ML Final Project

Gianfranco de Castro and Emmy Blumenthal

Spring '22

1 Overview of analysis

The objective of this analysis is to quantify the linear relationship between the applied transverse current and measured Hall voltage that appears in the classic Hall effect experiment. Provided from an author's previous work, we have obtained video files of sensors measuring the experimental dependent variables of the experiment as the applied voltage generating the transverse current for the Hall experiment is varied relatively quickly over time. An example frame from one of the video files is provided in figure (1). By systematically identifying the value each sensor reports at each frame, we can report the relationship between the Hall voltage and transverse current at high frequency. In order to identify a large number of frames each with the value of each measurement device, we employ techniques from supervised and unsupervised machine learning so that an experimenter only has to manually identify a limited number of digits to label a very large number of digits.

More specifically, the analysis consists of manually creating a crop to identify each digit on each detector; the crop is then iterated through each frame in order to produce a time sequence of tuples of digits, one for each digit place on the sensor display. We further pre-process the data by converting it to black-and-white (0 or 1)-valued pixels for model and training efficiency. A sub-sample of these digits is randomly selected and is then manually labelled. These labelled digits are then partitioned into training and validation datasets which will be used to train and assess our model, respectively. When the model is trained and tuned adequately, labelling all digits in all frames should be a relatively efficient task; this will allow us to ultimately create an estimate for the density of charge carriers in the sample by fitting the linear relationship.

2 Video frame pre-processing and manual labelling

Each video file was read frame-by-frame, and a crop was manually-selected so that each digits would be a 20×30 image consisting of binary-valued pixels. An example set of selected digits is displayed in figure (2). Each cropped and exported digit is labelled by its frame index (i.e., time step in experiment), what detector monitor it belongs to, and what position (10s, 1s, tenths, etc...) the digit is in. Additionally, 250 images of each digit have been manually-labelled by selecting random digits from the entire set of available digits (i.e., we randomly sampled the time indices). After systematically extracting and labelling data, it was exported to a set of .tsv files which can be found on the GitHub page (these are probably the most valuable byproduct of this project);

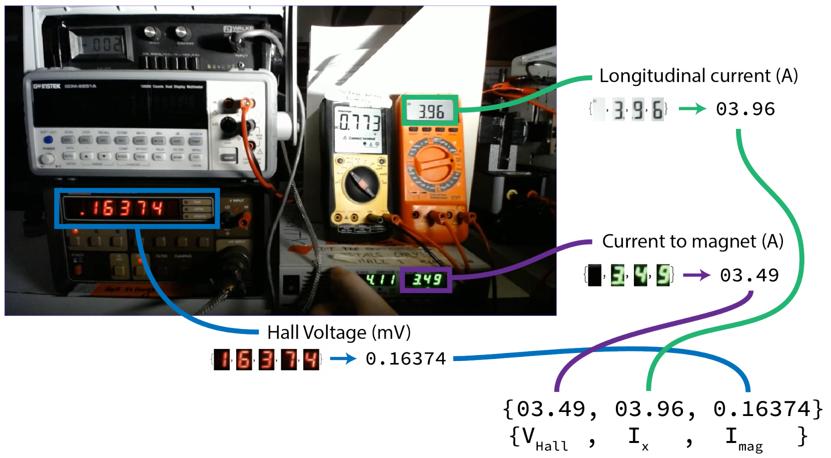


Figure 1:

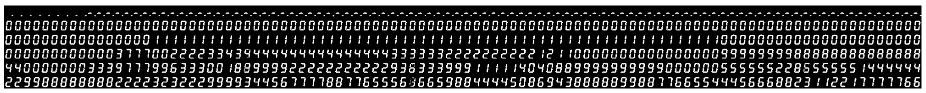


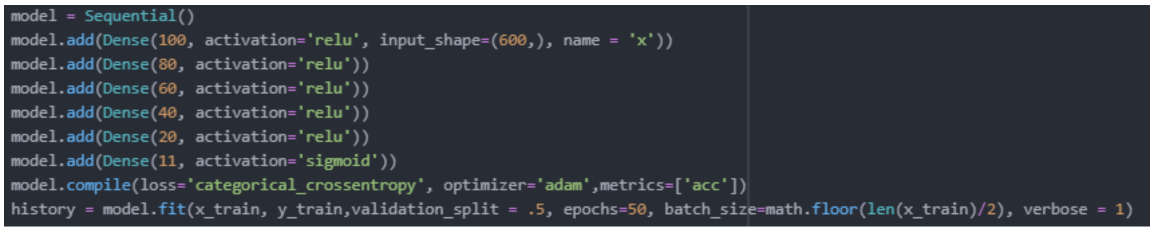
Figure 2: Example manually-cropped digits

