FAIR Document - Identifying Interaction Location in SuperCDMS Detectors

1 Problem description

The Super Cryogenic Dark Matter Search (SuperCDMS) experiment uses silicon and germanium particle detectors operated at temperatures of ~ 30 mK to search for Weakly Interacting Massive Particles (WIMPs), which are candidate dark matter particles that interact weakly with nuclei in the detectors. In operating these detectors, it is required not only to measure the energy of the interaction between the WIMP and the nuclei, but also to reconstruct where the interaction occurred, as the location can be used to separate background interactions from signal and to correct for variations with location of the energy response.

In this project, we address the problem of accurately reconstruct the locations of interactions in the SuperCDMS detectors using machine learning methods. The approach is to use data collected with a radioactive source at known locations to train and qualify ML models.

The problem may be expressed as:

$$y = f(\boldsymbol{x}; \theta) \tag{1}$$

where we want to obtain a function f parameterized by θ that maps $\boldsymbol{x}$ (the signals we have observed) to y (the interesting interaction information such as locations). Then the problem becomes: we look for the interesting interaction information (y) by observing collected signals ($\boldsymbol{x}$). Given a large number of observed pairs ($\boldsymbol{x}, y$), we:

- want to train a machine learning model which learns the complex mapping function conducted by the SuperCDMS detector (f in Eq. (1));
- further want our learned machine learning model to have a good generality, so that the learned model can be applied to events whose locations have never been observed before.

2 Data description

2.1 Detector testing

A prototype SuperCDMS germanium detector was tested at the University of Minnesota with a radioactive source mounted on a movable stage that can scan from the edge to the center of the detector. The detector is disk-shaped with sensors placed on the top and bottom surfaces to detect the particles emitted by the radioactive source (Fig. 1).

The sensors measure phonons (quantized vibrations of the crystal lattice) that are produced by the interacting particle and travel from the interaction location to the sensors. The number of phonons and the relative time of arrival at a particular sensor depends on the positions of the interaction and the sensor. The sensors are grouped into six regions on each side of the detector and each of these "channels" produces a waveform for every interaction. For the test performed at Minnesota, five channels on one side of the detector were used (Fig. 2).

The movable radioactive source was used to produce interactions at thirteen different locations on the detector along a radial path from the central axis to close to the the detector's outer edge (Fig. 3).
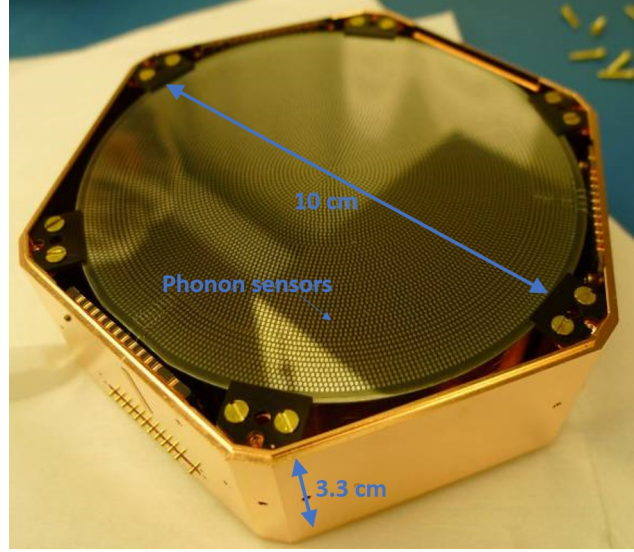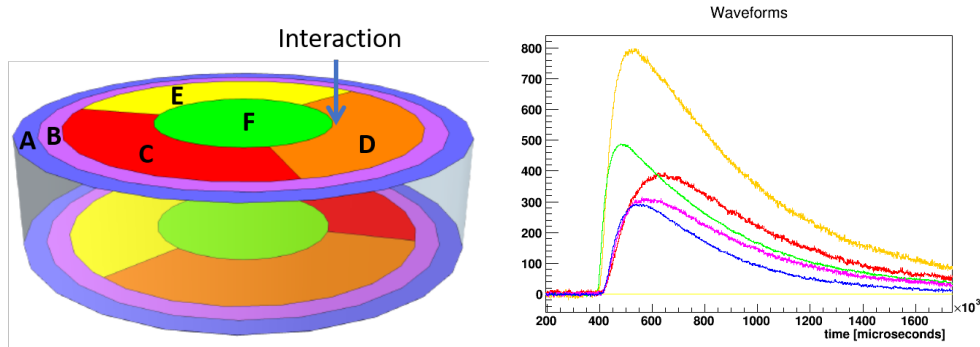
Figure 1: A SuperCDMS dark matter detector.

