Summer 8-2022

Using Ensemble Learning Techniques to Solve the Blind Drift Calibration Problem

Devin Scott Drake
Old Dominion University, drake127@vt.edu

USING ENSEMBLE LEARNING TECHNIQUES TO SOLVE THE BLIND DRIFT CALIBRATION PROBLEM

by

Devin Scott Drake
B.S. May 2020, Virginia Polytechnic Institute and State University

A Thesis Submitted to the Faculty of
Old Dominion University in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

COMPUTER SCIENCE

OLD DOMINION UNIVERSITY
August 2022

Approved by:

Yaohang Li (Director)

Ravi Mukkamala (Member)

Fengjiao Wang (Member)

ABSTRACT

USING ENSEMBLE LEARNING TECHNIQUES TO SOLVE THE BLIND DRIFT CALIBRATION PROBLEM

Devin Scott Drake
Old Dominion University, 2022
Director: Dr. Yaohang Li

Large sets of sensors deployed in nearly every practical environment are prone to drifting out of calibration. This drift can be sensor-based, with one or several sensors falling out of calibration, or system-wide, with changes to the physical system causing sensor-reading issues. Recalibrating sensors in either case can be both time and cost prohibitive. Ideally, some technique could be employed between the sensors and the final reading that recovers the drift-free sensor readings. This paper covers the employment of two ensemble learning techniques — stacking and bootstrap aggregation (or bagging) — to recover drift-free sensor readings from a suite of sensors. The ensembles are composed of two different deep learning network types: Long Short-Term Memory (LSTM) networks and Gated Recurrent Unit (GRU) networks. Standalone LSTM and GRU networks were also constructed, trained, and optimized to create a baseline against which the ensemble methods could be compared. The metrics used to compare the various models were Mean Squared Error (MSE), time and computing resources required, as well as a comparison of output graph shape compared to the drift-free sensor readings.

Both the stacking and bagging ensembles outperformed the standalone models (LSTM and GRU). The stacked ensemble achieved a lower MSE than the both the

LSTM and GRU models and a similar overall fit compared to the standalone models. This was achieved using less time to train the ensemble than either of the standalone models. The bagging ensemble achieved an MSE lower than both standalone models by a factor of nearly 100 and achieved a much tighter fit when compared to the standalone models, though did require nearly 30 times the number of CPU seconds to train. In both instances, the ensemble learning methods were determined to outperform the standalone models.

TABLE OF CONTENTS

Page

Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

Figure Page

Figure Page

Figure Page

1. INTRODUCTION

1.1 PROBLEM STATEMENT

Large modern structures generally require one to many sensors to maintain a safe and operational state. These sensors can measure anything from stress on components, to temperature, or even angular displacement. Traditionally, sensors are individually calibrated in controlled environments, providing a mapping (or function) for each sensor from which the true sensor reading can be obtained [1]. Unfortunately, this approach does not take into account deployment-related factors [2], such as changes as the sensor ages, damage to components, inconsistencies in power supplies, etc. These deployment-related factors generally tend to evolve over time, causing sensor readings to become increasingly inaccurate. This is what is known as sensor drift.

Removing sensors for individual re-calibration is typically considered the solution to sensor drift, but for large sensor networks, this is generally considered infeasible and ineffective for two reasons. First, removing each individual sensor and re-calibrating them in a controlled environment could be incredibly expensive, both monetarily and in terms of time, for sensor networks tens to hundreds of sensors [2, 3]. The second problem with out-of-system re-calibration is that it only accounts for error introduced from sensor aging and does not account for system-wide or system-specific factors that are introducing drift. For instance, if a sensor reading has drifted because it measures linear displacement and the components it is attached to have stretched due to fatigue, an in-lab re-calibration of that sensor will not cause its reading within the sensor network to be drift-free. Thus, it is important to be able to obtain a drift-free set of readings from

a sensor suite that is performed while the sensors are in-system. This is known as blind calibration [3] or blind drift calibration.

Assuming that N sensors are deployed to a system with S true dimensions, so long as $S < N$, the drift-free measurements lie in a low-rank S-dimensional subspace. Since the signal null space is the complement of the signal subspace, it also lies in a low dimensional matrix. Thus, there exists a projection function $P(\cdot)$ projecting measurements, Y, onto signal null space such that $P(Y) = P(X + D) = P(D)$ where X are the true sensor readings and D is the sensor drifts. This projection function obtains an estimate of the drift, which allows the sensor readings Y to be corrected such that X may be obtained [4]. In other words, so long as there are more sensors measuring a system than dimensions being measured, it is possible to holistically recover from drift. The goal of this paper is to use ensemble learning, which is a set of deep learning techniques, to accurately estimate X given Y.

1.2 LITERATURE REVIEW

The research covered by this thesis is novel in its implementation, but in the idea. Many other researchers have attempted to recover drift-free sensor readings using various different techniques. Balzano and Nowak [3] achieved good results blindly calibrating sensor networks, though their assumption was a “a linear model for the sensor calibration functions” [3, pp. 1]. Tan et al [2] proposed a two-tiered model approach, utilizing first a least squares regression for local sensor estimation followed by a “simple linear calibration scheme” to maximize overall system performance. Dorffer et al [5] proposed a model for blind calibration of a suite of crowd-sourced

sensors, but assumed an affine calibration model, which once again is a linear model. The work and research presented in this thesis differ from all of these pieces of work in that the methods described and explored do not require the assumption of a linear drift.

Several other researchers and research groups have attempted to tackle the problem as well without assuming linearity. Xiang et al. [6] attempt to remove drift from a suite of sensors by maximizing a nonlinear function. They do not, however, make use of deep learning techniques. Wang et al. [7] used kernel methods to map nonlinear functions to higher-order linear ones, somewhat effectively representing nonlinear drift. They then used Bayesian Learning to estimate drift-free sensor measurements. About a year later, the same group of researchers went on to employ deep learning techniques to solve the blind drift calibration problem, though opt to use a fully connected Convolutional Neural Network (CNN) they call a Projection-Recovery Network, or PRNet [8]. Neither of these techniques relate to the research presented in this paper. The first method once again effectively causes all drift to be estimated by a linear function. The second method utilizes CNNs, which were not used in any capacity for this project.

Lastly, a report generated from Old Dominion University (ODU) [4] compared a slew of different deep learning techniques, including Recurrent Neural Networks (RNNs). The data gathered for this paper was used as the data for this thesis. Additionally, the methods and techniques used for constructing and training the models were used as a starting point in this thesis. The research presented in this thesis goes one step further, however, as this paper employs ensemble learning techniques of networks similar to those presented in the study from ODU.

1.3 DATA COLLECTED AND PRESENTED

The data provided and used came from a project previously conducted by another graduate student at Old Dominion University in which a model section of a truss bridge was constructed and placed under a load. The bridge section had 16 pairs of strain sensors. Each pair had one calibrated sensor, to act as the ground truth sensor, and one uncalibrated sensor. Voltage readings were taken from the sensors once every second for approximately four days [4]. The resulting dataset contained 329,264 different time steps for all 32 sensors.

The raw sensor training data featured values ranging from 3.4511 volts to -788.307 volts, with a standard deviation of 193.736 and a mean of -390.262 volts. In order to properly process the training data, it was normalized. To do this, the mean was subtracted from each sensor reading and then the resulting value was divided by the standard deviation. The same process and values are done to normalize the testing and validation data. The raw, uncalibrated sensor data is shown in Figure 1. The normalized uncalibrated sensor values are shown in Figure 2. Figures 3 and 4 respectively show the normalized uncalibrated and ground truth sensor data.

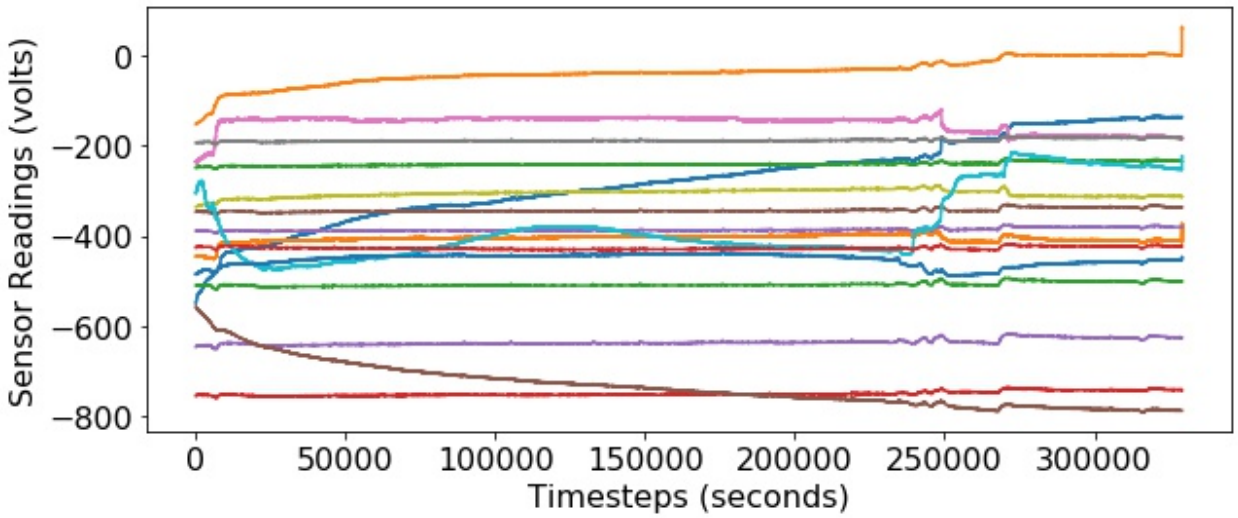

Figure 1: Raw Uncalibrated Sensor Data

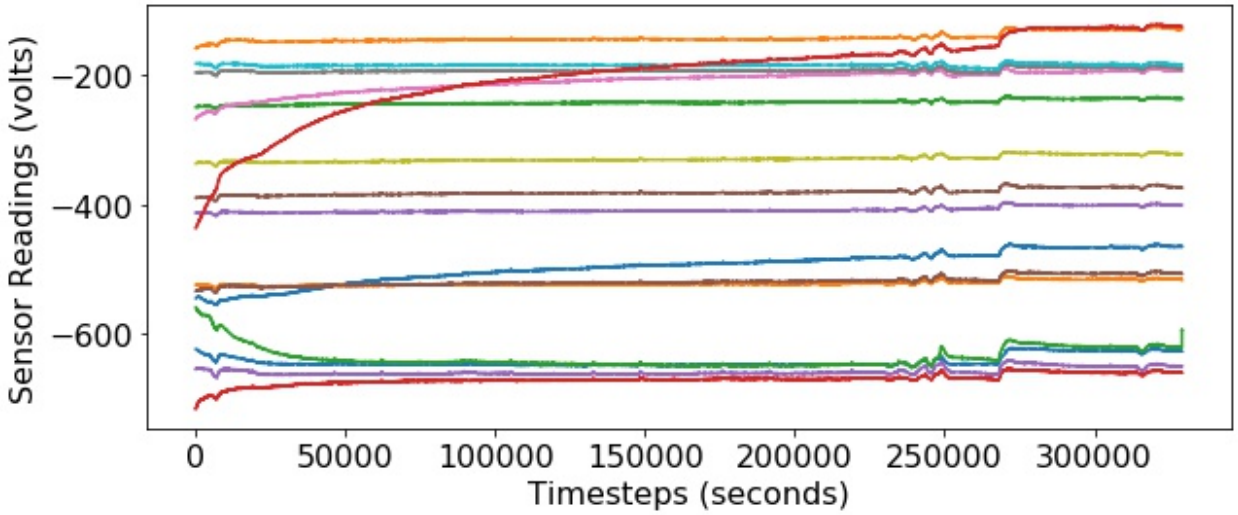

Figure 2: Raw Ground Truth Sensor Data

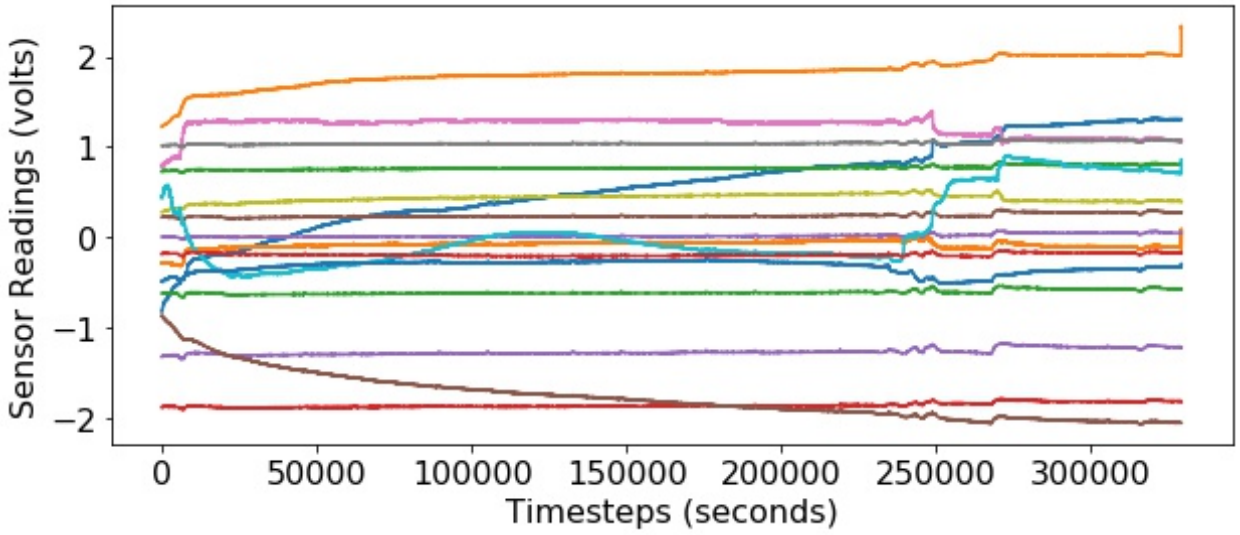

Figure 3: Normalized Uncalibrated Sensor Data

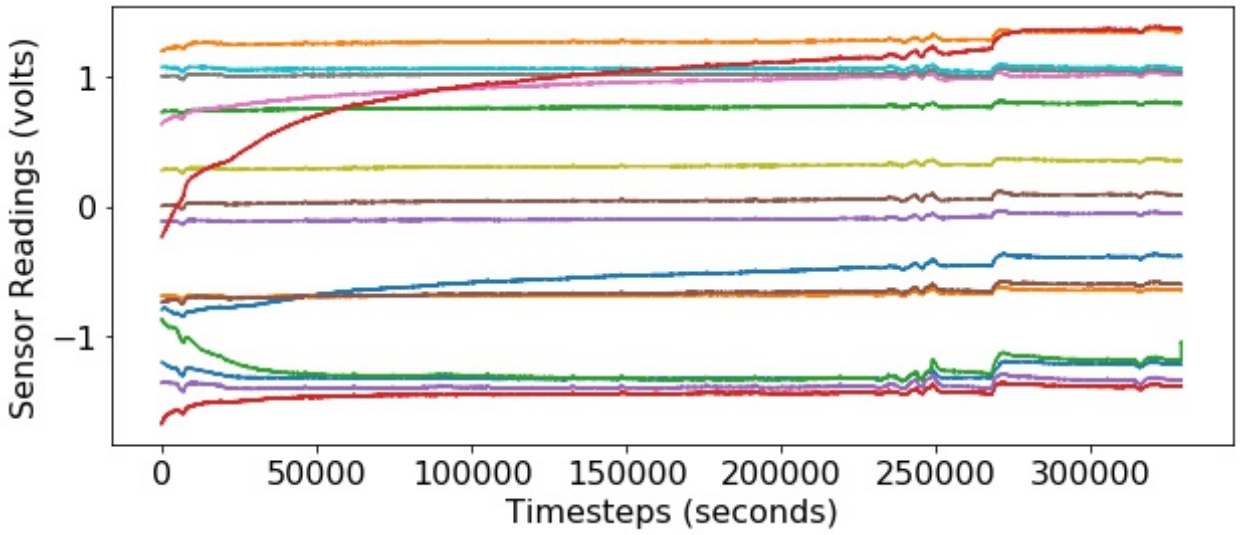

Figure 4: Normalized Ground Truth Sensor Data

As is most noticeable in all Figures above, some of the sensors have erroneous readings in some of the final readings. As a result, the final 500 time steps were dropped, leaving 328,764 usable time steps. From these time steps, the first 300,000 were used as training data and the remaining 28,764 as testing and model assessment data. This gives approximately a 91-9 split between training and testing data. Graphs of the final (normalized and trimmed) training and testing data are shown in Figures 5 and 6, respectively. The vertical line in both Figures represent the split between training and validation data.

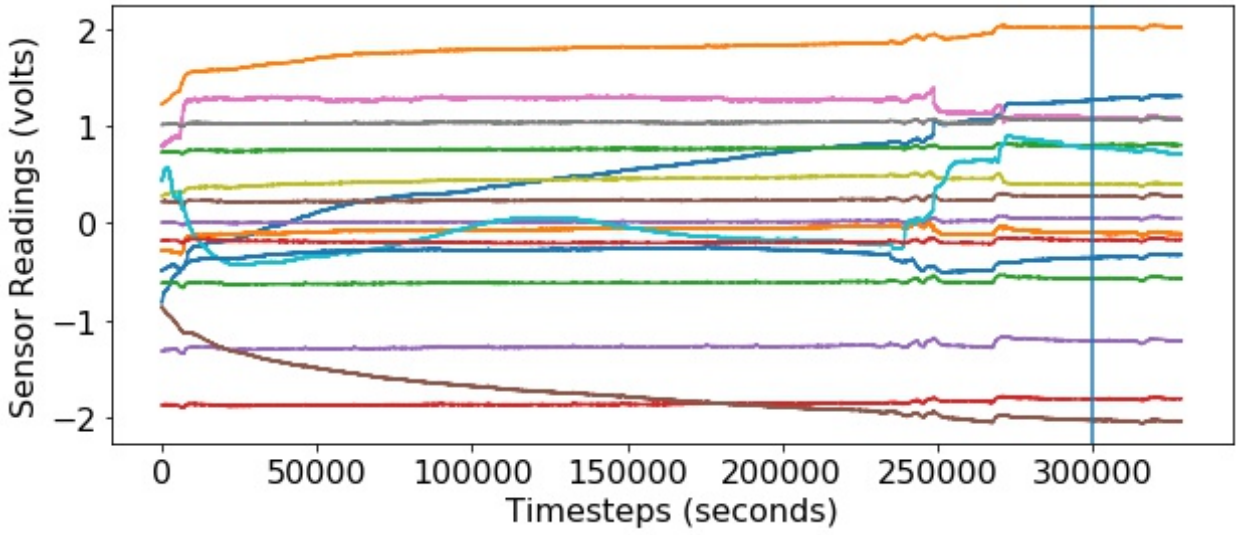

Figure 5: Normalized and Divided Uncalibrated Sensor Data

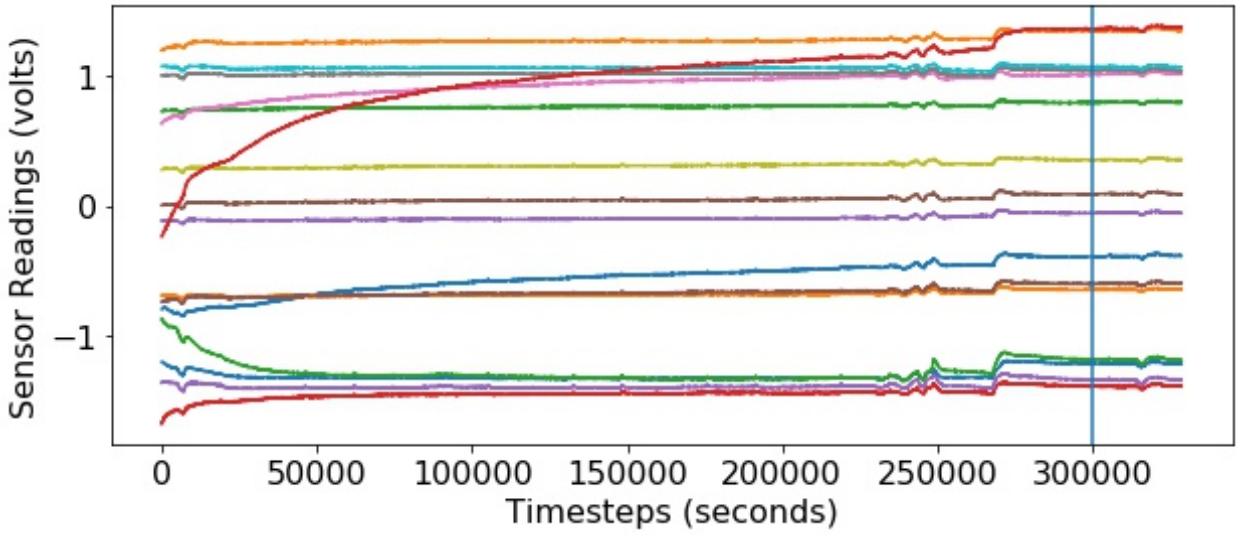

Figure 6: Normalized and Divided Ground Truth Sensor Data

1.4 ENSEMBLE LEARNING

Ensemble learning is a set of techniques used to enhance deep learning by using a collection of hypotheses – or an ensemble of hypotheses – to make predictions on an input. The hypotheses are generated by base models and combined to create the ensemble. The primary goals of ensemble learning are to reduce variance and bias [9, pp. 696-697]. If implemented correctly, ensemble learning should produce results that are better than any individual model. The purpose of this thesis is to explore the processes, procedures, and results of applying Boosted Aggregation (frequently called bagging) and Stacked Generalization (frequently called stacking) – two ensemble learning techniques – to the blind drift calibration problem. The ensemble learning results will be compared to the results of some of the best standalone neural networks

and the efficacy of ensemble learning to solve the blind drift calibration problem will be examined.

2. DIFFERENT TYPES OF NEURAL NETWORKS

This thesis will not delve into the specifics of how each of the presented and discussed types of neural networks function. Instead, a high level overview will be given and then the applicability of the network type in the scope of the presented problem will be discussed.

2.1 OVERVIEW AND COMMONALITIES

There are three main types of neural networks that will be examined and discussed: feedforward neural networks, which are a subset of artificial neural networks (ANN), recurrent neural networks (RNN), and convolutional neural networks (CNN). Each type uses nodes, which will also be called units, to approximate linear and nonlinear functions. Generally, for most networks, a unit receives inputs from one or many sources, computes a weighted sum of all inputs, then applies a nonlinear function to the input. This function is called the activation function. According to the universal approximation theorem, each network must have at least one layer of computational units that utilize a nonlinear activation function in order to be able to approximate a nonlinear function. A network with only two computational layers – one nonlinear layer and one linear layer – can approximate any function to varying degrees of accuracy [9, pp. 751-752].

Each type of network also utilizes the same general structure. All of the aforementioned network types receive inputs through input nodes, which feed the first layer of computational units, and end with output nodes, which are fed by the last layer

of non computational units. How the units are connected to other units (or possibly even themselves) are what cause the networks to differ.

2.2 FEEDFORWARD NETWORKS

Feedforward networks are perhaps the simplest form of neural networks used for deep learning and are directed, acyclic graphs [9, pp. 750-751]. Each computational unit generates an output (as described above) and passes that output along to its successor. That successor could be one or many other computational units, or output nodes.

Feedforward networks, along with all other forms of ANNs, generally process tabular data [10]. Due to the lack of “memory” in a simple feedforward network and the fixed size of the input layer force the networks to only examine a relatively small time window. As a result, feedforward networks miss long-term dependencies and trends that may be present in input data, causing them to generally be bad for predicting sequential data [9, pp. 773]. Since the input data for this problem is sequential, time-stamped data and the goal is to observe long-term trends and variations, feedforward networks would not be a good choice for detecting and correcting for sensor drift. With that being said, for the sake of completeness, the sensor data was fed into a feedforward network so that the outputs could be compared to that of better suited networks.

2.3 RECURRENT NEURAL NETWORKS

The second type of neural network that will be discussed are recurrent neural networks. RNNs are differentiated from ANNs in that RNNs allow for cycles in the computational graph. Each cycle has a delay and an associated timestamp such that each unit of the network can receive some computed (and weighted) value from its previous output. In other words, inputs received at earlier time steps can have an impact on the current time step, giving RNNs an internal state or memory [9, pp. 772-773].

There are a few noteworthy assumptions that must be made in order for an RNN to work. The first is the Markov Assumption, which assumes that the hidden state of a unit, z_t, suffices to capture and cover the information from all previous inputs to the unit [9, pp. 773]. More specifically, units of RNNs are first-order Markov processes, as the current state (output) of a unit at each time step depends only on the state of the previous time step. The state from the previous time step provides enough information to make the current time step conditionally independent of the past time steps [9, pp.463]. That is, $P\left(X_t|X_{0:t-1}\right) = P\left(X_t|X_{t-1}\right)$.

The second assumption that RNNs make regard the update function of the hidden state. Given some arbitrary function, $z_t = f_{z_t}\left(z_{t-1}, x_t\right)$, that takes the previous hidden state and the current output to compute the current hidden state, it can be assumed that this function is a time-homogeneous process. A time-homogeneous process is a process of change that itself does not change and f_{z_t} holds true for all time steps [9, pp. 463, 773]. This allows RNNs to detect long term dependencies that otherwise go undetected by feedforward networks [9, pp. 773]. This is something that is

useful given the nature of the data in the present problem. If the network is able to detect long-term changes, such as sensor drift, it should be able to account and correct for them.

2.4 CONVOLUTIONAL NEURAL NETWORKS

Convolutional neural networks are used to detect and identify patterns in single or multi-dimensional arrays of data. They use what are known as kernels, which are particular patterns that can be used to detect certain features in an input array. Kernels are applied continuously to the input array, separated by a distance known as a stride. The process of applying a kernel to an array is called a convolution [9, pp. 760-761]. Figure 7 demonstrates a kernel of length 4, $[3, -1, 1, 3]$, applied to a one dimensional array, $[5, 1, 9, 3, 6, 2, 9, 10, 3, 0]$, with a stride of 3.

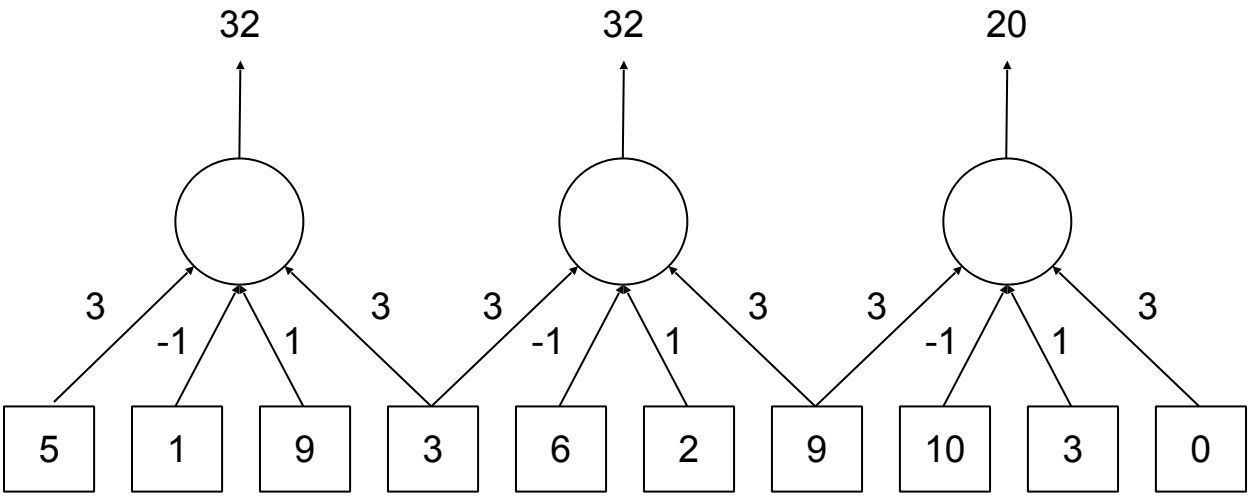

Figure 7: Convolutional Sample

A convolution can be reduced to a simple matrix multiplication problem as well. The corresponding matrix multiplication for Figure 7 would be

$$\begin{pmatrix} 3 & -1 & 1 & 3 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & -1 & 1 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 3 & -1 & 1 & 3 \end{pmatrix} \begin{pmatrix} 5 \\ 1 \\ 9 \\ 3 \\ 6 \\ 2 \\ 9 \\ 10 \\ 3 \\ 0 \end{pmatrix} = \begin{pmatrix} 32 \\ 32 \\ 20 \end{pmatrix}.$$

While CNNs are useful for detecting patterns in one-to-many dimensioned sets of data, their usefulness falls short of being able to make predictions in regression problems. Thus, a CNN would be useful for detecting sudden drifts in sensor data, such as a sensor that was damaged or bumped into, but would not be able to predict what the actual reading of a sensor should be following the detection of said drift. Since the problem at hand is to compensate for sensor drift, not just detect it, a CNN would not prove to be useful for solving the problem at hand.

2.5 SELECTION OF RECURRENT NEURAL NETWORKS

As mentioned above, the data received are sequential sensor readings for calibrated and uncalibrated sensors. The goal is to be able to detect sensor drift over time, correct for it, and make predictions as to what the actual sensor readings should be. Given the information presented in Sections 2.2 - 2.4, it should become clear that RNNs are the best choice for this type of data. Their internal memory through recurrence gives them the ability to detect long-term dependencies and changes that

feedforward networks or other simple ANNs would miss. CNNs are generally optimized for image processing and do not offer any additional benefits over RNNs for regression problems or problems with sequential data.

Although ANNs and feedforward networks might not be useful for making predictions on sequential data with long-distance dependencies and trends, they were still utilized and explored as a part of a stacked ensemble. The next section will discuss the results and findings of these underlying, standalone models.

3. STANDALONE GRU MODEL

3.1 DESIGN

As mentioned in Section 2.5, RNNs are the best choice for the given sensor drift regression problem. As such, one of the two selected models was a Gated Recurrent Unit (GRU). GRUs utilize recurrence, allowing units to use outputs from previous iterations (giving them memory). This allows GRUs to be effective and accurate at detecting long-term trends in sequential data, such as sensor drift. Figure 8 depicts the architecture for the GRU model. The model utilizes an Adam Optimizer.

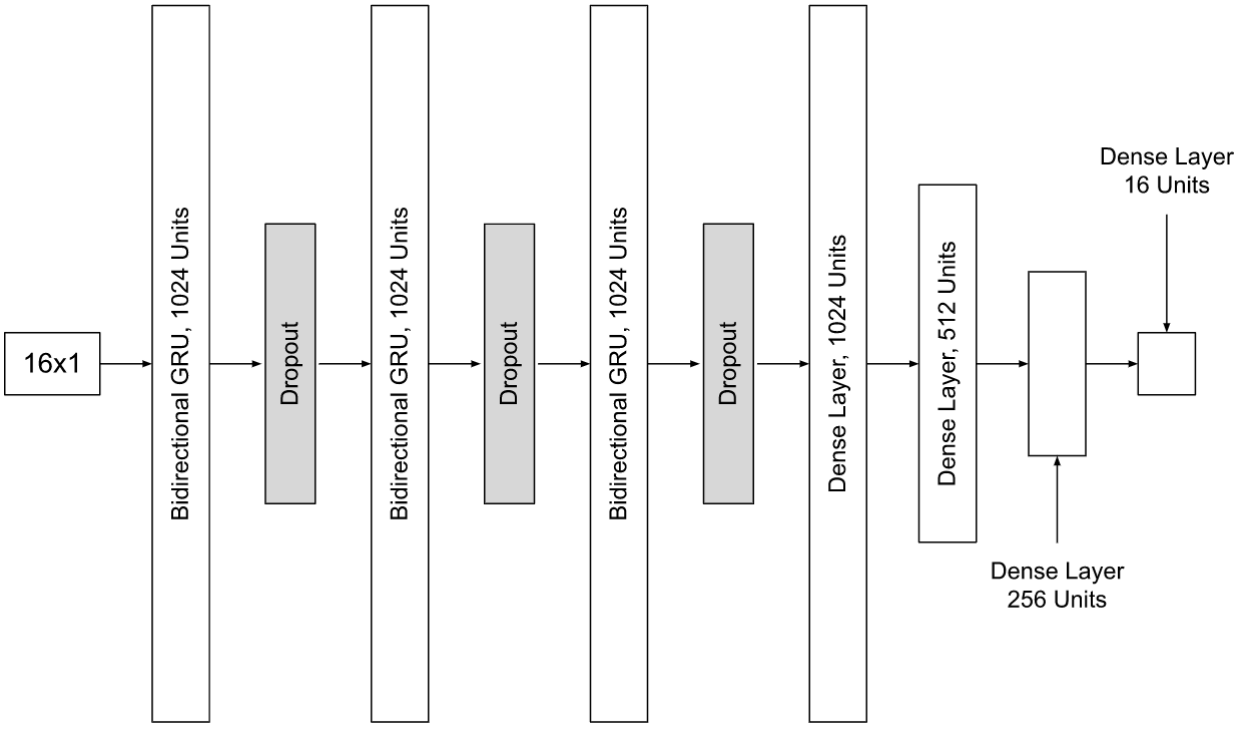

Figure 8: GRU Architecture

The input to the GRU model is a 16×1 dense layer, receiving one time slice of the readings from the 16 uncalibrated sensors at a time. This layer was followed by three bidirectional GRU layers, each of which is followed by a dropout layer. Each of these layers has 1024 units. Following the third dropout layer are four dense layers, which gradually stepped the network down from 1024 units back to 16 units, which gave the desired dimensionality ($16x1$) of the output layer (final dense layer). The activation function for all of the bidirectional GRU layers is ReLU. ReLU, or a rectified linear unit, is a piecewise function defined as $a(x) = max(0, x)$ [9, pp. 752]. The activation function for all of the dense layers was the linear activation function, defined as $a(x) = x$.