each file corresponds to a run, sensor, and digit place; each row represents a particular time step, and each column corresponds to a given pixel. The labelled images are then split into training and validation data sets. The data files were named like ”SampleElement-WhatWeVaried-SensorMeasured-DigitPlacementLeftToRight.tsv.” For example, ”CuCurrent3_HallVoltage_1.tsv” refers to the pixel values in the first digit of the Hall Voltage monitor for the third run with the copper sample where we vary the applied transverse current.

3 Supervised model design and results

Keras Implementation

For this implementation, we used dense layers and attempt to vary hyper-parameters in order to determine the network skeleton that yields the best performance. The specific details of the initial network are given in the image below. Note the neurons are set to the ones determined to yield optimal performance in the section below:



```
model = Sequential()
model.add(Dense(100, activation='relu', input_shape=(600,), name = 'x'))
model.add(Dense(80, activation='relu'))
model.add(Dense(60, activation='relu'))
model.add(Dense(40, activation='relu'))
model.add(Dense(20, activation='relu'))
model.add(Dense(11, activation='sigmoid'))
model.compile(loss='categorical_crossentropy', optimizer='adam',metrics=['acc'])
history = model.fit(x_train, y_train,validation_split = .5, epochs=50, batch_size=math.floor(len(x_train)/2), verbose = 1)
```

Figure 3: Initial Network structure

Varying Neurons

Starting with a relatively wide network - the same number of neurons as input parameters - we vary the neurons. We see that a thinner network tends to perform better, but see diminishing returns after a certain point if we go too thin.

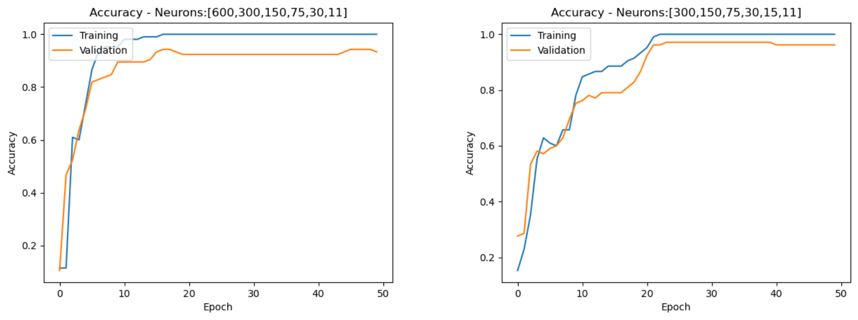


Figure 4: (i) [600,300,150,75,30,11], (ii) [300,150,75,30,15,11]

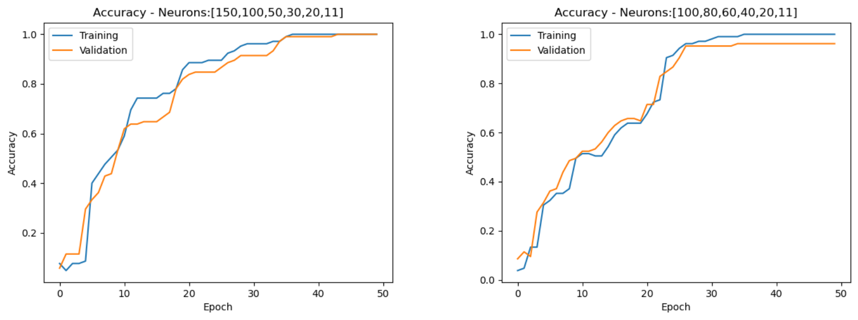


Figure 5: (iii) [150,100,50,30,20,11], (iv) [100,80,60,40,20,11]

We note convergence between the training and validation sets for the third set of neurons - [150,100,50,30,20,11]. Likewise, in the loss plots below, we note similar behavior. Additionally, as mentioned prior, going skinnier than our optimal set of neurons shows divergence in both accuracy and loss, similar to what happens when we use a wider network.