Figure 2: Pulses from an interaction in a SuperCDMS detector.

Data from many interactions were collected at each location. While the location of the source is known to high precision, the location of a specific interaction can vary from the source position by ± 0.5 mm.

2.2 Data from the detector test

For each interaction a set of parameters was extracted from the signals from each of the five sensors. These parameters represent information known to be sensitive to interaction location, including the relative timing between pulses in different channels, and features like the pulse shape. The relative amplitudes of the pulses are also relevant but due to instabilities in amplification during the test this data is not included. The parameters included for each interaction are:

- P[B,C,D,F]start, the time at which the pulse rises to 20% of its peak with respect to Channel A
- P[A,B,C,D,F]rise, the time it takes for a pulse to rise from 20% to 50% of its peak
- P[A,B,C,D,F]width, the width (in seconds) of the pulse at 80% of the pulse height
- P[A,B,C,D,F]fall, the time it takes for a pulse to fall from 40% to 20% of its peak

These parameters are illustrated in Fig. 4.
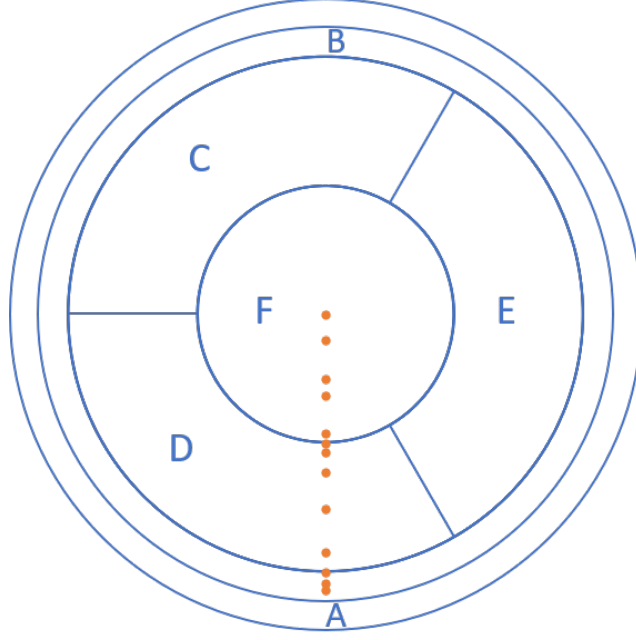
2

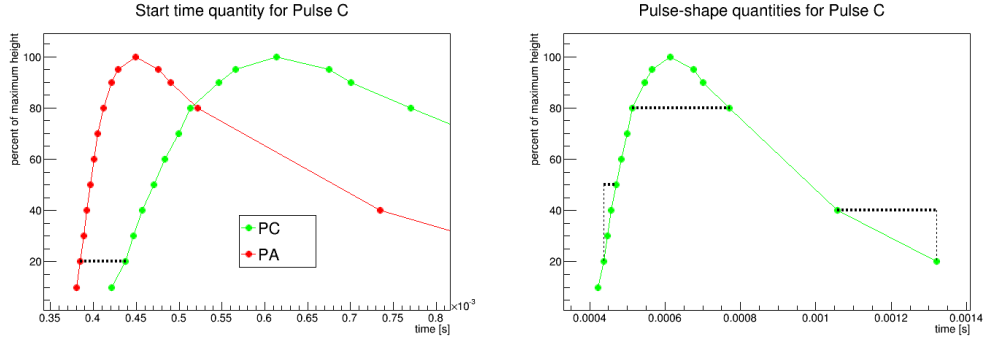Figure 3: Interaction locations included in the dataset.