3.2 EXPLORING AND SETTING SIGNIFICANT PARAMETERS

In order to achieve a near-optimal model and thus a near optimal solution, independent exploration of several key parameters was conducted. The parameters in question were the number of epochs used to train the model and the learning rate. The first parameter explored was the learning rate, as it is often considered the most important hyper-parameter [11 , pp. 8].

3.2.1 DROPOUT RATE

The purpose of using dropout between layers of a neural network is to reduce overfitting of the model by randomly dropping units in the network. This generally increases the time required to train the network by about two to three times what would normally be required to train the same network that did not have any dropout. This

training time, however, was determined to be worthwhile for the given dataset. Generally, optimal results are achieved when the probability of dropping a unit is set around 0.5 or 50% [12]. In order to balance training time requirements with proper model fit, a dropout rate of 0.4 was chosen.

3.2.2 LEARNING RATE

To determine the best learning rate, identical networks were trained on identical sensor data with varying initial learning rates (parameter input to the optimizer). The default learning rate when using an Adam optimizer, according to the creators of this optimizer, is 0.001 [13, pp. 2]. According to Keras, the default learning rate when using the Adam optimizer is 0.01 [14]. Thus, learning rates of 0.01, 0.001, 1e-05, 1e-06, and 1e-07 were used in an attempt to properly bracket the optimal learning rate. A learning rate of 0.01 fell victim to the exploding gradient problem [15] and did not produce a viable model and thus data from these runs are not presented. Figure 9 shows the validation losses during training for all of the aforementioned GRU models.

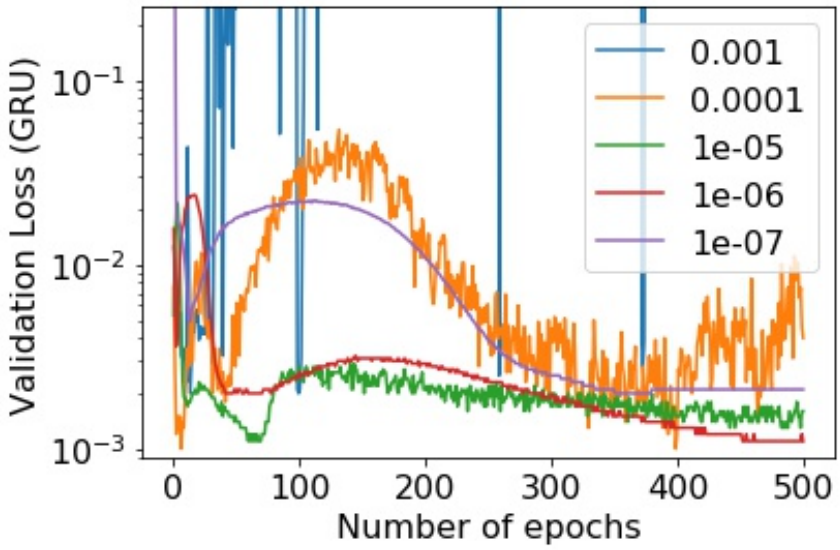

Figure 9: GRU Validation Losses with Varying Learning Rates

As can be seen from this Figure, only three of the models were stable and converged: those with learning rates of 1e-05, 1e-06, and 1e-07. Although the models with learning rates of 1e-5 and 1e-7 had comparable results, with mean squared errors (the main metric used for evaluating a model's performance; MSE) of 0.001177 and 0.001111 respectively, networks trained with higher learning rates require fewer epochs to converge. Thus, training a GRU network with an initial learning rate of 1e-05 is the more optimal choice.

3.2.3 NUMBER OF EPOCHS

Although the main purpose of Figure 9 was to determine the optimal learning rate to be used for the network, the validation loss curves also provide insight into how many epochs the network should be trained for. Looking at Figure 9, it can be seen that the

validation loss begins to level out and converge around 130 epochs. Thus, in order to determine the optimal number of training epochs, a network was trained that saved the model every 25 epochs with the goal of being able to observe the model's performance over relatively short intervals. The MSEs of the model with the above parameters are shown in Table 1. To produce this table, the saved models were used to predict the ground truth values of the uncalibrated sensor values from the holdout testing dataset described in Section 1.3 (same process used to evaluate the overall performance of all models).

Number of Epochs	MSE
25	0.00284
50	0.00156
75	0.00118
100	0.00260
125	0.00238
150	0.00178
175	0.00161
200	0.00167
225	0.00148
250	0.00146
275	0.00125
300	0.00114
325	0.00112
350	0.00116
375	0.00108
400	0.00113
425	**0.00105**
450	0.00111

Table 1: MSEs of the GRU Model Every 25 Epochs

As can be seen from this table, the lowest MSE was achieved when the model was trained for 425 epochs. Thus, it was determined that the optimal number of epochs to train the network was 425.

It is worth noting that Keras has a callbacks API exclusively designed to optimize the training process. For instance, one of the callbacks is the EarlyStopping callback, which ends training when certain conditions are met such as an increase to the validation loss. Using such a callback would eliminate the need to experimentally determine the optimal number of epochs a network should be trained for. Although this callback is highly tunable, getting the parameters right proved to be more cumbersome and less effective than manually determining this value. For instance, consider the 1e-05 curve from Figure 9. This curve is decreasing until around epoch 25, increasing from epoch 26 until around epoch 45, and then steadily decreasing until converging around epoch 130. In order for the EarlyStopping callback to be effective in this instance, the patience parameter would have to be set to at least 20 epochs. The patience parameter requires that the condition prescribed by the callback be met for N epochs before training is stopped [16]. For example, in this case, $N = 20$ would require that the validation loss increases for at least 20 epochs before training is stopped. As will be seen from other iterations and other models, the validation loss can increase a great deal over the course of 20 epochs as the model begins to become overfit. Thus, the use of the EarlyStopping callback was determined to be ineffective for the construction of these models.

3.3 RESULTS

The best results with the underlying GRU model were achieved during the test run to determine the proper number of epochs to use while training. The parameters for that run, including parameters that were not tweaked nor optimized for, are shown in

Table 2. Parameters that are asterisked are discussed in subsections above and are optimized. The remaining parameters were set and left at the recommended defaults [14].

Parameter	Value
Dropout Rate*	0.4
Learning Rate*	1e-05
Number of Epochs*	425
Beta 1	0.9
Beta 2	0.999
Epsilon	1e-08
Decay	0
Batch Size	128

Table 2: Underlying GRU Model Parameters

These parameters were used to obtain the lowest MSE seen in Table 1 of 0.00105. These were the best results obtained from a standalone GRU model, including models used for exploring significant parameters as well as several other iterations of training the GRU model aimed specifically at obtaining the best results. Graphical representations of the predictions of the GRU model are shown in Figures 10 through 25 below. In all cases, the uncalibrated sensor reading is shown in blue, the calibrated is shown in green, and the model's prediction is shown in orange.

