NTNU
Norwegian University of
Science and Technology

Sensor Fusion of Ultrasound and Motion Data for Gesture Recognition on Smartphones

Eirik Lid

Master of Science in Cybernetics and Robotics
Submission date: June 2018
Supervisor: Ole Morten Aamo, ITK
Co-supervisor: Øyvind Stamnes, Elliptic Labs

Norwegian University of Science and Technology
Department of Engineering Cybernetics

Abstract

Through the advances in computing power on smartphones in recent years, the potential for running computationally heavy systems, with higher demands for fast processing has become feasible to run on these devices. The field of machine learning is rapidly evolving and is continuing to influence more and more fields. How these methods work and how to apply them in human activity recognition problems is assessed in this thesis. An assessment of the current proximity classifier solution using ultrasound is done and used as a benchmark for the other models. With the expressive power of neural networks, attempts at combing ultrasound and inertia data for classification will be made. Different approaches to classify the gesture of bringing the phone up to the ear will be presented, analyzed, tested and discussed. These include using a sliding window approach with a multilayer perceptron network and using a recurrent neural network on the sequences. Important characteristics used to compare different machine learning models is presented. The process of recognizing gestures with inertial sensor data is explained and tested. Using just inertia data the neural network showed promising results and was used further in combination with ultrasound. A heuristic method of known transition pattern between output classes has been proposed and the reason for why it did not improve the performance of the model has been discussed.

State of the art methods to optimize hyperparameters in neural networks is presented and tools such as Hyperas is introduced to apply these methods on parameters such as dropout rate, activation function, and size of a layer. The possibility of training models with the Keras framework and deploying them to Android devices is introduced. The performance of the models is evaluated running on an app, which is evaluated empirically through experiments on different gesture scenarios. The final sensor fusion model was also tested by multiple users, to check the user independent performance of the model. The model's ability to distinguish between the approach and retract motion was quantified.

Sammendrag

På grunn av økt datakraft på smarttelefoner de siste årene, har potensialet for å kjøre tyngre systemer på disse enhetene, med høyere krav til rask behandling blitt mulig. Forskning på maskininlæring utvikler seg raskt og fortsetter å påvirke flere og flere fagfelt i større og større grad. Hvordan disse metodene fungerer og hvordan man bruker dem i menneskelig aktivitetsgjenkjenning, blir presentert i denne oppgaven. En vurdering av den nåværende modellen som bruker ultralyd er gjort, og denne brukes som sammenligningsgrunnlag for de andre modellene. Med nevrale nettverk vil forsøk på å kombinere ultralyd- og bevegelesesdata for klassifisering bli gjort. Ulike tilnærminger til å klassifisere bevegelsen av å løfte telefonen opp til øret vil bli presentert, analysert, testet og diskutert. Disse inkluderer bruk av et ”sliding widow” input for et nevral nett og bruk av ”recurrent neural network” på sekvensene. Viktige egenskaper som brukes til å sammenligne ulike maskinlæringsmodeller presenteres. Prosessen med å gjenkjenne bevegelser med akselerometer- og gyroskopdata blir forklart og testet. Ved hjelp av bare bevegelsesdata viste det nevrale nettverket lovende resultater, og ble brukt videre i kombinasjon med ultralyd. En heuristisk metode som utnytter kjent overgangsmønster mellom klasser har blitt foreslått, og grunner til hvorfor metoden ikke forbedret modellens ytelse er blitt diskutert.

Toppmoderne metoder for å optimalisere hyperparametere i nevrale nettverk presenteres og verktøy som Hyperas er introdusert for å anvende disse metodene på parametere som ”dropout”-rate, aktiveringsfunksjon og størrelse på et lag i nettverket. Muligheten til å trene modeller med Kerasn og distribuere dem til Android-enheter blir introdusert. Modellene kjøres i en app, for å evalueres empirisk gjennom eksperimenter på forskjellige gestscenarier. Den endelige sensorsammenkoblingsmodellen ble også testet av flere brukere for å kontrollere modellens ytelse ved hos forskjellige brukere. Modellenes evne til å skille mellom å løfte opp- og å legge ned telfonen ble og vurdert.

Preface

This master thesis is submitted as a part of the requirements for the Master of Science degree within the field of Engineering Cybernetics at the Norwegian University of Science and Technology (NTNU). It is written in collaboration with Elliptic Laboratories AS, who presented the problem formulation.

I would like to thank my supervisor, professor Ole Morten Aamo, for his guidance and for being a good discussion partner. A big thanks go to co-supervisor, Øyvind Nistad Stamnes, for constructive feedback, support and always being helpful regardless of what problem I needed help with.

Thanks are also due to Shaurya Sharma, for helping with the implementation on Keras and to Joachim Bjørne, for helping me whatever problem that would occur in with the ultrasound classifier in the Android implementation.

Finally, I would like to thank my family for their support and a special thanks to my fiancée Emelie Vårli Solheim, even though she has no idea what I am writing about.

The starting point for this thesis, given by Elliptic Laboratories AS, include 832 data sequences and a framework to load data into Keras.

My own contributions include altering the framework so it would be able to handle multiple transitions in a sequence, make a sequential data generator, record 52 additional sequences, set up a framework for optimizing model structure in Keras, an app has been made to test the Keras-model on device and to test and evaluate different neural network approaches to the problem.

Eirik Lid
Trondheim, 11th June, 2018

Table of Contents

List of Tables

List of Figures

Abbreviations

CNN	=	Convolutional Neural Network
HAR	=	Human Activity Recognition
IMU	=	Inertial Measurement Unit
IR	=	Infrared
MLP	=	Multi Layer Perceptron
NN	=	Neural Network
ML	=	Machine Learning
RNN	=	Recurrent Neural Network
US	=	Ultrasound

Chapter 1

Introduction

1.1 Background

Elliptic Labs delivers 3D interaction and sensing products for mobile, wearables and IoT devices powered by ultrasound. Many of these devices, in particular smartphones, have inertial sensors such as accelerometers and gyros. The data from these sensors can be used to complement Elliptic Labs ultrasound based sensing products. Elliptic Labs therefore wants to investigate if inertial sensors can be combined with their ultrasonic sensor solutions to improve the performance of their virtual proximity sensor.

1.2 Motivation

Some gestures require the user the user to move the device, for example picking up the phone when it is ringing. Other gestures require the phone to be kept somewhat still, for example when taking a picture with the device. Ultrasound measurements measure the distance using echo, and the velocity from Doppler shift in the frequency of the ultrasound signal. The measurements are all relative measurements. The relative motion of the device can lead to false positive classification using ultrasound. Adding information from inertia sensors can enhance the confidence of the ultrasound measurements, stating that something is moving towards or away from the device. The inertia sensor can give information about if it is the device that is being moved or if it is something moving towards the device.

1.3 Problem Description

State of the art machine learning algorithms for fusing inertial sensor data and ultrasound signals will be made. Robustness to variation in the motion when different users retract and extend the device will be considered. The complexity and size of the model will be accessed, as well as performance when running on a device. How inertia data performs on it own and in combination with ultrasound will be considered.

Specific research questions that will be answered is:

- How to make a model that is robust to variation in the approach and retract motion from user to user?
- Is it possible to separate the approach and retract motion using inertial sensors?
- What is a good model structure (size, layers, type of nodes) for this problem?
- Does the inertial sensor data improve the classification accuracy? How much?

The tasks for this project is:

- Create a suitable infrastructure for training, deploying and validating models. Use existing Elliptic Labs tools/infrastructure and the Google Tensorflow framework.
- Make a basic demo app in Android that can be used to evaluate the performance.
- Select, train and evaluate different machine learning algorithms for classifying near/-far events.

Elliptic Labs provides the following infrastructure to aid the development:

- A smartphone device and an app for data collection.
- A set of recordings containing both ultrasound signals, inertial measurements and metadata about the recording.
- An interface to load the recordings into Keras.

1.4 Related Work

In Human Activity Recognition most of the NN algorithm focus on offline classification of activities. Often several sensors are placed in different places on the body. The recordings are then taken and processed afterward. There are different techniques for different types of activities in HAR. Some focus on periodic activities such as walking, running, etc. Others focus on static postures such as sitting, standing, etc. The last group of activities is the sporadic ones. It is in this category the problem formulated belongs. Many of the techniques involve convolutional neural networks, to be able to segment the recording in different activities and is able to assess the whole sequence, using an offline algorithm. In online prediction, only previous data can be considered as well as the need for fast processing. Hammerla et al. (2016) proposes several methods, both for online and offline classification in HAR problems. Bulling et al. (2014) contains a comprehensive walk-through of different HAR problems using inertial sensors. This thesis will discuss several of the problems and techniques mentioned there.

Chapter 2

Machine Learning Theory

Machine learning is a sub-field of artificial intelligence, that utilizes data to make predictions and finding structures. (Raschka, 2017) Machine learning is commonly divided into two sub-groups, supervised learning and unsupervised learning. In supervised learning, the data is labeled so that the input variables are linked with an output variable. The purpose of supervised learning is to build a model that explains the relationship between the input- and output variables. In unsupervised learning there exist only input variables. The goal of unsupervised learning is to find hidden structures within the input data.

2.1 Supervised learning

When doing supervised learning it is normal to split the data into two sets, a training set, and a test set. This is visualized in figure 2.1. The training set is used to build a model and the test set is used to find out how well the model predicts unseen data.

The data can also be split into three categories, a training- and test set as before, as well as a validation set to tune the model before testing, visualized in figure 2.2. A reason for having a validation set is that it can be used to do hyperparameter tuning, without overfitting to the test set. The model is trained with several different parameters and the model with the highest accuracy on the validation set get selected. When training in this way, the validation set effects the model, even though the model is not trained on the validation set. This is due to the optimization towards good performance on the validation set, through tuning of hyperparameters. The validation set is also used to estimate the prediction error and the test set is used to asses the generalization error of the model. (Hastie, 2009)

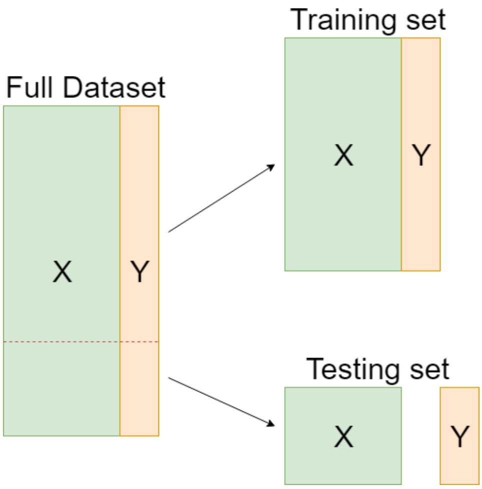


Figure 2.1: Example figure of split between training and test data.

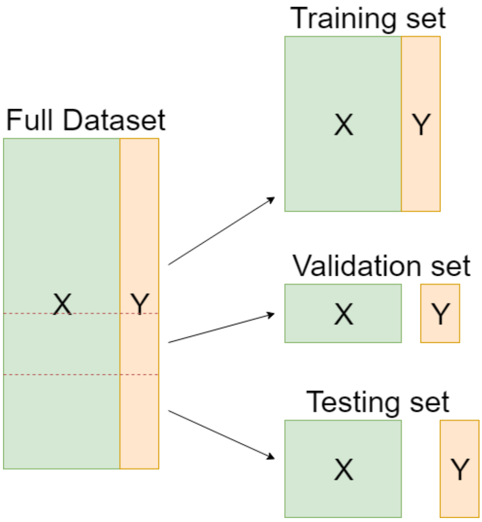


Figure 2.2: Example figure of split between training, validation and test data.

2.1.1 Loss function

The loss function is a function that is used to quantify how precise a machine learning models is, by stating how far from the truth the predictions it makes is. The loss function is a function of the parameters in the model, W and b. When the model is a regression model, i.e. it produces predictions with continuous values, the loss function is often defined as the squared difference between predicted values $\hat{y}$ and the true value y,

$$J(W, b) = \mathcal{L}(\hat{y}, y) = \frac{1}{2}(\hat{y} - y)^2 \tag{2.1}$$

With binary classification using logistic regression, the negative log-likelihood function is used,

$$J(W, b) = \mathcal{L}(\hat{y}, y) = -(y \log \hat{y} + (1 - y) \log(1 - \hat{y})) \tag{2.2}$$

For multi-class classification, with softmax regression over k classes, the cross-entropy loss is used,

$$J(W, b) = \mathcal{L}(\hat{y}, y) = -\sum_{j=1}^{k} y_j \log \hat{y}_j \tag{2.3}$$

Where y_j is a k-dimensional vector of zeros, except a 1 at the position of the true class. (Ng, 2013a)

2.1.2 Evaluation

The Bias-Variance Tradeoff

To evaluate how good the ML-model performs a sort of performance measure is needed. Assume that the underlying form form of the model is $Y = f(X) + \epsilon$, where ϵ is white noise, i.e. $E(\epsilon) = 0$ and $Var(\epsilon) = \sigma_\epsilon^2$. Having a model $\hat{f}(X)$ and using the squared-error loss, evaluated at $X = x_0$:

$$\begin{aligned} Err(x_0) &= E[(Y - \hat{f}(x_0))^2 | X = x_0] \\ &= \sigma_\epsilon^2 + [E\hat{f}(x_0) - f(x_0)]^2 + E[\hat{f}(x_0) - E\hat{f}(x_0)]^2 \\ &= \sigma_\epsilon^2 + Bias^2(\hat{f}(x_0)) + Var(\hat{f}(x_0)) \end{aligned} \tag{2.4}$$

The σ_ϵ^2 is the variance in the target Y and cannot be avoided unless $\sigma_\epsilon^2 = 0$. The bias is how much the average estimate differs from the true mean. The variance is the squared deviation of $\hat{f}(x_0)$ from its mean. (Hastie, 2009) This is a way of evaluating regression models. Evaluation with these metrics is often known as ”The Bias-Variance Tradeoff”. If the bias of the model is high, it means that the model is underfitted. This means that the model has not enough information from the training set, and thus it can not grasp the underlying model between x and y. If the variance is too big, the model could be overfitted. This means that the model is modeled too closely to the noise in the training set, and it will underperform on the test set. Normally reducing one of them will result in an increase in the other, that is the reason for calling it a tradeoff. (Le Calonnec, 2012)

Confusion Matrix

To evaluate classification a different approach must be used. A common way is to divide the results into a confusion matrix, as seen in table 2.1.