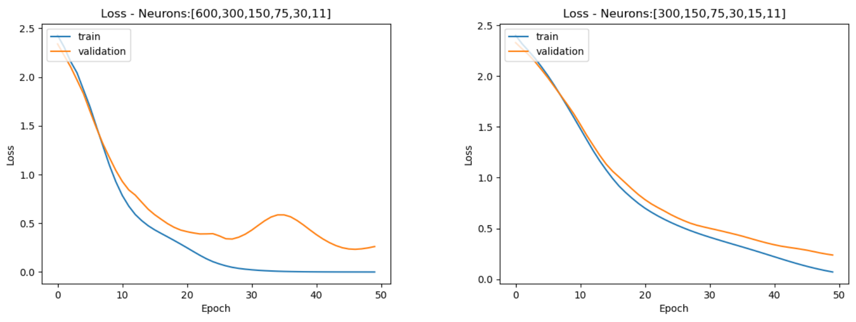


Figure 6: (i) [600,300,150,75,30,11], (ii) [300,150,75,30,15,11]

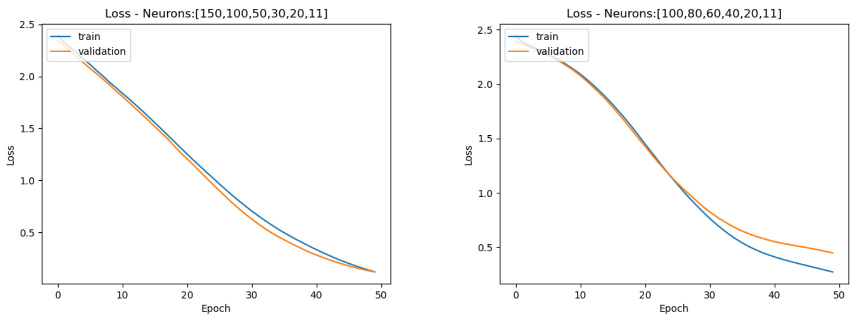


Figure 7: (iii) [150,100,50,30,20,11], (iv) [100,80,60,40,20,11]

Varying Layers

Starting from our base case of 6 layers as in the previous section, we now add and subtract two layers to determine whether a longer or shorter network performs best. We note a better performance in the shorter network, while the longer network shows similar divergence as with wider layers before. Additionally, for the longer network we had to increase training time. At first it looked like it would converge with more training, but unlike the shorter network, it diverges.

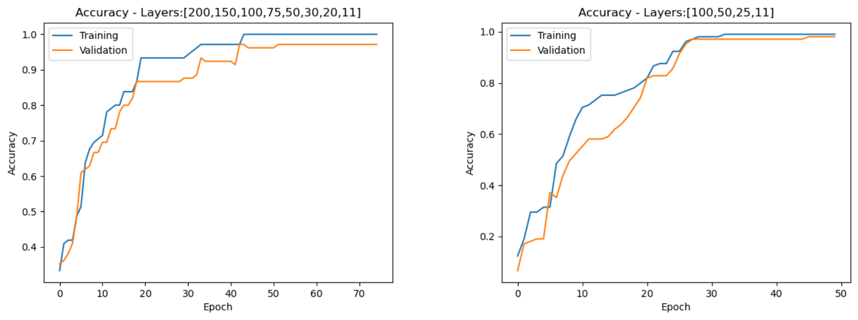


Figure 8: (i) [200,150,100,75,50,30,20,11], (ii) [100,50,25,11]

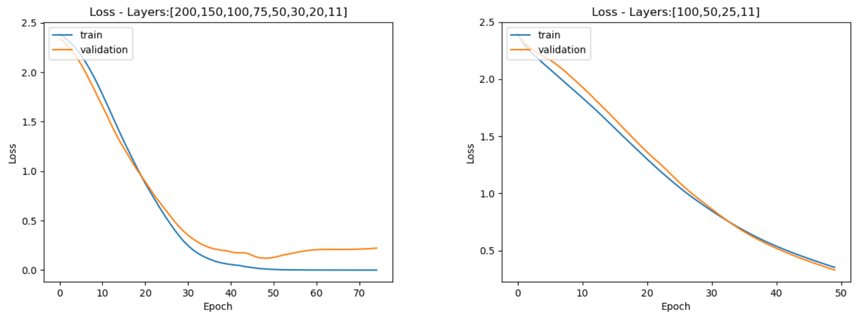


Figure 9: (i) [200,150,100,75,50,30,20,11], (ii) [100,50,25,11]

Training Time

We note that the loss does continue to decrease after a certain number of epochs past 50. In order to see this behavior, and see how far we can minimize it, we increase the training time and also vary the batch sizes, or frequency with which we update the parameters within an epoch. After bumping up the epochs from 50 to 200, we attempted a 4 layers and 5 layer network. Both performed extremely well with losses of essentially 0 and training and validation accuracies of ≈ 1.