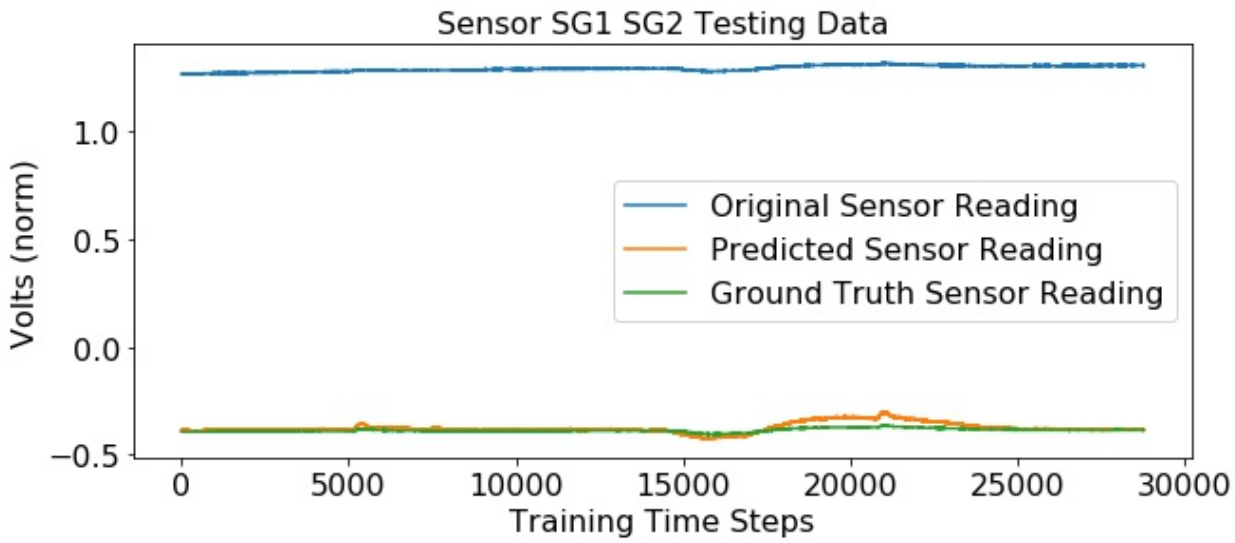

Figure 10: GRU Results for Sensors 1 and 2

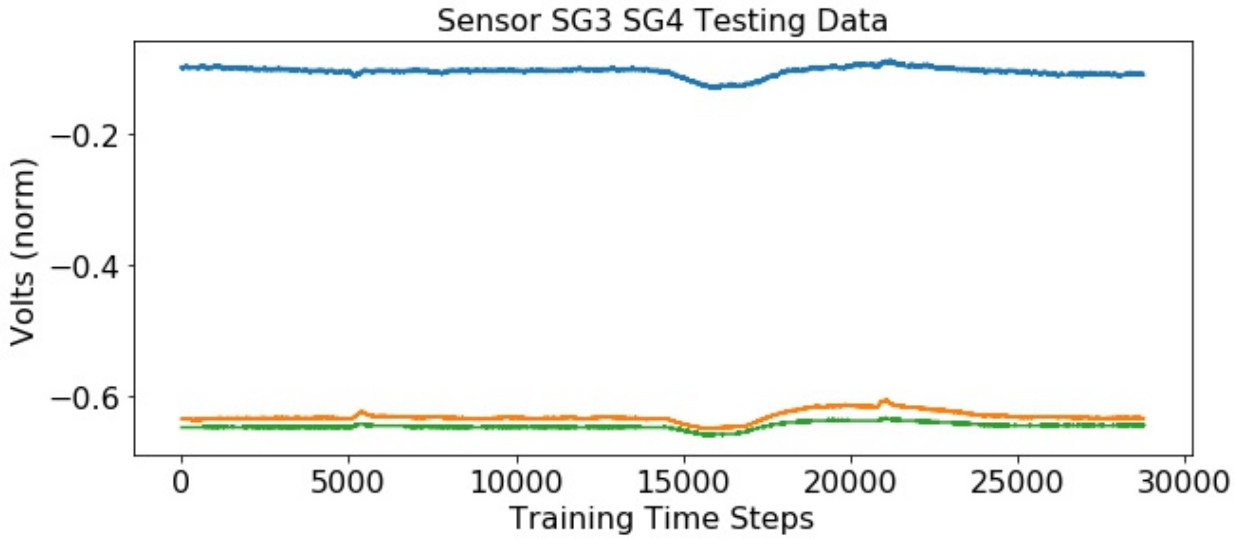

Figure 11: GRU Results for Sensors 3 and 4

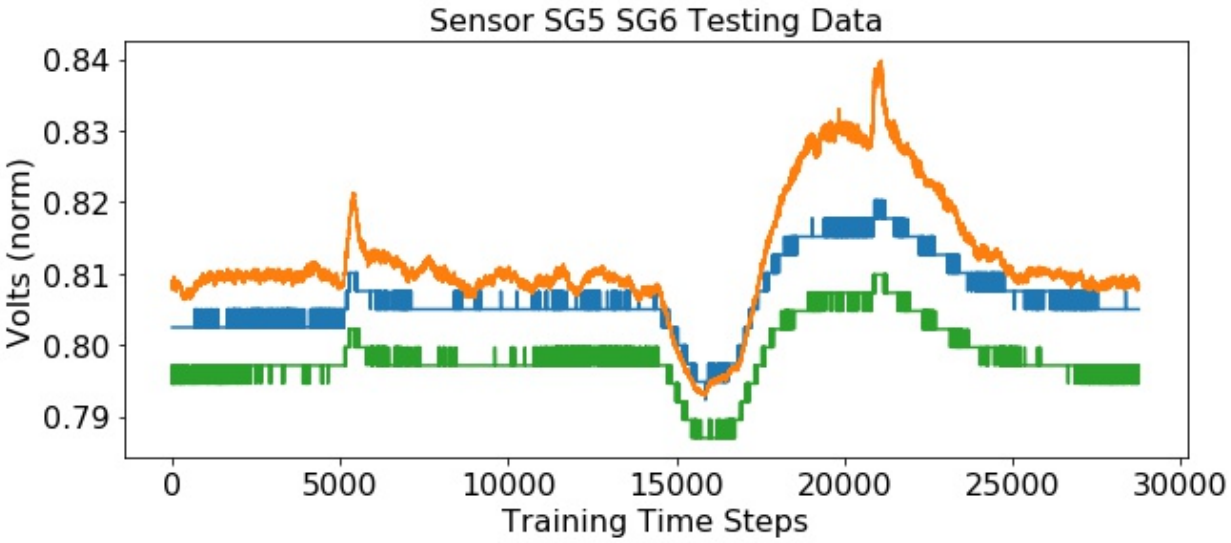

Figure 12: GRU Results for Sensors 5 and 6

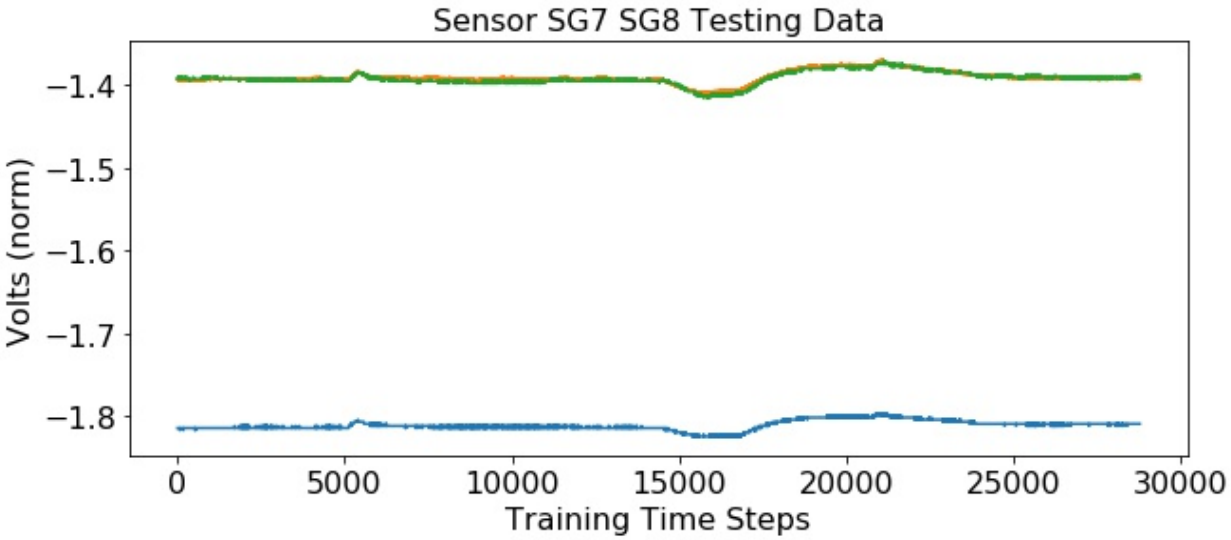

Figure 13: GRU Results for Sensors 7 and 8

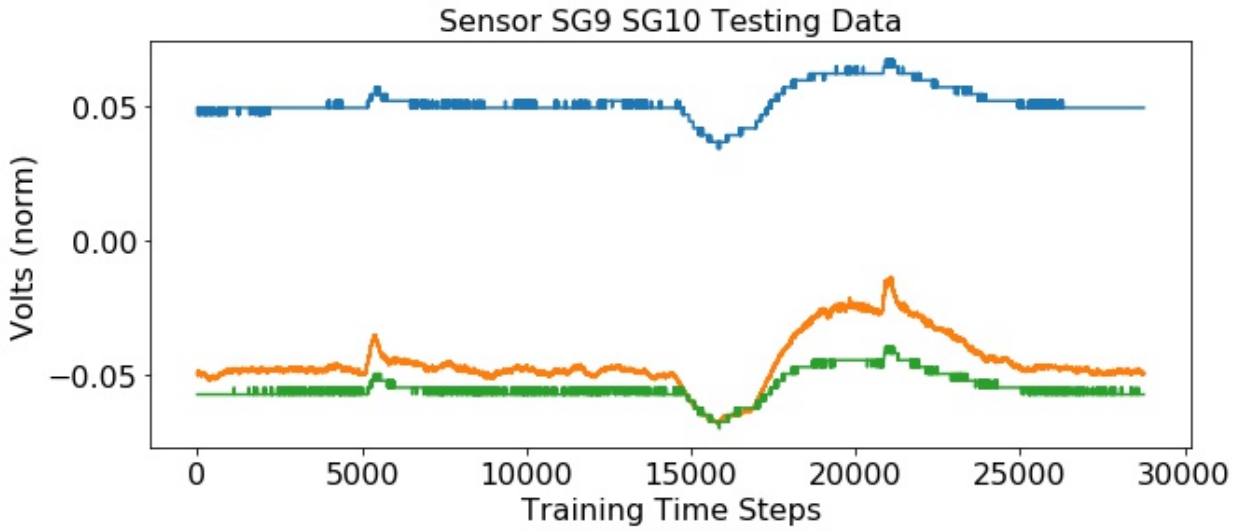

Figure 14: GRU Results for Sensors 9 and 10

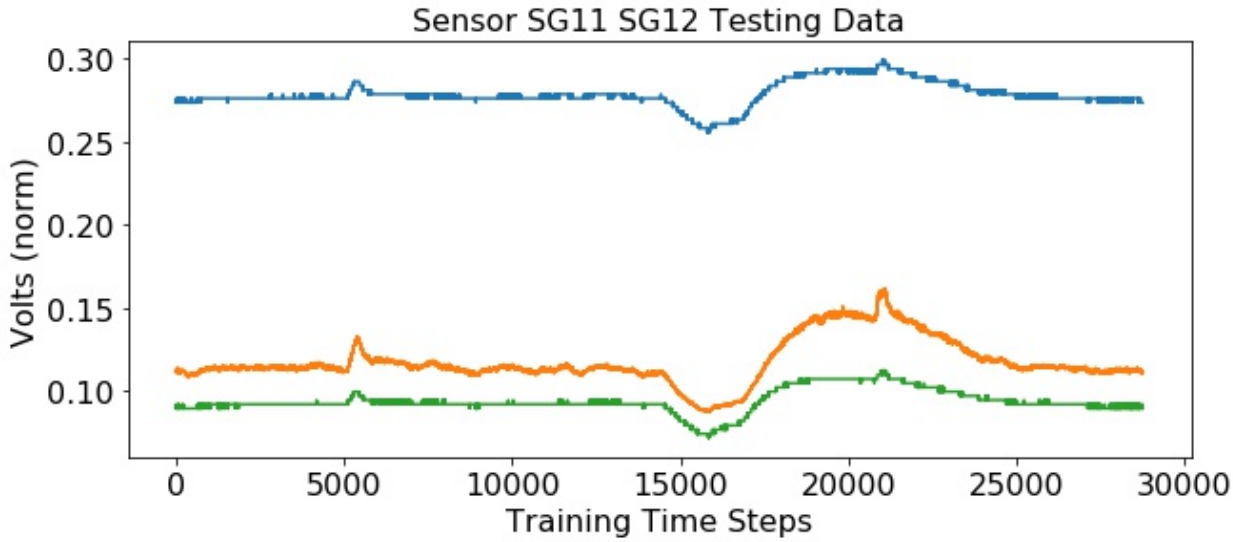

Figure 15: GRU Results for Sensors 11 and 12

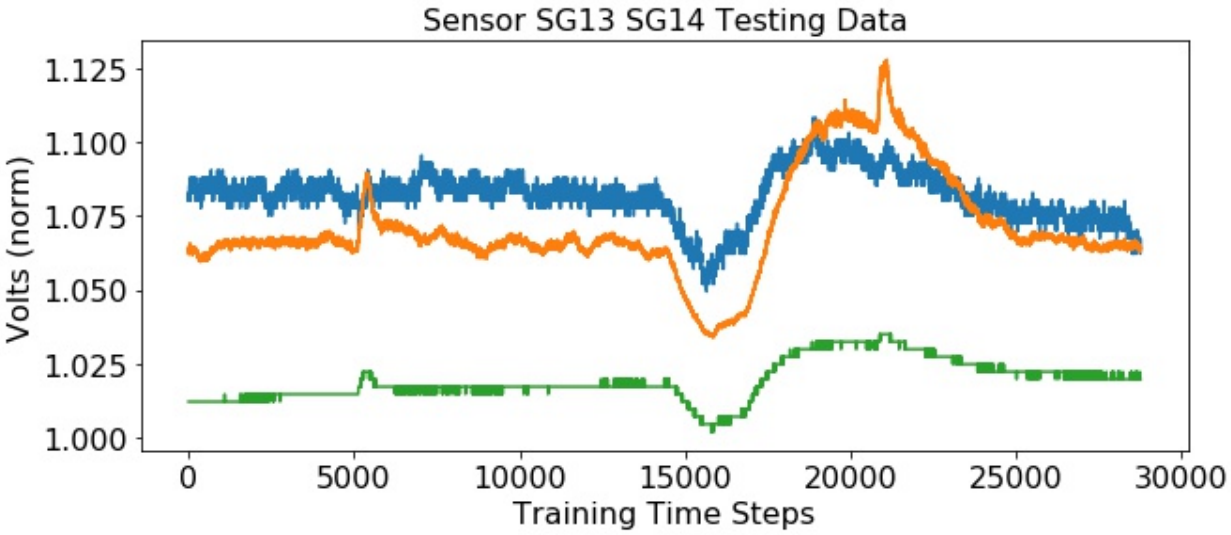

Figure 16: GRU Results for Sensors 13 and 14

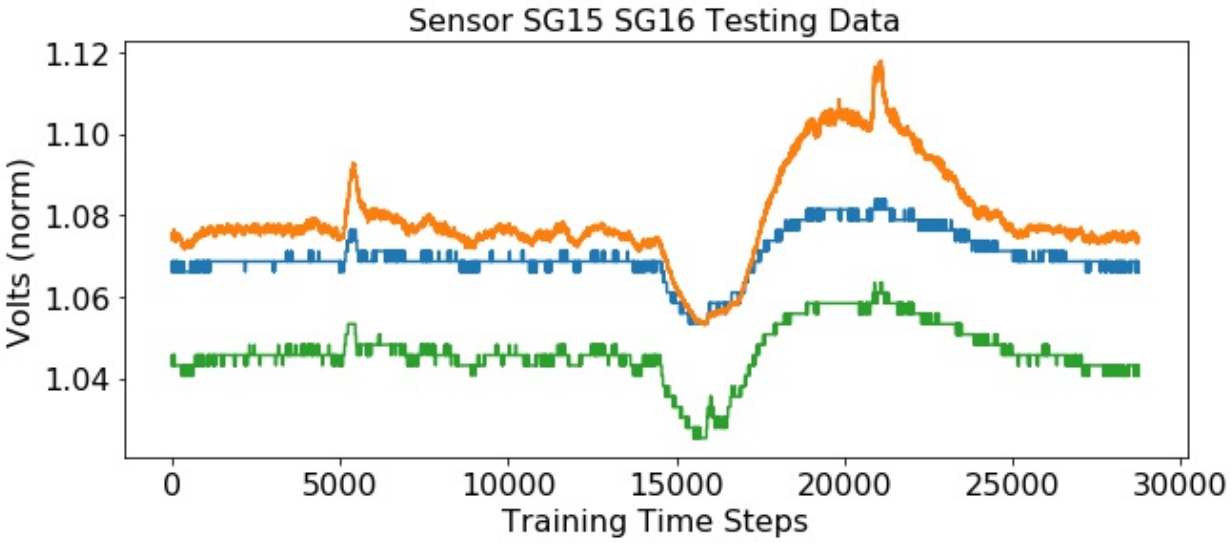

Figure 17: GRU Results for Sensors 15 and 16

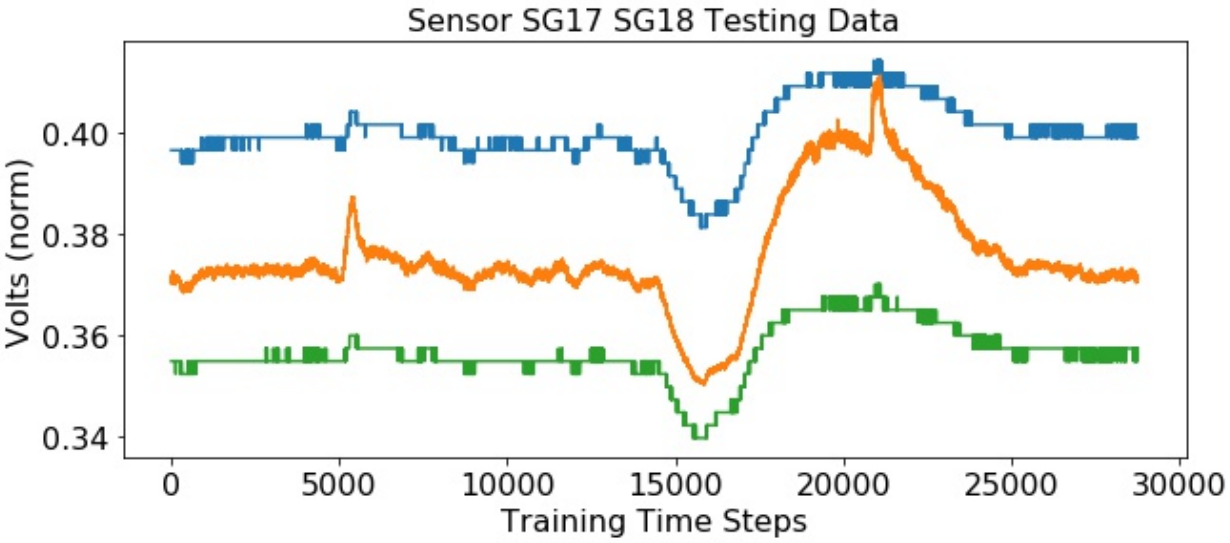

Figure 18: GRU Results for Sensors 17 and 18

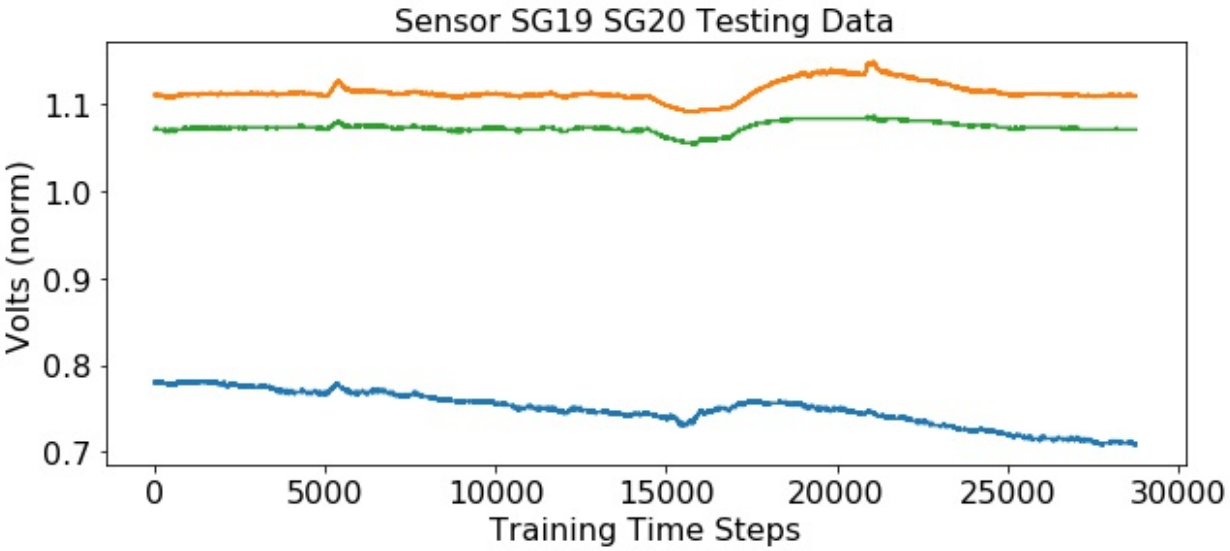

Figure 19: GRU Results for Sensors 19 and 20

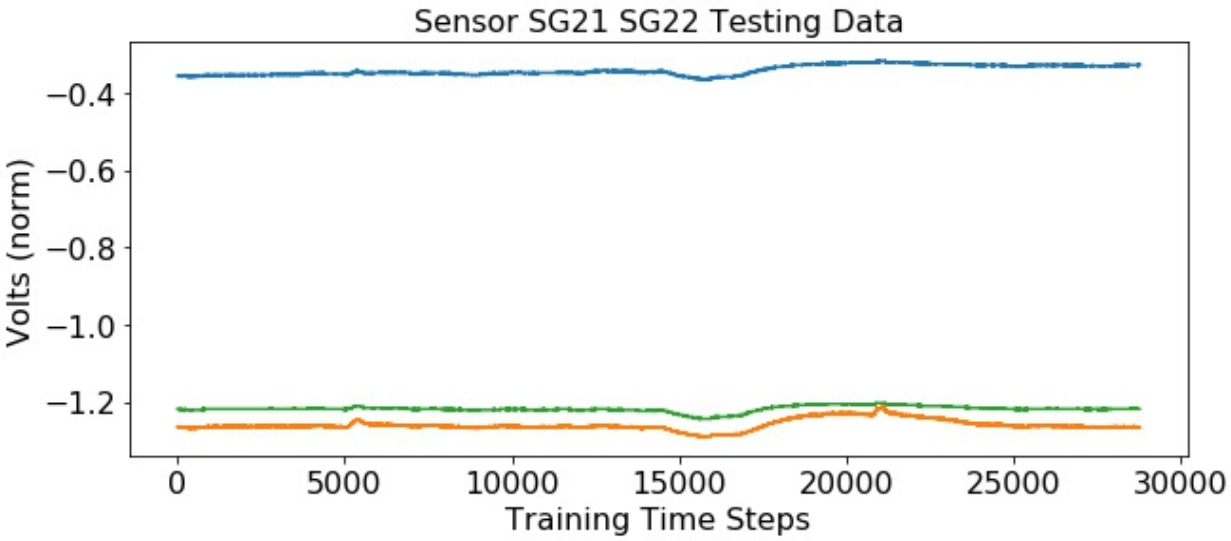

Figure 20: GRU Results for Sensors 21 and 22

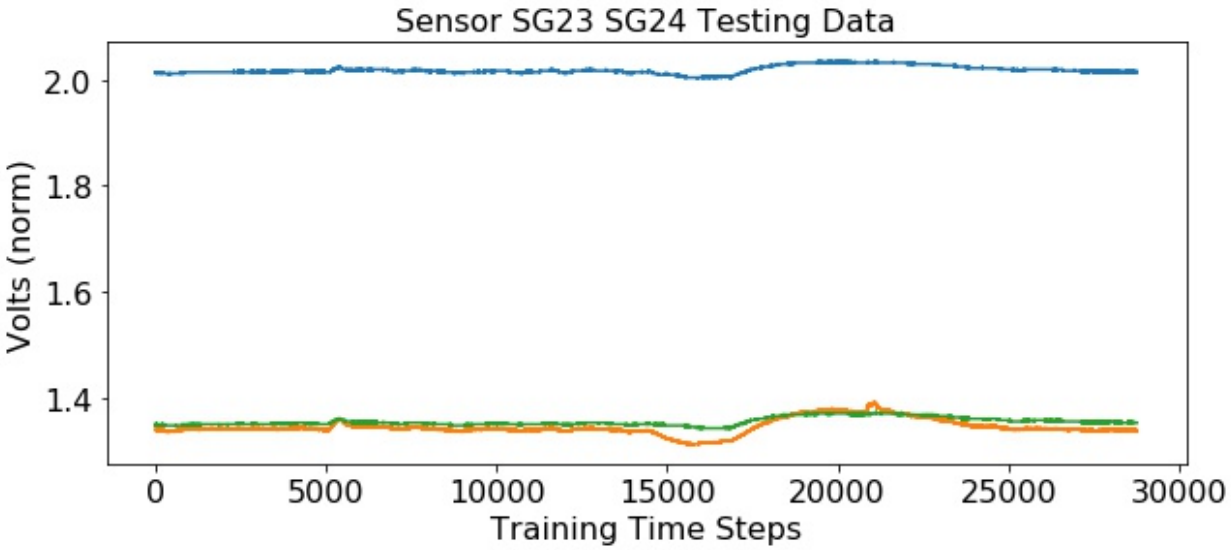

Figure 21: GRU Results for Sensors 23 and 24

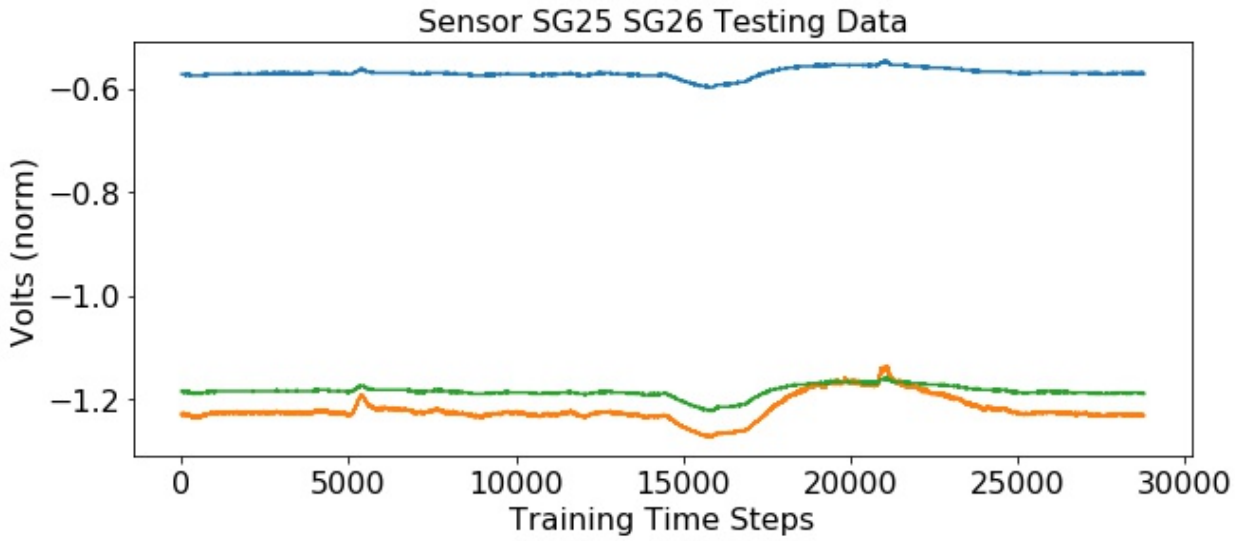

Figure 22: GRU Results for Sensors 25 and 26

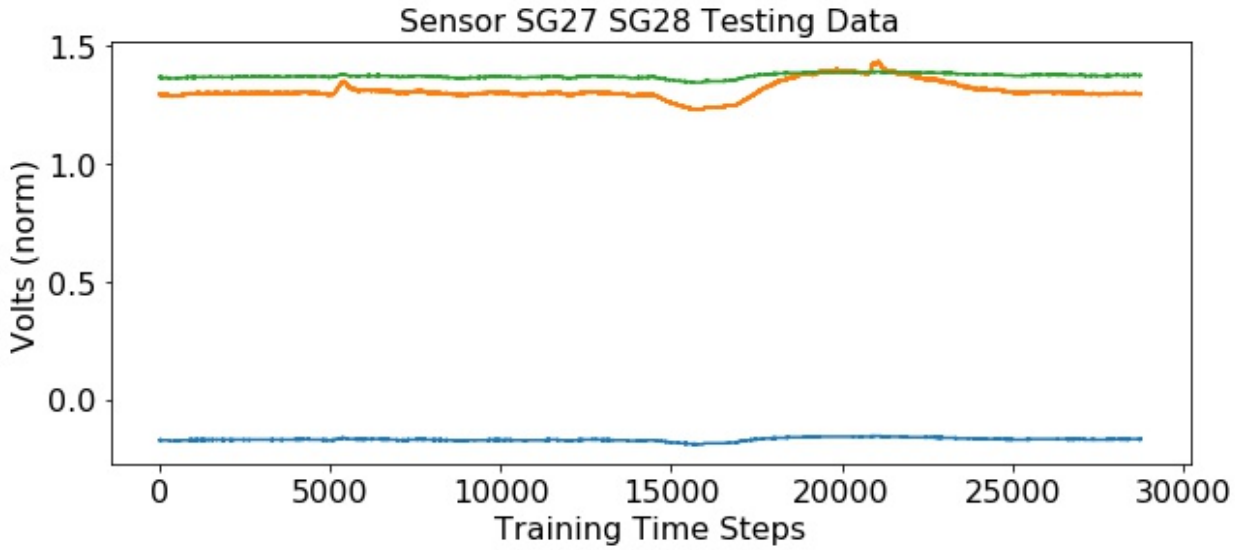

Figure 23: GRU Results for Sensors 27 and 28

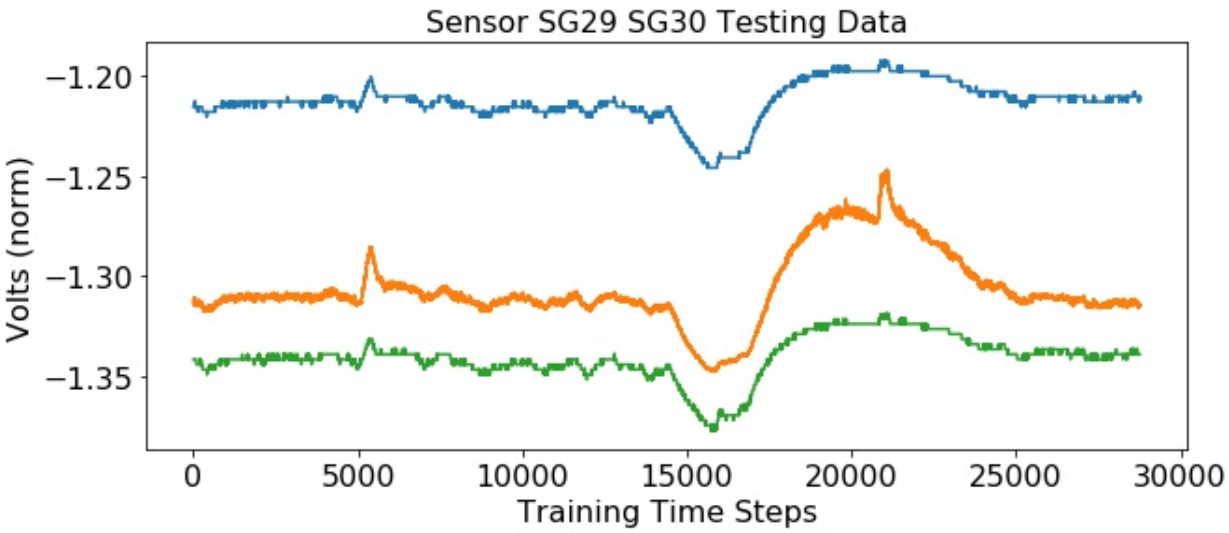

Figure 24: GRU Results for Sensors 29 and 30

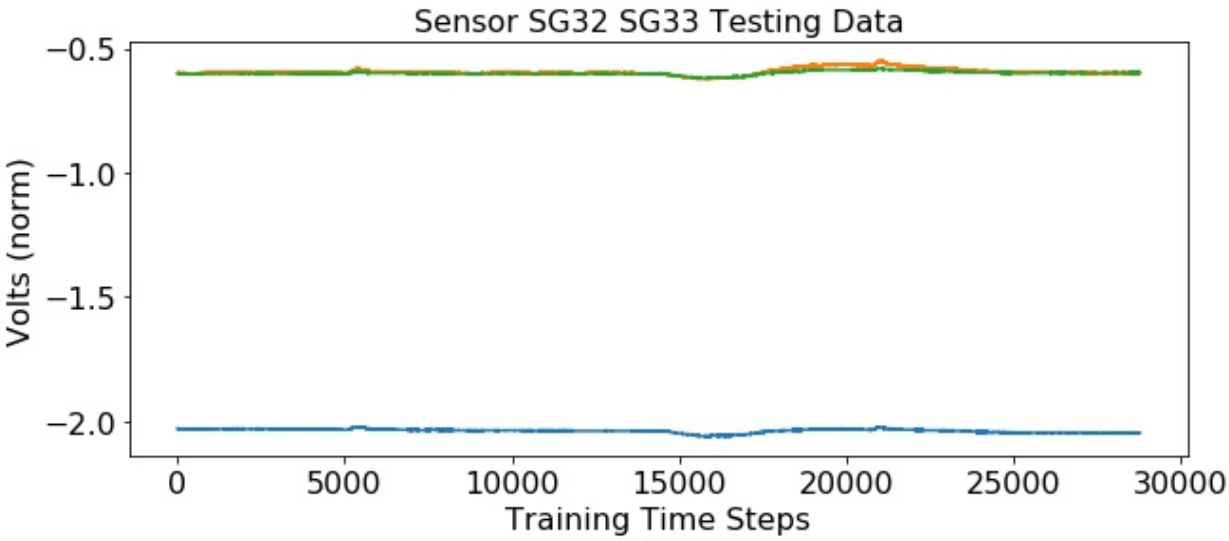

Figure 25: GRU Results for Sensors 32 and 33

3.4 OBSERVATIONS AND DISCUSSION

The GRU model had varying levels of success predicting the ground truth sensor values for the uncalibrated sensors, with predictions for some sensors being rather good, and others being not as great. Based on Figures above, the predictions for sensors 1, 3, 7, 23, and 32 appear to be very good (shown in Figures 10, 11, 13, 21, and 25, respectively). The overall shape of the predicted sensor curve very closely resembles that of the ground truth sensor curve, and the predicted values are consistently very close to the ground truth values. The predictions for sensors 9, 11, 17, 19, 21, 25, 27, and 29 (shown in Figures 14, 15, 18, 19, 20, 22, 23, and 24, respectively) are slightly worse, but still pretty good. The predicted sensor curve is somewhat close to that of the ground truth sensor curve, and the model correctly predicted drift in the correct direction. In other words, if the uncalibrated sensor was reporting voltages that were too high, the GRU network was able to detect this drift and predict values that were lower. The predictions for sensors 5, 13, and 15 (Figures 12, 16, and 17, respectively) were noticeably worse than the others. While the network generally output a predicted sensor curve that was roughly the same shape as the ground truth curve, the network incorrectly predicted the direction of the sensor drift. That is, if the uncalibrated sensor was reporting voltages that were too high, the GRU network would output predictions that were even higher.

Although the network made some less-than-stellar predictions for a couple of the sensors, the overall results of the GRU network were pretty good. The overall MSE was fairly low (0.00105) and most of the predicted sensor values were very close to the

ground truth sensor values. There were, however, a couple of noteworthy issues with the GRU network.

The first has already been touched on, but will be discussed again here in more depth: the network made rather poor predictions for a couple of sensors. The predictions shown in Figures 12, 16, and 17 – for sensors 5, 13, and 15 respectively – are either nearly the same as the uncalibrated sensor or, on occasion, worse than that of the uncalibrated sensor. In these instances, it would be hard to make a case for using the GRU model for recovering the ground truth sensor values from the uncalibrated sensor values. With that being said, though, the model recovered the ground truth sensor values accurately for 8 of the sensors, and nearly perfectly recovered the ground truth sensor values for 5 of the sensors. This means the GRU model made the outputs for 13 of the 16 sensors better, while the outputs for 3 sensors were either slightly worse or left relatively unchanged. This is still a good ratio.

The other noteworthy issue with the GRU model can be seen to varying degrees in nearly every Figure depicting sensor predictions above. When the data was being collected in the lab, there appears to have been some sort of disruption or disturbance that took place at around time steps 5,500 and 21,000 in the testing data. For nearly every sensor, the model overcompensated for this uptick in sensor voltage. This spike can most easily be seen in the predictions for sensors 13 and 15, in Figures 14 and 15, respectively. Although this overcompensation is less than ideal, it was more subtle for sensors the model more accurately predicted. That is, it is only incredibly noticeable and likely to be somewhat destructive for three of the sensors: 5, 13, and 15. Additionally, this overcompensation only appears to happen when all sensors

experience an uptick in the uncalibrated sensor values. In other words, the model correctly predicts other spikes in the uncalibrated sensor values so long as that spike is only shown in one or several of the sensors but not all of them. This, coupled with the logic provided above for the other issue leads to the same conclusion: while this overcompensation is not ideal, it does not appear to be an issue with a high likelihood of occurrence nor a high impact on the sensor predictions.

3.5 CONCLUSION

The overall efficacy of the GRU model was high. The MSE of the model was somewhat low (0.00105 volts for nearly 30,000 predicted voltages) and the predictions were quite good for 13 of the 16 sensors. While the model did fall victim to a small overcompensation issue discussed above, the impact and likelihood of occurrence were both determined to be low, causing this to be somewhat of a non-issue. The GRU model appears to be fairly well suited to recovering the ground truth sensor values from the uncalibrated sensor values. Any model that outperforms the GRU model will be a noteworthy and worthwhile improvement.

4. STANDALONE LSTM MODEL

4.1 DESIGN

As mentioned in Section 2.5, RNNs are the best choice for the given sensor drift regression problem. As such, the second selected model was Long Short-Term Memory (LSTM), another one of the more popular RNNs available. LSTMs utilize recurrence, allowing units to use outputs from previous iterations (giving them memory). This allows LSTMs to be effective and accurate at detecting long-term trends in sequential data, such as sensor drift. Figure 26 depicts the architecture for the LSTM model. The model utilizes an Adam Optimizer.

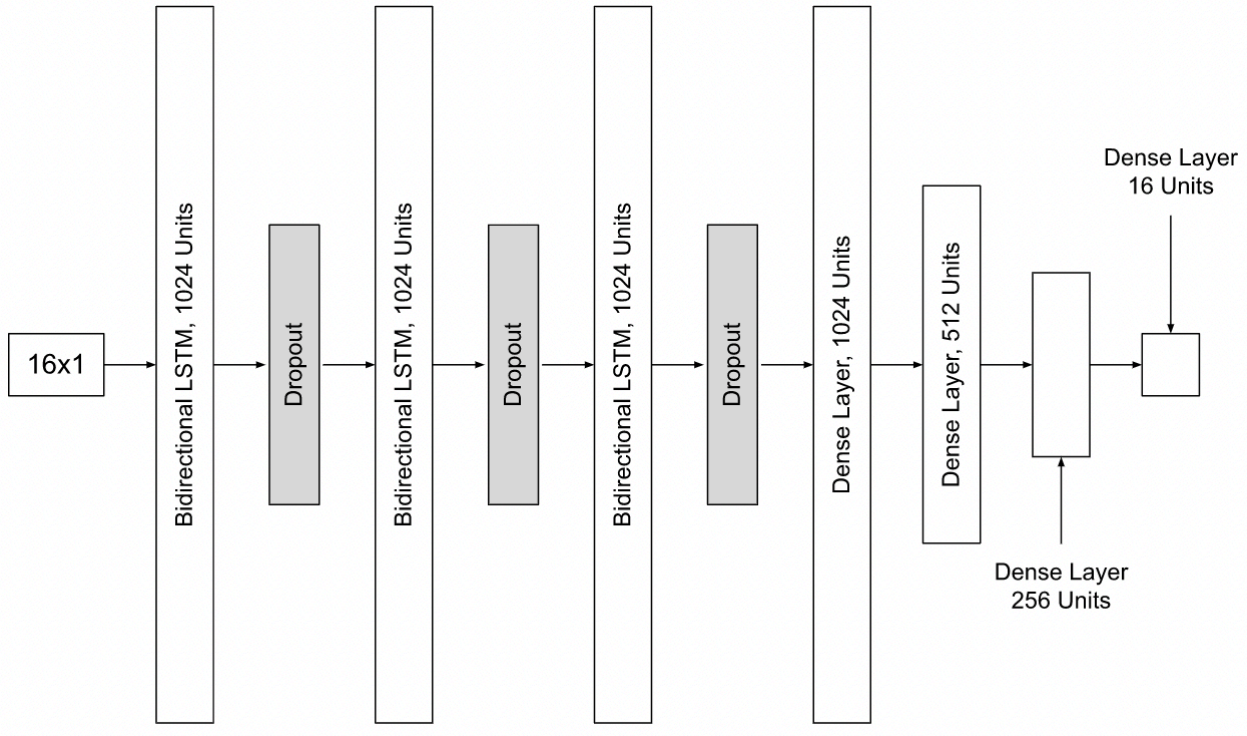

Figure 26: LSTM Architecture

The input to the LSTM model is a 16×1 dense layer, receiving one time slice of the readings from the 16 uncalibrated sensors at a time. This layer was followed by three bidirectional LSTM layers, each of which is followed by a dropout layer. Each of these layers has 1024 units. Following the third dropout layer are four dense layers, which gradually stepped the network down from 1024 units back to 16 units, which gave the desired dimensionality ($16x1$) of the output layer (final dense layer). The activation function for all of the bidirectional LSTM layers is ReLU. ReLU, or a rectified linear unit, is a piecewise function defined as $a(x) = max(0, x)$ [9, pp. 752]. The activation function for all of the dense layers was the linear activation function, defined as $a(x) = x$.

4.2 EXPLORING AND SETTING SIGNIFICANT PARAMETERS

In order to achieve a near-optimal model and thus a near optimal solution, independent exploration of several key parameters was conducted. The parameters in question were the number of epochs used to train the model and the learning rate. The first parameter explored was the learning rate, as it is often considered the most important hyper-parameter [11 , pp. 8].

4.1.1 DROPOUT RATE

Following the exact same logic as presented in Section 3.2.1, a dropout rate of 0.4 was used for the dropout layer between each of the bidirectional LSTM layers. Once again, this was done to balance proper fit with training time requirements.

4.2.2 LEARNING RATE

To determine the best learning rate, identical networks were trained on identical sensor data with varying initial learning rates (parameter input to the optimizer). As mentioned in Section 3.2.2, the default learning rate recommended when using an Adam optimizer is 0.001, while Keras recommends a default learning rate of 0.01. Thus, learning rates of 0.01, 0.001, 1e-05, 1e-06, and 1e-07 were used in an attempt to properly bracket the optimal learning rate. Just like the GRU model with a learning rate of 0.01, the LSTM model with a learning rate of 0.01 fell victim to the exploding gradient problem and did not produce a viable model. The data from this model will not be presented. Figure 27 shows the validation losses during training for all of the aforementioned LSTM models.

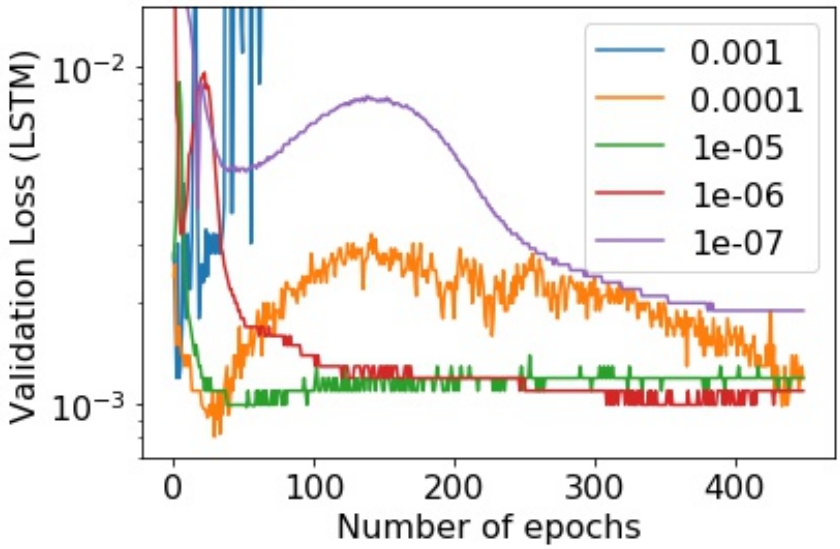

Figure 27: LSTM Validation Losses with Varying Learning Rates

The results for the LSTM networks are similar to those of the GRU networks, as the networks with learning rates of 0.001 and 0.0001 are unstable and do not converge (though the model with a learning rate of 0.0001 does get close to achieving convergence). The networks with learning rates of 1e-05, 1e-06, and 1e-07 all converged. The results for all three viable networks were also comparable, with MSEs of 0.00154, 0.00122, and 0.00213 for the learning rates of 1e-5, 1e-6 and 1e-7, respectively. Although the MSE for the 1e-06 model is the lowest, the results of the 1e-05 and 1e-07 were likely negatively affected by overfitting and underfitting, respectively. Looking at the shape of the 1e-06 graph, it would appear that 400 epochs is near optimal for that learning rate. Thus, for the same reason cited in Section 3.2.2 for the GRU network (higher learning rate requires fewer epochs), a learning rate of 1e-05 was selected for LSTM networks as well.

4.2.3 NUMBER OF EPOCHS

The shape of the 1e-05 curve in Figure 27 is quite a bit different (LSTM) from its GRU counterpart, shown in Figure 9. From this curve, it can be seen that the validation loss is relatively stable from 50 epochs until around 160 epochs, where the validation loss starts to climb as the model becomes overfit. Thus, an LSTM model was trained for 450 epochs that saved the model every 25 models with the goal being able to observe the models performance to determine the optimal number of epochs to be used for training the LSTM models. The MSEs of the model with the above parameters are shown in Table 3. To produce this table, the saved models were used to predict the ground truth values of the uncalibrated sensor values from the holdout testing dataset

described in Section 1.3 (same process used to evaluate the overall performance of all models).

Number of Epochs	MSE
25	0.00780
50	0.00271
75	0.00223
100	0.00179
125	0.00153
150	0.00143
175	0.00137
200	0.00130
225	0.00123
250	0.00121
275	0.00122
300	**0.00119**
325	0.00121
350	0.00125
375	0.00128
400	0.00128
425	0.00136
450	0.00142

Table 3: MSEs of the LSTM Model Every 25 Epochs

As can be seen from the table above, the lowest MSE was achieved after 300 epochs. Thus, it was determined that the optimal number of epochs to train the LSTM model was 300. This is a noticeable difference from the GRU model described in Section 3, which achieved its best performance when trained using 425 epochs. It is also worth noting for the LSTM model that the use of callbacks to determine the optimal number of epochs to train the model would not be effective in this case. Figure 28 shows the validation losses from the training of the model depicted in Table 3.

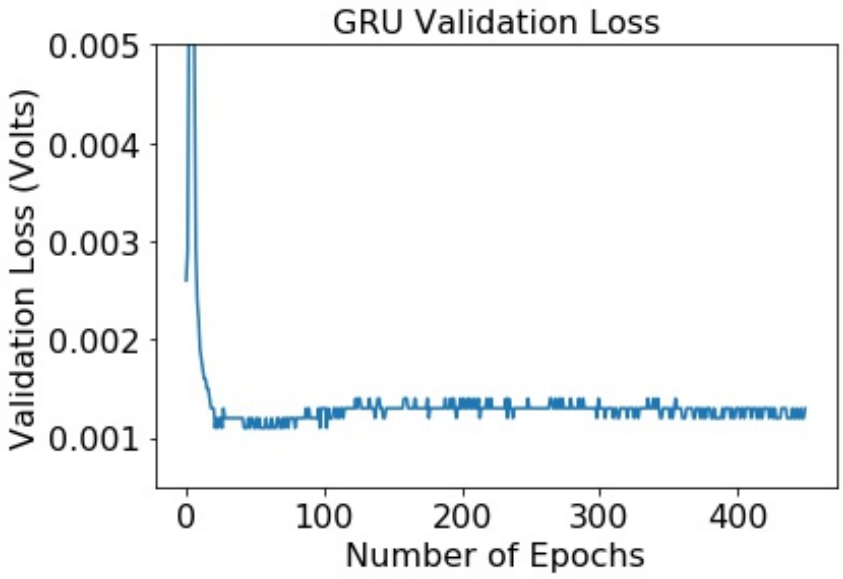

Figure 28: LSTM Training Validation Losses

As can be seen from this Figure, the validation loss for the LSTM model is slightly more stable than its GRU counterpart, but is still somewhat unstable. Utilizing

the EarlyStopping callback would require setting the patience parameter so high it would cease to be productive (explanation given in Section 3.2.3).

4.3 RESULTS

The best results with the underlying LSTM model were achieved during the test run to determine the proper number of epochs to use while training. The parameters for that run, including parameters that were not tweaked nor optimized for, are shown in Table 4. Parameters that are asterisked are discussed in subsections above and are optimized. The remaining parameters were set and left at the recommended defaults. Except for the number of epochs, the parameters used for the LSTM model are exactly the same as the parameters used for the GRU model.

Parameter	Value
Dropout Rate*	0.4
Learning Rate*	1e-05
Number of Epochs*	300
Beta 1	0.9
Beta 2	0.999
Epsilon	1e-08
Decay	0
Batch Size	128

Table 4: Underlying LSTM Model Parameters

These parameters were used to obtain the lowest MSE seen in Table 3 of 0.00119. These were the best results obtained from a standalone LSTM model, including models used for exploring significant parameters as well as several other iterations of training the LSTM model aimed specifically at obtaining the best results. Graphical representations of the predictions of the LSTM model are shown in Figures 29 through 44 below. Once again, the uncalibrated sensor reading is shown in blue, the calibrated is shown in green, and the model's prediction is shown in orange.

