Contents

[1 Creating labelled training & test data 2](#_Toc508542353)

[1.1 Collecting the data 2](#_Toc508542354)

[1.2 Computing the touch location(s) for *grounded touches* 2](#_Toc508542355)

[1.3 Adding self sensing data 4](#_Toc508542356)

[1.4 Creating a labelled training set 4](#_Toc508542357)

[1.5 Data augmentation 6](#_Toc508542358)

[2 Creating and training the CNN model 7](#_Toc508542359)

[2.1 The CNN architecture 7](#_Toc508542360)

[2.2 Training the model 9](#_Toc508542361)

[2.3 The detection model: non-max suppression 12](#_Toc508542362)

[3 Evaluation of the trained model on more difficult cases 13](#_Toc508542363)

[3.1 Five fingers – grounded touch (*Note: with 2nd version*) 13](#_Toc508542364)

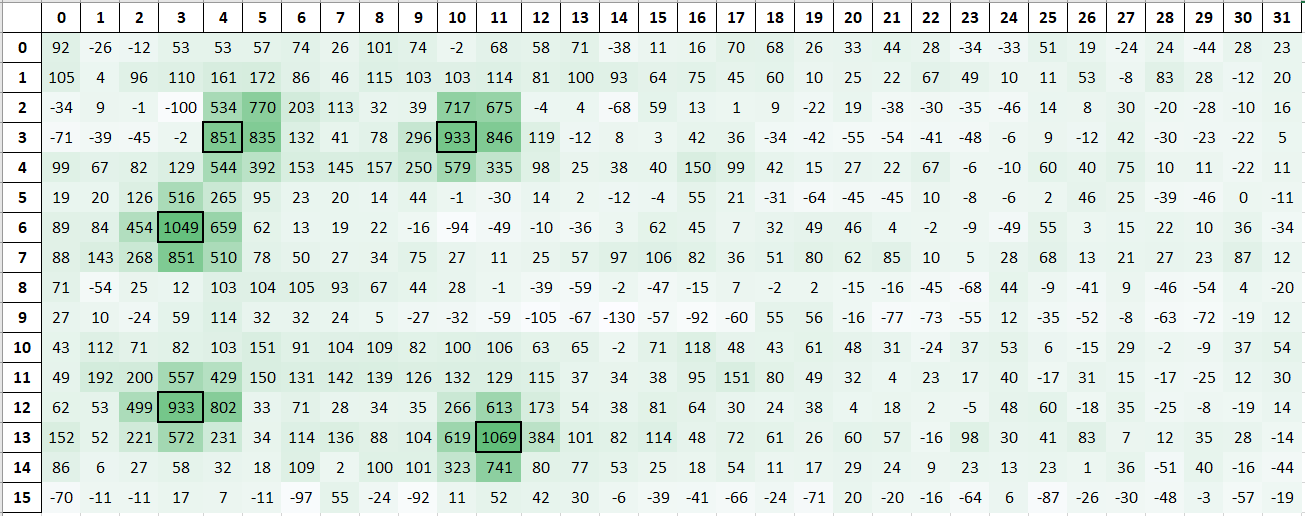
[3.2 Five fingers – floating touch (*Note: with 5th version*) 13](#_Toc508542365)

# Creating labelled training & test data

## Collecting the data

Training is done based on ***mutual*** *and* ***self******strength*** *data* (*baseline is subtracted* for model accuracy) with **multiple touches**. A total of 2705 **strength** frames were used (*2-5 fingers, no baseline subtraction, zebra display noise, typical AFE settings, multiple locations, thumb*). Both ***grounded*** *and* ***floating*** *touch* condition were recorded: 1943 grounded and 762 floating.

An example of mutual data is shown below. Note the data size: **(16, 32)**



A Matlab script will browse the data folders (for different conditions), read all Excel files (as saved by the GUI), extract all raw data matrices and finally, merge and save them into a new file as a **(2705, 512)** matrix (rows = *number of data*, columns = *flattened raw data; 16 \* 32 total*).