		Predicted Class	
		P’	**N’**
Actual Class	**P**	True Positive (TP)	False Negative (FN)
	N	False Positive (FP)	True Negative (TN)

Table 2.1: Confusion Matrix.

From the confusion matrix, the prediction error and accuracy can be calculated. The error is the ratio of wrong classified samples.

$$ERR = \frac{FP + FN}{FP + FN + TP + TN} \tag{2.5}$$

The accuracy is the ratio of correctly classified samples.

$$ACC = \frac{TP + TN}{FP + FN + TP + TN} = 1 - ERR \tag{2.6}$$

The precision is the rate of true positives among the positive predictions,

$$PRE = \frac{TP}{TP + FP} \tag{2.7}$$

The recall is the rate of true positives among the positive samples,

$$REC = \frac{TP}{TP + FN} \tag{2.8}$$

Often a combination of precision and recall is used, known as the F1-score,

$$F1 = 2\frac{PRE * REC}{PRE + REC} \tag{2.9}$$

Multi-class Confusion Matrix

When the classification is for multiple classes, the confusion matrix and the matrix changes a bit. Each prediction is matched up with the actual class of the sample, as seen in table 2.2.

		Predicted Class		
		A	**B**	**C**
Actual Class	**A**	M_{aa}	M_{ab}	M_{ac}
	B	M_{ba}	M_{bb}	M_{bc}
	C	M_{ca}	M_{cb}	M_{cc}

Table 2.2: Multi-class Confusion Matrix.

The average accuracy becomes,

$$ACC = \frac{\sum_i M_{ii}}{\sum_{i,j} M_{ij}} \tag{2.10}$$

The precision and recall can be defined in three ways; class independent, micro averaged and macro averaged.

The individual precision for a class i is,

$$PRE_i = \frac{TP_i}{TP_i + FP_i} = \frac{M_{ii}}{\sum_j M_{ji}} \tag{2.11}$$

and the recall is,

$$REC_i = \frac{TP_i}{TP_i + FN_i} = \frac{M_{ii}}{\sum_j M_{ij}} \tag{2.12}$$

when there is only one class per instance, the micro-averaged precision equals recall, since $\sum FP = \sum FN$, and the equation becomes,

$$PRE_\mu = REC_\mu = \frac{\sum_i TP_i}{\sum_i TP_i + FP_i} = \frac{\sum_i M_{ii}}{\sum_{i,j} M_{ij}} \tag{2.13}$$

which is identical to the average accuracy.

The macro average metrics is the sum of the individual class metrics, divided by the number of classes. For the macro-average precision with n classes, the equation becomes,

$$PRE_{macro} = \frac{\sum_i PRE_i}{n} = \frac{\sum_i \frac{TP_i}{TP_i + FP_i}}{n} \tag{2.14}$$

and the recall,

$$REC_{macro} = \frac{\sum_i REC_i}{n} = \frac{\sum_i \frac{TP_i}{TP_i + FN_i}}{n} \tag{2.15}$$

K-fold Cross Validation

K-fold cross validation is a method where the data is split into k different folds, without replacement. $K - 1$ folds are used to train the model and the last fold is to evaluate the performance of the model. This is done k times and makes k models and k estimations of the model performance. A visualization with white training folds and black test folds can be seen in figure 2.3.

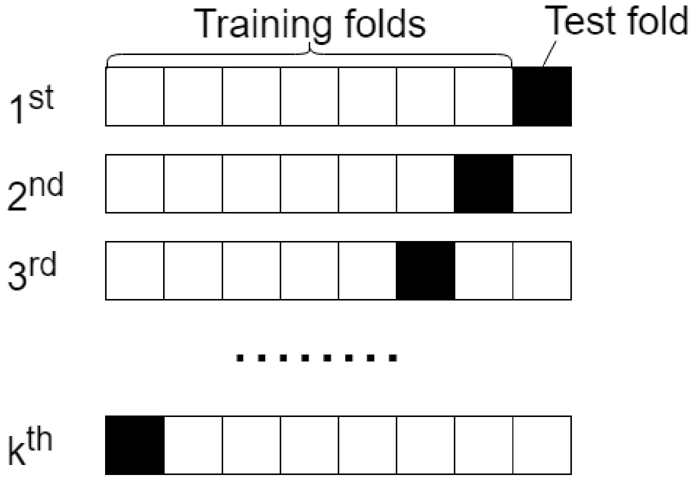


Figure 2.3: K-fold cross validation, visualized with $k = 8$ folds.

The case where $K = N$, i.e. the number of folds equals the number of samples, is known as leave-one-out cross validation. This is only used when the dataset is very small. (Hastie, 2009)

2.2 Neural Networks

The concept of a perceptron neural network is to have an input layer, hidden layers, and an output layer. The hidden layers consist of one or more perceptrons.

2.2.1 Perceptron

A perceptron consists of weights, a bias, and an activation function.

The inputs to the perceptron are given as a vector x. Each component is multiplied by its corresponding weight in the weight vector w, and summed up along with a constant

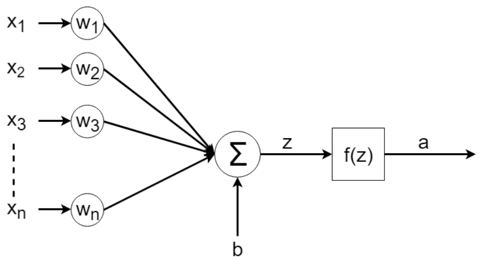


Figure 2.4: A perceptron layer, with input x, weights w, bias b, and activation function f.

bias b, to a scalar value z,

$$z = \sum_{i=1}^{n} w_i x_i + b = w^T x + b \tag{2.16}$$

The equation 2.16 combined with equation 2.18 is visualized in figure 2.4.

To simplify the notation and calculation of z, it is common to add a 1 as the first element in x and add the bias as the first element in w, often denoted as w_0. This gives,

$$z = \sum_{i=0}^{n} w_i x_i = w^T x \tag{2.17}$$

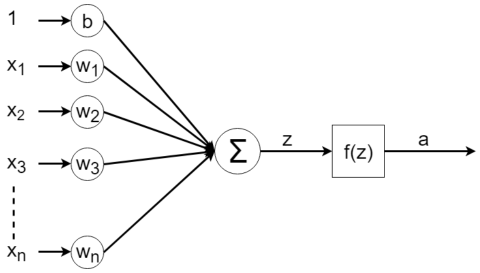


Figure 2.5: Perceptron layer with bias added to w.

The equation 2.17 combined with equation 2.18 is visualized in figure 2.5.

To produce the scalar output activation, a, of the perceptron an activation function f is applied to the sum z,

$$a = f(z) = f(w^T x) \tag{2.18}$$

Each hidden layer consists of one or more perceptron, where the output a is fed as input to the next layer. For each layer, all the activation outputs form an activation vector A. $A^{(0)} = x$ is defined to simplify the notation.

The sum of inputs in a hidden layer h is the activation from layer $h-1$ multiplied with the weight matrix $W^{(h)}$, which contain all the weights for each perceptron in layer h. This

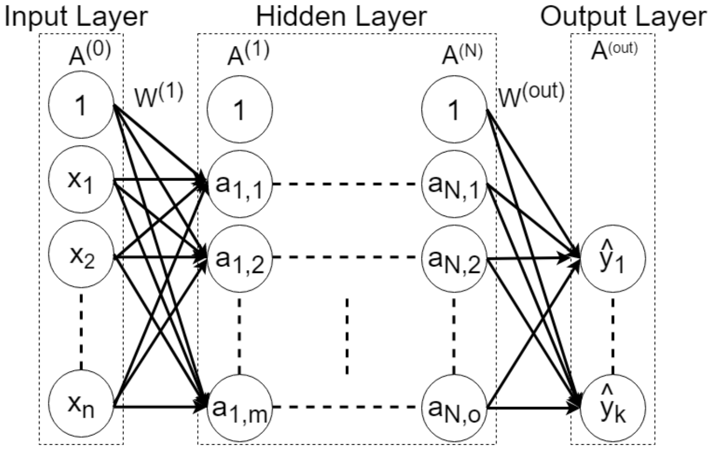


Figure 2.6: Multilayer Perceptron Neural Network.

forms a sum vector $Z^{(h)}$,

$$Z^{(h)} = A^{(h-1)}W^{(h)} \tag{2.19}$$

and the activation for the layer

$$A^{(h)} = f(Z^{(h)}) \tag{2.20}$$

this propagation trough layers is known as forward propagation. Together these layers form a Multilayer Perceptron neural network. The structure of MPL network and how the layers are connected is visualized in figure 2.6.

2.2.2 Backpropagtion

Backpropagation is a method for updating the weights and biases in a neural network.

First, define the error gradient, also called the sensitivity. (Mizutani et al., 2000)

$$\delta^{(h)} = \nabla_{z^{(h)}}\mathcal{L}(\hat{y}, y) \tag{2.21}$$

For the output layer $(N + 1)$ using the chain rule, the equation becomes

$$\delta^{(out)} = \nabla_{\hat{y}}\mathcal{L}(\hat{y}, y) \circ (f^{(out)})^{'}(z^{(out)}) \tag{2.22}$$

for the hidden layers $h = N, ..., 1$,

$$\delta^{(h)} = (W^{h+1}\delta^{(h+1)}) \circ f^{'}(z^{(h)}) \tag{2.23}$$

$f^{'}(z^{(h)})$ is the element wise derivation with regard to $z^{(h)}$. Ng (2013a) the gradients for layer h becomes,

$$\nabla_{W^{(h)}}J(W, b) = \delta^{(h)}a^{(h)T} \tag{2.24}$$

$$\nabla_{b^{(h)}}J(W, b) = \delta^{(h)} \tag{2.25}$$

For the update of the parameters, a few different techniques can be used. With stochastic gradient descent with a learning rate of α, the equation becomes,

$$W^{(h)} = W^{(h)} - \alpha \left[\nabla_{W^{(h)}} J(W, b) \right] \tag{2.26}$$

$$b^{(h)} = b^{(h)} - \alpha \left[\nabla_{b^{(h)}} J(W, b) \right] \tag{2.27}$$

The equations above are for backpropagation with a single training example. With more data, the update rule takes the average gradient. This yields these equations for m examples, known as batch gradient, (Ng, 2013b)

$$W^{(h)} = W^{(h)} - \alpha \left[\frac{1}{m} \sum_{i=1}^{m} \nabla_{W^{(h)}} J_i(W, b) \right] \tag{2.28}$$

$$b^{(h)} = b^{(h)} - \alpha \left[\frac{1}{m} \sum_{i=1}^{m} \nabla_{b^{(h)}} J_i(W, b) \right] \tag{2.29}$$

The different techniques to update the parameters is described in section 2.3.

2.2.3 Activation functions

An activation function is a function that is performed on the sum z in a perceptron. There are several functions, the most common functions is listed in table 2.3

Function	**Formula**
sigmoid	$f(x) = \frac{1}{1 + e^{-x}}$
tanh	$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
ReLU	$f(x) = max(0, x)$
Softmax	$f_i(x) = \frac{e^{x_i}}{\sum_{j=1}^{k} e^{x_j}}$

Table 2.3: Activation functions.

When doing multiclass classification, the output layer typically uses a softmax activation function to determine the probability for each class. The softmax function considers the layer as a whole and not just a single perceptron as the other activation functions.

2.2.4 Avoiding overfit

When the neural network has good performance on training data but does not generalize to the test data, it could mean that the network has overfitted to the training data. There exist several techniques to deal with these problems, among them is the dropout-layer for neural networks and regularization in the loss function.

Dropout

In the training phase, the dropout layer works by randomly dropping nodes in the coherent layer, with a probability p. How this effect the neural net can be seen in figure 2.7. In the testing phase, all activations are used, but they are reduced by a factor of p. This is done to accommodate for missing activations during the training phase. The activation of one node during training and testing is visualized in figure 2.8. The purpose of a dropout layer is to avoid over-fitting the model. (Srivastava et al., 2014)

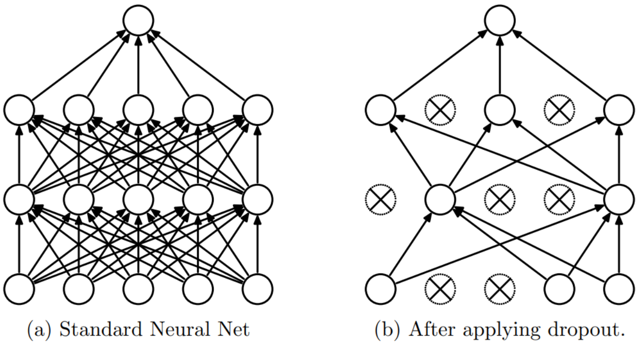


Figure 2.7: From (Srivastava et al., 2014): " Dropout Neural Net Model. **Left:** A standard neural net with 2 hidden layers. **Right:** An example of a thinned net produced by applying dropout to the network on the left. Crossed units have been dropped."

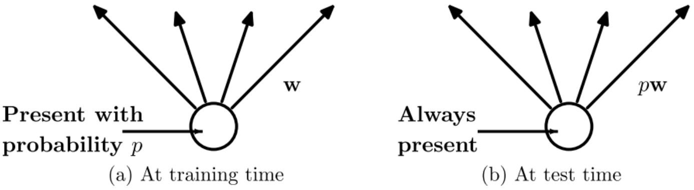


Figure 2.8: From (Srivastava et al., 2014): "**Left:** A unit at training time that is present with probability p and is connected to units in the next layer with weights **w**. **Right:** At test time, the unit is always present and the weights are multiplied by p. The output at test time is same as the expected output at training time."

Regularization

The concept of regularization is to avoid big dominant weights. A regularization term can be added to the loss function, often the L2-regularization term is used,

$$\lambda ||\mathbf{w}||^2 = \lambda \sum_{i=1}^{m} w_i^2 \tag{2.30}$$

The L2-regularization term is also known as L2 shrinkage or weight decay. The λ is the regularization parameter.