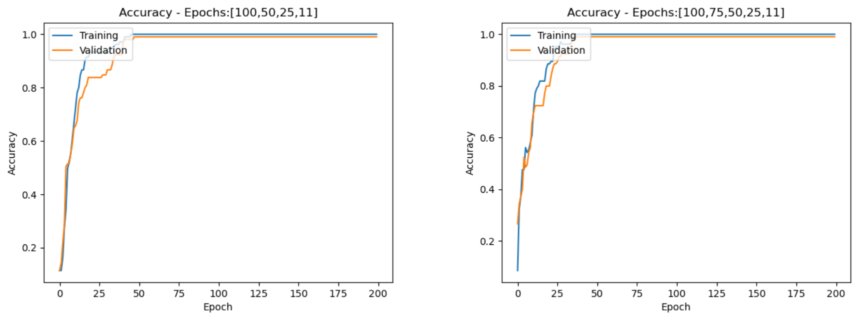


Figure 10: (i) [100,50,25,11], (ii) [100,75,50,25,11]

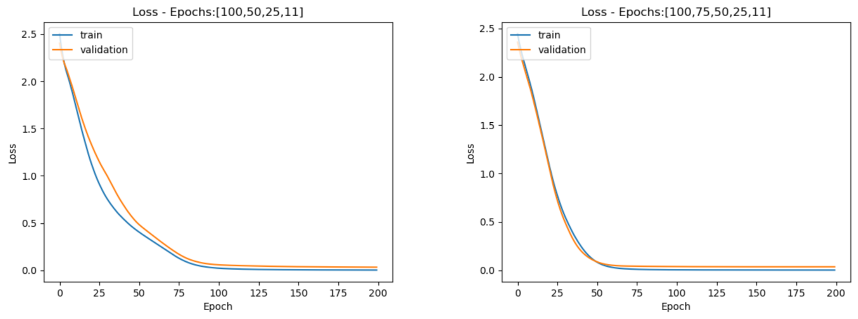


Figure 11: (i) [100,50,25,11], (ii) [100,75,50,25,11]

Looking at varying the batch sizes, we originally used half of the training size, which means we update halfway through each epoch. Below are plots where we instead try a batch size equal to the training size, so updating once per epoch, and a batch size that's much smaller, about a 20th of the training size. Interestingly, we note that our network learns much faster and more accurately while updating the parameters with more frequency with the smaller batch size. The results are shown below:

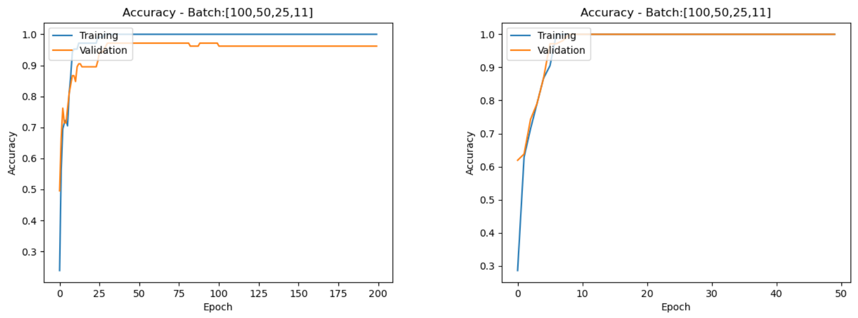


Figure 12: (i) Batch Size: $Len(x_{train})$, (ii) Batch Size: $Len(x_{train})/20$

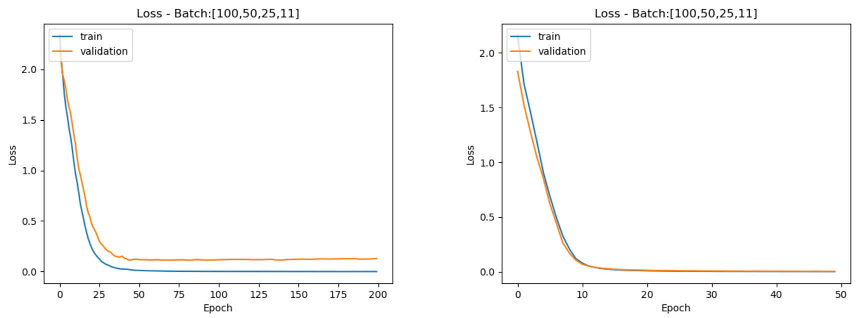


Figure 13: (i) Batch Size: $Len(x_{train})$, (ii) Batch Size: $Len(x_{train})/20$