**Improvement areas:**

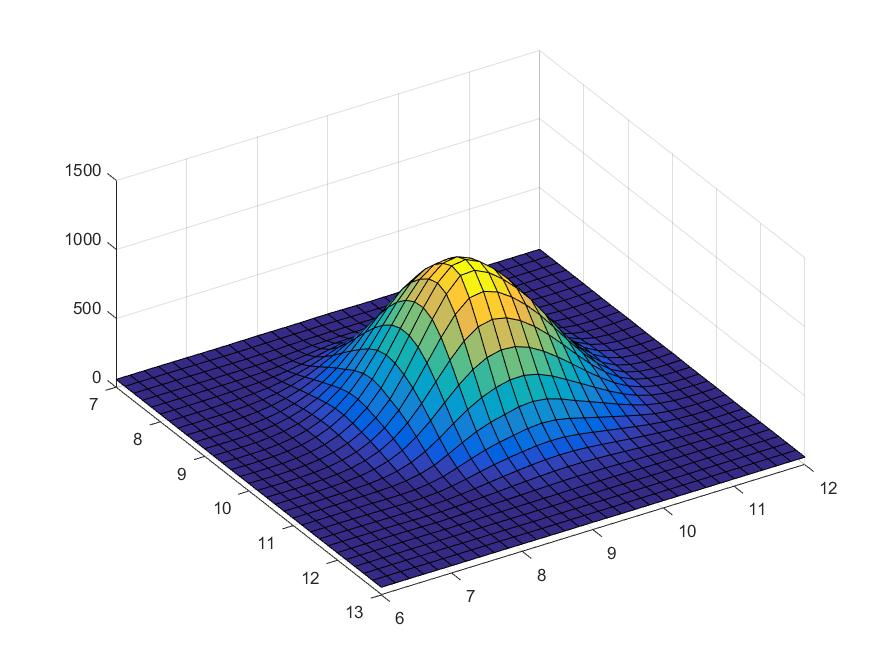
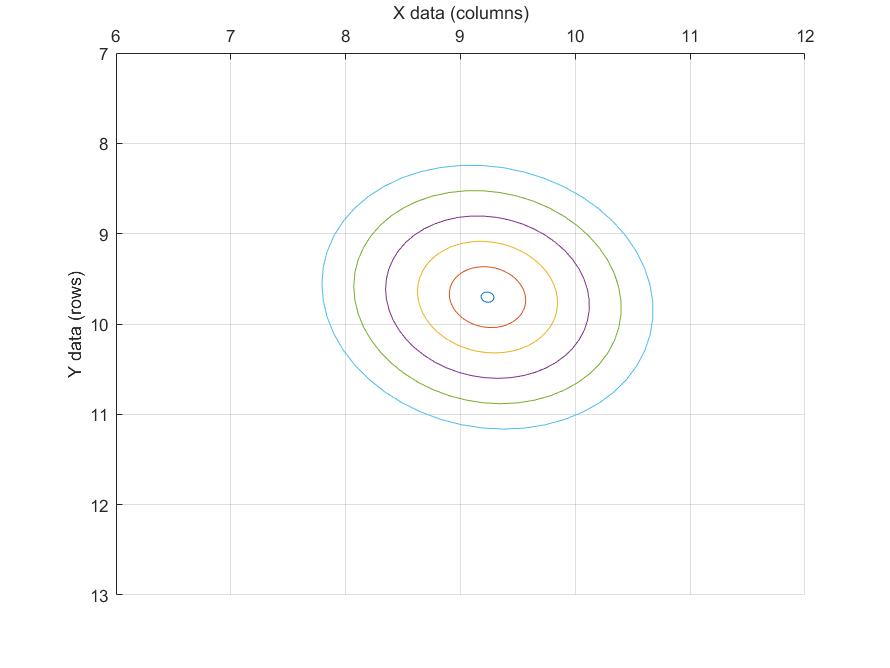
1. Identify all difficult scenarios (e.g. *charger noise, zebra display noise, low accumulations, shipment/testing noise, baseline drift, water condition,* etc.) and **collect data for each scenario**
2. Redo 1) for **differential mode** to train the neural network directly based on it (no integration).