2.3 Gradient decent optimization algorithms

The algorithm for updating weights after backpropagation is known as gradient descent. There a several approaches to how much the gradient should influence the weights, as well as how many samples that are needed before an update can be done. To simplify the notation the parameters W and b will be denoted as θ.

2.3.1 Batch gradient decent

Batch gradient descent is a weighted update after all samples in the training set has been accessed. The weight is known as the learning rate. Batch gradient descent is described in equations 2.28 and 2.29.

2.3.2 Stochastic gradient decent

Stochastic gradient descent is similar to the batch gradient, but an update of the weights is performed after every sample assessment. It is as described in 2.26 and 2.27.

2.3.3 Mini-batch gradient decent

When there is a lot of training examples, doing gradient descent on every single example can be computationally heavy. To accommodate for this mini-batch gradient decent divides the training data up into batches, and does backpropagation and updates the parameters based on the batch.

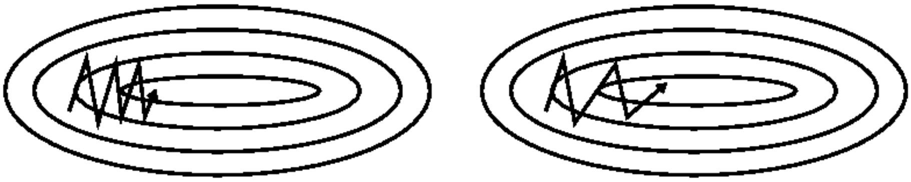
2.3.4 Momentum

The vanilla gradient descent algorithms have problems with slow convergence when the learning rate is too small. (Ruder, 2017)

It also struggles when the gradient is shifting around local optimums. To improve on this momentum is added, where a fraction of the previous update is kept in the new update.

$$\begin{aligned} v_t &= \gamma v_{t-1} + \alpha \nabla_\theta J(\theta) \\ \theta &= \theta - v_t \end{aligned} \tag{2.31}$$

γ is the momentum term and is typically set around 0.9. The effect of momentum in a 2D optimization problem can be seen in figure 2.9.



(a) Without momentum

(b) With momentum

Figure 2.9: Comparison of convergence to optimum, without momentum and with momentum. Source: Orr (1999).

2.3.5 Nesterov accelerated gradient

Instead of updating the momentum with the current gradient, the Nesterov accelerated gradient algorithm uses the gradient at the point where the current momentum would take new weight parameter. This behavior can be considered as a look ahead correction of the momentum.

$$\begin{aligned} v_t &= \gamma v_{t-1} + \alpha \nabla_\theta J(\theta - \gamma v_{t-1}) \\ \theta &= \theta - v_t \end{aligned} \tag{2.32}$$

2.3.6 Individual updates of parameters

When dealing with sparse data it can sometimes be hard to adjust the weights for infrequent classes. To accommodate this problem, some algorithms stores info from previous updates, so that an infrequent class gets more leverage on the model. These techniques also utilize some of the concepts mention earlier. These models include Adagrad, Adadelta, RMSprop, Adam, AdaMax, and Nadam, but these algorithms are beyond the scope of this thesis.

2.4 Other types of neural networks

2.4.1 CNN

Convolutional neural networks consist of weights which are used in a convolutional filter. This can only be used on data where the order in which features are stacked is non-trivial. This can be images, videos etc. The filters perform a convolution on a subset of the input corresponding to the filter size. This is done until the every input region is filtered. An example of how a convolutional neural network is applied to an image is seen in figure 2.10.

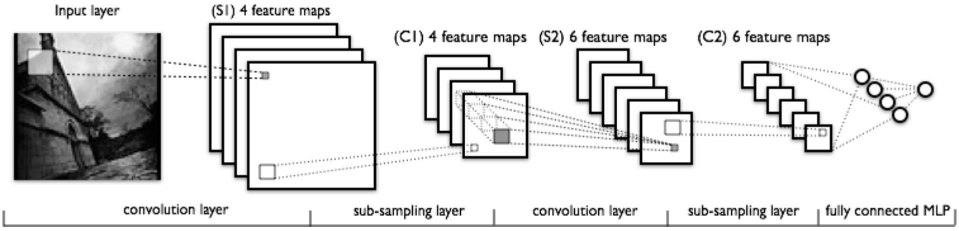


Figure 2.10: CNN example. S1 is the result after 4 different convolutional filters have been applied. The feature maps are then downsampled, into C1. How the next convolution is applied to give 6 feature maps can be in numerous ways, depending on the implementation. After a poling layer results to C2. C2 is then fed to a MLP network, which makes the prediction. Source: (CNNs).

Pooling

Pooling layers are used for down-sampling. This is done by selecting the max or mean within a local subsection. A typical size is 2x2 pooling and a stride of the pooling-filter of 2. (Chollet, 2017)

2.4.2 RNN

The NN models discussed so far has considered each data sample independently, and the order in which they appear has no influence on the result. However, for some data, the order can be very important, such as time-series, text, videos et al.

The sequential data can be split into different categories, depending on the form of the input and output.

The first one is many-to-one, where the input is a sequence, and the output is a class label. An example is sentiment analysis on sentences.

The second is one-to-many, where the input is fixed in size and the output is a sequence, an example is music generation.

The last is many-to-many. Here both the input and the output is sequences. This group can be further divided into two sub-groups, synchronized- and delayed many-to-many.

In the synchronized many-to-many, the input until time t corresponds to the output until time t. An example is speech recognition.

In the delayed version, however, the input and output do not necessarily follow the same order. There might be a difference in length as well. An example of delayed many-to-many sequential series is machine translation. (Raschka, 2017)

The difference between a standard NN and RNN is that in a RNN get information from the input and the hidden layer at the previous step. In the simplest form of RNNs, the information passed is the output of the layer. This is also called the state of the network. (Chollet, 2017) The equation for a RNN-cell with activation function f becomes

$$h^{(t)} = f\left(W_{hx} * x^{(t)} + W_{hh} * h^{(t-1)} + b_h\right) \tag{2.33}$$

where the W_{hx} matrix is the same as in MLP and W_{hh} is the weight matrix for the state h. A visualization of three simple RNN-nodes is seen in figure 2.11. Before the first step, the state is not defined. This initial state is often set to the zero matrix. (Chollet, 2017)

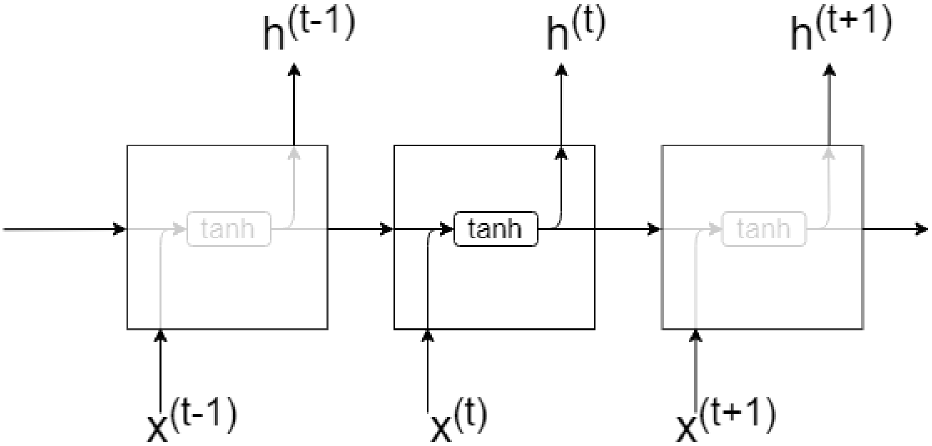


Figure 2.11: Visualization of a simple RNN structure.

A problem for RNN is to link some input that happens early and link it to a later event. This is known as the vanishing gradient problem. It is due to the nature of long-range dependencies. When doing backpropagation through time, BPTT, with a gradient update δw between two cells over a stretch of time t and T. δw will be multiplied with itself $T - t$ times. If δw is not 1, it will either become very small or very big. The former is known as the vanishing gradient problem, and the latter is known as the exploding gradient problem.(Bengio et al., 1994) The exploding gradient problem can be identified by infinite weight values in some of the weights in W_{hh}. The vanishing gradient problem, however, is more difficult to address since it causes only a very small update. (Ng, 2018)

Long Short-Term Memory

To deal with the vanishing gradient the Long Short-Term Memory method was introduced by Hochreiter and Schmidhuber (1997). A LSTM-cell is basically a memory cell. It takes the output from the previous cell $h^{(t-1)}$, the cell-state from the previous cell $c^{(t-1)}$ and the current input $x^{(t)}$. It uses different gates to determine how to update the cell-state and determine the new output. The forget gate is used to determine if the previous cell state shall be used,

$$f_t = \sigma\left(W_{xf}x^{(t)} + W_{hf}h^{(t-1)} + b_f\right) \tag{2.34}$$

this gate was not introduced in the original paper, but came later in Gers et al. (2002).

The input gate is used to update the cell state,

$$i_t = \sigma\left(W_{xi}x^{(t)} + W_{hi}h^{(t-1)} + b_i\right) \tag{2.35}$$

and a proposed cell state is made,

$$\tilde{c}^{(t)} = tanh\left(W_{xc}x^{(t)} + W_{hc}h^{(t-1)} + b_c\right) \tag{2.36}$$

the current cell state is computed as,

$$c^{(t)} = (i_t \odot \tilde{c}^{(t)}) + (f_t \odot c^{(t-1)}) \tag{2.37}$$

where $\odot$ is the element-wise product.
The output gate is used to update h,

$$o_t = \sigma\left(W_{xo}x^{(t)} + W_{ho}h^{(t-1)} + b_o\right) \tag{2.38}$$

and the new output becomes,

$$h^{(t)} = o_t \odot tanh(c^{(t)}) \tag{2.39}$$

A visualization of three LSTM-nodes is seen in figure 2.12.

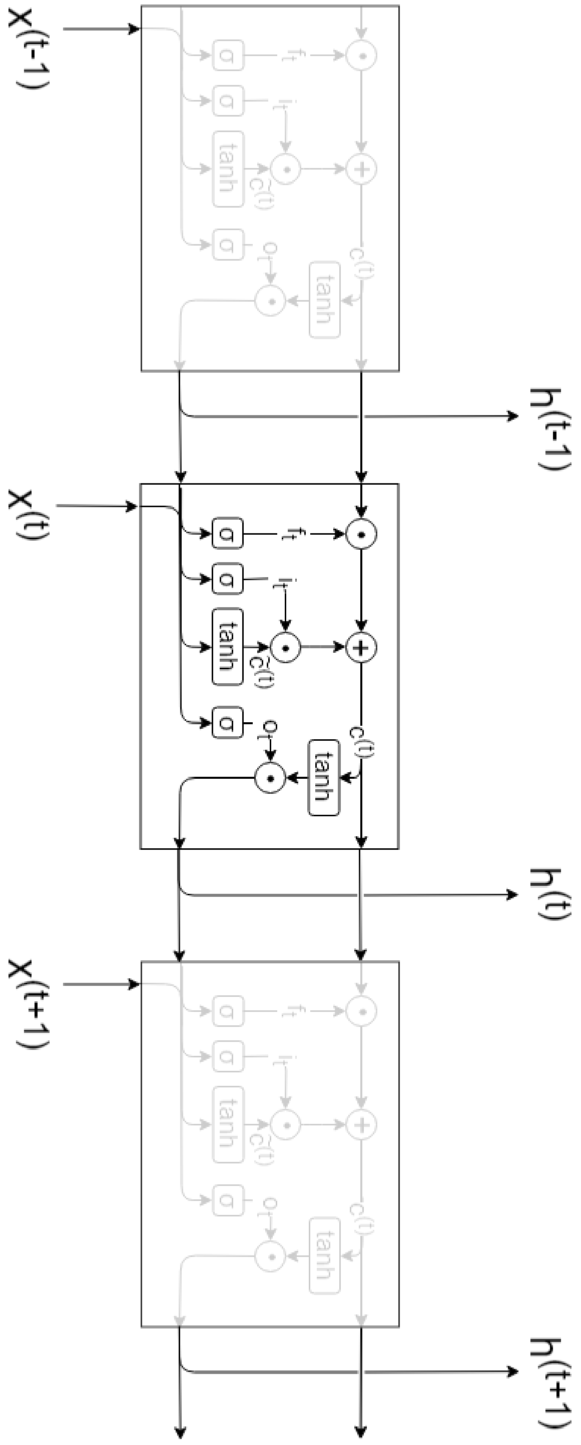


Figure 2.12: Visualization of a LSTM structure.

Gated Recurrent Unit

In 2014, Cho et al. (2014) introduced a simpler version of the LSTM. It only has two gates, the update gate, and the reset gate.

$$u_t = \sigma\left(W_{xu}x^{(t)} + W_{cu}c^{(t-1)} + b_u\right) \tag{2.40}$$

$$r_t = \sigma\left(W_{xr}x^{(t)} + W_{cr}c^{(t-1)} + b_r\right) \tag{2.41}$$

The reset gate is used in the proposal for the new cell state,

$$\tilde{c}^{(t)} = tanh\left(W_{xc}x^{(t)} + r_t \odot W_{cc}c^{(t-1)} + b_c\right) \tag{2.42}$$

The new cell is updated with the update gate,

$$c^{(t)} = (u_t \odot \tilde{c}^{(t)}) + ((1 - u_t) \odot c^{(t-1)}) \tag{2.43}$$

The output from GRU is simply the cell state,

$$h^{(t)} = c^{(t)} \tag{2.44}$$

A visualization of three GRU-nodes is seen in figure 2.13.

