticketing for public transportation systems, in: IEEE 18th International Conference on Intelligent Transportation Systems, IEEE, Las Palmas, Spain, 2015, pp. 1551–1556.
[4] T. Gyger, O. Desjeux, EasyRide: Active transponders for a fare collection system, IEEE Micro 21 (6) (2001) 36–42.
[5] C. Sarkar, J.J. Treurniet, S. Narayana, R.V. Prasad, W. de Boer, SEAT: Secure energy-efficient automated public transport ticketing system, IEEE Trans. Green Commun. Netw. 2 (1) (2018) 222–233.
[6] R. Meng, D.W. Grömling, R.R. Choudhury, S. Nelakuditi, RideSense: Towards ticketless transportation, in: 2016 IEEE Vehicular Networking Conference (VNC), IEEE, Columbus, OH, USA, 2016, pp. 1–8.
[7] M. Won, A. Mishra, S.H. Son, HybridBaro: Mining driving routes using barometer sensor of smartphone, IEEE Sens. J. 17 (19) (2017) 6397–6408.
[8] Shuochao Yao, Shaohan Hu, Yiran Zhao, Aston Zhang, Tarek Abdelzaher, Deepsense: A unified deep learning framework for time-series mobile sensing data processing, in: 26th International Conference on World Wide Web, ACM, Perth, Australia, 2017, pp. 351–360.
[9] Magnus Oplenskedal, Amir Taherkordi, Peter Herrmann, DeepMatch: Deep matching for in-vehicle presence detection in transportation, in: Proceedings of the 14th ACM International Conference on Distributed and Event-Based Systems, 2020, pp. 97–108.
[10] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, Sateesh Addepalli, Fog computing and its role in the internet of things, in: 1st Workshop on Mobile Cloud Computing (MCC), ACM, Helsinki, Finland, 2012, pp. 13–16.
[11] T. Gründel, H. Lorenz, K. Ringat, The ALLFA ticket in Dresden. Practical experience of fare management based on Be-In/Be-Out & automatic fare calculation, in: IPTS Conference, Seoul, South Korea, 2006.
[12] V. Kostakos, T. Camacho, C. Mantero, Wireless detection of end-to-end passenger trips on public transport buses, in: 13th IEEE International Conference on Intelligent Transportation Systems, 2010.
[13] Andrzej Kwiecień, Michał Maćkowski, Marek Kojder, Maciej Manczyk, Reliability of bluetooth smart technology for indoor localization system, in: International Conference on Computer Networks (CN), in: CCIS 522, Springer-Verlag, Br'unow, Poland, 2015, pp. 444–454.
[14] Sriharsha Kuchimanchi, Bluetooth Low Energy based Ticketing Systems (Master's thesis), Aalto University, Espoo, Finland, 2015.

[15] Kartik Sankaran, Minhui Zhu, Xiang Fa Guo, Akkihebbal L Ananda, Mun Choon Chan, Li-Shiuan Peh, Using mobile phone barometer for low-power transportation context detection, in: 12th ACM Conference on Embedded Network Sensor Systems, ACM, Memphis, TN, USA, 2014, pp. 191–205.

[16] Salvatore Vanini, Francesca Faraci, Alan Ferrari, Silvia Giordano, Using barometric pressure data to recognize vertical displacement activities on smartphones, Comput. Commun. 87 (2016) 37–48.

[17] Bo-Jhang Ho, Paul Martin, Prashanth Swaminathan, Mani Srivastava, From pressure to path: Barometer-based vehicle tracking, in: 2nd ACM Inter. Conf. on Embedded Systems for Energy-Efficient Built Environments (BuildSys), ACM, Seoul, South Korea, 2015, pp. 65–74.

[18] A. Dimri, H. Singh, N. Aggarwal, B. Raman, D. Bansal, K.K. Ramakrishnan, RoadSphygmo: Using barometer for traffic congestion detection, in: 8th International Conference on Communication Systems and Networks (COMSNETS), IEEE Computer, Bangalore, India, 2016, pp. 1–8.

[19] X. Wang, L. Kong, T. Wei, L. He, G. Chen, J. Wang, C. Xu, VLD: Smartphone-assisted vertical location detection for vehicles in urban environments, in: 2020 19th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN), 2020.

[20] Jindong Wang, Yiqiang Chen, Shuji Hao, Xiaohui Peng, Lisha Hu, Deep learning for sensor-based activity recognition: A survey, Pattern Recognit. Lett. 19 (2017) 3–11.