## Computing the touch location(s) for *grounded touches*

For *grounded touches*, a Matlab algorithm is used to label the data (*floating touch coordinates* are much harder to compute by an algorithm, due to the irregular mutual data, so they are processed ***manually***!). Knowing the number of touches, the peak (touch) locations are identified and a matrix is chosen around each of them. Depending on the relative distance between the touches (in order to avoid 2 touches in the same window), the window size is chosen as (3, 3), (5, 5) or (7, 7). An example of a (7, 7) window is shown below:



The matrix values are interpreted as Z values at the intersections of rows and columns (force and sense lines in the actual application). The **coordinates** are computed relatively to the **row/column number**:

The algorithm will then try to fit a 2D gaussian (see below figures):

Extracted values:

## Adding self sensing data

*Self sensing* data is an *additional input information* which mostly helps with *classification in floating conditio*n (e.g. differentiating big thumb from 2 adjacent touches or differentiating touch from large water drops), but also with *regression* (e.g. coordinate computation when mutual data is very low in floating condition due to fingers present on the same line).

In self mode, we obtain two 1D arrays: *Self Sense* (SS) of shape (1, 32) and *Self Force* (SF) of shape (1, 16). From them we can generate the self data matrix (input channel X1 of shape (16, 32)) as the average of the Self Sense and Self Force values:

An example is shown below for the case of 3 floating touches (marked by square contours in the matrix):



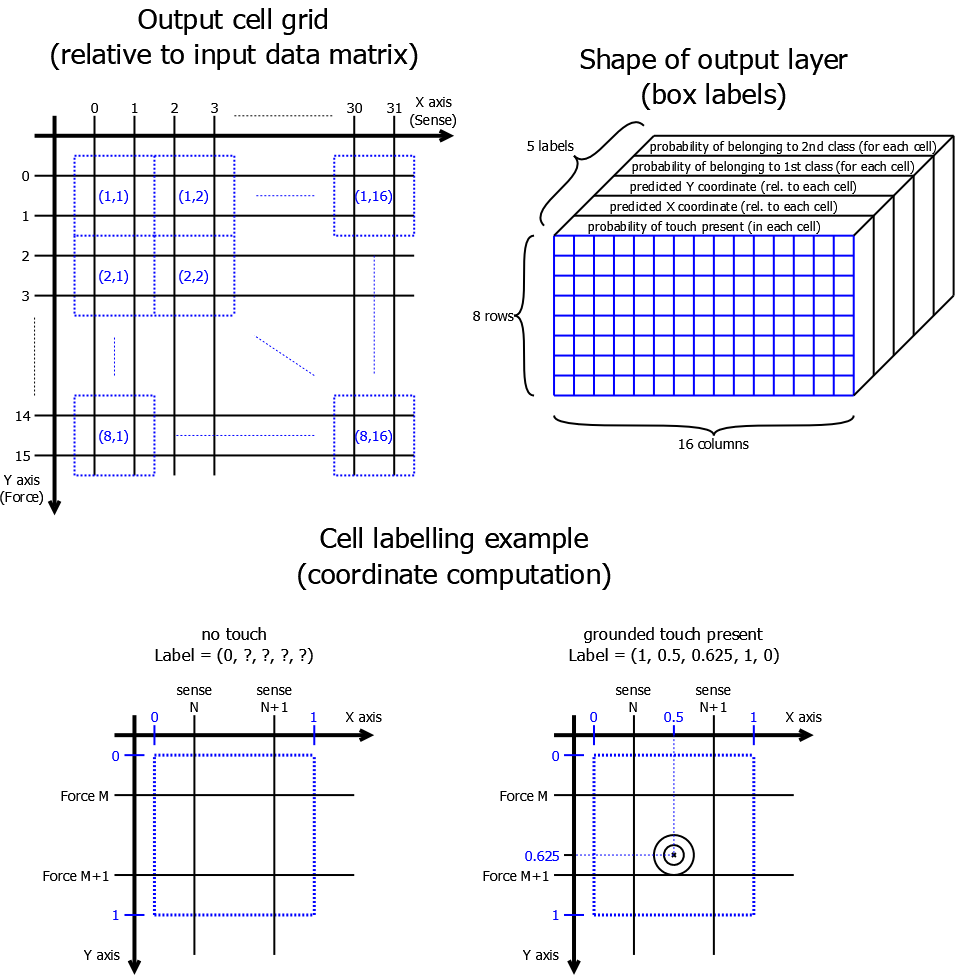
The SS and SF arrays are the very first row and column respectively (outside the self matrix).

Note that the touch locations are distinguishable despite the common mode values and noise (which should naturally be filtered by weights of opposite signs in the convolutional kernels).