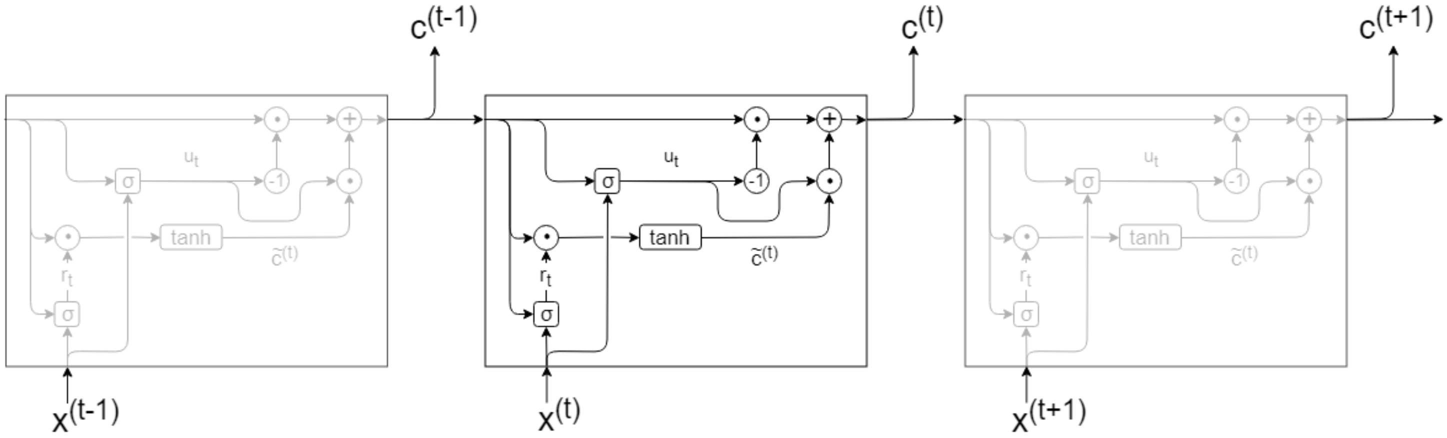


Figure 2.13: Visualization of a GRU structure

2.5 Hyperparameter tuning

Machine learning models consist of two types of parameters which describes how the structure of the model is. The first is normal parameters, where the parameter values are learned from data. In the context of deep learning, the normal parameters are called weights. All the parameters that are not updated from learning on data are called hyperparameters. This can be the number of layers in a neural network, the type of activation function, learning parameters, dropout and regularization rate and so on.

When doing hyperparameter tuning it is important to split the data into three sets, a training set, a validation set, and a test set. The training set is used to train the model, and the validation set is used to tune the hyperparameters. The test set is used to check that the model generalizes to unseen data as well.

Hyperparameters can be updated manually, given an experienced practitioner. There are some methods that can be used to optimize hyperparameters. Grid-search tests all combinations of the hyperparameters that shall be optimized. Since the computational overhead of training NNs can be large, this brute force method is mostly used for models that are not very computationally demanding. A method that is often used for NNs is random search. The hyperparameters are chosen randomly within predefined boundaries repeatedly. For each set of hyperparameters, the model is trained from scratch. The model with the best performance on the validation set is chosen in the end. Another approach is to use trees of Parzen estimators. (Bergstra et al., 2011) The field of hyperparameter optimization is still in the early stages and is expected to grow in importance in the years to come. (Chollet, 2017)

Chapter 3

Experiment

3.1 Dataset

Elliptic Labs provided a dataset containing 828 recording files, which together provided 10 415 248 sample points. Each sample point contains data from several sensors: accelerometer, gyroscope, infrared proximity, and ultrasound proximity data. The dataset contains recordings from several different users. The recordings contain data where the phone is either brought up to the ear in an ”answer call”-like fashion. Sequences with a motion to this position are considered as a positive recording. Other recordings contain sequences where the phone is kept at rest in the hand or on a table, considered as negative recording. The dataset contains recordings from several environments, indoor in an office, in public places such as a street or in a cafe, in vehicles such as cars, buses, trains, and aircraft. The recordings of the IMU-data have a sampling-period of 2.49 ms, which gives a sampling rate of roughly 402 Hz. The US data has a sampling rate of 10 Hz

In addition, Elliptic Labs provided an app to do more recordings. 52 new recordings have been made, containing 381 225 samples. These recordings have been made so that they address problematic scenarios detected during development.

The training data is the data provided by Elliptic Labs and the validation data is the new recordings. To test how well the model generalizes, the model will be tested on the device with an experiment, instead of using a test set. The app experiment is described in section 3.4.

3.1.1 Using ultrasound for proximity

The ultrasound data comes from a proximity algorithm provided by Elliptic Labs. To check how well the ultrasound proximity algorithm performs it is compared with the infrared proximity data, where infrared proximity is considered as ground truth. When the IR sensor gives negative output, the state is considered as Far. When the IR sensor gives positive output, the state is considered to be Near. The results are evaluated with the methods described in section 2.1.2.

		Ultrasound Proximity	
		P’	N’
Infrared Proximity	P	TP = 53.43%	FN = 8.83%
	N	FP = 0.15%	TN = 37.57%

Table 3.1: Confusion Matrix for Ultrasound- and Infrared Proximity data.

Metric	Result
Accuracy	0.910
Precision	0.997
Recall	0.858
F1	0.922

Table 3.2: Metrics for Ultrasound- and Infrared Proximity data.

The basis of figures for table 3.1 and table 3.2 is seen in appendix A, table A.1.

As seen in table 3.2, the ultrasound algorithm has very high precision and gives very reliable positive classifications. However, the ultrasound algorithm has quite a few false negatives. This gives a lower recall and is the main weakness of the classification.

In figure 3.1, it can often be seen that it has false negatives at the start of the IR positive sequence.

It is worth noting that the data in table 3.1 and table 3.2 is the data that the US classifier is tuned on.

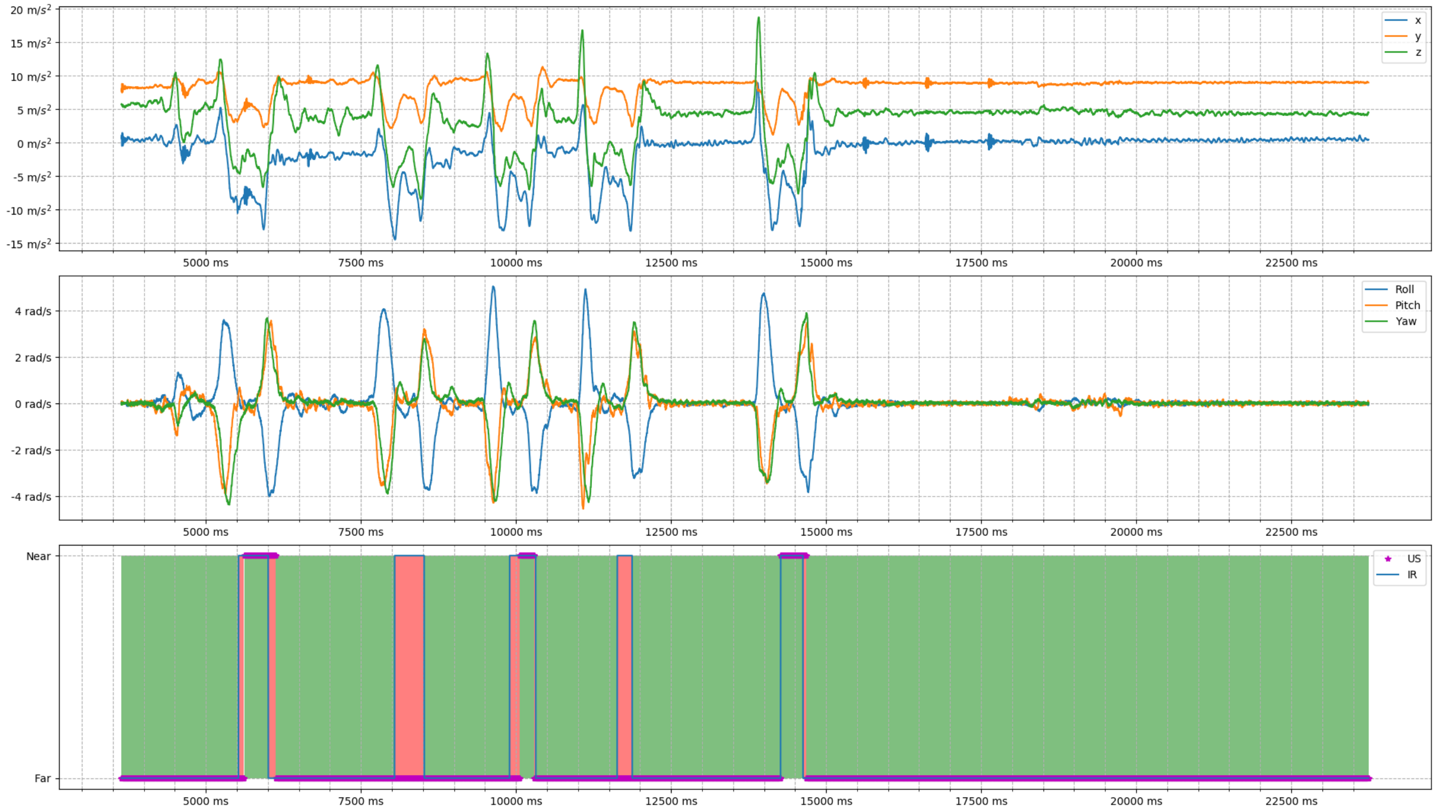


Figure 3.1: Accelerometer data, gyroscope data and IR and US combined. The green areas is where the IR data and US classification concur, and the red area is where they deviate.

Example used in plots

The example sequence used in the plots is one of the 52 new sequences. The sequence is seen in figure 3.1, figure 3.2, figure 4.1, figure 4.2, figure 4.3, figure 4.4, figure 4.5 and figure 4.6. It is selected because it contains several transitions, as well as it shows that the US classifier can be a bit slow, as well as it can have a too high threshold to produce a Near-event. The US classifier does not detect the Near-event at the second and fourth transition. At the third transition, the classifier reaches the Near 150 ms after the IR sensor. At first and the last transition the classifier is quite close to the IR sensor.

3.1.2 Altering the dataset to detect transitions between states

To be able to recognize the transition between the Far state and the Near state, a window of 200 ms was added before and after the transition, which was labeled as transition states in the dataset. They were called Far2Near and Near2Far. It is important to notice that most of the transition is longer than 200 ms, so the performance for these two classes is not expected to be as good as the two others. However, most of the behavior in the transition is captured within the 200 ms and it is with great certainty that there is a transition within the window in most cases. The alternatives would be to either label all the transitions by hand, or to use a segmentation algorithm to find the transitions.

The addition of leads to a quite unbalanced set, where the transition state has a lot fewer samples than Far and Near. To make sure that the transition states does not get neglected by the NNs, the training methods should consider this.

To accommodate for the imbalance in the set, a resampling technique could be used. By letting the classes with small sets appear more often than it normally would, the classes get more impact on the model. (Raschka, 2017) Another approach is to use a weighting of the classes. During training, the loss function will weight each error according to the class it belongs to. By giving a higher weight value to the small classes, these classes will have a bigger impact on the loss function, which again will give a bigger impact on the model.

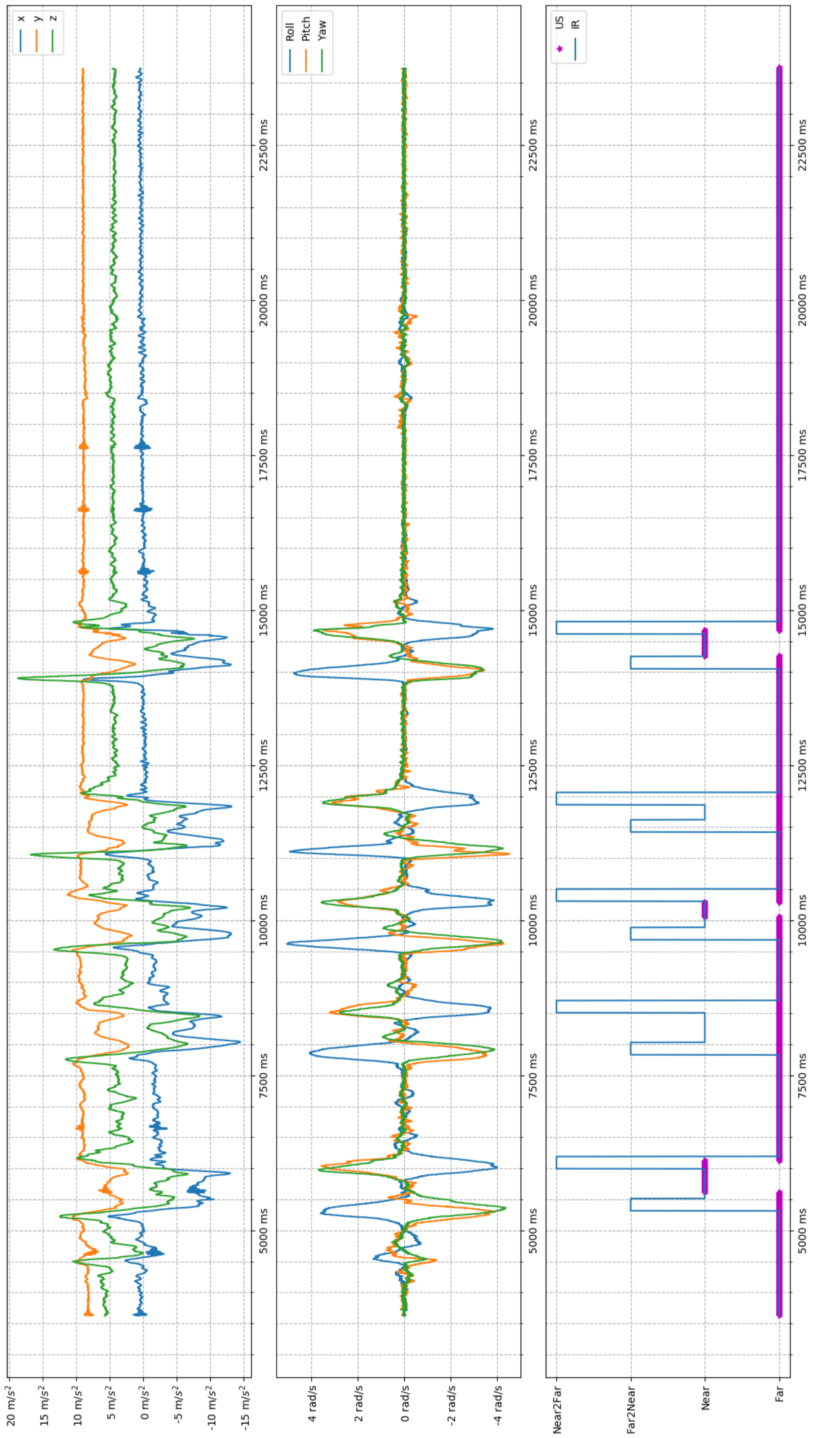


Figure 3.2: The example sequence, now with Far2Near and Near2Far states.