Figure 4: Pulse timing and shape parameters

2.3   Example of manual location reconstruction

As a benchmark of the quality of the location prediction, we have conducted a simple analysis just using the relationship between interaction location and a single pulse parameter (chosen for its high sensitivity to location position for this dataset). The results are shown in Fig. 5. The error bars in the parameter represent the standard deviation of the parameter values from all the interactions at the given location. The data were fit to a functional form (shown in red) which can be used to predict the location of an interaction whose source is not known.

3   Method

In this section we present the details of the methodology for solving Eq. (1) using a deep neural network (DNN). This is provided as an example of how ML may be used to approach this problem. We describe how the data are pre-processed (Section 3.1), the design of DNN models (Section 3.2), and present preliminary results that we obtained in Section 3.3. The details of the Github code are given in Section 3.4.
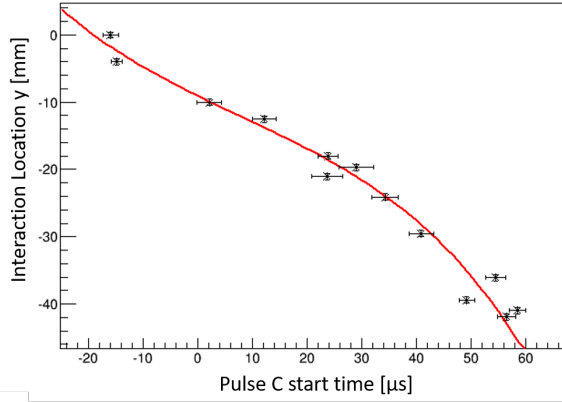
3

Figure 5: Interaction location as a function of a single pulse-timing parameter.

3.1 Data preprocessing

As we discussed in Section 1, in order to learn the mapping function f well, we need a large number of observed pairs $(\boldsymbol{x}, y)$. $\boldsymbol{x}$ in our current experimental data are 19 informative features extracted from 5 observed signals (pulses) (see Section 2.2) and y is the interaction location (in our current dataset, we have 13 different locations). In total, we obtain 7151 $(\boldsymbol{x}, y)$ pairs, of which the details are shown in Table 1.

Table 1: Dataset for MLS and HOS

Subset name	Position/Location	Sample number
MLS	0.0	924
	-3.969	500
	-9.988	613
	-17.992	357
	-19.700	376
	-21.034	747
	-24.077	567
	-36.116	560
	-39.400	386
	-41.010	606
HOS	-12.502	395
	-29.500	634
	-41.900	486

To train the DNN models and also test its generalization to new samples with never-seen positions during the process of model learning, we first divide the dataset into two separate subsets [1]:

- *Model-learning subset* (MLS). It contains 5636 data samples whose positions are in the set {0.0, -3.969, -9.988, -17.992, -19.700, -21.034, -24.077, -36.116, -39.400, -41.010}. We further split this subset into training set (80% of 5636 data samples) and validation set (20% of 5636 data samples). We then use the training set to train the DNN model and then validate the learned DNN model on the validation set.

- *Held-out subset* (HOS). It has 1515 data samples whose positions are in the set {-12.502, -29.500, -41.900}. We treat it as our test set and finally we test the performance of the learned DNN model on this subset.

---

[1]For simplicity, we split the dataset in this manner; while in practice, you can explore other settings (e.g., you can always add or remove data samples from HOS.)