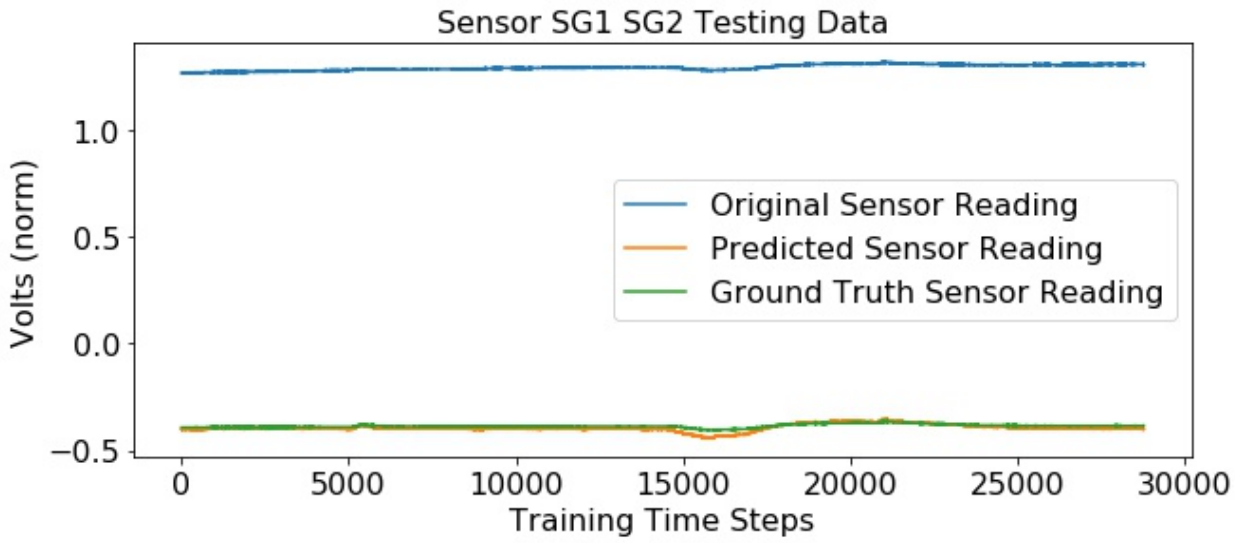

Figure 29: LSTM Results for Sensors 1 and 2

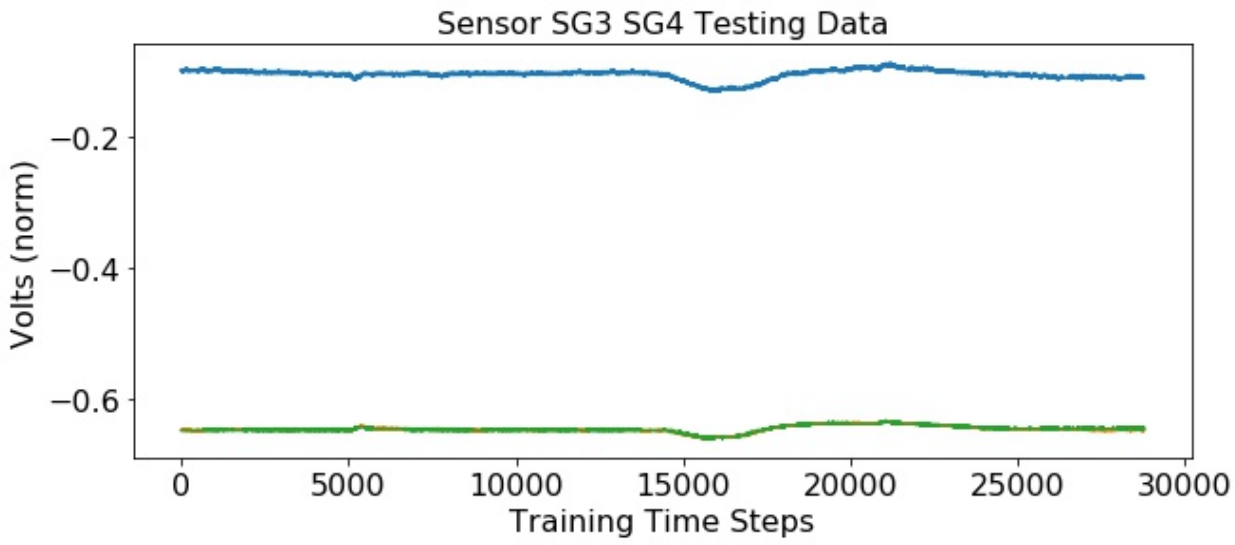

Figure 30: LSTM Results for Sensors 3 and 4

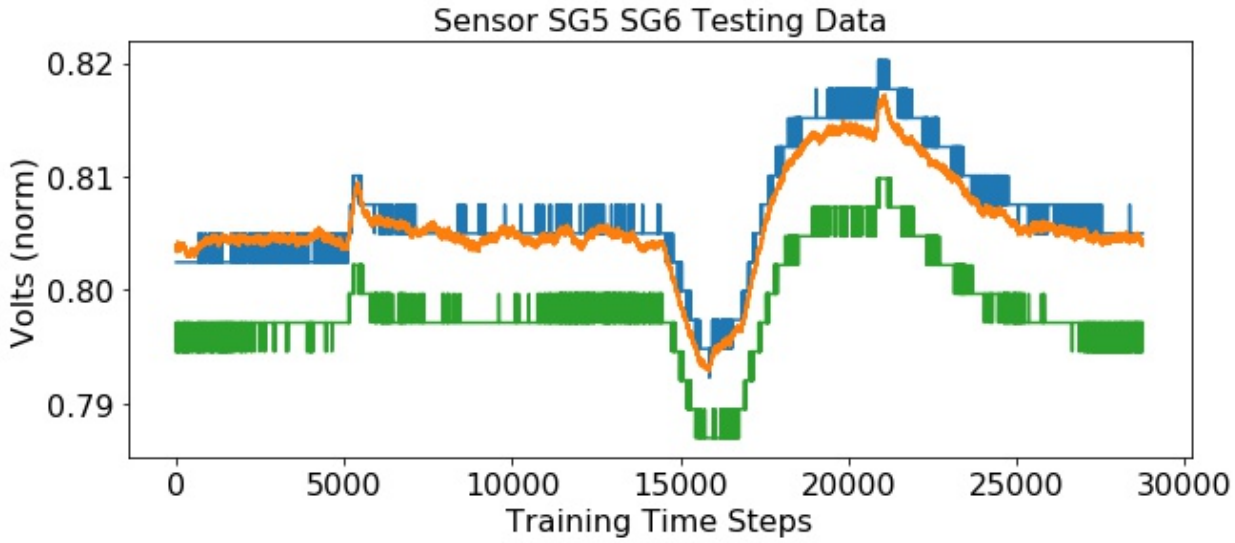

Figure 31: LSTM Results for Sensors 5 and 6

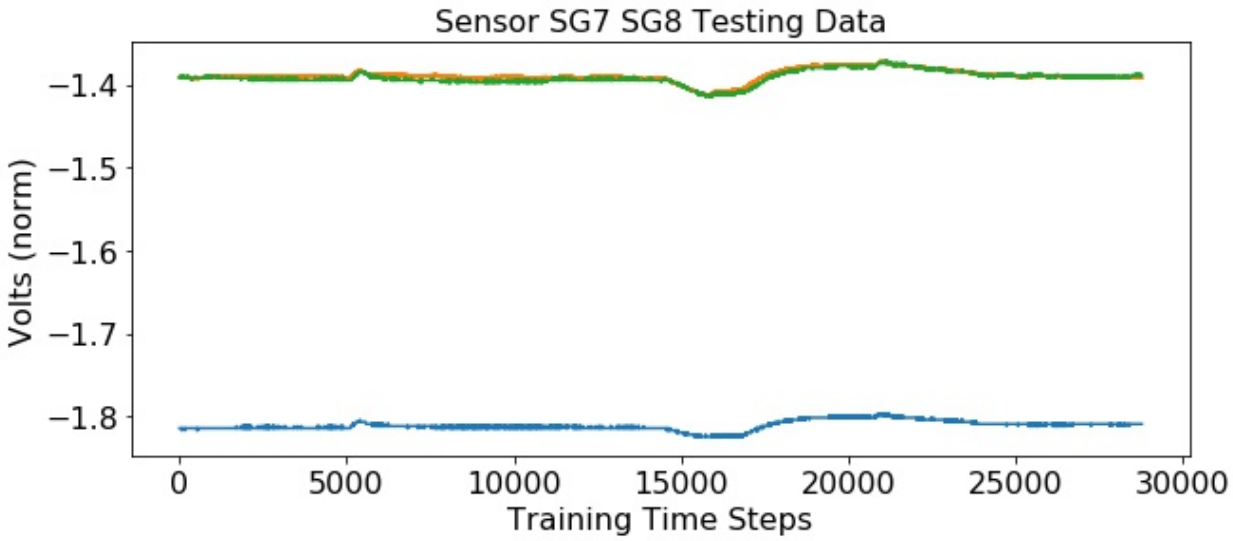

Figure 32: LSTM Results for Sensors 7 and 8

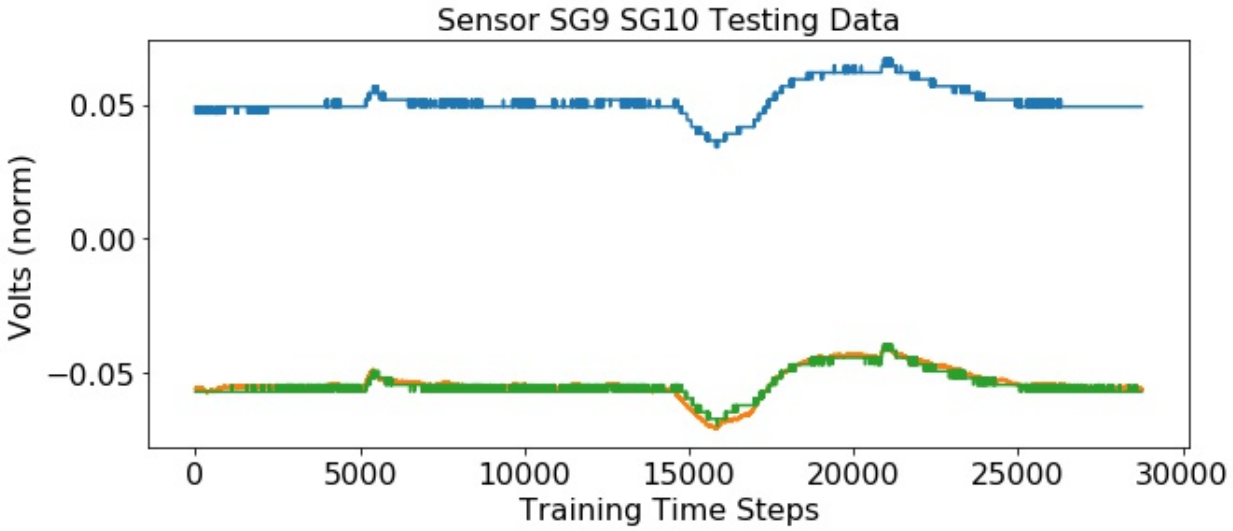

Figure 33: LSTM Results for Sensors 9 and 10

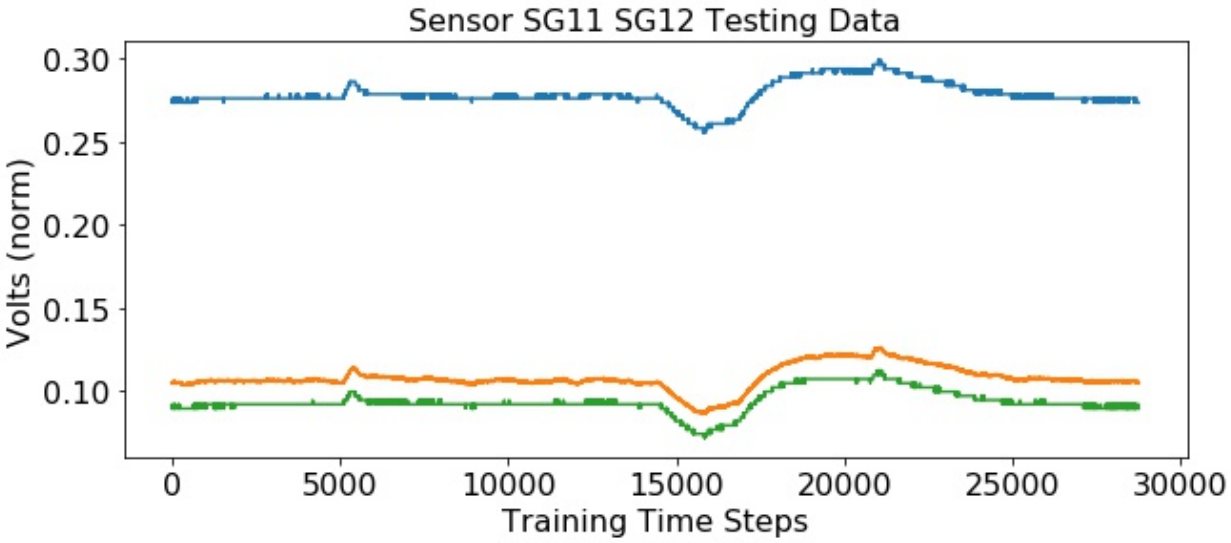

Figure 34: LSTM Results for Sensors 11 and 12

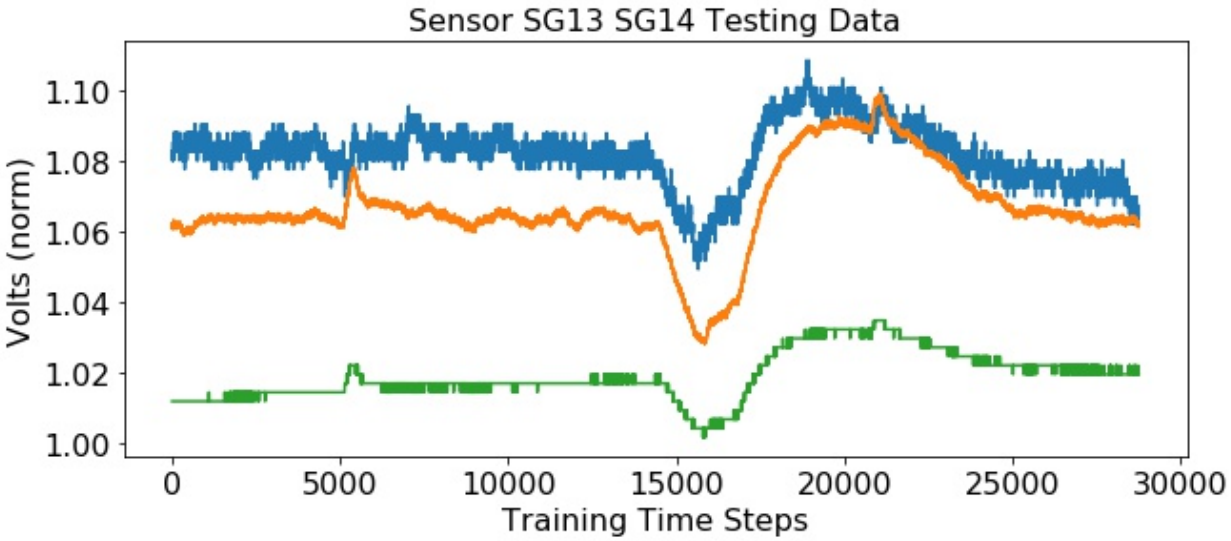

Figure 35: LSTM Results for Sensors 13 and 14

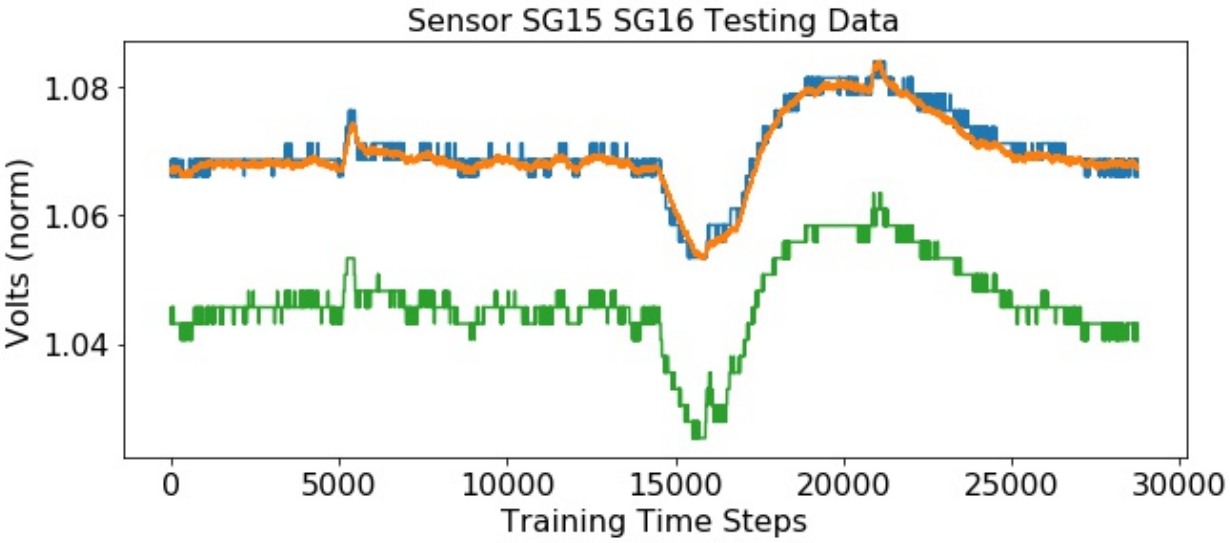

Figure 36: LSTM Results for Sensors 15 and 16

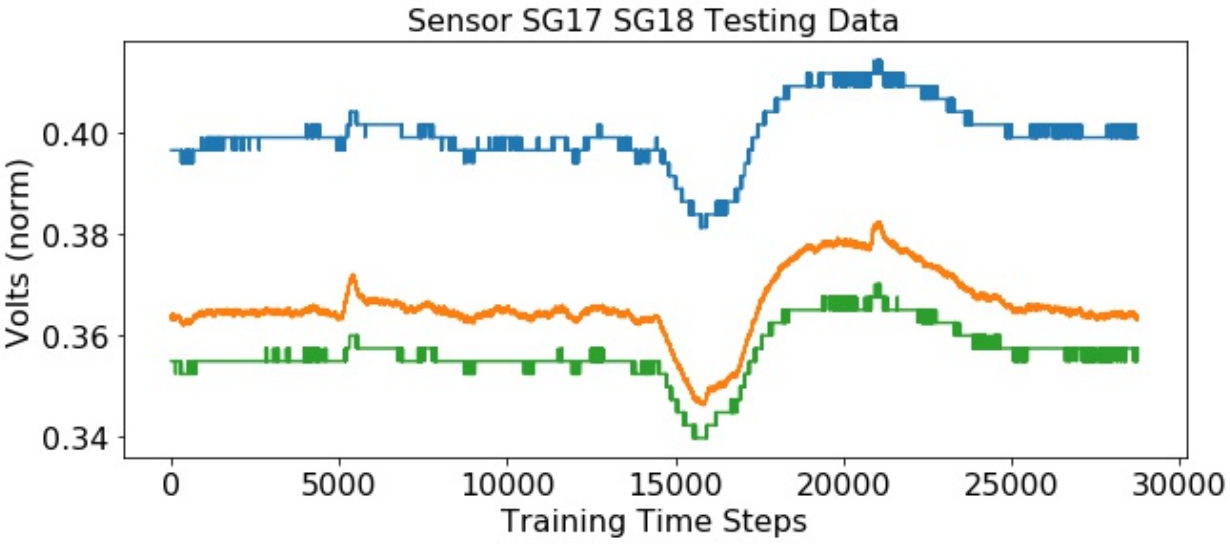

Figure 37: LSTM Results for Sensors 17 and 18

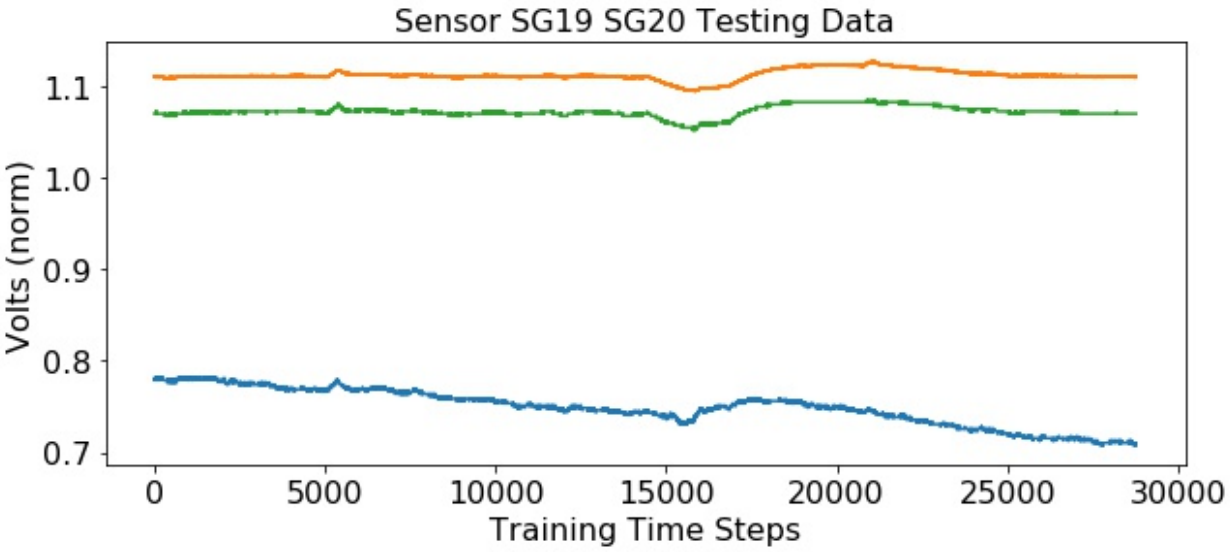

Figure 38: LSTM Results for Sensors 19 and 20

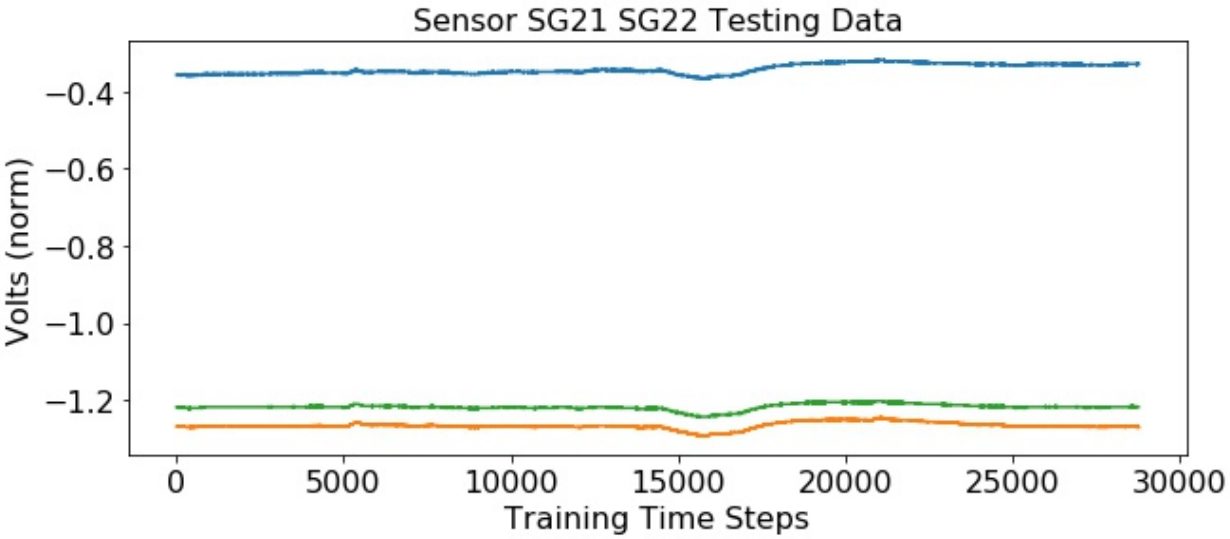

Figure 39: LSTM Results for Sensors 21 and 22

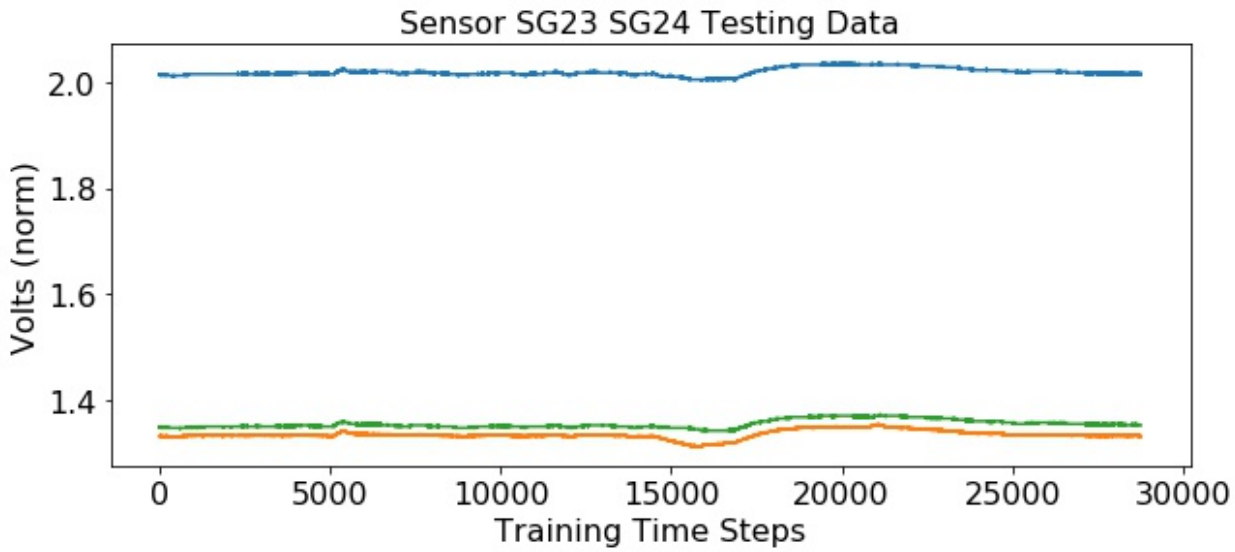

Figure 40: LSTM Results for Sensors 23 and 24

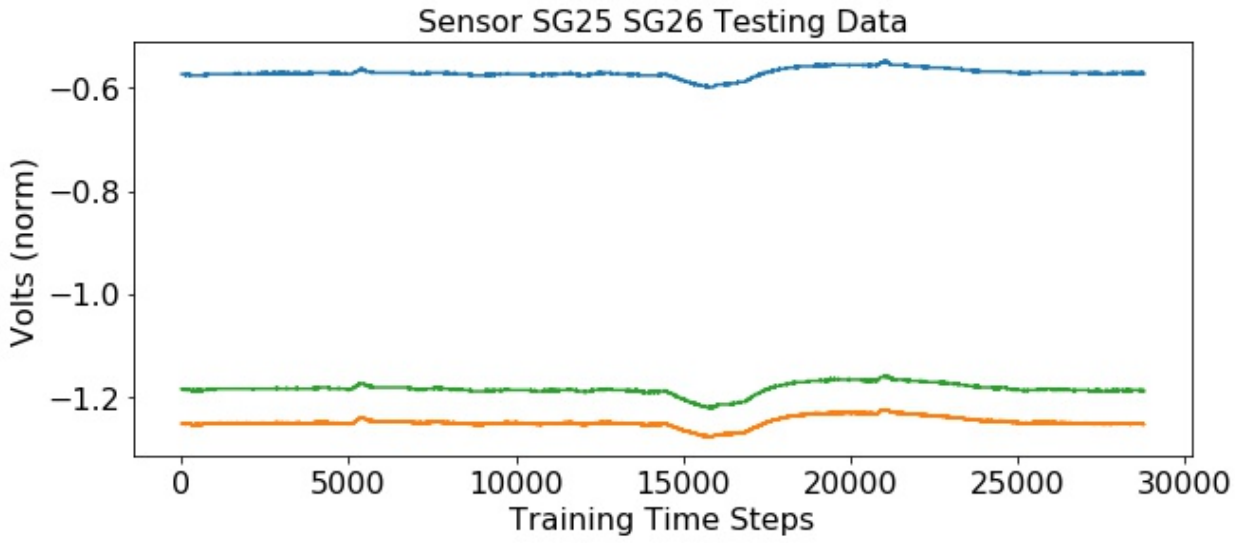

Figure 41: LSTM Results for Sensors 25 and 26

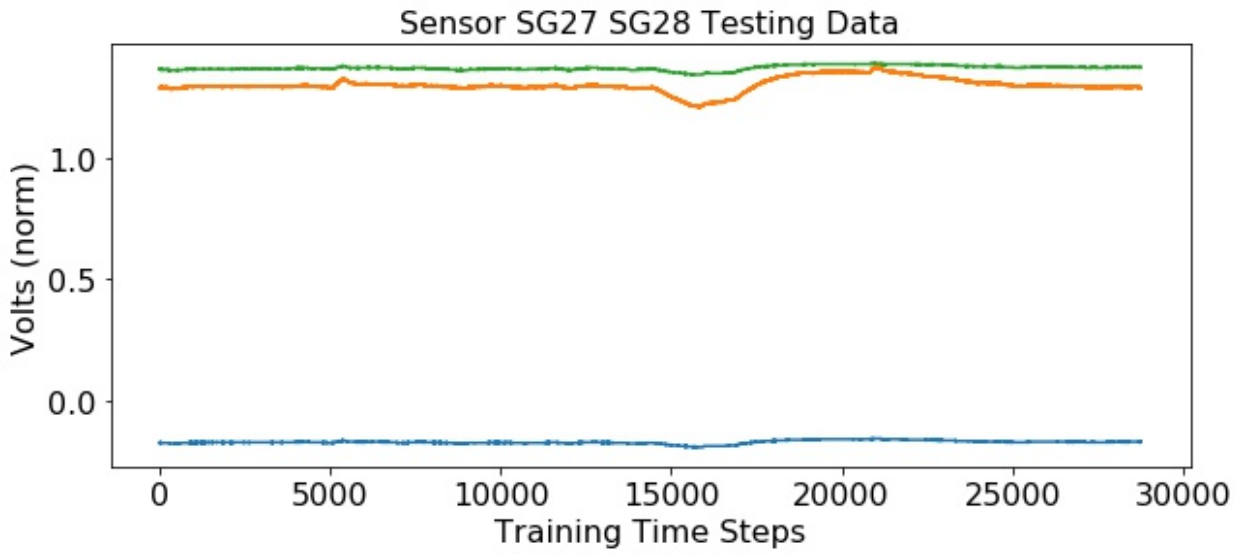

Figure 42: LSTM Results for Sensors 27 and 28

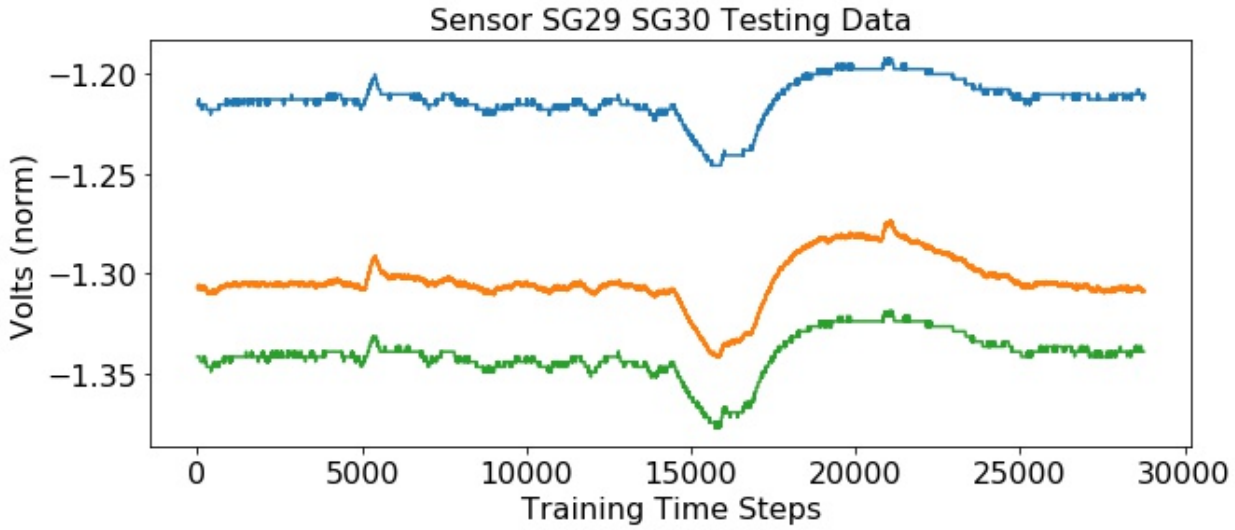

Figure 43: LSTM Results for Sensors 29 and 30

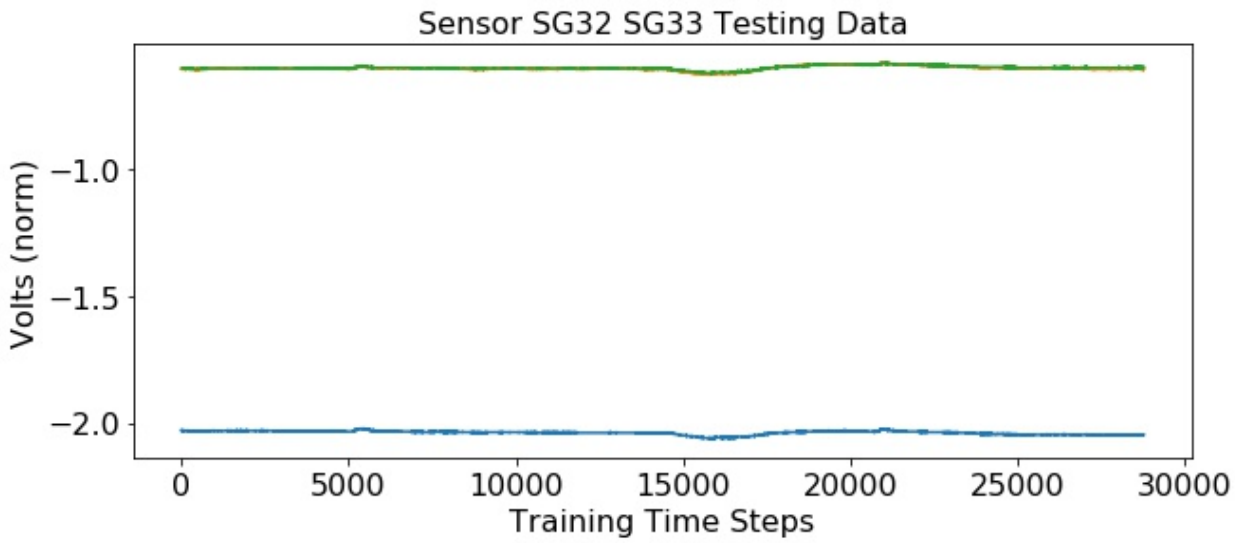

Figure 44: LSTM Results for Sensors 32 and 33

4.4 OBSERVATIONS AND DISCUSSION

The standalone LSTM model experienced varying levels of success when it came to predicting the ground truth sensor data from the uncalibrated sensor data. Once again, the merit of the predicted data ranged from very good, good, and somewhat poor. Looking at Figures 29 through 44, it can be seen that the LSTM model's prediction for sensors 1, 3, 7, 9, 23, and 32 (Figures 29, 30, 32, 33, 40, and 44, respectively) were very good. The model's predictions for sensors 11, 17, 19, 21, 25, 27, and 29 (Figures 33, 37, 38, 39, 41, 42, and 43, respectively) were somewhat good. Finally, the model's predictions for sensors 5, 13, and 15 (Figures 31, 35, and 36, respectively) were somewhat poor.

For the predictions that were labeled "very good" from the preceding paragraph, the shape of the predicted graph closely matched that of the ground truth data and the

curve of the predicted values is very closely fit to the ground truth sensor data. The sensors that were labeled as “somewhat good” exhibited the correct shape for the predicted values when compared to the ground truth curve, but the predicted curve did not have a very tight fit to the ground truth curve. Lastly, the sensors labeled as “somewhat poor” exhibited a generally correct predicted values curve, but that curve was substantially closer to the uncalibrated data curve than the ground truth curve. In these cases, the output of the LSTM model either made a marginal improvement or no improvement when compared to the uncalibrated input data.

Similar to the GRU model, the LSTM model generally overcompensates for the system-wide voltage increases experienced near the 5,500 and 21,000 time steps, although not as severely as the GRU model (shown in Section 3.3 and discussed in Section 3.4). This overcompensation is most notable in Figure 37. Generally speaking, however, the LSTM model appeared to be much better at not overcompensating for increases in voltages that were seen system-wide. This is perhaps due to a better fit, or perhaps due to the fact that the model was slightly more underfit due to being trained for considerably fewer epochs.

Additionally similar to the GRU model, as mentioned above, the LSTM model made predictions for a couple of the sensors that were not largely helpful. The sensors in question are sensors 5, 13, and 15. The predictions for sensors 5 and 13 were ever-so-slightly better than that of the uncalibrated input data. The predictions for sensor 15 were nearly identical to that of the uncalibrated input data. With that being said, there is one unique difference between the GRU model and the LSTM model when it comes to their corresponding poor predictions: the poor LSTM predictions are not detrimental.

4.5 CONCLUSION

Most of the LSTM's estimates of the ground truth sensor values were quite good. There were three sensors where the model made predictions that were either exactly the same as the uncalibrated data, or slightly better. Even though these predictions were not always better than the uncalibrated input data, the predictions were either the same as the uncalibrated data or slightly better. The shape of all of the predicted value curves were correct and the model did not generally overcompensate for system-wide voltage upticks. While the LSTM model had a higher overall MSE when compared to the GRU model, the overall performance of the LSTM model could be considered better on these facts alone. The LSTM model appears to be well equipped and proficient at recovering ground truth data from uncalibrated sensor data.

5. STACKING

Stacked Generalization, or stacking, is an ensemble learning technique that uses multiple models arranged in a manner in which one feeds directly into the next. All of the models are trained on the same data. Stacking typically reduces bias and generally leads to better performance than any one of the individual base models [9, pp. 699].

5.1 DESIGN

For this particular application, the models used were the full individual GRU model, a modified LSTM model, and an ad hoc fully connected neural network (FCNN). The architecture of the GRU model was identical to that of the standalone GRU model shown and discussed in Section 3.1 and Figure 8. This model was first trained on the full data set (the parameters will be discussed in Section 5.3). The output from the GRU model was then fed directly into the LSTM and FCNN models. Once training of the GRU model was complete, the parameters of the model were locked and unchanged. This means that while the LSTM and FCNN models were training, the GRU model was left unchanged. The architecture for the LSTM and FCNN models are shown in Figure 45.

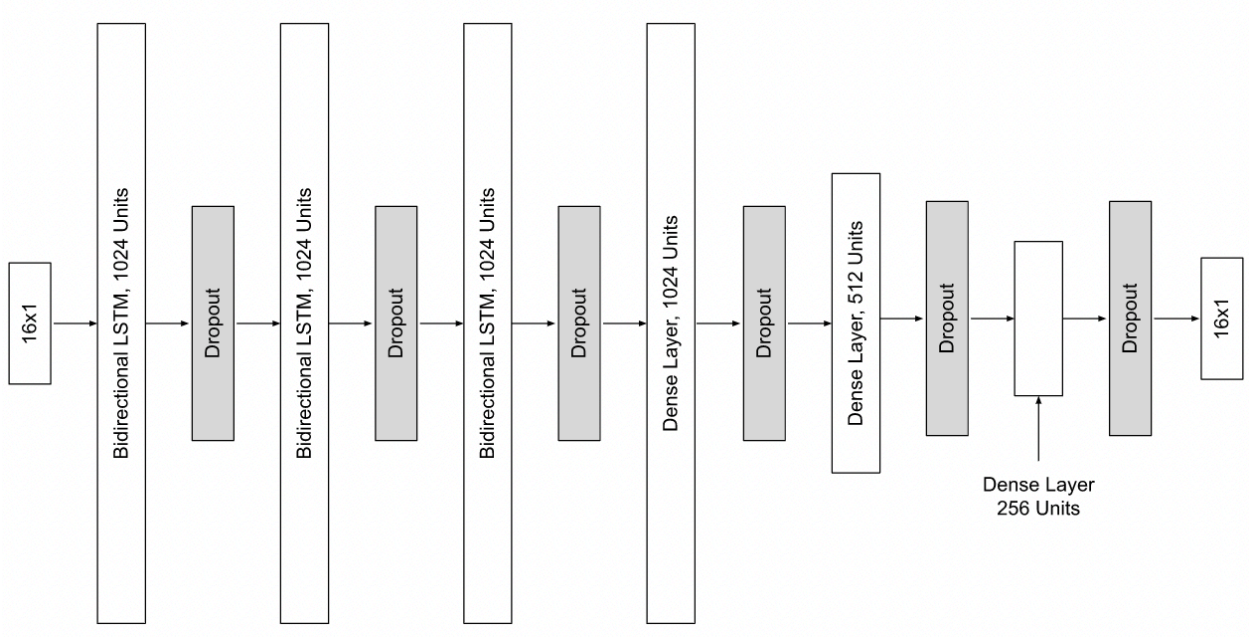

Figure 45: LSTM and FCNN Stacked Architecture

The derivation of the LSTM and FCNN models was a modification of the original LSTM model discussed in Section 4. This modification simply included adding dropout layers between the dense layers, creating a set of layers that act less as a means to step down the output of the LSTM layers and act more as a FCNN. The LSTM layers and the dense layers (FCNN) were trained in parallel. The resulting stacked architecture is a GRU model fed into an LSTM model fed into a FCNN. The rationale behind the decision to allow the dense layers to form their own FCNN layer is presented in Section 5.2. Both the GRU model and the LSTM/FCNN architectures were trained using slightly different sets of parameters. These parameters will be discussed at length in the Section that follows.