## Creating a labelled training set

At this stage, we know the touch locations for each of the 2705 raw data frames. The way each data is labelled/stored is by creating a cell grid (see figure below) and for each cell, we have 4 labels: 1 label for *the presence of touch in the cell* (‘1’ if touch, ‘0’ if no touch)*,* 2 labels for *the coordinates* (one for *X* and one for *Y*) and 2 labels for *the touch class* (‘10’ if 1st class, ‘01’ if 2nd class).The coordinates are values in the [0, 1] interval, corresponding to the relative touch location in the cell (see figure below).

Compared to previous implementations, having all labels in the [0, 1] interval will facilitate the training of the CNN (stronger gradients near 0 leading to faster learning; normalization can be omitted).



At the end of this initial labelling stage, each touch will correspond to a unique cell in the output stage. While this might seem good for discriminating *touch* from *no touch*, a problem arises when the touch is located at the edge of a cell. In this scenario, the model won’t know how to distinguish the touch (it will likely classify the both adjacent cells sharing the touch as *no touch*!).

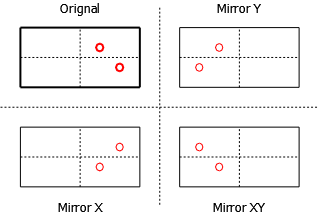
To mitigate this issue, a Matlab algorithm will *expand the touch labels* to the adjacent cell(s) if the touch is within **0.05** from it(them). Therefore, a touch may correspond to 1, 2 or 4 cells in the output grid. An algorithm called *non-maximum suppression* (described in a dedicated section below) will eliminate redundant touch predictions (so having extra “predicted” touches around a real touch is usually not an issue). Overall, this manner of labelling the touches is more robust and significantly increases the classification accuracy for touches at the grid edge (~20% of the total touch surface).

## Data augmentation

A cheap way of increasing the training data set is “*data augmentation*”. Basically, this means *creating new labelled data* from existing ones by geometrical *transformations* (translation, rotation, mirroring) taking into account the nature and symmetry of the data. *Note*: data augmentation can also be done by adding noise, but this isn’t always effective and the noise must be properly characterized.

For the touch application, the input data matrix may have specific *patterns on the rows* (constant, linear or parabolic noise shape for a given time slot along one Force line; e.g.: display noise) or *on the columns* (low frequency or beat frequency noise along the Sense lines where a touch is present; e.g.: charger noise, lamp noise). Therefore, the rows and columns can’t be rotated or taken out of sequence.

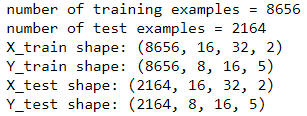
The only viable transformation is the *mirroring* (also taking into account that a touch can be present in any quadrant of the input matrix), which generates 3 new views for every input matrix, resulting in *4x more training data* (see below).



A Matlab script will augment the data giving a total of *10820 labelled input data* (*7772* for *grounded* touch and *3048* for *floating* touch).

Finally, the 10820data will be *permutated* (to randomize the order) and split into a *training* set (used in training the neural network model) and a *testing* set (used to evaluate the model accuracy on previously unseen data): **80%** (8656 data) used for training and **20%** (2164 data) used for testing.

To conclude, we now have the following data to be used in training and testing the CNN:

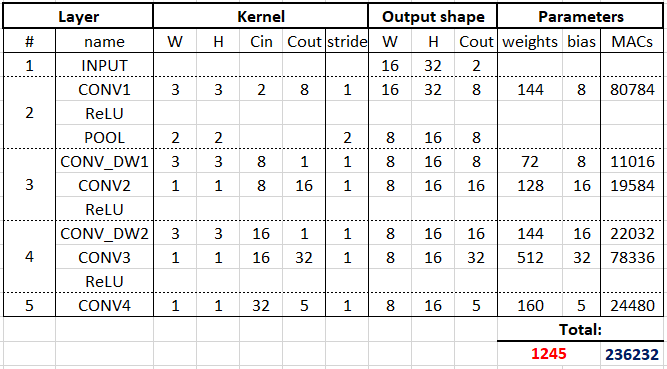


# Creating and training the CNN model

## The CNN architecture

The model does both the classification and the regression, so it **predicts the touch presence,** **the touch coordinates and the touch class** (e.g. grounded, floating, etc.).