3.2 Neural Network Framework

3.2.1 Keras

Keras is an API for high-level neural networks development. It runs on top of other deep learning toolkits, such as TensorFlow, CNTK, and Theano. Keras is designed for fast prototyping, with a user-friendly interface that is designed to have a minimum of actions needed for common use cases. A wide range of network types is supported, including CNNs, RNNs, and combinations of the two. Keras is able to run seamlessly on both CPU and GPU. If the TensorFlow backend is chosen, it also supports TPU processing through Google Cloud. (Chollet and others, 2015)

3.2.2 TensorFlow

TensorFlow is an open-source machine learning framework. It was developed in Googles Google Brain project and was later released for public use. TensorFlow is available for Python, C++, Java and Go. It also has bindings for several other languages. However, the Python API is considered most mature. TensorFlow comes with a range of operations, such as mathematical- and matrix operations, NN building blocks and control flow operations for multi-device computation. (Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo et al., 2015)

3.3 Setup

The experiment is conducted by building models based on data collected by Elliptic Labs and data collected afterward with the device and interface provided by Elliptic Labs. The models are built with Keras, and the Keras-model is then ported to TensorFlow. Then it is launched on an Android device to do the experiment.

3.3.1 Device

The device provided by EllipticLabs is a Google Pixel, running Android 8.0.0(Oreo).

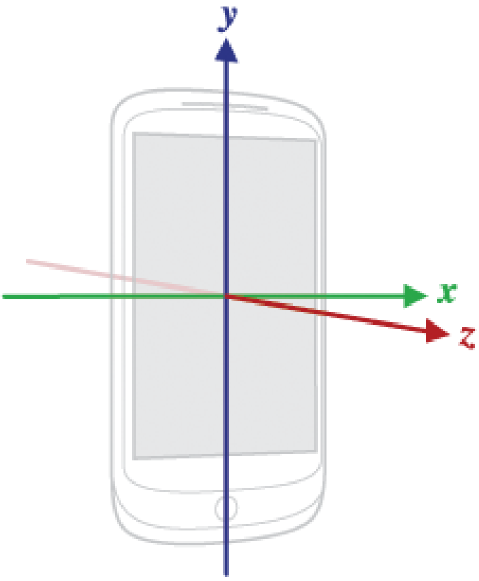


Figure 3.3: Reference axes on the Pixel device. Source: Google (2018).

3.3.2 Porting a model from Keras to Android

To be able to use a model that is trained in Keras, some actions need to be done. Android does not support running Keras models directly, but Android support running TensorFlow models. Since Keras runs TensorFlow in the backend, it is possible to get the TensorFlow session and retrieve the model from that session. However, some considerations need to be done before the model is runnable on an Android device.

First, the learning phase of the backend needs to be set to test. This is done with the following code,

```
from keras import backend as K
K.set_learning_phase(0)
```

Then the loaded model need to be retrieved using the current Tensorflow session, convert all variables in the model to constants and saved as a Protocol Buffer file.

```
sess = K.get_session()
constant_graph = graph_util.
    ↪ convert_variables_to_constants(sess, sess.graph.
    ↪ as_graph_def(), [output_node_name])
graph_io.write_graph(constant_graph, logdir,
    ↪ graph_file_name, as_text=False)
```

The Protocol Buffer file can be run on Android through the TensorFlowInferenceInterface class. The input is given to the model with the $feed()$ method, taking the name of the input node, the input data and the structure of the data as arguments to the method. The model is run with the $run()$ method, taking a list of output node names as argument. To get the output, the $fetch()$ method is run, taking the name of the node of interest and a list to save the output in as arguments.

3.3.3 Tuning of hyperparameters

To tune the hyperparameters of the model, Hyperas was used. Hyperas is is a wrapper for Hyperopt, which enables Keras models to be optimized with Hyperopt. Hyperopt is based on Bergstra et al. (2013) and supports two search algorithms, random search, and tree of Parzen estimators. The hyperparameters that were updated was the dropout-rate for each dropout-layer and the activation of the dense layers. When using RNN, the number of recurrent units was tuned as well.

3.3.4 Data generators

The neural networks were trained in mini-batches using Keras' $fit\_generator()$ method. Two different types of generator structures were used, depending on the needs for the network being trained. A generator is a function that yields data batches in an infinite loop. The first yields a random sample with a label, with a probability pre-defined by some label weight. The second generator yields samples in sequential order, as they appear in the recording.

When doing hyperparameter optimization with the random data generator, the model was trained with 10 epochs, each epoch ran for 5000 steps, where each step contains a batch of 32 samples. When the sequential generator is used, the number of steps equals the number of samples divided by the batch size.

3.4 Testing performance with app

To test how well the different neural networks perform, an app was made to be able to evaluate the output in real-time. This was done by exporting the NN model to an Android app, that took in accelerometer and gyroscope data, and fed it to the model. In the sensor fusion cases, ultrasound data was added as well through an API provided by Elliptic Labs. The app is based on the work in Valkov (2017) This is an app that takes in IMU data in 2000 ms batches and classifies the batch, by displaying the probabilities. The app that is used for the experiments is modified to do a classification after each sensor update, and update the color of the screen according to the classification every 100 ms.

The app will be used to evaluate how well the app generalizes, instead of using a separate test set. The reason behind this is that the app will show the real performance, whereas with a test set it is harder to interpret in which cases the model succeed and which cases it fails. The tests will contain several motion scenarios. The first is the motion of bringing the phone to the ear, both to the right- and left ear. This should be classified as a Near event, with coherent transition states. This motion is named Answer Call, and is the motion that the model is trained to recognize.

The second is to pick the phone up from the table and rotate the phone so one is able to look at it. This should be classified as a Far event. This motion is named Look Up. The reason for choosing to test this motion is that it is one of the other natural motions when the device is ringing.

The last scenario is to pick up the phone and rotate it forming a figure 8 pattern. This shall reveal if the model only uses the orientation of the phone to classify events. The motion is named Figure 8.

Each experiment trial will receive a positive mark if they predictions from the model are as described for the motion. If the predictions deviate from the description, the trial will receive a negative mark.

The marks will be summed up for each motion, and a total of 10 trials per motion will be performed.

3.5 Machine Learning Workflow

To be able to recognize changes over time, the dataset needs to be processed in such way that samples at different time steps can be compared.

3.5.1 Sliding window

When using a sliding window approach, each sample is altered to contain previous values. With a window of size T, the samples will contain the $T - 1$ previous values, as well as the current value. So at timestep t, the input to the network will be the samples from time t, till $t - T + 1$.

For example, if a window of 20 timesteps is used for a model that only takes IMU data. Each timestep contains 3 gyroscope values, 3 accelerometer values. The input to the model will contain $20 * (3 + 3) = 120$ values. If it is a sensor fusion model it would take data from the US classifier as well, so the input would be $20 * (3 + 3 + 1) = 140$ values.

3.5.2 Generator choice

Before training can begin it is important to consider the demands of the model regarding data generators. Some models require all the data to come in sequential order, and the sequential generator has to be used. Other models only require the data to be sequential within the time window given as input to the model.

3.5.3 Loss function

All of the models were trained by using the categorical cross entropy loss function.

3.5.4 Structure of the network

The size of the network, different activation functions, and different window sizes was explored before arriving on a final structure. The size of the window is 20 timesteps. The model has 3 dropout layers and 3 activation layers, in alternating order. The activation layers contain 64 nodes, 32 nodes, and 16 nodes, in the order in which they appear. The output layer consists of a softmax layer, with a number of nodes that corresponds to the number of classes.

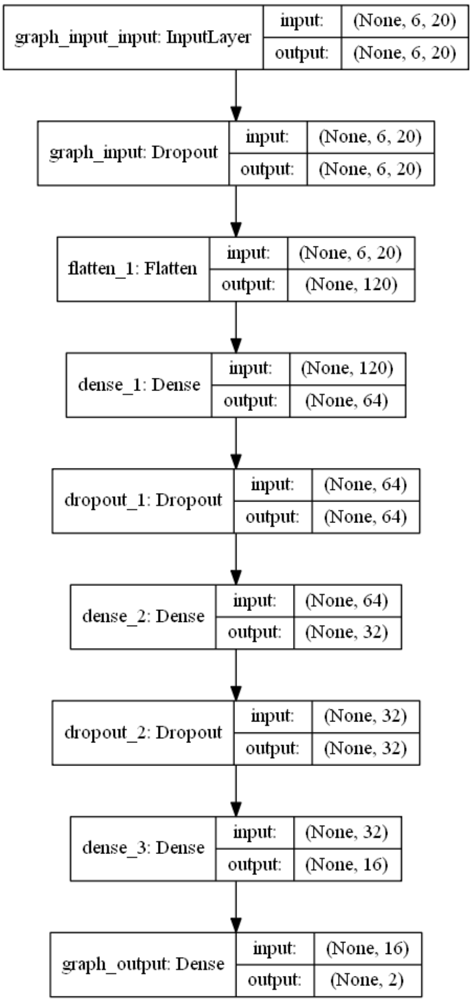


Figure 3.4: Visualization of the model structure.

Chapter 4

Results

The basis of figures for all tables in this section can be found in appendix A.

4.1 Using IMU data

The first models that were created used only inertial measurement data, i.e. accelerometer and gyroscope data.

4.1.1 Training with only Far and Near states

The first model did not use the transition states described in section 3.1.2. The input for the model was a sliding window with 20 timesteps, consisting of accelerometer and gyroscope data. For this model the three dropout layers where tuned with Hyperas. Dropout-rates between 0 and 1 were randomly selected, with a uniform distribution. The resulting dropout-rates can be seen in table 4.1. The activation function used in the dense layers was the sigmoid function. The prediction of this model on the example sequence is seen in figure 4.1.

Layer	graph_input	Dropout_1	Dropout_2
Dropout rate:	0.33	0.1	0.46

Table 4.1: Results from the hyperparameter tuning with only far and near states.

The basis of figures for table 4.2 and table 4.3 is seen in appendix A, table A.2.

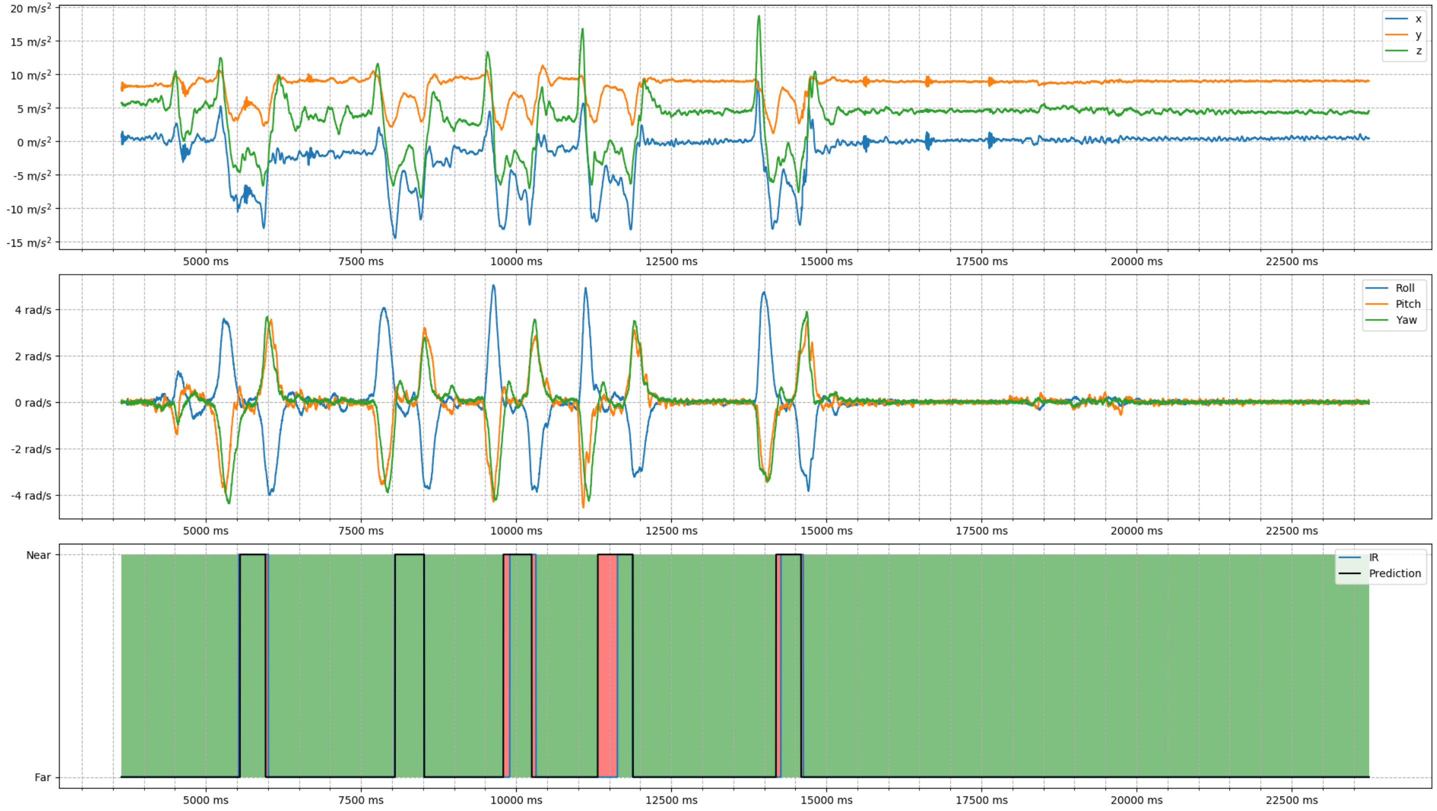


Figure 4.1: The example sequence, with predictions from far and near state model.