To facilitate the learning process, it is essential to normalize the features. In this work, we normalize the features to have a mean of zero and a standard deviation of one. We use sklearn [2] to achieve this goal. Fig. 6 showcases the feature information before (Fig. 6(a)) and after (Fig. 6(b)) normalization. We can observe that the numerical range of several features are small (e.g., PBstart) while some of them are large (e.g., PFfall), thus it is crucial to conduct normalization to mitigate the domination of learning by some of the features.

(a)   (b)
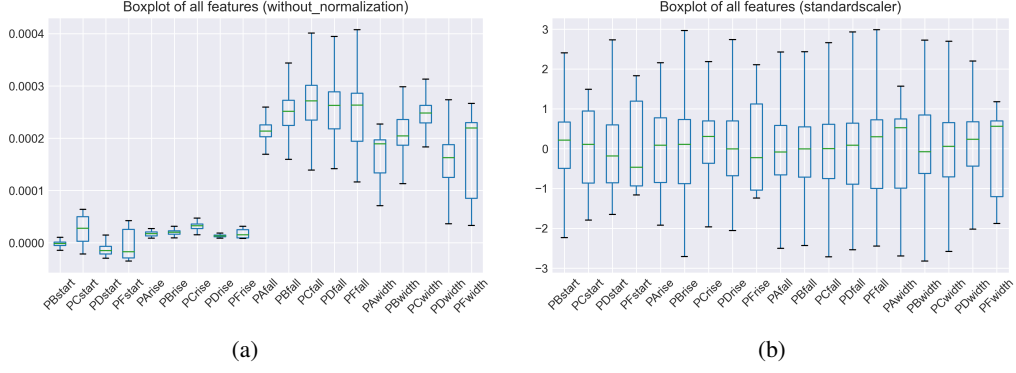
Figure 6: The numerical range of each feature before normalization (left) and after normalization (right).

3.2   Neural network models

We learn the mapping function f in Eq. (1) through deep fully-connected neural networks by solving the optimization problem:

$$\min_{\theta} \; \ell(y, f(\boldsymbol{x}; \theta)) \tag{2}$$

where $\boldsymbol{x}$ is the extracted features from observed signals, y is the true position/location (ground truth), ℓ is the loss function, and θ is the network weights. In our implementation, we use root mean squared error (RMSE) as our loss function.

We implement our neural network with Pytorch 1.9.0 [3]. The framework of our neural network model is shown in Fig. 7. It is a fully-connected network with non-linearity activation functions. In particular, in each hidden layer except the output layer, we employ a linear layer followed by the batch normalization [1], leaky rectified activation [2, 3], and dropout. For the output layer, we simply pass the learned features through a linear layer and obtain its prediction directly.
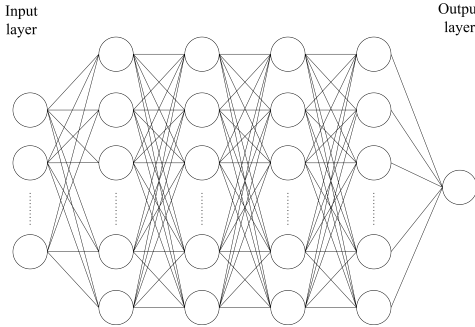

Figure 7: The framework of deep neural network models.

---

[2]`https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.`
`StandardScaler.html`

[3]`https://pytorch.org/`

In our experiments, we try neural networks with {2, 5, 10} hidden layers, indicating an increasing model complexity, and we set the number of neurons per hidden layer to be 32 [4]. To avoid the issue of overfitting, we employ dropout with a drop ratio of 0.5. We use Adam [4] as our optimizer and exploit a constant learning rate which is 0.001. We set the maximum training epochs to be 500 epochs for each model. After 500 epochs of training, the neural network model is converged well and does not show any overfitting based on the performance on both training and validation sets.

3.3 Preliminary results

In this section, we summarize the performance of our neural network models. We first show the training and validation performance curves in Fig. 8. We can observe that all three settings are converged well after 500 epochs and do not see overfitting issues. Furthermore, the neural network model with 10 hidden layers takes the longest training time to converge, which is expected due to the increased model complexity.