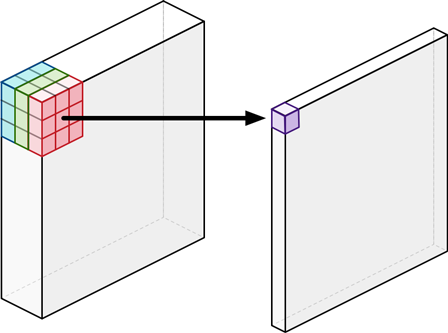
The chosen CNN architecture is summarized below:



*Note*: Cin and Cout refer to the number of input and output channels respectively.

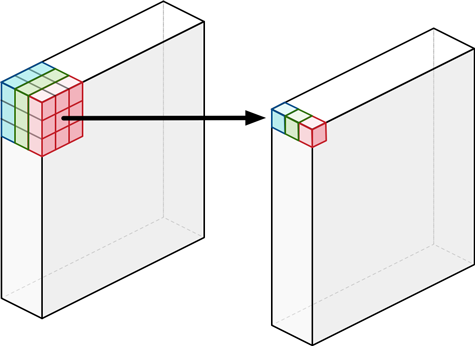
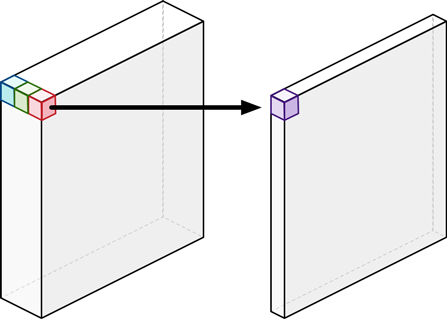
There are 5 layers (1 input layer and 4 convolutional layers). The input consists of 2 channels (mutual and self matrices) of size (16, 32).

The convolutional layers are of 2 types: standard *convolution over volumes* (CONVi) and *depthwise separable convolution* (CONV\_DWi). A standard convolution is shown below for an input with 3 channels (Cin = 3) and a kernel of (3, 3, 3, 1):



The resulting point is the dot product of the 27 input elements (highlighted in RGB) with the convolution kernel (a 3x3x3 volume not shown). Multiple output channels (Cout) for the convolution kernel mean that multiple 3x3xCin volumes are used and each of them will generate a “slice” (channel) like the one shown above. Stacked together, the output will have Cout such channels.

The disadvantage of the regular convolution is that it will combine every input element with every kernel element, being computationally wasteful, since not all combinations are needed/relevant. A depthwise separable convolution is much lighter in terms of parameters, with little loss in information (compared to a standard convolution). It is illustrated below:

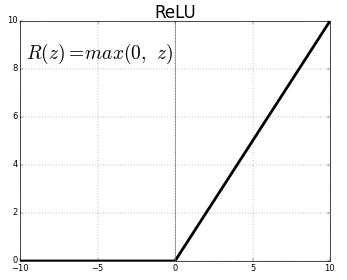
It consists of 2 parts: a *depthwise* convolution (seen on the left) and a *pointwise* convolution (seen on the right). A depthwise convolution doesn’t combine the input channels, performing the convolution on each channel separately (for the purpose of filtering each channel, performing edge detection, etc.). It is followed by a pointwise convolution (which is a regular convolution, with a 1x1xCin kernel), which will recombine the input channels as a weighted sum in order to create new features. To create more output channels, Cout such pointwise kernels are stacked.

This will reduce the number of parameters (compared to a standard convolution) by a factor of:

where “Cout” is the number of output channels and “f” is the kernel rank (3 in our case). For 8 output channels, the reduction factor is ~0.24, meaning a **~75% parameter reduction**!

*Note*: preceding each convolution, zero-padding is used (size of 1) in order to maintain the input shape (the stride is always 1 for convolutions).

The *ReLU (Rectified Linear Unit)* operation happens at the output of each layer, with the exception of the last one. The ReLU function is shown below:



*Note:* The ReLU in between the depthwise and the pointwise convolutions is optional (*not used* currently, as it was observed that information is lost and overall accuracy degraded).