		Classification	
		Near’	**Far’**
Infrared Proximity	**Near**	TP = 43.84%	FN = 5.05%
	Far	FP = 2.59%	TN = 48.52%

Table 4.2: Confusion Matrix for model on far and near state model.

Metric	Result
Accuracy	0.92
Precision	0.94
Recall	0.90
F1	0.92

Table 4.3: Metrics for model on far and near state model.

Motion:	Positive	Negative
Transition	3	7
Look up	0	10
Figure 8	0	10

Table 4.4: Results from app experiment on far and near state model.

4.1.2 Training with transition states

This model used the same input as the previous model, but the output was altered to include transition states as described in section 3.1.2. When training with transition states, the performance metrics need to be altered in a suitable way for a multiclass classification problem. Since there is a class imbalance, with many more far and near events than transition events, the metrics should consider this as well.

		Classification			
		Far	**Near**	**Far2Near**	**Near2Far**
Actual Class	**Far**	41.73%	1.12%	1.81%	1.45%
	Near	6.29%	41.53%	0.39%	0.67%
	Far2Near	0.36%	1.17%	0.90%	0.08%
	Near2Far	0.35%	0.23%	0.06%	1.85%

Table 4.5: Confusion Matrix for model with transition states.

Dropout Layer	graph_input	Dropout_1	Dropout_2
Dropout rate:	0.27	0.1	0.31
Activation Layer	dense_1	dense_2	dense_3
Activation Function:	ReLU	ReLU	Sigmoid

Table 4.6: Results from the hyperparameter tuning with transition states.

The basis of figures for table 4.5 and table 4.7 is seen in appendix A, table A.3.

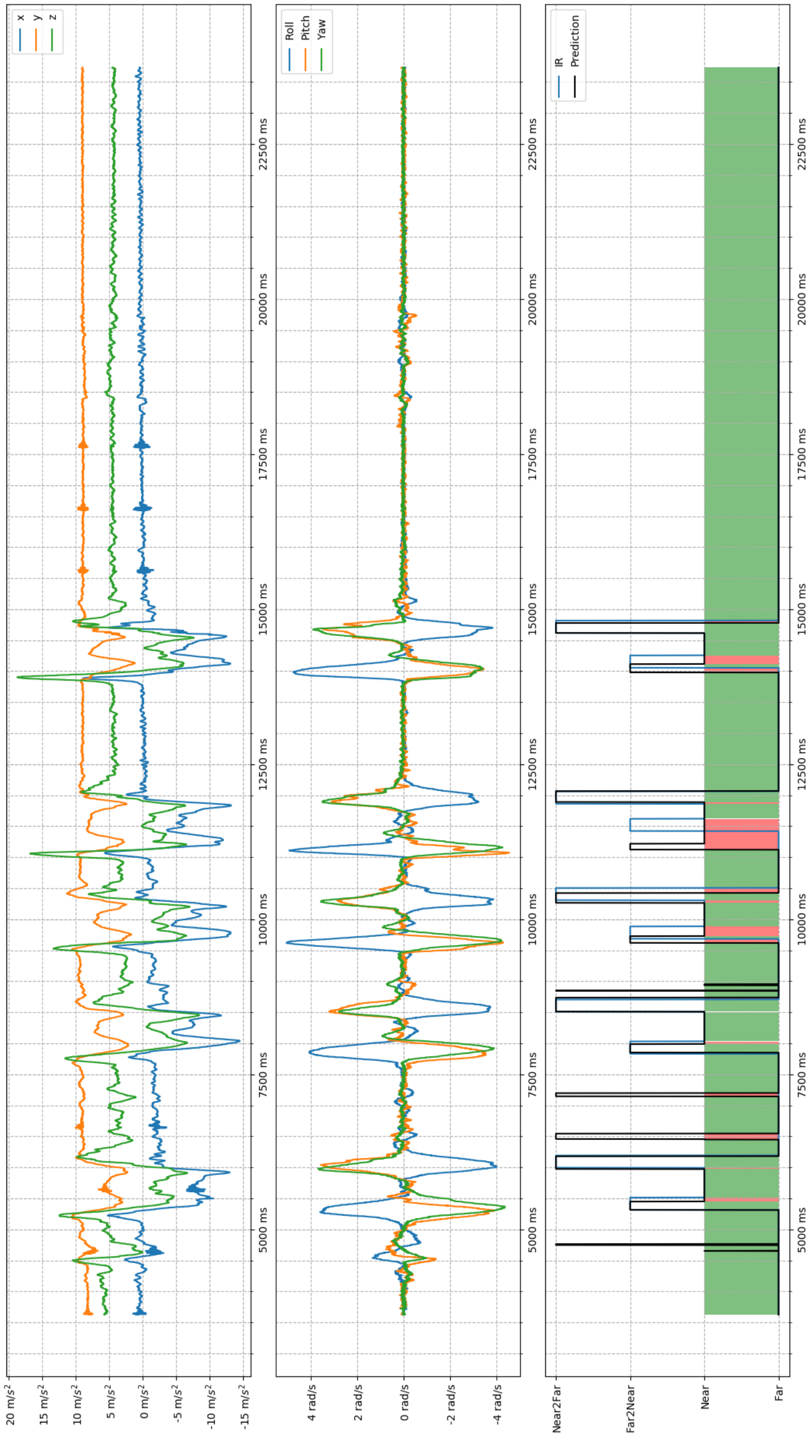


Figure 4.2: The example sequence, with predictions from model with transition states.

Metric	Macro-avg.	Far	Near	Far2Near	Near2Far
Accuracy	0.86				
Precision	0.64	0.86	0.94	0.29	0.46
Recall	0.71	0.91	0.85	0.36	0.74

Table 4.7: Metrics for model with transition states.

Motion:	Positive	Negative
Transition	9	1
Look up	10	0
Figure 8	6	4

Table 4.8: Results from app experiment on model with transition states.It is worth noting that the model gives transition states in Look up and Figure 8 motions, but no false Near states.

4.1.3 Using Gated Recurrent Units

When using GRU in the network, the samples are sent in one timestep at the time in sequential order, and the GRU keeps the previous samples in its memory, where the number of units decides how many timesteps that will be kept in memory. Because of this memory behavior, the sequential generator needs to be used. (Hammerla et al., 2016) The generator was run 325494 times per epoch, which is the number of samples divided by the batch size of 32.

The hyperparameters that were tuned was the number of units in the GRU-layer, the recurrent dropout between units in the GRU-layer, dropout layer rates and activation functions in dense layers.

The basis of figures for table 4.9 and table 4.10 is seen in appendix A, table A.4.

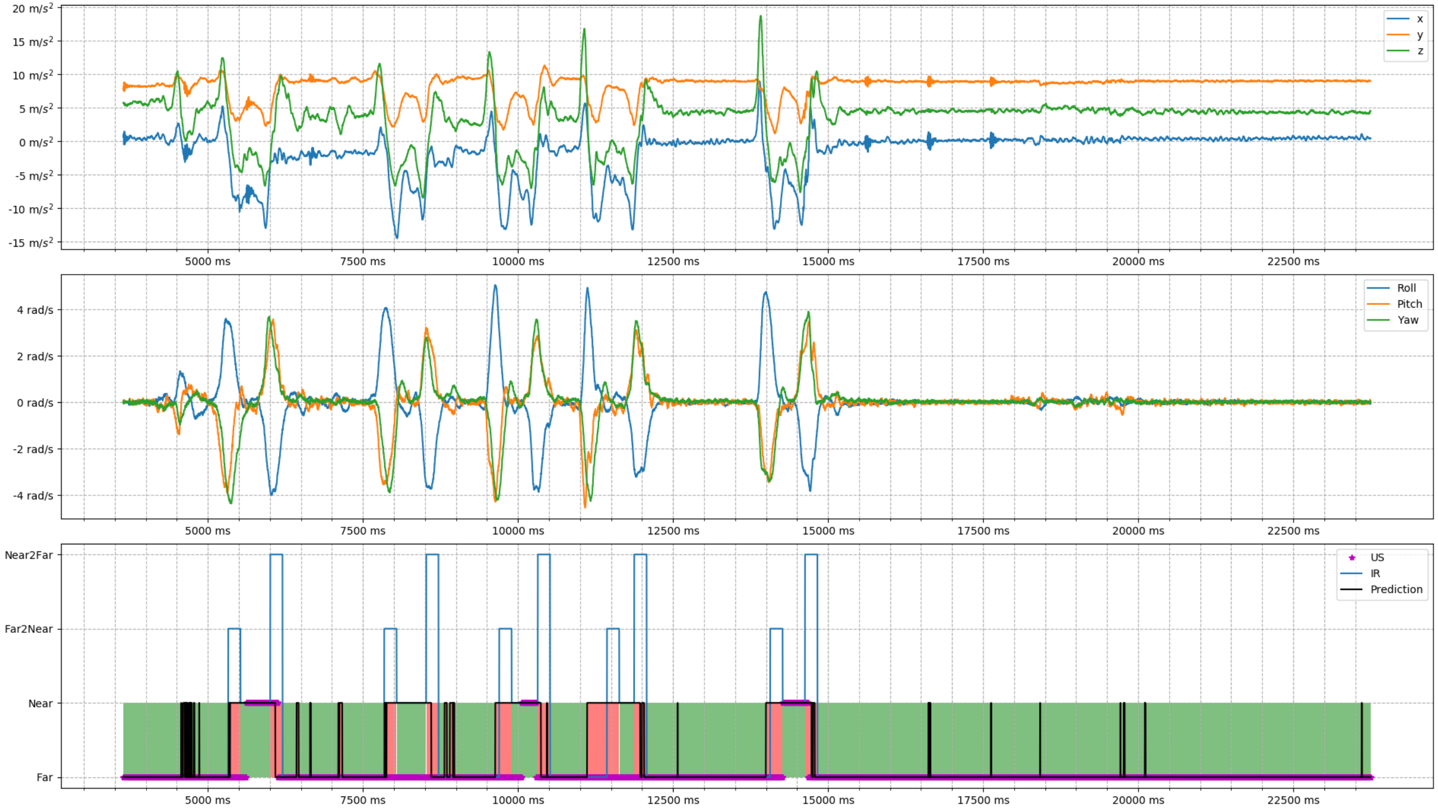


Figure 4.3: The example sequence, with predictions from GRU model.

		Classification			
		Far	Near	Far2Near	Near2Far
Actual Class	Far	38.34%	7.77%	0.00%	0.00%
	Near	0.69%	48.19%	0.00%	0.00%
	Far2Near	0.09%	2.43%	0.00%	0.00%
	Near2Far	0.61%	1.88%	0.00%	0.00%

Table 4.9: Confusion Matrix for GRU model.

Metric	Macro-avg.	Far	Near	Far2Near	Near2Far
Accuracy	0.86				
Precision	NaN	0.97	0.78	NaN	NaN
Recall	0.46	0.83	0.99	0	0

Table 4.10: Metrics for GRU model.

GRU units:	137		
Recurrent Dropout Rate:	0.22		
Dropout Layer	graph_input	Dropout_1	Dropout_2
Dropout rate:	0.02	0.85	0.61
Activation Layer	dense_1	dense_2	dense_3
Activation Function:	Sigmoid	ReLU	ReLU

Table 4.11: Results from the hyperparameter tuning of GRU model.

4.2 Sensor Fusion

To increase the performance of the models, the classification from the ultrasound was added as an input to the model.

For the models described in this section, the sliding window input was used. They had a window size of 20 timesteps. Each timestep had 3 gyroscope values, 3 accelerometer values and 1 value from the US classifier as input. This gives a total of 140 values as input.

4.2.1 Using known output pattern as heuristic

For the first few trials on making a sensor fusion model, it was observed that the model predicted the transition states in a somewhat good manner, however, it did not follow the expected cyclic structure in the output state. To try to improve this, a heuristic state machine was applied to the output to make sure it followed the correct state. The pseudo code to alter the output of the model can be seen in 4.1. Here $state$ is used as the output at each timestep.

Algorithm 4.1 Pseudo code for heuristic output state machine.

```
$\hat{y}$ is the proposed new state from the model
$state$ is the state currently held by the state machine
if $\hat{y} = state$ then
  $state$ unchanged
else
  if $\hat{y} = Far2Near$ and $state = Far$ then
    $state \leftarrow Far2Near$
  else if $\hat{y} = Near2Far$ and $state = Near$ then
    $state \leftarrow Near2Far$
  else if $state = Far2Near$ then
    $state \leftarrow Near$
  else if $state = Near2Far$ then
    $state \leftarrow Far$
  else
    $state$ unchanged
  end if
end if
```

The code only allows the state to change to a transition state if the transition corresponds to the current state. Also, the state that comes after the transition is set to the corresponding state of the transition, when the transition is done.

Original data

		Classification			
		Far	Near	Far2Near	Near2Far
Actual Class	Far	41.66%	0.91%	1.94%	1.60%
	Near	4.38%	43.49%	0.23%	0.79%
	Far2Near	0.85%	0.35%	1.23%	0.08%
	Near2Far	0.60%	0.16%	0.12%	1.62%

Table 4.12: Confusion Matrix for model without heuristic output correction.

Metric	Macro-avg.	Far	Near	Far2Near	Near2Far
Accuracy	0.88				
Precision	0.65	0.88	0.97	0.35	0.40
Recall	0.73	0.90	0.89	0.49	0.65

Table 4.13: Metrics for model without heuristic output correction.

The basis of figures for table 4.12 and table 4.13 is seen in appendix A, table A.5.