(a) DNN with 2 hidden layers   (b) DNN with 5 hidden layers   (c) DNN with 10 hidden layers
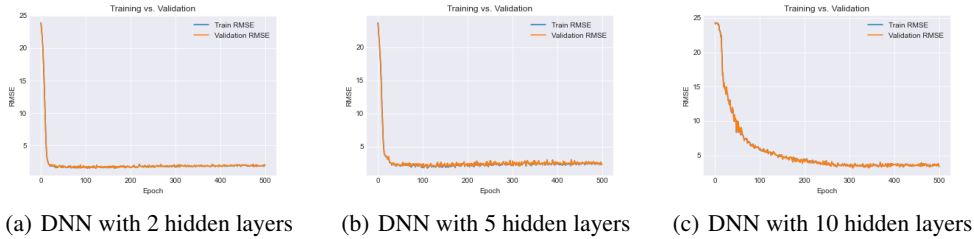
Figure 8: The performance (RMSE) curves on training and validation set.

We show the test performance on our test set (held-out subset) in Table 2. We can observe that simply increasing the model complexity does not boost the performance on our dataset, rather it hurts the performance. Therefore, we argue that to achieve better performance, it is worth exploring novel network architectures or training paradigms.

Table 2: The performance on training, validation, and test sets. The X in the DNN-X indicates the number of hidden layers.

Model	Data set	RMSE
DNN-2	Training set	1.530
	Validation set	1.533
	Test set (HOS)	1.741
DNN-5	Training set	1.723
	Validation set	1.857
	Test set (HOS)	5.179
DNN-10	Training set	3.200
	Validation set	3.202
	Test set (HOS)	7.072

To further see the performance of our DNN models on each position of test set (HOS), we visualize the histogram of predictions of NN-2, NN-5, and NN-10 in Fig. 9, Fig. 10, and Fig. 11, respectively. We can observe that NN-2 yields the best predictions across all test positions, which is consistent with the results in Table 2.

---

[4]It is worth exploring other settings (e.g., different network architectures, deeper and wider network models).
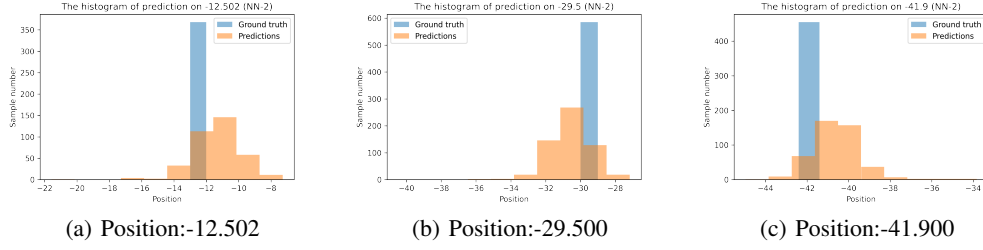
(a) Position:-12.502   (b) Position:-29.500   (c) Position:-41.900

Figure 9: The histogram of predictions (DNN with 2 layers).