There is also a POOL (Max Pooling) operation after the first convolution for the purpose of scaling down the size by a factor of 2 in each direction. It outputs the max of each 2x2 patch of the input:



One final thing to note on the architecture choice is the progression of the depth (number of output channels): *2 (input) → 8 → 16 → 32 → 5 (output)*. While the input and output are determined by the application, the size of the hidden layers is determined by 3 factors: *exponential progression* (we want to derive more and increasingly complex features: from edge detection in the 1st hidden layer to pattern/shape detection in the 3rd layer), *complexity requirement* (more and deeper layers will always give better accuracy, provided we have enough training examples) and *implementation restrictions* (finally, the hardware and processing time will be the main roadblocks, so limiting the weights to a reasonable number (~1000) is a must).

## Training the model

To train the model, we need to define a *loss (cost) function* which incorporates and weighs the different parts of the model: the touch presence probability, the predicted touch coordinates and the predicted touch class.

The total loss, as well as the weighted partial losses are defined as:

Note that there are 2 main types of loss functions being used: *mean squared error* (also known as loss; typically used for linear regression) and *focal loss* (a variant of cross entropy; typically used for logistic regression).

Since we mainly want to compute the loss for relevant values (e.g. *where a touch is located*), two logic coefficients are introduced:

The *coordinate loss* measures the loss at the touch locations between the labelled coordinates and the predicted coordinates .

The *class loss* is the same as the coordinate loss, between the class labels (for 2 classes, they can be (1, 0) or (0, 1)) and the predicted labels .

The *touch present* or *not present* losses are log-losses which heavily penalize the model whenever a significant mis-classification takes place: a touch is missed (label is 1, but ) or wrongly detected (label is 0, but ). The weights are chosen to balance the occurrence of “touch” and “no touch” labels (out of 128 total output cells, most are labelled as “no touch”) and to even out the *precision* and *recall* (accuracy metrics described below).

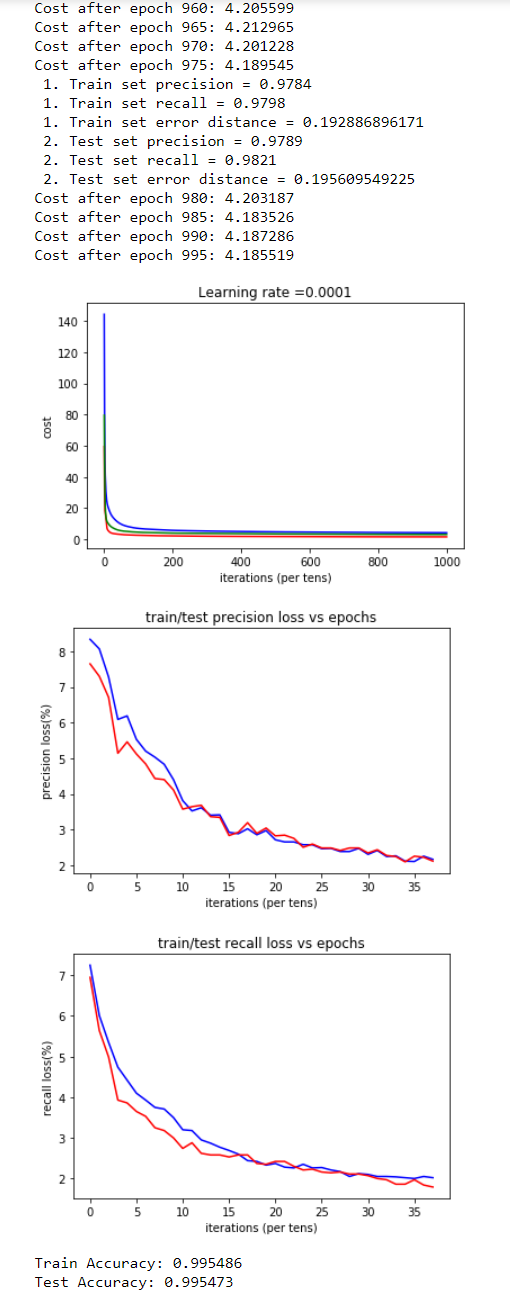
*Note* that (the touch presence probability being output) is actually taken as the *sigmoid* of the output channel **Y0** (the (8,16) matrix of the output layer, containing the probabilities of touch). This function is shown below:

With the cost function defined, the evaluation metric(s) also needs to be defined. The chosen metrics are *precision* and *recall* being defined as:

where TP = true positives (touches labelled and correctly detected), FP = false positives (touches not labelled, but wrongly detected), FN = false negatives (touches labelled, but not detected).