A wide array of different stacked configurations were tried, many of which more closely resembled the standalone GRU and LSTM architectures. The architecture and parameters presented in this Section yielded the best results of all configurations tried. The stacked model appeared to benefit from vastly different parameters from that of the standalone models and the bagged models that will be discussed in the next section. Four models that produced sub-optimal results will be shown in Section 5.2, as they provided valuable information and lessons learned.

5.2 UNSUCCESSFUL MODEL DESIGNS

This section will cover four failed to perform better than the standalone models. The first model that will be examined was a GRU that fed into an LSTM and then into an FCNN. The GRU model was trained separately from the LSTM and FCNN networks. The GRU model trained had the architecture presented in Section 3. The LSTM and FCNN network had the architecture shown in Figure 45. The LSTM/FCNN network was trained with a learning rate of 1e-05 for 300 epochs. The dropout rate between the bidirectional LSTM layers was 0.4 and was 0.1 between the dense layers (FCNN layers).

The best overall MSE achieved was 0.00132 after 100 epochs. Although this is approaching the MSE of both the standalone GRU and LSTM models, it is still noticeably higher than both. Additionally, the ensemble uses the GRU model that achieved an MSE 0.00105. These are not promising results. Similar to the standalone models, submodels were saved every 25 epochs. The results of these submodels are shown in Table 5.

Number of Epochs	MSE
25	0.00255
50	0.00239
75	0.00139
100	**0.00132**
125	0.00147
150	0.00152
175	0.00149
200	0.00148
225	0.00163
250	0.00158
275	0.00162
300	0.00168

Table 5: First Failed GRU, LSTM, FCNN Stacked Model Results

Additionally, the validation loss for the training of the LSTM and FCNN network are shown in Figure 46. As can be seen from this Figure, the validation loss achieves a stable minimum value around 100 epochs. With an increasing MSE from the saved models and a stable validation loss above 100 epochs, it is safe to assume that the ensemble did in fact achieve peak performance at 100 epochs and continuing to train the ensemble would not be beneficial to its performance.

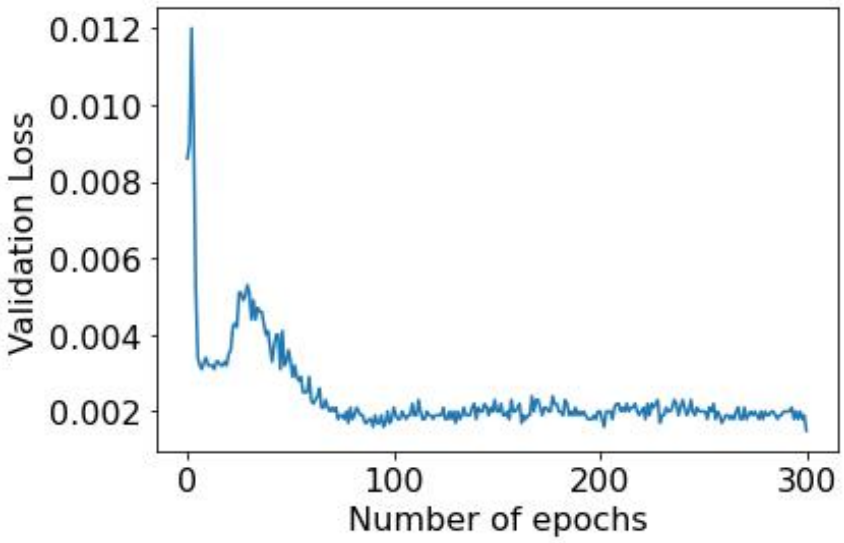

Figure 46: First Failed GRU, LSTM, FCNN Stacked Model Validation Loss

The second unsuccessful model that will be explored is a variation of the first. The second model is identical to the first model, except for the removal of the dropout layers between the dense layers in the LSTM/FCNN model. This returns the latter model to the architecture shown in Section 4, essentially removing the FCNN model from the architecture. In other words, the model is strictly the GRU model from Section 3 fed into the LSTM architecture from Section 4. The latter model was trained for 500 epochs using the model from Section 3 as its input. This model was saved every 25 epochs and the results are shown in Table 6. The best MSE achieved by this ensemble was 0.00163 after 75 and 100 epochs.

Number of Epochs	MSE
25	0.00343
50	0.00227
75	**0.00163**
100	**0.00163**
125	0.00165
150	0.00172
175	0.00167
200	0.00179
225	0.00168
250	0.00167
275	0.00193
300	0.00199
325	0.00197
350	0.00206
375	0.00210
400	0.00217
425	0.00228
450	0.00225
475	0.00216
500	0.00225

Table 6: First Failed GRU into LSTM Stacked Model Results

The importance of this model is it shows how the stacked ensemble performs worse when it is strictly a GRU model on top of an LSTM model. Based on these

results, it would appear that the best results are achieved when there are dropout layers between the dense layers, adding the ad hoc FCNN into the stack. The validation losses for the training of this model are shown in Figure 47.

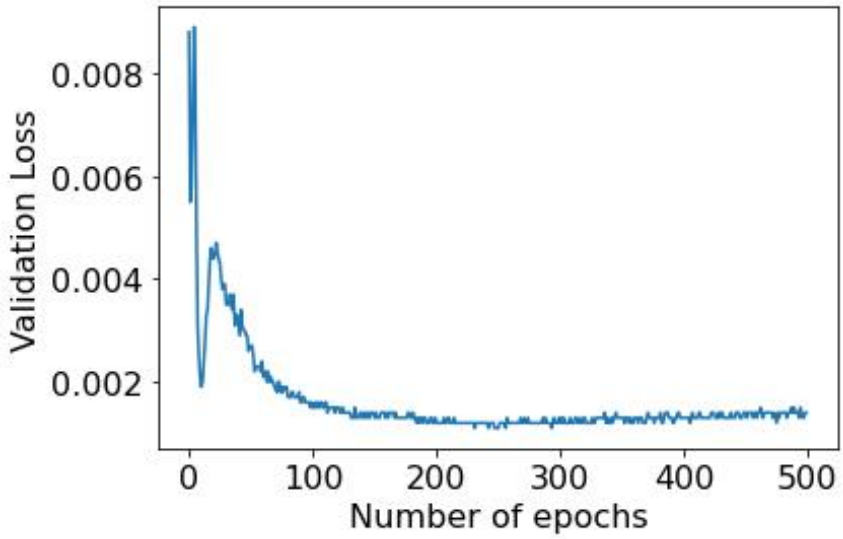

Figure 47: First Failed GRU into LSTM Stacked Model Validation Loss

The shape of this graph shows that the validation loss of the LSTM portion of the ensemble achieves a minimum at approximately 225 epochs. After approximately 250 epochs, the validation loss begins to increase. This points to the fact that the model achieved peak performance and started becoming overfit. Thus, it can be concluded that the lowest MSE produced by this ensemble with the given parameters is 0.00163, which is no better than either of the standalone models.

The next two unsuccessful stacked ensembles that will be examined were LSTM models from Section 4 fed into a GRU model with an architecture identical to the standalone model from Section 3. Similar to the model immediately preceding this one, there were no dropout layers between the dense layers. This model was also trained for 500 epochs and saved every 25 epochs and the performance of the model is shown in Table 7. The best MSE was 0.00274 after 25 epochs.

Number of Epochs	MSE
25	**0.00274**
50	0.02401
75	0.02118
100	0.02240
125	0.02629
150	0.02492
175	0.01911
200	0.02181
225	0.01754
250	0.01471
275	0.01417
300	0.01257
325	0.01206
350	0.01150
375	0.01085
400	0.01051
425	0.01020
450	0.00971
475	0.00868
500	0.00850

Table 7: First Failed LSTM into GRU Stacked Model Results

The performance pattern shown in Table 7 is quite peculiar. For additional context, the validation loss graph is provided in Figure 48. The shape of this curve should shed some light onto the somewhat obscure results of the ensemble.

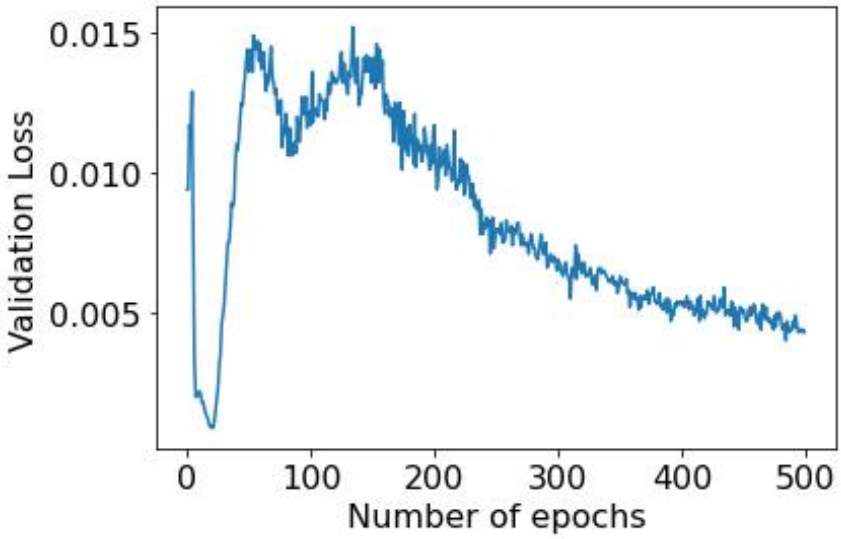

Figure 48: First Failed LSTM into GRU Stacked Model Validation Loss

As seen from this Figure, the lowest validation loss was achieved almost immediately. Next, the GRU model appears to become somewhat unstable and the validation loss shoots up, before stabilizing after approximately 175 epochs. The validation loss of the model then begins to decrease once again and starts to form a desirable shape. Although training finished while the validation loss was still decreasing, the results of what lies past 500 epochs was determined to be irrelevant. The training validation loss, as well as the MSE of the model, was still rather high at the

500 epoch mark. Although decreasing, it was determined to be highly unlikely that the MSE of the model would decrease to the point where this particular ensemble outperformed either of the standalone models. Additionally, any ensemble that requires an additional 500+ epochs of training with a mere hope of matching the performance of the standalone models would likely be cost-inhibitive. Thus, it was determined that the best use of resources was to focus on the immediate success achieved by the ensemble within the first 100 epochs.

Looking at the first 100 epochs, the validation loss is incredibly unstable, which likely points to a learning rate that is set too high. As such, the GRU portion of the model was re-trained for 200 epochs with a learning rate of 1e-06, with submodels being saved off every 10 epochs. The goal was to smooth the performance of the GRU model a bit, capturing and optimizing the success of the ensemble seen after approximately 30 epochs. Additionally, the learning rate was decreased from 1e-05 to 1e-06 in an attempt to slow the rate of convergence and cause the model to be more stable. The results of this run are shown in Table 8.

Number of Epochs	MSE
10	0.00437
20	0.00287
30	0.00256
40	0.00231
50	0.00188
60	0.00226
70	0.00249
80	0.00219
90	0.00176
100	0.00137
110	0.00111
120	**0.00107**
130	0.00111
140	0.00124
150	0.00149
160	0.00183
170	0.00229
180	0.00287
190	0.00354
200	0.00433

Table 8: Second Failed LSTM into GRU Stacked Model Results

As Table 8 shows, the lowest overall MSE was achieved after 120 epochs at 0.00107 volts. Although this model performed better than its predecessor, a MSE of 0.00107 is not an improvement over the standalone GRU model. For additional context, the validation loss graph is provided for this ensemble in Figure 49. The shape of the validation loss for the training of the GRU portion of the model looks exactly as expected given the results of the ensemble, bottoming out around 125 epochs before going back up.

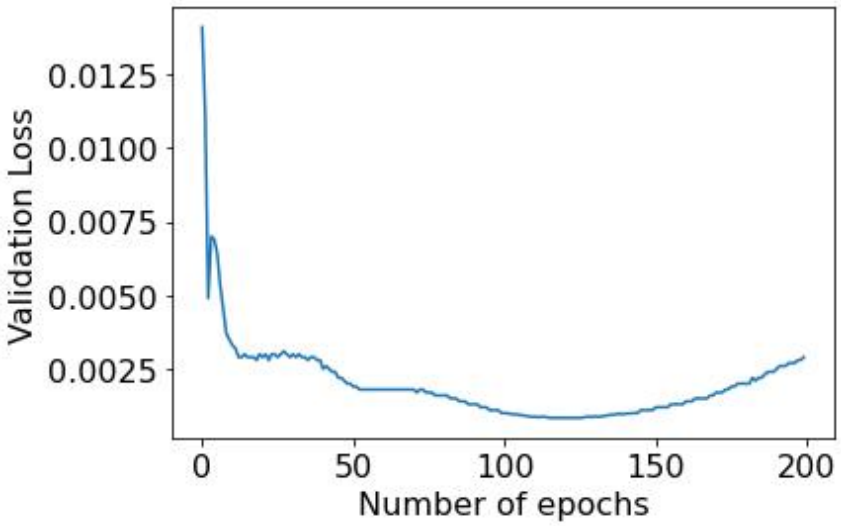

Figure 49: Second Failed LSTM into GRU Stacked Model Validation Loss

The results of this model additionally confirm the observation that the GRU portion of the previous LSTM into GRU stack was not worth pursuing past 500 epochs. With a lower learning rate, the validation loss curve is much more stable and achieves a minimum before increasing around 125 epochs. As such, one should not expect that a

model with the exact same architecture and parameters — except for a higher learning rate — would achieve a lower MSE at any point.

5.3 SIGNIFICANT PARAMETERS

Much like the standalone models, the basic parameters that were set were the learning rate, the dropout rate between various layers, and the number of epochs. These parameters were selected independently for both parameters. This subsection will cover the parameters for both the GRU model that was trained first, followed by the LSTM and FCNN models that were trained after. Optimizing hyperparameters for a stacked model is considerably more difficult, as the overall ensemble includes three different models trained during two different runs with different parameters. As such, the optimal GRU model might not be the one with the lowest overall MSE. Instead, it is the one that the LSTM and FCNN models are able to best utilize to make the final predictions for the 16 sensors. Thus, the selection of these parameters did not involve the same analytical rigor seen in Sections 3, 4 and 6. Setting up the stacking model was a little bit more “trial and error” coupled with lessons learned from the failures presented in Section 5.2

The general goal for the stacked model was to incorporate a GRU model fed into an LSTM model then finally into a FCNN model. Each model would be trained less than its corresponding standalone model to prevent overfitting. The model would also be trained in two different iterations, with the GRU model being trained separately from the LSTM/FCNN model.

5.2.1 DROPOUT RATE

The general idea behind choosing the dropout rate for both portions of the model was to use a significantly lower rate versus the standalone models. This is due to the instability and lack of success seen in the stacked models with high dropout rates. The idea was that a lower dropout rate would increase bias, but that having a series of generally more biased predictors fed one into another would not have detrimental results and would instead improve the predictive power of the ensemble. Thus, a rate of 0.1 was chosen for all dropout layers in all stacked models presented from this point forward.

5.2.2 NUMBER OF EPOCHS AND LEARNING RATE

Before the GRU model that was used as part of the ensemble was trained, a separate GRU model was trained with an aggressively small learning rate of 1e-07 and dropout rate of 0.1 between each RNN layer. This model was trained for 400 epochs. Unfortunately, submodels were not saved for this particular model, so step-by-step performance cannot be evaluated like previously done for other models. However, the validation loss for the model training was recorded and is shown in Figure 50. The validation loss data provided ample information for making a determination on the number of epochs to train the model.

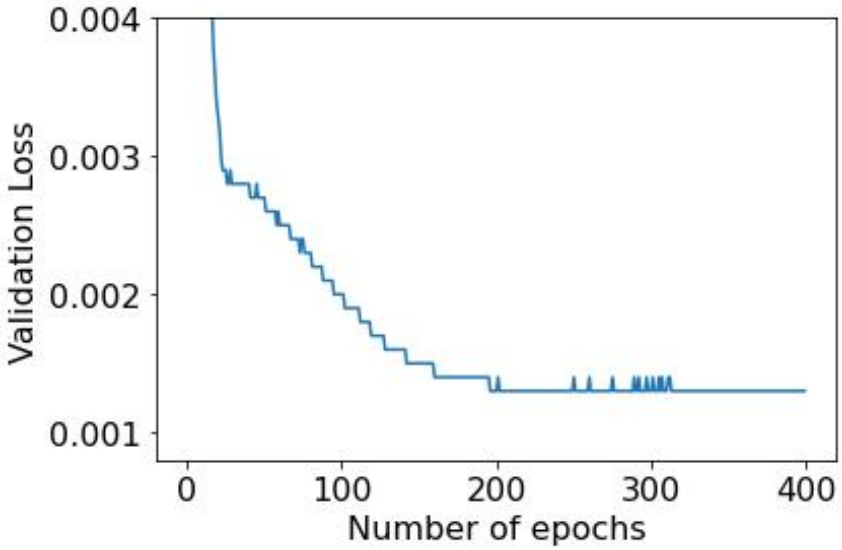

Figure 50: GRU Stacking Test Model Validation Loss

This Figure shows that this GRU model converges gently and steadily. The primary goal for the GRU model to be used as the foundation for the stacked ensemble was one that is slightly underfit, with training stopping right before the validation loss converges. Thus, it was determined, based on the data provided in Figure 50, that the model should be trained for 150 epochs using the same parameters. The learning rate was left rather low at 1e-07 to allow for a slow, steady, predictable convergence. To provide further evidence for this decision, a more zoomed-in look at the validation loss is shown in Figure 51.

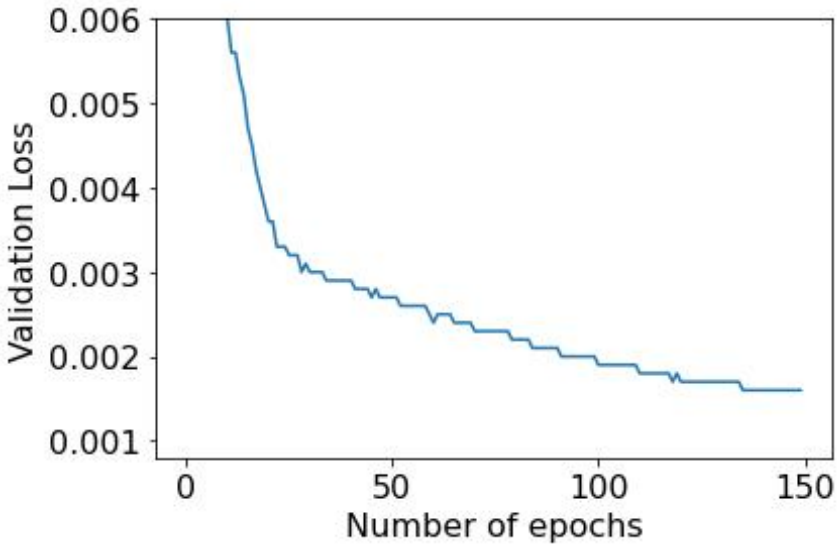

Figure 51: GRU Stacking Model Validation Loss

This Figure does a better job of depicting the desired shape. The validation loss curve appears to be leveling off, but still has not quite converged. Additionally, the curve is somewhat smooth and nearly monotonically decreasing. Thus, these parameters were accepted and the learning rate for the GRU model was set to 1e-07 and the number of epochs set to 150.

Now that the hyperparameters for the first model are set and the model is trained, the learning rate and number of epochs for the second model (two models that will be trained at the same time) can be set. As a starting point, the same learning rate (1e-07) was used, as well as the reduced dropout rate of 0.1 for all dropout layers. Once again, this was to allow for a relatively smooth and stable training process. This model was trained for 240 epochs and saved every 20 epochs. The validation loss curve is shown in Figure 52.

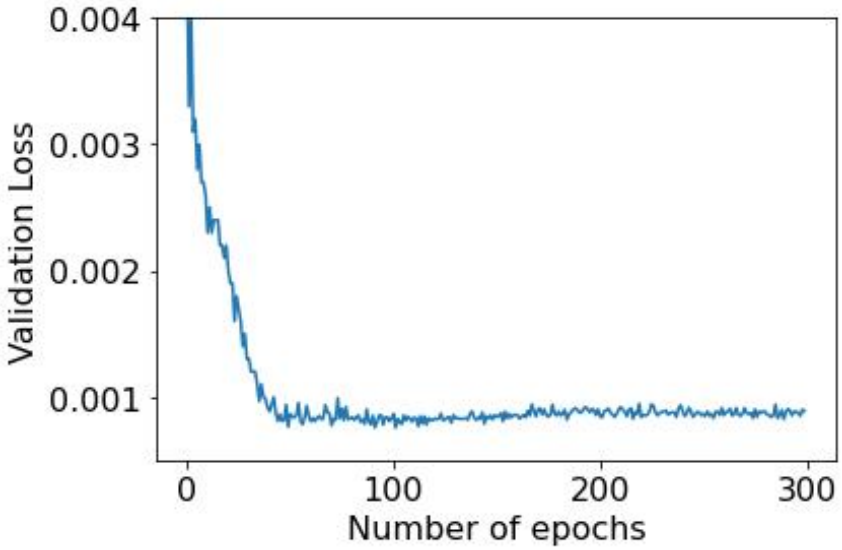

Figure 52: GRU, LSTM, FCNN Stack Validation Loss

As this Figure shows, the training of the second half of the ensemble resulted in a very smooth validation loss curve that converged nicely. Thus, a learning rate of 1e-07 appears to be appropriate. Based on the shape of this curve, coupled with the fact that the dropout is set relatively low, the number of epochs required to train the LSTM and FCNN models is probably somewhat low, likely somewhere between 50 and 100. To determine the exact number of epochs, the MSE of the submodels are shown in Table 9.

Number of Epochs	MSE
20	0.00314
40	0.00109
60	**0.00104**
100	0.00108
120	0.00112
140	0.00110
160	0.00112
180	0.00114
200	0.00124
220	0.00133
240	0.00133

Table 9: GRU, LSTM, FCNN Stacking Results

Based on the results of the submodels, it would appear that the optimal number of epochs to train the LSTM and FCNN models are around 60 epochs. Thus, in summary, the hyperparameters for the GRU model is a learning rate of 1e-07 for 150 epochs and the hyperparameters for the LSTM and FCNN models (trained together) is a learning rate of 1e-07 for 60 epochs.

5.4 RESULTS

The best performance achieved from the stacked ensemble was obtained from the run that generated Table 9, with an MSE of 0.00104. Figures of the predictions

generated from this ensemble are presented in Figures 53 through 68. Once again, the uncalibrated sensor reading is shown in blue, the calibrated is shown in green, and the model's prediction is shown in orange.