[21] Yi Zheng, Qi Liu, Enhong Chen, Yong Ge, J. Leon Zhao, Exploiting multi-channels deep convolutional neural networks for multivariate time series classification, Front. Comput. Sci. 10 (1) (2016) 96–112.

[22] P. Castrogiovanni, E. Fadda, G. Perboli, A. Rizzo, Smartphone data classification technique for detecting the usage of public or private transportation modes, IEEE Access 8 (2020) 58377–58391, http://dx.doi.org/10.1109/ACCESS.2020.2982218.

[23] Lin Wang, Hristijan Gjoreski, Mathias Ciliberto, Paula Lago, Kazuya Murao, Tsuyoshi Okita, Daniel Roggen, Summary of the Sussex-Huawei locomotion-transportation recognition challenge 2020, in: Adjunct Proceedings of the 2020 ACM International Joint Conference on Pervasive and Ubiquitous Computing and Proceedings of the 2020 ACM International Symposium on Wearable Computers, in: UbiComp-ISWC '20, Association for Computing Machinery, New York, NY, USA, ISBN: 9781450380768, 2020, pp. 351–358.

[24] Christian Mayer, Ruben Mayer, Majd Abdo, StreamLearner: Distributed incremental machine learning on event streams: Grand challenge, in: 11th ACM International Conference on Distributed and Event-Based Systems, ACM, Barcelona, Spain, 2017, pp. 298–303.

[25] Michael Spörk, Carlo Alberto Boano, Kay Römer, Performance and trade-offs of the new PHY modes of BLE 5, in: Proceedings of the ACM MobiHoc Workshop on Pervasive Systems in the IoT Era (PERSIST-IoT), ACM, 2019, pp. 7—12.

[26] Gregory Koch, Richard Zemel, Ruslan Salakhutdinov, Siamese neural networks for one-shot image recognition, 2015, p. 8, https://www.cs.cmu.edu/~rsalakhu/papers/oneshot1.pdf.

[27] Cheng Zhang, Wu Liu, Huadong Ma, Huiyuan Fu, Siamese neural network based gait recognition for human identification, in: IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), IEEE, Shanghai, China, 2016, pp. 2832–2836.

[28] Jane Bromley, Isabelle Guyon, Yann LeCun, Eduard Säckinger, Roopak Shah, Signature verification using a "Siamese" time delay neural network, in: Advances in Neural Information Processing Systems, Morgan Kaufmann Publishers, San Francisco, CA, USA, 1994, pp. 737–744.

[29] Ian Goodfellow, Yoshua Bengio, Aaron Courville, Deep Learning, MIT Press, Cambridge, MA, USA, 2016, pp. 505–528, Chapter Autoencoders. http://www.deeplearningbook.org.

[30] Naveen Sai Madiraju, Seid M. Sadat, Dimitry Fisher, Homa Karimabadi, Deep temporal clustering: Fully unsupervised learning of time-domain features, 2018, p. 11, arXiv cs, arXiv:1802.01059, Article 1802.01059.

[31] Jonathan Masci, Ueli Meier, Dan Cireşan, Jürgen Schmidhuber, Stacked convolutional auto-encoders for hierarchical feature extraction, in: International Conference on Artificial Neural Networks (ICANN), in: LNCS, vol. 6791, Springer-Verlag, Espoo, Finland, 2011, pp. 52–59.

[32] Akara Supratak, Hao Dong, Chao Wu, Yike Guo, DeepSleepNet: A model for automatic sleep stage scoring based on raw single-channel EEG, IEEE Trans. Neural Syst. Rehabil. Eng. 25 (11) (2017) 1998–2008.

[33] Tensorflow, Tensorflow 2.3, 2019, https://www.tensorflow.org/api_docs/python/tf. Accessed: 2020-11-27.

[34] Battery Historian, Batterystats and battery historian, 2019, https://developer.android.com/studio/profile/battery-historian. Accessed: 2019-10-23.

[35] Xiaomeng Chen, et al., Smartphone energy drain in the wild: Analysis and implications, ACM SIGMETRICS Perform. Eval. Rev. 43 (1) (2015) 151–164.

[36] Entur, Entur public transport API, 2020, https://developer.entur.org. Accessed: 2020-10-07.

[37] SIRI, SIRI standard, 2020, http://www.transmodel-cen.eu/standards/siri/. Accessed: 2020-10-07.