Optimizers

In this section we vary the type of optimizer implemented and see if our network structure is robust to these changes. In terms of the losses, they all seem to do pretty well, but SGD and Agarad show some noisy behavior in later epochs. We do not expect SGD to work as well as Adam since it is used for qualitative data, while we are categorizing our data. Nonetheless, we see decent performance from all optimizers with our best-performing model:

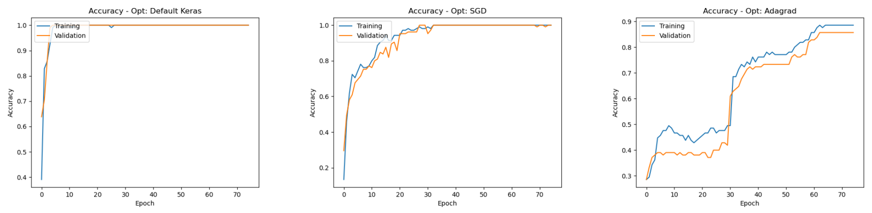


Figure 14: (i) Default, (ii) SGD, (iii) Adagrad

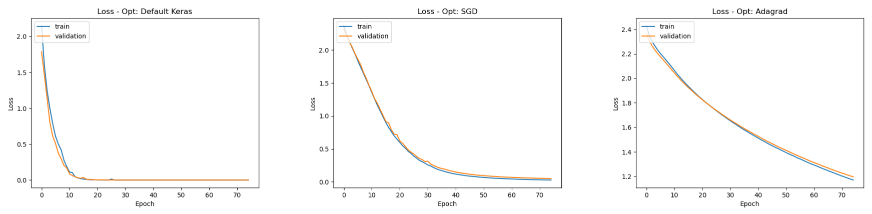


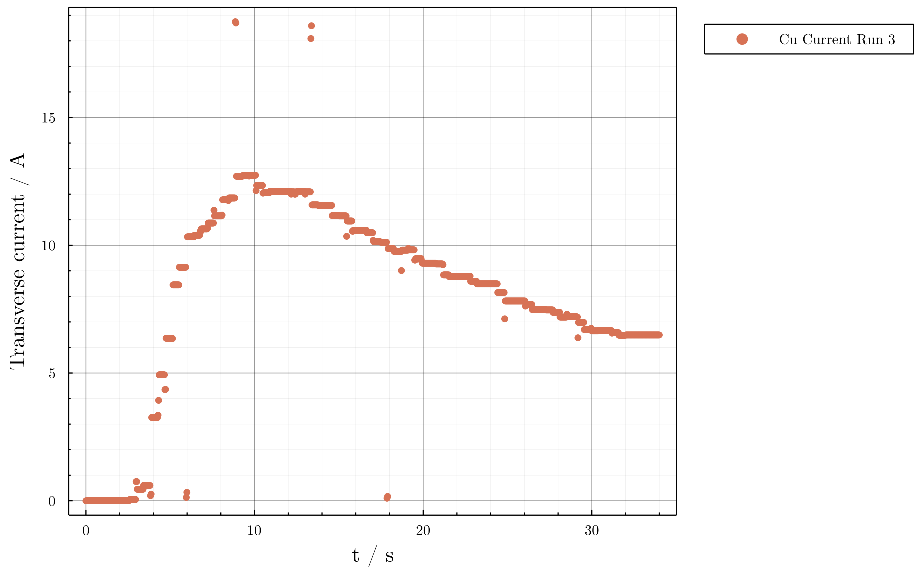
Figure 15: (i) Default, (ii) SGD, (iii) Adagrad

***Julia* implementation** In *Julia*'s Flux library, we implemented the model which was recommended by the previous section: we choose a chain of dense layers with widths 100, BatchNorm, 50, 25, 25, 11, followed by a soft-max layer. After training multiple models, one model for each detector, we use the models to label all digits; then, we use the stored data to assign to each time step a value of the transverse current and Hall voltage. The results are plotted in figure (16) and (17) for the copper current run 3 where the the transverse current applied was varied.