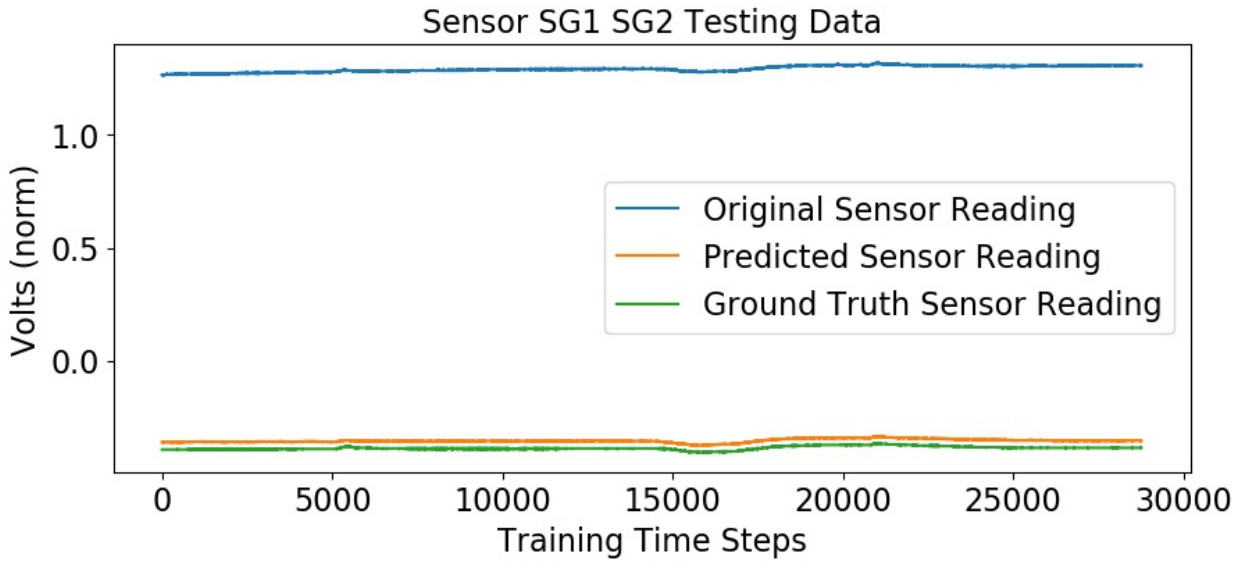

Figure 53: Stacking Results for Sensors 1 and 2

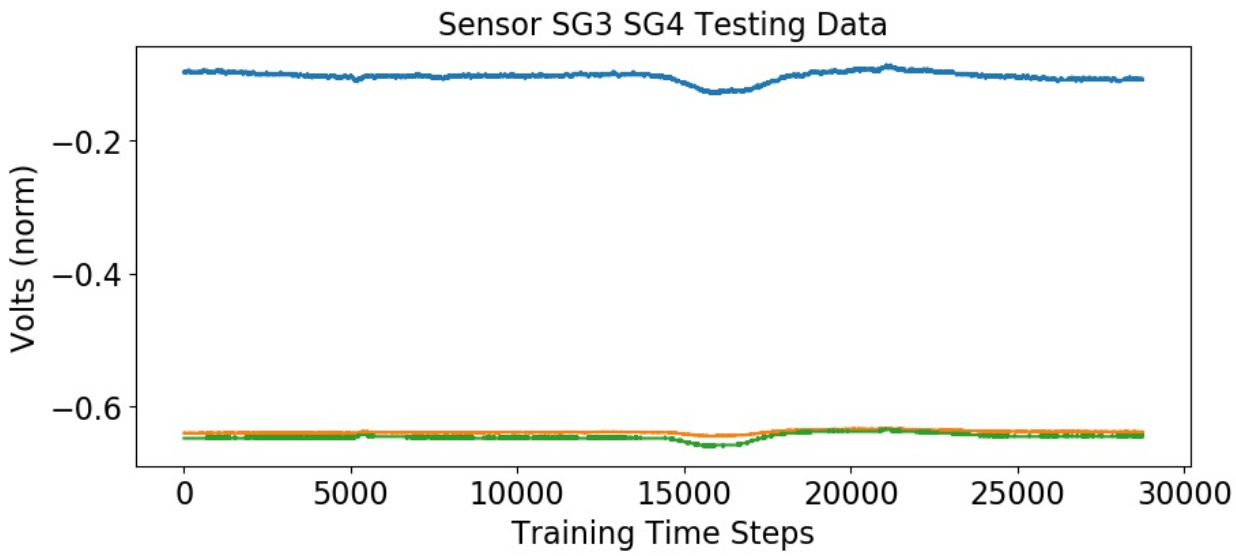

Figure 54: Stacking Results for Sensors 3 and 4

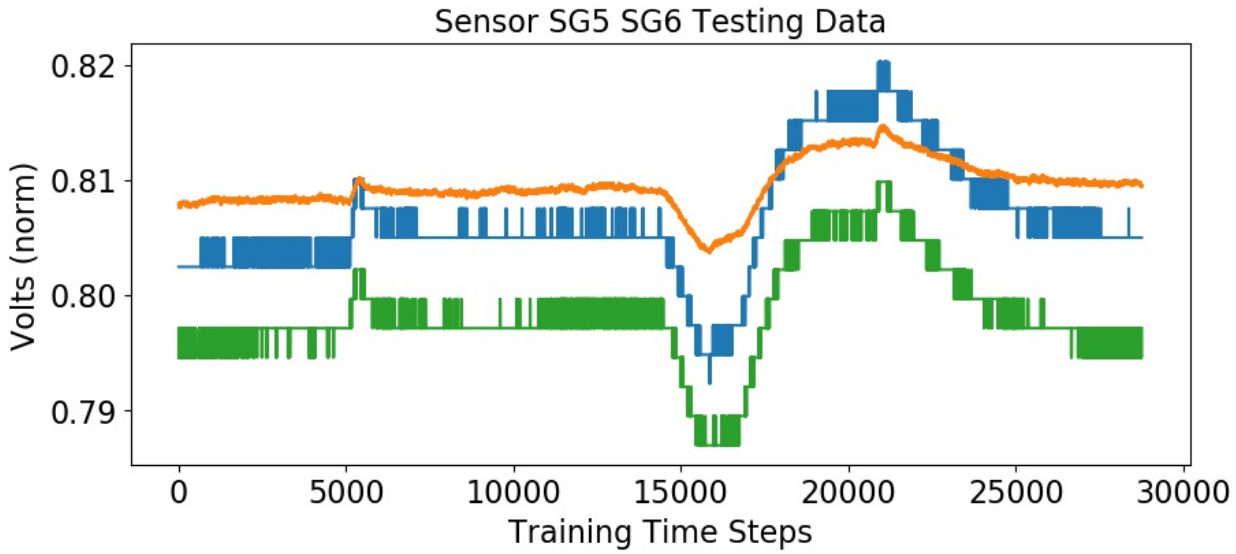

Figure 55: Stacking Results for Sensors 5 and 6

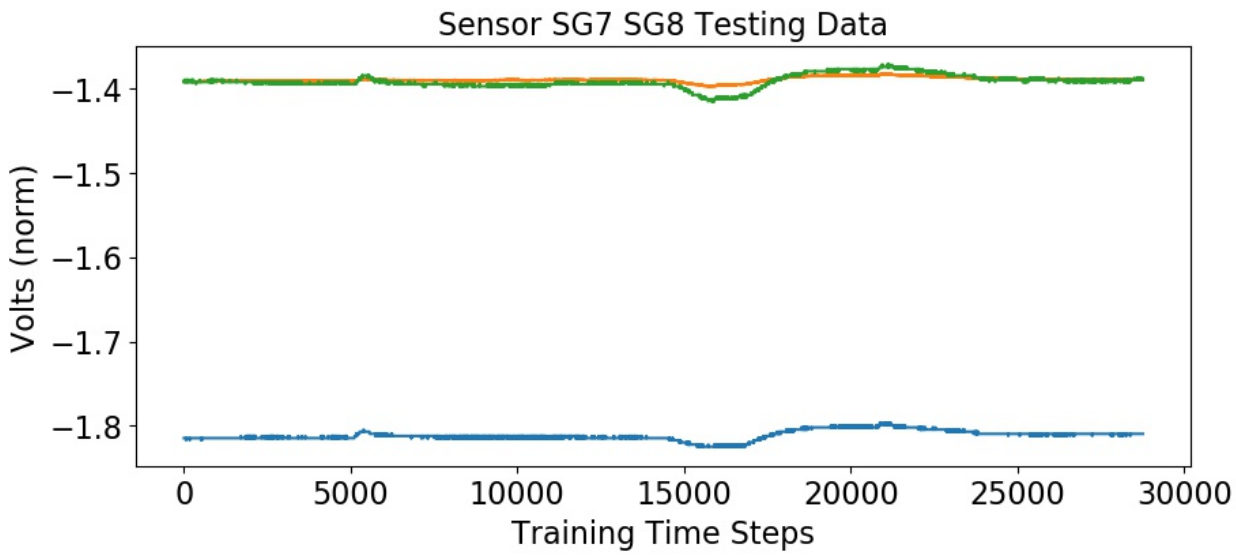

Figure 56: Stacking Results for Sensors 7 and 8

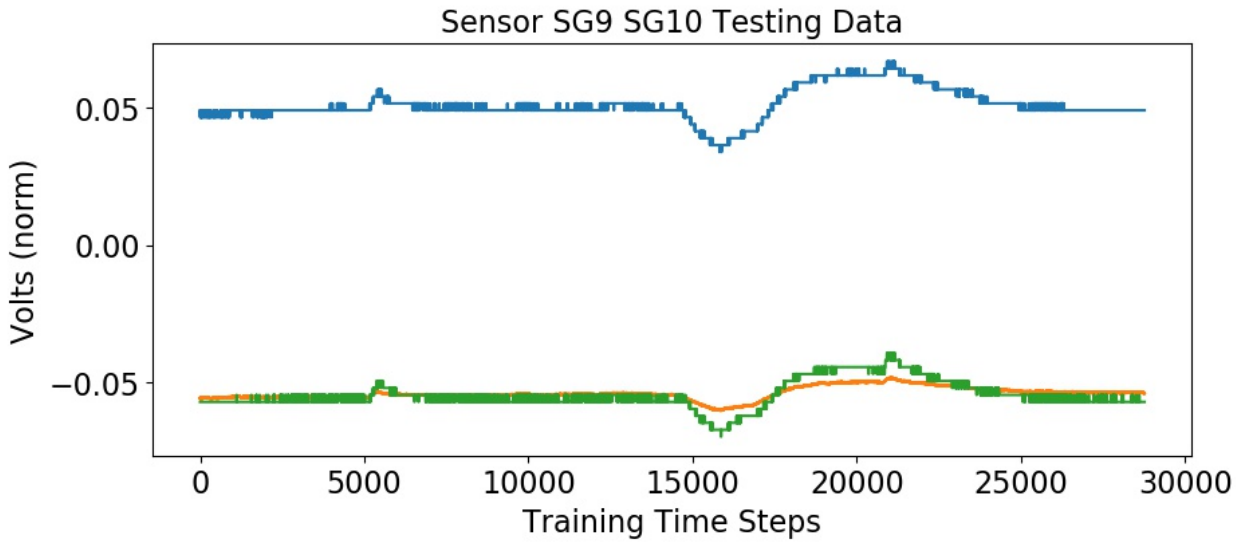

Figure 57: Stacking Results for Sensors 9 and 10

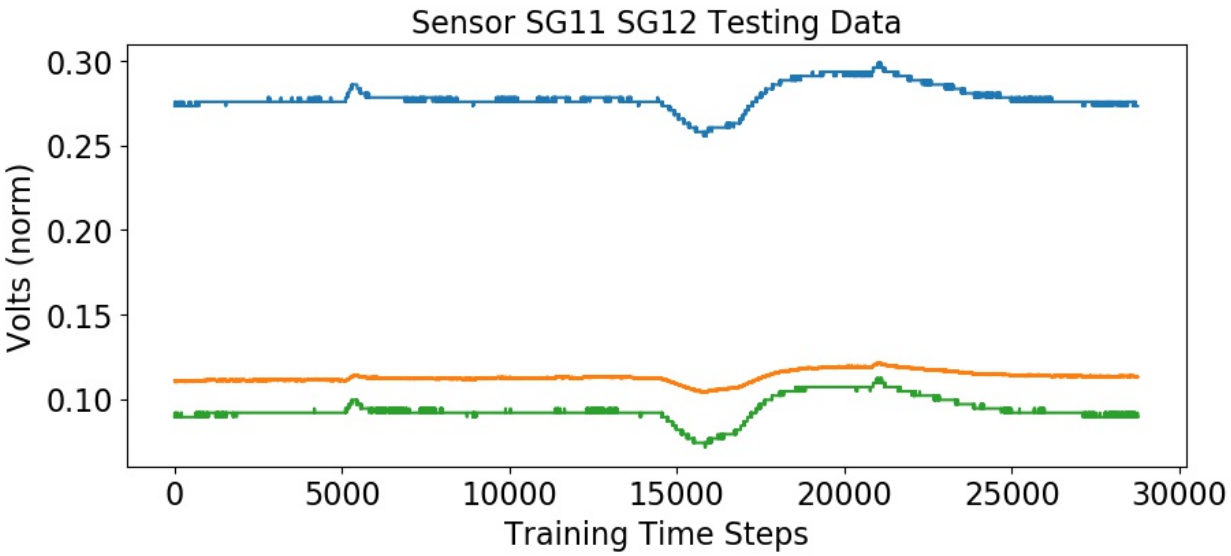

Figure 58: Stacking Results for Sensors 11 and 12

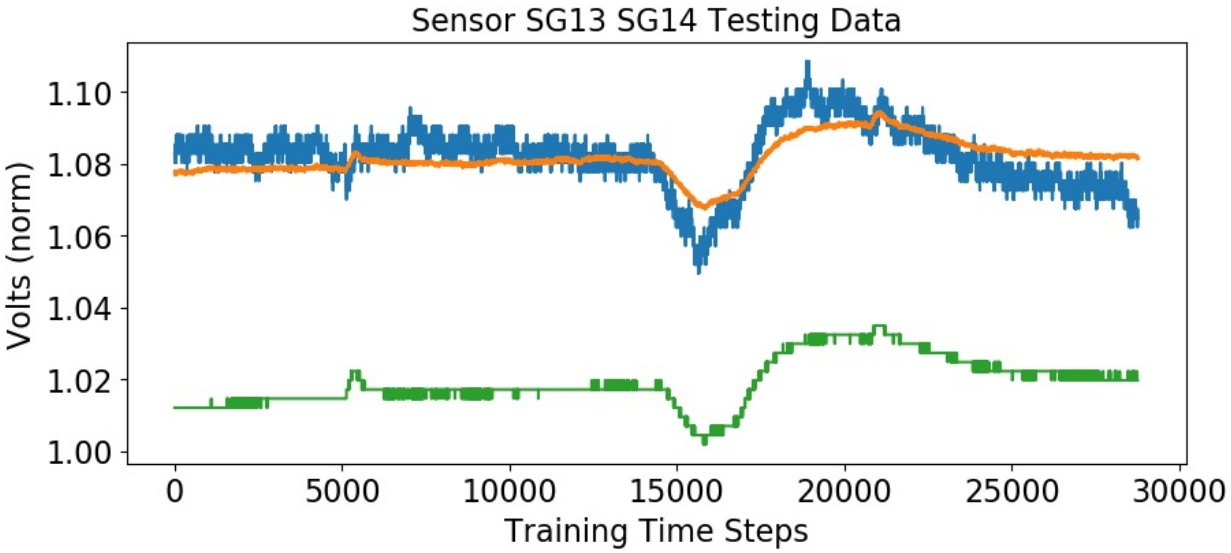

Figure 59: Stacking Results for Sensors 13 and 14

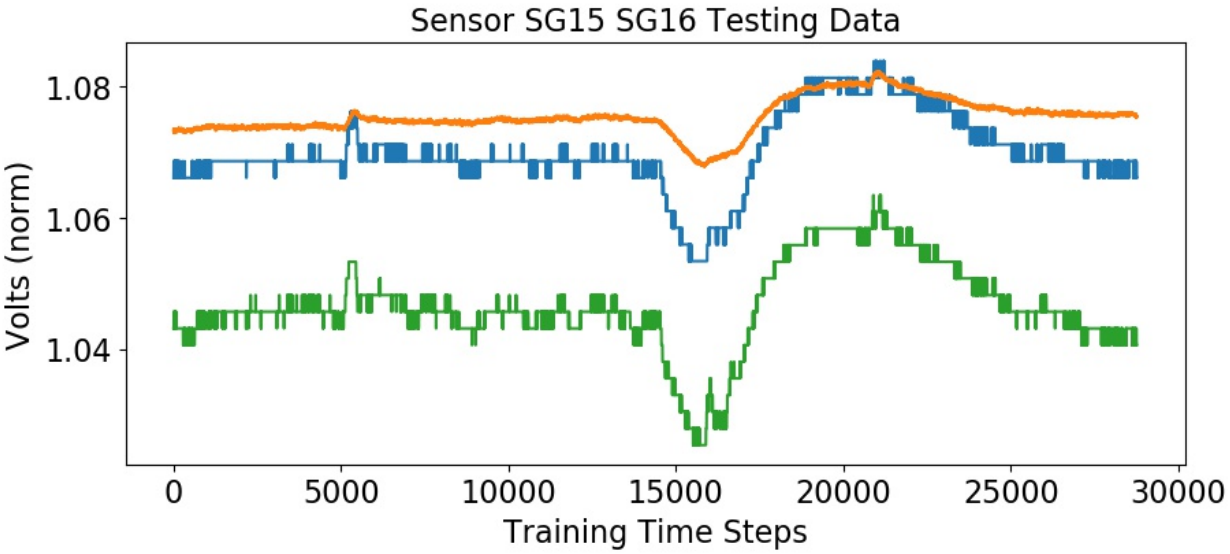

Figure 60: Stacking Results for Sensors 15 and 16

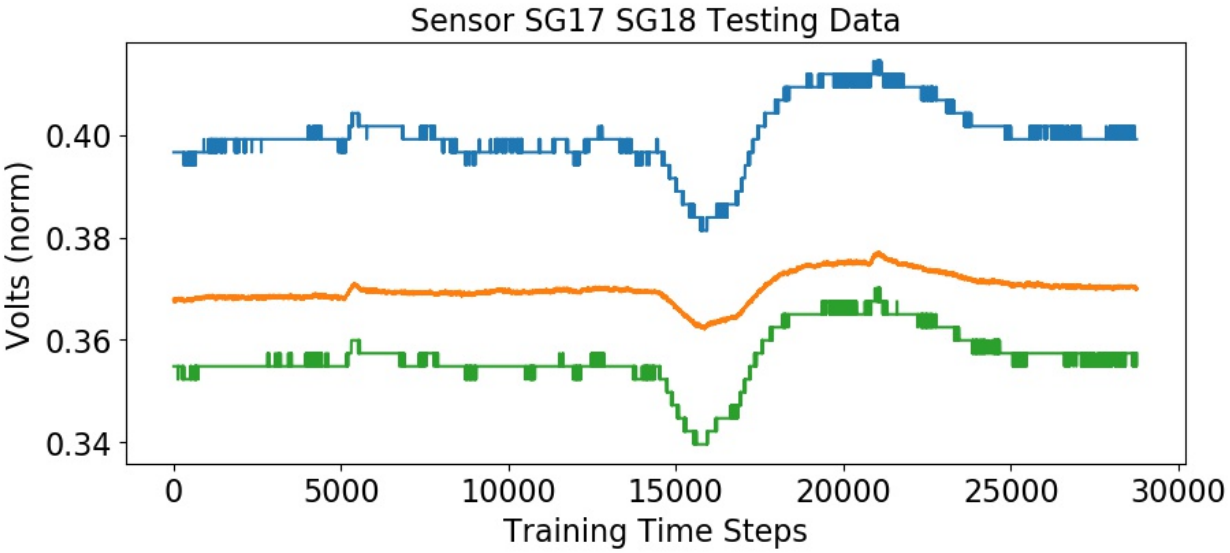

Figure 61: Stacking Results for Sensors 17 and 18

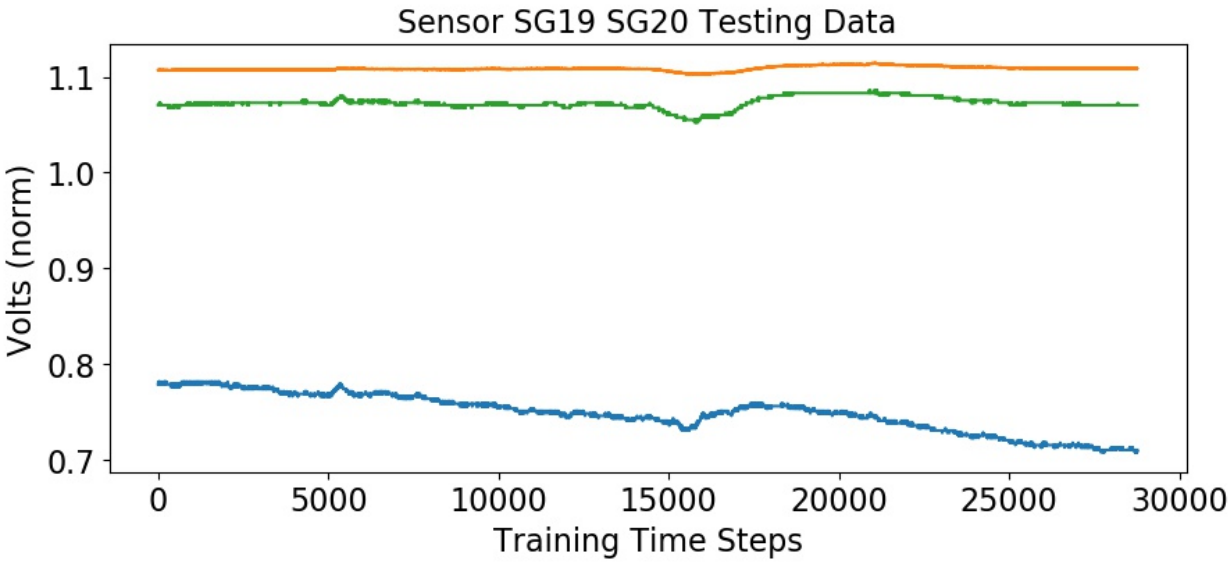

Figure 62: Stacking Results for Sensors 19 and 20

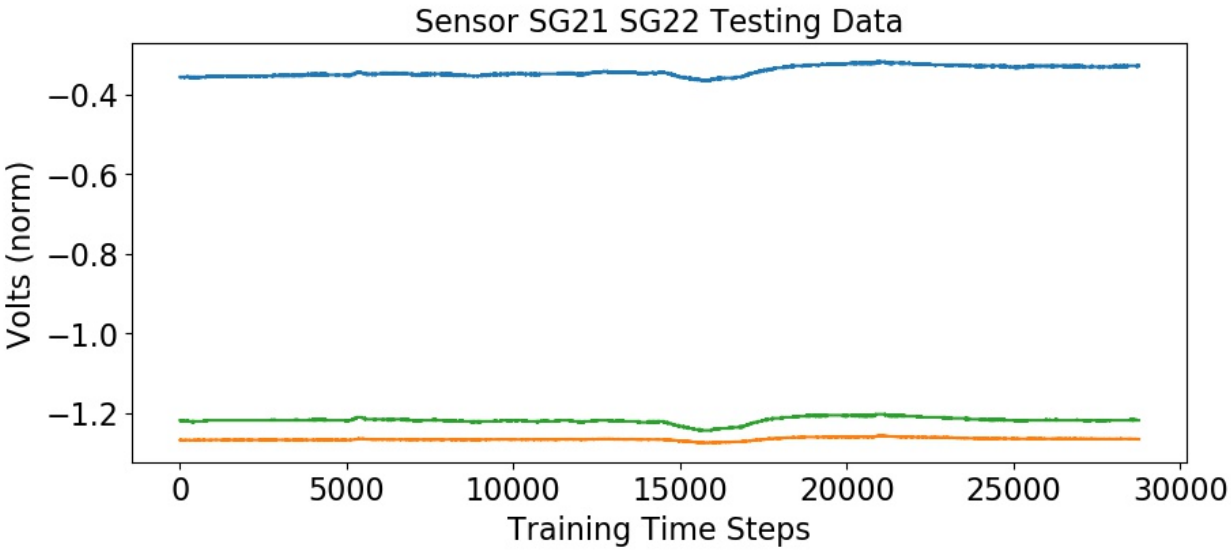

Figure 63: Stacking Results for Sensors 21 and 22

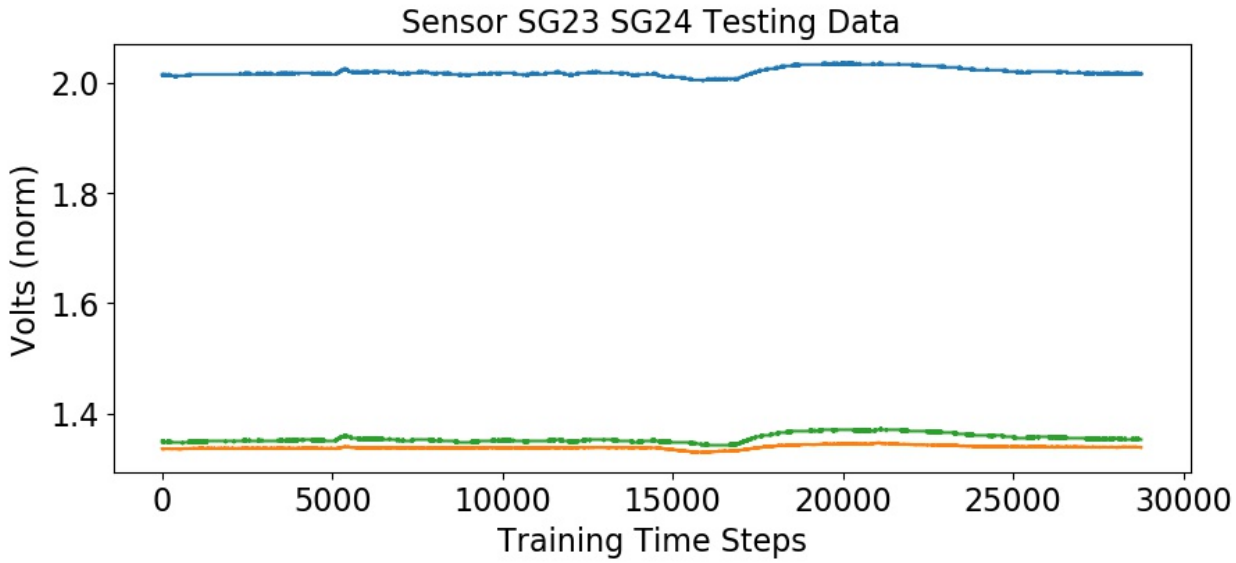

Figure 64: Stacking Results for Sensors 23 and 24

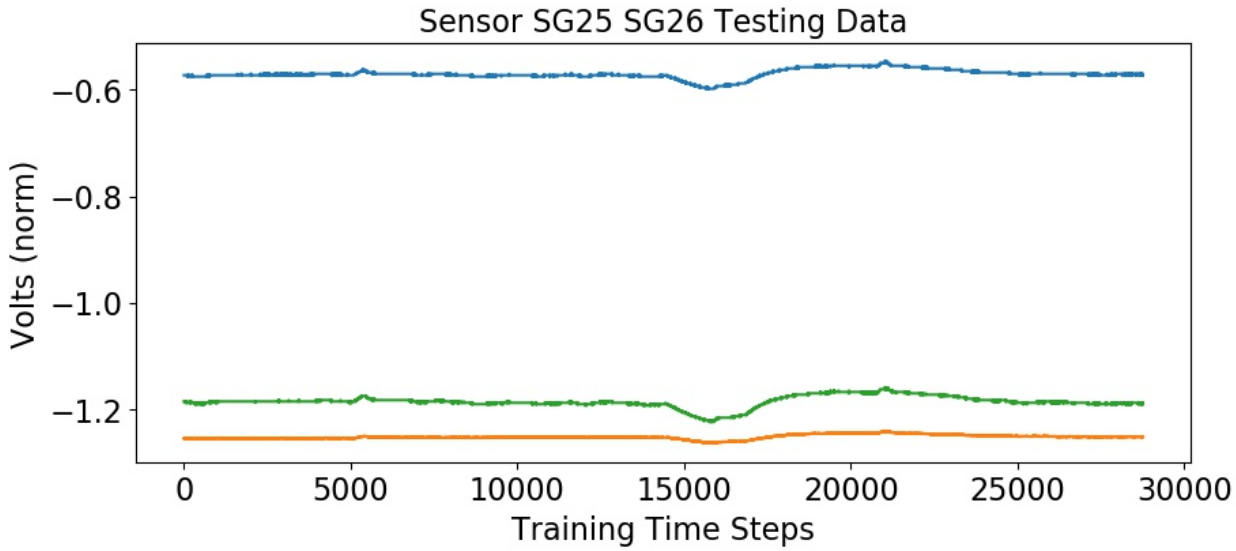

Figure 65: Stacking Results for Sensors 25 and 26

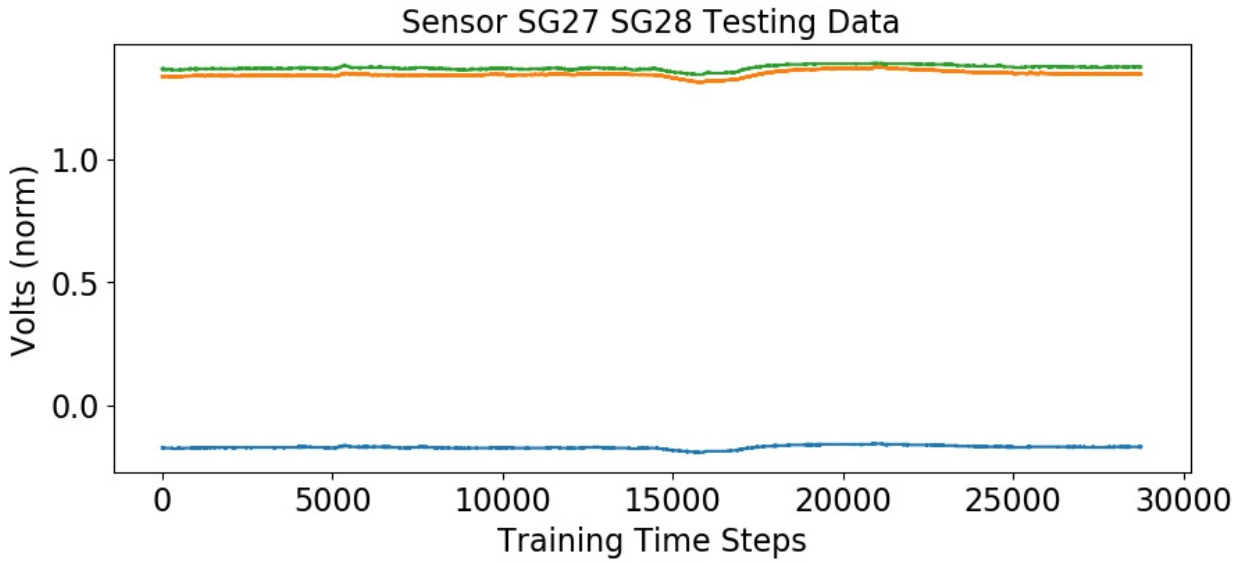

Figure 66: Stacking Results for Sensors 27 and 28

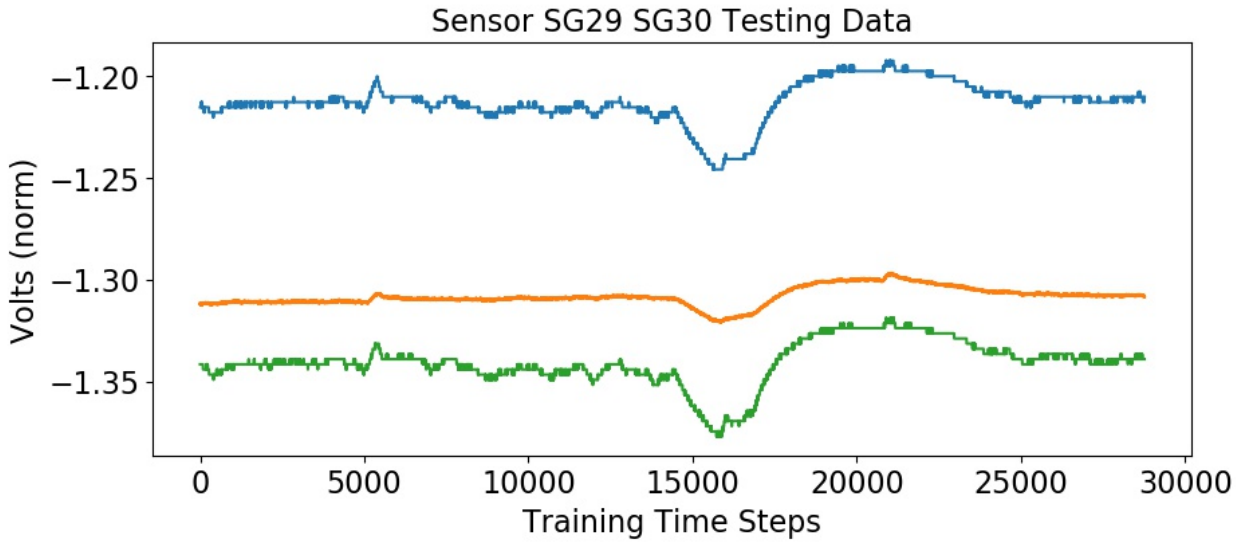

Figure 67: Stacking Results for Sensors 29 and 30

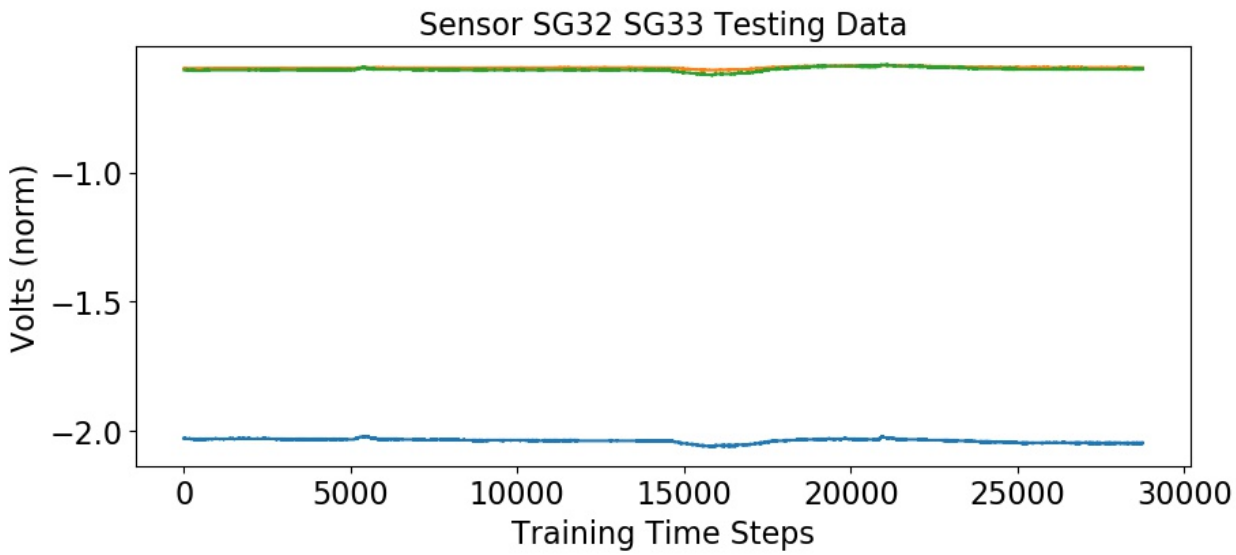

Figure 68: Stacking Results for Sensors 32 and 33

5.5 OBSERVATIONS AND DISCUSSION

The overall performance of the stacked ensemble is rather good. Of the 16 predictions made by the ensemble, 5 appear to be quite good (sensors 1, 3, 23, 27 and 32 shown in Figures 53, 54, 64, 66, and 68 respectively), 8 appear to be somewhat good (sensors 7, 9, 11, 17, 19, 21, 25, and 29 shown in Figures 56, 57, 58, 61, 62, 63, 65, and 67 respectively), and 3 are somewhat poor (sensors 5, 13, and 15 shown in Figures 55, 59, and 67 respectively). Predictions labeled “quite good” have both a good fit to the ground truth sensor curve, as well as generally having the correct shape. Predictions labeled as “somewhat good” are generally not quite the right shape and have a generally looser fit to their ground truth sensor. The predictions of both of these sets of sensors are an improvement over the uncalibrated sensor reading. The

predictions for the sensors that are labeled “somewhat poor” either closely hug the uncalibrated sensor prediction or are worse than the uncalibrated sensor prediction.

In sharp contrast to both the standalone GRU and LSTM models, the stacked ensemble does not overcompensate for the systemwide uptick seen around the 5,500 and 21,000 time steps. This is a notable improvement. Additionally, although the stacked ensemble still failed to make good predictions for all of the sensors, it did slightly outperform both of the standalone models in terms of MSE. Most of the predictions (13 of the 16) were better than the uncalibrated inputs for that particular sensor. The three that were worse (sensors 5, 13 and 15) will be examined in further detail in an attempt to determine if the error is a result of the data used or a shortcoming of the ensemble. That is, it is possible that the point where the data was split (the 300,000th time step) could be a poor choice for a particular sensor if that sensor was mid-drift, causing the ensemble to make an unexplainably poor prediction for that sensor. First the predictions for sensor 5 will be examined. The training data for sensor 5 is shown in Figure 69.

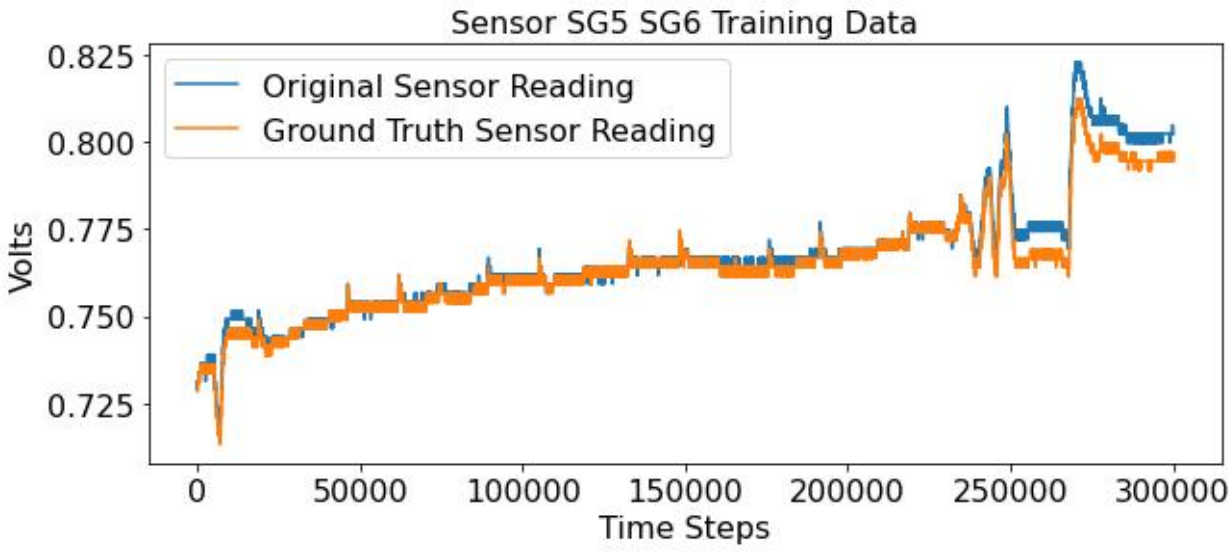

Figure 69: Sensor 5 Training Data

The general pattern of the training data for sensor 5 is that both the uncalibrated and ground truth sensors are approximately the same until around the 250,000 time step. At that point, the uncalibrated sensor begins to drift up following a significant decrease in voltage. This drift remains somewhat constant through the significant voltage increase around the 275,000 time step. This drift continues in the testing data, as shown in Figure 55. This does not offer a data-based explanation of why the ensemble predicted the drift-free measurement for sensor 5 to be higher than the uncalibrated sensor value. Additionally, this prediction is worse than the uncalibrated sensor reading, as the ground truth reading is below the uncalibrated sensor value, not above. Next, sensor 13 will be examined. The training data for sensor 13 is shown in Figure 70.

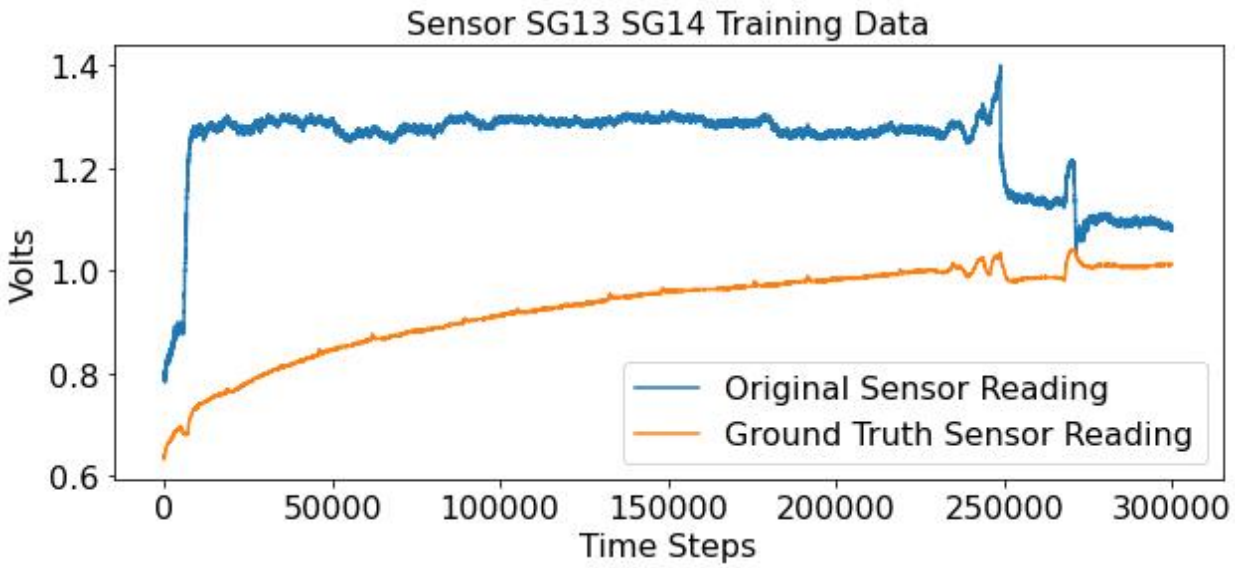

Figure 70: Sensor 13 Training Data

The general pattern for this sensor pair is quite different from the previous sensor set examined. The ground truth and uncalibrated sensor values start somewhat close before drifting far apart. After the initial drift, the gap between the two sensors begins to close. This remains relatively constant throughout the testing data, as shown in Figure 59. The ensemble predicts the drift-free measurement to be nearly exactly the same as the uncalibrated sensor. This error also does not appear to be an issue relating to the training and testing data, as it seems more likely that, based on the data, that the ensemble would predict a voltage that is too low based on the large drift present in most of the training data. The ensemble instead predicted virtually zero drift. Lastly, sensor 15 will be examined. The training data for sensor 15 is shown in Figure 71.