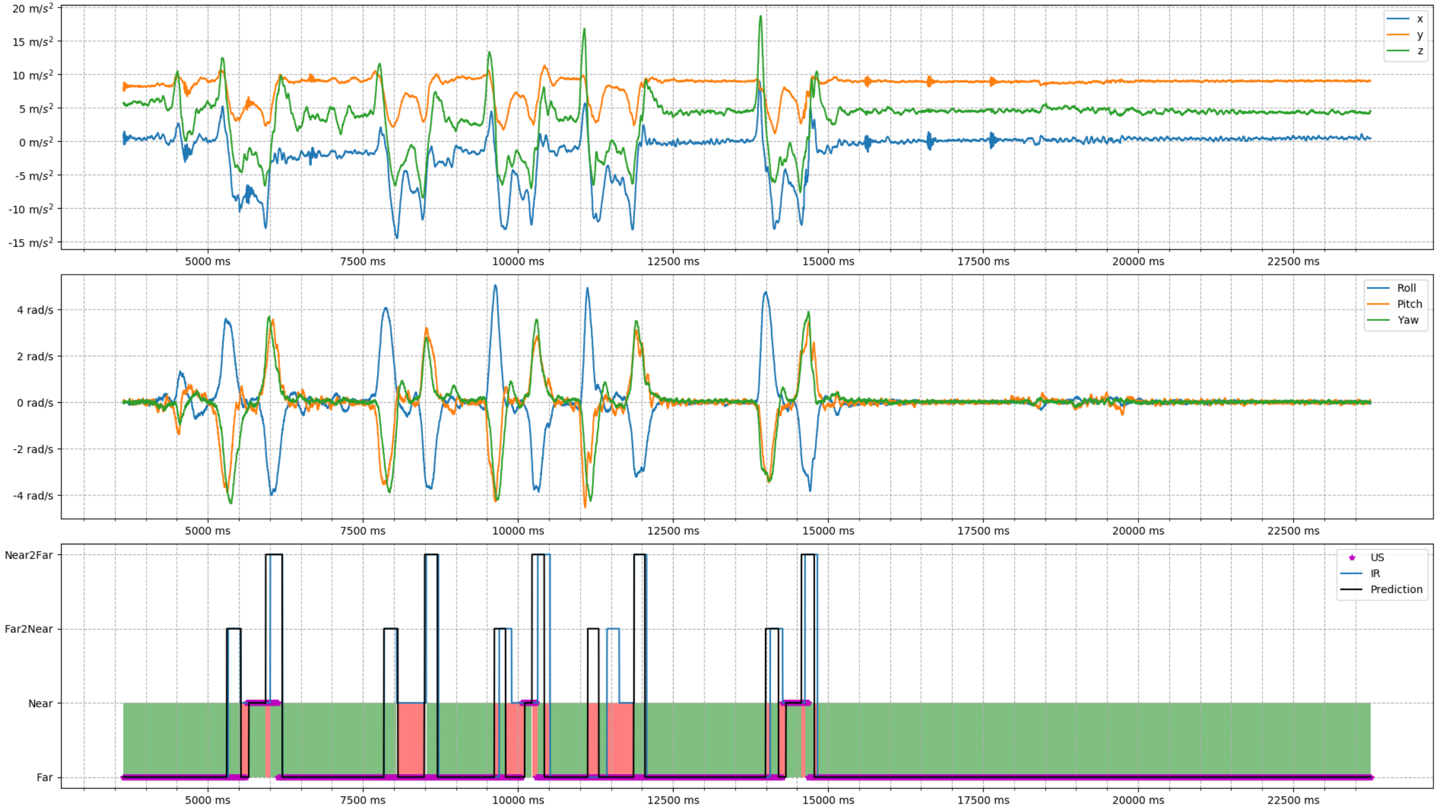


Figure 4.4: The example sequence, with a model without output correction.

Heuristic corrected data

		Classification			
		Far	**Near**	**Far2Near**	**Near2Far**
Actual Class	**Far**	38.96%	4.96%	1.71%	0.47
	Near	7.99%	39.99%	0.19%	0.71%
	Far2Near	0.31%	1.06%	1.10%	0.04%
	Near2Far	0.77%	0.25%	0.07%	1.40%

Table 4.14: Confusion Matrix for model with heuristic output correction.

Metric	Macro-avg.	Far	Near	Far2Near	Near2Far
Accuracy	0.82				
Precision	0.64	0.81	0.86	0.36	0.53
Recall	0.67	0.85	0.82	0.44	0.56

Table 4.15: Metrics for model without heuristic output correction.

The basis of figures for table 4.14 and table 4.15 is seen in appendix A, table A.6.

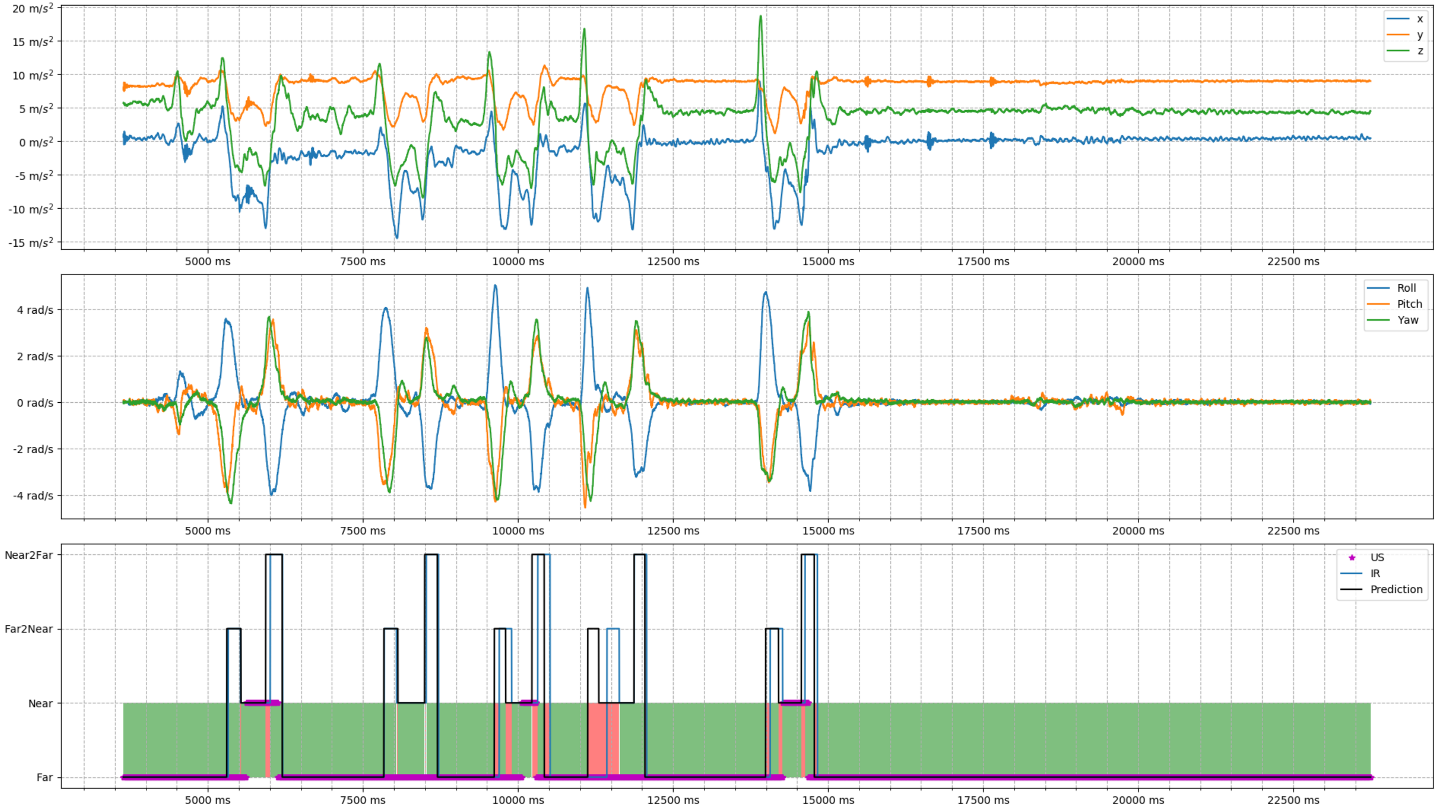


Figure 4.5: The example sequence, with a model with output correction.

Motion:	Positive	Negative
Transition	10	0
Look up	0	10
Figure 8	0	10

Table 4.16: Results from app experiment on model with heuristic state machine. For Look up and Figure 8 motions the model seems to get stuck in the near position.

4.2.2 Sensor fusion model optimized with hyperparameters

Since the output correction did not give better results, it was put out of further use. A model was trained with hyperparameter optimization to do sensor fusion.

		Classification			
		Far	**Near**	**Far2Near**	**Near2Far**
Actual Class	**Far**	44.52%	0.81%	0.68%	0.11%
	Near	1.77%	45.98%	0.23%	0.90%
	Far2Near	0.72%	0.48%	1.29%	0.03%
	Near2Far	1.24%	0.07%	0.00%	1.18%

Table 4.17: Confusion Matrix for sensor fusion model.

The basis of figures for table 4.17 and table 4.19 is seen in appendix A, table A.7.

This model was also tested on other persons. The persons was instructed to bring the device to the ear as if they were going to have a call. Each person did the experiment 10 times. The criteria to get a positive score is the same as for the transition movement. The results is seen in table 4.21.

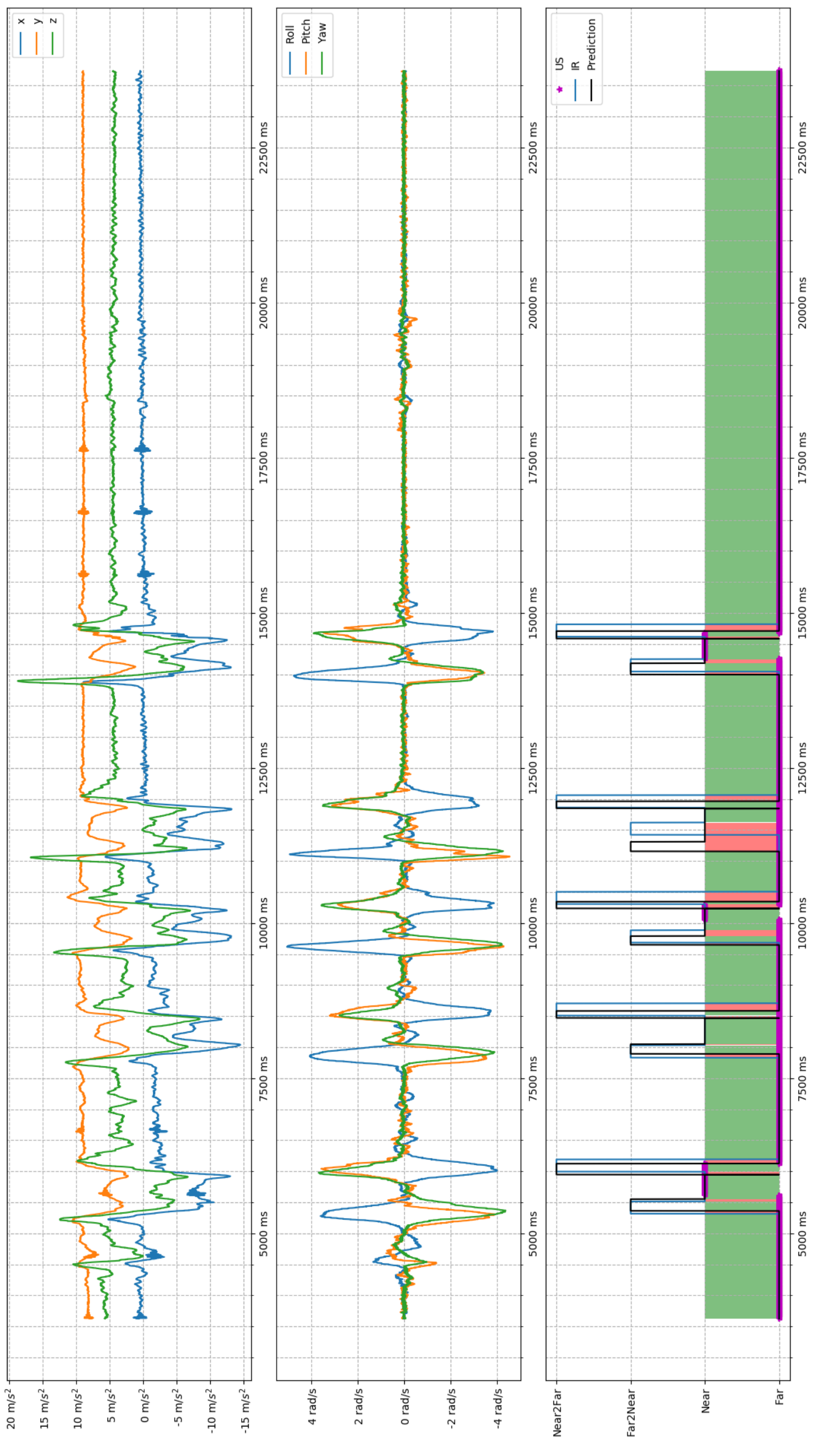


Figure 4.6: Senor fusion model predictions on the example sequence.

Dropout Layer	graph_input	Dropout_1	Dropout_2
Dropout rate:	0.77	0.26	0.66
Activation Layer	dense_1	dense_2	dense_3
Activation Function:	Sigmoid	Sigmoid	Sigmoid

Table 4.18: Results from the hyperparameter tuning for sensor fusion model.

Metric	Macro-avg.	Far	Near	Far2Near	Near2Far
Accuracy	0.93				
Precision	0.75	0.92	0.97	0.59	0.53
Recall	0.72	0.97	0.94	0.51	0.47

Table 4.19: Metrics for sensor fusion model.

Motion:	Positive	Negative
Transition	10	0
Look up	10	0
Figure 8	6	4

Table 4.20: Results from app experiment on sensor fusion model.

Test Subject:	Positive	Negative
Person 1	10	0
Person 2	9	1
Person 3	10	0

Table 4.21: Results from app experiment on different test subjects. It is worth noting that the model responded more slowly on Person 2.

Comparing Far2Near and Near2Far

To be able to see if the transition states can be separated based on IMU data, the subset of table 4.17 containing transition states will be evaluated.

		Predicted Class	
		Far2Near	**Near2Far**
Actual Class	**Far2Near**	51.63%	1.18%
	Near2Far	0.00%	47.19%

Table 4.22: Confusion Matrix for the transition states.

Metric	Results with Near2Far as positive	Results with Far2Near as positive
Accuracy	0.99	0.99
Precision	0.98	1.00
Recall	1.00	0.98
F1	0.99	0.99

Table 4.23: Metrics for transition state comparison.