(a) Position:-12.502   (b) Position:-29.500   (c) Position:-41.900
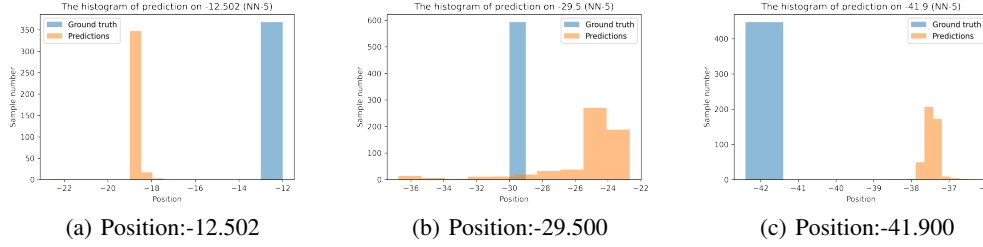
Figure 10: The histogram of predictions (DNN with 5 layers).
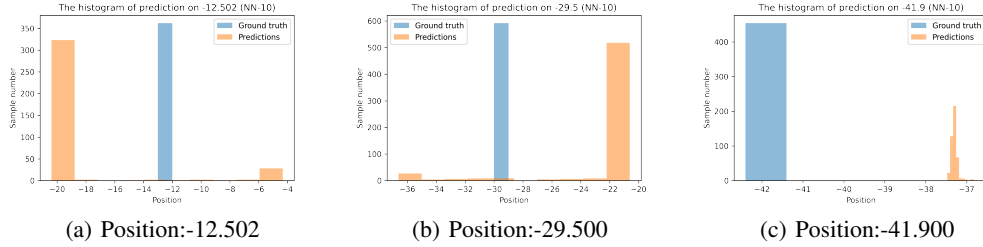

(a) Position:-12.502   (b) Position:-29.500   (c) Position:-41.900

Figure 11: The histogram of predictions (DNN with 10 layers).

3.4 Code availability

Our complementary Python code and Jupyter notebooks are available on Github[5]. The structure of folders and files on Github is shown in Fig. 12. We have provided a detailed Readme document inside each folder. In addition, for all Jupyter notebooks and Python code, we have provided detailed inline comments and descriptions.

3.4.1 Github: Data_Preprocessing

The purposes of data preprocessing are to explore and manipulate the dataset, and prepare the data for training our neural network models. This folder (see Fig. 13) contains three Jupyter notebooks.

- 1_txt2csv.ipynb which is used to extract the features (signals) and targets (positions/locations) from raw txt dataset and format them into csv files, so that it can be easily manipulated through dataframe.
- 2_prepare_dataset.ipynb which is used to divide the dataset into two subsets—MLS and HOS—and further split the MLS into training and validation sets (see Section 3.1 for details).
- 3_features_analysis.ipynb which is used to visualize the numerical range of each feature and analyze the features with different normalization methods.

---

[5] https://github.com/ml-deepai/FAIR-UMN

Figure 12: The structure of folders and files on Github.

Figure 13: The structure of Data_Preprocessing on Github.

3.4.2 Github: DNN_Models

This folder (see Fig. 14) is the core part of our method. It includes the details about the deep neural network model and its training/validation/test procedures.

- <u>model_files</u>. It is the backbone engine of our method. It includes the Python code to build the neural network models and also the corresponding auxiliary functions.
- <u>train_deep_regression.ipynb</u> which actually conducts the training, validation, and test procedures. We also define and set up all hyper-parameters in this Jupyter notebook.

3.4.3 Github: Results_Analysis

After training and testing the neural network models, we provide a straight-forward and simple script to extract and analyze the results. We archive this script in the folder named *Results_Analysis* on Github (see Fig. 15).

- <u>1_statistical_info.ipynb</u> which is used to (1) extract the predictions on training/validation/test set from different neural network models; (2) get statistical information (e.g., mean and standard deviation) of the predictions on test set (HOS).

  <u>2_visualization.ipynb</u> which is used to generate the histogram of predictions for each DNN model.

Figure 14: The structure of DNN_Models on Github.

Figure 15: The structure of Results_Analysis on Github.

References

[1] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in International conference on machine learning. PMLR, 2015, pp. 448–456.

[2] A. L. Maas, A. Y. Hannun, A. Y. Ng et al., "Rectifier nonlinearities improve neural network acoustic models," in Proc. icml, vol. 30, no. 1. Citeseer, 2013, p. 3.

[3] B. Xu, N. Wang, T. Chen, and M. Li, "Empirical evaluation of rectified activations in convolutional network," arXiv preprint arXiv:1505.00853, 2015.

[4] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," arXiv preprint arXiv:1412.6980, 2014.