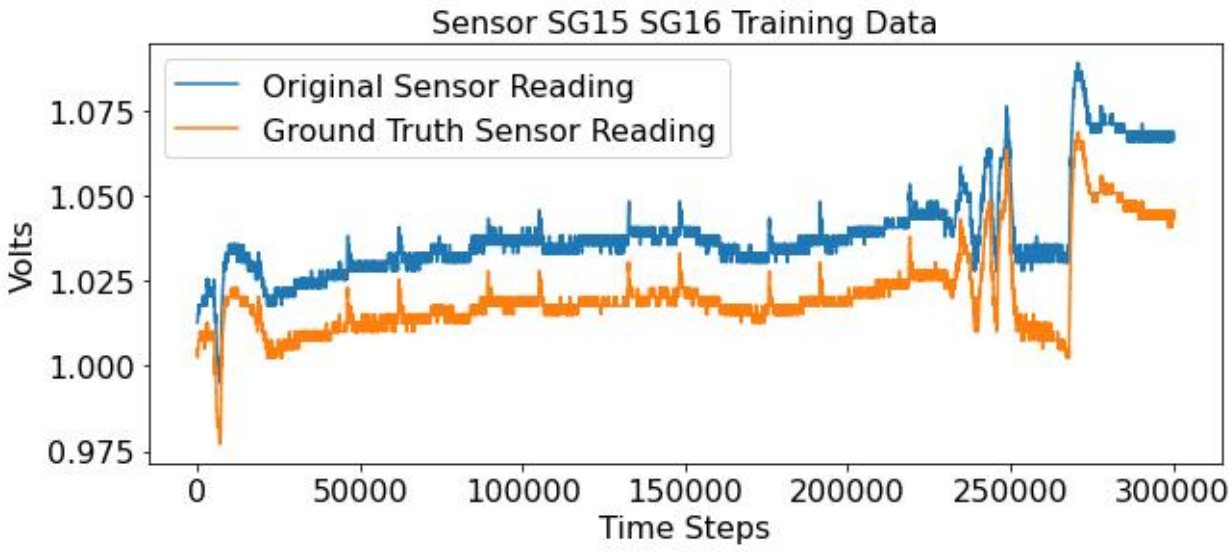

Figure 71: Sensor 15 Training Data

This set of training data appears to be quite simple. From time step 0 until time step 300,000, the amount of drift appears to remain relatively consistent, with both the uncalibrated and ground truth sensors moving in near perfect unison. Near somewhere around the 285,000th time step, there appears to be a small increase in the amount of drift, with the ground truth sensor decreasing while the uncalibrated sensor remains somewhat constant. This small increase in the amount of drift remains somewhat constant throughout the testing data, as shown in Figure 60. As such, one would expect the ensemble to likely under-predict the drift slightly, as the sensors were slightly closer for the majority of the training data than the testing data. Instead, the ensemble once again made a near drift-free prediction for the sensor. As such, it would appear that the ensemble's poor prediction for this sensor is likely not the result of the training and testing data.

All three sensors for which the ensemble made somewhat poor predictions appear to not be the result of the data nor the location of the split between the training and testing data. For these three sensors, the ensemble made poor and even slightly counter-productive predictions of the drift-free sensor readings.

5.6 CONCLUSION

The ensemble appears to be an improvement over both of the standalone models. This is based on several factors, such as overall error, commonalities in errors, and overall model cost. The stacked ensemble has a lower MSE (0.00104) when compared to the GRU model (0.00105) and the LSTM model (0.00119), though the amount the ensemble beats the standalone GRU model is rather small. What causes the stacked ensemble to be advantageous over the standalone GRU model is the training costs. The standalone GRU model took around 191 seconds per epoch for 425 epochs for a total of 81,175 seconds, or 22:32:55. The stacked ensemble required 185 seconds for 150 epochs (the GRU portion of the ensemble) and 211 seconds for 60 epochs (LSTM and FCNN portion of the ensemble), for a total training time of 40,410 seconds, or 11:13:30. That is, the stacked ensemble required approximately one half of the training time to achieve approximately the same performance. The performance of the stacked model was noticeably better than that of the standalone LSTM model, as well as requiring significantly less time to train. The standalone LSTM model took 262 seconds per epoch for 300 epochs, which equates to 78,600 seconds, or 21:50:00. This is definitely an improvement.

The stacked ensemble did not perform well making drift-free predictions for three sensors: 5, 13, and 15. Looking back at Sections 3.4 and 4.4, it can be seen that both the standalone GRU and standalone LSTM models struggled predicting these sensors as well. The stacked ensemble did not do any worse with regards to these three sensors. Following the same logic that was used in Sections 3.5 and 4.5, making good or even very good predictions for 13 of 16 sensors is quite good. The performance of the stacked ensemble coupled with the large reduction in training time leads to the conclusion that the stacked ensemble was rather successful at recovering drift-free sensor readings from uncalibrated sensors.

6. BAGGING

6.1 DESIGN

Bootstrap Aggregation, or Bagging, utilizes K individual models that were trained using K distinct and independent datasets to make predictions on data. The K predictions (where the i^{th} model is denoted as p_i, such that the i^{th} model's prediction of an input x would be $p_i(x)$) are aggregated by taking the un-weighted average of the K predictions [9, pp. 697]. In other words, the overall prediction $P(x)$ of the bagged model is calculated by:

$$P(x) = \sum_{i=1}^{K} p_i(x)$$

.

The underlying models, in this case, are GRU models. GRU was selected due to its standalone model performance and its superior cost-performance ratio, generally requiring less time-per-epoch when compared to an LSTM. Since bagging is a method used to optimize the performance of existing models by crowd-sourcing independent models, there is no need to change the underlying architecture of the models used. Thus, the architecture of the underlying models are identical to those discussed in Section 3.1 and shown in Figure 8.

6.2 DATA SELECTION

Before training can begin on the underlying models, the data used for training first has to be selected. A key element to bagging is independence of the underlying models, which increases the probability of the models being able to correctly predict the true sensor readings. Consider a simpler classification problem and five independent

models that correctly classify 75% of the time and an ensemble that classifies using the majority vote system (classified as whichever class gets 3 or more votes). Since the models are independent, the probability of at least three models correctly classifying the event (which would cause the ensemble to correctly classify the event) would be $C(5,3)\cdot\left(0.75^3\cdot 0.25^2\right)+C(5,4)\cdot\left(0.75^4\cdot 0.25\right)+C(5,5)\cdot\left(0.75^5\right)=0.896$ [9 pp. 697]. This is significantly better than the 75% accuracy of the underlying models. While true independence is an unreasonable assumption (since there is only one, finite dataset), it is important that the data selection process be as independent as possible, minimizing the amount of correlation between the models [9 pp. 697]. In order to achieve independence when selecting the training datasets, selection must be done with replacement [17, pp.732].

There is only one parameter required to select the new data set to train the submodels: what proportion of the original dataset is going to be selected for the training data. The selection of this ratio is discussed later in Section 6.3.1. This ratio is the same for all submodels that are trained independently for a bagging ensemble. A simple Python function was added to both the original GRU model that took in the training data that was originally identified in Section 1.3 and returned a training dataset and a test dataset.

The function randomly selects time slices of the data passed in and copies those time slices into a new list. The number of times this is done is defined by the size of the input dataset times the ratio passed to the function. Since each point was copied from the original list, there was an equal probability that each point could be selected every time, making the selections independent of each other (creating selection with

replacement). Once the appropriate number of time slices had been selected, the function then sorted the slices, putting them back into their original relative order. This was done because the ensemble is attempting to detect drift over time, so it is important to keep the training data in the correct order. If the data was left scrambled, the model would not be able to correctly correlate change with respect to time.

After the training data had been selected, the function then created a second dataset to be used for validation data. This dataset was composed of any time slice that was not included in the training data. The testing data did not need to be sorted since it was derived from a deep copy of the original dataset, so it was never out of order. Once these two datasets had been created, they were ready to be used for training the underlying model and the function returned these two lists of time slices as a tuple.

6.3 PARAMETER SELECTION

Although the standalone GRU model was painstakingly tuned, the underlying bagging models that will be used are optimized slightly differently. Thus, different parameters should be and will be used for training the underlying GRU models. Additionally, the bagging ensemble itself has two key parameters that need to be properly selected. This section will cover the selection of the parameters for the underlying models, as well as the parameters for the entire ensemble.

6.3.1 L/m RATIO AND K

The L/m ratio, which was the ratio mentioned in Section 6.2 as the ratio of the original dataset that was passed to the data selection function, defines what proportion of the original dataset size will be used for training. L is defined as the size of the training dataset and m is the size of the original dataset. Since the selection of the time slices are independent, only the size of the resulting dataset can be known, not the number of unique time slices it contains. While traditionally, bagging generally utilized an L/m ratio of 1.0, it has recently been shown that reducing the L/m ratio to somewhere between 0.6 and 0.9 improved the accuracy of the ensemble [18]. Since the sensor data collected and used for training and evaluation is relatively sparse sequential data, an L/m ratio of 0.9 was chosen and a bagging model was constructed, trained, and evaluated using this L/m ratio.

The number of predictors used in the ensemble, or K, generally only has a lower bound. It has been shown that as K increases, ensemble performance will either increase or remain the same. Typically, near peak performance can be achieved when $K \geq 30$ [18]. In order to maximize performance with runtime requirements, $K = 30$ was chosen.

6.3.2 DROPOUT RATE

As previously mentioned, the architecture of the GRU bagging model was left unchanged, but the hyperparameters used for training were optimized. One of those hyperparameters was the dropout rate between the GRU layers. In Section 3.2.1, it was determined that the dropout rate that would be used was 0.4 to balance training

time with proper fit. For bagging models, however, better results are achieved if the underlying models are overfit [19]. This intuitively makes sense. If all of the underlying models have a very general fit, the average of the models will result in a general fit. This is no better than using one model with a general fit. If however, all of the underlying models are overfit, their average is more likely to be unique. Since all of the models are trained using different data, they will all be overfit on a slightly different dataset. As a result, some of the models will likely over-predict drift for certain sensors, while others will likely underestimate drift. When these are averaged together, the result is likely to be closer to the true sensor reading than any of the individual models. This is the power of bagging. As a result, the dropout rate between the GRU layers was decreased from 0.4 to 0.2 to increase the fit.

6.3.3 NUMBER OF EPOCHS AND LEARNING RATE

As mentioned several times in previous sections, the goal when setting the learning rate is to allow the loss function – specifically the validation loss function – to converge (Section 3.2). Since bagging first involves training a standalone (not an ensemble of, yet) neural network, the best place to look for optimal hyperparameters was at the results of previous standalone runs for the GRU models. The optimal learning rate for both was previously determined to be 0.00001, or 1e-05 (see Section 3.2.1). Thus, the learning rate for the bagged GRU models was initially set to 1e-05.

The optimal number of epochs used to train a neural network is generally defined as the point in time, during training, in which the validation loss reaches a global minimum. Since the validation losses of every model trained and observed up until this

point have not been monotonically decreasing, that makes finding this global minimum a bit trickier. As a result, for parameter experimentation, a test model was intentionally over-trained and saved every 25 epochs so the results could be analyzed after the fact. This analysis would provide insight into optimal or near-optimal hyperparameters to be used to train the underlying bagging models.

It is worth noting that it is understood that the training and validation data for this test model – since it will follow the data selection process defined in Section 6.2 – will almost certainly be unique when compared to any other dataset used to train future underlying models. This is due to the independent data selection process used. As a result, iterations of training future bagging models will almost certainly yield different results. It will be shown, however, that the observations made on the test model are valid and the variations in the test and validation data do not have a significant impact on the overall performance of the ensemble.

The number of epochs for the test model was somewhat arbitrarily set to 300. Based on the information learned from the standalone models and the tweak to the dropout rate, this appeared to be a good starting point, as it was highly likely that the optimal number of epochs would be less than 300. The validation loss of this model is shown in Figure 72.

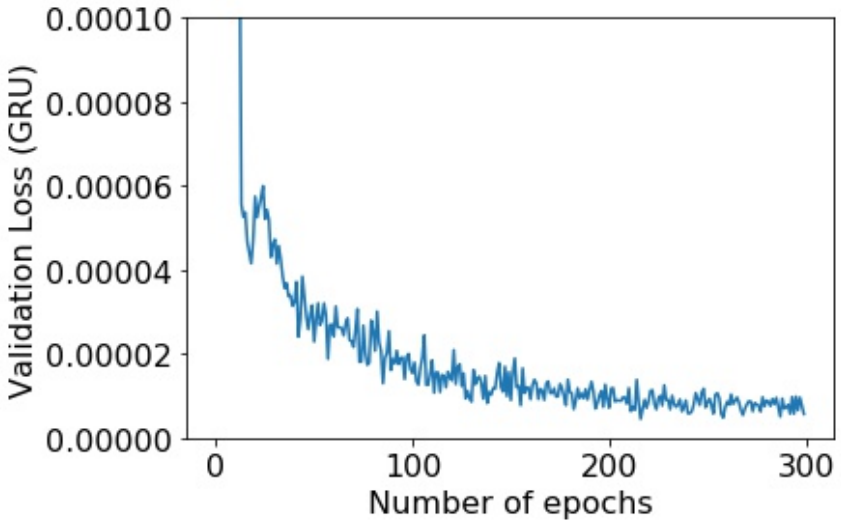

Figure 72: GRU L/m = 0.9 Validation Loss for 300 Epochs

From this Figure, it can be observed that the validation loss begins to converge after approximately 200 to 225 epochs, and remains stable until training ends at 300 epochs (since the validation loss smoothly converged, using a learning rate of 1e-05 is also validated). Thus, it can be assumed that the optimal number of training epochs is likely somewhere between 200 and 300 epochs. Fortunately, since the model was saved every 25 epochs, it does not have to be a guessing game. Table 10 shows the MSE of predictions made by the saved models on the testing data defined in Section 1.3.

Number of Epochs	MSE (e-05)
25	9.283
50	7.141
75	6.713
100	5.554
125	6.634
150	6.652
175	6.658
200	5.735
225	**4.480**
250	5.747
275	5.041
300	6.022

Table 10: Bagging Test Model MSE by Epoch

As can be seen from the table, the lowest MSE of the testing data is achieved at 225 epochs (bolded in the table). This aligns with the observations of the validation loss shown in Figure 72 that the optimal number of epochs likely falls between 200 and 300 epochs. With these two supporting pieces of evidence, it was determined that the optimal number of epochs to train the underlying bagging models was 225 epochs.

6.4 RESULTS

The underlying models were trained using the parameters mentioned above. Training took approximately 3 days and 8 hours to complete, training 12 models in

parallel at any given time. The bagging model was constructed by taking the unweighted average of the predictions generated by the models for the test dataset, as mentioned in Section 6.1. The overall MSE of the bagging model was 5.531e-05. This is noticeably better than the performance of the standalone GRU model from Section 3. Graphs showing each sensor set (drifted sensor and ground truth sensor) compared to the predictions of the ensemble are shown in Figures 73 through 88. Once again, the uncalibrated sensor reading is shown in blue, the calibrated is shown in green, and the model's prediction is shown in orange.