4 The Hall effect

The Hall effect experiment describes a relationship between an applied transverse voltage, I_x, an applied magnetic field, B_z, the density of charge carriers n, and charge carrier charge, e. This relationship is summarized as $V_H = I_x B_z/(nte)$.

Figure 16: Sensor readings



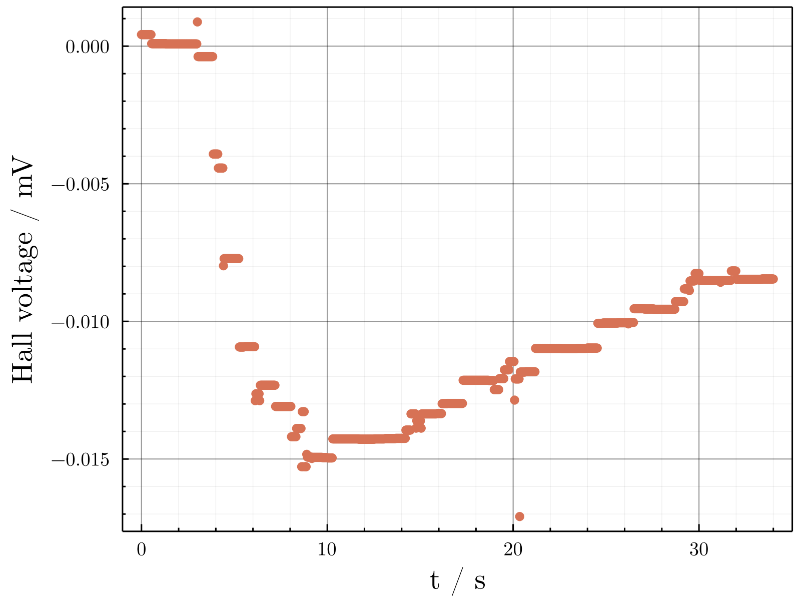
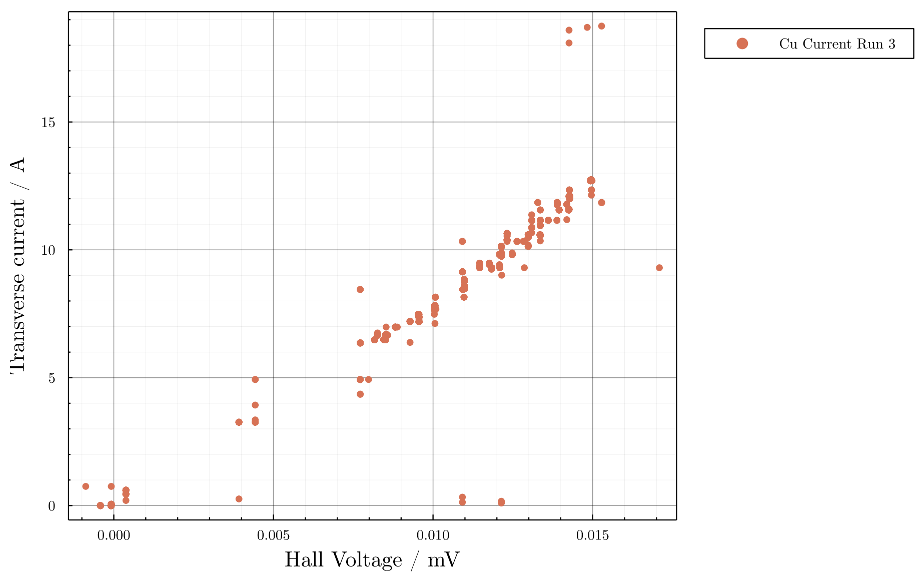


Figure 17: Hall effect



In our figures, we see that we are able to capture the data from the output monitors successfully. We also observe that when we plot the transverse current vs the Hall voltage, we obtain a relationship that looks linear. The erroneous points can be attributed to two things: in the experimental data, we chose to sweep from high to low transverse currents which means that we must 'ramp up' from a low applied transverse current to a greater one. This is apparent in the first few tens of frames. The other outlying points that seem to be an exception from apparent continuity in the curves can be attributed to mis-classified digits; these mis-classified digits likely come from over-fitting.

As an example showing that these outliers are due to mis-classification, we remove the BatchNorm layer and see that the amount of outlier points increases.

5 Further methods: unsupervised methods for efficient labelling

Ideally, an experimenter should have to label only very minimal data. Therefore, to minimize the amount of data necessary to label, using any method that identifies clusters or the most representative examples would make this process more efficient.