This makes the errors more interpretable and gives a better sense of the expected accuracy of the model. One final metric is the mean coordinate error in the TP case (even when a touch is correctly detected, it can be more or less close to the labelled coordinate). On the next page, the training progress and curves are shown (the X axis of the plots is 1-1000, the number of training iterations).

Both the precision and recall on the test set are **~98%** and the mean distance error is **~0.19** of the input grid size (meaning the distance between 2 adjacent lines, which is ~4mm). The 2 values below the plots (~99.5%) represent the percentage of output cells being correctly predicted (total of 8·16·10820 grid cells).



## The detection model: non-max suppression

The input data shape is **(16, 32)**. After passing it through the CNN, we obtain an output of shape **(8, 16, 5)**, where the last dimension contains (*probability of touch, predicted X coordinate, predicted Y coordinate, probability of 1st class, probability of 2nd class*). A Python algorithm will now **remove all redundant elements** (some of them may point to the same touch, but **we only want the highest likelihood to dictate the coordinates**).

* Since each of the (8, 16) output elements has its own probability the low-likelihood elements are removed (e.g. below 50% confidence).
* The highest probability element is chosen and the predicted X\_coord and Y\_coord values are stored.
* The **Euclidean distance** between the highest confidence coordinates and the remaining coordinates (after the first filtering, based on confidence threshold) is computed. If the distance is **smaller than a threshold** (e.g. **2**, since 2 touches can’t be within 2 adjacent nodes), the **respective elements are discarded**.

This algorithm is illustrated in the figure below (only the relevant part of the raw data is shown).

There are 3 possible touch locations, in boxes **(1, 1)**, **(1, 2)** and **(3, 2)**. The highest confidence box (box (1, 2), with 100% confidence) has the **coordinates of the first touch: (2.61, 0.96)**. Any other predicted touch within a **radius of 2** will be **discarded**. This is the case of the box (1, 1): it’s within a distance of 1.**69** from the first touch location.

The second predicted touch (box (3,2), with 58% confidence) is further than 2 from the first, so it will be kept. The algorithm stops after 2 detected touches, since there are no more high confidence boxes.



# Evaluation of the trained model on more difficult cases

## Five fingers – grounded touch (*Note: with 2nd version*)

The same example of 5 fingers that was used to test the 1st implementation, is now used to illustrate the improvement of the 2nd version.

Using the Matlab labelling algorithm, we get the following labels of the 5 touches:



Using the 2nd version mode (see figure on the next page), we get the following predictions:

The predictions have **very high confidence** (basically 100%), and are **very close to the labelled coordinates.**

Overall, the 2nd version is a big leap in accuracy compared to the original implementation and can go **to the next steps**:

* **evaluation** of C implementation on M4 emulator: **computation** **time**, **code** **size**, **accuracy** (*2F separation, jitter/precision under very noisy conditions*, etc).
* **reinforcement learning** based on **more data** with **more accurate labels.**
* **training** for detection of **new classes** (e.g. *floating finger*, *water differentiation*) and making use of **additional input features** (e.g. *self-sensing data*) and **pre-processing** (e.g. *baseline subtraction*).
* training a **recurrent neural network** to have temporal behaviour (use previous touches’ information to make better predictions).
* add high-level output features, like **gesture/pattern detection.**

## Five fingers – floating touch (*Note: with 5th version*)

The case of 5 fingers (using the 5th version) is exemplified on the last figure. Since the body is not properly grounded, the touches are “floating”, as can be observed by the presence of negative elevations. The grid elements containing the real touch centres are highlighted in black, while the corresponding predictions are highlighted in red.

Note that while all 5 touches are correctly detected, the probabilities are closer to 50% (mainly due to the cost function change from mean square error to focal loss). However, this has no impact the overall accuracy (e.g. a simple thresholding or softmax would make the probabilities closer to 0/100%, but the probability ordering and half-point would be unchanged). The focal loss heavily penalizes large misclassifications (e.g. label 1 being predicted as <0.5), tolerating small ones (e.g. 1 predicted as 0.8).