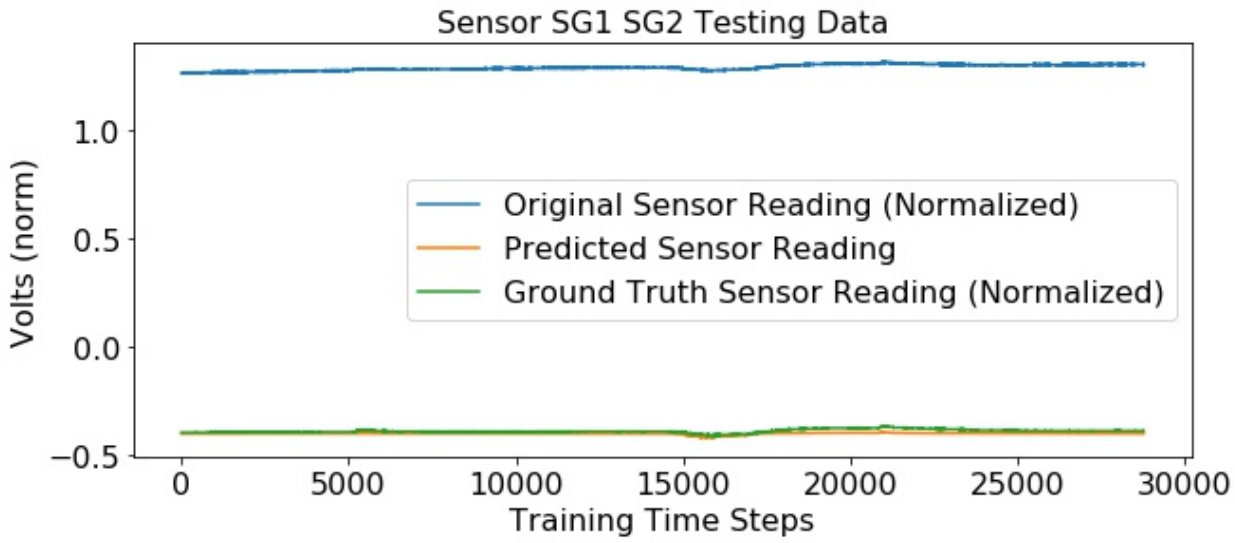

Figure 73: Bagging Results for Sensors 1 and 2

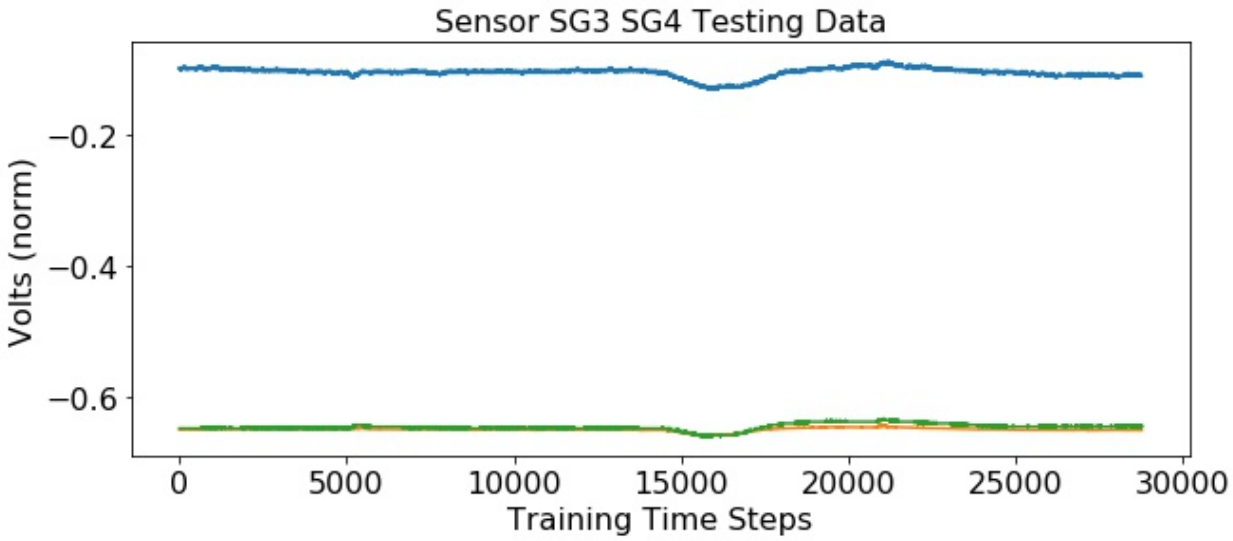

Figure 74: Bagging Results for Sensors 3 and 4

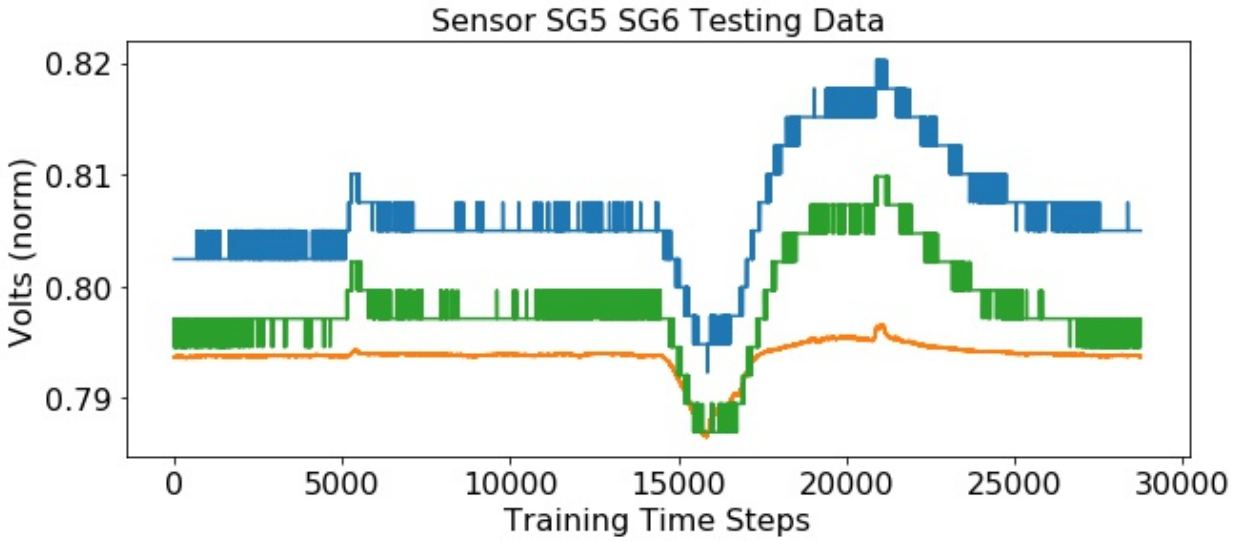

Figure 75: Bagging Results for Sensors 5 and 6

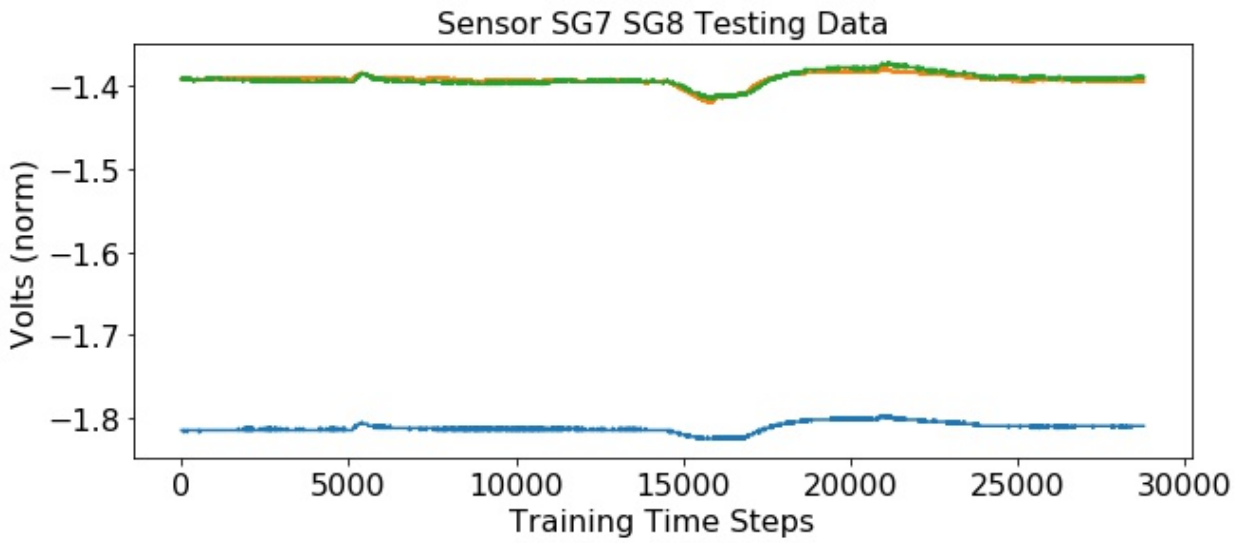

Figure 76: Bagging Results for Sensors 7 and 8

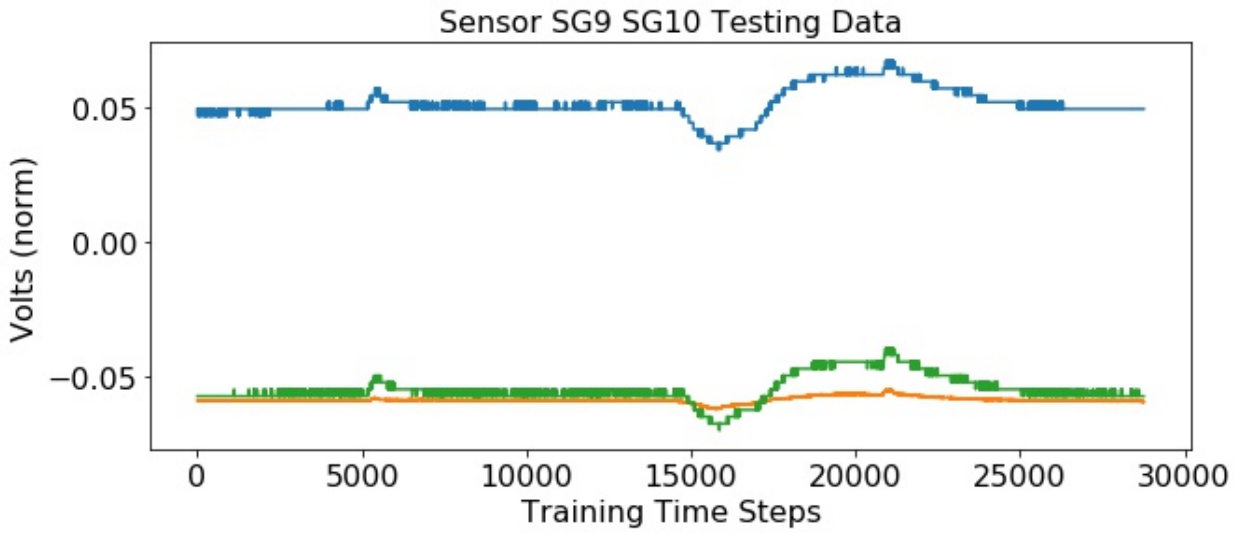

Figure 77: Bagging Results for Sensors 9 and 10

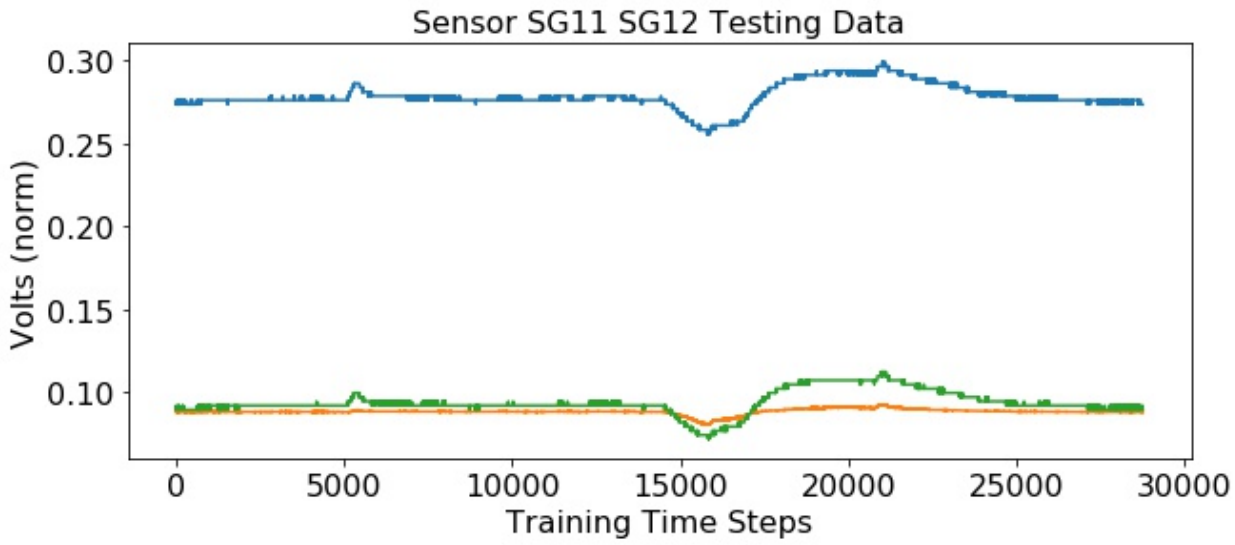

Figure 78: Bagging Results for Sensors 11 and 12

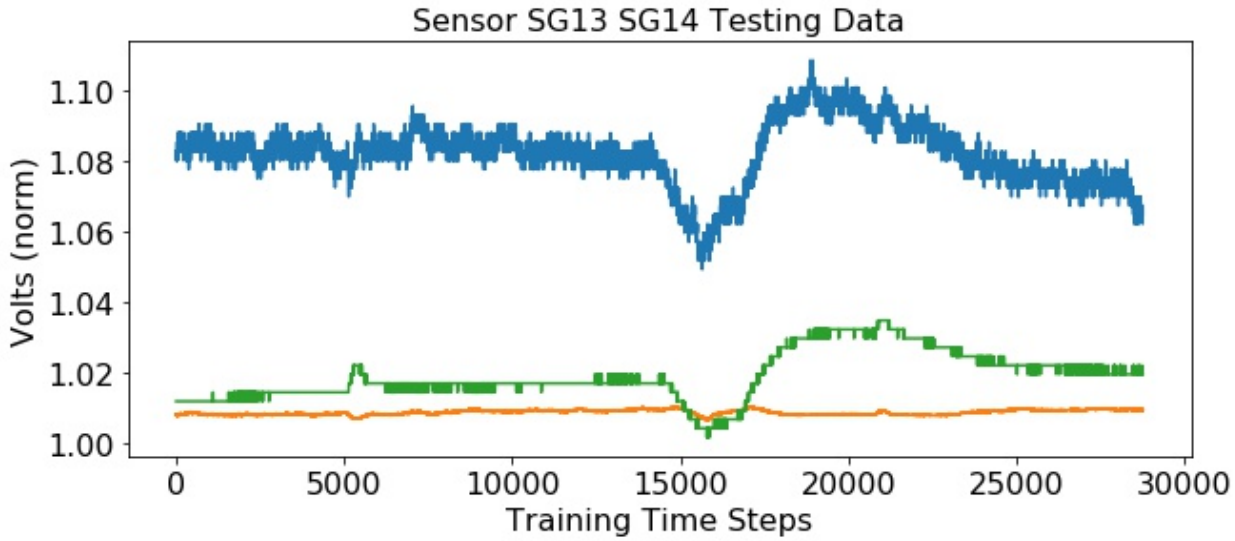

Figure 79: Bagging Results for Sensors 13 and 14

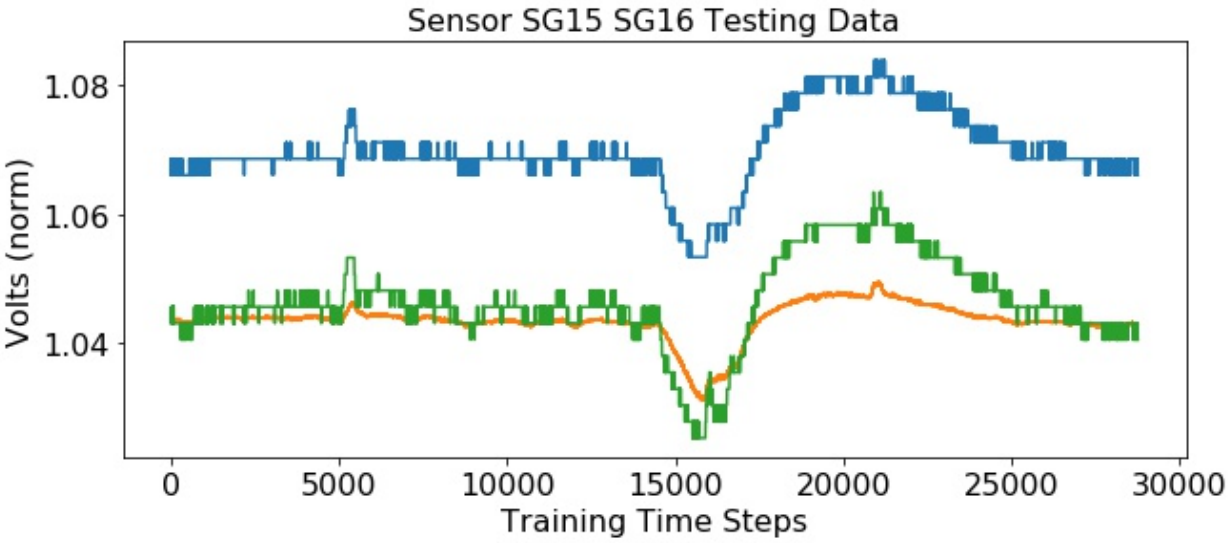

Figure 80: Bagging Results for Sensors 15 and 16

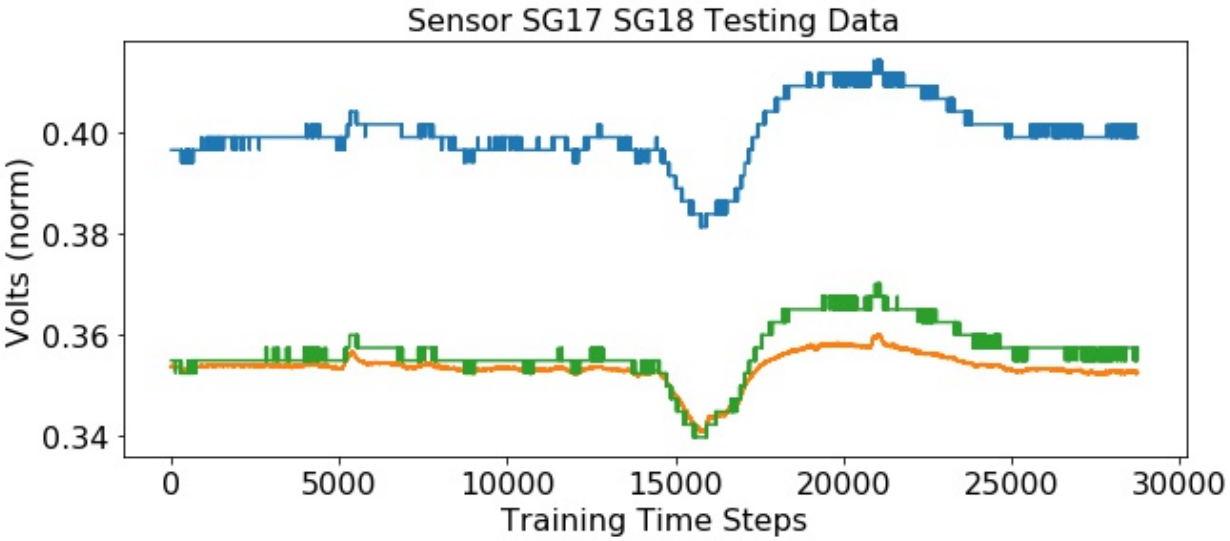

Figure 81: Bagging Results for Sensors 17 and 18

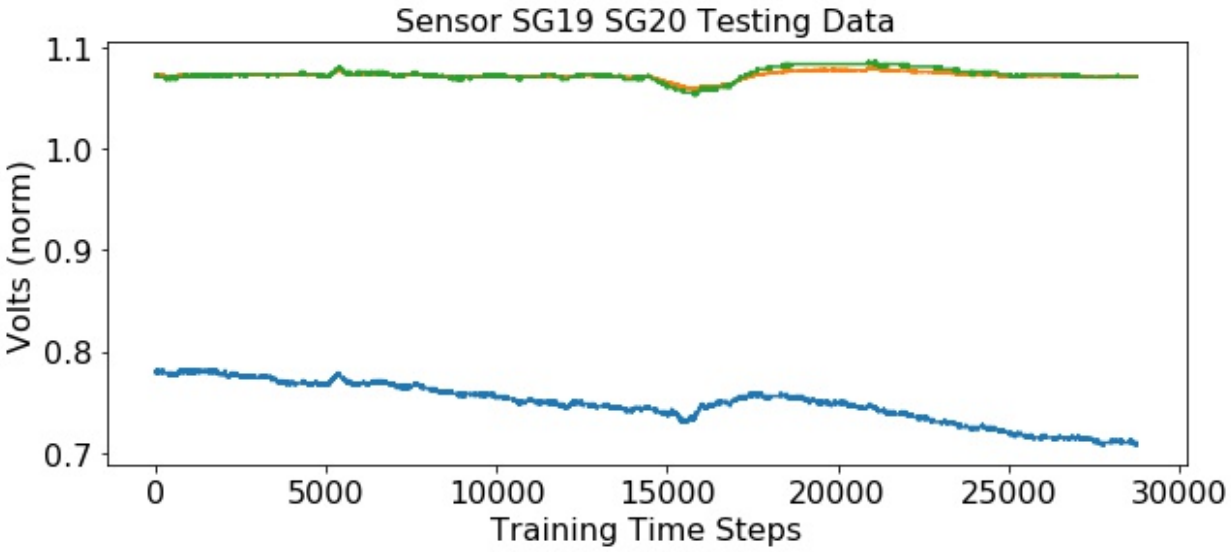

Figure 82: Bagging Results for Sensors 19 and 20

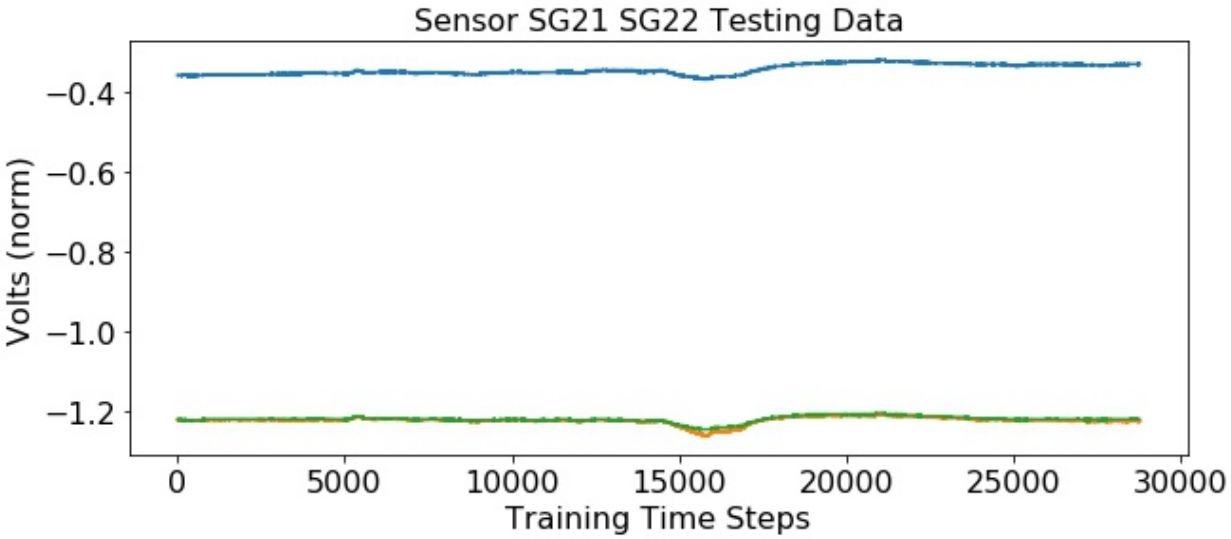

Figure 83: Bagging Results for Sensors 21 and 22

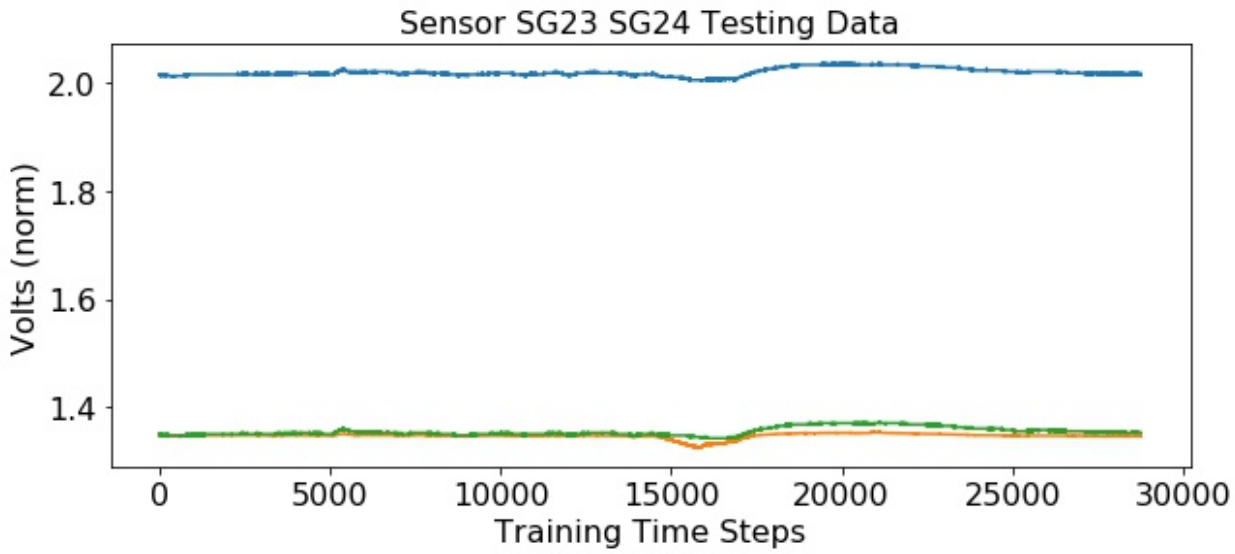

Figure 84: Bagging Results for Sensors 23 and 24

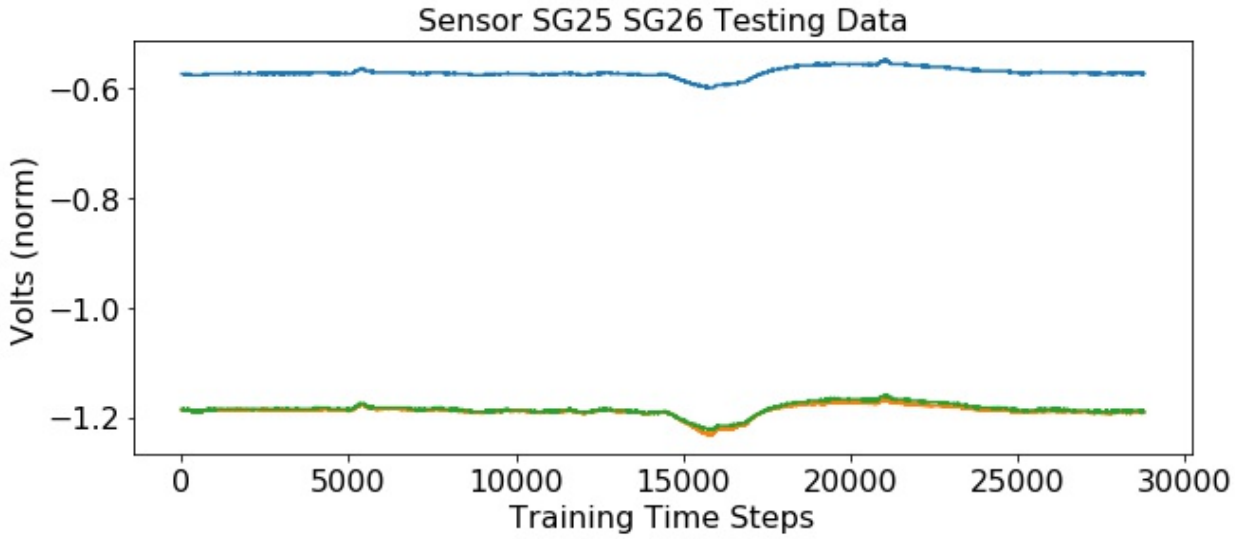

Figure 85: Bagging Results for Sensors 25 and 26

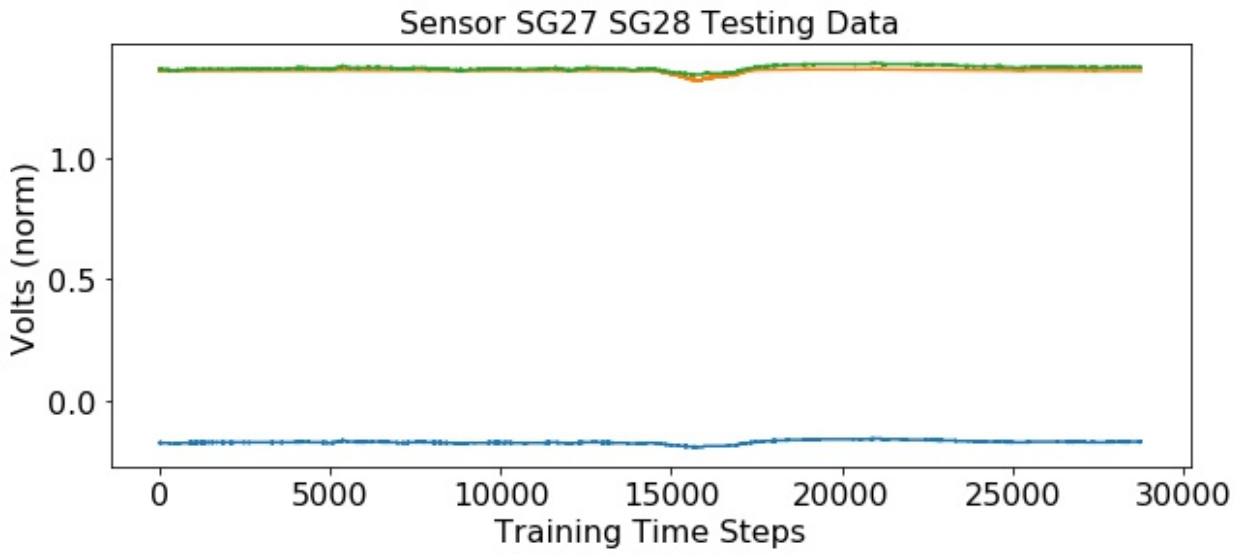

Figure 86: Bagging Results for Sensors 27 and 28

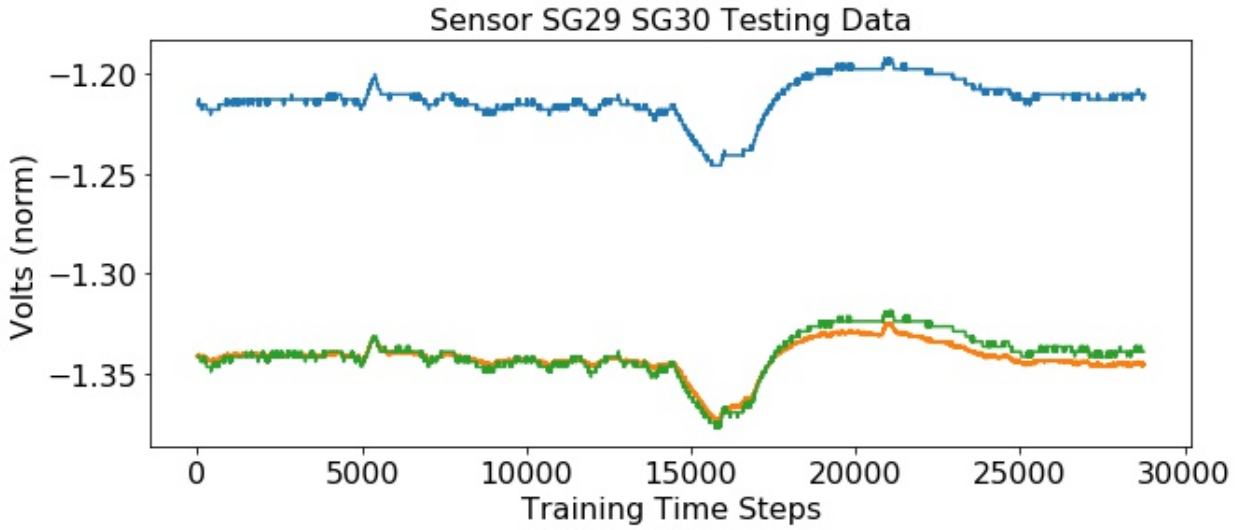

Figure 87: Bagging Results for Sensors 29 and 30

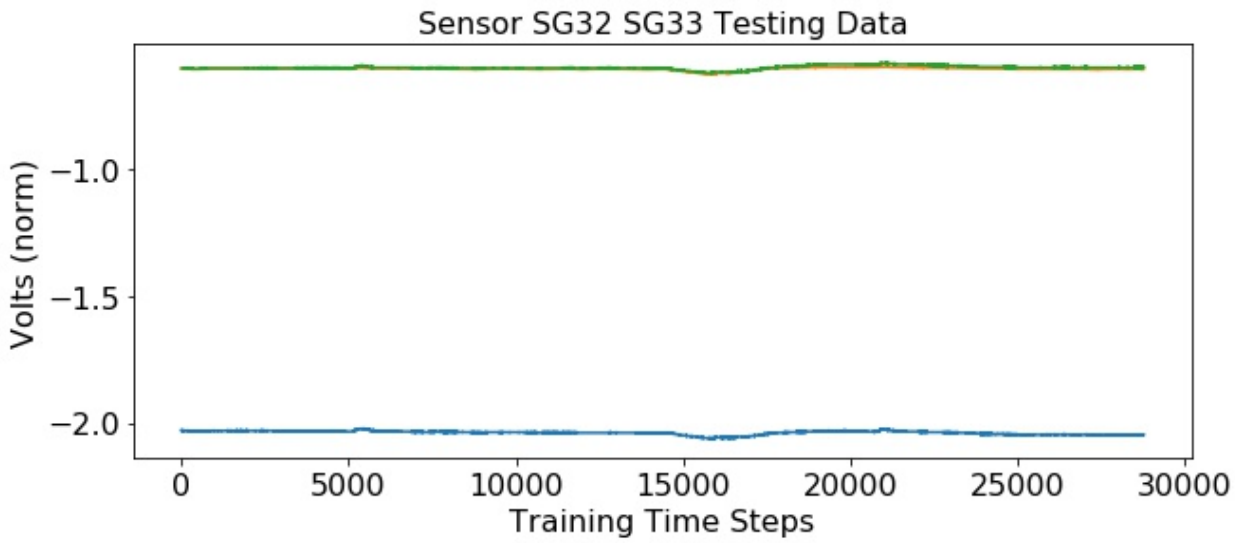

Figure 88: Bagging Results for Sensors 32 and 33

6.5 OBSERVATIONS AND DISCUSSION

The bagged model appears to do an excellent job of estimating the true sensor readings. The predictions for sensors 1, 3, 7, 19, 21, 23, 25, 27, 29, and 32 – which are the uncalibrated sensors for Figures 73, 74, 76, 82, 83, 84, 85, 86, 87, and 88 respectively – are very good and very tight to the ground truth sensor readings. The predictions for sensors 5, 9, 11, 15 and 17, which correspond to Figures 75, 77, 78, 80, and 81 respectively, have a looser fit to their corresponding ground truth sensor readings. The predictions for these sensors appear to be somewhat flatter than the ground truth and uncalibrated readings, both over and underestimating the ground truth values, though generally have the correct shape and are only off by a small fraction of a volt. All predictions of the ensemble avoid the problem seen in the standalone models with overcompensating for the system-wide voltage upticks around the 5,500 and

21,000 time steps. The only prediction that is slightly troubling is the prediction for sensor 13, which is shown in Figure 79. Although the overall error for the predictions for this sensor is low, the predictions have the opposite shape when compared to the ground truth. When both the uncalibrated sensor reading and the ground truth sensor reading go down, the predicted value goes up.

A common use for sensors in a large system is to detect change. In such cases, current sensor readings are compared to previous sensor readings in order to determine potential changes in state or developing trends. In cases where sensors are used actively to regulate the system, once the system has deviated far enough away from the status quo, actions are taken to return the system to its operational state. It is very possible that the system will have a set of actions or multiple sets of multiple actions that will directly oppose each other. Take for example a refrigeration system. If the goal is to keep a constant and cool temperature, the system will likely have a way to cool the system down and, if a temperature too low is also damaging, a way to increase the temperature of the system.

Given this scenario, now consider the predicted values for sensors 5, 6, 11, 15 and 17. Although the fit isn't as tight as the predictions 7, 11, and 25, when there is a change in the uncalibrated sensor reading, the predicted sensor reading changes and moves in the same direction. The predictions for sensor 13 respond to changes in the uncalibrated sensor, but deviate in the wrong direction. In the refrigeration example mentioned above, this would prove to be very problematic. As the temperature increases, the predicted value of the temperature would decrease. Eventually, the temperature would get high enough that the predicted temperature would decrease to

the point where the system would respond by attempting to warm the system up to re-attain the status quo. In this case, the system is in a positive feedback loop as opposed to the status-quo-maintaining negative feedback loop the rest of the sensors would be in. While this is very bad, there are two factors that, when coupled together, cause this to not be a huge concern.

Recall from Section 1.1 that one of the necessary conditions for blind drift calibration is that the number of sensors in the system exceeds the number of true dimensions of the system. Thus, it can be known that this sensor will not be the only sensor in the system. In the case of the sensor readings for the structure, 15 of the 16 sensor predictions change in the correct direction. If the readings were temperature readings, even halfway decent control software should be able to determine that the predicted reading for sensor 13 is an anomaly. Additionally, sensor 13 was the only sensor to exhibit this behavior. This would support the notion that the training data was likely the primary issue. Figure 70 from Section 5.5 shows the training data for sensor 13 and the ground truth data for that sensor (sensor 14), as this sensor set had similar issues when fed through the stacked model and both standalone models.

The training data for this sensor exhibits a pattern of the ground truth data continually rising while the uncalibrated sensor reading oscillates. Even when there are significant increases in the uncalibrated data, the ground truth sensor only increases a small amount. The sharp decreases in the uncalibrated data are not matched in the ground truth sensor, with only small decreases present. Thus, it is not surprising or alarming that the predictions for this sensor would incorrectly assume that changes in the uncalibrated sensor would result in slight increases to the predicted ground truth

sensor value. This is consistent with the data presented in Figure 79. Coupling these factors together, it can be reasonably concluded that this incorrect sensor reading is nothing more than a data driven anomaly. Although a similar anomaly could be present in other systems, it is also known that there will be other sensors present in the system as well.

A relevant parallel to this error are that of the control systems aboard the Boeing 737 Max 8 airplanes that experienced two tragic and fatal crashes in 2018 and 2019. Those particular airplanes are equipped with a system Boeing called the Maneuvering Characteristics Augmentation System, or MCAS, that was designed to counteract stalls that were the result of an aggressive angle-of-attack. To do this, the airplane and system relied on three angle-of-attack sensors. It was assumed that if one of the sensors disagreed with the other two, the system would ignore the obviously faulty sensor. This was not the case. Instead, the system would default to the angle-of-attack sensor on the pilot's side of the plane when it disagreed with the sensor on the co-pilot's side, even though there was still a third, backup sensor present [20]. This incident was regarded as a global faux pas on Boeing's part, costing the company tens of billions of dollars and killing several hundred people. With this particular case in mind, it would be foolish to allow this one out of 16 sensors to trigger any reaction, much less a strong one, from the system. Thus, such an error is not and should not be considered a significant error nor a significant problem.

6.6 CONCLUSION

Compared to both the standalone GRU and LSTM models, the bagged GRU ensemble takes considerably longer to train and make predictions. The average training time per model was roughly 78,750 seconds, or 21:52:30 (350 seconds per epoch for 225 epochs), for 30 models. This resulted in a total training time of 2,362,500 seconds. The time per epoch is somewhat surprising when compared to the training time for the standalone GRU model, first mentioned in Section 5.6. Each epoch of the bagged ensemble took nearly twice as long as an epoch of the standalone GRU model, despite being identical in architecture. This was likely due to hitting resource limits on the high-powered computing (HPC) cluster during training, since all 30 submodels were submitted at the same time. Either way, this cost appears to be fully worthwhile. The performance of the bagged GRU ensemble far exceeds that of the standalone GRU and LSTM models, outperforming it by nearly two factors of 10 (1e-03 versus 5e-05). Thus, the bagged GRU model is considered to not only be successful, but a noteworthy increase in the performance in the use of deep learning to solve the blind drift calibration problem.

7. CONCLUSION

7.1 OVERVIEW OF MODELS

In review, the goal of this thesis was to explore the improvements that ensemble learning could provide to the blind drift calibration problem. In order to do this, two different ensemble learning techniques were compared to two standalone deep learning models. The standalone models were two popular RNNs: a GRU model and an LSTM model. The ensemble techniques explored were stacking and bagging. The best MSE achieved for each model type is shown in Table 11. The overall cost per model is provided in Table 12. It is worth noting that the training time for each model is heavily hardware dependent. The times are provided merely as a reference for comparison between the different models (all models were trained on identical hardware with identical parameters). The factor difference between the models should not deviate across different training environments.

Architecture	MSE
Standalone GRU	0.00105
Standalone LSTM	0.00119
Stacked GRU onto LSTM onto FCNN	0.00104
Bagged GRU	0.00005531

Table 11: Best Results From All Models

Architecture	Total Training Time
Standalone GRU	81,175 seconds
Standalone LSTM	78,600 seconds
Stacked GRU onto LSTM onto FCNN	40,410 seconds
Bagged GRU	2,362,500 seconds

Table 12: Gross Total Training Time by Model

As mentioned in Section 6.6, the total time required for the bagged GRU ensemble is likely skewed high due to exhausting allowable computing resources. Using the same time-per-epoch as the standalone GRU model, the total amount of time required to train the ensemble drops to 1,289,250 seconds. This is a significant reduction and is likely a more accurate estimate of the time required to train the bagged GRU ensemble. This is purely a note to address the anomaly of the differing training time requirements. This estimated figure will not be used henceforth.

The major advantage bagging offers over stacking is the training process of bagging is embarrassingly parallel. Old Dominion University's Wahab HPC cluster allows for a maximum of 2048 cores to be reserved, though this number could fluctuate downward during periods of high demand [21]. Given the parameters used to train the bagged ensemble (the parameters were the same for all models trained for comparison's sake), this allowed 12 of the models to train at any given time. This results in 2.5 "batches" of models to be trained, or two iterations of 12 models and one iteration of 6 models. This lowers the total time required to train the bagged GRU

ensemble to 236,250 seconds, which is an order of magnitude lower than the total time required. This is a significant improvement. The bagged ensemble is the only model in this thesis that can be trained in parallel and the allocation of more GPU resources will result in a factor of scale reduction of the time required to train the model. This is what accounts for the factor of 10 reduction in the training time for the Bagged GRU model shown in Table 13.

Architecture	Total Training Time
Standalone GRU	81,175 seconds
Standalone LSTM	78,600 seconds
Stacked GRU onto LSTM onto FCNN	40,410 seconds
Bagged GRU (Parallel)	236,250 seconds

Table 13: Net Total Training Time by Model

7.2 ANALYSIS OF STACKED MODEL

The stacked model beat the performance of the standalone LSTM model and ever-so-slightly beat the performance of the GRU model. The biggest performance increase with regards to the stacked model came with the total time required to train the ensemble. Unfortunately, the stacked ensemble requires twice as much time to make a prediction, as each input has to flow through twice as many layers compared to the

standalone models. This problem cannot be solved with parallel computing resources. This is only a notable problem, however, if computing power is very scarce. Recovering drift-free measurements can be done in near real-time for both the standalone models and the stacked ensemble if respectable computing power is available. Due to the matched performance when compared to the best standalone model and the large decrease in training time, the stacked ensemble is a notable improvement to solving the blind drift calibration problem.

7.3 ANALYSIS OF BAGGED MODEL

The bagged model was very successful. The performance of the bagged model far exceeds that of the standalone GRU and LSTM models. The predictions made by the bagged model, which are discussed in-depth in Section 6.4 through 6.6, were quite good, outperforming the predictions of both standalone models in terms of both shape and fit. The significant downside of the bagged ensemble is the space and time costs. The ensemble is approximately 30 times larger than the standalone GRU model and requires more than 30 times the number of computations to make predictions (30 GRU models make predictions and then these 30 predictions need to be averaged). Although this overhead can be significantly reduced using parallel computing schemes, this could be detrimental in some cases if the system this drift-recovery technique was deployed to had some sort of low-powered control system comparable to an Arduino Uno or a Raspberry Pi. In those cases, the bagged solution would require additional, custom hardware to effectively deliver timely, drift-free measurements. Other than this, the

bagged ensemble provides a significant improvement over the standalone models when it comes to solving the blind drift calibration problem.

7.4 CONCLUSION OF OVERALL ENSEMBLE PERFORMANCE

Ensemble learning is very effective in a laboratory setting at predicting drift-free sensor readings given drifted sensor values. Standalone deep learning models, namely RNNs, are already quite proficient at this task, but ensemble learning models far outperform those models. Stacking achieved comparable performance when compared to the standalone models with a significant reduction in required training time. Bagging, while being significantly more resource intensive, drastically outperformed the standalone models. Ensemble learning proved to be excellent at solving the blind drift calibration problem.

REFERENCES

[1] N. Ramanathan; L. Balzano; M. Burt; D. Estrin; T. Harmon; C. Harvey, et al. (2006). Rapid Deployment with Confidence:Calibration and Fault Detection in Environmental Sensor Networks. *UCLA: Center for Embedded Network Sensing*. Retrieved from https://escholarship.org/uc/item/8v26b5qh

[2] R. Tan, G. Xing, Z. Yuan, X. Liu, J. Yao, "System-level calibration for data fusion in wireless sensor networks," ACM Transactions on Sensor Network, 9(3), 2013.

[3] L. Balzano and R. Nowak, "Blind Calibration of Sensor Networks," 2007 6th International Symposium on Information Processing in Sensor Networks, 2007, pp. 79-88, doi: 10.1109/IPSN.2007.4379667.

[4] Old Dominion University. Norfolk, Virginia (2021, November). Comparison of Deep Learning Blind Drift Calibration Methods on a Structure Lab Testbed (No. 1).

[5] C. Dorffer, M. Puigt, G. Delmaire and G. Roussel, "Blind mobile sensor calibration using an informed nonnegative matrix factorization with a relaxed rendezvous model," *2016 IEEE International Conference on Acoustics, Speech and Signal Processing* (ICASSP), 2016, pp. 2941-2945, doi: 10.1109/ICASSP.2016.7472216.

[6] C. Xiang, P. Yang, C. Tian, H. Cai and Y. Liu, "Calibrate without Calibrating: An Iterative Approach in Participatory Sensing Network," in *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 2, pp. 351-361, Feb. 2015, doi: 10.1109/TPDS.2014.2308205.

[7] Y. Wang, A. Yang, Z. Li, X. Chen, P. Wang and H. Yang, "Blind Drift Calibration of Sensor Networks Using Sparse Bayesian Learning," in *IEEE Sensors Journal*, vol. 16, no. 16, pp. 6249-6260, Aug.15, 2016, doi: 10.1109/JSEN.2016.2582539.

[8] Y. Wang, A. Yang, X. Chen, P. Wang, Y. Wang and H. Yang, "A Deep Learning Approach for Blind Drift Calibration of Sensor Networks," in *IEEE Sensors Journal*, vol. 17, no. 13, pp. 4158-4171, 1 July1, 2017, doi: 10.1109/JSEN.2017.2703885.

[9] Russell, S., & Norvig, P. (2020). “Artificial Intelligence: A Modern Approach (Pearson Series in Artifical Intelligence) (4th ed.).” Pearson. Hoboken, New Jersey.

[10] Pai, A. (2020, February 17). “CNN vs. RNN vs. ANN – Analyzing 3 Types of Neural Networks in Deep Learning.” Analytics Vidhya.Pai, A. (2020, February 17). Retrieved April 27, 2022, from https://www.analyticsvidhya.com/blog/2020/02/cnn-vs-rnn-vs-mlp-analyzing-3-types-of-neural-networks-in-deep-learning/#h2_5

[11] A. Bengio. “Practical Recommendations for Gradient-Based Training of Deep Architectures.” arXiv:1206.5533v2, 2012.

[12] 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, 1930.

[13] D. Kingma, J. Ba. "Adam: A method for stochastic optimization." arXiv:1412.6980, 2014.

[14] “Keras documentation: Optimizers.” (n.d.). Keras. Retrieved March 13, 2022, from https://keras.io/api/optimizers/

[15] Nielsen, M. (2015). “Neural Networks and Deep Learning (1st ed.).” Determination Press. San Francisco, California.

[16] “Keras documentation: Callbacks API.” (n.d.). Keras. Retrieved March 25, 2022, from https://keras.io/api/optimizers/

[17] Illowsky, B., & Dean, S. (2022). “De Anza College Introductory Statistics (1st ed.).” LibreText. Cupertino, California.

[18] L. Liu, S. P. Chin and T. D. Tran, "Reducing Sampling Ratios and Increasing Number of Estimates Improve Bagging in Sparse Regression," 2019 53rd Annual Conference on Information Sciences and Systems (CISS), 2019, pp. 1-5, doi: 10.1109/CISS.2019.8692865.

[19] Brownlee, J. (2018, December 24). How to Create a Bagging Ensemble of Deep Learning Models in Keras. Machine Learning Mastery. Retrieved May 10, 2022, from https://machinelearningmastery.com/how-to-create-a-random-split-cross-validation-and-bagging-ensemble-for-deep-learning-in-keras/

[20] Langewiesche, W. (2019, September 18). What Really Brought Down the Boeing 737 Max? The New York Times Magazine. Retrieved May 15, 2022, from https://www.nytimes.com/2019/09/18/magazine/boeing-737-max-crashes.html

[21] Partition Policies. (n.d.). Research Cloud Computing. Retrieved March 19, 2022, from https://wiki.hpc.odu.edu/en/Cluster/Wahab

VITA

Devin Scott Drake
Computer Science, Old Dominion University
3300 Engineering & Computational Sciences Building
Norfolk, VA 23529

Profile

I am finishing my second year as a Master's student at Old Dominion University studying computer science. As part of the degree, I completed the graduate-level cyber security certificate program. I work full time for the Department of the Navy at Naval Surface Warfare Center, Dahlgren Division, though I have been deployed to Kosovo with my Army National Guard unit since January 2022. My main interests in the field of Computer Science are software engineering and big data analysis.

Education

Old Dominion University — M.S. in Computer Science, Summer 2022 (proj.). GPA: 3.95

Old Dominion University — Graduate Certificate in Cyber Security, 2021. GPA: 4.0

Virginia Tech — B.S. in Computer Science (Minors in Mathematics and Cyber Security), 2020. GPA: 3.69

Experience

Naval Surface Warfare Center, Dahlgren Division — 2020 - Present
I am employed by Naval Surface Warfare Center Dahlgren Division (NSWCDD) in Dahlgren, VA as a Software Engineer working within the Integrated Combat Systems Department. I am on a team developing a Display Architecture using JavaFX. I did a rotation with a group developing a system to interact with an RDF Triple store data layer. Additionally, I did a second rotation with a group doing software assurance on tactical code (both rotations were 4 months).

Virginia Army National Guard — 2018 - Present
I am enlisted as a SPC serving in the Staunton, Virginia-based Headquarters and Headquarters Company, 116th Infantry Brigade Combat Team as a Combat Engineer. I deployed to Kosovo supporting the NATO mission Operation Joint Guardian.

Undergraduate Teaching Assistant — 2018 - 2020
I worked 4 consecutive semesters as an Undergraduate Teaching Assistant. Two semesters were in an Intro to Programming in Python, one semester in an Intro to Programming in C++, and one semester in an Intro to Software Design in Java course.