Table 4.23 shows the metrics in two ways. One in which Near2Far is considered as the positive label, and one in which Far2Near is considered the positive label. Since there are not as many Near2Far predictions as Far2Near predictions, the set is not completely balanced, and the metrics are altered a bit depending on the choice of positive label. The basis of figures for table 4.22 and table 4.23 is seen in appendix A, table A.8.

4.3 Comparing Ultrasound and Sensor Fusion

4.3.1 Ultrasound performance on validation set

When applying the US classifier to the validation set, the resulting confusion matrix is seen in table 4.24 and metrics is seen in table 4.25. This is calculated to have a measure of the US classifiers performance on data it is not tuned on.

The basis of figures for table 4.24 and table 4.25 is seen in appendix A, table A.9.

		Predicted Class	
		Far	Near
Actual Class	Far	49.92%	5.22%
	Far	1.20%	43.67%

Table 4.24: Confusion Matrix for the ultrasound classifier on the validation set.

Metric	Result
Accuracy	0.94
Precision	0.97
Recall	0.89
F1	0.93

Table 4.25: Metrics for ultrasound on validation set.

4.3.2 Making comparable metrics

Since the sensor fusion model contains transition states, and the ultrasound classifier only contains Far and Near states, the data in table 4.17 should be altered to make the metrics comparable. The transition states is relabeled as Far for both the dataset and the prediction. The metrics for the sensor fusion model becomes,

Metric	Result
Accuracy	0.96
Precision	0.97
Recall	0.94
F1	0.96

Table 4.26: Metrics for sensor fusion, with transition states labeled as Far.

By comparing table 4.25 and table 4.26, it can be seen that the sensor fusion model has equal precision and an improved recall and accuracy.

Chapter 5

Discussion

In this chapter the results presented will be discussed. The overall strengths and weaknesses of the approaches will be evaluated, as well as how to improve on the topics discussed.

5.1 Orientation of the device

The major contributor in all of the models is the orientation of the phone. Most of the cases start with the phone in laying position, i.e. the gravitation is along the z-axis. When the phone is brought up to the ear the gravity is decomposed to all of the axes.

Results from the app experiment show that the transition states are present in look motion and in the Figure 8 motion. However, since there is no gravitational component along the x-axis in the look motion, the IMU only and sensor fusion model avoids misclassifying the look motion as a Near event. With the Figure 8 motion, the device is occasionally oriented in the same way as when holding the phone to the ear, leading to false Near classifications.

5.2 Usage of RNN

The RNN models trained yielded not so good results. The predictions had a tendency to flicker, as seen in figure 4.3. Also, the model never gave predictions of the transition state. This is seen in table 4.9. According to Hammerla et al. (2016): ”Movement data which was collected adhering to a fixed protocol will be less suitable to train RNNs, as long-term context brings little to no benefit over localized pattern matching during training and inference (and may even lead to overfitting).”. This could explain why the RNN underperform versus the sliding-window approach.

5.3 Combining IMU and US data

When combining IMU and US data the models often took one out of two approaches. The first was to still rely on the orientation of the phone and only be somewhat aided by the US classifier. This behavior can be seen for the model in section 4.2.2. The other approach was to just rely on the US classifier to detect Near states. This behavior can be seen for the model in section 4.2.1.

The US classifier gives a binary output that corresponds to the state it believes the device is in. By either giving a floating prediction output with the confidence for each class or using the US properties used by the US classifier directly, the NNs could define its own threshold, rather than relying on the threshold within the US classifier.

5.4 State machine to get cyclic output

The state machine proposed in section 4.2.1 did not improve the performance of the model. It relied heavily on correct and precise classification on the transition states, and even though it improved some corner cases where Near events were classified as Far, it did not justify long stretches of errors due to missed transitions.

5.5 User independent evaluation

The sensor fusion model from section 4.2.2 showed good performance when tested on different users, seen in table 4.21. Observations from the experiment supported the theory that the models rely on the orientation of the device for classification.

5.6 Labeling of transition states and assessment of ground truth

The labeling of the transition states was done by setting a window of 200 ms before each change from Far to Near and after each change from Near to Far. However, most transitions do not have a duration of exactly 200 ms. The IR sensor also does not change value at exactly the same time, depending on many different factors. It can be seen on the sequence plots that the IR sensor sometimes has a delay before recognizing a Near state. Often the IR sensor detects a Near state before the end of the motion, probably because the device is moved along the jaw, before reaching the ear. This behavior makes gives false positives in the evaluation of the predicted transition states, when they indeed are correct.

Usage of offline HAR algorithms could be used to get a better segmentation of the transition states.

5.7 Categorical cross-entropy as loss functions

With categorical cross-entropy as loss function, each misclassified sample is considered on its own and the time aspect of the sequences is neglected. Methods to evaluate performance of classification with respect to time is presented in Ward et al. (2011) and Ward et al. (2006). Here the activity recognition problem is considered a segmentation problem. This type of performance metric could be more suitable for a problem where the timing does not necessarily need to be exact, but the events should be classified reliably.

5.8 Separation of transition states

Most of the models that were made rarely misclassified the transition states for one another. Some of them had overfitted to transitions only to the right ear and when brought to the left ear the model would either mistake the transitions for one another or not recognize any transition at all. Several models show good performance on the transition states for motions to both ears, probably utilizing the gyroscope measurement present in the translations, in combination with a rapid change in the accelerometer data.

Chapter 6

Conclusion

This thesis has presented an infrastructure to train, deploy and validate models. Through the Hyperas interface, the model structure can be optimized for the problem at hand. Multiple approaches on how to input the data to the neural networks have been tested, and the sliding window approach was the one who yielded best results.

The results showed that the sensor fusion model was able to distinguish the approach and retract motion with an accuracy of 0.99. The accuracy of the model overall was raised from 0.94 when just using ultrasound for classification, to 0.96 when combining ultrasound with inertia data. This increase in accuracy was due to better recall, that went from 0.89 to 0.94 while leaving the precision unchanged.

Observations of model performance when running on an Android app revealed that the model relied heavily on the orientation of the device when doing classification. All of the models performance has been evaluated when running on the device, where different motions have been tested repeatedly. The sensor fusion model was tested on several persons and had little variance in output between the test subjects.

Bibliography

Bengio, Y., Simard, P., Frasconi, P., 1994. Learning Long-Term Dependencies with Gradient Descent is Difficult. IEEE Transactions on Neural Networks 5 (2), 157–166.
URL http://proceedings.mlr.press/v28/pascanu13.pdf?spm=5176.100239.blogcont292826.13.57KVN0&file=pascanu13.pdfhttp://arxiv.org/abs/1211.5063

Bergstra, J., Bardenet, R., Bengio, Y., Kégl, B., 2011. Algorithms for Hyper-Parameter Optimization. In: Advances in Neural Information Processing Systems (NIPS). pp. 2546–2554.
URL https://papers.nips.cc/paper/4443-algorithms-for-hyper-parameter-optimization.pdf

Bergstra, J., Yamins, D., Cox, D. D., 2013. Making a Science of Model Search: Hyperparameter Optimization in Hundreds of Dimensions for Vision Architectures. JMLR.

Bulling, A., Blanke, U., Schiele, B., 2014. A tutorial on human activity recognition using body-worn inertial sensors. ACM Computing Surveys (CSUR) 1 (June), 1–33.
URL http://dx.doi.org/10.1145/2499621http://dl.acm.org/citation.cfm?id=2499621

Cho, K., van Merrienboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., Bengio, Y., 2014. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation.
URL https://arxiv.org/pdf/1406.1078.pdfhttp://arxiv.org/abs/1406.1078

Chollet, F., 2017. Deep Learning with Python. Manning Publications Company.
URL https://books.google.no/books?id=Yo3CAQAACAAJ

Chollet, F., others, 2015. Keras.
URL https://keras.io

(CNNs), A. B. G. t. D. C. N. N., ???? A Beginner's Guide to Deep Convolutional Neural Networks (CNNs).
URL https://deeplearning4j.org/convolutionalnetwork

Gers, F. A., Eck, D., Schmidhuber, J., 2002. Applying LSTM to Time Series Predictable Through Time-Window Approaches. In: Tagliaferri, R., Marinaro, M. (Eds.), Neural Nets WIRN Vietri-01. Springer London, London, pp. 193–200.

Google, 2018. Android API Guide.
URL https://developer.android.com/guide/topics/sensors/sensors_overview

Hammerla, N. Y., Halloran, S., Ploetz, T., 2016. Deep, Convolutional, and Recurrent Models for Human Activity Recognition using Wearables. Ijcai, 1533–1540.
URL https://www.ijcai.org/Proceedings/16/Papers/220.pdfhttp://arxiv.org/abs/1604.08880

Hastie, T., 2009. The Elements of Statistical Learning : Data Mining, Inference, and Prediction.

Hochreiter, S., Schmidhuber, J., 1997. Long Short-term Memory. Neural computation 9, 1735–1780.

Le Calonnec, Y., 2012. CS229 Supplemental Lecture Notes - Variance and Error Analysis. Tech. rep.
URL http://cs229.stanford.edu/notes/cs229-notes-all/error-analysis.pdf

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Rafal Jozefowicz, Yangqing Jia, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Mike Schuster, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yu, Y., Zheng., X., 2015. TensorFlow: Large-scale machine learning on heterogeneous systems.
URL https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/45166.pdfhttp://download.tensorflow.org/paper/whitepaper2015.pdf

Mizutani, E., Dreyfus, S., Nishio, K., 2000. On derivation of MLP backpropagation from the Kelley-Bryson optimal-control gradient formula and its application. Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks. IJCNN 2000. Neural Computing: New Challenges and Perspectives for the New Millennium, 167–172.
URL http://queue.ieor.berkeley.edu/People/Faculty/dreyfus-pubs/ijcnn2k.pdfhttp://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=857892

Ng, A., 2013a. CS229: Supplemental notes 6- on Backpropagation. Course CS229: Machine Learning, 1–2.
URL http://cs229.stanford.edu/notes/cs229-notes-backprop.pdf

Ng, A., 2013b. CS299 notes Deep Learning. Course CS229: Machine Learning 1 (1), 1–3.
URL http://cs229.stanford.edu/notes/cs229-notes-deep_learning.pdfhttp://www.stanford.edu/class/cs229/

Ng, A., 2018. Sequence Models - Recurrent Neural Networks — Coursera.
URL https://www.coursera.org/learn/nlp-sequence-models/home/week/1

Orr, G. B., 1999. Momentum and Learning Rate Adaptation. Willamette University.
URL https://www.willamette.edu/~gorr/classes/cs449/momrate.html

Raschka, S., 2017. Phyton machine learning : machine Learning and deep Learning with Python, scikit-learn, and TensorFlow.

Ruder, S., 2017. An overview of gradient descent optimization algorithms *.
URL https://arxiv.org/pdf/1609.04747.pdf

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., Salakhutdinov, R., 2014. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. Journal of Machine Learning Research 15, 1929–1958.
URL https://www.cs.toronto.edu/~hinton/absps/JMLRdropout.pdf

Valkov, V., 2017. Human Activity Recognition using LSTMs on Android.
URL https://medium.com/@curiousily/human-activity-recognition-using-lstms-on-android-tensorflow-for-hackers-part-vi-492da5adef64

Ward, J. A., Lukowicz, P., Gellersen, H., 2011. Performance metrics for activity recognition. ACM Transactions on Intelligent Systems and Technology (TIST) 2 (1), 1–23.
URL http://doi.acm.org/10.1145/1889681.1889687http://dx.doi.org/10.1145/1889681.1889687

Ward, J. A., Lukowicz, P., Tröster, G., 2006. Evaluating Performance in Continuous Context Recognition Using Event-Driven Error Characterisation. In: Hazas, M., Krumm, J., Strang, T. (Eds.), Location- and Context-Awareness. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 239–255.

Appendix

Appendix A

Basis of Figures

		Ultrasound Proximity	
		P’	N’
Infrared Proximity	P	TP = 5 565 728	FN = 919 878
	N	FP = 16 299	TN = 3 913 343

Table A.1: Basis of figures for table 3.1.

		Classification	
		Near’	**Far’**
Infrared Proximity	**Near**	TP = 145 399	FN = 16 740
	Far	FP = 8 589	TN = 160 968

Table A.2: Basis of figures for table 4.2.

			Classification		
		Far	**Near**	**Far2Near**	**Near2Far**
Actual Class	**Far**	138417	3725	5999	4799
	Near	20861	137753	1289	2236
	Far2Near	1185	3895	3000	269
	Near2Far	1150	762	211	6145

Table A.3: Basis of figures for table 4.5.

			Classification		
		Far	Near	Far2Near	Near2Far
Actual Class	Far	161841	32188	0	0
	Near	2306	171055	0	0
	Far2Near	557	9131	0	0
	Near2Far	2722	6883	0	0

Table A.4: Basis of figures for table 4.9.

			Classification		
		Far	**Near**	**Far2Near**	**Near2Far**
Actual Class	**Far**	138197	3032	6419	5292
	Near	14523	144249	749	2618
	Far2Near	2825	1152	4100	272
	Near2Far	1978	527	389	5374

Table A.5: Basis of figures for table 4.12.

			Classification		
		Far	Near	Far2Near	Near2Far
Actual Class	Far	129241	16465	5678	1556
	Near	26512	132645	623	2359
	Far2Near	1030	3522	3661	136
	Near2Far	2546	835	242	4645

Table A.6: Basis of figures for table 4.14.

		Classification			
		Far	Near	Far2Near	Near2Far
Actual Class	Far	147668	2673	2239	360
	Near	5870	152509	777	2983
	Far2Near	2384	1595	4272	98
	Near2Far	4123	240	0	3905

Table A.7: Basis of figures for table 4.17

		Predicted Class	
		Far2Near	Near2Far
Actual Class	Far2Near	4272	98
	Near2Far	0	3905

Table A.8: Basis of figures for table 4.22.

		Predicted Class	
		Far	Near
Actual Class	Far	165592	17303
	Far	3965	144836

Table A.9: Basis of figures for table 4.24.